Reverse Engineering для начинающих

Денис Юричев
<dennis@yurichev.com>

Версия этого текста (8 мая 2014 г.).
Возможно, более новая версии текста, а также англоязычная версия, также доступна по ссылке http://yurichev.com/RE-book.html. Версия для электронных читалок так же доступна по ссылке.

Вы также можете подписаться на мой twitter для получения информации о новых версиях этого текста, и т.д: @yurichev_ru [1], либо подписаться на список рассылки [2].

[1] https://twitter.com/yurichev_ru
[2] http://yurichev.com/mailing_lists.html

Пожалуйста жертвуйте!

Я писал эту книгу не менее года, здесь более 600 страниц, и она бесплатная.
Книги такого же уровня стоят от $20 до $50.

Больше об этом: 0.1.

Краткое оглавление

Оглавление

0.1 Предисловие

Здесь (будет) немного моих заметок о reverse engineering на русском языке для начинающих, для тех кто хочет научиться понимать создаваемый Си/Си++ компиляторами код для x86 (коего, практически, больше всего остального) и ARM.

У термина "reverse engineering" несколько популярных значений: 1) исследование скомпилированных программ; 2) сканирование трехмерной модели для последующего копирования; 3) восстановление структуры СУБД. Настоящий сборник заметок связан с первым значением

Рассмотренные темы

x86, ARM.

Затронутые темы

Oracle RDBMS (58), Itanium (66), донглы для защиты от копирования (55), LD_PRELOAD (47.2), переполнение стека, ELF[3], формат файла PE в win32 (48.2), x86-64 (23.1), критические секции (48.4), сисколлы (46), TLS[4], адресно-независимый код (PIC[5]) (47.1), profile-guided optimization (68.1), C++ STL (29.4), OpenMP (65), SEH ().

Мини-ЧаВО[6]

- Q: Нужно ли учиться понимать язык ассемблера в наше время?
  A: Да: ради того чтобы понимать лучше внутреннее устройство, отлаживать код лучше и быстрее.
- Q: Нужно ли учиться писать на языке ассемблера в наше время?
  A: Пожалуй, нет, если только не писать низкоуровневый код для ОС[7].
- Q: Но для написания очень оптимизированных процедур?
  A: Нет, современные компиляторы Си/Си++ делают это лучше.
- Q: Нужно ли знать внутреннее устройство микропроцессоров?
  A: Современные CPU[8] очень сложные. Если вы не собираетесь писать очень оптимизированный код или не работаете над кодегенератором компилятора, тогда устройство CPU можно изучать только в общих чертах [9]. В то же время для понимания и анализа кода достаточно только знать ISA[10], назначения регистров, т.е., "внешнюю" часть CPU, доступную для прикладного программиста.
- Q: И всё-таки, зачем мне учить ассемблер?
  A: В основном для лучшего понимания происходящего во время отладки и для исследования программ без наличия исходных кодов, включая зловреды (или вредоносы) [11].
- Q: Как можно найти работу reverse engineer-а?
  A: На reddit, посвященному RE[12], время от времени бывают hiring thread (2013 Q3, 2014). Посмотрите там.

[3] Формат исполняемых файлов, использующийся в Linux и некоторых других *NIX
[4] Thread Local Storage
[5] Position Independent Code: 47.1
[6] Часто задаваемые вопросы
[7] Операционная Система
[8] Central processing unit
[9] Очень хороший текст на эту тему: [10]
[10] Instruction Set Architecture (Архитектура набора команд)
[11] современные (2013) русскоязычные термины для malware
[12] http://www.reddit.com/r/ReverseEngineering/

Об авторе

Денис Юричев — опытный reverse engineer и программист. С его резюме можно ознакомиться на его сайте[13].

Благодарности

Андрей “herm1t” Баранович, Слава ”Avid” Казаков, Станислав ”Beaver” Бобрицкий, Александр Лысенко, Александр ”Lstar” Черненький, Андрей Зубинский, Владимир Ботов, Марк “Logxen” Купер, Shell Rocket, Yuan Jochen Kang, Arnaud Patard (rtp на #debian-arm IRC), и всем тем на github.com кто присылал замечания и коррективы.

Было использовано множество пакетов LaTeX: их авторов я также хотел бы поблагодарить.

Отзывы об этой книге

- “It’s very well done .. and for free .. amazing.”[14] Daniel Bilar, Siege Technologies, LLC.
- “...excellent and free”[15] Pete Finnigan, гуру по безопасности Oracle RDBMS.
- “... book is interesting, great job!” Michael Sikorski, автор книги *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- “... my compliments for the very nice tutorial!” Herbert Bos, профессор университета Vrije Universiteit Amsterdam.
- “... It is amazing and unbelievable.” Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- “Thanks for the great work and your book.” Joris van de Vis, SAP Netweaver & Security specialist.

Пожертвования

Как выясняется, быть (техническим) писателем требует много сил и работы.

Эта книга является свободной, находится в свободном доступе, и доступна в виде исходных кодов [16] (LaTeX), и всегда будет оставаться таковой.

В мои текущие планы насчет этой книги входит добавление информации на эти темы: PLANS[17].

Если вы хотите, чтобы я продолжал свою работу и писал на эти темы, вы можете рассмотреть идею пожертвования.

Я писал эту книгу более года [18], здесь более 500 страниц. Здесь ≈ 300 TeX-файлов, ≈ 90 исходников на Си/Си++, ≈ 350 различных листингов.

Цены на другие книги по этой же тематике на amazon.com колеблются в пределах от $20 до $50.

Со способами пожертвовать деньги можно ознакомиться на странице http://yurichev.com/donate.html

Имена всех жертвователей будут перечислены в книге! Жертвователи также имеют право просить меня дописывать в книгу что-то раньше, чем остальное.

Почему не попробовать издаться? Потому что это техническая литература, которая, как мне кажется, не может быть закончена или быть замороженной в бумажном виде. Такие технические справочники чем-то похожи на Wikipedia или библиотеку MSDN[19], они могут развиваться бесконечно долго. Кто-то может сесть и, не

[13] http://yurichev.com/Dennis_Yurichev.pdf
[14] https://twitter.com/daniel_bilar/status/436578617221742593
[15] https://twitter.com/petefinnigan/status/400551705797869568
[16] https://github.com/dennis714/RE-for-beginners
[17] https://github.com/dennis714/RE-for-beginners/blob/master/PLANS
[18] Самый первый коммит в git от марта 2013:
https://github.com/dennis714/RE-for-beginners/tree/1e57ef540d827c7f7a92fcb3a4626af3e13c7ee4
[19] Microsoft Developer Network

отрываясь, написать всё от начала до конца, опубликовать это и забыть. Как выясняется, это не я. Каждый день меня посещают мысли вроде “это было написано плохо, можно было бы и лучше написать”, “это плохой пример, я знаю получше”, “ещё одна вещь, которую я могу объяснить лучше и короче” и т.д. Как можно увидеть в истории коммитов исходников этой книги, я делаю много мелких изменений почти каждый день: https://github.com/dennis714/RE-for-beginners/commits/master.

Так что книга, наверное, будет в виде “rolling release”, как говорят о дистрибутивах Linux вроде Gentoo. Без релизов (и дедлайнов) вообще, а постепенная разработка. Я не знаю, сколько займет времени написать всё что я знаю. Может быть, 10 лет или больше. Конечно, это не очень удобно для читателей, желающих стабильности, но всё что я могу им предложить — это файл ChangeLog [20] , служащий как секция “что нового”. Те, кому интересно, могут проверять его время от времени, или мой блог/twitter [21] .

Жертвователи

6 * аноним, 2 * Олег Выговский, Daniel Bilar, James Truscott, Luis Rocha, Joris van de Vis, Richard S Shultz, Jang Minchang, Shade Atlas, Yao Xiao.

Об иллюстрациях

Читатели, привыкшие читать интернет-страницы, вероятно привыкли к тому что иллюстрации находятся там же, где и должны. Это потому что там нет разбивок на страницы, там она только одна. В книгах же, иллюстрации далеко не всегда удается вставить в том контексте где нужно. Так что, здесь бывает так, что они все находятся в конце секции, и по тексту на них ставятся ссылки вроде “илл.1.1”.

[20] https://github.com/dennis714/RE-for-beginners/blob/master/ChangeLog
[21] http://blog.yurichev.com/ https://twitter.com/yurichev_ru

Часть I

Образцы кода

Всё познается в сравнении

Автор неизвестен

Когда я учил Си, а затем Си++, я просто писал небольшие фрагменты кода, компилировал и смотрел что получилось на ассемблере. Так было намного проще понять. Я делал это такое количество раз, что связь между кодом на Си/Си++ и тем, что генерирует компилятор, вбилась мне в подсознание достаточно глубоко, поэтому я могу, глядя на код на ассемблере, сразу понимать, в общих чертах, что там было написано на Си. Возможно это поможет кому-то ещё, попробую описать некоторые примеры.

Глава 1

Краткое введенение в CPU

CPU это собственно устройство исполняющее все программы.
Немного терминологии:

Инструкция : примитивная команда CPU. Простейшие примеры: перемещение между регистрами, работа с памятью, примитивные арифметические операции. Как правило, каждый CPU имеет свой набор инструкций (ISA).

Машинный код : код понимаемый CPU. Каждая инструкция обычно кодируется несколькими байтами.

Язык ассемблера : машинный код плюс некоторые расширения призванные облегчить труд программиста: макросы, итд.

Регистр CPU : Каждый CPU имеет некоторый фиксированный набор регистров общего назначения (GPR[1]). ≈ 8 в x86, ≈ 16 в x86-64, ≈ 16 в ARM. Проще всего понимать регистр как временную переменную без типа. Можно представить что вы пишете на ЯП[2] высокого уровня и у вас только 8 переменных шириной 32 бита. Можно сделать очень много используя только их!

Откуда взялась разница между машинным кодом и ЯП высокого уровня? Человеку проще писать на ЯП высокого уровня вроде Си/Си++, Java, Python, а CPU проще работать с абстракциями куда более низкого уровня. Возможно, можно было бы придумать CPU исполняющий код ЯП высокого уровня, но он был бы значительно сложнее. И наоборот, человеку очень неудобно писать на ассемблере из-за его низкоуровневости, к тому же, крайне трудно обойтись без мелких ошибок. Программа переводящая код из ЯП высокого уровня в ассемблер называется *компилятором* [3].

[1] General Purpose Registers (регистры общего пользования)
[2] Язык Программирования
[3] В более старой русскоязычной литературе также часто встречается термин “транслятор”.

Глава 2

Hello, world!

Начнем с знаменитого примера из книги “The C programming Language” [17]:

```
#include <stdio.h>

int main()
{
    printf("hello, world");
    return 0;
};
```

2.1 x86

2.1.1 MSVC — x86

Компилируем в MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(Ключ /Fa означает сгенерировать листинг на ассемблере)

Листинг 2.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC выдает листинги в Intel-овском синтаксисе. Разница между Intel-синтаксисом и AT&T будет рассмотрена немного позже.

Компилятор сгенерировал файл `1.obj`, который впоследствии будет слинкован линкером в `1.exe`.

В нашем случае этот файл состоит из двух сегментов: `CONST` (для данных-констант) и `_TEXT` (для кода).

Строка `‘‘hello, world’’` в Си/Си++ имеет тип `const char*`, однако не имеет имени.

Но компилятору нужно как-то с ней работать, так что он дает ей внутреннее имя `$SG3830`.

Так что пример можно было бы переписать вот так:

```
#include <stdio.h>

const char *$SG3830="hello, world";

int main()
{
    printf($SG3830);
    return 0;
};
```

Вернемся к листингу на ассемблере. Как видно, строка заканчивается нулевым байтом — это требования стандарта Си/Си++ для строк. Больше о строках в Си: 36.1.

В сегменте кода `_TEXT` находится пока только одна функция: `main()`.

Функция `main()`, как и практически все функции, начинается с пролога и заканчивается эпилогом [1].

Далее следует вызов функции `printf()`: `CALL _printf`.

Перед этим вызовом адрес строки (или указатель на неё) с нашим приветствием при помощи инструкции `PUSH` помещается в стек.

После того, как функция `printf()` возвращает управление в функцию `main()`, адрес строки (или указатель на неё) всё еще лежит в стеке.

Так как он больше не нужен, то указатель стека (регистр `ESP`) корректируется.

`ADD ESP, 4` означает прибавить 4 к значению в регистре `ESP`.

Почему 4? Так как это 32-битный код, для передачи адреса нужно аккурат 4 байта. В x64-коде это 8 байт.

‘‘`ADD ESP, 4`’’ эквивалентно ‘‘`POP регистр`’’, но без использования какого-либо регистра[2].

Некоторые компиляторы, например, Intel C++ Compiler, в этой же ситуации могут вместо `ADD` сгенерировать `POP ECX` (подобное можно встретить, например, в коде Oracle RDBMS, им скомпилированном), что почти то же самое, только портится значение в регистре `ECX`.

Возможно, компилятор применяет `POP ECX`, потому что эта инструкция короче (1 байт против 3).

О стеке можно прочитать в соответствующем разделе (4).

После вызова `printf()` в оригинальном коде на Си/Си++ указано `return 0` — вернуть 0 в качестве результата функции `main()`.

В сгенерированном коде это обеспечивается инструкцией `XOR EAX, EAX`

`XOR`, на самом деле, как легко догадаться, “исключающее ИЛИ” [3] , но компиляторы часто используют его вместо простого `MOV EAX, 0` — снова потому, что опкод короче (2 байта против 5).

Бывает так, что некоторые компиляторы генерируют `SUB EAX, EAX`, что значит *отнять значение в* ***EAX*** *от значения в* ***EAX***, что в любом случае даст 0 в результате.

Самая последняя инструкция `RET` возвращает управление в вызывающую функцию. Обычно это код Си/Си++ CRT[4], который, в свою очередь, вернёт управление операционной системе.

2.1.2 GCC — x86

Теперь скомпилируем то же самое компилятором GCC 4.4.1 в Linux: `gcc 1.c -o 1`

Затем, при помощи IDA[5], посмотрим, как создалась функция `main()`.

(IDA, как и MSVC, показывает код в Intel-синтаксисе).

N.B. Мы также можем заставить GCC генерировать листинги в этом формате при помощи ключей `-S -masm=intel`

Листинг 2.2: GCC

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world"
```

[1] Об этом смотрите подробнее в разделе о прологе и эпилоге функции (3).

[2] Флаги процессора, впрочем, модифицируются

[3] http://en.wikipedia.org/wiki/Exclusive_or

[4] C runtime library: sec:CRT

[5] Interactive Disassembler

```
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Почти то же самое. Адрес строки "hello, world", лежащей в сегменте данных, вначале сохраняется в EAX, затем записывается в стек. А еще в прологе функции мы видим AND ESP, 0FFFFFFF0h — эта инструкция выравнивает значение в ESP по 16-байтной границе, делая все значения в стеке также выровненными по этой границе (процессор более эффективно работает с переменными, расположенными в памяти по адресам кратным 4 или 16)[6].

SUB ESP, 10h выделяет в стеке 16 байт. Хотя, как будет видно далее, здесь достаточно только 4.

Это происходит потому, что количество выделяемого места в локальном стеке тоже выровнено по 16-байтной границе.

Адрес строки (или указатель на строку) затем записывается прямо в стек без помощи инструкции PUSH. *var_10* по совместительству — и локальная переменная и одновременно аргумент для printf(). Подробнее об этом будет ниже.

Затем вызывается printf().

В отличие от MSVC, GCC в компиляции без включенной оптимизации генерирует MOV EAX, 0 вместо более короткого опкода.

Последняя инструкция LEAVE — это аналог команд MOV ESP, EBP и POP EBP — то есть возврат указателя стека и регистра EBP в первоначальное состояние.

Это необходимо, т.к., в начале функции мы модифицировали регистры ESP и EBP (при помощи MOV EBP, ESP / AND ESP, ...).

2.1.3 GCC: Синтаксис AT&T

Попробуем посмотреть, как выглядит то же самое в AT&T-синтаксисе языка ассемблера. Этот синтаксис больше распространен в UNIX-мире.

Листинг 2.3: компилируем в GCC 4.7.3

```
gcc -S 1_1.c
```

Получим такой файл:

Листинг 2.4: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
```

[6] Wikipedia: Выравнивание данных

```
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Здесь много макросов (начинающихся с точки). Они нам пока не интересны. Пока что, ради упрощения, мы можем их игнорировать и впредь (кроме макроса *.string*, при помощи которого кодируется последовательность символов, оканчивающихся нулем, такие же строки как в Си). И тогда получится следующее [7]:

Листинг 2.5: GCC 4.7.3

```
.LC0:
        .string "hello, world"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Основные отличия синтаксиса Intel и AT&T следующие:

- Операнды записываются наоборот.

  В Intel-синтаксисе: <инструкция> <операнд назначения> <операнд-источник>.

  В AT&T-синтаксисе: <инструкция> <операнд-источник> <операнд назначения>.

  Чтобы легче понимать разницу, можно запомнить следующее: когда вы работаете с Intel-синтаксисом — можете в уме ставить знак равенства (=) между операндами, а когда с AT&T-синтаксисом — мысленно ставьте стрелку направо (→) [8].

- AT&T: Перед именами регистров ставится знак процента (%), а перед числами знак доллара ($). Вместо квадратных скобок применяются круглые.

- AT&T: К каждой инструкции добавляется специальный символ, определяющий тип данных:

  – l — long (32 бита)
  – w — word (16 бит)
  – b — byte (8 бит)

Возвращаясь к результату компиляции: он идентичен тому, который мы посмотрели в IDA. Одна мелочь: `0FFFFFFF0h` записывается как `$-16`. Это то же самое: `16` в десятичной системе это `0x10` в шестнадцатеричной. `-0x10` будет как раз `0xFFFFFFF0` (в рамках 32-битных чисел).

Ещё: возвращаемый результат устанавливается в 0 обычной инструкцией `MOV` а не `XOR`. `MOV` просто загружает значение в регистр. Её название не очень удачное (данные не перемещаются), в других архитектурах подобная инструкция обычно носит название “load” или что-то в этом роде.

2.2 x86-64

2.2.1 MSVC — x86-64

Попробуем также 64-битный MSVC:

[7] Кстати, для уменьшения генерации “лишних” макросов, можно использовать такой ключ GCC: *-fno-asynchronous-unwind-tables*

[8] Кстати, в некоторых стандартных функциях библиотеки Си (например, memcpy(), strcpy()) также применяется расстановка аргументов как в Intel-синтаксисе: вначале указатель в памяти на блок назначения, затем указатель на блок-источник.

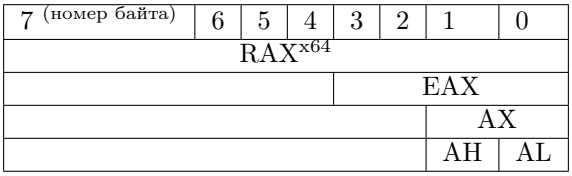
Листинг 2.6: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

В x86-64 все регистры были расширены до 64-х бит и теперь имеют префикс `R-`. Чтобы поменьше задействовать стек (иными словами, поменьше обращаться к внешней памяти), уже давно имелся довольно популярный метод передачи аргументов ф-ции через регистры (fastcall: 44.3). Т.е., часть аргументов ф-ции передается через регистры и часть — через стек. В Win64 первые 4 аргумента ф-ции передаются через регистры `RCX`, `RDX`, `R8`, `R9`. Это мы здесь и видим: указатель на строку в `printf()` теперь передается не через стек, а через регистр `RCX`.

Указатели теперь 64-битные, так что они передаются через 64-битные части регистров (имеющие префикс `R-`). Но для обратной совместимости, к регистрам можно обращаться и к 32-битным частям, используя префикс `E-`.

Вот, например, как выглядит регистр `RAX`/`EAX`/`AX`/`AL` в 64-битных x86-совместимых CPU:



Ф-ция `main()` возвращает значение типа *int*, который, в Си, вероятно для лучшей совместимости и портабельности, оставили 32-битным, вот почему в конце ф-ции `main()` обнуляется не `RAX` а `EAX`, т.е., 32-битная часть регистра.

2.2.2 GCC — x86-64

Попробуем GCC в 64-битном Linux:

Листинг 2.7: GCC 4.4.6 x64

```
.string "hello, world"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world"
        xor     eax, eax  ; количество задействованных векторных регистров
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

В Linux, *BSD и Mac OS X для x86-64 также принят способ передачи аргументов ф-ции через регистры [21]. 6 первых аргументов передаются через регистры `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, а остальные — через стек.

Так что указатель на строку передается через `EDI` (32-битную часть регистра). Но почему не через 64-битную часть, `RDI`?

Важно запомнить что в 64-битном режиме, все инструкции `MOV` записывающие что-либо в младшую 32-битную часть регистра, обнуляют старшие 32-бита [14]. То есть, инструкция `MOV EAX, 011223344h` корректно запишет это значение в `RAX`, старшие биты сбросятся в ноль.

Если посмотреть в IDA скомпилированный объектный файл (.o), увидим также опкоды всех инструкций [9]:

Листинг 2.8: GCC 4.4.6 x64

```
.text:00000000004004D0                  main  proc near
.text:00000000004004D0 48 83 EC 08      sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00   mov     edi, offset format ; "hello, world"
```

[9] Это нужно задать в Options → Disassembly → Number of opcode bytes

```
.text:00000000004004D9 31 C0            xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF   call    _printf
.text:00000000004004E0 31 C0            xor     eax, eax
.text:00000000004004E2 48 83 C4 08      add     rsp, 8
.text:00000000004004E6 C3               retn
.text:00000000004004E6                  main  endp
```

Как видно, инструкция, записывающая в EDI по адресу 0x4004D4, занимает 5 байт. Та же инструкция, записывающая 64-битное значение в RDI, будет занимать 7 байт. Возможно, GCC решил немного сэкономить. К тому же, вероятно, он уверен что сегмент данных где хранится строка, никогда не будет расположен в адресах выше 4GiB.

Здесь мы также видим обнуление регистра EAX перед вызовом printf(). Это делается потому, что по стандарту передачи аргументов в *NIX для x86-64 в EAX передается количество задействованных векторных регистров: "with variable arguments passes information about the number of vector registers used" [21].

2.3 ARM

Для экспериментов с процессором ARM я выбрал два компилятора: популярный в embedded-среде Keil Release 6/2013 и среду разработки Apple Xcode 4.6.3 (с компилятором LLVM-GCC 4.2), генерирующую код для ARM-совместимых процессоров и SOC[10] в iPod/iPhone/iPad, планшетных компьютеров для Windows 8 и Windows RT[11] и таких устройствах как Raspberry Pi.

Везде в этой книге, кроме как если указано иное, идет речь о 32-битном ARM.

2.3.1 Неоптимизирующий Keil + Режим ARM

Для начала скомпилируем наш пример в Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Компилятор *armcc* генерирует листинг на ассемблере в формате Intel, но он содержит некоторые высокоуровневые макросы, связанные с ARM[12], а нам важнее увидеть инструкции "как есть", так что посмотрим скомпилированный результат в IDA.

Листинг 2.9: Неоптимизирующий Keil + Режим ARM + IDA

```
.text:00000000                  main
.text:00000000 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2      ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB      BL      __2printf
.text:0000000C 00 00 A0 E3      MOV     R0, #0
.text:00000010 10 80 BD E8      LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF: main+4
```

Вот чуть-чуть фактов о процессоре ARM, которые желательно знать. Процессор ARM имеет по крайней мере два основных режима: режим ARM и thumb. В первом (ARM) режиме доступны все инструкции и каждая имеет размер 32 бита (или 4 байта). Во втором режиме (thumb) каждая инструкция имеет размер 16 бит (или 2 байта) [13]. Режим thumb может выглядеть привлекательнее тем, что программа на нем может быть 1) компактнее; 2) эффективнее исполняться на микроконтроллере с 16-битной шиной данных. Но за всё нужно платить: в режиме thumb куда меньше возможностей процессора, например, возможен доступ только к 8-и регистрам процессора, и, чтобы совершить некоторые действия, выполнимые в режиме ARM одной инструкцией, нужны несколько thumb-инструкций.

Начиная с ARMv7, имеется также поддержка инструкций thumb-2. Это thumb, расширенный до поддержки куда большего числа инструкций. Распространено заблуждение, что thumb-2 — это смесь ARM и thumb. Это не верно. Просто thumb-2 был дополнен до более полной поддержки возможностей процессора, что теперь может легко конкурировать с режимом ARM. Программа для процессора ARM может представлять смесь процедур, скомпилированных для обоих режимов. Основное количество приложений для iPod/iPhone/iPad скомпилировано для набора инструкций thumb-2, потому что Xcode делает так по умолчанию.

[10] System on Chip

[11] http://en.wikipedia.org/wiki/List_of_Windows_8_and_RT_tablet_devices

[12] например, он показывает инструкции PUSH/POP, отсутствующие в режиме ARM

[13] Кстати, инструкции фиксированного размера удобны тем, что всегда можно легко узнать адрес следующей (или предыдущей) инструкции. Эта особенность будет рассмотрена в секции о switch() (11.2.2).

В вышеприведённом примере можно легко увидеть, что каждая инструкция имеет размер 4 байта. Действительно, ведь мы же компилировали наш код для режима ARM а не thumb.

Самая первая инструкция, ‘‘STMFD SP!, {R4,LR}’’[14], работает как инструкция PUSH в x86, записывая значения двух регистров (R4 и LR[15]) в стек. Действительно, в выдаваемом листинге на ассемблере компилятор *armcc*, для упрощения, указывает здесь инструкцию ‘‘PUSH {r4,lr}’’. Но это не совсем точно, инструкция PUSH доступна только в режиме thumb, поэтому, во избежание путаницы, я предложил работать в IDA.

Итак, эта инструкция уменьшает SP[16], чтобы он указывал на место в стеке, свободное для записи новых значений, затем записывает значения регистров R4 и LR по адресу в памяти, на который указывает измененный регистр SP.

Эта инструкция, как и инструкция PUSH в режиме thumb, может сохранить в стеке одновременно несколько значений регистров, что может быть очень удобно. Кстати, такого в x86 нет. Также следует заметить, что STMFD — генерализация инструкции PUSH (то есть, расширяет её возможности), потому что может работать с любым регистром, а не только с SP, это тоже может быть очень удобно.

Инструкция ‘‘ADR R0, aHelloWorld’’ прибавляет значение регистра PC[17] к смещению, где хранится строка *“hello, world”*. Причем здесь PC, можно спросить? Притом, что это так называемый “адресно-независимый код” [18] он предназначен для исполнения будучи не привязанным к каким-либо адресам в памяти. В опкоде инструкции ADR указывается разница между адресом этой инструкции и местом, где хранится строка. Эта разница всегда будет постоянной, вне зависимости от того, куда был загружен ОС наш код. Поэтому всё, что нужно — это прибавить адрес текущей инструкции (из PC), чтобы получить текущий абсолютный адрес нашей Си-строки.

Инструкция ‘‘BL __2printf’’[19] вызывает функцию printf(). Работа этой инструкции состоит из двух фаз:

- записать адрес после инструкции BL (0xC) в регистр LR;
- затем собственно передать управление в printf(), записав адрес этой функции в регистр PC.

Ведь когда функция printf() закончит работу, нужно знать, куда вернуть управление, поэтому закончив работу, всякая функция передает управление по адресу, записанному в регистре LR.

В этом разница между “чистыми” RISC[20]-процессорами вроде ARM и CISC[21]-процессорами как x86, где адрес возврата записывается в стек[22].

Кстати, 32-битный абсолютный адрес, либо же смещение, невозможно закодировать в 32-битной инструкции BL, в ней есть место только для 24-х бит. Также следует отметить, что из-за того, что все инструкции в режиме ARM имеют длину 4 байта (32 бита), и инструкции могут находится только по адресам кратным 4, то последние 2 бита (всегда нулевых) можно не кодировать. В итоге имеем 26 бит, при помощи которых, можно закодировать смещение $\pm \approx 32M$.

Следующая инструкция ‘‘MOV R0, #0’’[23] просто записывает 0 в регистр R0. Ведь наша Си-функция возвращает 0, а возвращаемое значение всякая функция оставляет в R0.

Последняя инструкция — ‘‘LDMFD SP!, R4,PC’’[24] это инструкция, обратная от STMFD. Она загружает из стека значения для сохранения их в R4 и PC, увеличивая указатель стека SP. Это, в каком-то смысле, аналог POP. N.B. Самая первая инструкция STMFD сохранила в стеке R4 и LR, а *восстанавливаются* во время исполнения LDMFD регистры R4 и PC.

Как я уже описывал, в регистре LR обычно сохраняется адрес места, куда нужно всякой функции вернуть управление. Самая первая инструкция сохраняет это значение в стеке, потому что наша функция main() позже будет сама пользоваться этим регистром, в момент вызова printf(). А затем, в конце функции, это значение можно сразу записать в PC, таким образом, передав управление туда, откуда была вызвана наша функция. Так как функция main() обычно самая главная в Си/Си++, управление будет возвращено в загрузчик ОС, либо куда-то в CRT или что-то в этом роде.

DCB — директива ассемблера, описывающая массивы байт или ASCII-строк, аналог директивы DB в x86-ассемблере.

2.3.2 Неоптимизирующий Keil: Режим thumb

Скомпилируем тот же пример в Keil для режима thumb:

[14] Store Multiple Full Descending
[15] Link Register
[16] указатель стека. SP/ESP/RSP в x86/x64. SP в ARM.
[17] Program Counter. IP/EIP/RIP в x86/64. PC в ARM.
[18] Читайте больше об этом в соответствующем разделе (47.1)
[19] Branch with Link
[20] Reduced instruction set computing
[21] Complex instruction set computing
[22] Подробнее об этом будет описано в следующей главе (4)
[23] MOVe
[24] Load Multiple Full Descending

```
armcc.exe --thumb --c90 -O0 1.c
```

Получим (в IDA):

Листинг 2.10: Неоптимизирующий Keil + Режим thumb + IDA

```
.text:00000000             main
.text:00000000 10 B5          PUSH    {R4,LR}
.text:00000002 C0 A0          ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9    BL      __2printf
.text:00000008 00 20          MOVS    R0, #0
.text:0000000A 10 BD          POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF: main+2
```

Сразу бросаются в глаза двухбайтные (16-битные) опкоды — это, как я уже упоминал, thumb. Кроме инструкции BL. Но на самом деле она состоит из двух 16-битных инструкций. Это потому, что загрузить в PC смещение, по которому находится функция printf(), используя так мало места в одном 16-битном опкоде, нельзя. Поэтому первая 16-битная инструкция загружает старшие 10 бит смещения, а вторая — младшие 11 бит смещения. Как я уже упоминал, все инструкции в thumb-режиме имеют длину 2 байта (или 16 бит). Поэтому невозможна такая ситуация, когда thumb-инструкция начинается по нечетному адресу. Учитывая сказанное, последний бит адреса можно не кодировать. Таким образом, в итоге, в thumb-инструкции BL кодируется смещение $\pm \approx 2M$ от текущего адреса.

Остальные инструкции в функции: PUSH и POP работают почти так же, как и описанные STMFD/LDMFD, только регистр SP здесь не указывается явно. ADR работает так же, как и в предыдущем примере. MOVS записывает 0 в регистр R0 для возврата нуля.

2.3.3 Оптимизирующий Xcode (LLVM) + Режим ARM

Xcode 4.6.3 без включенной оптимизации выдает слишком много лишнего кода, поэтому остановимся на той версии, где как можно меньше инструкций: -O3.

Листинг 2.11: Оптимизирующий Xcode (LLVM) + Режим ARM

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV             R7, SP
__text:000028D0 00 00 40 E3   MOVT            R0, #0
__text:000028D4 00 00 8F E0   ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB   BL              _puts
__text:000028DC 00 00 A0 E3   MOV             R0, #0
__text:000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Инструкции STMFD и LDMFD нам уже знакомы.

Инструкция MOV просто записывает число 0x1686 в регистр R0 — это смещение, указывающее на строку “Hello world!”.

Регистр R7, по стандарту, принятому в [2] — это frame pointer , о нем будет рассказано позже.

Инструкция MOVT R0, #0 записывает 0 в старшие 16 бит регистра. Дело в том, что обычная инструкция MOV в режиме ARM может записывать какое-либо значение только в младшие 16 бит регистра, ведь в ней нельзя закодировать больше. Помните, что в режиме ARM опкоды всех инструкций ограничены длиной в 32 бита. Конечно, это ограничение не касается перемещений между регистрами. Поэтому для записи в старшие биты (от 16-го по 31-го включительно) существует дополнительная команда MOVT. Впрочем, здесь её использование избыточно, потому что инструкция ‘‘MOV R0, #0x1686’’ выше и так обнулила старшую часть регистра. Возможно, это недочет компилятора.

Инструкция ‘‘ADD R0, PC, R0’’ прибавляет PC к R0 для вычисления действительного адреса строки “Hello world!”. Как нам уже известно, это “адресно-независимый код”, поэтому такая корректива необходима.

Инструкция BL вызывает puts() вместо printf().

Компилятор заменил вызов printf() на puts(). Действительно, printf() с одним аргументом это почти аналог puts().

Почти, если принять условие, что в строке не будет управляющих символов `printf()`, начинающихся со знака процента. Тогда эффект от работы этих двух функций будет разным [25].

Зачем компилятор заменил один вызов на другой? Потому что `puts()` работает быстрее [26].

Видимо потому, что `puts()` проталкивает символы в stdout не сравнивая каждый со знаком процента.

Далее уже знакомая инструкция ‘‘`MOV R0, #0`’’, служащая для установки в 0 возвращаемого значения функции.

2.3.4 Оптимизирующий Xcode (LLVM) + Режим thumb-2

По умолчанию, Xcode 4.6.3 генерирует код для режима thumb-2 примерно в такой манере:

Листинг 2.12: Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
__text:00002B6C                    _hello_world
__text:00002B6C 80 B5          PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30    MOVW            R0, #0x13D8
__text:00002B72 6F 46          MOV             R7, SP
__text:00002B74 C0 F2 00 00    MOVT.W          R0, #0
__text:00002B78 78 44          ADD             R0, PC
__text:00002B7A 01 F0 38 EA    BLX             _puts
__text:00002B7E 00 20          MOVS            R0, #0
__text:00002B80 80 BD          POP             {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Инструкции `BL` и `BLX` в thumb, как мы помним, кодируются как пара 16-битных инструкций, а в thumb-2 эти *суррогатные* опкоды расширены так, что новые инструкции кодируются здесь как 32-битные инструкции. Это можно заметить по тому что опкоды thumb-2 инструкций всегда начинаются с `0xFx` либо с `0xEx`. Но в листинге IDA байты опкода переставлены местами, это из-за того, что в процессоре ARM инструкции кодируются так: в начале последний байт, потом первый (для thumb и thumb-2 режима), либо, (для инструкций в режиме ARM) в начале четвертый байт, затем третий, второй и первый (т.е., другой endianness). Так что мы видим здесь что инструкции `MOVW`, `MOVT.W` и `BLX` начинаются с `0xFx`.

Одна из thumb-2 инструкций это ‘‘`MOVW R0, #0x13D8`’’ — она записывает 16-битное число в младшую часть регистра `R0`.

Еще ‘‘`MOVT.W R0, #0`’’ — эта инструкция работает так же, как и `MOVT` из предыдущего примера, но она работает в thumb-2.

Помимо прочих отличий, здесь используется инструкция `BLX` вместо `BL`. Отличие в том, что помимо сохранения адреса возврата в регистре LR и передаче управления в функцию `puts()`, происходит смена режима процессора с thumb на ARM, либо наоборот. Здесь это нужно потому, что инструкция, куда ведет переход, выглядит так (она закодирована в режиме ARM):

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

Итак, внимательный читатель может задать справедливый вопрос: почему бы не вызывать `puts()` сразу в том же месте кода, где он нужен?

Но это не очень выгодно (в плане экономия места) и вот почему.

Практически любая программа использует внешние динамические библиотеки (будь то DLL в Windows, .so в *NIX либо .dylib в Mac OS X). В динамических библиотеках находятся часто используемые библиотечные функции, в том числе стандартная функция Си `puts()`.

В исполняемом бинарном файле (Windows PE .exe, ELF либо Mach-O) имеется секция импортов, список символов (функций либо глобальных переменных) импортируемых из внешних модулей, а также названия самих модулей.

Загрузчик ОС загружает необходимые модули и, перебирая импортируемые символы в основном модуле, проставляет правильные адреса каждого символа.

В нашем случае, *__imp__puts* это 32-битная переменная, куда загрузчик ОС запишет правильный адрес этой же функции во внешней библиотеке. Так что инструкция `LDR` просто берет 32-битное значение из этой переменной и, записывая его в регистр PC, просто передает туда управление.

[25] Также нужно заметить, что `puts()` не требует символа перевода строки ’\n’ в конце строки, поэтому его здесь нет.

[26] http://www.ciselant.de/projects/gcc_printf/gcc_printf.html

Чтобы уменьшить время работы загрузчика ОС, нужно чтобы ему пришлось записать адрес каждого символа только один раз, в соответствующее выделенное для них место.

К тому же, как мы уже убедились, нельзя одной инструкцией загрузить в регистр 32-битное число без обращений к памяти. Так что наиболее оптимально выделить отдельную функцию, работающую в режиме ARM, чья единственная цель — передавать управление дальше, в динамическую библиотеку. И затем ссылаться на эту короткую функцию из одной инструкции (так называемую thunk-функцию) из thumb-кода.

Кстати, в предыдущем примере (скомпилированном для режима ARM), переход при помощи инструкции `BL` ведет на такую же thunk-функцию, однако режим процессора не переключается (отсюда, отсутствие "X" в мнемонике инструкции).

Глава 3

Пролог и эпилог в функции

Пролог функции — это инструкции в самом начале функции. Как правило это что-то вроде такого фрагмента кода:

```
push    ebp
mov     ebp, esp
sub     esp, X
```

Эти инструкции делают следующее: сохраняют значение регистра EBP на будущее, выставляют EBP равным ESP, затем подготавливают место в стеке для хранения локальных переменных.

EBP сохраняет свое значение на протяжении всей функции, он будет использоваться здесь для доступа к локальным переменным и аргументам. Можно было бы использовать и ESP, но он постоянно меняется и это не очень удобно.

Эпилог функции аннулирует выделенное место в стеке, возвращает значение EBP на то что было и возвращает управление в вызывающую функцию:

```
mov    esp, ebp
pop    ebp
ret    0
```

Пролог и эпилог ф-ции обычно находятся в в дизассемблерах для размежевания ф-ций друг от друга.

3.1 Рекурсия

Наличие эпилога и пролога может несколько ухудшить эффективность рекурсии.

Например, однажды я написал функцию для поиска нужного узла в двоичном дереве. Рекурсивно она выглядела очень красиво, но из-за того, что при каждом вызове тратилось время на эпилог и пролог, все это работало в несколько раз медленнее чем та же функция, но без рекурсии.

Кстати, поэтому есть такая вещь как хвостовая рекурсия.

Глава 4

Стек

Стек в компьютерных науках — это одна из наиболее фундаментальных вещей [1].

Технически, это просто блок памяти в памяти процесса + регистр ESP или RSP в x86 или x64, либо SP в ARM, который указывает где-то в пределах этого блока.

Часто используемые инструкции для работы со стеком — это PUSH и POP (в x86 и thumb-режиме ARM). PUSH уменьшает ESP/RSP/SP на 4 в 32-битном режиме (или на 8 в 64-битном), затем записывает по адресу, на который указывает ESP/RSP/SP, содержимое своего единственного операнда.

POP это обратная операция — сначала достает из указателя стека значение и помещает его в операнд (который очень часто является регистром) и затем увеличивает указатель стека на 4 (или 8).

В самом начале регистр-указатель указывает на конец стека. PUSH уменьшает регистр-указатель, а POP — увеличивает. Конец стека находится в начале блока памяти, выделенного под стек. Это странно, но это так.

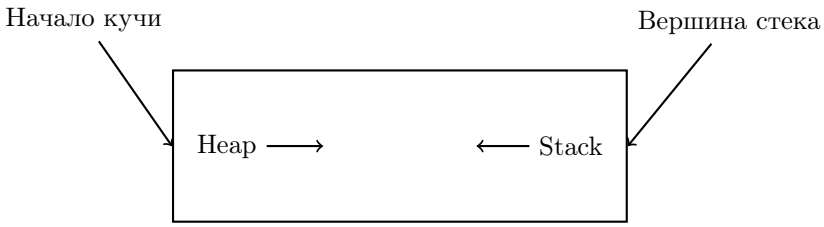
В процессоре ARM, тем не менее, есть поддержка стеков, растущих как в сторону уменьшения, так и в сторону увеличения.

Например, инструкции STMFD[2]/LDMFD[3], STMED[4]/LDMED[5] предназначены для descending-стека, т.е. уменьшающегося. Инструкции STMFA[6]/LMDFA[7], STMEA[8]/LDMEA[9] предназначены для ascending-стека, т.е. увеличивающегося.

4.1 Почему стек растет в обратную сторону?

Интуитивно мы можем подумать, что как и любая другая структура данных, стек мог бы расти вперед, т.е. в сторону увеличения адресов.

Причина, почему стек растет назад, вероятно, историческая. Когда компьютеры были большие и занимали целую комнату, было очень легко разделить сегмент на две части, для кучи и стека. Конечно, ведь заранее было неизвестно, насколько большой может быть куча или стек, так что это решение было самым простым.



В [26] можно прочитать:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected

[1] http://en.wikipedia.org/wiki/Call_stack
[2] Store Multiple Full Descending
[3] Load Multiple Full Descending
[4] Store Multiple Empty Descending
[5] Load Multiple Empty Descending
[6] Store Multiple Full Ascending
[7] Load Multiple Full Ascending
[8] Store Multiple Empty Ascending
[9] Load Multiple Empty Ascending

> and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

4.2 Для чего используется стек?

4.2.1 Сохранение адреса куда должно вернуться управление после вызова функции

x86

При вызове другой функции через `CALL` сначала в стек записывается адрес, указывающий на место аккурат после инструкции `CALL`, затем делается безусловный переход (почти как `JMP`) на адрес, указанный в операнде.

`CALL` — это аналог пары инструкций `PUSH address_after_call / JMP`.

`RET` вытаскивает из стека значение и передает управление по этому адресу — это аналог пары инструкций `POP tmp / JMP tmp`.

Крайне легко устроить переполнение стека, запустив бесконечную рекурсию:

```
void f()
{
        f();
};
```

MSVC 2008 предупреждает о проблеме:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause ↙
    ↳ runtime stack overflow
```

... но, тем не менее, создает нужный код:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

... причем, если включить оптимизацию (`/Ox`), то будет даже интереснее, без переполнения стека, но работать будет *корректно*[10]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                          ; f
```

GCC 4.4.1 генерирует точно такой же код в обоих случаях, хотя и не предупреждает о проблеме.

[10] здесь ирония

ARM

Программы для ARM также используют стек для сохранения RA[11], куда нужно вернуться, но несколько иначе. Как уже упоминалось в секции "Hello, world!" (2.3), RA записывается в регистр LR (link register). Но если есть необходимость вызывать какую-то другую функцию и использовать регистр LR еще раз, его значение желательно сохранить. Обычно это происходит в прологе функции, часто мы видим там инструкцию вроде `‘‘PUSH R4-R7,LR’’` , а в эпилоге `‘‘POP R4-R7,PC’’` — так сохраняются регистры, которые будут использоваться в текущей функции, в том числе LR.

Тем не менее, если некая функция не вызывает никаких более функций, в терминологии ARM она называется *leaf function*[12]. Как следствие, "leaf"-функция не использует регистр LR. А если эта функция небольшая, использует мало регистров, она может не использовать стек вообще. Таким образом, в ARM возможен вызов небольших leaf-функций не используя стек. Это может быть быстрее чем в x86, ведь внешняя память для стека не используется [13]. Либо это может быть полезным для тех ситуаций, когда память для стека еще не выделена либо недоступна.

4.2.2 Передача параметров для функции

Самый распространенный способ передачи параметров в x86 называется "cdecl":

```
push arg3
push arg2
push arg1
call f
add esp, 4*3
```

Вызываемая функция получает свои параметры также через указатель стека.

Следовательно, так будут расположены значения в стеке перед исполнением самой первой инструкции ф-ции f():

ESP	адрес возврата
ESP+4	аргумент#1, маркируется в IDA как `arg_0`
ESP+8	аргумент#2, маркируется в IDA как `arg_4`
ESP+0xC	аргумент#3, маркируется в IDA как `arg_8`
...	...

См. также в соответствующем разделе о других способах передачи аргументов через стек (44). Важно отметить, что, в общем, никто не заставляет программистов передавать параметры именно через стек, это не является требованием к исполняемому коду. Вы можете делать это совершенно иначе, не используя стек вообще.

К примеру, можно выделять в куче место для аргументов, заполнять их и передавать в функцию указатель на это место через `EAX`. И это вполне будет работать [14]. Однако, так традиционно сложилось, что в x86 и ARM передача аргументов происходит именно через стек.

Кстати, вызываемая ф-ция не имеет информации, сколько аргументов было ей было передано. Функции Си с переменным количеством аргументов (как `printf()`) определяют их количество по спецификатором строки формата (начинающиеся со знака %). Если написать что-то вроде

```
printf("%d %d %d", 1234);
```

`printf()` выведет 1234, затем еще два случайных числа, которые волею случая оказались в стеке рядом. Вот почему не так уж и важно, как объявлять ф-цию `main()`: как `main()`, `main(int argc, char *argv[])` либо `main(int argc, char *argv[], char *envp[])`.

В реальности, CRT-код вызывает `main()` примерно так:

```
push envp
push argv
push argc
```

[11] Адрес возврата

[12] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[13] Когда-то, очень давно, на PDP-11 и VAX на инструкцию CALL (вызов других функций) могло тратиться вплоть до 50% времени (возможно из-за работы с памятью), поэтому считалось, что много небольших функций это анти-паттерн [25, Chapter 4, Part II].

[14] Например, в книге Дональда Кнута "Искусство программирования", в разделе 1.4.1 посвященном подпрограммам [18, раздел 1.4.1], мы можем прочитать о возможности располагать параметры для вызываемой подпрограммы после инструкции `JMP`, передающей управление подпрограмме. Кнут описывает что это было особенно удобно для компьютеров System/360.

```
call main
...
```

Если вы объявляете `main()` как `main()` без аргументов, они, тем не менее, присутствуют в стеке, но не используются. Если вы объявите `main()` как `main(int argc, char *argv[])`, вы будете использовать два аргумента, а третий останется для вашей ф-ции "невидимым". Более того, можно даже объявить `main(int argc)`, и это будет работать.

4.2.3 Хранение локальных переменных

Функция может выделить для себя некоторое место в стеке для локальных переменных, просто отодвинув указатель стека глубже к концу стека.

Это снова не является необходимым требованием. Вы можете хранить локальные переменные где угодно. Но по традиции всё сложилось так.

4.2.4 x86: Функция alloca()

Интересен случай с функцией `alloca()`[15].

Эта функция работает как `malloc()`, но выделяет память прямо в стеке.

Память освобождать через `free()` не нужно, так как эпилог функции (3) вернет `ESP` назад в изначальное состояние и выделенная память просто аннулируется.

Интересна реализация функции `alloca()`.

Эта функция, если упрощенно, просто сдвигает `ESP` вглубь стека на столько байт, сколько вам нужно и возвращает `ESP` в качестве указателя на выделенный блок. Попробуем:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

(Функция `_snprintf()` работает так же, как и `printf()`, только вместо выдачи результата в stdout (т.е., на терминал или в консоль), записывает его в буфер `buf`. `puts()` выдает содержимое буфера `buf` в stdout. Конечно, можно было бы заменить оба этих вызова на один `printf()`, но мне нужно проиллюстрировать использование небольшого буфера.)

MSVC

Компилируем (MSVC 2010):

Листинг 4.1: MSVC 2010

```
...

    mov    eax, 600         ; 00000258H
    call   __alloca_probe_16
    mov    esi, esp
```

[15] В MSVC, реализацию функции можно посмотреть в файлах `alloca16.asm` и `chkstk.asm` в `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
    push   3
    push   2
    push   1
    push   OFFSET $SG2672
    push   600             ; 00000258H
    push   esi
    call   __snprintf

    push   esi
    call   _puts
    add    esp, 28         ; 0000001cH

...
```

Единственный параметр в `alloca()` передается через `EAX`, а не как обычно через стек [16]. После вызова `alloca()`, `ESP` теперь указывает на блок в 600 байт, который мы можем использовать под `buf`.

GCC + Синтаксис Intel

А GCC 4.4.1 обходится без вызова других функций:

Листинг 4.2: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                        ; выровнять указатель по 16-байтной границе
        mov     DWORD PTR [esp], ebx            ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600          ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx            ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + Синтаксис AT&T

Посмотрим на тот же код, только в синтаксисе AT&T:

Листинг 4.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
```

[16] Это потому, что alloca() — это не сколько функция, сколько т.н. *compiler intrinsic* (63).

Одна из причин, почему здесь нужна именно функция, а не несколько инструкций прямо в коде в том, что в реализации функции alloca() от MSVC[17] есть также код, читающий из только что выделенной памяти, чтобы ОС подключила физическую память к этому региону VM[18].

```
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Всё то же самое, что и в прошлом листинге.

N.B. Например, `movl $3, 20(%esp)` — это аналог `mov DWORD PTR [esp+20], 3` в Intel-синтаксисе: при адресации памяти в виде *регистр+смещение*, это записывается в AT&T синтаксисе как `смещение(%регистр)`.

4.2.5 (Windows) SEH

В стеке хранятся записи SEH[19] для функции (если они присутствуют).

Читайте больше о нем здесь: (48.3).

4.2.6 Защита от переполнений буфера

Здесь больше об этом (16.2).

4.3 Разметка типичного стека

Разметка типичного стека в 32-битной среде на момент начала ф-ции выглядит так:

...	...
ESP-0xC	локальная переменная #2, маркируется в IDA как `var_8`
ESP-8	локальная переменная #1, маркируется в IDA как `var_4`
ESP-4	сохраненное значение `EBP`
ESP	адрес возврата
ESP+4	аргумент#1, маркируется в IDA как `arg_0`
ESP+8	аргумент#2, маркируется в IDA как `arg_4`
ESP+0xC	аргумент#3, маркируется в IDA как `arg_8`
...	...

[19] Structured Exception Handling: 48.3

Глава 5

printf() с несколькими аргументами

Попробуем теперь немного расширить пример *Hello, world!* (2), написав в теле функции `main()`:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

5.1 x86: 3 аргумента

5.1.1 MSVC

Компилируем при помощи MSVC 2010 Express, и в итоге получим:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Все почти то же, за исключением того, что теперь видно, что аргументы для `printf()` заталкиваются в стек в обратном порядке: самый первый аргумент заталкивается последним.

Кстати, вспомним что переменные типа *int* в 32-битной системе, как известно, имеет ширину 32 бита, это 4 байта.

Итак, у нас всего 4 аргумента. $4 * 4 = 16$ — именно 16 байт занимают в стеке указатель на строку плюс еще 3 числа типа *int*.

Когда при помощи инструкции `‘‘ADD ESP, X’’` корректируется указатель стека `ESP` после вызова какой-либо функции, зачастую можно сделать вывод о том, сколько аргументов у вызываемой функции было, разделив X на 4.

Конечно, это относится только к cdecl-методу передачи аргументов через стек.

См. также в соответствующем разделе о способах передачи аргументов через стек (44).

Иногда бывает так, что подряд идут несколько вызовов разных функций, но стек корректируется только один раз, после последнего вызова:

```
push a1
push a2
call ...
...
push a1
call ...
```

```
...
push a1
push a2
push a3
call ...
add esp, 24
```

5.1.2 MSVC и OllyDbg

Попробуем этот же пример в OllyDbg. Это один из наиболее популярных win32-отладчиков user-режима. Мы можем компилировать наш пример в MSVC 2012 с опцией `/MD` что означает, линковать с библиотекой `MSVCR*.DLL`, чтобы импортируемые ф-ции были хорошо видны в отладчике.

Затем загружаем исполняемый файл в OllyDbg. Самый первый брякпойнт в `ntdll.dll`, нажмите F9 (запустить). Второй брякпойнт в CRT-коде. Теперь мы должны найти ф-цию `main()`.

Найдите этот код скроллируя окно кода до самого верха (MSVC располагает ф-цию `main()` в самом начале секции кода): илл. 5.3.

Кликните на инструкции `PUSH EBP`, нажмите F2 (установка брякпойнта) и нажмите F9 (запустить). Нам нужно произвести все эти манипуляции, чтобы пропустить CRT-код, потому что нам он пока не интересен.

Нажмите F8 (сделать шаг не входя в ф-цию) 6 раз, т.е., пропустить 6 инструкций: илл. 5.4.

Теперь PC указывает на инструкцию `CALL printf`. OllyDbg, как и другие отладчики, подсвечивает регистры со значениями, которые изменились. Так что, каждый раз, когда мы нажимаем, `EIP` изменяется и его значение подсвечивается красным. `ESP` также меняется, потому что значения заталкиваются в стек.

Где находятся эти значения в стеке? Посмотрите на правое/нижнее окно в отладчике:

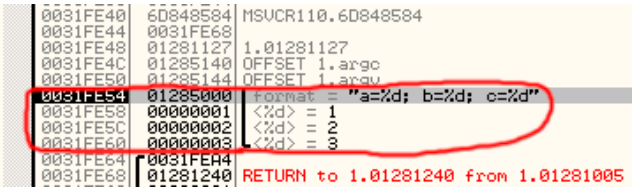


Рис. 5.1: OllyDbg: стек, после того как значения там сохранены (я сделал здесь округлую красную пометку в графическом редакторе)

Так что здесь видно 3 столбца: адрес в стеке, значение в стеке и еще дополнительный комментарий от OllyDbg. OllyDbg понимает `printf()`-строки, так что он показывает здесь и строку и 3 значения *привязанных* к ней.

Можно кликнуть правой кнопкой мыши на строке формата, кликнуть на “Follow in dump” и строка формата появится в окне слева внизу, где всегда виден какой-либо участок памяти. Эти значения в памяти можно редактировать. Можно изменить саму строку формата, и тогда результат работы нашего примера будет другой. В данном случае, пользы от этого немного, но для упражнения это полезно, чтобы начать чувствовать как тут всё работает.

Нажмите F8 (сделать шаг не входя в ф-цию).

В консоли мы видим вывод:

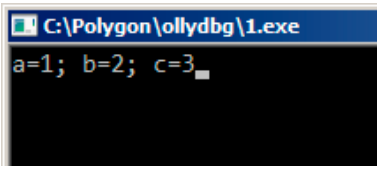


Рис. 5.2: Ф-ция `printf()` исполнилась

Посмотрим, как изменились регистры и состояние стека: илл. 5.5.

Регистр `EAX` теперь содержит `0xD` (13). That’s correct, `printf()` returns number of characters printed. Значение `EIP` изменилось: действительно, теперь здесь адрес инструкции после `CALL printf`. Значения регистров `ECX` и `EDX` также изменились. Очевидно, внутренности ф-ции `printf()` используют их для каких-то своих нужд.

Очень важный момент в том что значение ESP не изменилось. И состояние стека также! Мы ясно видим здесь и строку формата и соответствующие ей 3 значения, они все еще здесь. Действительно, по соглашению вызовов *cdecl*, вызывающая ф-ция не очищает аргументы из стека. Это должна делать вызывающая ф-ция.

Нажмите F8 снова, чтобы исполнилась инструкция ADD ESP, 10: илл. 5.6.

ESP изменился, но значения все еще в стеке! Конечно, никому не нужно заполнять эти значения нулями или что-то в этом роде. Потому что всё что выше указателя стека (SP) это *шум* или *мусор*, это всё не имеет особой ценности. Было бы очень затратно по времени очищать ненужные элементы стека, к тому же, никому это и не нужно.

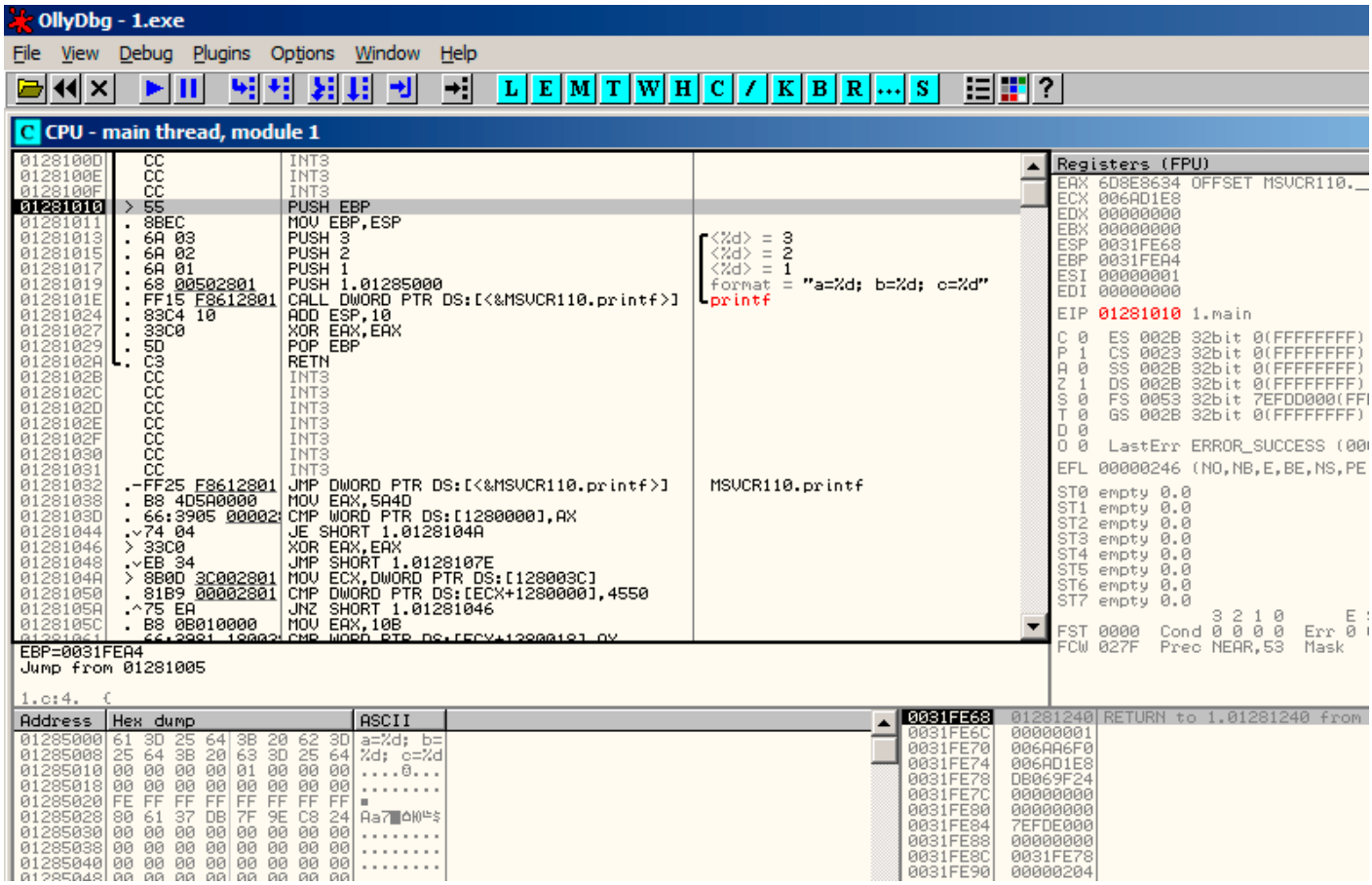


Рис. 5.3: OllyDbg: самое начало ф-ции main()

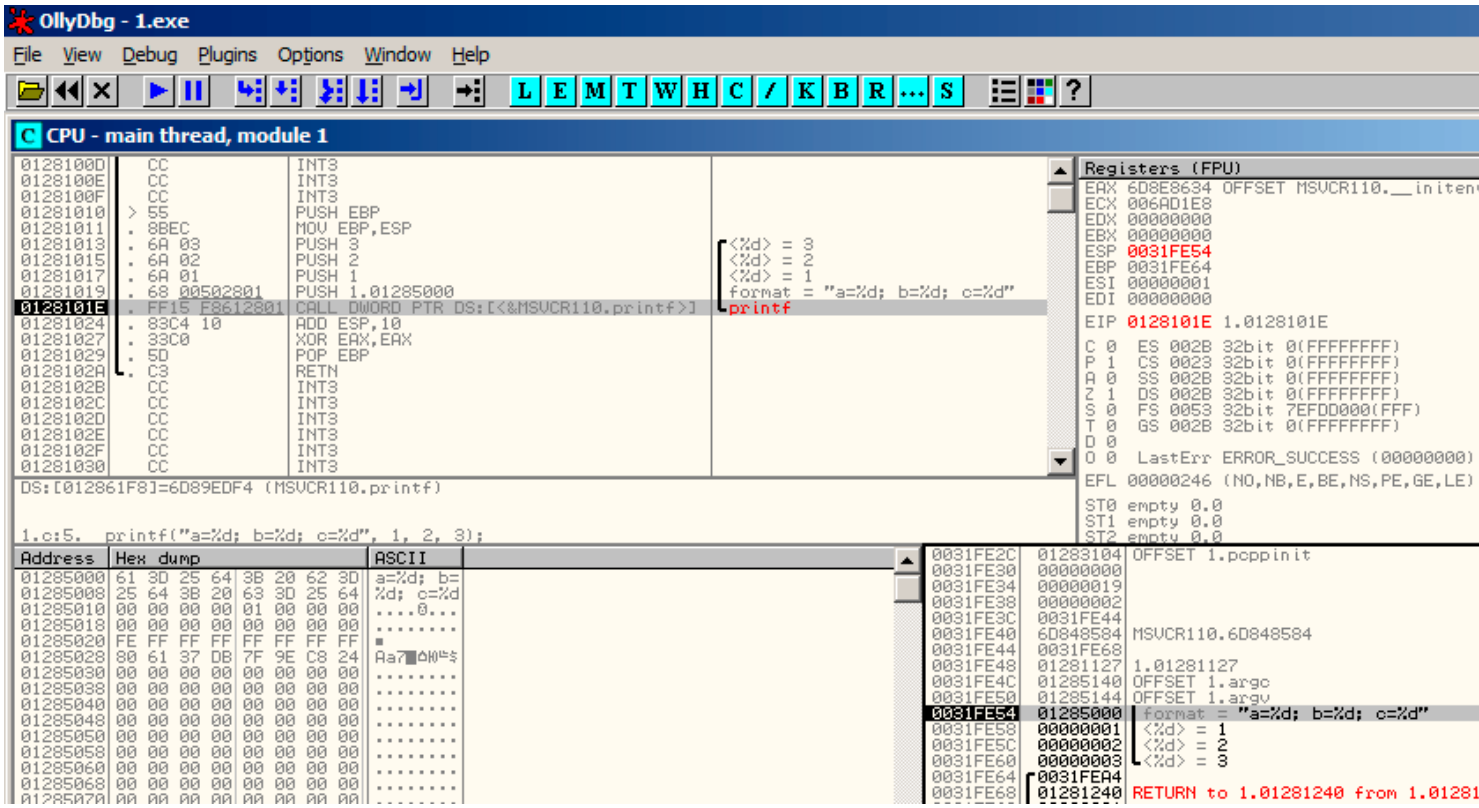


Рис. 5.4: OllyDbg: перед исполнением `printf()`

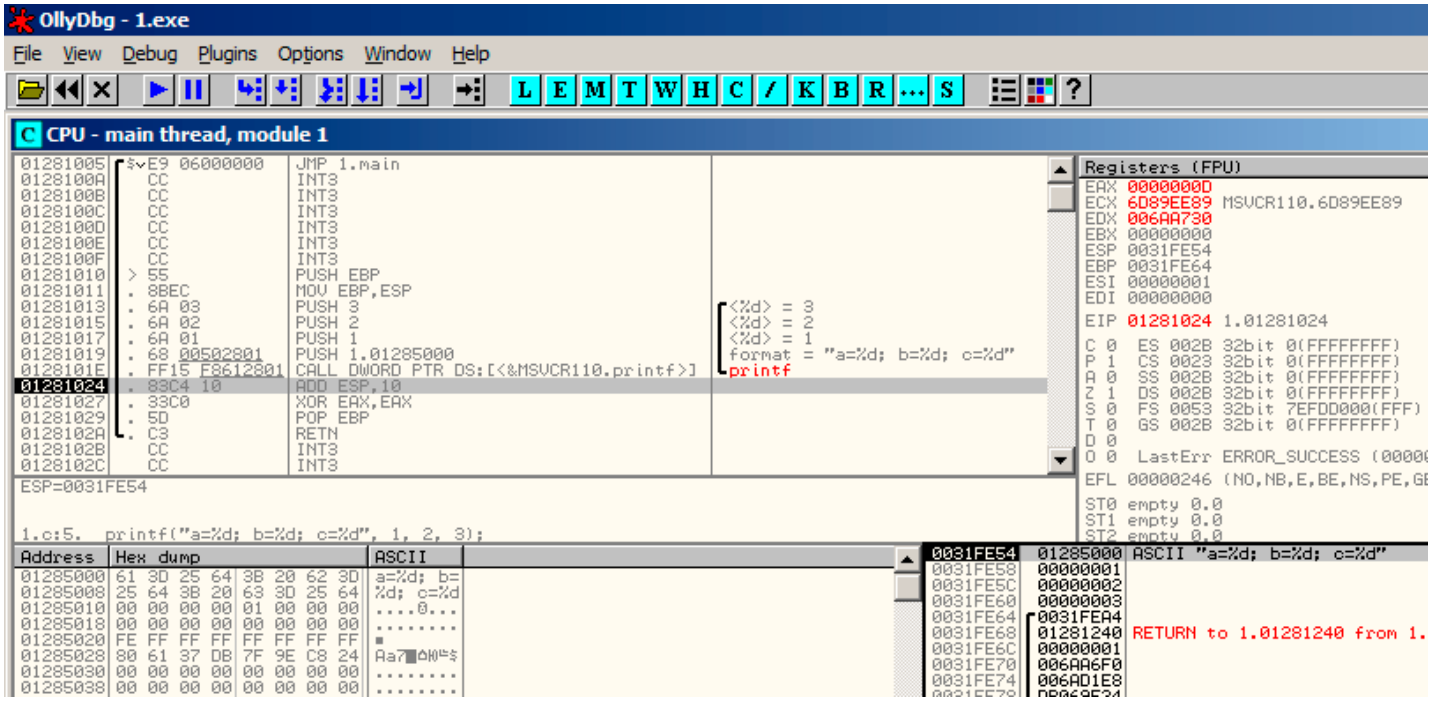


Рис. 5.5: OllyDbg: после исполнения `printf()`

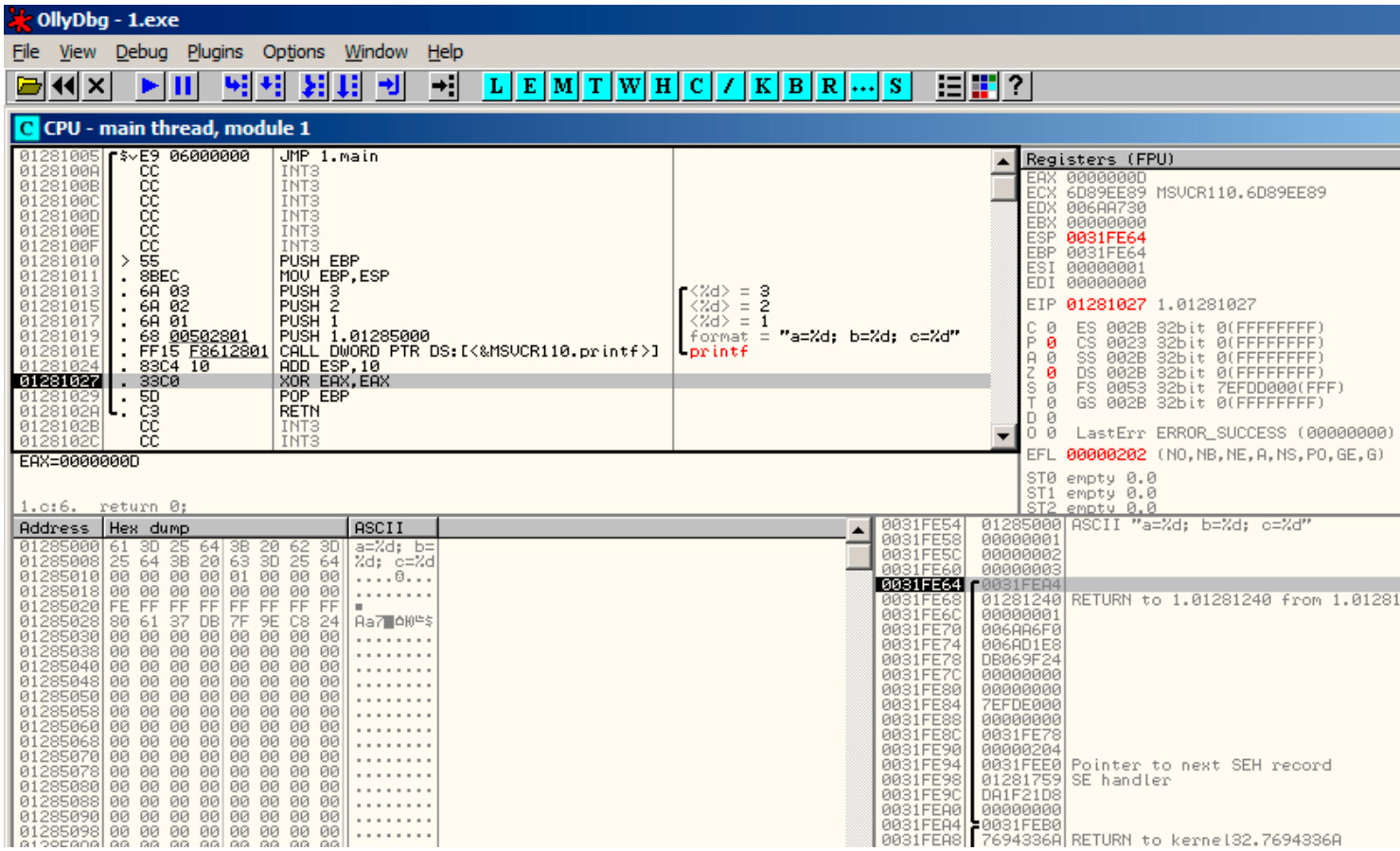


Рис. 5.6: OllyDbg: после исполнения инструкции `ADD ESP, 10`

5.1.3 GCC

Скомпилируем то же самое в Linux при помощи GCC 4.4.1 и посмотрим в IDA что вышло:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Можно сказать, что этот короткий код, созданный GCC, отличается от кода MSVC только способом помещения значений в стек. Здесь GCC снова работает со стеком напрямую без `PUSH`/`POP`.

5.1.4 GCC и GDB

Попробуем также этот пример и в GDB[1] в Linux.

`-g` означает генерировать отладочную информацию в выходном исполняемом файле.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

Листинг 5.1: установим брякпойнт на `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Запукаем. Здесь у нас нет исходного кода ф-ции `printf()` так что GDB не может показать его исходный код, но могла бы.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Выдать 10 элементов стека. Левый столбец это адрес в стеке.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Самый первый элемент это RA (`0x0804844a`). Мы можем удостовериться в этом, дизассемблируя память по этому адресу:

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Две инструкции `XCHG` это, вероятно, какой-то случайный мусор, который мы пока что можем игнорировать.

Второй элемент (`0x080484f0`) это адрес строки формата:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Остальные 3 элемента (1, 2, 3) это аргументы ф-ции `printf()`. Остальные элементы это может быть и мусор в стеке, но могут быть и значения от других ф-ций, их локальные переменные, итд. Пока что мы можем игнорировать их.

Исполняем “finish”. Это значит, исполнять до конца ф-ции. Здесь это означает: исполнять до завершения `printf()`.

[1] GNU debugger

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB показывает, что вернула `printf()` в `EAX` (13). Это, так же как и в примере с OllyDbg, количество напечатанных символов.

А еще мы видим "return 0;" и что это выражение находится в файле `1.c` в строке 6. Действительно, файл `1.c` лежит в текущем директории и GDB находит там эту строку. Как GDB знает, какая строка Си-кода сейчас исполняется? Это связано с тем фактом, что компилятор, генерируя отладочную информацию, также сохраняет информацию о соответствии строк в исходном коде и адресов инструкций. GDB это всё-таки отладчик уровня исходных текстов.

Посмотрим регистры. 13 в `EAX`:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Попробуем дизассемблировать текущие инструкции. Стрелка указывает на инструкцию, которая будет исполнена следующей.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB показывает дизассемблированный листинг в формате AT&T по умолчанию. Но можно также переключиться в формат Intel:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
```

```
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Исполняем следующую инструкцию. GDB покажет закрывающуюся скобку, означая, что это конец блока в ф-ции.

```
(gdb) step
7       };
```

Посмотрим регистры после исполнения инструкции `MOV EAX, 0`. `EAX` здесь уже действительно ноль.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

5.2 x64: 8 аргументов

Для того, чтобы посмотреть, как остальные аргументы будут передаваться через стек, изменим пример еще раз, увеличив количество передаваемых аргументов до 9 (строка формата `printf()` и 8 переменных типа *int*):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

5.2.1 MSVC

Как уже было сказано раннее, первые 4 аргумента в Win64 передаются в регистрах `RCX`, `RDX`, `R8`, `R9` , а остальные — через стек. Здесь мы это и видим. Впрочем, инструкция `PUSH` не используется, вместо нее, при помощи `MOV`, значения сразу записываются в стек.

Листинг 5.2: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; return 0
```

```
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

5.2.2 GCC

В *NIX-системах для x86-64 ситуация похожая, вот только первые 6 аргументов передаются через RDI, RSI, RDX, RCX, R8, R9. Остальные — через стек. GCC генерирует код записывающий указатель на строку в EDI вместо RDI— это мы уже рассмотрели чуть раньше: 2.2.2.

Почему перед вызовом printf() очищается регистр EAX, мы уже рассмотрели раннее: 2.2.2.

Листинг 5.3: GCC 4.4.6 -O3 x64

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; количество задействованных векторных регистров
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; return 0

        xor     eax, eax
        add     rsp, 40
        ret
```

5.2.3 GCC + GDB

Попробуем этот пример в GDB.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

Листинг 5.4: ставим брякпойнт на printf(), запускаем

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at ↙
    ↳ printf.c:29
29      printf.c: No such file or directory.
```

В регистрах RSI/RDX/RCX/R8/R9 всё предсказуемо. А RIP содержит адрес самой первой инструкции ф-ции printf().

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Листинг 5.5: смотрим на строку формата

```
(gdb) x/s $rdi
0x400628:       "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Дампим стек на этот раз с командой x/g — g означает *giant words*, т.е., 64-битные слова.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048      0x0000000100000000
```

Самый первый элемент стека, как и в прошлый раз, это RA. Через стек также передаются 3 значения: 6, 7, 8. Видно, что 8 передается с не очищенной старшей 32-битной частью: 0x00007fff00000008. Это нормально, ведь передаются числа типа *int*, а они 32-битные. Так что в старшей части регистра или памяти стека остался "случайный мусор".

GDB показывает всю ф-цию main(), если попытаться посмотреть, куда возвратится управление после исполнения printf():

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp,rsp
   0x0000000000400531 <+4>:     sub    rsp,0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp],0x6
```

```
   0x000000000040054c <+31>:    mov    r9d,0x5
   0x0000000000400552 <+37>:    mov    r8d,0x4
   0x0000000000400558 <+43>:    mov    ecx,0x3
   0x000000000040055d <+48>:    mov    edx,0x2
   0x0000000000400562 <+53>:    mov    esi,0x1
   0x0000000000400567 <+58>:    mov    edi,0x400628
   0x000000000040056c <+63>:    mov    eax,0x0
   0x0000000000400571 <+68>:    call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:    mov    eax,0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

Заканчиваем исполнение `printf()`, исполняем инструкцию обнуляющую `EAX`, удостоверяемся что в регистре `EAX` именно ноль. `RIP` указывает сейчас на инструкцию `LEAVE`, т.е., предпоследнюю в ф-ции `main()`.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%↙
    ↳ d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6               return 0;
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

5.3 ARM: 3 аргумента

В ARM традиционно принята такая схема передачи аргументов в функцию: 4 первых аргумента через регистры `R0-R3`; а остальные — через стек. Это немного похоже на то, как аргументы передаются в fastcall (44.3) или win64 (44.5.1).

5.3.1 Неоптимизирующий Keil + Режим ARM

Листинг 5.6: Неоптимизирующий Keil + Режим ARM

```
.text:00000014                printf_main1
.text:00000014 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000018 03 30 A0 E3    MOV     R3, #3
.text:0000001C 02 20 A0 E3    MOV     R2, #2
.text:00000020 01 10 A0 E3    MOV     R1, #1
.text:00000024 1D 0E 8F E2    ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
```

```
.text:00000028 0D 19 00 EB   BL      __2printf
.text:0000002C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

Итак, первые 4 аргумента передаются через регистры R0-R3, по порядку: указатель на формат-строку для printf() в R0, затем 1 в R1, 2 в R2 и 3 в R3.

Пока что здесь нет ничего необычного.

5.3.2 Оптимизирующий Keil + Режим ARM

Листинг 5.7: Оптимизирующий Keil + Режим ARM

```
.text:00000014              EXPORT printf_main1
.text:00000014             printf_main1
.text:00000014 03 30 A0 E3   MOV     R3, #3
.text:00000018 02 20 A0 E3   MOV     R2, #2
.text:0000001C 01 10 A0 E3   MOV     R1, #1
.text:00000020 1E 0E 8F E2   ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA   B       __2printf
```

Это оптимизированная версия (-O3) для режима ARM, и здесь мы видим последнюю инструкцию: B вместо привычной нам BL. Отличия между этой оптимизированной версией и предыдущей, скомпилированной без оптимизации, еще и в том, что здесь нет пролога и эпилога функции (инструкций, сохраняющих состояние регистров R0 и LR). Инструкция B просто переходит на другой адрес, без манипуляций с регистром LR, то есть это аналог JMP в x86. Почему это работает нормально? Потому что этот код эквивалентен предыдущему. Основных причин две: 1) стек не модифицируется, как и указатель стека SP; 2) вызов функции printf() последний, после него ничего не происходит. Функция printf(), отработав, просто вернет управление по адресу, записанному в LR. Но в LR находится адрес места, откуда была вызвана наша функция! А следовательно, управление из printf() вернется сразу туда. Следовательно, нет нужды сохранять LR, потому что нет нужны модифицировать LR. А нет нужды модифицировать LR, потому что нет иных вызовов функций, кроме printf(), к тому же, после этого вызова не нужно ничего здесь больше делать! Поэтому такая оптимизация возможна.

Еще один похожий пример описан в секции “switch()/case/default” , здесь (11.1.1).

5.3.3 Оптимизирующий Keil + Режим thumb

Листинг 5.8: Оптимизирующий Keil + Режим thumb

```
.text:0000000C             printf_main1
.text:0000000C 10 B5         PUSH    {R4,LR}
.text:0000000E 03 23         MOVS    R3, #3
.text:00000010 02 22         MOVS    R2, #2
.text:00000012 01 21         MOVS    R1, #1
.text:00000014 A4 A0         ADR     R0, aADBDCD     ; "a=%d; b=%d; c=%d\n"
.text:00000016 06 F0 EB F8   BL      __2printf
.text:0000001A 10 BD         POP     {R4,PC}
```

Здесь нет особых отличий от неоптимизированного варианта для режима ARM.

5.4 ARM: 8 аргументов

Снова воспользуемся примером с 9-ю аргументами из предыдущей секции: 5.2.

```
void printf_main2()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n",
        1, 2, 3, 4, 5, 6, 7, 8);
};
```

5.4.1 Оптимизирующий Keil: Режим ARM

```
.text:00000028                printf_main2
.text:00000028
.text:00000028                var_18 = -0x18
.text:00000028                var_14 = -0x14
.text:00000028                var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV    R3, #8
.text:00000034 07 20 A0 E3  MOV    R2, #7
.text:00000038 06 10 A0 E3  MOV    R1, #6
.text:0000003C 05 00 A0 E3  MOV    R0, #5
.text:00000040 04 C0 8D E2  ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV    R0, #4
.text:0000004C 00 00 8D E5  STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV    R3, #3
.text:00000054 02 20 A0 E3  MOV    R2, #2
.text:00000058 01 10 A0 E3  MOV    R1, #1
.text:0000005C 6E 0F 8F E2  ADR    R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g↙
    ↳ =%"...
.text:00000060 BC 18 00 EB  BL     __2printf
.text:00000064 14 D0 8D E2  ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR    PC, [SP+4+var_4],#4
```

Этот код можно условно разделить на несколько частей:

- Пролог функции:

  Самая первая инструкция ‘‘STR LR, [SP,#var_4]!’’ сохраняет в стеке LR, ведь нам придется использовать этот регистр для вызова printf().

  Вторая инструкция ‘‘SUB SP, SP, #0x14’’ уменьшает указатель стека SP, но, на самом деле, эта процедура нужна для выделения в локальном стеке места размером 0x14 (20) байт. Действительно, нам нужно передать 5 32-битных значений через стек в printf(), каждое значение занимает 4 байта, а $5 * 4 = 20$ — как раз. Остальные 4 32-битных значения будут переданы через регистры.

- Передача 5, 6, 7 и 8 через стек:

  Затем значения 5, 6, 7 и 8 записываются в регистры R0, R1, R2 и R3 соответственно. Затем инструкция ‘‘ADD R12, SP, #0x18+var_14’’ записывает в регистр R12 адрес места в стеке, куда будут помещены эти 4 значения. *var_14* — это макрос ассемблера, равный $-0x14$, такие макросы создает IDA, чтобы удобнее было показывать, как код обращается к стеку. Макросы *var_?*, создаваемые IDA, отражают локальные переменные в стеке. Так что в R12 будет записано $SP + 4$. Следующая инструкция ‘‘STMIA R12, R0-R3’’ записывает содержимое регистров R0-R3 по адресу в памяти, на который указывает R12. Инструкция STMIA означает *Store Multiple Increment After*. *Increment After* означает, что R12 будет увеличиваться на 4 после записи каждого значения регистра.

- Передача 4 через стек: 4 записывается в R0, затем это значение при помощи инструкции ‘‘STR R0, [SP,#0x18+var_18]’’ попадает в стек. *var_18* равен $-0x18$, смещение будет 0, так что, значение из регистра R0 (4) запишется туда, куда указывает SP.

- Передача 1, 2 и 3 через регистры:

  Значения для первых трех чисел (a, b, c) (1, 2, 3 соответственно) передаются в регистрах R1, R2 и R3 перед самим вызовом printf(), а остальные 5 значений передаются через стек, и вот как:

- Вызов printf():

- Эпилог функции:

  Инструкция ‘‘ADD SP, SP, #0x14’’ возвращает SP на прежнее место, аннулируя таким образом всё, что было записано в стеке. Конечно, то что было записано в стек, там пока и останется, но всё это будет многократно перезаписано во время исполнения последующих функций.

  Инструкция ‘‘LDR PC, [SP+4+var_4],#4’’ загружает в PC сохраненное значение LR из стека, обеспечивая таким образом выход из функции.

5.4.2 Оптимизирующий Keil: Режим thumb

```
.text:0000001C             printf_main2
.text:0000001C
.text:0000001C             var_18 = -0x18
.text:0000001C             var_14 = -0x14
.text:0000001C             var_8  = -8
.text:0000001C
.text:0000001C 00 B5        PUSH    {LR}
.text:0000001E 08 23        MOVS    R3, #8
.text:00000020 85 B0        SUB     SP, SP, #0x14
.text:00000022 04 93        STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22        MOVS    R2, #7
.text:00000026 06 21        MOVS    R1, #6
.text:00000028 05 20        MOVS    R0, #5
.text:0000002A 01 AB        ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3        STMIA   R3!, {R0-R2}
.text:0000002E 04 20        MOVS    R0, #4
.text:00000030 00 90        STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23        MOVS    R3, #3
.text:00000034 02 22        MOVS    R2, #2
.text:00000036 01 21        MOVS    R1, #1
.text:00000038 A0 A0        ADR     R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; ↙
    ↳ g=%"...
.text:0000003A 06 F0 D9 F8  BL      __2printf
.text:0000003E
.text:0000003E             loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0        ADD     SP, SP, #0x14
.text:00000040 00 BD        POP     {PC}
```

Это почти то же самое, что и в предыдущем примере, только код для thumb и значения помещаются в стек немного иначе: в начале 8 за первый раз, затем 5, 6, 7 за второй раз и 4 за третий раз.

5.4.3 Оптимизирующий Xcode (LLVM): Режим ARM

```
__text:0000290C             _printf_main2
__text:0000290C
__text:0000290C             var_1C = -0x1C
__text:0000290C             var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3   MOV    R3, #6
__text:00002930 05 10 A0 E3   MOV    R1, #5
__text:00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV    R9, #8
__text:00002940 01 10 A0 E3   MOV    R1, #1
__text:00002944 02 20 A0 E3   MOV    R2, #2
__text:00002948 03 30 A0 E3   MOV    R3, #3
__text:0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB   BL     _printf
__text:00002954 07 D0 A0 E1   MOV    SP, R7
__text:00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Почти то же самое, что мы уже видели, за исключением того, что STMFA (Store Multiple Full Ascending) — это синоним инструкции STMIB (Store Multiple Increment Before) . Эта инструкция увеличивает SP и только затем записывает в память значение очередного регистра, но не наоборот.

Второе, что бросается в глаза, это то, что инструкции как будто бы расположены случайно. Например, значение в регистре R0 подготавливается в трех местах, по адресам 0x2918, 0x2920 и 0x2928, когда это можно было бы сделать в одном месте. Однако, у оптимизирующего компилятора могут быть свои доводы о том, как лучше составлять инструкции друг с другом для лучшей эффективности исполнения. Процессор обычно пытается исполнять одновременно идущие друг за другом инструкции. К примеру, инструкции ‘‘MOVT R0, #0’’ и ‘‘ADD R0, PC, R0’’ не могут быть исполнены одновременно, потому что обе инструкции модифицируют регистр R0. А вот инструкции ‘‘MOVT R0, #0’’ и ‘‘MOV R2, #4’’ легко можно исполнить одновременно, потому что эффекты от их исполнения никак не конфликтуют друг с другом. Вероятно, компилятор старается генерировать код именно таким образом там, где это возможно.

5.4.4 Оптимизирующий Xcode (LLVM): Режим thumb-2

```
__text:00002BA0                 _printf_main2
__text:00002BA0
__text:00002BA0                 var_1C = -0x1C
__text:00002BA0                 var_18 = -0x18
__text:00002BA0                 var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5            PUSH     {R7,LR}
__text:00002BA2 6F 46            MOV      R7, SP
__text:00002BA4 85 B0            SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20      MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C      MOV.W    R12, #7
__text:00002BAE C0 F2 00 00      MOVT.W   R0, #0
__text:00002BB2 04 22            MOVS     R2, #4
__text:00002BB4 78 44            ADD      R0, PC  ; char *
__text:00002BB6 06 23            MOVS     R3, #6
__text:00002BB8 05 21            MOVS     R1, #5
__text:00002BBA 0D F1 04 0E      ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92            STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09      MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10      STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21            MOVS     R1, #1
__text:00002BCA 02 22            MOVS     R2, #2
__text:00002BCC 03 23            MOVS     R3, #3
__text:00002BCE CD F8 10 90      STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA      BLX      _printf
__text:00002BD6 05 B0            ADD      SP, SP, #0x14
__text:00002BD8 80 BD            POP      {R7,PC}
```

Почти то же самое, что и в предыдущем примере, лишь за тем исключением, что здесь используются thumb-инструкции.

5.5 Кстати

Кстати, разница между способом передачи параметров принятая в x86, x64, fastcall и ARM неплохо иллюстрирует тот важный момент, что процессору, в общем, все равно, как будут передаваться параметры функций. Можно создать гипотетический компилятор, который будет передавать их при помощи указателя на структуру с параметрами, не пользуясь стеком вообще.

Глава 6

scanf()

Теперь попробуем использовать scanf().

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Да, согласен, использовать `scanf()` в наши времена для того, чтобы спросить у пользователя что-то, не самая хорошая идея. Но я хотел проиллюстрировать передачу указателя на *int*.

6.1 Об указателях

Это одна из фундаментальных вещей в компьютерных науках. Часто большой массив, структуру или объект передавать в другую функцию никак не выгодно, а передать её адрес куда проще. К тому же, если вызываемая функция должна изменить что-то в этом большом массиве или структуре, то возвращать её полностью — это так же абсурдно. Так что самое простое, что можно сделать, это передать в функцию адрес массива или структуры, и пусть она что-то там изменит.

Указатель в Си/Си++ — это просто адрес какого-либо места в памяти.

В x86 адрес представляется в виде 32-битного числа (т.е., занимает 4 байта), а в x86–64 как 64-битное число (занимает 8 байт). Кстати, отсюда негодование некоторых людей, связанное с переходом на x86-64 — на этой архитектуре все указатели будут занимать места в 2 раза больше.

При некотором упорстве можно работать только с бестиповыми указателями (`void*`); например, стандартная функция Си `memcpy()`, копирующая блок из одного места памяти в другое, принимает на вход 2 указателя типа `void*`, потому что нельзя заранее предугадать, какого типа блок вы собираетесь копировать. Да в общем это и не важно, важно только знать размер блока.

Также указатели широко используются, когда функции нужно вернуть более одного значения (мы еще вернемся к этому в будущем (9)). *scanf()* — это как раз такой случай. Помимо того, что этой функции нужно показать, сколько значений было прочитано успешно, ей еще и нужно вернуть сами значения.

Тип указателя в Си/Си++ нужен для проверки типов на стадии компиляции. Внутри, в скомпилированном коде, никакой информации о типах указателей нет.

6.2 x86

6.2.1 MSVC

Что получаем на ассемблере компилируя MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; return 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Переменная `x` является локальной.

По стандарту Си/Си++ она доступна только из этой же функции и ниоткуда более. Так получилось, что локальные переменные располагаются в стеке. Может быть, можно было бы использовать и другие варианты, но в x86 это традиционно так.

Следующая после пролога инструкция `PUSH ECX` не ставит своей целью сохранить значение регистра `ECX`. (Заметьте отсутствие соответствующей инструкции `POP ECX` в конце функции)

Она на самом деле выделяет в стеке 4 байта для хранения `x` в будущем.

Доступ к `x` будет осуществляться при помощи объявленного макроса `_x$` (он равен -4) и регистра `EBP` указывающего на текущий фрейм.

Вообще, во все время исполнения функции, `EBP` указывает на текущий фрейм и через `EBP+смещение` можно иметь доступ как к локальным переменным функции, так и аргументам функции.

Можно было бы использовать `ESP`, но он во время исполнения функции постоянно меняется. Так что можно сказать, что `EBP` это *замороженное состояние* `ESP` на момент начала исполнения функции.

Разметка типичного стекового фрейма в 32-битной среде:

...	...
EBP-8	локальная переменная #2, маркируется в IDA как `var_8`
EBP-4	локальная переменная #1, маркируется в IDA как `var_4`
EBP	сохраненное значение `EBP`
EBP+4	адрес возврата
EBP+8	аргумент#1, маркируется в IDA как `arg_0`
EBP+0xC	аргумент#2, маркируется в IDA как `arg_4`
EBP+0x10	аргумент#3, маркируется в IDA как `arg_8`
...	...

У функции `scanf()` в нашем примере два аргумента.

Первый — указатель на строку содержащую "`%d`" и второй — адрес переменной `x`.

Вначале адрес `x` помещается в регистр `EAX` при помощи инструкции `lea eax, DWORD PTR _x$[ebp]`.

Инструкция `LEA` означает *load effective address*, но со временем она изменила свою функцию (B.6.2).

Можно сказать, что в данном случае `LEA` просто помещает в `EAX` результат суммы значения в регистре `EBP` и макроса `_x$`.

Это тоже что и `lea eax, [ebp-4]`.

Итак, от значения `EBP` отнимается 4 и помещается в `EAX`. Далее значение `EAX` заталкивается в стек и вызывается `scanf()`.

После этого вызывается `printf()`. Первый аргумент вызова которого, строка: "`You entered %d...\n`".

Второй аргумент: `mov ecx, [ebp-4]`, эта инструкция помещает в `ECX` не адрес переменной `x`, а его значение, что там сейчас находится.

Далее значение `ECX` заталкивается в стек и вызывается последний `printf()`.

6.2.2 MSVC + OllyDbg

Попробуем этот же пример в OllyDbg. Загружаем, нажимаем F8 (сделать шаг не входя в ф-цию) до тех пор, пока не окажемся в своем исполняемом файле, а не в `ntdll.dll`. Скроллим вверх, до тех пока не найдем `main()`. Кликаем на первой инструкции (`PUSH EBP`), нажимаем F2, затем F9 (Run) и брякпойнт срабатывает на начале `main()`.

Трассируем до того места, где готовится адрес переменной x: илл. 6.2.

На `EAX` в окне регистров можно нажать правой кнопкой и далее "Follow in stack". Этот адрес покажется в окне стека. Смотрите, это переменная в локальном стеке. Я нарисовал там красную стрелку. И там сейчас какой-то мусор (`0x77D478`). Адрес этого элемента стека сейчас, при помощи `PUSH`, запишется в этот же стек, рядом. Трассируем при помощи F8 вплоть до конца исполнения `scanf()`. А пока `scanf()` исполняется, в консольном окне, вводим, например, 123:

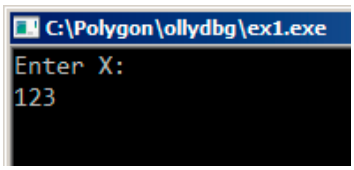


Рис. 6.1: Вывод в консоль

Вот тут `scanf()` отработал: илл. 6.3. `scanf()` вернул 1 в `EAX`, что означает, что он успешно прочитал одно значение. В наблюдаемом нами элементе стека теперь `0x7B` (123).

Чуть позже, это значение копируется из стека в регистр `ECX` и передается в `printf()`: илл. 6.4.

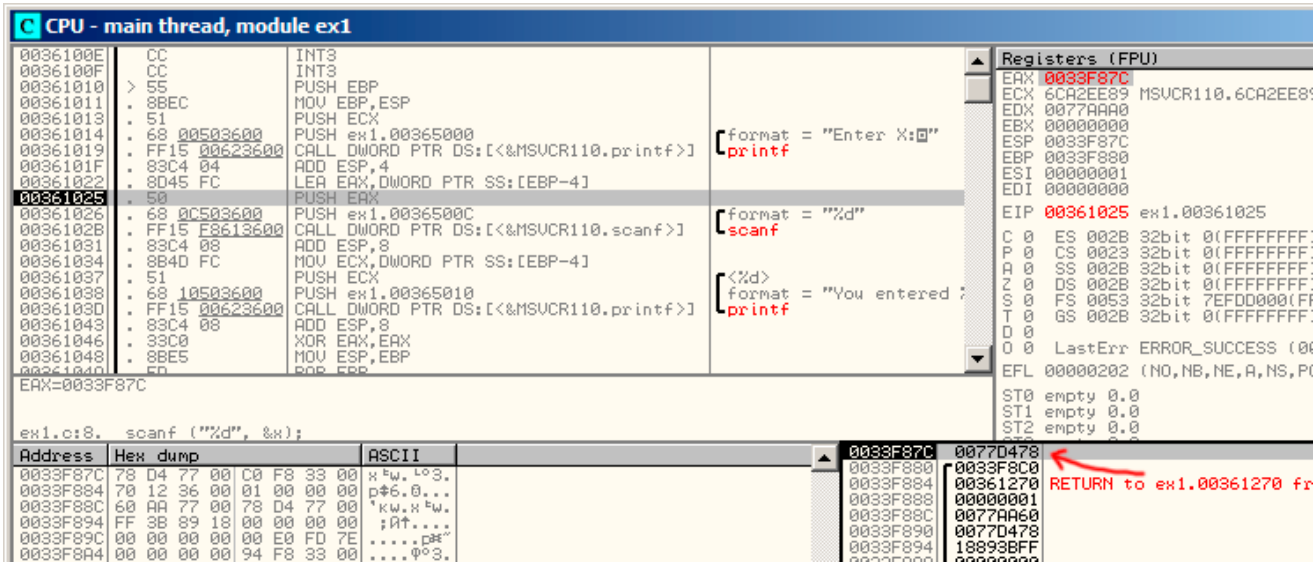


Рис. 6.2: OllyDbg: вычисляется адрес локальной переменной

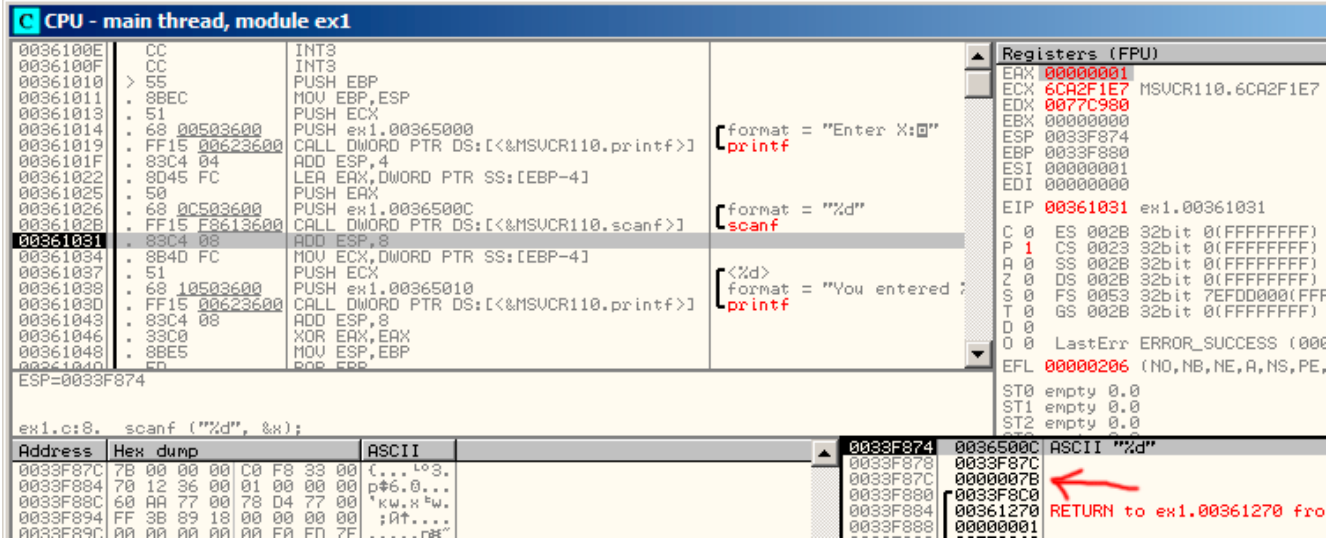


Рис. 6.3: OllyDbg: `scanf()` исполнилась

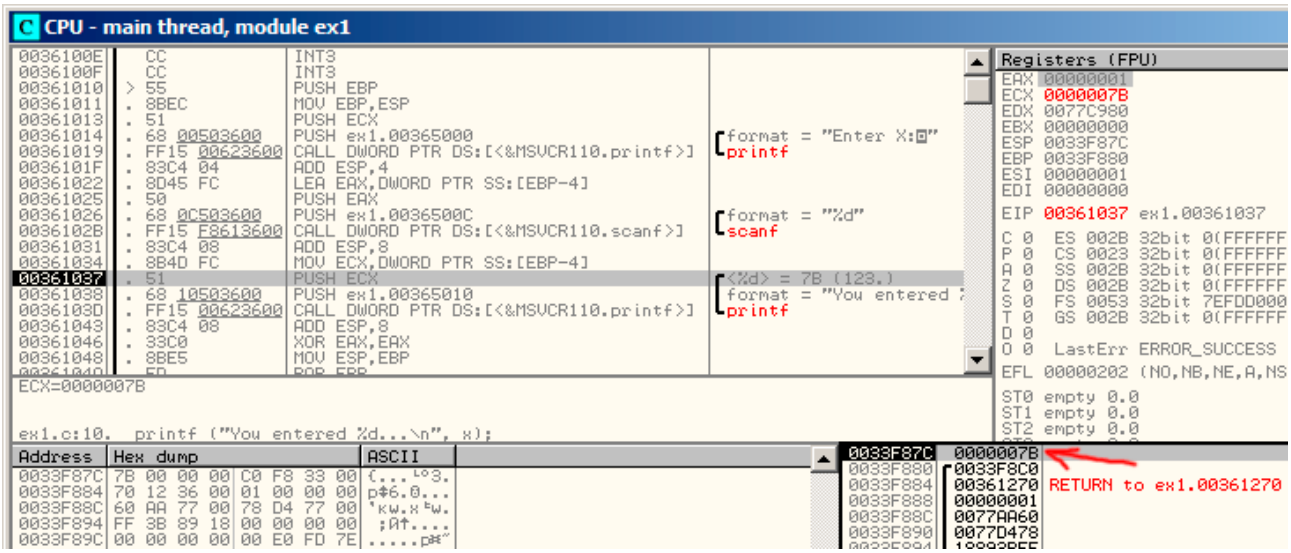


Рис. 6.4: OllyDbg: готовим значение для передачи в `printf()`

6.2.3 GCC

Попробуем тоже самое скомпилировать в Linux при помощи GCC 4.4.1:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
```

```
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

GCC заменил первый вызов `printf()` на `puts()`, почему это было сделано, уже было описано раннее (2.3.3). Далее все как и прежде — параметры заталкиваются через стек при помощи `MOV`.

6.3 x64

Всё то же самое, только используются регистры вместо стека для передачи аргументов функций.

6.3.1 MSVC

Листинг 6.1: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
```

6.3.2 GCC

Листинг 6.2: GCC 4.4.6 -O3 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
```

```
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 24
        ret
```

6.4 ARM

6.4.1 Оптимизирующий Keil + Режим thumb

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8           = -8
.text:00000042
.text:00000042 08 B5                       PUSH    {R3,LR}
.text:00000044 A9 A0                       ADR     R0, aEnterX     ; "Enter X:\n"
.text:00000046 06 F0 D3 F8                 BL      __2printf
.text:0000004A 69 46                       MOV     R1, SP
.text:0000004C AA A0                       ADR     R0, aD          ; "%d"
.text:0000004E 06 F0 CD F8                 BL      __0scanf
.text:00000052 00 99                       LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0                       ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000056 06 F0 CB F8                 BL      __2printf
.text:0000005A 00 20                       MOVS    R0, #0
.text:0000005C 08 BD                       POP     {R3,PC}
```

Чтобы `scanf()` мог вернуть значение, ему нужно передать указатель на переменную типа *int*. *int* — 32-битное значение, для его хранения нужно только 4 байта, и оно помещается в 32-битный регистр. Место для локальной переменной `x` выделяется в стеке, IDA наименовала её *var_8*, впрочем, место для нее выделять не обязательно, т.к., указатель стека SP уже указывает на место, свободное для использования сразу же. Так что значение указателя SP копируется в регистр `R1`, и вместе с format-строкой, передается в `scanf()`. Позже, при помощи инструкции `LDR`, это значение перемещается из стека в регистр `R1`, чтобы быть переданным в `printf()`.

Варианты, скомпилированные для ARM-режима процессора, а также варианты скомпилированные при помощи Xcode LLVM, не очень отличаются от этого, так что, мы можем пропустить их здесь.

6.5 Глобальные переменные

А что если переменная `x` из предыдущего примера будет глобальной переменной, а не локальной? Тогда к ней смогут обращаться из любого другого места, а не только из тела функции. Глобальные переменные считаются анти-паттерном, но ради примера мы можем себе это позволить.

```
#include <stdio.h>

int x;

int main()
{
        printf ("Enter X:\n");
```

```
        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

6.5.1 MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Ничего особенного, в целом. Теперь `x` объявлена в сегменте `_DATA`. Память для нее в стеке более не выделяется. Все обращения к ней происходит не через стек, а уже напрямую. Неинициализированные глобальные переменные не занимают места в исполняемом файле (и действительно, зачем в исполняемом файле нужно выделять место под изначально нулевые переменные?), но тогда, когда к этому месту в памяти кто-то обратится, ОС подставит туда блок состоящий из нулей[1].

Попробуем изменить объявление этой переменной:

```
int x=10; // default value
```

Выйдет в итоге:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Здесь уже по месту этой переменной записано `0xA` с типом DD (dword = 32 бита).

Если вы откроете скомпилированный .exe-файл в IDA, то увидите что x находится аккурат в начале сегмента `_DATA`, после этой переменной будут текстовые строки.

[1] Так работает VM

А вот если вы откроете в IDA, .exe скомпилированный в прошлом примере, где значение x не определено, то в IDA вы увидите:

```
.data:0040FA80 _x              dd ?                    ; DATA XREF: _main+10
.data:0040FA80                                         ; _main+22
.data:0040FA84 dword_40FA84    dd ?                    ; DATA XREF: _memset+1E
.data:0040FA84                                         ; unknown_libname_1+28
.data:0040FA88 dword_40FA88    dd ?                    ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                                         ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem           dd ?                    ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                                         ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90    dd ?                    ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                                         ; __calloc_impl+72
.data:0040FA94 dword_40FA94    dd ?                    ; DATA XREF: ___sbh_free_block+2FE
```

`_x` обозначен как `?`, наряду с другими переменными не требующими инициализации. Это означает, что при загрузке .exe в память, место под все это выделено будет. Но в самом .exe ничего этого нет. Неинициализированные переменные не занимают места в исполняемых файлах. Удобно для больших массивов, например.

6.5.2 MSVC: x86 + OllyDbg

Тут даже проще: илл. 6.5. Переменная хранится в сегменте данных. Кстати, после исполнения инструкции `PUSH`, заталкивающей адрес x, адрес появится в стеке, и на этом элементе можно нажать правой кнопкой, выбрать “Follow in dump”. И в окне памяти слева появится эта переменная.

После того как в консоли введем 123, здесь появится `0x7B`.

Почему самый первый байт это `7B`? По логике вещей, здесь должно было бы быть `00 00 00 7B`. Это называется endianness, и в x86 принят формат *little-endian*. Это означает что в начале записывается самый младший байт, а заканчивается самым старшим байтом. Больше об этом: 33.

Позже, из этого места в памяти, 32-битное значение загружается в `EAX` и передается в `printf()`.

Адрес переменной x в памяти `0xDC3390`. В OllyDbg мы можем посмотреть карту памяти процесса (Alt-M) и увидим что этот адрес внутри PE-сегмента `.data` нашей программы: илл. 6.6.

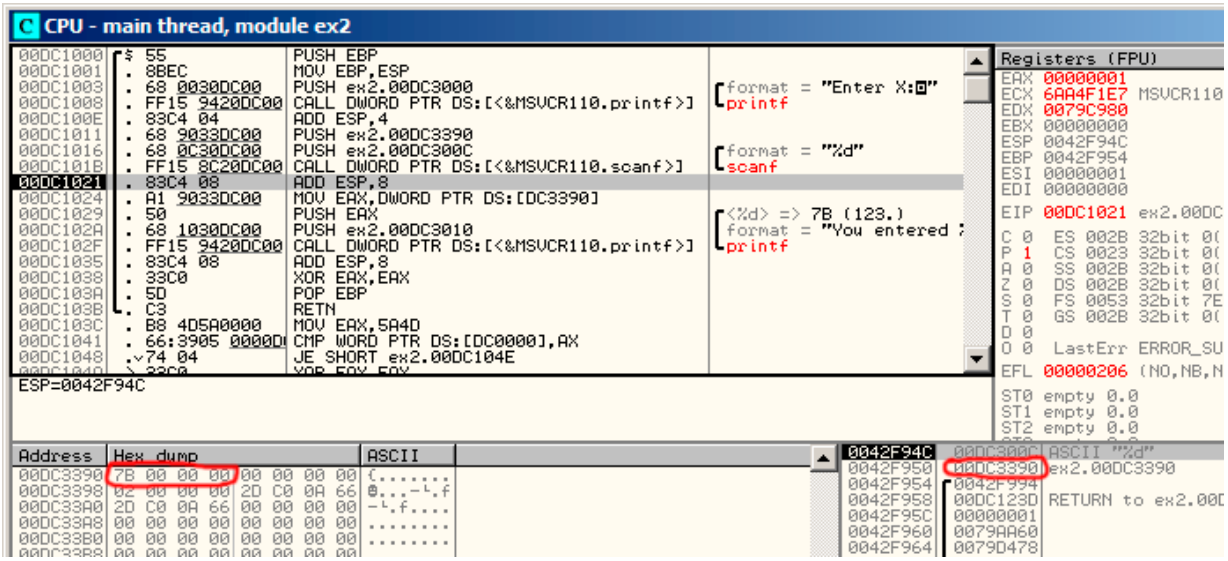


Рис. 6.5: OllyDbg: после исполнения `scanf()`

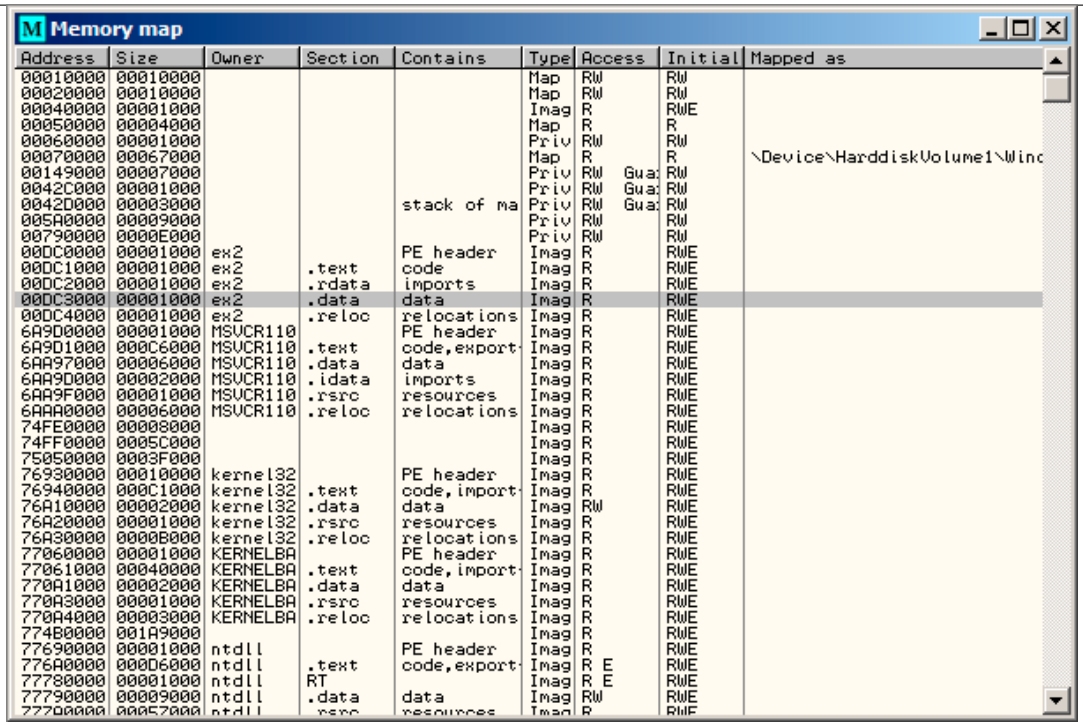


Рис. 6.6: OllyDbg: карта памяти процесса

6.5.3 GCC: x86

В Linux все также почти. За исключением того, что если значение `x` не определено, то эта переменная будет находится в сегменте `_bss`. В ELF этот сегмент имеет такие атрибуты:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Ну а если сделать статическое присвоение этой переменной какого-либо значения, например, 10, то она будет находится в сегменте `_data`, это сегмент с такими атрибутами:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

6.5.4 MSVC: x64

Листинг 6.3: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
```

```
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Почти такой же код как и в x86. Обратите внимание что для `scanf()` адрес переменной x передается при помощи инструкции `LEA`, а во второй `printf()` передается само значение переменной при помощи `MOV`. ‘‘`DWORD PTR`’’ — это часть языка ассемблера (не имеющая отношения к машинным кодам) показывающая, что тип переменной в памяти — именно 32-битный, и инструкция `MOV` должна быть здесь закодирована соответственно.

6.5.5 ARM: Оптимизирующий Keil + Режим thumb

```
.text:00000000 ; Segment type: Pure code
.text:00000000                 AREA .text, CODE
...
.text:00000000 main
.text:00000000                 PUSH    {R4,LR}
.text:00000002                 ADR     R0, aEnterX     ; "Enter X:\n"
.text:00000004                 BL      __2printf
.text:00000008                 LDR     R1, =x
.text:0000000A                 ADR     R0, aD          ; "%d"
.text:0000000C                 BL      __0scanf
.text:00000010                 LDR     R0, =x
.text:00000012                 LDR     R1, [R0]
.text:00000014                 ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016                 BL      __2printf
.text:0000001A                 MOVS    R0, #0
.text:0000001C                 POP     {R4,PC}
...
.text:00000020 aEnterX         DCB "Enter X:",0xA,0    ; DATA XREF: main+2
.text:0000002A                 DCB    0
.text:0000002B                 DCB    0
.text:0000002C off_2C          DCD x                   ; DATA XREF: main+8
.text:0000002C                                         ; main+10
.text:00000030 aD              DCB "%d",0              ; DATA XREF: main+A
.text:00000033                 DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF: main+14
.text:00000047                 DCB 0
.text:00000047 ; .text         ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048                 AREA .data, DATA
.data:00000048                 ; ORG 0x48
.data:00000048                 EXPORT x
.data:00000048 x               DCD 0xA                 ; DATA XREF: main+8
.data:00000048                                         ; main+10
.data:00000048 ; .data         ends
```

Итак, переменная `x` теперь глобальная, и она расположена, почему-то, в другом сегменте, а именно сегменте данных (*.data*). Можно спросить, почему текстовые строки расположены в сегменте кода (*.text*) а `x` нельзя было разместить тут же? Потому что эта переменная, и как следует из определения, она может меняться. И может даже быть, меняться часто. Сегмент кода нередко может быть расположен в ПЗУ микроконтроллера (не забывайте, мы сейчас имеем дело с embedded-микроэлектроникой, где дефицит памяти — это обычное дело), а изменяемые переменные — в ОЗУ. Хранить в ОЗУ неизменяемые данные, когда в наличии есть ПЗУ, не

экономно. К тому же, сегмент данных в ОЗУ с константами нужно было бы инициализировать перед работой, ведь, после включения ОЗУ, очевидно, она содержит в себе случайную информацию.

Далее, мы видим, в сегменте кода, хранится указатель на переменную `x` (`off_2C`) и вообще, все операции с переменной, происходят через этот указатель. Это связано с тем что переменная `x` может быть расположена где-то довольно далеко от данного участка кода, так что её адрес нужно сохранить в непосредственной близости к этому коду. Инструкция `LDR` в thumb-режиме может адресовать только переменные в пределах вплоть до 1020 байт от места где она находится. Эта же инструкция в ARM-режиме — переменные в пределах ±4095 байт, таким образом, адрес глобальной переменной `x` нужно расположить в непосредственной близости, ведь нет никакой гарантии, что компоновщик[2] сможет разместить саму переменную где-то рядом, она может быть даже в другом чипе памяти!

Еще одна вещь: если переменную объявить как *const*, то компилятор Keil разместит её в сегменте `.constdata`. Должно быть, впоследствии, компоновщик и этот сегмент сможет разместить в ПЗУ, вместе с сегментом кода.

6.6 Проверка результата scanf()

Как я уже упоминал, использовать `scanf()` в наше время это слегка старомодно. Но если уж жизнь заставила этим заниматься, нужно хотя бы проверять, сработал ли `scanf()` правильно или пользователь ввел вместо числа что-то другое, что `scanf()` не смог трактовать как число.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

По стандарту, `scanf()`[3] возвращает количество успешно полученных значений.

В нашем случае, если все успешно и пользователь ввел таки некое число, `scanf()` вернет 1. А если нет, то 0 или EOF.

Я добавил код проверяющий результат `scanf()` и в случае ошибки, он сообщает пользователю что-то другое.

Это работает предсказуемо:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

6.6.1 MSVC: x86

Вот, что выходит на ассемблере (MSVC 2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
```

[2]linker в англоязычной литературе

[3]MSDN: scanf, wscanf: http://msdn.microsoft.com/en-us/library/9y6s16x1(VS.71).aspx

```
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

Для того чтобы вызывающая функция имела доступ к результату вызываемой функции, вызываемая функция (в нашем случае `scanf()`) оставляет это значение в регистре `EAX`.

Мы проверяем его инструкцией `CMP EAX, 1` (*CoMPare*), то есть, сравниваем значение в `EAX` с 1.

Следующий за инструкцией `CMP`: условный переход `JNE`. Это означает *Jump if Not Equal*, то есть, условный переход *если не равно*.

Итак, если `EAX` не равен 1, то `JNE` заставит перейти процессор по адресу указанном в операнде `JNE`, у нас это `$LN2@main`. Передав управление по этому адресу, CPU как раз начнет исполнять вызов `printf()` с аргументом `'What you entered? Huh?'`. Но если все нормально, перехода не случится, и исполнится другой `printf()` с двумя аргументами: `'You entered %d...'` и значением переменной `x`.

А для того чтобы после этого вызова не исполнился сразу второй вызов `printf()`, после него имеется инструкция `JMP`, безусловный переход, он отправит процессор на место аккурат после второго `printf()` и перед инструкцией `XOR EAX, EAX`, которая собственно `return 0`.

Итак, можно сказать, что в подавляющих случаях сравнение какой-либо переменной с чем-то другим происходит при помощи пары инструкций `CMP` и `Jcc`, где *cc* это *condition code*. `CMP` сравнивает два значения и выставляет флаги процессора[4]. `Jcc` проверяет нужные ему флаги и выполняет переход по указанному адресу (или не выполняет).

Но на самом деле, как это не парадоксально поначалу звучит, `CMP` это почти то же самое что и инструкция `SUB`, которая отнимает числа одно от другого. Все арифметические инструкции также выставляют флаги в соответствии с результатом, не только `CMP`. Если мы сравним 1 и 1, от единицы отнимется единица, получится 0, и выставится флаг `ZF` (*zero flag*), означающий что последний полученный результат был 0. Ни при каких других значениях `EAX`, флаг `ZF` выставлен не будет, кроме тех, когда операнды равны друг другу. Инструкция `JNE` проверяет только флаг `ZF`, и совершает переход только если флаг не поднят. Фактически, `JNE` это синоним инструкции `JNZ` (*Jump if Not Zero*). Ассемблер транслирует обе инструкции в один и тот же опкод. Таким образом, можно `CMP` заменить на `SUB` и все будет работать также, но разница в том, что `SUB` все-таки испортит значение в первом операнде. `CMP` это *SUB без сохранения результата*.

6.6.2 MSVC: x86: IDA

Наверное, уже пора делать первые попытки анализа кода в IDA. Кстати, для начинающих, полезно компилировать в MSVC с ключом `/MD`, что означает что все эти стандартные ф-ции не будут скомпонованы с исполняемым файлу, а будут импортироваться из файла `MSVCR*.DLL`. Так будет легче увидеть, где какая стандартная ф-ция используется.

Анализируя код в IDA, очень полезно делать пометки для себя (и других). Например, разбирая этот пример, мы сразу видим что `JNZ` срабатывает в случае ошибки. Можно навести курсор на эту метку, нажать "n" и переименовать метку в "error". Еще одну метку — в "exit". Вот как у меня получилось в итоге:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
```

[4]См. также о флагах x86-процессора: http://en.wikipedia.org/wiki/FLAGS_register_(computing).

```
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X:\n"
.text:00401009       call    ds:printf
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

Так понимать код становится чуть легче. Впрочем, меру нужно знать во всем и комментировать каждую инструкцию не стоит.

В IDA также можно скрывать части ф-ций: нужно отметить часть, нажать “-” на цифровой клавиатуре и ввести текст.

Я скрыл две части и придумал им названия:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000       assume cs:_text
.text:00401000       ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024       cmp  eax, 1
.text:00401027       jnz  short error
.text:00401029 ; print result
.text:0040103B       jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call ds:printf
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

Раскрывать скрытые части ф-ций можно при помощи “+” на цифровой клавиатуре.

Нажав “пробел”, мы увидим как IDA может представить ф-цию в виде графа: илл. 6.7. После каждого условного перехода видны две стрелки: зеленая и красная. Зеленая ведет к тому блоку, который исполнится если переход сработал, а красная — если не сработал.

В этом режиме также можно сворачивать узлы и давать им названия (“group nodes”). Я сделал это для трех блоков: илл. 6.8.

Всё это очень полезно делать. Вообще, очень важная часть работы реверсера состоит в том, чтобы уменьшать количество имеющейся информации.

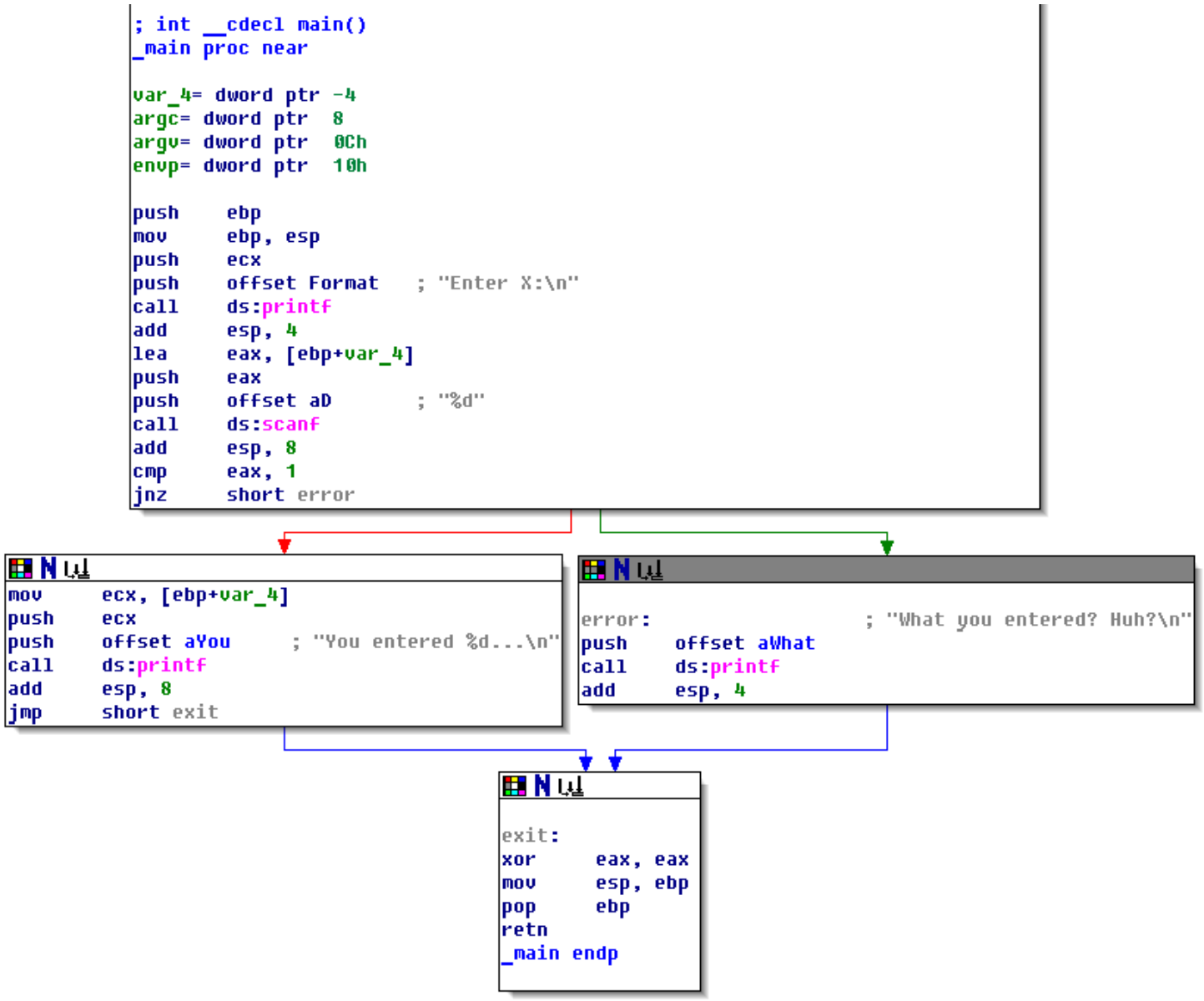


Рис. 6.7: Отображение в IDA в виде графа

```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format   ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD       ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

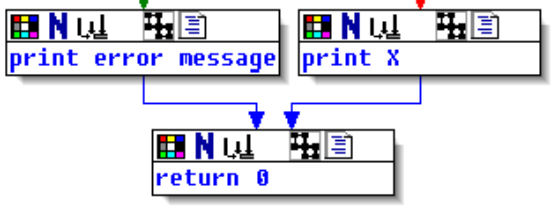


Рис. 6.8: Отображение в IDA в виде графа с тремя свернутыми блоками

6.6.3 MSVC: x86 + OllyDbg

Попробуем в OllyDbg немного хакнуть программу, и сделать вид что `scanf()` срабатывает всегда без ошибок.

Когда в `scanf()` передается адрес локальной перемейнной, изначальное, в данном случае, у нас в этой переменной находится некий мусор, а это `0x4CD478`: илл. 6.10.

Когда `scanf()` запускается, я ввожу в консоли что-то непохожее на число, например “asdasd”. `scanf()` заканчивается с 0 в `EAX`, что означает, что произошла ошибка: илл. 6.11.

Вместе с этим, мы можем посмотреть на локальную переменную в стеке, она не изменилась. Действительно, ведь что туда записала бы ф-ция `scanf()`? Она просто ничего не делала кроме возвращения нуля.

Теперь попробуем немного “хакнуть” нашу программу. Кликнем правой кнопкой на `EAX`, там, в числе опций, будет также “Set to 1”. Это то что нам нужно.

В `EAX` теперь 1, последующая проверка пройдет как надо, и `printf()` выведет значение переменной из стека.

Запускаем (F9) и видим в консоли следующее:

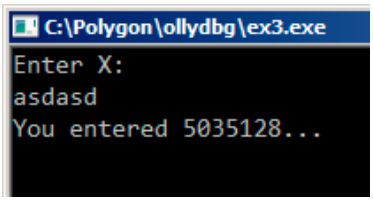


Рис. 6.9: консоль

Действительно, 5035128 это десятичное представление числа в стеке (`0x4CD478`)!

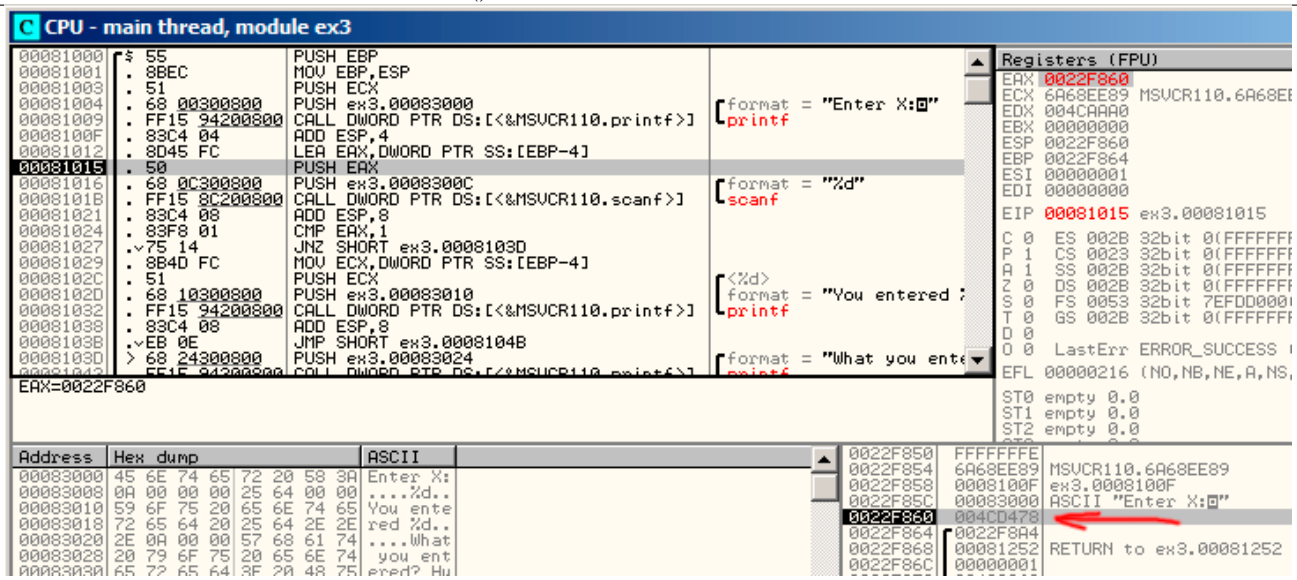


Рис. 6.10: OllyDbg: передача адреса переменной в `scanf()`

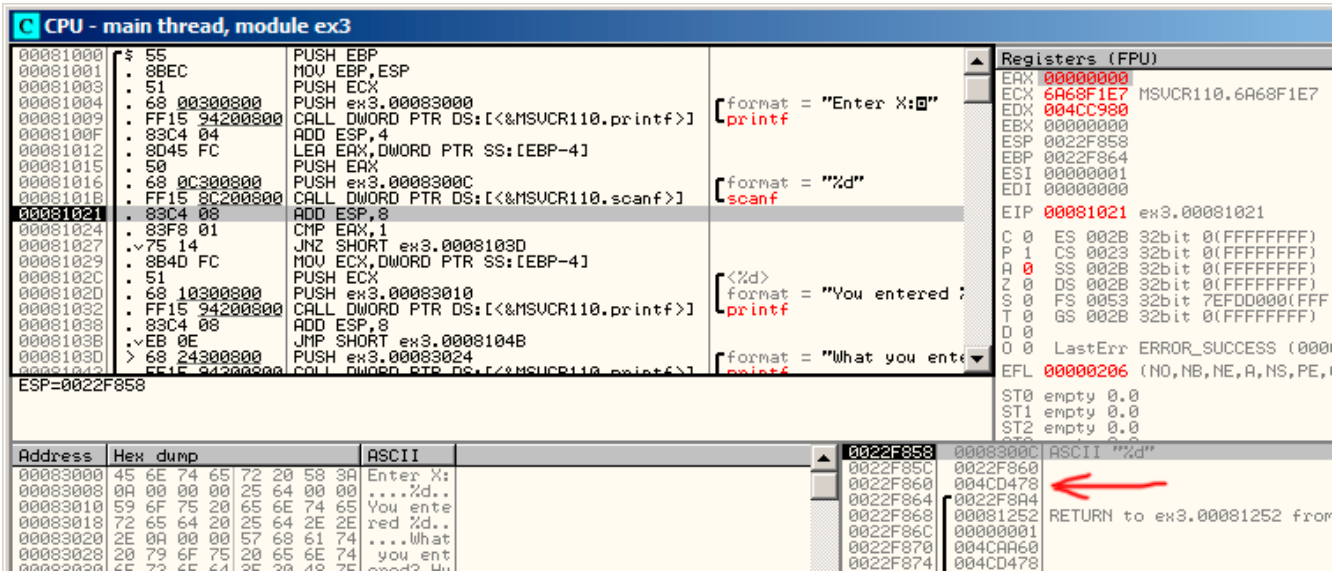


Рис. 6.11: OllyDbg: `scanf()` закончился с ошибкой

6.6.4 MSVC: x86 + Hiew

Это еще и может быть простым примером патчинга исполняемого файла. Мы можем попробовать пропатчить его таким образом, что программа всегда будет выводить числа, вне зависимости от того, что мы вводим.

Учитывая, что исполняемый файл скомпилирован с учетом импортирования ф-ций из `MSVCR*.DLL` (т.е., с опцией `/MD`)[5], мы можем отыскать ф-цию `main()` в самом начале секции `.text`. Откроем исполняемый файл в Hiew, найдем самое начало секции `.text` (Enter, F8, F6, Enter, Enter).

Мы увидим следующее: илл. 6.12.

Hiew находит ASCIIZ[6]-строки и показывает их, также как и имена импортируемых ф-ций.

Переведите курсор на адрес `.00401027` (с инструкцией `JNZ`, которую мы хотим заблокировать), нажмите F3, затем наберите “9090” (что означает NOP[7]-а): илл. 6.13.

Затем F9 (update). Теперь исполняемый файл записан на диск. Он будет вести себя так, как нам надо.

Два NOP-а возможно, не так эстетично, как могло бы быть. Другой способ пропатчить инструкцию, это записать 0 во второй байт опкода (смещение перехода), так что `JNZ` всегда будет переходить на следующую инструкцию.

[5]то, что еще называют “dynamic linking”
[6]ASCII Zero (ASCII-строка заканчивающаяся нулем)
[7]No OPeration

Можно пропатчить и наоборот: первый байт заменить на EB, второй байт (смещение перехода) не трогать. Получится всегда срабатывающий безусловный переход. Теперь сообщение об ошибке будет выдаваться всегда, даже если мы ввели число.

