Reverse Engineering for Beginners

Dennis Yurichev

Reverse Engineering for Beginners

Dennis Yurichev
<dennis(a)yurichev.com>

Text version (November 25, 2015).

The latest version (and Russian edition) of this text accessible at beginners.re. An A4-format version is also available.

You can also follow me on twitter to get information about updates of this text: @yurichev[1] or to subscribe to the mailing list[2].

The cover was made by Andy Nechaevsky: facebook.

[1]twitter.com/yurichev
[2]yurichev.com

Warning: this is a shortened LITE-version!

It is approximately 6 times shorter than full version (~150 pages) and intended to those who wants for very quick introduction to reverse engineering basics. There are nothing about MIPS, ARM, OllyDBG, GCC, GDB, IDA, there are no exercises, examples, etc.

If you still interesting in reverse engineering, full version of the book is always available on my website: beginners.re.

Contents

I Code patterns 1

1 A short introduction to the CPU 3

2 The simplest Function 5
2.1 x86 5

3 Hello, world! 7
3.1 x86 7
3.1.1 MSVC 7
3.2 x86-64 9
3.2.1 MSVC—x86-64 9
3.3 Conclusion 10

4 Function prologue and epilogue 11
4.1 Recursion 11

5 Stack 13
5.1 Why does the stack grow backwards? 13
5.2 What is the stack used for? 14
5.2.1 Save the function's return address 14
5.2.2 Passing function arguments 16
5.2.3 Local variable storage 17
5.2.4 x86: alloca() function 17
5.2.5 (Windows) SEH 19
5.2.6 Buffer overflow protection 19
5.2.7 Automatic deallocation of data in stack 19
5.3 A typical stack layout 19

6 `printf()` with several arguments 21
6.1 x86 21

6.1.1 x86: 3 arguments 21
6.1.2 x64: 8 arguments 23
6.2 Conclusion 24
6.3 By the way 25

7 scanf() 26
7.1 Simple example 26
7.1.1 About pointers 26
7.1.2 x86 27
7.1.3 x64 29
7.2 Global variables 30
7.2.1 MSVC: x86 31
7.2.2 MSVC: x64 33
7.3 scanf() result checking 34
7.3.1 MSVC: x86 34
7.3.2 MSVC: x86 + Hiew 37
7.3.3 MSVC: x64 38
7.4 Exercise 39

8 Accessing passed arguments 40
8.1 x86 40
8.1.1 MSVC 40
8.2 x64 42
8.2.1 MSVC 42

9 More about results returning 45
9.1 Attempt to use the result of a function returning *void* 45
9.2 What if we do not use the function result? 47

10 GOTO operator 48
10.1 Dead code 49

11 Conditional jumps 50
11.1 Simple example 50
11.1.1 x86 51
11.2 Calculating absolute value 56
11.2.1 Optimizing MSVC 56
11.3 Ternary conditional operator 57
11.3.1 x86 57
11.3.2 Let's rewrite it in an `if/else` way 59
11.4 Getting minimal and maximal values 60
11.4.1 32-bit 60
11.5 Conclusion 61
11.5.1 x86 61

11.5.2 Branchless . . . 62

12 switch()/case/default **63**
12.1 Small number of cases . . . 63
12.1.1 x86 . . . 63
12.1.2 Conclusion . . . 67
12.2 A lot of cases . . . 67
12.2.1 x86 . . . 67
12.2.2 Conclusion . . . 71
12.3 When there are several *case* statements in one block . . . 72
12.3.1 MSVC . . . 73
12.4 Fall-through . . . 75
12.4.1 MSVC x86 . . . 75

13 Loops **77**
13.1 Simple example . . . 77
13.1.1 x86 . . . 77
13.1.2 One more thing . . . 79
13.2 Memory blocks copying routine . . . 80
13.2.1 Straight-forward implementation . . . 80
13.3 Conclusion . . . 81

14 Simple C-strings processing **84**
14.1 strlen() . . . 84
14.1.1 x86 . . . 85

15 Replacing arithmetic instructions to other ones **87**
15.1 Multiplication . . . 87
15.1.1 Multiplication using addition . . . 87
15.1.2 Multiplication using shifting . . . 88
15.1.3 Multiplication using shifting, subtracting, and adding . . . 89
15.2 Division . . . 91
15.2.1 Division using shifts . . . 91

16 Arrays **93**
16.1 Simple example . . . 93
16.1.1 x86 . . . 94
16.2 Buffer overflow . . . 95
16.2.1 Reading outside array bounds . . . 95
16.2.2 Writing beyond array bounds . . . 97
16.3 One more word about arrays . . . 101
16.4 Array of pointers to strings . . . 102
16.4.1 x64 . . . 102
16.5 Multidimensional arrays . . . 104

16.5.1 Two-dimensional array example 106
16.5.2 Access two-dimensional array as one-dimensional 107
16.5.3 Three-dimensional array example 109
16.6 Conclusion 111

17 Manipulating specific bit(s) 112
17.1 Specific bit checking 112
17.1.1 x86 112
17.2 Setting and clearing specific bits 113
17.2.1 x86 114
17.3 Shifts 115
17.4 Counting bits set to 1 115
17.4.1 x86 118
17.4.2 x64 119
17.5 Conclusion 121
17.5.1 Check for specific bit (known at compile stage) 121
17.5.2 Check for specific bit (specified at runtime) 122
17.5.3 Set specific bit (known at compile stage) 123
17.5.4 Set specific bit (specified at runtime) 123
17.5.5 Clear specific bit (known at compile stage) 123
17.5.6 Clear specific bit (specified at runtime) 124

18 Linear congruential generator 125
18.1 x86 126
18.2 x64 127

19 Structures 129
19.1 MSVC: SYSTEMTIME example 129
19.1.1 Replacing the structure with array 131
19.2 Let's allocate space for a structure using malloc() 132
19.3 Fields packing in structure 135
19.3.1 x86 136
19.3.2 One more word 140
19.4 Nested structures 140
19.5 Bit fields in a structure 142
19.5.1 CPUID example 142

20 64-bit values in 32-bit environment 147
20.1 Returning of 64-bit value 147
20.1.1 x86 147
20.2 Arguments passing, addition, subtraction 148
20.2.1 x86 148
20.3 Multiplication, division 149
20.3.1 x86 150

20.4 Shifting right . . . 151
20.4.1 x86 . . . 151
20.5 Converting 32-bit value into 64-bit one . . . 152
20.5.1 x86 . . . 152

21 64 bits 153
21.1 x86-64 . . . 153

II Important fundamentals 155

22 Signed number representations 157

23 Memory 159

III Finding important/interesting stuff in the code 161

24 Communication with the outer world (win32) 163
24.1 Often used functions in the Windows API . . . 164
24.2 tracer: Intercepting all functions in specific module . . . 165

25 Strings 167
25.1 Text strings . . . 167
25.1.1 C/C++ . . . 167
25.1.2 Borland Delphi . . . 168
25.1.3 Unicode . . . 168
25.1.4 Base64 . . . 172
25.2 Error/debug messages . . . 173
25.3 Suspicious magic strings . . . 173

26 Calls to assert() 175

27 Constants 177
27.1 Magic numbers . . . 178
27.1.1 DHCP . . . 178
27.2 Searching for constants . . . 179

28 Finding the right instructions 180

29 Suspicious code patterns 183
29.1 XOR instructions . . . 183
29.2 Hand-written assembly code . . . 183

30 Using magic numbers while tracing 185

31 Other things 187
31.1 General idea . . . 187
31.2 Some binary file patterns . . . 187
31.3 Memory “snapshots” comparing . . . 188
31.3.1 Windows registry . . . 189
31.3.2 Blink-comparator . . . 189

IV Tools 190

32 Disassembler 191
32.1 IDA . . . 191

33 Debugger 192
33.1 tracer . . . 192

34 Decompilers 193

35 Other tools 194

V Books/blogs worth reading 195

36 Books 196
36.1 Windows . . . 196
36.2 C/C++ . . . 196
36.3 x86 / x86-64 . . . 196
36.4 ARM . . . 196
36.5 Cryptography . . . 196

37 Blogs 197
37.1 Windows . . . 197

38 Other 198

Afterword 200

39 Questions? 200

Acronyms used 203

Glossary 205

Index 207

Bibliography **210**

Preface

There are several popular meanings of the term "reverse engineering": 1) The reverse engineering of software: researching compiled programs; 2) The scanning of 3D structures and the subsequent digital manipulation required order to duplicate them; 3) recreating DBMS[3] structure. This book is about the first meaning.

Exercises and tasks

...are all moved to the separate website: `http://challenges.re`.

About the author

Dennis Yurichev is an experienced reverse engineer and programmer. He can be contacted by email: **dennis(a)yurichev.com**, or on Skype: **dennis.yurichev**.

Praise for *Reverse Engineering for Beginners*

- "It's very well done .. and for free .. amazing."[4] Daniel Bilar, Siege Technologies, LLC.
- "... excellent and free"[5] Pete Finnigan, Oracle RDBMS security guru.
- "... book is interesting, great job!" Michael Sikorski, author of *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.

[3]Database management systems
[4]twitter.com/daniel_bilar/status/436578617221742593
[5]twitter.com/petefinnigan/status/400551705797869568

- "... my compliments for the very nice tutorial!" Herbert Bos, full professor at the Vrije Universiteit Amsterdam, co-author of *Modern Operating Systems (4th Edition)*.
- "... 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.
- "... reasonable intro to some of the techniques."[6] Mike Stay, teacher at the Federal Law Enforcement Training Center, Georgia, US.
- "I love this book! I have several students reading it at the moment, plan to use it in graduate course."[7] Sergey Bratus, Research Assistant Professor at the Computer Science Department at Dartmouth College
- "Dennis @Yurichev has published an impressive (and free!) book on reverse engineering"[8] Tanel Poder, Oracle RDBMS performance tuning expert.
- "This book is some kind of Wikipedia to beginners..." Archer, Chinese Translator, IT Security Researcher.

Thanks

For patiently answering all my questions: Andrey "herm1t" Baranovich, Slava "Avid" Kazakov.

For sending me notes about mistakes and inaccuracies: Stanislav "Beaver" Bobrytskyy, Alexander Lysenko, Shell Rocket, Zhu Ruijin, Changmin Heo.

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), Aliaksandr Autayeu.

For translating the book into Simplified Chinese: Antiy Labs (antiy.cn) and Archer.

For translating the book into Korean: Byungho Min.

For proofreading: Alexander "Lstar" Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen.

Vasil Kolev did a great amount of work in proofreading and correcting many mistakes.

For illustrations and cover art: Andy Nechaevsky.

Thanks also to all the folks on github.com who have contributed notes and corrections.

[6]reddit
[7]twitter.com/sergeybratus/status/505590326560833536
[8]twitter.com/TanelPoder/status/524668104065159169

Many LaTeX packages were used: I would like to thank the authors as well.

Donors

Those who supported me during the time when I wrote significant part of the book:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10).

Thanks a lot to every donor!

mini-FAQ

Q: Why should one learn assembly language these days?
A: Unless you are an OS[9] developer, you probably don't need to code in assembly—modern compilers are much better at performing optimizations than humans [10]. Also, modern CPU[11]s are very complex devices and assembly knowledge doesn't really help one to understand their internals. That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, security/malware research. It is also a good way to gain a better understanding of your compiled code whilst debugging. This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?
A: In Adobe Acrobat Reader click Alt+LeftArrow.

[9] Operating System
[10] A very good text about this topic: [Fog13]
[11] Central processing unit

Q: I'm not sure if I should try to learn reverse engineering or not.
A: Perhaps, the average time to become familiar with the contents of the shortened LITE-version is 1-2 month(s).

Q: May I print this book? Use it for teaching?
A: Of course! That's why the book is licensed under the Creative Commons license. One might also want to build one's own version of book—read here to find out more.

Q: I want to translate your book to some other language.
A: Read my note to translators.

Q: How does one get a job in reverse engineering?
A: There are hiring threads that appear from time to time on reddit, devoted to RE[12] (2013 Q3, 2014). Try looking there. A somewhat related hiring thread can be found in the "netsec" subreddit: 2014 Q2.

Q: I have a question...
A: Send it to me by email (dennis(a)yurichev.com).

This is the A5-format version for e-book readers. Although the content is mostly the same, the illustrations are resized and probably not readable. You may try to change scale in your e-book reader. Otherwise, you can always view them in the A4-format version here: beginners.re.

About the Korean translation

In January 2015, the Acorn publishing company (www.acornpub.co.kr) in South Korea did a huge amount of work in translating and publishing my book (as it was in August 2014) into Korean.

It's now available at their website.

The translator is Byungho Min (twitter/tais9).

The cover art was done by my artistic friend, Andy Nechaevsky : facebook/andydinka.

They also hold the copyright to the Korean translation.

So, if you want to have a *real* book on your shelf in Korean and want to support my work, it is now available for purchase.

[12]reddit.com/r/ReverseEngineering/

Part I

Code patterns

Everything is comprehended in comparison

Author unknown

When the author of this book first started learning C and, later, C++, he used to write small pieces of code, compile them, and then look at the assembly language output. This made it very easy for him to understand what was going on in the code that he had written. [13]. He did it so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It's easy to imagine instantly a rough outline of C code's appearance and function. Perhaps this technique could be helpful for others.

Sometimes ancient compilers are used here, in order to get the shortest (or simplest) possible code snippet.

Optimization levels and debug information

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means disable optimization. Optimization can also be targeted towards code size or code speed.

A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact).

In addition to optimization levels and direction, a compiler can include in the resulting file some debug information, thus producing code for easy debugging.

One of the important features of the ´debug' code is that it might contain links between each line of the source code and the respective machine code addresses. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code.

Reverse engineers can encounter either version, simply because some developers turn on the compiler's optimization flags and others do not. Because of this, we'll try to work on examples of both debug and release versions of the code featured in this book, where possible.

[13]In fact, he still does it when he can't understand what a particular bit of code does.

Chapter 1

A short introduction to the CPU

The CPU is the device that executes the machine code a program consists of.

A short glossary:

Instruction : A primitive CPU command. The simplest examples include: moving data between registers, working with memory, primitive arithmetic operations . As a rule, each CPU has its own instruction set architecture (ISA[1]).

Machine code : Code that the CPU directly processes. Each instruction is usually encoded by several bytes.

Assembly language : Mnemonic code and some extensions like macros that are intended to make a programmer's life easier.

CPU register : Each CPU has a fixed set of general purpose registers (GPR[2]). ≈ 8 in x86, ≈ 16 in x86-64, ≈ 16 in ARM. The easiest way to understand a register is to think of it as an untyped temporary variable . Imagine if you were working with a high-level PL[3] and could only use eight 32-bit (or 64-bit) variables . Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a PL. The answer lies in the fact that humans and CPUs are not alike— it is much easier for humans to use a high-level PL like C/C++, Java, Python, etc., but it is easier for a CPU to use a much lower level of abstraction. Perhaps it would be possible to invent a CPU that can execute high-level PL code, but it would be many times more complex than the CPUs we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying

[1]Instruction Set Architecture
[2]General Purpose Registers
[3]Programming language

mistakes. The program that converts the high-level PL code into assembly is called a *compiler*.

Chapter 2

The simplest Function

The simplest possible function is arguably one that simply returns a constant value:

Here it is:

Listing 2.1: C/C++ Code

```
int f()
{
        return 123;
};
```

Lets compile it!

2.1 x86

Here's what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 2.2: Optimizing GCC/MSVC (assembly output)

```
f:
        mov     eax, 123
        ret
```

There are just two instructions: the first places the value 123 into the EAX register, which is used by convention for storing the return value and the second one is RET, which returns execution to the caller. The caller will take the result from the EAX register.

It is worth noting that MOV is a misleading name for the instruction in both x86 and ARM ISAs. The data is not in fact *moved*, but *copied*.

Chapter 3

Hello, world!

Let's use the famous example from the book "The C programming Language"[Ker88]:

```
#include <stdio.h>

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

3.1 x86

3.1.1 MSVC

Let's compile it in MSVC 2010:

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

(/Fa option instructs the compiler to generate assembly listing file)

Listing 3.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 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
```

The compiler generated the file, `1.obj`, which is to be linked into `1.exe`. In our case, the file contains two segments: CONST (for data constants) and _TEXT (for code).

The string `hello, world` in C/C++ has type `const char[]` [Str13, p176, 7.3.2], but it does not have its own name. The compiler needs to deal with the string somehow so it defines the internal name $SG3830 for it.

That is why the example may be rewritten as follows:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

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

Let's go back to the assembly listing. As we can see, the string is terminated by a zero byte, which is standard for C/C++ strings. More about C strings: 25.1.1 on page 167.

In the code segment, _TEXT, there is only one function so far: `main()`. The function `main()` starts with prologue code and ends with epilogue code (like almost any function)[1].

After the function prologue we see the call to the `printf()` function: CALL `_printf`. Before the call the string address (or a pointer to it) containing our greeting is placed on the stack with the help of the PUSH instruction.

When the `printf()` function returns the control to the `main()` function, the string address (or a pointer to it) is still on the stack. Since we do not need it

[1]You can read more about it in the section about function prologues and epilogues (4 on page 11).

anymore, the stack pointer (the ESP register) needs to be corrected.

ADD ESP, 4 means add 4 to the ESP register value. Why 4? Since this is a 32-bit program, we need exactly 4 bytes for address passing through the stack. If it was x64 code we would need 8 bytes. ADD ESP, 4 is effectively equivalent to POP register but without using any register[2].

For the same purpose, some compilers (like the Intel C++ Compiler) may emit POP ECX instead of ADD (e.g., such a pattern can be observed in the Oracle RDBMS code as it is compiled with the Intel C++ compiler). This instruction has almost the same effect but the ECX register contents will be overwritten. The Intel C++ compiler probably uses POP ECX since this instruction's opcode is shorter than ADD ESP, x (1 byte for POP against 3 for ADD).

Here is an example of using POP instead of ADD from Oracle RDBMS:

Listing 3.2: Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

After calling `printf()`, the original C/C++ code contains the statement `return 0` —return 0 as the result of the `main()` function. In the generated code this is implemented by the instruction XOR EAX, EAX. XOR is in fact just "eXclusive OR"[3] but the compilers often use it instead of MOV EAX, 0— again because it is a slightly shorter opcode (2 bytes for XOR against 5 for MOV).

Some compilers emit SUB EAX, EAX, which means *SUBtract the value in the* EAX *from the value in* EAX, which, in any case, results in zero.

The last instruction RET returns the control to the caller. Usually, this is C/C++ CRT[4] code, which, in turn, returns control to the OS.

3.2 x86-64

3.2.1 MSVC—x86-64

Let's also try 64-bit MSVC:

Listing 3.3: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H
```

[2]CPU flags, however, are modified

[3]wikipedia

[4]C runtime library

```
main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

In x86-64, all registers were extended to 64-bit and now their names have an R- prefix. In order to use the stack less often (in other words, to access external memory/cache less often), there exists a popular way to pass function arguments via registers (fastcall). I.e., a part of the function arguments is passed in registers, the rest—via the stack. In Win64, 4 function arguments are passed in the RCX, RDX, R8, R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but in the RCX register.

The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the R- prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the E- prefix.

This is how the RAX/EAX/AX/AL register looks like in x86-64:

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RAX^{x64}							
				EAX			
						AX	
						AH	AL

The `main()` function returns an *int*-typed value, which is, in C/C++, for better backward compatibility and portability, still 32-bit, so that is why the EAX register is cleared at the function end (i.e., the 32-bit part of the register) instead of RAX.

There are also 40 bytes allocated in the local stack. This is called the "shadow space", about which we are going to talk later: 8.2.1 on page 43.

3.3 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that the pointer to the string is now 64-bits in length. Indeed, modern CPUs are now 64-bit due to both the reduced cost of memory and the greater demand for it by modern applications. We can add much more memory to our computers than 32-bit pointers are able to address. As such, all pointers are now 64-bit.

Chapter 4

Function prologue and epilogue

A function prologue is a sequence of instructions at the start of a function. It often looks something like the following code fragment:

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

What these instruction do: save the value in the EBP register, set the value of the EBP register to the value of the ESP and then allocate space on the stack for local variables.

The value in the EBP stays the same over the period of the function execution and is to be used for local variables and arguments access. For the same purpose one can use ESP, but since it changes over time this approach is not too convenient.

The function epilogue frees the allocated space in the stack, returns the value in the EBP register back to its initial state and returns the control flow to the callee:

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

Function prologues and epilogues are usually detected in disassemblers for function delimitation.

4.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.

More about recursion in this book: **??** on page ??.

Chapter 5

Stack

The stack is one of the most fundamental data structures in computer science[1].

Technically, it is just a block of memory in process memory along with the ESP or RSP register in x86 or x64, or the SP[2] register in ARM, as a pointer within that block.

The most frequently used stack access instructions are PUSH and POP (in both x86 and ARM Thumb-mode). PUSH subtracts from ESP/RSP/SP 4 in 32-bit mode (or 8 in 64-bit mode) and then writes the contents of its sole operand to the memory address pointed by ESP/RSP/SP.

POP is the reverse operation: retrieve the data from the memory location that SP points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the stack pointer.

After stack allocation, the stack pointer points at the bottom of the stack. PUSH decreases the stack pointer and POP increases it. The bottom of the stack is actually at the beginning of the memory allocated for the stack block. It seems strange, but that's the way it is.