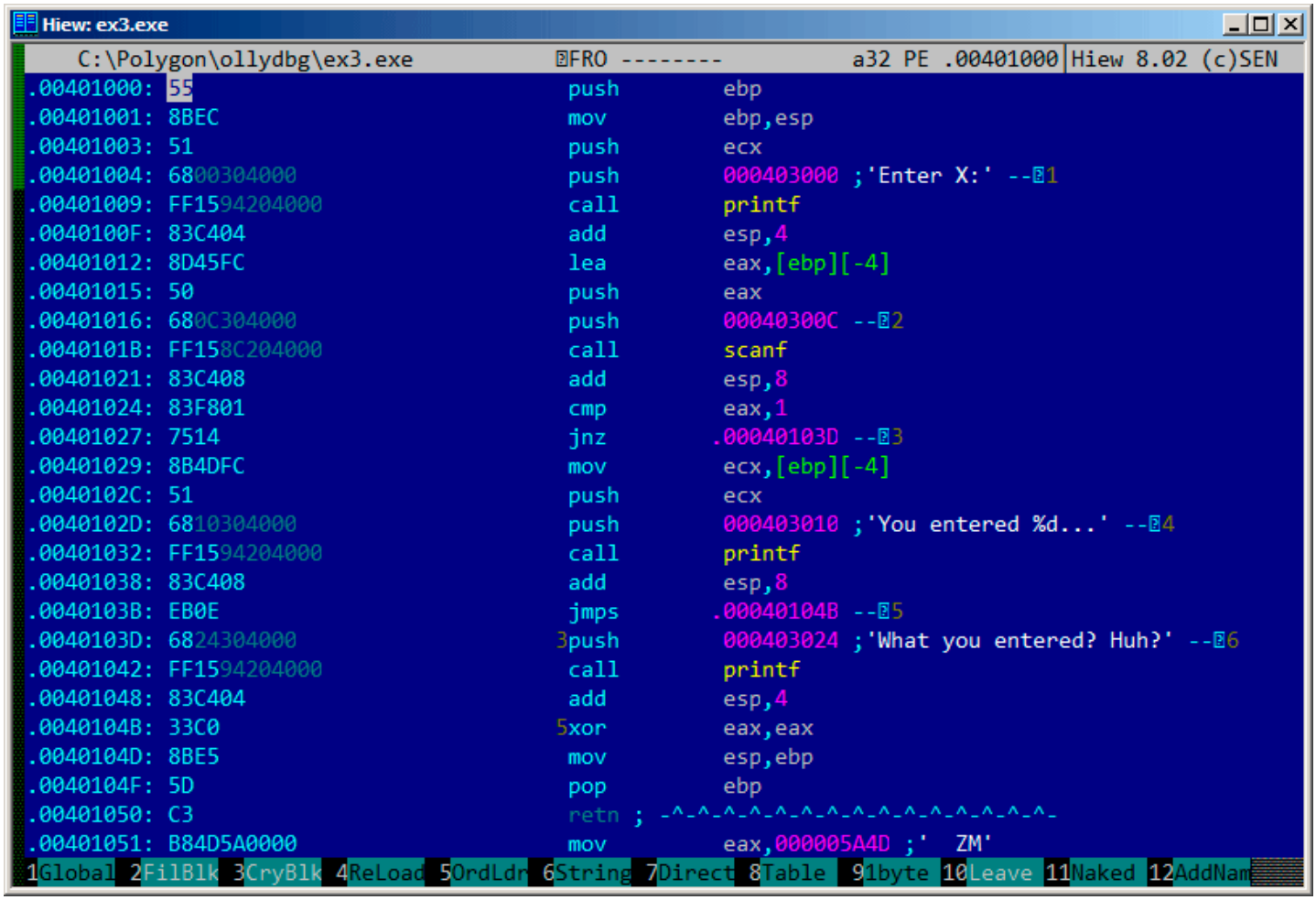


Рис. 6.12: Hiew: ф-ция `main()`

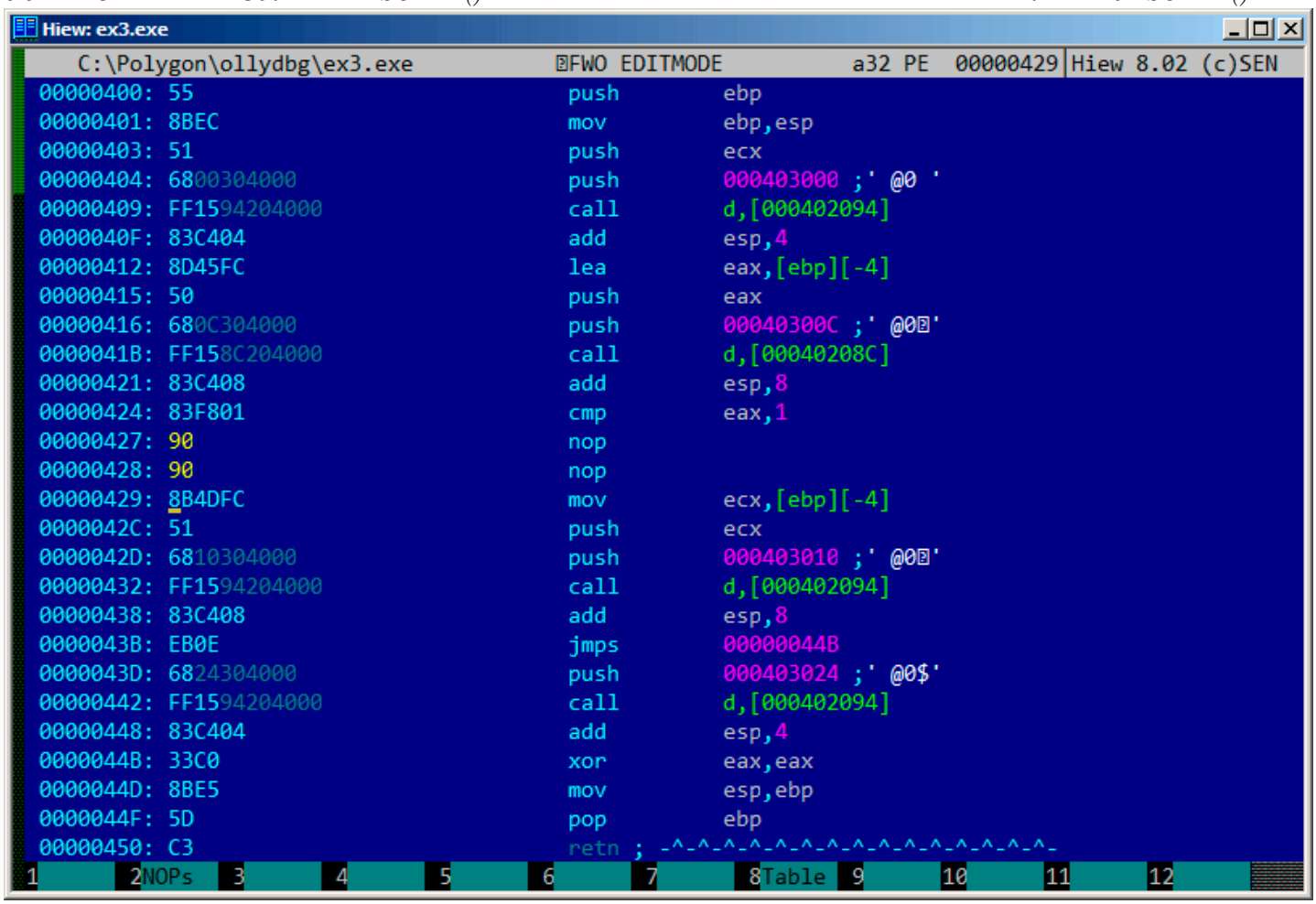


Рис. 6.13: Hiew: замена JNZ на два NOP-а

6.6.5 GCC: x86

Код созданный при помощи GCC 4.4.1 в Linux практически такой же, если не считать мелких отличий, которые мы уже рассмотрели раннее.

6.6.6 MSVC: x64

Так как мы работаем здесь с переменными типа *int*, а они в x86-64 остались 32-битными, то мы здесь видим как продолжают использоваться регистры с префиксом E-. Но для работы с указателями, конечно, используются 64-битные части регистров, с префиксом R-.

Листинг 6.4: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
```

```
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
        call    printf
$LN1@main:
        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

6.6.7 ARM: Оптимизирующий Keil + Режим thumb

Листинг 6.5: Оптимизирующий Keil + Режим thumb

```
var_8           = -8

                PUSH    {R3,LR}
                ADR     R0, aEnterX     ; "Enter X:\n"
                BL      __2printf
                MOV     R1, SP
                ADR     R0, aD          ; "%d"
                BL      __0scanf
                CMP     R0, #1
                BEQ     loc_1E
                ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
                BL      __2printf

loc_1A                                  ; CODE XREF: main+26
                MOVS    R0, #0
                POP     {R3,PC}

loc_1E                                  ; CODE XREF: main+12
                LDR     R1, [SP,#8+var_8]
                ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
                BL      __2printf
                B       loc_1A
```

Новые инструкции здесь для нас: `CMP` и BEQ[8].

`CMP` аналогична той что в x86, она отнимает один аргумент от второго и сохраняет флаги.

BEQ совершает переход по другому адресу, если операнды при сравнении были равны, либо если результат последнего вычисления был 0, либо если флаг Z равен 1. То же что и `JZ` в x86.

Всё остальное просто: исполнение разветвляется на две ветки, затем они сходятся там, где в `R0` записывается 0 как возвращаемое из функции значение и происходит выход из функции.

[8] (PowerPC, ARM) Branch if Equal

Глава 7

Доступ к переданным аргументам

Как мы уже успели заметить, вызывающая функция передает аргументы для вызываемой через стек. А как вызываемая функция имеет к ним доступ?

Листинг 7.1: простой пример

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

7.1 x86

7.1.1 MSVC

Имеем в итоге (MSVC 2010 Express):

Листинг 7.2: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_c$ = 16                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3rd argument
        push    2 ; 2nd argument
        push    1 ; 1st argument
        call    _f
```

```
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; return 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Итак, здесь видно: в функции `main()` заталкиваются три числа в стек и вызывается функция `f(int,int,int)`. Внутри `f()`, доступ к аргументам, также как и к локальным переменным, происходит через макросы: `_a$ = 8`, но разница в том, что эти смещения со знаком *плюс*, таким образом если прибавить макрос `_a$` к указателю на `EBP`, то адресуется *внешняя* часть фрейма стека относительно `EBP`.

Далее все более-менее просто: значение a помещается в `EAX`. Далее `EAX` умножается при помощи инструкции `IMUL` на то что лежит в `_b`, так в `EAX` остается произведение[1]этих двух значений. Далее к регистру `EAX` прибавляется то что лежит в `_c`. Значение из `EAX` никуда не нужно перекладывать, оно уже лежит где надо. Возвращаем управление вызываемой функции — она возьмет значение из `EAX` и отправит его в `printf()`.

7.1.2 MSVC + OllyDbg

Проиллюстрируем всё это в OllyDbg. Когда мы протрассируем до первой инструкции в `f()`, которая использует какой-то из аргументов (первый), мы увидим что `EBP` указывает на фрейм стека, я отметил его начало красной стрелкой. Самый первый элемент фрейма стека это сохраненное значение `EBP`, затем RA, третий элемент это первый аргумент ф-ции, затем второй аргумент и третий. Для доступа к первому аргументу ф-ции, нужно прибавить к `EBP` аккурат 8 (2 32-битных слова).

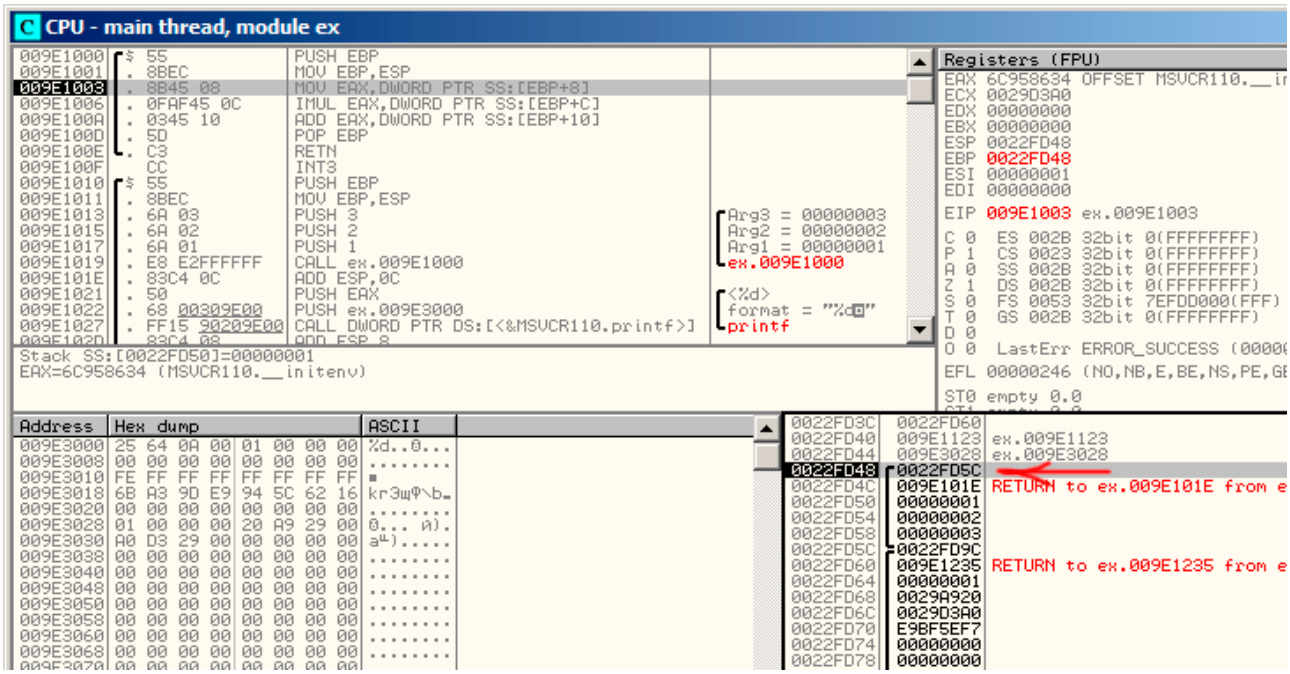


Рис. 7.1: OllyDbg: внутри ф-ции f()

7.1.3 GCC

Скомпилируем то же в GCC 4.4.1 и посмотрим результат в IDA:

Листинг 7.3: GCC 4.4.1

```
                public f
f               proc near
```

[1]результат умножения

```
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0] ; 1st argument
                imul    eax, [ebp+arg_4] ; 2nd argument
                add     eax, [ebp+arg_8] ; 3rd argument
                pop     ebp
                retn
f               endp

                public main
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_8], 3 ; 3rd argument
                mov     [esp+10h+var_C], 2 ; 2nd argument
                mov     [esp+10h+var_10], 1 ; 1st argument
                call    f
                mov     edx, offset aD  ; "%d\n"
                mov     [esp+10h+var_C], eax
                mov     [esp+10h+var_10], edx
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Практически то же самое, если не считать мелких отличий описанных раннее.

После вызова обоих ф-ций, указатель стека не возвращается назад, потому что предпоследняя инструкция LEAVE (B.6.2) сделает это за один раз, в конце исполнения.

7.2 x64

В x86-64 всё немного иначе, здесь аргументы ф-ции (4 или 6) передаются через регистры, а callee из регистров их и читает, вместо того чтобы обращаться к стеку.

7.2.1 MSVC

Оптимизирующий MSVC:

Листинг 7.4: MSVC 2012 /Ox x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
```

```
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - 1st argument
        ; EDX - 2nd argument
        ; R8D - 3rd argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Как видно, очень компактная ф-ция `f()`, берет аргументы прямо из регистров. Инструкция `LEA` используется здесь для сложения чисел, должно быть компилятор посчитал что так будет эффективнее нежели использование `ADD`. А в самой `main()` `LEA` также используется для подготовки первого и третьего аргумента: должно быть, компилятор решил, что `LEA` будет работать здесь быстрее, чем загрузка значения в регистр при помощи `MOV`.

Попробуем посмотреть вывод неоптимизирующего MSVC:

Листинг 7.5: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1st argument
                ; EDX - 2nd argument
                ; R8D - 3rd argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3rd argument
                mov     edx, 2 ; 2nd argument
                mov     ecx, 1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

Немного путаннее: все 3 аргумента из регистров зачем-то сохраняются в стеке. Это называется “shadow space” [2]: каждая ф-ция в Win64 может (хотя и не обязана) сохранять значения 4-х регистров там. Это делается по крайней мере из-за двух причин: 1) в большой ф-ции отвести целый регистр (а тем более 4 регистра) для

[2] http://msdn.microsoft.com/en-us/library/zthk2dkh(v=vs.80).aspx

входного аргумента, это слишком расточительно, так что к нему будет обращение через стек; 2) отладчик всегда знает где найти аргументы ф-ции в момент останова [3].

Место в стеке для "shadow space" выделяет именно caller.

7.2.2 GCC

Оптимизирующий GCC также делает понятный код:

Листинг 7.6: GCC 4.4.6 -O3 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Неоптимизирующий GCC:

Листинг 7.7: GCC 4.4.6 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; number of vector registers passed
```

[3] http://msdn.microsoft.com/en-us/library/ew5tede7(v=VS.90).aspx

```
        call    printf
        mov     eax, 0
        leave
        ret
```

В соглашении о вызовах System V *NIX [21] нет “shadow space”, но callee тоже иногда должен сохранять где-то аргументы, потому что, опять же, регистров может и не хватить на все действия. Что мы здесь и видим.

7.2.3 GCC: uint64_t вместо int

Наш пример работал с 32-битным *int*, поэтому использовались 32-битные части регистров с префиксом E-.

Его можно немного переделать, чтобы он заработал с 64-битными значениями:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Листинг 7.8: GCC 4.4.6 -O3 x64

```
f               proc near
                imul    rsi, rdi
                lea     rax, [rdx+rsi]
                retn
f               endp

main            proc near
                sub     rsp, 8
                mov     rdx, 3333333344444444h ; 3rd argument
                mov     rsi, 1111111122222222h ; 2nd argument
                mov     rdi, 1122334455667788h ; 1st argument
                call    f
                mov     edi, offset format ; "%lld\n"
                mov     rsi, rax
                xor     eax, eax ; number of vector registers passed
                call    _printf
                xor     eax, eax
                add     rsp, 8
                retn
main            endp
```

Собственно, всё то же самое, только используются регистры *целиком*, с префиксом R-.

7.3 ARM

7.3.1 Неоптимизирующий Keil + Режим ARM

```
.text:000000A4 00 30 A0 E1                MOV     R3, R0
.text:000000A8 93 21 20 E0                MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1                BX      LR
```

```
...
.text:000000B0                 main
.text:000000B0 10 40 2D E9                 STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3                 MOV     R2, #3
.text:000000B8 02 10 A0 E3                 MOV     R1, #2
.text:000000BC 01 00 A0 E3                 MOV     R0, #1
.text:000000C0 F7 FF FF EB                 BL      f
.text:000000C4 00 40 A0 E1                 MOV     R4, R0
.text:000000C8 04 10 A0 E1                 MOV     R1, R4
.text:000000CC 5A 0F 8F E2                 ADR     R0, aD_0        ; "%d\n"
.text:000000D0 E3 18 00 EB                 BL      __2printf
.text:000000D4 00 00 A0 E3                 MOV     R0, #0
.text:000000D8 10 80 BD E8                 LDMFD   SP!, {R4,PC}
```

В функции `main()` просто вызываются две функции, в первую (`f`) передается три значения.

Как я уже упоминал, первые 4 значения, в ARM обычно передаются в первых 4-х регистрах (`R0-R3`).

Функция `f`, как видно, использует три первых регистра (`R0-R2`) как аргументы.

Инструкция `MLA` (*Multiply Accumulate*) перемножает два первых операнда (`R3` и `R1`), прибавляет к произведению третий операнд (`R2`) и помещает результат в нулевой операнд (`R0`), через который, по стандарту, возвращаются значения функций.

Умножение и сложение одновременно[4] (*Fused multiply–add*) это много где применяемая операция, кстати, аналогичной инструкции в x86 нет, если не считать новых FMA-инструкций[5] в SIMD.

Самая первая инструкция `MOV R3, R0`, по-видимому, избыточна (можно было бы обойтись только одной инструкцией `MLA`), компилятор не оптимизировал её, ведь, это компиляция без оптимизации.

Инструкция `BX` возвращает управление по адресу записанному в LR и, если нужно, переключает режимы процессора с thumb на ARM или наоборот. Это может быть необходимым потому, что, как мы видим, функции `f` неизвестно, из какого кода она будет вызываться, из ARM или thumb. Поэтому, если она будет вызываться из кода thumb, `BX` не только вернет управление в вызывающую функцию, но также переключит процессор в режим thumb. Либо не переключит, если функция вызывалась из кода для режима ARM.

7.3.2 Оптимизирующий Keil + Режим ARM

```
.text:00000098                 f
.text:00000098 91 20 20 E0                 MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1                 BX      LR
```

А вот и функция `f` скомпилированная компилятором Keil в режиме полной оптимизации (`-O3`). Инструкция `MOV` была соптимизирована и теперь `MLA` использует все входящие регистры и помещает результат в `R0`, как раз, где вызываемая функция будет его читать и использовать.

7.3.3 Оптимизирующий Keil + Режим thumb

```
.text:0000005E 48 43                       MULS    R0, R1
.text:00000060 80 18                       ADDS    R0, R0, R2
.text:00000062 70 47                       BX      LR
```

В режиме thumb, инструкция `MLA` недоступна, так что компилятору пришлось сгенерировать код, делающий обе операции по отдельности. Первая инструкция `MULS` умножает `R0` на `R1` оставляя результат в `R1`. Вторая (`ADDS`) складывает результат и `R2`, оставляя результат в `R0`.

[4] wikipedia: Умножение-сложение

[5] https://en.wikipedia.org/wiki/FMA_instruction_set

Глава 8

И еще немного о возвращаемых результатах

Результат выполнения функции в x86 обычно возвращается [1] через регистр EAX, а если результат имеет тип байт или символ (*char*), то в самой младшей части EAX — AL. Если функция возвращает число с плавающей запятой, то будет использован регистр FPU ST(0). В ARM обычно результат возвращается в регистре R0.

Кстати, что будет если возвращаемое значение в ф-ции main() объявлять не как *int* а как *void*?
Т.н. startup-код вызывает main() примерно так:

```
push envp
push argv
push argc
call main
push eax
call exit
```

Т.е., иными словами:

```
exit(main(argc,argv,envp));
```

Если вы объявите main() как *void*, и ничего не будете возвращать явно (при помощи выражения *return*), то в единственный аргумент exit() попадет то, что лежало в регистре EAX на момент выхода из main(). Там, скорее всего, будет какие-то случайное число, оставшееся от работы вашей ф-ции. Так что, код завершения программы будет псевдослучайным.

Я могу это проиллюстрировать. Заметьте что у ф-ции main() тип возвращаемого значения именно *void*:

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Скомпилируем в Linux.
GCC 4.8.1 заменила printf() на puts() (мы видели это прежде: 2.3.3), но это нормально, потому что puts() возвращает количество выведенных символов, так же как и printf(). Обратите внимание на то что EAX не обнуляется перед выходом их main(). Это значит, EAX перед выходом из main() будет содержать то, что puts() оставит там.

Листинг 8.1: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
```

[1] См. также: MSDN: Return Values (C++): http://msdn.microsoft.com/en-us/library/7572ztz4.aspx

```
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Напишем небольшой скрипт на bash, показывающий статус возврата ("exit status" или "exit code"):

Листинг 8.2: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

И запустим:

```
$ tst.sh
Hello, world!
14
```

14 это как раз количество выведенных символов.

Вернемся к тому факту, что возвращаемое значение остается в регистре `EAX`. Вот почему старые компиляторы Си не способны создавать функции, возвращающие нечто большее нежели помещается в один регистр (обычно, тип *int*), а когда нужно, приходится возвращать через указатели, указываемые в аргументах. Хотя, позже и стало возможным, вернуть, скажем, целую структуру, но этот метод до сих пор не очень популярен. Если функция должна вернуть структуру, вызывающая функция должна сама, скрыто и прозрачно для программиста, выделить место и передать указатель на него в качестве первого аргумента. Это почти то же самое что и сделать это вручную, но компилятор прячет это.

Небольшой пример:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

... получим (MSVC 2010 `/Ox`):

```
$T3853 = 8                      ; size = 4
_a$ = 12                        ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC              ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
```

```
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP            ; get_some_values
```

Имя внутреннего макроса для передачи указателя на структуру здесь это `$T3853`.

Этот пример можно даже переписать используя расширения C99:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Листинг 8.3: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

Как видно, ф-ция просто заполняет поля в структуре, выделенной вызывающей ф-цией. Так что никаких проблем с эффективностью нет.

Глава 9

Указатели

Указатели также часто используются для возврата значений из функции (вспомните случай со `scanf()` (6)). Например, когда функции нужно вернуть сразу два значения.

9.1 Пример с глобальными переменными

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Это компилируется в:

Листинг 9.1: Оптимизирующий MSVC 2010 (/Ox /Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
_sum$ = 16                                              ; size = 4
_product$ = 20                                          ; size = 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP
```

```
_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456                                     ; 000001c8H
        push    123                                     ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28                                 ; 0000001cH
        xor     eax, eax
        ret     0
_main   ENDP
```

Посмотрим это в OllyDbg: илл. 9.1. В начале адреса обоих глобальных переменных передаются в `f1()`. Можно нажать "Follow in dump" на элементе стека и в окне слева увидим место в сегменте данных выделенных для двух переменных. Эти переменные обнулены, потому что, по стандарту, неинициализированные данные (BSS[1]) обнуляются перед началом исполнения. И они находятся в сегменте данных, о чем можно удостовериться нажав Alt-M и увидев карту памяти: илл. 9.5.

Трассируем (F7) до начала исполнения `f1()`илл. 9.2. В стеке видны и значения 456 (`0x1C8`) и 123 (`0x7B`), а также адреса двух глобальных переменных.

Трассируем до конца `f1()`. Мы видим в окне слева, как результаты вычисления появились в глобальных переменных илл. 9.3.

Теперь из глобальных переменных значения загружаются в регистры для передачи в `printf()`: илл. 9.4.

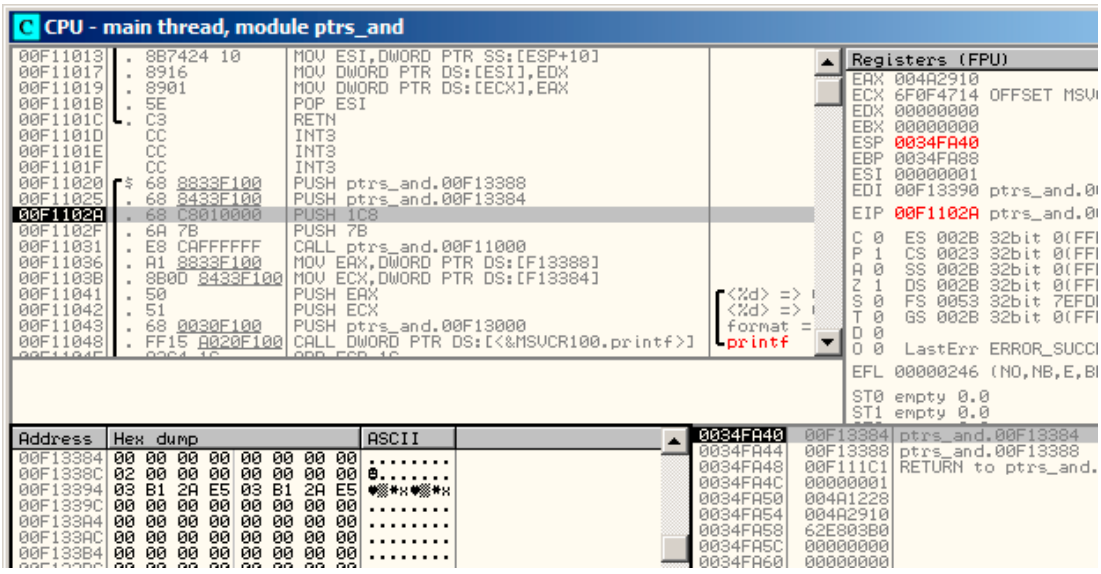


Рис. 9.1: OllyDbg: передаются адреса двух глобальных переменных в `f1()`

[1]Block Started by Symbol

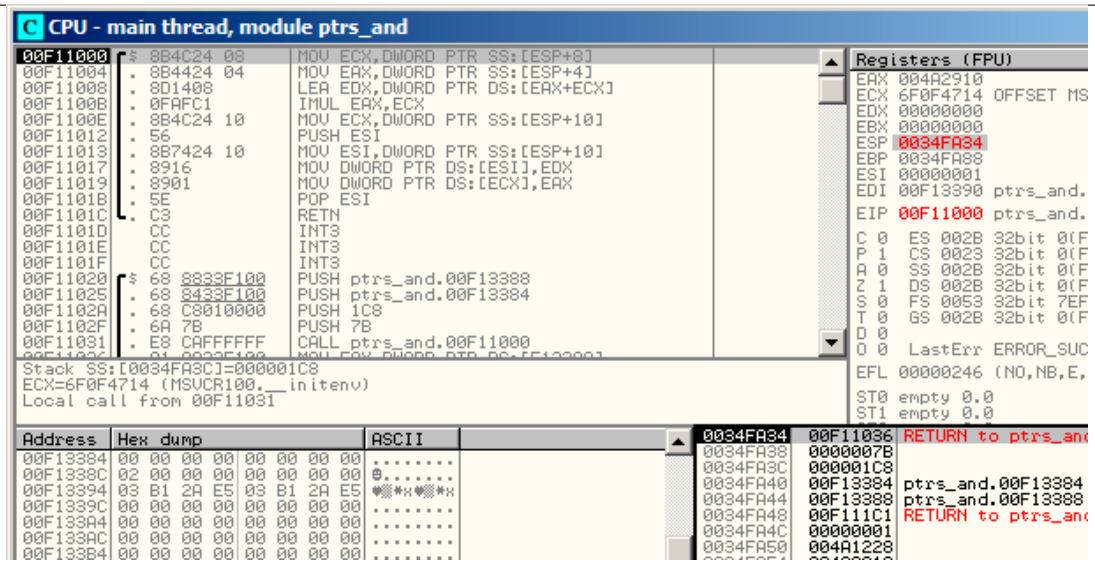


Рис. 9.2: OllyDbg: начало работы `f1()`

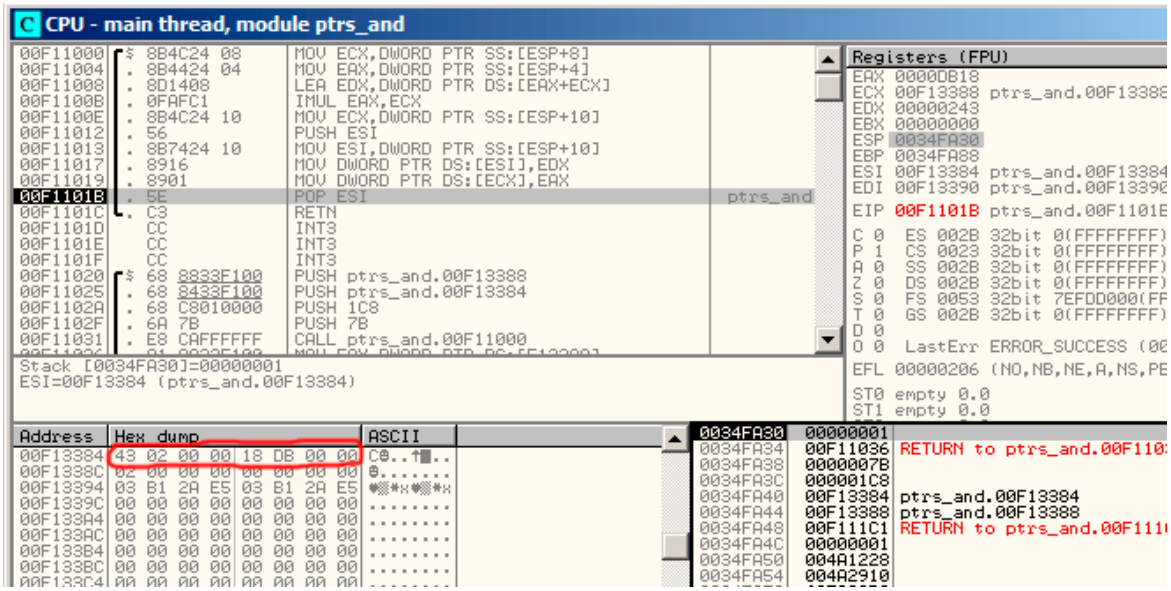


Рис. 9.3: OllyDbg: `f1()`заканчивает работу

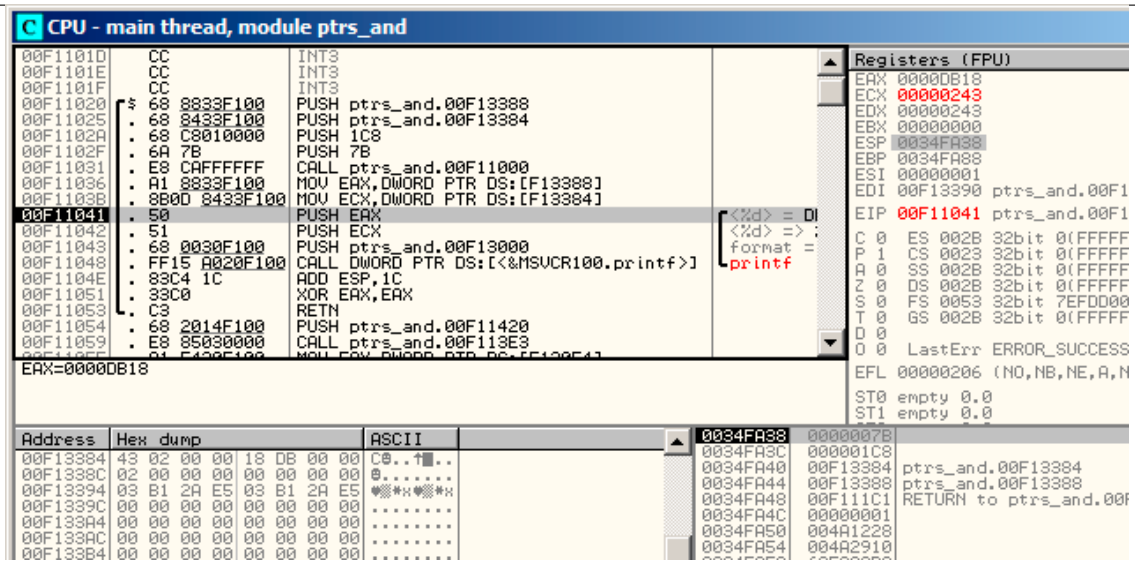


Рис. 9.4: OllyDbg: адреса глобальных переменных передаются в `printf()`

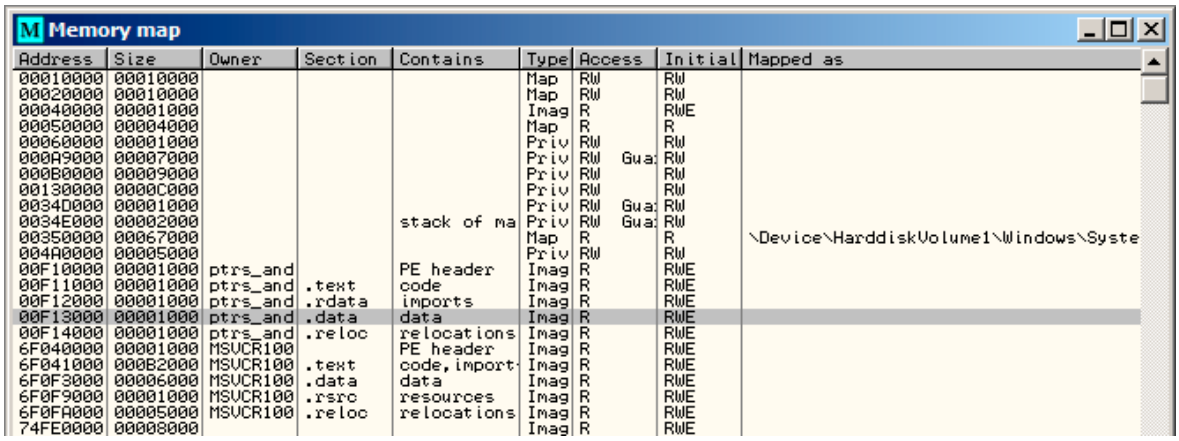


Рис. 9.5: OllyDbg: карта памяти

9.2 Пример с локальными переменными

Немного переделаем пример:

Листинг 9.2: теперь переменные локальные

```
void main()
{
        int sum, product; // теперь переменные здесь

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Код ф-ции `f1()`не изменится. Изменится только `main()`:

Листинг 9.3: Оптимизирующий MSVC 2010 (/Ox /Ob0)

```
_product$ = -8                                          ; size = 4
_sum$ = -4                                              ; size = 4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
```

```
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456                                     ; 000001c8H
        push    123                                     ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36                                 ; 00000024H
        ret     0
```

Снова посмотрим в OllyDbg. Адреса локальных переменных в стеке это `0x35FCF4` и `0x35FCF8`. Видно как они заталкиваются в стек: илл. 9.6.

Начало работы `f1()`. В стеке по адресам `0x35FCF4` и `0x35FCF8` пока находится случайный мусор илл. 9.7.

Конец работы `f1()`. В стеке по адресам `0x35FCF4` и `0x35FCF8` теперь значения `0xDB18` и `0x243`, это результаты работы `f1()`.

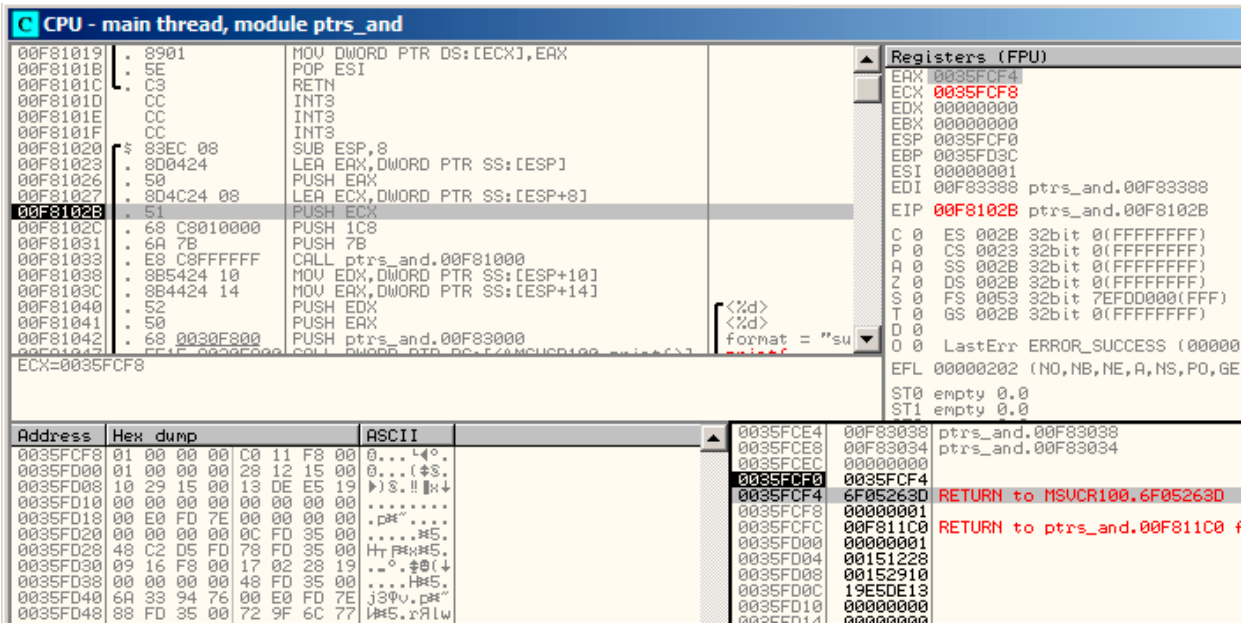


Рис. 9.6: OllyDbg: адреса локальных переменных заталкиваются в стек

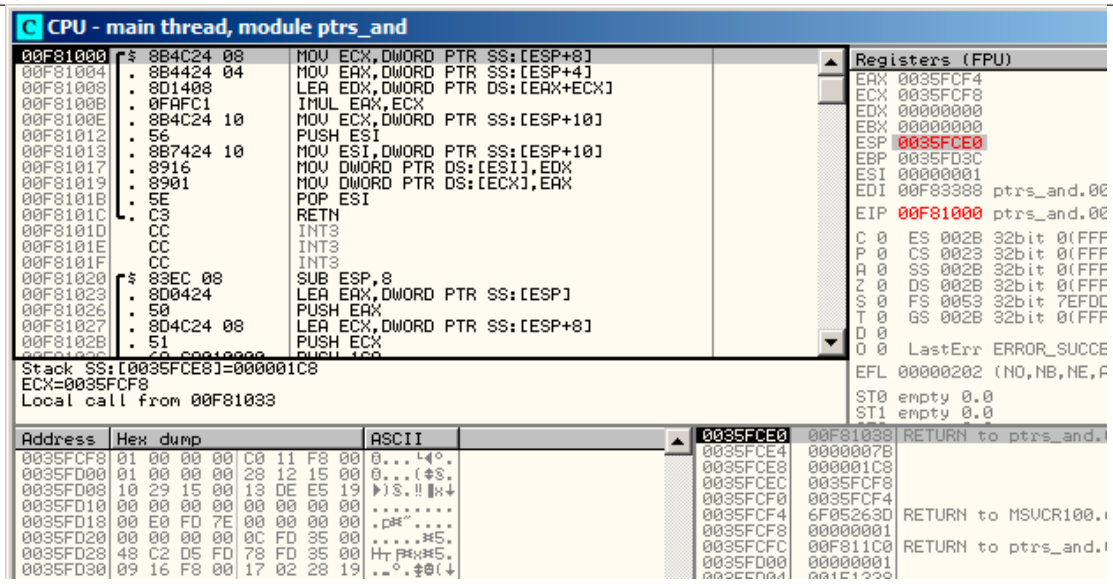


Рис. 9.7: OllyDbg: `f1()` начинает работу

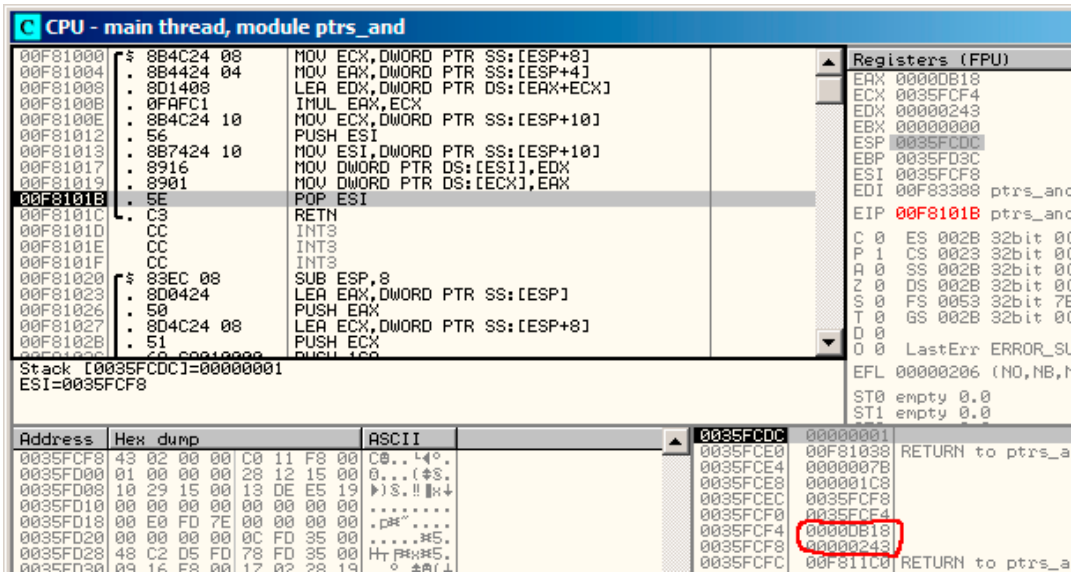


Рис. 9.8: OllyDbg: `f1()` заканчивает работу

9.3 Вывод

`f1()` может одинаково хорошо возвращать результаты работы в любые места памяти, находящиеся где угодно. В этом суть и удобство указателей.

Кстати, *references* в Си++ работают точно так же. Читайте больше об этом: (29.3).

Глава 10

Условные переходы

Об условных переходах.

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

10.1 x86

10.1.1 x86 + MSVC

Имеем в итоге функцию `f_signed()`:

Листинг 10.1: Неоптимизирующий MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
```

```
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

Первая инструкция JLE значит *Jump if Less or Equal*. То есть, если второй операнд больше первого или равен ему, произойдет переход туда, где будет следующая проверка. А если это условие не срабатывает, то есть второй операнд меньше первого, то перехода не будет, и сработает первый `printf()`. Вторая проверка это JNE: *Jump if Not Equal*. Переход не произойдет, если операнды равны. Третья проверка JGE: *Jump if Greater or Equal* — переход если первый операнд больше второго или равен ему. Кстати, если все три условных перехода сработают, ни один `printf()` не вызовется. Но, без внешнего вмешательства, это, пожалуй, невозможно.

Функция `f_unsigned()` точно такая же, за тем исключением, что используются инструкции JBE и JAE вместо JLE и JGE, об этом читайте ниже:

Далее функция `f_unsigned()`

Листинг 10.2: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
```

```
_f_unsigned ENDP
```

Здесь все то же самое, только инструкции условных переходов немного другие: JBE — *Jump if Below or Equal* и JAE — *Jump if Above or Equal*. Эти инструкции (JA/JAE/JBE/JBE) отличаются от JG/JGE/JL/JLE тем, что работают с беззнаковыми переменными.

Отступление: смотрите также секцию о представлении знака в числах (32). Таким образом, увидев где используется JG/JL вместо JA/JBE и наоборот, можно сказать почти уверенно насчет того, является ли тип переменной знаковым (signed) или беззнаковым (unsigned).

Далее функция main(), где ничего нового для нас нет:

Листинг 10.3: main()

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

10.1.2 x86 + MSVC + OllyDbg

Если попробовать этот пример в OllyDbg, можно увидеть как выставляются флаги. Начнем с ф-ции f_unsigned(), которая работает с беззнаковыми числами. В целом, в каждой ф-ции, CMP исполняется три раза, но для одних и тех же аргументов, так что флаги будут три раза одни и те же каждый раз.

Результат первого сравнения: илл. 10.1. Итак, флаги: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Для краткости, в OllyDbg флаги называются только одной буквой.

OllyDbg подсказывает, что первый переход (JBE) сработает. Действительно, если заглянуть в [14], прочитаем там, что JBE срабатывает в случаях если CF=1 или ZF=1. Условие здесь выполняется, так что переход срабатывает.

Следующий переход:илл. 10.2. OllyDbg подсказывает, что JNZ сработает. Действительно, JNZ срабатывает если ZF=0 (zero flag).

Третий переход JNB: илл. 10.3. В [14] мы можем найти что JNB срабатывает если CF=0 (carry flag). В нашем случае это не так, так что переход не срабатывает, и исполняется третий по счету printf().

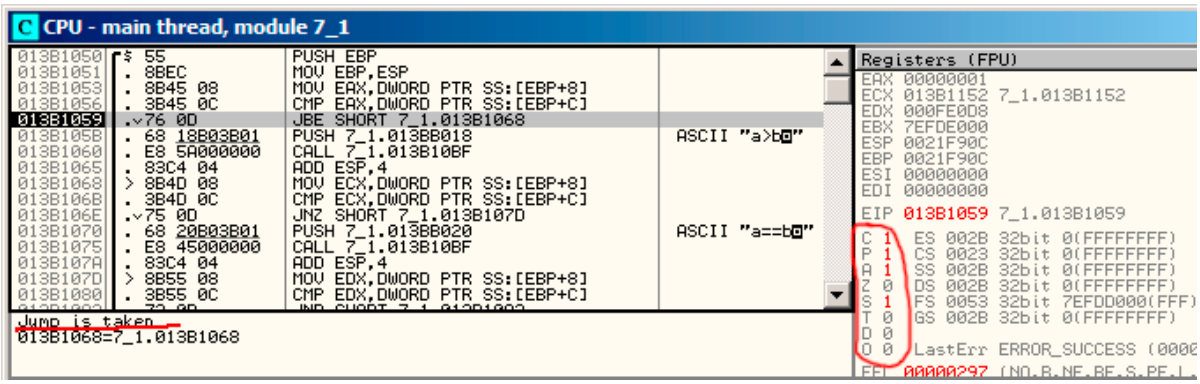


Рис. 10.1: OllyDbg: f_unsigned(): первый условный переход

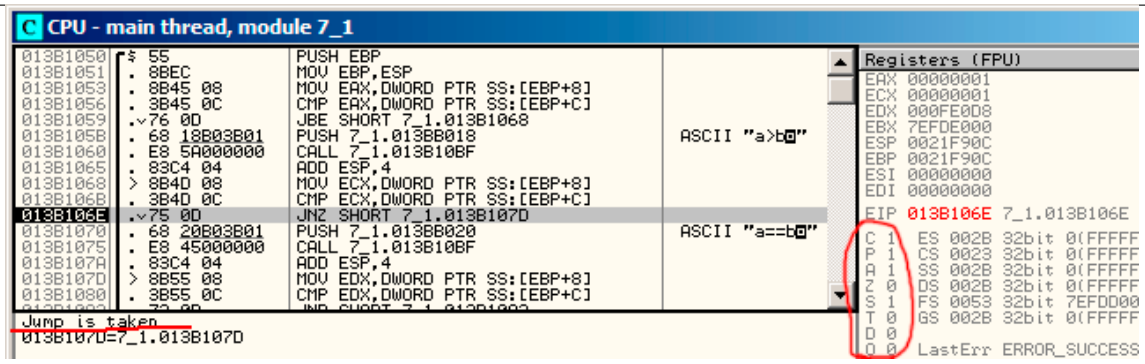


Рис. 10.2: OllyDbg: `f_unsigned()`: второй условный переход

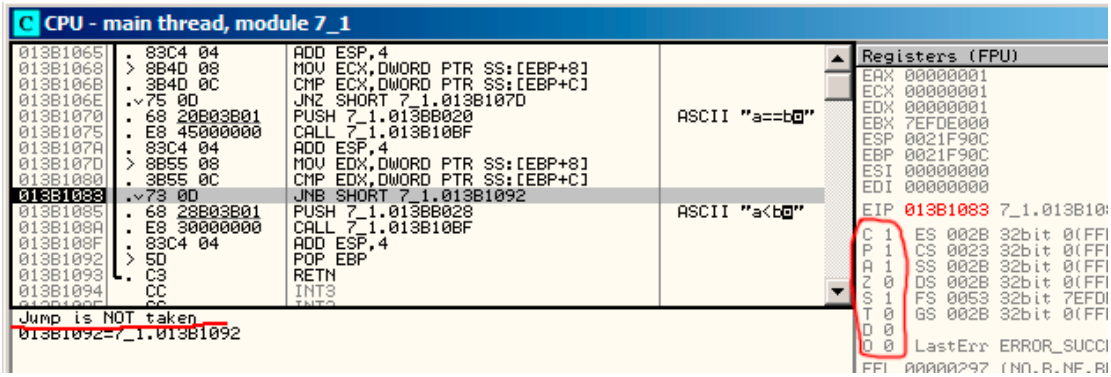


Рис. 10.3: OllyDbg: `f_unsigned()`: третий условный переход

Теперь можно попробовать в OllyDbg ф-цию `f_signed()` работающую с знаковыми величинами.

Флаги выставляются точно так же: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

Первый переход `JLE` сработает. илл. 10.4. В [14] мы можем прочитать что эта инструкция срабатывает если ZF=1 или SF≠OF. В нашем случае, SF≠OF, так что переход срабатывает.

Второй переход `JNZ` сработает: он срабатывает если ZF=0 (zero flag): илл. 10.5.

Третий переход `JGE` не сработает, потому что он срабатывает только если SF=OF, что в нашем случае не так: илл. 10.6.

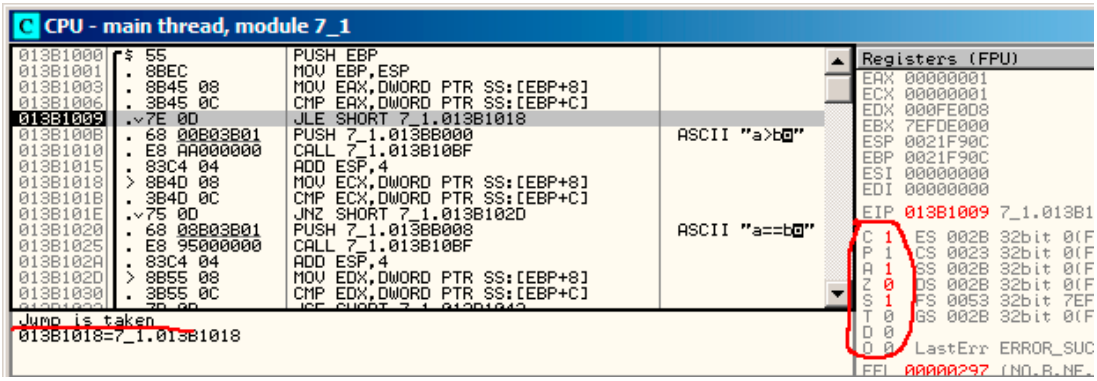


Рис. 10.4: OllyDbg: `f_unsigned()`: первый условный переход

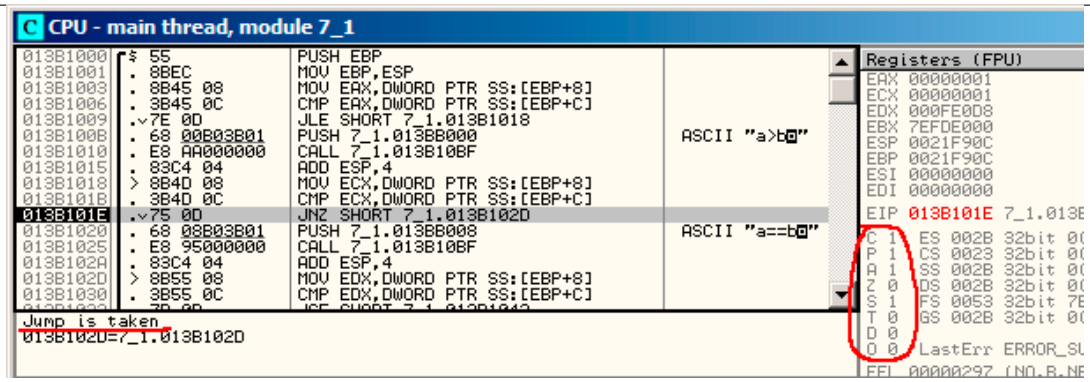


Рис. 10.5: OllyDbg: `f_unsigned()`: второй условный переход

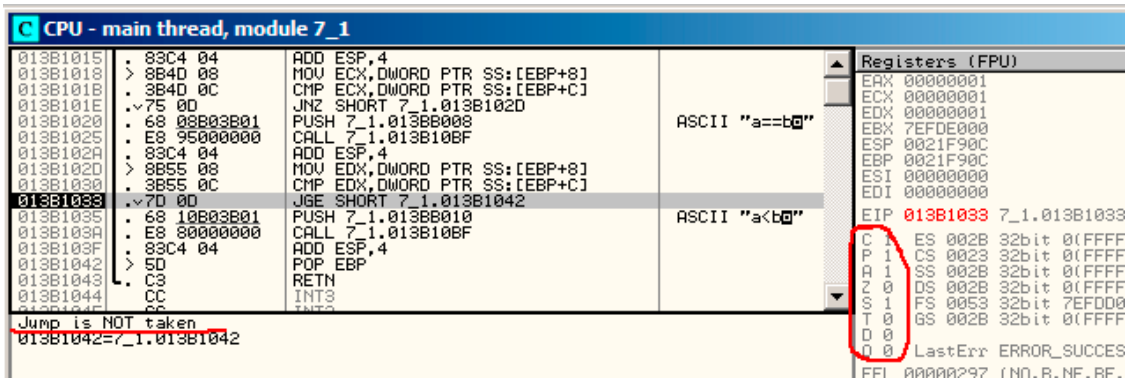


Рис. 10.6: OllyDbg: `f_unsigned()`: третий условный переход

10.1.3 x86 + MSVC + Hiew

Можем попробовать модифицировать исполняемый файл так, чтобы ф-ция `f_unsigned()` всегда показывала “a==b”, при любых входящих значениях.

Вот как она выглядит в Hiew: илл. 10.7.

Собственно, задаче три:

- заставить первый переход срабатывать всегда;
- заставить второй переход не срабатывать никогда;
- заставить третий переход срабатывать всегда.

Так мы направим путь исполнения кода (code flow) во второй `printf()`, и он всегда будет срабатывать и выводить на консоль “a==b”.

Для этого нужно пропатчить три инструкции (или байта):

- Первый переход теперь будет `JMP`, но смещение перехода (jump offset) останется прежним.
- Второй переход может быть и будет срабатывать иногда, но в любом случае он будет совершать переход только на следующую инструкцию, потому что мы выставляем смещение перехода (jump offset) в 0. В этих инструкциях, смещение перехода просто прибавляется к адресу следующей инструкции. И когда смещение 0, то тогда и переход будет на следующую инструкцию.
- Третий переход конвертируем в `JMP` точно так же, как и первый, он будет срабатывать всегда.

Что и делаем: илл. 10.8.

Если забыть про какой-то из переходов, то тогда будет срабатывать несколько вызовов `printf()`, а нам ведь нужно чтобы исполнялся только один.

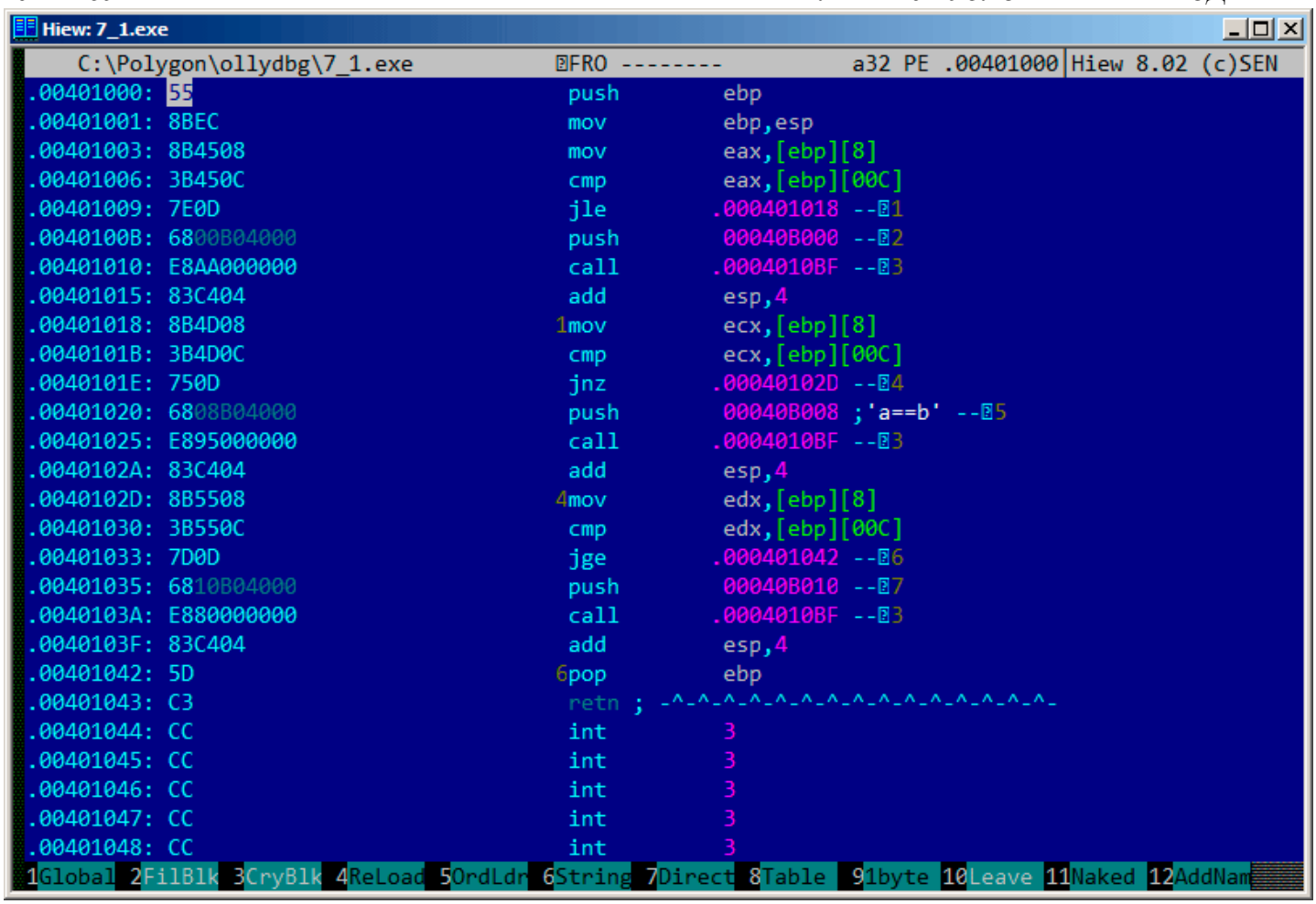


Рис. 10.7: Hiew: функция `f_unsigned()`

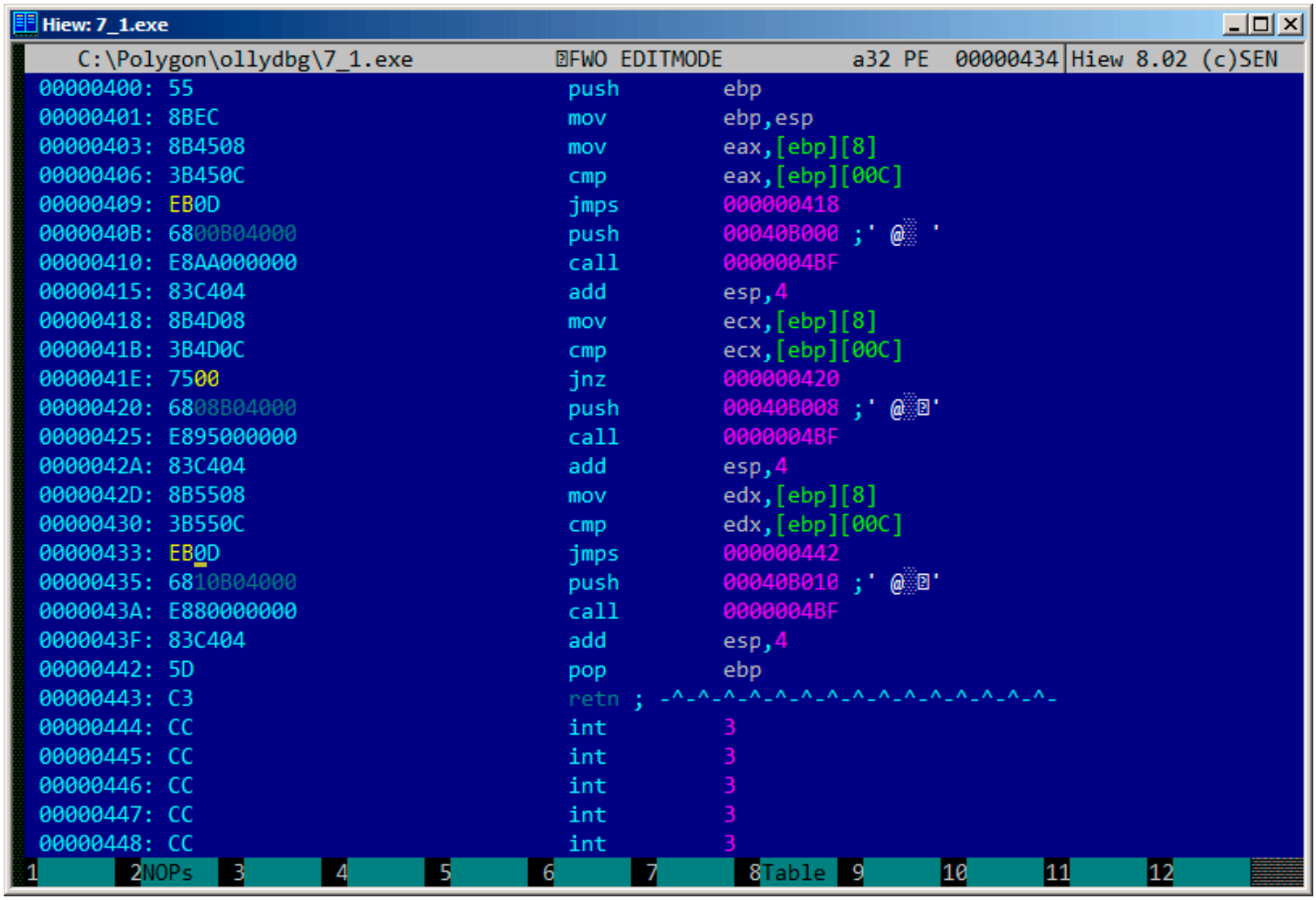


Рис. 10.8: Hiew: модифицируем функцию `f_unsigned()`

10.1.4 Неоптимизирующий GCC

Неоптимизирующий GCC 4.4.1 производит почти такой же код, за исключением `puts()` (2.3.3) вместо `printf()`.

10.1.5 Оптимизирующий GCC

Наблюдательный читатель может спросить, зачем исполнять `CMP` так много раз, если флаги всегда одни и те же? По видимому, оптимизирующий MSVC не может этого делать, но GCC 4.8.1 делает больше оптимизаций:

Листинг 10.4: GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

Мы здесь также видим `JMP puts` вместо `CALL puts / RETN`. Этот прием будет описан немного позже: 11.1.1.

Нужно сказать что x86-код такого типа редок. MSVC 2012, как видно, не может этого делать. С другой стороны, программисты на ассемблере прекрасно осведомлены о том, что инструкции Jcc можно располагать последовательно. Так что если вы видите это где-то, имеется немалая вероятность, что этот фрагмент кода был написан вручную.

Ф-ция f_unsigned() получилась не настолько эстетически короткой:

Листинг 10.5: GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx        ; эту инструкцию можно убрать
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

Так что, алгоритмы оптимизации GCC 4.8.1 наверное, еще пока не идеальны.

10.2 ARM

10.2.1 Оптимизирующий Keil + Режим ARM

Листинг 10.6: Оптимизирующий Keil + Режим ARM

```
.text:000000B8                     EXPORT f_signed
.text:000000B8             f_signed                ; CODE XREF: main+C
.text:000000B8 70 40 2D E9         STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1         MOV     R4, R1
.text:000000C0 04 00 50 E1         CMP     R0, R4
.text:000000C4 00 50 A0 E1         MOV     R5, R0
.text:000000C8 1A 0E 8F C2         ADRGT   R0, aAB         ; "a>b\n"
.text:000000CC A1 18 00 CB         BLGT    __2printf
.text:000000D0 04 00 55 E1         CMP     R5, R4
.text:000000D4 67 0F 8F 02         ADREQ   R0, aAB_0       ; "a==b\n"
.text:000000D8 9E 18 00 0B         BLEQ    __2printf
.text:000000DC 04 00 55 E1         CMP     R5, R4
```

```
.text:000000E0 70 80 BD A8          LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8          LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2          ADR     R0, aAB_1       ; "a<b\n"
.text:000000EC 99 18 00 EA          B       __2printf
.text:000000EC                  ; End of function f_signed
```

Многие инструкции в режиме ARM могут быть исполнены только при некоторых выставленных флагах. Это нередко используется для сравнения чисел, например.

К примеру, инструкция ADD на самом деле может быть представлена как ADDAL, AL означает *Always*, то есть, исполнять всегда. Предикаты кодируются в 4-х старших битах инструкции 32-битных ARM-инструкций (*condition field*). Инструкция безусловного перехода B, на самом деле условная и кодируется так же как и прочие инструкции условных переходов, но имеет AL в *condition field*, то есть, исполняется всегда, игнорируя флаги.

Инструкция ADRGT работает так же, как и ADR, но исполнится только в случае если предыдущая инструкция CMP, сравнивая два числа, обнаружила что одно из них больше второго (*Greater Than*).

Следующая инструкция BLGT ведет себя так же, как и BL и сработает только если результат сравнения был такой же (*Greater Than*). ADRGT записывает в R0 указатель на строку "a>b\n", а BLGT вызывает printf(). Следовательно, эти инструкции с суффиксом -GT, исполнятся только в том случае, если значение в R0 (там a) было больше чем значение в R4 (там b).

Далее мы увидим инструкции ADREQ и BLEQ. Они работают так же, как и ADR и BL, но исполнятся только в случае если значения при сравнении были равны. Перед ними еще один CMP (ведь вызов printf() мог испортить состояние флагов).

Далее мы увидим LDMGEFD, эта инструкция работает так же, как и LDMFD[1], но сработает только в случае если в результате сравнения одно из значений было больше или равно второму (*Greater or Equal*).

Смысл инструкции "LDMGEFD SP!, {R4-R6,PC}" в том, что это как бы эпилог функции, но он сработает только если $a >= b$, только тогда работа функции закончится. Но если это не так, то есть $a < b$, то исполнение дойдет до следующей инструкции "LDMFD SP!, {R4-R6,LR}", это еще один эпилог функции, эта инструкция восстанавливает состояние регистров R4-R6, но и LR вместо PC, таким образом, пока что не делая возврата из функции. Последние две инструкции вызывают printf() со строкой «a<b\n» в качестве единственного аргумента. Безусловный переход на printf() вместо возврата из функции, это то что мы уже рассматривали в секции «printf() с несколькими аргументами», здесь (5.3.2).

Функция f_unsigned точно такая же, но там используются инструкции ADRHI, BLHI, и LDMCSFD эти предикаты (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) аналогичны рассмотренным, но служат для работы с беззнаковыми значениями.

В функции main() ничего для нас нового нет:

Листинг 10.7: main()

```
.text:00000128                  EXPORT main
.text:00000128              main
.text:00000128 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148              ; End of function main
```

Так, в режиме ARM можно обойтись без условных переходов.

Почему это хорошо? Потому что ARM это RISC-процессор имеющий конвейер (pipeline) для исполнения инструкций. Если говорить коротко, то процессору с конвейером тяжело даются переходы вообще, поэтому есть спрос на возможность предсказывания переходов. Очень хорошо если программа имеют как можно меньшее переходов, как условных, так и безусловных, поэтому, инструкции с добавленными предикатами, указывающими, исполнять инструкцию или нет, могут избавить от некоторого количества условных переходов.

В x86 нет аналогичной возможности, если не считать инструкцию CMOVcc, это то же что и MOV, но она срабатывает только при определенных выставленных флагах, обычно, выставленных при помощи CMP во время сравнения.

[1]Load Multiple Full Descending

10.2.2 Оптимизирующий Keil + Режим thumb

Листинг 10.8: Оптимизирующий Keil + Режим thumb

```
.text:00000072             f_signed ; CODE XREF: main+6
.text:00000072 70 B5         PUSH    {R4-R6,LR}
.text:00000074 0C 00         MOVS    R4, R1
.text:00000076 05 00         MOVS    R5, R0
.text:00000078 A0 42         CMP     R0, R4
.text:0000007A 02 DD         BLE     loc_82
.text:0000007C A4 A0         ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8   BL      __2printf
.text:00000082
.text:00000082             loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42         CMP     R5, R4
.text:00000084 02 D1         BNE     loc_8C
.text:00000086 A4 A0         ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8   BL      __2printf
.text:0000008C
.text:0000008C             loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42         CMP     R5, R4
.text:0000008E 02 DA         BGE     locret_96
.text:00000090 A3 A0         ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8   BL      __2printf
.text:00000096
.text:00000096             locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD         POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

В режиме thumb, только инструкции `B` могут быть дополнены условием исполнения (*condition code*), так что, код для режима thumb выглядит привычнее.

`BLE` это обычный переход с условием *Less than or Equal*, `BNE` — *Not Equal*, `BGE` — *Greater than or Equal*.

Функция `f_unsigned` точно такая же, но для работы с беззнаковыми величинами, там используются инструкции`BLS` (*Unsigned lower or same*) и `BCS` (*Carry Set (Greater than or equal)*).

Глава 11

switch()/case/default

11.1 Если вариантов мало

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

11.1.1 x86

Это дает в итоге (MSVC 2010):

Листинг 11.1: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
```

```
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Наша функция со switch()-ем, с небольшим количеством вариантов, это практически аналог подобной конструкции:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Когда вариантов немного, и мы видим подобный код, невозможно сказать с уверенностью, был ли в оригинальном исходном коде switch(), либо просто набор if()-ов. То есть, switch() это синтаксический сахар для большого количества вложенных проверок при помощи if().

В самом выходном коде, в принципе, ничего особо нового для нас здесь, за исключением того, что компилятор зачем-то перекладывает входящую переменную (a) во временную в локальном стеке v64.

Если скомпилировать это при помощи GCC 4.4.1, то будет почти то же самое, даже с максимальной оптимизацией (ключ -O3).

Попробуем, включить оптимизацию кодегенератора MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

Листинг 11.2: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp    _printf
_f    ENDP
```

Вот здесь уже все немного по-другому, причем не без грязных хаков.

Первое: a помещается в EAX и от него отнимается 0. Звучит абсурдно, но нужно это для того, чтобы проверить, 0 ли в EAX был до этого? Если да, то выставится флаг ZF (что означает что результат отнимания 0 от числа стал

0) и первый условный переход JE (*Jump if Equal* или его синоним JZ — *Jump if Zero*) сработает на метку $LN4@f, где выводится сообщение 'zero'. Если первый переход не сработал, от значения отнимается по единице, и если на какой-то стадии образуется в результате 0, то сработает соответствующий переход.

И в конце концов, если ни один из условных переходов не сработал, управление передается printf() с аргументом 'something unknown'.

Второе: мы видим две, мягко говоря, необычные вещи: указатель на сообщение помещается в переменную a, и затем printf() вызывается не через CALL, а через JMP. Объяснение этому простое. Вызывающая функция заталкивает в стек некоторое значение и через CALL вызывает нашу функцию. CALL в свою очередь заталкивает в стек адрес возврата и делает безусловный переход на адрес нашей функции. Наша функция в самом начале (да и в любом её месте, потому что в теле функции нет ни одной инструкции, которая меняет что-то в стеке или в ESP) имеет следующую разметку стека:

- ESP — хранится RA
- ESP+4 — хранится значение a

С другой стороны, чтобы вызвать printf() нам нужна почти такая же разметка стека, только в первом аргументе нужен указатель на строку. Что, собственно, этот код и делает.

Он заменяет свой первый аргумент на другой и затем передает управление printf(), как если бы вызвали не нашу функцию f(), а сразу printf(). printf() выводит некую строку на stdout, затем исполняет инструкцию RET, которая из стека достает RA и управление передается в ту функцию, которая вызывала f(), минуя при этом саму f().

Все это возможно потому что printf() вызывается в f() в самом конце. Все это чем-то даже похоже на longjmp()[1]. И все это, разумеется, сделано для экономии времени исполнения.

Похожая ситуация с компилятором для ARM описана в секции "printf() с несколькими аргументами", здесь (5.3.2).

11.1.2 ARM: Оптимизирующий Keil + Режим ARM

```
.text:0000014C             f1
.text:0000014C 00 00 50 E3   CMP     R0, #0
.text:00000150 13 0E 8F 02   ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A   BEQ     loc_170
.text:00000158 01 00 50 E3   CMP     R0, #1
.text:0000015C 4B 0F 8F 02   ADREQ   R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A   BEQ     loc_170
.text:00000164 02 00 50 E3   CMP     R0, #2
.text:00000168 4A 0F 8F 12   ADRNE   R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02   ADREQ   R0, aTwo  ; "two\n"
.text:00000170
.text:00000170             loc_170 ; CODE XREF: f1+8
.text:00000170                     ; f1+14
.text:00000170 78 18 00 EA   B       __2printf
```

Мы снова не сможем сказать, глядя на этот код, был ли в оригинальном исходном коде switch() либо же несколько if()-в.

Так или иначе, мы снова видим здесь инструкции с предикатами, например, ADREQ (*(Equal)*), которая будет исполняться только если $R0 = 0$, и тогда, в R0 будет загружен адрес строки *«zero\n»*. Следующая инструкция BEQ перенаправит исполнение на loc_170, если $R0 = 0$. Кстати, наблюдательный читатель может спросить, сработает ли BEQ нормально, ведь ADREQ перед ним уже заполнила регистр R0 чем-то другим. Сработает, потому что BEQ проверяет флаги, установленные инструкцией CMP, а ADREQ флаги никак не модифицирует.

Кстати, в ARM имеется также для некоторых инструкций суффикс *-S*, указывающий, что эта инструкция будет модифицировать флаги, а при отсутствии суффикса — не будет. Например, инструкция ADD в отличие от ADDS сложит два числа, но флаги не изменит. Такие инструкции удобно использовать между CMP где выставляются флаги и, например, инструкциями перехода, где флаги используются.

Далее всё просто и знакомо. Вызов printf() один, и в самом конце, мы уже рассматривали подобный трюк здесь (5.3.2). К printf()-у в конце ведут три пути.

Обратите внимание на то что происходит если $a = 2$ и если a не попадает под сравниваемые константы. Инструкция "CMP R0, #2" нужна чтобы узнать $a = 2$ или нет. Если это не так, то при помощи ADRNE (*Not Equal*) в R0 будет загружен указатель на строку *«something unknown \n»*, ведь a уже было проверено на 0 и 1 до этого,

[1] http://en.wikipedia.org/wiki/Setjmp.h

и здесь a точно не попадает под эти константы. Ну а если $R0 = 2$, в R0 будет загружен указатель на строку *«two\n»* при помощи инструкции ADREQ.

11.1.3 ARM: Оптимизирующий Keil + Режим thumb

```
.text:000000D4              f1
.text:000000D4 10 B5        PUSH    {R4,LR}
.text:000000D6 00 28        CMP     R0, #0
.text:000000D8 05 D0        BEQ     zero_case
.text:000000DA 01 28        CMP     R0, #1
.text:000000DC 05 D0        BEQ     one_case
.text:000000DE 02 28        CMP     R0, #2
.text:000000E0 05 D0        BEQ     two_case
.text:000000E2 91 A0        ADR     R0, aSomethingUnkno ; "something unknown\n"
.text:000000E4 04 E0        B       default_case

.text:000000E6              zero_case ; CODE XREF: f1+4
.text:000000E6 95 A0        ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0        B       default_case

.text:000000EA              one_case ; CODE XREF: f1+8
.text:000000EA 96 A0        ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0        B       default_case

.text:000000EE              two_case ; CODE XREF: f1+C
.text:000000EE 97 A0        ADR     R0, aTwo ; "two\n"
.text:000000F0              default_case ; CODE XREF: f1+10
.text:000000F0                                  ; f1+14
.text:000000F0 06 F0 7E F8  BL      __2printf
.text:000000F4 10 BD        POP     {R4,PC}
.text:000000F4              ; End of function f1
```

Как я уже писал, в thumb-режиме нет возможности *присоединять* предикаты к большинству инструкций, так что thumb-код вышел похожим на код x86, вполне понятный.

11.2 И если много

А если ветвлений слишком много, то конечно генерировать слишком длинный код с многочисленными JE/JNE уже не так удобно.

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

11.2.1 x86

Имеем в итоге (MSVC 2010):

Листинг 11.3: MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
```

```
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad   2
$LN11@f:
    DD     $LN6@f ; 0
    DD     $LN5@f ; 1
    DD     $LN4@f ; 2
    DD     $LN3@f ; 3
    DD     $LN2@f ; 4
_f    ENDP
```

Здесь происходит следующее: в теле функции есть набор вызовов `printf()` с разными аргументами. Все они имеют, конечно же, адреса, а также внутренние символические метки, которые присвоил им компилятор. Помимо всего прочего, все эти метки складываются во внутреннюю таблицу `$LN11@f`.

В начале функции, если `a` больше 4, то сразу происходит переход на метку `$LN1@f`, где вызывается `printf()` с аргументом `'something unknown'`.

А если `a` меньше или равно 4, то это значение умножается на 4 и прибавляется адрес таблицы с переходами. Таким образом, получается адрес внутри таблицы, где лежит нужный адрес внутри тела функции. Например, возьмем `a` равным 2. $2 * 4 = 8$ (ведь все элементы таблицы — это адреса внутри 32-битного процесса, таким образом, каждый элемент занимает 4 байта). 8 прибавить к `$LN11@f` — это будет элемент таблицы, где лежит

`$LN4@f`. `JMP` вытаскивает из таблицы адрес `$LN4@f` и делает безусловный переход туда.

Эта таблица иногда называется *jumptable*.

А там вызывается `printf()` с аргументом `'two'`. Дословно, инструкция `jmp DWORD PTR $LN11@f[ecx*4]` означает *перейти по DWORD, который лежит по адресу* `$LN11@f + ecx * 4`.

`npad` (61) это макрос ассемблера, немного выровнять начало таблицы, дабы она располагалась по адресу кратному 4 (или 16). Это нужно для того чтобы процессор мог эффективнее загружать 32-битное значения из памяти, через шину с памятью, кэш-память, и т.д.

Посмотрим, что сгенерирует GCC 4.4.1:

Листинг 11.4: GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
        call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
        leave
        retn
f       endp
```

```
off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

Практически то же самое, за исключением мелкого нюанса: аргумент из `arg_0` умножается на 4 при помощи сдвига влево на 2 бита (это почти то же самое что и умножение на 4) (17.3.1). Затем адрес метки внутри функции берется из массива `off_804855C` и адресуется при помощи вычисленного индекса.

11.2.2 ARM: Оптимизирующий Keil + Режим ARM

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable 00000178 default case

00000180
00000180             loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA     B       zero_case        ; jumptable 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA     B       one_case         ; jumptable 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA     B       two_case         ; jumptable 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA     B       three_case       ; jumptable 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA     B       four_case        ; jumptable 00000178 case 4

00000194
00000194             zero_case ; CODE XREF: f2+4
00000194                       ; f2:loc_180
00000194 EC 00 8F E2     ADR     R0, aZero        ; jumptable 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8

0000019C
0000019C             one_case ; CODE XREF: f2+4
0000019C                      ; f2:loc_184
0000019C EC 00 8F E2     ADR     R0, aOne         ; jumptable 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF: f2+4
000001A4                      ; f2:loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo         ; jumptable 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF: f2+4
000001AC                        ; f2:loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree       ; jumptable 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8
```

```
000001B4
000001B4             four_case ; CODE XREF: f2+4
000001B4                       ; f2:loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF: f2+24
000001B8                       ; f2+2C
000001B8 66 18 00 EA     B       __2printf

000001BC
000001BC             default_case ; CODE XREF: f2+4
000001BC                          ; f2+8
000001BC D4 00 8F E2     ADR     R0, aSomethingUnkno ; jumptable 00000178 default case
000001C0 FC FF FF EA     B       loc_1B8
000001C0             ; End of function f2
```

В этом коде используется та особенность режима ARM, что все инструкции в этом режиме имеют длину 4 байта.

Итак, не будем забывать, что максимальное значение для a это 4, всё что выше, должно вызвать вывод строки *«something unknown\n»* .

Самая первая инструкция ‘‘`CMP R0, #5`’’ сравнивает входное значение в a с 5.

Следующая инструкция ‘‘`ADDCC PC, PC, R0,LSL#2`’’ [2] сработает только в случае если $R0 < 5$ (*CC=Carry clear / Less than*). Следовательно, если `ADDCC` не сработает (это случай с $R0 \geq 5$), выполнится переход на метку *default_case*.

Но если $R0 < 5$ и `ADDCC` сработает, то произойдет следующее:

Значение в `R0` умножается на 4. Фактически, `LSL#2` в конце инструкции означает “сдвиг влево на 2 бита”. Но как будет видно позже (17.3.1) в секции “Сдвиги” , сдвиг влево на 2 бита это как раз эквивалентно его умножению на 4.

Затем полученное $R0 * 4$ прибавляется к текущему значению PC, совершая, таким образом, переход на одну из расположенных ниже инструкций `B` (*Branch*).

На момент исполнения `ADDCC`, содержимое PC на 8 байт больше (`0x180`) чем адрес по которому расположена сама инструкция `ADDCC` (`0x178`), либо, говоря иным языком, на 2 инструкции больше.

Это связано с работой конвейера процессора ARM: пока исполняется инструкция `ADDCC`, процессор уже начинает обрабатывать инструкцию после следующей, поэтому PC указывает туда.

В случае, если $a = 0$, тогда к PC ничего не будет прибавлено, в PC запишется актуальный на тот момент PC (который больше на 8) и произойдет переход на метку *loc_180*, это на 8 байт дальше от места где находится инструкция `ADDCC`.

В случае, если $a = 1$, тогда в PC запишется $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 16 = 0x184$, это адрес метки *loc_184*.

При каждой добавленной к a единице, итоговый PC увеличивается на 4. 4 это как раз длина инструкции в режиме ARM и одновременно с этим, длина каждой инструкции `B`, их здесь следует 5 в ряд.

Каждая из этих пяти инструкций `B`, передает управление дальше, где собственно и происходит то, что запрограммировано в *switch()*. Там происходит загрузка указателя на свою строку, и т.д.

11.2.3 ARM: Оптимизирующий Keil + Режим thumb

```
000000F6                     EXPORT f2
000000F6                f2
000000F6 10 B5               PUSH    {R4,LR}
000000F8 03 00               MOVS    R3, R0
000000FA 06 F0 69 F8         BL      __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05                  DCB 5
000000FF 04 06 08 0A 0C 10   DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for switch statement
00000105 00                  ALIGN 2
00000106
00000106                zero_case ; CODE XREF: f2+4
00000106 8D A0               ADR     R0, aZero ; jumptable 000000FA case 0
00000108 06 E0               B       loc_118
```

[2] ADD — складывание чисел

```
0000010A
0000010A                    one_case ; CODE XREF: f2+4
0000010A 8E A0                  ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0                  B       loc_118

0000010E
0000010E                    two_case ; CODE XREF: f2+4
0000010E 8F A0                  ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0                  B       loc_118

00000112
00000112                    three_case ; CODE XREF: f2+4
00000112 90 A0                  ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0                  B       loc_118

00000116
00000116                    four_case ; CODE XREF: f2+4
00000116 91 A0                  ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                    loc_118 ; CODE XREF: f2+12
00000118                            ; f2+16
00000118 06 F0 6A F8            BL      __2printf
0000011C 10 BD                  POP     {R4,PC}

0000011E
0000011E                    default_case ; CODE XREF: f2+4
0000011E 82 A0                  ADR     R0, aSomethingUnkno ; jumptable 000000FA default case
00000120 FA E7                  B       loc_118


000061D0                        EXPORT __ARM_common_switch8_thumb
000061D0                    __ARM_common_switch8_thumb ; CODE XREF: example6_f2+4
000061D0 78 47                  BX      PC

000061D2 00 00                  ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                    __32__ARM_common_switch8_thumb ; CODE XREF: ↙
    ↳ __ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

В режимах thumb и thumb-2, уже нельзя надеяться на то что все инструкции будут иметь одну длину. Можно даже сказать, что в этих режимах инструкции переменной длины, как в x86.

Так что здесь добавляется специальная таблица, содержащая информацию о том, как много вариантов здесь, не включая default-варианта, и смещения, для каждого варианта, каждое кодирует метку, куда нужно передать управление в соответствующем случае.

Для того чтобы работать с таблицей и совершить переход, вызывается служебная функция *__ARM_common_switch8_thumb*. Она начинается с инструкции `"BX PC"` , чья функция — переключить процессор в ARM-режим. Далее функция, работающая с таблицей. Она слишком сложная для рассмотрения в данном месте, так что я пропущу объяснения.

Но можно отметить, что эта функция использует регистр LR как указатель на таблицу. Действительно, после вызова этой функции, в LR был записан адрес после инструкции `"BL __ARM_common_switch8_thumb"`, а там как раз и начинается таблица.

Еще можно отметить что код для этого выделен в отдельную функцию для того, чтобы и в других местах, в похожих случаях, обрабатывались *switch()*-и, и не нужно было каждый раз генерировать во всех этих местах такой фрагмент кода.

IDA распознала эту служебную функцию и таблицу автоматически, дописав комментарии к меткам вроде `jumptable 000000FA case 0`.

Глава 12

Циклы

12.1 x86

Для организации циклов, в архитектуре x86 есть старая инструкция `LOOP`, она проверяет значение регистра `ECX` и если оно не 0, делает декремент `ECX` и переход по метке указанной в операнде. Возможно, эта инструкция не слишком удобная, поэтому я не видел современных компиляторов, которые использовали бы её. Так что, если вы видите где-то `LOOP`, то это, с большой вероятностью, вручную написанный код на ассемблере.

Кстати, в качестве домашнего задания, вы можете попытаться объяснить, чем именно эта инструкция неудобна.

Циклы на Си/Си++ создаются при помощи `for()`, `while()`, `do/while()`.

Начнем с `for()`.

Это выражение описывает инициализацию, условие, что делать после каждой итерации (инкремент/декремент) и тело цикла.

```
for (инициализация; условие; после каждой итерации)
{
  тело_цикла;
}
```

Примерно так же, генерируемый код и будет состоять из этих четырех частей.

Возьмем пример:

```
#include <stdio.h>

void f(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                f(i);

        return 0;
};
```

Имеем в итоге (MSVC 2010):

Листинг 12.1: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; инициализация цикла
```

```
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; то что мы делаем после каждой итерации:
    add    eax, 1                  ; добавляем 1 к i
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; это условие проверяется *перед* каждой итерацией
    jge    SHORT $LN1@main         ; если i больше или равно 10, заканчиваем цикл
    mov    ecx, DWORD PTR _i$[ebp] ; тело цикла: вызов функции f(i)
    push   ecx
    call   _f
    add    esp, 4
    jmp    SHORT $LN2@main         ; переход на начало цикла
$LN1@main:                         ; конец цикла
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

В принципе, ничего необычного.

GCC 4.4.1 выдает примерно такой же код, с небольшой разницей:

Листинг 12.2: GCC 4.4.1

```
main            proc near               ; DATA XREF: _start+17

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; инициализация i
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    f
                add     [esp+20h+var_4], 1  ; инкремент i

loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; если i<=9, продолжаем цикл
                mov     eax, 0
                leave
                retn
main            endp
```

Интересно становится, если скомпилируем этот же код при помощи MSVC 2010 с включенной оптимизацией (`/Ox`):

Листинг 12.3: Оптимизирующий MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _f
    inc    esi
    add    esp, 4
```

```
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Здесь происходит следующее: переменную i компилятор не выделяет в локальном стеке, а выделяет целый регистр под нее: `ESI`. Это возможно для маленьких функций, где мало локальных переменных.

В принципе, все то же самое, только теперь одна важная особенность: `f()` не должна менять значение `ESI`. Наш компилятор уверен в этом, а если бы и была необходимость использовать регистр `ESI` в функции `f()`, то её значение сохранялось бы в стеке. Примерно так же, как и в нашем листинге: обратите внимание на `PUSH ESI/POP ESI` в начале и конце функции.

Попробуем GCC 4.4.1 с максимальной оптимизацией (`-O3`):

Листинг 12.4: Оптимизирующий GCC 4.4.1

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    f
                mov     [esp+10h+var_10], 3
                call    f
                mov     [esp+10h+var_10], 4
                call    f
                mov     [esp+10h+var_10], 5
                call    f
                mov     [esp+10h+var_10], 6
                call    f
                mov     [esp+10h+var_10], 7
                call    f
                mov     [esp+10h+var_10], 8
                call    f
                mov     [esp+10h+var_10], 9
                call    f
                xor     eax, eax
                leave
                retn
main            endp
```

Однако, GCC просто *развернул* цикл[1].

Делается это в тех случаях, когда итераций не слишком много, как в нашем примере, и можно немного сэкономить время, убрав все инструкции, обеспечивающие цикл. В качестве обратной стороны медали, размер кода увеличился.

OK, увеличим максимальное значение i в цикле до 100 и попробуем снова. GCC выдаст подобное:

Листинг 12.5: GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
```

[1] loop unwinding в англоязычной литературе

```
                push    ebx
                mov     ebx, 2                  ; i=2
                sub     esp, 1Ch

; выравнивание метки loc_80484D0 (начало тела цикла) по 16-байтной границе
                nop

loc_80484D0:
                mov     [esp+20h+var_20], ebx ; передать i как первый аргумент для f()
                add     ebx, 1    ; i++
                call    f
                cmp     ebx, 64h ; i==100?
                jnz     short loc_80484D0 ; если нет, продолжать
                add     esp, 1Ch
                xor     eax, eax ; вернуть 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

Это уже похоже на то что сделал MSVC 2010 в режиме оптимизации (`/Ox`). За исключением того, что под переменную `i` будет выделен регистр `EBX`. GCC уверен, что этот регистр не будет модифицироваться внутри `f()`, а если вдруг это и придётся там сделать, то его значение будет сохранено в начале функции, прямо как в `main()` здесь.

12.1.1 OllyDbg

Скомпилируем наш пример в MSVC 2010 с `/Ox` и `/Ob0` и загрузим в OllyDbg.

Оказывается, OllyDbg может обнаруживать простые циклы и показывать их в квадратных скобках, для удобства: илл. 12.1.

Трассируя (F8 (сделать шаг не входя в ф-цию)) мы видим как `ESI` увеличивается на 1. Например, здесь `ESI` =i=6: илл. 12.2.

9 это последнее значение цикла. Поэтому `JL` после инкремента не срабатывает и ф-ция заканчивается: илл. 12.3.

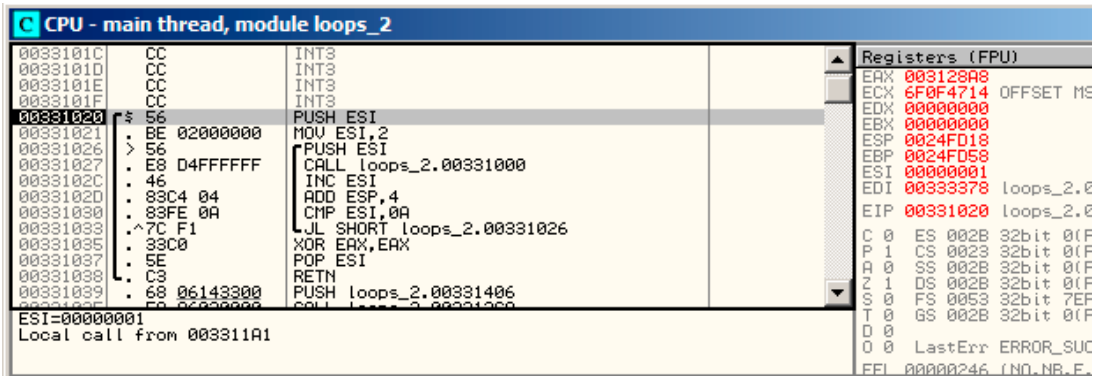


Рис. 12.1: OllyDbg: начало `main()`

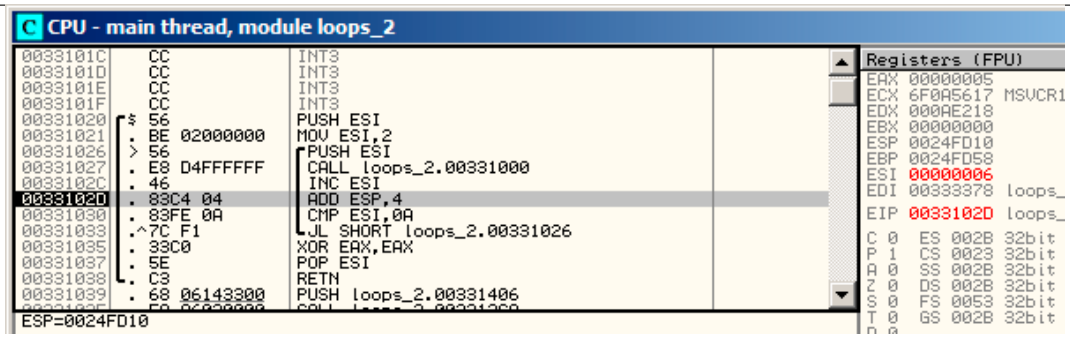


Рис. 12.2: OllyDbg: тело цикла только что отработало с i=6

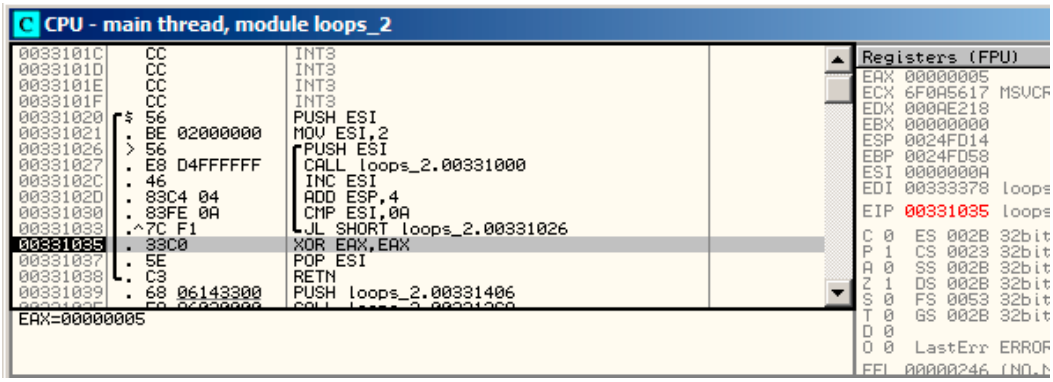


Рис. 12.3: OllyDbg: ESI =10, конец цикла

12.1.2 tracer

Как видно, трассировать вручную цикл в отладчике это не очень удобно. Это одна из причин, почему я писал tracer для себя.

Я открываю скомпилированный пример в IDA, нахожу там адрес инструкции PUSH ESI (передающей единственный аргумент в f()) а это 0x401026 у меня и запускаю tracer:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

Опция BPX просто ставит брякпойнт по адресу и затем будет выдавать состояние регистров.

В tracer.log после запуска я вижу следующее:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
```

```
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Видно как значение ESI последовательно изменяется от 2 до 9.

И дажее более того, в tracer можно собирать значения регистров по всем адресам внутри ф-ции. Там это называется *trace*. Каждая инструкция трассируется, значения самых интересных регистров запоминаются. Затем генерируется .idc-скрипт для IDA, который добавляет комментарии. Итак, в IDA я узнал что адрес `main()` это 0x00401020 и запускаю:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF означает установить брякпойнт на ф-ции.

Получаю в итоге скрипты `loops_2.exe.idc` и `loops_2.exe_clear.idc`. Загружаю `loops_2.exe.idc` в IDA и увижу следующее: илл. 12.4

Видно что ESI бывает от 2 до 9 в начале тела цикла, но после инкремента он в пределах [3..0xA]. Видно также что ф-ция `main()` заканчивается с 0 в EAX.

tracer также генерирует `loops_2.exe.txt`, содержащий адреса инструкций, сколько раз была исполнена каждая и значения регистров:

Листинг 12.6: loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested maximum level (1) reached, ↙
    ↳ skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

Так можно использовать grep.

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Рис. 12.4: IDA с загруженным .idc-скриптом

12.2 ARM

12.2.1 Неоптимизирующий Keil + Режим ARM

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF: main+1C
        MOV     R0, R4
        BL      f
        ADD     R4, R4, #1

loc_368  ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Счетчик итераций `i` будет храниться в регистре `R4`.

Инструкция "`MOV R4, #2`" просто инициализирует i.

Инструкции "`MOV R0, R4`" и "`BL f`" составляют тело цикла, первая инструкция готовит аргумент для функции `f()` и вторая собственно вызывает её.

Инструкция "`ADD R4, R4, #1`" прибавляет единицу к i при каждой итерации.

"`CMP R4, #0xA`" сравнивает `i` с `0xA` (10). Следующая за ней инструкция `BLT` (*Branch Less Than*) совершит переход, если i меньше чем 10.

В противном случае, в `R0` запишется 0 (потому что наша функция возвращает 0) и произойдет выход из функции.

12.2.2 Оптимизирующий Keil + Режим thumb

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2
```

```
loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      example7_f
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Практически, всё то же самое.

12.2.3 Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
_main
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

На самом деле, в моей функции `f()` было такое:

```
void f(int i)
{
    // do something here
    printf ("%d\n", i);
};
```

Так что, LLVM не только *развернул* цикл, но также и представил мою очень простую функцию `f()` как *inline-вую*, и вставил её тело вместо цикла 8 раз. Это возможно, когда функция очень простая, как та что у меня, и когда она вызывается не очень много раз, как здесь.

12.3 Еще кое-что

По генерируемому коду мы видим следующее: после инициализации i, тело цикла не исполняется, а исполняется сразу проверка условия i, а лишь затем исполняется тело цикла. Это правильно. Потому что если условие в

самом начале не выполняется, тело цикла исполнять нельзя. Так может быть, например, в таком случае:

```
for (i=0; i<total_entries_to_process; i++)
    тело_цикла;
```

Если *total_ entries_ to_ process* равно 0, тело цикла не должно исполниться ни разу. Поэтому проверка условия происходит перед тем как исполнить само тело.

Впрочем, оптимизирующий компилятор может переставить проверку условия и тело цикла местами, если он уверен, что описанная здесь ситуация невозможна, как в случае с нашим простейшим примером и компиляторами Keil, Xcode (LLVM), MSVC и GCC в режиме оптимизации.

Глава 13

strlen()

Еще немного о циклах. Часто, функция `strlen()`[1] реализуется при помощи `while()`. Например, как это сделано в стандартных библиотеках MSVC:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

13.1 x86

13.1.1 Неоптимизирующий MSVC

Итак, компилируем:

```
_eos$ = -4                      ; size = 4
_str$ = 8                       ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]   ; взять указатель на символ из str
    mov     DWORD PTR _eos$[ebp], eax   ; и переложить его в нашу локальную переменную eos
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]   ; ecx=eos

    ; взять байт, на который указывает ecx и положить его в edx с signed-расширением

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]   ; eax=eos
    add     eax, 1                      ; увеличить eax на единицу
    mov     DWORD PTR _eos$[ebp], eax   ; положить eax назад в eos
    test    edx, edx                    ; edx==0?
    je      SHORT $LN1@strlen_          ; да, то что лежит в edx это ноль, выйти из цикла
    jmp     SHORT $LN2@strlen_          ; продолжаем цикл
```

[1] подсчет длины строки в Си

```
$LN1@strlen_:

    ; здесь мы вычисляем разницу двух указателей

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                       ; отнимаем от разницы еще единицу и возвращаем ↙
    ↳ результат
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Здесь две новых инструкции: MOVSX (13.1.1) и TEST.

О первой: MOVSX (13.1.1) предназначен для того чтобы взять байт из какого-либо места в памяти и положить его, в нашем случае, в регистр EDX. Но регистр EDX — 32-битный. MOVSX (13.1.1) означает *MOV with Sign-Extent*. Оставшиеся биты с 8-го по 31-й MOVSX (13.1.1) сделает единицей, если исходный байт в памяти имеет знак *минус*, или заполнит нулями, если знак *плюс*.

И вот зачем все это.

По стандарту Си/Си++, тип *char* — знаковый. Если у нас есть две переменные, одна *char*, а другая *int* (*int* тоже знаковый), и если в первой переменной лежит −2 (что кодируется как 0xFE) и мы просто переложим это в *int*, то там будет 0x000000FE, а это, с точки зрения *int*, даже знакового, будет 254, но никак не −2. −2 в переменной *int* кодируется как 0xFFFFFFFE. И для того чтобы значение 0xFE из переменной типа *char* переложить в знаковый *int* с сохранением всего, нужно узнать его знак, и затем заполнить остальные биты. Это делает MOVSX (13.1.1).

См. также об этом раздел "*Представление знака в числах*" (32).

Хотя, конкретно здесь, компилятору врядли была особая надобность хранить значение *char* в регистре EDX а не его восьмибитной части, скажем, DL. Но получилось, как получилось: должно быть, register allocator компилятора сработал именно так.

Позже выполняется TEST EDX, EDX. Об инструкции TEST читайте в разделе о битовых полях (17). Но конкретно здесь, эта инструкция просто проверяет состояние регистра EDX на 0.

13.1.2 Неоптимизирующий GCC

Попробуем GCC 4.4.1:

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
```

```
                leave
                retn
strlen          endp
```

Результат очень похож на MSVC, вот только здесь используется MOVZX а не MOVSX (13.1.1). MOVZX означает *MOV with Zero-Extent*. Эта инструкция перекладывает какое-либо значение в регистр и остальные биты выставляет в 0. Фактически, преимущество этой инструкции только в том, что она позволяет заменить две инструкции сразу: xor eax, eax / mov al, [...].

С другой стороны, нам очевидно, что здесь можно было бы написать вот так: mov al, byte ptr [eax] / test al, al — это тоже самое, хотя старшие биты EAX будут "замусорены". Но, будем считать, что это погрешность компилятора — он не смог сделать код более экономным или более понятным. Строго говоря, компилятор вообще не нацелен на то чтобы генерировать понятный (для человека) код.

Следующая новая инструкция для нас — SETNZ. В данном случае, если в AL был не ноль, то test al, al выставит флаг ZF в 0, а SETNZ, если ZF==0 (*NZ* значит *not zero*) выставит 1 в AL. Смысл этой процедуры в том, что, если говорить человеческим языком, *если AL не ноль, то выполнить переход на* loc_80483F0. Компилятор выдал немного избыточный код, но не будем забывать, что оптимизация выключена.

13.1.3 Оптимизирующий MSVC

Теперь скомпилируем все то же самое в MSVC 2012, но с включенной оптимизацией (/Ox):

Листинг 13.1: MSVC 2012 /Ox /Ob0

```
_str$ = 8                       ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> указатель на строку
        mov     eax, edx                    ; переложить в EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; нет, продолжаем цикл
        sub     eax, edx                    ; вычисляем разницу указателей
        dec     eax                         ; декремент EAX
        ret     0
_strlen ENDP
```

Здесь все попроще стало. Но следует отметить, что компилятор обычно может так хорошо использовать регистры только на не очень больших функциях с не очень большим количеством локальных переменных.

INC/DEC — это инструкции инкремента-декремента, попросту говоря: увеличить на единицу или уменьшить.

13.1.4 Оптимизирующий MSVC + OllyDbg

Можем попробовать этот (соптимизированный) пример в OllyDbg. Вот самая первая итерация: илл. 13.1. Видно что OllyDbg обнаружил цикл и, для удобства, *свернула* инструкции тела цикла в скобке. Нажав правой кнопкой на EAX, можно выбрать "Follow in Dump" и позиция в окне памяти будет как раз там где надо. Здесь мы видим в памяти строку "hello!". После нее имеется как минимум 1 нулевой байт, затем случайный мусор. Если OllyDbg видит что в регистре содержится адрес указывающий на строку, он показывает эту строку.

Нажем F8 (сделать шаг не входя в ф-цию) столько раз, чтобы текущий адрес снова был в начале тела цикла: илл. 13.2. Видно что EAX уже содержит адрес второго символа в строке.

Будем нажимать F8 достаточное количество раз, чтобы выйти из цикла: илл. 13.3. Увидим что EAX теперь содержит адрес нулевого байта, следующего сразу за строкой. А EDX так и не менялся, он всё еще указывает на начало строки. Здесь сейчас будет вычисляться разница между этими двумя адресами.

Инструкция SUB исполнилась: илл. 13.4. Разница в EAX— 7. Действительно, длина строки "hello!" — 6, но вместе с нулевым байтом — 7. Но strlen() должна возвращать количество ненулевых символов в строке. Так что сейчас будет исполняться декремент и выход из ф-ции.

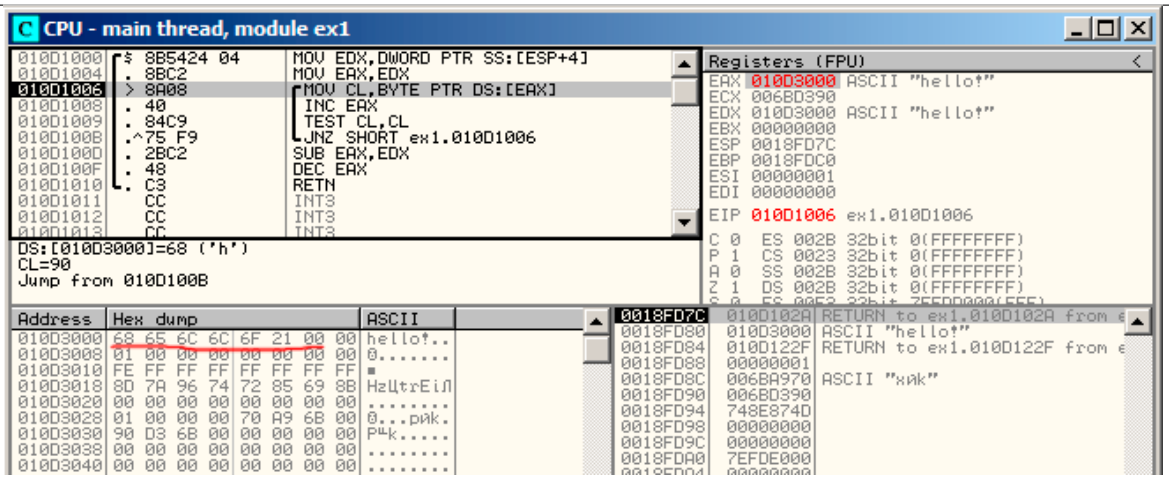


Рис. 13.1: OllyDbg: начало первой итерации

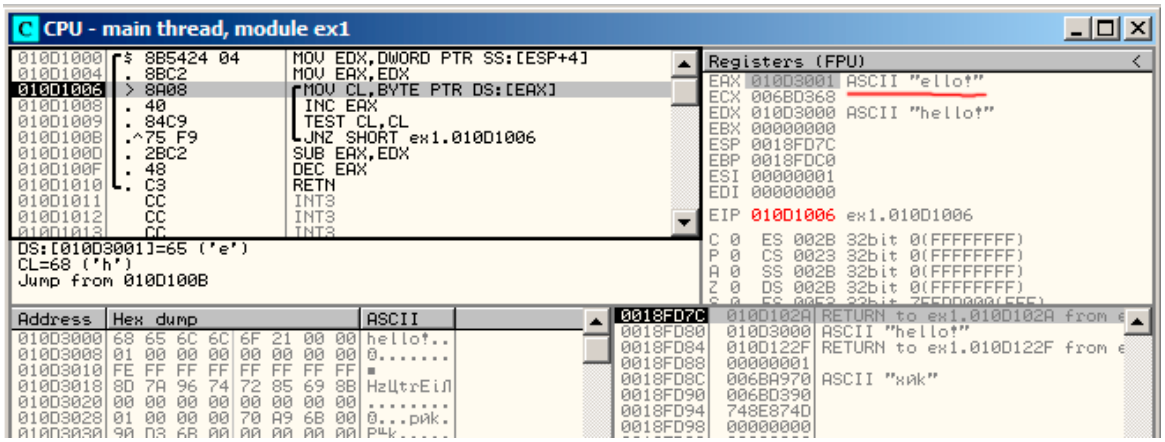


Рис. 13.2: OllyDbg: начало второй итерации

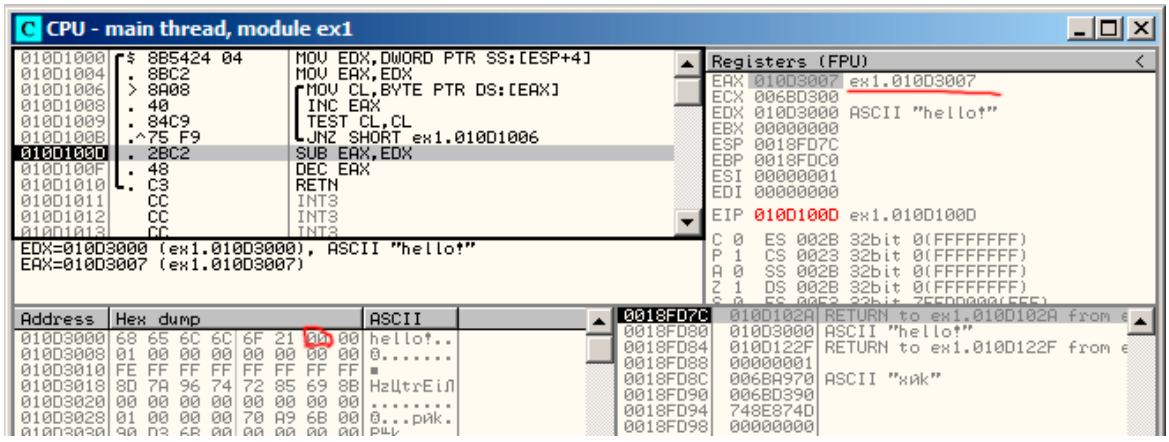


Рис. 13.3: OllyDbg: сейчас будет вычисление разницы указателей

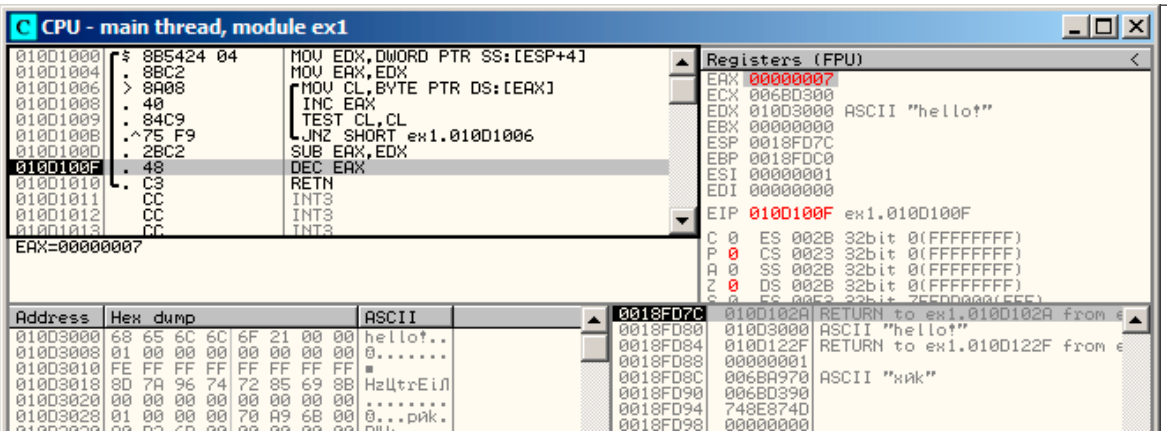


Рис. 13.4: OllyDbg: сейчас будет декремент EAX

13.1.5 Оптимизирующий GCC

Попробуем GCC 4.4.1 с включенной оптимизацией (ключ -O3:

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

Здесь GCC не очень отстает от MSVC за исключением наличия MOVZX.

Впрочем, MOVZX здесь явно можно заменить на mov dl, byte ptr [eax].

Но, возможно, компилятору GCC просто проще помнить, что у него под переменную типа *char* отведен целый 32-битный регистр и быть уверенным в том, что старшие биты регистра не будут замусорены.

Далее мы видим новую для нас инструкцию NOT. Эта инструкция инвертирует все биты в операнде. Можно сказать, что здесь это синонимично инструкции XOR ECX, 0ffffffffh. NOT и следующая за ней инструкция ADD вычисляют разницу указателей и отнимают от результата единицу. Только происходит это слегка по-другому. Сначала ECX, где хранится указатель на *str*, инвертируется и от него отнимается единица.

См. также раздел: “Представление знака в числах” (32).

Иными словами, в конце функции, после цикла, происходит примерно следующее:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

. . . что эквивалентно:

```
ecx=str;
eax=eos;
```

```
eax=eax-ecx;
eax=eax-1;
return eax
```

Но почему GCC решил, что так будет лучше? Снова не берусь сказать. Но я не сомневаюсь, что эти оба варианта работают примерно равноценно в плане эффективности и скорости.

13.2 ARM

13.2.1 Неоптимизирующий Xcode (LLVM) + Режим ARM

Листинг 13.2: Неоптимизирующий Xcode (LLVM) + Режим ARM

```
_strlen

eos  = -8
str  = -4

     SUB    SP, SP, #8 ; allocate 8 bytes for local variables
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; deallocate 8 bytes for local variables
     BX     LR
```

Неоптимизирующий LLVM генерирует слишком много кода, зато на этом примере можно посмотреть, как функции работают с локальными переменными в стеке. В нашей функции только локальных переменных две, это два указателя, *eos* и *str*.

В этом листинге, сгенерированном при помощи IDA, я переименовал *var_8* и *var_4* в *eos* и *str* вручную.

Итак, первые несколько инструкций просто сохраняют входное значение в переменных *str* и *eos*.

Начиная с метки *loc_2CB8*, начинается тело цикла.

Первые три инструкции в теле цикла (`LDR`, `ADD`, `STR`) загружают значение *eos* в `R0`, затем происходит инкремент значения и оно сохраняется назад в локальной переменной *eos* расположенной в стеке.

Следующая инструкция "`LDRSB R0, [R0]`" (*Load Register Signed Byte*) загружает байт из памяти по адресу `R0`, расширяет его до 32-бит считая его знаковым (signed) и сохраняет в `R0`. Это немного похоже на инструкцию `MOVSX` (13.1.1) в x86. Компилятор считает этот байт знаковым (signed), потому что тип *char* по стандарту Си — знаковый. Об это я уже немного писал (13.1.1) в этой же секции, но посвященной x86.

Следует также заметить, что, в ARM нет возможности использовать 8-битную или 16-битную часть регистра, как это возможно в x86. Вероятно, это связано с тем что за x86 тянется длинный шлейф совместимости со своими предками, такими как 16-битный 8086 и даже 8-битный 8080, а ARM разрабатывался с чистого листа как 32-битный RISC-процессор. Следовательно, чтобы работать с отдельными байтами на ARM, так или иначе, придется использовать 32-битные регистры.

Итак, `LDRSB` загружает символ из строки в `R0`, по одному. Следующие инструкции `CMP` и BEQ проверяют, является ли этот символ 0. Если не 0, то происходит переход на начало тела цикла. А если 0, выходим из цикла.

В конце функции вычисляется разница между *eos* и *str*, вычитается еще единица и вычисленное значение возвращается через `R0`.

N.B. В этой функции не сохранялись регистры. Это потому что, по стандарту, регистры R0-R3 называются также "scratch registers", они предназначены для передачи аргументов, их значения не нужно восстанавливать при выходе из функции, потому что они больше не нужны в вызывающей функции. Таким образом, их можно использовать как захочется А так как никакие больше регистры не используются, то и сохранять нечего. Поэтому, управление можно вернуть назад вызывающей функции простым переходом (BX), по адресу в регистре LR.

13.2.2 Оптимизирующий Xcode (LLVM) + Режим thumb

Листинг 13.3: Оптимизирующий Xcode (LLVM) + Режим thumb

```
_strlen
            MOV       R1, R0

loc_2DF6                          ; CODE XREF: _strlen+8
            LDRB.W    R2, [R1],#1
            CMP       R2, #0
            BNE       loc_2DF6
            MVNS      R0, R0
            ADD       R0, R1
            BX        LR
```

Оптимизирующий LLVM решил, что под переменные *eos* и *str* выделять место в стеке не обязательно, и эти переменные можно хранить прямо в регистрах. Перед началом тела цикла, *str* будет находиться в R0, а *eos* — в R1.

Инструкция "LDRB.W R2, [R1],#1" загружает в R2 байт из памяти по адресу R1, расширяя его как знаковый (signed), до 32-битного значения, но не только это. #1 в конце инструкции называется "Post-indexed addressing", это значит, что после загрузки байта, к R1 добавится единица. Это очень удобно для работы с массивами.

Такого режима адресации в x86 нет, но он есть в некоторых других процессорах, даже на PDP-11. Существует байка, что режимы пре-инкремента, пост-инкремента, пре-декремента и пост-декремента адреса в PDP-11, были "виновны" в появлении таких конструкций языка Си (который разрабатывался на PDP-11) как *ptr++, *++ptr, *ptr--, *--ptr. Кстати, это является труднозапоминаемой особенностью в Си. Дела обстоят так:

термин в Си	термин в ARM	выражение Си	как это работает
Пост-инкремент	post-indexed addressing	*ptr++	использовать значение *ptr, затем инкремент указателя ptr
Пост-декремент	post-indexed addressing	*ptr--	использовать значение *ptr, затем декремент указателя ptr
Пре-инкремент	pre-indexed addressing	*++ptr	инкремент указателя ptr, затем использовать значение *ptr
Пре-декремент	post-indexed addressing	*--ptr	декремент указателя ptr, затем использовать значение *ptr

Деннис Ритчи (один из создателей ЯП Си) указывал, что, это, вероятно, придумал Кен Томпсон (еще один создатель Си), потому что подобная возможность процессора имелась еще в PDP-7 [28] [29]. Таким образом, компиляторы с ЯП Си на тот процессор, где это есть, могут использовать это.

Далее в теле цикла можно увидеть CMP и BNE[2], они продолжают работу цикла до тех пор, пока не будет встречен 0.

После конца цикла MVNS[3] (инвертирование всех бит, аналог NOT на x86) и ADD вычисляют $eos - str - 1$. На самом деле, эти две инструкции вычисляют $R0 = str + eos$, что эквивалентно тому, что было в исходном коде, а почему это так, я уже описывал чуть раньше, здесь (13.1.5).

Вероятно, LLVM, как и GCC, посчитал что такой код будет короче, или быстрее.

13.2.3 Оптимизирующий Keil + Режим ARM

Листинг 13.4: Оптимизирующий Keil + Режим ARM

```
_strlen
                MOV     R1, R0
```

[2](PowerPC, ARM) Branch if Not Equal
[3]MoVe Not

```
loc_2C8 ; CODE XREF: _strlen+14
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Практически то же самое что мы уже видели, за тем исключением что выражение $str - eos - 1$ может быть вычислено не в самом конце функции, а прямо в теле цикла. Суффикс `-EQ`, как мы помним, означает что инструкция будет выполнена только если операнды в исполненной перед этим инструкции `CMP` были равны. Таким образом, если в `R0` будет 0, обе инструкции `SUBEQ` исполнятся и результат останется в `R0`.

Глава 14

Деление на 9

Простая функция:

```
int f(int a)
{
        return a/9;
};
```

14.1 x86

. . . компилируется вполне предсказуемо:

Листинг 14.1: MSVC

```
_a$ = 8             ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq            ; знаковое расширение EAX до EDX:EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f  ENDP
```

IDIV делит 64-битное число хранящееся в паре регистров EDX:EAX на значение в ECX. В результате, EAX будет содержать частное[1], а EDX — остаток от деления. Результат возвращается из функции через EAX, так что после операции деления, это значение не перекладывается больше никуда, оно уже там где надо. Из-за того, что IDIV требует пару регистров EDX:EAX, то перед этим инструкция CDQ расширяет EAX до 64-битного значения учитывая знак, так же, как это делает MOVSX (13.1.1). Со включенной оптимизацией (/Ox) получается:

Листинг 14.2: Оптимизирующий MSVC

```
_a$ = 8                     ; size = 4
_f    PROC

    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177   ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31          ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

[1] результат деления

Это — деление через умножение. Умножение конечно быстрее работает. Поэтому можно используя этот трюк [2] создать код эквивалентный тому что мы хотим и работающий быстрее.

В оптимизации компиляторов, это также называется “strength reduction”.

GCC 4.4.1 даже без включенной оптимизации генерирует примерно такой же код, как и MSVC с оптимизацией:

Листинг 14.3: Неоптимизирующий GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     edx, 954437177 ; 38E38E39h
        mov     eax, ecx
        imul    edx
        sar     edx, 1
        mov     eax, ecx
        sar     eax, 1Fh
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        pop     ebp
        retn
f       endp
```

14.2 ARM

В процессоре ARM, как и во многих других “чистых” (pure) RISC-процессорах нет инструкции деления. Нет также возможности умножения на 32-битную константу одной инструкцией. При помощи одного любопытного трюка (или *хака*)[3], можно обойтись только тремя действиями: сложением, вычитанием и битовыми сдвигами (17).

Пример деления 32-битного числа на 10 из [20, 3.3 Division by a Constant]. На выходе и частное и остаток.

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10             ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up remainder
    MOV    pc, lr
```

14.2.1 Оптимизирующий Xcode (LLVM) + Режим ARM

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV    R1, 0x38E38E39
__text:00002C60 10 F1 50 E7              SMMUL  R0, R0, R1
__text:00002C64 C0 10 A0 E1              MOV    R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0              ADD    R0, R1, R0,LSR#31
```

[2] Читайте подробнее о делении через умножение в [35, 10-3]

[3] hack

```
__text:00002C6C 1E FF 2F E1             BX      LR
```

Этот код почти тот же, что сгенерирован MSVC и GCC в режиме оптимизации. Должно быть, LLVM использует тот же алгоритм для поиска констант.

Наблюдательный читатель может спросить, как MOV записала в регистр сразу 32-битное число, ведь это невозможно в режиме ARM. Действительно невозможно, но как мы видим, здесь на инструкцию 8 байт вместо стандартных 4-х, на самом деле, здесь 2 инструкции. Первая инструкция загружает в младшие 16 бит регистра значение 0x8E39, а вторая инструкция, на самом деле MOVT, загружающая в старшие 16 бит регистра значение 0x383E. IDA распознала эту последовательность и для краткости, сократила всё это до одной "псевдо-инструкции".

Инструкция SMMUL (*Signed Most Significant Word Multiply*) умножает числа считая их знаковыми (signed) и оставляет в R0 старшие 32 бита результата, не сохраняя младшие 32 бита.

Инструкция ''MOV R1, R0,ASR#1'' это арифметический сдвиг право на один бит.

''ADD R0, R1, R0,LSR#31'' это $R0 = R1 + R0 >> 31$

Дело в том, что в режиме ARM нет отдельных инструкций для битовых сдвигов. Вместо этого, некоторые инструкции (MOV, ADD, SUB, RSB)[4] могут быть дополнены пометкой, сдвигать ли второй операнд и если да, то на сколько и как. ASR означает *Arithmetic Shift Right*, LSR — *Logican Shift Right*.

14.2.2 Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

В режиме thumb отдельные инструкции для битовых сдвигов есть, и здесь применяется одна из них — ASRS (арифметический сдвиг вправо).

14.2.3 Неоптимизирующий Xcode (LLVM) и Keil

Неоптимизирующий LLVM не занимается генерацией подобного кода, а вместо этого просто вставляет вызов библиотечной функции *___divsi3*.

A Keil во всех случаях вставляет вызов функции *__aeabi_idivmod*.

14.3 Как это работает

Вот как деление может быть заменено на умножение и деление на числа 2^n:

$$result = \frac{input}{divisor} = \frac{input \cdot \frac{2^n}{divisor}}{2^n} = \frac{input \cdot M}{2^n}$$

Где M это *magic*-коэффициент.

Как вычислить M:

$$M = \frac{2^n}{divisor}$$

Так что эти фрагменты кода обычно имеют форму:

$$result = \frac{input \cdot M}{2^n}$$

n это произвольное число, оно может быть 32 (тогда старшая часть результата умножения берется из регистра EDX или RDX) или 31 (тогда старшая часть результата умножения дополнительно сдвигается).

n выбирается так, чтобы улучшить точность результата.

Если делать знаковое деление, знак результата умножения также добавляется к результату.

Посмотрите на разницу:

[4] Эти инструкции также называются "data processing instructions"

```
int f3_32_signed(int a)
{
        return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
        return a/3;
};
```

В беззнаковой версии функции, *magic*-коэффициент это `0xAAAAAAAB` и результат умножения делится на 2^33.

В знаковой версии функции, *magic*-коэффициент это `0x55555556` и результат умножения делится на 2^{32}. Знак результата умножения также учитывается: старшие 32 бита результата сдвигаются на 32 (таким образом, оставляя знак в самом младшем бите`EAX`). 1 прибавляется к конечному результату, если знак отрицательный.

Листинг 14.4: MSVC 2012 /Ox

```
_f3_32_unsigned PROC
        mov     eax, -1431655765                        ; aaaaaaabH
        mul     DWORD PTR _a$[esp-4] ; беззнаковое деление
        shr     edx, 1
        mov     eax, edx
        ret     0
_f3_32_unsigned ENDP

_f3_32_signed PROC
        mov     eax, 1431655766                         ; 55555556H
        imul    DWORD PTR _a$[esp-4] ; знаковое деление
        mov     eax, edx
        shr     eax, 31                                 ; 0000001fH
        add     eax, edx ; прибавить 1 если знак отрицательный
        ret     0
_f3_32_signed ENDP
```

Читайте больше об этом в [35, 10-3].

14.4 Определение делителя

14.4.1 Вариант #1

Часто, код имеет вид:

```
        mov     eax, MAGICAL CONSTANT
        imul    input value
        sar     edx, SHIFTING COEFFICIENT ; знаковое деление на 2^x при помощи арифметического ↙
    ↳ сдвига вправо
        mov     eax, edx
        shr     eax, 31
        add     eax, edx
```

Определим 32-битную *magic*-коэффициент через M, коэффициент сдвига через C и делитель через D. Делитель, который нам нужен это:

$$D = \frac{2^{32+C}}{M}$$

Например:

Листинг 14.5: Оптимизирующий MSVC 2012

```
        mov     eax, 2021161081                         ; 78787879H
        imul    DWORD PTR _a$[esp-4]
        sar     edx, 3
        mov     eax, edx
```

```
        shr     eax, 31                                 ; 0000001fH
        add     eax, edx
```

Это:

$$D = \frac{2^{32+3}}{2021161081}$$

Числа больше чем 32-битные, так что я использовал Wolfram Mathematica для удобства:

Листинг 14.6: Wolfram Mathematica

```
In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.
```

Так что искомый делитель это 17.

При делении в x64, всё то же самое, только нужно использовать 2^{64} вместо 2^{32}:

```
uint64_t f1234(uint64_t a)
{
        return a/1234;
};
```

Листинг 14.7: MSVC 2012 x64 /Ox

```
f1234   PROC
        mov     rax, 7653754429286296943                ; 6a37991a23aead6fH
        mul     rcx
        shr     rdx, 9
        mov     rax, rdx
        ret     0
f1234   ENDP
```

Листинг 14.8: Wolfram Mathematica

```
In[1]:=N[2^(64+9)/16^^6a37991a23aead6f]
Out[1]:=1234.
```

14.4.2 Вариант #2

Бывает также вариант с пропущенным арифметическим сдвигом, например:

```
                mov     eax, 55555556h ; 1431655766
                imul    ecx
                mov     eax, edx
                shr     eax, 1Fh
```

Метод определения делителя упрощается:

$$D = \frac{2^{32}}{M}$$

Для моего примера, это:

$$D = \frac{2^{32}}{1431655766}$$

Снова использую Wolfram Mathematica:

Листинг 14.9: Wolfram Mathematica

```
In[1]:=N[2^32/16^^55555556]
Out[1]:=3.
```

Искомый делитель это 3.

Глава 15

Работа с FPU

FPU[1] — блок в процессоре работающий с числами с плавающей запятой.

Раньше он назывался сопроцессором. Он немного похож на программируемый калькулятор и стоит немного в стороне от CPU.

Перед изучением FPU полезно ознакомиться с тем как работают стековые машины[2], или ознакомиться с основами языка Forth[3].

Интересен факт, что в свое время (до 80486) сопроцессор был отдельным чипом на материнской плате, и вследствие его высокой цены, он стоял не всегда. Его можно было докупить отдельно и поставить [4].

Начиная с процессора 80486 DX, FPU уже всегда входит в его состав.

Этот факт может напоминать такой рудимент, как наличие инструкции вроде `FWAIT`, которая заставляет CPU ожидать, пока FPU закончит работу. Другой рудимент, это тот факт что опкоды FPU-инструкций начинаются с т.н. "escape"-опкодов (`D8..DF`), т.е., опкоды, передающиеся в FPU.

FPU имеет стек из восьми 80-битных регистров, каждый может содержать число в формате IEEE 754[5].

В Си/Си++ имеются два типа для работы с числами с плавающей запятой, это *float* (*число одинарной точности*[6], 32 бита) [7] и *double* (*число двойной точности*[8], 64 бита).

GCC также поддерживает тип *long double* (*extended precision*[9], 80 бит), но MSVC — нет.

Не смотря на то что *float* занимает столько же места сколько *int* на 32-битной архитектуре, представление чисел, разумеется, совершенно другое.

Число с плавающей точкой состоит из знака, мантиссы[10] и экспоненты.

Функция, имеющая *float* или *double* среди аргументов, получает эти значения через стек. Если функция возвращает *float* или *double*, она оставляет значение в регистре `ST(0)` — то есть, на вершине FPU-стека.