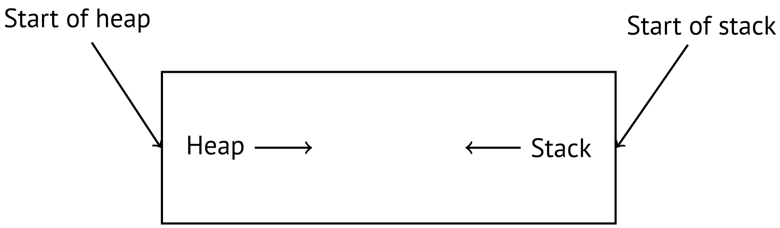
5.1 Why does the stack grow backwards?

Intuitively, we might think that the stack grows upwards, i.e. towards higher addresses, like any other data structure.

[1]wikipedia.org/wiki/Call_stack

[2]stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.

The reason that the stack grows backward is probably historical. When the computers were big and occupied a whole room, it was easy to divide memory into two parts, one for the heap and one for the stack. Of course, it was unknown how big the heap and the stack would be during program execution, so this solution was the simplest possible.



In [RT74] we can read:

> 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 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.

This reminds us how some students write two lecture notes using only one notebook: notes for the first lecture are written as usual, and notes for the second one are written from the end of notebook, by flipping it. Notes may meet each other somewhere in between, in case of lack of free space.

5.2 What is the stack used for?

5.2.1 Save the function's return address

x86

When calling another function with a CALL instruction, the address of the point exactly after the CALL instruction is saved to the stack and then an unconditional jump to the address in the CALL operand is executed.

The CALL instruction is equivalent to a PUSH address_after_call / JMP operand instruction pair.

RET fetches a value from the stack and jumps to it —that is equivalent to a POP tmp / JMP tmp instruction pair.

Overflowing the stack is straightforward. Just run eternal recursion:

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

MSVC 2008 reports the problem:

```
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
```

...but generates the right code anyway:

```
?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
```

... Also if we turn on the compiler optimization (/Ox option) the optimized code will not overflow the stack and will work *correctly*[3] instead:

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

[3]irony here

```
?f@@YAXXZ ENDP                                              ; f
```

5.2.2 Passing function arguments

The most popular way to pass parameters in x86 is called “cdecl”:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

Callee functions get their arguments via the stack pointer.

Therefore, this is how the argument values are located in the stack before the execution of the `f()` function's very first instruction:

ESP	return address
ESP+4	argument#1, marked in IDA[4] as `arg_0`
ESP+8	argument#2, marked in IDA as `arg_4`
ESP+0xC	argument#3, marked in IDA as `arg_8`
…	…

It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

For example, it is possible to allocate a space for arguments in the heap, fill it and pass it to a function via a pointer to this block in the EAX register. This will work[5]. However, it is a convenient custom in x86 and ARM to use the stack for this purpose.

By the way, the callee function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like `printf()`) determine their number using format string specifiers (which begin with the % symbol). If we write something like

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

`printf()` will print 1234, and then two random numbers, which were lying next to it in the stack.

[5]For example, in the “The Art of Computer Programming” book by Donald Knuth, in section 1.4.1 dedicated to subroutines [Knu98, section 1.4.1], we could read that one way to supply arguments to a subroutine is simply to list them after the JMP instruction passing control to subroutine. Knuth explains that this method was particularly convenient on IBM System/360.

That's why it is not very important how we declare the `main()` function: as `main()`, `main(int argc, char *argv[])` or `main(int argc, char *argv[], char *envp[])`.

In fact, the CRT-code is calling `main()` roughly as:

```
push envp
push argv
push argc
call main
...
```

If you declare `main()` as `main()` without arguments, they are, nevertheless, still present in the stack, but are not used. If you declare `main()` as `main(int argc, char *argv[])`, you will be able to use first two arguments, and the third will remain "invisible" for your function. Even more, it is possible to declare `main(int argc)`, and it will work.

5.2.3 Local variable storage

A function could allocate space in the stack for its local variables just by decreasing the stack pointer towards the stack bottom. Hence, it's very fast, no matter how many local variables are defined.

It is also not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.

5.2.4 x86: alloca() function

It is worth noting the `alloca()` function[6].

This function works like `malloc()`, but allocates memory directly on the stack.

The allocated memory chunk does not need to be freed via a `free()` function call, since the function epilogue (4 on page 11) returns ESP back to its initial state and the allocated memory is just *dropped*.

It is worth noting how `alloca()` is implemented.

In simple terms, this function just shifts ESP downwards toward the stack bottom by the number of bytes you need and sets ESP as a pointer to the *allocated* block. Let's try:

[6]In MSVC, the function implementation can be found in `alloca16.asm` and `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
#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);
};
```

`_snprintf()` function works just like `printf()`, but instead of dumping the result into stdout (e.g., to terminal or console), it writes it to the `buf` buffer. Function `puts()` copies the contents of `buf` to stdout. Of course, these two function calls might be replaced by one `printf()` call, but we have to illustrate small buffer usage.

MSVC

Let's compile (MSVC 2010):

Listing 5.1: MSVC 2010

```
...

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

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

    push    esi
    call    _puts
    add     esp, 28         ; 0000001cH

...
```

The sole `alloca()` argument is passed via EAX (instead of pushing it into the stack)[7]. After the `alloca()` call, ESP points to the block of 600 bytes and we can use it as memory for the `buf` array.

5.2.5 (Windows) SEH

SEH[10] records are also stored on the stack (if they are present)..

5.2.6 Buffer overflow protection

More about it here (16.2 on page 95).

5.2.7 Automatic deallocation of data in stack

Perhaps, the reason for storing local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (it is often ADD). Function arguments, as we could say, are also deallocated automatically at the end of function. In contrast, everything stored in the *heap* must be deallocated explicitly.

5.3 A typical stack layout

A typical stack layout in a 32-bit environment at the start of a function, before the first instruction execution looks like this:

[7] It is because alloca() is rather a compiler intrinsic than a normal function.
One of the reasons we need a separate function instead of just a couple of instructions in the code, is because the MSVC[8] alloca() implementation also has code which reads from the memory just allocated, in order to let the OS map physical memory to this VM[9] region.

[10] Structured Exception Handling

...	...
ESP-0xC	local variable #2, marked in IDA as `var_8`
ESP-8	local variable #1, marked in IDA as `var_4`
ESP-4	saved value of EBP
ESP	return address
ESP+4	argument#1, marked in IDA as `arg_0`
ESP+8	argument#2, marked in IDA as `arg_4`
ESP+0xC	argument#3, marked in IDA as `arg_8`
...	...

Chapter 6

printf() with several arguments

Now let's extend the *Hello, world!* (3 on page 7) example, replacing printf() in the main() function body with this:

```
#include <stdio.h>

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

6.1 x86

6.1.1 x86: 3 arguments

MSVC

When we compile it with MSVC 2010 Express we get:

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

...

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

Almost the same, but now we can see the `printf()` arguments are pushed onto the stack in reverse order. The first argument is pushed last.

By the way, variables of *int* type in 32-bit environment have 32-bit width, that is 4 bytes.

So, we have 4 arguments here. $4 * 4 = 16$ —they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type *int*.

When the stack pointer (ESP register) has changed back by the `ADD ESP, X` instruction after a function call, often, the number of function arguments could be deduced by simply dividing X by 4.

Of course, this is specific to the *cdecl* calling convention, and only for 32-bit environment.

In certain cases where several functions return right after one another, the compiler could merge multiple "`ADD ESP, X`" instructions into one, after the last call:

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

Here is a real-world example:

Listing 6.1: x86

```
.text:100113E7                push    3
.text:100113E9                call    sub_100018B0 ; takes one
    argument (3)
.text:100113EE                call    sub_100019D0 ; takes no
    arguments at all
.text:100113F3                call    sub_10006A90 ; takes no
    arguments at all
```

```
.text:100113F8                push    1
.text:100113FA                call    sub_100018B0 ; takes one ↙
    ↳ argument (1)
.text:100113FF                add     esp, 8       ; drops two ↙
    ↳ arguments from stack at once
```

6.1.2 x64: 8 arguments

To see how other arguments are passed via the stack, let's change our example again by increasing the number of arguments to 9 (`printf()` format string + 8 *int* variables):

```
#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;
};
```

MSVC

As it was mentioned earlier, the first 4 arguments has to be passed through the RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is exactly what we see here. However, the MOV instruction, instead of PUSH, is used for preparing the stack, so the values are stored to the stack in a straightforward manner.

Listing 6.2: 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
```

The observant reader may ask why are 8 bytes allocated for *int* values, when 4 is enough? Yes, one has to remember: 8 bytes are allocated for any data type shorter than 64 bits. This is established for the convenience's sake: it makes it easy to calculate the address of arbitrary argument. Besides, they are all located at aligned memory addresses. It is the same in the 32-bit environments: 4 bytes are reserved for all data types.

6.2 Conclusion

Here is a rough skeleton of the function call:

Listing 6.3: x86

```
...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; modify stack pointer (if needed)
```

Listing 6.4: x64 (MSVC)

```
MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc (if needed)
CALL function
; modify stack pointer (if needed)
```

6.3 By the way

By the way, this difference between the arguments passing in x86, x64, fastcall, ARM and MIPS is a good illustration of the fact that the CPU is oblivious to how the arguments are passed to functions. It is also possible to create a hypothetical compiler able to pass arguments via a special structure without using stack at all.

The CPU is not aware of calling conventions whatsoever.

We may also recall how newcoming assembly language programmers passing arguments into other functions: usually via registers, without any explicit order, or even via global variables. Of course, it works fine.

Chapter 7

scanf()

Now let's use scanf().

7.1 Simple example

```
#include <stdio.h>

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

        scanf ("%d", &x);

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

        return 0;
};
```

It's not clever to use `scanf()` for user interactions nowadays. But we can, however, illustrate passing a pointer to a variable of type *int*.

7.1.1 About pointers

Pointers are one of the fundamental concepts in computer science. Often, passing a large array, structure or object as an argument to another function is too expensive, while passing their address is much cheaper. In addition if the callee function needs to modify something in the large array or structure received as a parameter and return back the entire structure then the situation is close to absurd. So the simplest thing to do is to pass the address of the array or structure to the callee function, and let it change what needs to be changed.

A pointer in C/C++ is simply an address of some memory location.

In x86, the address is represented as a 32-bit number (i.e., it occupies 4 bytes), while in x86-64 it is a 64-bit number (occupying 8 bytes). By the way, that is the reason behind some people's indignation related to switching to x86-64—all pointers in the x64-architecture require twice as much space, including cache memory, which is "expensive" place.

It is possible to work with untyped pointers only, given some effort; e.g. the standard C function `memcpy()`, that copies a block from one memory location to another, takes 2 pointers of type `void*` as arguments, since it is impossible to predict the type of the data you would like to copy. Data types are not important, only the block size matters.

Pointers are also widely used when a function needs to return more than one value (we are going to get back to this later). *scanf()* is such a case. Besides the fact that the function needs to indicate how many values were successfully read, it also needs to return all these values.

In C/C++ the pointer type is only needed for compile-time type checking. Internally, in the compiled code there is no information about pointer types at all.

7.1.2 x86

MSVC

Here is what we get after compiling with 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
```

x is a local variable.

According to the C/C++ standard it must be visible only in this function and not from any other external scope. Traditionally, local variables are stored on the stack. There are probably other ways to allocate them, but in x86 that is the way it is.

The goal of the instruction following the function prologue, PUSH ECX, is not to save the ECX state (notice the absence of corresponding POP ECX at the function's end).

In fact it allocates 4 bytes on the stack for storing the x variable.

x is to be accessed with the assistance of the _x$ macro (it equals to -4) and the EBP register pointing to the current frame.

Over the span of the function's execution, EBP is pointing to the current stack frame making it possible to access local variables and function arguments via `EBP+offset`.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a *frozen state* of the value in ESP at the start of the function's execution.

Here is a typical stack frame layout in 32-bit environment:

...	...
EBP-8	local variable #2, marked in IDA as `var_8`
EBP-4	local variable #1, marked in IDA as `var_4`
EBP	saved value of EBP
EBP+4	return address
EBP+8	argument#1, marked in IDA as `arg_0`
EBP+0xC	argument#2, marked in IDA as `arg_4`
EBP+0x10	argument#3, marked in IDA as `arg_8`
...	...

The `scanf()` function in our example has two arguments.

The first one is a pointer to the string containing %d and the second is the address of the `x` variable.

First, the `x` variable's address is loaded into the EAX register by the `lea eax, DWORD PTR _x$[ebp]` instruction

We could say that in this case LEA simply stores the sum of the EBP register value and the `_x$` macro in the EAX register.

This is the same as `lea eax, [ebp-4]`.

So, 4 is being subtracted from the EBP register value and the result is loaded in the EAX register. Next the EAX register value is pushed into the stack and `scanf()` is being called.

`printf()` is being called after that with its first argument — a pointer to the string: `You entered %d...\n`.

The second argument is prepared with: `mov ecx, [ebp-4]`. The instruction stores the `x` variable value and not its address, in the ECX register.

Next the value in the ECX is stored on the stack and the last `printf()` is being called.

By the way

By the way, this simple example is a demonstration of the fact that compiler translates list of expressions in C/C++-block into sequential list of instructions. There are nothing between expressions in C/C++, and so in resulting machine code, there are nothing between, control flow slips from one expression to the next one.

7.1.3 x64

The picture here is similar with the difference that the registers, rather than the stack, are used for arguments passing.

MSVC

Listing 7.1: 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
```

7.2 Global variables

What if the x variable from the previous example was not local but a global one? Then it would have been accessible from any point, not only from the function body. Global variables are considered anti-pattern, but for the sake of the experiment, we could do this.

```
#include <stdio.h>

// now x is global variable
int x;

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

        scanf ("%d", &x);

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

        return 0;
};
```

7.2.1 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
```

In this case the x variable is defined in the _DATA segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the OS will allocate a block of zeroes there[1].

Now let's explicitly assign a value to the variable:

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

We got:

```
_DATA    SEGMENT
_x       DD       0aH

...
```

Here we see a value 0xA of DWORD type (DD stands for DWORD = 32 bit) for this variable.

If you open the compiled .exe in IDA, you can see the *x* variable placed at the beginning of the _DATA segment, and after it you can see text strings.

If you open the compiled .exe from the previous example in IDA, where the value of *x* was not set, you would see something like this:

```
.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
```

[1]That is how a VM behaves

_x is marked with ? with the rest of the variables that do not need to be initialized. This implies that after loading the .exe to the memory, a space for all these variables is to be allocated and filled with zeroes [ISO07, 6.7.8p10]. But in the .exe file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.

7.2.2 MSVC: x64

Listing 7.2: 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
```

The code is almost the same as in x86. Please note that the address of the x variable is passed to `scanf()` using a LEA instruction, while the variable's value is passed to the second `printf()` using a MOV instruction. DWORD PTR is a part of the assembly language (no relation to the machine code), indicating that the variable data size is 32-bit and the MOV instruction has to be encoded accordingly.

7.3 scanf() result checking

As was noted before, it is slightly old-fashioned to use `scanf()` today. But if we have to, we need to at least check if `scanf()` finishes correctly without an error.

```
#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;
};
```

By standard, the `scanf()`[2] function returns the number of fields it has successfully read.

In our case, if everything goes fine and the user enters a number `scanf()` returns 1, or in case of error (or EOF[3]) — 0.

Let's add some C code to check the `scanf()` return value and print error message in case of an error.

This works as expected:

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

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

7.3.1 MSVC: x86

Here is what we get in the assembly output (MSVC 2010):

[2] scanf, wscanf: MSDN

[3] End of file

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

The caller function (`main()`) needs the callee function (`scanf()`) result, so the callee returns it in the EAX register.

We check it with the help of the instruction `CMP EAX, 1` (*CoMPare*). In other words, we compare the value in the EAX register with 1.

A JNE conditional jump follows the CMP instruction. JNE stands for *Jump if Not Equal*.

So, if the value in the EAX register is not equal to 1, the CPU will pass the execution to the address mentioned in the JNE operand, in our case `$LN2@main`. Passing the control to this address results in the CPU executing `printf()` with the argument `What you entered? Huh?`. But if everything is fine, the conditional jump is not be be taken, and another `printf()` call is to be executed, with two arguments: `'You entered %d...'` and the value of x.

Since in this case the second `printf()` has not to be executed, there is a JMP preceding it (unconditional jump). It passes the control to the point after the second `printf()` and just before the `XOR EAX, EAX` instruction, which implements `return 0`.

So, it could be said that comparing a value with another is *usually* implemented by CMP/Jcc instruction pair, where *cc* is *condition code*. CMP compares two values and sets processor flags[4]. Jcc checks those flags and decides to either pass the control to the specified address or not.

[4]x86 flags, see also: wikipedia.

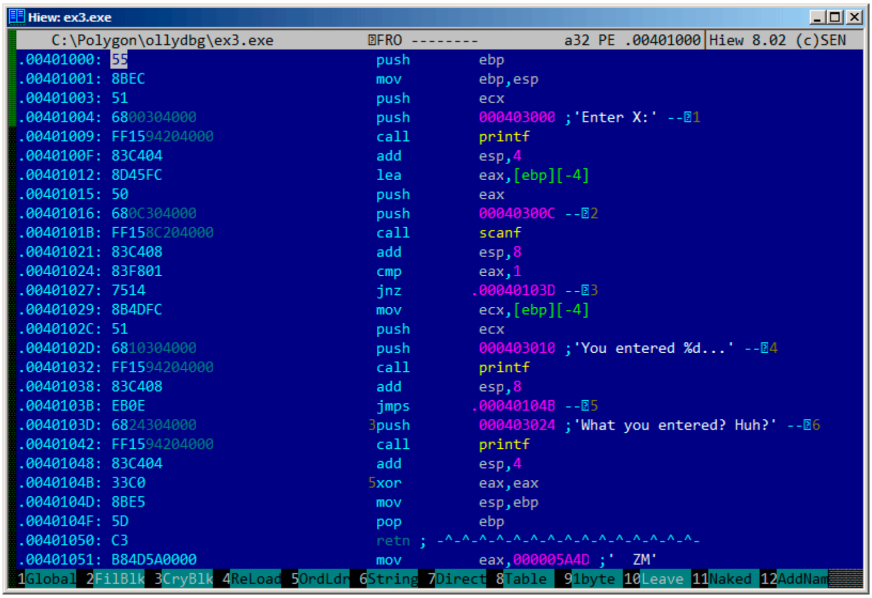
This could sound paradoxical, but the CMP instruction is in fact SUB (subtract). All arithmetic instructions set processor flags, not just CMP. If we compare 1 and 1, $1 - 1$ is 0 so the ZF flag would be set (meaning that the last result was 0). In no other circumstances ZF can be set, except when the operands are equal. JNE checks only the ZF flag and jumps only if it is not set. JNE is in fact a synonym for JNZ (*Jump if Not Zero*). Assembler translates both JNE and JNZ instructions into the same opcode. So, the CMP instruction can be replaced with a SUB instruction and almost everything will be fine, with the difference that SUB alters the value of the first operand. CMP is *SUB without saving the result, but affecting flags.*

7.3.2 MSVC: x86 + Hiew

This can also be used as a simple example of executable file patching. We may try to patch the executable so the program would always print the input, no matter what we enter.

Assuming that the executable is compiled against external `MSVCR*.DLL` (i.e., with `/MD` option)[5], we see the `main()` function at the beginning of the `.text` section. Let's open the executable in Hiew and find the beginning of the `.text` section (Enter, F8, F6, Enter, Enter).

We can see this:



```
Hiew: ex3.exe
    C:\Polygon\ollydbg\ex3.exe                □FRO --------         a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                                  push        ebp
.00401001: 8BEC                                mov         ebp,esp
.00401003: 51                                  push        ecx
.00401004: 6800304000                          push        000403000 ;'Enter X:' --□1
.00401009: FF1594204000                        call        printf
.0040100F: 83C404                              add         esp,4
.00401012: 8D45FC                              lea         eax,[ebp][-4]
.00401015: 50                                  push        eax
.00401016: 680C304000                          push        00040300C --□2
.0040101B: FF158C204000                        call        scanf
.00401021: 83C408                              add         esp,8
.00401024: 83F801                              cmp         eax,1
.00401027: 7514                                jnz        .00040103D --□3
.00401029: 8B4DFC                              mov         ecx,[ebp][-4]
.0040102C: 51                                  push        ecx
.0040102D: 6810304000                          push        000403010 ;'You entered %d...' --□4
.00401032: FF1594204000                        call        printf
.00401038: 83C408                              add         esp,8
.0040103B: EB0E                                jmps       .00040104B --□5
.0040103D: 6824304000                         3push        000403024 ;'What you entered? Huh?' --□6
.00401042: FF1594204000                        call        printf
.00401048: 83C404                              add         esp,4
.0040104B: 33C0                               5xor         eax,eax
.0040104D: 8BE5                                mov         esp,ebp
.0040104F: 5D                                  pop         ebp
.00401050: C3                                  retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                          mov         eax,000005A4D ;'  ZM'
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam
```

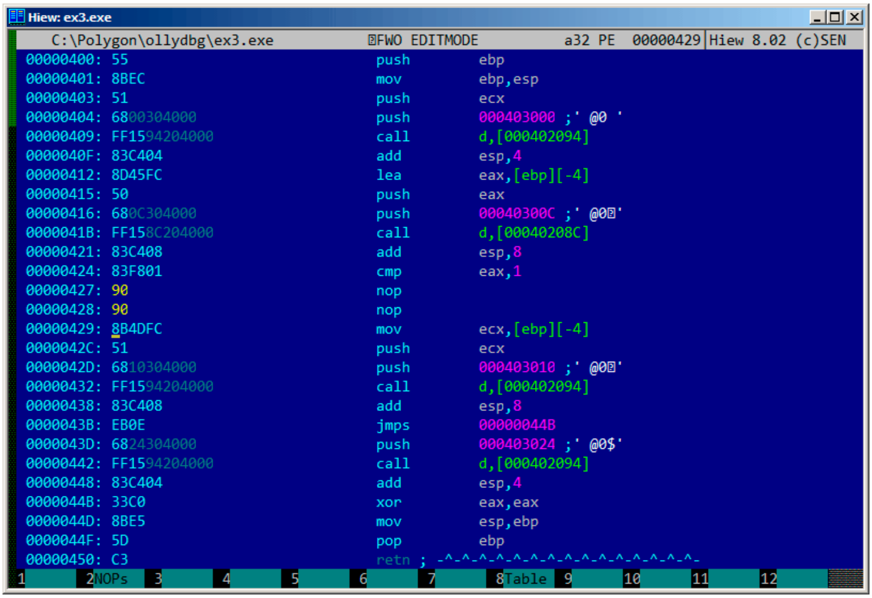
Figure 7.1: Hiew: `main()` function

Hiew finds ASCIIZ[6] strings and displays them, as it does with the imported functions' names.

[5] that's what also called "dynamic linking"

[6] ASCII Zero (null-terminated ASCII string)

Move the cursor to address `.00401027` (where the JNZ instruction, we have to bypass, is located), press F3, and then type "9090"(, meaning two NOP[7]s):