15.1 Простой пример

Рассмотрим простой пример:

```
double f (double a, double b)
```

[1] Floating-point unit
[2] http://en.wikipedia.org/wiki/Stack_machine
[3] http://en.wikipedia.org/wiki/Forth_(programming_language)
[4] Например, Джон Кармак использовал в своей игре Doom числа с фиксированной запятой, хранящиеся в обычных 32-битных GPR (16 бит на целую часть и 16 на дробную), чтобы Doom работал на 32-битных компьютерах без FPU, т.е., 80386 и 80486 SX
[5] http://en.wikipedia.org/wiki/IEEE_754-2008
[6] http://en.wikipedia.org/wiki/Single-precision_floating-point_format
[7] Формат представления float-чисел затрагивается в разделе *Работа с типом float как со структурой* (18.6.2).
[8] http://en.wikipedia.org/wiki/Double-precision_floating-point_format
[9] http://en.wikipedia.org/wiki/Extended_precision
[10] *significand* или *fraction* в англоязычной литературе

```
{
        return a/3.14 + b*4.1;
};
```

15.1.1 x86

Компилируем в MSVC 2010:

Листинг 15.1: MSVC 2010

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f  PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _a$[ebp]

; состояние стека сейчас: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; состояние стека сейчас: ST(0) = результат деления _a на 3.13

    fld    QWORD PTR _b$[ebp]

; состояние стека сейчас: ST(0) = _b; ST(1) = результат деления _a на 3.13

    fmul   QWORD PTR __real@4010666666666666

; состояние стека сейчас: ST(0) = результат _b * 4.1; ST(1) = результат деления _a на 3.13

    faddp  ST(1), ST(0)

; состояние стека сейчас: ST(0) = результат сложения

    pop    ebp
    ret    0
_f  ENDP
```

FLD берет 8 байт из стека и загружает их в регистр ST(0), автоматически конвертируя во внутренний 80-битный формат (*extended precision*).

FDIV делит содержимое регистра ST(0) на число, лежащее по адресу __real@40091eb851eb851f — там закодировано значение 3.14. Синтаксис ассемблера не поддерживает подобные числа, так что то что мы там видим, это шестнадцатеричное представление числа *3.14* в формате IEEE 754.

После выполнения FDIV, в ST(0) остается частное[11].

Кстати, есть еще инструкция FDIVP, которая делит ST(1) на ST(0), выталкивает эти числа из стека и заталкивает результат. Если вы знаете язык Forth[12], то это как раз оно и есть — стековая машина[13].

Следующая FLD заталкивает в стек значение *b*.

После этого, в ST(1) перемещается результат деления, а в ST(0) теперь будет *b*.

Следующий FMUL умножает *b* из ST(0) на значение __real@4010666666666666 — там лежит число 4.1, и оставляет результат в ST(0).

[11] результат деления

[12] http://en.wikipedia.org/wiki/Forth_(programming_language)

[13] http://en.wikipedia.org/wiki/Stack_machine

Самая последняя инструкция FADDP складывает два значения из вершины стека, в ST(1) и затем выталкивает значение, лежащее в ST(0), таким образом результат сложения остается на вершине стека в ST(0).

Функция должна вернуть результат в ST(0), так что больше ничего здесь не производится, кроме эпилога функции.

GCC 4.4.1 (с опцией -O3) генерирует похожий код, хотя и с некоторой разницей:

Листинг 15.2: Оптимизирующий GCC 4.4.1

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; состояние стека сейчас: ST(0) = 3.13

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; состояние стека сейчас: ST(0) = результат деления

                fld     ds:dbl_8048610 ; 4.1

; состояние стека сейчас: ST(0) = 4.1, ST(1) = результат деления

                fmul    [ebp+arg_8]

; состояние стека сейчас: ST(0) = результат умножения, ST(1) = результат деления

                pop     ebp
                faddp   st(1), st

; состояние стека сейчас: ST(0) = результат сложения

                retn
f               endp
```

Разница в том, что в стек сначала заталкивается 3.14 (в ST(0)), а затем значение из arg_0 делится на то что лежит в регистре ST(0).

FDIVR означает *Reverse Divide* — делить поменяв делитель и делимое местами. Точно такой же инструкции для умножения нет, потому что она была бы бессмыслена (ведь умножение — операция коммутативная), так что остается только FMUL без соответствующей ей -R инструкции.

FADDP не только складывает два значения, но также и выталкивает из стека одно значение. После этого, в ST(0) остается только результат сложения.

Этот фрагмент кода получен при помощи IDA, которая регистр ST(0) называет для краткости просто ST.

15.1.2 ARM: Оптимизирующий Xcode (LLVM) + Режим ARM

Пока в ARM не было стандартного набора инструкций для работы с плавающей точкой, разные производители процессоров могли добавлять свои расширения для работы с ними. Позже, был принят стандарт VFP (*Vector Floating Point*).

Важное отличие от x86 в том, что там вы работаете с FPU-стеком, а здесь стека нет, здесь вы работаете просто с регистрами.

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; load a
                VMOV            D18, R2, R3 ; load b
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
```

```
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF: f
dbl_2CA0        DCFD 4.1                ; DATA XREF: f+10
```

Итак, здесь мы видим использование новых регистров, с префиксом D. Это 64-битные регистры, их 32, и их можно использовать и для чисел с плавающей точкой двойной точности (double) и для SIMD (в ARM это называется NEON).

Имеются также 32 32-битных S-регистра, они применяются для работы с числами с плавающей точкой одинарной точности (float).

Запомнить легко: D-регистры предназначены для чисел double-точности, а S-регистры — для чисел single-точности.

Обе константы (3.14 и 4.1) хранятся в памяти в формате IEEE 754.

Инструкции `VLDR` и `VMOV` , как можно догадаться, это аналоги обычных `LDR` и `MOV`, но они работают с D-регистрами. Важно отметить, что эти инструкции, как и D-регистры, предназначены не только для работы с числами с плавающей точкой, но пригодны также и для работы с SIMD (NEON), и позже это также будет видно.

Аргументы передаются в функцию обычным путем, через R-регистры, однако, каждое число, имеющее двойную точность, занимает 64 бита, так что для передачи каждого нужны два R-регистра.

‘‘`VMOV D17, R0, R1`’’ в самом начале составляет два 32-битных значения из `R0` и `R1` в одно 64-битное и сохраняет в `D17`.

‘‘`VMOV R0, R1, D16`’’ в конце это обратная процедура, то что было в `D16` остается в двух регистрах `R0` и `R1`, потому что, число с двойной точностью, занимающее 64 бита, возвращается в паре регистров `R0` и `R1`.

`VDIV`, `VMUL` и `VADD`, это, собственно, инструкции для работы с числами с плавающей точкой, вычисляющие, соответственно, частное[14], произведение[15]и сумму[16].

Код для thumb-2 такой же.

15.1.3 ARM: Оптимизирующий Keil + Режим thumb

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

[14]результат деления
[15]результат умножения
[16]результат сложения

Keil компилировал для процессора, в котором может и не быть поддержки FPU или NEON. Так что числа с двойной точностью передаются в парах обычных R-регистров, а вместо FPU-инструкций вызываются сервисные библиотечные функции `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd` , эмулирующие умножение, деление и сложение чисел с плавающей точкой. Конечно, это медленнее чем FPU-сопроцессор, но лучше, чем ничего.

Кстати, похожие библиотеки для эмуляции сопроцессорных инструкций были очень распространены в x86, когда сопроцессор был редким и дорогим, и стоял далеко не на всех компьютерах.

Эмуляция FPU-сопроцессора в ARM называется *soft float* или *armel*, а использование FPU-инструкций сопроцессора — *hard float* или *armhf*.

Ядро Linux, например, для Raspberry Pi может поставляться в двух вариантах. В случае *soft float*, аргументы будут передаваться через R-регистры, а в случае *hard float*, через D-регистры.

И это то, что помешает использовать, например, armhf-библиотеки из armel-кода или наоборот, поэтому, весь код в дистрибутиве Linux должен быть скомпилирован в соответствии с выбранным соглашением о вызовах.

15.2 Передача чисел с плавающей запятой в аргументах

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

15.2.1 x86

Посмотрим, что у нас вышло (MSVC 2010):

Листинг 15.3: MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8  ; выделить место для первой переменной
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; выделить место для второй переменной
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; "вернуть" место от одной переменной.

; в локальном стеке сейчас все еще зарезервировано 8 байт для нас.
; результат сейчас в ST(0)

    fstp   QWORD PTR [esp] ; перегрузить результат из ST(0) в локальный стек для printf()
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD и FSTP перемещают переменные из/в сегмента данных в FPU-стек. pow()[17] достает оба значения из FPU-стека и возвращает результат в ST(0). printf() берет 8 байт из стека и трактует их как переменную типа *double*.

15.2.2 ARM + Неоптимизирующий Xcode (LLVM) + Режим thumb-2

```
_main

var_C           = -0xC

                PUSH            {R7,LR}
                MOV             R7, SP
                SUB             SP, SP, #4
                VLDR            D16, =32.01
                VMOV            R0, R1, D16
                VLDR            D16, =1.54
                VMOV            R2, R3, D16
                BLX             _pow
                VMOV            D16, R0, R1
                MOV             R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
                ADD             R0, PC
                VMOV            R1, R2, D16
                BLX             _printf
                MOVS            R1, 0
                STR             R0, [SP,#0xC+var_C]
                MOV             R0, R1
                ADD             SP, SP, #4
                POP             {R7,PC}

dbl_2F90        DCFD 32.01              ; DATA XREF: _main+6
dbl_2F98        DCFD 1.54               ; DATA XREF: _main+E
```

Как я уже писал, 64-битные числа с плавающей точкой передаются в парах R-регистров. Этот код слегка избыточен (наверное, потому что не включена оптимизация), ведь, можно было бы загружать значения напрямую в R-регистры минуя загрузку в D-регистры.

Итак, видно, что функция _pow получает первый аргумент в R0 и R1, а второй в R2 и R3. Функция оставляет результат в R0 и R1. Результат работы _pow перекладывается в D16, затем в пару R1 и R2, откуда printf() будет читать это число.

15.2.3 ARM + Неоптимизирующий Keil + Режим ARM

```
_main
                STMFD   SP!, {R4-R6,LR}
                LDR     R2, =0xA3D70A4  ; y
                LDR     R3, =0x3FF8A3D7
                LDR     R0, =0xAE147AE1 ; x
                LDR     R1, =0x40400147
                BL      pow
                MOV     R4, R0
                MOV     R2, R4
                MOV     R3, R1
                ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
                BL      __2printf
                MOV     R0, #0
                LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF: _main+8
; double x
```

[17] стандартная функция Си, возводящая число в степень

```
x               DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528       DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF: _main+24
```

Здесь не используются D-регистры, используются только пары R-регистров.

15.3 Пример с сравнением

Попробуем теперь вот это:

```
double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};
```

15.3.1 x86

Несмотря на кажущуюся простоту этой функции, понять, как она работает будет чуть сложнее.

Вот что выдал MSVC 2010:

Листинг 15.4: MSVC 2010

```
PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8            ; size = 8
_b$ = 16           ; size = 8
_d_max     PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _b$[ebp]

; состояние стека сейчас: ST(0) = _b
; сравниваем _b (в ST(0)) и _a, затем выталкиваем значение из стека

    fcomp  QWORD PTR _a$[ebp]

; стек теперь пустой

    fnstsw ax
    test   ah, 5
    jp     SHORT $LN1@d_max

; мы здесь если if a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
    fld    QWORD PTR _b$[ebp]
$LN2@d_max:
    pop    ebp
    ret    0
_d_max     ENDP
```

Итак, `FLD` загружает `_b` в регистр `ST(0)`.

`FCOMP` сравнивает содержимое `ST(0)` с тем что лежит в `_a` и выставляет биты `C3`/`C2`/`C0` в регистре статуса FPU. Это 16-битный регистр отражающий текущее состояние FPU.

Итак, биты C3/C2/C0 выставлены, но, к сожалению, у процессоров до Intel P6 [18] нет инструкций условного перехода, проверяющих эти биты. Возможно, так сложилось исторически (вспомните о том что FPU когда-то был вообще отдельным чипом). А у Intel P6 появились инструкции FCOMI/FCOMIP/FUCOMI/FUCOMIP — делающие тоже самое, только напрямую модифицирующие флаги ZF/PF/CF.

После этого, инструкция FCOMP выдергивает одно значение из стека. Это отличает её от FCOM, которая просто сравнивает значения, оставляя стек в таком же состоянии.

FNSTSW копирует содержимое регистра статуса в AX. Биты C3/C2/C0 занимают позиции, соответственно, 14, 10, 8, в этих позициях они и остаются в регистре AX, и все они расположены в старшей части регистра — AH.

- Если b>a в нашем случае, то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если a>b, то биты будут выставлены: 0, 0, 1.
- Если a=b, то биты будут выставлены так: 1, 0, 0.

После исполнения test ah, 5, бит C3 и C1 сбросится в ноль, на позициях 0 и 2 (внутри регистра AH) останутся соответственно C0 и C2.

Теперь немного о *parity flag*[19]. Еще один замечательный рудимент:

> One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they can not be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.[20]

Этот флаг выставляется в 1 если количество единиц в последнем результате — четно. И в 0 если — нечетно.

Таким образом, что мы имеем, флаг PF будет выставлен в 1, если C0 и C2 оба 1 или оба 0. И тогда сработает последующий JP (*jump if PF==1*). Если мы вернемся чуть назад и посмотрим значения C3/C2/C0 для разных вариантов, то увидим, что условный переход JP сработает в двух случаях: если b>a или если a==b (ведь бит C3 уже *вылетел* после исполнения test ah, 5).

Дальше все просто. Если условный переход сработал, то FLD загрузит значение _b в ST(0), а если не сработал, то загрузится _a и произойдет выход из функции.

Но это еще не все!

15.3.2 А теперь скомпилируем все это в MSVC 2010 с опцией /Ox

Листинг 15.5: Оптимизирующий MSVC 2010

```
_a$ = 8             ; size = 8
_b$ = 16            ; size = 8
_d_max      PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]

; состояне стека сейчас: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; сравнить _a и ST(1) = (_b)
    fnstsw  ax
    test    ah, 65                  ; 00000041H
    jne     SHORT $LN5@d_max

; копировать содержимое ST(0) в ST(1) и вытолкнуть значение из стека,
; оставив _a на вершине
    fstp    ST(1)

; состояние стека сейчас: ST(0) = _a
```

[18] Intel P6 это Pentium Pro, Pentium II, и далее

[19] флаг четности

```
    ret     0
$LN5@d_max:

; копировать содержимое ST(0) в ST(0) и вытолкнуть значение из стека,
; оставив _b на вершине
   fstp    ST(0)

; состояние стека сейчас: ST(0) = _b

    ret     0
_d_max      ENDP
```

FCOM отличается от FCOMP тем что просто сравнивает значения и оставляет стек в том же состоянии. В отличие от предыдущего примера, операнды здесь в другом порядке. Поэтому и результат сравнения в C3/C2/C0 будет другим чем раньше:

- Если a>b в нашем случае, то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если b>a, то биты будут выставлены: 0, 0, 1.
- Если a=b, то биты будут выставлены так: 1, 0, 0.

Инструкция test ah, 65 как бы оставляет только два бита — C3 и C0. Они оба будут нулями, если a>b: в таком случае переход JNE не сработает. Далее имеется инструкция FSTP ST(1) — эта инструкция копирует значение ST(0) в указанный операнд и выдергивает одно значение из стека. В данном случае, она копирует ST(0) (где сейчас лежит _a) в ST(1). После этого на вершине стека два раза лежат _a. Затем одно значение выдергивается. После этого в ST(0) остается _a и функция завершается.

Условный переход JNE сработает в двух других случаях: если b>a или a==b. ST(0) скопируется в ST(0), что как бы холостая операция, затем одно значение из стека вылетит и на вершине стека останется то что до этого лежало в ST(1) (то есть, _b). И функция завершится. Эта инструкция используется здесь видимо потому что в FPU нет инструкции которая просто выдергивает значение из стека и больше ничего.

Но и это еще не все.

15.3.3 GCC 4.4.1

Листинг 15.6: GCC 4.4.1

```
d_max proc near

b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
    mov     ebp, esp
    sub     esp, 10h

; переложим a и b в локальный стек:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; загружаем a и b в стек FPU
```

```
    fld     [ebp+a]
    fld     [ebp+b]

; текущее состояние стека: ST(0) - b; ST(1) - a

    fxch    st(1) ; эта инструкция меняет ST(1) и ST(0) местами

; текущее состояние стека: ST(0) - a; ST(1) - b

    fucompp     ; сравнить a и b и выдернуть из стека два значения, т.е., a и b
    fnstsw  ax  ; записать статус FPU в AX
    sahf        ; загрузить состояние флагов SF, ZF, AF, PF, и CF из AH
    setnbe  al  ; записать единицу в AL если CF=0 и ZF=0
    test    al, al              ; AL==0 ?
    jz      short loc_8048453 ; да
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

FUCOMPP — это почти то же что и FCOM, только выкидывает из стека оба значения после сравнения, а также несколько иначе реагирует на "не-числа".

Немного о *не-числах*:

FPU умеет работать со специальными переменными, которые числами не являются и называются "не числа" или NaN[21]. Это бесконечность, результат деления на ноль, и так далее. Нечисла бывают "тихие" и "сигнализирующие". С первыми можно продолжать работать и далее, а вот если вы попытаетесь совершить какую-то операцию с сигнализирующим нечислом, то сработает исключение.

Так вот, FCOM вызовет исключение если любой из операндов — какое-либо нечисло. FUCOM же вызовет исключение только если один из операндов именно "сигнализирующее нечисло".

Далее мы видим SAHF — это довольно редкая инструкция в коде не использующим FPU. 8 бит из AH перекладываются в младшие 8 бит регистра статуса процессора в таком порядке: SF:ZF:-:AF:-:PF:-:CF <- AH.

Вспомним, что FNSTSW перегружает интересующие нас биты C3/C2/C0 в AH, и соответственно они будут в позициях 6, 2, 0 в регистре AH.

Иными словами, пара инструкций `fnstsw ax / sahf` перекладывает биты C3/C2/C0 в флаги ZF, PF, CF.

Теперь снова вспомним, какие значения бит C3/C2/C0 будут при каких результатах сравнения:

- Если a больше b в нашем случае, то биты C3/C2/C0 должны быть выставлены так: 0, 0, 0.
- Если a меньше b, то биты будут выставлены: 0, 0, 1.
- Если a=b, то биты будут выставлены так: 1, 0, 0.

Иными словами, после инструкций FUCOMPP/FNSTSW/SAHF, мы получим такое состояние флагов:

- Если a>b в нашем случае, то флаги будут выставлены так: ZF=0, PF=0, CF=0.
- Если a<b, то флаги будут выставлены: ZF=0, PF=0, CF=1.
- Если a=b, то флаги будут выставлены так: ZF=1, PF=0, CF=0.

Инструкция SETNBE выставит в AL единицу или ноль, в зависимости от флагов и условий. Это почти аналог JNBE, за тем лишь исключением, что SETcc[22] выставляет 1 или 0 в AL, а Jcc делает переход или нет. SETNBE запишет 1 если только CF=0 и ZF=0. Если это не так, то запишет 0 в AL.

CF будет 0 и ZF будет 0 одновременно только в одном случае: если a>b.

Тогда в AL будет записана единица, последующий условный переход JZ взят не будет, и функция вернет _a. В остальных случаях, функция вернет _b.

Но и это еще не конец.

[21] http://ru.wikipedia.org/wiki/NaN

[22] *cc* это *condition code*

15.3.4 GCC 4.4.1 с оптимизацией -O3

Листинг 15.7: Оптимизирующий GCC 4.4.1

```
                public d_max
d_max           proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; состояние стека сейчас: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; состояние стека сейчас: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; сравнить _a и _b
                fnstsw  ax
                sahf
                ja      short loc_8048448
; записать ST(0) в ST(0) (холостая операция), выкинуть значение лежащее на вершине стека, ↙
    ↳ оставить _b
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; записать _a в ST(0), выкинуть значение лежащее на вершине стека, оставить _a на вершине стека
                fstp    st(1)

loc_804844A:
                pop     ebp
                retn
d_max           endp
```

Почти все что здесь есть уже описано мною, кроме одного: использование JA после SAHF. Действительно, инструкции условных переходов "больше", "меньше", "равно" для сравнения беззнаковых чисел (JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) проверяют только флаги CF и ZF. И биты C3/C2/C0 после сравнения перекладываются в эти флаги аккурат так, чтобы перечисленные инструкции переходов могли работать. JA сработает если CF и ZF обнулены.

Таким образом, перечисленные инструкции условного перехода можно использовать после инструкций FNSTSW/SAHF.

Вполне возможно, что биты статуса FPU C3/C2/C0 преднамеренно были размещены таким образом, чтобы переноситься на базовые флаги процессора без перестановок.

15.3.5 ARM + Оптимизирующий Xcode (LLVM) + Режим ARM

Листинг 15.8: Оптимизирующий Xcode (LLVM) + Режим ARM

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copy b to D16
VMOV            R0, R1, D16
BX              LR
```

Очень простой случай. Входные величины помещаются в D17 и D16 и сравниваются при помощи инструкции VCMPE. Как и в сопроцессорах x86, сопроцессор в ARM имеет свой собственный регистр статуса и флагов, (FPSCR), потому как есть необходимость хранить специфичные для его работы флаги.

И так же, как и в x86, в ARM нет инструкций условного перехода, проверяющих биты в регистре статуса сопроцессора, так что имеется инструкция VMRS , копирующая 4 бита (N, Z, C, V) из статуса сопроцессора в биты *общего* статуса (регистр APSR).

VMOVGT это аналог MOVGT, инструкция, сработающая если при сравнении один операнд был больше чем второй (*GT — Greater Than*).

Если она сработает, в D16 запишется значение *b*, лежащее в тот момент в D17.

А если не сработает, то в D16 останется лежать значение *a*.

Предпоследняя инструкция VMOV подготовит то что было в D16 для возврата через пару регистров R0 и R1.

15.3.6 ARM + Оптимизирующий Xcode (LLVM) + Режим thumb-2

Листинг 15.9: Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Почти то же самое что и в предыдущем примере, за парой отличий. Дело в том, многие инструкции в режиме ARM можно дополнять условием, которое если справедливо, то инструкция выполнится.

Но в режиме thumb такого нет. В 16-битных инструкций просто нет места для лишних 4 битов, при помощи которых можно было бы закодировать условие выполнения.

Поэтому в thumb-2 добавили возможность дополнять thumb-инструкции условиями.

Здесь, в листинге сгенерированном при помощи IDA, мы видим инструкцию VMOVGT, такую же как и в предыдущем примере.

Но в реальности, там закодирована обычная инструкция VMOV, просто IDA добавила суффикс -GT к ней, потому что перед этой инструкцией стоит ‘‘IT GT’’.

Инструкция IT определяет так называемый *if-then block*. После этой инструкции, можно указывать до четырех инструкций, к которым будет добавлен суффикс условия. В нашем примере, ‘‘IT GT’’ означает, что следующая за ней инструкция будет исполнена, если условие *GT* (*Greater Than*) справедливо.

Теперь более сложный пример, кстати, из “Angry Birds” (для iOS):

Листинг 15.10: Angry Birds Classic

```
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
```

ITE означает *if-then-else* и кодирует суффиксы для двух следующих за ней инструкций. Первая из них исполнится, если условие, закодированное в ITE (*NE, not equal*) будет в тот момент справедливо, а вторая — если это условие не сработает. (Обратное условие от NE это EQ (*equal*)).

Еще чуть сложнее, и снова этот фрагмент из “Angry Birds”:

Листинг 15.11: Angry Birds Classic

```
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
```

4 символа “T” в инструкции означают что 4 следующие инструкции будут исполнены если условие соблюдается. Поэтому IDA добавила ко всем четырем инструкциям суффикс -EQ.

А если бы здесь было, например, ITEEE EQ (*if-then-else-else-else*), тогда суффиксы для следующих четырех инструкций были бы расставлены так:

```
-EQ
-NE
-NE
-NE
```

Еще фрагмент из “Angry Birds”:

Листинг 15.12: Angry Birds Classic

```
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
```

ITTE (*if-then-then-else*) означает что первая и вторая инструкции исполнятся, если условие LE (*Less or Equal*) справедливо, а третья — если справедливо обратное условие (GT — *Greater Than*).

Компиляторы способны генерировать далеко не все варианты. Например, в вышеупомянутой игре “Angry Birds” (версия *classic* для iOS) встречаются только такие варианты инструкции IT: IT, ITE, ITT, ITTE, ITTT, ITTTT. Как я это узнал? В IDA можно сгенерировать листинг, так я и сделал, только в опциях я установил так чтобы показывались 4 байта для каждого опкода. Затем, зная, что старшая часть 16-битного опкода IT это 0xBF, я сделал при помощи grep это:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

Кстати, если писать на ассемблере для режима thumb-2 вручную, и дополнять инструкции суффиксами условия, то ассемблер автоматически будет добавлять инструкцию IT с соответствующими флагами, там, где надо.

15.3.7 ARM + Неоптимизирующий Xcode (LLVM) + Режим ARM

Листинг 15.13: Неоптимизирующий Xcode (LLVM) + Режим ARM

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Почти то же самое что мы уже видели, но много избыточного кода из-за хранения a и b, а также выходного значения, в локальном стеке.

15.3.8 ARM + Оптимизирующий Keil + Режим thumb

Листинг 15.14: Оптимизирующий Keil + Режим thumb

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
                MOVS    R5, R3
                MOVS    R6, R0
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil не генерирует специальную инструкцию для сравнения чисел с плавающей запятой, потому что не рассчитывает на то что она будет поддерживаться, а простым сравнением побитово здесь не обойтись. Для сравнения вызывается библиотечная функция `__aeabi_cdrcmple`. N.B. Результат сравнения эта функция оставляет в флагах, чтобы следующая за вызовом инструкция `BCS` (*Carry set - Greater than or equal*) могла работать без дополнительного кода.

15.4 x64

О том как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: 24.

Глава 16

Массивы

Массив, это просто набор переменных в памяти, обязательно лежащих рядом, и обязательно одного типа [1].

16.1 Простой пример

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

16.1.1 x86

Компилируем:

Листинг 16.1: MSVC

```
_TEXT    SEGMENT
_i$ = -84                       ; size = 4
_a$ = -80                       ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84        ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp    DWORD PTR _i$[ebp], 20   ; 00000014H
    jge    SHORT $LN4@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
```

[1] AKA[2] “гомогенный контейнер”

```
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    mov    edx, DWORD PTR _a$[ebp+ecx*4]
    push   edx
    mov    eax, DWORD PTR _i$[ebp]
    push   eax
    push   OFFSET $SG2463
    call   _printf
    add    esp, 12        ; 0000000cH
    jmp    SHORT $LN2@main
$LN1@main:
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Однако, ничего особенного, просто два цикла, один заполняет цикл, второй печатает его содержимое. Команда `shl ecx, 1` используется для умножения `ECX` на 2, об этом ниже (17.3.1).

Под массив выделено в стеке 80 байт, это 20 элементов по 4 байта.

То, что делает GCC 4.4.1:

Листинг 16.2: GCC 4.4.1

```
                public main
main            proc near               ; DATA XREF: _start+17

var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1          ; i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
```

```
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

Кстати, переменная *a* в нашем примере имеет тип *int** (то есть, указатель на *int*) — вы можете попробовать передать в другую функцию указатель на массив, но точнее было бы сказать, что передается указатель на первый элемент массива (а адреса остальных элементов массива можно вычислить очевидным образом). Если индексировать этот указатель как *a[idx]*, *idx* просто прибавляется к указателю и возвращается элемент, расположенный там, куда ссылается вычисленный указатель.

Вот любопытный пример: строка символов вроде *"string"* это массив из символов, и она имеет тип *const char**.К этому указателю также можно применять индекс. И поэтому можно написать даже так: `"string"[i]` — это совершенно легальное выражение в Си/Си++!

16.1.2 ARM + Неоптимизирующий Keil + Режим ARM

```
            EXPORT _main
_main
            STMFD   SP!, {R4,LR}
            SUB     SP, SP, #0x50       ; выделить место для 20-и переменных типа int

; первый цикл

            MOV     R4, #0              ; i
            B       loc_4A0
loc_494
            MOV     R0, R4,LSL#1        ; R0=R4*2
            STR     R0, [SP,R4,LSL#2]   ; сохранить R0 в SP+R4<<2 (то же что и SP+R4*4)
            ADD     R4, R4, #1          ; i=i+1

loc_4A0
            CMP     R4, #20             ; i<20?
            BLT     loc_494             ; да, запустить тело цикла снова

; второй цикл

            MOV     R4, #0              ; i
            B       loc_4C4
loc_4B0
            LDR     R2, [SP,R4,LSL#2]   ; (второй аргумент printf) R2=*(SP+R4<<4) (то же что и ↙
    ↳ *(SP+R4*4))
            MOV     R1, R4              ; (первый аргумент printf) R1=i
            ADR     R0, aADD            ; "a[%d]=%d\n"
            BL      __2printf
```

```
            ADD     R4, R4, #1          ; i=i+1

loc_4C4
            CMP     R4, #20             ; i<20?
            BLT     loc_4B0             ; да, запустить тело цикла снова
            MOV     R0, #0              ; значение для возврата
            ADD     SP, SP, #0x50       ; освободить место в стеке, выделенное для 20 переменных
            LDMFD   SP!, {R4,PC}
```

Тип *int* требует 32 бита для хранения, или 4 байта, так что для хранения 20 переменных типа *int*, нужно 80 (0x50) байт, поэтому инструкция ‘‘SUB SP, SP, #0x50’’ в эпилоге функции выделяет в локальном стеке под массив именно столько места.

И в первом и во втором цикле, итератор цикла i будет постоянно находится в регистре R4.

Число, которое нужно записать в массив, вычисляется так $i * 2$, и это эквивалентно сдвигу на 1 бит влево, инструкция ‘‘MOV R0, R4,LSL#1’’ делает это.

‘‘STR R0, [SP,R4,LSL#2]’’ записывает содержимое R0 в массив. Указатель на элемент массива вычисляется так: SP указывает на начало массива, R4 это i. Так что сдвигаем i на 2 бита влево, что эквивалентно умножению на 4 (ведь каждый элемент массива занимает 4 байта) и прибавляем это к адресу начала массива.

Во втором цикле используется обратная инструкция ‘‘LDR R2, [SP,R4,LSL#2]’’, она загружает из массива нужное значение, и указатель на него вычисляется точно так же.

16.1.3 ARM + Оптимизирующий Keil + Режим thumb

```
_main
            PUSH    {R4,R5,LR}
; выделить место для 20 переменных типа int + еще одной переменной
            SUB     SP, SP, #0x54

; первый цикл

            MOVS    R0, #0          ; i
            MOV     R5, SP          ; указатель на первый элемент массива

loc_1CE
            LSLS    R1, R0, #1      ; R1=i<<1 (то же что и i*2)
            LSLS    R2, R0, #2      ; R2=i<<2 (то же что и i*4)
            ADDS    R0, R0, #1      ; i=i+1
            CMP     R0, #20         ; i<20?
            STR     R1, [R5,R2]     ; сохранить R1 в *(R5+R2) (то же что и R5+i*4)
            BLT     loc_1CE         ; да, i<20, запустить тело цикла снова

; второй цикл

            MOVS    R4, #0          ; i=0
loc_1DC
            LSLS    R0, R4, #2      ; R0=i<<2 (то же что и i*4)
            LDR     R2, [R5,R0]     ; загрузить из *(R5+R0) (то же что и R5+i*4)
            MOVS    R1, R4
            ADR     R0, aADD        ; "a[%d]=%d\n"
            BL      __2printf
            ADDS    R4, R4, #1      ; i=i+1
            CMP     R4, #20         ; i<20?
            BLT     loc_1DC         ; да, i<20, запустить тело цикла снова
            MOVS    R0, #0          ; значение для возврата
; освободить место в стеке, выделенное для 20-и переменных типа int и еще одной переменной
            ADD     SP, SP, #0x54
            POP     {R4,R5,PC}
```

Код для thumb очень похожий. В thumb имеются отдельные инструкции для битовых сдвигов (как LSLS), вычисляющие и число для записи в массив и адрес каждого элемента массива.

Компилятор почему-то выделил в локальном стеке немного больше места, однако последние 4 байта не используются.

16.2 Переполнение буфера

Итак, индексация массива — это просто *массив[индекс]*. Если вы присмотритесь к коду, в цикле печати значений массива через `printf()` вы не увидите проверок индекса, *меньше ли он двадцати?* А что будет если он будет больше двадцати? Эта одна из особенностей Си/Си++, за которую их, собственно, и ругают.

Вот код который и компилируется и работает:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[100]=%d\n", a[100]);

        return 0;
};
```

Вот в это (MSVC 2010):

```
_TEXT    SEGMENT
_i$ = -84                       ; size = 4
_a$ = -80                       ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84            ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20 ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main:
    mov    eax, DWORD PTR _a$[ebp+400]
    push   eax
    push   OFFSET $SG2460
    call   _printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

У меня оно при запуске выдало вот это:

```
a[100]=760826203
```

Это просто *что-то*, что волею случая лежало в стеке рядом с массивом, через 400 байт от его первого элемента.

Действительно, а как могло бы быть иначе? Компилятор мог бы встроить какой-то код, каждый раз проверяющий индекс на соответствие пределам массива, как в языках программирования более высокого уровня[3], что делало бы запускаемый код медленнее.

Итак, мы прочитали какое-то число из стека явно *нелегально*, а что если мы запишем?

Вот что мы пишем:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

И вот что имеем на ассемблере:

```
_TEXT    SEGMENT
_i$ = -84                ; size = 4
_a$ = -80                ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84                     ; 00000054H
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30           ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]       ; явный промах компилятора. эта инструкция лишняя.
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; а здесь в качестве второго операнда подошел бы ECX.
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Запускаете скомпилированную программу, и она падает. Немудрено. Но давайте теперь узнаем, где именно.

Отладчик я уже давно не использую, так как надоело для всяких мелких задач вроде подсмотреть состояние регистров, запускать что-то, двигать мышью, и т.д. Поэтому я написал очень минималистическую утилиту для себя, tracer, коей обхожусь.

Помимо всего прочего, я могу использовать мою утилиту просто чтобы посмотреть где и какое исключение произошло. Итак, пробую:

```
generic tracer 0.4 (WIN32), http://conus.info/gt
```

[3]Java, Python, и т.д.

```
New process: C:\PRJ\...\1.exe, PID=7988
EXCEPTION_ACCESS_VIOLATION: 0x15 (<symbol (0x15) is in unknown module>), ExceptionInformation↙
    ↳ [0]=8
EAX=0x00000000 EBX=0x7EFDE000 ECX=0x0000001D EDX=0x0000001D
ESI=0x00000000 EDI=0x00000000 EBP=0x00000014 ESP=0x0018FF48
EIP=0x00000015
FLAGS=PF ZF IF RF
PID=7988|Process exit, return code -1073740791
```

Итак, следите внимательно за регистрами.

Исключение произошло по адресу 0x15. Это явно нелегальный адрес для кода — по крайней мере, win32-кода! Мы там как-то очутились, причем, сами того не хотели. Интересен также тот факт, что в EBP хранится 0x14, а в ECX и EDX — 0x1D.

И еще немного изучим разметку стека.

После того как управление передалось в main(), в стек было сохранено значение EBP. Затем, для массива + переменной i было выделено 84 байта. Это (20+1)*sizeof(int). ESP сейчас указывает на переменную _i в локальном стеке и при исполнении следующего PUSH что-либо, *что-либо* появится рядом с _i.

Вот так выглядит разметка стека пока управление находится внутри main():

ESP	4 байта для i
ESP+4	80 байт для массива a[20]
ESP+84	сохраненное значение EBP
ESP+88	адрес возврата

Команда a[19]=чего_нибудь записывает последний *int* в пределах массива (пока что в пределах!)

Команда a[20]=чего_нибудь записывает *чего_нибудь* на место где сохранено значение EBP.

Обратите внимание на состояние регистров на момент падения процесса. В нашем случае, в 20-й элемент записалось значение 20. И вот все дело в том, что заканчиваясь, эпилог функции восстанавливал значение EBP. (20 в десятичной системе это как раз 0x14 в шестнадцатеричной). Далее выполнилась инструкция RET, которая на самом деле эквивалентна POP EIP.

Инструкция RET вытащила из стека адрес возврата (это адрес где-то внутри CRT), которая вызвала main()), а там было записано 21 в десятичной системе, то есть 0x15 в шестнадцатеричной. И вот процессор оказался по адресу 0x15, но исполняемого кода там нет, так что случилось исключение.

Добро пожаловать! Это называется *buffer overflow*[4].

Замените массив *int* на строку (массив *char*), нарочно создайте слишком длинную строку, просуньте её в ту программу, в ту функцию, которая не проверяя длину строки скопирует её в слишком короткий буфер, и вы сможете указать программе, по какому именно адресу перейти. Не все так просто в реальности, конечно, но началось все с этого [5].

Попробуем то же самое в GCC 4.4.1. У нас выходит такое:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 60h
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
loc_80483D1:
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
```

[4] http://en.wikipedia.org/wiki/Stack_buffer_overflow

[5] Классическая статья об этом: [22]

```
                leave
                retn
main            endp
```

Запуск этого в Linux выдаст: `Segmentation fault`.

Если запустить полученное в отладчике GDB, получим:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388       -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0       0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Значения регистров немного другие чем в примере win32, это потому что разметка стека чуть другая.

16.3 Защита от переполнения буфера

В наше время пытаются бороться с этой напастью, не взирая на халатность программистов на Си/Си++. В MSVC есть опции вроде[6]:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Один из методов, это в прологе функции вставлять в область локальных переменных некоторое случайное значение и в эпилоге функции, перед выходом, это число проверять. И если проверка не прошла, то не выполнять инструкцию `RET` а остановиться (или зависнуть). Процесс зависнет, но это лучше, чем удаленная атака на ваш хост.

Это случайное значение иногда называют “канарейкой” [7], по аналогии с шахтной канарейкой[8], их использовали шахтеры в свое время, чтобы определять, есть ли в шахте опасный газ. Канарейки очень к нему чувствительны и либо проявляли сильное беспокойство, либо гибли от газа.

Если скомпилировать наш простейший пример работы с массивом (16.1) в MSVC с опцией RTC1 или RTCs, в конце функции будет вызов функции `@_RTC_CheckStackVars@8`, проверяющей корректность “канарейки”.

Посмотрим, как дела обстоят в GCC. Возьмем пример из секции про `alloca()` (4.2.4):

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>
```

[6] Wikipedia: описания защит, которые компилятор может вставлять в код: http://en.wikipedia.org/wiki/Buffer_overflow_protection

[7] “canary” в англоязычной литературе

[8] http://miningwiki.ru/wiki/%D0%9A%D0%B0%D0%BD%D0%B0%D1%80%D0%B5%D0%B9%D0%BA%D0%B0_%D0%B2_%D1%88%D0%B0%D1%85%D1%82%D0%B5

```
void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

По умолчанию, без дополнительных ключей, GCC 4.7.3 вставит в код проверку “канарейки”:

Листинг 16.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0  ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs:20                 ; canary
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20                 ; canary
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5:
        call    __stack_chk_fail
```

Случайное значение находится в `gs:20`. Оно записывается в стек, затем, в конце функции, значение в стеке сравнивается с корректной “канарейкой” в `gs:20`. Если значения не равны, будет вызвана функция `__stack_chk_fail` и в консоли мы увидим что-то вроде такого (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
======= Memory map: ========
```

```
08048000-08049000 r-xp 00000000 08:01 2097586    /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586    /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586    /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0          [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602    /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781    /lib/i386-linux-gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0          [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794    /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

gs — это так называемый сегментный регистр, эти регистры широко использовались во времена MS-DOS и DOS-экстендеров. Сейчас их функция немного изменилась. Если говорить коротко, в Linux `gs` всегда указывает на TLS (45) — там находится различная информация, специфичная для выполняющегося треда (кстати, в win32 эту же роль играет сегментный регистр `fs`, он всегда указывает на TIB[9] [10]).

Больше информации можно почерпнуть из исходных кодов Linux (по крайней мере, в версии 3.11): в файле *arch/x86/include/asm/stackprotector.h* в комментариях описывается эта переменная.

16.3.1 Оптимизирующий Xcode (LLVM) + Режим thumb-2

Возвращаясь к нашему простому примеру (16.1), можно посмотреть, как LLVM добавит проверку “канарейки”:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH     {R4-R7,LR}
    ADD      R7, SP, #0xC
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
```

[9] Thread Information Block

[10] https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

```
    MOVW    R0, #aObjc_methtype ; "objc_methtype"
    MOVS    R2, #0
    MOVT.W  R0, #0
    MOVS    R5, #0
    ADD     R0, PC
    LDR.W   R8, [R0]
    LDR.W   R0, [R8]
    STR     R0, [SP,#0x64+canary]
    MOVS    R0, #2
    STR     R2, [SP,#0x64+var_64]
    STR     R0, [SP,#0x64+var_60]
    MOVS    R0, #4
    STR     R0, [SP,#0x64+var_5C]
    MOVS    R0, #6
    STR     R0, [SP,#0x64+var_58]
    MOVS    R0, #8
    STR     R0, [SP,#0x64+var_54]
    MOVS    R0, #0xA
    STR     R0, [SP,#0x64+var_50]
    MOVS    R0, #0xC
    STR     R0, [SP,#0x64+var_4C]
    MOVS    R0, #0xE
    STR     R0, [SP,#0x64+var_48]
    MOVS    R0, #0x10
    STR     R0, [SP,#0x64+var_44]
    MOVS    R0, #0x12
    STR     R0, [SP,#0x64+var_40]
    MOVS    R0, #0x14
    STR     R0, [SP,#0x64+var_3C]
    MOVS    R0, #0x16
    STR     R0, [SP,#0x64+var_38]
    MOVS    R0, #0x18
    STR     R0, [SP,#0x64+var_34]
    MOVS    R0, #0x1A
    STR     R0, [SP,#0x64+var_30]
    MOVS    R0, #0x1C
    STR     R0, [SP,#0x64+var_2C]
    MOVS    R0, #0x1E
    STR     R0, [SP,#0x64+var_28]
    MOVS    R0, #0x20
    STR     R0, [SP,#0x64+var_24]
    MOVS    R0, #0x22
    STR     R0, [SP,#0x64+var_20]
    MOVS    R0, #0x24
    STR     R0, [SP,#0x64+var_1C]
    MOVS    R0, #0x26
    STR     R0, [SP,#0x64+var_18]
    MOV     R4, 0xFDA ; "a[%d]=%d\n"
    MOV     R0, SP
    ADDS    R6, R0, #4
    ADD     R4, PC
    B       loc_2F1C

; начало второго цикла

loc_2F14
    ADDS    R0, R5, #1
    LDR.W   R2, [R6,R5,LSL#2]
    MOV     R5, R0

loc_2F1C
```

```
    MOV     R0, R4
    MOV     R1, R5
    BLX     _printf
    CMP     R5, #0x13
    BNE     loc_2F14
    LDR.W   R0, [R8]
    LDR     R1, [SP,#0x64+canary]
    CMP     R0, R1
    ITTTT EQ            ; канарейка все еще верна?
    MOVEQ   R0, #0
    ADDEQ   SP, SP, #0x54
    LDREQ.W R8, [SP+0x64+var_64],#4
    POPEQ   {R4-R7,PC}
    BLX     ___stack_chk_fail
```

Во-первых, как видно, LLVM "развернул" цикл и все значения записываются в массив по одному, уже вычисленные, потому что LLVM посчитал что так будет быстрее. Кстати, инструкции режима ARM позволяют сделать это еще быстрее и это может быть вашим домашним заданием.

В конце функции мы видим сравнение "канареек" — той что лежит в локальном стеке и корректной, на которую ссылается регистр R8. Если они равны, срабатывает блок из четырех инструкций при помощи ''ITTTT EQ'', это запись 0 в R0, эпилог функции и выход из нее. Если "канарейки" не равны, блок не срабатывает и происходит переход на функцию ___stack_chk_fail, которая, вероятно, остановит работу программы.

16.4 Еще немного о массивах

Теперь понятно, почему нельзя написать в исходном коде на Си/Си++ что-то вроде [11]:

```
void f(int size)
{
    int a[size];
...
};
```

Все просто потому, чтобы выделять место под массив в локальном стеке или же сегменте данных (если массив глобальный), компилятору нужно знать его размер, чего он, на стадии компиляции, разумеется, знать не может.

Если вам нужен массив произвольной длины, то выделите столько, сколько нужно, через malloc(), затем обращайтесь к выделенному блоку байт как к массиву того типа, который вам нужен. Либо используйте возможность стандарта C99 [15, 6.7.5/2], но внутри это будет похоже на alloca() (4.2.4)

16.5 Многомерные массивы

Внутри, многомерный массив выглядит так же, как и линейный.

Ведь память компьютера линейная, это одномерный массив. Но для удобства, этот одномерный массив легко представить как многомерный.

К примеру, элементы массива $a[3][4]$ будут так расположены в одномерном массиве из 12-и ячеек:

[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][1]
[1][2]
[1][3]
[2][0]
[2][1]
[2][2]
[2][3]

[11] Впрочем, по стандарту C99 это возможно [15, 6.7.5/2]: GCC может это сделать выделяя место под массив динамически в стеке (как alloca() (4.2.4))

Вот как можно представить двухмерный массив с порядковыми номерами элементов в одномерном линейном массиве в памяти:

0	1	2	3
4	5	6	7
8	9	10	11

То есть, чтобы адресовать нужный элемент, в начале умножаем первый индекс на 4 (ширину матрицы), затем прибавляем второй индекс. Это называется *row-major order*, и такой способ представления массивов и матриц используется по крайней мере в Си/Си++, Python. Термин row-major order означает по-русски примерно следующее: "в начале записываем элементы первой строки, затем второй ... и элементы последней строки в самом конце".

Другой способ представления называется *column-major order* (индексы массива используются в обратном порядке) и это используется по крайней мере в FORTRAN, MATLAB, R. Термин column-major order означает по-русски следующее: "в начале записываем элементы первого столбца, затем второго ... и элементы последнего столбца в самом конце".

То же самое и для многомерных массивов.

Попробуем:

Листинг 16.4: простой пример

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

16.5.1 x86

В итоге (MSVC 2010):

Листинг 16.5: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_z$ = 16                ; size = 4
_value$ = 20            ; size = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=value
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

В принципе, ничего удивительного. В `insert()` для вычисления адреса нужного элемента массива, три входных аргумента перемножаются по формуле $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, чтобы представить массив

трехмерным. Не забывайте также, что тип *int* 32-битный (4 байта), поэтому все коэффициенты нужно умножить на 4.

Листинг 16.6: GCC 4.4.1

```
                public insert
insert          proc near

x               = dword ptr  8
y               = dword ptr  0Ch
z               = dword ptr  10h
value           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+x]
                mov     eax, [ebp+y]
                mov     ecx, [ebp+z]
                lea     edx, [eax+eax]              ; edx=y*2
                mov     eax, edx                    ; eax=y*2
                shl     eax, 4                      ; eax=(y*2)<<4 = y*2*16 = y*32
                sub     eax, edx                    ; eax=y*32 - y*2=y*30
                imul    edx, ebx, 600               ; edx=x*600
                add     eax, edx                    ; eax=eax+edx=y*30 + x*600
                lea     edx, [eax+ecx]              ; edx=y*30 + x*600 + z
                mov     eax, [ebp+value]
                mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=value
                pop     ebx
                pop     ebp
                retn
insert          endp
```

Компилятор GCC решил всё сделать немного иначе. Для вычисления одной из операций ($30y$), GCC создал код, где нет самой операции умножения. Происходит это так: $(y+y) \ll 4 - (y+y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Таким образом, для вычисления $30y$ используется только операция сложения, операция битового сдвига и операция вычитания. Это работает быстрее.

16.5.2 ARM + Неоптимизирующий Xcode (LLVM) + Режим thumb

Листинг 16.7: Неоптимизирующий Xcode (LLVM) + Режим thumb

```
_insert

value    = -0x10
z        = -0xC
y        = -8
x        = -4

; выделить место в локальном стеке для 4 переменных типа int
SUB      SP, SP, #0x10
MOV      R9, 0xFC2 ; a
ADD      R9, PC
LDR.W    R9, [R9]
STR      R0, [SP,#0x10+x]
STR      R1, [SP,#0x10+y]
STR      R2, [SP,#0x10+z]
STR      R3, [SP,#0x10+value]
LDR      R0, [SP,#0x10+value]
LDR      R1, [SP,#0x10+z]
LDR      R2, [SP,#0x10+y]
LDR      R3, [SP,#0x10+x]
```

```
MOV      R12, 2400
MUL.W    R3, R3, R12
ADD      R3, R9
MOV      R9, 120
MUL.W    R2, R2, R9
ADD      R2, R3
LSLS     R1, R1, #2 ; R1=R1<<2
ADD      R1, R2
STR      R0, [R1]   ; R1 - адрес элемента массива
; освободить место в локальном стеке, выделенное для 4 переменных
ADD      SP, SP, #0x10
BX       LR
```

Неоптимизирующий LLVM сохраняет все переменные в локальном стеке, хотя это и избыточно. Адрес элемента массива вычисляется по уже рассмотренной формуле.

16.5.3 ARM + Оптимизирующий Xcode (LLVM) + Режим thumb

Листинг 16.8: Оптимизирующий Xcode (LLVM) + Режим thumb

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1,LSL#4  ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC            ; R9 = указатель на массив
LDR.W   R9, [R9]
MLA.W   R0, R0, R12, R9   ; R0 - x, R12 - 2400, R9 - указатель на a. R0=x*2400 + указатель на a
ADD.W   R0, R0, R1,LSL#3  ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + указатель на a + y*15*8 =
                          ; указатель на a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 - z, R3 - значение. адрес=R0+z*4 =
                          ; указатель на a + y*30*4 + x*600*4 + z*4
BX      LR
```

Тут используются уже описанные трюки для замены умножения на операции сдвига, сложения и вычитания.

Также мы видим новую для себя инструкцию `RSB` (*Reverse Subtract*). Она работает так же, как и `SUB`, только меняет операнды местами. Зачем? `SUB`, `RSB`, это те инструкции, ко второму операнду которых можно применить коэффициент сдвига, как мы видим и здесь: (`LSL#4`). Но этот коэффициент можно применить только ко второму операнду. Для коммутативных операций, таких как сложение или умножение, там операнды можно менять местами и это не влияет на результат. Но вычитание — операция некоммутативная, так что, для этих случаев существует инструкция `RSB`.

Инструкция "LDR.W R9, [R9]" работает как `LEA` (B.6.2) в x86, и здесь она ничего не делает, она избыточна. Вероятно, компилятор несоптимизировал её.

Глава 17

Битовые поля

Немало функций задают различные флаги в аргументах при помощи битовых полей[1]. Наверное, вместо этого, можно было бы использовать набор переменных типа *bool*, но это было бы не очень экономно.

17.1 Проверка какого-либо бита

17.1.1 x86

Например в Win32 API:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS↙
    ↳ , FILE_ATTRIBUTE_NORMAL, NULL);
```

Получаем (MSVC 2010):

Листинг 17.1: MSVC 2010

```
        push    0
        push    128                                     ; 00000080H
        push    4
        push    0
        push    1
        push    -1073741824                             ; c0000000H
        push    OFFSET $SG78813
        call    DWORD PTR __imp__CreateFileA@28
        mov     DWORD PTR _fh$[ebp], eax
```

Заглянем в файл WinNT.h:

Листинг 17.2: WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Все ясно, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, и это значение используется как второй аргумент для `CreateFile()`[2] function.

Как `CreateFile()` будет проверять флаги?

Заглянем в KERNEL32.DLL от Windows XP SP3 x86 и найдем в функции `CreateFileW()` в том числе и такой фрагмент кода:

Листинг 17.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429                test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D                mov     [ebp+var_8], 1
.text:7C83D434                jz      short loc_7C83D417
.text:7C83D436                jmp     loc_7C810817
```

[1] bit fields в англоязычной литературе

[2] MSDN: CreateFile function

Здесь мы видим инструкцию TEST, впрочем, она берет не весь второй аргумент функции, но только его самый старший байт (ebp+dwDesiredAccess+3) и проверяет его на флаг 0x40 (имеется ввиду флаг GENERIC_WRITE).

TEST это то же что и AND, только без сохранения результата (вспомните что CMP это то же что и SUB, только без сохранения результатов (6.6.1)).

Логика данного фрагмента кода примерно такая:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Если после операции AND останется этот бит, то флаг ZF не будет поднят и условный переход JZ не сработает. Переход возможен только если в переменной dwDesiredAccess отсутствует бит 0x40000000 — тогда результат AND будет 0, флаг ZF будет поднят и переход сработает.

Попробуем GCC 4.4.1 и Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;

        handle=open ("file", O_RDWR | O_CREAT);
};
```

Получим:

Листинг 17.4: GCC 4.4.1

```
                public main
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_1C], 42h
                mov     [esp+20h+var_20], offset aFile ; "file"
                call    _open
                mov     [esp+20h+var_4], eax
                leave
                retn
main            endp
```

Заглянем в реализацию функции open() в библиотеке libc.so.6, но обнаружим что там только вызов сисколла:

Листинг 17.5: open() (libc.so.6)

```
.text:000BE69B                mov     edx, [esp+4+mode] ; mode
.text:000BE69F                mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3                mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7                mov     eax, 5
.text:000BE6AC                int     80h             ; LINUX - sys_open
```

Значит, битовые поля флагов open() вероятно проверяются где-то в ядре Linux.

Разумеется, и стандартные библиотеки Linux и ядро Linux можно получить в виде исходников, но нам интересно попробовать разобраться без них.

Итак, при вызове сисколла sys_open, управление в конечном итоге передается в do_sys_open в ядре Linux 2.6. Оттуда — в do_filp_open() (эта функция находится в исходниках ядра в файле fs/namei.c).

N.B. Помимо передачи параметров функции через стек, существует также возможность передавать некоторые из них через регистры. Это называется в том числе fastcall (44.3). Это работает немного быстрее, так

как процессору не нужно обращаться к стеку, лежащему в памяти для чтения аргументов. В GCC есть опция *regparm*[3], и с её помощью можно задать, сколько аргументов можно передать через регистры.

Ядро Linux 2.6 собирается с опцией `-mregparm=3` [4] [5].

И для нас это означает, что первые три аргумента функции будут передаваться через регистры `EAX`, `EDX` и `ECX`, а остальные через стек. Разумеется, если аргументов у функции меньше трех, то будет задействована только часть регистров.

Итак, качаем ядро 2.6.31, собираем его в Ubuntu: `make vmlinux`, открываем в IDA, находим функцию `do_filp_open()`. В начале мы увидим подобное (комментарии мои):

Листинг 17.6: do_filp_open() (linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (пятый аргумент)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (первый аргумент)
                mov     [ebp+var_7C], edx ; pathname (второй аргумент)
                mov     [ebp+var_78], ecx ; open_flag (третий аргумент)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; EBX <- open_flag
```

GCC сохраняет значения первых трех аргументов в локальном стеке. Иначе, если эти три регистра не трогать вообще, то функции компилятора, распределяющей переменные по регистрам (так называемый register allocator), будет очень тесно.

Далее находим примерно такой фрагмент кода:

Листинг 17.7: do_filp_open() (linux kernel 2.6.31)

```
loc_C01EF6B4:                           ; CODE XREF: do_filp_open+4F
                test    bl, 40h         ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

`0x40` — это то чему равен макрос `O_CREAT`. `open_flag` проверяется на наличие бита `0x40` и если бит равен 1, то выполняется следующие за `JNZ` инструкции.

17.1.2 ARM

В ядре Linux 3.8.0 бит `O_CREAT` проверяется немного иначе.

Листинг 17.8: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
```

[3] http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm

[4] http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

[5] См. также файл `arch\x86\include\asm\calling.h` в исходниках ядра

```
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}


static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

Вот как это выглядит в IDA, ядро скомпилированное для режима ARM:

Листинг 17.9: do_last() (vmlinux)

```
...
.text:C0169EA8                 MOV             R9, R3  ; R3 - (4th argument) open_flag
...
.text:C0169ED4                 LDR             R6, [R9] ; R6 - open_flag
...
.text:C0169F68                 TST             R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C                 BNE             loc_C016A128
.text:C0169F70                 LDR             R2, [R4,#0x10]
.text:C0169F74                 ADD             R12, R4, #8
.text:C0169F78                 LDR             R3, [R4,#0xC]
.text:C0169F7C                 MOV             R0, R4
.text:C0169F80                 STR             R12, [R11,#var_50]
.text:C0169F84                 LDRB            R3, [R2,R3]
.text:C0169F88                 MOV             R2, R8
.text:C0169F8C                 CMP             R3, #0
.text:C0169F90                 ORRNE           R1, R1, #3
.text:C0169F94                 STRNE           R1, [R4,#0x24]
.text:C0169F98                 ANDS            R3, R6, #0x200000
.text:C0169F9C                 MOV             R1, R12
.text:C0169FA0                 LDRNE           R3, [R4,#0x24]
.text:C0169FA4                 ANDNE           R3, R3, #1
.text:C0169FA8                 EORNE           R3, R3, #1
.text:C0169FAC                 STR             R3, [R11,#var_54]
.text:C0169FB0                 SUB             R3, R11, #-var_38
.text:C0169FB4                 BL              lookup_fast
...
.text:C016A128 loc_C016A128                            ; CODE XREF: do_last.isra.14+DC
.text:C016A128                 MOV             R0, R4
.text:C016A12C                 BL              complete_walk
...
```

TST это аналог инструкции TEST в x86.

Мы можем "узнать" визуально этот фрагмент кода по тому что в одном случае исполнится функция lookup_fast(), а в другом complete_walk(). Это соответствует исходному коду функции do_last().

Макрос O_CREAT здесь так же равен 0x40.

17.2 Установка/сброс отдельного бита

Например:

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};
```

17.2.1 x86

Имеем в итоге (MSVC 2010):

Листинг 17.10: MSVC 2010

```
_rt$ = -4         ; size = 4
_a$ = 8           ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384                ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                 ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f  ENDP
```

Инструкция OR здесь добавляет в переменную еще один бит, игнорируя остальные.

А AND сбрасывает некий бит. Можно также сказать, что AND здесь копирует все биты, кроме одного. Действительно, во втором операнде AND выставлены в единицу те биты, которые нужно сохранить, кроме одного, копировать который мы не хотим (и который 0 в битовой маске). Так проще понять и запомнить.

Если скомпилировать в MSVC с оптимизацией (/Ox), то код будет еще короче:

Листинг 17.11: Оптимизирующий MSVC

```
_a$ = 8                     ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    and    eax, -513        ; fffffdffH
    or     eax, 16384       ; 00004000H
```

```
    ret    0
_f    ENDP
```

Попробуем GCC 4.4.1 без оптимизации:

Листинг 17.12: Неоптимизирующий GCC

```
                public f
f               proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
                and     [ebp+var_4], 0FFFFFDFFh
                mov     eax, [ebp+var_4]
                leave
                retn
f               endp
```

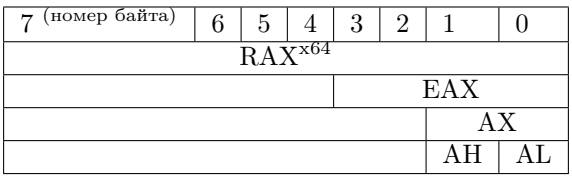
Также избыточный код, хотя короче, чем у MSVC без оптимизации.

Попробуем теперь GCC с оптимизацией `-O3`:

Листинг 17.13: Оптимизирующий GCC

```
                public f
f               proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f               endp
```

Уже короче. Важно отметить что через регистр `AH`, компилятор работает с частью регистра `EAX`, эта его часть от 8-го до 15-го бита включительно.



N.B. В 16-битном процессоре 8086 аккумулятор имел название `AX` и состоял из двух 8-битных половин — `AL` (младшая часть) и `AH` (старшая). В 80386 регистры были расширены до 32-бит, аккумулятор стал называться `EAX`, но в целях совместимости, к его *более старым* частям все еще можно обращаться как к `AX`/`AH`/`AL`.

Из-за того, что все x86 процессоры — наследники 16-битного 8086, эти *старые* 16-битные опкоды короче нежели более новые 32-битные. Поэтому, инструкция ‘‘`or ah, 40h`’’ занимает только 3 байта. Было бы логичнее сгенерировать здесь ‘‘`or eax, 04000h`’’, но это уже 5 байт, или даже 6 (если регистр в первом операнде не `EAX`).

Если мы скомпилируем этот же пример не только с включенной оптимизацией `-O3`, но еще и с опцией `regparm=3`, о которой я писал немного выше, то получится еще короче:

Листинг 17.14: Оптимизирующий GCC

```
                public f
```

```
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

Действительно — первый аргумент уже загружен в EAX, и прямо здесь можно начинать с ним работать. Интересно, что и пролог функции (“push ebp / mov ebp,esp”) и эпилог (“pop ebp”) функции можно смело выкинуть за ненадобностью, но возможно GCC еще не так хорош для подобных оптимизаций по размеру кода. Впрочем, в реальной жизни, подобные короткие функции лучше всего автоматически делать в виде *inline-функций* (27).

17.2.2 ARM + Оптимизирующий Keil + Режим ARM

Листинг 17.15: Оптимизирующий Keil + Режим ARM

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

BIC это “логическое и”, аналог AND в x86. ORR это “логическое или”, аналог OR в x86.
Пока всё понятно.

17.2.3 ARM + Оптимизирующий Keil + Режим thumb

Листинг 17.16: Оптимизирующий Keil + Режим thumb

```
01 21 89 03          MOVS    R1, 0x4000
08 43                ORRS    R0, R1
49 11                ASRS    R1, R1, #5   ; generate 0x200 and place to R1
88 43                BICS    R0, R1
70 47                BX      LR
```

Вероятно, Keil решил, что код в режиме thumb, получающий 0x200 из 0x4000, будет компактнее нежели код, записывающий 0x200 в какой-нибудь регистр.

Поэтому, при помощи инструкции ASRS (арифметический сдвиг вправо), это значение вычисляется как $0x4000 \gg 5$.

17.2.4 ARM + Оптимизирующий Xcode (LLVM) + Режим ARM

Листинг 17.17: Оптимизирующий Xcode (LLVM) + Режим ARM

```
42 0C C0 E3          BIC             R0, R0, #0x4200
01 09 80 E3          ORR             R0, R0, #0x4000
1E FF 2F E1          BX              LR
```

Код, который был сгенерирован LLVM, в исходном коде, на самом деле, выглядел бы так:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

И он делает то же самое что нам нужно. Но почему 0x4200? Возможно, это артефакт оптимизатора LLVM [6]. Возможно, ошибка оптимизатора компилятора, но создаваемый код все же работает верно.

Об аномалиях компиляторов, подробнее читайте здесь (64).

Для режима Thumb, Оптимизирующий Xcode (LLVM) генерирует точно такой же код.

[6] Это был LLVM build 2410.2.00 входящий в состав Apple Xcode 4.6.3

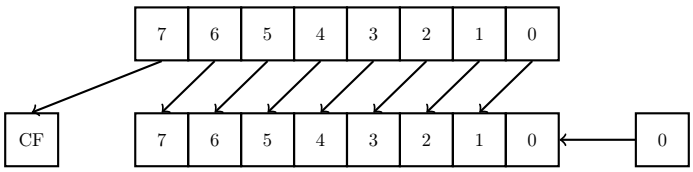
17.3 Сдвиги

Битовые сдвиги в Си/Си++ реализованы при помощи операторов ≪ и ≫.

Вот этот несложный пример иллюстрирует функцию, считающую количество бит-единиц во входной переменной:

```
#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};
```

В этом цикле, счетчик итераций i считает от 0 до 31, а $1 \ll i$ будет от 1 до `0x80000000`. Описывая это словами, можно сказать *сдвинуть единицу на n бит влево*. Т.е., в некотором смысле, выражение $1 \ll i$ последовательно выдаст все возможные позиции бит в 32-битном числе. Кстати, освободившийся бит справа всегда обнуляется. Макрос `IS_SET` проверяет наличие этого бита в `a`.



Макрос `IS_SET` на самом деле это операция логического И (*AND*) и она возвращает 0 если бита там нет, либо эту же битовую маску, если бит там есть. В Си/Си++, конструкция `if()` срабатывает, если выражение внутри её не ноль, пусть хоть 123456, поэтому все будет работать.

17.3.1 x86

Компилируем (MSVC 2010):

Листинг 17.18: MSVC 2010

```
_rt$ = -8               ; size = 4
_i$ = -4                ; size = 4
_a$ = 8                 ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]  ; инкремент i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32   ; 00000020H
    jge    SHORT $LN2@f             ; цикл закончился?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                  ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f             ; результат исполнения инструкции AND был 0?
```

```
                                    ; тогда пропускаем следующие команды
    mov    eax, DWORD PTR _rt$[ebp] ; нет, не ноль
    add    eax, 1                   ; инкремент rt
    mov    DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp    SHORT $LN3@f
$LN2@f:
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Вот так работает SHL (*SHift Left*).

Скомпилируем то же и в GCC 4.4.1:

Листинг 17.19: GCC 4.4.1

```
                public f
f               proc near

rt              = dword ptr -0Ch
i               = dword ptr -8
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483EF
loc_80483D0:
                mov     eax, [ebp+i]
                mov     edx, 1
                mov     ebx, edx
                mov     ecx, eax
                shl     ebx, cl
                mov     eax, ebx
                and     eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_80483EB
                add     [ebp+rt], 1
loc_80483EB:
                add     [ebp+i], 1
loc_80483EF:
                cmp     [ebp+i], 1Fh
                jle     short loc_80483D0
                mov     eax, [ebp+rt]
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn
f               endp
```

Инструкции сдвига также активно применяются при делении или умножении на числа-степени двойки (1, 2, 4, 8, и т.д.).

Например:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

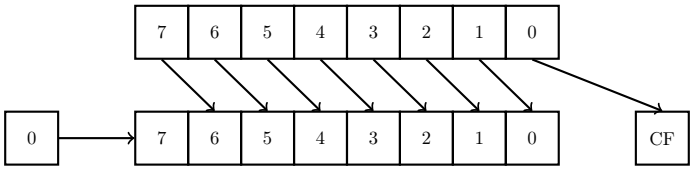
Имеем в итоге (MSVC 2010):

Листинг 17.20: MSVC 2010

```
_a$ = 8                                                 ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

Инструкция SHR (*SHift Right*) в данном примере сдвигает число на 2 бита вправо. При этом, освободившиеся два бита слева (т.е., самые старшие разряды), выставляются в нули. А самые младшие 2 бита выкидываются. Фактически, эти два выкинутых бита — остаток от деления.

Инструкция SHR работает так же, как и SHL, только в другую сторону.



Для того, чтобы это проще понять, представьте себе десятичную систему счисления и число 23. 23 можно разделить на 10 просто откинув последний разряд (3 — это остаток от деления). После этой операции останется 2 как частное.

Так и с умножением. Умножить на 4 это просто сдвинуть число на 2 бита влево, вставив 2 нулевых бита справа (как два самых младших бита). Это как умножить 3 на 100 — нужно просто дописать два нуля справа.

17.3.2 ARM + Оптимизирующий Xcode (LLVM) + Режим ARM

Листинг 17.21: Оптимизирующий Xcode (LLVM) + Режим ARM

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; set flags according to R1 & (R2<<R3)
                ADD             R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; if ZF flag is cleared by TST, R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST это то же что и TEST в x86.

Как я уже указывал (14.2.1), в режиме ARM нет отдельной инструкции для сдвигов. Однако, модификаторами LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) и RRX (*Rotate Right with Extend*) можно дополнять некоторые инструкции, такие как MOV, TST, CMP, ADD, SUB, RSB[7].

Эти модификаторы указывают, как сдвигать второй операнд, и на сколько.

Таким образом, инструкция "TST R1, R2,LSL R3" здесь работает как $R1 \wedge (R2 \ll R3)$.

17.3.3 ARM + Оптимизирующий Xcode (LLVM) + Режим thumb-2

Почти такое же, только здесь применяется пара инструкций LSL.W/TST вместо одной TST, ведь в режиме thumb нельзя добавлять указывать модификатор LSL прямо в TST.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
```

[7] Эти инструкции также называются "data processing instructions"

```
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
                BNE             loc_2F7A
                BX              LR
```

17.4 Пример вычисления CRC32

Это распространенный табличный способ вычисления хеша алгоритмом CRC32[8].

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long  ub4;
typedef  unsigned char  ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
  0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
  0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
  0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
  0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
  0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
  0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
  0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
  0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
  0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
  0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
  0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
  0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
  0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
  0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
  0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
  0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
  0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
  0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
  0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
  0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
  0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
  0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
  0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
  0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
  0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
  0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
```

[8]Исходник взят тут: http://burtleburtle.net/bob/c/crc.c

```
  0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
  0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
  0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
  0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

/* how to derive the values in crctab[] from polynomial 0xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n");
  }
}


/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
{
  ub4  i;
  const ub1 *k = key;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
  return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
  char s[1000];
  while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
  return 0;
}
```

Нас интересует функция `crc()`. Кстати, обратите внимание на два инициализатора в выражении `for()`: `hash=len, i=0`. Стандарт Си/Си++, конечно, допускает это. А в итоговом коде, вместо одной операции инициализации цикла, будет две.

Компилируем в MSVC с оптимизацией (`/Ox`). Для краткости, я приведу только функцию `crc()`, с некоторыми комментариями.

```
_key$ = 8           ; size = 4
_len$ = 12          ; size = 4
_hash$ = 16         ; size = 4
_crc    PROC
    mov    edx, DWORD PTR _len$[esp-4]
    xor    ecx, ecx ; i будет лежать в регистре ECX
    mov    eax, edx
    test   edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
```

```
    push   esi
    mov    esi, DWORD PTR _key$[esp+4] ; ESI = key
    push   edi
$LL3@crc:

; работаем с байтами используя 32-битные регистры. в EDI положим байт с адреса key+i

    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax  ; EBX = (hash = len)
    and    ebx, 255  ; EBX = hash & 0xff


; XOR EDI, EBX (EDI=EDI^EBX) - эта операция задействует все 32 бита каждого регистра
; но остальные биты (8-31) будут обнулены всегда, так что все OK
; они обнулены потому что для EDI это было сделано инструкцией MOVZX выше
; а старшие биты EBX были сброшены инструкцией AND EBX, 255 (255 = 0xff)

    xor    edi, ebx

; EAX=EAX>>8; образовавшиеся из ниоткуда биты в результате (биты 24-31) будут заполнены нулями
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - выбираем элемент из таблицы crctab[] под номером EDI
    xor    eax, DWORD PTR _crctab[edi*4]
    inc    ecx            ; i++
    cmp    ecx, edx       ; i<len ?
    jb     SHORT $LL3@crc ; да
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc:
    ret    0
_crc    ENDP
```

Попробуем то же самое в GCC 4.4.1 с опцией `-O3`:

```
                public crc
crc             proc near

key             = dword ptr  8
hash            = dword ptr  0Ch

                push    ebp
                xor     edx, edx
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+key]
                push    ebx
                mov     ebx, [ebp+hash]
                test    ebx, ebx
                mov     eax, ebx
                jz      short loc_80484D3
                nop                     ; выравнивание
                lea     esi, [esi+0]    ; выравнивание; ESI не меняется здесь

loc_80484B8:
                mov     ecx, eax        ; сохранить предыдущее состояние хеша в ECX
                xor     al, [esi+edx]   ; AL=*(key+i)
                add     edx, 1          ; i++
                shr     ecx, 8          ; ECX=hash>>8
                movzx   eax, al         ; EAX=*(key+i)
                mov     eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
```

```
                xor     eax, ecx        ; hash=EAX^ECX
                cmp     ebx, edx
                ja      short loc_80484B8

loc_80484D3:
                pop     ebx
                pop     esi
                pop     ebp
                retn
crc             endp
\
```

GCC немного выровнял начало тела цикла по 8-байтной границе, для этого добавил `NOP` и `lea esi, [esi+0]` (что тоже *холостая операция*). Подробнее об этом смотрите в разделе о npad (61).

Глава 18

Структуры

В принципе, структура в Си/Си++ это, с некоторыми допущениями, просто всегда лежащий рядом, и в той же последовательности, набор переменных, не обязательно одного типа [1].

18.1 Пример SYSTEMTIME

Возьмем, к примеру, структуру SYSTEMTIME[2] из win32 описывающую время.

Она объявлена так:

Листинг 18.1: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Пишем на Си функцию для получения текущего системного времени:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Что в итоге (MSVC 2010):

Листинг 18.2: MSVC 2010

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
```

[1] AKA “гетерогенный контейнер”

[2] MSDN: SYSTEMTIME structure

```
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push   eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Под структуру в стеке выделено 16 байт — именно столько будет `sizeof(WORD)*8` (в структуре 8 переменных с типом WORD).

Обратите внимание на тот факт, что структура начинается с поля `wYear`. Можно сказать, что в качестве аргумента для `GetSystemTime()`[3]передается указатель на структуру SYSTEMTIME, но можно также сказать, что передается указатель на поле `wYear`, что одно и тоже! `GetSystemTime()` пишет текущий год в тот WORD на который указывает переданный указатель, затем сдвигается на 2 байта вправо, пишет текущий месяц, и т.д., и т.д.

Тот факт, что поля структуры это просто переменные расположенные рядом, я могу проиллюстрировать следующим образом. Глядя на описание структуры `SYSTEMTIME`, я могу переписать этот простой пример так:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */);

    return;
};
```

Компилятор немного поворчит:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to '↙
    ↳ LPSYSTEMTIME'
```

Тем не менее, выдаст такой код:

Листинг 18.3: MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16   ; size = 16
```

[3]MSDN: GetSystemTime function

```
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

И это работает так же!

Любопытно что результат на ассемблере неотличим от предыдущего. Таким образом, глядя на этот код, никогда нельзя сказать с уверенностью, была ли там объявлена структура, либо просто набор переменных.

Тем не менее, никто в здравом уме делать так не будет. Потому что это неудобно. К тому же, иногда, поля в структуре могут меняться разработчиками, переставляться местами, и т.д.

18.2 Выделяем место для структуры через malloc()

Однако, бывает и так, что проще хранить структуры не в стеке, а в куче:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Скомпилируем на этот раз с оптимизацией (/Ox) чтобы было проще увидеть то, что нам нужно.

Листинг 18.4: Оптимизирующий MSVC

```
_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP
```

Итак, `sizeof(SYSTEMTIME) = 16`, именно столько байт выделяется при помощи `malloc()`. Она возвращает указатель на только что выделенный блок памяти в `EAX`, который копируется в `ESI`. Win32 функция `GetSystemTime()` обязуется сохранить состояние `ESI`, поэтому здесь оно нигде не сохраняется и продолжает использоваться после вызова `GetSystemTime()`.

Новая инструкция — `MOVZX` (*Move with Zero eXtent*). Она нужна почти там же где и `MOVSX` (13.1.1), только всегда очищает остальные биты в 0. Дело в том, что `printf()` требует 32-битный тип *int*, а в структуре лежит WORD — это 16-битный беззнаковый тип. Поэтому копируя значение из WORD в *int*, нужно очистить биты от 16 до 31, иначе там будет просто случайный мусор, оставшийся от предыдущих действий с регистрами.

В этом примере я тоже могу представить структуру как массив WORD-ов:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Получим такое:

Листинг 18.5: Оптимизирующий MSVC

```
$SG78594 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

И снова мы получаем идентичный код, неотличимый от предыдущего. Но и снова я должен отметить, что в реальности так лучше не делать.

18.3 struct tm

18.3.1 Linux

В Линуксе, для примера, возьмем структуру `tm` из `time.h`:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

Компилируем при помощи GCC 4.4.1:

Листинг 18.6: GCC 4.4.1

```
main proc near
     push    ebp
     mov     ebp, esp
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; первый агрумент для time()
     call    time
     mov     [esp+3Ch], eax
     lea     eax, [esp+3Ch]  ; берем указатель на то что вернула time()
     lea     edx, [esp+10h]  ; по ESP+10h будет начинаться структура struct tm
     mov     [esp+4], edx    ; передаем указатель на начало структуры
     mov     [esp], eax      ; передаем указатель на результат time()
     call    localtime_r
     mov     eax, [esp+24h]  ; tm_year
     lea     edx, [eax+76Ch] ; edx=eax+1900
     mov     eax, offset format ; "Year: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+20h]      ; tm_mon
     mov     eax, offset aMonthD ; "Month: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+1Ch]     ; tm_mday
     mov     eax, offset aDayD  ; "Day: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+18h]     ; tm_hour
     mov     eax, offset aHourD ; "Hour: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+14h]        ; tm_min
     mov     eax, offset aMinutesD ; "Minutes: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+10h]
     mov     eax, offset aSecondsD ; "Seconds: %d\n"
     mov     [esp+4], edx          ; tm_sec
     mov     [esp], eax
     call    printf
     leave
     retn
main endp
```

К сожалению, по какой-то причине, IDA не сформировала названия локальных переменных в стеке. Но так как мы уже опытные реверсеры :-) то можем обойтись и без этого в таком простом примере.

Обратите внимание на `lea edx, [eax+76Ch]` — эта инструкция прибавляет `0x76C` к `EAX`, но не модифицирует флаги. См. также соответствующий раздел об инструкции `LEA` (B.6.2).

Чтобы проиллюстрировать то что структура — это просто набор переменных лежащих в одном месте, переделаем немного пример, заглянув предварительно в файл *time.h*:

Листинг 18.7: time.h

```
struct tm
{
```

```
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

N.B. В `localtime_r` передается указатель именно на `tm_sec`, т.е., на первый элемент “структуры”.

В итоге, и этот компилятор поворчит:

Листинг 18.8: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [↙
    ↳ enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

Тем не менее, сгенерирует такое:

Листинг 18.9: GCC 4.7.3

```
main      proc near

var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 30h
          call    __main
```

```
        mov     [esp+30h+var_30], 0 ; arg 0
        call    time
        mov     [esp+30h+unix_time], eax
        lea     eax, [esp+30h+tm_sec]
        mov     [esp+30h+var_2C], eax
        lea     eax, [esp+30h+unix_time]
        mov     [esp+30h+var_30], eax
        call    localtime_r
        mov     eax, [esp+30h+tm_year]
        add     eax, 1900
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_mon]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_mday]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_hour]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_min]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
        call    printf
        mov     eax, [esp+30h+tm_sec]
        mov     [esp+30h+var_2C], eax
        mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
        call    printf
        leave
        retn
main    endp
```

Этот код почти идентичен уже рассмотренному, и нельзя сказать, была ли структура в оригинальном исходном коде либо набор переменных.

И это работает. Однако, в реальности так лучше не делать. Обычно, компилятор располагает переменные в локальном стеке в том же порядке, в котором они объявляются в функции. Тем не менее, никакой гарантии нет.

Кстати, какой-нибудь другой компилятор может предупредить, что переменные `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, но не `tm_sec`, используются без инициализации. Действительно, ведь компилятор не знает что они будут заполнены при вызове функции `localtime_r()`.

Я выбрал именно этот пример для иллюстрации, потому что все члены структуры имеют тип *int*, а члены структуры `SYSTEMTIME` — 16-битные `WORD`, и если их объявлять так же, как локальные переменные, то они будут выровнены по 32-битной границе и ничего не выйдет (потому что `GetSystemTime()` заполнит их неверно). Читайте об этом в следующей секции: “Упаковка полей в структуре”.

Так что, структура — это просто набор переменных лежащих в одном месте, рядом. Я мог бы сказать что структура — это такой синтаксический сахар, заставляющий компилятор удерживать их в одном месте. Впрочем, я не специалист по языкам программирования, так что, скорее всего, ошибаюсь с этим термином. Кстати, когда-то, в очень ранних версиях Си (перед 1972) структур не было вовсе [29].

18.3.2 ARM + Оптимизирующий Keil + Режим thumb

Этот же пример:

Листинг 18.10: Оптимизирующий Keil + Режим thumb

```
var_38 = -0x38
var_34 = -0x34
```

```
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

       PUSH    {LR}
       MOVS    R0, #0          ; timer
       SUB     SP, SP, #0x34
       BL      time
       STR     R0, [SP,#0x38+timer]
       MOV     R1, SP          ; tp
       ADD     R0, SP, #0x38+timer ; timer
       BL      localtime_r
       LDR     R1, =0x76C
       LDR     R0, [SP,#0x38+var_24]
       ADDS    R1, R0, R1
       ADR     R0, aYearD      ; "Year: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_28]
       ADR     R0, aMonthD     ; "Month: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_2C]
       ADR     R0, aDayD       ; "Day: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_30]
       ADR     R0, aHourD      ; "Hour: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_34]
       ADR     R0, aMinutesD   ; "Minutes: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_38]
       ADR     R0, aSecondsD   ; "Seconds: %d\n"
       BL      __2printf
       ADD     SP, SP, #0x34
       POP     {PC}
```

18.3.3 ARM + Оптимизирующий Xcode (LLVM) + Режим thumb-2

IDA "узнала" структуру tm (потому что IDA "знает" типы аргументов библиотечных функций, таких как `localtime_r()`), поэтому показала здесь обращения к отдельным элементам структуры и присвоила им имена.

Листинг 18.11: Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
var_38 = -0x38
var_34 = -0x34

       PUSH {R7,LR}
       MOV  R7, SP
       SUB  SP, SP, #0x30
       MOVS R0, #0  ; time_t *
       BLX  _time
       ADD  R1, SP, #0x38+var_34 ; struct tm *
       STR  R0, [SP,#0x38+var_38]
       MOV  R0, SP  ; time_t *
       BLX  _localtime_r
       LDR  R1, [SP,#0x38+var_34.tm_year]
       MOV  R0, 0xF44 ; "Year: %d\n"
       ADD  R0, PC  ; char *
       ADDW R1, R1, #0x76C
       BLX  _printf
```

```
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_min]
        MOV  R0, 0xF28 ; "Minutes: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34]
        MOV  R0, 0xF25 ; "Seconds: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        ADD  SP, SP, #0x30
        POP  {R7,PC}

...

00000000 tm        struc ; (sizeof=0x2C, standard type)
00000000 tm_sec    DCD ?
00000004 tm_min    DCD ?
00000008 tm_hour   DCD ?
0000000C tm_mday   DCD ?
00000010 tm_mon    DCD ?
00000014 tm_year   DCD ?
00000018 tm_wday   DCD ?
0000001C tm_yday   DCD ?
00000020 tm_isdst  DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone   DCD ? ; offset
0000002C tm        ends
```

18.4 Упаковка полей в структуре

Достаточно немаловажный момент, это упаковка полей в структурах[4].

Возьмем простой пример:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};
```

[4] См. также: Wikipedia: Выравнивание данных

Как видно, мы имеем два поля *char* (занимающий один байт) и еще два — *int* (по 4 байта).

18.4.1 x86

Компилируется это все в:

```
_s$ = 8 ; size = 16
?f@@YAXUs@@@Z PROC ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+12]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+8]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+4]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP ; f
_TEXT    ENDS
```

Мы видим здесь что адрес каждого поля в структуре выравнивается по 4-байтной границе. Так что каждый *char* здесь занимает те же 4 байта что и *int*. Зачем? Затем что процессору удобнее обращаться по таким адресам и кэшировать данные из памяти.

Но это не экономично по размеру данных.

Попробуем скомпилировать тот же исходник с опцией (`/Zp1`) (*/Zp[n] pack structures on n-byte boundary*).

Листинг 18.12: MSVC /Zp1

```
_TEXT    SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC    ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+6]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+5]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+1]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP    ; f
```

Теперь структура занимает 10 байт и все *char* занимают по байту. Что это дает? Экономию места. Недостаток — процессор будет обращаться к этим полям не так эффективно по скорости, как мог бы.

Как нетрудно догадаться, если структура используется много в каких исходниках и объектных файлах, все они должны быть откомпилированы с одним и тем же соглашением об упаковке структур.

Помимо ключа MSVC `/Zp`, указывающего, по какой границе упаковывать поля структур, есть также опция компилятора `#pragma pack`, её можно указывать прямо в исходнике. Это справедливо и для MSVC[5] и GCC[6].

[5] MSDN: Working with Packing Structures

[6] Structure-Packing Pragmas

Давайте теперь вернемся к SYSTEMTIME, которая состоит из 16-битных полей. Откуда наш компилятор знает что их надо паковать по однобайтной границе?

В файле WinNT.h попадается такое:

Листинг 18.13: WinNT.h

```
#include "pshpack1.h"
```

И такое:

Листинг 18.14: WinNT.h

```
#include "pshpack4.h"                    // 4 byte packing is the default
```

Сам файл PshPack1.h выглядит так:

Листинг 18.15: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Собственно, так и задается компилятору, как паковать объявленные после #pragma pack структуры.

18.4.2 ARM + Оптимизирующий Keil + Режим thumb

Листинг 18.16: Оптимизирующий Keil + Режим thumb

```
.text:0000003E             exit ; CODE XREF: f+16
.text:0000003E 05 B0               ADD     SP, SP, #0x14
.text:00000040 00 BD               POP     {PC}


.text:00000280             f
.text:00000280
.text:00000280             var_18 = -0x18
.text:00000280             a      = -0x14
.text:00000280             b      = -0x10
.text:00000280             c      = -0xC
.text:00000280             d      = -8
.text:00000280
.text:00000280 0F B5               PUSH    {R0-R3,LR}
.text:00000282 81 B0               SUB     SP, SP, #4
.text:00000284 04 98               LDR     R0, [SP,#16]    ; d
.text:00000286 02 9A               LDR     R2, [SP,#8]     ; b
.text:00000288 00 90               STR     R0, [SP]
.text:0000028A 68 46               MOV     R0, SP
.text:0000028C 03 7B               LDRB    R3, [R0,#12]    ; c
.text:0000028E 01 79               LDRB    R1, [R0,#4]     ; a
.text:00000290 59 A0               ADR     R0, aADBDCDDD   ; "a=%d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF         BL      __2printf
.text:00000296 D2 E6               B       exit
```

Как мы помним, здесь передается не указатель на структуру, а сама структура, а так как в ARM первые 4 аргумента функции передаются через регистры, то поля структуры передаются через R0-R3.

Инструкция LDRB загружает один байт из памяти и расширяет до 32-бит учитывая знак. Это то же что и инструкция MOVSX (13.1.1) в x86. Она здесь применяется для загрузки полей a и c из структуры.

Еще что бросается в глаза, так это то что вместо эпилога функции, переход на эпилог другой функции! Действительно, то была совсем другая, не относящаяся к этой, функция, однако, она имела точно такой же эпилог (видимо, тоже хранила в стеке 5 локальных переменных ($5 * 4 = 0x14$)). К тому же, она находится рядом (обратите внимание на адреса). Действительно, нет никакой разницы, какой эпилог исполнять, если он работает так же, как нам нужно. Keil решил использовать часть другой ф-ции, вероятно, из-за экономии. Эпилог занимает 4 байта, а переход — только 2.

18.4.3 ARM + Оптимизирующий Xcode (LLVM) + Режим thumb-2

Листинг 18.17: Оптимизирующий Xcode (LLVM) + Режим thumb-2

```
var_C = -0xC

      PUSH   {R7,LR}
      MOV    R7, SP
      SUB    SP, SP, #4
      MOV    R9, R1 ; b
      MOV    R1, R0 ; a
      MOVW   R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
      SXTB   R1, R1  ; prepare a
      MOVT.W R0, #0
      STR    R3, [SP,#0xC+var_C] ; place d to stack for printf()
      ADD    R0, PC  ; format-string
      SXTB   R3, R2  ; prepare c
      MOV    R2, R9  ; b
      BLX    _printf
      ADD    SP, SP, #4
      POP    {R7,PC}
```

SXTB (*Signed Extend Byte*) это также аналог MOVSX (13.1.1) в x86, только работает не с памятью, а с регистром. Всё остальное — так же.

18.5 Вложенные структуры

Теперь, как насчет ситуаций, когда одна структура определяет внутри себя еще одну структуру?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};
```

... в этом случае, оба поля `inner_struct` просто будут располагаться между полями a,b и d,e в `outer_struct`.

Компилируем (MSVC 2010):

Листинг 18.18: MSVC 2010

```
_s$ = 8 ; size = 24
_f  PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+20] ; e
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+16] ; d
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+12] ; c.b
    push   edx
    mov    eax, DWORD PTR _s$[ebp+8] ; c.a
    push   eax
    mov    ecx, DWORD PTR _s$[ebp+4] ; b
    push   ecx
    movsx  edx, BYTE PTR _s$[ebp] ;a
    push   edx
    push   OFFSET $SG2466
    call   _printf
    add    esp, 28
    pop    ebp
    ret    0
_f  ENDP
```

Очень любопытный момент в том, что глядя на этот код на ассемблере, мы даже не видим, что была использована какая-то еще другая структура внутри этой! Так что, пожалуй, можно сказать, что все вложенные структуры в итоге разворачиваются в одну, *линейную* или *одномерную* структуру.

Конечно, если заменить объявление `struct inner_struct c;` на `struct inner_struct *c;` (объявляя таким образом указатель), ситуация будет совсем иная.

18.6 Работа с битовыми полями в структуре

18.6.1 Пример CPUID

Язык Си/Си++ позволяет указывать, сколько именно бит отвести для каждого поля структуры. Это удобно если нужно экономить место в памяти. К примеру, для переменной типа *bool* достаточно одного бита. Но, это не очень удобно, если нужна скорость.

Рассмотрим пример с инструкцией `CPUID`[7]. Эта инструкция возвращает информацию о том, какой процессор имеется в наличии и какие возможности он имеет.

Если перед исполнением инструкции в `EAX` будет 1, то `CPUID` вернет упакованную в `EAX` такую информацию о процессоре:

3:0	Stepping
7:4	Model
11:8	Family
13:12	Processor Type
19:16	Extended Model
27:20	Extended Family

MSVC 2010 имеет макрос для `CPUID`, а GCC 4.4.1 — нет. Поэтому для GCC сделаем эту функцию сами, используя его встроенный ассемблер[8].

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
```

[7] http://en.wikipedia.org/wiki/CPUID

[8] Подробнее о встроенном ассемблере GCC

```
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

После того как CPUID заполнит EAX/EBX/ECX/EDX, у нас они отразятся в массиве b[]. Затем, мы имеем указатель на структуру CPUID_1_EAX, и мы указываем его на значение EAX из массива b[].

Иными словами, мы трактуем 32-битный *int* как структуру.

Затем мы читаем из структуры.

Компилируем в MSVC 2008 с опцией /Ox:

Листинг 18.19: Оптимизирующий MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
```

```
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
    and    edx, 15
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 16
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call   _printf

    shr    esi, 20
    and    esi, 255
    push   esi
    push   OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call   _printf
    add    esp, 48
    pop    esi

    xor    eax, eax
    pop    ebx

    add    esp, 16
    ret    0
_main    ENDP
```

Инструкция `SHR` сдвигает значение из `EAX` на то количество бит, которое нужно *пропустить*, то есть, мы игнорируем некоторые биты *справа*.

А инструкция `AND` очищает биты *слева* которые нам не нужны, или же, говоря иначе, она оставляет по маске только те биты в `EAX`, которые нам сейчас нужны.

Попробуем GCC 4.4.1 с опцией `-O3`.

Листинг 18.20: Оптимизирующий GCC 4.4.1

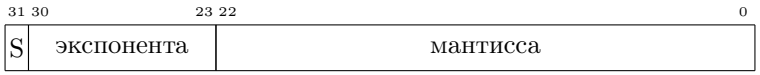
```
main            proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ; "extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     [esp+8], esi
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
    call    ___printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

Практически, то же самое. Единственное что стоит отметить это то, что GCC решил зачем-то объединить

вычисление `extended_model_id` и `extended_family_id` в один блок, вместо того чтобы вычислять их перед соответствующим вызовом `printf()`.

18.6.2 Работа с типом float как со структурой

Как уже раннее указывалось в секции о FPU (15), и *float* и *double* содержат в себе знак, мантиссу и экспоненту. Однако, можем ли мы работать с этими полями напрямую? Попробуем с *float*.



(S — знак)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8;  // exponent + 0x3FF
    unsigned int sign : 1;      // sign bit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiple d by 2^n (n here is 2)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

Структура `float_as_struct` занимает в памяти столько же места сколько и *float*, то есть 4 байта или 32 бита.

Далее мы выставляем во входящем значении отрицательный знак, а также прибавляя двойку к экспоненте, мы тем самым умножаем всё значение на 2^2, то есть на 4.

Компилируем в MSVC 2008 без оптимизации:

Листинг 18.21: Неоптимизирующий MSVC 2008

```
_t$ = -8  ; size = 4
_f$ = -4  ; size = 4
__in$ = 8 ; size = 4
?f@@YAMM@Z PROC  ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8
```

```
    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - выставляем знак минус
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - выкидываем мантиссу
    and    eax, 255         ; 000000ffH - оставляем здесь только экспоненту
    add    eax, 2           ; прибавляем к ней два
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - пододвигаем результат на место бит 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - выкидываем экспоненту

    ; складываем оригинальное значение без экспоненты с новой только что вычисленной ↙
    ↳ экспонентой
    or     ecx, eax
    mov    DWORD PTR _t$[ebp], ecx

    push   4
    lea    edx, DWORD PTR _t$[ebp]
    push   edx
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    call   _memcpy
    add    esp, 12

    fld    DWORD PTR _f$[ebp]

    mov    esp, ebp
    pop    ebp
    ret    0
?f@@YAMM@Z ENDP  ; f
```

Слегка избыточно. В версии скомпилированной с флагом `/Ox` нет вызовов `memcpy()`, там работа происходит сразу с переменной f. Но по неоптимизированной версии будет проще понять.

А что сделает GCC 4.4.1 с опцией `-O3`?

Листинг 18.22: Оптимизирующий GCC 4.4.1

```
; f(float)
      public _Z1ff
_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr  8

      push    ebp
      mov     ebp, esp
      sub     esp, 4
      mov     eax, [ebp+arg_0]
```

```
      or      eax, 80000000h  ; выставить знак '-'
      mov     edx, eax
      and     eax, 807FFFFFh  ; оставить в eax только знак и мантиссу
      shr     edx, 23         ; подготовить экспоненту
      add     edx, 2          ; прибавить 2
      movzx   edx, dl         ; сбросить все биты кроме 7:0 в EAX в 0
      shl     edx, 23         ; подвинуть новую только что вычисленную экспоненту на свое место
      or      eax, edx        ; сложить новую экспоненту и оригинальное значение без экспоненты
      mov     [ebp+var_4], eax
      fld     [ebp+var_4]
      leave
      retn
_Z1ff endp

      public main
main  proc near
      push    ebp
      mov     ebp, esp
      and     esp, 0FFFFFFF0h
      sub     esp, 10h
      fld     ds:dword_8048614 ; -4.936
      fstp    qword ptr [esp+8]
      mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
      mov     dword ptr [esp], 1
      call    ___printf_chk
      xor     eax, eax
      leave
      retn
main  endp
```

Да, функция `f()` в целом понятна. Однако, что интересно, еще при компиляции, не взирая на мешанину с полями структуры, GCC умудрился вычислить результат функции `f(1.234)` и сразу подставить его в аргумент для `printf()`!

Глава 19

Объединения (union)

19.1 Пример генератора случайных чисел

Если нам нужны случайные значения с плавающей запятой в интервале от 0 до 1, самое простое это взять ГПСЧ[1] вроде Mersenne twister выдающий случайные 32-битные числа в виде DWORD, преобразовать это число в *float* и затем разделить на `RAND_MAX` (`0xFFFFFFFF` в данном случае) — полученное число будет в интервале от 0 до 1.

Но как известно, операция деления — это медленная операция. Сможем ли мы избежать её, как в случае с делением через умножение? (14)

Вспомним состав числа с плавающей запятой: это бит знака, биты мантиссы и биты экспоненты. Для получения случайного числа, нам нужно просто заполнить случайными битами все биты мантиссы!

Экспонента не может быть нулевой (иначе число будет денормализованным), так что в эти биты мы запишем `01111111` — это будет означать что экспонента равна единице. Далее заполняем мантиссу случайными битами, знак оставляем в виде 0 (что значит наше число положительное), и вуаля. Генерируемые числа будут в интервале от 1 до 2, так что нам еще нужно будет отнять единицу.

В моем примере[2] применяется очень простой линейный конгруэнтный генератор случайных чисел, выдающий 32-битные числа. Генератор инициализируется текущим временем в стиле UNIX.

Далее, тип *float* представляется в виде *union* — это конструкция Си/Си++ позволяющая интерпретировать часть памти по-разному. В нашем случае, мы можем создать переменную типа `union` и затем обращаться к ней как к *float* или как к *uint32_t*. Можно сказать, что это хак, причем грязный.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

union uint32_t_float
{
    uint32_t i;
    float f;
};

// from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;

int main()
{
    uint32_t_float tmp;

    uint32_t RNG_state=time(NULL); // initial seed
    for (int i=0; i<100; i++)
    {
        RNG_state=RNG_state*RNG_a+RNG_c;
        tmp.i=RNG_state & 0x007fffff | 0x3F800000;
        float x=tmp.f-1;
```

[1] Генератор псевдослучайных чисел

[2] идея взята здесь: http://xor0110.wordpress.com/2010/09/24/how-to-generate-floating-point-random-numbers-efficiently

```
        printf ("%f\n", x);
    };
    return 0;
};
```

Листинг 19.1: MSVC 2010 (/Ox)

```
$SG4232    DB    '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r    ; 1

tv140 = -4                          ; size = 4
_tmp$ = -4                          ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    and    esp, -64                 ; fffffc0H
    sub    esp, 56                  ; 00000038H
    push   esi
    push   edi
    push   0
    call   __time64
    add    esp, 4
    mov    esi, eax
    mov    edi, 100                 ; 00000064H
$LN3@main:

; собственно, генерируем случайное 32-битное число

    imul   esi, 1664525             ; 0019660dH
    add    esi, 1013904223          ; 3c6ef35fH
    mov    eax, esi

; оставляем биты необходимые только для мантиссы

    and    eax, 8388607             ; 007fffffH

; выставляем экспоненту в 1

    or     eax, 1065353216          ; 3f800000H

; записываем это значение как int

    mov    DWORD PTR _tmp$[esp+64], eax
    sub    esp, 8

; загружаем это значение уже как float

    fld    DWORD PTR _tmp$[esp+72]

; отнимаем единицу от него

    fsub   QWORD PTR __real@3ff0000000000000
    fstp   DWORD PTR tv140[esp+72]
    fld    DWORD PTR tv140[esp+72]
    fstp   QWORD PTR [esp]
    push   OFFSET $SG4232
    call   _printf
    add    esp, 12                  ; 0000000cH
    dec    edi
```

```
    jne    SHORT $LN3@main
    pop    edi
    xor    eax, eax
    pop    esi
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

А результат GCC будет почти таким же.

Глава 20

Указатели на функции

Указатель на функцию, в целом, как и любой другой указатель, просто адрес указывающий на начало функции в сегменте кода.

Это применяется часто в т.н. callback-ах [1].

Известные примеры:

- `qsort()`[2], `atexit()`[3] из стандартной библиотеки Си;
- сигналы в *NIX ОС[4];
- запуск тредов: `CreateThread()` (win32), `pthread_create()` (POSIX);
- множество функций win32, например `EnumChildWindows()`[5].

Итак, функция `qsort()` это реализация алгоритма "быстрой сортировки". Функция может сортировать что угодно, любые типы данных, но при условии, что вы имеете функцию сравнения двух элементов данных и `qsort()` может вызывать её.

Эта функция сравнения может определяться так:

```
int (*compare)(const void *, const void *)
```

Воспользуемся немного модифицированным примером, который я нашел вот здесь:

```
1  /* ex3 Sorting ints with qsort */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int comp(const void * _a, const void * _b)
7  {
8    const int *a=(const int *)_a;
9    const int *b=(const int *)_b;
10
11   if (*a==*b)
12     return 0;
13   else
14     if (*a < *b)
15         return -1;
16      else
17       return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22    int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
```

[1] http://en.wikipedia.org/wiki/Callback_(computer_science)
[2] http://en.wikipedia.org/wiki/Qsort_(C_standard_library)
[3] http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html
[4] http://en.wikipedia.org/wiki/Signal.h
[5] http://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx

```
23    int i;
24
25   /* Sort the array */
26   qsort(numbers,10,sizeof(int),comp) ;
27   for (i=0;i<9;i++)
28     printf("Number = %d\n",numbers[ i ]) ;
29   return 0;
30 }
```

20.1 MSVC

Компилируем в MSVC 2010 (я убрал некоторые части для краткости) с опцией `/Ox`:

Листинг 20.1: Оптимизирующий MSVC 2010: /Ox /GS- /MD

```
__a$ = 8                                                ; size = 4
__b$ = 12                                               ; size = 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     ecx, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp   ENDP


_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88         ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; fffe7960H
        call    _qsort
        add     esp, 16                                 ; 00000010H

...
```

Ничего особо удивительного здесь мы не видим. В качестве четвертого аргумента, в `qsort()` просто передается адрес метки `_comp`, где собственно и располагается функция `comp()`.

Как `qsort()` вызывает её?

Посмотрим в MSVCR80.DLL (эта DLL куда в MSVC вынесены функции из стандартных библиотек Си):

Листинг 20.2: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int (__cdecl *)(const ↙
    ↳ void *, const void *))
.text:7816CBF0                 public _qsort
.text:7816CBF0 _qsort          proc near
.text:7816CBF0
.text:7816CBF0 lo              = dword ptr -104h
.text:7816CBF0 hi              = dword ptr -100h
.text:7816CBF0 var_FC          = dword ptr -0FCh
.text:7816CBF0 stkptr          = dword ptr -0F8h
.text:7816CBF0 lostk           = dword ptr -0F4h
.text:7816CBF0 histk           = dword ptr -7Ch
.text:7816CBF0 base            = dword ptr  4
.text:7816CBF0 num             = dword ptr  8
.text:7816CBF0 width           = dword ptr  0Ch
.text:7816CBF0 comp            = dword ptr  10h
.text:7816CBF0
.text:7816CBF0                 sub     esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                           ; CODE XREF: _qsort+B1
.text:7816CCE0                 shr     eax, 1
.text:7816CCE2                 imul    eax, ebp
.text:7816CCE5                 add     eax, ebx
.text:7816CCE7                 mov     edi, eax
.text:7816CCE9                 push    edi
.text:7816CCEA                 push    ebx
.text:7816CCEB                 call    [esp+118h+comp]
.text:7816CCF2                 add     esp, 8
.text:7816CCF5                 test    eax, eax
.text:7816CCF7                 jle     short loc_7816CD04
```

`comp` — это четвертый аргумент функции. Здесь просто передается управление по адресу указанному в `comp`. Перед этим подготавливается два аргумента для функции `comp()`. Далее, проверяется результат её выполнения.

Вот почему использование указателей на функции — это опасно. Во-первых, если вызвать `qsort()` с неправильным указателем на функцию, то `qsort()`, дойдя до этого вызова, может передать управление неизвестно куда, процесс упадет, и эту ошибку можно будет найти не сразу.

Во-вторых, типизация callback-функции должна строго соблюдаться, вызов не той функции с не теми аргументами не того типа, может привести к плачевным результатам, хотя падение процесса это и не проблема, проблема — это найти ошибку, ведь компилятор на стадии компиляции может вас и не предупредить о потенциальных неприятностях.

20.1.1 MSVC + OllyDbg

Загрузим наш пример в OllyDbg и установим брякпойнт на ф-ции `comp()`.

Как значения сравниваются, мы можем увидеть во время самого первого вызова `comp()`: илл. 20.1. Для удобства, OllyDbg показывает сравниваемые значения в окне под окном кода. Мы можем так же увидеть что SP указывает на RA где находится место в ф-ции `qsort()` (на самом деле, находится в `MSVCR100.DLL`).

Трассируя (F8) до инструкции `RETN` и нажав F8 еще один раз, мы возвращаемся в ф-цию `qsort()`: илл. 20.2. Это был вызов ф-ции сравнения.

Вот также скриншот момента второго вызова ф-ции `comp()` — теперь сравниваемые значения другие: илл. 20.3.

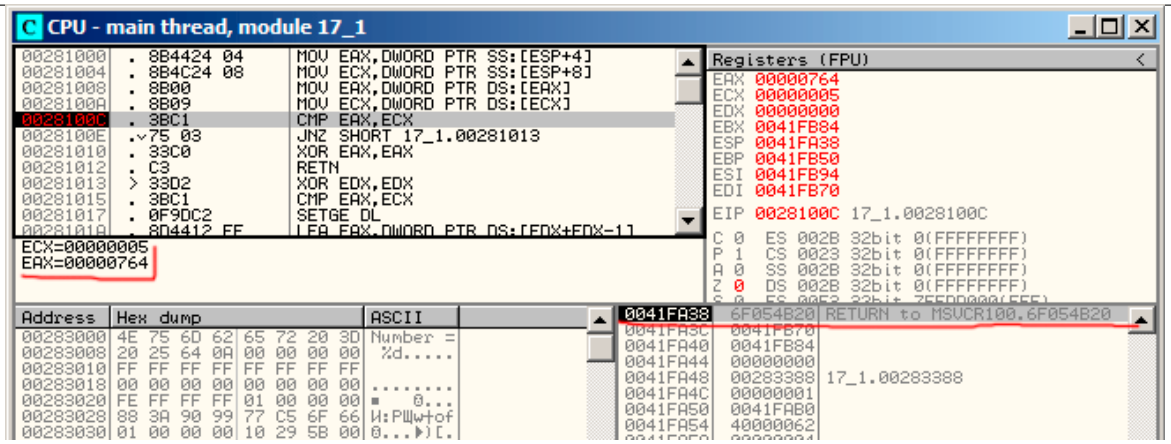


Рис. 20.1: OllyDbg: первый вызов `comp()`

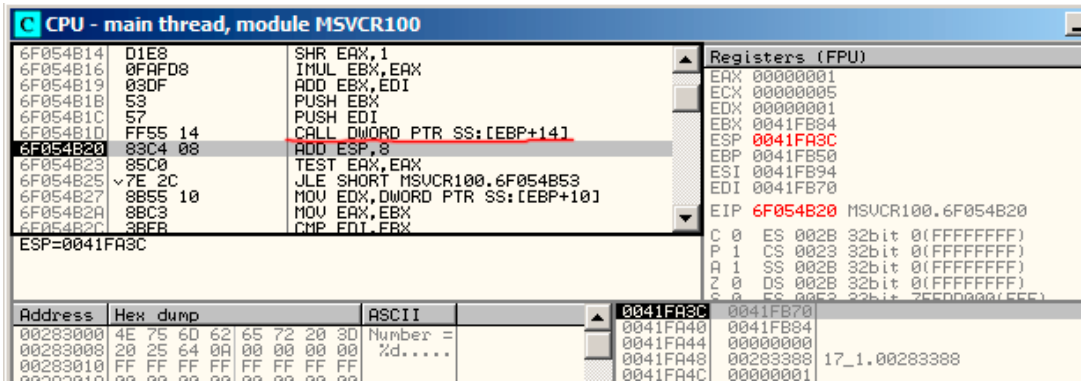


Рис. 20.2: OllyDbg: код в `qsort()` сразу после вызова `comp()`

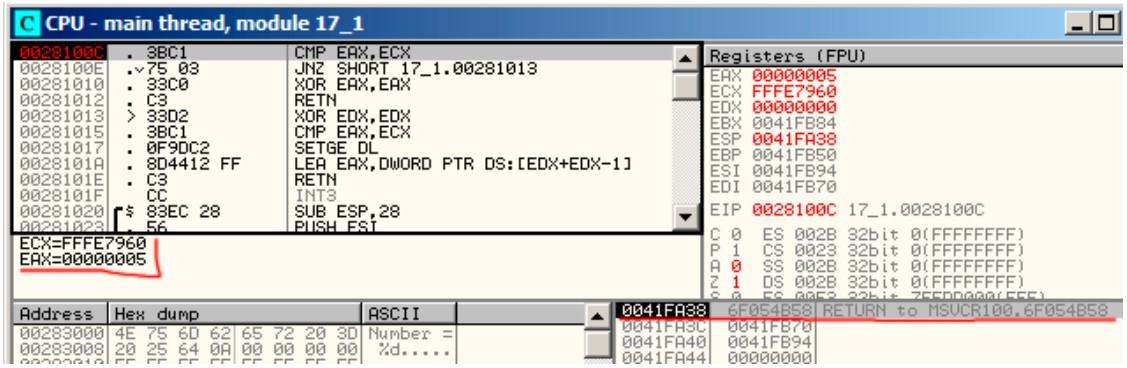


Рис. 20.3: OllyDbg: второй вызов `comp()`

20.1.2 MSVC + tracer

Посмотрим, какие пары сравниваются. Эти 10 чисел будут сортироваться: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Я нашел адрес первой инструкции `CMP` в `comp()`, и это `0x0040100C` и я ставлю брякпойнт на нем:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Получаю информацию о регистрах на брякпойнте:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
```

```
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Я отфильтровал `EAX` и `ECX` и получил:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

Это 34 пары. Следовательно, алгоритму быстрой сортировки нужно 34 операции сравнения для сортировки этих 10-и чисел.

20.1.3 MSVC + tracer (code coverage)

Но можно также и воспользоваться возможностью tracer накапливать все возможные состояния регистров и показать их в IDA.

Трассируем все инструкции в ф-ции `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

Получем .idc-скрипт для загрузки в IDA и загружаем его: илл. 20.4.

Имя этой ф-ции (PtFuncCompare) дала IDA — видимо, потому что видит что указатель на эту ф-цию передается в `qsort()`.

Мы видим что указатели a и b указывают на разные места внутри массива, но шаг между указателями — 4, что логично, ведь в массиве хранятся 32-битные значения.

Видно что инструкции по адресам `0x401010` и `0x401012` никогда не исполнялись (они и остались белыми): действительно, ф-ция `comp()` никогда не возвращала 0, потому что в массиве нет одинаковых элементов.

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008                 mov     eax, [eax]       ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100A                 mov     ecx, [ecx]       ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100C                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                              ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017                 setnl   dl               ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                     ; EAX=1, 0xffffffff
.text:0040101E PtFuncCompare   endp
.text:0040101F
```

Рис. 20.4: tracer и IDA. N.B.: некоторые значения обрезаны справа

20.2 GCC

Не слишком большая разница:

Листинг 20.3: GCC

```
                lea     eax, [esp+40h+var_28]
                mov     [esp+40h+var_40], eax
                mov     [esp+40h+var_28], 764h
                mov     [esp+40h+var_24], 2Dh
                mov     [esp+40h+var_20], 0C8h
                mov     [esp+40h+var_1C], 0FFFFFF9Eh
                mov     [esp+40h+var_18], 0FF7h
                mov     [esp+40h+var_14], 5
                mov     [esp+40h+var_10], 0FFFFCFC7h
                mov     [esp+40h+var_C], 43Fh
                mov     [esp+40h+var_8], 58h
                mov     [esp+40h+var_4], 0FFFE7960h
                mov     [esp+40h+var_34], offset comp
                mov     [esp+40h+var_38], 4
                mov     [esp+40h+var_3C], 0Ah
                call    _qsort
```

Функция `comp()`:

```
                public comp
comp            proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
```

```
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                mov     edx, [eax]
                xor     eax, eax
                cmp     [ecx], edx
                jnz     short loc_8048458
                pop     ebp
                retn
loc_8048458:
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

Реализация `qsort()` находится в `libc.so.6`, и представляет собой просто wrapper [6] для `qsort_r()`.

Она, в свою очередь, вызывает `quicksort()`, где есть вызовы определенной нами функции через переданный указатель:

Листинг 20.4: (файл libc.so.6, версия glibc — 2.10.1)

```
.text:0002DDF6                mov     edx, [ebp+arg_10]
.text:0002DDF9                mov     [esp+4], esi
.text:0002DDFD                mov     [esp], edi
.text:0002DE00                mov     [esp+8], edx
.text:0002DE04                call    [ebp+arg_C]
...
```

20.2.1 GCC + GDB (с исходными кодами)

Очевидно, у нас есть исходный код нашего примера на Си (20), так что мы можем установить брякпойнт (`b`) на номере строки (11-й — это номер строки где происходит первое сравнение). Нам также нужно скомпилировать наш пример с ключом `-g`, чтобы в исполняемом файле была полная отладочная информация. Мы можем так же выводить значения используя имена переменных (`p`): отладочная информация также содержит информацию о том, в каком регистре и/или элементе локального стека находится какая переменная.

Мы можем также увидеть стек (`bt`) и обнаружить что в Glibc используется какая-то вспомогательная ф-ция с именем `msort_with_tmp()`.

Листинг 20.5: GDB-сессия

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out
```

[6] понятие близкое к thunk function

```
Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11        if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11        if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <↙
    ↳ comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

20.2.2 GCC + GDB (без исходных кодов)

Но часто никаких исходных кодов нет вообще, так что мы можем дизассемблировать ф-цию `comp()` (`disas`), найти самую первую инструкцию `CMP` и установить брякпойнт (`b`) по этому адресу. На каждом брякпойнте мы будем видеть содержимое регистров (`info registers`). Информация из стека так же доступна (`bt`), но частичная: здесь нет номеров строк для ф-ции `comp()`.

Листинг 20.6: GDB-сессия

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
   0x08048453 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:     mov    DWORD PTR [ebp-0x8],eax
```

```
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:    mov    edx,DWORD PTR [eax]
   0x08048464 <+23>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:    mov    eax,DWORD PTR [eax]
   0x08048469 <+28>:    cmp    edx,eax
   0x0804846b <+30>:    jne    0x8048474 <comp+39>
   0x0804846d <+32>:    mov    eax,0x0
   0x08048472 <+37>:    jmp    0x804848e <comp+65>
   0x08048474 <+39>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:    mov    edx,DWORD PTR [eax]
   0x08048479 <+44>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47>:    mov    eax,DWORD PTR [eax]
   0x0804847e <+49>:    cmp    edx,eax
   0x08048480 <+51>:    jge    0x8048489 <comp+60>
   0x08048482 <+53>:    mov    eax,0xffffffff
   0x08048487 <+58>:    jmp    0x804848e <comp+65>
   0x08048489 <+60>:    mov    eax,0x1
   0x0804848e <+65>:    leave
   0x0804848f <+66>:    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8       -1073745672
edx            0x764    1892
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff0fc       -1073745668
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104       -1073745660
edx            0xffffff9e       -98
ebx            0xb7fc0000       -1208221696
esp            0xbfffee58       0xbfffee58
ebp            0xbfffee68       0xbfffee68
esi            0xbffff108       -1073745656
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x282    [ SF IF ]
cs             0x73     115
```

```
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e       -98
ecx            0xbffff100       -1073745664
edx            0xc8     200
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff104       -1073745660
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <↙
    ↳ comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp>) at msort.c:307
#9  0x0804850d in main ()
```

Глава 21

64-битные значения в 32-битной среде

В среде, где GPR-ы 32-битные, 64-битные значения передаются как пары 32-битных значений [1].

21.1 Передача аргументов, сложение, вычитание

```
#include <stdint.h>

uint64_t f1 (uint64_t a, uint64_t b)
{
        return a+b;
};

void f1_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f1(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f1(12345678901234, 23456789012345));
#endif
};

uint64_t f2 (uint64_t a, uint64_t b)
{
        return a-b;
};
```

Листинг 21.1: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f1     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f1     ENDP

_f1_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f1
        push    edx
```

[1] Кстати, в 16-битной среде, 32-битные значения передаются 16-битными парами точно так же

```
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f1_test ENDP

_f2     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f2     ENDP
```

В `f1_test()` видно как каждое 64-битное число передается двумя 32-битными значениями, сначала старшая часть, затем младшая.

Сложение и вычитание происходит также парами.

При сложении, в начале складываются младшие 32 бита. Если при сложении был перенос, выставляется флаг CF. Следующая инструкция `ADC` складывает старшие части чисел, но также прибавляет единицу если CF=1.

Вычитание также происходит парами. Первый `SUB` может также включить флаг переноса CF, который затем будет проверяться в `SBB`: если флаг переноса включен, то от результата отнимется единица.

64-битные значения в 32-битной среде возвращаются из ф-ций в паре регистров `EDX:EAX`. Легко увидеть, как результат работы `f1()` затем передается в `printf()`.

Листинг 21.2: GCC 4.8.1 -O1 -fno-inline

```
_f1:
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f1_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f1
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\12\0"
        call    _printf
        add     esp, 28
        ret

_f2:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

Код GCC почти такой же.

21.2 Умножение, деление

```
#include <stdint.h>

uint64_t f3 (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t f4 (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f5 (uint64_t a, uint64_t b)
{
        return a % b;
};
```

Листинг 21.3: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f3     PROC
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __allmul ; long long multiplication
        ret     0
_f3     ENDP

_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f4     PROC
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __aulldiv ; unsigned long long division
        ret     0
_f4     ENDP

_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f5     PROC
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __aullrem ; unsigned long long remainder
        ret     0
_f5     ENDP
```

Умножение и деление это более сложная операция, так что обычно, компилятор встраивает вызовы библиотечных ф-ций, делающих это.

Значение этих библиотечных ф-ций, здесь: E.

Листинг 21.4: GCC 4.8.1 -O3 -fno-inline

```
_f3:
```

```
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f4:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___udivdi3 ; unsigned division
        add     esp, 28
        ret

_f5:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___umoddi3 ; unsigned modulo
        add     esp, 28
        ret
```

GCC делает почти то же самое, тем не менее, встраивает код умножения прямо в ф-цию, посчитав что так будет эффективнее. У GCC другие имена библиотечных ф-ций: D.

21.3 Сдвиг вправо

```
#include <stdint.h>

uint64_t f6 (uint64_t a)
{
        return a>>7;
};
```

Листинг 21.5: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_f6     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
```

```
        shr     edx, 7
        ret     0
_f6     ENDP
```

Листинг 21.6: GCC 4.8.1 -O3 -fno-inline

```
_f6:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
        ret
```

Сдвиг происходит также в две операции: в начале сдвигается младшая часть, затем старшая. Но младшая часть сдвигается при помощи инструкции SHRD, она сдвигает значение в EDX на 7 бит, но подтягивает новые биты из EAX, т.е., из старшей части. Старшая часть сдвигается более известной инструкцией SHR: действительно, ведь освободившиеся биты в старшей части нужно просто заполнить нулями.

21.4 Конвертирование 32-битного значения в 64-битное

```
#include <stdint.h>

int64_t f7 (int64_t a, int64_t b, int32_t c)
{
        return a*b+c;
};

int64_t f7_main ()
{
        return f7(12345678901234, 23456789012345, 12345);
};
```

Листинг 21.7: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_c$ = 24        ; size = 4
_f7     PROC
        push    esi
        push    DWORD PTR _b$[esp+4]
        push    DWORD PTR _b$[esp+4]
        push    DWORD PTR _a$[esp+12]
        push    DWORD PTR _a$[esp+12]
        call    __allmul ; long long multiplication
        mov     ecx, eax
        mov     eax, DWORD PTR _c$[esp]
        mov     esi, edx
        cdq     ; input: 32-bit value in EAX; output: 64-bit value in EDX:EAX
        add     eax, ecx
        adc     edx, esi
        pop     esi
        ret     0
_f7     ENDP

_f7_main PROC
        push    12345           ; 00003039H
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f7
```

```
        add     esp, 20         ; 00000014H
        ret     0
_f7_main ENDP
```

Здесь появляется необходимость расширить 32-битное знаковое значение из *c* в 64-битное знаковое. Конвертировать беззнаковые значения очень просто: нужно просто выставить в 0 все биты в старшей части. Но для знаковых типов это не подходит: знак числа должен быть скопирован в старшую часть числа-результата. Здесь это делает инструкция `CDQ`, она берет входное значение в `EAX`, расширяет число до 64-битного, и оставляет его в паре регистров `EDX:EAX`. Иными словами, инструкция `CDQ` узнает знак числа в `EAX` (просто берет самый старший бит в `EAX`) и в зависимости от этого, выставляет все 32 бита в `EDX` в 0 или в 1. Её работа в каком-то смысле напоминает работу инструкции `MOVSX` (13.1.1).

Листинг 21.8: GCC 4.8.1 -O3 -fno-inline

```
_f7:
        push    edi
        push    esi
        push    ebx
        mov     esi, DWORD PTR [esp+16]
        mov     edi, DWORD PTR [esp+24]
        mov     ebx, DWORD PTR [esp+20]
        mov     ecx, DWORD PTR [esp+28]
        mov     eax, esi
        mul     edi
        imul    ebx, edi
        imul    ecx, esi
        mov     esi, edx
        add     ecx, ebx
        mov     ebx, eax
        mov     eax, DWORD PTR [esp+32]
        add     esi, ecx
        cdq     ; input: 32-bit value in EAX; output: 64-bit value in EDX:EAX
        add     eax, ebx
        adc     edx, esi
        pop     ebx
        pop     esi
        pop     edi
        ret

_f7_main:
        sub     esp, 28
        mov     DWORD PTR [esp+16], 12345       ; 00003039H
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f7
        add     esp, 28
        ret
```

GCC генерирует такой же код как и MSVC, но обходится без вызова библиотечной ф-ции для перемножения значений.

См.также: 32-битные значения в 16-битной среде: 31.4.

Глава 22

SIMD

SIMD[1] это акроним: *Single Instruction, Multiple Data*.

Как можно судить по названию, это обработка множества данных исполняя только одну инструкцию.

Как и FPU, эта подсистема процессора выглядит так же отдельным процессором внутри x86.

SIMD в x86 начался с MMX. Появилось 8 64-битных регистров MM0-MM7.

Каждый MMX-регистр может содержать 2 32-битных значения, 4 16-битных или же 8 байт. Например, складывая значения двух MMX-регистров, можно складывать одновременно 8 8-битных значений.

Простой пример, это некий графический редактор, который хранит открытое изображение как двумерный массив. Когда пользователь меняет яркость изображения, редактору нужно, например, прибавить некий коэффициент ко всем пикселям, или отнять. Для простоты можно представить, что изображение у нас бело-серо-черное и каждый пиксель занимает один байт, то с помощью MMX можно менять яркость сразу у восьми пикселей.

Когда MMX только появилось, эти регистры на самом деле располагались в FPU-регистрах. Можно было использовать либо FPU либо MMX в одно и то же время. Можно подумать, что Intel решило немного сэкономить на транзисторах, но на самом деле причина такого симбиоза проще — более старая ОС не знающая о дополнительных регистрах процессора не будет сохранять их во время переключения задач, а вот регистры FPU сохранять будет. Таким образом, процессор с MMX + старая ОС + задача использующая возможности MMX = все это может работать вместе.

SSE — это расширение регистров до 128 бит, теперь уже отдельно от FPU.

AVX — расширение регистров до 256 бит.

Немного о практическом применении.

Конечно же, копирование блоков в памяти (`memcpy`), сравнение (`memcmp`), и подобное.

Еще пример: имеется алгоритм шифрования DES, который берет 64-битный блок, 56-битный ключ, шифрует блок с ключом и образуется 64-битный результат. Алгоритм DES можно легко представить в виде очень большой электронной цифровой схемы, с проводами, элементами И, ИЛИ, НЕ.

Идея bitslice DES[2] — это обработка сразу группы блоков и ключей одновременно. Скажем, на x86 переменная типа *unsigned int* вмещает в себе 32 бита, так что там можно хранить промежуточные результаты сразу для 32-х блоков-ключей, используя 64+56 переменных типа *unsigned int*.

Я написал утилиту для перебора паролей/хешей Oracle RDBMS (которые основаны на алгоритме DES), переделав алгоритм bitslice DES для SSE2 и AVX — и теперь возможно шифровать одновременно 128 или 256 блоков-ключей:

http://conus.info/utils/ops_SIMD/

22.1 Векторизация

Векторизация[3] это когда у вас есть цикл, который берет на вход несколько массивов и выдает, например, один массив данных. Тело цикла берет некоторые элементы из входных массивов, что-то делает с ними и помещает в выходной. Важно, что операция применяемая ко всем элементам одна и та же. Векторизация — это обрабатывать несколько элементов одновременно.

Векторизация — это не самая новая технология: автор сих строк видел её по крайней мере на линейке суперкомпьютеров Cray Y-MP от 1988, когда работал на его версии-“лайт” Cray Y-MP EL [4].

Например:

[1] Single instruction, multiple data

[2] http://www.darkside.com.au/bitslice/

[3] Wikipedia: vectorization

[4] Удаленно. Он находится в музее суперкомпьютеров: http://www.cray-cyber.org

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Этот фрагмент кода берет элементы из A и B, перемножает и сохраняет результат в C.

Если представить, что каждый элемент массива — это 32-битный *int*, то их можно загружать сразу по 4 из A в 128-битный XMM-регистр, из B в другой XMM-регистр и выполнив инструкцию *PMULLD* (*Перемножить запакованные знаковые DWORD и сохранить младшую часть результата*) и PMULHW (*Перемножить запакованные знаковые DWORD и сохранить старшую часть результата*), можно получить 4 64-битных произведения сразу.

Таким образом, тело цикла исполняется 1024/4 раза вместо 1024, что в 4 раза меньше, и, конечно, быстрее.

Некоторые компиляторы умеют делать автоматическую векторизацию в простых случаях, например Intel C++[5].

Я написал очень простую функцию:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];

        return 0;
};
```

22.1.1 Intel C++

Компилирую при помощи Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Имеем такое (в IDA):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr  0Ch
ar3    = dword ptr  10h

        push    edi
        push    esi
        push    ebx
        push    esi
        mov     edx, [esp+10h+sz]
        test    edx, edx
        jle     loc_15B
        mov     eax, [esp+10h+ar3]
        cmp     edx, 6
        jle     loc_143
        cmp     eax, [esp+10h+ar2]
        jbe     short loc_36
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        neg     esi
        cmp     ecx, esi
        jbe     short loc_55
```

[5]Еще о том, как Intel C++ умеет автоматически векторизовать циклы: Excerpt: Effective Automatic Vectorization

```
loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax        ; edi = ar1
        and     edi, 0Fh        ; is ar1 16-byte aligned?
        jz      short loc_9A    ; yes
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
        mov     edx, [ecx+esi*4]
        add     edx, [ebx+esi*4]
        mov     [eax+esi*4], edx
        inc     esi
        cmp     esi, edi
        jb      short loc_C1
        mov     ecx, [esp+10h+var_10]
```

```
        mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
        mov     esi, [esp+10h+ar2]
        lea     esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
        test    esi, 0Fh
        jz      short loc_109   ; yes!
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
        movdqu  xmm1, xmmword ptr [ebx+edi*4]
        movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to ↙
    ↳ xmm0
        paddd   xmm1, xmm0
        movdqa  xmmword ptr [eax+edi*4], xmm1
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_ED
        jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
        movdqu  xmm0, xmmword ptr [ebx+edi*4]
        paddd   xmm0, xmmword ptr [esi+edi*4]
        movdqa  xmmword ptr [eax+edi*4], xmm0
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
          ; f(int,int *,int *,int *)+164
        cmp     ecx, edx
        jnb     short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp     short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
          ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
```

```
        jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
         ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
        pop     esi
        pop     edi
        retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
         ; f(int,int *,int *,int *)+9F
        xor     ecx, ecx
        jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

Инструкции, имеющие отношение к SSE2 это:

- `MOVDQU` (*Move Unaligned Double Quadword*) — она просто загружает 16 байт из памяти в XMM-регистр.
- `PADDD` (*Add Packed Integers*) — складывает сразу 4 пары 32-битных чисел и оставляет в первом операнде результат. Кстати, если произойдет переполнение, то исключения не произойдет и никакие флаги не установятся, запишутся просто младшие 32 бита результата. Если один из операндов `PADDD` — адрес значения в памяти, то требуется чтобы адрес был выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение [6].
- `MOVDQA` (*Move Aligned Double Quadword*) — тоже что и `MOVDQU`, только подразумевает что адрес в памяти выровнен по 16-байтной границе. Если он не выровнен, произойдет исключение. `MOVDQA` работает быстрее чем `MOVDQU`, но требует вышеозначенного.

Итак, эти SSE2-инструкции исполнятся только в том случае если еще осталось просуммировать 4 пары переменных типа *int* плюс если указатель `ar3` выровнен по 16-байтной границе.

Более того, если еще и `ar2` выровнен по 16-байтной границе, то будет выполняться этот фрагмент кода:

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

А иначе, значение из `ar2` загрузится в `XMM0` используя инструкцию `MOVDQU`, которая не требует выровненного указателя, зато может работать чуть медленнее:

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to xmm0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

А во всех остальных случаях, будет исполняться код, который был бы, как если бы не была включена поддержка SSE2.

22.1.2 GCC

Но и GCC умеет кое-что векторизировать[7], если компилировать с опциями `-O3` и включить поддержку SSE2: `-msse2`.

Вот что вышло (GCC 4.4.1):

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
```

[6] О выравнивании данных см. также: Wikipedia: Выравнивание данных

[7] Подробнее о векторизации в GCC: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1
```

```
loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Почти то же самое, хотя и не так дотошно как Intel C++.

22.2 Реализация strlen() при помощи SIMD

Прежде всего, следует заметить, что SIMD-инструкции можно вставлять в Си/Си++ код при помощи специальных макросов[8]. В MSVC, часть находится в файле intrin.h.

Имеется возможность реализовать функцию strlen()[9] при помощи SIMD-инструкций, работающий в 2-2.5 раза быстрее обычной реализации. Эта функция будет загружать в XMM-регистр сразу 16 байт и проверять каждый на ноль.

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

(пример базируется на исходнике отсюда).

Компилируем в MSVC 2010 с опцией /Ox:

```
_pos$75552 = -4          ; size = 4
_str$ = 8                ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
    and      esp, -16            ; fffffff0H
    mov      eax, DWORD PTR _str$[ebp]
    sub      esp, 12             ; 0000000cH
    push     esi
    mov      esi, eax
    and      esi, -16            ; fffffff0H
    xor      edx, edx
    mov      ecx, eax
    cmp      esi, eax
    je       SHORT $LN4@strlen_sse
    lea      edx, DWORD PTR [eax+1]
```

[8] MSDN: MMX, SSE, and SSE2 Intrinsics

[9] strlen() — стандартная функция Си для подсчета длины строки

```
    npad    3
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test    eax, eax
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16                     ; 00000010H
    pcmpeqb xmm1, xmm0
    add     edx, 16                     ; 00000010H
    pmovmskb eax, xmm1
    test    eax, eax
    je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf     eax, eax
    mov     ecx, eax
    mov     DWORD PTR _pos$75552[esp+16], eax
    lea     eax, DWORD PTR [ecx+edx]
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
?strlen_sse2@@YAIPBD@Z ENDP                 ; strlen_sse2
```

Итак, прежде всего, мы проверяем указатель `str`, выровнен ли он по 16-байтной границе. Если нет, то мы вызовем обычную реализацию `strlen()`.

Далее мы загружаем по 16 байт в регистр `XMM1` при помощи команды `MOVDQA`.

Наблюдательный читатель может спросить, почему в этом месте мы не можем использовать `MOVDQU`, которая может загружать откуда угодно не взирая на факт, выровнен ли указатель?

Да, можно было бы сделать вот как: если указатель выровнен, загружаем используя `MOVDQA`, иначе используем работающую чуть медленнее `MOVDQU`.

Однако здесь кроется не сразу заметная проблема, которая проявляется вот в чем:

В ОС линии Windows NT, и не только, память выделяется страницами по 4 KiB (4096 байт). Каждый win32-процесс якобы имеет в наличии 4 GiB, но на самом деле, только некоторые части этого адресного пространства присоединены к реальной физической памяти. Если процесс обратится к блоку памяти, которого не существует, сработает исключение. Так работает виртуальная память[10].

Так вот, функция, читающая сразу по 16 байт, имеет возможность нечаянно вылезти за границу выделенного блока памяти. Предположим, ОС выделила программе 8192 (0x2000) байт по адресу 0x008c0000. Таким образом, блок занимает байты с адреса 0x008c0000 по 0x008c1fff включительно.

За этим блоком, то есть начиная с адреса 0x008c2000 нет вообще ничего, т.е., ОС не выделяла там память. Обращение к памяти начиная с этого адреса вызовет исключение.

И предположим, что программа хранит некую строку из, скажем, пяти символов почти в самом конце блока, что не является преступлением:

[10] http://en.wikipedia.org/wiki/Page_(computer_memory)

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	здесь случайный мусор
0x008c1fff	здесь случайный мусор

В обычных условиях, программа вызывает `strlen()` передав ей указатель на строку `'hello'` лежащую по адресу 0x008c1ff8. `strlen()` будет читать по одному байту до 0x008c1ffd, где ноль, и здесь она закончит работу.

Теперь, если мы напишем свою реализацию `strlen()` читающую сразу по 16 байт, с любого адреса, будь он выровнен по 16-байтной границе или нет, `MOVDQU` попытается загрузить 16 байт с адреса 0x008c1ff8 по 0x008c2008, и произойдет исключение. Это ситуация которой, конечно, хочется избежать.

Поэтому мы будем работать только с адресами, выровненными по 16 байт, что в сочетании со знанием что размер страницы ОС также, как правило, выровнен по 16 байт, даст некоторую гарантию что наша функция не будет пытаться читать из мест в невыделенной памяти.

Вернемся к нашей функции.

`_mm_setzero_si128()` — это макрос, генерирующий `pxor xmm0, xmm0` — инструкция просто обнуляет регистр `XMM0`.

`_mm_load_si128()` — это макрос для `MOVDQA`, он просто загружает 16 байт по адресу из указателя в `XMM1`.

`_mm_cmpeq_epi8()` — это макрос для `PCMPEQB`, это инструкция которая побайтово сравнивает значения из двух XMM регистров.

И если какой-то из байт равен другому, то в результирующем значении будет выставлено на месте этого байта `0xff`, либо 0, если байты не были равны.

Например.

```
XMM1: 11223344556677880000000000000000
XMM0: 11ab3444007877881111111111111111
```

После исполнения `pcmpeqb xmm1, xmm0`, регистр `XMM1` будет содержать:

```
XMM1: ff0000ff0000ffff0000000000000000
```

Эта инструкция в нашем случае, сравнивает каждый 16-байтный блок с блоком состоящим из 16-и нулевых байт, выставленным в `XMM0` при помощи `pxor xmm0, xmm0`.

Следующий макрос `_mm_movemask_epi8()` — это инструкция `PMOVMSKB`.

Она очень удобна как раз для использования в паре с `PCMPEQB`.

`pmovmskb eax, xmm1`

Эта инструкция выставит самый первый бит `EAX` в единицу, если старший бит первого байта в регистре `XMM1` является единицей. Иными словами, если первый байт в регистре `XMM1` является `0xff`, то первый бит в `EAX` будет также единицей, иначе нулем.

Если второй байт в регистре `XMM1` является `0xff`, то второй бит в `EAX` также будет единицей. Иными словами, инструкция отвечает на вопрос, *какие из байт в `XMM1` являются `0xff`?* В результате приготовит 16 бит и запишет в `EAX`. Остальные биты в `EAX` обнулятся.

Кстати, не забывайте также вот о какой особенности нашего алгоритма:

На вход может прийти 16 байт вроде `hello\x00garbage\x00ab`

Это строка `'hello'`, после нее терминирующий ноль, затем немного мусора в памяти.

Если мы загрузим эти 16 байт в `XMM1` и сравним с нулевым `XMM0`, то в итоге получим такое (я использую здесь порядок с MSB[11] до LSB[12]):

```
XMM1: 0000ff00000000000000ff0000000000
```

Это означает что инструкция сравнения обнаружила два нулевых байта, что и не удивительно.

`PMOVMSKB` в нашем случае подготовит `EAX` вот так (в двоичном представлении): *0010000000100000b*.

Совершенно очевидно, что далее наша функция должна учитывать только первый встретившийся нулевой бит и игнорировать все остальное.

Следующая инструкция — `BSF` (*Bit Scan Forward*). Это инструкция находит самый младший бит во втором операнде и записывает его позицию в первый операнд.

```
EAX=0010000000100000b
```

[11] most significant bit
[12] least significant bit

После исполнения этой инструкции `bsf eax, eax`, в `EAX` будет 5, что означает, что единица найдена в пятой позиции (считая с нуля).

Для использования этой инструкции, в MSVC также имеется макрос `_BitScanForward`.

А дальше все просто. Если нулевой байт найден, его позиция прибавляется к тому что мы уже насчитали и возвращается результат.

Почти всё.

Кстати, следует также отметить, что компилятор MSVC сгенерировал два тела цикла сразу, для оптимизации.

Кстати, в SSE 4.2 (который появился в Intel Core i7) все эти манипуляции со строками могут быть еще проще: http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

Глава 23

64 бита

23.1 x86-64

Это расширение x86-архитуктуры до 64 бит.

С точки зрения начинающего reverse engineer-а, наиболее важные отличия от 32-битного x86 это:

- Почти все регистры (кроме FPU и SIMD) расширены до 64-бит и получили префикс r-. И еще 8 регистров добавлено. В итоге имеются эти GPR-ы: `RAX`, `RBX`, `RCX`, `RDX`, `RBP`, `RSP`, `RSI`, `RDI`, `R8`, `R9`, `R10`, `R11`, `R12`, `R13`, `R14`, `R15`.

  К ним также можно обращаться так же, как и прежде. Например, для доступа к младшим 32 битам `RAX` можно использовать `EAX`.

  У новых регистров `r8-r15` также имеются их *младшие части*: `r8d-r15d` (младшие 32-битные части), `r8w-r15w` (младшие 16-битные части), `r8b-r15b` (младшие 8-битные части).

  Удвоено количество SIMD-регистров: с 8 до 16: `XMM0-XMM15`.

- В win64 передача всех параметров немного иная, это немного похоже на fastcall (44.3). Первые 4 аргумента записываются в регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт чтобы вызываемая функция могла сохранить там первые 4 аргумента и использовать эти регистры по своему усмотрению. Короткие функции могут использовать аргументы прямо из регистров, но бóльшие функции могут сохранять их значения на будущее.

  Соглашение System V AMD64 ABI (Linux, *BSD, Mac OS X) [21] также напоминает fastcall, использует 6 регистров `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` для первых шести аргументов. Остальные передаются через стек.

  См. также в соответствующем разделе о способах передачи аргументов через стек (44).

- Сишный *int* остается 32-битным для совместимости.

- Все указатели теперь 64-битные.

  На это иногда сетуют: ведь теперь для хранения всех указателей нужно в 2 раза больше места в памяти, в т.ч. и в кэш-памяти, не смотря на то что x64-процессоры адресуют только 48 бит внешней RAM[1].

Из-за того, что регистров общего пользования теперь вдвое больше, у компиляторов теперь больше свободного места для маневра называемого register allocation. Для нас это означает, что в итоговом коде будет меньше локальных переменных.

Для примера, функция вычисляющая первый S-бокс алгоритма шифрования DES, она обрабатывает сразу 32/64/128/256 значений, в зависимости от типа `DES_type` (uint32, uint64, SSE2 или AVX), методом bitslice DES (больше об этом методе читайте здесь (22)):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */
```

[1] Random-access memory

```
#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
```

```
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

Здесь много локальных переменных. Конечно, далеко не все они будут в локальном стеке. Компилируем обычным MSVC 2008 с опцией `/Ox`:

Листинг 23.1: Оптимизирующий MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32         ; size = 4
_x24$ = 36          ; size = 4
_out2$ = 36         ; size = 4
_out3$ = 40         ; size = 4
_out4$ = 44         ; size = 4
_s1    PROC
    sub    esp, 20                    ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
```

```
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    and    ecx, ebx
    mov    DWORD PTR _x1$[esp+36], eax
    xor    eax, ebx
    mov    esi, ebp
    or     esi, edx
    mov    DWORD PTR _x4$[esp+36], esi
    and    esi, DWORD PTR _a6$[esp+32]
    mov    DWORD PTR _x7$[esp+32], ecx
    mov    edx, esi
    xor    edx, eax
    mov    DWORD PTR _x6$[esp+36], edx
    mov    edx, DWORD PTR _a3$[esp+32]
    xor    edx, ebx
    mov    ebx, esi
    xor    ebx, DWORD PTR _a5$[esp+32]
    mov    DWORD PTR _x8$[esp+36], edx
    and    ebx, edx
    mov    ecx, edx
    mov    edx, ebx
    xor    edx, ebp
    or     edx, DWORD PTR _a6$[esp+32]
    not    ecx
    and    ecx, DWORD PTR _a6$[esp+32]
    xor    edx, edi
    mov    edi, edx
    or     edi, DWORD PTR _a2$[esp+32]
    mov    DWORD PTR _x3$[esp+36], ebp
    mov    ebp, DWORD PTR _a2$[esp+32]
    xor    edi, ebx
    and    edi, DWORD PTR _a1$[esp+32]
    mov    ebx, ecx
    xor    ebx, DWORD PTR _x7$[esp+32]
    not    edi
    or     ebx, ebp
    xor    edi, ebx
    mov    ebx, edi
    mov    edi, DWORD PTR _out2$[esp+32]
    xor    ebx, DWORD PTR [edi]
    not    eax
    xor    ebx, DWORD PTR _x6$[esp+36]
    and    eax, edx
    mov    DWORD PTR [edi], ebx
    mov    ebx, DWORD PTR _x7$[esp+32]
    or     ebx, DWORD PTR _x6$[esp+36]
    mov    edi, esi
    or     edi, DWORD PTR _x1$[esp+36]
    mov    DWORD PTR _x28$[esp+32], ebx
```

```
    xor    edi, DWORD PTR _x8$[esp+36]
    mov    DWORD PTR _x24$[esp+32], edi
    xor    edi, ecx
    not    edi
    and    edi, edx
    mov    ebx, edi
    and    ebx, ebp
    xor    ebx, DWORD PTR _x28$[esp+32]
    xor    ebx, eax
    not    eax
    mov    DWORD PTR _x33$[esp+32], ebx
    and    ebx, DWORD PTR _a1$[esp+32]
    and    eax, ebp
    xor    eax, ebx
    mov    ebx, DWORD PTR _out4$[esp+32]
    xor    eax, DWORD PTR [ebx]
    xor    eax, DWORD PTR _x24$[esp+32]
    mov    DWORD PTR [ebx], eax
    mov    eax, DWORD PTR _x28$[esp+32]
    and    eax, DWORD PTR _a3$[esp+32]
    mov    ebx, DWORD PTR _x3$[esp+36]
    or     edi, DWORD PTR _a3$[esp+32]
    mov    DWORD PTR _x36$[esp+32], eax
    not    eax
    and    eax, edx
    or     ebx, ebp
    xor    ebx, eax
    not    eax
    and    eax, DWORD PTR _x24$[esp+32]
    not    ebp
    or     eax, DWORD PTR _x3$[esp+36]
    not    esi
    and    ebp, eax
    or     eax, edx
    xor    eax, DWORD PTR _a5$[esp+32]
    mov    edx, DWORD PTR _x36$[esp+32]
    xor    edx, DWORD PTR _x4$[esp+36]
    xor    ebp, edi
    mov    edi, DWORD PTR _out1$[esp+32]
    not    eax
    and    eax, DWORD PTR _a2$[esp+32]
    not    ebp
    and    ebp, DWORD PTR _a1$[esp+32]
    and    edx, esi
    xor    eax, edx
    or     eax, DWORD PTR _a1$[esp+32]
    not    ebp
    xor    ebp, DWORD PTR [edi]
    not    ecx
    and    ecx, DWORD PTR _x33$[esp+32]
    xor    ebp, ebx
    not    eax
    mov    DWORD PTR [edi], ebp
    xor    eax, ecx
    mov    ecx, DWORD PTR _out3$[esp+32]
    xor    eax, DWORD PTR [ecx]
    pop    edi
    pop    esi
    xor    eax, ebx
    pop    ebp
    mov    DWORD PTR [ecx], eax
```

```
    pop    ebx
    add    esp, 20                    ; 00000014H
    ret    0
_s1    ENDP
```

5 переменных компилятору пришлось разместить в локальном стеке.

Теперь попробуем то же самое только в 64-битной версии MSVC 2008:

Листинг 23.2: Оптимизирующий MSVC 2008

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov    QWORD PTR [rsp+24], rbx
    mov    QWORD PTR [rsp+32], rbp
    mov    QWORD PTR [rsp+16], rdx
    mov    QWORD PTR [rsp+8], rcx
    push   rsi
    push   rdi
    push   r12
    push   r13
    push   r14
    push   r15
    mov    r15, QWORD PTR a5$[rsp]
    mov    rcx, QWORD PTR a6$[rsp]
    mov    rbp, r8
    mov    r10, r9
    mov    rax, r15
    mov    rdx, rbp
    not    rax
    xor    rdx, r9
    not    r10
    mov    r11, rax
    and    rax, r9
    mov    rsi, r10
    mov    QWORD PTR x36$1$[rsp], rax
    and    r11, r8
    and    rsi, r8
    and    r10, r15
    mov    r13, rdx
    mov    rbx, r11
    xor    rbx, r9
    mov    r9, QWORD PTR a2$[rsp]
    mov    r12, rsi
    or     r12, r15
    not    r13
    and    r13, rcx
    mov    r14, r12
    and    r14, rcx
    mov    rax, r14
    mov    r8, r14
    xor    r8, rbx
```

```
    xor    rax, r15
    not    rbx
    and    rax, rdx
    mov    rdi, rax
    xor    rdi, rsi
    or     rdi, rcx
    xor    rdi, r10
    and    rbx, rdi
    mov    rcx, rdi
    or     rcx, r9
    xor    rcx, rax
    mov    rax, r13
    xor    rax, QWORD PTR x36$1$[rsp]
    and    rcx, QWORD PTR a1$[rsp]
    or     rax, r9
    not    rcx
    xor    rcx, rax
    mov    rax, QWORD PTR out2$[rsp]
    xor    rcx, QWORD PTR [rax]
    xor    rcx, r8
    mov    QWORD PTR [rax], rcx
    mov    rax, QWORD PTR x36$1$[rsp]
    mov    rcx, r14
    or     rax, r8
    or     rcx, r11
    mov    r11, r9
    xor    rcx, rdx
    mov    QWORD PTR x36$1$[rsp], rax
    mov    r8, rsi
    mov    rdx, rcx
    xor    rdx, r13
    not    rdx
    and    rdx, rdi
    mov    r10, rdx
    and    r10, r9
    xor    r10, rax
    xor    r10, rbx
    not    rbx
    and    rbx, r9
    mov    rax, r10
    and    rax, QWORD PTR a1$[rsp]
    xor    rbx, rax
    mov    rax, QWORD PTR out4$[rsp]
    xor    rbx, QWORD PTR [rax]
    xor    rbx, rcx
    mov    QWORD PTR [rax], rbx
    mov    rbx, QWORD PTR x36$1$[rsp]
    and    rbx, rbp
    mov    r9, rbx
    not    r9
    and    r9, rdi
    or     r8, r11
    mov    rax, QWORD PTR out1$[rsp]
    xor    r8, r9
    not    r9
    and    r9, rcx
    or     rdx, rbp
    mov    rbp, QWORD PTR [rsp+80]
    or     r9, rsi
    xor    rbx, r12
    mov    rcx, r11
```

```
    not    rcx
    not    r14
    not    r13
    and    rcx, r9
    or     r9, rdi
    and    rbx, r14
    xor    r9, r15
    xor    rcx, rdx
    mov    rdx, QWORD PTR a1$[rsp]
    not    r9
    not    rcx
    and    r13, r10
    and    r9, r11
    and    rcx, rdx
    xor    r9, rbx
    mov    rbx, QWORD PTR [rsp+72]
    not    rcx
    xor    rcx, QWORD PTR [rax]
    or     r9, rdx
    not    r9
    xor    rcx, r8
    mov    QWORD PTR [rax], rcx
    mov    rax, QWORD PTR out3$[rsp]
    xor    r9, r13
    xor    r9, QWORD PTR [rax]
    xor    r9, r8
    mov    QWORD PTR [rax], r9
    pop    r15
    pop    r14
    pop    r13
    pop    r12
    pop    rdi
    pop    rsi
    ret    0
s1    ENDP
```

Компилятор ничего не выделил в локальном стеке, а `x36` это синоним для `a5`.

Кстати, видно, что функция сохраняет регистры `RCX`, `RDX` в отведенных для этого вызываемой функцией местах, а `R8` и `R9` не сохраняет, а начинает использовать их сразу.

Кстати, существуют процессоры с еще большим количеством GPR, например, Itanium — 128 регистров.

23.2 ARM

64-битные инструкции в ARM появились в ARMv8.

23.3 Числа с плавающей запятой

О том как происходит работа с числами с плавающей запятой в x86-64, читайте здесь: 24.

Глава 24

Работа с числами с плавающей запятой в x64 используя SIMD

Разумеется, FPU остался в x86-совместимых процессорах в то время, когда ввели расширение x64. Но в то время уже везде были SIMD-расширения (SSE[1], SSE2, итд), которые, к тому же, тоже умеют работать с числами с плавающей запятой. Формат чисел остается тот же (IEEE 754).

Так что, компиляторы для x86-64 обычно используют SIMD-инструкции. Это, можно сказать, хорошая новость, потому что работать с ними легче.

Примеры будем использовать из секции о FPU 15.

24.1 Простой пример

```
double f (double a, double b)
{
        return a/3.14 + b*4.1;
};
```

Листинг 24.1: MSVC 2012 x64 /Ox

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Собственно, входные значения с плавающей запятой передаются через регистры `XMM0-XMM3`, а остальные — через стек [2].

a передается через `XMM0`, *b* — через `XMM1`. Но XMM-регистры (как мы уже знаем из секции о SIMD22) 128-битные, а значения типа *double*— 64-битные, так что используется только младшая половина регистра.

`DIVSD` это SSE-инструкия, означает “Divide Scalar Double-Precision Floating-Point Values”, и просто делит значение типа *double* на другое, лежащие в младших половинах операндов.

Константы закодированы компилятором в формате IEEE 754.

`MULSD` и `ADDSD` работают так же, только производят умножение и сложение.

Результат работы ф-ции типа *double* ф-ция оставляет в регистре `XMM0`.

Как работает неоптимизирующий MSVC:

[1] Streaming SIMD Extensions
[2] MSDN: Parameter Passing

Листинг 24.2: MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Чуть более избыточно. Входные аргументы сохраняются в "shadow space" (7.2.1), причем, только младшие половины регистров, т.е., только 64-битные значения типа *double*.

Результат работы компилятора GCC точно такой же.

24.2 Передача чисел с плавающей запятой в аргументах

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

Они передаются в младших половинах регистров XMM0-XMM3.

Листинг 24.3: MSVC 2012 x64 /Ox

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r   ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r   ; 1.54

main    PROC
        sub     rsp, 40                                 ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Инструкции MOVSDX нет в документации от Intel [14] и AMD [1], там она называется просто MOVSD. Таким образом, в процессорах x86 две инструкции с одинаковым именем (о второй: B.6.2). Возможно, в Microsoft решили избежать путанницы и переименовали инструкцию в MOVSDX. Она просто загружает значение в младшую половину XMM-регистра.

Ф-ция pow() берет аргументы из XMM0 и XMM1, и возвращает результат в XMM0. Далее он перекладывается в RDX для printf(). Почему? Честно говоря, не знаю, может быть, это потому что printf() — ф-ция с переменным количеством аргументов?

Листинг 24.4: GCC 4.4.6 x64 -O3

```
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
main:
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
        call    pow
        ; результат сейчас в XMM0
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 1 ; количество переданных векторных регистров
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
.LC0:
        .long   171798692
        .long   1073259479
.LC1:
        .long   2920577761
        .long   1077936455
```

GCC работает понятнее. Значение для printf() передается в XMM0. Кстати, вот тот случай, когда в EAX для printf() записывается 1 — это значит, что будет передан один аргумент в векторных регистрах, так того требует стандарт [21].

24.3 Пример с сравнением

```
double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};
```

Листинг 24.5: MSVC 2012 x64 /Ox

```
a$ = 8
b$ = 16
d_max   PROC
        comisd  xmm0, xmm1
        ja      SHORT $LN2@d_max
        movaps  xmm0, xmm1
$LN2@d_max:
        fatret  0
d_max   ENDP
```

Оптимизирующий MSVC генерирует очень понятный код.

Инструкция COMISD это “Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”. Собственно, это она и делает.

Неоптимизирующий MSVC генерирует более избыточно, но тоже всё понятно:

Листинг 24.6: MSVC 2012 x64

```
a$ = 8
b$ = 16
```

```
d_max   PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max:
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret  0
d_max   ENDP
```

А вот GCC 4.4.6 дошел в оптимизации дальше и применил инструкцию `MAXSD` ("Return Maximum Scalar Double-Precision Floating-Point Value"), которая просто выбирает максимальное значение!

Листинг 24.7: GCC 4.4.6 x64 -O3

```
d_max:
        maxsd   xmm0, xmm1
        ret
```

24.4 Краткий итог

Во всех приведенных примерах, в XMM-регистрах используется только младшая половина регистра, там хранится значение в формате IEEE 754.

Собственно, все инструкции с суффиксом `-SD` ("Scalar Double-Precision") — это инструкции для работы с числами с плавающей запятой в формате IEEE 754, хранящиеся в младшей 64-битной половине XMM-регистра.

Всё удобнее чем это было в FPU, видимо, сказывается тот факт что расширения SIMD развивались не так хаотично как FPU в прошлом. Стековая модель регистров не используется.

Если вы попробуете заменить в этих примерах *double* на *float*, то инструкции будут использоваться те же, только с суффиксом `-SS` ("Scalar Single-Precision"), например, `MOVSS`, `COMISS`, `ADDSS`, итд.

"Scalar" означает что SIMD-регистр будет хранить только одно значение, вместо нескольких. Инструкции, работающие с несколькими значениями в регистре одновременно, имеют "Packed" в названии.

Глава 25

Конвертирование температуры

Еще один крайне популярный пример из книг по программированию для начинающих, это простейшая программа для конвертирования температуры по Фаренгейту в температуру по Цельсию.

$$C = \frac{5 \cdot (F - 32)}{9}$$

Я также добавил простейшую обработку ошибок: 1) мы должны проверять правильность ввода пользователем; 2) мы должны проверять результат, не ниже ли он −273 по Цельсию (что, как мы можем помнить из школьных уроков физики, ниже абсолютного ноля).

Ф-ция `exit()` заканчивает программу тут же, без возврата в вызывающую ф-цию.

25.1 Целочисленные значения

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%d", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %d\n", celsius);
};
```

25.1.1 MSVC 2012 x86 /Ox

Листинг 25.1: MSVC 2012 x86 /Ox

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H
```

```
_fahr$ = -4                                             ; size = 4
_main   PROC
        push    ecx
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; call printf()
        lea     eax, DWORD PTR _fahr$[esp+12]
        push    eax
        push    OFFSET $SG4230          ; '%d'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        mov     eax, DWORD PTR _fahr$[esp+8]
        add     eax, -32                                ; fffffe0H
        lea     ecx, DWORD PTR [eax+eax*4]
        mov     eax, 954437177                          ; 38e38e39H
        imul    ecx
        sar     edx, 1
        mov     eax, edx
        shr     eax, 31                                 ; 0000001fH
        add     eax, edx
        cmp     eax, -273                               ; fffffeefH
        jge     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        push    eax
        push    OFFSET $SG4234          ; 'Celsius: %d'
        call    esi                     ; call printf()
        add     esp, 8
        ; return 0 - at least by C99 standard
        xor     eax, eax
        pop     esi
        pop     ecx
        ret     0
$LN8@main:
_main   ENDP
```

Что мы можем сказать об этом:

- Адрес ф-ции `printf()` в начале загружается в регистр `ESI` так что последующие вызовы `printf()` происходят просто при помощи инструкции `CALL ESI`. Это очень популярная техника компиляторов, может присутствовать, если имеются несколько вызовов одной и той же ф-ции в одном месте, и/или имеется свободный регистр для этого.
- Мы видим инструкцию `ADD EAX, -32` в том месте где от значения должно отняться 32. $EAX = EAX + (-32)$ эквивалентно $EAX = EAX - 32$ и как-то компилятор решил использовать `ADD` вместо `SUB`. Может быть оно того стоит.

- Инструкция LEA используются там где нужно умножить значение на 5: `lea ecx, DWORD PTR [eax+eax*4]`. Да, $i + i * 4$ эквивалентно $i * 5$ и LEA работает быстрее чем IMUL. Кстати, пара инструкций `SHL EAX, 2` / `ADD EAX, EAX` может быть использована здесь вместо LEA— некоторые компиляторы так и делают.
- Деление через умножение (14) также используется здесь.
- Ф-ция `main()` возвращает 0 хотя `return 0` в конце ф-ции отсутствует. В стандарте C99 [15, 5.1.2.2.3] указано что `main()` будет возвращать 0 в случае отсутствия выражения `return`. Это правило работает только для ф-ции `main()`. И хотя, MSVC не поддерживает C99, может быть частично и поддерживает?

25.1.2 MSVC 2012 x64 /Ox

Код почти такой же, хотя я заметил инструкцию `INT 3` после каждого вызова `exit()`:

```
        xor     ecx, ecx
        call    QWORD PTR __imp_exit
        int     3
```

`INT 3` это брякпойнт для отладчика.

Известно что ф-ция `exit()` из тех, что никогда не возвращают управление [1], так что если управление все же возвращается, значит происходит что-то крайне странное, и пришло время запускать отладчик.

25.2 Числа с плавающей запятой

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        double celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%lf", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;

        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 использует инструкции FPU...

Листинг 25.2: MSVC 2010 x86 /Ox

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
__real@4040000000000000 DQ 04040000000000000r   ; 32
```

[1] еще одна популярная из того же ряда это `longjmp()`

```
_fahr$ = -8                                             ; size = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4038          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; call printf
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4040          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4041          ; 'Error while parsing your input'
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN2@main:
        fld     QWORD PTR _fahr$[esp+12]
        fsub    QWORD PTR __real@4040000000000000 ; 32
        fmul    QWORD PTR __real@4014000000000000 ; 5
        fdiv    QWORD PTR __real@4022000000000000 ; 9
        fld     QWORD PTR __real@c071100000000000 ; -273
        fcomp   ST(1)
        fnstsw  ax
        test    ah, 65                                  ; 00000041H
        jne     SHORT $LN1@main
        push    OFFSET $SG4043          ; 'Error: incorrect temperature!'
        fstp    ST(0)
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN1@main:
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4044          ; 'Celsius: %lf'
        call    esi
        add     esp, 12                                 ; 0000000cH
        ; return 0
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN10@main:
_main   ENDP
```

... но MSVC от года 2012 использует инструкции SIMD вместо этого:

Листинг 25.3: MSVC 2010 x86 /Ox

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4040000000000000 DQ 04040000000000000r   ; 32
__real@4022000000000000 DQ 04022000000000000r   ; 9
```

```
__real@4014000000000000 DQ 04014000000000000r   ; 5

_fahr$ = -8                                             ; size = 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in Fahrenheit:'
        call    esi                     ; call printf
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4230          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing your input'
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        movsd   xmm1, QWORD PTR _fahr$[esp+12]
        subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
        movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
        mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
        divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
        comisd  xmm0, xmm1
        jbe     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect temperature!'
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        sub     esp, 8
        movsd   QWORD PTR [esp], xmm1
        push    OFFSET $SG4234          ; 'Celsius: %lf'
        call    esi                     ; call printf
        add     esp, 12                                 ; 0000000cH
        ; return 0
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN8@main:
_main   ENDP
```

Конечно, SIMD-инструкции доступны и в x86-режиме, включая те что работают с числами с плавающей запятой. Их использовать в каком-то смысле проще, так что новый компилятор от Microsoft теперь применяет их.

Мы можем также заметить что значение −273 загружается в регистр `XMM0` слишком рано. И это нормально, потому что компилятор может генерировать инструкции далеко не в том порядке, в котором они появляются в исходном коде.

Глава 26

C99 restrict

А вот причина, из-за которой программы на FORTRAN, в некоторых случаях, работают быстрее чем на Си.

```
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

Это очень простой пример, в котором есть одна особенность: указатель на массив `update_me` может быть указателем на массив `sum`, `product`, или даже `sum_product` — ведь нет ничего криминального в том чтобы аргументам функции быть такими, верно?

Компилятор знает об этом, поэтому генерирует код, где в теле цикла будет 4 основных стадии:

- вычислить следующий `sum[i]`
- вычислить следующий `product[i]`
- вычислить следующий `update_me[i]`
- вычислить следующий `sum_product[i]` — на этой стадии придется снова загружать из памяти подсчитанные `sum[i]` и `product[i]`

Возможно ли соптимизировать последнюю стадию? Ведь подсчитанные `sum[i]` и `product[i]` не обязательно снова загружать из памяти, ведь мы их только что подсчитали. Можно, но компилятор не уверен, что на третьей стадии ничего не затерлось! Это называется “pointer aliasing”, ситуация, когда компилятор не может быть уверен что память на которую указывает какой-то указатель, не изменилась.

restrict в стандарте Си C99 [15, 6.7.3/1] это обещание, даваемое компилятору программистом, что аргументы функции, отмеченные этим ключевым словом, всегда будут указывать на разные места в памяти и пересекаться не будут.

Если быть более точным, и описывать это формально, *restrict* показывает, что только данный указатель будет использоваться для доступа к этому объекту, с которым мы работаем через этот указатель, больше никакой указатель для этого использоваться не будет. Можно даже сказать, что к всякому объекту, доступ будет осуществляться только через один единственный указатель, если он отмечен как *restrict*.

Добавим это ключевое слово к каждому аргументу-указателю:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* ↙
    ↳ restrict sum_product,
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
```

```
                sum_product[i]=sum[i]+product[i];
        };
};
```

Посмотрим результаты:

Листинг 26.1: GCC x64: f1()

```
f1:
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6:
        mov     r11, rdi
        mov     rdi, rax
.L4:
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d         ; сохранить в sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d         ; сохранить в product[]
        mov     DWORD PTR [r12+r11*4], ebx    ; сохранить в update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]         ; перезагрузить sum[i]
        add     r10d, DWORD PTR [rax]         ; перезагрузить product[i]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d  ; сохранить в sum_product[]
        jne     .L6
.L1:
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

Листинг 26.2: GCC x64: f2()

```
f2:
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
.L11:
        mov     rax, rdi
```

```
        mov     rdi, r11
.L10:
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d  ; сохранить в update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx    ; сохранить в sum[]
        mov     DWORD PTR [r9+rax*4], r11d   ; сохранить в product[]
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; сохранить в sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7:
        pop     rbx rsi rdi rbp r12 r13
        ret
```

Разница между скомпилированной функцией `f1()` и `f2()` такая: в `f1()`, `sum[i]` и `product[i]` загружаются снова посреди тела цикла, а в `f2()` этого нет, используются уже подсчитанные значения, ведь мы "пообещали" компилятору, что никто и ничто не изменит значения в `sum[i]` и `product[i]` во время исполнения тела цикла, поэтому он "уверен", что значения из памяти можно не загружать снова. Очевидно, второй вариант будет работать быстрее.

Но что будет если указатели в аргументах функций все же будут пересекаться? Это останется на совести программиста, а результаты вычислений будут неверными.

Вернемся к FORTRAN. Компиляторы с этого ЯП, по умолчанию, все указатели считают таковыми, поэтому, когда в Си не было возможности указать *restrict*, FORTRAN в этих случаях мог генерировать более быстрый код.

Насколько это практично? Там, где функция работает с несколькими большими блоками в памяти. Такого очень много в линейной алгебре, например. Очень много линейной алгебры используется на суперкомпьютерах/HPC[1], возможно, поэтому, традиционно, там часто используется FORTRAN, до сих пор [19].

Ну а когда итераций цикла не очень много, конечно, тогда прирост скорости не будет ощутимым.

[1] High-Performance Computing

Глава 27

Inline-функции

Inline-код это когда компилятор, вместо того чтобы генерировать инструкцию вызова небольшой функции, просто вставляет её тело прямо в это место.

Листинг 27.1: Простой пример

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
        return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
        int celsius=atol(argv[1]);
        printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

... это компилируется вполне предсказуемо, хотя, если включить оптимизации GCC (-O3), мы увидим:

Листинг 27.2: GCC 4.8.1 -O3

```
_main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        call    ___main
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp], eax
        call    _atol
        mov     edx, 1717986919
        mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
        lea     ecx, [eax+eax*8]
        mov     eax, ecx
        imul    edx
        sar     ecx, 31
        sar     edx
        sub     edx, ecx
        add     edx, 32
        mov     DWORD PTR [esp+4], edx
        call    _printf
        leave
        ret
```

(Здесь деление заменено умножением(14).)

Да, наша маленькая ф-ция была помещена прямо перед вызовом `printf()`. Почему? Это может быть быстрее чем исполнять код самой ф-ции плюс затраты на вызов и возврат.

В прошлом, такие ф-ции нужно было маркировать ключевым словом “inline” в определении ф-ции, хотя, в наше время, такие ф-ции выбираются компилятором автоматически.

Другая очень частая оптимизация это вставка кода строковых ф-ций таких как *strcpy()*, *strcmp()*, и т.д.

Листинг 27.3: Еще один простой пример

```
bool is_bool (char *s)
{
        if (strcmp (s, "true")==0)
                return true;
        if (strcmp (s, "false")==0)
                return false;

        assert(0);
};
```

Листинг 27.4: GCC 4.8.1 -O3

```
_is_bool:
        push    edi
        mov     ecx, 5
        push    esi
        mov     edi, OFFSET FLAT:LC0 ; "true\0"
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        repz cmpsb
        je      L3
        mov     esi, DWORD PTR [esp+32]
        mov     ecx, 6
        mov     edi, OFFSET FLAT:LC1 ; "false\0"
        repz cmpsb
        seta    cl
        setb    dl
        xor     eax, eax
        cmp     cl, dl
        jne     L8
        add     esp, 20
        pop     esi
        pop     edi
        ret
```

Вот пример очень часто попадающегося фрагмента кода strcmp() генерируемого MSVC:

Листинг 27.5: MSVC

```
    mov     dl, [eax]
    cmp     dl, [ecx]
    jnz     short loc_10027FA0
    test    dl, dl
    jz      short loc_10027F9C
    mov     dl, [eax+1]
    cmp     dl, [ecx+1]
    jnz     short loc_10027FA0
    add     eax, 2
    add     ecx, 2
    test    dl, dl
    jnz     short loc_10027F80

loc_10027F9C: ; CODE XREF: f1+448
    xor     eax, eax
    jmp     short loc_10027FA5

loc_10027FA0: ; CODE XREF: f1+444
              ; f1+450
```

```
    sbb     eax, eax
    sbb     eax, 0FFFFFFFFh
```

Я написал небольшой скрипт для IDA для поиска и сворачивания таких очень часто попадающихся inline-функций:
https://github.com/yurichev/IDA_scripts.

Глава 28

Неверно дизассемблированный код

Практикующие reverse engineer-ы часто сталкиваются с неверно дизассемблированным кодом.

28.1 Дизассемблирование началось в неверном месте (x86)

В отличие от ARM и MIPS (где у каждой инструкции длина или 2 или 4 байта), x86-инструкции имеют переменную длину, так что, любой дизассемблер, начиная работу с середины x86-инструкции, может выдать неверные результаты.

Как пример:

```
add     [ebp-31F7Bh], cl
dec     dword ptr [ecx-3277Bh]
dec     dword ptr [ebp-2CF7Bh]
inc     dword ptr [ebx-7A76F33Ch]
fdiv    st(4), st
db 0FFh
dec     dword ptr [ecx-21F7Bh]
dec     dword ptr [ecx-22373h]
dec     dword ptr [ecx-2276Bh]
dec     dword ptr [ecx-22B63h]
dec     dword ptr [ecx-22F4Bh]
dec     dword ptr [ecx-23343h]
jmp     dword ptr [esi-74h]
xchg    eax, ebp
clc
std
db 0FFh
db 0FFh
mov     word ptr [ebp-214h], cs
mov     word ptr [ebp-238h], ds
mov     word ptr [ebp-23Ch], es
mov     word ptr [ebp-240h], fs
mov     word ptr [ebp-244h], gs
pushf
pop     dword ptr [ebp-210h]
mov     eax, [ebp+4]
mov     [ebp-218h], eax
lea     eax, [ebp+4]
mov     [ebp-20Ch], eax
mov     dword ptr [ebp-2D0h], 10001h
mov     eax, [eax-4]
mov     [ebp-21Ch], eax
mov     eax, [ebp+0Ch]
mov     [ebp-320h], eax
mov     eax, [ebp+10h]
mov     [ebp-31Ch], eax
mov     eax, [ebp+4]
```

```
mov     [ebp-314h], eax
call    ds:IsDebuggerPresent
mov     edi, eax
lea     eax, [ebp-328h]
push    eax
call    sub_407663
pop     ecx
test    eax, eax
jnz     short loc_402D7B
```

В начале мы видим неверно дизассемблированные инструкции, но потом, так или иначе, дизассемблер находит верный след.

28.2 Как выглядят случайные данные в дизассемблированном виде?

Общее, что можно сразу заметить, это:

- Необычно большой разброс инструкций. Самые частые x86-инструкции это `PUSH`, `MOV`, `CALL`, но здесь мы видим инструкции из любых групп: FPU-инструкции, инструкции `IN`/`OUT`, редкие и системные инструкции, всё друг с другом смешано в одном месте.
- Большие и случайные значения, смещения, immediates.
- Переходы с неверными смещениями часто имеют адрес перехода в середину другой инструкции.

Листинг 28.1: случайный шум (x86)

```
    mov     bl, 0Ch
    mov     ecx, 0D38558Dh
    mov     eax, ds:2C869A86h
    db      67h
    mov     dl, 0CCh
    insb
    movsb
    push    eax
    xor     [edx-53h], ah
    fcom    qword ptr [edi-45A0EF72h]
    pop     esp
    pop     ss
    in      eax, dx
    dec     ebx
    push    esp
    lds     esp, [esi-41h]
    retf
    rcl     dword ptr [eax], cl
    mov     cl, 9Ch
    mov     ch, 0DFh
    push    cs
    insb
    mov     esi, 0D9C65E4Dh
    imul    ebp, [ecx], 66h
    pushf
    sal     dword ptr [ebp-64h], cl
    sub     eax, 0AC433D64h
    out     8Ch, eax
    pop     ss
    sbb     [eax], ebx
    aas
    xchg    cl, [ebx+ebx*4+14B31Eh]
    jecxz   short near ptr loc_58+1
    xor     al, 0C6h
    inc     edx
```

```
    db      36h
    pusha
    stosb
    test    [ebx], ebx
    sub     al, 0D3h ; 'L'
    pop     eax
    stosb

loc_58: ; CODE XREF: seg000:0000004A
    test    [esi], eax
    inc     ebp
    das
    db      64h
    pop     ecx
    das
    hlt

    pop     edx
    out     0B0h, al
    lodsb
    push    ebx
    cdq
    out     dx, al
    sub     al, 0Ah
    sti
    outsd
    add     dword ptr [edx], 96FCBE4Bh
    and     eax, 0E537EE4Fh
    inc     esp
    stosd
    cdq
    push    ecx
    in      al, 0CBh
    mov     ds:0D114C45Ch, al
    mov     esi, 659D1985h
    enter   6FE8h, 0D9h
    enter   6FE6h, 0D9h
    xchg    eax, esi
    sub     eax, 0A599866Eh
    retn

    pop     eax
    dec     eax
    adc     al, 21h ; '!'
    lahf
    inc     edi
    sub     eax, 9062EE5Bh
    bound   eax, [ebx]

loc_A2: ; CODE XREF: seg000:00000120
    wait
    iret

    jnb     short loc_D7
    cmpsd
    iret

    jnb     short loc_D7
    sub     ebx, [ecx]
    in      al, 0Ch
    add     esp, esp
```

```
    mov     bl, 8Fh
    xchg    eax, ecx
    int     67h
    pop     ds
    pop     ebx
    db      36h
    xor     esi, [ebp-4Ah]
    mov     ebx, 0EB4F980Ch
    repne add bl, dh
    imul    ebx, [ebp+5616E7A5h], 67A4D1EEh
    xchg    eax, ebp
    scasb
    push    esp
    wait
    mov     dl, 11h
    mov     ah, 29h ; ')'
    fist    dword ptr [edx]

loc_D7: ; CODE XREF: seg000:000000A4
        ; seg000:000000A8 ...
    dec     dword ptr [ebp-5D0E0BA4h]
    call    near ptr 622FEE3Eh
    sbb     ax, 5A2Fh
    jmp     dword ptr cs:[ebx]

    xor     ch, [edx-5]
    inc     esp
    push    edi
    xor     esp, [ebx-6779D3B8h]
    pop     eax
    int     3               ; Trap to Debugger
    rcl     byte ptr [ebx-3Eh], cl
    xor     [edi], bl
    sbb     al, [edx+ecx*4]
    xor     ah, [ecx-1DA4E05Dh]
    push    edi
    xor     ah, cl
    popa
    cmp     dword ptr [edx-62h], 46h ; 'F'
    dec     eax
    in      al, 69h
    dec     ebx
    iret

    or      al, 6
    jns     short near ptr loc_D7+3
    shl     byte ptr [esi], 42h
    repne adc [ebx+2Ch], eax
    icebp
    cmpsd
    leave
    push    esi
    jmp     short loc_A2

    and     eax, 0F2E41FE9h
    push    esi
    loop    loc_14F
    add     ah, fs:[edx]

loc_12D: ; CODE XREF: seg000:00000169
    mov     dh, 0F7h
```

```
    add     [ebx+7B61D47Eh], esp
    mov     edi, 79F19525h
    rcl     byte ptr [eax+22015F55h], cl
    cli
    sub     al, 0D2h ; 'T'
    dec     eax
    mov     ds:0A81406F5h, eax
    sbb     eax, 0A7AA179Ah
    in      eax, dx

loc_14F: ; CODE XREF: seg000:00000128
    and     [ebx-4CDFAC74h], ah
    pop     ecx
    push    esi
    mov     bl, 2Dh ; '-'
    in      eax, 2Ch
    stosd
    inc     edi
    push    esp

locret_15E: ; CODE XREF: seg000:loc_1A0
    retn    0C432h

    and     al, 86h
    cwde
    and     al, 8Fh
    cmp     ebp, [ebp+7]
    jz      short loc_12D
    sub     bh, ch
    or      dword ptr [edi-7Bh], 8A16C0F7h
    db      65h
    insd
    mov     al, ds:0A3A5173Dh
    dec     ecx
    push    ds
    xor     al, cl
    jg      short loc_195
    push    6Eh ; 'n'
    out     0DDh, al
    inc     edi
    sub     eax, 6899BBF1h
    leave
    rcr     dword ptr [ecx-69h], cl
    sbb     ch, [edi+5EDDCB54h]

loc_195: ; CODE XREF: seg000:0000017F
    push    es
    repne sub ah, [eax-105FF22Dh]
    cmc
    and     ch, al

loc_1A0: ; CODE XREF: seg000:00000217
    jnp     short near ptr locret_15E+1
    or      ch, [eax-66h]
    add     [edi+edx-35h], esi
    out     dx, al
    db      2Eh
    call    far ptr 1AAh:6832F5DDh
    jz      short near ptr loc_1DA+1
    sbb     esp, [edi+2CB02CEFh]
    xchg    eax, edi
```

```
    xor     [ebx-766342ABh], edx

loc_1C1: ; CODE XREF: seg000:00000212
    cmp     eax, 1BE9080h
    add     [ecx], edi
    aad     0
    imul    esp, [edx-70h], 0A8990126h
    or      dword ptr [edx+10C33693h], 4Bh
    popf

loc_1DA: ; CODE XREF: seg000:000001B2
    mov     ecx, cs
    aaa
    mov     al, 39h ; '9'
    adc     byte ptr [eax-77F7F1C5h], 0C7h
    add     [ecx], bl
    retn    0DD42h

    db      3Eh
    mov     fs:[edi], edi
    and     [ebx-24h], esp
    db      64h
    xchg    eax, ebp
    push    cs
    adc     eax, [edi+36h]
    mov     bh, 0C7h
    sub     eax, 0A710CBE7h
    xchg    eax, ecx
    or      eax, 51836E42h
    xchg    eax, ebx
    inc     ecx
    jb      short near ptr loc_21E+3
    db      64h
    xchg    eax, esp
    and     dh, [eax-31h]
    mov     ch, 13h
    add     ebx, edx
    jnb     short loc_1C1
    db      65h
    adc     al, 0C5h
    js      short loc_1A0
    sbb     eax, 887F5BEEh

loc_21E: ; CODE XREF: seg000:00000207
    mov     eax, 888E1FD6h
    mov     bl, 90h
    cmp     [eax], ecx
    rep int 61h             ; reserved for user interrupt
    and     edx, [esi-7EB5C9EAh]
    fisttp  qword ptr [eax+esi*4+38F9BA6h]
    jmp     short loc_27C

    fadd    st, st(2)
    db      3Eh
    mov     edx, 54C03172h
    retn

    db      64h
    pop     ds
    xchg    eax, esi
    rcr     ebx, cl
```

```
    cmp     [di+2Eh], ebx
    repne xor [di-19h], dh
    insd
    adc     dl, [eax-0C4579F7h]
    push    ss
    xor     [ecx+edx*4+65h], ecx
    mov     cl, [ecx+ebx-32E8AC51h]
    or      [ebx], ebp
    cmpsb
    lodsb
    iret
```

Листинг 28.2: случайный шум (x86-64)

```
    lea     esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
    lea     ecx, cs:0FFFFFFFFB2A6780Fh
    mov     al, 96h
    mov     ah, 0CEh
    push    rsp
    lods    byte ptr [esi]

    db  2Fh ; /

    pop     rsp
    db      64h
    retf    0E993h

    cmp     ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]

    db  9Ah

    rcr     byte ptr [rax+1Dh], cl
    lodsd
    xor     [rbp+6CF20173h], edx
    xor     [rbp+66F8B593h], edx
    push    rbx
    sbb     ch, [rbx-0Fh]
    stosd
    int     87h
    db      46h, 4Ch
    out     33h, rax
    xchg    eax, ebp
    test    ecx, ebp
    movsd
    leave
    push    rsp

    db  16h

    xchg    eax, esi
    pop     rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
```

```
    out     dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
    pop     rbp

    db  60h ; `

    movsxd  rbp, dword ptr [rbp-17h]
    pop     rbx
    out     7Dh, al
    add     eax, 0D79BE769h

    db  1Fh

    retf    0CAB9h

    jl      short near ptr loc_B3D+4
    sal     dword ptr [rbx+rbp+4Dh], 0D3h
    mov     cl, 41h ; 'A'
    imul    eax, [rbp-5B77E717h], 1DDE6E5h
    imul    ecx, ebx, 66359BCCh
    xlat

    db  60h ; `

    cmp     bl, [rax]
    and     ebp, [rcx-57h]
    stc
    sub     [rcx+1A533AB4h], al
    jmp     short loc_C05

    db  4Bh ; K

    int     3               ; Trap to Debugger
    xchg    ebx, [rsp+rdx-5Bh]

    db 0D6h

    mov     esp, 0C5BA61F7h
    out     0A3h, al        ; Interrupt Controller #2, 8259A
    add     al, 0A6h
    pop     rbx
    cmp     bh, fs:[rsi]
    and     ch, cl
    cmp     al, 0F3h

    db  0Eh

    xchg    dh, [rbp+rax*4-4CE9621Ah]
    stosd
    xor     [rdi], ebx
    stosb
    xchg    eax, ecx
    push    rsi
    insd
    fidiv   word ptr [rcx]
    xchg    eax, ecx
    mov     dh, 0C0h ; 'L'
    xchg    eax, esp
    push    rsi
```

```
    mov     dh, [rdx+rbp+6918F1F3h]
    xchg    eax, ebp
    out     9Dh, al

loc_BC0: ; CODE XREF: seg000:0000000000000C26
    or      [rcx-0Dh], ch
    int     67h             ;  - LIM EMS
    push    rdx
    sub     al, 43h ; 'C'
    test    ecx, ebp
    test    [rdi+71F372A4h], cl

    db    7

    imul    ebx, [rsi-0Dh], 2BB30231h
    xor     ebx, [rbp-718B6E64h]
    jns     short near ptr loc_C56+1
    ficomp  dword ptr [rcx-1Ah]
    and     eax, 69BEECC7h
    mov     esi, 37DA40F6h
    imul    r13, [rbp+rdi*8+529F33CDh], 0FFFFFFFFF35CDD30h
    or      [rbx], edx
    imul    esi, [rbx-34h], 0CDA42B87h

    db  36h ; 6
    db  1Fh


loc_C05: ; CODE XREF: seg000:0000000000000B86
    add     dh, [rcx]
    mov     edi, 0DD3E659h
    ror     byte ptr [rdx-33h], cl
    xlat
    db      48h
    sub     rsi, [rcx]

    db  1Fh
    db    6

    xor     [rdi+13F5F362h], bh
    cmpsb
    sub     esi, [rdx]
    pop     rbp
    sbb     al, 62h ; 'b'
    mov     dl, 33h ; '3'

    db  4Dh ; M
    db  17h

    jns     short loc_BC0
    push    0FFFFFFFFFFFFFF86h

loc_C2A: ; CODE XREF: seg000:0000000000000C8F
    sub     [rdi-2Ah], eax

    db 0FEh

    cmpsb
    wait
    rcr     byte ptr [rax+5Fh], cl
    cmp     bl, al
```

```
    pushfq
    xchg    ch, cl

    db  4Eh ; N
    db  37h ; 7

    mov     ds:0E43F3CCD3D9AB295h, eax
    cmp     ebp, ecx
    jl      short loc_C87
    retn    8574h

    out     3, al           ; DMA controller, 8237A-5.
                            ; channel 1 base address and word count

loc_C4C: ; CODE XREF: seg000:0000000000000C7F
    cmp     al, 0A6h
    wait
    push    0FFFFFFFFFFFFFFBEh

    db  82h

    ficom   dword ptr [rbx+r10*8]

loc_C56: ; CODE XREF: seg000:0000000000000BDE
    jnz     short loc_C76
    xchg    eax, edx
    db      26h
    wait
    iret

    push    rcx

    db  48h ; H
    db  9Bh
    db  64h ; d
    db  3Eh ; >
    db  2Fh ; /

    mov     al, ds:8A7490CA2E9AA728h
    stc

    db  60h ; `

    test    [rbx+rcx], ebp
    int     3               ; Trap to Debugger
    xlat

loc_C72: ; CODE XREF: seg000:0000000000000CC6
    mov     bh, 98h

    db  2Eh ; .
    db 0DFh


loc_C76: ; CODE XREF: seg000:loc_C56
    jl      short loc_C91
    sub     ecx, 13A7CCF2h
    movsb
    jns     short near ptr loc_C4C+1
    cmpsd
    sub     ah, ah
```

```
    cdq

    db  6Bh ; k
    db  5Ah ; Z


loc_C87: ; CODE XREF: seg000:0000000000000C45
    or      ecx, [rbx+6Eh]
    rep in eax, 0Eh         ; DMA controller, 8237A-5.
                            ; Clear mask registers.
                            ; Any OUT enables all 4 channels.
    cmpsb
    jnb     short loc_C2A

loc_C91: ; CODE XREF: seg000:loc_C76
    scasd
    add     dl, [rcx+5FEF30E6h]
    enter   0FFFFFFFFFFFFC733h, 7Ch
    insd
    mov     ecx, gs
    in      al, dx
    out     2Dh, al
    mov     ds:6599E434E6D96814h, al
    cmpsb
    push    0FFFFFFFFFFFFFFD6h
    popfq
    xor     ecx, ebp
    db      48h
    insb
    test    al, cl
    xor     [rbp-7Bh], cl
    and     al, 9Bh

    db  9Ah

    push    rsp
    xor     al, 8Fh
    cmp     eax, 924E81B9h
    clc
    mov     bh, 0DEh
    jbe     short near ptr loc_C72+1

    db  1Eh

    retn    8FCAh

    db 0C4h ; -


loc_CCD: ; CODE XREF: seg000:0000000000000D22
    adc     eax, 7CABFBF8h

    db  38h ; 8

    mov     ebp, 9C3E66FCh
    push    rbp
    dec     byte ptr [rcx]
    sahf
    fidivr  word ptr [rdi+2Ch]

    db  1Fh
```

```
    db      3Eh
    xchg    eax, esi

loc_CE2: ; CODE XREF: seg000:0000000000000D5E
    mov     ebx, 0C7AFE30Bh
    clc
    in      eax, dx
    sbb     bh, bl
    xchg    eax, ebp

    db  3Fh ; ?

    cmp     edx, 3EC3E4D7h
    push    51h
    db      3Eh
    pushfq
    jl      short loc_D17
    test    [rax-4CFF0D49h], ebx

    db  2Fh ; /

    rdtsc
    jns     short near ptr loc_D40+4
    mov     ebp, 0B2BB03D8h
    in      eax, dx

    db  1Eh

    fsubr   dword ptr [rbx-0Bh]
    jns     short loc_D70
    scasd
    mov     ch, 0C1h ; '+'
    add     edi, [rbx-53h]

    db 0E7h


loc_D17: ; CODE XREF: seg000:0000000000000CF7
    jp      short near ptr unk_D79
    scasd
    cmc
    sbb     ebx, [rsi]
    fsubr   dword ptr [rbx+3Dh]
    retn

    db    3

    jnp     short near ptr loc_CCD+4
    db      36h
    adc     r14b, r13b

    db  1Fh

    retf

    test    [rdi+rdi*2], ebx
    cdq
    or      ebx, edi
    test    eax, 310B94BCh
    ffreep  st(7)
```

```
    cwde
    sbb     esi, [rdx+53h]
    push    5372CBAAh

loc_D40: ; CODE XREF: seg000:0000000000000D02
    push    53728BAAh
    push    0FFFFFFFFF85CF2FCh

    db  0Eh

    retn    9B9Bh

    movzx   r9, dword ptr [rdx]
    adc     [rcx+43h], ebp
    in      al, 31h

    db  37h ; 7

    jl      short loc_DC5
    icebp
    sub     esi, [rdi]
    clc
    pop     rdi
    jb      short near ptr loc_CE2+1
    or      al, 8Fh
    mov     ecx, 770EFF81h
    sub     al, ch
    sub     al, 73h ; 's'
    cmpsd
    adc     bl, al
    out     87h, eax        ; DMA page register 74LS612:
                            ; Channel 0 (address bits 16-23)

loc_D70: ; CODE XREF: seg000:0000000000000D0E
    adc     edi, ebx
    db      49h
    outsb
    enter   33E5h, 97h
    xchg    eax, ebx

unk_D79   db 0FEh ; CODE XREF: seg000:loc_D17
          db 0BEh
          db 0E1h
          db  82h


loc_D7D: ; CODE XREF: seg000:0000000000000DB3
    cwde

    db    7
    db  5Ch ; \
    db  10h
    db  73h ; s
    db 0A9h
    db  2Bh ; +
    db  9Fh


loc_D85: ; CODE XREF: seg000:0000000000000DD1
    dec     dh
    jnz     short near ptr loc_DD3+3
```

```
    mov     ds:7C1758CB282EF9BFh, al
    sal     ch, 91h
    rol     dword ptr [rbx+7Fh], cl
    fbstp   tbyte ptr [rcx+2]
    repne mov al, ds:4BFAB3C3ECF2BE13h
    pushfq
    imul    edx, [rbx+rsi*8+3B484EE9h], 8EDC09C6h
    cmp     [rax], al
    jg      short loc_D7D
    xor     [rcx-638C1102h], edx
    test    eax, 14E3AD7h
    insd

    db  38h ; 8
    db  80h
    db 0C3h


loc_DC5: ; CODE XREF: seg000:0000000000000D57
         ; seg000:0000000000000DD8
    cmp     ah, [rsi+rdi*2+527C01D3h]
    sbb     eax, 5FC631F0h
    jnb     short loc_D85

loc_DD3: ; CODE XREF: seg000:0000000000000D87
    call    near ptr 0FFFFFFFFC03919C7h
    loope   near ptr loc_DC5+3
    sbb     al, 0C8h
    std
```

Листинг 28.3: случайный шум (ARM в режиме ARM)

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC],#-0x964
LDCGE   p6, c14, [R0],#0x168
STCCSL  p9, c9, [LR],#0x14C
CMNHIP  PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2,c15,c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA   R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB    SP, [R8],PC,ROR#18
LDCCS   p9, c13, [R6,#0x1BC]
LDRGE   R8, [R9,#0x66E]
STRNEB  R5, [R8],#-0x8C3
STCCSL  p15, c9, [R7,#-0x84]
RSBLS   LR, R2, R11,ASR LR
SVCGT   0x9B0362
SVCGT   0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR     R0, [R3],#-0xCE4
LDCGT   p15, c8, [R1,#0x2CC]
LDRCCB  R1, [R11],-R7,ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
```

```
USATNE  R10, #7, SP,LSL#11
LDRGEB  LR, [R1],#0xE56
STRPLT  R9, [LR],#0x567
LDRLT   R11, [R1],#-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP,LSL#25
LDCL    p8, c14, [R12,#-0x288]
STCNEL  p2, c6, [R6,#-0xBC]!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4,LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7,ASR R0
LDRMI   R7, [LR],R3,LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
EORMI   R12, R6, #0xC5
MCRRCS  p1, 0xF, R1,R3,c2
```

Листинг 28.4: случайный шум (ARM в режиме Thumb)

```
    LSRS    R3, R6, #0x12
    LDRH    R1, [R7,#0x2C]
    SUBS    R0, #0x55 ; 'U'
    ADR     R1, loc_3C
    LDR     R2, [SP,#0x218]
    CMP     R4, #0x86
    SXTB    R7, R4
    LDR     R4, [R1,#0x4C]
    STR     R4, [R4,R2]
    STR     R0, [R6,#0x20]
    BGT     0xFFFFFF72
    LDRH    R7, [R2,#0x34]
    LDRSH   R0, [R2,R4]
    LDRB    R2, [R7,R2]

    DCB 0x17
    DCB 0xED

    STRB    R3, [R1,R1]
    STR     R5, [R0,#0x6C]
    LDMIA   R3, {R0-R5,R7}
    ASRS    R3, R2, #3
    LDR     R4, [SP,#0x2C4]
    SVC     0xB5
    LDR     R6, [R1,#0x40]
    LDR     R5, =0xB2C5CA32
    STMIA   R6, {R1-R4,R6}
    LDR     R1, [R3,#0x3C]
    STR     R1, [R5,#0x60]
    BCC     0xFFFFFF70
    LDR     R4, [SP,#0x1D4]
    STR     R5, [R5,#0x40]
    ORRS    R5, R7

loc_3C ; DATA XREF: ROM:00000006
    B       0xFFFFFF98
```

```
    ASRS    R4, R1, #0x1E
    ADDS    R1, R3, R0
    STRH    R7, [R7,#0x30]
    LDR     R3, [SP,#0x230]
    CBZ     R6, loc_90
    MOVS    R4, R2
    LSRS    R3, R4, #0x17
    STMIA   R6!, {R2,R4,R5}
    ADDS    R6, #0x42 ; 'B'
    ADD     R2, SP, #0x180
    SUBS    R5, R0, R6
    BCC     loc_B0
    ADD     R2, SP, #0x160
    LSLS    R5, R0, #0x1A
    CMP     R7, #0x45
    LDR     R4, [R4,R5]

    DCB 0x2F ; /
    DCB 0xF4

    B       0xFFFFFD18

    ADD     R4, SP, #0x2C0
    LDR     R1, [SP,#0x14C]
    CMP     R4, #0xEE

    DCB  0xA
    DCB 0xFB

    STRH    R7, [R5,#0xA]
    LDR     R3, loc_78

    DCB 0xBE ; -
    DCB 0xFC

    MOVS    R5, #0x96

    DCB 0x4F ; O
    DCB 0xEE

    B       0xFFFFFAE6

    ADD     R3, SP, #0x110

loc_78 ; DATA XREF: ROM:0000006C
    STR     R1, [R3,R6]
    LDMIA   R3!, {R2,R5-R7}
    LDRB    R2, [R4,R2]
    ASRS    R4, R0, #0x13
    BKPT    0xD1
    ADDS    R5, R0, R6
    STR     R5, [R3,#0x58]
```

Листинг 28.5: случайный шум(MIPS little endian)

```
    lw      $t9, 0xCB3($t5)
    sb      $t5, 0x3855($t0)
    sltiu   $a2, $a0, -0x657A
    ldr     $t4, -0x4D99($a2)
    daddi   $s0, $s1, 0x50A4
```

```
    lw      $s7, -0x2353($s4)
    bgtzl   $a1, 0x17C5C

    .byte 0x17
    .byte 0xED
    .byte 0x4B  # K
    .byte 0x54  # T

    lwc2    $31, 0x66C5($sp)
    lwu     $s1, 0x10D3($a1)
    ldr     $t6, -0x204B($zero)
    lwc1    $f30, 0x4DBE($s2)
    daddiu  $t1, $s1, 0x6BD9
    lwu     $s5, -0x2C64($v1)
    cop0    0x13D642D
    bne     $gp, $t4, 0xFFFF9EF0
    lh      $ra, 0x1819($s1)
    sdl     $fp, -0x6474($t8)
    jal     0x78C0050
    ori     $v0, $s2, 0xC634
    blez    $gp, 0xFFFEA9D4
    swl     $t8, -0x2CD4($s2)
    sltiu   $a1, $k0, 0x685
    sdc1    $f15, 0x5964($at)
    sw      $s0, -0x19A6($a1)
    sltiu   $t6, $a3, -0x66AD
    lb      $t7, -0x4F6($t3)
    sd      $fp, 0x4B02($a1)

    .byte 0x96
    .byte 0x25  # %
    .byte 0x4F  # O
    .byte 0xEE

    swl     $a0, -0x1AC9($k0)
    lwc2    $4, 0x5199($ra)
    bne     $a2, $a0, 0x17308

    .byte 0xD1
    .byte 0xBE
    .byte 0x85
    .byte 0x19

    swc2    $8, 0x659D($a2)
    swc1    $f8, -0x2691($s6)
    sltiu   $s6, $t4, -0x2691
    sh      $t9, -0x7992($t4)
    bne     $v0, $t0, 0x163A4
    sltiu   $a3, $t2, -0x60DF
    lbu     $v0, -0x11A5($v1)
    pref    0x1B, 0x362($gp)
    pref    7, 0x3173($sp)
    blez    $t1, 0xB678
    swc1    $f3, flt_CE4($zero)
    pref    0x11, -0x704D($t4)
    ori     $k1, $s2, 0x1F67
    swr     $s6, 0x7533($sp)
    swc2    $15, -0x67F4($k0)
    ldl     $s3, 0xF2($t7)
    bne     $s7, $a3, 0xFFFE973C
    sh      $s1, -0x11AA($a2)
```

```
    bnel    $a1, $t6, 0xFFFE566C
    sdr     $s1, -0x4D65($zero)
    sd      $s2, -0x24D7($t8)
    scd     $s4, 0x5C8D($t7)

    .byte 0xA2
    .byte 0xE8
    .byte 0x5C  # \
    .byte 0xED

    bgtz    $t3, 0x189A0
    sd      $t6, 0x5A2F($t9)
    sdc2    $10, 0x3223($k1)
    sb      $s3, 0x5744($t9)
    lwr     $a2, 0x2C48($a0)
    beql    $fp, $s2, 0xFFFF3258
```

Также важно помнить, что хитрым образом написанный код для распаковки и дешифровки (включая самомодифицирующийся), также может выглядеть как случайный шум, тем не менее, он исполняется корректно.

28.3 Информационная энтропия среднестатистического кода

Результаты работы утилиты *ent*[1].

(Энтропия идеально сжатого (или зашифрованного) файла — 8 бит на байт; файла с нулями любой длины — 0 бит на байт.)

Здесь видно что код для CPU с 4-байтными инструкциями (ARM в режиме ARM и MIPS) наименее экономичны в этом смысле.

28.3.1 x86

Секция `.text` файла `ntoskrnl.exe` из Windows 2003:

```
Entropy = 6.662739 bits per byte.

Optimum compression would reduce the size
of this 593920 byte file by 16 percent.
...
```

Секция `.text` файла `ntoskrnl.exe` из Windows 7 x64:

```
Entropy = 6.549586 bits per byte.

Optimum compression would reduce the size
of this 1685504 byte file by 18 percent.
...
```

28.3.2 ARM (Thumb)

AngryBirds Classic:

```
Entropy = 7.058766 bits per byte.

Optimum compression would reduce the size
of this 3336888 byte file by 11 percent.
...
```

[1] http://www.fourmilab.ch/random/

28.3.3 ARM (режим ARM)

Linux Kernel 3.8.0:

```
Entropy = 6.036160 bits per byte.

Optimum compression would reduce the size
of this 6946037 byte file by 24 percent.
...
```

28.3.4 MIPS (little endian)

Секция `.text` файла `user32.dll` из Windows NT 4:

```
Entropy = 6.098227 bits per byte.

Optimum compression would reduce the size
of this 433152 byte file by 23 percent.
....
```

Глава 29

Си++

29.1 Классы

29.1.1 Простой пример

Внутреннее представление классов в Си++ почти такое же, как и представление структур.

Давайте попробуем простой пример с двумя переменными, двумя конструкторами и одним методом:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };

    void dump()
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

MSVC — x86

Вот как выглядит `main()` на ассемблере:

Листинг 29.1: MSVC

```
_c2$ = -16 ; size = 8
_c1$ = -8  ; size = 8
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

Вот что происходит. Под каждый экземпляр класса *c* выделяется по 8 байт, столько же, сколько нужно для хранения двух переменных.

Для *c1* вызывается конструктор по умолчанию без аргументов `??0c@@QAE@XZ`. Для *c2* вызывается другой конструктор `??0c@@QAE@HH@Z` и передаются два числа в качестве аргументов.

А указатель на объект (*this* в терминологии Си++) передается в регистре `ECX`. Это называется thiscall (29.1.1) — метод передачи указателя на объект.

В данном случае, MSVC делает это через `ECX`. Необходимо помнить, что это не стандартизированный метод, и другие компиляторы могут делать это иначе, например, через первый аргумент функции (как GCC).

Почему у имен функций такие странные имена? Это name mangling.

В Си++, у класса, может иметься несколько методов с одинаковыми именами, но аргументами разных типов — это полиморфизм. Ну и конечно, у разных классов могут быть методы с одинаковыми именами.

Name mangling позволяет закодировать имя класса + имя метода + типы всех аргументов метода в одной ASCII-строке, которая затем используется как внутреннее имя функции. Это все потому что ни компоновщик[1], ни загрузчик DLL ОС (мангленные имена могут быть среди экспортов/импортов в DLL), ничего не знают о Си++ или ООП[2].

Далее вызывается два раза `dump()`.

Теперь смотрим на код в конструкторах:

Листинг 29.2: MSVC

```
_this$ = -4       ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
```

[1] linker

[2] Объектно-Ориентированное Программирование

```
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8     ; size = 4
_b$ = 12    ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _a$[ebp]
    mov  DWORD PTR [eax], ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR _b$[ebp]
    mov  DWORD PTR [edx+4], eax
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c
```

Конструкторы — это просто функции, они используют указатель на структуру в ECX, перекладывают его себе в локальную переменную, хотя это и не обязательно.

Из стандарта Си++ мы знаем [16, 12.1] что конструкторы не должны возвращать значение. В реальности, внутри, конструкторы возвращают указатель на созданный объект, т.е., *this*.

И еще метод dump():

Листинг 29.3: MSVC

```
_this$ = -4          ; size = 4
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax+4]
    push ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR [edx]
    push eax
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    mov  esp, ebp
    pop  ebp
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

Все очень просто, dump() берет указатель на структуру состоящую из двух *int* через ECX, выдергивает оттуда две переменные и передает их в printf().

А если скомпилировать с оптимизацией (/Ox), то кода будет намного меньше:

Листинг 29.4: MSVC

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    mov  eax, ecx
    mov  DWORD PTR [eax], 667
```

```
    mov  DWORD PTR [eax+4], 999
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8  ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov  edx, DWORD PTR _b$[esp-4]
    mov  eax, ecx
    mov  ecx, DWORD PTR _a$[esp-4]
    mov  DWORD PTR [eax], ecx
    mov  DWORD PTR [eax+4], edx
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

Вот и все. Единственное о чем еще нужно сказать, это о том что в функции `main()`, когда вызывался второй конструктор с двумя аргументами, за ним не корректировался стек при помощи `add esp, X`. В то же время, в конце конструктора вместо `RET` имеется `RET 8`.

Это потому что здесь используется thiscall (29.1.1), который, вместе с stdcall (44.2) (все это — методы передачи аргументов через стек), предлагает вызываемой функции корректировать стек. Инструкция `ret X` сначала прибавляет `X` к `ESP`, затем передает управление вызывающей функции.

См. также в соответствующем разделе о способах передачи аргументов через стек (44).

Еще, кстати, нужно отметить, что именно компилятор решает, когда вызывать конструктор и деструктор — но это и так известно из основ языка Си++.

MSVC — x86-64

Как мы уже знаем, в x86-84 первые 4 аргумента ф-ции передаются через регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — через стек. Тем не менее, указатель на объект *this* передается через `RCX`, а первый аргумент метода — в `RDX`, итд. Здесь это видно во внутренностях метода `c(int a, int b)`:

Листинг 29.5: MSVC 2012 x64 /Ox

```
; void dump()

?dump@c@@QEAAXXZ PROC ; c::dump
    mov    r8d, DWORD PTR [rcx+4]
    mov    edx, DWORD PTR [rcx]
    lea    rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@ ; '%d; %d'
    jmp    printf
?dump@c@@QEAAXXZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov    DWORD PTR [rcx], edx   ; 1st argument: a
    mov    DWORD PTR [rcx+4], r8d ; 2nd argument: b
    mov    rax, rcx
    ret    0
??0c@@QEAA@HH@Z ENDP ; c::c
```

```
; default ctor

??0c@@QEAA@XZ PROC ; c::c
    mov    DWORD PTR [rcx], 667
    mov    DWORD PTR [rcx+4], 999
    mov    rax, rcx
    ret    0
??0c@@QEAA@XZ ENDP ; c::c
```

Тип *int* в x64 остается 32-битным [3], поэтому здесь используются 32-битные части регистров.
В методе `dump()` вместо `RET` мы видим `JMP printf`, этот *хак* мы рассматривали раннее: 11.1.1.

GCC — x86

В GCC 4.4.1 всё почти так же, за исключением некоторых различий.

Листинг 29.6: GCC 4.4.1

```
     public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Ev
    mov  [esp+20h+var_18], 6
    mov  [esp+20h+var_1C], 5
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Eii
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    mov  eax, 0
    leave
    retn
main endp
```

Здесь мы видим, что применяется иной *name mangling* характерный для стандартов GNU [4] Во-вторых, указатель на экземпляр передается как первый аргумент функции — конечно же, скрыто от программиста.
Это первый конструктор:

```
                public _ZN1cC1Ev ; weak
_ZN1cC1Ev       proc near               ; CODE XREF: main+10

arg_0           = dword ptr  8

                push    ebp
```

[3] Вероятно, так решили для упрощения портирования Си/Си++-кода на x64
[4] Еще о name mangling разных компиляторов: [12].

```
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax], 667
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax+4], 999
                pop     ebp
                retn
_ZN1cC1Ev       endp
```

Он просто записывает два числа по указателю переданному в первом (и единственном) аргументе.
Второй конструктор:

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_4]
                mov     [eax], edx
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_8]
                mov     [eax+4], edx
                pop     ebp
                retn
_ZN1cC1Eii      endp
```

Это функция, аналог которой мог бы выглядеть так:

```
void ZN1cC1Eii (int *obj, int a, int b)
{
  *obj=a;
  *(obj+1)=b;
};
```

... что, в общем, предсказуемо.
И функция `dump()`:

```
                public _ZN1c4dumpEv
_ZN1c4dumpEv    proc near

var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+arg_0]
                mov     edx, [eax+4]
                mov     eax, [ebp+arg_0]
                mov     eax, [eax]
                mov     [esp+18h+var_10], edx
                mov     [esp+18h+var_14], eax
                mov     [esp+18h+var_18], offset aDD ; "%d; %d\n"
                call    _printf
                leave
                retn
```

```
_ZN1c4dumpEv    endp
```

Эта функция *во внутреннем представлении* имеет один аргумент, через который передается указатель на объект[5] (*this*).

Таким образом, если брать в учет только эти простые примеры, разница между MSVC и GCC в способе кодирования имен функций (*name mangling*) и передаче указателя на экземпляр класса (через `ECX` или через первый аргумент).

GCC — x86-64

Первые 6 аргументов, как мы уже знаем, передаются через 6 регистров `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` [21], а указатель на *this* через первый (`RDI`) что мы здесь и видим. Тип *int* 32-битный и здесь. *Хак* с `JMP` вместо `RET` используется и здесь.

Листинг 29.7: GCC 4.4.6 x64

```
; default ctor

_ZN1cC2Ev:
    mov  DWORD PTR [rdi], 667
    mov  DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_ZN1cC2Eii:
    mov  DWORD PTR [rdi], esi
    mov  DWORD PTR [rdi+4], edx
    ret

; dump()

_ZN1c4dumpEv:
    mov  edx, DWORD PTR [rdi+4]
    mov  esi, DWORD PTR [rdi]
    xor  eax, eax
    mov  edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp  printf
```

29.1.2 Наследование классов

О наследованных классах можно сказать, что это та же простая структура которую мы уже рассмотрели, только расширяемая в наследуемых классах.

Возьмем очень простой пример:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
```

[5] экземпляр класса

```
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, ↙
    ↳ height, depth);
        };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
    s.dump();

    return 0;
};
```

Исследуя сгенерированный код для функций/методов `dump()`, а также `object::print_color()`, посмотрим, какая будет разметка памяти для структур-объектов (для 32-битного кода).

Итак, методы `dump()` разных классов сгенерированные MSVC 2008 с опциями `/Ox` и `/Ob0` [6]

Листинг 29.8: Оптимизирующий MSVC 2008 /Ob0

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; 'string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax

; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
    call _printf
    add  esp, 8
```

[6] опция `/Ob0` означает отмену inline expansion, ведь вставка компилятором тела функции/метода прямо в код где он вызывается только затруднит наши эксперименты

```
    ret  0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

Листинг 29.9: Оптимизирующий MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H ; `string'
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Листинг 29.10: Оптимизирующий MSVC 2008 /Ob0

```
?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
    push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN?$CFd?0?5radius@
    call _printf
    add  esp, 12
    ret  0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

Итак, разметка полей получается следующая:
(базовый класс *object*)

смещение	описание
+0x0	int color

(унаследованные классы)
box:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere:

смещение	описание
+0x0	int color
+0x4	int radius

Посмотрим тело `main()`:

Листинг 29.11: Оптимизирующий MSVC 2008 /Ob0

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub  esp, 24
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+40]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+32]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?dump@box@@QAEXXZ ; box::dump
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?dump@sphere@@QAEXXZ ; sphere::dump
    xor  eax, eax
    add  esp, 24
    ret  0
_main ENDP
```

Наследованные классы всегда должны добавлять свои поля после полей базового класса для того, чтобы методы базового класса могли продолжать работать со своими полями.

Когда метод `object::print_color()` вызывается, ему в качестве `this` передается указатель и на объект типа *box* и на объект типа *sphere*, так как он может легко работать с классами *box* и *sphere*, потому что поле *color* в этих классах всегда стоит по тому же адресу (по смещению `0x0`).

Можно также сказать что методу `object::print_color()` даже не нужно знать, с каким классом он работает, до тех пор, пока будет соблюдаться условие *закрепления* полей по тем же адресам, а это условие соблюдается всегда.

А если вы создадите класс-наследник класса *box*, например, то компилятор будет добавлять новые поля уже за полем *depth*, оставляя уже имеющиеся поля класса *box* по тем же адресам.

Так, метод `box::dump()` будет нормально работать обращаясь к полям *color*/*width*/*height*/*depth* всегда находящимся по известным адресам.

Код на GCC практически точно такой же, за исключением способа передачи `this` (он, как уже было указано, передается в первом аргументе, вместо регистра `ECX`).

29.1.3 Инкапсуляция

Инкапсуляция — это сокрытие данных в *private* секциях класса, например, чтобы разрешить доступ к ним только для методов этого класса, но не более.

Однако, маркируется ли как-нибудь в коде тот факт, что некоторое поле — приватное, а некоторое другое — нет?

Нет, никак не маркируется.

Попробуем простой пример:

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
```

```
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, ↙
    ↳ height, depth);
        };
};
```

Снова скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код метода `box::dump()`:

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$CFd?0?5width?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Разметка полей в классе выходит такой:

смещение	описание
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

Все поля приватные и недоступные для модификации из других функций, но, зная эту разметку, сможем ли мы создать код модифицирующий эти поля?

Для этого я добавил функцию `hack_oop_encapsulation()`, которая если обладает приведенным ниже телом, то просто не скомпилируется:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled:
                // "error C2248: 'box::width' : cannot access private member declared in class ↙
    ↳ 'box'"
};
```

Тем не менее, если преобразовать тип *box* к типу *указатель на массив int*, и если модифицировать полученный массив *int*-ов, тогда всё получится.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
};
```

Код этой функции довольно прост — можно сказать, функция берет на вход указатель на массив *int*-ов и записывает *123* во второй *int*:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; hack_oop_encapsulation
```

Проверим, как это работает:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Запускаем:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

Выходит, инкапсуляция — это защита полей класса только на стадии компиляции. Компилятор Си++ не позволит сгенерировать код прямо модифицирующий защищенные поля, тем не менее, используя *грязные трюки* — это вполне возможно.

29.1.4 Множественное наследование

Множественное наследование — это создание класса наследующего поля и методы от двух или более классов.

Снова напишем простой пример:

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
```

```
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
            return density;
        };
        void dump()
        {
            printf ("this is solid_object. density=%d\n", density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
        void dump()
        {
            printf ("this is solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, ↙
    ↪ height, depth, density);
        };
        int get_weight() { return get_volume() * get_density(); };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};
```

Снова скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код методов `box::dump()`, `solid_object::dump()` `solid_box::dump()`:

Листинг 29.12: Оптимизирующий MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    mov  edx, DWORD PTR [ecx+4]
    push eax
    mov  eax, DWORD PTR [ecx]
    push edx
    push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?$CFd?0?5height?$DN?$CFd@
    call _printf
```

```
    add  esp, 16
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Листинг 29.13: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax
; 'this is solid_object. density=%d', 0aH
    push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5density?$DN?$CFd@
    call _printf
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Листинг 29.14: Оптимизирующий MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d', 0aH
    push OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?$DN?$CFd?0?5hei@
    call _printf
    add  esp, 20
    ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

Выходит, имеем такую разметку в памяти для всех трех классов:
класс *box*:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth

класс *solid_ object*:

смещение	описание
+0x0	density

Можно сказать, что разметка класса *solid_ box* будет *объединённой*:
класс *solid_ box*:

смещение	описание
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

Код методов `box::get_volume()` и `solid_object::get_density()` тривиален:

Листинг 29.15: Оптимизирующий MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
```

```
    mov  eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret  0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Листинг 29.16: Оптимизирующий MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::get_density
```

А вот код метода `solid_box::get_weight()` куда интереснее:

Листинг 29.17: Оптимизирующий MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    push edi
    lea  ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::get_density
    mov  ecx, esi
    mov  edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop  edi
    pop  esi
    ret  0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` просто вызывает два метода, но для `get_volume()` он передает просто указатель на `this`, а для `get_density()`, он передает указатель на `this` сдвинутый на 12 байт (либо `0xC` байт), а там, в разметке класса `solid_box`, как раз начинаются поля класса `solid_object`.

Так, метод `solid_object::get_density()` будет полагать что работает с обычным классом `solid_object`, а метод `box::get_volume()` будет работать только со своими тремя полями, полагая, что работает с обычным экземпляром класса `box`.

Таким образом, можно сказать, что экземпляр класса-наследника нескольких классов представляет в памяти просто *объединённый* класс, содержащий все унаследованные поля. А каждый унаследованный метод вызывается с передачей ему указателя на соответствующую часть структуры.

29.1.5 Виртуальные методы

И снова простой пример:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
```

```
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d, depth=%d\n", color, width, ↙
    ↳ height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

У класса *object* есть виртуальный метод `dump()`, впоследствии заменяемый в классах-наследниках *box* и *sphere*.

Если в какой-то среде, где неизвестно, какого типа является экземпляр класса, как в функции `main()` в примере, вызывается виртуальный метод `dump()`, где-то должна сохраняться информация о том, какой же метод в итоге вызвать.

Скомпилируем в MSVC 2008 с опциями `/Ox` и `/Ob0` и посмотрим код функции `main()`:

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub  esp, 32
    push 30
    push 20
    push 10
    push 1
```

```
    lea  ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov  eax, DWORD PTR _b$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _b$[esp+32]
    call edx
    mov  eax, DWORD PTR _s$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _s$[esp+32]
    call edx
    xor  eax, eax
    add  esp, 32
    ret  0
_main ENDP
```

Указатель на функцию `dump()` берется откуда-то из экземпляра класса (объекта). Где мог записаться туда адрес нового метода-функции? Только в конструкторах, больше негде: ведь в функции `main()` ничего более не вызывается. [7]

Посмотрим код конструктора класса *box*:

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type Descriptor'
    DD    00H
    DB    '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::`RTTI Base Class Descriptor at (0,-1,0,64)'
    DD    01H
    DD    00H
    DD    0ffffffffH
    DD    00H
    DD    040H
    DD    FLAT:??_R3box@@8

??_R2box@@8 DD    FLAT:??_R1A@?0A@EA@box@@8 ; box::`RTTI Base Class Array'
    DD    FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD    00H ; box::`RTTI Class Hierarchy Descriptor'
    DD    00H
    DD    02H
    DD    FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::`RTTI Complete Object Locator'
    DD    00H
    DD    00H
    DD    FLAT:??_R0?AVbox@@@8
    DD    FLAT:??_R3box@@8

??_7box@@6B@ DD    FLAT:??_R4box@@6B@ ; box::`vftable'
    DD    FLAT:?dump@box@@UAEXXZ

_color$ = 8   ; size = 4
_width$ = 12  ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20  ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
    push esi
```

[7] Об указателях на функции читайте больше в соответствующем разделе:(20)

```
    mov  esi, ecx
    call ??0object@@QAE@XZ ; object::object
    mov  eax, DWORD PTR _color$[esp]
    mov  ecx, DWORD PTR _width$[esp]
    mov  edx, DWORD PTR _height$[esp]
    mov  DWORD PTR [esi+4], eax
    mov  eax, DWORD PTR _depth$[esp]
    mov  DWORD PTR [esi+16], eax
    mov  DWORD PTR [esi], OFFSET ??_7box@@6B@
    mov  DWORD PTR [esi+8], ecx
    mov  DWORD PTR [esi+12], edx
    mov  eax, esi
    pop  esi
    ret  16
??0box@@QAE@HHHH@Z ENDP ; box::box
```

Здесь мы видим, что разметка класса немного другая: в качестве первого поля имеется указатель на некую таблицу `box::‘vftable’` (название оставлено компилятором MSVC).

В этой таблице есть ссылка на таблицу с названием `box::‘RTTI Complete Object Locator’` и еще ссылка на метод `box::dump()`. Итак, это называется таблица виртуальных методов и RTTI[8]. Таблица виртуальных методов хранит в себе адреса методов, а RTTI хранит информацию о типах вообще. Кстати, RTTI-таблицы это именно те таблицы, информация из которых используются при вызове *dynamic_ cast* и *typeid* в C++. Вы можете увидеть, что здесь хранится даже имя класса в виде обычной строки. Так, какой-нибудь метод базового класса *object* может вызвать виртуальный метод `object::dump()` что в итоге вызовет нужный метод унаследованного класса, потому что информация о нем присутствует прямо в этой структуре класса.

Работа с этими таблицами и поиск адреса нужного метода, занимает какое-то время процессора, возможно, поэтому считается что работа с виртуальными методами медленна.

В сгенерированном коде от GCC RTTI-таблицы устроены чуть-чуть иначе.

29.2 ostream

Начнем снова с примера типа “hello world”, на этот раз используя ostream:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Из практически любого учебника Си++, известно, что операцию « можно заменить для других типов. Что и делается в ostream. Видно, что в реальности вызывается `operator«` для ostream:

Листинг 29.18: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H

_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

Немного переделаем пример:

```
#include <iostream>
```

[8]Run-time type information

```
int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

И снова, из многих учебников по Си++, известно, что результат каждого `operator<<` в ostream передается в следующий. Действительно:

Листинг 29.19: MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax             ; результат работы предыдущей ф-ции
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?↙
    ↳ $char_traits@D@std@@@0@AAV10@PBD@Z ; std::operator<<<std::char_traits<char> >
    add  esp, 8

    xor  eax, eax
    ret  0
_main ENDP
```

Если заменить `operator<<` на f(), то этот код можно было бы переписать примерно так:

```
f(f(std::cout, "Hello, "), "world!")
```

GCC генерирует практически такой же код как и MSVC.

29.3 References

References в Си++ это тоже указатели (9), но их называют *безопасными* (safe), потому что работая с ними, труднее сделать ошибку [16, 8.3.2]. Например, reference всегда должен указывать объект того же типа и не может быть NULL [6, 8.6]. Более того, reference нельзя менять, нельзя его заставить указывать на другой объект (reseat) [6, 8.5].

Если мы попробуем изменить пример с указателями (9) чтобы он использовал reference вместо указателей:

```
void f2 (int x, int y, int & sum, int & product)
{
        sum=x+y;
        product=x*y;
};
```

То выяснится, что скомпилированный код абсолютно такой же как и в примере с указателями (9):

Листинг 29.20: Оптимизирующий MSVC 2010

```
_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
_sum$ = 16                                              ; size = 4
_product$ = 20                                          ; size = 4
?f2@@YAXHHAAH0@Z PROC                                   ; f2
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul eax, ecx
        mov ecx, DWORD PTR _product$[esp-4]
```

```
        push esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
?f2@@YAXHHAAH0@Z ENDP                                   ; f2
```

(Почему у C++ функций такие странные имена, описано здесь: 29.1.1.)

29.4 STL

N.B.: все примеры здесь были проверены только в 32-битной среде. x64-версии не были проверены.

29.4.1 std::string

Как устроена структура

Многие строковые библиотеки ([37, 2.2]) обеспечивают структуру содержащую ссылку на буфер собственно со строкой, переменная всегда содержащую длину строки (что очень удобно для массы ф-ций [37, 2.2.1]) и переменную содержащую текущий размер буфера. Строка в буфере обыкновенно оканчивается нулем: это для того чтобы указатель на буфер можно было передавать в ф-ции требующие на вход обычную сишную ASCIIZ-строку.

Стандарт Си++ ([16]) не описывает, как именно нужно реализовывать std::string, но как правило они реализованы как описано выше, с небольшими дополнениями.

По стандарту, std::string это не класс (как, например, QString в Qt), а темплейт, это сделано для того чтобы поддерживать строки содержащие разного типа символы: как минимум char и wchar_t.

Здесь пока не будет листингов на ассемблере, потому что проиллюстрировать внутренности std::string в MSVC и GCC можно и без этого.

MSVC В реализации MSVC, вместо ссылки на буфер может содержаться сам буфер (если строка короче 16-и символов).

Это означает что каждая короткая строка будет занимать в памяти по крайней мере 16 + 4 + 4 = 24 байт для 32-битной среды либо 16 + 8 + 8 = 32 байта в 64-битной, а если строка длиннее 16-и символов, то прибавьте еще длину самой строки.

Листинг 29.21: пример для MSVC

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
    {
        char buf[16];
        char* ptr;
    } u;
    size_t size;     // AKA 'Mysize' in MSVC
    size_t capacity; // AKA 'Myres' in MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr : p->u.buf, p->size, p->↙
    ↳ capacity);
};

int main()
{
    std::string s1="short string";
```

```
    std::string s2="string longer that 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // that works without using c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};
```

Собственно, из этого исходника почти всё ясно.

Несколько замечаний:

Если строка короче 16-и символов, то отдельный буфер для строки в куче выделяться не будет. Это удобно потому что на практике, действительно немало строк короткие. Вероятно, разработчики в Microsoft выбрали размер в 16 символов как разумный баланс.

Теперь очень важный момент в конце ф-ции main(): я не пользуюсь методом c_str(), тем не менее, если это скомпилировать и запустить, то обе строки появятся в консоли!

Работает это вот почему.

В первом случае строка короче 16-и символов и в начале объекта std::string (его можно рассматривать просто как структуру) расположен буфер с этой строкой. printf() трактует указатель как указатель на массив символов оканчивающийся нулем и поэтому всё работает.

Вывод второй строки (длиннее 16-и символов) даже еще опаснее: это вообще типичная программистская ошибка (или опечатка), забыть дописать c_str(). Это работает потому что в это время в начале структуры расположен указатель на буфер. Это может надолго остаться незамеченным: до тех пока там не появится строка короче 16-и символов, тогда процесс упадет.

GCC В реализации GCC в структуре есть еще одна переменная — reference count.

Интересно, что указатель на экземпляр класса std::string в GCC указывает не на начало самой структуры, а на указатель на буфера. В libstdc++-v3\include\bits\basic_string.h мы можем прочитать что это сделано для удобства отладки:

```
   *  The reason you want _M_data pointing to the character %array and
   *  not the _Rep is so that the debugger can see the string
   *  contents. (Probably we should add a non-inline member to get
   *  the _Rep for the debugger to use, so users can check the actual
   *  string length.)
```

исходный код basic_string.h

В моем примере я учитываю это:

Листинг 29.22: пример для GCC

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string));
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer that 16 bytes";
```

```
    dump_std_string(s1);
    dump_std_string(s2);

    // GCC type checking workaround:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

Нужны еще небольшие хаки чтобы сымитировать типичную ошибку, о которой я уже написал, из-за более ужесточенной проверки типов в GCC, тем не менее, printf() работает и здесь без c_str().

Чуть более сложный пример

```
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Листинг 29.23: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub  esp, 72

    push 7
    push OFFSET $SG39512
    lea  ecx, DWORD PTR _s1$[esp+80]
    mov  DWORD PTR _s1$[esp+100], 15
    mov  DWORD PTR _s1$[esp+96], 0
    mov  BYTE PTR _s1$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ;↙
    ↳  std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    push 7
    push OFFSET $SG39514
    lea  ecx, DWORD PTR _s2$[esp+80]
    mov  DWORD PTR _s2$[esp+100], 15
    mov  DWORD PTR _s2$[esp+96], 0
    mov  BYTE PTR _s2$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ;↙
    ↳  std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign

    lea  eax, DWORD PTR _s2$[esp+72]
    push eax
    lea  eax, DWORD PTR _s1$[esp+76]
    push eax
    lea  eax, DWORD PTR _s3$[esp+80]
```

```
    push eax
    call ??$?HDU?$char_traits@D@std@@V?$allocator@D@1@@std@@YA?AV?$basic_string@DU?↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@0@ABV10@0@Z ; std::operator+<char,std::char_traits<↙
    ↳ char>,std::allocator<char> >

    ; inlined c_str() method:
    cmp  DWORD PTR _s3$[esp+104], 16
    lea  eax, DWORD PTR _s3$[esp+84]
    cmovae eax, DWORD PTR _s3$[esp+84]

    push eax
    push OFFSET $SG39581
    call _printf
    add  esp, 20

    cmp  DWORD PTR _s3$[esp+92], 16
    jb   SHORT $LN119@main
    push DWORD PTR _s3$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN119@main:
    cmp  DWORD PTR _s2$[esp+92], 16
    mov  DWORD PTR _s3$[esp+92], 15
    mov  DWORD PTR _s3$[esp+88], 0
    mov  BYTE PTR _s3$[esp+72], 0
    jb   SHORT $LN151@main
    push DWORD PTR _s2$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN151@main:
    cmp  DWORD PTR _s1$[esp+92], 16
    mov  DWORD PTR _s2$[esp+92], 15
    mov  DWORD PTR _s2$[esp+88], 0
    mov  BYTE PTR _s2$[esp+72], 0
    jb   SHORT $LN195@main
    push DWORD PTR _s1$[esp+72]
    call ??3@YAXPAX@Z               ; operator delete
    add  esp, 4
$LN195@main:
    xor  eax, eax
    add  esp, 72
    ret  0
_main ENDP
```

Собственно, компилятор не конструирует строки статически: да в общем-то и как это возможно, если буфер с ней нужно хранить в куче? Вместо этого в сегменте данных хранятся обычные ASCIIZ-строки, а позже, во время выполнения, при помощи метода “assign”, конструируются строки s1 и s2. При помощи `operator+`, создается строка s3.

Обратите внимание на то что вызов метода c_str() отсутствует, потому что его код достаточно короткий и компилятор вставил его прямо здесь: если строка короче 16-и байт, то в регистре EAX остается указатель на буфер, а если длиннее, то из этого же места достается адрес на буфер расположенный в куче.

Далее следуют вызовы трех деструкторов, причем, они вызываются только если строка длиннее 16-и байт: тогда нужно освободить буфера в куче. В противном случае, так как все три объекта std::string хранятся в стеке, они освобождаются автоматически после выхода из ф-ции.

Следовательно, работа с короткими строками более быстрая из-за мѐньшего обращения к куче.

Код на GCC даже проще (из-за того, что в GCC, как я уже указывал, не реализована возможность хранить короткую строку прямо в структуре):

Листинг 29.24: GCC 4.8.1

```
.LC0:
    .string "Hello, "
```

```
.LC1:
    .string "world!\n"
main:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea  ebx, [esp+28]
    lea  edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea  esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

    ; inlined c_str():
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts

    mov  eax, DWORD PTR [esp+28]
    lea  ebx, [esp+19]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+24]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+20]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    lea  esp, [ebp-12]
    xor  eax, eax
    pop  ebx
    pop  esi
```

```
    pop  edi
    pop  ebp
    ret
```

Можно заметить, что в деструкторы передается не указатель на объект, а указатель на место за 12 байт (или 3 слова) перед ним, то есть, на настоящее начало структуры.

std::string как глобальная переменная

Опытные программисты на Си++ могут возразить: глобальные переменные STL[9]-типов вполне можно объявлять.

Да, действительно:

```
#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

Листинг 29.25: MSVC 2012

```
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    mov  eax, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP

??__Es@@YAXXZ PROC ; `dynamic initializer for 's'', COMDAT
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov  ecx, OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV12@PBDI@Z ;↙
    ↳  std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??__Fs@@YAXXZ ; `dynamic atexit destructor for 's''
    call _atexit
    pop  ecx
    ret  0
??__Es@@YAXXZ ENDP ; `dynamic initializer for 's''

??__Fs@@YAXXZ PROC ; `dynamic atexit destructor for 's'', COMDAT
    push ecx
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 16
    jb   SHORT $LN23@dynamic
    push esi
    mov  esi, DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A
    lea  ecx, DWORD PTR $T2[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ ; std::_Wrap_alloc<std::allocator<↙
    ↳ char> >::_Wrap_alloc<std::allocator<char> >
```

[9](C++) Standard Template Library: 29.4

```
    push OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    lea  ecx, DWORD PTR $T2[esp+12]
    call ??$destroy@PAD@?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAEXPAPAD@Z ; std::_Wrap_alloc<↙
    ↳ std::allocator<char> >::destroy<char *>
    lea  ecx, DWORD PTR $T1[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ ; std::_Wrap_alloc<std::allocator<↙
    ↳ char> >::_Wrap_alloc<std::allocator<char> >
    push esi
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    pop  esi
$LN23@dynamic:
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+20, 15
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A+16, 0
    mov  BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@A, 0
    pop  ecx
    ret  0
??__Fs@@YAXXZ ENDP ; 'dynamic atexit destructor for 's''
```

В реальности, из CRT, еще до вызова main(), вызывается специальная ф-ция, в которой перечислены все конструкторы подобных переменных. Более того: при помощи atexit() регистрируется ф-ция, которая будет вызвана в конце работы программы: в этой ф-ции компилятор собирает деструкторы всех подобных глобальных переменных.

GCC работает похожим образом:

Листинг 29.26: GCC 4.8.1

```
main:
    push ebp
    mov  ebp, esp
    and  esp, -16
    sub  esp, 16
    mov  eax, DWORD PTR s
    mov  DWORD PTR [esp], eax
    call puts
    xor  eax, eax
    leave
    ret
.LC0:
    .string "a string"
_GLOBAL__sub_I_s:
    sub  esp, 44
    lea  eax, [esp+31]
    mov  DWORD PTR [esp+8], eax
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], OFFSET FLAT:s
    call _ZNSsC1EPKcRKSaIcE
    mov  DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
    mov  DWORD PTR [esp+4], OFFSET FLAT:s
    mov  DWORD PTR [esp], OFFSET FLAT:_ZNSsD1Ev
    call __cxa_atexit
    add  esp, 44
    ret
.LFE645:
    .size  _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long  _GLOBAL__sub_I_s
    .globl s
    .bss
    .align 4
    .type  s, @object
```

```
    .size  s, 4
s:
    .zero  4
    .hidden __dso_handle
```

Он даже не выделяет отдельной ф-ции в которой будут собраны деструкторы: каждый деструктор передается в atexit() по одному.

29.4.2 std::list

Хорошо известный всем двусвязный список: каждый элемент имеет два указателя, на следующий и на предыдущий элементы.

Это означает что расход памяти увеличивается на 2 слова на каждый элемент (8 байт в 32-битной среде или 16 байт в 64-битной).

Это также циркулярный список, что означает что последний элемент имеет указатель на первый и наоборот: первый имеет указатель на последний.

C++ STL просто добавляет указатели "next" и "previous" к той вашей структуре, которую вы желаете объединить в список.

Попробуем разобраться с примером в котором простая структура из двух переменных, мы объеденим её в список.

Хотя и стандарт Си++ [16] не предлагает, как он должен быть реализован, реализации MSVC и GCC прямолинейны и похожи друг на друга, так что этот исходный код для обоих:

```
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
        n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
```

```
#ifdef _MSC_VER
    // GCC implementation doesn't have "size" field
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
    printf ("node at .end:\n");
    tmp=l.end();
    dump_List_node ((struct List_node *)*(void**)&tmp);

    printf ("* let's count from the begin:\n");
    std::list<struct a>::iterator it=l.begin();
    printf ("1st element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("2nd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("3rd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

    printf ("* let's count from the end:\n");
    std::list<struct a>::iterator it2=l.end();
    printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

    printf ("removing last element...\n");
    l.pop_back();
    dump_List_val((unsigned int*)(void*)&l);
};
```

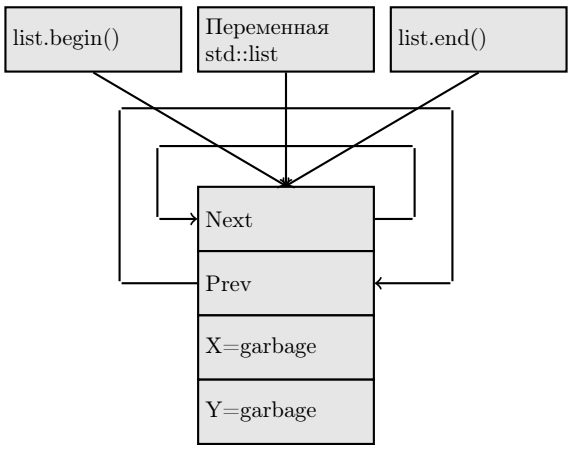
GCC

Начнем с GCC.

При запуске увидим длинный вывод, будем разбирать его по частям.

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

Видим пустой список. Не смотря на то что он пуст, имеется один элемент с мусором в переменных x и y. Оба указателя "next" и "prev" указывают на себя:



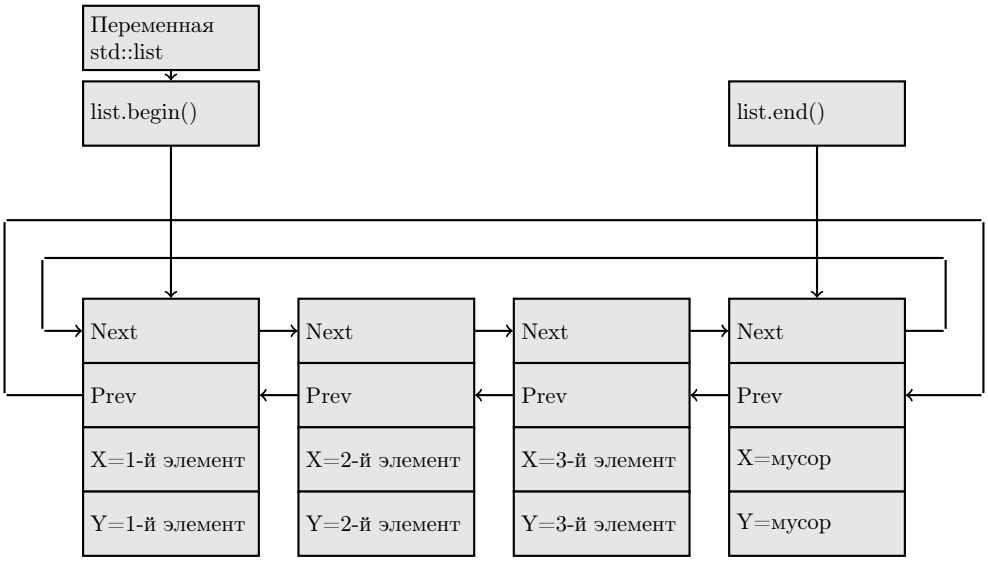
Это тот момент, когда итераторы .begin и .end равны друг другу.

Вставим 3 элемента и список в памяти будет представлен так:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Последний элемент всё еще на 0x0028fe90, он не будет передвинут куда-либо до самого уничтожения списка. Он все еще содержит случайный мусор в полях x и y (5 и 6). Случайно совпало так, что эти значения точно такие же как и в последнем элементе, но это не значит, что они имеют какое-то значение.

Вот как эти 3 элемента хранятся в памяти:



Переменная l всегда указывает на первый элемент.

Итераторы .begin() и .end() ни на что не указывают, и вообще отсутствуют в памяти, но указатели на эти элементы будут возвращены, когда соответствующие методы будут вызваны.

Иметь элемент с "мусором" это очень популярная практика в реализации двусвязных списков. Без него, многие операции были бы сложнее, и, следовательно, медленнее.

Итератор на самом деле это просто указатель на элемент. list.begin() и list.end() просто возвращают указатели.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

Тот факт что список циркулярный, очень помогает: если иметь указатель только на первый элемент, т.е., тот что в переменной l, очень легко получить указатель на последний элемент, без необходимости обходить все элементы списка. Вставка элемента в конец списка также быстра благодаря этой особенности.

operator- и operator++ просто выставляют текущее значение итератора на current_node->prev или current_node->n
Обратные итераторы (.rbegin, .rend) работают точно так же, только наоборот.

operator* на итераторе просто возвращает указатель на место в структуре, где начинается пользовательская структура, т.е., указатель на самый первый элемент структуры (x).

Вставка в список и удаление очень просты: просто выделите новый элемент (или освободите) и исправьте все указатели так, чтобы они были верны.

Вот почему итератор может стать недействительным после удаления элемента: он может всё еще указывать на уже освобожденный элемент. И конечно же, информация из освобожденного элемента, на который указывает итератор, не может использоваться более.

В реализации GCC (по крайней мере 4.8.1) не сохраняется текущая длина списка: это выливается в медленный метод .size(): он должен пройти по всему списку считая элементы, просто потому что нет другого способа получить эту информацию. Это означает что эта операция $O(n)$, т.е., она работает тем медленнее, чем больше элементов в списке.

Листинг 29.27: GCC 4.8.1 -O3 -fno-inline-small-functions

```
main proc near
    push ebp
    mov  ebp, esp
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  ebx, [esp+10h]
    mov  dword ptr [esp], offset s ; "* empty list:"
    mov  [esp+10h], ebx
    mov  [esp+14h], ebx
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea  esi, [esp+18h]
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 1 ; X for new element
    mov  dword ptr [esp+1Ch], 2 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a↙
    ↳  const&)
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 3 ; X for new element
    mov  dword ptr [esp+1Ch], 4 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,std::allocator<a>>::push_front(a↙
    ↳  const&)
    mov  dword ptr [esp], 10h
    mov  dword ptr [esp+18h], 5 ; X for new element
    mov  dword ptr [esp+1Ch], 6 ; Y for new element
    call _Znwj           ; operator new(uint)
    cmp  eax, 0FFFFFFF8h
    jz   short loc_80002A6
    mov  ecx, [esp+1Ch]
    mov  edx, [esp+18h]
    mov  [eax+0Ch], ecx
    mov  [eax+8], edx
```

```
loc_80002A6: ; CODE XREF: main+86
    mov  [esp+4], ebx
    mov  [esp], eax
    call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::__detail::_List_node_base::_M_hook↙
    ↳ (std::__detail::_List_node_base*)
    mov  dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
    call puts
    mov  eax, [esp+10h]
    mov  [esp], eax
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
    mov  dword ptr [esp], offset aNodeAt_end ; "node at .end:"
    call puts
    mov  [esp], ebx
    call _Z14dump_List_nodeP9List_node ; dump_List_node(List_node *)
    mov  dword ptr [esp], offset aLetSCountFromT ; "* let's count from the begin:"
    call puts
    mov  esi, [esp+10h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi] ; operator++: get ->next pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the end:"
    call puts
    mov  eax, [esp+1Ch]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+0Ch], eax
    mov  eax, [esp+18h]
    mov  [esp+8], eax
    call __printf_chk
```

```
    mov  esi, [esp+14h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi+4] ; operator--: get ->prev pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi+4] ; operator--: get ->prev pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aRemovingLastEl ; "removing last element..."
    call puts
    mov  esi, [esp+14h]
    mov  [esp], esi
    call _ZNSt8__detail15_List_node_base9_M_unhookEv ; std::__detail::_List_node_base::↙
    ↳ _M_unhook(void)
    mov  [esp], esi ; void *
    call _ZdlPv ; operator delete(void *)
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  [esp], ebx
    call _ZNSt10_List_baseI1aSaIS0_EE8_M_clearEv ; std::_List_base<a,std::allocator<a>>::↙
    ↳ _M_clear(void)
    lea  esp, [ebp-8]
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  ebp
    retn
main endp
```

Листинг 29.28: Весь вывод

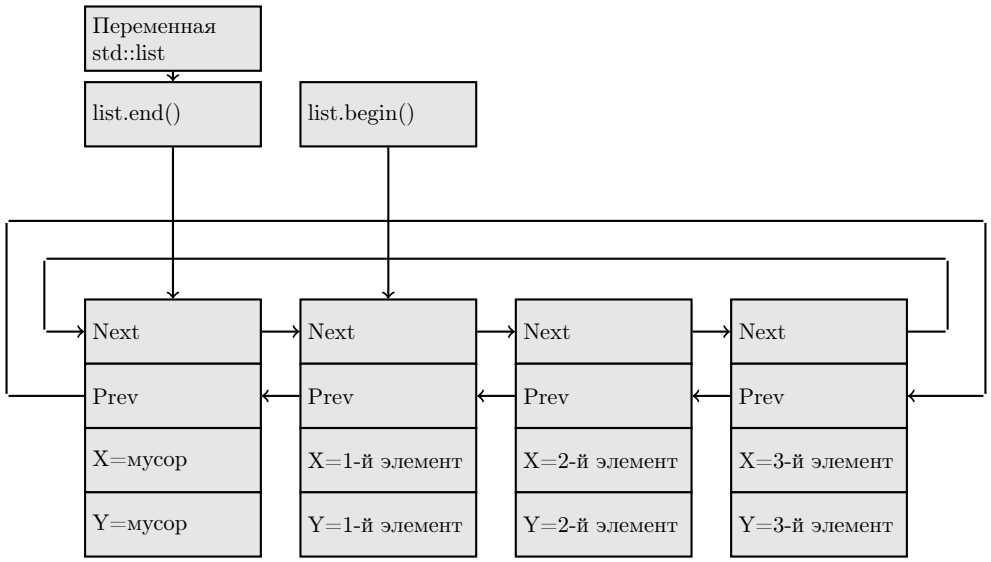
```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
```

```
3rd element: 5 6
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

MSVC

Реализация MSVC (2012) точно такая же, только еще и сохраняет текущий размер списка. Это означает что метод .size() очень быстр ($O(1)$): просто прочитать одно значение из памяти. С другой стороны, переменная хранящая размер должна корректироваться при каждой вставке/удалении.

Реализация MSVC также немного отлична в смысле расстановки элементов:



У GCC его элемент с "мусором" в самом конце списка, а у MSVC в самом начале.

Листинг 29.29: MSVC 2012 /Fa2.asm /Ox /GS- /Ob1

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main    PROC
    sub  esp, 16
    push ebx
    push esi
    push edi
    push 0
    push 0
    lea  ecx, DWORD PTR _l$[esp+36]
    mov  DWORD PTR _l$[esp+40], 0
    ; allocate first "garbage" element
    call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V?↙
    ↳ $allocator@Ua@@@std@@@std@@@std@@QAEPAU?$_List_node@Ua@@PAX@2@PAU32@0@Z ; std::↙
    ↳ _List_alloc<0,std::_List_base_types<a,std::allocator<a> > >::_Buynode0
    mov  edi, DWORD PTR __imp__printf
    mov  ebx, eax
    push OFFSET $SG40685 ; '* empty list:'
    mov  DWORD PTR _l$[esp+32], ebx
    call edi  ; printf
    lea  eax, DWORD PTR _l$[esp+32]
```

```
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  esi, DWORD PTR [ebx]
    add  esp, 8
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push esi
    mov  DWORD PTR _t1$[esp+40], 1 ; data for a new node
    mov  DWORD PTR _t1$[esp+44], 2 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU?↙
    ↪ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
    ↪ const &>
    mov  DWORD PTR [esi+4], eax
    mov  ecx, DWORD PTR [eax+4]
    mov  DWORD PTR _t1$[esp+28], 3 ; data for a new node
    mov  DWORD PTR [ecx], eax
    mov  esi, DWORD PTR [ebx]
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push esi
    mov  DWORD PTR _t1$[esp+44], 4 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU?↙
    ↪ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
    ↪ const &>
    mov  DWORD PTR [esi+4], eax
    mov  ecx, DWORD PTR [eax+4]
    mov  DWORD PTR _t1$[esp+28], 5 ; data for a new node
    mov  DWORD PTR [ecx], eax
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [ebx+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push ebx
    mov  DWORD PTR _t1$[esp+44], 6 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?$allocator@Ua@@@std@@@std@@QAEPAU?↙
    ↪ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,std::allocator<a> >::_Buynode<a ↙
    ↪ const &>
    mov  DWORD PTR [ebx+4], eax
    mov  ecx, DWORD PTR [eax+4]
    push OFFSET $SG40689 ; '* 3-elements list:'
    mov  DWORD PTR _l$[esp+36], 3
    mov  DWORD PTR [ecx], eax
    call edi ; printf
    lea  eax, DWORD PTR _l$[esp+32]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    push OFFSET $SG40831 ; 'node at .begin:'
    call edi ; printf
    push DWORD PTR [ebx] ; get next field of node $l$ variable points to
    call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
    push OFFSET $SG40835 ; 'node at .end:'
    call edi ; printf
    push ebx ; pointer to the node $l$ variable points to!
    call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
```

```
    push OFFSET $SG40839 ; '* let''s count from the begin:'
    call edi ; printf
    mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40846 ; '1st element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40848 ; '2nd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40850 ; '3rd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
    add  esp, 64
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40852 ; 'element at .end(): %d %d'
    call edi ; printf
    push OFFSET $SG40853 ; '* let''s count from the end:'
    call edi ; printf
    push DWORD PTR [ebx+12] ; use x and y fields from the node $1$ variable points to
    push DWORD PTR [ebx+8]
    push OFFSET $SG40860 ; 'element at .end(): %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40862 ; '3rd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40864 ; '2nd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40866 ; '1st element: %d %d'
    call edi ; printf
    add  esp, 64
    push OFFSET $SG40867 ; 'removing last element...'
    call edi ; printf
    mov  edx, DWORD PTR [ebx+4]
    add  esp, 4

    ; prev=next?
    ; it is the only element, "garbage one"?
    ; if yes, do not delete it!
    cmp  edx, ebx
    je   SHORT $LN349@main
    mov  ecx, DWORD PTR [edx+4]
    mov  eax, DWORD PTR [edx]
    mov  DWORD PTR [ecx], eax
    mov  ecx, DWORD PTR [edx]
    mov  eax, DWORD PTR [edx+4]
    push edx
```

```
    mov  DWORD PTR [ecx+4], eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  DWORD PTR _l$[esp+32], 2
$LN349@main:
    lea  eax, DWORD PTR _l$[esp+28]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  eax, DWORD PTR [ebx]
    add  esp, 4
    mov  DWORD PTR [ebx], ebx
    mov  DWORD PTR [ebx+4], ebx
    cmp  eax, ebx
    je   SHORT $LN412@main
$LL414@main:
    mov  esi, DWORD PTR [eax]
    push eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  eax, esi
    cmp  esi, ebx
    jne  SHORT $LL414@main
$LN412@main:
    push  ebx
    call  ??3@YAXPAX@Z ; operator delete
    add   esp, 4
    xor   eax, eax
    pop   edi
    pop   esi
    pop   ebx
    add   esp, 16
    ret   0
_main ENDP
```

В отличие от GCC, код MSVC выделяет элемент с “мусором” в самом начале ф-ции при помощи ф-ции “Buynode”, она также используется и во время выделения остальных элементов (код GCC выделяет самый первый элемент в локальном стеке).

Листинг 29.30: Весь вывод

```
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y=4522072
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
```

```
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y=4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list

Это то же самое что и std::list, но только односвязный список, т.е., имеющий только поле "next" в каждом элементе. Таким образом расход памяти меньше, но возможности идти по списку назад здесь нет.

29.4.3 std::vector

Я бы назвал `std::vector` "безопасной оболочкой (wrapper)" PODT[10] массива в Си. Изнутри он очень похож на `std::string` (29.4.1): он имеет указатель на буфер, указатель на конец массива и указатель на конец буфера.

Элементы массива просто лежат в памяти впритык друг к другу, так же, как и в обычном массиве (16). В C++11 появился метод `.data()` возвращающий указатель на этот буфер, это похоже на `.c_str()` в `std::string`.

Выделенный буфер в куче может быть больше чем сам массив.

Реализации MSVC и GCC почти одинаковые, отличаются только имена полей в структуре[11], так что здесь один исходник работающий для обоих компиляторов. И снова здесь Си-подобный код для вывода структуры `std::vector`:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // GCC structure is the same, names are: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst, in->Mylast, in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};

int main()
{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
```

[10](C++) Plain Old Data Type
[11]внутренности GCC: http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01371.html

```
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // bounds checking
    printf ("%d\n", c[8]); // operator[], no bounds checking
};
```

Примерный вывод программы скомпилированной в MSVC:

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

Как можно заметить, выделенного буфера в самом начале ф-ции `main()` пока нет. После первого вызова `push_back()` буфер выделяется. И далее, после каждого вызова `push_back()` и длина массива и вместимость буфера (*capacity*) увеличиваются. Но адрес буфера также меняется, потому что вызов ф-ции `push_back()` перевыделяет буфер в куче каждый раз. Это дорогая операция, вот почему очень важно предсказать размер будущего массива и зарезервировать место для него при помощи метода `.reserve()`. Самое последнее число

это мусор: там нет элементов массива в этом месте, вот откуда это случайное число. Это иллюстрация того факта что метод `operator[]` в `std::vector` не проверяет индекс на правильность. Метод `.at()` с другой стороны, проверяет, и подкидывает исключение `std::out_of_range` в случае ошибки.

Давайте посмотрим код:

Листинг 29.31: MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
__Pos$ = 8  ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::vector<int,std::allocator<int> ↙
    ↳ >::at, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _this$[ebp]
    mov  edx, DWORD PTR [eax+4]
    sub  edx, DWORD PTR [ecx]
    sar  edx, 2
    cmp  edx, DWORD PTR __Pos$[ebp]
    ja   SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPPO@invalid?5vector?$DMT?$DO?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z
$LN1@at:
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax]
    mov  edx, DWORD PTR __Pos$[ebp]
    lea  eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov  esp, ebp
    pop  ebp
    ret  4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int,std::allocator<int> ↙
    ↳ >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8  ; size = 4
$T6 = -4  ; size = 4
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 36
    mov  DWORD PTR _c$[ebp], 0     ; Myfirst
    mov  DWORD PTR _c$[ebp+4], 0   ; Mylast
    mov  DWORD PTR _c$[ebp+8], 0   ; Myend
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T6[ebp], 1
    lea  ecx, DWORD PTR $T6[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
```

```
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T5[ebp], 2
    lea  eax, DWORD PTR $T5[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T4[ebp], 3
    lea  edx, DWORD PTR $T4[ebp]
    push edx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T3[ebp], 4
    lea  ecx, DWORD PTR $T3[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 6
    lea  ecx, DWORD PTR _c$[ebp]
    call ?reserve@?$vector@HV?$allocator@H@std@@@std@@QAEXI@Z ; std::vector<int,std::allocator<↙
    ↪ int> >::reserve
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T2[ebp], 5
    lea  ecx, DWORD PTR $T2[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T1[ebp], 6
    lea  eax, DWORD PTR $T1[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?$allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std::↙
    ↪ allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
```

```
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 5
    lea  ecx, DWORD PTR _c$[ebp]
    call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ; std::vector<int,std::allocator<int ↙
    ↳ > >::at
    mov  edx, DWORD PTR [eax]
    push edx
    push OFFSET $SG52650 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    mov  eax, 8
    shl  eax, 2
    mov  ecx, DWORD PTR _c$[ebp]
    mov  edx, DWORD PTR [ecx+eax]
    push edx
    push OFFSET $SG52651 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    lea  ecx, DWORD PTR _c$[ebp]
    call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ ; std::vector<int,std::allocator<int ↙
    ↳ > >::_Tidy
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

Мы видим как метод `.at()` проверяет границы и подкидывает исключение в случае ошибки. Число, которое выводит последний вызов `printf()` берется из памяти, без всяких проверок.

Читатель может спросить, почему бы не использовать переменные "size" и "capacity", как это сделано в `std::string`. Я подозреваю что это для более быстрой проверки границ. Но я не уверен.

Код генерируемый GCC почти такой же, в целом, но метод `.at()` вставлен прямо в код:

Листинг 29.32: GCC 4.8.1 -fno-inline-small-functions -O1

```
main proc near
     push ebp
     mov  ebp, esp
     push edi
     push esi
     push ebx
     and  esp, 0FFFFFFF0h
     sub  esp, 20h
     mov  dword ptr [esp+14h], 0
     mov  dword ptr [esp+18h], 0
     mov  dword ptr [esp+1Ch], 0
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 1
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back( ↙
    ↳ int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 2
```

```
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(↙
   ↳ int  const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 3
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(↙
   ↳ int  const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  dword ptr [esp+10h], 4
    lea  eax, [esp+10h]
    mov  [esp+4], eax
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(↙
   ↳ int  const&)
    lea  eax, [esp+14h]
    mov  [esp], eax
    call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
    mov  ebx, [esp+14h]
    mov  eax, [esp+1Ch]
    sub  eax, ebx
    cmp  eax, 17h
    ja   short loc_80001CF
    mov  edi, [esp+18h]
    sub  edi, ebx
    sar  edi, 2
    mov  dword ptr [esp], 18h
    call _Znwj           ; operator new(uint)
    mov  esi, eax
    test edi, edi
    jz   short loc_80001AD
    lea  eax, ds:0[edi*4]
    mov  [esp+8], eax    ; n
    mov  [esp+4], ebx    ; src
    mov  [esp], esi      ; dest
    call memmove

loc_80001AD: ; CODE XREF: main+F8
    mov  eax, [esp+14h]
    test eax, eax
    jz   short loc_80001BD
    mov  [esp], eax      ; void *
    call _ZdlPv          ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
    mov  [esp+14h], esi
    lea  eax, [esi+edi*4]
    mov  [esp+18h], eax
    add  esi, 18h
    mov  [esp+1Ch], esi
```

```
loc_80001CF: ; CODE XREF: main+DD
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 5
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(↙
    ↳ int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 6
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,std::allocator<int>>::push_back(↙
    ↳ int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  eax, [esp+14h]
     mov  edx, [esp+18h]
     sub  edx, eax
     cmp  edx, 17h
     ja   short loc_8000246
     mov  dword ptr [esp], offset aVector_m_range ; "vector::_M_range_check"
     call _ZSt20__throw_out_of_rangePKc ; std::__throw_out_of_range(char  const*)

loc_8000246:                            ; CODE XREF: main+19C
     mov  eax, [eax+14h]
     mov  [esp+8], eax
     mov  dword ptr [esp+4], offset aD ; "%d\n"
     mov  dword ptr [esp], 1
     call __printf_chk
     mov  eax, [esp+14h]
     mov  eax, [eax+20h]
     mov  [esp+8], eax
     mov  dword ptr [esp+4], offset aD ; "%d\n"
     mov  dword ptr [esp], 1
     call __printf_chk
     mov  eax, [esp+14h]
     test eax, eax
     jz   short loc_80002AC
     mov  [esp], eax      ; void *
     call _ZdlPv          ; operator delete(void *)
     jmp  short loc_80002AC

     mov  ebx, eax
     mov  edx, [esp+14h]
     test edx, edx
     jz   short loc_80002A4
     mov  [esp], edx      ; void *
     call _ZdlPv          ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
     mov  [esp], ebx
```

```
    call _Unwind_Resume


loc_80002AC: ; CODE XREF: main+1EA
             ; main+1F4
    mov  eax, 0
    lea  esp, [ebp-0Ch]
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
                ; .eh_frame:080005BC
    retn
main endp
```

Метод `.reserve()` точно так же вставлен прямо в код `main()`. Он вызывает `new()` если буфер слишком мал для нового массива, вызывает `memmove()` для копирования содержимого буфера, и вызывает `delete()` для освобождения старого буфера.

Посмотрим, что выводит программа будучи скомпилированная GCC:

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
```

```
element 5: 6
6
0
```

Мы можем заметить, что буфер растет иначе чем в MSVC.

При помощи простых экспериментов становится ясно, что в реализации MSVC буфер увеличивается на ~50% каждый раз, когда он должен был увеличен, а у GCC он увеличивается на 100% каждый раз, т.е., удваивается.

29.4.4 std::map и std::set

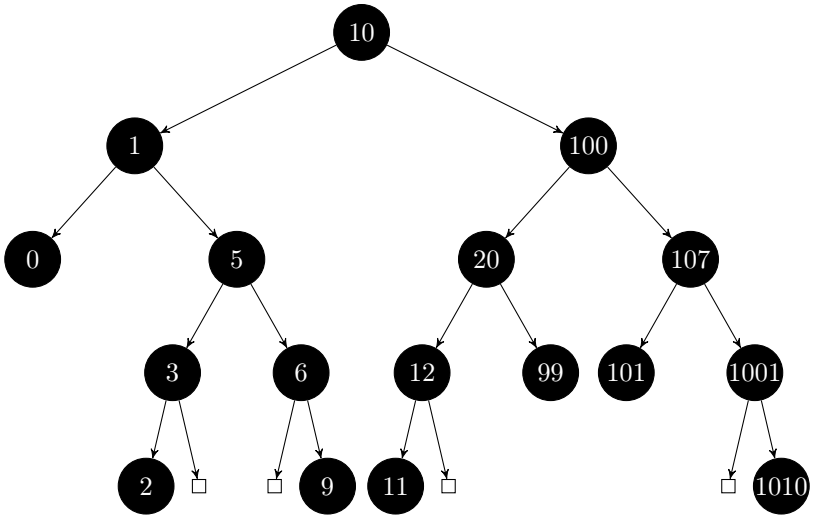
Двоичное дерево — это еще одна фундаментальная структура данных. Как следует из названия, это дерево, но у каждого узла максимум 2 связи с другими узлами. Каждый узел имеет ключ и/или значение.

Обычно, именно при помощи двоичных деревьев реализуются "словари" пар ключ-значения (AKA "ассоциативные массивы").

Двоичные деревья имеют по крайней мере три важных свойства:

- Все ключи всегда хранятся в отсортированном виде.
- Могут хранится ключи любых типов. Алгоритмы для работы с двоичными деревьями не зависят от типа ключа, для работы им нужна только ф-ция для сравнения ключей.
- Поиск необходимого ключа относительно быстрый по сравнению со списками или массивами.

Очень простой пример: давайте сохраним вот эти числа в двоичном дереве: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



Все ключи меньше чем значение ключа узла, сохраняются по левой стороне. Все ключи больше чем значение ключа узла, сохраняются по правой стороне.

Таким образом, алгоритм для поиска нужного ключа прост: если искомое значение меньше чем значение текущего узла: двигаемся влево, если больше: двигаемся вправо, останавливаемся если они равны. Таким образом, алгоритм может искать числа, текстовые строки, и т.д., только при помощи ф-ции сравнения ключей.

Все ключи имеют уникальные значения.

Учитывая это, нужно $\approx \log_2 n$ шагов для поиска ключа в сбалансированном дереве, содержащем n ключей. Это ≈ 10 шагов для ≈ 1000 ключей, или ≈ 13 шагов для ≈ 10000 ключей. Неплохо, но для этого дерево всегда должно быть сбалансировано: т.е., ключи должны быть равномерно распределены на всех ярусах. Операции вставки и удаления проводят дополнительную работу по обслуживанию дерева и сохранения его в сбалансированном состоянии.

Известно несколько популярных алгоритмом балансировки, включая AVL-деревья и красно-черные деревья. Последний дополняет узел значением "цвета" для упрощения балансировки, таким образом каждый узел может быть "красным" или "черным".

Реализации `std::map` и `std::set` обоих GCC и MSVC используют красно-черные деревья.

`std::set` содержит только ключи. `std::map` это "расширенная" версия set: здесь имеется еще и значение (value) на каждом узле.

MSVC

```
#include <map>
#include <set>
#include <string>
#include <iostream>

// structure is not packed!
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d Isnil=%d\n",
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnil);
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->second);
    }

    if (traverse)
    {
        if (n->Isnil==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnil==0)
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnil==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
```

```
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnil==0)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
```

```
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false);
};
```

Листинг 29.33: MSVC 2012

```
dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0x005BB3A0 Color=0 Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0x005BB3A0 Color=1 Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0x005BB3A0 Color=0 Isnil=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
```

```
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0x005BB5E0 Color=0 Isnil=0
first=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=1
```

Структура не запакована, так что оба значения типа *char* занимают по 4 байта.

В `std::map`, `first` и `second` могут быть представлены как одно значение типа `std::pair`. `std::set` имеет только одно значение в этом месте структуры.

Текущий размер дерева всегда присутствует, как и в случае реализации `std::list` в MSVC (29.4.2).

Как и в случае с `std::list`, итераторы это просто указатели на узлы. Итератор `.begin()` указывает на минимальный ключ. Этот указатель нигде не сохранен (как в списках), минимальный ключ дерева нужно находить каждый раз. `operator-` и `operator++` перемещают указатель не текущий узел на узел-предшественник или узел-преемник, т.е., узлы содержащие предыдущий и следующий ключ. Алгоритмы для всех этих операций описаны в [7].

Итератор `.end()` указывает на корневой узел, он имеет 1 в `Isnil`, что означает что у узла нет ключа и/или значения. Так что его можно рассматривать как "landing zone" в HDD[12].

GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>
```

[12]Hard disk drive

```
struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool dump_keys_and_values↙
    ↳ )
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p M_color=%d\n",
            n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
        };
    };

    if (traverse==false)
        return;

    if (n->M_left)
        dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
    if (n->M_right)
        dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
        struct map_pair *p=(struct map_pair *)point_after_struct;
```

```
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, M_node_count=%d\n",
        m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";

    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, true);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, false, false);

    // set
```

```
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, true);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, false, false);
};
```

Листинг 29.34: GCC 4.8.1

```
dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0x007A4C20 M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0x00000000 M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0x00000000 M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
```

```
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0x007A4CC0 M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0x01D5D850 M_color=1
key=123
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0x00000000 M_color=0
key=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0x01D5D8D0 M_color=0
```

Реализация в GCC очень похожа [13]. Разница только в том, что здесь нет поля `Isnil`, так что структура занимает немного меньше места в памяти чем та что реализована в MSVC. Корневой узел это также место, куда указывает итератор `.end()`, не имеющий ключа и/или значения.

Демонстрация перебалансировки (GCC)

Вот также демонстрация показывающая нам как дерево может перебалансироваться после вставок.

Листинг 29.35: GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>
```

[13] http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html

```
struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
```

```
    printf ("\n");
    printf ("100, 1001 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    printf ("\n");
};
```

Листинг 29.36: GCC 4.8.1

```
123, 456 are inserted
root----123
        R-------456

11, 12 are inserted
root----123
        L-------11
                R-------12
        R-------456

100, 1001 are inserted
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001

667, 1, 4, 7 are inserted
root----12
        L-------4
                L-------1
                R-------11
                        L-------7
        R-------123
                L-------100
                R-------667
                        L-------456
                        R-------1001
```

Глава 30

Обфускация

Обфускация это попытка спрятать код (или его значение) от reverse engineer-а.

30.1 Текстовые строки

Как я указывал в (36) текстовые строки могут быть крайне полезны. Знающие об этом программисты могут попытаться их спрятать так, чтобы их не было видно в IDA или любом шестнадцатеричном редакторе.

Вот простейший метод.

Вот как строка может быть сконструирована:

```
mov     byte ptr [ebx], 'h'
mov     byte ptr [ebx+1], 'e'
mov     byte ptr [ebx+2], 'l'
mov     byte ptr [ebx+3], 'l'
mov     byte ptr [ebx+4], 'o'
mov     byte ptr [ebx+5], ' '
mov     byte ptr [ebx+6], 'w'
mov     byte ptr [ebx+7], 'o'
mov     byte ptr [ebx+8], 'r'
mov     byte ptr [ebx+9], 'l'
mov     byte ptr [ebx+10], 'd'
```

Строка также может сравниваться с другой:

```
mov     ebx, offset username
cmp     byte ptr [ebx], 'j'
jnz     fail
cmp     byte ptr [ebx+1], 'o'
jnz     fail
cmp     byte ptr [ebx+2], 'h'
jnz     fail
cmp     byte ptr [ebx+3], 'n'
jnz     fail
jz      it_is_john
```

В обоих случаях, эти строки нельзя так просто нати в шестнадцатеричном редакторе.

Кстати, точно также со строками можно работать в тех случаях, когда строку нельзя разместить в сегменте данных, например, в PIC, или в шелл-коде.

Еще один виденный мною метод с использованием ф-ции `sprintf()` для конструирования:

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

Код выглядит ужасно, но как простейшая мера для анти-реверсинга, это может помочь.

Текстовые строки могут также присутствовать в зашифрованном виде, в таком случае, их использование будет предварять вызов ф-ции для дешифровки.

30.2 Исполняемый код

30.2.1 Вставка мусора

Обфускация исполняемого кода это вставка случайного мусора (между настоящим кодом), который исполняется, но не делает ничего полезного.

Просто пример:

```
add     eax, ebx
mul     ecx
```

Листинг 30.1: obfuscated code

```
xor     esi, 011223344h ; мусор
add     esi, eax        ; мусор
add     eax, ebx
mov     edx, eax        ; мусор
shl     edx, 4          ; мусор
mul     ecx
xor     esi, ecx        ; мусор
```

Здесь код-мусор использует регистры, которые не используются в настоящем коде (ESI и EDX). Впрочем, промежуточные результаты полученные при исполнении настоящего кода вполне могут использоваться кодом-мусором для бóльшей путанницы — почему нет?

30.2.2 Замена инструкций на раздутые эквиваленты

- MOV op1, op2 может быть заменена на пару PUSH op2 / POP op1.
- JMP label может быть заменена на пару PUSH label / RET. IDA не покажет ссылок на эту метку.
- CALL label может быть заменена на тройку PUSH label_after_CALL_instruction / PUSH label / RET.
- PUSH op также можно заменить на пару SUB ESP, 4 (или 8) / MOV [ESP], op.

30.2.3 Всегда исполняющийся/никогда не исполняющийся код

Если разработчик уверен что в ESI всегда будет 0 в этом месте:

```
mov     esi, 1
...     ; какой-то не трогающий ESI код
dec     esi
...     ; какой-то не трогающий ESI код
cmp     esi, 0
jz      real_code
; фальшивый багаж
real_code:
```

Reverse engineer-у понадобится какое-то время чтобы с этим разобраться.

Это также называется *opaque predicate*.

Еще один пример (и снова разработчик уверен что ESI — всегда ноль):

```
add     eax, ebx        ; реальный код
mul     ecx             ; реальный код
add     eax, esi        ; opaque predicate. вместо ADD тут может быть XOR
```

30.2.4 Сделать побольше путанницы

```
instruction 1
instruction 2
instruction 3
```

Можно заменить на:

```
begin:          jmp     ins1_label

ins2_label:     instruction 2
                jmp     ins3_label

ins3_label:     instruction 3
                jmp     exit:

ins1_label:     instruction 1
                jmp     ins2_label
exit:
```

30.2.5 Использование косвенных указателей

```
dummy_data1     db      100h dup (0)
message1        db      'hello world',0

dummy_data2     db      200h dup (0)
message2        db      'another message',0

func            proc
                ...
                mov     eax, offset dummy_data1 ; PE or ELF reloc here
                add     eax, 100h
                push    eax
                call    dump_string
                ...
                mov     eax, offset dummy_data2 ; PE or ELF reloc here
                add     eax, 200h
                push    eax
                call    dump_string
                ...
func            endp
```

IDA покажет ссылки на `dummy_data1` и `dummy_data2`, но не на сами текстовые строки.

К глобальным переменным и даже ф-циям можно обращаться так же.

30.3 Виртуальная машина / псевдо-код

Программист может также создать свой собственный ЯП или ISA и интерпретатор для него. (Как версии Visual Basic перед 5.0, .NET, Java machine). Reverse engineer-у придется потратить какое-то время для понимания деталей всех инструкций в ISA. Ему также возможно придется писать что-то вроде дизассемблера/декомпилятора.

30.4 Еще кое-что

Моя попытка (хотя и слабая) пропатчить компилятор Tiny C чтобы он выдавал обфусцированный код: http://blog.yurichev.com/node/58.

Использование инструкции `MOV` для сложных вещей: [8].

Глава 31

Windows 16-bit

16-битные программы под Windows в наше время редки, хотя я иногда вожусь с ними, в смысле ретрокомпьютинга, либо защищенные донглами (55).

16-битные версии Windows были вплоть до 3.11. 96/98/ME также поддерживает 16-битный код, как и все 32-битные OS линейки Windows NT. 64-битные версии Windows NT не поддерживают 16-битный код вообще.

Код напоминает тот что под MS-DOS.

Исполняемые файлы имеют не MZ-тип, и не PE-тип, а NE-тип (так называемый "new executable").

Все рассмотренные здесь примеры скомпилированы компилятором OpenWatcom 1.9 используя эти опции: `wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c`

31.1 Пример#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBeep(MB_ICONEXCLAMATION);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 30h ; '0'   ; MB_ICONEXCLAMATION constant
                push    ax
                call    MESSAGEBEEP
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

Пока всё просто.