```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe     FWO EDITMODE        a32 PE  00000429 Hiew 8.02 (c)SEN
00000400: 55                       push      ebp
00000401: 8BEC                     mov       ebp,esp
00000403: 51                       push      ecx
00000404: 6800304000               push      000403000 ;' @0 '
00000409: FF1594204000             call      d,[000402094]
0000040F: 83C404                   add       esp,4
00000412: 8D45FC                   lea       eax,[ebp][-4]
00000415: 50                       push      eax
00000416: 680C304000               push      00040300C ;' @0 '
0000041B: FF158C204000             call      d,[00040208C]
00000421: 83C408                   add       esp,8
00000424: 83F801                   cmp       eax,1
00000427: 90                       nop
00000428: 90                       nop
00000429: 8B4DFC                   mov       ecx,[ebp][-4]
0000042C: 51                       push      ecx
0000042D: 6810304000               push      000403010 ;' @0 '
00000432: FF1594204000             call      d,[000402094]
00000438: 83C408                   add       esp,8
0000043B: EB0E                     jmps      00000044B
0000043D: 6824304000               push      000403024 ;' @0$'
00000442: FF1594204000             call      d,[000402094]
00000448: 83C404                   add       esp,4
0000044B: 33C0                     xor       eax,eax
0000044D: 8BE5                     mov       esp,ebp
0000044F: 5D                       pop       ebp
00000450: C3                       retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
1      2NOPs  3      4      5      6      7      8Table  9      10      11      12
```

Figure 7.2: Hiew: replacing JNZ with two NOPs

Then press F9 (update). Now the executable is saved to the disk. It will behave as we wanted.

Two NOPs are probably not the most æsthetic approach. Another way to patch this instruction is to write just 0 to the second opcode byte (jump offset), so that JNZ will always jump to the next instruction.

We could also do the opposite: replace first byte with EB while not touching the second byte (jump offset). We would get an unconditional jump that is always triggered. In this case the error message would be printed every time, no matter the input.

7.3.3 MSVC: x64

Since we work here with *int*-typed variables, which are still 32-bit in x86-64, we see how the 32-bit part of the registers (prefixed with `E-`) are used here as well.

[7]No OPeration

While working with pointers, however, 64-bit register parts are used, prefixed with R-.

Listing 7.3: 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
```

7.4 Exercise

- http://challenges.re/53

Chapter 8

Accessing passed arguments

Now we figured out that the caller function is passing arguments to the callee via the stack. But how does the callee access them?

Listing 8.1: simple example

```
#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;
};
```

8.1 x86

8.1.1 MSVC

Here is what we get after compilation (MSVC 2010 Express):

Listing 8.2: 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
```

What we see is that the `main()` function pushes 3 numbers onto the stack and calls `f(int,int,int)`. Argument access inside `f()` is organized with the help of macros like: `_a$ = 8`, in the same way as local variables, but with positive offsets (addressed with *plus*). So, we are addressing the *outer* side of the stack frame by adding the `_a$` macro to the value in the EBP register.

Then the value of a is stored into EAX. After `IMUL` instruction execution, the value in EAX is a product of the value in EAX and the content of _b. After that, ADD adds the value in _c to EAX. The value in EAX does not need to be moved: it is already where it must be. On returning to caller, it takes the EAX value and use it as an argument to `printf()`.

8.2 x64

The story is a bit different in x86-64. Function arguments (first 4 or first 6 of them) are passed in registers i.e. the callee reads them from registers instead of reading them from the stack.

8.2.1 MSVC

Optimizing MSVC:

Listing 8.3: Optimizing MSVC 2012 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
```

As we can see, the compact function `f()` takes all its arguments from the registers. The `LEA` instruction here is used for addition, apparently the compiler considered it faster than ADD. `LEA` is also used in the `main()` function to prepare the first and third `f()` arguments. The compiler must have decided that this would work faster than the usual way of loading values into a register using MOV instruction.

Let's take a look at the non-optimizing MSVC output:

Listing 8.4: 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
```

It looks somewhat puzzling because all 3 arguments from the registers are saved to the stack for some reason. This is called "shadow space" [1]: every Win64 may (but is not required to) save all 4 register values there. This is done for two reasons: 1) it is too lavish to allocate a whole register (or even 4 registers) for an input argument, so it will be accessed via stack; 2) the debugger is always aware where to find the function arguments at a break[2].

So, some large functions can save their input arguments in the "shadows space" if they need to use them during execution, but some small functions (like ours) may

[1] MSDN
[2] MSDN

not do this.

It is a caller responsibility to allocate “shadow space” in the stack.

Chapter 9

More about results returning

In x86, the result of function execution is usually returned[1] in the EAX register. If it is byte type or a character (*char*), then the lowest part of register EAX (AL) is used. If a function returns a *float* number, the FPU register `ST(0)` is used instead.

9.1 Attempt to use the result of a function returning *void*

So, what if the `main()` function return value was declared of type *void* and not *int*?

The so-called startup-code is calling `main()` roughly as follows:

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

In other words:

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

If you declare `main()` as *void*, nothing is to be returned explicitly (using the *return* statement), then something random, that was stored in the EAX register at the end

[1]See also: MSDN: Return Values (C++): MSDN

of `main()` becomes the sole argument of the exit() function. Most likely, there will be a random value, left from your function execution, so the exit code of program is pseudorandom.

We can illustrate this fact. Please note that here the `main()` function has a *void* return type:

```
#include <stdio.h>

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

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` with `puts()`, but that's OK, since `puts()` returns the number of characters printed out, just like `printf()`. Please notice that EAX is not zeroed before `main()`'s end. This implies that the value of EAX at the end of `main()` contains what `puts()` has left there.

Listing 9.1: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Let' s write a bash script that shows the exit status:

Listing 9.2: tst.sh

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

And run it:

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

14 is the number of characters printed.

9.2 What if we do not use the function result?

`printf()` returns the count of characters successfully output, but the result of this function is rarely used in practice. It is also possible to call a function whose essence is in returning a value, and not use it:

```
int f()
{
    // skip first 3 random values
    rand();
    rand();
    rand();
    // and use 4th
    return rand();
};
```

The result of the rand() function is left in EAX, in all four cases. But in the first 3 cases, the value in EAX is just thrown away.

Chapter 10

GOTO operator

The GOTO operator is generally considered as anti-pattern. [Dij68], Nevertheless, it can be used reasonably [Knu74], [Yur13, p. 1.3.2].

Here is a very simple example:

```
#include <stdio.h>

int main()
{
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

Here is what we have got in MSVC 2012:

Listing 10.1: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

The *goto* statement has been simply replaced by a JMP instruction, which has the same effect: unconditional jump to another place.

The second `printf()` could be executed only with human intervention, by using a debugger or by patching the code.

10.1 Dead code

The second `printf()` call is also called "dead code" in compiler terms. This means that the code will never be executed. So when you compile this example with optimizations, the compiler removes "dead code", leaving no trace of it:

Listing 10.2: Optimizing MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

However, the compiler forgot to remove the "skip me!" string.

Chapter 11

Conditional jumps

11.1 Simple example

```
#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;
};
```

11.1.1 x86

x86 + MSVC

Here is how the `f_signed()` function looks like:

Listing 11.1: Non-optimizing 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
```

The first instruction, `JLE`, stands for *Jump if Less or Equal*. In other words, if the second operand is larger or equal to the first one, the control flow will be passed to the specified in the instruction address or label. If this condition does not trigger because the second operand is smaller than the first one, the control flow would not be altered and the first printf() would be executed. The second check is JNE: *Jump if Not Equal*. The control flow will not change if the operands are equal.

The third check is JGE: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equal. So, if all three conditional jumps are triggered, none of the printf() calls would be executed whatsoever. This is impossible without special intervention.

Now let's take a look at the f_unsigned() function. The f_unsigned() function is the same as f_signed(), with the exception that the JBE and JAE instructions are used instead of JLE and JGE, as follows:

Listing 11.2: 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
```

As already mentioned, the branch instructions are different: JBE—*Jump if Below or Equal* and JAE—*Jump if Above or Equal*. These instructions (JA/JAE/JB/JBE) differ from JG/JGE/JL/JLE in the fact that they work with unsigned numbers.

See also the section about signed number representations (22 on page 157). That is why if we see JG/JL in use instead of JA/JB or vice-versa, we can be almost sure that the variables are signed or unsigned, respectively.

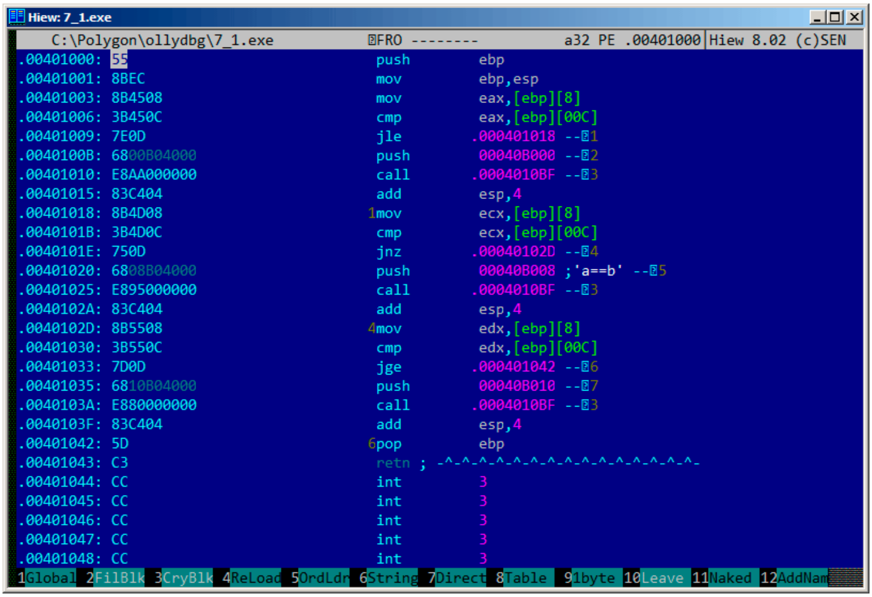
Here is also the main() function, where there is nothing much new to us:

Listing 11.3: 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
```

x86 + MSVC + Hiew

We can try to patch the executable file in a way that the `f_unsigned()` function would always print "a==b", no matter the input values. Here is how it looks in Hiew:


Hiew: 7_1.exe
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam

```
    C:\Polygon\ollydbg\7_1.exe              □FRO --------          a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55                            push        ebp
.00401001: 8BEC                          mov         ebp,esp
.00401003: 8B4508                        mov         eax,[ebp][8]
.00401006: 3B450C                        cmp         eax,[ebp][00C]
.00401009: 7E0D                          jle        .000401018 --□1
.0040100B: 6800B04000                    push        00040B000 --□2
.00401010: E8AA000000                    call       .0004010BF --□3
.00401015: 83C404                        add         esp,4
.00401018: 8B4D08                       1mov         ecx,[ebp][8]
.0040101B: 3B4D0C                        cmp         ecx,[ebp][00C]
.0040101E: 750D                          jnz        .00040102D --□4
.00401020: 6808B04000                    push        00040B008 ;'a==b' --□5
.00401025: E895000000                    call       .0004010BF --□3
.0040102A: 83C404                        add         esp,4
.0040102D: 8B5508                       4mov         edx,[ebp][8]
.00401030: 3B550C                        cmp         edx,[ebp][00C]
.00401033: 7D0D                          jge        .000401042 --□6
.00401035: 6810B04000                    push        00040B010 --□7
.0040103A: E880000000                    call       .0004010BF --□3
.0040103F: 83C404                        add         esp,4
.00401042: 5D                           6pop         ebp
.00401043: C3                            retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401044: CC                            int         3
.00401045: CC                            int         3
.00401046: CC                            int         3
.00401047: CC                            int         3
.00401048: CC                            int         3
```

Figure 11.1: Hiew: `f_unsigned()` function

Essentially, we need to accomplish three tasks:

- force the first jump to always trigger;
- force the second jump to never trigger;
- force the third jump to always trigger.

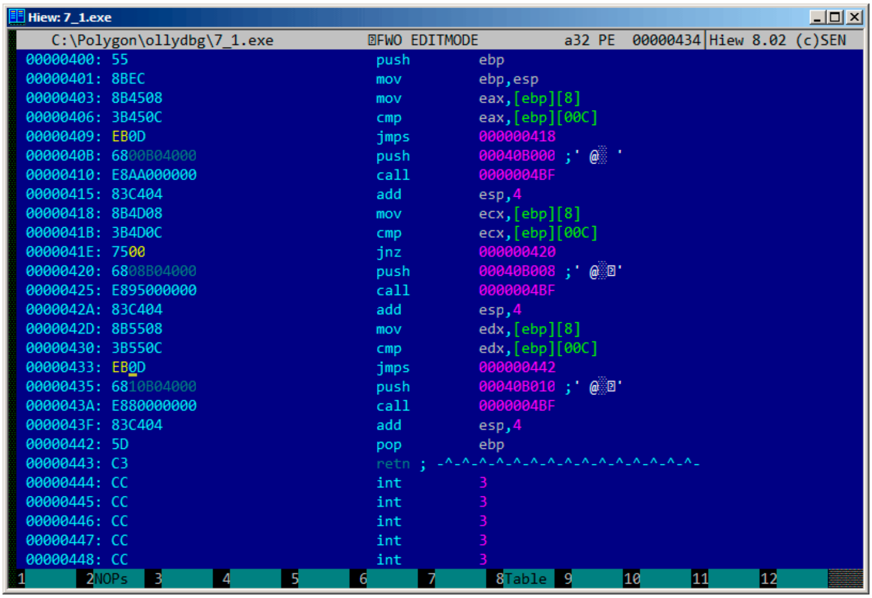
Thus we can direct the code flow to always pass through the second `printf()`, and output "a==b".

Three instructions (or bytes) has to be patched:

- The first jump becomes JMP, but the jump offset would remain the same.
- The second jump might be triggered sometimes, but in any case it will jump to the next instruction, because, we set the jump offset to 0. In these instructions the jump offset is added to the address for the next instruction. So if the offset is 0, the jump will transfer the control to the next instruction.

- The third jump we replace with JMP just as we do with the first one, so it will always trigger.

Here is the modified code:



```
Hiew: 7_1.exe
    C:\Polygon\ollydbg\7_1.exe          FWO EDITMODE       a32 PE   00000434 Hiew 8.02 (c)SEN
 00000400: 55                           push        ebp
 00000401: 8BEC                         mov         ebp,esp
 00000403: 8B4508                       mov         eax,[ebp][8]
 00000406: 3B450C                       cmp         eax,[ebp][00C]
 00000409: EB0D                         jmps        000000418
 0000040B: 6800B04000                   push        00040B000 ;' @ '
 00000410: E8AA000000                   call        0000004BF
 00000415: 83C404                       add         esp,4
 00000418: 8B4D08                       mov         ecx,[ebp][8]
 0000041B: 3B4D0C                       cmp         ecx,[ebp][00C]
 0000041E: 7500                         jnz         000000420
 00000420: 6808B04000                   push        00040B008 ;' @ '
 00000425: E895000000                   call        0000004BF
 0000042A: 83C404                       add         esp,4
 0000042D: 8B5508                       mov         edx,[ebp][8]
 00000430: 3B550C                       cmp         edx,[ebp][00C]
 00000433: EB0D                         jmps        000000442
 00000435: 6810B04000                   push        00040B010 ;' @ '
 0000043A: E880000000                   call        0000004BF
 0000043F: 83C404                       add         esp,4
 00000442: 5D                           pop         ebp
 00000443: C3                           retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
 00000444: CC                           int         3
 00000445: CC                           int         3
 00000446: CC                           int         3
 00000447: CC                           int         3
 00000448: CC                           int         3
1     2NOPs    3     4     5     6     7     8Table  9     10    11    12
```

Figure 11.2: Hiew: let’s modify the `f_unsigned()` function

If we miss to change any of these jumps, then several `printf()` calls may execute, while we want to execute only one.

11.2 Calculating absolute value

A simple function:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

11.2.1 Optimizing MSVC

This is how the code is usually generated:

Listing 11.4: Optimizing MSVC 2012 x64

```
i$ = 8
my_abs  PROC
; ECX = input
        test    ecx, ecx
; check for sign of input value
; skip NEG instruction if sign is positive
        jns     SHORT $LN2@my_abs
; negate value
        neg     ecx
$LN2@my_abs:
; prepare result in EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

11.3 Ternary conditional operator

The ternary conditional operator in C/C++ has the following syntax:

```
expression ? expression : expression
```

Here is an example:

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

11.3.1 x86

Old and non-optimizing compilers generate assembly code just as if an `if/else` statement was used:

Listing 11.5: Non-optimizing MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; compare input value with 10
        cmp     DWORD PTR _a$[ebp], 10
; jump to $LN3@f if not equal
        jne     SHORT $LN3@f
; store pointer to the string into temporary variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten↙
    ↳ '
; jump to exit
        jmp     SHORT $LN4@f
$LN3@f:
; store pointer to the string into temporary variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not↙
    ↳  ten'
$LN4@f:
; this is exit. copy pointer to the string from temporary
    variable to EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 11.6: Optimizing MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; compare input value with 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; jump to $LN4@f if equal
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

Newer compilers are more concise:

Listing 11.7: Optimizing MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

```

```
a$ = 8
f       PROC
; load pointers to the both strings
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; compare input value with 10
        cmp     ecx, 10
; if equal, copy value from RDX ("it is ten")
; if not, do nothing. pointer to the string "it is not ten" is
   still in RAX as for now.
        cmove   rax, rdx
        ret     0
f       ENDP
```

Optimizing GCC 4.8 for x86 also uses the `CMOVcc` instruction, while the non-optimizing GCC 4.8 uses conditional jumps.

11.3.2 Let's rewrite it in an `if/else` way

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

Interestingly, optimizing GCC 4.8 for x86 was also able to use `CMOVcc` in this case:

Listing 11.8: Optimizing GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
.LFB0:
; compare input value with 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; if comparison result is Not Equal, copy EDX value to EAX
; if not, do nothing
        cmovne  eax, edx
        ret
```

But the optimizing MSVC 2012 is not that good (yet).

11.4 Getting minimal and maximal values

11.4.1 32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Listing 11.9: Non-optimizing MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; compare A and B:
        cmp     eax, DWORD PTR _b$[ebp]
; jump, if A is greater or equal to B:
        jge     SHORT $LN2@my_min
; reload A to EAX if otherwise and jump to exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; this is redundant JMP
$LN2@my_min:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
```

```
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; compare A and B:
        cmp     eax, DWORD PTR _b$[ebp]
; jump if A is less or equal to B:
        jle     SHORT $LN2@my_max
; reload A to EAX if otherwise and jump to exit
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; this is redundant JMP
$LN2@my_max:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

These two functions differ only in the conditional jump instruction: JGE ("Jump if Greater or Equal") is used in the first one and JLE ("Jump if Less or Equal") in the second.

There is one unneeded JMP instruction in each function, which MSVC probably left by mistake.

11.5 Conclusion

11.5.1 x86

Here's the rough skeleton of a conditional jump:

Listing 11.10: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

11.5.2 Branchless

If the body of a condition statement is very short, the conditional move instruction can be used: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

Chapter 12

switch()/case/default

12.1 Small number of cases

```
#include <stdio.h>

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;
    };
};

int main()
{
    f (2); // test
};
```

12.1.1 x86

Non-optimizing MSVC

Result (MSVC 2010):

Listing 12.1: 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
```

Our function with a few cases in switch() is in fact analogous to this construction:

```
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");
};
```

If we work with switch() with a few cases it is impossible to be sure if it was a real switch() in the source code, or just a pack of if() statements. This implies that switch() is like syntactic sugar for a large number of nested if()s.

There is nothing especially new to us in the generated code, with the exception of the compiler moving input variable a to a temporary local variable `tv64` [1].

If we compile this in GCC 4.4.1, we'll get almost the same result, even with maximal optimization turned on (`-O3` option).