31.2 Пример #2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
```

```
        MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; 0x18. "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; 0x10. "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp

dseg02:0010 aCaption        db 'caption',0
dseg02:0018 aHelloWorld     db 'hello, world',0
```

Пара важных моментов: соглашение о передаче аргументов здесь PASCAL: оно указывает что самый последний аргумент должен передаваться первым (MB_YESNOCANCEL), а самый первый аргмент — последним (NULL). Это соглашение также указывает вызываемой ф-ции восстановить указатель стека: поэтому инструкция RETN имеет аргумент 0Ah означая что указатель нужно сдвинуть вперед на 10 байт во время возврата из ф-ции.

Указатели передаются парами: сначала сегмент данных, потом указатель внутри сегмента. В этом примере только один сегмент, так что DS всегда указывает на сегмент данных в исполняемом файле.

31.3 Пример #3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax       ; NULL
                push    ax
                push    ds
```

```
                mov     ax, offset aHelloWorld ; "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                cmp     ax, 2           ; IDCANCEL
                jnz     short loc_2F
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedCanc ; "you pressed cancel"
                jmp     short loc_49
loc_2F:
                cmp     ax, 6           ; IDYES
                jnz     short loc_3D
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedYes ; "you pressed yes"
                jmp     short loc_49
loc_3D:
                cmp     ax, 7           ; IDNO
                jnz     short loc_57
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedNo ; "you pressed no"
loc_49:
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax
                push    ax
                call    MESSAGEBOX
loc_57:
                xor     ax, ax
                pop     bp
                retn    0Ah
WinMain         endp
```

Немного расширенная версия примера из предыдущей секции.

31.4 Пример #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
        return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
        return a*b+c;
};
```

```
long PASCAL func3 (long a, long b, long c, int d)
{
        return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        func1 (123, 456, 789);
        func2 (600000, 700000, 800000);
        func3 (600000, 700000, 800000, 123);
        return 0;
};
```

```
func1           proc near

c               = word ptr  4
b               = word ptr  6
a               = word ptr  8

                push    bp
                mov     bp, sp
                mov     ax, [bp+a]
                imul    [bp+b]
                add     ax, [bp+c]
                pop     bp
                retn    6
func1           endp

func2           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_8]
                mov     dx, [bp+arg_A]
                mov     bx, [bp+arg_4]
                mov     cx, [bp+arg_6]
                call    sub_B2 ; long 32-bit multiplication
                add     ax, [bp+arg_0]
                adc     dx, [bp+arg_2]
                pop     bp
                retn    12
func2           endp

func3           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh
```

```
arg_C           = word ptr  10h

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_A]
                mov     dx, [bp+arg_C]
                mov     bx, [bp+arg_6]
                mov     cx, [bp+arg_8]
                call    sub_B2 ; long 32-bit multiplication
                mov     cx, [bp+arg_2]
                add     cx, ax
                mov     bx, [bp+arg_4]
                adc     bx, dx          ; BX=high part, CX=low part
                mov     ax, [bp+arg_0]
                cwd                     ; AX=low part d, DX=high part d
                sub     cx, ax
                mov     ax, cx
                sbb     bx, dx
                mov     dx, bx
                pop     bp
                retn    14
func3           endp

WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 123
                push    ax
                mov     ax, 456
                push    ax
                mov     ax, 789
                push    ax
                call    func1
                mov     ax, 9       ; high part of 600000
                push    ax
                mov     ax, 27C0h   ; low part of 600000
                push    ax
                mov     ax, 0Ah     ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch     ; high part of 800000
                push    ax
                mov     ax, 3500h   ; low part of 800000
                push    ax
                call    func2
                mov     ax, 9       ; high part of 600000
                push    ax
                mov     ax, 27C0h   ; low part of 600000
                push    ax
                mov     ax, 0Ah     ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch     ; high part of 800000
                push    ax
                mov     ax, 3500h   ; low part of 800000
                push    ax
                mov     ax, 7Bh     ; 123
                push    ax
                call    func3
```

```
                xor     ax, ax      ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

32-битные значения (тип данных `long` означает 32-бита, а *int* здесь 16-битный) в 16-битном коде (и в MS-DOS и в Win16) передаются парами). Это так же как и 64-битные значения передаются в 32-битной среде (21).

`sub_B2 here` здесь это библиотечная ф-ция написанная разработчиками компилятора, делающая "long multiplication", т.е., перемножает два 32-битных значения. Другие ф-ции компиляторов делающие то же самое перечислены здесь: E, D.

Пара инструкций `ADD`/`ADC` используется для сложения этих составных значений: `ADD` может установить или сбросить флаг `CF`, `ADC` будет использовать его. Пара инструкций `SUB`/`SBB` используется для вычитания: `SUB` может установить или сбросить флаг `CF`, `SBB` будет использовать его.

32-битные значения возвращаются из ф-ций в паре регистров `DX:AX`.

Константы так же передаются как пары в `WinMain()`.

Константа 123 типа *int* в начале конвертируется (учитывая знак) в 32-битное значение используя инструкция `CWD`.

31.5 Пример #5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

void PASCAL remove_digits (char *s)
{
        while (*s)
        {
                if (*s>='0' && *s<='9')
                        *s='-';
                s++;
        };
};
```

```
char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        string_compare ("asd", "def");
        string_compare_far ("asd", "def");
        remove_digits (str);
        MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
        return 0;
};
```

```
string_compare  proc near

arg_0 = word ptr  4
arg_2 = word ptr  6

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
      mov     al, [bx]
      cmp     al, [si]
      jz      short loc_1C
      xor     ax, ax
      jmp     short loc_2B


loc_1C: ; CODE XREF: string_compare+Ej
      test    al, al
      jz      short loc_22
      jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
      mov     ax, 1
      jmp     short loc_2B


loc_27: ; CODE XREF: string_compare+18j
      inc     bx
      inc     si
      jmp     short loc_12


loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
      pop     si
      pop     bp
      retn    4
string_compare  endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr  4
arg_2 = word ptr  6
arg_4 = word ptr  8
```

```
arg_6 = word ptr  0Ah

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
      mov     es, [bp+arg_6]
      mov     al, es:[bx]
      mov     es, [bp+arg_2]
      cmp     al, es:[si]
      jz      short loc_4C
      xor     ax, ax
      jmp     short loc_67


loc_4C: ; CODE XREF: string_compare_far+16j
      mov     es, [bp+arg_6]
      cmp     byte ptr es:[bx], 0
      jz      short loc_5E
      mov     es, [bp+arg_2]
      cmp     byte ptr es:[si], 0
      jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
      mov     ax, 1
      jmp     short loc_67


loc_63: ; CODE XREF: string_compare_far+2Cj
      inc     bx
      inc     si
      jmp     short loc_3A


loc_67: ; CODE XREF: string_compare_far+1Aj
        ; string_compare_far+31j
      pop     si
      pop     bp
      retn    8
string_compare_far endp

remove_digits   proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr  4

      push    bp
      mov     bp, sp
      mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
      mov     al, [bx]
      test    al, al
      jz      short loc_86
      cmp     al, 30h ; '0'
      jb      short loc_83
      cmp     al, 39h ; '9'
      ja      short loc_83
      mov     byte ptr [bx], 2Dh ; '-'
```

```
loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
      inc     bx
      jmp     short loc_72


loc_86: ; CODE XREF: remove_digits+Aj
      pop     bp
      retn    2
remove_digits   endp

WinMain proc near ; CODE XREF: start+EDp
      push    bp
      mov     bp, sp
      mov     ax, offset aAsd ; "asd"
      push    ax
      mov     ax, offset aDef ; "def"
      push    ax
      call    string_compare
      push    ds
      mov     ax, offset aAsd ; "asd"
      push    ax
      push    ds
      mov     ax, offset aDef ; "def"
      push    ax
      call    string_compare_far
      mov     ax, offset aHello1234World ; "hello 1234 world"
      push    ax
      call    remove_digits
      xor     ax, ax
      push    ax
      push    ds
      mov     ax, offset aHello1234World ; "hello 1234 world"
      push    ax
      push    ds
      mov     ax, offset aCaption ; "caption"
      push    ax
      mov     ax, 3 ; MB_YESNOCANCEL
      push    ax
      call    MESSAGEBOX
      xor     ax, ax
      pop     bp
      retn    0Ah
WinMain endp
```

Здесь мы можем увидеть разницу между указателями “near” и указателями “far” еще один ужасный артефакт сегментированной памяти 16-битного 8086.

Читайте больше об этом: 67.

Указатели “near” (“близкие”) это те которые указывают в пределах текущего сегмента. Поэтому, ф-ция `string_compare()` берет на вход только 2 16-битных значения и работает с данными расположеными в сегменте, на который указывает `DS` (инструкция `mov al, [bx]` на самом деле работает как `mov al, ds:[bx]` — `DS` используется здесь неявно).

Указатели “far” (далекие) могут указывать на данные в другом сегменте памяти. Поэтому `string_compare_far()` берет на вход 16-битную пару как указатель, загружает старшую часть в сегментный регистр `ES` и обращается к данным через него (`mov al, es:[bx]`). Указатели “far” также используются в моем win16-примере касательно `MessageBox()`: 31.2. Действительно, ядро Windows должно знать, из какого сегмента данных читать текстовые строки, так что ему нужна полная информация.

Причина этой разница в том что компактная программа вполне может обойтись одним сегментом данных размером 64 килобайта, так что старшую часть указателя передавать не нужна (ведь она одинаковая везде). Бо́льшие программы могут использовать несколько сегментов данных размером 64 килобайта, так что нужно

указывать каждый раз, в каком сегменте расположены данные.

То же касается и сегментов кода. Компактная программа может расположиться в пределах одного 64kb-сегмента, тогда ф-ции в ней будут вызываться инструкцией `CALL NEAR`, а возвращаться управление используя `RETN`. Но если сегментов кода несколько, тогда и адрес вызываемой ф-ции будет задаваться парой, вызываться она будет используя `CALL FAR`, а возвращаться управление используя `RETF`.

Это то что задается в компиляторе указывая "memory model".

Компиляторы под MS-DOS и Win16 имели разные библиотеки под разные модели памяти: они отличались типами указателей для кода и данных.

31.6 Пример #6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        struct tm *t;
        time_t unix_time;

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->↙
    ↳ tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};
```

```
WinMain         proc near

var_4           = word ptr -4
var_2           = word ptr -2

                push    bp
                mov     bp, sp
                push    ax
                push    ax
                xor     ax, ax
                call    time_
                mov     [bp+var_4], ax   ; low part of UNIX time
                mov     [bp+var_2], dx   ; high part of UNIX time
                lea     ax, [bp+var_4]   ; take a pointer of high part
                call    localtime_
                mov     bx, ax           ; t
                push    word ptr [bx]    ; second
                push    word ptr [bx+2]  ; minute
                push    word ptr [bx+4]  ; hour
                push    word ptr [bx+6]  ; day
                push    word ptr [bx+8]  ; month
                mov     ax, [bx+0Ah]     ; year
```

```
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax            ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax
                mov     sp, bp
                pop     bp
                retn    0Ah
WinMain         endp
```

Время в формате UNIX это 32-битное значение, так что оно возвращается в паре регистров DX:AX и сохраняется в двух локальны 16-битных переменных. Потом указатель на эту пару передается в ф-цию localtime(). Ф-ция localtime() имеет структуру struct tm расположенную у себя где-то внутри, так что только указатель на нее возвращается. Кстати, это также означает что функцию нельзя вызывать еще раз, пока её результаты не были использованы.

Для ф-ций time() и localtime() используется Watcom-соглашение о вызовах: первые четыре аргумента передаются через регистры AX, DX, BX и CX, а остальные аргументы через стек. Ф-ции, использующие это соглашение, маркируется символом подчеркивания в конце имени.

Для вызова ф-ции sprintf() используется обычное соглашение *cdecl* (44.1) вместо PASCAL или Watcom, так что аргументы передаются привычным образом.

31.6.1 Глобальные переменные

Это тот же пример, только переменные теперь глобальные:

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->↙
    ↪ tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);
```

```
        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};
```

```
unix_time_low   dw 0
unix_time_high  dw 0
t               dw 0

WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax
                call    time_
                mov     unix_time_low, ax
                mov     unix_time_high, dx
                mov     ax, offset unix_time_low
                call    localtime_
                mov     bx, ax
                mov     t, ax               ; will not be used in future...
                push    word ptr [bx]       ; seconds
                push    word ptr [bx+2]     ; minutes
                push    word ptr [bx+4]     ; hour
                push    word ptr [bx+6]     ; day
                push    word ptr [bx+8]     ; month
                mov     ax, [bx+0Ah]        ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

t не будет использоваться, но компилятор создал код, записывающий в эту переменную. Потому что он не уверен, может быть это значение где-то еще будет прочитано.

Часть II

Важные фундаментальные вещи

Глава 32

Представление знака в числах

Методов представления чисел с знаком "плюс" или "минус" несколько[1], а в x86 применяется метод "дополнительный код" или "two's complement".

двоичное	шестнадцатеричное	беззнаковое	знаковое (дополнительный код)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

Разница в подходе к знаковым/беззнаковым числам, собственно, нужна потому что, например, если представить `0xFFFFFFFE` и `0x0000002` как беззнаковое, то первое число (4294967294) больше второго (2). Если их оба представить как знаковые, то первое будет −2, которое, разумеется, меньше чем второе (2). Вот почему инструкции для условных переходов (10) представлены в обоих версиях — и для знаковых сравнений (например `JG`, `JL`) и для беззнаковых (`JA`, `JBE`).

Для простоты, вот что нужно знать:

- Числа бывают знаковые и беззнаковые.
- Знаковые типы в Си/Си++: *int* (-2147483646..2147483647 или `0x80000000..0x7FFFFFFF`), *char* (-127..128 или `0x7F..0x80`). Беззнаковые: `unsigned int` (0..4294967295 или `0..0xFFFFFFFF`), `unsigned char` (0..255 или `0..0xFF`), `size_t`.
- У знаковых чисел знак определяется самым старшим битом: 1 означает "минус", 0 означает "плюс".
- Инструкции сложения и вычитания работают одинаково хорошо и для знаковых и для беззнаковых значений. Но для операций умножения и деления, в x86 имеются разные инструкции: `IDIV`/`IMUL` для знаковых и `DIV`/`MUL` для беззнаковых.
- Еще инструкции работающие с знаковыми числами: `CBW/CWD/CWDE/CDQ/CDQE` (B.6.3), `MOVSX` (13.1.1), `SAR` (B.6.3).

32.1 Переполнение integer

Бывает так, что ошибки представления знаковых/беззнаковых могут привести к уязвимости *переполнение integer*.

Например, есть некий сервис, который принимает по сети некие пакеты. В пакете есть заголовок где указана длина пакета. Это 32-битное значение. В процессе приема пакета, сервис проверяет это значение и сверяет,

[1] http://en.wikipedia.org/wiki/Signed_number_representations

больше ли оно чем максимальный размер пакета, скажем, константа `MAX_PACKET_SIZE` (например, 10 килобайт), и если да, то пакет отвергается как некорректный. Сравнение знаковое. Злоумышленник подставляет значение `0xFFFFFFFF`. Это число трактуется как знаковое −1 и оно меньше чем 10000. Проверка проходит. Продолжаем дальше и копируем этот пакет куда-нибудь себе в сегмент данных... вызов функции `memcpy (dst, src, 0xFFFFFFFF)` скорее всего, затрет много чего внутри процесса.

Немного подробнее: [3].

Глава 33

Endianness (порядок байт)

Endianness (порядок байт) это способ представления чисел в памяти.

33.1 Big-endian (от старшего к младшему)

Число `0x12345678` будет представлено в памяти так:

адрес в памяти	значение байта
+0	0x12
+1	0x34
+2	0x56
+3	0x78

CPU с таким порядком включают в себя Motorola 68k, IBM POWER.

33.2 Little-endian (от младшего к старшему)

Число `0x12345678` будет представлено в памяти так:

адрес в памяти	значение байта
+0	0x78
+1	0x56
+2	0x34
+3	0x12

CPU с таким порядком байт включают в себя Intel x86.

33.3 Bi-endian (переключаемый порядок)

CPU поддерживающие оба порядка, и его можно переключать, включают в себя ARM, PowerPC, SPARC, MIPS, IA64[1], итд.

33.4 Конвертирование

Сетевые пакеты TCP/IP используют соглашение big-endian, вот почему программа работающая на little-endian архитектуре должна конвертировать значения используя ф-ции `htonl()` и `htons()`.

Порядок байт big-endian в среде TCP/IP также называется “network byte order”, а little-endian — “host byte order”.

Инструкция `BSWAP` также может использоваться для конвертирования.

[1]Intel Architecture 64 (Itanium): 66

Часть III

Поиск в коде того что нужно

Современное ПО, в общем-то, минимализмом не отличается.

Но не потому, что программисты слишком много пишут, а потому что к исполняемым файлам обыкновенно прикомпилируют все подряд библиотеки. Если бы все вспомогательные библиотеки всегда выносили во внешние DLL, мир был бы иным. (Еще одна причина для Си++ — STL и прочие библиотеки шаблонов.)

Таким образом, очень полезно сразу понимать, какая функция из стандартной библиотеки или более-менее известной (как Boost[2], libpng[3]), а какая — имеет отношение к тому что мы пытаемся найти в коде.

Переписывать весь код на Си/Си++, чтобы разобраться в нем, безусловно, не имеет никакого смысла.

Одна из важных задач reverse engineer-а это быстрый поиск в коде того что собственно его интересует.

Дизассемблер IDA позволяет делать поиск как минимум строк, последовательностей байт, констант. Можно даже сделать экспорт кода в текстовый файл .lst или .asm и затем натравить на него `grep`, `awk`, и т.д.

Когда вы пытаетесь понять, что делает тот или иной код, это запросто может быть какая-то опенсорсная библиотека вроде libpng. Поэтому, когда находите константы, или текстовые строки которые выглядят явно знакомыми, всегда полезно их *погуглить*. А если вы найдете искомый опенсорсный проект где это используется, то тогда будет достаточно будет просто сравнить вашу функцию с ней. Это решит часть проблем.

К примеру, если программа использует какие-то XML-файлы, первым шагом может быть установление, какая именно XML-библиотека для этого используется, ведь часто используется какая-то стандартная (или очень известная) вместо самодельной.

К примеру, однажды я пытался разобраться как происходит компрессия/декомпрессия сетевых пакетов в SAP 6.0. Это очень большая программа, но к ней идет подробный .PDB-файл с отладочной информацией, и это очень удобно. Я в конце концов пришел к тому что одна из функций декомпрессирующая пакеты называется CsDecomprLZC(). Не сильно раздумывая, я решил погуглить и оказалось что функция с таким же названием имеется в MaxDB (это опен-сорсный проект SAP)[4].

http://www.google.com/search?q=CsDecomprLZC

Каково же было мое удивление, когда оказалось, что в MaxDB используется точно такой же алгоритм, скорее всего, с таким же исходником.

[2] http://www.boost.org/

[3] http://www.libpng.org/pub/png/libpng.html

[4] Больше об этом в соответствующей секции (57.1)

Глава 34

Идентификация исполняемых файлов

34.1 Microsoft Visual C++

Версии MSVC и DLL которые могут быть импортированы:

Маркетинговая версия	Внутренняя версия	Версия CL.EXE	Импортируемые DLL	Дата выхода
6	6.0	12.00	msvcrt.dll, msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll, msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll, msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll, msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll, msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll, msvcp120.dll	October 17, 2013

msvcp*.dll содержит ф-ции связанные с Си++, так что если она импортируется, скорее всего, вы имеете дело с программой на Си++.

34.1.1 Name mangling

Имена обычно начинаются с символа `?`.

О name mangling в MSVC читайте также здесь: 29.1.1.

34.2 GCC

Кроме компиляторов под *NIX, GCC имеется так же и для win32-окружения: в виде Cygwin и MinGW.

34.2.1 Name mangling

Имена обычно начинаются с символов `_Z`.

О name mangling в GCC читайте также здесь: 29.1.1.

34.2.2 Cygwin

cygwin1.dll часто импортируется.

34.2.3 MinGW

msvcrt.dll может импортироваться.

34.3 Intel FORTRAN

libifcoremd.dll, libifportmd.dll и libiomp5md.dll (поддрежка OpenMP) могут импортироваться.

В libifcoremd.dll много ф-ций с префиксом `for_`, что значит FORTRAN.

34.4 Watcom, OpenWatcom

34.4.1 Name mangling

Имена обычно начинаются с символа W.

Например, так кодируется метод "method" класса "class" не имеющий аргументов и возвращающий *void*:

```
W?method$_class$n__v
```

34.5 Borland

Вот пример name mangling в Borland Delphi и C++Builder:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

Имена всегда начинаются с символа @ затем следует имя класса, имя метода и закодированные типы аргументов.

Эти имена могут присутствовать с импортах .exe, экспортах .dll, отладочной информации, и т.д.

Borland Visual Component Libraries (VCL) находятся в файлах .bpl вместо .dll, например, vcl50.dll, rtl60.dll.

Другие DLL которые могут импортироваться: BORLNDMM.DLL.

34.5.1 Delphi

Почти все исполняемые файлы имеют текстовую строку "Boolean" в самом начале сегмента кода, среди остальных имен типов.

Вот очень характерное для Delphi начало сегмента `.text`, этот блок следует сразу за заголовком win32 PE-файла:

```
00000400  04 10 40 00 03 07 42 6f  6f 6c 65 61 6e 01 00 00  |..@...Boolean...|
00000410  00 00 01 00 00 00 00 10  40 00 05 46 61 6c 73 65  |........@..False|
00000420  04 54 72 75 65 8d 40 00  2c 10 40 00 09 08 57 69  |.True.@.,.@...Wi|
00000430  64 65 43 68 61 72 03 00  00 00 00 ff ff 00 00 90  |deChar..........|
00000440  44 10 40 00 02 04 43 68  61 72 01 00 00 00 00 ff  |D.@...Char......|
00000450  00 00 00 90 58 10 40 00  01 08 53 6d 61 6c 6c 69  |....X.@...Smalli|
00000460  6e 74 02 00 80 ff ff ff  7f 00 00 90 70 10 40 00  |nt..........p.@.|
00000470  01 07 49 6e 74 65 67 65  72 04 00 00 00 80 ff ff  |..Integer.......|
00000480  ff 7f 8b c0 88 10 40 00  01 04 42 79 74 65 01 00  |......@...Byte..|
00000490  00 00 00 ff 00 00 00 90  9c 10 40 00 01 04 57 6f  |..........@...Wo|
000004a0  72 64 03 00 00 00 00 ff  ff 00 00 90 b0 10 40 00  |rd............@.|
000004b0  01 08 43 61 72 64 69 6e  61 6c 05 00 00 00 00 ff  |..Cardinal......|
000004c0  ff ff ff 90 c8 10 40 00  10 05 49 6e 74 36 34 00  |......@...Int64.|
000004d0  00 00 00 00 00 00 80 ff  ff ff ff ff ff ff 7f 90  |................|
000004e0  e4 10 40 00 04 08 45 78  74 65 6e 64 65 64 02 90  |..@...Extended..|
000004f0  f4 10 40 00 04 06 44 6f  75 62 6c 65 01 8d 40 00  |..@...Double..@.|
00000500  04 11 40 00 04 08 43 75  72 72 65 6e 63 79 04 90  |..@...Currency..|
00000510  14 11 40 00 0a 06 73 74  72 69 6e 67 20 11 40 00  |..@...string .@.|
00000520  0b 0a 57 69 64 65 53 74  72 69 6e 67 30 11 40 00  |..WideString0.@.|
00000530  0c 07 56 61 72 69 61 6e  74 8d 40 00 40 11 40 00  |..Variant.@.@.@.|
00000540  0c 0a 4f 6c 65 56 61 72  69 61 6e 74 98 11 40 00  |..OleVariant..@.|
00000550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000560  00 00 00 00 00 00 00 00  00 00 00 00 98 11 40 00  |..............@.|
00000570  04 00 00 00 00 00 00 00  18 4d 40 00 24 4d 40 00  |.........M@.$M@.|
```

```
00000580  28 4d 40 00 2c 4d 40 00  20 4d 40 00 68 4a 40 00  |(M@.,M@. M@.hJ@.|
00000590  84 4a 40 00 c0 4a 40 00  07 54 4f 62 6a 65 63 74  |.J@..J@..TObject|
000005a0  a4 11 40 00 07 07 54 4f  62 6a 65 63 74 98 11 40  |..@...TObject..@|
000005b0  00 00 00 00 00 00 00 06  53 79 73 74 65 6d 00 00  |........System..|
000005c0  c4 11 40 00 0f 0a 49 49  6e 74 65 72 66 61 63 65  |..@...IInterface|
000005d0  00 00 00 00 01 00 00 00  00 00 00 00 00 c0 00 00  |................|
000005e0  00 00 00 00 46 06 53 79  73 74 65 6d 03 00 ff ff  |....F.System....|
000005f0  f4 11 40 00 0f 09 49 44  69 73 70 61 74 63 68 c0  |..@...IDispatch.|
00000600  11 40 00 01 00 04 02 00  00 00 00 00 c0 00 00 00  |.@..............|
00000610  00 00 00 46 06 53 79 73  74 65 6d 04 00 ff ff 90  |...F.System.....|
00000620  cc 83 44 24 04 f8 e9 51  6c 00 00 83 44 24 04 f8  |..D$...Ql...D$..|
00000630  e9 6f 6c 00 00 83 44 24  04 f8 e9 79 6c 00 00 cc  |.ol...D$...yl...|
00000640  cc 21 12 40 00 2b 12 40  00 35 12 40 00 01 00 00  |.!.@.+.@.5.@....|
00000650  00 00 00 00 00 00 00 00  00 c0 00 00 00 00 00 00  |................|
00000660  46 41 12 40 00 08 00 00  00 00 00 00 00 8d 40 00  |FA.@..........@.|
00000670  bc 12 40 00 4d 12 40 00  00 00 00 00 00 00 00 00  |..@.M.@.........|
00000680  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000690  bc 12 40 00 0c 00 00 00  4c 11 40 00 18 4d 40 00  |..@.....L.@..M@.|
000006a0  50 7e 40 00 5c 7e 40 00  2c 4d 40 00 20 4d 40 00  |P~@.\~@.,M@. M@.|
000006b0  6c 7e 40 00 84 4a 40 00  c0 4a 40 00 11 54 49 6e  |l~@..J@..J@..TIn|
000006c0  74 65 72 66 61 63 65 64  4f 62 6a 65 63 74 8b c0  |terfacedObject..|
000006d0  d4 12 40 00 07 11 54 49  6e 74 65 72 66 61 63 65  |..@...TInterface|
000006e0  64 4f 62 6a 65 63 74 bc  12 40 00 a0 11 40 00 00  |dObject..@...@..|
000006f0  00 06 53 79 73 74 65 6d  00 00 8b c0 00 13 40 00  |..System......@.|
00000700  11 0b 54 42 6f 75 6e 64  41 72 72 61 79 04 00 00  |..TBoundArray...|
00000710  00 00 00 00 03 00 00 00  6c 10 40 00 06 53 79 00  |........l.@..Sy.|
00000720  73 74 65 6d 28 13 40 00  04 09 54 44 61 74 65 54  |stem(.@...TDateT|
00000730  69 6d 65 01 ff 25 48 e0  c4 00 8b c0 ff 25 44 e0  |ime..%H......%D.|
```

34.6 Другие известные DLL

- vcomp*.dll — Реализация OpenMP от Microsoft.

Глава 35

Связь с внешним миром (win32)

Обращения к файлам и реестру: для самого простого анализа может помочь утилита Process Monitor[1] от SysInternals.

Для анализа обращения программы к сети, может помочь Wireshark[2].

Затем всё-таки придётся смотреть внутрь.

Первое на что нужно обратить внимание, это какие функции из API[3] ОС и какие функции стандартных библиотек используются.

Если программа поделена на главный исполняемый файл и группу DLL-файлов, то имена функций в этих DLL, бывает так, могут помочь.

Если нас интересует, что именно приводит к вызову `MessageBox()` с определенным текстом, то первое что можно попробовать сделать: найти в сегменте данных этот текст, найти ссылки на него, и найти, откуда может передаться управление к интересующему нас вызову `MessageBox()`.

Если речь идет о компьютерной игре, и нам интересно какие события в ней более-менее случайны, мы можем найти функцию `rand()` или её заменитель (как алгоритм Mersenne twister), и посмотреть, из каких мест эта функция вызывается и что самое главное: как используется результат этой функции.

Но если это не игра, а `rand()` используется, то также весьма любопытно, зачем. Бывают неожиданные случаи вроде использования `rand()` в алгоритме для сжатия данных (для имитации шифрования): http://blog.yurichev.com/node/44.

35.1 Часто используемые ф-ции Windows API

Это ф-ции которые можно увидеть в числе импортируемых. Но также нельзя забывать, что далеко не все они были использованы в коде написанном автором. Немалая часть может вызываться из библиотечных ф-ций и CRT-кода.

- Работа с реестром (advapi32.dll): RegEnumKeyEx[4] [5], RegEnumValue[6] [5], RegGetValue[7] [5], RegOpenKeyEx[8] [5], RegQueryValueEx[9] [5].
- Работа с текстовыми .ini-файлами (kernel32.dll): GetPrivateProfileString [10] [5].
- Диалоговые окна (user32.dll): MessageBox [11] [5], MessageBoxEx [12] [5], SetDlgItemText [13] [5], GetDlgItemText [14] [5].
- Работа с ресурсами(48.2.8): (user32.dll): LoadMenu [15] [5].

[1] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
[2] http://www.wireshark.org/
[3] Application programming interface
[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724862(v=vs.85).aspx
[5] Может иметь суффикс -A для ASCII-версии и -W для Unicode-версии
[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724865(v=vs.85).aspx
[7] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724868(v=vs.85).aspx
[8] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724897(v=vs.85).aspx
[9] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724911(v=vs.85).aspx
[10] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724353(v=vs.85).aspx
[11] http://msdn.microsoft.com/en-us/library/ms645505(VS.85).aspx
[12] http://msdn.microsoft.com/en-us/library/ms645507(v=vs.85).aspx
[13] http://msdn.microsoft.com/en-us/library/ms645521(v=vs.85).aspx
[14] http://msdn.microsoft.com/en-us/library/ms645489(v=vs.85).aspx
[15] http://msdn.microsoft.com/en-us/library/ms647990(v=vs.85).aspx

- Работа с TCP/IP-сетью (ws2_32.dll): WSARecv [16], WSASend [17].
- Работа с файлами (kernel32.dll): CreateFile [18] [5], ReadFile [19], ReadFileEx [20], WriteFile [21], WriteFileEx [22].
- Высокоуровневая работа с Internet (wininet.dll): WinHttpOpen [23].
- Проверка цифровой подписи исполняемого файла (wintrust.dll): WinVerifyTrust [24].
- Стандартная библиотека MSVC (при случае динамического связывания) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

35.2 tracer: Перехват всех ф-ций в отдельном модуле

В tracer есть INT3-брякпойнты, хотя и срабатывающие только один раз, но зато их можно установить на все сразу ф-ции в некоей DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Либо, поставим INT3-прерывание на все функции, имена которых начинаются с префикса `xml`:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

В качестве обратной стороны медали, такие прерывания срабатывают только один раз.

Tracer покажет вызов какой-либо функции, если он случится, но только один раз. Еще один недостаток — увидеть аргументы функции также нельзя.

Тем не менее, эта возможность очень удобна для тех ситуаций, когда вы знаете что некая программа использует некую DLL, но не знаете какие именно функции в этой DLL. И функций много.

Например, попробуем узнать, что использует cygwin-утилита uptime:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Так мы можем увидеть все ф-ции из библиотеки cygwin1.dll, которые были вызваны хотя бы один раз, и откуда:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3))
```

[16] http://msdn.microsoft.com/en-us/library/windows/desktop/ms741688(v=vs.85).aspx
[17] http://msdn.microsoft.com/en-us/library/windows/desktop/ms742203(v=vs.85).aspx
[18] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx
[19] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx
[20] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365468(v=vs.85).aspx
[21] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx
[22] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365748(v=vs.85).aspx
[23] http://msdn.microsoft.com/en-us/library/windows/desktop/aa384098(v=vs.85).aspx
[24] http://msdn.microsoft.com/library/windows/desktop/aa388208.aspx

Глава 36

Строки

36.1 Текстовые строки

Обычные строки в Си заканчиваются нулем (ASCIIZ-строки).

Причина, почему формат строки в Си именно такой (оканчивающийся нулем) вероятно историческая. В [27] мы можем прочитать:

> A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

Строки выглядят в Hiew или FAR Manager точно так же:

```
int main()
{
        printf ("Hello, world!\n");
};
```

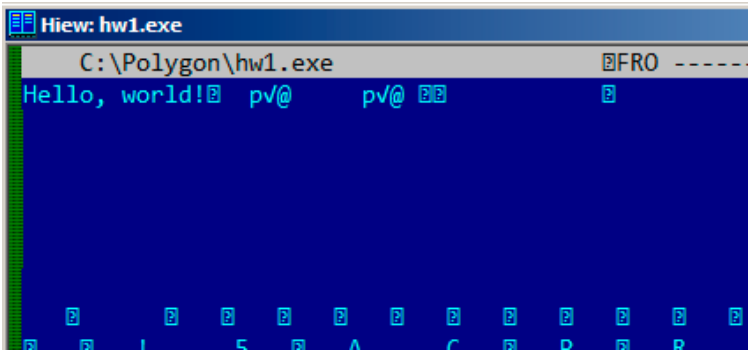


Рис. 36.1: Hiew

Когда кодируются строки в Pascal и Delphi, сама строка предваряется 8-битным или 32-битным значением, в котором закодирована длина строки.

Например:

Листинг 36.1: Delphi

```
CODE:00518AC8                 dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0

...

CODE:00518AFC                 dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

36.1.1 Unicode

Нередко уникодом называют все способы передачи символов, когда символ занимает 2 байта или 16 бит. Это распространенная терминологическая ошибка. Уникод — это стандарт, присваивающий номер каждому символу многих письменностей мира, но не описывающий способ кодирования.

Наиболее популярные способы кодирования: UTF-8 (наиболее часто используется в Интернете и *NIX-системах) и UTF-16LE (используется в Windows).

UTF-8

UTF-8 это один из очень удачных способов кодирования символов. Все символы латинницы кодируются так же, как и в ASCII-кодировке, а символы выходящие за пределы ASCII-7-таблицы, кодируются несколькими байтами. 0 кодируется, как и прежде, нулевыми байтом, так что все стандартные ф-ции Си продолжают работать с UTF-8-строками так же как и с обычными строками.

Посмотрим как символы из разных языков кодируются в UTF-8 и как это выглядит в FAR, в кодировке 437 [1]:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

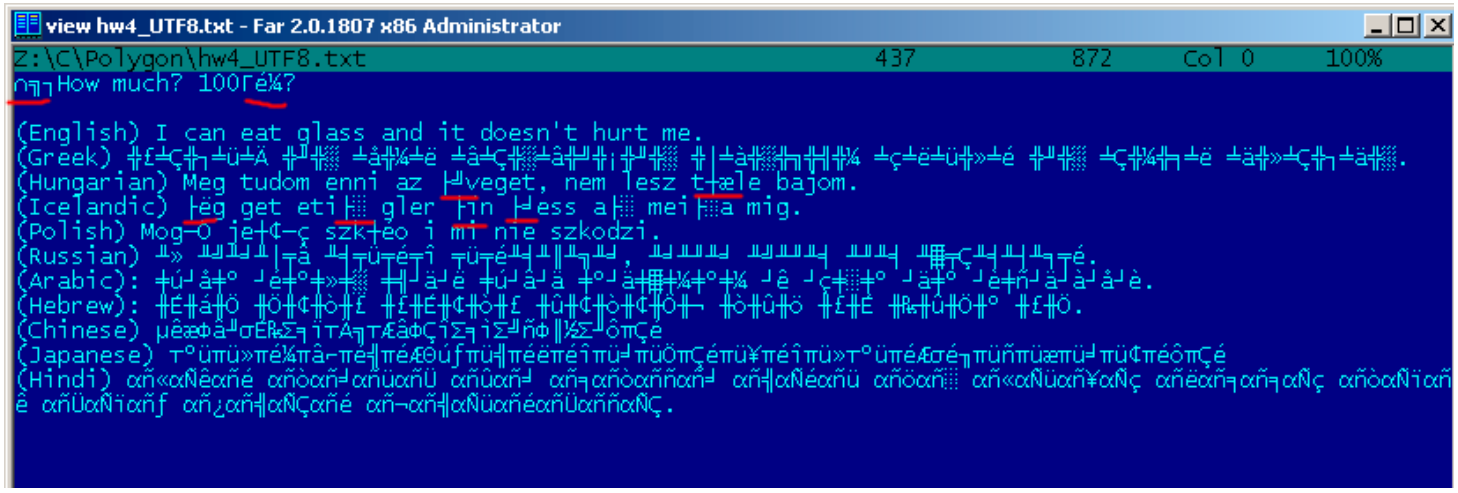


Рис. 36.2: FAR: UTF-8

Видно что строка на английском языке выглядит точно так же, как и в ASCII-кодировке. В венгерском языке используются латинница плюс латинские буквы с диакритическими знаками. Здесь видно что эти буквы кодируются несколькими байтами, я подчеркнул их красным. То же самое с исландским и польским языками. В самом начале я также применил символ валюты “Евро”, который кодируется тремя байтами. Остальные системы письма здесь никак не связаны с латинницей. По крайней мере о русском, арабском, иврите и хинди мы можем сказать что здесь видны повторяющиеся байты, что не удивительно, ведь, обычно буквы из одной и той же системы письменности расположены в одной или нескольких таблицах уникода, поэтому часто их коды начинаются с одних и тех же цифр.

В самом начале, перед строкой “How much?”, видны три байта, которые на самом деле BOM[2]. BOM показывает, какой способ кодирования будет сейчас использоваться.

[1] Я взял пример и переводы на разные языки здесь: http://www.columbia.edu/~fdc/utf8/

[2] Byte order mark

UTF-16LE

Многие ф-ции win32 в Windows имееют суффикс `-A` и `-W`. Первые ф-ции работают с обычными строками, вторые с UTF-16LE-строками (*wide*). Во втором случае, каждый символ обычно хранится в 16-битной переменной типа *short*.

Строки с латинскими буквами выглядят в Hiew или FAR как перемежающиеся с нулевыми байтами:

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

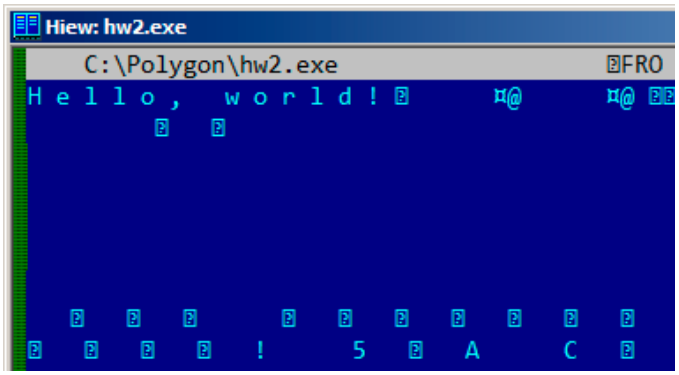


Рис. 36.3: Hiew

Подобное можно часто увидеть в системных файлах Windows NT:

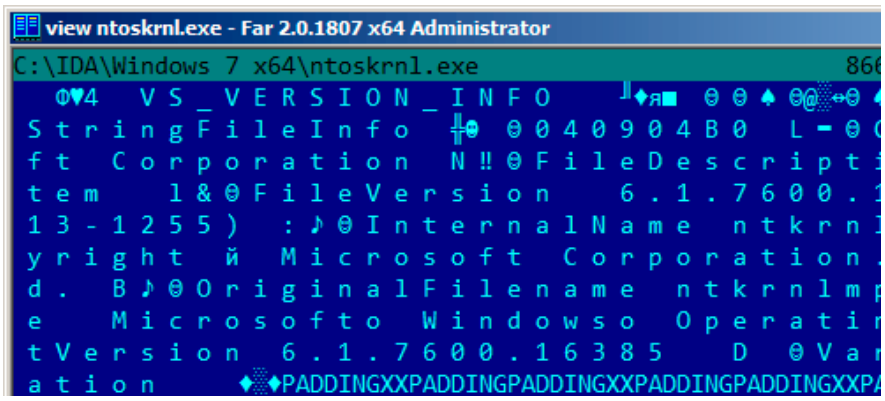


Рис. 36.4: Hiew

В IDA, уникодом называется именно строки с символами занимающими 2 байта:

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Вот как может выглядеть строка на русском языке (“И снова здравствуйте!”) закодированная в UTF-16LE:

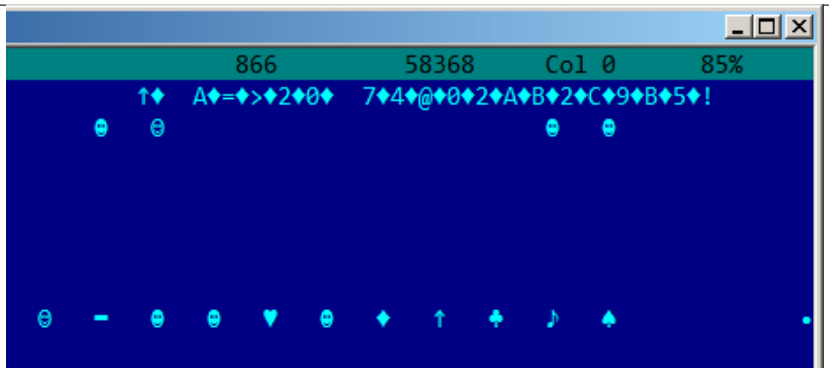


Рис. 36.5: Hiew: UTF-16LE

То что бросается в глаза — это то что символы перемежаются ромбиками (который имеет код 4). Действительно, в уникоде кирилличные символы находятся в четвертом блоке [3]. Таким образом, все кирилличные символы в UTF-16LE находятся в диапазоне `0x400-0x4FF`.

Вернемся к примеру, где одна и та же строка написана разными языками. Здесь посмотрим в кодировке UTF-16LE.

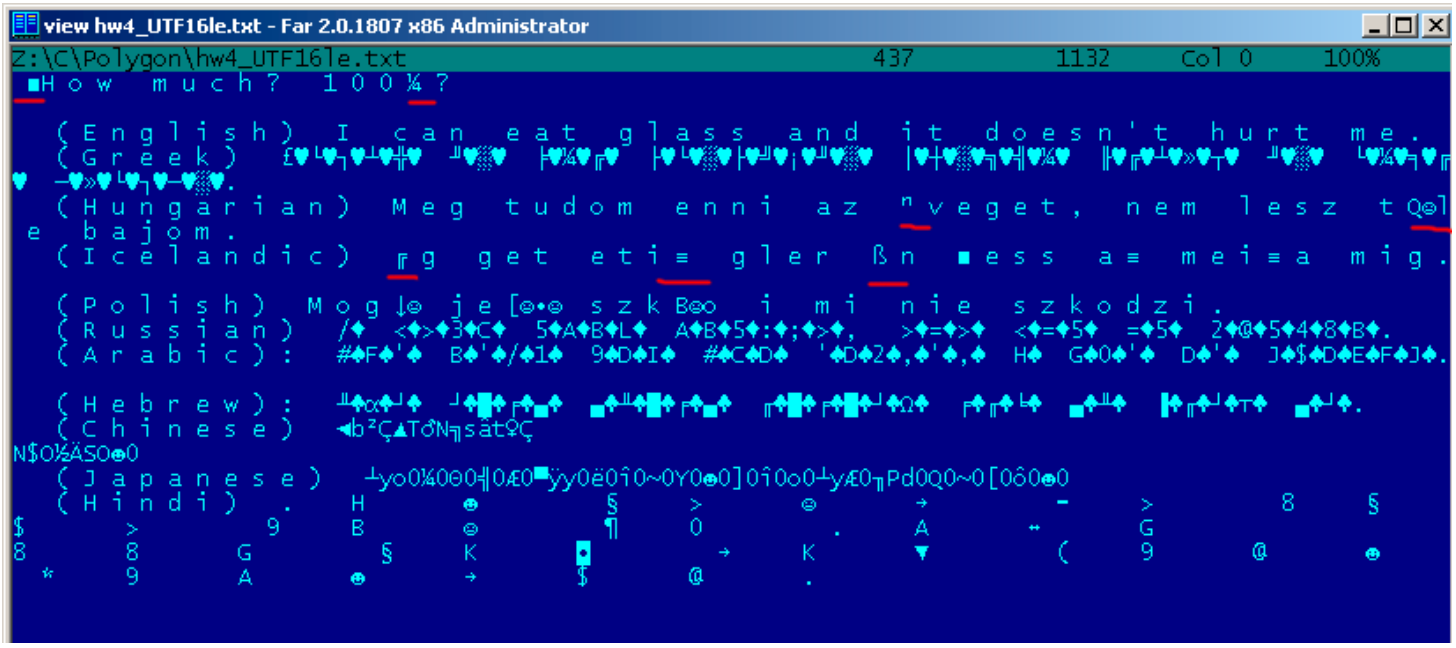


Рис. 36.6: FAR: UTF-16LE

Здесь мы также видим BOM в самом начале. Все латинские буквы перемежаются с нулевыми байтом. Некоторые буквы с диакритическими знаками (венгерский и исландский языки) также подчеркнуты красным.

36.2 Сообщения об ошибках и отладочные сообщения

Очень сильно помогают отладочные сообщения, если они имеются. В некотором смысле, отладочные сообщения, это отчет о том, что сейчас происходит в программе. Зачастую, это `printf()`-подобные функции, которые пишут куда-нибудь в лог, а бывает так что и не пишут ничего, но вызовы остались, так как эта сборка — не отладочная, а *release*. Если в отладочных сообщениях дампятся значения некоторых локальных или глобальных переменных, это тоже может помочь, как минимум, узнать их имена. Например, в Oracle RDBMS одна из таких функций: `ksdwrt()`.

Осмысленные текстовые строки вообще очень сильно могут помочь. Дизассемблер IDA может сразу указать, из какой функции и из какого её места используется эта строка. Встречаются и смешные случаи[4].

Сообщения об ошибках также могут помочь найти то что нужно. В Oracle RDBMS сигнализация об ошибках проходит при помощи вызова некоторой группы функций. Тут еще немного об этом[5].

[3] https://en.wikipedia.org/wiki/Cyrillic_(Unicode_block)
[4] http://blog.yurichev.com/node/32
[5] http://blog.yurichev.com/node/43

Можно довольно быстро найти, какие функции сообщают о каких ошибках, и при каких условиях. Это, кстати, одна из причин, почему в защите софта от копирования, бывает так, что сообщение об ошибке заменяется невнятным кодом или номером ошибки. Мало кому приятно, если взломщик быстро поймет, из за чего именно срабатывает защита от копирования, просто по сообщению об ошибке.

Один из примеров шифрования сообщений об ошибке, здесь: 55.2.

Глава 37

Вызовы assert()

Может также помочь наличие `assert()` в коде: обычно этот макрос оставляет название файла-исходника, номер строки, и условие.

Наиболее полезная информация содержится в assert-условии, по нему можно судить по именам переменных или именам полей структур. Другая полезная информация — это имена файлов, по их именам можно попытаться предположить, что там за код. Также, по именам файлов можно опознать какую-либо очень известную опен-сорсную библиотеку.

Листинг 37.1: Пример информативных вызовов assert()

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Полезно “гуглить” и условия и имена файлов, это может вывести вас к опен-сорсной бибилотеке. Например, если “погуглить” “sp->lzw_nbits <= BITS_MAX”, это вполне предсказуемо выводит на опенсорсный код, что-то связанное с LZW-компрессией.

Глава 38

Константы

Люди, включая программистов, часто используют круглые числа вроде 10, 100, 1000, в т.ч. и в коде.

Практикующие реверсеры, обычно, хорошо знают их в шестнадцатеричном представлении: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Некоторые алгоритмы, особенно криптографические, используют хорошо различимые константы, которые при помощи IDA легко находить в коде.

Например алгоритм MD5[1] инициализирует свои внутренние переменные так:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Если в коде найти использование этих четырех констант подряд — очень высокая вероятность что эта функция имеет отношение к MD5.

Еще такой пример это алгоритмы CRC16/CRC32, часто, алгоритмы вычисления контрольной суммы по CRC используют заранее заполненные таблицы, вроде:

Листинг 38.1: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        ...
```

См. также таблицу CRC32: 17.4.

38.1 Magic numbers

Немало форматов файлов определяет стандартный заголовок файла где используются *magic numbers*[2].

Скажем, все исполняемые файлы для Win32 и MS-DOS начинаются с двух символов "MZ"[3].

В начале MIDI-файла должно быть "MThd". Если у нас есть использующая для чего-нибудь MIDI-файлы программа очень вероятно, что она будет проверять MIDI-файлы на правильность хотя бы проверяя первые 4 байта.

Это можно сделать при помощи:

(*buf* указывает на начало загруженного в память файла)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...либо вызвав функцию сравнения блоков памяти `memcmp()` или любой аналогичный код, вплоть до инструкции `CMPSB` (B.6.3).

Найдя такое место мы получаем как минимум информацию о том, где начинается загрузка MIDI-файла, во-вторых, мы можем увидеть где располагается буфер с содержимым файла, и что еще оттуда берется, и как используется.

[1] http://ru.wikipedia.org/wiki/MD5

[2] http://en.wikipedia.org/wiki/Magic_number_(programming)

[3] http://en.wikipedia.org/wiki/DOS_MZ_executable

38.1.1 DHCP

Это касается также и сетевых протоколов. Например, сетевые пакеты протокола DHCP содержат так называемую *magic cookie*: `0x63538263`. Какой-либо код, генерирующий пакеты по протоколу DHCP где-то и как-то должен внедрять в пакет также и эту константу. Найдя её в коде мы сможем найти место где происходит это и не только это. *Что-либо* что получает пакеты по DHCP должно где-то как-то проверять *magic cookie*, сравнивая это поле пакета с константой.

Например, берем файл dhcpcore.dll из Windows 7 x64 и ищем эту константу. И находим, два раза: оказывается, эта константа используется в функциях с красноречивыми названиями `DhcpExtractOptionsForValidation()` и `DhcpExtractFullOptions()`:

Листинг 38.2: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ; DATA XREF: ↙
    ↳ DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ; DATA XREF: ↙
    ↳ DhcpExtractFullOptions+97
```

А вот те места в функциях где происходит обращение к константам:

Листинг 38.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

И:

Листинг 38.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

38.2 Поиск констант

В IDA это очень просто, Alt-B или Alt-I. А для поиска константы в большом количестве файлов, либо для поиска их в неисполняемых файлах, я написал небольшую утилиту *binary grep*[4].

[4] https://github.com/yurichev/bgrep

Глава 39

Поиск нужных инструкций

Если программа использует инструкции сопроцессора, и их не очень много, то можно попробовать вручную проверить отладчиком какую-то из них.

К примеру, нас может заинтересовать, при помощи чего Microsoft Excel считает результаты формул введенных пользователем. Например, операция деления.

Если загрузить excel.exe (из Office 2010) версии 14.0.4756.1000 в IDA, затем сделать полный листинг и найти все инструкции `FDIV` (но кроме тех, которые в качестве второго операнда используют константы — они, очевидно, не подходят нам):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

. . . то окажется, что их всего 144.

Мы можем вводить в Excel строку вроде `=(1/3)` и проверить все эти инструкции.

Проверяя каждую инструкцию в отладчике или tracer (проверять эти инструкции можно по 4 за раз), окажется, что нам везет и срабатывает всего лишь 14-я по счету:

```
.text:3011E919 DC 33                                    fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

В `ST(0)` содержится первый аргумент (1), второй содержится в `[EBX]`.

Следующая за `FDIV` инструкция записывает результат в память:

```
.text:3011E91B DD 1E                                    fstp    qword ptr [esi]
```

Если поставить breakpoint на ней, то мы можем видеть результат:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

А также, в рамках пранка[1], модифицировать его на лету:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel показывает в этой ячейке 666, что окончательно убеждает нас в том что мы нашли нужное место.

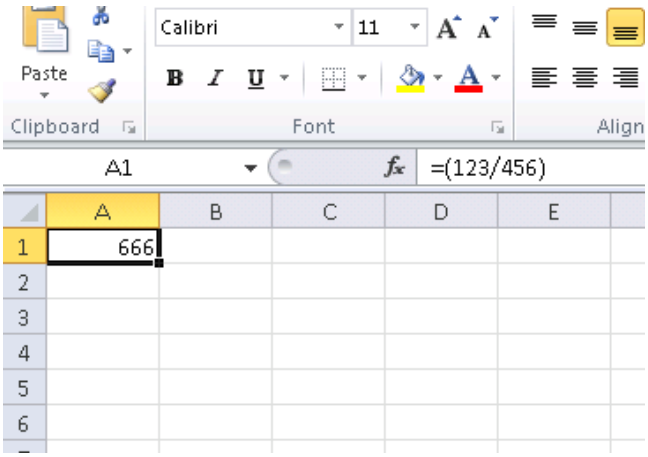


Рис. 39.1: Пранк сработал

Если попробовать ту же версию Excel, только x64, то окажется что там инструкций `FDIV` всего 12, причем нужная нам — третья по счету.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Видимо, все дело в том, что много операций деления переменных типов *float* и *double* компилятор заменил на SSE-инструкции вроде `DIVSD`, коих здесь теперь действительно много (`DIVSD` присутствует в количестве 268 инструкций).

[1] practical joke

Глава 40

Подозрительные паттерны кода

40.1 Инструкции XOR

Инструкции вроде `XOR op, op` (например, `XOR EAX, EAX`) обычно используются для обнуления регистра, однако, если операнды разные, то применяется операция именно *исключающего или*. Эта операция очень редко применяется в обычном программировании, но применяется очень часто в криптографии, включая любительскую. Особенно подозрительно, если второй операнд это большое число. Это может указывать на шифрование, вычисление контрольной суммы, итд.

Одно из исключений из этого наблюдения о котором стоит сказать, то, что генерация и проверка значения "канарейки" (16.3) часто происходит используя инструкцию `XOR`.

Этот AWK-скрипт можно использовать для обработки листингов (.lst) созданных IDA:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp")↙
    ↪  { print $1, $2, tmp, ",", $4 } }' filename.lst
```

Нельзя также забывать, что если использовать подобный скрипт, то, возможно, он захватит и неверно дизассемблированный код (28).

40.2 Вручную написанный код на ассемблере

Современные компиляторы не генерируют инструкции `LOOP` и `RCL`. С другой стороны, эти инструкции хорошо знакомы кодерам, предпочитающим писать прямо на ассемблере. Подобные инструкции отмечены как (M) в списке инструкций в приложении: B.6. Если такие инструкции встретились, можно сказать с какой-то вероятностью, что этот фрагмент кода написан вручную.

Также, пролог/эпилог функции обычно не встречается в ассемблерном коде, написанном вручную.

Как правило, в вручную написанном коде, нет никакого четкого метода передачи аргументов в функцию.

Пример из ядра Windows 2003 (файл ntoskrnl.exe):

```
MultiplyTest proc near               ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                          ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                       ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near               ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
```

```
            mov     eax, 417A000h
            mul     ecx
            cmp     edx, 2
            stc
            jnz     short locret_62057F
            cmp     eax, 0FE7A000h
            stc
            jnz     short locret_62057F
            clc
locret_62057F:                          ; CODE XREF: Multiply+10
                                        ; Multiply+18
            retn
Multiply    endp
```

Действительно, если заглянуть в исходные коды WRK[1] v1.2, данный код можно найти в файле *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

[1] Windows Research Kernel

Глава 41

Использование magic numbers для трассировки

Нередко бывает нужно узнать, как используется то или иное значение, прочитанное из файла либо взятое из пакета, принятого по сети. Часто, ручное слежение за нужной переменной это трудный процесс. Один из простых методов (хотя и не полностью надежный на 100%) это использование вашей собственной *magic number*.

Это чем-то напоминает компьютерную томографию: пациенту перед сканированием вводят в кровь рентгеноконтрастный препарат, хорошо отсвечивающий в рентгеновских лучах. Известно, как кровь нормального человека расходится, например, по почкам, и если в этой крови будет препарат, то при томографии будет хорошо видно, достаточно ли хорошо кровь расходится по почкам и нет ли там камней, например, и прочих образований.

Мы можем взять 32-битное число вроде `0x0badf00d`, либо чью-то дату рождения вроде `0x11101979` и записать это, занимающее 4 байта число, в какое-либо место файла используемого исследуемой нами программой.

Затем, при трассировки этой программы, в том числе, при помощи tracer в режиме *code coverage*, а затем при помощи *grep* или простого поиска по текстовому файлу с результатами трассировки, мы можем легко увидеть, в каких местах кода использовалось это значение, и как.

Пример результата работы tracer в режиме *cc*, к которому легко применить утилиту *grep*:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Это справедливо также и для сетевых пакетов. Важно только, чтобы наш *magic number* был как можно более уникален и не присутствовал в самом коде.

Помимо tracer, такой эмулятор MS-DOS как DosBox, в режиме heavydebug, может писать в отчет информацию обо всех состояниях регистра на каждом шаге исполнения программы[1], так что этот метод может пригодиться и для исследования программ под DOS.

[1] См. также мой пост в блоге об этой возможности в DosBox: http://blog.yurichev.com/node/55

Глава 42

Прочее

RTTI (29.1.5)-информация также может быть полезна для идентификации Си++-классов.

Глава 43

Старые методы, тем не менее, интересные

43.1 Сравнение "снимков" памяти

Метод простого сравнения двух снимков памяти для поиска изменений часто применялся для взлома игр на 8-битных компьютерах и взлома файлов с записанными рекордными очками.

К примеру, если вы имеете загруженную игру на 8-битном компьютере (где самой памяти не очень много, но игра занимает еще меньше), и вы знаете что сейчас у вас, условно, 100 пуль, вы можете сделать "снимок" всей памяти и сохранить где-то. Затем просто стреляете куда угодно, у вас станет 99 пуль, сделать второй "снимок", и затем сравнить эти два снимка: где-то наверняка должен быть байт, который в начале был 100, а затем стал 99. Если учесть, что игры на тех маломощных домашних компьютерах обычно были написаны на ассемблере и подобные переменные там были глобальные, то можно с уверенностью сказать, какой адрес в памяти всегда отвечает за количество пуль. Если поискать в дизассемблированном коде игры все обращения по этому адресу, несложно найти код, отвечающий за уменьшение пуль и записать туда инструкцию NOP или несколько NOP-в, так мы получим игру в которой у игрока всегда будет 100 пуль, например. А так как игры на тех домашних 8-битных компьютерах всегда загружались по одним и тем же адресам, и версий одной игры редко когда было больше одной продолжительное время, то геймеры-энтузиасты знали, по какому адресу (используя инструкцию языка BASIC POKE) что записать после загрузки игры, чтобы хакнуть её. Это привело к появлению списков "читов" состоящих из инструкций POKE, публикуемых в журналах посвященным 8-битным играм. См. также: http://en.wikipedia.org/wiki/PEEK_and_POKE.

Точно так же легко модифицировать файлы с сохраненными рекордами (кто сколько очков набрал), впрочем, это может сработать не только с 8-битными играми. Нужно заметить, какой у вас сейчас рекорд и где-то сохранить файл с очками. Затем, когда очков станет другое количество, просто сравнить два файла, можно даже DOS-утилитой FC[1] (файлы рекордов, часто, бинарные). Где-то будут отличаться несколько байт, и легко будет увидеть, какие именно отвечают за количество очков. Впрочем, разработчики игр осведомлены о таких хитростях и могут защититься от этого.

43.1.1 Реестр Windows

А еще можно вспомнить сравнение реестра Windows до инсталляции программы и после. Это также популярный метод поиска, какие элементы реестра программа будет использовать.

[1] утилита MS-DOS для сравнения двух файлов побайтово

Часть IV

Специфичное для ОС

Глава 44

Способы передачи аргументов при вызове функций

44.1 cdecl

Этот способ передачи аргументов через стек чаще всего используется в языках Си/Си++.

Вызывающая функция заталкивает в стек аргументы в обратном порядке: сначала последний аргумент в стек, затем предпоследний, и в самом конце — первый аргумент. Вызывающая функция должна также затем вернуть указатель стека в нормальное состояние, после возврата вызываемой функции.

Листинг 44.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

44.2 stdcall

Это почти то же что и *cdecl*, за исключением того, что вызываемая функция сама возвращает `ESP` в нормальное состояние, выполнив инструкцию `RET x` вместо `RET`, где `x = количество_аргументов * sizeof(int)`[1]. Вызывающая функция не будет корректировать указатель стека при помощи инструкции `add esp, x`.

Листинг 44.2: stdcall

```
push arg3
push arg2
push arg1
call function

function:
... do something ...
ret 12
```

Этот способ используется почти везде в системных библиотеках win32, но не в win64 (о win64 смотрите ниже).

Например, мы можем взять ф-цию из 7.1 и изменить её немного добавив модификатор `__stdcall`:

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

Он будет скомпилирован почти так же как и 7.2, но вы увидите `RET 12` вместо `RET`. SP не будет корректироваться в вызывающей ф-ции.

Как следствие, количество аргументов ф-ции легко узнать из инструкции `RETN n` просто разделите n на 4.

[1]Размер переменной типа *int* — 4 в x86-системах и 8 в x64-системах

Листинг 44.3: MSVC 2010

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_c$ = 16                                                ; size = 4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12                                      ; 0000000cH
_f2@12  ENDP

; ...
        push    3
        push    2
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

44.2.1 Функции с переменным количеством аргументов

Функции вроде `printf()`, должно быть, единственный случай функций в Си/Си++ с переменным количеством аргументов, но с их помощью можно легко проследить очень важную разницу между *cdecl* и *stdcall*. Начнем с того, что компилятор знает сколько аргументов было у `printf()`. Однако, вызываемая функция `printf()`, которая уже давно скомпилирована и находится в системной библиотеке MSVCRT.DLL (если говорить о Windows), не знает сколько аргументов ей передали, хотя может установить их количество по строке формата. Таким образом, если бы `printf()` была *stdcall*-функцией и возвращала указатель стека в первоначальное состояние подсчитав количество аргументов в строке формата, это была бы потенциально опасная ситуация, когда одна опечатка программиста могла бы вызывать неожиданные падения программы. Таким образом, для таких функций *stdcall* явно не подходит, а подходит *cdecl*.

44.3 fastcall

Это общее название для передачи некоторых аргументов через регистры, а всех остальных — через стек.На более старых процессорах, это работало потенциально быстрее чем *cdecl*/*stdcall* (ведь стек в памяти использовался меньше). Впрочем, на современных, намного более сложных CPU, выигрыша может не быть.

Это не стандартизированный способ, поэтому разные компиляторы делают это по-своему. Разумеется, если у вас есть, скажем, две DLL, одна использует другую, и обе они собраны с *fastcall* но разными компиляторами, очень вероятно, будут проблемы.

MSVC и GCC передает первый и второй аргумент через `ECX` и `EDX` а остальные аргументы через стек.

Указатель стека должен быть возвращен в первоначальное состояние вызываемой функцией, как в случае *stdcall*.

Листинг 44.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4
```

Например, мы можем взять ф-цию из 7.1 и изменить её немного добавив модификатор `__fastcall`:

```
int __fastcall f3 (int a, int b, int c)
{
        return a*b+c;
};
```

Вот как он будет скомпилирован:

Листинг 44.5: MSVC 2010 /Ox /Ob0

```
_c$ = 8                                                 ; size = 4
@f3@12  PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

Видно что вызываемая ф-ция сама возвращает SP при помощи инструкции RETN с операндом. Так что и здесь можно легко вычислять количество аргументов.

44.3.1 GCC regparm

Это в некотором роде, развитие *fastcall*[2]. Опцией -mregparm=x можно указывать, сколько аргументов компилятор будет передавать через регистры. Максимально 3. В этом случае будут задействованы регистры EAX, EDX и ECX.

Разумеется, если аргументов у функции меньше трех, то будет задействована только часть регистров.

Вызывающая функция возвращает указатель стека в первоначальное состояние.

Для примера, см. (17.1.1).

44.3.2 Watcom/OpenWatcom

Здесь это называется "register calling convention". Первые 4 аргумента передаются через регистры EAX, EDX, EBX и ECX. Все остальные — через стек. Функции имеют символ подчеркивания добавленный к концу имени ф-ции, для отличия их от тех, которые имеют другой способ передачи аргументов.

44.4 thiscall

В C++, это передача в функцию-метод указателя *this* на объект.

В MSVC указатель *this* обычно передается в регистре ECX.

В GCC указатель *this* обычно передается как самый первый аргумент. Таким образом, внутри будет видно: у всех функций-методов на один аргумент больше.

Для примера, см. (29.1.1).

[2] http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm

44.5 x86-64

44.5.1 Windows x64

В win64 метод передачи всех параметров немного похож на `fastcall`. Первые 4 аргумента записываются в регистры `RCX`, `RDX`, `R8`, `R9`, а остальные — в стек. Вызывающая функция также должна подготовить место из 32 байт или для четырех 64-битных значений, чтобы вызываемая функция могла сохранить там первые 4 аргумента. Короткие функции могут использовать переменные прямо из регистров, но бóльшие могут сохранять их значения на будущее.

Вызывающая функция должна вернуть указатель стека в первоначальное состояние.

Это же соглашение используется и в системных библиотеках Windows x86-64 (вместо *stdcall* в win32).

Пример:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

Листинг 44.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72                                 ; 00000048H

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72                                 ; 00000048H
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72                                 ; 00000048H

        mov     eax, DWORD PTR g$[rsp]
```

```
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT:$SG2937
        call    printf

        add     rsp, 72                                 ; 00000048H
        ret     0
f1      ENDP
```

Здесь мы легко видим как 7 аргументов передаются: 4 через регистры и остальные 3 через стек. Код пролога ф-ции f1() сохраняет аргументы в “scratch space” — место в стеке предназначенное именно для этого. Это делается потому что компилятор может быть не уверен, достаточно ли ему будет остальных регистров для работы исключая эти 4, которые иначе будут заняты аргументами до конца исполнения ф-ции. Выделение “scratch space” в стеке лежит на ответственности вызывающей ф-ции.

Листинг 44.7: MSVC 2012 /Ox /0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72                                 ; 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72                                 ; 00000048H
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72                                 ; 00000048H

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
```

```
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72                                 ; 00000048H
        ret     0
main    ENDP
```

Если компилировать этот пример с оптимизацией, то выйдет почти то же самое, только "scratch space" не используется, потому что незачем.

Обратите также внимание на то как MSVC 2012 оптимизирует примитивную загрузку значений в регистры используя `LEA` (B.6.2). Я не уверен, что это того стоит, но может быть.

Передача *this*

Указатель *this* передается через `RCX`, первый аргумент метода через `RDX`, итд. Для примера, см. также: 29.1.1.

44.5.2 Linux x64

Метод передачи аргументов в Linux для x86-64 почти такой же, как и в Windows, но 6 регистров используется вместо 4 (`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`), и здесь нет "scratch space", но callee может сохранять значения регистров в стеке, если нужно.

Листинг 44.8: GCC 4.7.3 -O3

```
.LC0:
        .string "%d %d %d %d %d %d %d\n"
f1:
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main:
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

N.B.: здесь значения записываются в 32-битные части регистров (например EAX) а не в весь 64-битный регистр (RAX). Это связано с тем что в x86-64, запись в младшую 32-битную часть 64-битного регистра автоматически обнуляет старшие 32 бита. Вероятно, это сделано для упрощения портирования кода под x86-64.

44.6 Возвращение переменных типа *float*, *double*

Во всех соглашениях кроме Win64, переменная типа *float* или *double* возвращается через регистр FPU `ST(0)`.

В Win64 переменные типа *float* и *double* возвращаются в регистре `XMM0` вместо `ST(0)`.

44.7 Модификация аргументов

Иногда программисты на Си/Си++ (и не только этих ЯП) задаются вопросом, что будет если модифицировать аргументы? Ответ прост: аргументы хранятся в стеке, именно там и будет происходит модификация. А вызывающие ф-ции не использует их после вызова ф-ции (обратного случая в своей практике я не видел ни разу).

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

Листинг 44.9: MSVC 2012

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

Следовательно, модифицировать аргументы ф-ции можно запросто. Разумеется, если это не *references* в Си++ (29.3), и если вы не модифицируете данные по указателю (тогда эффект распространится не только на текущую ф-цию).

Глава 45

Thread Local Storage

Это область данных, отдельная для каждого треда. Каждый тред может хранить там то, что ему нужно. Один из известных примеров, это стандартная глобальная переменная в Си *errno*. Несколько тредов одновременно могут вызывать функции возвращающие код ошибки в *errno*, поэтому глобальная переменная здесь не будет работать корректно, для мультитредовых программ *errno* нужно хранить в в TLS.

В C++11 ввели модификатор *thread_local* , показывающий что каждый тред будет иметь свою версию этой переменной, и её можно инициализировать, и она расположена в TLS [1]:

Листинг 45.1: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

[2]

Если говорить о PE-файлах, то в исполняемом файле значение *tmp* будет именно в секции отведенной TLS.

[1] В C11 также есть поддержка тредов, хотя и опциональная
[2]Компилируется в MinGW GCC 4.8.1, но не в MSVC 2012

Глава 46

Системные вызовы (syscall-ы)

Как известно, все работающие процессы в ОС делятся на две категории: имеющие полный доступ ко всему "железу" ("kernel space") и не имеющие ("user space").

В первой категории ядро ОС и, обычно, драйвера.

Во второй категории всё прикладное ПО.

Это разделение очень важно для безопасности ОС: очень важно чтобы никакой процесс не мог испортить что-то в других процессах или даже в самом ядре ОС. С другой стороны, падающий драйвер или ошибка внутри ядра ОС обычно приводит к kernel panic или BSOD[1].

Защита x86-процессора устроена так что возможно разделить всё на 4 слоя защиты (rings), но и в Linux, и в Windows, используются только 2: ring0 ("kernel space") и ring3 ("user space").

Системные вызовы (syscall-ы) это точка где соединяются вместе оба эти пространства. Это, можно сказать, самое главное API предоставляемое прикладному ПО.

В Windows NT таблица сисколлов находится в SSDT[2].

Работа через syscall-ы популярна у авторов шеллкодов в вирусов, потому что там обычно бывает трудно определить адреса нужных ф-ций в системных библиотеках, а syscall-ами проще пользоваться, хотя и придется писать больше кода из-за более низкого уровня абстракции этого API. Также нельзя еще забывать, что номера syscall-ов, например, в Windows, могут отличаться от версии к версии.

46.1 Linux

В Linux вызов syscall-а обычно происходит через `int 0x80`. В регистре `EAX` передается номер вызова, в остальных регистрах —- параметры.

Листинг 46.1: Простой пример использования пары syscall-ов

```
section .text
global  _start

_start:
        mov     edx,len ; buf len
        mov     ecx,msg ; buf
        mov     ebx,1   ; file descriptor. stdout is 1
        mov     eax,4   ; syscall number. sys_write is 4
        int     0x80

        mov     eax,1   ; syscall number. sys_exit is 4
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Компиляция:

[1]Black Screen of Death

[2]System Service Dispatch Table

```
nasm -f elf32 1.s
ld 1.o
```

Полный список syscall-ов в Linux: http://syscalls.kernelgrok.com/.

Для перехвата и трассировки системных вызовов в Linux, можно применять strace(51).

46.2 Windows

Вызов происходит через `int 0x2e` либо используя специальную x86-инструкцию `SYSENTER`.

Полный список syscall-ов в Windows: http://j00ru.vexillium.org/ntapi/.

Смотрите также:

“Windows Syscall Shellcode” by Piotr Bania:
http://www.symantec.com/connect/articles/windows-syscall-shellcode.

Глава 47

Linux

47.1 Адресно-независимый код

Во время анализа динамических библиотек (.so) в Linux, часто можно заметить такой шаблонный код:

Листинг 47.1: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near         ; CODE XREF: sub_17350+3
.text:0012D5E3                                         ; sub_173CC+4 ...
.text:0012D5E3                 mov     ebx, [esp+0]
.text:0012D5E6                 retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near               ; CODE XREF: tmpfile+73

...

.text:000576C0                 push    ebp
.text:000576C1                 mov     ecx, large gs:0
.text:000576C8                 push    edi
.text:000576C9                 push    esi
.text:000576CA                 push    ebx
.text:000576CB                 call    __x86_get_pc_thunk_bx
.text:000576D0                 add     ebx, 157930h
.text:000576D6                 sub     esp, 9Ch

...

.text:000579F0                 lea     eax, (a__gen_tempname - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6                 mov     [esp+0ACh+var_A0], eax
.text:000579FA                 lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/↙
    ↪ posix/tempname.c"
.text:00057A00                 mov     [esp+0ACh+var_A8], eax
.text:00057A04                 lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ; "! \"invalid ↙
    ↪ KIND in __gen_tempname\""
.text:00057A0A                 mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12                 mov     [esp+0ACh+var_AC], eax
.text:00057A15                 call    __assert_fail
```

Все указатели на строки корректируются при помощи некоторой константы из регистра EBX, которая вычисляется в начале каждой функции. Это так называемый адресно-независимый код (PIC), он предназначен для исполнения будучи расположенным по любому адресу в памяти, вот почему он не содержит никаких абсолютных адресов в памяти.

PIC был очень важен в ранних компьютерных системах и важен сейчас во встраиваемых[1], не имеющих поддержки виртуальной памяти (все процессы расположены в одном непрерывном блоке памяти). Он до сих

[1] embedded

пор используется в *NIX системах для динамических библиотек, потому что динамическая библиотека может использоваться одновременно в нескольких процессах, будучи загружена в память только один раз. Но все эти процессы могут загрузить одну и ту же динамическую библиотеку по разным адресам, вот почему динамическая библиотека должна работать корректно, не привязываясь к абсолютным адресам.

Простой эксперимент:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Скомпилируем в GCC 4.7.3 и посмотрим итоговый файл .so в IDA:

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Листинг 47.2: GCC 4.7.3

```
.text:00000440                 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near         ; CODE XREF: _init_proc+4
.text:00000440                                         ; deregister_tm_clones+4 ...
.text:00000440                 mov     ebx, [esp+0]
.text:00000443                 retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                 public f1
.text:00000570 f1              proc near
.text:00000570
.text:00000570 var_1C          = dword ptr -1Ch
.text:00000570 var_18          = dword ptr -18h
.text:00000570 var_14          = dword ptr -14h
.text:00000570 var_8           = dword ptr -8
.text:00000570 var_4           = dword ptr -4
.text:00000570 arg_0           = dword ptr  4
.text:00000570
.text:00000570                 sub     esp, 1Ch
.text:00000573                 mov     [esp+1Ch+var_8], ebx
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
.text:00000582                 mov     [esp+1Ch+var_4], esi
.text:00000586                 mov     eax, ds:(global_variable_ptr - 2000h)[ebx]
.text:0000058C                 mov     esi, [eax]
.text:0000058E                 lea     eax, (aReturningD - 2000h)[ebx] ; "returning %d\n"
.text:00000594                 add     esi, [esp+1Ch+arg_0]
.text:00000598                 mov     [esp+1Ch+var_18], eax
.text:0000059C                 mov     [esp+1Ch+var_1C], 1
.text:000005A3                 mov     [esp+1Ch+var_14], esi
.text:000005A7                 call    ___printf_chk
.text:000005AC                 mov     eax, esi
.text:000005AE                 mov     ebx, [esp+1Ch+var_8]
.text:000005B2                 mov     esi, [esp+1Ch+var_4]
.text:000005B6                 add     esp, 1Ch
.text:000005B9                 retn
.text:000005B9 f1              endp
```

Так и есть: указатели на строку *«returning %d\n»* и переменную *global_variable* корректируются при каждом исполнении функции Функция `__x86_get_pc_thunk_bx()` возвращает адрес точки после вызова самой себя (здесь: `0x57C`) в `EBX`. Это очень простой способ получить значение указателя на текущую инструкцию (`EIP`)

в произвольном месте. Константа 0x1A84 связана с разницей между началом этой функции и так называемой *Global Offset Table Procedure Linkage Table* (GOT PLT), секцией, сразу же за *Global Offset Table* (GOT), где находится указатель на *global_variable*. IDA показывает смещения уже обработанными, чтобы их было проще понимать, но на самом деле код такой:

```
.text:00000577                call    __x86_get_pc_thunk_bx
.text:0000057C                add     ebx, 1A84h
.text:00000582                mov     [esp+1Ch+var_4], esi
.text:00000586                mov     eax, [ebx-0Ch]
.text:0000058C                mov     esi, [eax]
.text:0000058E                lea     eax, [ebx-1A30h]
```

Так что, EBX указывает на секцию GOT PLT и для вычисления указателя на *global_variable*, которая хранится в GOT, нужно вычесть 0xC. А чтобы вычислить указатель на *«returning %d\n»*, нужно вычесть 0x1A30.

Кстати, вот зачем в AMD64 появилась поддержка адресации относительно RIP[2], просто для упрощения PIC-кода.

Скомпилируем тот же код на Си при помощи той же версии GCC, но для x64.

IDA упростит код на выходе убирая упоминания RIP, так что я буду использовать *objdump* вместо:

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0x2008b9]        # 200fe0 <_DYNAMIC+0↙
    ↳ x1d0>
 727:   53                      push   rbx
 728:   89 fb                   mov    ebx,edi
 72a:   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        # 751 <_fini+0x9>
 731:   bf 01 00 00 00          mov    edi,0x1
 736:   03 18                   add    ebx,DWORD PTR [rax]
 738:   31 c0                   xor    eax,eax
 73a:   89 da                   mov    edx,ebx
 73c:   e8 df fe ff ff          call   620 <__printf_chk@plt>
 741:   89 d8                   mov    eax,ebx
 743:   5b                      pop    rbx
 744:   c3                      ret
```

0x2008b9 это разница между адресом инструкции по 0x720 и *global_variable*, а 0x20 это разница между инструкцией по 0x72A и строкой *«returning %d\n»*.

Как видно, необходимость очень часто пересчитывать адреса делает исполнение немного медленнее (хотя это и стало лучше в x64). Так что если вы заботитесь о скорости исполнения, то, наверное, нужно задуматься о статической компоновке (static linking) ([11]).

47.1.1 Windows

Такой механизм не используется в Windows DLL. Если загрузчику в Windows приходится загружать DLL в другое место, он "патчит" DLL прямо в памяти (на местах *FIXUP*-ов) чтобы скорректировать все адреса. Это приводит к тому что загруженную один раз DLL нельзя использовать одновременно в разных процессах, желающих расположить её по разным адресам — потому что каждый загруженный в память экземпляр DLL *доводится* до того чтобы работать только по этим адресам.

47.2 Трюк с *LD_PRELOAD* в Linux

Это позволяет загружать свои динамические библиотеки перед другими, даже перед системными, такими как libc.so.6.

Что в свою очередь, позволяет "подставлять" написанные нами ф-ции перед оригинальными из системных библиотек. Например, легко перехватывать все вызовы к time(), read(), write(), и т.д.

Попробуем узнать, сможем ли мы обмануть утилиту uptime Как известно, она сообщает, как долго компьютер работает. При помощи strace(51), можно увидеть, что эту информацию утилита получает из файла /proc/uptime :

```
$ strace uptime
...
```

[2]указатель инструкций в AMD64

```
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

Это не реальный файл на диске, это виртуальный файл, содержимое которого генерируется на лету в ядре Linux. Там просто два числа:

```
$ cat /proc/uptime
416690.91 415152.03
```

Из wikipedia, можно узнать:

> The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

[3]

Попробуем написать свою динамическую библиотеку, в которой будет open(), read(), close() с нужной нам функциональностью.

Во-первых, наш open() будет сравнивать имя открываемого файла с тем что нам нужно, и если да, то будет запоминать дескриптор открытого файла. Во-вторых, read(), если будет вызываться для этого дескриптора, будет подменять вывод, а в остальных случаях, будет вызывать настоящий read() из libc.so.6. А также close(), будет следить, закрывается ли файл за которым мы следим.

Для того чтобы найти адреса настоящих ф-ций в libc.so.6, используем dlopen() и dlsym().

Нам это нужно, потому что нам нужно передавать управление “настоящим” ф-циями.

С другой стороны, если бы мы перехватывали, скажем, strcmp(), и следили бы за всеми сравнениями строк в программе, то, наверное, strcmp() можно было бы и самому реализовать, не пользуясь настоящей ф-цией [4].

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;
```

[3] https://en.wikipedia.org/wiki/Uptime

[4] Например, посмотрите как обеспечивается простейший перехват strcmp() в статье [5] от Yong Huang

```
        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

Компилируем как динамическую библиотеку:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Запускаем *uptime*, подгружая нашу библиотеку перед остальными:

```
LD_PRELOAD=‘pwd‘/fool_uptime.so uptime
```

Видим такое:

```
01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Если переменная окружения *LD_PRELOAD* будет всегда указывать на путь и имя файла нашей библиотеки, то она будет загружаться для всех запускаемых программ.

Еще примеры:

- Перехват time() в Sun Solaris http://yurichev.com/mirrors/LD_PRELOAD/sun_hack.txt
- Очень простой перехват strcmp() (Yong Huang) http://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
- Kevin Pulo — Fun with LD_PRELOAD. Много примеров и идей. http://yurichev.com/mirrors/LD_PRELOAD/lca2009.pdf
- Перехват ф-ций работы с файлами для компрессии и декомпрессии файлов на лету (zlibc). ftp://metalab.unc.edu/pub/Linux/libs/compression

Глава 48

Windows NT

48.1 CRT (win32)

Начинается ли исполнение программы прямо с ф-ции `main()`? Нет, не начинается. Если открыть любой исполняемый файл в IDA или Hiew, то OEP[1] указывает на совсем другой код. Это код, который делает некоторые приготовления перед тем как запустить ваш код. Он называется стартап-код или CRT-код (C RunTime).

Ф-ция `main()` принимает на вход массив из параметров, переданных в командной строке, а также переменные окружения. Но в реальности в программу передается командная строка в виде простой строки, это именно CRT-код находит там пробелы и разрезает строку на части. CRT-код также готовит массив переменных окружения `envp`. В GUI[2]-приложениях win32, вместо `main()` имеется ф-ция `WinMain` со своими аргументами:

```
int CALLBACK WinMain(
  _In_  HINSTANCE hInstance,
  _In_  HINSTANCE hPrevInstance,
  _In_  LPSTR lpCmdLine,
  _In_  int nCmdShow
);
```

CRT-код готовит и их.

А также, число, возвращаемое ф-цией `main()`, это код ошибки возвращаемый программой. В CRT это значение передается в `ExitProcess()`, принимающей в качестве аргумента код ошибки.

Как правило, каждый компилятор имеет свой CRT-код.

Вот типичный для MSVC 2008 CRT-код.

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

        push    14h
        push    offset stru_4092D0
        call    __SEH_prolog4
        mov     eax, 5A4Dh
        cmp     ds:400000h, ax
        jnz     short loc_401096
        mov     eax, ds:40003Ch
        cmp     dword ptr [eax+400000h], 4550h
        jnz     short loc_401096
        mov     ecx, 10Bh
        cmp     [eax+400018h], cx
        jnz     short loc_401096
```

[1] Original Entry Point
[2] Graphical user interface

```
        cmp     dword ptr [eax+400074h], 0Eh
        jbe     short loc_401096
        xor     ecx, ecx
        cmp     [eax+4000E8h], ecx
        setnz   cl
        mov     [ebp+var_1C], ecx
        jmp     short loc_40109A


loc_401096: ; CODE XREF: ___tmainCRTStartup+18
            ; ___tmainCRTStartup+29 ...
        and     [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
        push    1
        call    __heap_init
        pop     ecx
        test    eax, eax
        jnz     short loc_4010AE
        push    1Ch
        call    _fast_error_exit
        pop     ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
        call    __mtinit
        test    eax, eax
        jnz     short loc_4010BF
        push    10h
        call    _fast_error_exit
        pop     ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
        call    sub_401F2B
        and     [ebp+ms_exc.disabled], 0
        call    __ioinit
        test    eax, eax
        jge     short loc_4010D9
        push    1Bh
        call    __amsg_exit
        pop     ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
        call    ds:GetCommandLineA
        mov     dword_40B7F8, eax
        call    ___crtGetEnvironmentStringsA
        mov     dword_40AC60, eax
        call    __setargv
        test    eax, eax
        jge     short loc_4010FF
        push    8
        call    __amsg_exit
        pop     ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
        call    __setenvp
        test    eax, eax
        jge     short loc_401110
        push    9
        call    __amsg_exit
        pop     ecx
```

```
loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
        push    1
        call    __cinit
        pop     ecx
        test    eax, eax
        jz      short loc_401123
        push    eax
        call    __amsg_exit
        pop     ecx

loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
        mov     eax, envp
        mov     dword_40AC80, eax
        push    eax             ; envp
        push    argv            ; argv
        push    argc            ; argc
        call    _main
        add     esp, 0Ch
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN28
        push    eax             ; uExitCode
        call    $LN32

$LN28:      ; CODE XREF: ___tmainCRTStartup+105
        call    __cexit
        jmp     short loc_401186


$LN27:      ; DATA XREF: .rdata:stru_4092D0
        mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
        mov     ecx, [eax]
        mov     ecx, [ecx]
        mov     [ebp+var_24], ecx
        push    eax
        push    ecx
        call    __XcptFilter
        pop     ecx
        pop     ecx

$LN24:
        retn


$LN14:      ; DATA XREF: .rdata:stru_4092D0
        mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 401044
        mov     eax, [ebp+var_24]
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN29
        push    eax             ; int
        call    __exit


$LN29:      ; CODE XREF: ___tmainCRTStartup+135
        call    __c_exit

loc_401186: ; CODE XREF: ___tmainCRTStartup+112
        mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
        mov     eax, [ebp+var_20]
        call    __SEH_epilog4
```

```
        retn
```

Здесь можно увидеть по крайней мере вызов ф-ции `GetCommandLineA()`, затем `setargv()` и `setenvp()`, которые, видимо, заполняют глобальные переменные-указатели `argc`, `argv`, `envp`.

В итоге, вызывается `main()` с этими аргументами.

Также имеются вызовы ф-ций с говорящими именами вроде `heap_init()`, `ioinit()`.

Куча действительно инициализируется в CRT: если вы попытаетесь использовать `malloc()`, программа упадет с такой ошибкой:

```
runtime error R6030
- CRT not initialized
```

Инициализация глобальных объектов в Си++ происходит до вызова `main()`, именно в CRT: 29.4.1.

Значение, возвращаемое из `main()` передается или в `cexit()`, или же в `$LN32`, которая далее вызывает `doexit()`.

Можно ли обойтись без CRT? Можно, если вы знаете что делаете.

В линкере от MSVC точка входа задается опцией `/ENTRY`.

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Компилируем в MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

Получаем вполне работающий .exe размером 2560 байт, внутри которого есть только PE-заголовок, инструкции, вызывающие `MessageBox`, две строки в сегменте данных, импортируемая из `user32.dll` ф-ция `MessageBox`, и более ничего.

Это работает, но вы уже не сможете вместо `main()` написать `WinMain` с его четырьмя аргументами. Вернее, написать-то сможете, но доступа к этим аргументам не будет, потому что они не будут подготовлены на момент исполнения.

Кстати, можно еще короче сделать .exe если уменьшить выравнивание PE[3]-секций (которое, по умолчанию, 4096 байт).

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Линкер скажет:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Получим .exe размером 720 байт. Он запускается в Windows 7 x86, но не x64 (там выдает ошибку при загрузке). При желании, размер можно еще сильнее ужать, но, как видно, возникают проблемы с совместимостью с разными версиями Windows.

48.2 Win32 PE

PE это формат исполняемых файлов, принятый в Windows.

Разница между .exe, .dll, и .sys в том, что у .exe и .sys обычно нет экспортов, только импорты.

У DLL[4], как и у всех PE-файлов, есть точка входа (OEP) (там располагается ф-ция DllMain()), но обычно эта ф-ция ничего не делает.

.sys это обычно драйвера устройств.

Для драйверов, Windows требует чтобы контрольная сумма в PE-файле была проставлена и была верной [5].

А начиная с Windows Vista, PE-файлы-драйвера должны быть также подписаны при помощи электронной подписи, иначе они не будут загружаться.

В начале всякого PE-файла есть крохотная DOS-программа, выводящая на консоль сообщение вроде “This program cannot be run in DOS mode.” — если запустить эту программу в DOS либо Windows 3.1, выведется это сообщение.

[3] Portable Executable: 48.2
[4] Dynamic-link library
[5] Например, Hiew(53) умеет её подсчитывать

48.2.1 Терминология

- Модуль — это отдельный файл, .exe или.dll.
- Процесс — это некая загруженная в память и работающая программа. Как правило состоит из одного .exe-файла и массы .dll-файлов.
- Память процесса — память с которой работает процесс. У каждого процесса — своя. Там обычно имеются загруженные модули, память стека, кучи, и т.д.
- VA[6] — это адрес, который будет использоваться в самой программе.
- Базовый адрес — это адрес, по которому модуль будет загружен в пространство процесса.
- RVA[7] — это VA-адрес минус базовый адрес. Многие адреса в таблицах PE-файла используют именно RVA-адреса.
- IAT[8] — массив адресов импортированных символов [9]. Иногда, директория `IMAGE_DIRECTORY_ENTRY_IAT` указывает на IAT. Важно отметить, что IDA (по крайней мере 6.1) может выделить псевдо-секцию с именем `.idata` для IAT, даже если IAT является частью совсем другой секции!
- INT[10] — массив имен символов для импортирования[11].

48.2.2 Базовый адрес

Дело в том, что несколько авторов модулей могут готовить DLL-файлы для других, и нет возможности договориться о том, какие адреса и кому будут отведены.

Поэтому, если у двух необходимых для загрузки процесса DLL одинаковые базовые адреса, одна из них будет загружена по этому базовому адресу, а вторая — по другому свободному месту в памяти процесса, и все виртуальные адреса во второй DLL будут скорректированы.

Очень часто линкер в MSVC генерирует .exe-файлы с базовым адресом `0x400000`, и с секцией кода начинающейся с `0x401000`. Это значит, что RVA начала секции кода — `0x1000`. А DLL часто генерируются этим линкером с базовым адресом `0x10000000` [12].

Помимо всего прочего, есть еще одна причина намеренно загружать модули по разным адресам, а точнее, по случайным.

Это ASLR[13] [14].

Дело в том, что некий шеллкод, пытающийся исполниться на зараженной системе, должен вызывать какие-то системные ф-ции.

И в старых ОС (в линейке Windows NT: до Windows Vista), системные DLL (такие как kernel32.dll, user32.dll) загружались все время по одним и тем же адресам, а если еще и вспомнить, что версии этих DLL редко менялись, то адреса отдельных ф-ций, можно сказать, фиксированы и шеллкод может вызывать их напрямую.

Чтобы избежать этого, методика ASLR загружает и вашу программу, и все модули ей необходимые, по случайным адресам, разным при каждом запуске.

В PE-файлах, поддержка ASLR отмечается выставлением флага `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [30].

48.2.3 Subsystem

Имеется также поле *subsystem*, обычно это native (.sys-драйвер), console (консольное приложение) или GUI (не консольное).

[6] Virtual Address
[7] Relative Virtual Address
[8] Import Address Table
[9] [24]
[10] Import Name Table
[11] [24]
[12] Это можно изменять опцией /BASE в линкере
[13] Address Space Layout Randomization
[14] https://ru.wikipedia.org/wiki/Address_Space_Layout_Randomization

48.2.4 Версия ОС

В PE-файле имеется минимальный номер версии Windows, необходимый для загрузки модуля. Соответствие номеров версий в файле и кодовых наименований Windows, можно посмотреть здесь[15].

Например, MSVC 2005 еще компилирует .exe-файлы запускающиеся на Windows NT4 (версия 4.00), а вот MSVC 2008 уже нет (генерируемые файлы имеют версию 5.00, для запуска необходима как минимум Windows 2000).

MSVC 2012 по умолчанию генерирует .exe-файлы версии 6.00, для запуска нужна как минимум Windows Vista, хотя изменив настройки компиляции [16], можно заставить генерировать и под Windows XP.

48.2.5 Секции

Разделение на секции присутствует, по-видимому, во всех форматах исполняемых файлов.

Сделано это для того, чтобы отделить код от данных, а данные — от константных данных.

- На секции кода будет стоять флаг *IMAGE_SCN_CNT_CODE* или *IMAGE_SCN_MEM_EXECUTE* — это исполняемый код.
- На секции данных — флаги *IMAGE_SCN_CNT_INITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* и *IMAGE_SCN_MEM_WRITE*.
- На пустой секции с неинициализированными данными — *IMAGE_SCN_CNT_UNINITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* и *IMAGE_SCN_MEM_WRITE*.
- А на секции с константными данными, то есть, защищенными от записи, могут быть флаги *IMAGE_SCN_CNT_INITIALIZED_DATA* и *IMAGE_SCN_MEM_READ* без *IMAGE_SCN_MEM_WRITE*. Если попытаться записать что-то в эту секцию, процесс упадет.

В PE-файле можно задавать название для секции, но это не важно. Часто (но не всегда) секция кода называется `.text`, секция данных — `.data`, константных данных — `.rdata` *(readable data)*. Еще популярные имена секций:

- `.idata` — секция импортов. IDA может создавать псевдо-секцию с этим же именем: 48.2.1.
- `.edata` — секция экспортов
- `.pdata` — секция содержащая информацию об исключениях в Windows NT для MIPS, IA64 и x64: 48.3.3
- `.reloc` — секция релоков
- `.bss` — неинициализированные данные
- `.tls` — thread local storage (TLS)
- `.rsrc` — ресурсы
- `.CRT` — может присутствует в бинарных файлах скомпилированных очень старыми версиями MSVC

Запаковщики/зашифровщики PE-файлов часто затирают имена секций, или меняют на свои.

В MSVC можно объявлять данные в произвольно названной секции [17].

Некоторые компиляторы и линкеры могут добавлять также секцию с отладочными символами и вообще отладочной информацией (например, MinGW). Хотя это не так в современных версиях MSVC (там принято отладочную информацию сохранять в отдельных PDB-файлах).

Вот как секция описывается в файле:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
```

[15] https://en.wikipedia.org/wiki/Windows_NT#Releases
[16] http://blogs.msdn.com/b/vcblog/archive/2012/10/08/10357555.aspx
[17] http://msdn.microsoft.com/en-us/library/windows/desktop/cc307397.aspx

```
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[18]

Еще немного терминологии: *PointerToRawData* называется "Offset" и *VirtualAddress* называется "RVA" в Hiew.

48.2.6 Релоки

Также известны как FIXUP-ы (по крайней мере в Hiew).

Это также присутствует почти во всех форматах загружаемых и исполняемых файлов [19].

Как видно, модули могут загружаться по другим базовым адресам, но как же тогда работать с глобальными переменными, например? Ведь нужно обращаться к ним по адресу. Одно из решений это адресно-независимый код(47.1). Но это далеко не всегда удобно.

Поэтому имеется таблица релоков. Там просто перечислены адреса мест в модуле подлежащими коррекции при загрузке по другому базовому адресу.

Например, по `0x410000` лежит некая глобальная переменная, и вот как обеспечивается её чтение:

```
A1 00 00 41 00          mov         eax,[000410000]
```

Базовый адрес модуля `0x400000`, а RVA глобальной переменной `0x10000`.

Если загружать модуль по базовому адресу `0x500000`, нужно чтобы адрес этой переменной в этой инструкции стал `0x510000`.

Как видно, адрес переменной закодирован в самой инструкции `MOV`, после байта `0xA1`.

Поэтому адрес четырех байт, после `0xA1`, записывается в таблицу релоков.

Если модуль загружается по другому базовому адресу загрузчик OC обходит все адреса в таблице, находит каждое 32-битное слово по этому адресу, отнимает от него настоящий, оригинальный базовый адрес (в итоге получается RVA), и прибавляет к нему новый базовый адрес.

А если модуль загружается по своему оригинальному базовому адресу, ничего не происходит.

Так можно обходиться со всеми глобальными переменными.

Релоки могут быть разных типов, однако в Windows для x86-процессоров, тип обычно *IMAGE_REL_BASED_HIGHLOW*.

Кстати, релоки маркируются темным в Hiew, например илл. 6.12.

48.2.7 Экспорты и импорты

Как известно, любая исполняемая программа должна как-то пользоваться сервисами OC и прочими DLL-библиотеками.

Можно сказать, что нужно связывать функции из одного модуля (обычно DLL) и места их вызовов в другом модуле (.exe-файл или другая DLL).

Для этого, у каждой DLL есть "экспорты", это таблица ф-ций плюс их адреса в модуле.

А у .exe-файла, либо DLL, есть "импорты", это таблица ф-ций требующихся для исполнения включая список имен DLL-файлов.

Загрузчик OC, после загрузки основного .exe-файла, проходит по таблице импортов: загружает дополнительные DLL-файлы, находит имена ф-ций среди экспортов в DLL и прописывает их адреса в IAT в головном .exe-модуле.

Как видно, во время загрузки, загрузчику нужно много сравнивать одни имена ф-ций с другими, а сравнение строк — это не очень быстрая процедура, так что, имеется также поддержка "ординалов" или "hint"-ов, это когда в таблице импортов проставлены номера ф-ций вместо их имен.

Так их быстрее находить в загружаемой DLL. В таблице экспортов ординалы присутствуют всегда.

К примеру, программы использующие библиотеки MFC[20], обычно загружают mfc*.dll по ординалам, и в таких программах, в INT, нет имен ф-ций MFC.

При загрузке такой программы в IDA, она спросит у вас путь к файлу mfc*.dll, чтобы установить имена ф-ций. Если в IDA не указать путь к этой DLL, то вместо имен ф-ций будет что-то вроде *mfc80_123*.

[18] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680341(v=vs.85).aspx
[19] Даже .exe-файлы в MS-DOS
[20] Microsoft Foundation Classes

Секция импортов

Под таблицу импортов и всё что с ней связано иногда отводится отдельная секция (с названием вроде `.idata`), но это не обязательно.

Импорты — это запутанная тема еще и из-за терминологической путаницы. Попробуем собрать всё в одно место.

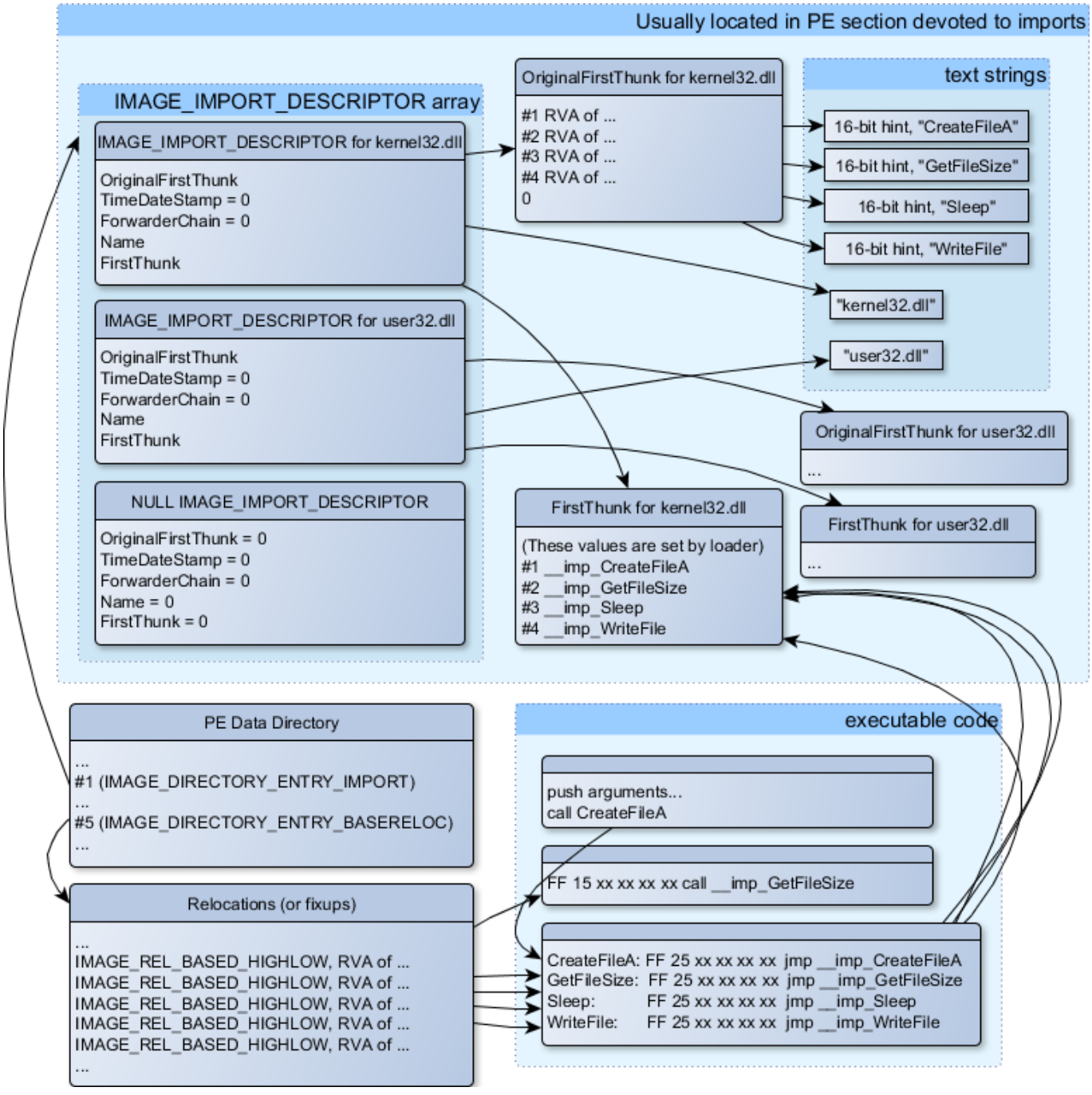


Рис. 48.1: схема, объединяющая все структуры в PE-файлы, связанные с импортами

Самая главная структура — это массив *IMAGE_IMPORT_DESCRIPTOR*. Каждый элемент на каждую импортируемую DLL.

У каждого элемента есть RVA-адрес текстовой строки (имя DLL) (*Name*).

OriginalFirstThink это RVA-адрес таблицы INT. Это массив RVA-адресов, каждый из которых указывает на текстовую строку где записано имя ф-ции. Каждую строку предваряет 16-битное число (“hint”) — “ординал” ф-ции.

Если при загрузке удается найти ф-цию по ординалу, тогда сравнение текстовых строк не будет происходить. Массив оканчивается нулем. Есть также указатель на таблицу IAT с названием *FirstThunk*, это просто RVA-адрес места, где загрузчик будет проставлять адреса найденных ф-ций.

Места где загрузчик проставляет адреса, IDA именует их так: *__imp_CreateFileA*, и т.д.
Есть по крайней мере два способа использовать адреса, проставленные загрузчиком.

- В коде будут просто инструкции вроде *call __imp_CreateFileA*, а так как, поле с адресом импортируемой ф-ции это как бы глобальная переменная, то в таблице релоков добавляется адрес (плюс 1 или 2) в инструкции *call*, на случай если модуль будет загружен по другому базовому адресу.

  Но как видно, это приводит к увеличению таблицы релоков. Ведь вызовов импортируемой ф-ции у вас в модуле может быть очень много. К тому же, чем больше таблица релоков, тем дольше загрузка.

- На каждую импортируемую ф-цию выделяется только один переход на импортируемую ф-цию используя инструкцию `JMP` плюс релок на эту инструкцию. Такие места-"переходники" называются также "thunk"-ами. А все вызовы импортируемой ф-ции это просто инструкция `CALL` на соответствующий "thunk". В данном случае, дополнительные релоки не нужны, потому что эти CALL-ы имеют относительный адрес, и корректировать их не надо.

Оба этих два метода могут комбинироваться. Вероятно, линкер создает отдельный "thunk", если вызовов слишком много, но по умолчанию — не создает.

Кстати, массив адресов ф-ций, на который указывает FirstThunk, не обязательно может быть в секции IAT. К примеру, я написал утилиту PE_add_import[21] для добавления импорта в уже существующий .exe-файл. Раньше, в прошлых версиях утилиты, на месте ф-ции, вместо которой вы хотите подставить вызов в другую DLL, моя утилита вписывала такой код:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

При этом, FirstThunk указывает прямо на первую инструкцию. Иными словами, загрузчик, загружая yourdll.dll, прописывает адрес ф-ции *function* прямо в коде.

Надо также отметить что обычно секция кода защищена от записи, так что, моя утилита добавляет флаг *IMAGE_SCN_MEM_WRITE* для секции кода. Иначе при загрузке такой программы, она упадет с ошибкой 5 (access denied).

Может возникнуть вопрос: а что если я поставляю программу с набором DLL, которые никогда не будут меняться, может как-то можно ускорить процесс загрузки?

Да, можно прописать адреса импортируемых ф-ций в массивы FirstThunk заранее. Для этого в структуре *IMAGE_IMPORT_DESCRIPTOR* имеется поле *Timestamp*. И если там присутствует какое-то значение, то загрузчик сверяет это значение с датой-временем DLL-файла. И если они равны, то загрузчик больше ничего не делает, и загрузка будет происходить быстрее. Это называется "old-style binding" [22]. В Windows SDK для этого имеется утилита BIND.EXE. Для ускорения загрузки вашей программы, Matt Pietrek в [24], предлагает делать binding сразу после инсталляции вашей программы на компьютере конечного пользователя.

Запаковщики/зашифровщики PE-файлов могут также сжимать/шифровать таблицу импортов. В этом случае, загрузчик Windows, конечно же, не загрузит все нужные DLL. Поэтому распаковщик/расшифровщик делает это сам, при помощи вызовов *LoadLibrary()* и *GetProcAddress()*.

В стандартных DLL входящих в состав Windows, часто, IAT находится в самом начале PE-файла. Возможно это для оптимизации. Ведь .exe-файл при загрузке не загружается в память весь (вспомните что инсталляторы огромного размера подозрительно быстро запускаются), он "мапится" (map), и подгружается в память частями по мере обращения к этой памяти. И возможно в Microsoft решили что так будет быстрее.

48.2.8 Ресурсы

Ресурсы в PE-файле — это набор иконок, картинок, текстовых строк, описаний диалогов. Возможно, их в свое время решили отделить от основного кода, чтобы все эти вещи были многоязычными, и было проще выбирать текст или картинку того языка, который установлен в ОС.

В качестве побочного эффекта, их легко редактировать и сохранять обратно в исполняемый файл, даже не обладая специальными знаниями, например, редактором ResHack(48.2.11).

[21] http://yurichev.com/PE_add_imports.html
[22] http://blogs.msdn.com/b/oldnewthing/archive/2010/03/18/9980802.aspx. Существует также "new-style binding", про него напишу позже

48.2.9 .NET

Программы на .NET компилируются не в машинный код, а в свой собственный байткод. Собственно, в .exe-файлы байткод вместо обычного кода, однако, точка входа (OEP) указывает на крохотный фрагмент x86-кода:

```
jmp         mscoree.dll!_CorExeMain
```

А в mscoree.dll и находится .NET-загрузчик, который уже сам будет работать с PE-файлом. Так было в ОС до Windows XP. Начиная с XP, загрузчик ОС уже сам определяет, что это .NET-файл и запускает его не исполняя этой инструкции `JMP` [23].

48.2.10 TLS

Эта секция содержит в себе инициализированные данные для TLS(45) (если нужно). При старте нового треда, его TLS-данные инициализируются данными из этой секции.

Помимо всего прочего, спецификация PE-файла предусматривает инициализацию TLS-секции, т.н., TLS callbacks. Если они присутствуют, то они будут вызваны перед тем как передать управление на главную точку входа (OEP). Это широко используется запаковщиками/защифровщиками PE-файлов.

48.2.11 Инструменты

- objdump (из cygwin) для вывода всех структур PE-файла.
- Hiew(53) как редактор.
- pefile — Python-библиотека для работы с PE-файлами [24].
- ResHack AKA Resource Hacker — редактор ресурсов [25].
- PE_add_import[26] — простая утилита для добавления символа/-ов в таблицу импортов PE-файла.
- PE_patcher[27] — простая утилита для модификации PE-файлов.
- PE_search_str_refs[28] — простая утилита для поиска ф-ции в PE-файле, где используется некая текстовая строка.

48.2.12 Further reading

- Daniel Pistelli — The .NET File Format [29]

48.3 Windows SEH

48.3.1 Забудем на время о MSVC

SEH в Windows предназначен для обработки исключений, тем не менее, с Си++ и ООП он никак не связан. Здесь мы рассмотрим SEH изолированно от Си++ и расширений MSVC.

Каждый процесс имеет цепочку SEH-обработчиков, и адрес последнего записан в TIB. Когда происходит исключение (деление на ноль, обращение по неверному адресу в памяти, пользовательское исключение, поднятое при помощи `RaiseException()`), ОС находит последний обработчик в TIB и вызывает его, передав ему информацию о состоянии CPU в момент исключения (все значения регистров, и т.д.). Обработчик выясняет, то ли это исключение, для которого он создавался? Если да, то он обрабатывает исключение. Если нет, то показывает ОС что он не может его обработать и ОС вызывает следующий обработчик в цепочке, и так до тех пор, пока не найдется обработчик способный обработать исключение.

В самом конце цепочки находится стандартный обработчик, показывающий всем очень известное окно, сообщающее что процесс упал, сообщает также состояние CPU в момент падения и позволяет собрать и отправить информацию обработчикам в Microsoft.

[23] http://msdn.microsoft.com/en-us/library/xh0859k0(v=vs.110).aspx
[24] https://code.google.com/p/pefile/
[25] http://www.angusj.com/resourcehacker/
[26] http://yurichev.com/PE_add_imports.html
[27] http://yurichev.com/PE_patcher.html
[28] http://yurichev.com/PE_search_str_refs.html
[29] http://www.codeproject.com/Articles/12585/The-NET-File-Format

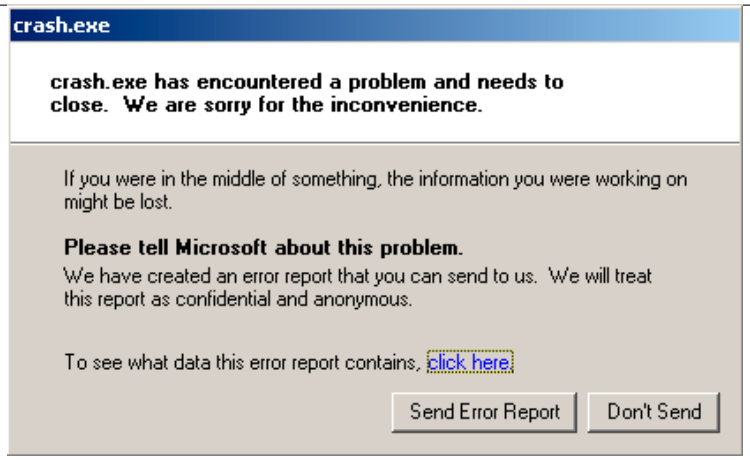


Рис. 48.2: Windows XP

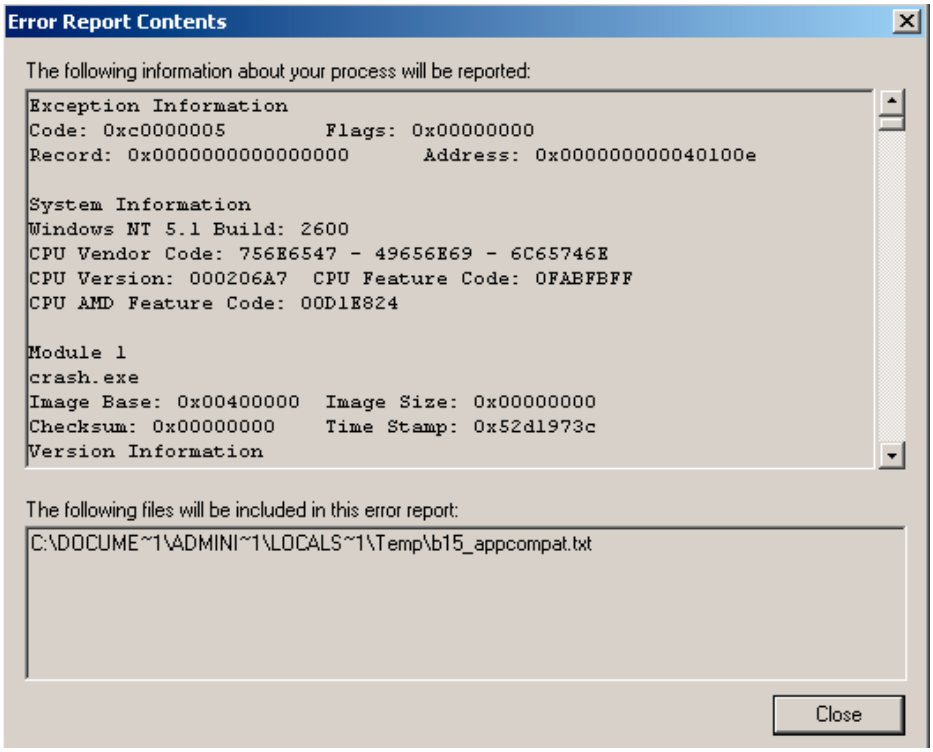


Рис. 48.3: Windows XP

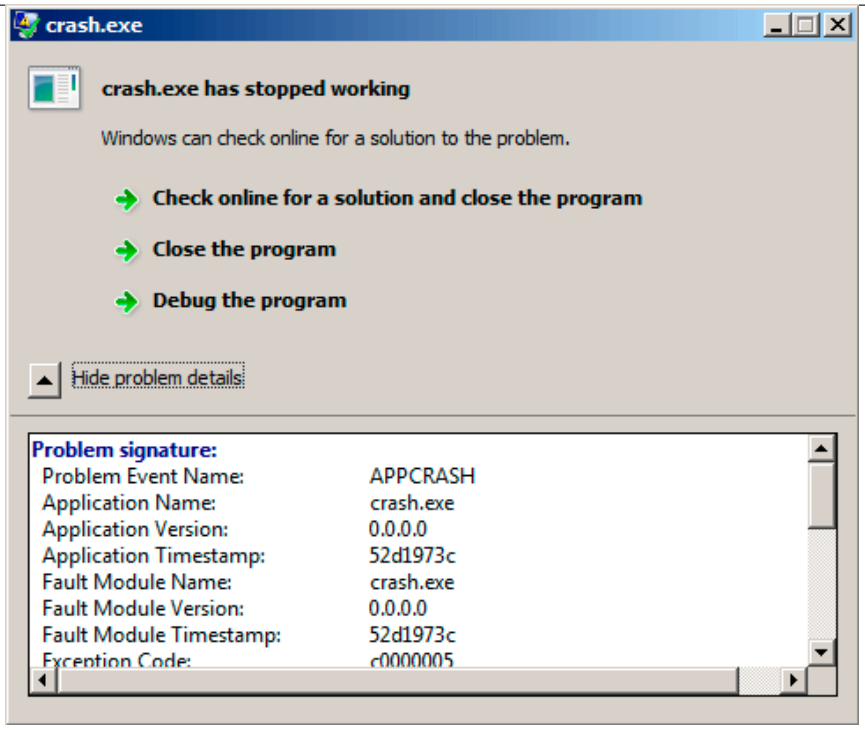


Рис. 48.4: Windows 7

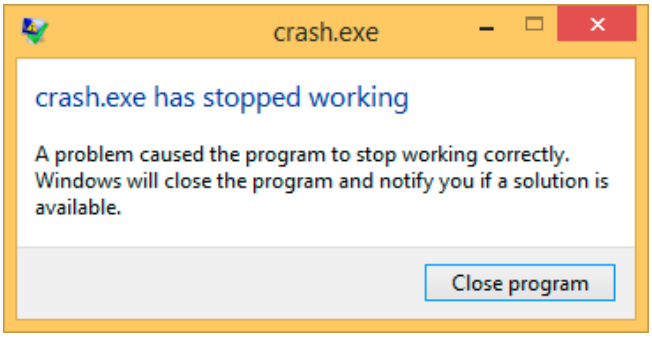


Рис. 48.5: Windows 8.1

Раньше этот обработчик назывался Dr. Watson [30].

Кстати, некоторые разработчики делают свой собственный обработчик, отправляющий информацию о падении программы им самим. Он регистрируется при помощи ф-ции `SetUnhandledExceptionFilter()` и будет вызван если ОС не знает как иначе обработать исключение. А, например, Oracle RDBMS в этом случае генерирует огромные дампы, содержащие всю возможную информацию и состоянии CPU и памяти.

Попробуем написать свой примитивный обработчик исключений [31]:

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
```

[30] https://en.wikipedia.org/wiki/Dr._Watson_(debugger)

[31] Пример основан на примере из [23]

Он должен компилироваться с опцией SAFESEH: `cl seh1.cpp /link /safeseh:no`

Подробнее об опции SAFESEH здесь:
http://msdn.microsoft.com/en-us/library/9a89h429.aspx

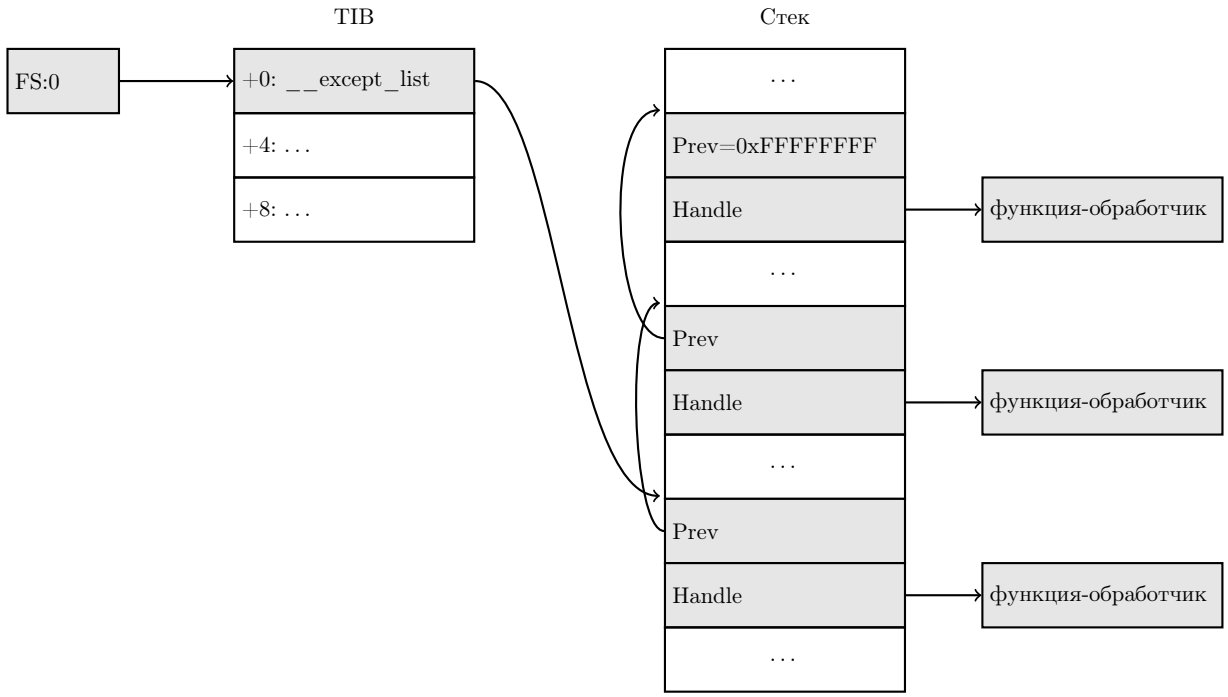
```
{
        unsigned i;

        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
        {
                printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n");
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer to our handler

        // install exception handler
        __asm
        {                               // make EXCEPTION_REGISTRATION record:
                push    handler         // address of handler function
                push    FS:[0]          // address of previous handler
                mov     FS:[0],ESP      // add new EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                               // remove our EXECEPTION_REGISTRATION record
                mov     eax,[ESP]       // get pointer to previous record
                mov     FS:[0], EAX     // install previous record
                add     esp, 8          // clean our EXECEPTION_REGISTRATION off stack
        }

        return 0;
}
```

Сегментный регистр FS: в win32 указывает на TIB. Самый первый элемент TIB это указатель на последний обработчик в цепочке. Мы сохраняем его в стеке и записываем туда адрес своего обработчика. Эта структура называется `_EXCEPTION_REGISTRATION`, это простейший односвязный список, и эти элементы хранятся прямо в стеке.

Листинг 48.1: MSVC/VC/crt/src/exsup.inc

```
\_EXCEPTION\_REGISTRATION struc
    prev    dd      ?
    handler dd      ?
\_EXCEPTION\_REGISTRATION ends
```

Так что каждое поле "handler" указывает на обработчик, а каждое поле "prev" указывает на предыдущую структуру в стеке. Самая последняя структура имеет `0xFFFFFFFF` (-1) в поле "prev".



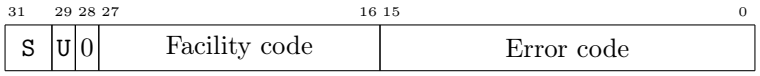
После инсталляции своего обработчика, вызываем `RaiseException()` [32]. Это пользовательские исключения. Обработчик проверяет код. Если код `0xE1223344`, то он возвращает `ExceptionContinueExecution`, что сигнализирует системе что обработчик изменил состояние CPU (обычно это EIP/ESP) и что OS может возобновить исполнение треда. Если вы немного измените код так что обработчик будет возвращать `ExceptionContinueSearch`, то ОС будет вызывать остальные обработчики в цепочке, и врядли найдется тот, кто обработает ваше исключение, ведь информации о нем (вернее, его коде) ни у кого нет. Вы увидите стандартное окно Windows о падении процесса.

Какова разница между системными исключениями и пользовательскими? Вот системные:

[32] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680552(v=vs.85).aspx

как определен в WinBase.h	как определен в ntstatus.h	численное знач
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

Так определяется код:



S это код статуса: 11 — ошибка; 10 — предупреждение; 01 — информация; 00 — успех. U — является ли этот код пользовательским, а не системным.

Вот почему я выбрал 0xE1223344 — 0xE (1110b) означает что это 1) пользовательское исключение; 2) ошибка. Хотя, если быть честным, этот пример нормально работает и без этих старших бит.

Далее мы пытаемся прочитать значение из памяти по адресу 0. Конечно, в win32 по этому адресу обычно ничего нет, и сработает исключение. Однако, первый обработчик, который будет заниматься этим делом — ваш, и он узнает об этом первым, проверяя код на соответствие с константной `EXCEPTION_ACCESS_VIOLATION`.

А если заглянуть в то что получилось на ассемблере, то можно увидеть что код читающий из памяти по адресу 0, выглядит так:

Листинг 48.2: MSVC 2010

```
...
xor     eax, eax
mov     eax, DWORD PTR [eax] ; exception will occur here
push    eax
push    OFFSET msg
call    _printf
add     esp, 8
...
```

Возможно ли “на лету” исправить ошибку и предложить программе исполняться далее? Да, наш обработчик может изменить значение в `EAX` и предложить ОС исполнить эту же инструкцию еще раз. Что мы и делаем. `printf()` напечатает 1234, потому что после работы нашего обработчика, `EAX` будет не 0, а будет содержать адрес глобальной переменной `new_value`. Программа будет исполняться далее.

Собственно, вот что происходит: срабатывает защита менеджера памяти в CPU, он останавливает работу треда, отыскивает в ядре Windows обработчик исключений, тот, в свою очередь, начинает вызывать обработчики из цепочки SEH, по одному.

Я компилирую это всё в MSVC 2010, но конечно же, нет никакой гарантии что для указателя будет использован именно регистр `EAX`.

Этот трюк с подменой адреса эффектно выглядит, и я его привожу здесь для наглядной иллюстрации работы SEH. Тем не менее, я затрудняюсь припомнить, применяется ли где-то подобное на практике для исправления ошибок "на лету".

Почему SEH-записи хранятся именно в стеке а не в каком-то другом месте? Вероятно, потому что тогда ОС не будет заботиться об освобождении этой информации, эти записи пропадают как ненужные когда ф-ция заканчивает работу. Но я не уверен на 100% и могу ошибаться. Это чем-то похоже на alloca(): (4.2.4).

48.3.2 Теперь вспомним MSVC

Должно быть, программистам Microsoft были нужны исключения в Си, но не в Си++, так что они добавили нестандартное расширение Си в MSVC[33]. Оно не связано с исключениями в Си++.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

Блок "finally" может присутствовать вместо код обработчика:

```
__try
{
    ...
}
__finally
{
    ...
}
```

Код-фильтр это выражение, отвечающее на вопрос, соответствует ли код этого обработчика к поднятому исключению. Если ваш код слишком большой и не помещается в одно выражение, отдельная функция-фильтр может быть определена.

Таких конструкций много в ядре Windows. Вот несколько примеров оттуда (WRK):

Листинг 48.3: WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Листинг 48.4: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
```

[33] http://msdn.microsoft.com/en-us/library/swezty51.aspx

```
                  MorePages ?
                    (PAGE_SIZE - PageOffset) :
                    (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                          &Status ) ) {
```

Вот пример кода-фильтра:

Листинг 48.5: WRK-v1.2/base/ntos/cache/copysup.c

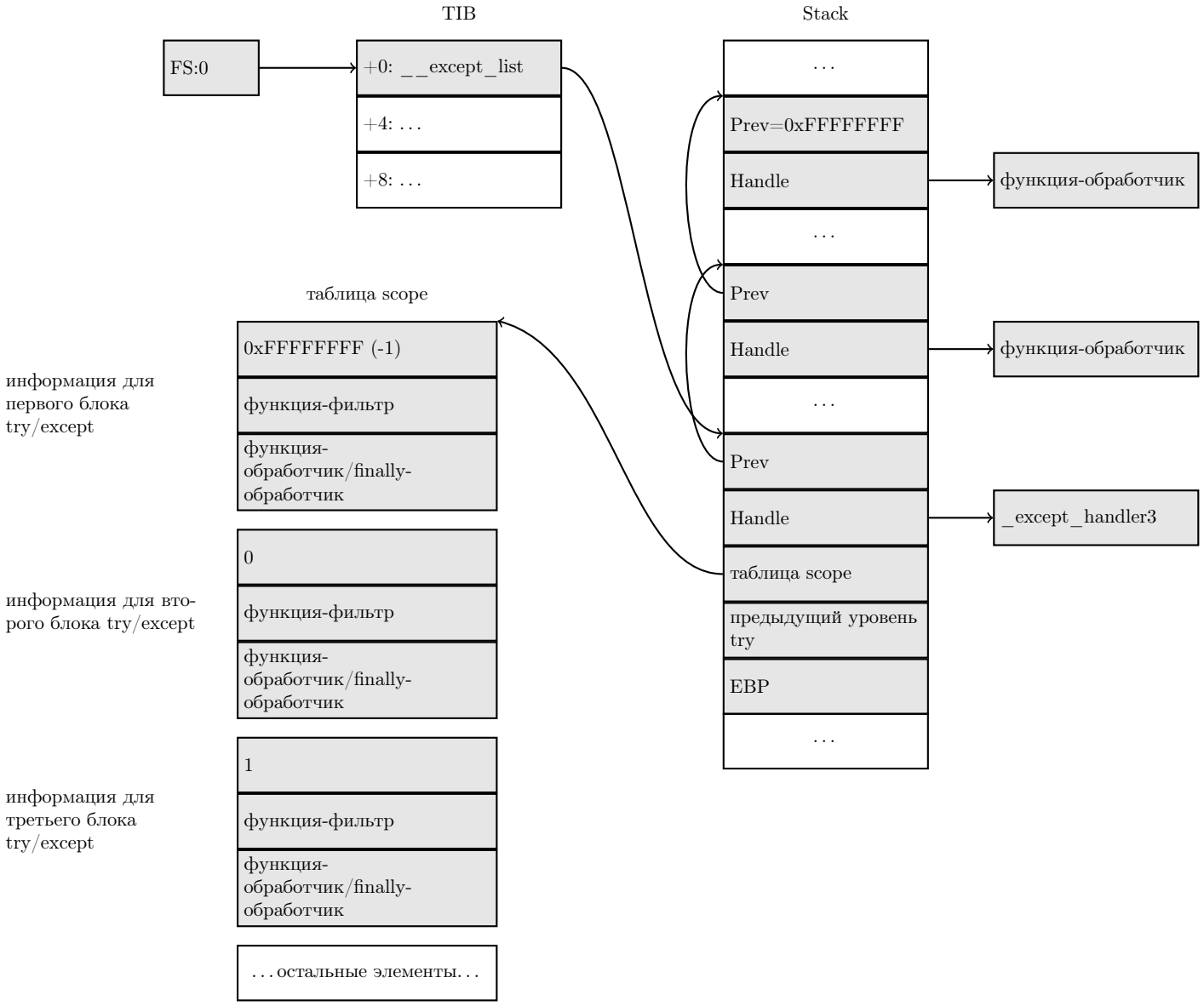
```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )

/*++

Routine Description:

    This routine serves as a exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
                    status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

Внутри, SEH это расширение исключений поддерживаемых OS. Но функция обработчик теперь или `_except_handler3` (для SEH3) или `_except_handler4` (для SEH4). Код обработчика от MSVC, расположен в его библиотеках, или же в msvcr*.dll. Очень важно понимать что SEH это специфичное для MSVC. Другие компиляторы могут предлагать что-то совершенно другое.

SEH3

SEH3 имеет `_except_handler3` как функцию-обработчик, и расширяет структуру `_EXCEPTION_REGISTRATION` добавляя указатель на *scope table* и переменную *previous try level*. SEH4 расширяет *scope table* добавляя еще 4 значения связанных с защитой от переполнения буфера.

Scope table это таблица, состоящая из указателей на код фильтра и обработчика, для каждого уровня вложенности *try/except*.



И снова, очень важно понимать, что OS заботится только о полях *prev/handle* , и больше ничего. Это работа функции `_except_handler3` читать другие поля, читать *scope table* и решать, какой обработчик исполнять и когда.

Исходный код ф-ции `_except_handler3` закрыт. Хотя, Sanos OS, имеющая слой совместимости с win32, имеет некоторые ф-ции написанные заново, которые в каком-то смысле эквивалентны тем что в Windows [34]. Другие попытки реализации имеются в Wine[35] и ReactOS[36].

Если указатель *filter* ноль, *handler* указывает на код *finally*.

Во время исполнения, значение *previous try level* в стеке меняется, чтобы ф-ция `_except_handler3` знала о текущем уровне вложенности, чтобы знать, какой элемент таблицы *scope table* использовать.

SEH3: пример с одним блоком try/except

[34] https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c
[35] https://github.com/mirrors/wine/blob/master/dlls/msvcrt/except_i386.c
[36] http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Листинг 48.6: MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table

CONST     SEGMENT
$T74622   DD    0ffffffffH    ; previous try level
          DD    FLAT:$L74617  ; filter
          DD    FLAT:$L74618  ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1                              ; previous try level
    push   OFFSET FLAT:$T74622             ; scope table
    push   OFFSET FLAT:__except_handler3   ; handler
    mov    eax, DWORD PTR fs:__except_list
    push   eax                             ; prev
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -16
    push   ebx   ; saved 3 registers
    push   esi   ; saved 3 registers
    push   edi   ; saved 3 registers
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0  ; previous try level
    push   OFFSET FLAT:$SG74605 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET FLAT:$SG74606 ; 'hello #2!'
```

```
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try level
    jmp    SHORT $L74616

    ; filter code

$L74617:
$L74627:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74619:
$L74626:
    ret    0

    ; handler code

$L74618:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back to -1
$L74616:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Здесь мы видим как структура SEH конструируется в стеке. *Scope table* расположена в сегменте `CONST` — действительно, эти поля не будут меняться. Интересно, как меняется переменная *previous try level*. Исходное значение `0xFFFFFFFF` (−1). Момент, когда тело `try` открывается, обозначен инструкцией, записывающей 0 в эту переменную. В момент, когда тело `try` закрывается, −1 возвращается в нее назад. Мы также видим адреса кода фильтра и обработчика. Так мы можем легко увидеть структуру конструкций *try/except* в ф-ции.

Так как код инициализации SEH-структур в прологе ф-ций может быть общим для нескольких ф-ций, иногда компилятор вставляет в прологе вызов ф-ции `SEH_prolog()`, которая всё это делает. А код для деинициализации SEH в ф-ции `SEH_epilog()`.

Запустим этот пример в tracer:

```
tracer.exe -l:2.exe --dump-seh
```

Листинг 48.7: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
```

```
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088 ↙
    ↳ (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d) ↙
    ↳ handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16↙
    ↳ )
```

Мы видим что цепочка SEH состоит из 4-х обработчиков.

Первые два расположены в нашем примере. Два? Но ведь мы же сделали только один? Да, второй был установлен в CRT-функции `_mainCRTStartup()`, и судя по всему, он обрабатывает как минимум исключения связанные с FPU. Его код можно посмотреть в инсталляции MSVC: `crt/src/winxfltr.c`.

Третий это SEH4 в ntdll.dll, и четвертый это обработчик, не имеющий отношения к MSVC, расположенный в ntdll.dll, имеющий "говорящее" название ф-ции.

Как видно, в цепочке присутствуют обработчики трех типов: один не связан с MSVC вообще (последний) и два связанных с MSVC: SEH3 и SEH4.

SEH3: пример с двумя блоками try/except

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
```

```
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
                EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Теперь здесь два блока `try`. Так что *scope table* теперь содержит два элемента, один элемент на каждый блок. *Previous try level* меняется вместе с тем, как исполнение доходит до очередного `try`-блока, либо выходит из него.

Листинг 48.8: MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB    'access violation, can''t recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8   ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push   OFFSET FLAT:$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table

CONST    SEGMENT
```

```
$T74644   DD    0ffffffffH   ; previous try level for outer block
          DD    FLAT:$L74634 ; outer block filter
          DD    FLAT:$L74635 ; outer block handler
          DD    00H          ; previous try level for inner block
          DD    FLAT:$L74638 ; inner block filter
          DD    FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT:$T74644
    push   OFFSET FLAT:__except_handler3
    mov    eax, DWORD PTR fs:__except_list
    push   eax
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to↙
    ↳ 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to↙
    ↳ 1
    push   OFFSET FLAT:$SG74617 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867    ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ↙
    ↳ back to 0
    jmp    SHORT $L74615

    ; inner block filter

$L74638:
$L74650:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74643[ebp], eax
    mov    eax, DWORD PTR $T74643[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
```

```
$L74640:
$L74648:
    ret    0

    ; inner block handler

$L74639:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74621 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set previous try level ↙
    ↳ back to 0

$L74615:
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set previous try level ↙
    ↳ back to -1
    jmp    SHORT $L74633

    ; outer block filter

$L74634:
$L74651:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74642[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T74642[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$L74636:
$L74649:
    ret    0

    ; outer block handler

$L74635:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74623 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set previous try level ↙
    ↳ back to -1
$L74633:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Если установить брякпойнт на ф-цию `printf()` вызываемую из обработчика, мы можем увидеть что добавился еще один SEH-обработчик. Наверное, это еще какая-то дополнительная механика скрытая внутри процесса обработки исключений. Тут мы также видим *scope table* состоящую из двух элементов.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

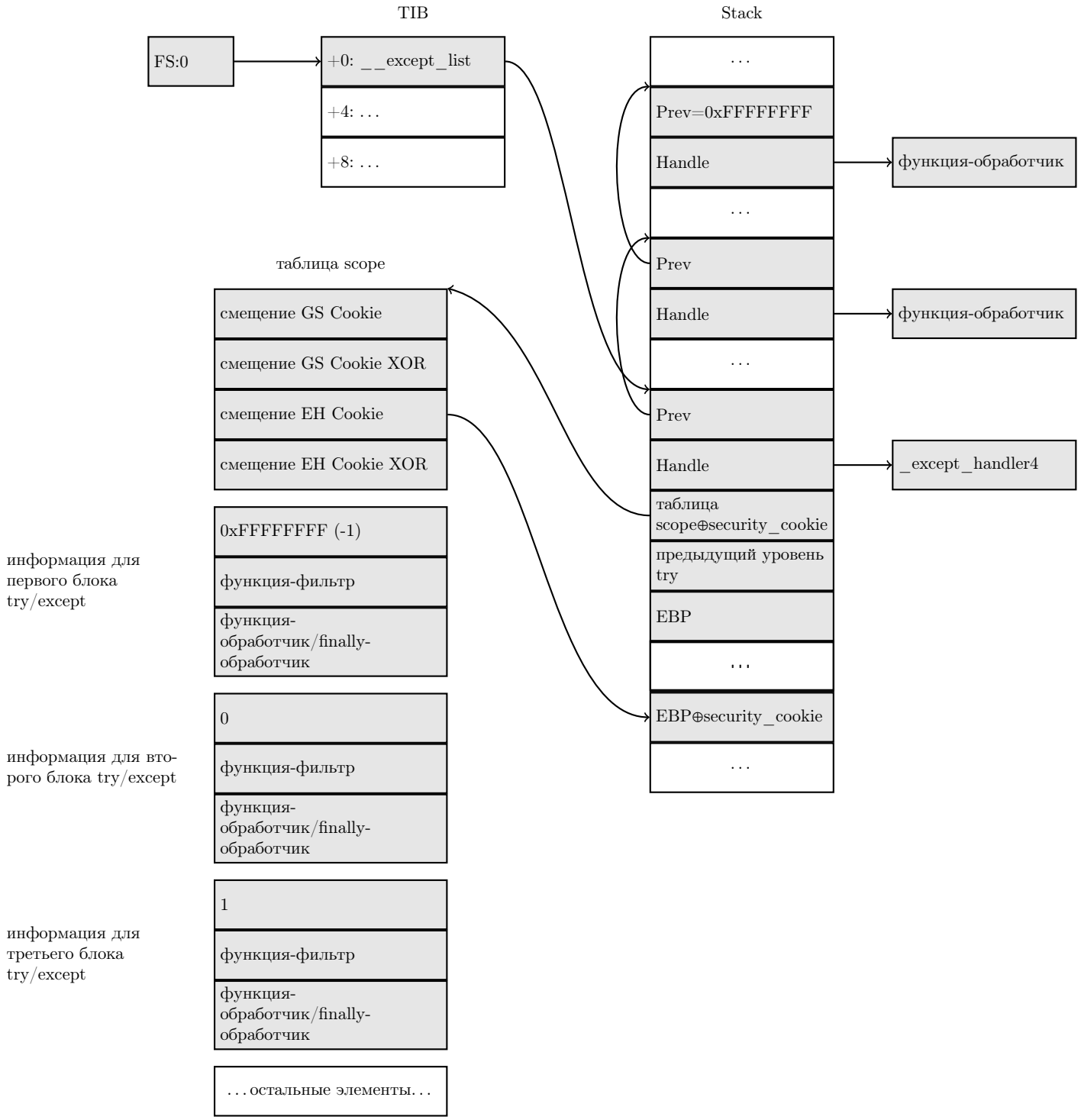
Листинг 48.9: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b ↙
    ↳ (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100 ↙
    ↳ (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) ↙
    ↳ handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16↙
    ↳ )
```

SEH4

Во время атаки переполнения буфера (16.2) адрес *scope table* может быть перезаписан, так что начиная с MSVC 2005, SEH3 был дополнен защитой от переполнения буфера, до SEH4. Указатель на *scope table* теперь про-XOR-ен с security cookie. *Scope table* расширена, теперь имеет заголовок содержащий 2 указателя на *security cookies*. Каждый элемент имеет смешение внутри стека на другое значение: это адрес фрейма (EBP) также про-XOR-еный с `security_cookie` расположенный в стеке. Это значение будет прочитано во время обработки исключения и проверено на правильность. *Security cookie* в стеке случайное каждый раз, так что атакующий, как мы надеемся, не мог предсказать его.

Изначальное значение *previous try level* это −2 в SEH4 вместо −1.



Оба примера скомпилированные в MSVC 2012 с SEH4:

Листинг 48.10: MSVC 2012: one try block example

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table

xdata$x          SEGMENT
__sehtable$_main DD 0ffffffeH   ; GS Cookie Offset
    DD           00H            ; GS Cookie XOR Offset
    DD           0ffffffccH     ; EH Cookie Offset
    DD           00H            ; EH Cookie XOR Offset
```

```
    DD          0fffffffeH      ; previous try level
    DD          FLAT:$LN12@main ; filter
    DD          FLAT:$LN8@main  ; handler
xdata$x         ENDS

$T2 = -36       ; size = 4
_p$ = -32       ; size = 4
tv68 = -28      ; size = 4
__$SEHRec$ = -24 ; size = 24
_main   PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor    eax, ebp
    push   eax                               ; ebp ^ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

    ; filter

$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0
```

```
    ; handler

$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Листинг 48.11: MSVC 2012: two try blocks example

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0fffffffeH         ; GS Cookie Offset
                 DD    00H             ; GS Cookie XOR Offset
                 DD    0ffffffc8H      ; EH Cookie Offset
                 DD    00H             ; EH Cookie Offset
                 DD    0fffffffeH      ; previous try level for outer block
                 DD    FLAT:$LN19@main ; outer block filter
                 DD    FLAT:$LN9@main  ; outer block handler
                 DD    00H             ; previous try level for inner block
                 DD    FLAT:$LN18@main ; inner block filter
                 DD    FLAT:$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
```

```
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored pointer to scope table
    xor    eax, ebp                                ; ebp ^ security_cookie
    push   eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]        ; pointer to ↙
    ↳ VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try ↙
    ↳ level=0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try ↙
    ↳ level=1
    push   OFFSET $SG85497 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level ↙
    ↳ back to 0
    jmp    SHORT $LN2@main

    ; inner block filter

$LN12@main:
$LN18@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T3[ebp], eax
    cmp    DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN5@main
    mov    DWORD PTR tv72[ebp], 1
    jmp    SHORT $LN6@main
$LN5@main:
    mov    DWORD PTR tv72[ebp], 0
$LN6@main:
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret    0

    ; inner block handler

$LN13@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try ↙
    ↳ level back to 0
$LN2@main:
```

```
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ↙
    ↘ back to -2
    jmp    SHORT $LN7@main

    ; outer block filter

$LN8@main:
$LN19@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret    0

    ; outer block handler

$LN9@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level ↙
    ↘ back to -2
$LN7@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_code$ = 8  ; size = 4
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push   OFFSET $SG85488 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
```

```
    jmp    SHORT $LN3@filter_use
$LN2@filter_use:
    push   OFFSET $SG85490 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP
```

Вот значение *cookies*: `Cookie Offset` это разница между адресом записанного в стеке значения EBP и значения $EBP \oplus security\_cookie$ в стеке. `Cookie XOR Offset` это дополнительная разница между значением $EBP \oplus security\_cookie$ и тем что записано в стеке. Если это уравнение не верно, то процесс остановится из-за разрушения стека:

$$security\_cookie \oplus (CookieXOROffset + addressofsavedEBP) == stack[addressofsavedEBP + CookieOffset]$$

Если `Cookie Offset` равно −2, это значит что оно не присутствует.

Проверка *cookies* также реализована в моем tracer, смотрите https://github.com/dennis714/tracer/blob/master/SEH.c для деталей.

Возможность переключиться назад на SEH3 все еще присутствует в компиляторах после (и включая) MSVC 2005, нужно включить опцию `/GS-`, впрочем, CRT-код будет продолжать использовать SEH4.

48.3.3 Windows x64

Как видно, это не самая быстрая штука, устанавливать SEH-структуры в каждом прологе функции. Еще одна проблема производительности это менять переменную *previous try level* много раз в течении исполнении ф-ции. Так что в x64 всё сильно изменилось, теперь все указатели на `try`-блоки, ф-ции фильтров и обработчиков, теперь записаны в другом PE-сегменте `.pdata`, откуда обработчик исключений OS берет всю информацию.

Вот два примера из предыдущей секции, скомпилированных для x64:

Листинг 48.12: MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
```

```
main    PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t recover'
        call    printf
        npad    1
$LN8@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
_TEXT   ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x  ENDS
```

Листинг 48.13: MSVC 2012

```
$SG86277 DB     'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB     'yes, that is our exception', 0aH, 00H
$SG86281 DB     'not our exception', 0aH, 00H
$SG86288 DB     'hello!', 0aH, 00H
$SG86290 DB     '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB     'access violation, can''t recover', 0aH, 00H
$SG86294 DB     'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD  imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
```

```
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
$unwind$filter_user_exceptions DD 020601H
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main:
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t recover'
        call    printf
        npad    1
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call    printf
        npad    1
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
```

```
main    ENDP

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$:
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1
$LN8@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$1 ENDP
text$x  ENDS

_TEXT   SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
        call    printf
        mov     eax, 1
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use:
        lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
```

```
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT   ENDS
```

Смотрите [32] для более детального описания.

Помимо информации об исключениях, секция `.pdata` также содержит начала и концы почти всех ф-ций, так что эту информацию можно использовать в каких-либо утилитах предназначенных для автоматизации анализа.

48.3.4 Больше о SEH

[23], [32].

48.4 Windows NT: Критические секции

Критические секции в любой ОС очень важны в мультитредовой среде, используются в основном для обеспечения гарантии что только один тред будет иметь доступ к данным, блокируя остальные треды и прерывания.

Вот как объявлена структура `CRITICAL_SECTION` объявлена в линейке OS Windows NT:

Листинг 48.14: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    //  The following three fields control entering and exiting the critical
    //  section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Вот как работает ф-ция EnterCriticalSection():

Листинг 48.15: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD:
                mov     eax, large fs:18h
```

```
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
                pop     ebp
                retn    4

... skipped
```

Самая важная инструкция в этом фрагменте кода это BTR (с префиксом LOCK): нулевой бит сохраняется в флаге CF и очищается в памяти. Это атомарная операция, блокирующая доступ всех остальных процессоров к этому значению в памяти (обратите внимание на префикс LOCK перед инструкцией BTR. Если бит в LockCount был 1, хорошо, сбросить его и вернуться из ф-ции: мы в критической секции. Если нет — критическая секция уже занята другим тредом, тогда ждем.
Ожидание там сделано через вызов WaitForSingleObject().

А вот как работает ф-ция LeaveCriticalSection():

Листинг 48.16: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
                lea     edi, [esi+4]    ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... skipped
```

XADD это “обменять и прибавить”. В данном случае, это значит прибавить 1 к значению в LockCount, сохранить результат в регистре EBX, и в то же время 1 записывается в LockCount. Эта операция также атомарная, потому что также имеет префикс LOCK, что означает что другие CPU или ядра CPU в системе не будут иметь доступа к этой ячейке памяти.

Префикс LOCK очень важен: два треда, каждый из которых работает на разных CPU или ядрах CPU, могут попытаться одновременно войти в критическую секцию, одновременно модифицируя значение в памяти, и это

может привести к непредсказуемым результатам.

Часть V

Инструменты

Глава 49

Дизассемблер

49.1 IDA

Старая бесплатная версия доступна для скачивания [1].

Краткий справочник хот-кеев: F.1

[1] http://www.hex-rays.com/idapro/idadownfreeware.htm

Глава 50

Отладчик

50.1 tracer

Я использую *tracer*[1] вместо отладчика.

Со временем я отказался использовать отладчик, потому что все что мне нужно от него: это иногда подсмотреть какие-либо аргументы какой-либо функции во время исполнения или состояние регистров в определенном месте. Каждый раз загружать отладчик для этого это слишком, поэтому я написал очень простую утилиту *tracer*. Она консольная, запускается из командной строки, позволяет перехватывать исполнение функций, ставить брякпойнты на произвольные места, смотреть состояние регистров, модифицировать их, и так далее.

Но для учебы, очень полезно трассировать код руками в отладчике, наблюдать как меняются значения регистров (например, как минимум классический SoftICE, OllyDbg, WinDbg подсвечивают измененные регистры), флагов, данные, менять их самому, смотреть реакцию, и т.д.

50.2 OllyDbg

Очень популярный отладчик пользовательской среды win32:
http://www.ollydbg.de/.

Краткий справочник хот-кеев:

50.3 GDB

Не очень популярный отладчик у реверсеров, тем не менее, крайне удобный. Некоторые команды: F.5.

[1] http://yurichev.com/tracer-ru.html

Глава 51

Трассировка системных вызовов

51.0.1 strace / dtruss

Позволяет показать, какие системные вызовы (syscalls(46)) прямо сейчас вызывает процесс. Например:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

В Mac OS X для этого же имеется dtruss.

В Cygwin также есть strace, впрочем, если я верно понял, он показывает результаты только для .exe-файлов скомпилированных для среды самого cygwin.

Глава 52

Декомпиляторы

Пока существует только один, публично доступный, декомпилятор в Си высокого качества: Hex-Rays: https://www.hex-rays.com/products/decompiler/

Глава 53

Прочие инструменты

- Microsoft Visual Studio Express[1]: Усеченная бесплатная версия Visual Studio, пригодная для простых экспериментов. Some useful options: F.3.
- Hiew[2] для мелкой модификации кода в исполняемых файлах.
- binary grep: небольшая утилита для поиска констант (либо просто последовательности байт) в большом кол-ве файлов, включая неисполняемые: https://github.com/yurichev/bgrep.

[1] http://www.microsoft.com/express/Downloads/
[2] http://www.hiew.ru/

Часть VI

Еще примеры

Глава 54

Ручная декомпиляция + использование SMT-солвера Z3 для взлома любительской криптографии

Любительская криптография обычно (непреднамеренно) очень слабая и может быть легко сломана — для криптографов, конечно.

Но представим на время что мы не в числе этих профессионалов.

Я нашел эту необратимую хэш-ф-цию, которая конвертирует одно 64-битное значение в другое, и нам нужно попытаться развернуть её работу назад.

> Но что такое хеш-функция? Простейший пример это CRC32, алгоритм "более мощный" чем простая контрольная сумма, для проверки целостности данных. Невозможно восстановить оригинальный текст из хеша, там просто меньше информации: ведь текст может быть очень длинным, но результат CRC32 всегда ограничен 32 битами. Но CRC32 не надежна в криптографическом смысле: известны методы как изменить текст таким образом, чтобы получить нужный результат. Криптографические хеш-функции защищены от этого. Такие ф-ции как MD5, SHA1, и т.д, широко используются для хеширования паролей для хранения их в базе. Действительно: БД форума в интернете может и не хранить пароли (иначе злоумышленник получивший доступ к БД сможет узнать все пароли), а только хеши. К тому же, скрипту интернет-форума вовсе не обязательно знать ваш пароль, он только должен сверить его хеш с тем что лежит в БД, и дать вам доступ если сверка проходит. Один из самых простых способов взлома это просто перебирать все пароли и ждать пока результат будет такой же как тот что нам нужен. Другие методы намного сложнее.

54.1 Ручная декомпиляция

Вот листинг на ассемблере в IDA:

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
```

```
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

Пример был скомпилирован в GCC, так что первый аргумент передается в ECX.

Если вы не имеете Hex-Rays, либо вы не доверяете его результатам, мы можем попробовать переписать всё это на Си вручную. Один из методов, это представить регистры CPU в виде локальных переменных Си и заменить каждую инструкцию эквивалентным выражением, например:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Если быть очень аккуратным, этот код можно скомпилировать и он даже будет работать, точно так же как оригинальный.

Затем, будем переписывать его постепенно, не забывая об использовании регистров. Внимание и фокусирование здесь крайне важно — любая самая мелкая опечатка может испортить всю работу!

Первый шаг:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
```

```
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Следующий шаг:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Мы находим деление через умножение (14). Действительно, найдем делитель в Wolfram Mathematica:

Листинг 54.1: Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

Получаем:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;
```

```
        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Еще один шаг:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Простым сокращением, мы видим, что вычислялось вовсе не частное, а остаток от деления:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

Заканчиваем на приятно отформатированном исходном коде:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>
```

```
#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

Так как мы не криптоаналитики, мы не можем найти простой способ найти входное значение для определенного выходного значения. Коэффициенты инструкций сдвигов выглядят очень пугающе — это гарантия что ф-ция не биективная, она имеет коллизии, или, говоря проще, возможны несколько значений на входе для одного на выходе.

Брут-форс это тоже не решение, т.к., значения 64-битные, и это совершенно нереально.

54.2 Попробуем Z3 SMT-солвер

Но все же, без всяких специальных знаний из криптографии, мы можем попытаться взломать алгоритм при помощи великолепного SMT-солвера от Microsoft Research под названием Z3[1]. На самом деле, это автоматический доказыватель теорем, но мы будем использовать его как SMT-солвер. Упрощенно говоря, мы можем думать о нем как о системе, способной решать очень большие системы уравнений.

Вот исходный код на Питоне:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

Это будет наш первый солвер.

[1] http://z3.codeplex.com/

На строке 7 мы видим объявление переменных. Это просто 64-битные переменные. `i1..i6` это промежуточные переменные, отражающие значения в регистрах между исполнениями инструкций.

Потом добавляем т.н. констрайнты, в строках 10..15. Самый последний констрайнт в строке 17 это наиболее важный: мы будем искать входное значение для нашего алгоритма, при котором он выдаст на выходе 10816636949158156260.

Собственно, SMT-солвер ищет (любые) значения, удовлетворяющие всем констрайнтам.

RotateRight, RotateLeft, URem — это ф-ции из Питоновского Z3 API для описания выражений, они не связаны с ЯП Python.

Запускаем:

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

“sat” означает “satisfiable”, т.е., солвер нашел по крайней мере одно решение. Решение выведено внутри квадратных скобок. Две последние строки это пара входного/выходного значения в шестнадцатеричном виде. Да, действительно, если мы запустим нашу ф-цию с `0x12EE577B63E80B73` на входе, алгоритм выдаст искомое значение.

Но, как мы заметили раннее, ф-ция не биективная, так что тут могут быть и другие корректные входные значения. Z3 SMT-солвер не выдает результаты больше одного, но мы можем хакнуть наш пример немного, добавив констрайнт в строке 19, означая, что мы ищем какие угодно другие результаты кроме этого:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 s.add(inp!=0x12EE577B63E80B73)
20
21 print s.check()
22 m=s.model()
23 print m
24 print (" inp=0x%X" % m[inp].as_long())
25 print ("outp=0x%X" % m[outp].as_long())
```

Действительно, получаем еще один верный результат:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
```

```
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

Это можно автоматизировать. Каждый найденный результат можно добавлять в качестве констрайнта и искать следующий. Пример немного сложнее:

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

# copypasted from http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation
result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not suppported")
            # create a constant from declaration
            c=d()
            if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
                raise Z3Exception("arrays and uninterpreted sorts are not supported")
            block.append(c != m[d])
        s.add(Or(block))
    else:
            print "results total=",len(result)
            break
```

Получаем:

```
1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
```

```
1906652839273745429
1113002487612852123 7
1574171089455590914 1
6518338857701133333
5975809943035972467
1519918197989074827 5
1058749596146336037 1
results total= 16
```

Так что имеется 16 верных входных значений для `0x92EE577B63E80B73` на выходе.

Второй это 1234567890 — действительно, я это и использовал изначально, когда готовил этот пример.

Попробуем изучить алгоритм немного больше. В порыве садистских желаний, попробуем найти, есть ли здесь какая-нибудь возможная пара входов/выходов, в которых младшие 32-битные части равны друг другу?

Уберем констрайнт *outp* и добавим другой, в строке 17:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

И действительно:

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Можем упражняться в садизме и далее: пусть последние 16-бит всегда будут `0x1234`:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
8
9  s = Solver()
```

```
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18 s.add(outp & 0xFFFF == 0x1234)
19
20 print s.check()
21 m=s.model()
22 print m
23 print (" inp=0x%X" % m[inp].as_long())
24 print ("outp=0x%X" % m[outp].as_long())
```

Это так же возможно:

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 работает крайне быстро и это означает что алгоритм слаб, и вообще не относится к криптографическим (как и почти вся любительская криптография).

Можно ли попытаться сделать что-то подобное с настоящими криптоалгоритмами этими методами? Настоящие алгоритмы, такие как AES, RSA, итд, так же могут быть представлены в виде огромных систем уравнений, но они большие настолько, что с ними нельзя работать на компьютерах, ни сейчас, ни в обозримом будущем. Разумеется, криптографы об этом всем прекрасно знают.

Еще одна статья которую я написал о Z3: [36].

Глава 55

Донглы

Иногда я делаю замену донглам или “эмуляторы донглов” и здесь немного примеров, как это происходит [1].

Об одном неописанном здесь случае вы также можете прочитать здесь: [36].

55.1 Пример #1: MacOS Classic и PowerPC

Как-то я получил программу для MacOS Classic [2], для PowerPC. Компания, разработавшая этот продукт давно исчезла, так что (легальный) пользователь боялся того что донгла может сломаться.

Если запустить программу без подключенной донглы, можно увидеть окно с надписью "Invalid Security Device". Мне повезло потому что этот текст можно было легко найти внутри исполняемого файла.

Я не был знаком ни с Mac OS Classic, ни с PowerPC, но решил попробовать.

IDA открывает исполняемый файл легко, показывая его тип как "PEF (Mac OS or Be OS executable)"(действительно, это стандартный тип файлов в Mac OS Classic).

В поисках текстовой строки с сообщение об ошибке, я попал на этот фрагмент кода:

```
...

seg000:000C87FC 38 60 00 01                li      %r3, 1
seg000:000C8800 48 03 93 41                bl      check1
seg000:000C8804 60 00 00 00                nop
seg000:000C8808 54 60 06 3F                clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40                bne     OK
seg000:000C8810 80 62 9F D8                lwz     %r3, TC_aInvalidSecurityDevice

...
```

Да, это код PowerPC. Это очень типичный процессор для RISC 1990-х. Каждая инструкция занимает 4 байта (как и в MIPS и ARM) и их имена немного похожи на имена инструкций MIPS.

`check1()` это имя которое я дал этой ф-ции позже. `BL` это инструкция *Branch Link* т.е., предназначенная для вызова подпрограмм. Самое важное место — это инструкция BNE срабатывающая если проверка наличия донглы прошла успешно, либо не срабатывающая в случае ошибки: и тогда адрес текстовой строки с сообщением об ошибке будет загружен в регистр r3 для последующей передачи в функцию отображения диалогового окна.

Из [33] я узнал, что регистр r3 используется для возврата значений (и еще r4 если значение 64-битное).

Еще одна пока что неизвестная инструкция `CLRLWI`. Из [13] я узнал, что эта инструкция одновременно и очищает и загружает. В нашем случае, она очищает 24 старших бита из значения в r3 и записывает всё это в r0, так что это аналог `MOVZX` в x86 (13.1.2), но также устанавливает флаги, так что BNE может проверить их потом.

Посмотрим внутрь `check1()`:

```
seg000:00101B40             check1: # CODE XREF: seg000:00063E7Cp
seg000:00101B40                     # sub_64070+160p ...
seg000:00101B40
seg000:00101B40             .set arg_8,  8
seg000:00101B40
seg000:00101B40 7C 08 02 A6         mflr    %r0
```

[1] Больше об этом читайте тут: http://yurichev.com/dongles.html

[2] MacOS перед тем как перейти на UNIX

```
seg000:00101B44 90 01 00 08         stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0         stwu    %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39         bl      check2
seg000:00101B50 60 00 00 00         nop
seg000:00101B54 80 01 00 48         lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40         addi    %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6         mtlr    %r0
seg000:00101B60 4E 80 00 20         blr
seg000:00101B60             # End of function check1
```

Как можно увидеть в IDA, эта ф-ция вызывается из многих мест в программе, но только значение в регистре r3 проверяется сразу после каждого вызова. Всё что эта ф-ция делает это только вызывает другую ф-цию, так что это thunk function: здесь присутствует и пролог ф-ции и эпилог, но регистр r3 не трогается, так что `check1()` возвращает то, что возвращает `check2()`.

BLR[3] это похоже возврат из ф-ции, но так как IDA делает всю разметку ф-ций автоматически, наверное, мы можем пока не интересоваться этим. Так как это типичный RISC, похоже, подпрограммы вызываются используя link register, точно как в ARM.

Ф-ция `check2()` более сложная:

```
seg000:00118684             check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684             .set var_18, -0x18
seg000:00118684             .set var_C, -0xC
seg000:00118684             .set var_8, -8
seg000:00118684             .set var_4, -4
seg000:00118684             .set arg_8,  8
seg000:00118684
seg000:00118684 93 E1 FF FC   stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6   mflr    %r0
seg000:0011868C 83 E2 95 A8   lwz     %r31, off_1485E8 # dword_24B704
seg000:00118690               .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8   stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4   stw     %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78   mr      %r29, %r3
seg000:0011869C 90 01 00 08   stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01   cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0   stwu    %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C   bne     loc_1186B8
seg000:001186B0 38 60 00 01   li      %r3, 1
seg000:001186B4 48 00 00 6C   b       exit
seg000:001186B8
seg000:001186B8             loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5   bl      sub_118A8C
seg000:001186BC 60 00 00 00   nop
seg000:001186C0 3B C0 00 00   li      %r30, 0
seg000:001186C4
seg000:001186C4             skip:       # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F   clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18   beq     loc_1186E0
seg000:001186CC 38 61 00 38   addi    %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00   lwz     %r4, dword_24B704
seg000:001186D4 48 00 C0 55   bl      .RBEFINDNEXT
seg000:001186D8 60 00 00 00   nop
seg000:001186DC 48 00 00 1C   b       loc_1186F8
seg000:001186E0
seg000:001186E0             loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00   lwz     %r5, dword_24B704
seg000:001186E4 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2   li      %r3, 0x1234
```

[3](PowerPC) Branch to Link Register

```
seg000:001186EC 48 00 BF 99   bl      .RBEFINDFIRST
seg000:001186F0 60 00 00 00   nop
seg000:001186F4 3B C0 00 01   li      %r30, 1
seg000:001186F8
seg000:001186F8             loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C   beq     must_jump
seg000:00118700 38 60 00 00   li      %r3, 0          # error
seg000:00118704 48 00 00 1C   b       exit
seg000:00118708
seg000:00118708             must_jump:  # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78   mr      %r3, %r29
seg000:0011870C 48 00 00 31   bl      check3
seg000:00118710 60 00 00 00   nop
seg000:00118714 54 60 06 3F   clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC   beq     skip
seg000:0011871C 38 60 00 01   li      %r3, 1
seg000:00118720
seg000:00118720             exit:       # CODE XREF: check2+30j
seg000:00118720                         # check2+80j
seg000:00118720 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC   lwz     %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6   mtlr    %r0
seg000:00118730 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118738 4E 80 00 20   blr
seg000:00118738             # End of function check2
```

Снова повезло: имена некоторых ф-ций оставлены в исполняемом файле (в символах в отладочной секции? Я не уверен, т.к. я не знаком с этим форматом файлов, может быть это что-то вроде PE-экспортов (48.2.7))? как например `.RBEFINDNEXT()` and `.RBEFINDFIRST()`. В итоге, эти ф-ции вызывают другие ф-ции с именами вроде `.GetNextDeviceViaUSB()`, `.USBSendPKT()`, так что они явно работают с каким-то USB-устройством.

Тут даже есть ф-ция с названием `.GetNextEve3Device()` — звучит знакомо, в 1990-х годах была донгла Sentinel Eve3 для ADB-порта (присутствующих на Макинтошах).

В начале посмотрим на то как устанавливается регистр r3 одновременно игнорируя всё остальное. Мы знаем, что "хорошее" значение в r3 должно быть не нулевым, а нулевой r3 приведет к выводу диалогового окна с сообщением об ошибке.

В ф-ции имеются две инструкции `li %r3, 1` и одна `li %r3, 0` (*Load Immediate*, т.е., загрузить значение в регистр). Самая первая инструкция находится на `0x001186B0` — и честно говоря, я не знаю что это означает, нужно больше времени на изучение ассемблера PowerPC.

А вот то что мы видим дальше понять проще: вызывается `.RBEFINDFIRST()` и в случае ошибки, 0 будет записан в r3 и мы перейдем на *exit*, а иначе будет вызвана ф-ция `check3()` — если и она будет выполнена с ошибкой, будет вызвана `.RBEFINDNEXT()` вероятно, для поиска другого USB-устройства.

N.B.: `clrlwi. %r0, %r3, 16` это аналог того что мы уже видели, но она очищает 16 старших бит, т.е., `.RBEFINDFIRST()` вероятно возвращает 16-битное значение.

`B` означает *branch* — безусловный переход.

BEQ это обратная инструкция от BNE.

Посмотрим на `check3()`:

```
seg000:0011873C             check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C             .set var_18, -0x18
seg000:0011873C             .set var_C, -0xC
seg000:0011873C             .set var_8, -8
seg000:0011873C             .set var_4, -4
seg000:0011873C             .set arg_8,  8
seg000:0011873C
seg000:0011873C 93 E1 FF FC   stw     %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6   mflr    %r0
seg000:00118744 38 A0 00 00   li      %r5, 0
seg000:00118748 93 C1 FF F8   stw     %r30, var_8(%sp)
```

```
seg000:0011874C 83 C2 95 A8   lwz     %r30, off_1485E8 # dword_24B704
seg000:00118750               .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4   stw     %r29, var_C(%sp)
seg000:00118754 3B A3 00 00   addi    %r29, %r3, 0
seg000:00118758 38 60 00 00   li      %r3, 0
seg000:0011875C 90 01 00 08   stw     %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0   stwu    %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118768 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D   bl      .RBEREAD
seg000:00118770 60 00 00 00   nop
seg000:00118774 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C   beq     loc_118784
seg000:0011877C 38 60 00 00   li      %r3, 0
seg000:00118780 48 00 02 F0   b       exit
seg000:00118784
seg000:00118784             loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2   cmplwi  %r0, 0x1100
seg000:0011878C 41 82 00 0C   beq     loc_118798
seg000:00118790 38 60 00 00   li      %r3, 0
seg000:00118794 48 00 02 DC   b       exit
seg000:00118798
seg000:00118798             loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011879C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01   li      %r3, 1
seg000:001187A4 38 A0 00 00   li      %r5, 0
seg000:001187A8 48 00 C0 21   bl      .RBEREAD
seg000:001187AC 60 00 00 00   nop
seg000:001187B0 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C   beq     loc_1187C0
seg000:001187B8 38 60 00 00   li      %r3, 0
seg000:001187BC 48 00 02 B4   b       exit
seg000:001187C0
seg000:001187C0             loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B   cmplwi  %r0, 0x09AB
seg000:001187C8 41 82 00 0C   beq     loc_1187D4
seg000:001187CC 38 60 00 00   li      %r3, 0
seg000:001187D0 48 00 02 A0   b       exit
seg000:001187D4
seg000:001187D4             loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9   bl      sub_B7BAC
seg000:001187D8 60 00 00 00   nop
seg000:001187DC 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001187E0 2C 00 00 05   cmpwi   %r0, 5
seg000:001187E4 41 82 01 00   beq     loc_1188E4
seg000:001187E8 40 80 00 10   bge     loc_1187F8
seg000:001187EC 2C 00 00 04   cmpwi   %r0, 4
seg000:001187F0 40 80 00 58   bge     loc_118848
seg000:001187F4 48 00 01 8C   b       loc_118980
seg000:001187F8
seg000:001187F8             loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B   cmpwi   %r0, 0xB
seg000:001187FC 41 82 00 08   beq     loc_118804
seg000:00118800 48 00 01 80   b       loc_118980
seg000:00118804
seg000:00118804             loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118808 38 81 00 38   addi    %r4, %sp, 0x50+var_18
```

```
seg000:0011880C 38 60 00 08   li      %r3, 8
seg000:00118810 38 A0 00 00   li      %r5, 0
seg000:00118814 48 00 BF B5   bl      .RBEREAD
seg000:00118818 60 00 00 00   nop
seg000:0011881C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C   beq     loc_11882C
seg000:00118824 38 60 00 00   li      %r3, 0
seg000:00118828 48 00 02 48   b       exit
seg000:0011882C
seg000:0011882C             loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30   cmplwi  %r0, 0xFEA0
seg000:00118834 41 82 00 0C   beq     loc_118840
seg000:00118838 38 60 00 00   li      %r3, 0
seg000:0011883C 48 00 02 34   b       exit
seg000:00118840
seg000:00118840             loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01   li      %r3, 1
seg000:00118844 48 00 02 2C   b       exit
seg000:00118848
seg000:00118848             loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011884C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A   li      %r3, 0xA
seg000:00118854 38 A0 00 00   li      %r5, 0
seg000:00118858 48 00 BF 71   bl      .RBEREAD
seg000:0011885C 60 00 00 00   nop
seg000:00118860 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C   beq     loc_118870
seg000:00118868 38 60 00 00   li      %r3, 0
seg000:0011886C 48 00 02 04   b       exit
seg000:00118870
seg000:00118870             loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3   cmplwi  %r0, 0xA6E1
seg000:00118878 41 82 00 0C   beq     loc_118884
seg000:0011887C 38 60 00 00   li      %r3, 0
seg000:00118880 48 00 01 F0   b       exit
seg000:00118884
seg000:00118884             loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118888 28 1F 00 02   cmplwi  %r31, 2
seg000:0011888C 40 82 00 0C   bne     loc_118898
seg000:00118890 38 60 00 01   li      %r3, 1
seg000:00118894 48 00 01 DC   b       exit
seg000:00118898
seg000:00118898             loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011889C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B   li      %r3, 0xB
seg000:001188A4 38 A0 00 00   li      %r5, 0
seg000:001188A8 48 00 BF 21   bl      .RBEREAD
seg000:001188AC 60 00 00 00   nop
seg000:001188B0 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001188B4 41 82 00 0C   beq     loc_1188C0
seg000:001188B8 38 60 00 00   li      %r3, 0
seg000:001188BC 48 00 01 B4   b       exit
seg000:001188C0
seg000:001188C0             loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C   cmplwi  %r0, 0x1C20
```

```
seg000:001188C8 41 82 00 0C   beq     loc_1188D4
seg000:001188CC 38 60 00 00   li      %r3, 0
seg000:001188D0 48 00 01 A0   b       exit
seg000:001188D4
seg000:001188D4             loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03   cmplwi  %r31, 3
seg000:001188D8 40 82 01 94   bne     error
seg000:001188DC 38 60 00 01   li      %r3, 1
seg000:001188E0 48 00 01 90   b       exit
seg000:001188E4
seg000:001188E4             loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00   lwz     %r6, dword_24B704
seg000:001188E8 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C   li      %r3, 0xC
seg000:001188F0 38 A0 00 00   li      %r5, 0
seg000:001188F4 48 00 BE D5   bl      .RBEREAD
seg000:001188F8 60 00 00 00   nop
seg000:001188FC 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C   beq     loc_11890C
seg000:00118904 38 60 00 00   li      %r3, 0
seg000:00118908 48 00 01 68   b       exit
seg000:0011890C
seg000:0011890C             loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40   cmplwi  %r0, 0x40FF
seg000:00118914 41 82 00 0C   beq     loc_118920
seg000:00118918 38 60 00 00   li      %r3, 0
seg000:0011891C 48 00 01 54   b       exit
seg000:00118920
seg000:00118920             loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118924 28 1F 00 02   cmplwi  %r31, 2
seg000:00118928 40 82 00 0C   bne     loc_118934
seg000:0011892C 38 60 00 01   li      %r3, 1
seg000:00118930 48 00 01 40   b       exit
seg000:00118934
seg000:00118934             loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118938 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D   li      %r3, 0xD
seg000:00118940 38 A0 00 00   li      %r5, 0
seg000:00118944 48 00 BE 85   bl      .RBEREAD
seg000:00118948 60 00 00 00   nop
seg000:0011894C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C   beq     loc_11895C
seg000:00118954 38 60 00 00   li      %r3, 0
seg000:00118958 48 00 01 18   b       exit
seg000:0011895C
seg000:0011895C             loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF   cmplwi  %r0, 0xFC7
seg000:00118964 41 82 00 0C   beq     loc_118970
seg000:00118968 38 60 00 00   li      %r3, 0
seg000:0011896C 48 00 01 04   b       exit
seg000:00118970
seg000:00118970             loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03   cmplwi  %r31, 3
seg000:00118974 40 82 00 F8   bne     error
seg000:00118978 38 60 00 01   li      %r3, 1
seg000:0011897C 48 00 00 F4   b       exit
seg000:00118980
```

```
seg000:00118980             loc_118980: # CODE XREF: check3+B8j
seg000:00118980                         # check3+C4j
seg000:00118980 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118984 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00   li      %r31, 0
seg000:0011898C 38 60 00 04   li      %r3, 4
seg000:00118990 38 A0 00 00   li      %r5, 0
seg000:00118994 48 00 BE 35   bl      .RBEREAD
seg000:00118998 60 00 00 00   nop
seg000:0011899C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C   beq     loc_1189AC
seg000:001189A4 38 60 00 00   li      %r3, 0
seg000:001189A8 48 00 00 C8   b       exit
seg000:001189AC
seg000:001189AC             loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A   cmplwi  %r0, 0xAED0
seg000:001189B4 40 82 00 0C   bne     loc_1189C0
seg000:001189B8 3B E0 00 01   li      %r31, 1
seg000:001189BC 48 00 00 14   b       loc_1189D0
seg000:001189C0
seg000:001189C0             loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28   cmplwi  %r0, 0x2818
seg000:001189C4 41 82 00 0C   beq     loc_1189D0
seg000:001189C8 38 60 00 00   li      %r3, 0
seg000:001189CC 48 00 00 A4   b       exit
seg000:001189D0
seg000:001189D0             loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0                         # check3+288j
seg000:001189D0 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:001189D4 28 00 00 02   cmplwi  %r0, 2
seg000:001189D8 40 82 00 20   bne     loc_1189F8
seg000:001189DC 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:001189E0 41 82 00 10   beq     good2
seg000:001189E4 48 00 4C 69   bl      sub_11D64C
seg000:001189E8 60 00 00 00   nop
seg000:001189EC 48 00 00 84   b       exit
seg000:001189F0
seg000:001189F0             good2:      # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01   li      %r3, 1
seg000:001189F4 48 00 00 7C   b       exit
seg000:001189F8
seg000:001189F8             loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00   lwz     %r6, dword_24B704
seg000:001189FC 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05   li      %r3, 5
seg000:00118A04 38 A0 00 00   li      %r5, 0
seg000:00118A08 48 00 BD C1   bl      .RBEREAD
seg000:00118A0C 60 00 00 00   nop
seg000:00118A10 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118A14 41 82 00 0C   beq     loc_118A20
seg000:00118A18 38 60 00 00   li      %r3, 0
seg000:00118A1C 48 00 00 54   b       exit
seg000:00118A20
seg000:00118A20             loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3   cmplwi  %r0, 0xD300
seg000:00118A28 40 82 00 0C   bne     loc_118A34
seg000:00118A2C 3B E0 00 01   li      %r31, 1
seg000:00118A30 48 00 00 14   b       good1
seg000:00118A34
```

```
seg000:00118A34             loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB   cmplwi  %r0, 0xEBA1
seg000:00118A38 41 82 00 0C   beq     good1
seg000:00118A3C 38 60 00 00   li      %r3, 0
seg000:00118A40 48 00 00 30   b       exit
seg000:00118A44
seg000:00118A44             good1:      # CODE XREF: check3+2F4j
seg000:00118A44                         # check3+2FCj
seg000:00118A44 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:00118A48 28 00 00 03   cmplwi  %r0, 3
seg000:00118A4C 40 82 00 20   bne     error
seg000:00118A50 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:00118A54 41 82 00 10   beq     good
seg000:00118A58 48 00 4B F5   bl      sub_11D64C
seg000:00118A5C 60 00 00 00   nop
seg000:00118A60 48 00 00 10   b       exit
seg000:00118A64
seg000:00118A64             good:       # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01   li      %r3, 1
seg000:00118A68 48 00 00 08   b       exit
seg000:00118A6C
seg000:00118A6C             error:      # CODE XREF: check3+19Cj
seg000:00118A6C                         # check3+238j ...
seg000:00118A6C 38 60 00 00   li      %r3, 0
seg000:00118A70
seg000:00118A70             exit:       # CODE XREF: check3+44j
seg000:00118A70                         # check3+58j ...
seg000:00118A70 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC   lwz     %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6   mtlr    %r0
seg000:00118A80 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20   blr
seg000:00118A88             # End of function check3
```

Здесь много вызовов `.RBEREAD()`. Эта ф-ция вероятно читает какие-то значения из донглы, которые потом сравниваются здесь при помощи `CMPLWI`.

Мы также видим в регистр r3 записывается перед каждым вызовом `.RBEREAD()` одно из этих значений: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Вероятно адрес в памяти или что-то в этом роде?

Да, действительно, если погуглить имена этих ф-ций, можно легко найти документацию к Sentinel Eve3!

Мне даже, наверное, не нужно изучать остальные инструкции PowerPC: всё что делает эта ф-ция это просто вызывает `.RBEREAD()`, сравнивает его результаты с константами и возвращает 1 если результат сравнения положительный или 0 в другом случае.

Всё ясно: `check1()` должна всегда возвращать 1 или иное ненулевое значение. Но так как я не очень уверен в своих знаниях инструкций PowerPC, я буду осторожен и пропатчу переходы в `check2` на адресах `0x001186FC` и `0x00118718`.

На `0x001186FC` я записал байты 0x48 и 0 таким образом превращая инструкцию BEQ в инструкцию `B` (безусловный переход): Я заметил этот опкод прямо в коде даже без обращения к [13].

На `0x00118718` я записал байт 0x60 и еще 3 нулевых байта, таким образом превращая её в инструкцию NOP: Этот опкод я тоже подсмотрел прямо в коде.

Резюмируя, такие простые модификации можно делать в IDA даже с минимальными знаниями ассемблера.

55.2 Пример #2: SCO OpenServer

Древняя программа для SCO OpenServer от 1997 разработанная давно исчезнувшей компанией.

Специальный драйвер донглы инсталлируется в системе, он содержит такие текстовые строки: “Copyright 1989, Rainbow Technologies, Inc., Irvine, CA” и “Sentinel Integrated Driver Ver. 3.0 ”.

После инсталляции драйвера, в /dev появляются такие устройства:

```
/dev/rbsl8
```

```
/dev/rbsl9
/dev/rbsl10
```

Без подключенной донглы, программа сообщает об ошибке, но сообщение об ошибке не удается найти в исполняемых файлах.

Еще раз спасибо IDA, она легко загружает исполняемые файлы формата COFF использующиеся в SCO OpenServer.

Я попробовал также поискать строку “rbsl” , и действительно, её можно найти в таком фрагменте кода:

```
.text:00022AB8        public SSQC
.text:00022AB8 SSQC   proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr  8
.text:00022AB8
.text:00022AB8        push    ebp
.text:00022AB9        mov     ebp, esp
.text:00022ABB        sub     esp, 44h
.text:00022ABE        push    edi
.text:00022ABF        mov     edi, offset unk_4035D0
.text:00022AC4        push    esi
.text:00022AC5        mov     esi, [ebp+arg_0]
.text:00022AC8        push    ebx
.text:00022AC9        push    esi
.text:00022ACA        call    strlen
.text:00022ACF        add     esp, 4
.text:00022AD2        cmp     eax, 2
.text:00022AD7        jnz     loc_22BA4
.text:00022ADD        inc     esi
.text:00022ADE        mov     al, [esi-1]
.text:00022AE1        movsx   eax, al
.text:00022AE4        cmp     eax, '3'
.text:00022AE9        jz      loc_22B84
.text:00022AEF        cmp     eax, '4'
.text:00022AF4        jz      loc_22B94
.text:00022AFA        cmp     eax, '5'
.text:00022AFF        jnz     short loc_22B6B
.text:00022B01        movsx   ebx, byte ptr [esi]
.text:00022B04        sub     ebx, '0'
.text:00022B07        mov     eax, 7
.text:00022B0C        add     eax, ebx
.text:00022B0E        push    eax
.text:00022B0F        lea     eax, [ebp+var_44]
.text:00022B12        push    offset aDevSlD  ; "/dev/sl%d"
.text:00022B17        push    eax
.text:00022B18        call    nl_sprintf
.text:00022B1D        push    0               ; int
.text:00022B1F        push    offset aDevRbsl8 ; char *
.text:00022B24        call    _access
.text:00022B29        add     esp, 14h
.text:00022B2C        cmp     eax, 0FFFFFFFFh
.text:00022B31        jz      short loc_22B48
.text:00022B33        lea     eax, [ebx+7]
.text:00022B36        push    eax
.text:00022B37        lea     eax, [ebp+var_44]
.text:00022B3A        push    offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F        push    eax
.text:00022B40        call    nl_sprintf
.text:00022B45        add     esp, 0Ch
.text:00022B48
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j
```

```
.text:00022B48        mov     edx, [edi]
.text:00022B4A        test    edx, edx
.text:00022B4C        jle     short loc_22B57
.text:00022B4E        push    edx             ; int
.text:00022B4F        call    _close
.text:00022B54        add     esp, 4
.text:00022B57
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57        push    2               ; int
.text:00022B59        lea     eax, [ebp+var_44]
.text:00022B5C        push    eax             ; char *
.text:00022B5D        call    _open
.text:00022B62        add     esp, 8
.text:00022B65        test    eax, eax
.text:00022B67        mov     [edi], eax
.text:00022B69        jge     short loc_22B78
.text:00022B6B
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B        mov     eax, 0FFFFFFFFh
.text:00022B70        pop     ebx
.text:00022B71        pop     esi
.text:00022B72        pop     edi
.text:00022B73        mov     esp, ebp
.text:00022B75        pop     ebp
.text:00022B76        retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78        pop     ebx
.text:00022B79        pop     esi
.text:00022B7A        pop     edi
.text:00022B7B        xor     eax, eax
.text:00022B7D        mov     esp, ebp
.text:00022B7F        pop     ebp
.text:00022B80        retn
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84        mov     al, [esi]
.text:00022B86        pop     ebx
.text:00022B87        pop     esi
.text:00022B88        pop     edi
.text:00022B89        mov     ds:byte_407224, al
.text:00022B8E        mov     esp, ebp
.text:00022B90        xor     eax, eax
.text:00022B92        pop     ebp
.text:00022B93        retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94        mov     al, [esi]
.text:00022B96        pop     ebx
.text:00022B97        pop     esi
.text:00022B98        pop     edi
.text:00022B99        mov     ds:byte_407225, al
.text:00022B9E        mov     esp, ebp
.text:00022BA0        xor     eax, eax
.text:00022BA2        pop     ebp
.text:00022BA3        retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4        movsx   eax, ds:byte_407225
.text:00022BAB        push    esi
.text:00022BAC        push    eax
```

```
.text:00022BAD        movsx   eax, ds:byte_407224
.text:00022BB4        push    eax
.text:00022BB5        lea     eax, [ebp+var_44]
.text:00022BB8        push    offset a46CCS   ; "46%c%c%s"
.text:00022BBD        push    eax
.text:00022BBE        call    nl_sprintf
.text:00022BC3        lea     eax, [ebp+var_44]
.text:00022BC6        push    eax
.text:00022BC7        call    strlen
.text:00022BCC        add     esp, 18h
.text:00022BCF        cmp     eax, 1Bh
.text:00022BD4        jle     short loc_22BDA
.text:00022BD6        mov     [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA        lea     eax, [ebp+var_44]
.text:00022BDD        push    eax
.text:00022BDE        call    strlen
.text:00022BE3        push    eax             ; unsigned int
.text:00022BE4        lea     eax, [ebp+var_44]
.text:00022BE7        push    eax             ; void *
.text:00022BE8        mov     eax, [edi]
.text:00022BEA        push    eax             ; int
.text:00022BEB        call    _write
.text:00022BF0        add     esp, 10h
.text:00022BF3        pop     ebx
.text:00022BF4        pop     esi
.text:00022BF5        pop     edi
.text:00022BF6        mov     esp, ebp
.text:00022BF8        pop     ebp
.text:00022BF9        retn
.text:00022BFA        db 0Eh dup(90h)
.text:00022BFA SSQC   endp
```

Действительно, должна же как-то программа обмениваться информацией с драйвером.
Единственное место где вызывается ф-ция `SSQC()` это thunk function:

```
.text:0000DBE8        public SSQ
.text:0000DBE8 SSQ    proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                  ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8 arg_0  = dword ptr  8
.text:0000DBE8
.text:0000DBE8        push    ebp
.text:0000DBE9        mov     ebp, esp
.text:0000DBEB        mov     edx, [ebp+arg_0]
.text:0000DBEE        push    edx
.text:0000DBEF        call    SSQC
.text:0000DBF4        add     esp, 4
.text:0000DBF7        mov     esp, ebp
.text:0000DBF9        pop     ebp
.text:0000DBFA        retn
.text:0000DBFB SSQ    endp
```

А вот SSQ() вызывается по крайней мере из двух разных ф-ций.
Одна из них:

```
.data:0040169C _51_52_53       dd offset aPressAnyKeyT_0 ; DATA XREF: init_sys+392r
.data:0040169C                                           ; sys_info+A1r
.data:0040169C                                           ; "PRESS ANY KEY TO CONTINUE: "
.data:004016A0                 dd offset a51             ; "51"
.data:004016A4                 dd offset a52             ; "52"
.data:004016A8                 dd offset a53             ; "53"
```

```
...

.data:004016B8 _3C_or_3E       dd offset a3c           ; DATA XREF: sys_info:loc_D67Br
.data:004016B8                                         ; "3C"
.data:004016BC                 dd offset a3e           ; "3E"

; these names I gave to the labels:
.data:004016C0 answers1        dd 6B05h                ; DATA XREF: sys_info+E7r
.data:004016C4                 dd 3D87h
.data:004016C8 answers2        dd 3Ch                  ; DATA XREF: sys_info+F2r
.data:004016CC                 dd 832h
.data:004016D0 _C_and_B        db 0Ch                  ; DATA XREF: sys_info+BAr
.data:004016D0                                         ; sys_info:OKr
.data:004016D1 byte_4016D1     db 0Bh                  ; DATA XREF: sys_info+FDr
.data:004016D2                 db    0

...

.text:0000D652                 xor     eax, eax
.text:0000D654                 mov     al, ds:ctl_port
.text:0000D659                 mov     ecx, _51_52_53[eax*4]
.text:0000D660                 push    ecx
.text:0000D661                 call    SSQ
.text:0000D666                 add     esp, 4
.text:0000D669                 cmp     eax, 0FFFFFFFFh
.text:0000D66E                 jz      short loc_D6D1
.text:0000D670                 xor     ebx, ebx
.text:0000D672                 mov     al, _C_and_B
.text:0000D677                 test    al, al
.text:0000D679                 jz      short loc_D6C0
.text:0000D67B
.text:0000D67B loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B                 mov     eax, _3C_or_3E[ebx*4]
.text:0000D682                 push    eax
.text:0000D683                 call    SSQ
.text:0000D688                 push    offset a4g      ; "4G"
.text:0000D68D                 call    SSQ
.text:0000D692                 push    offset a0123456789 ; "0123456789"
.text:0000D697                 call    SSQ
.text:0000D69C                 add     esp, 0Ch
.text:0000D69F                 mov     edx, answers1[ebx*4]
.text:0000D6A6                 cmp     eax, edx
.text:0000D6A8                 jz      short OK
.text:0000D6AA                 mov     ecx, answers2[ebx*4]
.text:0000D6B1                 cmp     eax, ecx
.text:0000D6B3                 jz      short OK
.text:0000D6B5                 mov     al, byte_4016D1[ebx]
.text:0000D6BB                 inc     ebx
.text:0000D6BC                 test    al, al
.text:0000D6BE                 jnz     short loc_D67B
.text:0000D6C0
.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0                 inc     ds:ctl_port
.text:0000D6C6                 xor     eax, eax
.text:0000D6C8                 mov     al, ds:ctl_port
.text:0000D6CD                 cmp     eax, edi
.text:0000D6CF                 jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1           ; sys_info+B6j
```

```
.text:0000D6D1                  mov     edx, [ebp+var_8]
.text:0000D6D4                  inc     edx
.text:0000D6D5                  mov     [ebp+var_8], edx
.text:0000D6D8                  cmp     edx, 3
.text:0000D6DB                  jle     loc_D641
.text:0000D6E1
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1           ; sys_info+51j ...
.text:0000D6E1                  pop     ebx
.text:0000D6E2                  pop     edi
.text:0000D6E3                  mov     esp, ebp
.text:0000D6E5                  pop     ebp
.text:0000D6E6                  retn
.text:0000D6E8 OK:       ; CODE XREF: sys_info+F0j
.text:0000D6E8           ; sys_info+FBj
.text:0000D6E8                  mov     al, _C_and_B[ebx]
.text:0000D6EE                  pop     ebx
.text:0000D6EF                  pop     edi
.text:0000D6F0                  mov     ds:ctl_model, al
.text:0000D6F5                  mov     esp, ebp
.text:0000D6F7                  pop     ebp
.text:0000D6F8                  retn
.text:0000D6F8 sys_info         endp
```

"3C" и "3E" — это звучит знакомо: когда-то была донгла Sentinel Pro от Rainbow без памяти, предоставляющая только одну секретную крипто-хеширующую ф-цию.

О том, что такое хэш-функция, было описано здесь: 54.

Но вернемся к нашей программе. Так что программа может только проверить подключена ли донгла или нет. Никакой больше информации в такую донглу без памяти записать нельзя. Двухсимвольные коды — это команды (можно увидеть как они обрабатываются в ф-ции `SSQC()`) а все остальные строки хешируются внутри донглы превращаясь в 16-битное число. Алгоритм был секретный, так что нельзя было написать замену драйверу или сделать электронную копию донглы идеально эмулирующую алгоритм. С другой стороны, всегда можно было перехватить все обращения к ней и найти те константы, с которыми сравнивается результат хеширования. Но надо сказать, вполне возможно создать устойчивую защиту от копирования базирующуюся на секретной хеш-функции: пусть она шифрует все файлы с которыми ваша программа работает.

Но вернемся к нашему коду.

Коды 51/52/53 используются для выбора номера принтеровского LPT-порта. 3x/4x используются для выбора "family" так донглы Sentinel Pro можно отличать друг от друга: ведь более одной донглы может быть подключено к LPT-порту.

Единственная строка, передающаяся в хеш-функцию это "0123456789". Затем результат сравнивается с несколькими правильными значениями. Если результат правилен, 0xC или 0xB будет записано в глобальную переменную `ctl_model`.

Еще одна строка для хеширования: "PRESS ANY KEY TO CONTINUE: но результат не проверяется. Не знаю зачем это, может быть по ошибке. (Это очень странное чувство: находить ошибки в столь древнем ПО.)

Давайте посмотрим, где проверяется значение глобальной переменной `ctl_mode`.

Одно из таких мест:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14   = dword ptr -14h
.text:0000D708 var_10   = byte ptr -10h
.text:0000D708 var_8    = dword ptr -8
.text:0000D708 var_2    = word ptr -2
.text:0000D708
.text:0000D708          push    ebp
.text:0000D709          mov     eax, ds:net_env
.text:0000D70E          mov     ebp, esp
.text:0000D710          sub     esp, 1Ch
.text:0000D713          test    eax, eax
.text:0000D715          jnz     short loc_D734
.text:0000D717          mov     al, ds:ctl_model
.text:0000D71C          test    al, al
```

```
.text:0000D71E          jnz     short loc_D77E
.text:0000D720          mov     [ebp+var_8], offset aIeCvulnvvOkgT_ ; "Ie-cvulnvV\\\bOKG]T_"
.text:0000D727          mov     edx, 7
.text:0000D72C          jmp     loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7           ; prep_sys+33j
.text:0000D7E7          push    edx
.text:0000D7E8          mov     edx, [ebp+var_8]
.text:0000D7EB          push    20h
.text:0000D7ED          push    edx
.text:0000D7EE          push    16h
.text:0000D7F0          call    err_warn
.text:0000D7F5          push    offset station_sem
.text:0000D7FA          call    ClosSem
.text:0000D7FF          call    startup_err
```

Если оно 0, шифрованное сообщение об ошибке будет передано в ф-цию дешифрования, и оно будет показано. Ф-ция дешифровки сообщений об ошибке похоже применяет простой xoring:

```
.text:0000A43C err_warn         proc near               ; CODE XREF: prep_sys+E8p
.text:0000A43C                                          ; prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55           = byte ptr -55h
.text:0000A43C var_54           = byte ptr -54h
.text:0000A43C arg_0            = dword ptr  8
.text:0000A43C arg_4            = dword ptr  0Ch
.text:0000A43C arg_8            = dword ptr  10h
.text:0000A43C arg_C            = dword ptr  14h
.text:0000A43C
.text:0000A43C                  push    ebp
.text:0000A43D                  mov     ebp, esp
.text:0000A43F                  sub     esp, 54h
.text:0000A442                  push    edi
.text:0000A443                  mov     ecx, [ebp+arg_8]
.text:0000A446                  xor     edi, edi
.text:0000A448                  test    ecx, ecx
.text:0000A44A                  push    esi
.text:0000A44B                  jle     short loc_A466
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:                                ; CODE XREF: err_warn+28j
.text:0000A453                  xor     eax, eax
.text:0000A455                  mov     al, [edx+edi]
.text:0000A458                  xor     eax, esi
.text:0000A45A                  add     esi, 3
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
.text:0000A466
.text:0000A466 loc_A466:                                ; CODE XREF: err_warn+Fj
.text:0000A466                  mov     [ebp+edi+var_54], 0
.text:0000A46B                  mov     eax, [ebp+arg_0]
.text:0000A46E                  cmp     eax, 18h
.text:0000A473                  jnz     short loc_A49C
.text:0000A475                  lea     eax, [ebp+var_54]
.text:0000A478                  push    eax
.text:0000A479                  call    status_line
```

```
.text:0000A47E                  add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:                                ; CODE XREF: err_warn+72j
.text:0000A481                  push    50h
.text:0000A483                  push    0
.text:0000A485                  lea     eax, [ebp+var_54]
.text:0000A488                  push    eax
.text:0000A489                  call    memset
.text:0000A48E                  call    pcv_refresh
.text:0000A493                  add     esp, 0Ch
.text:0000A496                  pop     esi
.text:0000A497                  pop     edi
.text:0000A498                  mov     esp, ebp
.text:0000A49A                  pop     ebp
.text:0000A49B                  retn
.text:0000A49C
.text:0000A49C loc_A49C:                                ; CODE XREF: err_warn+37j
.text:0000A49C                  push    0
.text:0000A49E                  lea     eax, [ebp+var_54]
.text:0000A4A1                  mov     edx, [ebp+arg_0]
.text:0000A4A4                  push    edx
.text:0000A4A5                  push    eax
.text:0000A4A6                  call    pcv_lputs
.text:0000A4AB                  add     esp, 0Ch
.text:0000A4AE                  jmp     short loc_A481
.text:0000A4AE err_warn         endp
```

Вот почему не получилось найти сообщение об ошибке в исполняемых файлах, потому что оно было зашифровано, это очень популярная практика.

Еще один вызов хеширующей ф-ции передает строку “offln” и сравнивает результат с константами `0xFE81` и `0x12A9`. Если результат не сходится, происходит работа с какой-то ф-цией `timer()` (может быть для ожидания плохо подключенной донглы и нового запроса?), затем дешифрует еще одно сообщение об ошибке и выводит его.

```
.text:0000DA55 loc_DA55:                                ; CODE XREF: sync_sys+24Cj
.text:0000DA55                  push    offset aOffln   ; "offln"
.text:0000DA5A                  call    SSQ
.text:0000DA5F                  add     esp, 4
.text:0000DA62                  mov     dl, [ebx]
.text:0000DA64                  mov     esi, eax
.text:0000DA66                  cmp     dl, 0Bh
.text:0000DA69                  jnz     short loc_DA83
.text:0000DA6B                  cmp     esi, 0FE81h
.text:0000DA71                  jz      OK
.text:0000DA77                  cmp     esi, 0FFFFF8EFh
.text:0000DA7D                  jz      OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE XREF: sync_sys+201j
.text:0000DA83                  mov     cl, [ebx]
.text:0000DA85                  cmp     cl, 0Ch
.text:0000DA88                  jnz     short loc_DA9F
.text:0000DA8A                  cmp     esi, 12A9h
.text:0000DA90                  jz      OK
.text:0000DA96                  cmp     esi, 0FFFFFFF5h
.text:0000DA99                  jz      OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE XREF: sync_sys+220j
.text:0000DA9F                  mov     eax, [ebp+var_18]
.text:0000DAA2                  test    eax, eax
.text:0000DAA4                  jz      short loc_DAB0
.text:0000DAA6                  push    24h
.text:0000DAA8                  call    timer
```

```
.text:0000DAAD                  add     esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0:                                ; CODE XREF: sync_sys+23Cj
.text:0000DAB0                  inc     edi
.text:0000DAB1                  cmp     edi, 3
.text:0000DAB4                  jle     short loc_DA55
.text:0000DAB6                  mov     eax, ds:net_env
.text:0000DABB                  test    eax, eax
.text:0000DABD                  jz      short error

...

.text:0000DAF7 error:                                   ; CODE XREF: sync_sys+255j
.text:0000DAF7                                          ; sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name I gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                          ; sync_sys+2ABj
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
.text:0000D9BD                  mov     edx, [ebp+var_C] ; key
.text:0000D9C0                  mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                  push    edx
.text:0000D9C4                  push    20h
.text:0000D9C6                  push    ecx
.text:0000D9C7                  push    18h
.text:0000D9C9                  call    err_warn
.text:0000D9CE                  push    0Fh
.text:0000D9D0                  push    190h
.text:0000D9D5                  call    sound
.text:0000D9DA                  mov     [ebp+var_18], 1
.text:0000D9E1                  add     esp, 18h
.text:0000D9E4                  call    pcv_kbhit
.text:0000D9E9                  test    eax, eax
.text:0000D9EB                  jz      short loc_D9FB

...

; this name I gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, 48h, 6, 5Ah, 49h, 4Ch, 2 dup(47h↙
    ↳ )
.data:00401736                  db 51h, 4Fh, 47h, 61h, 20h, 22h, 3Ch, 24h, 33h, 36h, 76h
.data:00401736                  db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1Fh, 7, 1Eh, 1Ah
```

Заставить работать программу без донглы довольно просто: просто пропатчить все места после инструкций `CMP` где происходят соответствующие сравнения.

Еще одна возможность — это написать свой драйвер для SCO OpenServer.

55.2.1 Дешифровка сообщений об ошибке

Кстати, мы также можем дешифровать все сообщения об ошибке. Алкгоритм, находящийся в ф-ции `err_warn()` действительно, крайне прост:

Листинг 55.1: Ф-ция дешифровки

```
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
```

```
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453                  xor     eax, eax
.text:0000A455                  mov     al, [edx+edi] ; загружаем байт для дешифровки
.text:0000A458                  xor     eax, esi      ; дешифруем его
.text:0000A45A                  add     esi, 3        ; изменяем ключ для следующего байта
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
```

Как видно, не только сама строка поступает на вход, но также и ключ для дешифровки:

```
.text:0000DAF7 error:                                   ; CODE XREF: sync_sys+255j
.text:0000DAF7                                          ; sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name I gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE XREF: sync_sys+29Dj
.text:0000D9B6                                          ; sync_sys+2ABj
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
.text:0000D9BD                  mov     edx, [ebp+var_C] ; key
.text:0000D9C0                  mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                  push    edx
.text:0000D9C4                  push    20h
.text:0000D9C6                  push    ecx
.text:0000D9C7                  push    18h
.text:0000D9C9                  call    err_warn
```

Алгоритм это очень простой xoring: каждый байт XOR-ится с ключом, но ключ увеличивается на 3 после обработки каждого байта.

Я написал небольшой скрипт на Python для проверки своих догадок:

Листинг 55.2: Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
        sys.stdout.write ("%c" % (i^tmp))
        tmp=tmp+3
sys.stdout.flush()
```

И он выводит: “check security device connection”. Так что да, это дешифрованное сообщение.

Здесь есть также и другие сообщения, с соответствующими ключами. Но надо сказать, их можно дешифровать и без ключей. В начале, мы можем заметить что ключ это просто байт. Это потому что самая важная часть ф-ции дешифровки (`XOR`) оперирует байтами. Ключ находится в регистре `ESI`, но только младшие 8 бит (т.е., байт) регистра используются. Следовательно, ключ может быть больше 255, но его значение будет округляться.

И как следствие, мы можем попробовать обычный перебор всех ключей в диапазоне 0..255. Мы также можем пропускать сообщения содержащие непечатные символы.

Листинг 55.3: Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],

[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]


def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
        print ("message #%d" % cnt)
        for key in range(0,256):
                result=[]
                tmp=key
                for i in msg:
                        result.append (i^tmp)
                        tmp=tmp+3
                if is_string_printable (result):
                        print ("key=", key, "value=", "".join(list(map(chr, result))))
        cnt=cnt+1
```

И мы получим:

Листинг 55.4: Results

```
message #1
key= 20 value= ‘eb^h%|‘‘hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!‘nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!’!#!
message #3
key= 7 value= Bk<waoqNUpu$‘yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%gxqd‘‘_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.‘mwhj
key= 10 value= Ol!td‘tMhwx’efwfbf!tubuvnm!anvok
```

```
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{‘whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#‘juknuhydk!vrbsp!Zy‘dbefe
message #5
key= 17 value= check security device station
key= 18 value= ‘ijbh!td‘tmhwx’efwfbf!tubuVnm!’!
```

Тут есть какой-то мусор, но мы можем быстро отыскать сообщения на английском языке!

Кстати, так как алгоритм использует простой XOR, та же ф-ция может использоваться и для шифрования сообщения. Если нужно, мы можем зашифровать наши собственные сообщения, и пропатчить программу вставив их.

55.3 Пример #3: MS-DOS

Еще одна очень старая программа для MS-DOS от 1995 также разработанная давно исчезнувшей компанией.

Во времена перед DOS-экстендерами, всё ПО для MS-DOS рассчитывалось на процессоры 8086 или 80286, так что в своей массе весь код был 16-битным. 16-битный код в основном такой же, какой вы уже видели в этой книге, но все регистры 16-битные, и доступно меньше инструкций.

Среда MS-DOS не могла иметь никаких драйверов, и ПО работало с “голым” железом через порты, так что здесь вы можете увидеть инструкции OUT/IN, которые в наше время присутствуют в основном только в драйверах (в современных OS нельзя обращаться на прямую к портам из user mode).

Учитывая это, ПО для MS-DOS должно работать с донглой обращаясь к принтерному LPT-порту напрямую. Так что мы можем просто поискать эти инструкции. И да, вот они:

```
seg030:0034             out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034                               ; sent_pro+2Ap ...
seg030:0034
seg030:0034             arg_0    = byte ptr  6
seg030:0034
seg030:0034 55                   push    bp
seg030:0035 8B EC                mov     bp, sp
seg030:0037 8B 16 7E E7          mov     dx, _out_port ; 0x378
seg030:003B 8A 46 06             mov     al, [bp+arg_0]
seg030:003E EE                   out     dx, al
seg030:003F 5D                   pop     bp
seg030:0040 CB                   retf
seg030:0040             out_port endp
```

(Все имена меток в этом примере даны мною).

Функция out_port() вызывается только из одной ф-ции:

```
seg030:0041             sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041
seg030:0041             var_3    = byte ptr -3
seg030:0041             var_2    = word ptr -2
seg030:0041             arg_0    = dword ptr  6
seg030:0041
seg030:0041 C8 04 00 00          enter   4, 0
seg030:0045 56                   push    si
seg030:0046 57                   push    di
seg030:0047 8B 16 82 E7          mov     dx, _in_port_1 ; 0x37A
seg030:004B EC                   in      al, dx
seg030:004C 8A D8                mov     bl, al
seg030:004E 80 E3 FE             and     bl, 0FEh
seg030:0051 80 CB 04             or      bl, 4
seg030:0054 8A C3                mov     al, bl
seg030:0056 88 46 FD             mov     [bp+var_3], al
seg030:0059 80 E3 1F             and     bl, 1Fh
seg030:005C 8A C3                mov     al, bl
seg030:005E EE                   out     dx, al
```

```
seg030:005F 68 FF 00            push    0FFh
seg030:0062 0E                  push    cs
seg030:0063 E8 CE FF            call    near ptr out_port
seg030:0066 59                  pop     cx
seg030:0067 68 D3 00            push    0D3h
seg030:006A 0E                  push    cs
seg030:006B E8 C6 FF            call    near ptr out_port
seg030:006E 59                  pop     cx
seg030:006F 33 F6               xor     si, si
seg030:0071 EB 01               jmp     short loc_359D4
seg030:0073
seg030:0073             loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46                  inc     si
seg030:0074
seg030:0074             loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00         cmp     si, 96h
seg030:0078 7C F9               jl      short loc_359D3
seg030:007A 68 C3 00            push    0C3h
seg030:007D 0E                  push    cs
seg030:007E E8 B3 FF            call    near ptr out_port
seg030:0081 59                  pop     cx
seg030:0082 68 C7 00            push    0C7h
seg030:0085 0E                  push    cs
seg030:0086 E8 AB FF            call    near ptr out_port
seg030:0089 59                  pop     cx
seg030:008A 68 D3 00            push    0D3h
seg030:008D 0E                  push    cs
seg030:008E E8 A3 FF            call    near ptr out_port
seg030:0091 59                  pop     cx
seg030:0092 68 C3 00            push    0C3h
seg030:0095 0E                  push    cs
seg030:0096 E8 9B FF            call    near ptr out_port
seg030:0099 59                  pop     cx
seg030:009A 68 C7 00            push    0C7h
seg030:009D 0E                  push    cs
seg030:009E E8 93 FF            call    near ptr out_port
seg030:00A1 59                  pop     cx
seg030:00A2 68 D3 00            push    0D3h
seg030:00A5 0E                  push    cs
seg030:00A6 E8 8B FF            call    near ptr out_port
seg030:00A9 59                  pop     cx
seg030:00AA BF FF FF            mov     di, 0FFFFh
seg030:00AD EB 40               jmp     short loc_35A4F
seg030:00AF
seg030:00AF             loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00            mov     si, 4
seg030:00B2
seg030:00B2             loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7               shl     di, 1
seg030:00B4 8B 16 80 E7         mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC                  in      al, dx
seg030:00B9 A8 80               test    al, 80h
seg030:00BB 75 03               jnz     short loc_35A20
seg030:00BD 83 CF 01            or      di, 1
seg030:00C0
seg030:00C0             loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+        test    [bp+var_2], 8
seg030:00C5 74 05               jz      short loc_35A2C
seg030:00C7 68 D7 00            push    0D7h ; '+'
seg030:00CA EB 0B               jmp     short loc_35A37
seg030:00CC
```

```
seg030:00CC             loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00               push    0C3h
seg030:00CF 0E                     push    cs
seg030:00D0 E8 61 FF               call    near ptr out_port
seg030:00D3 59                     pop     cx
seg030:00D4 68 C7 00               push    0C7h
seg030:00D7
seg030:00D7             loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E                     push    cs
seg030:00D8 E8 59 FF               call    near ptr out_port
seg030:00DB 59                     pop     cx
seg030:00DC 68 D3 00               push    0D3h
seg030:00DF 0E                     push    cs
seg030:00E0 E8 51 FF               call    near ptr out_port
seg030:00E3 59                     pop     cx
seg030:00E4 8B 46 FE               mov     ax, [bp+var_2]
seg030:00E7 D1 E0                  shl     ax, 1
seg030:00E9 89 46 FE               mov     [bp+var_2], ax
seg030:00EC 4E                     dec     si
seg030:00ED 75 C3                  jnz     short loc_35A12
seg030:00EF
seg030:00EF             loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06               les     bx, [bp+arg_0]
seg030:00F2 FF 46 06               inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07               mov     al, es:[bx]
seg030:00F8 98                     cbw
seg030:00F9 89 46 FE               mov     [bp+var_2], ax
seg030:00FC 0B C0                  or      ax, ax
seg030:00FE 75 AF                  jnz     short loc_35A0F
seg030:0100 68 FF 00               push    0FFh
seg030:0103 0E                     push    cs
seg030:0104 E8 2D FF               call    near ptr out_port
seg030:0107 59                     pop     cx
seg030:0108 8B 16 82 E7            mov     dx, _in_port_1 ; 0x37A
seg030:010C EC                     in      al, dx
seg030:010D 8A C8                  mov     cl, al
seg030:010F 80 E1 5F               and     cl, 5Fh
seg030:0112 8A C1                  mov     al, cl
seg030:0114 EE                     out     dx, al
seg030:0115 EC                     in      al, dx
seg030:0116 8A C8                  mov     cl, al
seg030:0118 F6 C1 20               test    cl, 20h
seg030:011B 74 08                  jz      short loc_35A85
seg030:011D 8A 5E FD               mov     bl, [bp+var_3]
seg030:0120 80 E3 DF               and     bl, 0DFh
seg030:0123 EB 03                  jmp     short loc_35A88
seg030:0125
seg030:0125             loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD               mov     bl, [bp+var_3]
seg030:0128
seg030:0128             loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80               test    cl, 80h
seg030:012B 74 03                  jz      short loc_35A90
seg030:012D 80 E3 7F               and     bl, 7Fh
seg030:0130
seg030:0130             loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7            mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3                  mov     al, bl
seg030:0136 EE                     out     dx, al
seg030:0137 8B C7                  mov     ax, di
seg030:0139 5F                     pop     di
```

```
seg030:013A 5E                  pop     si
seg030:013B C9                  leave
seg030:013C CB                  retf
seg030:013C             sent_pro endp
```

Это также “хеширующая” донгла Sentinel Pro как и в предыдущем примере. Я заметил это по тому что текстовые строки передаются и здесь, 16-битные значения также возвращаются и сравниваются с другими.

Так вот как происходит работа с Sentinel Pro через порты. Адрес выходного порта обычно 0x378, т.е., принтерного порта, данные для него во времена перед USB отправлялись прямо сюда. Порт однонаправленный, потому что когда его разрабатывали, никто не мог предположить, что кому-то понадобится получать информацию из принтера [4]. Единственный способ получить информацию из принтера это регистр статуса на порту 0x379, он содержит такие биты как “paper out”, “ack”, “busy” — так принтер может сигнализировать о том, что он готов или нет, и о том, есть ли в нем бумага. Так что донгла возвращает информацию через какой-то из этих бит, по одному биту на каждой итерации.

`_in_port_2` содержит адрес статуса (0x379) и `_in_port_1` содержит адрес управляющего регистра (0x37A).

Судя по всему, донгла возвращает информацию только через флаг “busy” на `seg030:00B9`: каждый бит записывается в регистре `DI` позже возвращаемый в самом конце ф-ции.

Что означают все эти отсылаемые в выходной порт байты? Я не знаю. Возможно, команды донглы. Но честно говоря, нам и не обязательно знать: нашу задачу можно легко решить и без этих знаний.

Вот ф-ция проверки донглы:

```
00000000 struct_0        struc ; (sizeof=0x1B)
00000000 field_0         db 25 dup(?)            ; string(C)
00000019 _A              dw ?
0000001B struct_0        ends

dseg:3CBC 61 63 72 75+_Q   struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+       ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+     struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+     struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+     struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+     struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+     struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+     struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+     struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+     struct_0 <'copy', 0F557h>


seg030:0145             check_dongle proc far ; CODE XREF: sub_3771D+3EP
seg030:0145
seg030:0145             var_6 = dword ptr -6
seg030:0145             var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00       enter   6, 0
seg030:0149 56                push    si
seg030:014A 66 6A 00          push    large 0         ; newtime
seg030:014D 6A 00             push    0               ; cmd
seg030:014F 9A C1 18 00+      call    _biostime
seg030:0154 52                push    dx
seg030:0155 50                push    ax
seg030:0156 66 58             pop     eax
seg030:0158 83 C4 06          add     sp, 6
seg030:015B 66 89 46 FA       mov     [bp+var_6], eax
seg030:015F 66 3B 06 D8+      cmp     eax, _expiration
seg030:0164 7E 44             jle     short loc_35B0A
seg030:0166 6A 14             push    14h
```

[4] Если учитывать только Centronics и не учитывать последующий стандарт IEEE 1284 — в нем из принтера можно получать информацию.

```
seg030:0168 90                nop
seg030:0169 0E                push    cs
seg030:016A E8 52 00          call    near ptr get_rand
seg030:016D 59                pop     cx
seg030:016E 8B F0             mov     si, ax
seg030:0170 6B C0 1B          imul    ax, 1Bh
seg030:0173 05 BC 3C          add     ax, offset _Q
seg030:0176 1E                push    ds
seg030:0177 50                push    ax
seg030:0178 0E                push    cs
seg030:0179 E8 C5 FE          call    near ptr sent_pro
seg030:017C 83 C4 04          add     sp, 4
seg030:017F 89 46 FE          mov     [bp+var_2], ax
seg030:0182 8B C6             mov     ax, si
seg030:0184 6B C0 12          imul    ax, 18
seg030:0187 66 0F BF C0       movsx   eax, ax
seg030:018B 66 8B 56 FA       mov     edx, [bp+var_6]
seg030:018F 66 03 D0          add     edx, eax
seg030:0192 66 89 16 D8+      mov     _expiration, edx
seg030:0197 8B DE             mov     bx, si
seg030:0199 6B DB 1B          imul    bx, 27
seg030:019C 8B 87 D5 3C       mov     ax, _Q._A[bx]
seg030:01A0 3B 46 FE          cmp     ax, [bp+var_2]
seg030:01A3 74 05             jz      short loc_35B0A
seg030:01A5 B8 01 00          mov     ax, 1
seg030:01A8 EB 02             jmp     short loc_35B0C
seg030:01AA
seg030:01AA             loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                        ; check_dongle+5Ej
seg030:01AA 33 C0               xor     ax, ax
seg030:01AC
seg030:01AC             loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E                  pop     si
seg030:01AD C9                  leave
seg030:01AE CB                  retf
seg030:01AE             check_dongle endp
```

А так как эта ф-ция может вызываться слишком часто, например, перед выполнением каждой важной возможности ПО, а обращение к донгле вообще-то медленное (и из-за медленного принтерного порта и из-за медленного MCU[5] в донгле), так что они, наверное, добавили возможность пропускать проверку донглы слишком часто, используя текущее время в ф-ции `biostime()`.

Ф-ция `get_rand()` использует стандартную ф-цию Си:

```
seg030:01BF             get_rand proc far ; CODE XREF: check_dongle+25p
seg030:01BF
seg030:01BF             arg_0    = word ptr  6
seg030:01BF
seg030:01BF 55                   push    bp
seg030:01C0 8B EC                mov     bp, sp
seg030:01C2 9A 3D 21 00+         call    _rand
seg030:01C7 66 0F BF C0          movsx   eax, ax
seg030:01CB 66 0F BF 56+         movsx   edx, [bp+arg_0]
seg030:01D0 66 0F AF C2          imul    eax, edx
seg030:01D4 66 BB 00 80+         mov     ebx, 8000h
seg030:01DA 66 99                cdq
seg030:01DC 66 F7 FB             idiv    ebx
seg030:01DF 5D                   pop     bp
seg030:01E0 CB                   retf
seg030:01E0             get_rand endp
```

[5]Microcontroller unit

Так что текстовая строка выбирается случайно, отправляется в донглу и результат хеширования сверяется с корректным значением.

Текстовые строки, похоже, выбирались так же случайно.

И вот как вызывается главная процедура проверки донглы:

```
seg033:087B 9A 45 01 96+   call    check_dongle
seg033:0880 0B C0          or      ax, ax
seg033:0882 74 62          jz      short OK
seg033:0884 83 3E 60 42+   cmp     word_620E0, 0
seg033:0889 75 5B          jnz     short OK
seg033:088B FF 06 60 42    inc     word_620E0
seg033:088F 1E             push    ds
seg033:0890 68 22 44       push    offset aTrupcRequiresA ; "This Software Requires a Software ↙
    ↳ Lock\n"
seg033:0893 1E             push    ds
seg033:0894 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+   call    _strcpy
seg033:089C 83 C4 08       add     sp, 8
seg033:089F 1E             push    ds
seg033:08A0 68 42 44       push    offset aPleaseContactA ; "Please Contact ..."
seg033:08A3 1E             push    ds
seg033:08A4 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+   call    _strcat
```

Заставить работать программу без донглы очень просто: просто заставить ф-цию `check_dongle()` возвращать всегда 0.

Например, вставив такой код в самом её начале:

```
mov ax,0
retf
```

Наблюдательный читатель может заметить что ф-ция Си `strcpy()` имеет 2 аргумента, но здесь мы видим, что передается 4:

```
seg033:088F 1E                         push    ds
seg033:0890 68 22 44                   push    offset aTrupcRequiresA ; "This Software ↙
    ↳ Requires a Software Lock\n"
seg033:0893 1E                         push    ds
seg033:0894 68 60 E9                   push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+               call    _strcpy
seg033:089C 83 C4 08                   add     sp, 8
```

Об этом больше читайте здесь: 67.

Так что, `strcpy()`, как и любая другая ф-ция принимающая указатель (-и) в аргументах, работает с 16-битными парами.

Вернемся к нашему примеру. `DS` сейчас указывает на сегмент данных размещенный в исполняемом файле, там, где хранится текстовая строка.

В ф-ции `sent_pro()` каждый байт строки загружается на `seg030:00EF`: инструкция `LES` загружает из переданного аргумента пару ES:BX одновременно. `MOV` на `seg030:00F5` загружает байт из памяти, на который указывает пара ES:BX.

На `seg030:00F2` инкрементируется только 16-битное слово, но не значение сегмента. Это значит, что переданная в ф-цию строка не может находится на границе двух сегментов.

Глава 56

“QR9”: Любительская криптосистема, вдохновленная кубиком Рубика

Любительские криптосистемы иногда встречаются довольно странные.

Однажды меня попросили разобраться с одним таким любительским криптоалгоритмом встроенным в утилиту для шифрования, исходный код которой был утерян[1].

Вот листинг этой утилиты для шифрования, полученный при помощи IDA:

```
.text:00541000 set_bit         proc near               ; CODE XREF: rotate1+42
.text:00541000                                         ; rotate2+42 ...
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
.text:00541009                 mov     esi, [esp+4+arg_0]
.text:0054100D                 test    al, al
.text:0054100F                 mov     eax, [esp+4+arg_4]
.text:00541013                 mov     dl, 1
.text:00541015                 jz      short loc_54102B
.text:00541017                 shl     dl, cl
.text:00541019                 mov     cl, cube64[eax+esi*8]
.text:00541020                 or      cl, dl
.text:00541022                 mov     cube64[eax+esi*8], cl
.text:00541029                 pop     esi
.text:0054102A                 retn
.text:0054102B
.text:0054102B loc_54102B:                             ; CODE XREF: set_bit+15
.text:0054102B                 shl     dl, cl
.text:0054102D                 mov     cl, cube64[eax+esi*8]
.text:00541034                 not     dl
.text:00541036                 and     cl, dl
.text:00541038                 mov     cube64[eax+esi*8], cl
.text:0054103F                 pop     esi
.text:00541040                 retn
.text:00541040 set_bit         endp
.text:00541040
.text:00541041                 align 10h
.text:00541050
.text:00541050 ; ============== S U B R O U T I N E =======================================
.text:00541050
.text:00541050
```

[1] Я также получил разрешение от клиента на публикацию деталей алгоритма

```
.text:00541050 get_bit         proc near               ; CODE XREF: rotate1+16
.text:00541050                                         ; rotate2+16 ...
.text:00541050
.text:00541050 arg_0           = dword ptr  4
.text:00541050 arg_4           = dword ptr  8
.text:00541050 arg_8           = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr     al, cl
.text:00541065                 and     al, 1
.text:00541067                 retn
.text:00541067 get_bit         endp
.text:00541067
.text:00541068                 align 10h
.text:00541070
.text:00541070 ; =============== S U B R O U T I N E =======================================
.text:00541070
.text:00541070
.text:00541070 rotate1         proc near               ; CODE XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0           = dword ptr  4
.text:00541070
.text:00541070                 sub     esp, 40h
.text:00541073                 push    ebx
.text:00541074                 push    ebp
.text:00541075                 mov     ebp, [esp+48h+arg_0]
.text:00541079                 push    esi
.text:0054107A                 push    edi
.text:0054107B                 xor     edi, edi        ; EDI is loop1 counter
.text:0054107D                 lea     ebx, [esp+50h+internal_array_64]
.text:00541081
.text:00541081 first_loop1_begin:                      ; CODE XREF: rotate1+2E
.text:00541081                 xor     esi, esi        ; ESI is loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:                      ; CODE XREF: rotate1+25
.text:00541083                 push    ebp             ; arg_0
.text:00541084                 push    esi
.text:00541085                 push    edi
.text:00541086                 call    get_bit
.text:0054108B                 add     esp, 0Ch
.text:0054108E                 mov     [ebx+esi], al   ; store to internal array
.text:00541091                 inc     esi
.text:00541092                 cmp     esi, 8
.text:00541095                 jl      short first_loop2_begin
.text:00541097                 inc     edi
.text:00541098                 add     ebx, 8
.text:0054109B                 cmp     edi, 8
.text:0054109E                 jl      short first_loop1_begin
.text:005410A0                 lea     ebx, [esp+50h+internal_array_64]
.text:005410A4                 mov     edi, 7          ; EDI is loop1 counter, initial state is↙
    ↳ 7
.text:005410A9
.text:005410A9 second_loop1_begin:                     ; CODE XREF: rotate1+57
.text:005410A9                 xor     esi, esi        ; ESI is loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:                     ; CODE XREF: rotate1+4E
.text:005410AB                 mov     al, [ebx+esi]   ; value from internal array
```

```
.text:005410AE                 push    eax
.text:005410AF                 push    ebp             ; arg_0
.text:005410B0                 push    edi
.text:005410B1                 push    esi
.text:005410B2                 call    set_bit
.text:005410B7                 add     esp, 10h
.text:005410BA                 inc     esi             ; increment loop2 counter
.text:005410BB                 cmp     esi, 8
.text:005410BE                 jl      short second_loop2_begin
.text:005410C0                 dec     edi             ; decrement loop2 counter
.text:005410C1                 add     ebx, 8
.text:005410C4                 cmp     edi, 0FFFFFFFFh
.text:005410C7                 jg      short second_loop1_begin
.text:005410C9                 pop     edi
.text:005410CA                 pop     esi
.text:005410CB                 pop     ebp
.text:005410CC                 pop     ebx
.text:005410CD                 add     esp, 40h
.text:005410D0                 retn
.text:005410D0 rotate1         endp
.text:005410D0
.text:005410D1                 align 10h
.text:005410E0
.text:005410E0 ; ============== S U B R O U T I N E =======================================
.text:005410E0
.text:005410E0
.text:005410E0 rotate2         proc near               ; CODE XREF: rotate_all_with_password+7A
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0           = dword ptr  4
.text:005410E0
.text:005410E0                 sub     esp, 40h
.text:005410E3                 push    ebx
.text:005410E4                 push    ebp
.text:005410E5                 mov     ebp, [esp+48h+arg_0]
.text:005410E9                 push    esi
.text:005410EA                 push    edi
.text:005410EB                 xor     edi, edi        ; loop1 counter
.text:005410ED                 lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:                             ; CODE XREF: rotate2+2E
.text:005410F1                 xor     esi, esi        ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:                             ; CODE XREF: rotate2+25
.text:005410F3                 push    esi             ; loop2
.text:005410F4                 push    edi             ; loop1
.text:005410F5                 push    ebp             ; arg_0
.text:005410F6                 call    get_bit
.text:005410FB                 add     esp, 0Ch
.text:005410FE                 mov     [ebx+esi], al   ; store to internal array
.text:00541101                 inc     esi             ; increment loop1 counter
.text:00541102                 cmp     esi, 8
.text:00541105                 jl      short loc_5410F3
.text:00541107                 inc     edi             ; increment loop2 counter
.text:00541108                 add     ebx, 8
.text:0054110B                 cmp     edi, 8
.text:0054110E                 jl      short loc_5410F1
.text:00541110                 lea     ebx, [esp+50h+internal_array_64]
.text:00541114                 mov     edi, 7          ; loop1 counter is initial state 7
.text:00541119
.text:00541119 loc_541119:                             ; CODE XREF: rotate2+57
```

```
.text:00541119                 xor     esi, esi        ; loop2 counter
.text:0054111B
.text:0054111B loc_54111B:                             ; CODE XREF: rotate2+4E
.text:0054111B                 mov     al, [ebx+esi]   ; get byte from internal array
.text:0054111E                 push    eax
.text:0054111F                 push    edi             ; loop1 counter
.text:00541120                 push    esi             ; loop2 counter
.text:00541121                 push    ebp             ; arg_0
.text:00541122                 call    set_bit
.text:00541127                 add     esp, 10h
.text:0054112A                 inc     esi             ; increment loop2 counter
.text:0054112B                 cmp     esi, 8
.text:0054112E                 jl      short loc_54111B
.text:00541130                 dec     edi             ; decrement loop2 counter
.text:00541131                 add     ebx, 8
.text:00541134                 cmp     edi, 0FFFFFFFFh
.text:00541137                 jg      short loc_541119
.text:00541139                 pop     edi
.text:0054113A                 pop     esi
.text:0054113B                 pop     ebp
.text:0054113C                 pop     ebx
.text:0054113D                 add     esp, 40h
.text:00541140                 retn
.text:00541140 rotate2         endp
.text:00541140
.text:00541141                 align 10h
.text:00541150
.text:00541150 ; =============== S U B R O U T I N E =======================================
.text:00541150
.text:00541150
.text:00541150 rotate3         proc near               ; CODE XREF: rotate_all_with_password+66
.text:00541150
.text:00541150 var_40          = byte ptr -40h
.text:00541150 arg_0           = dword ptr  4
.text:00541150
.text:00541150                 sub     esp, 40h
.text:00541153                 push    ebx
.text:00541154                 push    ebp
.text:00541155                 mov     ebp, [esp+48h+arg_0]
.text:00541159                 push    esi
.text:0054115A                 push    edi
.text:0054115B                 xor     edi, edi
.text:0054115D                 lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161:                             ; CODE XREF: rotate3+2E
.text:00541161                 xor     esi, esi
.text:00541163
.text:00541163 loc_541163:                             ; CODE XREF: rotate3+25
.text:00541163                 push    esi
.text:00541164                 push    ebp
.text:00541165                 push    edi
.text:00541166                 call    get_bit
.text:0054116B                 add     esp, 0Ch
.text:0054116E                 mov     [ebx+esi], al
.text:00541171                 inc     esi
.text:00541172                 cmp     esi, 8
.text:00541175                 jl      short loc_541163
.text:00541177                 inc     edi
.text:00541178                 add     ebx, 8
.text:0054117B                 cmp     edi, 8
.text:0054117E                 jl      short loc_541161
```

```
.text:00541180                 xor     ebx, ebx
.text:00541182                 lea     edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:                             ; CODE XREF: rotate3+54
.text:00541186                 mov     esi, 7
.text:0054118B
.text:0054118B loc_54118B:                             ; CODE XREF: rotate3+4E
.text:0054118B                 mov     al, [edi]
.text:0054118D                 push    eax
.text:0054118E                 push    ebx
.text:0054118F                 push    ebp
.text:00541190                 push    esi
.text:00541191                 call    set_bit
.text:00541196                 add     esp, 10h
.text:00541199                 inc     edi
.text:0054119A                 dec     esi
.text:0054119B                 cmp     esi, 0FFFFFFFFh
.text:0054119E                 jg      short loc_54118B
.text:005411A0                 inc     ebx
.text:005411A1                 cmp     ebx, 8
.text:005411A4                 jl      short loc_541186
.text:005411A6                 pop     edi
.text:005411A7                 pop     esi
.text:005411A8                 pop     ebp
.text:005411A9                 pop     ebx
.text:005411AA                 add     esp, 40h
.text:005411AD                 retn
.text:005411AD rotate3         endp
.text:005411AD
.text:005411AE                 align 10h
.text:005411B0
.text:005411B0 ; ============== S U B R O U T I N E =======================================
.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near      ; CODE XREF: crypt+1F
.text:005411B0                                         ; decrypt+36
.text:005411B0
.text:005411B0 arg_0           = dword ptr  4
.text:005411B0 arg_4           = dword ptr  8
.text:005411B0
.text:005411B0                 mov     eax, [esp+arg_0]
.text:005411B4                 push    ebp
.text:005411B5                 mov     ebp, eax
.text:005411B7                 cmp     byte ptr [eax], 0
.text:005411BA                 jz      exit
.text:005411C0                 push    ebx
.text:005411C1                 mov     ebx, [esp+8+arg_4]
.text:005411C5                 push    esi
.text:005411C6                 push    edi
.text:005411C7
.text:005411C7 loop_begin:                             ; CODE XREF: rotate_all_with_password+9F
.text:005411C7                 movsx   eax, byte ptr [ebp+0]
.text:005411CB                 push    eax             ; C
.text:005411CC                 call    _tolower
.text:005411D1                 add     esp, 4
.text:005411D4                 cmp     al, 'a'
.text:005411D6                 jl      short next_character_in_password
.text:005411D8                 cmp     al, 'z'
.text:005411DA                 jg      short next_character_in_password
.text:005411DC                 movsx   ecx, al
.text:005411DF                 sub     ecx, 'a'
```

```
.text:005411E2                 cmp     ecx, 24
.text:005411E5                 jle     short skip_subtracting
.text:005411E7                 sub     ecx, 24
.text:005411EA
.text:005411EA skip_subtracting:                       ; CODE XREF: rotate_all_with_password+35
.text:005411EA                 mov     eax, 55555556h
.text:005411EF                 imul    ecx
.text:005411F1                 mov     eax, edx
.text:005411F3                 shr     eax, 1Fh
.text:005411F6                 add     edx, eax
.text:005411F8                 mov     eax, ecx
.text:005411FA                 mov     esi, edx
.text:005411FC                 mov     ecx, 3
.text:00541201                 cdq
.text:00541202                 idiv    ecx
.text:00541204                 sub     edx, 0
.text:00541207                 jz      short call_rotate1
.text:00541209                 dec     edx
.text:0054120A                 jz      short call_rotate2
.text:0054120C                 dec     edx
.text:0054120D                 jnz     short next_character_in_password
.text:0054120F                 test    ebx, ebx
.text:00541211                 jle     short next_character_in_password
.text:00541213                 mov     edi, ebx
.text:00541215
.text:00541215 call_rotate3:                           ; CODE XREF: rotate_all_with_password+6F
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:                           ; CODE XREF: rotate_all_with_password+5A
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:                             ; CODE XREF: rotate_all_with_password+83
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:                           ; CODE XREF: rotate_all_with_password+57
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:                             ; CODE XREF: rotate_all_with_password+97
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
.text:00541246                 dec     edi
.text:00541247                 jnz     short loc_54123D
.text:00541249
.text:00541249 next_character_in_password:             ; CODE XREF: rotate_all_with_password+26
.text:00541249                                         ; rotate_all_with_password+2A ...
.text:00541249                 mov     al, [ebp+1]
```

```
.text:0054124C                 inc     ebp
.text:0054124D                 test    al, al
.text:0054124F                 jnz     loop_begin
.text:00541255                 pop     edi
.text:00541256                 pop     esi
.text:00541257                 pop     ebx
.text:00541258
.text:00541258 exit:                                   ; CODE XREF: rotate_all_with_password+A
.text:00541258                 pop     ebp
.text:00541259                 retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:0054125A                 align 10h
.text:00541260
.text:00541260 ; ============== S U B R O U T I N E =======================================
.text:00541260
.text:00541260
.text:00541260 crypt           proc near               ; CODE XREF: crypt_file+8A
.text:00541260
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:                             ; CODE XREF: crypt+41
.text:0054126A                 mov     eax, [esp+10h+arg_8]
.text:0054126E                 mov     ecx, 10h
.text:00541273                 mov     esi, ebx
.text:00541275                 mov     edi, offset cube64
.text:0054127A                 push    1
.text:0054127C                 push    eax
.text:0054127D                 rep movsd
.text:0054127F                 call    rotate_all_with_password
.text:00541284                 mov     eax, [esp+18h+arg_4]
.text:00541288                 mov     edi, ebx
.text:0054128A                 add     ebp, 40h
.text:0054128D                 add     esp, 8
.text:00541290                 mov     ecx, 10h
.text:00541295                 mov     esi, offset cube64
.text:0054129A                 add     ebx, 40h
.text:0054129D                 cmp     ebp, eax
.text:0054129F                 rep movsd
.text:005412A1                 jl      short loc_54126A
.text:005412A3                 pop     edi
.text:005412A4                 pop     esi
.text:005412A5                 pop     ebp
.text:005412A6                 pop     ebx
.text:005412A7                 retn
.text:005412A7 crypt           endp
.text:005412A7
.text:005412A8                 align 10h
.text:005412B0
.text:005412B0 ; ============== S U B R O U T I N E =======================================
.text:005412B0
.text:005412B0
```

```
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt         proc near               ; CODE XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0           = dword ptr  4
.text:005412B0 arg_4           = dword ptr  8
.text:005412B0 Src             = dword ptr  0Ch
.text:005412B0
.text:005412B0                 mov     eax, [esp+Src]
.text:005412B4                 push    ebx
.text:005412B5                 push    ebp
.text:005412B6                 push    esi
.text:005412B7                 push    edi
.text:005412B8                 push    eax             ; Src
.text:005412B9                 call    __strdup
.text:005412BE                 push    eax             ; Str
.text:005412BF                 mov     [esp+18h+Src], eax
.text:005412C3                 call    __strrev
.text:005412C8                 mov     ebx, [esp+18h+arg_0]
.text:005412CC                 add     esp, 8
.text:005412CF                 xor     ebp, ebp
.text:005412D1
.text:005412D1 loc_5412D1:                             ; CODE XREF: decrypt+58
.text:005412D1                 mov     ecx, 10h
.text:005412D6                 mov     esi, ebx
.text:005412D8                 mov     edi, offset cube64
.text:005412DD                 push    3
.text:005412DF                 rep movsd
.text:005412E1                 mov     ecx, [esp+14h+Src]
.text:005412E5                 push    ecx
.text:005412E6                 call    rotate_all_with_password
.text:005412EB                 mov     eax, [esp+18h+arg_4]
.text:005412EF                 mov     edi, ebx
.text:005412F1                 add     ebp, 40h
.text:005412F4                 add     esp, 8
.text:005412F7                 mov     ecx, 10h
.text:005412FC                 mov     esi, offset cube64
.text:00541301                 add     ebx, 40h
.text:00541304                 cmp     ebp, eax
.text:00541306                 rep movsd
.text:00541308                 jl      short loc_5412D1
.text:0054130A                 mov     edx, [esp+10h+Src]
.text:0054130E                 push    edx             ; Memory
.text:0054130F                 call    _free
.text:00541314                 add     esp, 4
.text:00541317                 pop     edi
.text:00541318                 pop     esi
.text:00541319                 pop     ebp
.text:0054131A                 pop     ebx
.text:0054131B                 retn
.text:0054131B decrypt         endp
.text:0054131B
.text:0054131C                 align 10h
.text:00541320
.text:00541320 ; =============== S U B R O U T I N E =======================================
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near               ; CODE XREF: _main+42
.text:00541320
.text:00541320 Str             = dword ptr  4
.text:00541320 Filename        = dword ptr  8
```

```
.text:00541320 password        = dword ptr  0Ch
.text:00541320
.text:00541320                 mov     eax, [esp+Str]
.text:00541324                 push    ebp
.text:00541325                 push    offset Mode     ; "rb"
.text:0054132A                 push    eax             ; Filename
.text:0054132B                 call    _fopen          ; open file
.text:00541330                 mov     ebp, eax
.text:00541332                 add     esp, 8
.text:00541335                 test    ebp, ebp
.text:00541337                 jnz     short loc_541348
.text:00541339                 push    offset Format   ; "Cannot open input file!\n"
.text:0054133E                 call    _printf
.text:00541343                 add     esp, 4
.text:00541346                 pop     ebp
.text:00541347                 retn
.text:00541348
.text:00541348 loc_541348:                             ; CODE XREF: crypt_file+17
.text:00541348                 push    ebx
.text:00541349                 push    esi
.text:0054134A                 push    edi
.text:0054134B                 push    2               ; Origin
.text:0054134D                 push    0               ; Offset
.text:0054134F                 push    ebp             ; File
.text:00541350                 call    _fseek
.text:00541355                 push    ebp             ; File
.text:00541356                 call    _ftell          ; get file size
.text:0054135B                 push    0               ; Origin
.text:0054135D                 push    0               ; Offset
.text:0054135F                 push    ebp             ; File
.text:00541360                 mov     [esp+2Ch+Str], eax
.text:00541364                 call    _fseek          ; rewind to start
.text:00541369                 mov     esi, [esp+2Ch+Str]
.text:0054136D                 and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370                 add     esi, 40h        ; align size to 64-byte border
.text:00541373                 push    esi             ; Size
.text:00541374                 call    _malloc
.text:00541379                 mov     ecx, esi
.text:0054137B                 mov     ebx, eax        ; allocated buffer pointer -> to EBX
.text:0054137D                 mov     edx, ecx
.text:0054137F                 xor     eax, eax
.text:00541381                 mov     edi, ebx
.text:00541383                 push    ebp             ; File
.text:00541384                 shr     ecx, 2
.text:00541387                 rep stosd
.text:00541389                 mov     ecx, edx
.text:0054138B                 push    1               ; Count
.text:0054138D                 and     ecx, 3
.text:00541390                 rep stosb               ; memset (buffer, 0, aligned_size)
.text:00541392                 mov     eax, [esp+38h+Str]
.text:00541396                 push    eax             ; ElementSize
.text:00541397                 push    ebx             ; DstBuf
.text:00541398                 call    _fread          ; read file
.text:0054139D                 push    ebp             ; File
.text:0054139E                 call    _fclose
.text:005413A3                 mov     ecx, [esp+44h+password]
.text:005413A7                 push    ecx             ; password
.text:005413A8                 push    esi             ; aligned size
.text:005413A9                 push    ebx             ; buffer
.text:005413AA                 call    crypt           ; do crypt
.text:005413AF                 mov     edx, [esp+50h+Filename]
```

```
.text:005413B3                 add     esp, 40h
.text:005413B6                 push    offset aWb      ; "wb"
.text:005413BB                 push    edx             ; Filename
.text:005413BC                 call    _fopen
.text:005413C1                 mov     edi, eax
.text:005413C3                 push    edi             ; File
.text:005413C4                 push    1               ; Count
.text:005413C6                 push    3               ; Size
.text:005413C8                 push    offset aQr9     ; "QR9"
.text:005413CD                 call    _fwrite         ; write file signature
.text:005413D2                 push    edi             ; File
.text:005413D3                 push    1               ; Count
.text:005413D5                 lea     eax, [esp+30h+Str]
.text:005413D9                 push    4               ; Size
.text:005413DB                 push    eax             ; Str
.text:005413DC                 call    _fwrite         ; write original file size
.text:005413E1                 push    edi             ; File
.text:005413E2                 push    1               ; Count
.text:005413E4                 push    esi             ; Size
.text:005413E5                 push    ebx             ; Str
.text:005413E6                 call    _fwrite         ; write crypted file
.text:005413EB                 push    edi             ; File
.text:005413EC                 call    _fclose
.text:005413F1                 push    ebx             ; Memory
.text:005413F2                 call    _free
.text:005413F7                 add     esp, 40h
.text:005413FA                 pop     edi
.text:005413FB                 pop     esi
.text:005413FC                 pop     ebx
.text:005413FD                 pop     ebp
.text:005413FE                 retn
.text:005413FE crypt_file      endp
.text:005413FE
.text:005413FF                 align 10h
.text:00541400
.text:00541400 ; ============== S U B R O U T I N E =======================================
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file    proc near               ; CODE XREF: _main+6E
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
```

```
.text:0054142A                pop     esi
.text:0054142B                pop     ebp
.text:0054142C                pop     ebx
.text:0054142D                retn
.text:0054142E
.text:0054142E loc_54142E:                            ; CODE XREF: decrypt_file+1A
.text:0054142E                push    2               ; Origin
.text:00541430                push    0               ; Offset
.text:00541432                push    esi             ; File
.text:00541433                call    _fseek
.text:00541438                push    esi             ; File
.text:00541439                call    _ftell
.text:0054143E                push    0               ; Origin
.text:00541440                push    0               ; Offset
.text:00541442                push    esi             ; File
.text:00541443                mov     ebp, eax
.text:00541445                call    _fseek
.text:0054144A                push    ebp             ; Size
.text:0054144B                call    _malloc
.text:00541450                push    esi             ; File
.text:00541451                mov     ebx, eax
.text:00541453                push    1               ; Count
.text:00541455                push    ebp             ; ElementSize
.text:00541456                push    ebx             ; DstBuf
.text:00541457                call    _fread
.text:0054145C                push    esi             ; File
.text:0054145D                call    _fclose
.text:00541462                add     esp, 34h
.text:00541465                mov     ecx, 3
.text:0054146A                mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                mov     esi, ebx
.text:00541471                xor     edx, edx
.text:00541473                repe cmpsb
.text:00541475                jz      short loc_541489
.text:00541477                push    offset aFileIsNotCrypt ; "File is not crypted!\n"
.text:0054147C                call    _printf
.text:00541481                add     esp, 4
.text:00541484                pop     edi
.text:00541485                pop     esi
.text:00541486                pop     ebp
.text:00541487                pop     ebx
.text:00541488                retn
.text:00541489
.text:00541489 loc_541489:                            ; CODE XREF: decrypt_file+75
.text:00541489                mov     eax, [esp+10h+Src]
.text:0054148D                mov     edi, [ebx+3]
.text:00541490                add     ebp, 0FFFFFFF9h
.text:00541493                lea     esi, [ebx+7]
.text:00541496                push    eax             ; Src
.text:00541497                push    ebp             ; int
.text:00541498                push    esi             ; int
.text:00541499                call    decrypt
.text:0054149E                mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                push    offset aWb_0    ; "wb"
.text:005414A7                push    ecx             ; Filename
.text:005414A8                call    _fopen
.text:005414AD                mov     ebp, eax
.text:005414AF                push    ebp             ; File
.text:005414B0                push    1               ; Count
.text:005414B2                push    edi             ; Size
.text:005414B3                push    esi             ; Str
```

```
.text:005414B4                 call    _fwrite
.text:005414B9                 push    ebp             ; File
.text:005414BA                 call    _fclose
.text:005414BF                 push    ebx             ; Memory
.text:005414C0                 call    _free
.text:005414C5                 add     esp, 2Ch
.text:005414C8                 pop     edi
.text:005414C9                 pop     esi
.text:005414CA                 pop     ebp
.text:005414CB                 pop     ebx
.text:005414CC                 retn
.text:005414CC decrypt_file    endp
```

Все имена функций и меток даны мною в процессе анализа.

Я начал с самого верха. Вот функция, берущая на вход два имени файла и пароль.

```
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename, int password)
.text:00541320 crypt_file      proc near
.text:00541320
.text:00541320 Str             = dword ptr  4
.text:00541320 Filename        = dword ptr  8
.text:00541320 password        = dword ptr  0Ch
.text:00541320
```

Открыть файл и сообщить об ошибке в случае ошибки:

```
.text:00541320                 mov     eax, [esp+Str]
.text:00541324                 push    ebp
.text:00541325                 push    offset Mode     ; "rb"
.text:0054132A                 push    eax             ; Filename
.text:0054132B                 call    _fopen          ; open file
.text:00541330                 mov     ebp, eax
.text:00541332                 add     esp, 8
.text:00541335                 test    ebp, ebp
.text:00541337                 jnz     short loc_541348
.text:00541339                 push    offset Format   ; "Cannot open input file!\n"
.text:0054133E                 call    _printf
.text:00541343                 add     esp, 4
.text:00541346                 pop     ebp
.text:00541347                 retn
.text:00541348
.text:00541348 loc_541348:
```

Узнать размер файла используя `fseek()`/`ftell()`:

```
.text:00541348 push    ebx
.text:00541349 push    esi
.text:0054134A push    edi
.text:0054134B push    2               ; Origin
.text:0054134D push    0               ; Offset
.text:0054134F push    ebp             ; File

; переместить текущую позицию файла на конец
.text:00541350 call    _fseek
.text:00541355 push    ebp             ; File
.text:00541356 call    _ftell          ; узнать текущую позицию
.text:0054135B push    0               ; Origin
.text:0054135D push    0               ; Offset
.text:0054135F push    ebp             ; File
.text:00541360 mov     [esp+2Ch+Str], eax

; переместить текущую позицию файла на начало
.text:00541364 call    _fseek
```

Этот фрагмент кода вычисляет длину файла, выровненную по 64-байтной границе. Это потому что этот алгоритм шифрования работает только с блоками размерами 64 байта. Работает очень просто: разделить длину файла на 64, забыть об остатке, прибавить 1, умножить на 64. Следующий код удаляет остаток от деления, как если бы это значение уже было разделено на 64 и добавляет 64. Это почти то же самое.

```
.text:00541369 mov     esi, [esp+2Ch+Str]

; сбросить в ноль младшие 6 бит
.text:0054136D and     esi, 0FFFFFFC0h

; выровнять размер по 64-байтной границе
.text:00541370 add     esi, 40h
```

Выделить буфер с выровненным размером:

```
.text:00541373                 push    esi             ; Size
.text:00541374                 call    _malloc
```

Вызвать memset(), т.е., очистить выделенный буфер[2].

```
.text:00541379 mov     ecx, esi
.text:0054137B mov     ebx, eax        ; указатель на выделенный буфер -> to EBX
.text:0054137D mov     edx, ecx
.text:0054137F xor     eax, eax
.text:00541381 mov     edi, ebx
.text:00541383 push    ebp             ; File
.text:00541384 shr     ecx, 2
.text:00541387 rep stosd
.text:00541389 mov     ecx, edx
.text:0054138B push    1               ; Count
.text:0054138D and     ecx, 3
.text:00541390 rep stosb               ; memset (buffer, 0, выровненный_размер)
```

Чтение файла используя стандартную функцию Си `fread()`.

```
.text:00541392                 mov     eax, [esp+38h+Str]
.text:00541396                 push    eax             ; ElementSize
.text:00541397                 push    ebx             ; DstBuf
.text:00541398                 call    _fread          ; read file
.text:0054139D                 push    ebp             ; File
.text:0054139E                 call    _fclose
```

Вызов `crypt()`. Эта функция берет на вход буфер, длину буфера (выровненную) и строку пароля.

```
.text:005413A3                 mov     ecx, [esp+44h+password]
.text:005413A7                 push    ecx             ; password
.text:005413A8                 push    esi             ; aligned size
.text:005413A9                 push    ebx             ; buffer
.text:005413AA                 call    crypt           ; do crypt
```

Создать выходной файл. Кстати, разработчик забыл вставить проверку, создался ли файл успешно! Результат открытия файла, впрочем, проверяется.

```
.text:005413AF                 mov     edx, [esp+50h+Filename]
.text:005413B3                 add     esp, 40h
.text:005413B6                 push    offset aWb      ; "wb"
.text:005413BB                 push    edx             ; Filename
.text:005413BC                 call    _fopen
.text:005413C1                 mov     edi, eax
```

Теперь хэндл созданного файла в регистре `EDI`. Записываем сигнатуру "QR9".

```
.text:005413C3                 push    edi             ; File
.text:005413C4                 push    1               ; Count
```

[2] malloc() + memset() можно было бы заменить на calloc()

```
.text:005413C6                 push    3               ; Size
.text:005413C8                 push    offset aQr9     ; "QR9"
.text:005413CD                 call    _fwrite         ; write file signature
```

Записываем настоящую длину файла (не выровненную):

```
.text:005413D2                 push    edi             ; File
.text:005413D3                 push    1               ; Count
.text:005413D5                 lea     eax, [esp+30h+Str]
.text:005413D9                 push    4               ; Size
.text:005413DB                 push    eax             ; Str
.text:005413DC                 call    _fwrite         ; write original file size
```

Записываем шифрованный буфер:

```
.text:005413E1                 push    edi             ; File
.text:005413E2                 push    1               ; Count
.text:005413E4                 push    esi             ; Size
.text:005413E5                 push    ebx             ; Str
.text:005413E6                 call    _fwrite         ; write encrypted file
```

Закрыть файл и освободить выделенный буфер:

```
.text:005413EB                 push    edi             ; File
.text:005413EC                 call    _fclose
.text:005413F1                 push    ebx             ; Memory
.text:005413F2                 call    _free
.text:005413F7                 add     esp, 40h
.text:005413FA                 pop     edi
.text:005413FB                 pop     esi
.text:005413FC                 pop     ebx
.text:005413FD                 pop     ebp
.text:005413FE                 retn
.text:005413FE crypt_file      endp
```

Переписанный на Си код:

```
void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);
```

```
        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);
};
```

Процедура дешифрования почти такая же:

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, void *Src)
.text:00541400 decrypt_file    proc near
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
.text:0054142A                 pop     esi
.text:0054142B                 pop     ebp
.text:0054142C                 pop     ebx
.text:0054142D                 retn
.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E                 push    2               ; Origin
.text:00541430                 push    0               ; Offset
.text:00541432                 push    esi             ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi             ; File
.text:00541439                 call    _ftell
.text:0054143E                 push    0               ; Origin
.text:00541440                 push    0               ; Offset
.text:00541442                 push    esi             ; File
.text:00541443                 mov     ebp, eax
.text:00541445                 call    _fseek
.text:0054144A                 push    ebp             ; Size
.text:0054144B                 call    _malloc
.text:00541450                 push    esi             ; File
.text:00541451                 mov     ebx, eax
.text:00541453                 push    1               ; Count
.text:00541455                 push    ebp             ; ElementSize
.text:00541456                 push    ebx             ; DstBuf
```

```
.text:00541457                call    _fread
.text:0054145C                push    esi             ; File
.text:0054145D                call    _fclose
```

Проверяем сигнатуру (первые 3 байта):

```
.text:00541462                add     esp, 34h
.text:00541465                mov     ecx, 3
.text:0054146A                mov     edi, offset aQr9_0 ; "QR9"
.text:0054146F                mov     esi, ebx
.text:00541471                xor     edx, edx
.text:00541473                repe cmpsb
.text:00541475                jz      short loc_541489
```

Сообщить об ошибке если сигнатура отсутствует:

```
.text:00541477                push    offset aFileIsNotCrypt ; "File is not crypted!\n"
.text:0054147C                call    _printf
.text:00541481                add     esp, 4
.text:00541484                pop     edi
.text:00541485                pop     esi
.text:00541486                pop     ebp
.text:00541487                pop     ebx
.text:00541488                retn
.text:00541489
.text:00541489 loc_541489:
```

Вызвать `decrypt()`.

```
.text:00541489                mov     eax, [esp+10h+Src]
.text:0054148D                mov     edi, [ebx+3]
.text:00541490                add     ebp, 0FFFFFFF9h
.text:00541493                lea     esi, [ebx+7]
.text:00541496                push    eax             ; Src
.text:00541497                push    ebp             ; int
.text:00541498                push    esi             ; int
.text:00541499                call    decrypt
.text:0054149E                mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                push    offset aWb_0    ; "wb"
.text:005414A7                push    ecx             ; Filename
.text:005414A8                call    _fopen
.text:005414AD                mov     ebp, eax
.text:005414AF                push    ebp             ; File
.text:005414B0                push    1               ; Count
.text:005414B2                push    edi             ; Size
.text:005414B3                push    esi             ; Str
.text:005414B4                call    _fwrite
.text:005414B9                push    ebp             ; File
.text:005414BA                call    _fclose
.text:005414BF                push    ebx             ; Memory
.text:005414C0                call    _free
.text:005414C5                add     esp, 2Ch
.text:005414C8                pop     edi
.text:005414C9                pop     esi
.text:005414CA                pop     ebp
.text:005414CB                pop     ebx
.text:005414CC                retn
.text:005414CC decrypt_file   endp
```

Переписанный на Си код:

```
void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
```

```
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};
```

OK, посмотрим глубже.

Функция `crypt()`:

```
.text:00541260 crypt           proc near
.text:00541260
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:
```

Этот фрагмент кода копирует часть входного буфера во внутренний буфер, который я позже назвал "cube64". Длина в регистре `ECX`. `MOVSD` означает *скопировать 32-битное слово*, так что, 16 32-битных слов это как раз 64 байта.

```
.text:0054126A                  mov     eax, [esp+10h+arg_8]
.text:0054126E                  mov     ecx, 10h
.text:00541273                  mov     esi, ebx   ; EBX is pointer within input buffer
.text:00541275                  mov     edi, offset cube64
.text:0054127A                  push    1
.text:0054127C                  push    eax
.text:0054127D                  rep movsd
```

Вызвать `rotate_all_with_password()`:

```
.text:0054127F                  call    rotate_all_with_password
```

Скопировать зашифрованное содержимое из "cube64" назад в буфер:

```
.text:00541284                  mov     eax, [esp+18h+arg_4]
.text:00541288                  mov     edi, ebx
.text:0054128A                  add     ebp, 40h
.text:0054128D                  add     esp, 8
.text:00541290                  mov     ecx, 10h
.text:00541295                  mov     esi, offset cube64
.text:0054129A                  add     ebx, 40h  ; add 64 to input buffer pointer
.text:0054129D                  cmp     ebp, eax  ; EBP contain amount of crypted data.
.text:0054129F                  rep movsd
```

Если `EBP` не больше чем длина во входном аргументе, тогда переходим к следующему блоку.

```
.text:005412A1                  jl      short loc_54126A
.text:005412A3                  pop     edi
.text:005412A4                  pop     esi
.text:005412A5                  pop     ebp
.text:005412A6                  pop     ebx
.text:005412A7                  retn
.text:005412A7 crypt            endp
```

Реконструированная функция `crypt()`:

```
void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};
```

OK, углубимся в функцию `rotate_all_with_password()`. Она берет на вход два аргумента: строку пароля и число. В функции `crypt()`, число 1 используется и в `decrypt()` (где `rotate_all_with_password()` функция вызывается также), число 3.

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0           = dword ptr  4
.text:005411B0 arg_4           = dword ptr  8
.text:005411B0
.text:005411B0                  mov     eax, [esp+arg_0]
.text:005411B4                  push    ebp
.text:005411B5                  mov     ebp, eax
```

Проверяем символы в пароле. Если это ноль, выходим:

```
.text:005411B7                 cmp     byte ptr [eax], 0
.text:005411BA                 jz      exit
.text:005411C0                 push    ebx
.text:005411C1                 mov     ebx, [esp+8+arg_4]
.text:005411C5                 push    esi
.text:005411C6                 push    edi
.text:005411C7
.text:005411C7 loop_begin:
```

Вызываем `tolower()`, стандартную функцию Си.

```
.text:005411C7                 movsx   eax, byte ptr [ebp+0]
.text:005411CB                 push    eax             ; C
.text:005411CC                 call    _tolower
.text:005411D1                 add     esp, 4
```

Хмм, если пароль содержит символ не из латинского алфавита, он пропускается! Действительно, если мы запускаем утилиту для шифрования используя символы не латинского алфавита, похоже, они просто игнорируются.

```
.text:005411D4                 cmp     al, 'a'
.text:005411D6                 jl      short next_character_in_password
.text:005411D8                 cmp     al, 'z'
.text:005411DA                 jg      short next_character_in_password
.text:005411DC                 movsx   ecx, al
```

Отнимем значение “a” (97) от символа.

```
.text:005411DF                 sub     ecx, 'a'  ; 97
```

После вычитания, тут будет 0 для “a”, 1 для “b”, и так далее. И 25 для “z”.

```
.text:005411E2                 cmp     ecx, 24
.text:005411E5                 jle     short skip_subtracting
.text:005411E7                 sub     ecx, 24
```

Похоже, символы “y” и “z” также исключительные. После этого фрагмента кода, “y” становится 0, а “z” — 1. Это значит, что 26 латинских букв становятся значениями в интервале 0..23, (всего 24).

```
.text:005411EA
.text:005411EA skip_subtracting:                       ; CODE XREF: rotate_all_with_password+35
```

Это, на самом деле, деление через умножение. Читайте об этом больше в секции “Деление на 9” (14).

Это код, на самом деле, делит значение символа пароля на 3.

```
.text:005411EA                 mov     eax, 55555556h
.text:005411EF                 imul    ecx
.text:005411F1                 mov     eax, edx
.text:005411F3                 shr     eax, 1Fh
.text:005411F6                 add     edx, eax
.text:005411F8                 mov     eax, ecx
.text:005411FA                 mov     esi, edx
.text:005411FC                 mov     ecx, 3
.text:00541201                 cdq
.text:00541202                 idiv    ecx
```

`EDX` — остаток от деления.

```
.text:00541204 sub     edx, 0
.text:00541207 jz      short call_rotate1 ; если остаток 0, перейти к rotate1
.text:00541209 dec     edx
.text:0054120A jz      short call_rotate2 ; .. если он 1, перейти к rotate2
.text:0054120C dec     edx
.text:0054120D jnz     short next_character_in_password
.text:0054120F test    ebx, ebx
```

```
.text:00541211 jle     short next_character_in_password
.text:00541213 mov     edi, ebx
```

Если остаток 2, вызываем `rotate3()`. `EDX` это второй аргумент функции `rotate_all_with_password()`. Как я уже писал, 1 это для шифрования, 3 для дешифрования. Так что здесь цикл, функции rotate1/2/3 будут вызываться столько же раз, сколько значение переменной в первом аргументе.

```
.text:00541215 call_rotate3:
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short next_character_in_password
.text:00541237
.text:00541237 call_rotate1:
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
.text:00541246                 dec     edi
.text:00541247                 jnz     short loc_54123D
.text:00541249
```

Достать следующий символ из строки пароля.

```
.text:00541249 next_character_in_password:
.text:00541249                 mov     al, [ebp+1]
```

Инкремент указателя на символ в строке пароля:

```
.text:0054124C                 inc     ebp
.text:0054124D                 test    al, al
.text:0054124F                 jnz     loop_begin
.text:00541255                 pop     edi
.text:00541256                 pop     esi
.text:00541257                 pop     ebx
.text:00541258
.text:00541258 exit:
.text:00541258                 pop     ebp
.text:00541259                 retn
.text:00541259 rotate_all_with_password endp
```

Реконструированный код на Си:

```
void rotate_all (char *pwd, int v)
{
```

```
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
                        };
                };

                p++;
        };
};
```

Углубимся еще дальше и исследуем функции rotate1/2/3. Каждая функция вызывает еще две. В итоге я назвал их `set_bit()` и `get_bit()`.

Начнем с `get_bit()`:

```
.text:00541050 get_bit         proc near
.text:00541050
.text:00541050 arg_0           = dword ptr  4
.text:00541050 arg_4           = dword ptr  8
.text:00541050 arg_8           = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr     al, cl
.text:00541065                 and     al, 1
.text:00541067                 retn
.text:00541067 get_bit         endp
```

... иными словами: подсчитать индекс в массиве cube64: *arg_4 + arg_0 * 8*. Затем сдвинуть байт из массива вправо на количество бит заданных в arg_8. Изолировать самый младший бит и вернуть его

Посмотрим другую функцию, `set_bit()`:

```
.text:00541000 set_bit         proc near
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
```

```
.text:00541009                  mov     esi, [esp+4+arg_0]
.text:0054100D                  test    al, al
.text:0054100F                  mov     eax, [esp+4+arg_4]
.text:00541013                  mov     dl, 1
.text:00541015                  jz      short loc_54102B
```

DL тут равно 1. Сдвигаем эту единицу на количество указанное в arg_8. Например, если в arg_8 число 4, тогда значение в DL станет 0x10 или 1000 в двоичной системе счисления.

```
.text:00541017                  shl     dl, cl
.text:00541019                  mov     cl, cube64[eax+esi*8]
```

Вытащить бит из массива и явно выставить его.

```
.text:00541020                  or      cl, dl
```

Сохранить его назад:

```
.text:00541022                  mov     cube64[eax+esi*8], cl
.text:00541029                  pop     esi
.text:0054102A                  retn
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B                  shl     dl, cl
```

Если arg_C не ноль...

```
.text:0054102D                  mov     cl, cube64[eax+esi*8]
```

...инвертировать DL. Например, если состояние DL после сдвига 0x10 или 1000 в двоичной системе, здесь будет 0xEF после инструкции NOT или 11101111 в двоичной системе.

```
.text:00541034                  not     dl
```

Эта инструкция сбрасывает бит, иными словами, она сохраняет все биты в CL которые также выставлены в DL кроме тех в DL, что были сброшены. Это значит, что если в DL, например, 11101111 в двоичной системе, все биты будут сохранены кроме пятого (считая с младшего бита).

```
.text:00541036                  and     cl, dl
```

Сохранить его назад

```
.text:00541038                  mov     cube64[eax+esi*8], cl
.text:0054103F                  pop     esi
.text:00541040                  retn
.text:00541040 set_bit          endp
```

Это почти то же самое что и get_bit(), кроме того, что если arg_C ноль, тогда функция сбрасывает указанный бит в массиве, либо же, в противном случае, выставляет его в 1.

Мы также знаем что размер массива 64. Первые два аргумента и у set_bit() и у get_bit() могут быть представлены как двумерные координаты. Таким образом, массив — это матрица 8*8.

Представление на Си всего того, что мы уже знаем:

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

char cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<shift);
        else
                REMOVE_BIT (cube[x][y], 1<<shift);
};
```

```
int get_bit (int x, int y, int shift)
{
        if ((cube[x][y]>>shift)&1==1)
                return 1;
        return 0;
};
```

Теперь вернемся к функциям rotate1/2/3.

```
.text:00541070 rotate1         proc near
.text:00541070
```

Выделение внутреннего массива размером 64 байта в локальном стеке:

```
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0           = dword ptr  4
.text:00541070
.text:00541070                 sub     esp, 40h
.text:00541073                 push    ebx
.text:00541074                 push    ebp
.text:00541075                 mov     ebp, [esp+48h+arg_0]
.text:00541079                 push    esi
.text:0054107A                 push    edi
.text:0054107B                 xor     edi, edi        ; EDI is loop1 counter
```

EBX указывает на внутренний массив

```
.text:0054107D                 lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

Здесь два вложенных цикла:

```
.text:00541081 first_loop1_begin:
.text:00541081    xor     esi, esi        ; ESI это счетчик второго цикла
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083    push    ebp             ; arg_0
.text:00541084    push    esi             ; счетчик первого цикла
.text:00541085    push    edi             ; счетчик второго цикла
.text:00541086    call    get_bit
.text:0054108B    add     esp, 0Ch
.text:0054108E    mov     [ebx+esi], al   ; записываем во внутренний массив
.text:00541091    inc     esi             ; инкремент счетчика первого цикла
.text:00541092    cmp     esi, 8
.text:00541095    jl      short first_loop2_begin
.text:00541097    inc     edi             ; инкремент счетчика второго цикла

; инкремент указателя во внутреннем массиве на 8 на каждой итерации первого цикла
.text:00541098    add     ebx, 8
.text:0054109B    cmp     edi, 8
.text:0054109E    jl      short first_loop1_begin
```

Мы видим, что оба счетчика циклов в интервале 0..7. Также, они используются как первый и второй аргумент `get_bit()`. Третий аргумент `get_bit()` это единственный аргумент `rotate1()`. То что возвращает `get_bit()` будет сохранено во внутреннем массиве.

Снова приготовить указатель на внутренний массив:

```
.text:005410A0    lea     ebx, [esp+50h+internal_array_64]
.text:005410A4    mov     edi, 7          ; EDI здесь счетчик первого цикла, значение на старте↙
    ↳  - 7
.text:005410A9
.text:005410A9 second_loop1_begin:
.text:005410A9    xor     esi, esi        ; ESI - счетчик второго цикла
```

```
.text:005410AB
.text:005410AB second_loop2_begin:
.text:005410AB    mov     al, [ebx+esi]   ; значение из внутреннего массива
.text:005410AE    push    eax
.text:005410AF    push    ebp             ; arg_0
.text:005410B0    push    edi             ; счетчик первого цикла
.text:005410B1    push    esi             ; счетчик второго цикла
.text:005410B2    call    set_bit
.text:005410B7    add     esp, 10h
.text:005410BA    inc     esi             ; инкремент счетчика второго цикла
.text:005410BB    cmp     esi, 8
.text:005410BE    jl      short second_loop2_begin
.text:005410C0    dec     edi             ; декремент счетчика первого цикла
.text:005410C1    add     ebx, 8          ; инкремент указателя во внутреннем массиве
.text:005410C4    cmp     edi, 0FFFFFFFFh
.text:005410C7    jg      short second_loop1_begin
.text:005410C9    pop     edi
.text:005410CA    pop     esi
.text:005410CB    pop     ebp
.text:005410CC    pop     ebx
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0 rotate1         endp
```

...этот код помещает содержимое из внутреннего массива в глобальный массив cube используя функцию `set_bit()`, *но*, в обратном порядке! Теперь счетчик первого цикла в интервале 7 до 0, уменьшается на 1 на каждой итерации!

Представление кода на Си выглядит так:

```
void rotate1 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, j, v);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (j, 7-i, v, tmp[x][y]);
};
```

Не очень понятно, но если мы посмотрим в функцию `rotate2()`:

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr  4
.text:005410E0
.text:005410E0     sub     esp, 40h
.text:005410E3     push    ebx
.text:005410E4     push    ebp
.text:005410E5     mov     ebp, [esp+48h+arg_0]
.text:005410E9     push    esi
.text:005410EA     push    edi
.text:005410EB     xor     edi, edi        ; счетчик первого цикла
.text:005410ED     lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor     esi, esi        ; счетчик второго цикла
.text:005410F3
.text:005410F3 loc_5410F3:
```

```
.text:005410F3     push    esi             ; счетчик второго цикла
.text:005410F4     push    edi             ; счетчик первого цикла
.text:005410F5     push    ebp             ; arg_0
.text:005410F6     call    get_bit
.text:005410FB     add     esp, 0Ch
.text:005410FE     mov     [ebx+esi], al   ; записать во внутренний массив
.text:00541101     inc     esi             ; инкремент счетчика первого цикла
.text:00541102     cmp     esi, 8
.text:00541105     jl      short loc_5410F3
.text:00541107     inc     edi             ; инкремент счетчика второго цикла
.text:00541108     add     ebx, 8
.text:0054110B     cmp     edi, 8
.text:0054110E     jl      short loc_5410F1
.text:00541110     lea     ebx, [esp+50h+internal_array_64]
.text:00541114     mov     edi, 7          ; первоначальное значение счетчика первого цикла - 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor     esi, esi        ; счетчик второго цикла
.text:0054111B
.text:0054111B loc_54111B:
.text:0054111B     mov     al, [ebx+esi]   ; взять байт из внутреннего массива
.text:0054111E     push    eax
.text:0054111F     push    edi             ; счетчик первого цикла
.text:00541120     push    esi             ; счетчик второго цикла
.text:00541121     push    ebp             ; arg_0
.text:00541122     call    set_bit
.text:00541127     add     esp, 10h
.text:0054112A     inc     esi             ; инкремент счетчика первого цикла
.text:0054112B     cmp     esi, 8
.text:0054112E     jl      short loc_54111B
.text:00541130     dec     edi             ; декремент счетчика второго цикла
.text:00541131     add     ebx, 8
.text:00541134     cmp     edi, 0FFFFFFFFh
.text:00541137     jg      short loc_541119
.text:00541139     pop     edi
.text:0054113A     pop     esi
.text:0054113B     pop     ebp
.text:0054113C     pop     ebx
.text:0054113D     add     esp, 40h
.text:00541140     retn
.text:00541140 rotate2 endp
```

Почти то же самое, за исключением иного порядка аргументов в `get_bit()` и `set_bit()`. Перепишем это на Си-подобный код:

```
void rotate2 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (v, i, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (v, j, 7-i, tmp[i][j]);
};
```

Перепишем так же функцию `rotate3()`:

```
void rotate3 (int v)
{
```

```
        bool tmp[8][8];
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, v, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (7-j, v, i, tmp[i][j]);
};
```

Теперь всё проще. Если мы представим cube64 как трехмерный куб 8*8*8, где каждый элемент это бит, то `get_bit()` и `set_bit()` просто берут на вход координаты бита.

Функции rotate1/2/3 просто поворачивают все биты на определенной плоскости. Три функции, каждая на каждую сторону куба и аргумент `v` выставляет плоскость в интервале 0..7

Может быть, автор алгоритма думал о кубике Рубика 8*8*8 [3]?!

Да, действительно.

Рассмотрим функцию `decrypt()`, я переписал её:

```
void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};
```

Почти то же самое что и crypt(), *но* строка пароля разворачивается стандартной функцией Си strrev() [4] и `rotate_all()` вызывается с аргументом 3.

Это значит, что, в случае дешифровки, rotate1/2/3 будут вызываться трижды.

Это почти кубик Рубика! Если вы хотите вернуть его состояние назад, делайте то же самое в обратном порядке и направлении! Чтобы вернуть эффект от поворота плоскости по часовой стрелке, нужно повернуть её же против часовой стрелки трижды.

`rotate1()`, вероятно, поворот "лицевой" плоскости. `rotate2()`, вероятно, поворот "верхней" плоскости. `rotate3()`, вероятно, поворот "левой" плоскости.

Вернемся к ядру функции `rotate_all()`

```
q=c-'a';
if (q>24)
        q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; // left
```

[3] http://en.wikipedia.org/wiki/Rubik's_Cube

[4] http://msdn.microsoft.com/en-us/library/9hby7w40(VS.80).aspx

```
};
```

Так понять проще: каждый символ пароля определяет сторону (одну из трех) и плоскость (одну из восьми). 3*8 = 24, вот почему два последних символа латинского алфавита переопределяются так чтобы алфавит состоял из 24-х элементов.

Алгоритм очевидно слаб: в случае коротких паролей, в бинарном редакторе файлов можно будет увидеть, что в зашифрованных файлах остались незашифрованные символы.

Весь исходный код в реконструированном виде:

```
#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<z);
        else
                REMOVE_BIT (cube[x][y], 1<<z);
};

bool get_bit (int x, int y, int z)
{
        if ((cube[x][y]>>z)&1==1)
                return true;
        return false;
};

void rotate_f (int row)
{
        bool tmp[8][8];
        int x, y;

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        tmp[x][y]=get_bit (x, y, row);

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
        bool tmp[8][8];
        int y, z;

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        tmp[y][z]=get_bit (row, y, z);

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        set_bit (row, z, 7-y, tmp[y][z]);
};
```

```
void rotate_l (int row)
{
        bool tmp[8][8];
        int x, z;

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        tmp[x][z]=get_bit (x, row, z);

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 (quotient); break;
                        };
                };

                p++;
        };
};

void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};
```

```
void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);

};

void decrypt_file(char *fin, char* fout, char *pw)
{
```

```
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};

// run: input output 0/1 password
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
        if (argc!=5)
        {
                printf ("Incorrect parameters!\n");
                return 1;
        };

        if (strcmp (argv[3], "0")==0)
                crypt_file (argv[1], argv[2], argv[4]);
        else
                if (strcmp (argv[3], "1")==0)
                        decrypt_file (argv[1], argv[2], argv[4]);
                else
                        printf ("Wrong param %s\n", argv[3]);

        return 0;
```

```
};
```

Глава 57

SAP

57.1 Касательно сжимания сетевого траффика в клиенте SAP

(Трассировка связи между переменной окружения TDW_NOCOMPRESS SAPGUI[1] до “назойливого всплывающего окна” и самой функции сжатия данных.)

Известно, что сетевой траффик между SAPGUI и SAP по умолчанию не шифруется а сжимается (читайте здесь[2] и здесь[3]).

Известно также что если установить переменную окружения *TDW_NOCOMPRESS* в 1, можно выключить сжатие сетевых пакетов.

Но вы увидите окно, которое нельзя будет закрыть:

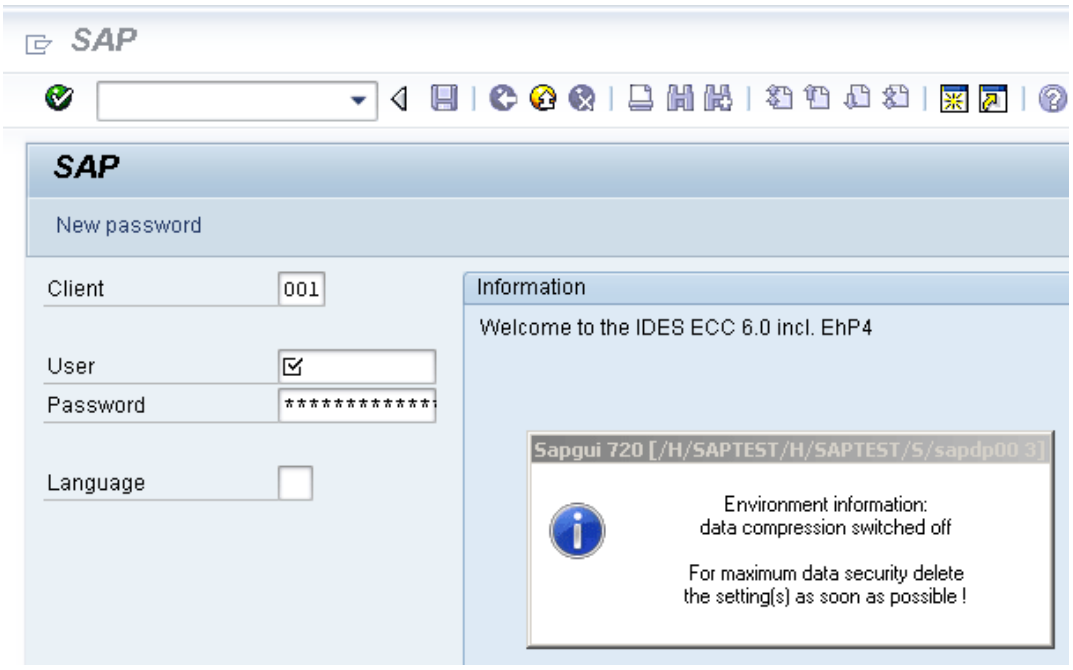


Рис. 57.1: Скриншот

Посмотрим, сможем ли мы как-то убрать это окно.

Но в начале давайте посмотрим, что мы уже знаем. Первое: мы знаем, что переменна окружения *TDW_NOCOMPRESS* проверяется где-то внутри клиента SAPGUI. Второе: строка вроде “data compression switched off” также должна где-то присутствовать. При помощи файлового менеджера FAR я нашел обе эти строки в файле SAPguilib.dll.

Так что давайте откроем файл SAPguilib.dll в IDA и поищем там строку *“TDW_NOCOMPRESS”*. Да, она присутствует и имеется только одна ссылка на эту строку.

Мы увидим такой фрагмент кода (все смещения верны для версии SAPGUI 720 win32, SAPguilib.dll версия файла 7200,1,0,9009):

[1] GUI-клиент от SAP

[2] http://blog.yurichev.com/node/44

[3] http://blog.yurichev.com/node/47

```
.text:6440D51B                lea     eax, [ebp+2108h+var_211C]
.text:6440D51E                push    eax             ; int
.text:6440D51F                push    offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524                mov     byte ptr [edi+15h], 0
.text:6440D528                call    chk_env
.text:6440D52D                pop     ecx
.text:6440D52E                pop     ecx
.text:6440D52F                push    offset byte_64443AF8
.text:6440D534                lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537                call    ds:mfc90_1603
.text:6440D53D                test    eax, eax
.text:6440D53F                jz      short loc_6440D55A
.text:6440D541                lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544                call    ds:mfc90_910
.text:6440D54A                push    eax             ; Str
.text:6440D54B                call    ds:atoi
.text:6440D551                test    eax, eax
.text:6440D553                setnz   al
.text:6440D556                pop     ecx
.text:6440D557                mov     [edi+15h], al
```

Строка возвращаемая функцией `chk_env()` через второй аргумент, обрабатывается далее строковыми функциями MFC, затем вызывается `atoi()`[4]. После этого, число сохраняется в `edi+15h`.

Обратите также внимание на функцию `chk_env` (это я так назвал её):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.text:64413F20 chk_env         proc near
.text:64413F20
.text:64413F20 DstSize         = dword ptr -0Ch
.text:64413F20 var_8           = dword ptr -8
.text:64413F20 DstBuf          = dword ptr -4
.text:64413F20 VarName         = dword ptr  8
.text:64413F20 arg_4           = dword ptr  0Ch
.text:64413F20
.text:64413F20                 push    ebp
.text:64413F21                 mov     ebp, esp
.text:64413F23                 sub     esp, 0Ch
.text:64413F26                 mov     [ebp+DstSize], 0
.text:64413F2D                 mov     [ebp+DstBuf], 0
.text:64413F34                 push    offset unk_6444C88C
.text:64413F39                 mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C                 call    ds:mfc90_820
.text:64413F42                 mov     eax, [ebp+VarName]
.text:64413F45                 push    eax             ; VarName
.text:64413F46                 mov     ecx, [ebp+DstSize]
.text:64413F49                 push    ecx             ; DstSize
.text:64413F4A                 mov     edx, [ebp+DstBuf]
.text:64413F4D                 push    edx             ; DstBuf
.text:64413F4E                 lea     eax, [ebp+DstSize]
.text:64413F51                 push    eax             ; ReturnSize
.text:64413F52                 call    ds:getenv_s
.text:64413F58                 add     esp, 10h
.text:64413F5B                 mov     [ebp+var_8], eax
.text:64413F5E                 cmp     [ebp+var_8], 0
```

[4]Стандартная функция Си, конвертирующая число в строке в число

```
.text:64413F62                 jz      short loc_64413F68
.text:64413F64                 xor     eax, eax
.text:64413F66                 jmp     short loc_64413FBC
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68                 cmp     [ebp+DstSize], 0
.text:64413F6C                 jnz     short loc_64413F72
.text:64413F6E                 xor     eax, eax
.text:64413F70                 jmp     short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72                 mov     ecx, [ebp+DstSize]
.text:64413F75                 push    ecx
.text:64413F76                 mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79                 call    ds:mfc90_2691
.text:64413F7F                 mov     [ebp+DstBuf], eax
.text:64413F82                 mov     edx, [ebp+VarName]
.text:64413F85                 push    edx             ; VarName
.text:64413F86                 mov     eax, [ebp+DstSize]
.text:64413F89                 push    eax             ; DstSize
.text:64413F8A                 mov     ecx, [ebp+DstBuf]
.text:64413F8D                 push    ecx             ; DstBuf
.text:64413F8E                 lea     edx, [ebp+DstSize]
.text:64413F91                 push    edx             ; ReturnSize
.text:64413F92                 call    ds:getenv_s
.text:64413F98                 add     esp, 10h
.text:64413F9B                 mov     [ebp+var_8], eax
.text:64413F9E                 push    0FFFFFFFFh
.text:64413FA0                 mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3                 call    ds:mfc90_5835
.text:64413FA9                 cmp     [ebp+var_8], 0
.text:64413FAD                 jz      short loc_64413FB3
.text:64413FAF                 xor     eax, eax
.text:64413FB1                 jmp     short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3                 mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64413FB6                 call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC                 mov     esp, ebp
.text:64413FBE                 pop     ebp
.text:64413FBF                 retn
.text:64413FBF chk_env         endp
```

Да. Функция `getenv_s()`[5] это *безопасная* версия функции `getenv()`[6] в MSVC.

Тут также имеются манипуляции со строками при помощи функций из MFC.

Множество других переменных окружения также проверяются. Здесь список всех переменных проверяемых SAPGUI а также сообщение записываемое им в лог-файл, если переменная включена:

[5] http://msdn.microsoft.com/en-us/library/tb2sfw2z(VS.80).aspx

[6] Стандартная функция Си, возвращающая значение переменной окружения

DPTRACE	“GUI-OPTION: Trace set to %d”
TDW_HEXDUMP	“GUI-OPTION: Hexdump enabled”
TDW_WORKDIR	“GUI-OPTION: working directory ‘%s’”
TDW_SPLASHSRCEENOFF	“GUI-OPTION: Splash Screen Off” / “GUI-OPTION: Splash Screen On”
TDW_REPLYTIMEOUT	“GUI-OPTION: reply timeout %d milliseconds”
TDW_PLAYBACKTIMEOUT	“GUI-OPTION: PlaybackTimeout set to %d milliseconds”
TDW_NOCOMPRESS	“GUI-OPTION: no compression read”
TDW_EXPERT	“GUI-OPTION: expert mode”
TDW_PLAYBACKPROGRESS	“GUI-OPTION: PlaybackProgress”
TDW_PLAYBACKNETTRAFFIC	“GUI-OPTION: PlaybackNetTraffic”
TDW_PLAYLOG	“GUI-OPTION: /PlayLog is YES, file %s”
TDW_PLAYTIME	“GUI-OPTION: /PlayTime set to %d milliseconds”
TDW_LOGFILE	“GUI-OPTION: TDW_LOGFILE ‘%s’”
TDW_WAN	“GUI-OPTION: WAN - low speed connection enabled”
TDW_FULLMENU	“GUI-OPTION: FullMenu enabled”
SAP_CP / SAP_CODEPAGE	“GUI-OPTION: SAP_CODEPAGE ‘%d’”
UPDOWNLOAD_CP	“GUI-OPTION: UPDOWNLOAD_CP ‘%d’”
SNC_PARTNERNAME	“GUI-OPTION: SNC name ‘%s’”
SNC_QOP	“GUI-OPTION: SNC_QOP ‘%s’”
SNC_LIB	“GUI-OPTION: SNC is set to: %s”
SAPGUI_INPLACE	“GUI-OPTION: environment variable SAPGUI_INPLACE is on”

Настройки для каждой переменной записываются в массив через указатель в регистре `EDI`. `EDI` выставляется перед вызовом функции:

```
.text:6440EE00                lea     edi, [ebp+2884h+var_2884] ; options here like +0x15...
.text:6440EE03                lea     ecx, [esi+24h]
.text:6440EE06                call    load_command_line
.text:6440EE0B                mov     edi, eax
.text:6440EE0D                xor     ebx, ebx
.text:6440EE0F                cmp     edi, ebx
.text:6440EE11                jz      short loc_6440EE42
.text:6440EE13                push    edi
.text:6440EE14                push    offset aSapguiStoppedA ; "Sapgui stopped after ↙
    ↳ commandline interp"...
.text:6440EE19                push    dword_644F93E8
.text:6440EE1F                call    FEWTraceError
```

А теперь, можем ли мы найти строку *“data record mode switched on”*? Да, и есть только одна ссылка на эту строку в функции `CDwsGui::PrepareInfoWindow()`. Откуда я узнал имена классов/методов? Здесь много специальных отладочных вызовов, пишущих в лог-файл вроде:

```
.text:64405160                push    dword ptr [esi+2854h]
.text:64405166                push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: ↙
    ↳ sapgui env"...
.text:6440516B                push    dword ptr [esi+2848h]
.text:64405171                call    dbg
.text:64405176                add     esp, 0Ch
```

... или:

```
.text:6440237A                push    eax
.text:6440237B                push    offset aCclientStart_6 ; "CClient::Start: set shortcut ↙
    ↳ user to '\%"...
.text:64402380                push    dword ptr [edi+4]
.text:64402383                call    dbg
.text:64402388                add     esp, 0Ch
```

Они **очень** полезны.

Посмотрим содержимое функции “назойливого всплывающего окна”:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam         = byte ptr -3Ch
```

```
.text:64404F4F var_38          = dword ptr -38h
.text:64404F4F var_34          = dword ptr -34h
.text:64404F4F rc              = tagRECT ptr -2Ch
.text:64404F4F cy              = dword ptr -1Ch
.text:64404F4F h               = dword ptr -18h
.text:64404F4F var_14          = dword ptr -14h
.text:64404F4F var_10          = dword ptr -10h
.text:64404F4F var_4           = dword ptr -4
.text:64404F4F
.text:64404F4F                 push    30h
.text:64404F51                 mov     eax, offset loc_64438E00
.text:64404F56                 call    __EH_prolog3
.text:64404F5B                 mov     esi, ecx        ; ECX is pointer to object
.text:64404F5D                 xor     ebx, ebx
.text:64404F5F                 lea     ecx, [ebp+var_14]
.text:64404F62                 mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65                 call    ds:mfc90_316
.text:64404F6B                 mov     [ebp+var_4], ebx
.text:64404F6E                 lea     edi, [esi+2854h]
.text:64404F74                 push    offset aEnvironmentInf ; "Environment information:\n"
.text:64404F79                 mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B                 call    ds:mfc90_820
.text:64404F81                 cmp     [esi+38h], ebx
.text:64404F84                 mov     ebx, ds:mfc90_2539
.text:64404F8A                 jbe     short loc_64404FA9
.text:64404F8C                 push    dword ptr [esi+34h]
.text:64404F8F                 lea     eax, [ebp+var_14]
.text:64404F92                 push    offset aWorkingDirecto ; "working directory: '\%s'\n"
.text:64404F97                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98                 call    ebx ; mfc90_2539
.text:64404F9A                 add     esp, 0Ch
.text:64404F9D                 lea     eax, [ebp+var_14]
.text:64404FA0                 push    eax
.text:64404FA1                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
.text:64404FA3                 call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9                 mov     eax, [esi+38h]
.text:64404FAC                 test    eax, eax
.text:64404FAE                 jbe     short loc_64404FD3
.text:64404FB0                 push    eax
.text:64404FB1                 lea     eax, [ebp+var_14]
.text:64404FB4                 push    offset aTraceLevelDAct ; "trace level \%d activated\n"
.text:64404FB9                 push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA                 call    ebx ; mfc90_2539
.text:64404FBC                 add     esp, 0Ch
.text:64404FBF                 lea     eax, [ebp+var_14]
.text:64404FC2                 push    eax
.text:64404FC3                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)
```

```
.text:64404FC5                call    ds:mfc90_941
.text:64404FCB                xor     ebx, ebx
.text:64404FCD                inc     ebx
.text:64404FCE                mov     [ebp+var_10], ebx
.text:64404FD1                jmp     short loc_64404FD6
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3                xor     ebx, ebx
.text:64404FD5                inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6                cmp     [esi+38h], ebx
.text:64404FD9                jbe     short loc_64404FF1
.text:64404FDB                cmp     dword ptr [esi+2978h], 0
.text:64404FE2                jz      short loc_64404FF1
.text:64404FE4                push    offset aHexdumpInTrace ; "hexdump in trace activated\n"
.text:64404FE9                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB                call    ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1                cmp     byte ptr [esi+78h], 0
.text:64404FF5                jz      short loc_64405007
.text:64404FF7                push    offset aLoggingActivat ; "logging activated\n"
.text:64404FFC                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE                call    ds:mfc90_945
.text:64405004                mov     [ebp+var_10], ebx
.text:64405007
.text:64405007 loc_64405007:
.text:64405007                cmp     byte ptr [esi+3Dh], 0
.text:6440500B                jz      short bypass
.text:6440500D                push    offset aDataCompressio ; "data compression switched off\↙
    ↳ n"
.text:64405012                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                call    ds:mfc90_945
.text:6440501A                mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D                mov     eax, [esi+20h]
.text:64405020                test    eax, eax
.text:64405022                jz      short loc_6440503A
.text:64405024                cmp     dword ptr [eax+28h], 0
.text:64405028                jz      short loc_6440503A
.text:6440502A                push    offset aDataRecordMode ; "data record mode switched on\n↙
    ↳ "
.text:6440502F                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031                call    ds:mfc90_945
.text:64405037                mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A                mov     ecx, edi
.text:6440503C                cmp     [ebp+var_10], ebx
```

```
.text:6440503F                  jnz     loc_64405142
.text:64405045                  push    offset aForMaximumData ; "\nFor maximum data security ↙
    ↳ delete\nthe s"...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A                  call    ds:mfc90_945
.text:64405050                  xor     edi, edi
.text:64405052                  push    edi             ; fWinIni
.text:64405053                  lea     eax, [ebp+pvParam]
.text:64405056                  push    eax             ; pvParam
.text:64405057                  push    edi             ; uiParam
.text:64405058                  push    30h             ; uiAction
.text:6440505A                  call    ds:SystemParametersInfoA
.text:64405060                  mov     eax, [ebp+var_34]
.text:64405063                  cmp     eax, 1600
.text:64405068                  jle     short loc_64405072
.text:6440506A                  cdq
.text:6440506B                  sub     eax, edx
.text:6440506D                  sar     eax, 1
.text:6440506F                  mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072                  push    edi             ; hWnd
.text:64405073                  mov     [ebp+cy], 0A0h
.text:6440507A                  call    ds:GetDC
.text:64405080                  mov     [ebp+var_10], eax
.text:64405083                  mov     ebx, 12Ch
.text:64405088                  cmp     eax, edi
.text:6440508A                  jz      loc_64405113
.text:64405090                  push    11h             ; i
.text:64405092                  call    ds:GetStockObject
.text:64405098                  mov     edi, ds:SelectObject
.text:6440509E                  push    eax             ; h
.text:6440509F                  push    [ebp+var_10]    ; hdc
.text:644050A2                  call    edi ; SelectObject
.text:644050A4                  and     [ebp+rc.left], 0
.text:644050A8                  and     [ebp+rc.top], 0
.text:644050AC                  mov     [ebp+h], eax
.text:644050AF                  push    401h            ; format
.text:644050B4                  lea     eax, [ebp+rc]
.text:644050B7                  push    eax             ; lprc
.text:644050B8                  lea     ecx, [esi+2854h]
.text:644050BE                  mov     [ebp+rc.right], ebx
.text:644050C1                  mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8                  call    ds:mfc90_3178
.text:644050CE                  push    eax             ; cchText
.text:644050CF                  lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:644050D5                  call    ds:mfc90_910
.text:644050DB                  push    eax             ; lpchText
.text:644050DC                  push    [ebp+var_10]    ; hdc
.text:644050DF                  call    ds:DrawTextA
.text:644050E5                  push    4               ; nIndex
.text:644050E7                  call    ds:GetSystemMetrics
.text:644050ED                  mov     ecx, [ebp+rc.bottom]
.text:644050F0                  sub     ecx, [ebp+rc.top]
.text:644050F3                  cmp     [ebp+h], 0
.text:644050F7                  lea     eax, [eax+ecx+28h]
```

```
.text:644050FB                  mov     [ebp+cy], eax
.text:644050FE                  jz      short loc_64405108
.text:64405100                  push    [ebp+h]         ; h
.text:64405103                  push    [ebp+var_10]    ; hdc
.text:64405106                  call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108                  push    [ebp+var_10]    ; hDC
.text:6440510B                  push    0               ; hWnd
.text:6440510D                  call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113                  mov     eax, [ebp+var_38]
.text:64405116                  push    80h             ; uFlags
.text:6440511B                  push    [ebp+cy]        ; cy
.text:6440511E                  inc     eax
.text:6440511F                  push    ebx             ; cx
.text:64405120                  push    eax             ; Y
.text:64405121                  mov     eax, [ebp+var_34]
.text:64405124                  add     eax, 0FFFFFED4h
.text:64405129                  cdq
.text:6440512A                  sub     eax, edx
.text:6440512C                  sar     eax, 1
.text:6440512E                  push    eax             ; X
.text:6440512F                  push    0               ; hWndInsertAfter
.text:64405131                  push    dword ptr [esi+285Ch] ; hWnd
.text:64405137                  call    ds:SetWindowPos
.text:6440513D                  xor     ebx, ebx
.text:6440513F                  inc     ebx
.text:64405140                  jmp     short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
.text:64405142                  push    offset byte_64443AF8

; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147                  call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D                  cmp     dword_6450B970, ebx
.text:64405153                  jl      short loc_64405188
.text:64405155                  call    sub_6441C910
.text:6440515A                  mov     dword_644F858C, ebx
.text:64405160                  push    dword ptr [esi+2854h]
.text:64405166                  push    offset aCdwsguiPrepare ; "\nCDwsGui::PrepareInfoWindow: ↙
    ↳ sapgui env"...
.text:6440516B                  push    dword ptr [esi+2848h]
.text:64405171                  call    dbg
.text:64405176                  add     esp, 0Ch
.text:64405179                  mov     dword_644F858C, 2
.text:64405183                  call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188                  or      [ebp+var_4], 0FFFFFFFFh
.text:6440518C                  lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F                  call    ds:mfc90_601
.text:64405195                  call    __EH_epilog3
.text:6440519A                  retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

ECX в начале функции содержит в себе указатель на объект (потому что это тип функции thiscall (29.1.1)). В нашем случае, класс имеет тип, очевидно, *CDwsGui*. В зависимости от включенных опций в объекте, разные сообщения добавляются к итоговому сообщению.

Если переменная по адресу this+0x3D не ноль, компрессия сетевых пакетов будет выключена:

```
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ; "data compression switched off\↙
    ↳ n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
```

Интересно, что в итоге, состояние переменной *var_10* определяет, будет ли показано сообщение вообще:

```
.text:6440503C                 cmp     [ebp+var_10], ebx
.text:6440503F                 jnz     exit ; bypass drawing

; добавляет строки "For maximum data security delete" / "the setting(s) as soon as possible !":

.text:64405045                 push    offset aForMaximumData ; "\nFor maximum data security ↙
    ↳ delete\nthe s"...
.text:6440504A                 call    ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)
.text:64405050                 xor     edi, edi
.text:64405052                 push    edi             ; fWinIni
.text:64405053                 lea     eax, [ebp+pvParam]
.text:64405056                 push    eax             ; pvParam
.text:64405057                 push    edi             ; uiParam
.text:64405058                 push    30h             ; uiAction
.text:6440505A                 call    ds:SystemParametersInfoA
.text:64405060                 mov     eax, [ebp+var_34]
.text:64405063                 cmp     eax, 1600
.text:64405068                 jle     short loc_64405072
.text:6440506A                 cdq
.text:6440506B                 sub     eax, edx
.text:6440506D                 sar     eax, 1
.text:6440506F                 mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:

начинает рисовать:

.text:64405072                 push    edi             ; hWnd
.text:64405073                 mov     [ebp+cy], 0A0h
.text:6440507A                 call    ds:GetDC
```

Давайте проверим нашу теорию на практике.

JNZ в этой строке ...

```
.text:6440503F                 jnz     exit ; пропустить отрисовку
```

...заменим просто на JMP и получим SAPGUI работающим без этого назойливого всплывающего окна!

Копнем немного глубже и проследим связь между смещением 0x15 в load_command_line() (Это я дал имя этой функции) и переменной this+0x3D в *CDwsGui::PrepareInfoWindow*. Уверены ли мы что это одна и та же переменная?

Начинаю искать все места где в коде используется константа 0x15. Для таких небольших программ как SAPGUI, это иногда срабатывает. Вот первое что я нашел:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19
.text:64404C19 arg_0           = dword ptr  4
.text:64404C19
.text:64404C19                 push    ebx
.text:64404C1A                 push    ebp
.text:64404C1B                 push    esi
.text:64404C1C                 push    edi
.text:64404C1D                 mov     edi, [esp+10h+arg_0]
.text:64404C21                 mov     eax, [edi]
.text:64404C23                 mov     esi, ecx ; ESI/ECX are pointers to some unknown object.
.text:64404C25                 mov     [esi], eax
.text:64404C27                 mov     eax, [edi+4]
.text:64404C2A                 mov     [esi+4], eax
.text:64404C2D                 mov     eax, [edi+8]
.text:64404C30                 mov     [esi+8], eax
.text:64404C33                 lea     eax, [edi+0Ch]
.text:64404C36                 push    eax
.text:64404C37                 lea     ecx, [esi+0Ch]

; demangled name:  ATL::CStringT::operator=(class ATL::CStringT ... &)
.text:64404C3A                 call    ds:mfc90_817
.text:64404C40                 mov     eax, [edi+10h]
.text:64404C43                 mov     [esi+10h], eax
.text:64404C46                 mov     al, [edi+14h]
.text:64404C49                 mov     [esi+14h], al
.text:64404C4C                 mov     al, [edi+15h] ; copy byte from 0x15 offset
.text:64404C4F                 mov     [esi+15h], al ; to 0x15 offset in CDwsGui object
```

Эта функция вызывается из функции с названием *CDwsGui::CopyOptions*! И снова спасибо отладочной информации.

Но настоящий ответ находится в функции *CDwsGui::Init()*:

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                 mov     eax, [ebp+arg_0]
.text:6440B0C2                 push    [ebp+arg_4]
.text:6440B0C5                 mov     [esi+2844h], eax
.text:6440B0CB                 lea     eax, [esi+28h] ; ESI is pointer to CDwsGui object
.text:6440B0CE                 push    eax
.text:6440B0CF                 call    CDwsGui__CopyOptions
```

Теперь ясно: массив заполняемый в `load_command_line()` на самом деле расположен в классе *CDwsGui* но по адресу `this+0x28`. 0x15 + 0x28 это 0x3D. OK, мы нашли место, куда наша переменная копируется.

Посмотрим также и другие места, где используется смещение 0x3D. Одно из таких мест находится в функции *CDwsGui::SapguiRun* (и снова спасибо отладочным вызовам):

```
.text:64409D58                 cmp     [esi+3Dh], bl   ; ESI is pointer to CDwsGui object
.text:64409D5B                 lea     ecx, [esi+2B8h]
.text:64409D61                 setz    al
.text:64409D64                 push    eax             ; arg_10 of CConnectionContext::↙
    ↳ CreateNetwork
.text:64409D65                 push    dword ptr [esi+64h]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D68                 call    ds:mfc90_910
.text:64409D68                                         ; no arguments
.text:64409D6E                 push    eax
.text:64409D6F                 lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:64409D75                 call    ds:mfc90_910
.text:64409D75                                         ; no arguments
.text:64409D7B                 push    eax
```

```
.text:64409D7C                 push    esi
.text:64409D7D                 lea     ecx, [esi+8]
.text:64409D80                 call    CConnectionContext__CreateNetwork
```

Проверим нашу идею. Заменяем `setz al` здесь на `xor eax, eax / nop`, убираем переменную окружения TDW_NOCOMPRESS и запускаем SAPGUI. Wow! Назойливого окна больше нет (как и ожидалось: ведь переменной окружении также нет), но в Wireshark мы видим, что сетевые пакеты больше не сжимаются! Очевидно, это то самое место где флаг отражающий сжатие пакетов выставляется в объекте `CConnectionContext`.

Так что, флаг сжатия передается в пятом аргументе функции *CConnectionContext::CreateNetwork*. Внутри этой функции, вызывается еще одна:

```
...
.text:64403476                 push    [ebp+compression]
.text:64403479                 push    [ebp+arg_C]
.text:6440347C                 push    [ebp+arg_8]
.text:6440347F                 push    [ebp+arg_4]
.text:64403482                 push    [ebp+arg_0]
.text:64403485                 call    CNetwork__CNetwork
```

Флаг отвечающий за сжатие здесь передается в пятом аргументе для конструктора *CNetwork::CNetwork*.

И вот как конструктор *CNetwork* выставляет некоторые флаги в объекте *CNetwork* в соответствии с пятым аргументом **и** еще какую-то переменную, возможно, также отвечающую за сжатие сетевых пакетов.

```
.text:64411DF1                 cmp     [ebp+compression], esi
.text:64411DF7                 jz      short set_EAX_to_0
.text:64411DF9                 mov     al, [ebx+78h]   ; another value may affect compression?
.text:64411DFC                 cmp     al, '3'
.text:64411DFE                 jz      short set_EAX_to_1
.text:64411E00                 cmp     al, '4'
.text:64411E02                 jnz     short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
.text:64411E04                 xor     eax, eax
.text:64411E06                 inc     eax             ; EAX -> 1
.text:64411E07                 jmp     short loc_64411E0B
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09
.text:64411E09                 xor     eax, eax        ; EAX -> 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B                 mov     [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

Теперь мы знаем, что флаг отражающий сжатие данных сохраняется в классе *CNetwork* по адресу *this+0x3A4*.

Поищем теперь значение `0x3A4` в SAPguilib.dll. Находим второе упоминание этого значения в функции *CDwsGui::OnClientMessageWrite* (бесконечная благодарность отладочной информации):

```
.text:64406F76 loc_64406F76:
.text:64406F76                 mov     ecx, [ebp+7728h+var_7794]
.text:64406F79                 cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                 jnz     compression_flag_is_zero
.text:64406F86                 mov     byte ptr [ebx+7], 1
.text:64406F8A                 mov     eax, [esi+18h]
.text:64406F8D                 mov     ecx, eax
.text:64406F8F                 test    eax, eax
.text:64406F91                 ja      short loc_64406FFF
.text:64406F93                 mov     ecx, [esi+14h]
.text:64406F96                 mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99                 push    dword ptr [edi+2868h] ; int
.text:64406F9F                 lea     edx, [ebp+7728h+var_77A4]
.text:64406FA2                 push    edx             ; int
```

```
.text:64406FA3                  push    30000           ; int
.text:64406FA8                  lea     edx, [ebp+7728h+Dst]
.text:64406FAB                  push    edx             ; Dst
.text:64406FAC                  push    ecx             ; int
.text:64406FAD                  push    eax             ; Src
.text:64406FAE                  push    dword ptr [edi+28C0h] ; int
.text:64406FB4                  call    sub_644055C5        ; actual compression routine
.text:64406FB9                  add     esp, 1Ch
.text:64406FBC                  cmp     eax, 0FFFFFFF6h
.text:64406FBF                  jz      short loc_64407004
.text:64406FC1                  cmp     eax, 1
.text:64406FC4                  jz      loc_6440708C
.text:64406FCA                  cmp     eax, 2
.text:64406FCD                  jz      short loc_64407004
.text:64406FCF                  push    eax
.text:64406FD0                  push    offset aCompressionErr ; "compression error [rc = \%d]- ↙
    ↳ program wi"...
.text:64406FD5                  push    offset aGui_err_compre ; "GUI_ERR_COMPRESS"
.text:64406FDA                  push    dword ptr [edi+28D0h]
.text:64406FE0                  call    SapPcTxtRead
```

Заглянем в функцию *sub_644055C5*. Всё что в ней мы находим это вызов memcpy() и еще какую-то функцию, названную IDA *sub_64417440*.

И теперь заглянем в *sub_64417440*. Увидим там:

```
.text:6441747C                  push    offset aErrorCsrcompre ; "\nERROR: CsRCompress: invalid ↙
    ↳ handle"
.text:64417481                  call    eax ; dword_644F94C8
.text:64417483                  add     esp, 4
```

Voilà! Мы находим функцию которая собственно и сжимает сетевые пакеты. Как я уже разобрался [7], эта функция используется в SAP и в опен-сорсном проекте MaxDB. Так что эта функция доступна в виде исходников.

Последняя проверка:

```
.text:64406F79                  cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                  jnz     compression_flag_is_zero
```

Заменим JNZ на безусловный переход JMP. Уберем переменную окружения TDW_NOCOMPRESS. Voilà! В Wireshark мы видим, что сетевые пакеты, исходящие от клиента, не сжаты. Ответы сервера, впрочем, сжаты.

Так что мы нашли связь между переменной окружения и местом где функция сжатия данных вызывается, а также может быть отключена.

57.2 Функции проверки пароля в SAP 6.0

Когда я в очередной раз вернулся к своему SAP 6.0 IDES заинсталлированному в виртуальной машине VMware, я обнаружил что забыл пароль, впрочем, затем я вспомнил его, но теперь я получаю такую ошибку: *«Password logon no longer possible - too many failed attempts»*, потому что я потратил все попытки на то, чтобы вспомнить его.

Первая очень хорошая новость состоит в том, что с SAP поставляется полный файл *disp+work.pdb*, он содержит все: имена функций, структуры, типы, локальные переменные, имена аргументов, и т.д. Какой щедрый подарок!

Я нашел утилиту TYPEINFODUMP[8] для дампа содержимого PDB-файлов во что-то более читаемое и grep-абельное.

Вот пример её работы: информация о функции + её аргументах + её локальных переменных:

```
FUNCTION ThVmcSysEvent
  Address:         10143190  Size:      675 bytes  Index:    60483  TypeIndex:    60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
```

[7] http://conus.info/utils/SAP_pkt_decompr.txt
[8] http://www.debuginfo.com/tools/typeinfodump.html

```
PARAMETER events
  Address: Reg335+288  Size:        4 bytes  Index:    60488  TypeIndex:    60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:        1 bytes  Index:    60490  TypeIndex:    60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:        8 bytes  Index:    60492  TypeIndex:    60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:         12274af0  Size:        8 bytes  Index:    60495  TypeIndex:    60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:        8 bytes  Index:    60498  TypeIndex:    60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64  Size:      204 bytes  Index:    60501  TypeIndex:    60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:        4 bytes  Index:    60508  TypeIndex:    60509
  Type: int
Flags: 90
```

А вот пример дампа структуры:

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
  Type:  DBSL_MODULETYPE
  Offset:        0  Index:        3  TypeIndex:    38653
MEMBER module
  Type:  wchar_t module[40]
  Offset:        4  Index:        3  TypeIndex:      831
MEMBER stmtnum
  Type:  long
  Offset:       84  Index:        3  TypeIndex:      440
MEMBER timestamp
  Type:  wchar_t timestamp[15]
  Offset:       88  Index:        3  TypeIndex:     6612
```

Вау!

Вторая хорошая новость: *отладочные* вызовы, коих здесь очень много, очень полезны.

Здесь вы можете увидеть глобальную переменную *ct_level*[9], отражающую уровень трассировки.

В *disp+work.exe* очень много таких отладочных вставок:

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh        ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl     ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
```

[9]Еще об уровне трассировки: http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e10000000a155369/content.htm

```
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

Если текущий уровень трассировки выше или равен заданному в этом коде порогу, отладочное сообщение будет записано в лог-файл вроде *dev_ w0*, *dev_ disp* и прочие файлы *dev**.

Попробуем grep-ать файл полученный при помощи утилиты TYPEINFODUMP:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

Я получил:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor''::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Попробуем так же искать отладочные сообщения содержащие слова *«password»* и *«locked»*. Одна из таких это строка *«user was locked by subsequently failed password logon attempts»* на которую есть ссылка в функции *password_ attempt_ limit_ exceeded()*.

Другие строки, которые эта найденная функция может писать в лог-файл это: *«password logon attempt will be rejected immediately (preventing dictionary attacks)»*, *«failed-logon lock: expired (but not removed due to 'read-only' operation)»*, *«failed-logon lock: expired => removed»*.

Немного поэкспериментировав с этой функцией, я быстро понял что проблема именно в ней. Она вызывается из функции *chckpass()* — одна из функций проверяющих пароль.

В начале, я хочу убедиться, что я на верном пути:

Запускаю свой tracer:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1 ↙
    ↳     ", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

Функции вызываются так: *syssigni() -> DyISigni() -> dychkusr() -> usrexist() -> chckpass()*.

Число 0x35 возвращается из *chckpass()* в этом месте:

```
.text:00000001402ED567 loc_1402ED567:                          ; CODE XREF: chckpass+B4
.text:00000001402ED567                 mov     rcx, rbx        ; usr02
.text:00000001402ED56A                 call    password_idle_check
.text:00000001402ED56F                 cmp     eax, 33h
.text:00000001402ED572                 jz      loc_1402EDB4E
.text:00000001402ED578                 cmp     eax, 36h
.text:00000001402ED57B                 jz      loc_1402EDB3D
.text:00000001402ED581                 xor     edx, edx        ; usr02_readonly
.text:00000001402ED583                 mov     rcx, rbx        ; usr02
.text:00000001402ED586                 call    password_attempt_limit_exceeded
.text:00000001402ED58B                 test    al, al
.text:00000001402ED58D                 jz      short loc_1402ED5A0
.text:00000001402ED58F                 mov     eax, 35h
.text:00000001402ED594                 add     rsp, 60h
.text:00000001402ED598                 pop     r14
.text:00000001402ED59A                 pop     r12
.text:00000001402ED59C                 pop     rdi
.text:00000001402ED59D                 pop     rsi
.text:00000001402ED59E                 pop     rbx
.text:00000001402ED59F                 retn
```

Отлично, давайте проверим:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode,↙
    ↳ rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) ↙
    ↳ (called from 0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (called↙
    ↳  from 0x1402e9794 (disp+work.exe!chngpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Великолепно! Теперь я могу успешно залогиниться.

Кстати, я могу сделать вид что вообще забыл пароль, заставляя *chckpass()* всегда возвращать ноль, и этого достаточно для отключения проверки пароля:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus ↙
    ↳    ", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0
```

Что еще можно сказать бегло анализируя функцию *password_ attempt_ limit_ exceeded()*, это то что в начале можно увидеть следующий вызов:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
```

```
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E19D4
cmp     ax, 'n'
jz      short loc_1402E19D4
cmp     ax, '0'
jnz     short loc_1402E19DE
```

Очевидно, функция *sapgparam()* используется чтобы узнать значение какой-либо переменной конфигурации. Эта функция может вызываться из 1768 разных мест. Вероятно, при помощи этой информации, мы можем легко находить те места кода, на которые влияют определенные переменные конфигурации.

Замечательно! Имена функций очень понятны, куда понятнее чем в Oracle RDBMS. По всей видимости, процесс *disp+work* весь написан на Си++. Вероятно, он был переписан не так давно?

Глава 58

Oracle RDBMS

58.1 Таблица V$VERSION в Oracle RDBMS

Oracle RDBMS 11.2 это очень большая программа, основной модуль `oracle.exe` содержит около 124 тысячи функций. Для сравнения, ядро Windows 7 x64 (ntoskrnl.exe) — около 11 тысяч функций, а ядро Linux 3.9.8 (с драйверами по умолчанию) — 31 тысяч функций.

Начнем с одного простого вопроса. Откуда Oracle RDBMS берет информацию, когда мы в SQL*Plus пишем вот такой вот простой запрос:

```
SQL> select * from V$VERSION;
```

И получаем:

```
BANNER
--------------------------------------------------------------------------------

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Начнем. Где в самом Oracle RDBMS мы можем найти строку `V$VERSION`?

Для win32-версии, эта строка имеется в файле `oracle.exe`, это легко увидеть. Но мы также можем использовать объектные (.o) файлы от версии Oracle RDBMS для Linux, потому что в них сохраняются имена функций и глобальных переменных, а в `oracle.exe` для win32 этого нет.

Итак, строка `V$VERSION` имеется в файле `kqf.o`, в самой главной Oracle-библиотеке `libserver11.a`.

Ссылка на эту текстовую строку имеется в таблице `kqfviw`, размещенной в этом же файле `kqf.o`:

Листинг 58.1: kqf.o

```
.rodata:0800C4A0 kqfviw          dd 0Bh                  ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                                         ; kqfgbn+34
.rodata:0800C4A4                 dd offset _2__STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0800C4A8                 dd 4
.rodata:0800C4AC                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0                 dd 3
.rodata:0800C4B4                 dd 0
.rodata:0800C4B8                 dd 195h
.rodata:0800C4BC                 dd 4
.rodata:0800C4C0                 dd 0
.rodata:0800C4C4                 dd 0FFFFFC1CBh
.rodata:0800C4C8                 dd 3
.rodata:0800C4CC                 dd 0
.rodata:0800C4D0                 dd 0Ah
.rodata:0800C4D4                 dd offset _2__STRING_10104_0 ; "V$WAITSTAT"
.rodata:0800C4D8                 dd 4
.rodata:0800C4DC                 dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4E0                 dd 3
```

```
.rodata:0800C4E4                dd 0
.rodata:0800C4E8                dd 4Eh
.rodata:0800C4EC                dd 3
.rodata:0800C4F0                dd 0
.rodata:0800C4F4                dd 0FFFFC003h
.rodata:0800C4F8                dd 4
.rodata:0800C4FC                dd 0
.rodata:0800C500                dd 5
.rodata:0800C504                dd offset _2__STRING_10105_0 ; "GV$BH"
.rodata:0800C508                dd 4
.rodata:0800C50C                dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C510                dd 3
.rodata:0800C514                dd 0
.rodata:0800C518                dd 269h
.rodata:0800C51C                dd 15h
.rodata:0800C520                dd 0
.rodata:0800C524                dd 0FFFFC1EDh
.rodata:0800C528                dd 8
.rodata:0800C52C                dd 0
.rodata:0800C530                dd 4
.rodata:0800C534                dd offset _2__STRING_10106_0 ; "V$BH"
.rodata:0800C538                dd 4
.rodata:0800C53C                dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C540                dd 3
.rodata:0800C544                dd 0
.rodata:0800C548                dd 0F5h
.rodata:0800C54C                dd 14h
.rodata:0800C550                dd 0
.rodata:0800C554                dd 0FFFFC1EEh
.rodata:0800C558                dd 5
.rodata:0800C55C                dd 0
```

Кстати, нередко, при изучении внутренностей Oracle RDBMS, появляется вопрос, почему имена функций и глобальных переменных такие странные. Вероятно, дело в том, что Oracle RDBMS очень старый продукт сам по себе и писался на Си еще в 1980-х. А в те времена стандарт Си гарантировал поддержку имен переменных длиной только до шести символов включительно: «6 significant initial characters in an external identifier»[1]

Вероятно, таблица `kqfviw` содержащая в себе многие (а может даже и все) view с префиксом V$, это служебные view (fixed views), присутствующие всегда. Бегло оценив цикличность данных, мы легко видим, что в каждом элементе таблицы `kqfviw` 12 полей 32-битных полей. В IDA легко создать структуру из 12-и элементов и применить её ко всем элементам таблицы. Для версии Oracle RDBMS 11.2, здесь 1023 элемента в таблице, то есть, здесь описываются 1023 всех возможных *fixed view*. Позже, мы еще вернемся к этому числу.

Как видно, мы не очень много можем узнать чисел в этих полях. Самое первое число всегда равно длине строки-названия view (без терминирующего ноля). Это справедливо для всех элементов. Но эта информация не очень полезна.

Мы также знаем, что информацию обо всех fixed views можно получить из *fixed view* под названием `V$FIXED_VIEW_DEFIN` (кстати, информация для этого view также берется из таблиц `kqfviw` и `kqfvip`). Кстати, там тоже 1023 элемента.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

Итак, `V$VERSION` это как бы *thunk view* для другого, с названием `GV$VERSION`, который, в свою очередь:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';
```

[1] Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988)

```
VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$VERSION
select inst_id, banner from x$version
```

Таблицы с префиксом X$ в Oracle RDBMS — это также служебные таблицы, они не документированы, не могут изменятся пользователем, и обновляются динамически.

Попробуем поискать текст `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` в файле `kqf.o` и находим ссылку на него в таблице `kqfvip`:

.

Листинг 58.2: kqf.o

```
rodata:080185A0 kqfvip          dd offset _2__STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0                                             ; kqfgvt+F
.rodata:080185A0                                             ; "select inst_id,decode(indx,1,'data ↙
    ↳ bloc"...
.rodata:080185A4                 dd offset kqfv459_c_0
.rodata:080185A8                 dd 0
.rodata:080185AC                 dd 0

...

.rodata:08019570                 dd offset _2__STRING_11378_0 ; "select  BANNER from GV$VERSION↙
    ↳  where in"...
.rodata:08019574                 dd offset kqfv133_c_0
.rodata:08019578                 dd 0
.rodata:0801957C                 dd 0
.rodata:08019580                 dd offset _2__STRING_11379_0 ; "select inst_id,decode(bitand(↙
    ↳ cfflg,1),0"...
.rodata:08019584                 dd offset kqfv403_c_0
.rodata:08019588                 dd 0
.rodata:0801958C                 dd 0
.rodata:08019590                 dd offset _2__STRING_11380_0 ; "select  STATUS , NAME, ↙
    ↳ IS_RECOVERY_DEST"...
.rodata:08019594                 dd offset kqfv199_c_0
```

Таблица, по всей видимости, имеет 4 поля в каждом элементе. Кстати, здесь так же 1023 элемента. Второе поле указывает на другую таблицу, содержащую поля этого *fixed view*. Для `V$VERSION`, эта таблица только из двух элементов, первый это 6 и второй это строка `BANNER` (число это длина строки) и далее *терминирующий* элемент содержащий 0 и *нулевую* Си-строку:

Листинг 58.3: kqf.o

```
.rodata:080BBAC4 kqfv133_c_0     dd 6                    ; DATA XREF: .rodata:08019574
.rodata:080BBAC8                 dd offset _2__STRING_5017_0 ; "BANNER"
.rodata:080BBACC                 dd 0
.rodata:080BBAD0                 dd offset _2__STRING_0_0
```

Объединив данные из таблиц `kqfviw` и `kqfvip`, мы получим SQL-запросы, которые исполняются, когда пользователь хочет получить информацию из какого-либо *fixed view*.

Я написал программу oracle tables[2], которая собирает всю эту информацию из объектных файлов от Oracle RDBMS под Linux. Для `V$VERSION`, мы можем найти следующее:

Листинг 58.4: Результат работы oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where inst_id = USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

[2] http://yurichev.com/oracle_tables.html

и:

Листинг 58.5: Результат работы oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params:
[INST_ID] [BANNER]
```

Fixed view GV$VERSION отличается от V$VERSION тем, что содержит еще и поле отражающее идентификатор *instance*. Но так или иначе, мы теперь упираемся в таблицу X$VERSION. Как и прочие X$-таблицы, она не документирована, однако, мы можем оттуда что-то прочитать:

```
SQL> select * from x$version;

ADDR           INDX    INST_ID
-------- ---------- ----------
BANNER
--------------------------------------------------------------------------------

0DBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production

...
```

Эта таблица содержит дополнительные поля вроде ADDR и INDX.

Бегло листая содержимое файла kqf.o в IDA мы можем увидеть еще одну таблицу где есть ссылка на строку X$VERSION, это kqftab:

Листинг 58.6: kqf.o

```
.rodata:0803CAC0                dd 9                    ; element number 0x1f6
.rodata:0803CAC4                dd offset _2__STRING_13113_0 ; "X$VERSION"
.rodata:0803CAC8                dd 4
.rodata:0803CACC                dd offset _2__STRING_13114_0 ; "kqvt"
.rodata:0803CAD0                dd 4
.rodata:0803CAD4                dd 4
.rodata:0803CAD8                dd 0
.rodata:0803CADC                dd 4
.rodata:0803CAE0                dd 0Ch
.rodata:0803CAE4                dd 0FFFFC075h
.rodata:0803CAE8                dd 3
.rodata:0803CAEC                dd 0
.rodata:0803CAF0                dd 7
.rodata:0803CAF4                dd offset _2__STRING_13115_0 ; "X$KQFSZ"
.rodata:0803CAF8                dd 5
.rodata:0803CAFC                dd offset _2__STRING_13116_0 ; "kqfsz"
.rodata:0803CB00                dd 1
.rodata:0803CB04                dd 38h
.rodata:0803CB08                dd 0
.rodata:0803CB0C                dd 7
.rodata:0803CB10                dd 0
.rodata:0803CB14                dd 0FFFFC09Dh
.rodata:0803CB18                dd 2
.rodata:0803CB1C                dd 0
```

Здесь очень много ссылок на названия X$-таблиц, вероятно, на все те что имеются в Oracle RDBMS этой версии. Но мы снова упираемся в то что не имеем достаточно информации. У меня нет никакой идеи, что означает строка kqvt. Вообще, префикс kq может означать *kernel* и *query*. v, может быть, *version*, а t — *type*? Я не знаю, честно говоря.

Таблицу с очень похожим названием мы можем найти в kqf.o:

Листинг 58.7: kqf.o

```
.rodata:0808C360 kqvt_c_0        kqftap_param <4, offset _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, ↙
    ↳ 0>
.rodata:0808C360                                         ; DATA XREF: .rodata:08042680
.rodata:0808C360                                         ; "ADDR"
.rodata:0808C384                 kqftap_param <4, offset _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0,↙
    ↳  0> ; "INDX"
.rodata:0808C3A8                 kqftap_param <7, offset _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0,↙
    ↳  0> ; "INST_ID"
.rodata:0808C3CC                 kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 50h,↙
    ↳  0, 0> ; "BANNER"
.rodata:0808C3F0                 kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

Она содержит информацию об именах полей в таблице `X$VERSION`. Единственная ссылка на эту таблицу имеется в таблице `kqftap`:

Листинг 58.8: kqf.o

```
.rodata:08042680                 kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0> ; ↙
    ↳ element 0x1f6
```

Интересно что здесь этот элемент проходит так же под номером `0x1f6` (502-й), как и ссылка на строку `X$VERSION` в таблице `kqftab`. Вероятно, таблицы `kqftap` и `kqftab` дополняют друг друга, как и `kqfvip` и `kqfviw`. Мы также видим здесь ссылку на функцию с названием `kqvrow()`. А вот это уже кое-что!

Я сделал так чтобы моя программа oracle tables[3] могла дампить и эти таблицы. Для `X$VERSION` получается:

Листинг 58.9: Результат работы oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

При помощи tracer, можно легко проверить, что эта ф-ция вызывается 6 раз кряду (из ф-ции `qerfxFetch()`) при получении строк из `X$VERSION`.

Запустим tracer в режиме `cc` (он добавит комментарий к каждой исполненной инструкции):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_  proc near

var_7C    = byte ptr -7Ch
var_18    = dword ptr -18h
var_14    = dword ptr -14h
Dest      = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4
arg_8     = dword ptr  10h
arg_C     = dword ptr  14h
arg_14    = dword ptr  1Ch
arg_18    = dword ptr  20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

          push    ebp
          mov     ebp, esp
          sub     esp, 7Ch
          mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
          mov     ecx, TlsIndex   ; [69AEB08h]=0
```

[3] http://yurichev.com/oracle_tables.html

```
          mov     edx, large fs:2Ch
          mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
          cmp     eax, 2           ; EAX=1
          mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
          jz      loc_2CE1288
          mov     ecx, [eax]       ; [EAX]=0..5
          mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
             ; _kqvrow_+1A9
          cmp     ecx, 5           ; ECX=0..5
          ja      loc_56C11C7
          mov     edi, [ebp+arg_18] ; [EBP+20h]=0
          mov     [ebp+var_14], edx ; EDX=0xc98c938
          mov     [ebp+var_8], ebx ; EBX=0
          mov     ebx, eax         ; EAX=0xcdfe554
          mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
          mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db↙
    ↳ , 0x2ce11f6, 0x2ce1236, 0x2ce127a
          jmp     edx              ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, ↙
    ↳ 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
          push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
          mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
          xor     edx, edx
          mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
          push    edx              ; EDX=0
          push    edx              ; EDX=0
          push    50h
          push    ecx              ; ECX=0x8a172b4
          push    dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58
          call    _kghalf          ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          mov     esi, ds:__imp__vsnnum ; [59771A8h]=0x61bc49e0
          mov     [ebp+Dest], eax ; EAX=0xce2ffb0
          mov     [ebx+8], eax     ; EAX=0xce2ffb0
          mov     [ebx+4], eax     ; EAX=0xce2ffb0
          mov     edi, [esi]       ; [ESI]=0xb200100
          mov     esi, ds:__imp__vsnstr ; [597D6D4h]=0x65852148, "- Production"
          push    esi              ; ESI=0x65852148, "- Production"
          mov     ebx, edi         ; EDI=0xb200100
          shr     ebx, 18h         ; EBX=0xb200100
          mov     ecx, edi         ; EDI=0xb200100
          shr     ecx, 14h         ; ECX=0xb200100
          and     ecx, 0Fh         ; ECX=0xb2
          mov     edx, edi         ; EDI=0xb200100
          shr     edx, 0Ch         ; EDX=0xb200100
          movzx   edx, dl          ; DL=0
          mov     eax, edi         ; EDI=0xb200100
          shr     eax, 8           ; EAX=0xb200100
          and     eax, 0Fh         ; EAX=0xb2001
          and     edi, 0FFh        ; EDI=0xb200100
          push    edi              ; EDI=0
          mov     edi, [ebp+arg_18] ; [EBP+20h]=0
          push    eax              ; EAX=1
          mov     eax, ds:__imp__vsnban ; [597D6D8h]=0x65852100, "Oracle Database 11g ↙
    ↳ Enterprise Edition Release %d.%d.%d.%d.%d %s"
          push    edx              ; EDX=0
```

```
          push    ecx             ; ECX=2
          push    ebx             ; EBX=0xb
          mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
          push    eax             ; EAX=0x65852100, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release %d.%d.%d.%d.%d %s"
          mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
          push    eax             ; EAX=0xce2ffb0
          call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf tracing nested maximum level (1) ↙
    ↳ reached, skipping this CALL
          add     esp, 38h
          mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
             ; _kqvrow_+128 ...
          test    edi, edi        ; EDI=0
          jnz     __VInfreq__kqvrow
          mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
          mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
          mov     eax, ebx        ; EBX=0xcdfe554
          mov     ebx, [ebp+var_8] ; [EBP-8]=0
          lea     eax, [eax+4]    ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production↙
    ↳ ", "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production", "PL/SQL ↙
    ↳ Release 11.2.0.1.0 - Production", "TNS for 32-bit Windows: Version 11.2.0.1.0 - ↙
    ↳ Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
          mov     esp, ebp
          pop     ebp
          retn                    ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
          mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0 - Production"
          mov     dword ptr [ebx], 2
          mov     [ebx+4], edx    ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0 - Production"
          push    edx             ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition ↙
    ↳ Release 11.2.0.1.0 - Production"
          call    _kkxvsn         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          pop     ecx
          mov     edx, [ebx+4]    ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
          movzx   ecx, byte ptr [edx] ; [EDX]=0x50
          test    ecx, ecx        ; ECX=0x50
          jnz     short loc_2CE1192
          mov     edx, [ebp+var_14]
          mov     esi, [ebp+var_C]
          mov     eax, ebx
          mov     ebx, [ebp+var_8]
          mov     ecx, [eax]
          jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
          push    0
          push    50h
          mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
          mov     [ebx+4], edx    ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
          push    edx             ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
          call    _lmxver         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          add     esp, 0Ch
```

```
          mov     dword ptr [ebx], 3
          jmp     short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
          mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
          mov     [ebp+var_18], 50h
          mov     [ebx+4], edx    ; EDX=0xce2ffb0
          push    0
          call    _npinli         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          pop     ecx
          test    eax, eax        ; EAX=0
          jnz     loc_56C11DA
          mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
          lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
          push    edx             ; EDX=0xd76c93c
          push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
          push    dword ptr [ecx+13278h] ; [ECX+13278h]=0xacce190
          call    _nrtnsvrs       ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
          mov     dword ptr [ebx], 4
          jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
          lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
          push    edx             ; EDX=0xd76c8d8
          push    0
          mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for 32-bit Windows: Version ↙
    ↳ 11.2.0.1.0 - Production"
          mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↙
    ↳ - Production"
          mov     ecx, 50h
          mov     [ebp+var_18], ecx ; ECX=0x50
          push    ecx             ; ECX=0x50
          push    esi             ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 ↙
    ↳ - Production"
          call    _lxvers         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          add     esp, 10h
          mov     edx, [ebp+var_18] ; [EBP-18h]=0x50
          mov     dword ptr [ebx], 5
          test    edx, edx        ; EDX=0x50
          jnz     loc_2CE1192
          mov     edx, [ebp+var_14]
          mov     esi, [ebp+var_C]
          mov     eax, ebx
          mov     ebx, [ebp+var_8]
          mov     ecx, 5
          jmp     loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
          mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
          mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
          mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
          mov     eax, ebx        ; EBX=0xcdfe554
          mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
```

```
          mov     eax, [eax+8]    ; [EAX+8]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
          test    eax, eax        ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
          jz      short loc_2CE12A7
          push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
          push    eax             ; EAX=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production"
          mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
          push    eax             ; EAX=0x8a172b4
          push    dword ptr [edx+10494h] ; [EDX+10494h]=0xc98cd58
          call    _kghfrf         ; tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
          add     esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
          xor     eax, eax
          mov     esp, ebp
          pop     ebp
          retn                    ; EAX=0
_kqvrow_  endp
```

Так можно легко увидеть, что номер строки таблицы задается извне. Сама ф-ция возвращает строку, формируя её так:

Строка 1	Использует глобальные переменные `vsnstr`, `vsnnum`, `vsnban`. Вызывает `sprintf()`.
Строка 2	Вызывает `kkxvsn()`.
Строка 3	Вызывает `lmxver()`.
Строка 4	Вызывает `npinli()`, `nrtnsvrs()`.
Строка 5	Вызывает `lxvers()`.

Так вызываются соответствующие ф-ции для определения номеров версий отдельных модулей.

58.2 Таблица X$KSMLRU в Oracle RDBMS

В заметке *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* упоминается некая служебная таблица:

> There is a fixed table called X$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.
>
> If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.
>
> One unusual thing about the X$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

Однако, как можно легко убедиться, эта системная таблица очищается всякий раз, когда кто-то делает запрос к ней. Сможем ли мы найти причину, почему это происходит? Если вернуться к уже рассмотренным таблицам `kqftab` и `kqftap` полученных при помощи oracle tables[4], содержащим информацию о X$-таблицах, мы узнаем что для того чтобы подготовить строки этой таблицы, вызывается ф-ция `ksmlrs()`:

Листинг 58.10: Результат работы oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
```

[4] http://yurichev.com/oracle_tables.html

```
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

Действительно, при помощи tracer легко убедиться что эта ф-ция вызывается каждый раз, когда мы обращаемся к таблице `X$KSMLRU`.

Здесь есть ссылки на ф-ции `ksmsplu_sp()` и `ksmsplu_jp()`, каждая из которых в итоге вызывает `ksmsplu()`. В конце ф-ции `ksmsplu()` мы видим вызов `memset()`:

Листинг 58.11: ksm.o

```
...

.text:00434C50 loc_434C50:                             ; DATA XREF: .rdata:off_5E50EA8
.text:00434C50                 mov     edx, [ebp-4]
.text:00434C53                 mov     [eax], esi
.text:00434C55                 mov     esi, [edi]
.text:00434C57                 mov     [eax+4], esi
.text:00434C5A                 mov     [edi], eax
.text:00434C5C                 add     edx, 1
.text:00434C5F                 mov     [ebp-4], edx
.text:00434C62                 jnz     loc_434B7D
.text:00434C68                 mov     ecx, [ebp+14h]
.text:00434C6B                 mov     ebx, [ebp-10h]
.text:00434C6E                 mov     esi, [ebp-0Ch]
.text:00434C71                 mov     edi, [ebp-8]
.text:00434C74                 lea     eax, [ecx+8Ch]
.text:00434C7A                 push    370h            ; Size
.text:00434C7F                 push    0               ; Val
.text:00434C81                 push    eax             ; Dst
.text:00434C82                 call    __intel_fast_memset
.text:00434C87                 add     esp, 0Ch
.text:00434C8A                 mov     esp, ebp
.text:00434C8C                 pop     ebp
.text:00434C8D                 retn
.text:00434C8D _ksmsplu        endp
```

Такие конструкции (`memset (block, 0, size)`) очень часто используются для простого обнуления блока памяти. Мы можем попробовать рискнуть, заблокировав вызов `memset()` и посмотреть, что будет?

Запускаем tracer со следующей опцией: поставить точку останова на `0x434C7A` (там где начинается передача параметров для ф-ции `memset()`) так, чтобы tracer в этом месте установил указатель инструкций процессора (`EIP`) на место, где уже произошла очистка переданных параметров в `memset()` (по адресу `0x434C8A`): Можно сказать, при помощи этого, мы симулируем безусловный переход с адреса `0x434C7A` на `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)
```

(Важно: все эти адреса справедливы только для win32-версии Oracle RDBMS 11.2)

Действительно, после этого мы можем обращаться к таблице `X$KSMLRU` сколько угодно, и она уже не очищается!

~~Не делайте этого дома ("Разрушители легенд")~~ Не делайте этого на своих production-серверах.

Впрочем, это не обязательно полезное или желаемое поведение системы, но как эксперимент по поиску нужного кода, нам это подошло!

58.3 Таблица V$TIMER в Oracle RDBMS

V$TIMER это еще один служебный *fixed view*, отражающий какое-то часто меняющееся значение:

> V$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(Из документации Oracle RDBMS [5])

Интересно что периоды разные в Oracle для Win32 и для Linux. Сможем ли мы найти функцию, отвечающую за генерирование этого значения?

Как видно, эта информация, в итоге, берется из системной таблицы X$KSUTM.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------

GV$TIMER
select inst_id,ksutmtim from x$ksutm
```

Здесь мы упираемся в небольшую проблему, в таблицах kqftab/kqftap нет указателей на функцию, которая бы генерировала значение:

Листинг 58.12: Результат работы oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

Попробуем в таком случае просто поискать строку KSUTMTIM, и находим ссылку на нее в такой функции:

```
kqfd_DRN_ksutm_c proc near              ; DATA XREF: .rodata:0805B4E8

arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h

                push    ebp
```

[5] http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm

```
                mov     ebp, esp
                push    [ebp+arg_C]
                push    offset ksugtm
                push    offset _2__STRING_1263_0 ; "KSUTMTIM"
                push    [ebp+arg_8]
                push    [ebp+arg_0]
                call    kqfd_cfui_drain
                add     esp, 14h
                mov     esp, ebp
                pop     ebp
                retn
kqfd_DRN_ksutm_c endp
```

Сама ф-ция `kqfd_DRN_ksutm_c()` упоминается в таблице `kqfd_tab_registry_0` вот так:

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
dd 0
dd offset kqfd_DRN_ksutm_c
```

Упоминается также некая ф-ция `ksugtm()`. Посмотрим, что там (в Linux x86):

Листинг 58.13: ksu.o

```
ksugtm          proc near

var_1C          = byte ptr -1Ch
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 1Ch
                lea     eax, [ebp+var_1C]
                push    eax
                call    slgcs
                pop     ecx
                mov     edx, [ebp+arg_4]
                mov     [edx], eax
                mov     eax, 4
                mov     esp, ebp
                pop     ebp
                retn
ksugtm          endp
```

В win32-версии тоже самое.

Искомая ли эта функция? Попробуем узнать:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Пробуем несколько раз:

```
SQL> select * from V$TIMER;

     HSECS
----------
  27294929

SQL> select * from V$TIMER;

     HSECS
----------
  27295006
```

```
SQL> select * from V$TIMER;

     HSECS
----------
  27295167
```

Листинг 58.14: вывод tracer

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch↙
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                     ".|..            "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch↙
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                                     ".}..            "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch↙
    ↳ +0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.              "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                     ".}..            "
```

Действительно — значение то, что мы видим в SQL*Plus, и оно возвращается через второй аргумент.

Посмотрим, что в ф-ции `slgcs()` (Linux x86):

```
slgcs           proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    esi
                mov     [ebp+var_4], ebx
                mov     eax, [ebp+arg_0]
                call    $+5
                pop     ebx
                nop                     ; PIC mode
                mov     ebx, offset _GLOBAL_OFFSET_TABLE_
                mov     dword ptr [eax], 0
                call    sltrgatime64    ; PIC mode
                push    0
                push    0Ah
                push    edx
                push    eax
                call    __udivdi3       ; PIC mode
                mov     ebx, [ebp+var_4]
                add     esp, 10h
                mov     esp, ebp
                pop     ebp
                retn
slgcs           endp
```

(это просто вызов `sltrgatime64()` и деление его результата на 10 (14))

И в win32-версии:

```
_slgcs          proc near               ; CODE XREF: _dbgefgHtElResetCount+15
                                        ; _dbgerRunActions+1528
                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+8]
                mov     dword ptr [eax], 0
                call    ds:__imp__GetTickCount@0 ; GetTickCount()
                mov     edx, eax
                mov     eax, 0CCCCCCCDh
                mul     edx
                shr     edx, 3
                mov     eax, edx
                mov     esp, ebp
                pop     ebp
                retn
_slgcs          endp
```

Это просто результат `GetTickCount()` [6] поделенный на 10 (14).

Вуаля! Вот почему в win32-версии и версии Linux x86 разные результаты, потому что они получаются разными системными ф-циями ОС.

Drain по-английски дренаж, отток, водосток. Таким образом, возможно имеется ввиду *подключение* определенного столбца системной таблице к функции.

Я добавил поддержку таблицы `kqfd_tab_registry_0` в oracle tables[7], теперь мы можем видеть, при помощи каких ф-ций, столбцы в системных таблицах *подключаются* к значениям, например:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

OPN, возможно, *open*, а *DRN*, вероятно, означает *drain*.

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx
[7] http://yurichev.com/oracle_tables.html

Глава 59

Вручную написанный на ассемблере код

59.1 Тестовый файл EICAR

Этот .COM-файл предназначен для тестирования антивирусов, его можно запустить в MS-DOS и он выведет такую строку: "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!" [1].

Он примечателен тем, что он полностью состоит только из печатных ASCII-символов, следовательно, его можно набрать в любом текстовом редакторе:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Попробуем его разобрать:

```
; изначальное состояние: SP=0FFFEh, SS:[SP]=0
0100 58            pop     ax
; AX=0, SP=0
0101 35 4F 21      xor     ax, 214Fh
; AX = 214Fh и SP = 0
0104 50            push    ax
; AX = 214Fh, SP = FFFEh и SS:[FFFE] = 214Fh
0105 25 40 41      and     ax, 4140h
; AX = 140h, SP = FFFEh и SS:[FFFE] = 214Fh
0108 50            push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h и SS:[FFFE] = 214Fh
0109 5B            pop     bx
; AX = 140h, BX = 140h, SP = FFFEh и SS:[FFFE] = 214Fh
010A 34 5C         xor     al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh и SS:[FFFE] = 214Fh
010C 50            push    ax
010D 5A            pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFFE] = 214Fh
010E 58            pop     ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28      xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50            push    ax
0113 5E            pop     si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37         sub     [bx], si
0116 43            inc     bx
0117 43            inc     bx
0118 29 37         sub     [bx], si
011A 7D 24         jge     short near ptr word_10140
011C 45 49 43 ...  db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B   word_10140  dw 2B48h  ; CD 21 (INT 21) будет здесь
0142 48 2A               dw 2A48h  ; CD 20 (INT 20) будет здесь
0144 0D                  db  0Dh
0145 0A                  db  0Ah
```

[1] https://ru.wikipedia.org/wiki/EICAR-Test-File{https://en.wikipedia.org/wiki/EICAR_test_file}

Я добавил везде комментарии, показывающие состояние регистров и стека после каждой инструкции.

Собственно, все эти инструкции нужны только для того чтобы исполнить следующий код:

```
B4 09     MOV AH, 9
BA 1C 01  MOV DX, 11Ch
CD 21     INT 21h
CD 20     INT 20h
```

`INT 21h` с ф-цией 9 (переданной в `AH`) просто выводит строку, адрес которой передан в `DS:DX`. Кстати, строка должна быть завершена символом '$'. Вероятно, это наследие CP/M и эта ф-ция в DOS осталась для совместимости. `INT 20h` возвращает управление в DOS.

Но, как видно, далеко не все опкоды этих инструкций печатные. Так что основная часть EICAR-файла это:

- подготовка нужных значений регистров (AH и DX);
- подготовка в памяти опкодов для INT 21 и INT 20;
- исполнение INT 21 и INT 20.

Кстати, подобная техника широко используется для создания шеллкодов, где нужно создать x86-код, который будет нужно передать в виде текстовой строки.

Здесь также список всех x86-инструкций с печатаемыми опкодоами: B.6.6.

Глава 60

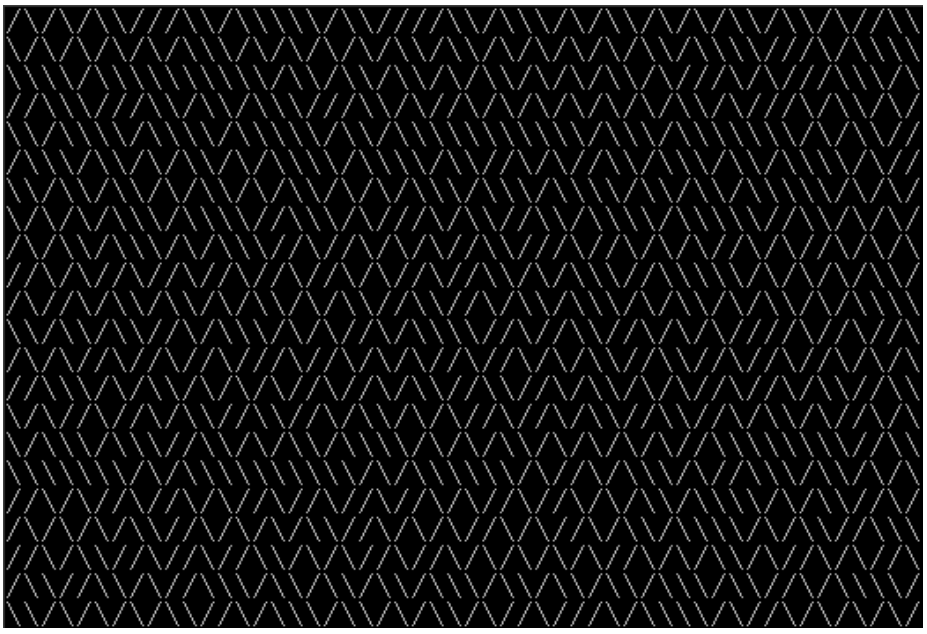
Демо

Демо (или демомейкинг) были великолепным упражнением в математике, программировании компьютерной графики и очень плотному программированию на ассемблере вручную.

60.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10

Все примеры здесь для .COM-файлов под MS-DOS.

В [9] можно прочитать об одном из простейших генераторов случайных лабиринтов. Он просто бесконечно и случайно печатает символ слэша или обратный слэша, выдавая в итоге что-то вроде:



Здесь несколько известных реализаций для 16-битного x86.

60.1.1 Версия 42-х байт от Trixter

Листинг взят с его сайта[1], но комментарии — мои.

```
00000000: B001        mov         al,1        ; установить видеорежим 40x25
00000002: CD10        int         010
00000004: 30FF        xor         bh,bh       ; установить видеостраницу для вызова int 10h
00000006: B9D007      mov         cx,007D0    ; вывод 2000 символов
00000009: 31C0        xor         ax,ax
0000000B: 9C          pushf                   ; сохранить флаги
; узнать случайное число из чипа таймера
0000000C: FA          cli                     ; запретить прерывания
0000000D: E643        out         043,al      ; записать 0 в порт 43h
```

[1] http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/

```
; прочитать 16-битное значение из порта 40h
0000000F: E440          in          al,040
00000011: 88C4          mov         ah,al
00000013: E440          in          al,040
00000015: 9D            popf                    ; разрешить прерывания возвращая значение флага ↙
    ↳ IF
00000016: 86C4          xchg        ah,al
; здесь мы имеем псевдослучайное 16-битное значение
00000018: D1E8          shr         ax,1
0000001A: D1E8          shr         ax,1
; в CF сейчас находится второй бит из значения
0000001C: B05C          mov         al,05C ;'\'
; если CF=1, пропускаем следующую инструкцию
0000001E: 7202          jc          000000022
; если CF=0, загружаем в регистр AL другой символ
00000020: B02F          mov         al,02F ;'/'
; вывод символа
00000022: B40E          mov         ah,00E
00000024: CD10          int         010
00000026: E2E1          loop        000000009 ; цикл в 2000 раз
00000028: CD20          int         020       ; возврат в DOS
```

Псевдослучайное число на самом деле это время прошедшее со старта системы, получаемое из чипа таймера 8253, это значение увеличивается на единицу 18.2 раза в секунду.

Записывая ноль в порт `43h`, мы имеем ввиду что команда это "выбрать счетчик 0 "counter latch "двоичный счетчик"(а не значение BCD[2]).

Прерывания снова разрешаются при помощи инструкции `POPF`, которая также возвращает флаг `IF`.

Инструкцию `IN` нельзя использовать с другими регистрами кроме `AL`, поэтому здесь перетасовка.

60.1.2 Моя попытка укоротить версию Trixter: 27 байт

Мы можем сказать что мы используем таймер не для того чтобы получить точное время, но псевдо-случайное число, так что мы можем не тратить время (и код) на запрещение прерываний. Еще можно сказать что так как мы берем бит из младшей 8-битной части, то мы можем считывать только её.

Я немного укоротил код и вышло 27 байт:

```
00000000: B9D007 mov    cx,007D0 ; вывести только 2000 символов
00000003: 31C0   xor    ax,ax    ; команда чипу таймера
00000005: E643   out    043,al
00000007: E440   in     al,040   ; читать 8 бит из таймера
00000009: D1E8   shr    ax,1     ; переместить второй бит в флаг CF
0000000B: D1E8   shr    ax,1
0000000D: B05C   mov    al,05C   ; подготовить '\'
0000000F: 7202   jc     000000013
00000011: B02F   mov    al,02F   ; подготовить '/'
; вывести символ на экран
00000013: B40E   mov    ah,00E
00000015: CD10   int    010
00000017: E2EA   loop   000000003
; выход в DOS
00000019: CD20   int    020
```

60.1.3 Использование случайного мусора в памяти как источника случайных чисел

Так как это MS-DOS, защиты памяти здесь нет вовсе, так что мы можем читать с какого угодно адреса. И даже более того: простая инструкция `LODSB` будет читать байт по адресу `DS:SI`, но это не проблема если правильные значения не установлены в регистры, пусть она читает 1) случайные байты; 2) из случайного места в памяти!

Так что на странице Trixter-а[3]можно найти предложение использовать `LODSB` без всякой инициализации.

[2]Binary-coded decimal

[3]http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/

Есть также предложение использовать инструкцию SCASB вместо, потому что она выставляет флаги в соответствии с прочитанным значением.

Еще одна идея насчет минимизации кода это использовать прерывание DOS INT 29h которое просто печатает символ на экране из регистра AL.

Это то что сделали Peter Ferrie и Андрей "herm1t" Баранович (11 и 10 байт) [4]:

Листинг 60.1: Андрей "herm1t" Баранович: 11 байт

```
00000000: B05C       mov         al,05C    ;'\'
; читать байт в AL из случайного места в памяти
00000002: AE         scasb
; PF = четность(AL - случайный_байт) = четность(5Ch - случайный_байт)
00000003: 7A02       jp          000000007
00000005: B02F       mov         al,02F    ;'/'
00000007: CD29       int         029       ; вывод AL на экран
00000009: EBF5       jmp         000000000 ; бесконечный цикл
```

SCASB также использует значение в регистре AL, она вычитает значение слайчного байта в памяти из 5Ch в AL. JP это редкая инструкция, здесь она используется для проверки флага четности (PF), который вычисляется по формуле в листинге. Как следствие, выводимый символ определяется не каким-то конкретным битом из случайного байта в памяти, а суммой бит, и это (надеемся) сделает результат более распределенным.

Можно сделать еще короче, если использовать недокументированную x86-инструкцию SALC (AKA SETALC) ("Set AL CF"). Она появилась в CPU и выставляет AL в 0xFF если CF это 1 или 0 если наоборот. Так что этот код не запустится на 8086/8088.

Листинг 60.2: Peter Ferrie: 10 байт

```
; AL в этом месте имеет случайное значение
00000000: AE         scasb
; CF устанавливается по результату вычитания случайного байта памяти из AL.
; так что он здесь случаен, в каком-то смысле
00000001: D6         setalc
; AL выставляется в 0xFF если CF=1 или в 0 если наоборот
00000002: 242D       and         al,02D ;'-'
; AL здесь 0x2D либо 0
00000004: 042F       add         al,02F ;'/'
; AL здесь 0x5C либо 0x2F
00000006: CD29       int         029       ; вывести AL на экране
00000008: EBF6       jmps        000000000 ; бесконечный цикл
```

Так что можно избавиться и от условных переходов. ASCII[5]-код обратного слэша ("\") это 0x5C и 0x2F для слэша ("/"). Так что нам нужно конвертировать один (псевдо-случайный) бит из флага CF в значение 0x5C или 0x2F.

Это делается легко: применяя операцию "И" ко всем битам в AL (где все 8 бит либо выставлены, либо сброшены) с 0x2D мы имеем просто 0 или 0x2D. Прибавляя значение 0x2F к этому значению, мы получаем 0x5C или 0x2F. И просто выводим это на экран.

60.1.4 Вывод

Также стоит отметить, что результат может быть разным в эмуляторе DOSBox, Windows NT и даже MS-DOS, из-за разных условий: чип таймера может эмулироваться по-разному, изначальные значения регистров также могут быть разными.

[4] http://pferrie.host22.com/misc/10print.htm
[5] American Standard Code for Information Interchange

Часть VII

Прочее

Глава 61

npad

Это макрос в ассемблере, для выравнивания некоторой метки по некоторой границе.

Это нужно для тех *нагруженных* меток, куда чаще всего передается управление, например, начало тела цикла. Для того чтобы процессор мог эффективнее вытягивать данные или код из памяти, через шину с памятью, кэширование, и т.д.

Взято из `listing.inc` (MSVC):

Это, кстати, любопытный пример различных вариантов `NOP`-ов. Все эти инструкции не дают никакого эффекта, но отличаются разной длиной.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
  if size eq 3
    ; lea ecx, [ecx+00]
    DB 8DH, 49H, 00H
  else
   if size eq 4
     ; lea esp, [esp+00]
     DB 8DH, 64H, 24H, 00H
   else
    if size eq 5
      add eax, DWORD PTR 0
    else
     if size eq 6
       ; lea ebx, [ebx+00000000]
       DB 8DH, 9BH, 00H, 00H, 00H, 00H
     else
      if size eq 7
        ; lea esp, [esp+00000000]
        DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
      else
       if size eq 8
        ; jmp .+8; .npad 6
        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
       else
```

```
        if size eq 9
         ; jmp .+9; .npad 7
         DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
        else
         if size eq 10
          ; jmp .+A; .npad 7; .npad 1
          DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
         else
          if size eq 11
           ; jmp .+B; .npad 7; .npad 2
           DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
          else
           if size eq 12
            ; jmp .+C; .npad 7; .npad 3
            DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
           else
            if size eq 13
             ; jmp .+D; .npad 7; .npad 4
             DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24H, 00H
            else
             if size eq 14
              ; jmp .+E; .npad 7; .npad 5
              DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, 00H, 00H, 00H
             else
              if size eq 15
               ; jmp .+F; .npad 7; .npad 6
               DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, 00H, 00H, 00H, 00H
              else
               %out error: unsupported npad size
               .err
              endif
             endif
            endif
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

Глава 62

Модификация исполняемых файлов

62.1 Текстовые строки

Сишные строки проще всего модифицировать (если они не зашифрованы) в любом шестнадцатеричном редакторе. Эта техника доступна даже для тех, кто вовсе не разбирается в машинном коде и форматах исполняемых файлов. Новая строка не должна быть длиннее старой, потому что имеется риск затереть какую-то другую переменную или код. Используя этот метод, очень много ПО было *локализовано* во времена MS-DOS, как минимум, в странах бывшего СССР, в 80-х и 90-х. Отсюда наличие очень странных аббревиатур и сокращений в *локализованном* ПО: там просто не было места для более длинных строк.

В строках в Delphi, длина строки также должна быть поправлена, если нужно.

62.2 x86-код

Часто необходимые задачи:

- Часто нужно просто запретить исполнение какой-либо инструкции. И чаще всего, это можно сделать заполняя её байтом `0x90` (NOP).
- Условные переходы, имеющие опкод вроде `74 xx` (`JZ`), так же могут быть заполнены двумя NOP-ами. Также возможно запретить исполнение условного перехода записав 0 во второй байт (*jump offset*).
- Еще одна часто необходимая задача это сделать условный переход всегда срабатывающим: это возможно при помощи записи `0xEB` вместо опкода, это значит `JMP`.
- Исполнение ф-ции может быть запрещено, если записать `RETN` (0xC3) в её начале. Это справедливо для всех ф-ций кроме `stdcall` (44.2). При модификации ф-ций `stdcall`, нужно в начале определить количество аргументов (например, отыскав `RETN` в этой ф-ции), и использовать `RETN` с 16-битным аргументом (0xC2).
- Иногда, запрещенная ф-ция должна возвращать 0 или 1. Это можно сделать при помощи `MOV EAX, 0` или `MOV EAX, 1`, но это слишком многословно. Способ получше это `XOR EAX, EAX` (2 байта) или `XOR EAX, EAX / INC EAX` (3 байта).

ПО может быть защищено от модификаций. Эта защита чаще всего реализуется путем чтения кода и вычисления контрольной суммы. Следовательно, код должен быть прочитан перед тем как защита сработает. Это можно определить установив брякпойнт на чтение памяти.

В tracer имеется опция BPM для этого.

Релоки в исполняемых PE-файлах (48.2.6) не должны быть тронуты, потому что загрузчик Windows перезапишет ваш новый код. (Они выделяются серым в Hiew, например: илл. 6.12). В качестве последней меры, можно записать `JMP` для обхода релока, либо же придется модифицировать таблицу релоков.

Глава 63

Compiler intrinsic

Специфичная для компилятора ф-ция не являющаяся обычной библиотечной ф-цией. Компилятор вместо её вызова генерирует определенный машинный код. Нередко, это псевдофункции для определенной инструкции CPU.

Например, в языках Си/Си++ нет операции циклического сдвига, а во многих CPU она есть. Чтобы программисту были доступны эти инструкции, в MSVC есть псевдофункции *_rotl()* и *_rotr()*[1], которые компилятором напрямую транслируются в x86-инструкции `ROL`/`ROR`.

Еще один пример это ф-ции позволяющие генерировать SSE-инструкции прямо в коде.

Полный список intrinsics от MSVC: http://msdn.microsoft.com/en-us/library/26td21ds.aspx.

[1] http://msdn.microsoft.com/en-us/library/5cc576c4.aspx

Глава 64

Аномалии компиляторов

Intel C++ 10.1 которым скомпилирован Oracle RDBMS 11.2 Linux86, может сгенерировать два JZ идущих подряд, причем на второй JZ нет ссылки ниоткуда. Второй JZ таким образом, не имеет никакого смысла.

Листинг 64.1: kdli.o из libserver11.a

```
.text:08114CF1                  loc_8114CF1: ; CODE XREF: __PGOSF539_kdlimemSer+89A
.text:08114CF1                               ; __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08              mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14           movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01              test    dl, 1
.text:08114CFB 0F 85 17 08 00 00     jnz     loc_8115518
.text:08114D01 85 C9                 test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00     jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00     jz      loc_8115518
.text:08114D0F 8B 53 08              mov     edx, [ebx+8]
.text:08114D12 89 55 FC              mov     [ebp+var_4], edx
.text:08114D15 31 C0                 xor     eax, eax
.text:08114D17 89 45 F4              mov     [ebp+var_C], eax
.text:08114D1A 50                    push    eax
.text:08114D1B 52                    push    edx
.text:08114D1C E8 03 54 00 00        call    len2nbytes
.text:08114D21 83 C4 08              add     esp, 8
```

Листинг 64.2: оттуда же

```
.text:0811A2A5                  loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5                               ; kdliSerLengths+1C1
.text:0811A2A5 8B 7D 08              mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10              mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14           movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01              test    dl, 1
.text:0811A2B2 75 3E                 jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01              and     eax, 1
.text:0811A2B7 74 1F                 jz      short loc_811A2D8
.text:0811A2B9 74 37                 jz      short loc_811A2F2
.text:0811A2BB 6A 00                 push    0
.text:0811A2BD FF 71 08              push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF        call    len2nbytes
```

Возможно, это ошибка его кодегенератора, не выявленная тестами (ведь результирующий код и так работает нормально).

Еще одна такая ошибка компилятора описана здесь (17.2.4).

Я показываю здесь подобные случаи для того, чтобы легче было понимать, что подобные ошибки компиляторов все же имеют место быть, и не следует ломать голову над тем, почему он сгенерировал такой странный код.

Глава 65

OpenMP

OpenMP это один из простейших способов распараллелить работу простого алгоритма.

В качестве примера, попытаемся написать программу для вычисления криптографического *nonce*. В моем простейшем примере, *nonce* это число, добавляемое к нешифрованному тексту, чтобы получить хэш с какой-то особенностью. Например, на одной из стадии, протокол Bitcoin требует найти такую *nonce*, чтобы в результате хэширования подряд шли определенное количество нулей. Это еще называется "proof of work" [1] (т.е., система доказывает, что она произвела какие-то очень ресурсоёмкие вычисления и затратила время на это).

Мой пример не связан с Bitcoin, он будет пытаться добавлять числа к строке "hello, world!_" чтобы найти такое число, при котором строка вида "hello, world!_<number>" после хеширования алгоритмом SHA512 будет содержать как минимум 3 нулевых байта в начале.

Ограничимся перебором всех чисел в интервале 0..INT32_MAX-1 (т.е., `0x7FFFFFFE` или 2147483646).

Алгоритм очень простой:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
        uint8_t buf[32];
        struct sha512_ctx ctx;
        uint8_t res[64];

        // update statistics
        int t=omp_get_thread_num();

        if (__min[t]==-1)
                __min[t]=nonce;
        if (__max[t]==-1)
                __max[t]=nonce;
```

[1] https://ru.wikipedia.org/wiki/Proof-of-work

```
        __min[t]=min(__min[t], nonce);
        __max[t]=max(__max[t], nonce);

        // idle if valid nonce found
        if (found)
                return;

        memset (buf, 0, sizeof(buf));
        sprintf (buf, "hello, world!_%d", nonce);

        sha512_init_ctx (&ctx);
        sha512_process_bytes (buf, strlen(buf), &ctx);
        sha512_finish_ctx (&ctx, &res);
        if (res[0]==0 && res[1]==0 && res[2]==0)
        {
                printf ("found (thread %d): [%s]. seconds spent=%d\n", t, buf, time(NULL)-start↙
    ↳ );
                found=1;
        };
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
};

int main()
{
        int32_t i;
        int threads=omp_get_max_threads();
        printf ("threads=%d\n", threads);

        __min=(int32_t*)malloc(threads*sizeof(int32_t));
        __max=(int32_t*)malloc(threads*sizeof(int32_t));
        for (i=0; i<threads; i++)
                __min[i]=__max[i]=-1;

        start=time(NULL);

        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);

        for (i=0; i<threads; i++)
                printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i, __min[i], i, __max[i]);

        free(__min); free(__max);
};
```

`check_nonce()` просто добавляет число к строке, хеширует алгоритмом SHA512 и проверяет 3 нулевых байта в начале.

Очень важная часть кода это:

```
        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);
```

Да, вот настолько просто, без `#pragma` мы просто вызываем `check_nonce()` для каждого числа от 0 до `INT32_MAX` (`0x7fffffff` или 2147483647). С `#pragma`, компилятор добавляет специальный код, который разрежет интервал цикла на меньшие интервалы, чтобы запустить их на доступных ядрах CPU [2].

[2] N.B.: Я намеренно использовал простейший пример, но на практике, применение OpenMP может быть труднее и сложнее

Пример может быть скомпилирован [3] в MSVC 2012:

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Или в GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

65.1 MSVC

Вот как MSVC 2012 генерирует главный цикл:

Листинг 65.1: MSVC 2012

```
        push    OFFSET _main$omp$1
        push    0
        push    1
        call    __vcomp_fork
        add     esp, 16                                 ; 00000010H
```

Ф-ции с префиксом vcomp связаны с OpenMP и находятся в файле vcomp*.dll. Так что тут запускается группа тредов.

Посмотрим на _mainomp1:

Листинг 65.2: MSVC 2012

```
$T1 = -8                                                ; size = 4
$T2 = -4                                                ; size = 4
_main$omp$1 PROC                                        ; COMDAT
        push    ebp
        mov     ebp, esp
        push    ecx
        push    ecx
        push    esi
        lea     eax, DWORD PTR $T2[ebp]
        push    eax
        lea     eax, DWORD PTR $T1[ebp]
        push    eax
        push    1
        push    1
        push    2147483646                              ; 7ffffffeH
        push    0
        call    __vcomp_for_static_simple_init
        mov     esi, DWORD PTR $T1[ebp]
        add     esp, 24                                 ; 00000018H
        jmp     SHORT $LN6@main$omp$1
$LL2@main$omp$1:
        push    esi
        call    _check_nonce
        pop     ecx
        inc     esi
$LN6@main$omp$1:
        cmp     esi, DWORD PTR $T2[ebp]
        jle     SHORT $LL2@main$omp$1
        call    __vcomp_for_static_end
        pop     esi
        leave
        ret     0
_main$omp$1 ENDP
```

[3] файлы sha512.(c|h) и u64.h можно взять из библиотеки OpenSSL: http://www.openssl.org/source/

Эта ф-ция будет запущена n раз параллельно, где n это число ядер CPU. `vcomp_for_static_simple_init()` вычисляет интервал для конструкта for() для текущего треда, в зависимости от текущего номера треда. Значения начала и конца цикла записаны в локальных переменных `$T1` и `$T2`. Вы также можете заметить `7ffffffeh` (или 2147483646) как аргумент для ф-ции `vcomp_for_static_simple_init()` это количество итераций всего цикла, оно будет поделено на равные части.

Потом мы видим новый цикл с вызовом ф-ции `check_nonce()` делающей всю работу.

Я также добавил немного кода в начале ф-ции `check_nonce()` для сбора статистики, с какими аргументами эта ф-ция вызывалась.

Вот что мы видим если запустим:

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7ffffffe
```

Да, результат правильный, первые 3 байта это нули:

```
C:\...\sha512sum test
000000↙
    ↳ f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403df6e3fe6019f5764fc9975e505a7395fed78↙
    ↳
0fee50eb38dd4c0279cb114672e2 *test
```

Оно требует ≈ 2..3 секунды на моем 4-х ядерном Intel Xeon E3-1220 3.10 GHz. В task manager я вижу 5 тредов: один главный тред + 4 запущенных. Я не делал никаких оптимизаций, чтобы оставить пример в как можно более простом виде. Но, наверное, этот алгоритм может работать быстрее. У моего CPU 4 ядра, вот почему OpenMP запустил именно 4 треда.

Глядя на таблицу статистики, можно легко увидеть, что цикл был разделен очень точно на 4 равных части. Ну хорошо, почти равных, если не учитывать последний бит.

Имеются также прагмы и для атомарных операций.

Посмотрим, как вот этот код будет скомпилирован:

```
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
```

Листинг 65.3: MSVC 2012

```
        push    edi
        push    OFFSET _checked
        call    __vcomp_atomic_add_i4
; Line 55
        push    OFFSET _$vcomp$critsect$
        call    __vcomp_enter_critsect
        add     esp, 12                                 ; 0000000cH
; Line 56
        mov     ecx, DWORD PTR _checked
        mov     eax, ecx
        cdq
        mov     esi, 100000                             ; 000186a0H
        idiv    esi
        test    edx, edx
        jne     SHORT $LN1@check_nonc
```

```
; Line 57
        push    ecx
        push    OFFSET ??_C@_0M@NPNHLIOO@checked?$DN?$CFd?6?$AA@
        call    _printf
        pop     ecx
        pop     ecx
$LN1@check_nonc:
        push    DWORD PTR _$vcomp$critsect$
        call    __vcomp_leave_critsect
        pop     ecx
```

Как выясняется, ф-ция `vcomp_atomic_add_i4()` в vcomp*.dll это просто крохотная ф-ция имеющая инструкцию `LOCK XADD`[4].

`vcomp_enter_critsect()` в конце концов вызывает ф-цию win32 API `EnterCriticalSection()` [5].

65.2 GCC

GCC 4.8.1 выдает программу показывающую точно такую же таблицу со статистикой, так что, реализация GCC делит цикл на части точно так же.

Листинг 65.4: GCC 4.8.1

```
        mov     edi, OFFSET FLAT:main._omp_fn.0
        call    GOMP_parallel_start
        mov     edi, 0
        call    main._omp_fn.0
        call    GOMP_parallel_end
```

В отличие от реализации MSVC, то, что делает код GCC, это запускает 3 треда, но также запускает четвертый прямо в текущем треде. Так что здесь всего 4 треда а не 5 как в случае с MSVC.

Вот ф-ция `main._omp_fn.0`:

Листинг 65.5: GCC 4.8.1

```
main._omp_fn.0:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 40
        mov     QWORD PTR [rbp-40], rdi
        call    omp_get_num_threads
        mov     ebx, eax
        call    omp_get_thread_num
        mov     esi, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     ecx, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     eax, edx
        cmp     esi, eax
        jl      .L15
.L18:
        imul    esi, ecx
        mov     edx, esi
        add     eax, edx
        lea     ebx, [rax+rcx]
        cmp     eax, ebx
        jge     .L14
```

[4] О префиксе LOCK читайте больше: B.6.1

[5] О критических секциях читайте больше тут: 48.4

```
        mov     DWORD PTR [rbp-20], eax
.L17:
        mov     eax, DWORD PTR [rbp-20]
        mov     edi, eax
        call    check_nonce
        add     DWORD PTR [rbp-20], 1
        cmp     DWORD PTR [rbp-20], ebx
        jl      .L17
        jmp     .L14
.L15:
        mov     eax, 0
        add     ecx, 1
        jmp     .L18
.L14:
        add     rsp, 40
        pop     rbx
        pop     rbp
        ret
```

Здесь мы видим это деление явно: вызывая `omp_get_num_threads()` и `omp_get_thread_num()` мы получаем количество запущенных тредов, а также номер текущего треда, и затем определяем интервал цикла. И затем запускаем `check_nonce()`.

GCC также вставляет инструкцию `LOCK ADD` прямо в том месте кода, где MSVC сгенерировал вызов отдельной ф-ции в DLL:

Листинг 65.6: GCC 4.8.1

```
        lock add        DWORD PTR checked[rip], 1
        call    GOMP_critical_start
        mov     ecx, DWORD PTR checked[rip]
        mov     edx, 351843721
        mov     eax, ecx
        imul    edx
        sar     edx, 13
        mov     eax, ecx
        sar     eax, 31
        sub     edx, eax
        mov     eax, edx
        imul    eax, eax, 100000
        sub     ecx, eax
        mov     eax, ecx
        test    eax, eax
        jne     .L7
        mov     eax, DWORD PTR checked[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
        mov     eax, 0
        call    printf
.L7:
        call    GOMP_critical_end
```

Ф-ции с префиксом GOMP это часть библиотеки GNU OpenMP. В отличие от vcomp*.dll, её исходный код свободно доступен: https://github.com/mirrors/gcc/tree/master/libgomp.

Глава 66

Itanium

Еще одна очень интересная архитектура (хотя и почти провальная) это Intel Itanium (IA64). Другие OOE[1]-процессоры сами решают, как переставлять инструкции и исполнять их параллельно, EPIC[2] это была попытка сдвинуть эти решения на компилятор: дать ему возможность самому группировать инструкции во время компиляции.

Это вылилось в очень сложные компиляторы

Вот один пример IA64-кода: простой криптоалгоритм из ядра Linux:

Листинг 66.1: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
        struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];

        n = TEA_ROUNDS;

        while (n-- > 0) {
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

И вот как он был скомпилирован:

Листинг 66.2: Linux Kernel 3.2.0.4 для Itanium 2 (McKinley)

```
0090|                                    tea_encrypt:
0090|08 80 80 41 00 21                      adds r16 = 96, r32              // ptr to ctx->KEY↙
    ↳ [2]
```

[1]Out-of-order execution

[2]Explicitly parallel instruction computing

```
0096|80 C0 82 00 42 00                     adds r8 = 88, r32               // ptr to ctx->KEY↙
    ↳ [0]
009C|00 00 04 00                           nop.i 0
00A0|09 18 70 41 00 21                     adds r3 = 92, r32               // ptr to ctx->KEY↙
    ↳ [1]
00A6|F0 20 88 20 28 00                     ld4 r15 = [r34], 4              // load z
00AC|44 06 01 84                           adds r32 = 100, r32;;           // ptr to ctx->KEY↙
    ↳ [3]
00B0|08 98 00 20 10 10                     ld4 r19 = [r16]                 // r19=k2
00B6|00 01 00 00 42 40                     mov r16 = r0                    // r0 always contain↙
    ↳  zero
00BC|00 08 CA 00                           mov.i r2 = ar.lc                // save lc register
00C0|05 70 00 44 10 10 9E FF FF FF 7F 20   ld4 r14 = [r34]                 // load y
00CC|92 F3 CE 6B                           movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00                     nop.m 0
00D6|50 01 20 20 20 00                     ld4 r21 = [r8]                  // r21=k0
00DC|F0 09 2A 00                           mov.i ar.lc = 31                // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10                     ld4 r20 = [r3];;                // r20=k1
00E6|20 01 80 20 20 00                     ld4 r18 = [r32]                 // r18=k3
00EC|00 00 04 00                           nop.i 0
00F0|
00F0|                                  loc_F0:
00F0|09 80 40 22 00 20                     add r16 = r16, r17              // r16=sum, r17=↙
    ↳ TEA_DELTA
00F6|D0 71 54 26 40 80                     shladd r29 = r14, 4, r21        // r14=y, r21=k0
00FC|A3 70 68 52                           extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20                     add r30 = r16, r14
0106|B0 E1 50 00 40 40                     add r27 = r28, r20;;            // r20=k1
010C|D3 F1 3C 80                           xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20                     xor r25 = r27, r26;;
0116|F0 78 64 00 40 00                     add r15 = r15, r25              // r15=z
011C|00 00 04 00                           nop.i 0;;
0120|00 00 00 00 01 00                     nop.m 0
0126|80 51 3C 34 29 60                     extr.u r24 = r15, 5, 27
012C|F1 98 4C 80                           shladd r11 = r15, 4, r19        // r19=k2
0130|0B B8 3C 20 00 20                     add r23 = r15, r16;;
0136|A0 C0 48 00 40 00                     add r10 = r24, r18              // r18=k3
013C|00 00 04 00                           nop.i 0;;
0140|0B 48 28 16 0F 20                     xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00                     xor r22 = r23, r9
014C|00 00 04 00                           nop.i 0;;
0150|11 00 00 00 01 00                     nop.m 0
0156|E0 70 58 00 40 A0                     add r14 = r14, r22
015C|A0 FF FF 48                           br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15                     st4 [r33] = r15, 4              // store z
0166|00 00 00 02 00 00                     nop.m 0
016C|20 08 AA 00                           mov.i ar.lc = r2;;              // restore lc ↙
    ↳ register
0170|11 00 38 42 90 11                     st4 [r33] = r14                 // store y
0176|00 00 00 02 00 80                     nop.i 0
017C|08 00 84 00                           br.ret.sptk.many b0;;
```

Прежде всего, все инструкции IA64 сгруппированы в пачки (bundle) из трех инструкций. Каждая пачка имеет размер 16 байт и состоит из template-кода и трех инструкций. IDA показывает пачки как 6+6+4 байт — вы можете легко заметить эту повторяющуюся структуру.

Все 3 инструкции каждой пачки обычно исполняются одновременно, если только у какой-то инструкции нет "стоп-бита".

Вероятно, инженеры Intel и HP собрали статистику наиболее встречающихся шаблонных сочетаний инструкций и решили ввести типы пачек (AKA "templates"): код пачки определяет типы инструкций в пачке. Их всего 12. Например, нулевой тип это MII, что означает: первая инструкция это Memory (загрузка или запись в память), вторая и третья это I (инструкция, работающая с целочисленными значениями). Еще один пример, тип 0x1d:

MFB: первая инструкция это Memory (загрузка или запись в память), вторая это Float (инструкция, работающая с FPU), третья это Branch (инструкция перехода).

Если компилятор не может подобрать подходящую инструкцию в соответствующее место пачки, он может вставить NOP: вы можете здесь увидеть инструкции `nop.i` (NOP на том месте где должна была бы находиться целочисленная инструкция) или `nop.m` (инструкция обращения к памяти должна была находиться здесь). Если вручную писать на ассемблере, NOP-ы могут вставляться автоматически.

И это еще не все. Пачки тоже могут быть объединены в группы. Каждая пачка может иметь "стоп-бит", так что все следующие друг за другом пачки вплоть до той, что имеет стоп-бит, могут быть исполнены одновременно. На практике, Itanium 2 может исполнять 2 пачки одновременно, таким образом, исполнять 6 инструкций одновременно.

Так что все инструкции внутри пачки и группы не могут мешать друг другу (т.е., не должны иметь data hazard-ов). А если это так, то результаты будут непредсказуемые.

На ассемблере, каждый стоп-бит маркируется как `;;` (две точки с запятой) после инструкции. Так, инструкции на [180-19c] могут быть исполнены одновременно: они не мешают друг другу. Следующая группа: [1a0-1bc].

Мы также видим стоп-бит на 22c. Следующая инструкция на 230 также имеет стоп-бит. Это значит, что эта инструкция должна исполняться изолированно от всех остальных (как в CISC). Действительно: следующая инструкция на 236 использует результат полученный от нее (значение в регистре r10), так что они не могут исполняться одновременно. Должно быть, компилятор не смог найти лучший способ распараллелить инструкции, или, иными словами, загрузить CPU насколько это возможно, отсюда так много стоп-битов и NOP-ов. Писать на ассемблере вручную это также очень трудная задача: программист должен группировать инструкции вручную.

У программиста остается возможность добавлять стоп-биты к каждой инструкции, но это сведет на нет всю мощность Itanium, ради которой он создавался.

Интересные примеры написания IA64-кода вручную можно найти в исходниках ядра Linux:
`http://lxr.free-electrons.com/source/arch/ia64/lib/`.

Еще одна вводная статья об ассемблере Itanium: [5].

Еще одна интересная особенность Itanium это *speculative execution* (исполнение инструкций заранее, когда еще не известно, нужно ли это) и бит NaT ("not a thing"), отдаленно напоминающий NaN-числа:
`http://blogs.msdn.com/b/oldnewthing/archive/2004/01/19/60162.aspx`.

Глава 67

Модель памяти в 8086

Разбирая 16-битные программы для MS-DOS или Win16 (55.3 или 31.5), мы можем увидеть что указатель состоит из двух 16-битных значений. Что это означает? О да, еще один дивный артефакт MS-DOS и 8086.

8086/8088 был 16-битным процессором, но мог адресовать 20-битное адресное пространство (таким образом мог адресовать 1MB внешней памяти). Внешняя адресное пространство было разделено между RAM (максимум 640KB), ROM[1], окна для видеопамяти, EMS-карт, и т.д.

Припомним также что 8086/8088 был на самом деле наследником 8-битного процессора 8080. Процессор 8080 имел 16-битное адресное пространство, т.е., мог адресовать только 64KB. И возможно в расчете на портирование старого ПО[2], 8086 может поддерживать 64-килобайтные окна, одновременно много таких, расположенных внутри одномегабайтного адресного пространства. Это, в каком-то смысле, игрушечная виртуализация. Все регистры 8086 16-битные, так что, чтобы адресовать больше, специальные сегментные регистры (CS, DS, ES, SS) были введены. Каждый 20-битный указатель вычисляется используя значения из пары состоящей из сегментного регистра и адресного регистра (например DS:BX) вот так

$$real\_address = (segment\_register \ll 4) + address\_register$$

Например, окно памяти для графики (EGA[3], VGA[4]) на старых IBM PC-совместимых компьютерах имело размер 64KB. Для доступа к нему, значение 0xA000 должно быть записано в один из сегментных регистров, например, в DS. Тогда DS:0 будет адресовать самый первый байт видеопамяти, а DS:0xFFFF — самый последний байт. А реальный адрес на 20-битной адресной шине, на самом деле будет от 0xA0000 до 0xAFFFF.

Программа может содержать жесткопривязанные адреса вроде 0x1234, но ОС может иметь необходимость загрузить программу по другим адресам, так что она пересчитает значения для сегментных регистров так, что программа будет нормально работать не обращая внимания на то, в каком месте памяти она была расположена.

Так что, любой указатель в окружении старой MS-DOS на самом деле состоял из адреса сегмента и адреса внутри сегмента, т.е., из двух 16-битных значений. 20-битного значения было бы достаточно для этого, хотя, тогда пришлось бы вычислять адреса слишком часто: так что передача большего количества информации в стеке — это более хороший баланс между экономией места и удобством.

Кстати, из-за всего этого, не было возможным выделить блок памяти больше чем 64KB.

В 80286 сегментные регистры получили новую роль селекторов, имеющих немного другую ф-цию.

Когда появился процессор 80386 и компьютеры с большей памятью, MS-DOS была всё еще популярна, так что появились DOS-экстендеры: на самом деле это уже был шаг к “серьезным” ОС, они переключали CPU в защищенный режим и предлагали куда лучшее API для программ, которые всё еще предполагалось запускать в MS-DOS. Широко известные примеры это DOS/4GW (игра DOOM была скомпилирована под него), Phar Lap, PMODE

Кстати, точно такой же способ адресации памяти был и в 16-битной линейке Windows 3.x, перед Win32.

[1] Read-only memory
[2] Хотя я и не уверен на 100%
[3] Enhanced Graphics Adapter
[4] Video Graphics Array

Глава 68

Перестановка basic block-ов

68.1 Profile-guided optimization

Этот метод оптимизации кода может перемещать некоторые basic block-и в другую секцию исполняемого бинарного файла.

Очевидно, в ф-ции есть места которые исполняются чаще всего (например, тела циклов) и реже всего (например, код обработки ошибок, обработчики исключений).

Компилятор добавляет дополнительный (instrumentation) код в исполняемый файл, затем разработчик запускает его с тестами для сбора статистики. Затем компилятор, при помощи собранной статистики, приготавливает итоговый исполняемый файл где весь редко исполняемый код перемещен в другую секцию.

В результате, весь часто исполняемый код ф-ции становится компактным, что очень важно для скорости исполнения и кэш-памяти.

Пример из Oracle RDBMS, который скомпилирован при помощи Intel C++:

Листинг 68.1: orageneric11.dll (win32)

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue:
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error                ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...

; address 0x60B953F0
```

```
__VInfreq__skgfsync:
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ; "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue

error:
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function called with invalid FIB/IOV ↙
    ↳ structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

Расстояние между двумя адресами приведенных фрагментов кода почти 9 МБ.

Весь редко исполняемый код помещен в конце секции кода DLL-файла, среди редко исполняемых частей прочих ф-ций. Эта часть ф-ции была отмечена компилятором Intel C++ префиксом `VInfreg`. Мы видим часть ф-ции которая записывает в лог-файл (вероятно, в случае ошибки или предупреждения или чего-то в этом роде) которая, наверное, не исполнялась слишком часто когда разработчики Oracle собирали статистику (если вообще исполнялась). Basic block записывающий в лог-файл, в конце концов возвращает управление в "горячую" часть ф-ции

Другая "редкая" часть это basic block возвращающий код ошибки 27050.

В ELF-файлах для Linux весь редко исполняемый код перемещается компилятором Intel C++ в другую секцию (`text.unlikely`) оставляя весь "горячий" код в секции `text.hot`.

С точки зрения reverse engineer-а, эта информация может помочь разделить ф-цию на её основу и части, отвечающие за обработку ошибок.

Часть VIII

Что стоит почитать

Глава 69

Книги

69.1 Windows

[30].

69.2 Си/Си++

[16].

69.3 x86 / x86-64

[14], [1]

69.4 ARM

Документация от ARM: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

Глава 70

Блоги

70.1 Windows

- Microsoft: Raymond Chen
- http://www.nynaeve.net/

Глава 71

Прочее

Имеются два отличных субреддита на reddit.com посвященных RE[1]: ReverseEngineering и REMath (для тем посвященных пересечению RE и математики).

Имеется также часть сайта Stack Exchange посвященная RE:
http://reverseengineering.stackexchange.com/.

[1]Reverse Engineering

Часть IX

Задачи

Почти для всех задач, если не указано иное, два вопроса:

1) Что делает эта функция? Ответ должен состоять из одной фразы.

2) Перепишите эту функцию на Си/Си++.

Пользоваться Google-м, для поиска каких-либо зацепок, разрешается. Впрочем, для усложения своей задачи, вы можете попробовать и без Google.

Подсказки и ответы собраны в приложении к этой книге.

Глава 72

Уровень 1

Задачи первого уровня, это те, которые можно решать голове.

72.1 Задача 1.1

72.1.1 MSVC 2012 x64 + /Ox

```
a$ = 8
b$ = 16
f       PROC
        cmp     ecx, edx
        cmovg   edx, ecx
        mov     eax, edx
        ret     0
f       ENDP
```

72.1.2 Keil (ARM)

```
        CMP     r0,r1
        MOVLE   r0,r1
        BX      lr
```

72.1.3 Keil (thumb)

```
        CMP     r0,r1
        BGT     |L0.6|
        MOVS    r0,r1
|L0.6|
        BX      lr
```

72.2 Задача 1.2

Почему инструкция LOOP больше не используется компиляторами?

72.3 Задача 1.3

Возьмите пример из секции “Циклы” (12), скомпилируйте его в вашей любимой ОС и компиляторе, и модифицируйте исполняемый файл так, чтобы цикл был в пределах [6..20].

72.4 Задача 1.4

Эта программа запрашивает пароль. Найдите его.

Как дополнительное упражнение, попробуйте изменить пароль, модифицируя исполняемый файл, в т.ч., на более короткий или более длинный.

Попытайтесь также вызвать аварийное завершение программы только при помощи ввода строки.

- win32: http://yurichev.com/RE-exercises/1/4/password1.exe
- Linux x86: http://yurichev.com/RE-exercises/1/4/password1_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/1/4/password1_MacOSX64.tar

Глава 73

Уровень 2

Для решения задач второго уровня, вам вероятно понадобится текстовый редактор или тетрадка с ручкой.

73.1 Задача 2.1

Это стандартная функция из библиотек Си. Исходник взят из OpenWatcom.

73.1.1 MSVC 2010

```
_TEXT    SEGMENT
_input$ = 8                         ; size = 1
_f PROC
    push   ebp
    mov    ebp, esp
    movsx  eax, BYTE PTR _input$[ebp]
    cmp    eax, 97          ; 00000061H
    jl     SHORT $LN1@f
    movsx  ecx, BYTE PTR _input$[ebp]
    cmp    ecx, 122         ; 0000007aH
    jg     SHORT $LN1@f
    movsx  edx, BYTE PTR _input$[ebp]
    sub    edx, 32          ; 00000020H
    mov    BYTE PTR _input$[ebp], dl
$LN1@f:
    mov    al, BYTE PTR _input$[ebp]
    pop    ebp
    ret    0
_f ENDP
_TEXT    ENDS
```

73.1.2 GCC 4.4.1 + -O3

```
_f              proc near

input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                movzx   eax, byte ptr [ebp+input]
                lea     edx, [eax-61h]
                cmp     dl, 19h
                ja      short loc_80483F2
                sub     eax, 20h

loc_80483F2:
```

```
                pop     ebp
                retn
_f              endp
```

73.1.3 Keil (ARM) + -O3

```
        SUB     r1,r0,#0x61
        CMP     r1,#0x19
        SUBLS   r0,r0,#0x20
        ANDLS   r0,r0,#0xff
        BX      lr
```

73.1.4 Keil (thumb) + -O3

```
        MOVS    r1,r0
        SUBS    r1,r1,#0x61
        CMP     r1,#0x19
        BHI     |L0.14|
        SUBS    r0,r0,#0x20
        LSLS    r0,r0,#24
        LSRS    r0,r0,#24
|L0.14|
        BX      lr
```

73.2 Задача 2.2

Это так же стандартная функция из библиотек Си. Исходник взят из OpenWatcom и немного переделан This is also standard C library function. Source code is taken from OpenWatcom and modified slightly.

Эта функция использует стандартные функции Си: isspace() и isdigit().

73.2.1 MSVC 2010 + /Ox

```
EXTRN   _isdigit:PROC
EXTRN   _isspace:PROC
EXTRN   ___ptr_check:PROC
; Function compile flags: /Ogtpy
_TEXT   SEGMENT
_p$ = 8                         ; size = 4
_f   PROC
    push   ebx
    push   esi
    mov    esi, DWORD PTR _p$[esp+4]
    push   edi
    push   0
    push   esi
    call   ___ptr_check
    mov    eax, DWORD PTR [esi]
    push   eax
    call   _isspace
    add    esp, 12                  ; 0000000cH
    test   eax, eax
    je     SHORT $LN6@f
    npad   2
$LL7@f:
    mov    ecx, DWORD PTR [esi+4]
    add    esi, 4
    push   ecx
```

```
    call   _isspace
    add    esp, 4
    test   eax, eax
    jne    SHORT $LL7@f
$LN6@f:
    mov    bl, BYTE PTR [esi]
    cmp    bl, 43                  ; 0000002bH
    je     SHORT $LN4@f
    cmp    bl, 45                  ; 0000002dH
    jne    SHORT $LN5@f
$LN4@f:
    add    esi, 4
$LN5@f:
    mov    edx, DWORD PTR [esi]
    push   edx
    xor    edi, edi
    call   _isdigit
    add    esp, 4
    test   eax, eax
    je     SHORT $LN2@f
$LL3@f:
    mov    ecx, DWORD PTR [esi]
    mov    edx, DWORD PTR [esi+4]
    add    esi, 4
    lea    eax, DWORD PTR [edi+edi*4]
    push   edx
    lea    edi, DWORD PTR [ecx+eax*2-48]
    call   _isdigit
    add    esp, 4
    test   eax, eax
    jne    SHORT $LL3@f
$LN2@f:
    cmp    bl, 45                  ; 0000002dH
    jne    SHORT $LN14@f
    neg    edi
$LN14@f:
    mov    eax, edi
    pop    edi
    pop    esi
    pop    ebx
    ret    0
_f    ENDP
_TEXT    ENDS
```

73.2.2 GCC 4.4.1

Задача немного усложняется тем, что GCC представил isspace() и isdigit() как inline-функции и вставил их тела прямо в код.

```
_f              proc near

var_10          = dword ptr -10h
var_9           = byte ptr -9
input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                jmp     short loc_8048410
loc_804840C:
                add     [ebp+input], 4
```

```
loc_8048410:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 2000h
                test    eax, eax
                jnz     short loc_804840C
                mov     eax, [ebp+input]
                mov     eax, [eax]
                mov     [ebp+var_9], al
                cmp     [ebp+var_9], '+'
                jz      short loc_8048444
                cmp     [ebp+var_9], '-'
                jnz     short loc_8048448

loc_8048444:
                add     [ebp+input], 4

loc_8048448:
                mov     [ebp+var_10], 0
                jmp     short loc_8048471

loc_8048451:
                mov     edx, [ebp+var_10]
                mov     eax, edx
                shl     eax, 2
                add     eax, edx
                add     eax, eax
                mov     edx, eax
                mov     eax, [ebp+input]
                mov     eax, [eax]
                lea     eax, [edx+eax]
                sub     eax, 30h
                mov     [ebp+var_10], eax
                add     [ebp+input], 4

loc_8048471:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 800h
                test    eax, eax
                jnz     short loc_8048451
                cmp     [ebp+var_9], 2Dh
                jnz     short loc_804849A
                neg     [ebp+var_10]

loc_804849A:
                mov     eax, [ebp+var_10]
                leave
```

```
                retn
_f              endp
```

73.2.3 Keil (ARM) + -O3

```
        PUSH    {r4,lr}
        MOV     r4,r0
        BL      __rt_ctype_table
        LDR     r2,[r0,#0]
|L0.16|
        LDR     r0,[r4,#0]
        LDRB    r0,[r2,r0]
        TST     r0,#1
        ADDNE   r4,r4,#4
        BNE     |L0.16|
        LDRB    r1,[r4,#0]
        MOV     r0,#0
        CMP     r1,#0x2b
        CMPNE   r1,#0x2d
        ADDEQ   r4,r4,#4
        B       |L0.76|
|L0.60|
        ADD     r0,r0,r0,LSL #2
        ADD     r0,r3,r0,LSL #1
        SUB     r0,r0,#0x30
        ADD     r4,r4,#4
|L0.76|
        LDR     r3,[r4,#0]
        LDRB    r12,[r2,r3]
        CMP     r12,#0x20
        BEQ     |L0.60|
        CMP     r1,#0x2d
        RSBEQ   r0,r0,#0
        POP     {r4,pc}
```

73.2.4 Keil (thumb) + -O3

```
        PUSH    {r4-r6,lr}
        MOVS    r4,r0
        BL      __rt_ctype_table
        LDR     r2,[r0,#0]
        B       |L0.14|
|L0.12|
        ADDS    r4,r4,#4
|L0.14|
        LDR     r0,[r4,#0]
        LDRB    r0,[r2,r0]
        LSLS    r0,r0,#31
        BNE     |L0.12|
        LDRB    r1,[r4,#0]
        CMP     r1,#0x2b
        BEQ     |L0.32|
        CMP     r1,#0x2d
        BNE     |L0.34|
|L0.32|
        ADDS    r4,r4,#4
|L0.34|
        MOVS    r0,#0
        B       |L0.48|
```

```
|L0.38|
        MOVS     r5,#0xa
        MULS     r0,r5,r0
        ADDS     r4,r4,#4
        SUBS     r0,r0,#0x30
        ADDS     r0,r3,r0
|L0.48|
        LDR      r3,[r4,#0]
        LDRB     r5,[r2,r3]
        CMP      r5,#0x20
        BEQ      |L0.38|
        CMP      r1,#0x2d
        BNE      |L0.62|
        RSBS     r0,r0,#0
|L0.62|
        POP      {r4-r6,pc}
```

73.3 Задача 2.3

Это так же стандартная функция из библиотек Си, а вернее, две функции, работающие в паре. Исходник взят из MSVC 2010 и немного переделан.

Суть переделки в том, что эта функция может корректно работать в мульти-тредовой среде, а я, для упрощения (или запутывания) убрал поддержку этого.

73.3.1 MSVC 2010 + /Ox

```
_BSS    SEGMENT
_v    DD    01H DUP (?)
_BSS    ENDS

_TEXT    SEGMENT
_s$ = 8                             ; size = 4
f1    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp]
    mov    DWORD PTR _v, eax
    pop    ebp
    ret    0
f1    ENDP
_TEXT    ENDS
PUBLIC    f2

_TEXT    SEGMENT
f2    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _v
    imul   eax, 214013      ; 000343fdH
    add    eax, 2531011     ; 00269ec3H
    mov    DWORD PTR _v, eax
    mov    eax, DWORD PTR _v
    shr    eax, 16          ; 00000010H
    and    eax, 32767       ; 00007fffH
    pop    ebp
    ret    0
f2    ENDP
_TEXT    ENDS
END
```

73.3.2 GCC 4.4.1

```
                public f1
f1              proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     ds:v, eax
                pop     ebp
                retn
f1              endp

                public f2
f2              proc near
                push    ebp
                mov     ebp, esp
                mov     eax, ds:v
                imul    eax, 343FDh
                add     eax, 269EC3h
                mov     ds:v, eax
                mov     eax, ds:v
                shr     eax, 10h
                and     eax, 7FFFh
                pop     ebp
                retn
f2              endp

bss             segment dword public 'BSS' use32
                assume cs:_bss
                dd ?
bss             ends
```

73.3.3 Keil (ARM) + -O3

```
f1 PROC
        LDR      r1,|L0.52|
        STR      r0,[r1,#0]  ; v
        BX       lr
        ENDP

f2 PROC
        LDR      r0,|L0.52|
        LDR      r2,|L0.56|
        LDR      r1,[r0,#0]  ; v
        MUL      r1,r2,r1
        LDR      r2,|L0.60|
        ADD      r1,r1,r2
        STR      r1,[r0,#0]  ; v
        MVN      r0,#0x8000
        AND      r0,r0,r1,LSR #16
        BX       lr
        ENDP

|L0.52|
        DCD      ||.data||
|L0.56|
        DCD      0x000343fd
```

```
|L0.60|
        DCD      0x00269ec3
```

73.3.4 Keil (thumb) + -O3

```
f1 PROC
        LDR      r1,|L0.28|
        STR      r0,[r1,#0]  ; v
        BX       lr
        ENDP

f2 PROC
        LDR      r0,|L0.28|
        LDR      r2,|L0.32|
        LDR      r1,[r0,#0]  ; v
        MULS     r1,r2,r1
        LDR      r2,|L0.36|
        ADDS     r1,r1,r2
        STR      r1,[r0,#0]  ; v
        LSLS     r0,r1,#1
        LSRS     r0,r0,#17
        BX       lr
        ENDP

|L0.28|
        DCD      ||.data||
|L0.32|
        DCD      0x000343fd
|L0.36|
        DCD      0x00269ec3
```

73.4 Задача 2.4

Это стандартная функция из библиотек Си. Исходник взят из MSVC 2010.

73.4.1 MSVC 2010 + /Ox

```
PUBLIC    _f
_TEXT    SEGMENT
_arg1$ = 8             ; size = 4
_arg2$ = 12            ; size = 4
_f    PROC
    push   esi
    mov    esi, DWORD PTR _arg1$[esp]
    push   edi
    mov    edi, DWORD PTR _arg2$[esp+4]
    cmp    BYTE PTR [edi], 0
    mov    eax, esi
    je     SHORT $LN7@f
    mov    dl, BYTE PTR [esi]
    push   ebx
    test   dl, dl
    je     SHORT $LN4@f
    sub    esi, edi
    npad   6
$LL5@f:
    mov    ecx, edi
    test   dl, dl
```

```
    je      SHORT $LN2@f
$LL3@f:
    mov     dl, BYTE PTR [ecx]
    test    dl, dl
    je      SHORT $LN14@f
    movsx   ebx, BYTE PTR [esi+ecx]
    movsx   edx, dl
    sub     ebx, edx
    jne     SHORT $LN2@f
    inc     ecx
    cmp     BYTE PTR [esi+ecx], bl
    jne     SHORT $LL3@f
$LN2@f:
    cmp     BYTE PTR [ecx], 0
    je      SHORT $LN14@f
    mov     dl, BYTE PTR [eax+1]
    inc     eax
    inc     esi
    test    dl, dl
    jne     SHORT $LL5@f
    xor     eax, eax
    pop     ebx
    pop     edi
    pop     esi
    ret     0
_f    ENDP
_TEXT    ENDS
END
```

73.4.2 GCC 4.4.1

```
                public f
f               proc near

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                mov     eax, [ebp+arg_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_8048443
                mov     eax, [ebp+arg_0]
                jmp     short locret_8048453

loc_80483F4:
                mov     eax, [ebp+var_4]
                mov     [ebp+var_8], eax
                mov     eax, [ebp+arg_4]
                mov     [ebp+var_C], eax
                jmp     short loc_804840A

loc_8048402:
```

```
                add     [ebp+var_8], 1
                add     [ebp+var_C], 1

loc_804840A:
                mov     eax, [ebp+var_8]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_8]
                movzx   edx, byte ptr [eax]
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                cmp     dl, al
                jz      short loc_8048402

loc_804842E:
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_804843D
                mov     eax, [ebp+var_4]
                jmp     short locret_8048453

loc_804843D:
                add     [ebp+var_4], 1
                jmp     short loc_8048444


loc_8048443:
                nop

loc_8048444:
                mov     eax, [ebp+var_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_80483F4
                mov     eax, 0

locret_8048453:
                leave
                retn
f               endp
```

73.4.3 Keil (ARM) + -O3

```
        PUSH    {r4,lr}
        LDRB    r2,[r1,#0]
        CMP     r2,#0
        POPEQ   {r4,pc}
        B       |L0.80|
|L0.20|
        LDRB    r12,[r3,#0]
        CMP     r12,#0
        BEQ     |L0.64|
        LDRB    r4,[r2,#0]
        CMP     r4,#0
```

```
        POPEQ   {r4,pc}
        CMP     r12,r4
        ADDEQ   r3,r3,#1
        ADDEQ   r2,r2,#1
        BEQ     |L0.20|
        B       |L0.76|
|L0.64|
        LDRB    r2,[r2,#0]
        CMP     r2,#0
        POPEQ   {r4,pc}
|L0.76|
        ADD     r0,r0,#1
|L0.80|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        MOVNE   r3,r0
        MOVNE   r2,r1
        MOVEQ   r0,#0
        BNE     |L0.20|
        POP     {r4,pc}
```

73.4.4 Keil (thumb) + -O3

```
        PUSH    {r4,r5,lr}
        LDRB    r2,[r1,#0]
        CMP     r2,#0
        BEQ     |L0.54|
        B       |L0.46|
|L0.10|
        MOVS    r3,r0
        MOVS    r2,r1
        B       |L0.20|
|L0.16|
        ADDS    r3,r3,#1
        ADDS    r2,r2,#1
|L0.20|
        LDRB    r4,[r3,#0]
        CMP     r4,#0
        BEQ     |L0.38|
        LDRB    r5,[r2,#0]
        CMP     r5,#0
        BEQ     |L0.54|
        CMP     r4,r5
        BEQ     |L0.16|
        B       |L0.44|
|L0.38|
        LDRB    r2,[r2,#0]
        CMP     r2,#0
        BEQ     |L0.54|
|L0.44|
        ADDS    r0,r0,#1
|L0.46|
        LDRB    r2,[r0,#0]
        CMP     r2,#0
        BNE     |L0.10|
        MOVS    r0,#0
|L0.54|
        POP     {r4,r5,pc}
```

73.5 Задача 2.5

Задача, скорее, на эрудицию, нежели на чтение кода.

Функция взята из OpenWatcom The function is taken from OpenWatcom.

73.5.1 MSVC 2010 + /Ox

```
_DATA    SEGMENT
COMM     __v:DWORD
_DATA    ENDS
PUBLIC   __real@3e45798ee2308c3a
PUBLIC   __real@4147ffff80000000
PUBLIC   __real@4150017ec0000000
PUBLIC   _f
EXTRN    __fltused:DWORD
CONST    SEGMENT
__real@3e45798ee2308c3a DQ 03e45798ee2308c3ar    ; 1e-008
__real@4147ffff80000000 DQ 04147ffff80000000r    ; 3.14573e+006
__real@4150017ec0000000 DQ 04150017ec0000000r    ; 4.19584e+006
CONST    ENDS
_TEXT    SEGMENT
_v1$ = -16               ; size = 8
_v2$ = -8                ; size = 8
_f    PROC
    sub    esp, 16       ; 00000010H
    fld    QWORD PTR __real@4150017ec0000000
    fstp   QWORD PTR _v1$[esp+16]
    fld    QWORD PTR __real@4147ffff80000000
    fstp   QWORD PTR _v2$[esp+16]
    fld    QWORD PTR _v1$[esp+16]
    fld    QWORD PTR _v1$[esp+16]
    fdiv   QWORD PTR _v2$[esp+16]
    fmul   QWORD PTR _v2$[esp+16]
    fsubp  ST(1), ST(0)
    fcomp  QWORD PTR __real@3e45798ee2308c3a
    fnstsw ax
    test   ah, 65        ; 00000041H
    jne    SHORT $LN1@f
    or     DWORD PTR __v, 1
$LN1@f:
    add    esp, 16       ; 00000010H
    ret    0
_f    ENDP
_TEXT    ENDS
```

73.6 Задача 2.6

73.6.1 MSVC 2010 + /Ox

```
PUBLIC    _f
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_k0$ = -12            ; size = 4
_k3$ = -8             ; size = 4
_k2$ = -4             ; size = 4
_v$ = 8               ; size = 4
_k1$ = 12             ; size = 4
_k$ = 12              ; size = 4
_f    PROC
```

```
    sub    esp, 12    ; 0000000cH
    mov    ecx, DWORD PTR _v$[esp+8]
    mov    eax, DWORD PTR [ecx]
    mov    ecx, DWORD PTR [ecx+4]
    push   ebx
    push   esi
    mov    esi, DWORD PTR _k$[esp+16]
    push   edi
    mov    edi, DWORD PTR [esi]
    mov    DWORD PTR _k0$[esp+24], edi
    mov    edi, DWORD PTR [esi+4]
    mov    DWORD PTR _k1$[esp+20], edi
    mov    edi, DWORD PTR [esi+8]
    mov    esi, DWORD PTR [esi+12]
    xor    edx, edx
    mov    DWORD PTR _k2$[esp+24], edi
    mov    DWORD PTR _k3$[esp+24], esi
    lea    edi, DWORD PTR [edx+32]
$LL8@f:
    mov    esi, ecx
    shr    esi, 5
    add    esi, DWORD PTR _k1$[esp+20]
    mov    ebx, ecx
    shl    ebx, 4
    add    ebx, DWORD PTR _k0$[esp+24]
    sub    edx, 1640531527   ; 61c88647H
    xor    esi, ebx
    lea    ebx, DWORD PTR [edx+ecx]
    xor    esi, ebx
    add    eax, esi
    mov    esi, eax
    shr    esi, 5
    add    esi, DWORD PTR _k3$[esp+24]
    mov    ebx, eax
    shl    ebx, 4
    add    ebx, DWORD PTR _k2$[esp+24]
    xor    esi, ebx
    lea    ebx, DWORD PTR [edx+eax]
    xor    esi, ebx
    add    ecx, esi
    dec    edi
    jne    SHORT $LL8@f
    mov    edx, DWORD PTR _v$[esp+20]
    pop    edi
    pop    esi
    mov    DWORD PTR [edx], eax
    mov    DWORD PTR [edx+4], ecx
    pop    ebx
    add    esp, 12                     ; 0000000cH
    ret    0
_f    ENDP
```

73.6.2 Keil (ARM) + -O3

```
        PUSH    {r4-r10,lr}
        ADD     r5,r1,#8
        LDM     r5,{r5,r7}
        LDR     r2,[r0,#4]
        LDR     r3,[r0,#0]
        LDR     r4,|L0.116|
```

```
        LDR     r6,[r1,#4]
        LDR     r8,[r1,#0]
        MOV     r12,#0
        MOV     r1,r12
|L0.40|
        ADD     r12,r12,r4
        ADD     r9,r8,r2,LSL #4
        ADD     r10,r2,r12
        EOR     r9,r9,r10
        ADD     r10,r6,r2,LSR #5
        EOR     r9,r9,r10
        ADD     r3,r3,r9
        ADD     r9,r5,r3,LSL #4
        ADD     r10,r3,r12
        EOR     r9,r9,r10
        ADD     r10,r7,r3,LSR #5
        EOR     r9,r9,r10
        ADD     r1,r1,#1
        CMP     r1,#0x20
        ADD     r2,r2,r9
        STRCS   r2,[r0,#4]
        STRCS   r3,[r0,#0]
        BCC     |L0.40|
        POP     {r4-r10,pc}

|L0.116|
        DCD     0x9e3779b9
```

73.6.3 Keil (thumb) + -O3

```
        PUSH    {r1-r7,lr}
        LDR     r5,|L0.84|
        LDR     r3,[r0,#0]
        LDR     r2,[r0,#4]
        STR     r5,[sp,#8]
        MOVS    r6,r1
        LDM     r6,{r6,r7}
        LDR     r5,[r1,#8]
        STR     r6,[sp,#4]
        LDR     r6,[r1,#0xc]
        MOVS    r4,#0
        MOVS    r1,r4
        MOV     lr,r5
        MOV     r12,r6
        STR     r7,[sp,#0]
|L0.30|
        LDR     r5,[sp,#8]
        LSLS    r6,r2,#4
        ADDS    r4,r4,r5
        LDR     r5,[sp,#4]
        LSRS    r7,r2,#5
        ADDS    r5,r6,r5
        ADDS    r6,r2,r4
        EORS    r5,r5,r6
        LDR     r6,[sp,#0]
        ADDS    r1,r1,#1
        ADDS    r6,r7,r6
        EORS    r5,r5,r6
        ADDS    r3,r5,r3
        LSLS    r5,r3,#4
```

```
        ADDS    r6,r3,r4
        ADD     r5,r5,lr
        EORS    r5,r5,r6
        LSRS    r6,r3,#5
        ADD     r6,r6,r12
        EORS    r5,r5,r6
        ADDS    r2,r5,r2
        CMP     r1,#0x20
        BCC     |L0.30|
        STR     r3,[r0,#0]
        STR     r2,[r0,#4]
        POP     {r1-r7,pc}

|L0.84|
        DCD     0x9e3779b9
```

73.7 Задача 2.7

Это взята функция из ядра Linux 2.6.

73.7.1 MSVC 2010 + /Ox

```
_table    db 000h, 080h, 040h, 0c0h, 020h, 0a0h, 060h, 0e0h
    db 010h, 090h, 050h, 0d0h, 030h, 0b0h, 070h, 0f0h
    db 008h, 088h, 048h, 0c8h, 028h, 0a8h, 068h, 0e8h
    db 018h, 098h, 058h, 0d8h, 038h, 0b8h, 078h, 0f8h
    db 004h, 084h, 044h, 0c4h, 024h, 0a4h, 064h, 0e4h
    db 014h, 094h, 054h, 0d4h, 034h, 0b4h, 074h, 0f4h
    db 00ch, 08ch, 04ch, 0cch, 02ch, 0ach, 06ch, 0ech
    db 01ch, 09ch, 05ch, 0dch, 03ch, 0bch, 07ch, 0fch
    db 002h, 082h, 042h, 0c2h, 022h, 0a2h, 062h, 0e2h
    db 012h, 092h, 052h, 0d2h, 032h, 0b2h, 072h, 0f2h
    db 00ah, 08ah, 04ah, 0cah, 02ah, 0aah, 06ah, 0eah
    db 01ah, 09ah, 05ah, 0dah, 03ah, 0bah, 07ah, 0fah
    db 006h, 086h, 046h, 0c6h, 026h, 0a6h, 066h, 0e6h
    db 016h, 096h, 056h, 0d6h, 036h, 0b6h, 076h, 0f6h
    db 00eh, 08eh, 04eh, 0ceh, 02eh, 0aeh, 06eh, 0eeh
    db 01eh, 09eh, 05eh, 0deh, 03eh, 0beh, 07eh, 0feh
    db 001h, 081h, 041h, 0c1h, 021h, 0a1h, 061h, 0e1h
    db 011h, 091h, 051h, 0d1h, 031h, 0b1h, 071h, 0f1h
    db 009h, 089h, 049h, 0c9h, 029h, 0a9h, 069h, 0e9h
    db 019h, 099h, 059h, 0d9h, 039h, 0b9h, 079h, 0f9h
    db 005h, 085h, 045h, 0c5h, 025h, 0a5h, 065h, 0e5h
    db 015h, 095h, 055h, 0d5h, 035h, 0b5h, 075h, 0f5h
    db 00dh, 08dh, 04dh, 0cdh, 02dh, 0adh, 06dh, 0edh
    db 01dh, 09dh, 05dh, 0ddh, 03dh, 0bdh, 07dh, 0fdh
    db 003h, 083h, 043h, 0c3h, 023h, 0a3h, 063h, 0e3h
    db 013h, 093h, 053h, 0d3h, 033h, 0b3h, 073h, 0f3h
    db 00bh, 08bh, 04bh, 0cbh, 02bh, 0abh, 06bh, 0ebh
    db 01bh, 09bh, 05bh, 0dbh, 03bh, 0bbh, 07bh, 0fbh
    db 007h, 087h, 047h, 0c7h, 027h, 0a7h, 067h, 0e7h
    db 017h, 097h, 057h, 0d7h, 037h, 0b7h, 077h, 0f7h
    db 00fh, 08fh, 04fh, 0cfh, 02fh, 0afh, 06fh, 0efh
    db 01fh, 09fh, 05fh, 0dfh, 03fh, 0bfh, 07fh, 0ffh

f               proc near

arg_0           = dword ptr  4
```

```
                mov     edx, [esp+arg_0]
                movzx   eax, dl
                movzx   eax, _table[eax]
                mov     ecx, edx
                shr     edx, 8
                movzx   edx, dl
                movzx   edx, _table[edx]
                shl     ax, 8
                movzx   eax, ax
                or      eax, edx
                shr     ecx, 10h
                movzx   edx, cl
                movzx   edx, _table[edx]
                shr     ecx, 8
                movzx   ecx, cl
                movzx   ecx, _table[ecx]
                shl     dx, 8
                movzx   edx, dx
                shl     eax, 10h
                or      edx, ecx
                or      eax, edx
                retn
f               endp
```

73.7.2 Keil (ARM) + -O3

```
f2 PROC
        LDR      r1,|L0.76|
        LDRB     r2,[r1,r0,LSR #8]
        AND      r0,r0,#0xff
        LDRB     r0,[r1,r0]
        ORR      r0,r2,r0,LSL #8
        BX       lr
        ENDP

f3 PROC
        MOV      r3,r0
        LSR      r0,r0,#16
        PUSH     {lr}
        BL       f2
        MOV      r12,r0
        LSL      r0,r3,#16
        LSR      r0,r0,#16
        BL       f2
        ORR      r0,r12,r0,LSL #16
        POP      {pc}
        ENDP

|L0.76|
        DCB      0x00,0x80,0x40,0xc0
        DCB      0x20,0xa0,0x60,0xe0
        DCB      0x10,0x90,0x50,0xd0
        DCB      0x30,0xb0,0x70,0xf0
        DCB      0x08,0x88,0x48,0xc8
        DCB      0x28,0xa8,0x68,0xe8
        DCB      0x18,0x98,0x58,0xd8
        DCB      0x38,0xb8,0x78,0xf8
        DCB      0x04,0x84,0x44,0xc4
        DCB      0x24,0xa4,0x64,0xe4
        DCB      0x14,0x94,0x54,0xd4
```

```
        DCB     0x34,0xb4,0x74,0xf4
        DCB     0x0c,0x8c,0x4c,0xcc
        DCB     0x2c,0xac,0x6c,0xec
        DCB     0x1c,0x9c,0x5c,0xdc
        DCB     0x3c,0xbc,0x7c,0xfc
        DCB     0x02,0x82,0x42,0xc2
        DCB     0x22,0xa2,0x62,0xe2
        DCB     0x12,0x92,0x52,0xd2
        DCB     0x32,0xb2,0x72,0xf2
        DCB     0x0a,0x8a,0x4a,0xca
        DCB     0x2a,0xaa,0x6a,0xea
        DCB     0x1a,0x9a,0x5a,0xda
        DCB     0x3a,0xba,0x7a,0xfa
        DCB     0x06,0x86,0x46,0xc6
        DCB     0x26,0xa6,0x66,0xe6
        DCB     0x16,0x96,0x56,0xd6
        DCB     0x36,0xb6,0x76,0xf6
        DCB     0x0e,0x8e,0x4e,0xce
        DCB     0x2e,0xae,0x6e,0xee
        DCB     0x1e,0x9e,0x5e,0xde
        DCB     0x3e,0xbe,0x7e,0xfe
        DCB     0x01,0x81,0x41,0xc1
        DCB     0x21,0xa1,0x61,0xe1
        DCB     0x11,0x91,0x51,0xd1
        DCB     0x31,0xb1,0x71,0xf1
        DCB     0x09,0x89,0x49,0xc9
        DCB     0x29,0xa9,0x69,0xe9
        DCB     0x19,0x99,0x59,0xd9
        DCB     0x39,0xb9,0x79,0xf9
        DCB     0x05,0x85,0x45,0xc5
        DCB     0x25,0xa5,0x65,0xe5
        DCB     0x15,0x95,0x55,0xd5
        DCB     0x35,0xb5,0x75,0xf5
        DCB     0x0d,0x8d,0x4d,0xcd
        DCB     0x2d,0xad,0x6d,0xed
        DCB     0x1d,0x9d,0x5d,0xdd
        DCB     0x3d,0xbd,0x7d,0xfd
        DCB     0x03,0x83,0x43,0xc3
        DCB     0x23,0xa3,0x63,0xe3
        DCB     0x13,0x93,0x53,0xd3
        DCB     0x33,0xb3,0x73,0xf3
        DCB     0x0b,0x8b,0x4b,0xcb
        DCB     0x2b,0xab,0x6b,0xeb
        DCB     0x1b,0x9b,0x5b,0xdb
        DCB     0x3b,0xbb,0x7b,0xfb
        DCB     0x07,0x87,0x47,0xc7
        DCB     0x27,0xa7,0x67,0xe7
        DCB     0x17,0x97,0x57,0xd7
        DCB     0x37,0xb7,0x77,0xf7
        DCB     0x0f,0x8f,0x4f,0xcf
        DCB     0x2f,0xaf,0x6f,0xef
        DCB     0x1f,0x9f,0x5f,0xdf
        DCB     0x3f,0xbf,0x7f,0xff
```

73.7.3 Keil (thumb) + -O3

```
f2 PROC
        LDR     r1,|L0.48|
        LSLS    r2,r0,#24
        LSRS    r2,r2,#24
```

```
        LDRB     r2,[r1,r2]
        LSLS     r2,r2,#8
        LSRS     r0,r0,#8
        LDRB     r0,[r1,r0]
        ORRS     r0,r0,r2
        BX       lr
        ENDP

f3 PROC
        MOVS     r3,r0
        LSLS     r0,r0,#16
        PUSH     {r4,lr}
        LSRS     r0,r0,#16
        BL       f2
        LSLS     r4,r0,#16
        LSRS     r0,r3,#16
        BL       f2
        ORRS     r0,r0,r4
        POP      {r4,pc}
        ENDP

|L0.48|
        DCB      0x00,0x80,0x40,0xc0
        DCB      0x20,0xa0,0x60,0xe0
        DCB      0x10,0x90,0x50,0xd0
        DCB      0x30,0xb0,0x70,0xf0
        DCB      0x08,0x88,0x48,0xc8
        DCB      0x28,0xa8,0x68,0xe8
        DCB      0x18,0x98,0x58,0xd8
        DCB      0x38,0xb8,0x78,0xf8
        DCB      0x04,0x84,0x44,0xc4
        DCB      0x24,0xa4,0x64,0xe4
        DCB      0x14,0x94,0x54,0xd4
        DCB      0x34,0xb4,0x74,0xf4
        DCB      0x0c,0x8c,0x4c,0xcc
        DCB      0x2c,0xac,0x6c,0xec
        DCB      0x1c,0x9c,0x5c,0xdc
        DCB      0x3c,0xbc,0x7c,0xfc
        DCB      0x02,0x82,0x42,0xc2
        DCB      0x22,0xa2,0x62,0xe2
        DCB      0x12,0x92,0x52,0xd2
        DCB      0x32,0xb2,0x72,0xf2
        DCB      0x0a,0x8a,0x4a,0xca
        DCB      0x2a,0xaa,0x6a,0xea
        DCB      0x1a,0x9a,0x5a,0xda
        DCB      0x3a,0xba,0x7a,0xfa
        DCB      0x06,0x86,0x46,0xc6
        DCB      0x26,0xa6,0x66,0xe6
        DCB      0x16,0x96,0x56,0xd6
        DCB      0x36,0xb6,0x76,0xf6
        DCB      0x0e,0x8e,0x4e,0xce
        DCB      0x2e,0xae,0x6e,0xee
        DCB      0x1e,0x9e,0x5e,0xde
        DCB      0x3e,0xbe,0x7e,0xfe
        DCB      0x01,0x81,0x41,0xc1
        DCB      0x21,0xa1,0x61,0xe1
        DCB      0x11,0x91,0x51,0xd1
        DCB      0x31,0xb1,0x71,0xf1
        DCB      0x09,0x89,0x49,0xc9
        DCB      0x29,0xa9,0x69,0xe9
        DCB      0x19,0x99,0x59,0xd9
```

```
        DCB     0x39,0xb9,0x79,0xf9
        DCB     0x05,0x85,0x45,0xc5
        DCB     0x25,0xa5,0x65,0xe5
        DCB     0x15,0x95,0x55,0xd5
        DCB     0x35,0xb5,0x75,0xf5
        DCB     0x0d,0x8d,0x4d,0xcd
        DCB     0x2d,0xad,0x6d,0xed
        DCB     0x1d,0x9d,0x5d,0xdd
        DCB     0x3d,0xbd,0x7d,0xfd
        DCB     0x03,0x83,0x43,0xc3
        DCB     0x23,0xa3,0x63,0xe3
        DCB     0x13,0x93,0x53,0xd3
        DCB     0x33,0xb3,0x73,0xf3
        DCB     0x0b,0x8b,0x4b,0xcb
        DCB     0x2b,0xab,0x6b,0xeb
        DCB     0x1b,0x9b,0x5b,0xdb
        DCB     0x3b,0xbb,0x7b,0xfb
        DCB     0x07,0x87,0x47,0xc7
        DCB     0x27,0xa7,0x67,0xe7
        DCB     0x17,0x97,0x57,0xd7
        DCB     0x37,0xb7,0x77,0xf7
        DCB     0x0f,0x8f,0x4f,0xcf
        DCB     0x2f,0xaf,0x6f,0xef
        DCB     0x1f,0x9f,0x5f,0xdf
        DCB     0x3f,0xbf,0x7f,0xff
```

73.8 Задача 2.8

73.8.1 MSVC 2010 + /O1

(/O1: оптимизация по размеру кода).

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
?s@@YAXPAN00@Z PROC     ; s, COMDAT
    mov    eax, DWORD PTR _b$[esp-4]
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    edx, DWORD PTR _c$[esp-4]
    push   esi
    push   edi
    sub    ecx, eax
    sub    edx, eax
    mov    edi, 200     ; 000000c8H
$LL6@s:
    push   100          ; 00000064H
    pop    esi
$LL3@s:
    fld    QWORD PTR [ecx+eax]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [edx+eax]
    add    eax, 8
    dec    esi
    jne    SHORT $LL3@s
    dec    edi
    jne    SHORT $LL6@s
    pop    edi
    pop    esi
    ret    0
?s@@YAXPAN00@Z ENDP     ; s
```

73.8.2 Keil (ARM) + -O3

```
        PUSH    {r4-r12,lr}
        MOV     r9,r2
        MOV     r10,r1
        MOV     r11,r0
        MOV     r5,#0
|L0.20|
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        MOV     r4,#0
        ADD     r8,r10,r0,LSL #5
        ADD     r7,r11,r0,LSL #5
        ADD     r6,r9,r0,LSL #5
|L0.44|
        ADD     r0,r8,r4,LSL #3
        LDM     r0,{r2,r3}
        ADD     r1,r7,r4,LSL #3
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADD     r2,r6,r4,LSL #3
        ADD     r4,r4,#1
        STM     r2,{r0,r1}
        CMP     r4,#0x64
        BLT     |L0.44|
        ADD     r5,r5,#1
        CMP     r5,#0xc8
        BLT     |L0.20|
        POP     {r4-r12,pc}
```

73.8.3 Keil (thumb) + -O3

```
        PUSH    {r0-r2,r4-r7,lr}
        MOVS    r4,#0
        SUB     sp,sp,#8
|L0.6|
        MOVS    r1,#0x19
        MOVS    r0,r4
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#8]
        LDR     r1,[sp,#0xc]
        ADDS    r2,r0,r2
        STR     r2,[sp,#0]
        LDR     r2,[sp,#0x10]
        MOVS    r5,#0
        ADDS    r7,r0,r2
        ADDS    r0,r0,r1
        STR     r0,[sp,#4]
|L0.32|
        LSLS    r6,r5,#3
        ADDS    r0,r0,r6
        LDM     r0!,{r2,r3}
        LDR     r0,[sp,#0]
        ADDS    r1,r0,r6
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADDS    r2,r7,r6
        ADDS    r5,r5,#1
        STM     r2!,{r0,r1}
```

```
        CMP     r5,#0x64
        BGE     |L0.62|
        LDR     r0,[sp,#4]
        B       |L0.32|
|L0.62|
        ADDS    r4,r4,#1
        CMP     r4,#0xc8
        BLT     |L0.6|
        ADD     sp,sp,#0x14
        POP     {r4-r7,pc}
```

73.9 Задача 2.9

73.9.1 MSVC 2010 + /O1

(/O1: оптимизация по размеру кода).

```
tv315 = -8              ; size = 4
tv291 = -4              ; size = 4
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_c$ = 16                ; size = 4
?m@@YAXPAN00@Z PROC     ; m, COMDAT
    push   ebp
    mov    ebp, esp
    push   ecx
    push   ecx
    mov    edx, DWORD PTR _a$[ebp]
    push   ebx
    mov    ebx, DWORD PTR _c$[ebp]
    push   esi
    mov    esi, DWORD PTR _b$[ebp]
    sub    edx, esi
    push   edi
    sub    esi, ebx
    mov    DWORD PTR tv315[ebp], 100  ; 00000064H
$LL9@m:
    mov    eax, ebx
    mov    DWORD PTR tv291[ebp], 300  ; 0000012cH
$LL6@m:
    fldz
    lea    ecx, DWORD PTR [esi+eax]
    fstp   QWORD PTR [eax]
    mov    edi, 200                   ; 000000c8H
$LL3@m:
    dec    edi
    fld    QWORD PTR [ecx+edx]
    fmul   QWORD PTR [ecx]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [eax]
    jne    HORT $LL3@m
    add    eax, 8
    dec    DWORD PTR tv291[ebp]
    jne    SHORT $LL6@m
    add    ebx, 800                   ; 00000320H
    dec    DWORD PTR tv315[ebp]
    jne    SHORT $LL9@m
    pop    edi
    pop    esi
    pop    ebx
```

```
    leave
    ret    0
?m@@YAXPAN00@Z ENDP                 ; m
```

73.9.2 Keil (ARM) + -O3

```
        PUSH    {r0-r2,r4-r11,lr}
        SUB     sp,sp,#8
        MOV     r5,#0
|L0.12|
        LDR     r1,[sp,#0xc]
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        ADD     r1,r1,r0,LSL #5
        STR     r1,[sp,#0]
        LDR     r1,[sp,#8]
        MOV     r4,#0
        ADD     r11,r1,r0,LSL #5
        LDR     r1,[sp,#0x10]
        ADD     r10,r1,r0,LSL #5
|L0.52|
        MOV     r0,#0
        MOV     r1,r0
        ADD     r7,r10,r4,LSL #3
        STM     r7,{r0,r1}
        MOV     r6,r0
        LDR     r0,[sp,#0]
        ADD     r8,r11,r4,LSL #3
        ADD     r9,r0,r4,LSL #3
|L0.84|
        LDM     r9,{r2,r3}
        LDM     r8,{r0,r1}
        BL      __aeabi_dmul
        LDM     r7,{r2,r3}
        BL      __aeabi_dadd
        ADD     r6,r6,#1
        STM     r7,{r0,r1}
        CMP     r6,#0xc8
        BLT     |L0.84|
        ADD     r4,r4,#1
        CMP     r4,#0x12c
        BLT     |L0.52|
        ADD     r5,r5,#1
        CMP     r5,#0x64
        BLT     |L0.12|
        ADD     sp,sp,#0x14
        POP     {r4-r11,pc}
```

73.9.3 Keil (thumb) + -O3

```
        PUSH    {r0-r2,r4-r7,lr}
        MOVS    r0,#0
        SUB     sp,sp,#0x10
        STR     r0,[sp,#0]
|L0.8|
        MOVS    r1,#0x19
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#0x10]
```

```
        LDR     r1,[sp,#0x14]
        ADDS    r2,r0,r2
        STR     r2,[sp,#4]
        LDR     r2,[sp,#0x18]
        MOVS    r5,#0
        ADDS    r7,r0,r2
        ADDS    r0,r0,r1
        STR     r0,[sp,#8]
|L0.32|
        LSLS    r4,r5,#3
        MOVS    r0,#0
        ADDS    r2,r7,r4
        STR     r0,[r2,#0]
        MOVS    r6,r0
        STR     r0,[r2,#4]
|L0.44|
        LDR     r0,[sp,#8]
        ADDS    r0,r0,r4
        LDM     r0!,{r2,r3}
        LDR     r0,[sp,#4]
        ADDS    r1,r0,r4
        LDM     r1,{r0,r1}
        BL      __aeabi_dmul
        ADDS    r3,r7,r4
        LDM     r3,{r2,r3}
        BL      __aeabi_dadd
        ADDS    r2,r7,r4
        ADDS    r6,r6,#1
        STM     r2!,{r0,r1}
        CMP     r6,#0xc8
        BLT     |L0.44|
        MOVS    r0,#0xff
        ADDS    r5,r5,#1
        ADDS    r0,r0,#0x2d
        CMP     r5,r0
        BLT     |L0.32|
        LDR     r0,[sp,#0]
        ADDS    r0,r0,#1
        CMP     r0,#0x64
        STR     r0,[sp,#0]
        BLT     |L0.8|
        ADD     sp,sp,#0x1c
        POP     {r4-r7,pc}
```

73.10 Задача 2.10

Если это скомпилировать и запустить, появится некоторое число. Откуда оно берется? Откуда оно берется если скомпилировать в MSVC с оптимизациями (`/Ox`)?

```
#include <stdio.h>

int main()
{
        printf ("%d\n");

        return 0;
};
```

73.11 Задача 2.11

В рамках шутки, “обманите” ваш Windows Task Manager чтобы он показывал больше процессоров/ядер процессоров чем есть в вашем компьютере на самом деле:

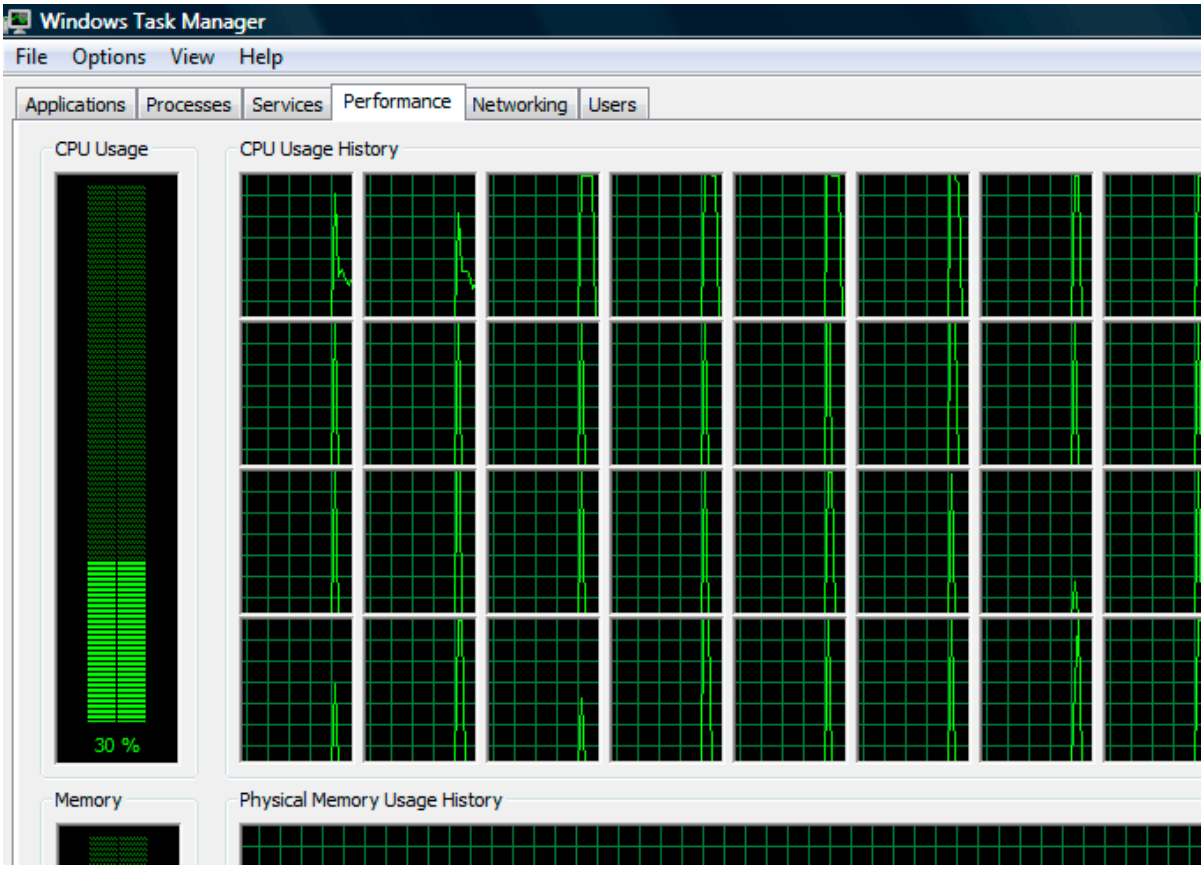


Рис. 73.1: Обманутый Windows Task Manager

73.12 Задача 2.12

Это довольно известный алгоритм. Как он называется?

73.12.1 MSVC 2012 x64 + /Ox

```
s$ = 8
f       PROC
        cmp     BYTE PTR [rcx], 0
        mov     r9, rcx
        je      SHORT $LN13@f
        npad    8
$LL5@f:
        movzx   edx, BYTE PTR [rcx]
        lea     eax, DWORD PTR [rdx-97]
        cmp     al, 25
        ja      SHORT $LN3@f
        movsx   r8d, dl
        mov     eax, 1321528399                         ; 4ec4ec4fH
        sub     r8d, 84                                 ; 00000054H
        imul    r8d
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31
        add     edx, eax
```

```
        imul    edx, 26
        sub     r8d, edx
        add     r8b, 97                                 ; 00000061H
        jmp     SHORT $LN14@f
$LN3@f:
        lea     eax, DWORD PTR [rdx-65]
        cmp     al, 25
        ja      SHORT $LN1@f
        movsx   r8d, dl
        mov     eax, 1321528399                         ; 4ec4ec4fH
        sub     r8d, 52                                 ; 00000034H
        imul    r8d
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31
        add     edx, eax
        imul    edx, 26
        sub     r8d, edx
        add     r8b, 65                                 ; 00000041H
$LN14@f:
        mov     BYTE PTR [rcx], r8b
$LN1@f:
        inc     rcx
        cmp     BYTE PTR [rcx], 0
        jne     SHORT $LL5@f
$LN13@f:
        mov     rax, r9
        ret     0
f       ENDP
```

73.12.2 Keil (ARM)

```
f PROC
        PUSH    {r4-r6,lr}
        MOV     r4,r0
        MOV     r5,r0
        B       |L0.84|
|L0.16|
        SUB     r1,r0,#0x61
        CMP     r1,#0x19
        BHI     |L0.48|
        SUB     r0,r0,#0x54
        MOV     r1,#0x1a
        BL      __aeabi_idivmod
        ADD     r0,r1,#0x61
        B       |L0.76|
|L0.48|
        SUB     r1,r0,#0x41
        CMP     r1,#0x19
        BHI     |L0.80|
        SUB     r0,r0,#0x34
        MOV     r1,#0x1a
        BL      __aeabi_idivmod
        ADD     r0,r1,#0x41
|L0.76|
        STRB    r0,[r4,#0]
|L0.80|
        ADD     r4,r4,#1
|L0.84|
        LDRB    r0,[r4,#0]
```

```
        CMP     r0,#0
        MOVEQ   r0,r5
        BNE     |L0.16|
        POP     {r4-r6,pc}
        ENDP
```

73.12.3 Keil (thumb)

```
f PROC
        PUSH    {r4-r6,lr}
        MOVS    r4,r0
        MOVS    r5,r0
        B       |L0.50|
|L0.8|
        MOVS    r1,r0
        SUBS    r1,r1,#0x61
        CMP     r1,#0x19
        BHI     |L0.28|
        SUBS    r0,r0,#0x54
        MOVS    r1,#0x1a
        BL      __aeabi_idivmod
        ADDS    r1,r1,#0x61
        B       |L0.46|
|L0.28|
        MOVS    r1,r0
        SUBS    r1,r1,#0x41
        CMP     r1,#0x19
        BHI     |L0.48|
        SUBS    r0,r0,#0x34
        MOVS    r1,#0x1a
        BL      __aeabi_idivmod
        ADDS    r1,r1,#0x41
|L0.46|
        STRB    r1,[r4,#0]
|L0.48|
        ADDS    r4,r4,#1
|L0.50|
        LDRB    r0,[r4,#0]
        CMP     r0,#0
        BNE     |L0.8|
        MOVS    r0,r5
        POP     {r4-r6,pc}
        ENDP
```

73.13 Задача 2.13

Это довольно известный криптоалгоритм прошлого. Как он называется?

73.13.1 MSVC 2012 + /Ox

```
_in$ = 8                                                ; size = 2
_f      PROC
        movzx   ecx, WORD PTR _in$[esp-4]
        lea     eax, DWORD PTR [ecx*4]
        xor     eax, ecx
        add     eax, eax
        xor     eax, ecx
        shl     eax, 2
```

```
        xor     eax, ecx
        and     eax, 32                                 ; 00000020H
        shl     eax, 10                                 ; 0000000aH
        shr     ecx, 1
        or      eax, ecx
        ret     0
_f      ENDP
```

73.13.2 Keil (ARM)

```
f PROC
        EOR      r1,r0,r0,LSR #2
        EOR      r1,r1,r0,LSR #3
        EOR      r1,r1,r0,LSR #5
        AND      r1,r1,#1
        LSR      r0,r0,#1
        ORR      r0,r0,r1,LSL #15
        BX       lr
        ENDP
```

73.13.3 Keil (thumb)

```
f PROC
        LSRS     r1,r0,#2
        EORS     r1,r1,r0
        LSRS     r2,r0,#3
        EORS     r1,r1,r2
        LSRS     r2,r0,#5
        EORS     r1,r1,r2
        LSLS     r1,r1,#31
        LSRS     r0,r0,#1
        LSRS     r1,r1,#16
        ORRS     r0,r0,r1
        BX       lr
        ENDP
```

73.14 Задача 2.14

Еще один хорошо известный алгоритм. Ф-ция берет на вход 2 значения и возвращает одно.

73.14.1 MSVC 2012

```
_rt$1 = -4                                              ; size = 4
_rt$2 = 8                                               ; size = 4
_x$ = 8                                                 ; size = 4
_y$ = 12                                                ; size = 4
?f@@YAIII@Z PROC                                        ; f
        push    ecx
        push    esi
        mov     esi, DWORD PTR _x$[esp+4]
        test    esi, esi
        jne     SHORT $LN7@f
        mov     eax, DWORD PTR _y$[esp+4]
        pop     esi
        pop     ecx
        ret     0
$LN7@f:
```

```
        mov     edx, DWORD PTR _y$[esp+4]
        mov     eax, esi
        test    edx, edx
        je      SHORT $LN8@f
        or      eax, edx
        push    edi
        bsf     edi, eax
        bsf     eax, esi
        mov     ecx, eax
        mov     DWORD PTR _rt$1[esp+12], eax
        bsf     eax, edx
        shr     esi, cl
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        je      SHORT $LN22@f
$LN23@f:
        jbe     SHORT $LN2@f
        xor     esi, edx
        xor     edx, esi
        xor     esi, edx
$LN2@f:
        cmp     esi, 1
        je      SHORT $LN22@f
        sub     edx, esi
        bsf     eax, edx
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        jne     SHORT $LN23@f
$LN22@f:
        mov     ecx, edi
        shl     esi, cl
        pop     edi
        mov     eax, esi
$LN8@f:
        pop     esi
        pop     ecx
        ret     0
?f@@YAIII@Z ENDP
```

73.14.2 Keil (ARM mode)

```
||f1|| PROC
        CMP      r0,#0
        RSB      r1,r0,#0
        AND      r0,r0,r1
        CLZ      r0,r0
        RSBNE    r0,r0,#0x1f
        BX       lr
        ENDP

f PROC
        MOVS     r2,r0
        MOV      r3,r1
        MOVEQ    r0,r1
        CMPNE    r3,#0
        PUSH     {lr}
```

```
        POPEQ   {pc}
        ORR     r0,r2,r3
        BL      ||f1||
        MOV     r12,r0
        MOV     r0,r2
        BL      ||f1||
        LSR     r2,r2,r0
|L0.196|
        MOV     r0,r3
        BL      ||f1||
        LSR     r0,r3,r0
        CMP     r2,r0
        EORHI   r1,r2,r0
        EORHI   r0,r0,r1
        EORHI   r2,r1,r0
        BEQ     |L0.240|
        CMP     r2,#1
        SUBNE   r3,r0,r2
        BNE     |L0.196|
|L0.240|
        LSL     r0,r2,r12
        POP     {pc}
        ENDP
```

73.14.3 GCC 4.6.3 for Raspberry Pi (ARM mode)

```
f:
        subs    r3, r0, #0
        beq     .L162
        cmp     r1, #0
        moveq   r1, r3
        beq     .L162
        orr     r2, r1, r3
        rsb     ip, r2, #0
        and     ip, ip, r2
        cmp     r2, #0
        rsb     r2, r3, #0
        and     r2, r2, r3
        clz     r2, r2
        rsb     r2, r2, #31
        clz     ip, ip
        rsbne   ip, ip, #31
        mov     r3, r3, lsr r2
        b       .L169
.L171:
        eorhi   r1, r1, r2
        eorhi   r3, r1, r2
        cmp     r3, #1
        rsb     r1, r3, r1
        beq     .L167
.L169:
        rsb     r0, r1, #0
        and     r0, r0, r1
        cmp     r1, #0
        clz     r0, r0
        mov     r2, r0
        rsbne   r2, r0, #31
        mov     r1, r1, lsr r2
        cmp     r3, r1
        eor     r2, r1, r3
```

```
        bne     .L171
.L167:
        mov     r1, r3, asl ip
.L162:
        mov     r0, r1
        bx      lr
```

73.15 Задача 2.15

И снова известный алгоритм. Что он делает?

Обратите внимание, что код для x86 использует FPU, а для x64 — SIMD-инструкции. Это нормально: 24.

73.15.1 MSVC 2012 x64 /Ox

```
__real@412e848000000000 DQ 0412e848000000000r   ; 1e+006
__real@4010000000000000 DQ 0401000000000000r   ; 4
__real@4008000000000000 DQ 0400800000000000r   ; 3
__real@3f800000 DD 03f800000r                   ; 1

tmp$1 = 8
tmp$2 = 8
f       PROC
        movsdx  xmm3, QWORD PTR __real@4008000000000000
        movss   xmm4, DWORD PTR __real@3f800000
        mov     edx, DWORD PTR ?RNG_state@?1??get_rand@@9@9
        xor     ecx, ecx
        mov     r8d, 200000                             ; 00030d40H
        npad    2
$LL4@f:
        imul    edx, 1664525                            ; 0019660dH
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        imul    edx, 1664525                            ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN3@f
        inc     ecx
$LN3@f:
        imul    edx, 1664525                            ; 0019660dH
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     eax, edx
```

```
        and     eax, 8388607                            ; 007fffffH
        imul    edx, 1664525                            ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN15@f
        inc     ecx
$LN15@f:
        imul    edx, 1664525                            ; 0019660dH
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        imul    edx, 1664525                            ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN16@f
        inc     ecx
$LN16@f:
        imul    edx, 1664525                            ; 0019660dH
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        imul    edx, 1664525                            ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
```

```
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN17@f
        inc     ecx
$LN17@f:
        imul    edx, 1664525                            ; 0019660dH
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        imul    edx, 1664525                            ; 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN18@f
        inc     ecx
$LN18@f:
        dec     r8
        jne     $LL4@f
        movd    xmm0, ecx
        mov     DWORD PTR ?RNG_state@?1??get_rand@@9@9, edx
        cvtdq2ps xmm0, xmm0
        cvtps2pd xmm1, xmm0
        mulsd   xmm1, QWORD PTR __real@4010000000000000
        divsd   xmm1, QWORD PTR __real@412e848000000000
        cvtpd2ps xmm0, xmm1
        ret     0
f       ENDP
```

73.15.2 GCC 4.4.6 -O3 x64

```
f1:
        mov     eax, DWORD PTR v1.2084[rip]
```

```
        imul    eax, eax, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2084[rip], eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [rsp-4], eax
        movss   xmm0, DWORD PTR [rsp-4]
        subss   xmm0, DWORD PTR .LC0[rip]
        ret
f:
        push    rbp
        xor     ebp, ebp
        push    rbx
        xor     ebx, ebx
        sub     rsp, 16
.L6:
        xor     eax, eax
        call    f1
        xor     eax, eax
        movss   DWORD PTR [rsp], xmm0
        call    f1
        movss   xmm1, DWORD PTR [rsp]
        mulss   xmm0, xmm0
        mulss   xmm1, xmm1
        lea     eax, [rbx+1]
        addss   xmm1, xmm0
        movss   xmm0, DWORD PTR .LC1[rip]
        ucomiss xmm0, xmm1
        cmova   ebx, eax
        add     ebp, 1
        cmp     ebp, 1000000
        jne     .L6
        cvtsi2ss        xmm0, ebx
        unpcklps        xmm0, xmm0
        cvtps2pd        xmm0, xmm0
        mulsd   xmm0, QWORD PTR .LC2[rip]
        divsd   xmm0, QWORD PTR .LC3[rip]
        add     rsp, 16
        pop     rbx
        pop     rbp
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
v1.2084:
        .long   305419896
.LC0:
        .long   1077936128
.LC1:
        .long   1065353216
.LC2:
        .long   0
        .long   1074790400
.LC3:
        .long   0
        .long   1093567616
```

73.15.3 GCC 4.8.1 -O3 x86

```
f1:
        sub     esp, 4
```

```
        imul    eax, DWORD PTR v1.2023, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2023, eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [esp], eax
        fld     DWORD PTR [esp]
        fsub    DWORD PTR .LC0
        add     esp, 4
        ret
f:
        push    esi
        mov     esi, 1000000
        push    ebx
        xor     ebx, ebx
        sub     esp, 16
.L7:
        call    f1
        fstp    DWORD PTR [esp]
        call    f1
        lea     eax, [ebx+1]
        fld     DWORD PTR [esp]
        fmul    st, st(0)
        fxch    st(1)
        fmul    st, st(0)
        faddp   st(1), st
        fld1
        fucomip st, st(1)
        fstp    st(0)
        cmova   ebx, eax
        sub     esi, 1
        jne     .L7
        mov     DWORD PTR [esp+4], ebx
        fild    DWORD PTR [esp+4]
        fmul    DWORD PTR .LC3
        fdiv    DWORD PTR .LC4
        fstp    DWORD PTR [esp+8]
        fld     DWORD PTR [esp+8]
        add     esp, 16
        pop     ebx
        pop     esi
        ret

v1.2023:
        .long   305419896
.LC0:
        .long   1077936128
.LC3:
        .long   1082130432
.LC4:
        .long   1232348160
```

73.15.4 Keil (ARM mode): для процессора Cortex-R4F

```
f1      PROC
        LDR      r1,|L0.184|
        LDR      r0,[r1,#0]  ; v1
        LDR      r2,|L0.188|
        VMOV.F32 s1,#3.00000000
        MUL      r0,r0,r2
```

```
        LDR      r2,|L0.192|
        ADD      r0,r0,r2
        STR      r0,[r1,#0]  ; v1
        BFC      r0,#23,#9
        ORR      r0,r0,#0x40000000
        VMOV     s0,r0
        VSUB.F32 s0,s0,s1
        BX       lr
        ENDP

f       PROC
        PUSH     {r4,r5,lr}
        MOV      r4,#0
        LDR      r5,|L0.196|
        MOV      r3,r4
|L0.68|
        BL       f1
        VMOV.F32 s2,s0
        BL       f1
        VMOV.F32 s1,s2
        ADD      r3,r3,#1
        VMUL.F32 s1,s1,s1
        VMLA.F32 s1,s0,s0
        VMOV     r0,s1
        CMP      r0,#0x3f800000
        ADDLT    r4,r4,#1
        CMP      r3,r5
        BLT      |L0.68|
        VMOV     s0,r4
        VMOV.F64 d1,#4.00000000
        VCVT.F32.S32 s0,s0
        VCVT.F64.F32 d0,s0
        VMUL.F64 d0,d0,d1
        VLDR     d1,|L0.200|
        VDIV.F64 d2,d0,d1
        VCVT.F32.F64 s0,d2
        POP      {r4,r5,pc}
        ENDP

|L0.184|
        DCD      ||.data||
|L0.188|
        DCD      0x0019660d
|L0.192|
        DCD      0x3c6ef35f
|L0.196|
        DCD      0x000f4240
|L0.200|
        DCFD     0x412e848000000000 ; 1000000

        DCD      0x00000000
        AREA ||.data||, DATA, ALIGN=2
v1
        DCD      0x12345678
```

73.16 Задача 2.16

Известная функция. Что она вычисляет? Почему стек переполняется если на вход подать числа 4 и 2? Есть ли здесь какая-то ошибка?

73.16.1 MSVC 2012 x64 /Ox

```
m$ = 48
n$ = 56
f       PROC
$LN14:
        push    rbx
        sub     rsp, 32
        mov     eax, edx
        mov     ebx, ecx
        test    ecx, ecx
        je      SHORT $LN11@f
$LL5@f:
        test    eax, eax
        jne     SHORT $LN1@f
        mov     eax, 1
        jmp     SHORT $LN12@f
$LN1@f:
        lea     edx, DWORD PTR [rax-1]
        mov     ecx, ebx
        call    f
$LN12@f:
        dec     ebx
        test    ebx, ebx
        jne     SHORT $LL5@f
$LN11@f:
        inc     eax
        add     rsp, 32
        pop     rbx
        ret     0
f       ENDP
```

73.16.2 Keil (ARM) -O3

```
f PROC
        PUSH    {r4,lr}
        MOVS    r4,r0
        ADDEQ   r0,r1,#1
        POPEQ   {r4,pc}
        CMP     r1,#0
        MOVEQ   r1,#1
        SUBEQ   r0,r0,#1
        BEQ     |L0.48|
        SUB     r1,r1,#1
        BL      f
        MOV     r1,r0
        SUB     r0,r4,#1
|L0.48|
        POP     {r4,lr}
        B       f
        ENDP
```

73.16.3 Keil (thumb) -O3

```
f PROC
        PUSH    {r4,lr}
        MOVS    r4,r0
        BEQ     |L0.26|
        CMP     r1,#0
```

```
        BEQ     |L0.30|
        SUBS    r1,r1,#1
        BL      f
        MOVS    r1,r0
|L0.18|
        SUBS    r0,r4,#1
        BL      f
        POP     {r4,pc}
|L0.26|
        ADDS    r0,r1,#1
        POP     {r4,pc}
|L0.30|
        MOVS    r1,#1
        B       |L0.18|
        ENDP
```

73.17 Задача 2.17

Эта программа выдает в *stdout* какую-то информацию, каждый раз — разную. Что это?

Скомпилированные бинарные файлы:

- Linux x64: http://yurichev.com/RE-exercises/2/17/17_Linux_x64.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/17/17_MacOSX_x64.tar
- Win32: http://yurichev.com/RE-exercises/2/17/17_win32.exe
- Win64: http://yurichev.com/RE-exercises/2/17/17_win64.exe

Для версий под Windows, возможно, нужно будет установить MSVC 2012 redist.

73.18 Задача 2.18

Эта программа запрашивает пароль. Найдите его.

Кстати, не только один пароль может подойти. Попробуйте найти еще.

Как дополнительное упражнение, попробуйте изменить пароль модифицируя исполняемый файл.

- Win32: http://yurichev.com/RE-exercises/2/18/password2.exe
- Linux x86: http://yurichev.com/RE-exercises/2/18/password2_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/18/password2_MacOSX64.tar

73.19 Задача 2.19

То же что и в упражнении 2.18.

- Win32: http://yurichev.com/RE-exercises/2/19/password3.exe
- Linux x86: http://yurichev.com/RE-exercises/2/19/password3_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/19/password3_MacOSX64.tar

Глава 74

Уровень 3

Для решения задач третьего уровня вам придется потратить какое-то ощутимое время, вплоть до одного дня.

74.1 Задача 3.1

Довольно известный алгоритм, так же включен в стандартную библиотеку Си. Исходник взят из glibc 2.11.1. Скомпилирован в GCC 4.4.1 с ключом `-Os` (оптимизация по размеру кода). Листинг сделан дизассемблером IDA 4.9 из ELF-файла созданным GCC и линкером.

Для тех кто хочет использовать IDA в процессе изучения, вот здесь лежат .elf и .idb файлы, .idb можно открыть при помощи бесплатной IDA 4.9:
http://yurichev.com/RE-exercises/3/1/

```
f               proc near

var_150         = dword ptr -150h
var_14C         = dword ptr -14Ch
var_13C         = dword ptr -13Ch
var_138         = dword ptr -138h
var_134         = dword ptr -134h
var_130         = dword ptr -130h
var_128         = dword ptr -128h
var_124         = dword ptr -124h
var_120         = dword ptr -120h
var_11C         = dword ptr -11Ch
var_118         = dword ptr -118h
var_114         = dword ptr -114h
var_110         = dword ptr -110h
var_C           = dword ptr -0Ch
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h
arg_10          = dword ptr  18h

                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                sub     esp, 14Ch
                mov     ebx, [ebp+arg_8]
                cmp     [ebp+arg_4], 0
                jz      loc_804877D
                cmp     [ebp+arg_4], 4
                lea     eax, ds:0[ebx*4]
                mov     [ebp+var_130], eax
                jbe     loc_804864C
```

```
                mov     eax, [ebp+arg_4]
                mov     ecx, ebx
                mov     esi, [ebp+arg_0]
                lea     edx, [ebp+var_110]
                neg     ecx
                mov     [ebp+var_118], 0
                mov     [ebp+var_114], 0
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                mov     [ebp+var_11C], edx
                mov     [ebp+var_134], ecx
                mov     [ebp+var_124], eax
                lea     eax, [ebp+var_118]
                mov     [ebp+var_14C], eax
                mov     [ebp+var_120], ebx

loc_8048433:                            ; CODE XREF: f+28C
                mov     eax, [ebp+var_124]
                xor     edx, edx
                push    edi
                push    [ebp+arg_10]
                sub     eax, esi
                div     [ebp+var_120]
                push    esi
                shr     eax, 1
                imul    eax, [ebp+var_120]
                lea     edx, [esi+eax]
                push    edx
                mov     [ebp+var_138], edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048482
                xor     eax, eax

loc_804846D:                            ; CODE XREF: f+CC
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804846D

loc_8048482:                            ; CODE XREF: f+B5
                push    ebx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    [ebp+var_124]
                call    [ebp+arg_C]
                mov     edx, [ebp+var_138]
                add     esp, 10h
                test    eax, eax
                jns     short loc_80484F6
                mov     ecx, [ebp+var_124]
                xor     eax, eax

loc_80484AB:                            ; CODE XREF: f+10D
```

```
                movzx   edi, byte ptr [edx+eax]
                mov     bl, [ecx+eax]
                mov     [edx+eax], bl
                mov     ebx, edi
                mov     [ecx+eax], bl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_80484AB
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    esi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_80484F6
                xor     eax, eax

loc_80484E1:                            ; CODE XREF: f+140
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_80484E1

loc_80484F6:                            ; CODE XREF: f+ED
                                        ; f+129
                mov     eax, [ebp+var_120]
                mov     edi, [ebp+var_124]
                add     edi, [ebp+var_134]
                lea     ebx, [esi+eax]
                jmp     short loc_8048513

loc_804850D:                            ; CODE XREF: f+17B
                add     ebx, [ebp+var_120]

loc_8048513:                            ; CODE XREF: f+157
                                        ; f+1F9
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    ebx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048537
                jmp     short loc_804850D

loc_8048531:                            ; CODE XREF: f+19D
                add     edi, [ebp+var_134]

loc_8048537:                            ; CODE XREF: f+179
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
```

```
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_8048531
                cmp     ebx, edi
                jnb     short loc_8048596
                xor     eax, eax
                mov     [ebp+var_128], edx

loc_804855F:                            ; CODE XREF: f+1BE
                mov     cl, [ebx+eax]
                mov     dl, [edi+eax]
                mov     [ebx+eax], dl
                mov     [edi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804855F
                mov     edx, [ebp+var_128]
                cmp     edx, ebx
                jnz     short loc_8048582
                mov     edx, edi
                jmp     short loc_8048588

loc_8048582:                            ; CODE XREF: f+1C8
                cmp     edx, edi
                jnz     short loc_8048588
                mov     edx, ebx

loc_8048588:                            ; CODE XREF: f+1CC
                                        ; f+1D0
                add     ebx, [ebp+var_120]
                add     edi, [ebp+var_134]
                jmp     short loc_80485AB

loc_8048596:                            ; CODE XREF: f+1A1
                jnz     short loc_80485AB
                mov     ecx, [ebp+var_134]
                mov     eax, [ebp+var_120]
                lea     edi, [ebx+ecx]
                add     ebx, eax
                jmp     short loc_80485B3

loc_80485AB:                            ; CODE XREF: f+1E0
                                        ; f:loc_8048596
                cmp     ebx, edi
                jbe     loc_8048513

loc_80485B3:                            ; CODE XREF: f+1F5
                mov     eax, edi
                sub     eax, esi
                cmp     eax, [ebp+var_130]
                ja      short loc_80485EB
                mov     eax, [ebp+var_124]
                mov     esi, ebx
                sub     eax, ebx
                cmp     eax, [ebp+var_130]
                ja      short loc_8048634
                sub     [ebp+var_11C], 8
```

```
                mov     edx, [ebp+var_11C]
                mov     ecx, [edx+4]
                mov     esi, [edx]
                mov     [ebp+var_124], ecx
                jmp     short loc_8048634

loc_80485EB:                            ; CODE XREF: f+209
                mov     edx, [ebp+var_124]
                sub     edx, ebx
                cmp     edx, [ebp+var_130]
                jbe     short loc_804862E
                cmp     eax, edx
                mov     edx, [ebp+var_11C]
                lea     eax, [edx+8]
                jle     short loc_8048617
                mov     [edx], esi
                mov     esi, ebx
                mov     [edx+4], edi
                mov     [ebp+var_11C], eax
                jmp     short loc_8048634

loc_8048617:                            ; CODE XREF: f+252
                mov     ecx, [ebp+var_11C]
                mov     [ebp+var_11C], eax
                mov     [ecx], ebx
                mov     ebx, [ebp+var_124]
                mov     [ecx+4], ebx

loc_804862E:                            ; CODE XREF: f+245
                mov     [ebp+var_124], edi

loc_8048634:                            ; CODE XREF: f+21B
                                        ; f+235 ...
                mov     eax, [ebp+var_14C]
                cmp     [ebp+var_11C], eax
                ja      loc_8048433
                mov     ebx, [ebp+var_120]

loc_804864C:                            ; CODE XREF: f+2A
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                add     ecx, [ebp+var_130]
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                cmp     ecx, eax
                mov     [ebp+var_120], eax
                jbe     short loc_804866B
                mov     ecx, eax

loc_804866B:                            ; CODE XREF: f+2B3
                mov     esi, [ebp+arg_0]
                mov     edi, [ebp+arg_0]
                add     esi, ebx
                mov     edx, esi
                jmp     short loc_80486A3

loc_8048677:                            ; CODE XREF: f+2F1
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
```

```
                mov     [ebp+var_13C], ecx
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                mov     ecx, [ebp+var_13C]
                test    eax, eax
                jns     short loc_80486A1
                mov     edi, edx

loc_80486A1:                            ; CODE XREF: f+2E9
                add     edx, ebx

loc_80486A3:                            ; CODE XREF: f+2C1
                cmp     edx, ecx
                jbe     short loc_8048677
                cmp     edi, [ebp+arg_0]
                jz      loc_8048762
                xor     eax, eax

loc_80486B2:                            ; CODE XREF: f+313
                mov     ecx, [ebp+arg_0]
                mov     dl, [edi+eax]
                mov     cl, [ecx+eax]
                mov     [edi+eax], cl
                mov     ecx, [ebp+arg_0]
                mov     [ecx+eax], dl
                inc     eax
                cmp     ebx, eax
                jnz     short loc_80486B2
                jmp     loc_8048762

loc_80486CE:                            ; CODE XREF: f+3C3
                lea     edx, [esi+edi]
                jmp     short loc_80486D5

loc_80486D3:                            ; CODE XREF: f+33B
                add     edx, edi

loc_80486D5:                            ; CODE XREF: f+31D
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    esi
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_80486D3
                add     edx, ebx
                cmp     edx, esi
                mov     [ebp+var_124], edx
                jz      short loc_804876F
                mov     edx, [ebp+var_134]
                lea     eax, [esi+ebx]
                add     edx, eax
                mov     [ebp+var_11C], edx
                jmp     short loc_804875B
```

```
loc_8048710:                            ; CODE XREF: f+3AA
                mov     cl, [eax]
                mov     edx, [ebp+var_11C]
                mov     [ebp+var_150], eax
                mov     byte ptr [ebp+var_130], cl
                mov     ecx, eax
                jmp     short loc_8048733

loc_8048728:                            ; CODE XREF: f+391
                mov     al, [edx+ebx]
                mov     [ecx], al
                mov     ecx, [ebp+var_128]

loc_8048733:                            ; CODE XREF: f+372
                mov     [ebp+var_128], edx
                add     edx, edi
                mov     eax, edx
                sub     eax, edi
                cmp     [ebp+var_124], eax
                jbe     short loc_8048728
                mov     dl, byte ptr [ebp+var_130]
                mov     eax, [ebp+var_150]
                mov     [ecx], dl
                dec     [ebp+var_11C]

loc_804875B:                            ; CODE XREF: f+35A
                dec     eax
                cmp     eax, esi
                jnb     short loc_8048710
                jmp     short loc_804876F

loc_8048762:                            ; CODE XREF: f+2F6
                                        ; f+315
                mov     edi, ebx
                neg     edi
                lea     ecx, [edi-1]
                mov     [ebp+var_134], ecx

loc_804876F:                            ; CODE XREF: f+347
                                        ; f+3AC
                add     esi, ebx
                cmp     esi, [ebp+var_120]
                jbe     loc_80486CE

loc_804877D:                            ; CODE XREF: f+13
                lea     esp, [ebp-0Ch]
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
f               endp
```

74.2 Задача 3.2

Имеется небольшой исполняемый файл, внутри которого находится довольно известная криптосистема. Попробуйте её идентифицировать.

- Windows x86: http://yurichev.com/RE-exercises/3/2/unknown_cryptosystem.exe
- Linux x86: http://yurichev.com/RE-exercises/3/2/unknown_encryption_linux86.tar

- Mac OS X (x64): http://yurichev.com/RE-exercises/3/2/unknown_encryption_MacOSX.tar

74.3 Задача 3.3

Имеется небольшой исполняемый файл, некая утилита. Она открывает другой файл, читает его, что-то вычисляет и показывает число с плавающей точкой. Попробуйте разобраться, что она делает.

- Windows x86: http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3.exe
- Linux x86: http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3_Linux86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3_MacOSX.tar

74.4 Задача 3.4

Утилита, шифрующая и дешифрующая файлы, по паролю. Есть зашифрованный текстовый файл, пароль неизвестен. Зашифрованный файл — это текст на английском языке. Утилита использует сравнительно мощный алгоритм шифрования, тем не менее, он был применен с очень грубой ошибкой. И из-за ошибки расшифровать файл вполне возможно с минимумом затрат.

Попробуйте найти ошибку и расшифровать файл.

- Windows x86: http://yurichev.com/RE-exercises/3/4/amateur_cryptor.exe
- Текстовый файл: http://yurichev.com/RE-exercises/3/4/text_encrypted

74.5 Задача 3.5

Это имитация защиты от копирования использующей ключевой файл. В ключевом файле имя пользователя и серийный номер.

Задачи две:

- (Простая) при помощи tracer либо иного отладчика, заставьте эту программу принимать измененный ключевой файл.
- (Средняя) ваша задача заключается в том, чтобы изменить в файле имя пользователя на другое, но при этом, модифицировать саму программу нельзя.
- Windows x86: http://yurichev.com/RE-exercises/3/5/super_mega_protection.exe
- Linux x86: http://yurichev.com/RE-exercises/3/5/super_mega_protection.tar
- Mac OS X (x64) http://yurichev.com/RE-exercises/3/5/super_mega_protection_MacOSX.tar
- Ключевой файл: http://yurichev.com/RE-exercises/3/5/sample.key

74.6 Задача 3.6

Это очень примитивный игрушечный веб-сервер, поддерживающий только статические файлы, без CGI[1], и т.д. В нем сознательно оставлено по крайней мере 4 уязвимости. Постарайтесь найти их все и использовать для взлома удаленной машины.

- Windows x86: http://yurichev.com/RE-exercises/3/6/webserv_win32.rar
- Linux x86: http://yurichev.com/RE-exercises/3/6/webserv_Linux_x86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/6/webserv_MacOSX_x64.tar

74.7 Задача 3.7

При помощи tracer или любого другого win32-отладчика, найдите скрытые мины во время игры, в стандартной игре Windows MineSweeper.

Подсказка: в [34] имеются некоторые описания внутренностей игры MineSweeper.

[1] Common Gateway Interface

74.8 Задача 3.8

Это достаточно известный алгоритм компрессии данных. Но из-за ошибки (или даже опечатки) он разжимает неверно. В этом можно убедиться на этих примерах.
Это исходный текст: http://yurichev.com/RE-exercises/3/8/test.txt
Это корректно сжатый текст: http://yurichev.com/RE-exercises/3/8/test.compressed
Это некорректно разжатый текст: http://yurichev.com/RE-exercises/3/8/test.uncompressed_incorrectly.

Попробуйте найти и исправить ошибку. При некотором упорстве, это можно сделать при помощи модификации исполняемого файла.

- Windows x86: http://yurichev.com/RE-exercises/3/8/compressor_win32.exe
- Linux x86: http://yurichev.com/RE-exercises/3/8/compressor_linux86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/8/compressor_MacOSX64.tar

Глава 75

crackme / keygenme

Несколько моих keygenme:
http://crackmes.de/users/yonkie/

Послесловие

Глава 76

Вопросы?

Совершенно по любым вопросам, вы можете не раздумывая писать автору: <dennis@yurichev.com>

Пожалуйста, присылайте мне информацию о замеченных ошибках (включая грамматические), и т.д.

Приложение

Приложение А

Общая терминология

слово обычно, слово это переменная помещающаяся в GPR CPU. В компьютерах старше персональных, память часто измерялась не в байтах, а в словах.

Приложение B

x86

B.1 Терминология

Общее для 16-bit (8086/80286), 32-bit (80386, и т.д.), 64-bit.

byte 8-бит. Для определения переменных и массива байт используется директива ассемблера DB. Байты передаются в 8-битных частях регистров: `AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L`.

word 16-бит. —"— директива ассемблера DW. Слова передаются в 16-битных частях регистров: `AX/BX/CX/DX/SI/DI/R*W`.

double word ("dword") 32-бит. —"— директива ассемблера DD. Двойные слова передаются в регистрах (x86) или в 32-битных частях регистров (x64). В 16-битном коде, двойные слова передаются в парах 16-битных регистров.

quad word ("qword") 64-бит. —"— директива ассемблера DQ. В 32-битном коде, учетверенные слова передаются в парах 32-битных регистров.

tbyte (10 байт) 80-бит или 10 байт (используется для регистров IEEE 754 FPU).

paragraph (16 байт) — термин был популярен в среде MS-DOS.

Типы данных с той же шириной (BYTE, WORD, DWORD) точно такие же и в Windows API.

B.2 Регистры общего пользования

Ко многим регистрам можно обращаться как к частям размером в байт или 16-битное слово. Это всё — наследие от более старых процессоров Intel (вплоть до 8-битного 8080), все еще поддерживаемое для обратной совместимости. Например, в RISC процессорах, такой возможности, как правило, нет.

Регистры, имеющие префикс R- появились только в x86-64, а префикс E- — в 80386. Таким образом, R-регистры 64-битные, а E-регистры — 32-битные.

В x86-64 добавили еще 8 GPR: R8-R15.

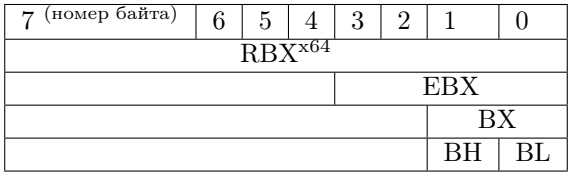
N.B.: В документации от Intel, для обращения к самому младшему байту к имени регистра нужно добавлять суффикс *L*: *R8L*, но IDA называет эти регистры добавляя суффикс *B*: *R8B*.

B.2.1 RAX/EAX/AX/AL

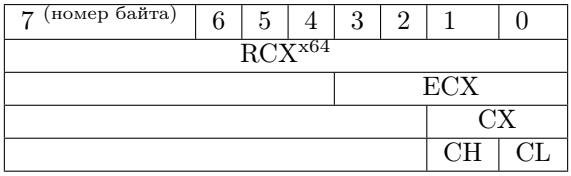
<table>
<tr><td>7 (номер байта)</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

AKA аккумулятор. Результат ф-ции обычно возвращается через этот регистр.

B.2.2 RBX/EBX/BX/BL

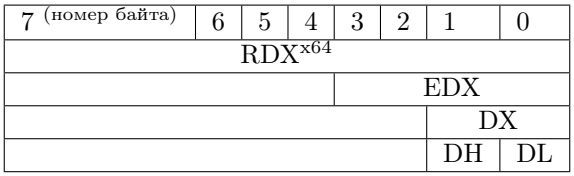


B.2.3 RCX/ECX/CX/CL

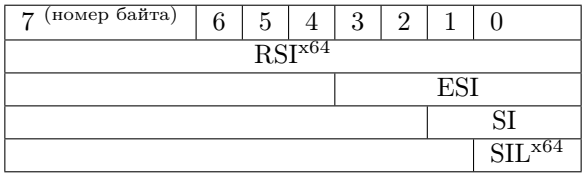


AKA счетчик: используется в этой роли в инструкциях с префиксом REP и в инструкциях сдвига (SHL/SHR/RxL/RxR)

B.2.4 RDX/EDX/DX/DL

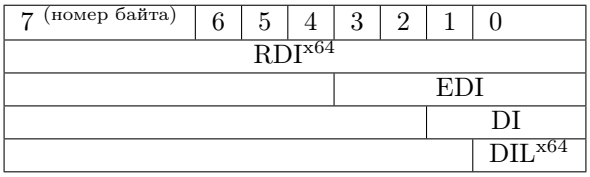


B.2.5 RSI/ESI/SI/SIL



AKA “source”. Используется как источник в инструкциях REP MOVSx, REP CMPSx.

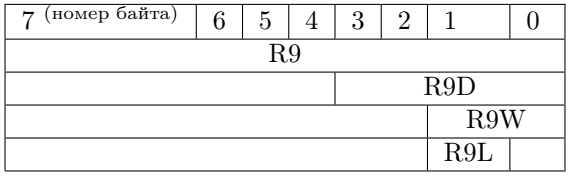
B.2.6 RDI/EDI/DI/DIL



AKA “destination”. Используется как указатель на место назначения в инструкции REP MOVSx, REP STOSx.

B.2.7 R8/R8D/R8W/R8L

B.2.8 R9/R9D/R9W/R9L



B.2.9 R10/R10D/R10W/R10L

7 (номер байта)	6	5	4	3	2	1	0
R10							
				R10D			
						R10W	
						R10L	

B.2.10 R11/R11D/R11W/R11L

7 (номер байта)	6	5	4	3	2	1	0
R11							
				R11D			
						R11W	
						R11L	

B.2.11 R12/R12D/R12W/R12L

7 (номер байта)	6	5	4	3	2	1	0
R12							
				R12D			
						R12W	
						R12L	

B.2.12 R13/R13D/R13W/R13L

7 (номер байта)	6	5	4	3	2	1	0
R13							
				R13D			
						R13W	
						R13L	

B.2.13 R14/R14D/R14W/R14L

7 (номер байта)	6	5	4	3	2	1	0
R14							
				R14D			
						R14W	
						R14L	

B.2.14 R15/R15D/R15W/R15L

7 (номер байта)	6	5	4	3	2	1	0
R15							
				R15D			
						R15W	
						R15L	

B.2.15 RSP/ESP/SP/SPL

7 (номер байта)	6	5	4	3	2	1	0
RSP[x64]							
				ESP			
						SP	
						SPL[x64]	

AKA указатель стека. Обычно всегда указывает на текущий стек, кроме тех случаев, когда он не инициализирован.

B.2.16 RBP/EBP/BP/BPL

7 (номер байта)	6	5	4	3	2	1	0
RBP^{x64}							
				EBP			
						BP	
							BPL^{x64}

AKA frame pointer. Обычно используется для доступа к локальным переменным ф-ции и аргументам, Больше о нем: (6.2.1).

B.2.17 RIP/EIP/IP

7 (номер байта)	6	5	4	3	2	1	0
RIP^{x64}							
				EIP			
						IP	

AKA “instruction pointer” [1]. Обычно всегда указывает на исполняющуюся инструкцию. Напрямую модифицировать регистр нельзя, хотя можно делать так (что равноценно):

```
mov eax...
jmp eax
```

Либо:

```
push val
ret
```

B.2.18 CS/DS/ES/SS/FS/GS

16-битные регистры, содержащие селектор кода (CS), данных (DS), стека (SS).

FS в win32 указывает на TLS, а в Linux на эту роль был выбран GS. Это сделано для более быстрого доступа к TLS и прочим структурам там вроде TIB.
В прошлом эти регистры использовались как сегментные регистры (67).

B.2.19 Регистр флагов

AKA EFLAGS.

[1] Иногда называется также “program counter”

Бит (маска)	Аббревиатура (значение)	Описание
0 (1)	CF (Carry)	Флаг переноса. Инструкции CLC/STC/CMC используются для установки/сброса/инвертирования этого флага
2 (4)	PF (Parity)	Флаг четности (15.3.1).
4 (0x10)	AF (Adjust)	
6 (0x40)	ZF (Zero)	Выставляется в 0 если результат последней операции был 0.
7 (0x80)	SF (Sign)	Флаг знака.
8 (0x100)	TF (Trap)	Применяется при отладке. Если включен, то после исполнения каждой инструкции будет сгенерировано исключение.
9 (0x200)	IF (Interrupt enable)	Разрешены ли прерывания. Инструкции CLI/STI используются для установки/сброса этого флага
10 (0x400)	DF (Direction)	Задается направление для инструкций REP MOVSx, REP CMPSx, REP LODSx, REP SCASx. Инструкции CLD/STD используются для установки/сброса этого флага
11 (0x800)	OF (Overflow)	Переполнение.
12, 13 (0x3000)	IOPL (I/O privilege level)^{80286}	
14 (0x4000)	NT (Nested task)^{80286}	
16 (0x10000)	RF (Resume)^{80386}	Применяется при отладке. Если включить, CPU проигнорирует хардварную точку останова в DRx.
17 (0x20000)	VM (Virtual 8086 mode)^{80386}	
18 (0x40000)	AC (Alignment check)^{80486}	
19 (0x80000)	VIF (Virtual interrupt)^{Pentium}	
20 (0x100000)	VIP (Virtual interrupt pending)^{Pentium}	
21 (0x200000)	ID (Identification)^{Pentium}	

Остальные флаги зарезервированы.

B.3 FPU-регистры

8 80-битных регистров работающих как стек: ST(0)-ST(7). N.B.: IDA называет ST(0) просто ST. Числа хранятся в формате IEEE 754.

Формат значения *long double*:

(S — знак, I — целочисленная часть)

B.3.1 Регистр управления

Регистр, при помощи которого можно задавать поведение FPU.

Бит	Аббревиатура (значение)	Описание
0	IM (Invalid operation Mask)	
1	DM (Denormalized operand Mask)	
2	ZM (Zero divide Mask)	
3	OM (Overflow Mask)	
4	UM (Underflow Mask)	
5	PM (Precision Mask)	
7	IEM (Interrupt Enable Mask)	Разрешение исключений, по умолчанию 1 (запрещено)
8, 9	PC (Precision Control)	Управление точностью 00 — 24 бита (REAL4) 10 — 53 бита (REAL8) 11 — 64 бита (REAL10)
10, 11	RC (Rounding Control)	Управление округлением 00 — (по умолчанию) округлять к ближайшему 01 — округлять к $-\infty$ 10 — округлять к $+\infty$ 11 — округлять к 0
12	IC (Infinity Control)	0 — (по умолчанию) считать $+\infty$ и $-\infty$ за беззнаковое 1 — учитывать и $+\infty$ и $-\infty$

Флагами PM, UM, OM, ZM, DM, IM задается, генерировать ли исключения в случае соответствующих ошибок.

B.3.2 Регистр статуса

Регистр только для чтения.

Бит	Аббревиатура (значение)	Описание
15	B (Busy)	Работает ли сейчас FPU (1) или закончил и результаты готовы (0)
14	C3	
13, 12, 11	TOP	указывает, какой сейчас регистр является нулевым
10	C2	
9	C1	
8	C0	
7	IR (Interrupt Request)	
6	SF (Stack Fault)	
5	P (Precision)	
4	U (Underflow)	
3	O (Overflow)	
2	Z (Zero)	
1	D (Denormalized)	
0	I (Invalid operation)	

Биты SF, P, U, O, Z, D, I сигнализируют об исключениях.

О C3, C2, C1, C0 читайте больше: (15.3.1).

N.B.: когда используется регистр ST(x), FPU прибавляет x к TOP по модулю 8 и получается номер внутреннего регистра.

B.3.3 Tag Word

Этот регистр отражает текущее содержимое регистров чисел.

Бит	Аббревиатура (значение)
15, 14	Tag(7)
13, 12	Tag(6)
11, 10	Tag(5)
9, 8	Tag(4)
7, 6	Tag(3)
5, 4	Tag(2)
3, 2	Tag(1)
1, 0	Tag(0)

Для каждого тэга:

- 00 — Регистр содержит ненулевое значение
- 01 — Регистр содержит 0
- 10 — Регистр содержит специальное число (NAN[2], ∞, или денормализованное число)
- 11 — Регистр пуст

B.4 SIMD-регистры

B.4.1 MMX-регистры

8 64-битных регистров: MM0..MM7.

B.4.2 SSE и AVX-регистры

SSE: 8 128-битных регистров: XMM0..XMM7. В x86-64 добавлено еще 8 регистров: XMM8..XMM15.

AVX это расширение всех регистры до 256 бит.

B.5 Отладочные регистры

Применяются для работы с т.н. hardware breakpoints.

- DR0 — адрес точки останова #1
- DR1 — адрес точки останова #2
- DR2 — адрес точки останова #3
- DR3 — адрес точки останова #4
- DR6 — здесь отображается причина останова
- DR7 — здесь можно задать типы точек останова

B.5.1 DR6

Бит (маска)	Описание
0 (1)	B0 — сработала точка останова #1
1 (2)	B1 — сработала точка останова #2
2 (4)	B2 — сработала точка останова #3
3 (8)	B3 — сработала точка останова #4
13 (0x2000)	BD — была попытка модифицировать один из регистров DRx. может быть выставлен если бит GD выставлен.
14 (0x4000)	BS — точка останова типа single step (флаг TF был выставлен в EFLAGS). Наивысший приоритет. Другие биты также могут быть выставлены.
15 (0x8000)	BT (task switch flag)

N.B. Точка останова single step это срабатывающая после каждой инструкции. Может быть включена выставлением флага TF в EFLAGS (B.2.19).

B.5.2 DR7

В этом регистре задаются типы точек останова.

[2]Not a Number

Бит (маска)	Описание
0 (1)	L0 — разрешить точку останова #1 для текущей задачи
1 (2)	G0 — разрешить точку останова #1 для всех задач
2 (4)	L1 — разрешить точку останова #2 для текущей задачи
3 (8)	G1 — разрешить точку останова #2 для всех задач
4 (0x10)	L2 — разрешить точку останова #3 для текущей задачи
5 (0x20)	G2 — разрешить точку останова #3 для всех задач
6 (0x40)	L3 — разрешить точку останова #4 для текущей задачи
7 (0x80)	G3 — разрешить точку останова #4 для всех задач
8 (0x100)	LE — не поддерживается начиная с P6
9 (0x200)	GE — не поддерживается начиная с P6
13 (0x2000)	GD — исключение будет вызвано если какая-либо инструкция MOV попытается модифицировать один из DRx-регистров
16,17 (0x30000)	точка останова #1: R/W — тип
18,19 (0xC0000)	точка останова #1: LEN — длина
20,21 (0x300000)	точка останова #2: R/W — тип
22,23 (0xC00000)	точка останова #2: LEN — длина
24,25 (0x3000000)	точка останова #3: R/W — тип
26,27 (0xC000000)	точка останова #3: LEN — длина
28,29 (0x30000000)	точка останова #4: R/W — тип
30,31 (0xC0000000)	точка останова #4: LEN — длина

Так задается тип точки останова (R/W):

- 00 — исполнение инструкции
- 01 — запись в память
- 10 — обращения к I/O-портам (недоступно из user-mode)
- 11 — обращение к памяти (чтение или запись)

N.B.: отдельного типа для чтения из памяти действительно нет.

Так задается длина точки останова (LEN):

- 00 — 1 байт
- 01 — 2 байта
- 10 — не определено для 32-битного режима, 8 байт для 64-битного
- 11 — 4 байта

B.6 Инструкции

Инструкции, отмеченные как (M) обычно не генерируются компилятором: если вы видите её, вероятно, это вручную написанный фрагмент кода, либо это т.н. compiler intrinsic (63).

Только наиболее используемые инструкции перечислены здесь. Обращайтесь к [14] или [1] для полной документации.

B.6.1 Префиксы

LOCK используется чтобы предоставить эксклюзивный доступ к памяти в многопроцессорной среде. Для упрощения, можно сказать, что когда исполняется инструкция с этим префиксом, остальные процессоры в системе останавливаются. Чаще все это используется для критических секций, семафоров, мьютексов. Обычно используется с ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. Читайте больше о критических секциях (48.4).

REP используется с инструкциями MOVSx и STOSx instructions: инструкция будет исполняться в цикле, счетчик расположен в регистре CX/ECX/RCX. Для более детального описания, читайте больше об инструкциях MOVSx (B.6.2) и STOSx (B.6.2).

Работа инструкций с префиксом REP зависит от флага DF, он задает направление.

REPE/REPNE (AKA REPZ/REPNZ) используется с инструкциями CMPSx и SCASx instructions: инструкция будет исполняться в цикле, счетчик расположен в регистре `CX`/`ECX`/`RCX`. Выполнение будет прервано если ZF будет 0 (REPE) либо если ZF будет 1 (REPNE).

Для более детального описания, читайте больше об инструкциях CMPSx (B.6.3) и SCASx (B.6.2).

Работа инструкций с префиксами REPE/REPNE зависит от флага DF, он задает направление.

B.6.2 Наиболее часто используемые инструкции

Их можно заучить в первую очередь.

ADC (*add with carry*) сложить два значения, инкремент если выставлен флаг CF. часто используется для складывания больших значений, например, складывания двух 64-битных значений в 32-битной среде используя две инструкции ADD и ADC, например:

```
; работа с 64-битными значениями: прибавить val1 к val2.
; .lo означает младшие 32 бита, .hi - старшие
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; использовать CF выставленный или очищенный в предыдущей инструкции
```

Еще один пример: 21.

ADD сложить два значения

AND логическое “И”

CALL вызвать другую ф-цию: `PUSH address_after_CALL_instruction; JMP label`

CMP сравнение значений и установка флагов, то же что и `SUB`, но только без записи результата

DEC декремент. Флаг CF не модифицируется.

IMUL умножение с учетом знаковых значений

INC инкремент. Флаг CF не модифицируется.

JCXZ, JECXZ, JRCXZ (M) переход если CX/ECX/RCX=0

JMP перейти на другой адрес. Опкод имеет т.н. jump offset.

Jcc (где cc — condition code)

Немало этих инструкций имеют синонимы (отмечены с AKA), это сделано для удобства. Синонимичные инструкции транслируются в один и тот же опкод. Опкод имеет т.н. jump offset.

JAE AKA JNC: переход если больше или равно (беззнаковый): CF=0

JA AKA JNBE: переход если больше (беззнаковый): CF=0 и ZF=0

JBE переход если меньше или равно (беззнаковый): CF=1 или ZF=1

JB AKA JC: переход если меньше (беззнаковый): CF=1

JC AKA JB: переход если CF=1

JE AKA JZ: переход если равно или ноль: ZF=1

JGE переход если больше или равно (знаковый): SF=OF

JG переход если больше (знаковый): ZF=0 и SF=OF

JLE переход если меньше или равно (знаковый): ZF=1 или SF≠OF

JL переход если меньше (знаковый): SF≠OF

JNAE AKA JC: переход если не больше или равно (беззнаковый) CF=1

JNA переход если не больше (беззнаковый) CF=1 и ZF=1

JNBE переход если не меньше или равно (беззнаковый): CF=0 и ZF=0

JNB AKA JNC: переход если не меньше (беззнаковый): CF=0

JNC AKA JAE: переход если CF=0, синонимично JNB.

JNE AKA JNZ: переход если не равно или не ноль: ZF=0

JNGE переход если не больше или равно (знаковый): SF≠OF

JNG переход если не больше (знаковый): ZF=1 или SF≠OF

JNLE переход если не меньше (знаковый): ZF=0 и SF=OF

JNL переход если не меньше (знаковый): SF=OF

JNO переход если не переполнение: OF=0

JNS переход если флаг SF сброшен

JNZ AKA JNE: переход если не равно или не ноль: ZF=0

JO переход если переполнение: OF=1

JPO переход если сброшен флаг PF

JP AKA JPE: переход если выставлен флаг PF

JS переход если выставлен флаг SF

JZ AKA JE: переход если равно или ноль: ZF=1

LAHF скопировать некоторые биты флагов в AH

LEAVE аналог команд `MOV ESP, EBP` и `POP EBP` — то есть возврат указателя стека и регистра `EBP` в первоначальное состояние.

LEA (*Load Effective Address*) сформировать адрес

Это инструкция которая задумывалась вовсе не для складывания и умножения чисел, а для формирования адреса например из указателя на массив и прибавления индекса к нему [3].

То есть, разница между `MOV` и `LEA` в том, что `MOV` формирует адрес в памяти и загружает значение из памяти, либо записывает его туда, а `LEA` только формирует адрес.

Тем не менее, её можно использовать для любых других вычислений.

`LEA` удобна тем, что производимые ею вычисления не модифицируют флаги CPU.

```
int f(int a, int b)
{
        return a*8+b;
};
```

Листинг B.1: MSVC 2010 /Ox

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_f      PROC
        mov     eax, DWORD PTR _b$[esp-4]
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [eax+ecx*8]
        ret     0
_f      ENDP
```

Intel C++ использует LEA даже больше:

```
int f1(int a)
{
        return a*13;
};
```

Листинг B.2: Intel C++ 2011

```
_f1     PROC NEAR
        mov       ecx, DWORD PTR [4+esp]      ; ecx = a
        lea       edx, DWORD PTR [ecx+ecx*8]  ; edx = a*9
        lea       eax, DWORD PTR [edx+ecx*4]  ; eax = a*9 + a*4 = a*13
        ret
```

[3] См. также: http://en.wikipedia.org/wiki/Addressing_mode

Эти две инструкции вместо одной IMUL будут работать быстрее.

MOVSB/MOVSW/MOVSD/MOVSQ скопировать байт/ 16-битное слово/ 32-битное слово/ 64-битное слово на который указывает SI/ESI/RSI куда указывает DI/EDI/RDI.

Вместе с префиксом REP, инструкция будет исполняться в цикле, счетчик будет находится в регистре CX/ECX/RCX: это работает как memcpy() в Си. Если размер блока известен компилятору на стадии компиляции, memcpy() часто компилируется в короткий фрагмент кода использующий REP MOVSx, иногда даже несколько инструкций.

Эквивалент memcpy(EDI, ESI, 15):

```
; скопировать 15 байт из ESI в EDI
CLD         ; установить направление на "вперед"
MOV ECX, 3
REP MOVSD   ; скопировать 12 байт
MOVSW       ; скопировать еще 2 байта
MOVSB       ; скопировать оставшийся байт
```

(Вероятно, так быстрее чем копировать 15 байт используя просто одну REP MOVSB).

MOVSX загрузить с расширением знака см. также: (13.1.1)

MOVZX загрузить и очистить все остальные биты см. также: (13.1.2)

MOV загрузить значение. эта инструкция была названа неудачно (данные не перемещаются), что является результатом путаницы: в других архитектурах эта же инструкция называется “LOAD” или что-то в этом роде.

Важно: если в 32-битном режиме при помощи MOV записывать младшую 16-байтную часть регистра, то старшие 16 бит останутся такими же. Но если в 64-битном режиме модифицировать 32-битную часть регистра, то старшие 32 бита обнуляются.

Вероятно, это сделано для упрощения портирования кода под x86-64.

MUL умножение с учетом беззнаковых значений

NEG смена знака: $op = -op$

NOP NOP. Её опкод 0x90, что на самом деле это холостая инструкция `XCHG EAX,EAX`. Это значит, что в x86 (как и во многих RISC) нет отдельной NOP-инструкции. Еще примеры подобных операций: (61).

NOP может быть сгенерировать компилятором для выравнивания меток по 16-байтной границе. Другое очень популярное использование NOP это вставка её вручную (патчинг) на месте какой-либо инструкции вроде условного перехода, чтобы запретить её исполнение.

NOT op1: $op1 = \neg op1$. логическое “НЕ”

OR логическое “ИЛИ”

POP взять значение из стека: `value=SS:[ESP]; ESP=ESP+4 (или 8)`

PUSH записать значение в стек: `ESP=ESP-4 (или 8); SS:[ESP]=value`

RET : возврат из процедуры: `POP tmp; JMP tmp`.

В реальности, RET это макрос ассемблера, в среде Windows и *NIX транслирующийся в RETN (“return near”) ибо, во времена MS-DOS, где память адресовалась немного иначе (67), в RETF (“return far”).

`RET` может иметь операнд. Тогда его работа будет такой: `POP tmp; ADD ESP op1; JMP tmp`. `RET` с операндом обычно завершает ф-ции с соглашением о вызовах *stdcall*, см. также: 44.2.

SAHF скопировать биты из AH в флаги, см. также: 15.3.3

SBB (*subtraction with borrow*) вычесть одно значение из другого, декремент результата если флаг CF выставлен. часто используется для вычитания больших значений, например, для вычитания двух 64-битных значений в 32-битной среде используя инструкции SUB и SBB, например:

```
; работа с 64-битными значениями: вычесть val2 из val1
; .lo означает младшие 32 бита, .hi - старшие
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; использовать CF выставленный или очищенный в предыдущей инструкции
```

Еще один пример: 21.

SCASB/SCASW/SCASD/SCASQ (M) сравнить байт/ 16-битное слово/ 32-битное слово/ 64-битное слово записанный в AX/EAX/RAX со значением, адрес которого находится в DI/EDI/RDI. Выставить флаги так же, как это делает `CMP`.

Эта инструкция часто используется с префиксом REPNE: продолжать сканировать буфер до тех пор, пока не встретится специальное значение, записанное в AX/EAX/RAX. Отсюда "NE" в REPNE: продолжать сканирование если сравниваемые значения не равны и остановиться если равны.

Она часто используется как стандартная ф-ция Си strlen(), для определения длины ASCIIZ-строки:

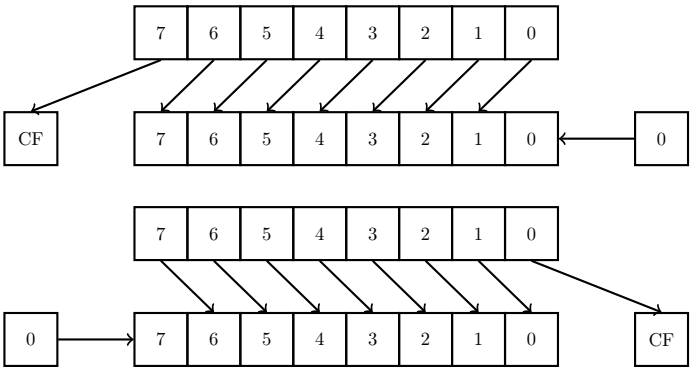
Пример:

```
lea     edi, string
mov     ecx, 0FFFFFFFFh ; сканировать 2^32-1 байт, т.е., почти "бесконечно"
xor     eax, eax        ; конец строки это 0
repne scasb
add     edi, 0FFFFFFFFh ; скорректировать

; теперь EDI указывает на последний символ в ASCIIZ-строке.

; узнать длину строки
; сейчас ECX = -1-strlen

not     ecx
dec     ecx

; теперь в ECX хранится длина строки
```

Если использовать другое значение AX/EAX/RAX, ф-ция будет работать как стандартная ф-ция Си memchr(), т.е., для поиска определенного байта.

SHL сдвинуть значение влево

SHR сдвинуть значение вправо:



Эти инструкции очень часто применяются для умножения и деления на 2^n. Еще одно очень частое применение это работа с битовыми полями: 17.

SHRD op1, op2, op3: сдвинуть значение в op2 вправо на op3 бит, подтягивая биты из op1.

Пример: 21.

STOSB/STOSW/STOSD/STOSQ записать байт/ 16-битное слово/ 32-битное слово/ 64-битное слово из AX/EAX/RAX в место, адрес которого находится в DI/EDI/RDI.

Вместе с префиксом REP, инструкция будет исполняться в цикле, счетчик будет находится в регистре CX/ECX/RCX: это работает как memset() в Си. Если размер блока известен компилятору на стадии компиляции, memset() часто компилируется в короткий фрагмент кода использующий REP STOSx, иногда даже несколько инструкций.

Эквивалент memset(EDI, 0xAA, 15):

```
; записать 15 байт 0xAA в EDI
CLD                 ; установить направление на "вперед"
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD           ; записать 12 байт
STOSW               ; записать еще 2 байта
STOSB               ; записать оставшийся байт
```

(Вероятно, так быстрее чем заполнять 15 байт используя просто одну REP STOSB).

SUB вычесть одно значение из другого. часто встречающийся вариант `SUB reg,reg` означает обнуление reg.

TEST то же что и AND, но без записи результатов, см. также: 17

XCHG обменять местами значения в операндах

XOR op1, op2: XOR[4] значений. $op1 = op1 \oplus op2$. часто встречающийся вариант `XOR reg,reg` означает обнуление reg.

B.6.3 Реже используемые инструкции

BSF *bit scan forward*, см. также: 22.2

BSR *bit scan reverse*

BSWAP *(byte swap)*, смена порядка байт в значении.

BTC bit test and complement

BTR bit test and reset

BTS bit test and set

BT bit test

CBW/CWD/CWDE/CDQ/CDQE Расширить значение учитывая его знак:

CBW : конвертировать байт в AL в слово в AX

CWD : конвертировать слово в AX в двойное слово в DX:AX

CWDE : конвертировать слово в AX в двойное слово в EAX

CDQ : конвертировать двойное слово в EAX в четверное слово в EDX:EAX

CDQE (x64): конвертировать двойное слово в EAX в четверное слово в RAX

Эти инструкции учитывают знак значения, расширяя его в старшую часть выходного значения. См. также: 21.4.

CLD сбросить флаг DF.

CLI (M) сбросить флаг IF

CMC (M) инвертировать флаг CF

CMOVcc условный MOV: загрузить значение если условие верно Коды точно такие же, как и в инструкциях Jcc (B.6.2).

CMPSB/CMPSW/CMPSD/CMPSQ (M) сравнить байт/ 16-битное слово/ 32-битное слово/ 64-битное слово из места, адрес которого находится в SI/ESI/RSI со значением, адрес которого находится в DI/EDI/RDI. Выставить флаги так же, как это делает `CMP`.

Вместе с префиксом REPE, инструкция будет исполняться в цикле, счетчик будет находится в регистре CX/ECX/RCX, процесс будет продолжаться пока флаг ZF=0 (т.е., до тех пор, пока все сравниваемые значения равны, отсюда “E” в REPE).

Это работает как memcmp() в Си.

Пример из ядра Windows NT (WRK v1.2):

[4] eXclusive OR (исключающее “ИЛИ”)

Листинг B.3: base\ntos\rtl\i386\movemem.asm

```
; ULONG
; RtlCompareMemory (
;    IN PVOID Source1,
;    IN PVOID Source2,
;    IN ULONG Length
;    )
;
; Routine Description:
;
;    This function compares two blocks of memory and returns the number
;    of bytes that compared equal.
;
; Arguments:
;
;    Source1 (esp+4) - Supplies a pointer to the first block of memory to
;       compare.
;
;    Source2 (esp+8) - Supplies a pointer to the second block of memory to
;       compare.
;
;    Length (esp+12) - Supplies the Length, in bytes, of the memory to be
;       compared.
;
; Return Value:
;
;    The number of bytes that compared equal is returned as the function
;    value. If all bytes compared equal, then the length of the original
;    block of memory is returned.
;
;--

RcmSource1      equ     [esp+12]
RcmSource2      equ     [esp+16]
RcmLength       equ     [esp+20]

CODE_ALIGNMENT
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

        push    esi                     ; save registers
        push    edi                     ;
        cld                             ; clear direction
        mov     esi,RcmSource1          ; (esi) -> first block to compare
        mov     edi,RcmSource2          ; (edi) -> second block to compare

;
;   Compare dwords, if any.
;

rcm10:  mov     ecx,RcmLength           ; (ecx) = length in bytes
        shr     ecx,2                   ; (ecx) = length in dwords
        jz      rcm20                   ; no dwords, try bytes
        repe    cmpsd                   ; compare dwords
        jnz     rcm40                   ; mismatch, go find byte

;
;   Compare residual bytes, if any.
;

rcm20:  mov     ecx,RcmLength           ; (ecx) = length in bytes
```

```
        and     ecx,3                   ; (ecx) = length mod 4
        jz      rcm30                   ; 0 odd bytes, go do dwords
        repe    cmpsb                   ; compare odd bytes
        jnz     rcm50                   ; mismatch, go report how far we got

;
;   All bytes in the block match.
;

rcm30:  mov     eax,RcmLength           ; set number of matching bytes
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

;
;   When we come to rcm40, esi (and edi) points to the dword after the
;   one which caused the mismatch.  Back up 1 dword and find the byte.
;   Since we know the dword didn't match, we can assume one byte won't.
;

rcm40:  sub     esi,4                   ; back up
        sub     edi,4                   ; back up
        mov     ecx,5                   ; ensure that ecx doesn't count out
        repe    cmpsb                   ; find mismatch byte

;
;   When we come to rcm50, esi points to the byte after the one that
;   did not match, which is TWO after the last byte that did match.
;

rcm50:  dec     esi                     ; back up
        sub     esi,RcmSource1          ; compute bytes that matched
        mov     eax,esi                 ;
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

stdENDP _RtlCompareMemory
```

N.B.: эта ф-ция использует сравнение 32-битных слов (CMPSD) если длина блоков кратна 4-м байтам, либо побайтовое сравнение (CMPSB) если не кратна.

CPUID получить информацию о доступных возможностях CPU. см. также: (18.6.1).

DIV деление с учетом беззнаковых значений

IDIV деление с учетом знаковых значений

INT (M): `INT x` аналогична `PUSHF; CALL dword ptr [x*4]` в 16-битной среде. Она активно использовалась в MS-DOS, работая как сисколл. Аргументы записывались в регистры AX/BX/CX/DX/SI/DI и затем происходил переход на таблицу векторов прерываний (расположенную в самом начале адресного пространства). Она была очень популярна потому что имела короткий опкод (2 байта) и программе использующая сервисы MS-DOS не нужно было заморачиваться узнавая адреса всех ф-ций этих сервисов. Обработчик прерываний возвращал управление назад при помощи инструкции IRET.

Самое используемое прерывание в MS-DOS было 0x21, там была основная часть его API. См. также: [4] самый крупный список всех известных прерываний и вообще там много информации о MS-DOS.

Во времена после MS-DOS, эта инструкция все еще использовалась как сискол, и в Linux и в Windows (46), но позже была заменена инструкцией SYSENTER или SYSCALL.

INT 3 (M): эта инструкция стоит немного в стороне от `INT`, она имеет собственный 1-байтный опкод (`0xCC`), и активно используется в отладке. Часто, отладчик просто записывает байт `0xCC` по адресу в памяти где устанавливается брякпойнт, и когда исключение поднимается, оригинальный байт будет восстановлен и

оригинальная инструкция по этому адресу исполнена заново.

В Windows NT, исключение `EXCEPTION_BREAKPOINT` поднимается, когда CPU исполняет эту инструкцию. Это отладочное событие может быть перехвачено и обработано отладчиком, если он загружен. Если он не загружен, Windows предложит запустить один из зарегистрированных в системе отладчиков. Если MSVS[5] установлена, его отладчик может быть загружен и подключен к процессу. В целях защиты от reverse engineering, множество анти-отладочных методов проверяют целостность загруженного кода.

В MSVC есть compiler intrinsic для этой инструкции: `__debugbreak()`[6].

В win32 также имеется ф-ция в kernel32.dll с названием `DebugBreak()`[7], которая также исполняет `INT 3`.

IN (M) получить данные из порта. Эту инструкцию обычно можно найти в драйверах OS либо в старом коде для MS-DOS, например (55.3).

IRET : использовалась в среде MS-DOS для возврата из обработчика прерываний, после того как он был вызван при помощи инструкции INT. Эквивалентна `POP tmp; POPF; JMP tmp`.

LOOP (M) декремент CX/ECX/RCX, переход если он всё еще не ноль.

OUT (M) послать данные в порт. Эту инструкцию обычно можно найти в драйверах OS либо в старом коде для MS-DOS, например (55.3).

POPA (M) восстанавливает значения регистров (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX из стека.

POPCNT population count. считает количество бит выставленных в 1 в значении. AKA "hamming weight". AKA "NSA instruction" из-за слухов:

> This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as "population count." It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I've heard this called the canonical NSA instruction, demanded by almost all computer contracts.

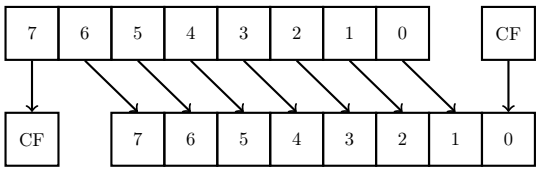
[31]

POPF восстановить флаги из стека (AKA регистр EFLAGS)

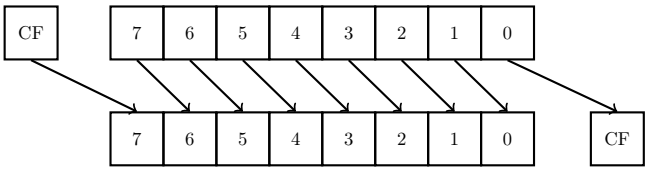
PUSHA (M) сохраняет значения регистров (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI в стеке.

PUSHF сохранить в стеке флаги (AKA регистр EFLAGS)

RCL (M) вращать биты налево через флаг CF:



RCR (M) вращать биты направо через флаг CF:



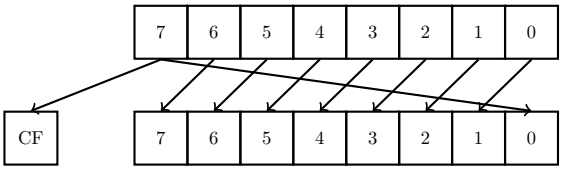
[5]Microsoft Visual Studio

[6]http://msdn.microsoft.com/en-us/library/f408b4et.aspx

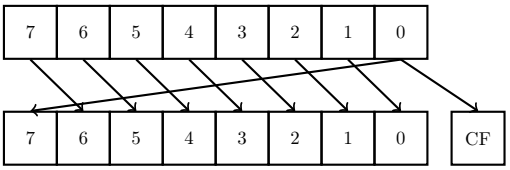
[7]http://msdn.microsoft.com/en-us/library/windows/desktop/ms679297(v=vs.85).aspx

ROL/ROR (M) циклический сдвиг

ROL: вращать налево:



ROR: вращать направо:

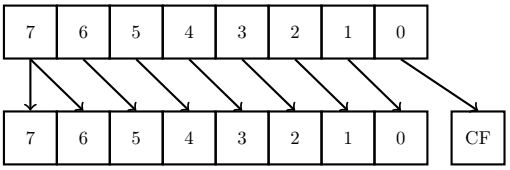


Не смотря на то что многие CPU имеют эти инструкции, в Си/Си++ нет соответствующих операций, так что компиляторы с этих ЯП обычно не генерируют код использующий эти инструкции.

Чтобы программисту были доступны эти инструкции, в MSVC есть псевдофункции (compiler intrinsics) *_rotl()* и *_rotr()*[8], которые транслируются компилятором напрямую в эти инструкции.

SAL Арифметический сдвиг влево, синонимично `SHL`

SAR Арифметический сдвиг вправо



Таким образом, бит знака всегда остается на месте MSB[9].

SETcc op: загрузить 1 в op (только байт) если условие верно или 0 если наоборот. Коды точно такие же, как и в инструкциях Jcc (B.6.2).

STC (M) установить флаг CF

STD (M) установить флаг DF

STI (M) установить флаг IF

SYSCALL (AMD) вызов сисколла (46)

SYSENTER (Intel) вызов сисколла (46)

UD2 (M) неопределенная инструкция, вызывает исключение. применяется для тестирования.

B.6.4 Инструкции FPU

-R в названии инструкции обычно означает что операнды поменяны местами, -P означает что один элемент выталкивается из стека после исполнения инструкции, -PP означает что выталкиваются два элемента.

-P инструкции часто бывают полезны, когда нам уже больше не нужно хранить значение в FPU-стеке.

FABS заменить значение в ST(0) на абсолютное значение ST(0)

FADD op: ST(0)=op+ST(0)

FADD ST(0), ST(i): ST(0)=ST(0)+ST(i)

[8] http://msdn.microsoft.com/en-us/library/5cc576c4.aspx

[9] Most significant bit/byte (самый старший бит/байт)

FADDP ST(1)=ST(0)+ST(1); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются суммой

FCHS : ST(0)=-ST(0)

FCOM сравнить ST(0) с ST(1)

FCOM op: сравнить ST(0) с op

FCOMP сравнить ST(0) с ST(1); вытолкнуть один элемент из стека

FCOMPP сравнить ST(0) с ST(1); вытолкнуть два элемента из стека

FDIVR op: ST(0)=op/ST(0)

FDIVR ST(i), ST(j): ST(i)=ST(j)/ST(i)

FDIVRP op: ST(0)=op/ST(0); вытолкнуть один элемент из стека

FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i); вытолкнуть один элемент из стека

FDIV op: ST(0)=ST(0)/op

FDIV ST(i), ST(j): ST(i)=ST(i)/ST(j)

FDIVP ST(1)=ST(0)/ST(1); вытолкнуть один элемент из стека, таким образом, делимое и делитель в стеке заменяются частным

FILD op: сконвертировать целочисленный op и затолкнуть его в стек.

FIST op: конвертировать ST(0) в целочисленное op

FISTP op: конвертировать ST(0) в целочисленное op; вытолкнуть один элемент из стека

FLD1 затолкнуть 1 в стек

FLDCW op: загрузить FPU control word (B.3) из 16-bit op.

FLDZ затолкнуть ноль в стек

FLD op: затолкнуть op в стек.

FMUL op: ST(0)=ST(0)*op

FMUL ST(i), ST(j): ST(i)=ST(i)*ST(j)

FMULP op: ST(0)=ST(0)*op; вытолкнуть один элемент из стека

FMULP ST(i), ST(j): ST(i)=ST(i)*ST(j); вытолкнуть один элемент из стека

FSINCOS : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)

FSQRT : $ST(0) = \sqrt{ST(0)}$

FSTCW op: записать FPU control word (B.3) в 16-bit op после проверки ожидающих исключений.

FNSTCW op: записать FPU control word (B.3) в 16-bit op.

FSTSW op: записать FPU status word (B.3.2) в 16-bit op после проверки ожидающих исключений.

FNSTSW op: записать FPU status word (B.3.2) в 16-bit op.

FST op: копировать ST(0) в op

FSTP op: копировать ST(0) в op; вытолкнуть один элемент из стека

FSUBR op: ST(0)=op-ST(0)

FSUBR ST(0), ST(i): ST(0)=ST(i)-ST(0)

FSUBRP ST(1)=ST(0)-ST(1); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются разностью

FSUB op: ST(0)=ST(0)-op

FSUB ST(0), ST(i): ST(0)=ST(0)-ST(i)

FSUBP ST(1)=ST(1)-ST(0); вытолкнуть один элемент из стека, таким образом, складываемые значения в стеке заменяются разностью

FUCOM ST(i): сравнить ST(0) и ST(i)

FUCOM : сравнить ST(0) и ST(1)

FUCOMP : сравнить ST(0) и ST(1); вытолкнуть один элемент из стека.

FUCOMPP : сравнить ST(0) и ST(1); вытолкнуть два элемента из стека.

Инструкция работает так же, как и FCOM, за тем исключением что исключение срабатывает только если один из операндов SNaN, но числа QNaN нормально обрабатываются.

FXCH ST(i) обменять местами значения в ST(0) и ST(i)

FXCH обменять местами значения в ST(0) и ST(1)

B.6.5 SIMD-инструкции

B.6.6 Инструкции с печатаемым ASCII-опкодом

(В 32-битном режиме).

Это может пригодиться для создания шеллкодов. См. также: 59.1.

ASCII-символ	шестнадцатеричный код	x86-инструкция
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3a	CMP
;	3b	CMP
<	3c	CMP
=	3d	CMP
?	3f	AAS
@	40	INC
A	41	INC
B	42	INC
C	43	INC
D	44	INC
E	45	INC
F	46	INC
G	47	INC
H	48	DEC
I	49	DEC
J	4a	DEC
K	4b	DEC
L	4c	DEC
M	4d	DEC
N	4e	DEC
O	4f	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH

T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5a	POP
[	5b	POP
\	5c	POP
]	5d	POP
^	5e	POP
_	5f	POP
`	60	PUSHA
a	61	POPA
f	66	(в 32-битном режиме) переключиться на 16-битный размер операнда
g	67	(в 32-битном режиме) переключиться на 16-битный размер адреса
h	68	PUSH
i	69	IMUL
j	6a	PUSH
k	6b	IMUL
p	70	JO
q	71	JNO
r	72	JB
s	73	JAE
t	74	JE
u	75	JNE
v	76	JBE
w	77	JA
x	78	JS
y	79	JNS
z	7a	JP

В итоге: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

Приложение C

ARM

C.1 Терминология

ARM изначально разрабатывался как 32-битный CPU, поэтому *слово* здесь, в отличие от x86, 32-битное.

byte 8-бит. Для определения переменных и массива байт используется директива ассемблера DCB.

halfword 16-бит. —"— директива ассемблера DCW.

word 32-бит. —"— директива ассемблера DCD.

C.2 Регистры общего пользования

- R0 — результат ф-ции обычно возвращается через R0
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R13 — AKA SP (указатель стека)
- R14 — AKA LR (link register)
- R15 — AKA PC (program counter)

`R0-R3` называются также "scratch registers": аргументы ф-ции обычно передаются через них, и эти значения не обязательно восстанавливать перед выходом из ф-ции.

C.3 Current Program Status Register (CPSR)

Бит	Описание
0..4	M — processor mode
5	T — Thumb state
6	F — FIQ disable
7	I — IRQ disable
8	A — imprecise data abort disable
9	E — data endianness
10..15, 25, 26	IT — if-then state
16..19	GE — greater-than-or-equal-to
20..23	DNM — do not modify
24	J — Java state
27	Q — sticky overflow
28	V — overflow
29	C — carry/borrow/extend
30	Z — zero bit
31	N — negative/less than

C.4 Регистры VPF (для чисел с плавающей точкой) и NEON

<table>
<tr><td>0..31 bits</td><td>32..64</td><td>65..96</td><td>97..127</td></tr>
<tr><td colspan="4">Q0 128 bits</td></tr>
<tr><td colspan="2">D0 64 bits</td><td colspan="2">D1</td></tr>
<tr><td>S0 32 bits</td><td>S1</td><td>S2</td><td>S3</td></tr>
</table>

S-регистры 32-битные, используются для хранения чисел с одинарной точностью.

D-регистры 64-битные, используются для хранения чисел с двойной точностью.

D- и S-регистры занимают одно и то же место в памяти CPU — можно обращаться к D-регистрам через S-регистры (хотя это и бессмысленно).

Точно также, NEON Q-регистры имеют размер 128 бит и занимают то же физическое место в памяти CPU что и остальные регистры, предназначенные для чисел с плавающей точкой.

В VFP присутствует 32 S-регистров: S0..S31.

В VPFv2 были добавлены 16 D-регистров, которые занимают то же место что и S0..S31.

В VFPv3 (NEON или “Advanced SIMD”) добавили еще 16 D-регистров, в итоге это D0..D31, но регистры D16..D31 не делят место с другими S-регистрами.

В NEON или “Advanced SIMD” были добавлены также 16 128-битных Q-регистров, делящих место с регистрами D0..D31.

Приложение D

Некоторые библиотечные функции GCC

имя	значение
`__divdi3`	знаковое деление
`__moddi3`	остаток от знакового деления
`__udivdi3`	беззнаковое деление
`__umoddi3`	остаток от беззнакового деления

Приложение E

Некоторые библиотечные функции MSVC

`ll` в имени функции означает “long long”, т.е., 64-битный тип данных.

имя	значение
`__alldiv`	знаковое деление
`__allmul`	умножение
`__allrem`	остаток от знакового деления
`__allshl`	сдвиг влево
`__allshr`	знаковый сдвиг вправо
`__aulldiv`	беззнаковое деление
`__aullrem`	остаток от беззнакового деления
`__aullshr`	беззнаковый сдвиг вправо

Процедуры умножения и сдвига влево, одни и те же и для знаковых чисел и для беззнаковых, поэтому здесь только одна ф-ция для каждой операции.

Исходные коды этих ф-ций можно найти в установленной MSVS, в `VC/crt/src/intel/*.asm`.

Приложение F

Cheatsheets

F.1 IDA

Краткий справочник хот-кеев:

клавиша	значение
Space	переключать между листингом и просмотром кода в виде графа
C	конвертировать в код
D	конвертировать в данные
A	конвертировать в строку
*	конвертировать в массив
U	сделать неопределенным
O	сделать смещение из операнда
H	сделать десятичное число
R	сделать символ
B	сделать двоичное число
Q	сделать шестнадцатеричное число
N	переменовать идентификатор
?	калькулятор
G	переход на адрес
:	добавить комментарий
Ctrl-X	показать ссылки на текущую ф-цию, метку, переменную (в т.ч., в стеке)
X	показать ссылки на ф-цию, метку, переменную, итд
Alt-I	искать константу
Ctrl-I	искать следующее вхождение константы
Alt-B	искать последовательность байт
Ctrl-B	искать следующее вхождение последовательности байт
Alt-T	искать текст (включая инструкции, итд)
Ctrl-T	искать следующее вхождение текста
Alt-P	редактировать текущую функцию
Enter	перейти к ф-ции, переменной, итд
Esc	вернуться назад
Num -	свернуть ф-цию или отмеченную область
Num +	снова показать ф-цию или область

Сворачивание ф-ции или области может быть удобно чтобы прятать те части ф-ции, чья функция вам стала уже ясна. это используется в моем скрипте[1] для сворачивания некоторых очень часто используемых фрагментов inline-кода.

F.2 OllyDbg

Краткий справочник хот-кеев:

[1] https://github.com/yurichev/IDA_scripts

хот-кей	значение
F7	трассировать внутрь
F8	сделать шаг не входя в ф-цию
F9	запуск
Ctrl-F2	перезапуск

F.3 MSVC

Некоторые полезные опции, которые я использовал в книге.

опция	значение
/O1	оптимизация по размеру кода
/Ob0	не заменять вызовы inline-ф-ций их кодом
/Ox	максимальная оптимизация
/GS-	отключить проверки переполнений буфера
/Fa(file)	генерировать листинг на ассемблере
/Zi	генерировать отладочную информацию
/Zp(n)	паковать структуры по границе в n байт
/MD	выходной исполняемый файл будет использовать `MSVCR*.DLL`

F.4 GCC

Некоторые полезные опции, которые я использовал в книге.

опция	значение
-Os	оптимизация по размеру кода
-O3	максимальная оптимизация
-regparm=	как много аргументов будет передаваться через регистры
-o file	задать имя выходного файла
-g	генерировать отладочную информацию в итоговом исполняемом файле
-S	генерировать листинг на ассемблере
-masm=intel	генерировать листинг в Intel-синтаксисе

F.5 GDB

Некоторые команды, которые я использовал в книге:

опция	значение
break filename.c:number	установить брякпойнт на номере строки в исходном файле
break function	установить брякпойнт на ф-ции
break *address	установить брякпойнт на адресе
b	—”—
p variable	вывести значение переменной
run	запустить
r	—”—
cont	продолжить исполнение
c	—”—
bt	вывести стек
set disassembly-flavor intel	установить Intel-синтаксис
disas	disassemble current function
disas function	дизассемблировать ф-цию
disas function,+50	disassemble portion
disas $eip,+0x10	—”—
info registers	вывести все регистры
info locals	вывести локальные переменные (если известны)
x/w ...	вывести память как 32-битные слова
x/w $rdi	вывести память как 32-битные слова по адресу, на который указывает `RDI`
x/10w ...	вывести 10 слов памяти
x/s ...	вывести строку из памяти
x/i ...	трактовать память как код
x/10c ...	вывести 10 символов
x/b ...	вывести байты
x/h ...	вывести 16-битные полуслова
x/g ...	вывести 64-битные слова
finish	исполнять до конца ф-ции
next	следующая инструкция (не заходить в ф-ции)
step	следующая инструкция (заодить в ф-ции)
frame n	переключить фрейм стека
info break	список брякпойнтов
del n	удалить брякпойнт

Приложение G

Ответы на задачи

G.1 Уровень 1

G.1.1 Задача 1.1

Это ф-ция возвращающая максимальное значение из двух.

G.1.2 Задача 1.4

Исходный код: http://yurichev.com/RE-exercise-solutions/1/4/password1.c

G.2 Уровень 2

G.2.1 Задача 2.1

Решение: `toupper()`.

Исходник на Си:

```
char toupper ( char c )
{
    if( c >= 'a' && c <= 'z' ) {
        c = c - 'a' + 'A';
    }
    return( c );
}
```

G.2.2 Задача 2.2

Ответ: `atoi()`

Исходник на Си:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int atoi ( const *p )  /* convert ASCII string to integer */
{
    int i;
    char s;

    while( isspace ( *p ) )
        ++p;
    s = *p;
    if( s == '+' || s == '-' )
        ++p;
    i = 0;
    while( isdigit(*p) ) {
```

```
        i = i * 10 + *p - '0';
        ++p;
    }
    if( s == '-' )
        i = - i;
    return( i );
}
```

G.2.3 Задача 2.3

Ответ: `srand()` / `rand()`.

Исходник на Си:

```
static unsigned int v;

void srand (unsigned int s)
{
        v = s;
}

int rand ()
{
        return( ((v = v * 214013L
            + 2531011L) >> 16) & 0x7fff );
}
```

G.2.4 Задача 2.4

Ответ: `strstr()`.

Исходник на Си:

```
char * strstr (
        const char * str1,
        const char * str2
        )
{
        char *cp = (char *) str1;
        char *s1, *s2;

        if ( !*str2 )
            return((char *)str1);

        while (*cp)
        {
                s1 = cp;
                s2 = (char *) str2;

                while ( *s1 && *s2 && !(*s1-*s2) )
                        s1++, s2++;

                if (!*s2)
                        return(cp);

                cp++;
        }

        return(NULL);

}
```

G.2.5 Задача 2.5

Подсказка #1: Не забывайте, что `__v` — глобальная переменная.

Подсказка #2: Эта функция вызывается startup-кодом перед вызовом `main()`.

Ответ: это проверка на наличие FDIV-ошибки в ранних процессорах Pentium[1].

Исходник на Си:

```
unsigned _v; // _v

enum e {
    PROB_P5_DIV = 0x0001
};

void f( void ) // __verify_pentium_fdiv_bug
{
    /*
        Verify we have got the Pentium FDIV problem.
        The volatiles are to scare the optimizer away.
    */
    volatile double     v1     = 4195835;
    volatile double     v2   = 3145727;

    if( (v1 - (v1/v2)*v2) > 1.0e-8 ) {
        _v |= PROB_P5_DIV;
    }
}
```

G.2.6 Задача 2.6

Подсказка: если погуглить применяемую здесь константу, это может помочь.

Ответ: шифрование алгоритмом TEA[2].

Исходник на Си (взято с http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm):

```
void f (unsigned int* v, unsigned int* k) {
    unsigned int v0=v[0], v1=v[1], sum=0, i;           /* set up */
    unsigned int delta=0x9e3779b9;                     /* a key schedule constant */
    unsigned int k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                                  /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

G.2.7 Задача 2.7

Подсказка: таблица содержит заранее вычисленные значения. Можно было бы обойтись и без нее, но тогда функция работала бы чуть медленнее.

Ответ: эта функция переставляет все биты во входном 32-битном слове наоборот. Это `lib/bitrev.c` из ядра Linux.

Исходник на Си:

```
const unsigned char byte_rev_table[256] = {
        0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
        0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
        0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
        0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
        0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
```

[1] http://en.wikipedia.org/wiki/Pentium_FDIV_bug

[2] Tiny Encryption Algorithm

```
        0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
        0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
        0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
        0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
        0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
        0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
        0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
        0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
        0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
        0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
        0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
        0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
        0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
        0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
        0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
        0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
        0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
        0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
        0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
        0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
        0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
        0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
        0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
        0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
        0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
        0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
        0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff,
};

unsigned char bitrev8(unsigned char byte)
{
        return byte_rev_table[byte];
}

unsigned short bitrev16(unsigned short x)
{
        return (bitrev8(x & 0xff) << 8) | bitrev8(x >> 8);
}

/**
 * bitrev32 - reverse the order of bits in a unsigned int value
 * @x: value to be bit-reversed
 */

unsigned int bitrev32(unsigned int x)
{
        return (bitrev16(x & 0xffff) << 16) | bitrev16(x >> 16);
}
```

G.2.8 Задача 2.8

Ответ: сложение двух матриц размером 100 на 200 элементов типа *double*.

Исходник на Си/Си++:

```
#define M    100
#define N    200

void s(double *a, double *b, double *c)
{
  for(int i=0;i<N;i++)
    for(int j=0;j<M;j++)
```

```
      *(c+i*M+j)=*(a+i*M+j) + *(b+i*M+j);
};
```

G.2.9 Задача 2.9

Ответ: умножение двух матриц размерами 100*200 и 100*300 элементов типа *double*, результат: матрица 100*300.

Исходник на Си/Си++:

```
#define M     100
#define N     200
#define P     300

void m(double *a, double *b, double *c)
{
  for(int i=0;i<M;i++)
    for(int j=0;j<P;j++)
    {
      *(c+i*M+j)=0;
      for (int k=0;k<N;k++) *(c+i*M+j)+=*(a+i*M+j) * *(b+i*M+j);
    }
};
```

G.2.10 Задача 2.11

Подсказка: Task Manager узнает количество процессоров/ядер используя вызов ф-ции `NtQuerySystemInformation(SystemBasicInformation, ..., ..., ...)`, этот вызов можно найти и подставлять туда другое значение.

И разумеется, Task Manager будет показывать некорректные результаты в графиках истории загрузки процессоров/ядер.

G.2.11 Задача 2.12

Это алгоритм примитивного шифрования ROT13, некогда популярный в UseNet и мейл-листах [3].

Исходный код: http://yurichev.com/RE-exercise-solutions/2/12/ROT13.c

G.2.12 Задача 2.13

Это криптоалгоритм linear feedback shift register (регистр сдвига с линейной обратной связью) [4].

Исходный код: http://yurichev.com/RE-exercise-solutions/2/13/LFSR.c

G.2.13 Задача 2.14

Это алгоритм нахождения наименьшего общего делителя.

Исходный код: http://yurichev.com/RE-exercise-solutions/2/14/GCD.c

G.2.14 Задача 2.15

Это вычисление числа Пи методом Монте-Карло.

Исходный код: http://yurichev.com/RE-exercise-solutions/2/15/monte.c

G.2.15 Задача 2.16

Это функция Аккермана [5].

```
int ack (int m, int n)
{
        if (m==0)
                return n+1;
```

[3] https://en.wikipedia.org/wiki/ROT13
[4] https://en.wikipedia.org/wiki/Linear_feedback_shift_register
[5] https://en.wikipedia.org/wiki/Ackermann_function

```
        if (n==0)
                return ack (m-1, 1);
        return ack(m-1, ack (m, n-1));
};
```

G.2.16 Задача 2.17

Это одномерный клеточный автомат по *правилу 110*:
https://en.wikipedia.org/wiki/Rule_110.

Исходный код: http://yurichev.com/RE-exercise-solutions/2/17/CA.c

G.2.17 Задача 2.18

Исходный код: http://yurichev.com/RE-exercise-solutions/2/18/

G.2.18 Задача 2.19

Исходный код: http://yurichev.com/RE-exercise-solutions/2/19/

G.3 Уровень 3

G.3.1 Задача 3.1

Подсказка #1: В этом коде есть одна особенность, по которой можно значительно сузить поиск функции в glibc..

Ответ: особенность — это вызов callback-функции (20), указатель на которую передается в четвертом аргументе. Это `quicksort()`.

Исходник на Си: http://yurichev.com/RE-exercise-solutions/3/1/2_1.c

G.3.2 Задача 3.2

Подсказка: проще всего конечно же искать по значениями в таблицах.

Исходник на Си с комментариями:
http://yurichev.com/RE-exercise-solutions/3/2/gost.c

G.3.3 Задача 3.3

Исходник на Си с комментариями:
http://yurichev.com/RE-exercise-solutions/3/3/entropy.c

G.3.4 Задача 3.4

Исходник на Си с комментариями, а также расшифрованный файл: http://yurichev.com/RE-exercise-solutions/3/4/

G.3.5 Задача 3.5

Подсказка: как видно, строка где указано имя пользователя занимает не весь ключевой файл.

Байты за терминирующим нулем вплоть до смещения `0x7F` игнорируются программой.

Исходник на Си с комментариями:
http://yurichev.com/RE-exercise-solutions/3/5/crc16_keyfile_check.c

G.3.6 Задача 3.6

Исходник на Си с комментариями:
http://yurichev.com/RE-exercise-solutions/3/6/

В качестве еще одного упражнения, теперь вы можете попробовать исправить уязвимости в этом веб-сервере.

G.3.7 Задача 3.8

Исходник на Си с комментариями:
http://yurichev.com/RE-exercise-solutions/3/8/

Список принятых сокращений

ОС Операционная Система . . . xv

ЧаВО Часто задаваемые вопросы . . . xv

ООП Объектно-Ориентированное Программирование . . . 249

ЯП Язык Программирования . . . 3

ГПСЧ Генератор псевдослучайных чисел . . . 176

RA Адрес возврата . . . 17

PE Portable Executable: 48.2 . . . 364

SP указатель стека. SP/ESP/RSP в x86/x64. SP в ARM. . . . 10

DLL Dynamic-link library . . . 364

PC Program Counter. IP/EIP/RIP в x86/64. PC в ARM. . . . 10

LR Link Register . . . 10

IDA Interactive Disassembler . . . 5

IAT Import Address Table . . . 365

INT Import Name Table . . . 365

RVA Relative Virtual Address . . . 365

VA Virtual Address . . . 365

OEP Original Entry Point . . . 361

MSVC Microsoft Visual C++

MSVS Microsoft Visual Studio . . . 596

ASLR Address Space Layout Randomization . . . 365

MFC Microsoft Foundation Classes . . . 367

TLS Thread Local Storage . . . xv

AKA Also Known As (Также известный как)

CRT C runtime library: sec:CRT . . . 5

CPU Central processing unit . . . xv

FPU Floating-point unit . . . 113

CISC Complex instruction set computing . . . 10

RISC Reduced instruction set computing . . . 10

GUI Graphical user interface . . . 361

RTTI Run-time type information . . . 265

BSS Block Started by Symbol . . . 66

SIMD Single instruction, multiple data . . . 195

BSOD Black Screen of Death . . . 353

ISA Instruction Set Architecture (Архитектура набора команд) . . . xv

CGI Common Gateway Interface . . . 574

HPC High-Performance Computing . . . 225

SOC System on Chip . . . 9

SEH Structured Exception Handling: 48.3 . . . 20

ELF Формат исполняемых файлов, использующийся в Linux и некоторых других *NIX . . . xv

TIB Thread Information Block . . . 136

TEA Tiny Encryption Algorithm . . . 610

PIC Position Independent Code: 47.1 . . . xv

NAN Not a Number . . . 587

NOP No OPeration . . . 51

BEQ (PowerPC, ARM) Branch if Equal . . . 54

BNE (PowerPC, ARM) Branch if Not Equal . . . 106

BLR (PowerPC) Branch to Link Register . . . 415

XOR eXclusive OR (исключающее “ИЛИ”) . . . 593

MCU Microcontroller unit . . . 436

RAM Random-access memory . . . 206

ROM Read-only memory . . . 519

EGA Enhanced Graphics Adapter . . . 519

VGA Video Graphics Array . . . 519

API Application programming interface . . . 327

ASCII American Standard Code for Information Interchange . . . 503

ASCIIZ ASCII Zero (ASCII-строка заканчивающаяся нулем) . . . 51

IA64 Intel Architecture 64 (Itanium): 66 . . . 321

EPIC Explicitly parallel instruction computing . . . 516

OOE Out-of-order execution . . . 516

MSDN Microsoft Developer Network . . . xvi

MSB Most significant bit/byte (самый старший бит/байт) . . . 597

STL (C++) Standard Template Library: 29.4 . . . 272

PODT (C++) Plain Old Data Type . . . 284

HDD Hard disk drive . . . 296

VM Virtual Memory (виртуальная память)

WRK Windows Research Kernel . . . 340

GPR General Purpose Registers (регистры общего пользования) . . . 3

SSDT System Service Dispatch Table . . . 353

RE Reverse Engineering . . . 525

SSE Streaming SIMD Extensions 214

BCD Binary-coded decimal 502

BOM Byte order mark 330

GDB GNU debugger 26

Литература

[1] AMD. AMD64 Architecture Programmer's Manual. 2013. Также доступно здесь: http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/.

[2] Apple. iOS ABI Function Call Guide. 2010. Также доступно здесь: http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf.

[3] blexim. Basic integer overflows. Phrack, 2002. Также доступно здесь: http://yurichev.com/mirrors/phrack/p60-0x0a.txt.

[4] Ralf Brown. The x86 interrupt list. Также доступно здесь: http://www.cs.cmu.edu/~ralf/files.html.

[5] Mike Burrell. Writing effcient itanium 2 assembly code. Также доступно здесь: http://yurichev.com/mirrors/RE/itanium.pdf.

[6] Marshall Cline. C++ faq. Также доступно здесь: http://www.parashift.com/c++-faq-lite/index.html.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009.

[8] Stephen Dolan. mov is turing-complete. 2013. Также доступно здесь: http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf.

[9] Nick Montfort et al. 10 PRINT CHR$(205.5+RND(1)); : GOTO 10. The MIT Press, 2012. Также доступно здесь: http://trope-tank.mit.edu/10_PRINT_121114.pdf.

[10] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs / An optimization guide for assembly programmers and compiler makers. 2013. http://agner.org/optimize/microarchitecture.pdf.

[11] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. 2013. http://agner.org/optimize/optimizing_cpp.pdf.

[12] Agner Fog. Calling conventions. 2014. http://www.agner.org/optimize/calling_conventions.pdf.

[13] IBM. PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. 2000. Также доступно здесь: http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf.

[14] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C. 2013. Также доступно здесь: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[15] ISO. ISO/IEC 9899:TC3 (C C99 standard). 2007. Также доступно здесь: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[16] ISO. ISO/IEC 14882:2011 (C++ 11 standard). 2013. Также доступно здесь: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[17] Brian W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[18] Donald E. Knuth. The Art of Computer Programming Volumes 1-3 Boxed Set. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998.

[19] Eugene Loh. The ideal hpc programming language. Queue, 8(6):30:30–30:38, June 2010.

[20] Advanced RISC Machines Ltd. The ARM Cookbook. 1994. Также доступно здесь: http://yurichev.com/ref/ARM%20Cookbook%20(1994).

[21] Michael Matz / Jan Hubicka / Andreas Jaeger / Mark Mitchell. System v application binary interface. amd64 architecture processor supplement, 2013. Также доступно здесь: http://x86-64.org/documentation/abi.pdf.

[22] Aleph One. Smashing the stack for fun and profit. Phrack, 1996. Также доступно здесь: http://yurichev.com/mirrors/phrack/p49-0x0e.txt.

[23] Matt Pietrek. A crash course on the depths of win32™ structured exception handling. MSDN magazine.

[24] Matt Pietrek. An in-depth look into the win32 portable executable file format. MSDN magazine, 2002.

[25] Eric S. Raymond. The Art of UNIX Programming. Pearson Education, 2003. Также доступно здесь: http://catb.org/esr/writings/taoup/html/.

[26] D. M. Ritchie and K. Thompson. The unix time sharing system. 1974. Также доступно здесь: http://dl.acm.org/citation.cfm?id=361061.

[27] Dennis M. Ritchie. The evolution of the unix time-sharing system. 1979.

[28] Dennis M. Ritchie. Where did ++ come from? (net.lang.c). http://yurichev.com/mirrors/C/c_dmr_postincrement.txt, 1986. [Online; accessed 2013].

[29] Dennis M. Ritchie. The development of the c language. SIGPLAN Not., 28(3):201–208, March 1993. Также доступно здесь: http://yurichev.com/mirrors/C/dmr-The%20Development%20of%20the%20C%20Language-1993.pdf.

[30] Mark E. Russinovich and David A. Solomon with Alex Ionescu. Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition. 2009.

[31] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 1994.

[32] Igor Skochinsky. Compiler internals: Exceptions and rtti, 2012. Также доступно здесь: http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf.

[33] SunSoft Steve Zucker and IBM Kari Karhi. SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement. 1995. Также доступно здесь: http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf.

[34] trew. Introduction to reverse engineering win32 applications. uninformed. Также доступно здесь: http://yurichev.com/mirrors/RE/uninformed_v1a7.pdf.

[35] Henry S. Warren. Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[36] Dennis Yurichev. Finding unknown algorithm using only input/output pairs and z3 smt solver. 2012. Также доступно здесь: http://yurichev.com/writings/z3_rockey.pdf.

[37] Dennis Yurichev. Заметки о языке программирования Си/Си++. 2013. Также доступно здесь: http://yurichev.com/writings/C-notes-ru.pdf.

Словарь терминов

декремент Уменьшение на 1. 91, 102, 106, 589, 591, 596

инкремент Увеличение на 1. 91, 94, 96, 102, 437, 457, 589

произведение Результат умножения. 196

указатель стека Регистр указывающий на место в стеке.. 5, 6, 10, 15, 18, 21, 32, 33, 41, 57, 307, 345–348, 583, 590, 601, 615

хвостовая рекурсия Это когда компилятор или интерпретатор превращает рекурсию (с которой возможно это проделать, т.е., *хвостовую*) в итерацию для эффективности: http://en.wikipedia.org/wiki/Tail_call. 14

частное Результат деления. 151, 408

anti-pattern Нечто широко известное как плохое решение. 17, 41

atomic operation "$\alpha\tau o\mu o\varsigma$" означает "неделимый" в греческом языке, так что атомарная операция это операция которая гарантированно не будет прервана другими тредами. 396, 513

basic block группа инструкций не имеющая инструкций переходов, а также не имеющая переходов в середину блока извне. В IDA он выглядит как просто список инструкций без строк-разрывов . 520, 521

callee Вызываемая ф-ция. 57, 60, 347, 350

caller Ф-ция вызывающая другую ф-цию. 59, 345

compiler intrinsic Специфичная для компилятора ф-ция не являющаяся обычной библиотечной ф-цией. Компилятор вместо её вызова генерирует определенный машинный код. Нередко, это псевдофункции для определенной инструкции CPU. Читайте больше: (63). 596

CP/M Control Program for Microcomputers: очень простая дисковая ОС использовавшаяся перед MS-DOS. 500

dongle Небольшое устройство подключаемое к LPT-порту для принтера (в прошлом) или к USB. Исполняло функции security token-а, имела память и, иногда, секретную (крипто-)хеширующую функцию.. 414

endianness Порядок байт: 33. 12, 43, 593

GiB Гибибайт: 2^{30} или 1024 мебибайт или 1073741824 байт. 9

heap (куча) обычно, большой кусок памяти предоставляемый ОС, так что прикладное ПО может делить его как захочет. malloc()/free() работают с кучей.. 15, 17, 158, 268, 270, 284, 285, 364, 365

jump offset Часть опкода JMP или Jcc инструкции, просто прибавляется к адресу следующей инструкции, и так вычисляется новый PC. Может быть отрицательным.. 75, 589

kernel mode Режим CPU с неограниченными возможностями в котором он исполняет ядро OS и драйвера. ср. user mode.. 622

keygenme Программа, имитирующая защиту вымышленной программы, для которой нужно сделать генератор ключей/лицензий. 576

leaf function Ф-ция не вызывающая больше никаких ф-ций. 17

link register (RISC) Регистр в котором обычно записан адрес возврата. Это позволяет вызывать leaf-функции без использования стека, т.е., быстрее.. 17, 415, 601

loop unwinding Это когда вместо организации цикла на n итераций, компилятор генерирует n копий тела цикла, для экономии на инструкциях, обеспечивающих сам цикл. 93

name mangling применяется как минимум в Си++, где компилятору нужно закодировать имя класса, метода и типы аргументов в одной строке, которая будет внутренним именем ф-ции. читайте также здесь: 29.1.1. 249, 324, 325

NaN не число: специальные случаи чисел с плавающей запятой, обычно сигнализирующие об ошибках . 122, 518

NEON AKA “Advanced SIMD” — SIMD от ARM. 602

NOP “no operation”, холостая инструкция. 343

POKE Инструкция языка BASIC записывающая байт по определенному адресу. 343

register allocator Ф-ция компилятора распределяющая локальные переменные по регистрам процессора. 101, 144, 206

reverse engineering процесс понимания как устроена некая вещь, иногда, с целью клонирования оной. xv, 596

security cookie Случайное значение, разное при каждом исполнении. Читайте больше об этом тут: 16.3. 385

stack frame Часть стека, в которой хранится информация связанная с текущей ф-цией: локальные переменные, аргументы ф-ции, RA, итд. 37, 56, 385

thunk function Крохотная функция делающая только одно: вызывающая другую функцию.. 13, 185, 415, 424

tracer Моя простейшая утилита для отладки. Читайте больше об этом тут: 50.1. 95, 96, 132, 328, 337, 341, 380, 391, 482, 489, 494, 497, 507, 574

user mode Режим CPU с ограниченными возможностями в котором он исполняет прикладное ПО. ср. kernel mode.. 432, 621

Windows NT Windows NT, 2000, XP, Vista, 7, 8. 203, 306, 331, 353, 365, 395, 503, 596

xoring нередко применяемое в английском языке, означает применение операции XOR. 385, 427, 430

Предметный указатель