Optimizing MSVC

Now let's turn on optimization in MSVC (`/Ox`): `cl 1.c /Fa1.asm /Ox`

Listing 12.2: 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, 00↙
    ↳ H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00↙
    ↳ H
    jmp    _printf
$LN4@f:
```

[1]Local variables in stack are prefixed with `tv`—that's how MSVC names internal variables for its needs

```
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, ↙
    ↘ 00H
    jmp    _printf
_f    ENDP
```

Here we can see some dirty hacks.

First: the value of a is placed in EAX and 0 is subtracted from it. Sounds absurd, but it is done to check if the value in EAX was 0. If yes, the ZF flag is to be set (e.g. subtracting from 0 is 0) and the first conditional jump JE (*Jump if Equal* or synonym JZ —*Jump if Zero*) is to be triggered and control flow is to be passed to the $LN4@f label, where the 'zero' message is being printed. If the first jump doesn't get triggered, 1 is subtracted from the input value and if at some stage the result is 0, the corresponding jump is to be triggered.

And if no jump gets triggered at all, the control flow passes to printf() with string argument 'something unknown'.

Second: we see something unusual for us: a string pointer is placed into the a variable, and then printf() is called not via CALL, but via JMP. There is a simple explanation for that: the caller pushes a value to the stack and calls our function via CALL. CALL itself pushes the return address (RA[2]) to the stack and does an unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction that moves the stack pointer) has the following stack layout:

- ESP—points to RA
- ESP+4—points to the a variable

On the other side, when we need to call printf() here we need exactly the same stack layout, except for the first printf() argument, which needs to point to the string. And that is what our code does.

It replaces the function's first argument with the address of the string and jumps to printf(), as if we didn't call our function f(), but directly printf(). printf() prints a string to stdout and then executes the RET instruction, which POPs RA from the stack and control flow is returned not to f() but rather to f()'s callee, bypassing the end of the f() function.

All this is possible because printf() is called right at the end of the f() function in all cases. In some way, it is similar to the longjmp()[3] function. And of course, it is all done for the sake of speed.

[2]Return Address
[3]wikipedia

12.1.2 Conclusion

A *switch()* with few cases is indistinguishable from an *if/else* construction, for example: listing.12.1.1.

12.2 A lot of cases

If a `switch()` statement contains a lot of cases, it is not very convenient for the compiler to emit too large code with a lot JE/JNE instructions.

```
#include <stdio.h>

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;
    };
};

int main()
{
    f (2); // test
};
```

12.2.1 x86

Non-optimizing MSVC

We get (MSVC 2010):

Listing 12.3: 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 ; align next label
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
```

```
_f      ENDP
```

What we see here is a set of `printf()` calls with various arguments. All they have not only addresses in the memory of the process, but also internal symbolic labels assigned by the compiler. All these labels are also mentioned in the `$LN11@f` internal table.

At the function start, if a is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument `'something unknown'` is called.

But if the value of a is less or equals to 4, then it gets multiplied by 4 and added with the `$LN11@f` table address. That is how an address inside the table is constructed, pointing exactly to the element we need. For example, let's say a is equal to 2. $2 * 4 = 8$ (all table elements are addresses in a 32-bit process and that is why all elements are 4 bytes wide). The address of the `$LN11@f` table + 8 is the table element where the `$LN4@f` label is stored. JMP fetches the `$LN4@f` address from the table and jumps to it.

This table is sometimes called *jumptable* or *branch table*[4].

Then the corresponding `printf()` is called with argument `'two'`. Literally, the `jmp DWORD PTR $LN11@f[ecx*4]` instruction implies *jump to the DWORD that is stored at address* `$LN11@f + ecx * 4`.

is assembly language macro that aligning the next label so that it is to be stored at an address aligned on a 4 byte (or 16 byte) boundary. This is very suitable for the processor since it is able to fetch 32-bit values from memory through the memory bus, cache memory, etc, in a more effective way if it is aligned.

Non-optimizing GCC

Let's see what GCC 4.4.1 generates:

Listing 12.4: 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
```

[4]The whole method was once called *computed GOTO* in early versions of FORTRAN: wikipedia. Not quite relevant these days, but what a term!

```
       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
```

It is almost the same, with a little nuance: argument `arg_0` is multiplied by 4 by shifting it to left by 2 bits (it is almost the same as multiplication by 4) (15.2.1 on page 92). Then the address of the label is taken from the `off_804855C` array, stored in EAX, and then `JMP  EAX` does the actual jump.

12.2.2 Conclusion

Rough skeleton of *switch()*:

Listing 12.5: x86

```
MOV REG, input
CMP REG, 4 ; maximal number of cases
JA default
SHL REG, 2 ; find element in table.  shift for 3 bits in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
    ; do something
    JMP exit
case2:
    ; do something
    JMP exit
case3:
    ; do something
    JMP exit
case4:
    ; do something
    JMP exit
case5:
    ; do something
    JMP exit

default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

The jump to the address in the jump table may also be implemented using this instruction: `JMP jump_table[REG*4]`. Or `JMP jump_table[REG*8]` in x64.

A *jumptable* is just array of pointers, like the one described later: 16.4 on page 102.

12.3 When there are several *case* statements in one block

Here is a very widespread construction: several *case* statements for a single block:

```
#include <stdio.h>

void f(int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
                printf ("22\n");
                break;
        default:
                printf ("default\n");
                break;
        };
};
```

```
int main()
{
        f(4);
};
```

It's too wasteful to generate a block for each possible case, so what is usually done is to generate each block plus some kind of dispatcher.

12.3.1 MSVC

Listing 12.6: Optimizing MSVC 2010

```
$SG2798 DB      '1, 2, 7, 10', 0aH, 00H
$SG2800 DB      '3, 4, 5', 0aH, 00H
$SG2802 DB      '8, 9, 21', 0aH, 00H
$SG2804 DB      '22', 0aH, 00H
$SG2806 DB      'default', 0aH, 00H

_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        dec     eax
        cmp     eax, 21
        ja      SHORT $LN1@f
        movzx   eax, BYTE PTR $LN10@f[eax]
        jmp     DWORD PTR $LN11@f[eax*4]
$LN5@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
        jmp     DWORD PTR __imp__printf
$LN4@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
        jmp     DWORD PTR __imp__printf
$LN3@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
        jmp     DWORD PTR __imp__printf
$LN2@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
        jmp     DWORD PTR __imp__printf
$LN1@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
        jmp     DWORD PTR __imp__printf
        npad    2 ; align $LN11@f table on 16-byte boundary
```

```
31 $LN11@f:
32         DD      $LN5@f ; print '1, 2, 7, 10'
33         DD      $LN4@f ; print '3, 4, 5'
34         DD      $LN3@f ; print '8, 9, 21'
35         DD      $LN2@f ; print '22'
36         DD      $LN1@f ; print 'default'
37 $LN10@f:
38         DB      0 ; a=1
39         DB      0 ; a=2
40         DB      1 ; a=3
41         DB      1 ; a=4
42         DB      1 ; a=5
43         DB      1 ; a=6
44         DB      0 ; a=7
45         DB      2 ; a=8
46         DB      2 ; a=9
47         DB      0 ; a=10
48         DB      4 ; a=11
49         DB      4 ; a=12
50         DB      4 ; a=13
51         DB      4 ; a=14
52         DB      4 ; a=15
53         DB      4 ; a=16
54         DB      4 ; a=17
55         DB      4 ; a=18
56         DB      4 ; a=19
57         DB      2 ; a=20
58         DB      2 ; a=21
59         DB      3 ; a=22
60 _f      ENDP
```

We see two tables here: the first table (`$LN10@f`) is an index table, and the second one (`$LN11@f`) is an array of pointers to blocks.

First, the input value is used as an index in the index table (line 13).

Here is a short legend for the values in the table: 0 is the first *case* block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block.

There we get an index for the second table of code pointers and we jump to it (line 14).

What is also worth noting is that there is no case for input value 0. That's why we see the DEC instruction at line 10, and the table starts at $a = 1$, because there is no need to allocate a table element for $a = 0$.

This is a very widespread pattern.

So why is this economical? Why isn't it possible to make it as before (12.2.1 on page 69), just with one table consisting of block pointers? The reason is that the elements in index table are 8-bit, hence it's all more compact.

12.4 Fall-through

Another very popular usage of `switch()` is the fall-through. Here is a small example:

```
 1  #define R 1
 2  #define W 2
 3  #define RW 3
 4
 5  void f(int type)
 6  {
 7          int read=0, write=0;
 8
 9          switch (type)
10          {
11          case RW:
12                  read=1;
13          case W:
14                  write=1;
15                  break;
16          case R:
17                  read=1;
18                  break;
19          default:
20                  break;
21          };
22          printf ("read=%d, write=%d\n", read, write);
23  };
```

If $type = 1$ (R), $read$ is to be set to 1, if $type = 2$ (W), $write$ is to be set to 2. In case of $type = 3$ (RW), both $read$ and $write$ is to be set to 1.

The code at line 14 is executed in two cases: if $type = RW$ or if $type = W$. There is no "break" for "case RW"x and that's OK.

12.4.1 MSVC x86

Listing 12.7: MSVC 2012

```
$SG1305 DB      'read=%d, write=%d', 0aH, 00H
```

```
_write$ = -12   ; size = 4
_read$ = -8     ; size = 4
tv64 = -4       ; size = 4
_type$ = 8      ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     DWORD PTR _read$[ebp], 0
        mov     DWORD PTR _write$[ebp], 0
        mov     eax, DWORD PTR _type$[ebp]
        mov     DWORD PTR tv64[ebp], eax
        cmp     DWORD PTR tv64[ebp], 1 ; R
        je      SHORT $LN2@f
        cmp     DWORD PTR tv64[ebp], 2 ; W
        je      SHORT $LN3@f
        cmp     DWORD PTR tv64[ebp], 3 ; RW
        je      SHORT $LN4@f
        jmp     SHORT $LN5@f
$LN4@f: ; case RW:
        mov     DWORD PTR _read$[ebp], 1
$LN3@f: ; case W:
        mov     DWORD PTR _write$[ebp], 1
        jmp     SHORT $LN5@f
$LN2@f: ; case R:
        mov     DWORD PTR _read$[ebp], 1
$LN5@f: ; default
        mov     ecx, DWORD PTR _write$[ebp]
        push    ecx
        mov     edx, DWORD PTR _read$[ebp]
        push    edx
        push    OFFSET $SG1305 ; 'read=%d, write=%d'
        call    _printf
        add     esp, 12
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

The code mostly resembles what is in the source. There are no jumps between labels `$LN4@f` and `$LN3@f`: so when code flow is at `$LN4@f`, $read$ is first set to 1, then $write$. This is why it's called fall-through: code flow falls through one piece of code (setting $read$) to another (setting $write$). If $type = W$, we land at `$LN3@f`, so no code setting $read$ to 1 is executed.

Chapter 13

Loops

13.1 Simple example

13.1.1 x86

There is a special LOOP instruction in x86 instruction set for checking the value in register ECX and if it is not 0, to decrement ECX and pass control flow to the label in the LOOP operand. Probably this instruction is not very convenient, and there are no any modern compilers which emit it automatically. So, if you see this instruction somewhere in code, it is most likely that this is a manually written piece of assembly code.

In C/C++ loops are usually constructed using `for()`, `while()` or `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is the counter bigger than a limit?), what is done at each iteration (increment/decrement) and of course loop body.

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

The generated code is consisting of four parts as well.

Let's start with a simple example:

```
#include <stdio.h>

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

int main()
{
        int i;

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

        return 0;
};
```

Result (MSVC 2010):

Listing 13.1: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; loop initialization
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; here is what we do after
    each iteration:
    add    eax, 1                  ; add 1 to (i) value
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; this condition is checked
    *before* each iteration
    jge    SHORT $LN1@main         ; if (i) is biggest or equals
    to 10, lets finish loop'
    mov    ecx, DWORD PTR _i$[ebp] ; loop body:
    call printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; jump to loop begin
$LN1@main:                         ; loop end
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

As we see, nothing special.

Listing 13.2: Optimizing MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

What happens here is that space for the i variable is not allocated in the local stack anymore, but uses an individual register for it, ESI. This is possible in such small functions where there aren't many local variables.

One very important thing is that the f() function must not change the value in ESI. Our compiler is sure here. And if the compiler decides to use the ESI register in f() too, its value would have to be saved at the function's prologue and restored at the function's epilogue, almost like in our listing: please note PUSH ESI/POP ESI at the function start and end.

13.1.2 One more thing

In the generated code we can see: after initializing i, the body of the loop is not to be executed, as the condition for i is checked first, and only after that loop body can be executed. And that is correct. Because, if the loop condition is not met at the beginning, the body of the loop must not be executed. This is possible in the following case:

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

If *total_entries_to_process* is 0, the body of the loo must not be executed at all. This is why the condition checked before the execution.

However, an optimizing compiler may swap the condition check and loop body, if it sure that the situation described here is not possible (like in the case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode).

13.2 Memory blocks copying routine

Real-world memory copy routines may copy 4 or 8 bytes at each iteration, use SIMD[1], vectorization, etc. But for the sake of simplicity, this example is the simplest possible.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t ↙
    ↘ cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

13.2.1 Straight-forward implementation

Listing 13.3: GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block

; initialize counter (i) at 0
        xor     eax, eax
.L2:
; all bytes copied? exit then:
        cmp     rax, rdx
        je      .L5
; load byte at RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; store byte at RDI+i:
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
```

[1]Single instruction, multiple data

```
        ret
```

13.3 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

Listing 13.4: x86

```
    mov [counter], 2 ; initialization
    jmp check
body:
    ; loop body
    ; do something here
    ; use counter variable in local stack
    add [counter], 1 ; increment
check:
    cmp [counter], 9
    jle body
```

The increment operation may be represented as 3 instructions in non-optimized code:

Listing 13.5: x86

```
    MOV [counter], 2 ; initialization
    JMP check
body:
    ; loop body
    ; do something here
    ; use counter variable in local stack
    MOV REG, [counter] ; increment
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

If the body of the loop is short, a whole register can be dedicated to the counter variable:

Listing 13.6: x86

```
    MOV EBX, 2 ; initialization
    JMP check
body:
    ; loop body
    ; do something here
    ; use counter in EBX, but do not modify it!
    INC EBX ; increment
check:
    CMP EBX, 9
    JLE body
```

Some parts of the loop may be generated by compiler in different order:

Listing 13.7: x86

```
    MOV [counter], 2 ; initialization
    JMP label_check
label_increment:
    ADD [counter], 1 ; increment
label_check:
    CMP [counter], 10
    JGE exit
    ; loop body
    ; do something here
    ; use counter variable in local stack
    JMP label_increment
exit:
```

Usually the condition is checked *before* loop body, but the compiler may rearrange it in a way that the condition is checked *after* loop body. This is done when the compiler is sure that the condition is always *true* on the first iteration, so the body of the loop is to be executed at least once:

Listing 13.8: x86

```
    MOV REG, 2 ; initialization
body:
    ; loop body
    ; do something here
    ; use counter in REG, but do not modify it!
    INC REG ; increment
    CMP REG, 10
    JL body
```

Using the LOOP instruction. This is rare, compilers are not using it. When you see it, it's a sign that this piece of code is hand-written:

Listing 13.9: x86

```
    ; count from 10 to 1
    MOV ECX, 10
body:
    ; loop body
    ; do something here
    ; use counter in ECX, but do not modify it!
    LOOP body
```

Chapter 14

Simple C-strings processing

14.1 strlen()

Let's talk about loops one more time. Often, the `strlen()` function[1] is implemented using a `while()` statement. Here is how it is done in the MSVC standard libraries:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

[1] counting the characters in a string in the C language

14.1.1 x86

Non-optimizing MSVC

Let's compile:

```
_eos$ = -4                              ; size = 4
_str$ = 8                               ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; place pointer to string
    from "str"
    mov     DWORD PTR _eos$[ebp], eax  ; place it to local
    variable "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit
    value to EDX with sign extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax  ; place EAX back to "eos"
    test    edx, edx                   ; EDX is zero?
    je      SHORT $LN1@strlen_         ; yes, then finish loop
    jmp     SHORT $LN2@strlen_         ; continue loop
$LN1@strlen_:

    ; here we calculate the difference between two pointers

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                      ; subtract 1 and return
    result
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

We get two new instructions here: MOVSX and TEST.

The first one—MOVSX—takes a byte from an address in memory and stores the value in a 32-bit register. MOVSX stands for *MOV with Sign-Extend*. MOVSX sets the rest of the bits, from the 8th to the 31th, to 1 if the source byte is *negative* or to 0 if is *positive*.

And here is why.

By default, the *char* type is signed in MSVC and GCC. If we have two values of which one is *char* and the other is *int*, (*int* is signed too), and if the first value contain -2 (coded as 0xFE) and we just copy this byte into the *int* container, it makes 0x000000FE, and this from the point of signed *int* view is 254, but not -2. In signed int, -2 is coded as 0xFFFFFFFE. So if we need to transfer 0xFE from a variable of *char* type to *int*, we need to identify its sign and extend it. That is what MOVSX does.

You can also read about it in "*Signed number representations*" section (22 on page 157).

It's hard to say if the compiler needs to store a *char* variable in EDX, it could just take a 8-bit register part (for example DL). Apparently, the compiler's register allocator works like that.

Then we see TEST EDX, EDX. You can read more about the TEST instruction in the section about bit fields (17 on page 112). Here this instruction just checks if the value in EDX equals to 0.

Optimizing MSVC

Now let's compile all this in MSVC 2012, with optimizations turned on (/Ox):

Listing 14.1: Optimizing MSVC 2012 /Ob0

```
_str$ = 8                               ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to
    the string
        mov     eax, edx                    ; move to EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; no, continue loop
        sub     eax, edx                    ; calculate pointers
    difference
        dec     eax                         ; decrement EAX
        ret     0
_strlen ENDP
```

Now it is all simpler. Needless to say, the compiler could use registers with such efficiency only in small functions with a few local variables.

INC/DEC— are increment/decrement instructions, in other words: add or substract 1 to/from a variable.

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

Chapter 15

Replacing arithmetic instructions to other ones

In the pursuit of optimization, one instruction may be replaced by another, or even with a group of instructions. For example, ADD and SUB can replace each other: line 18 in listing.**??**.

15.1 Multiplication

15.1.1 Multiplication using addition

Here is a simple example:

Listing 15.1: Optimizing MSVC 2010

```
unsigned int f(unsigned int a)
{
        return a*8;
};
```

Multiplication by 8 is replaced by 3 addition instructions, which do the same. Apparently, MSVC's optimizer decided that this code can be faster.

```
_TEXT   SEGMENT
_a$ = 8                                                  ; size ↙
    ↘ = 4
_f      PROC
; File c:\polygon\c\2.c
```

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

```
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

15.1.2 Multiplication using shifting

Multiplication and division instructions by a numbers that's a power of 2 are often replaced by shift instructions.

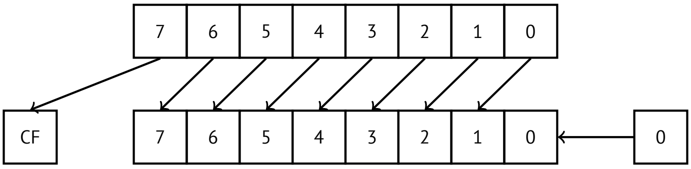
```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

Listing 15.2: Non-optimizing MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

Multiplication by 4 is just shifting the number to the left by 2 bits and inserting 2 zero bits at the right (as the last two bits). It is just like multiplying 3 by 100 —we need to just add two zeroes at the right.

That's how the shift left instruction works:



The added bits at right are always zeroes.

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

15.1.3 Multiplication using shifting, subtracting, and adding

It’s still possible to get rid of the multiplication operation when you multiply by numbers like 7 or 17 again by using shifting. The mathematics used here is relatively easy.

32-bit

```
#include <stdint.h>

int f1(int a)
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Listing 15.3: Optimizing MSVC 2012

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
```

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

```
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

Listing 15.4: Optimizing MSVC 2012

```
; a*7
f1:
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

15.2 Division

15.2.1 Division using shifts

Example of division by 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

We get (MSVC 2010):

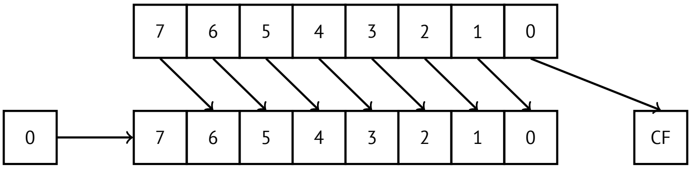
Listing 15.5: MSVC 2010

CHAPTER 15. REPLACING ARITHMETIC INSTRUCTIONS TO OTHER ONES

```
_a$ = 8                                                         ; size ⤦
    ↪ = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

The SHR (*SHift Right*) instruction in this example is shifting a number by 2 bits to the right. The two freed bits at left (e.g., two most significant bits) are set to zero. The two least significant bits are dropped. In fact, these two dropped bits are the division operation remainder.

The SHR instruction works just like SHL, but in the other direction.



It is easy to understand if you imagine the number 23 in the decimal numeral system. 23 can be easily divided by 10 just by dropping last digit (3—division remainder). 2 is left after the operation as a quotient.

So the remainder is dropped, but that's OK, we work on integer values anyway, these are not a real numbers!

Chapter 16

Arrays

An array is just a set of variables in memory that lie next to each other and that have the same type[1].

16.1 Simple example

```
#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;
};
```

[1] AKA[2] "homogeneous container"

16.1.1 x86

MSVC

Let's compile:

Listing 16.1: MSVC 2008

```
_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
    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
```

Nothing very special, just two loops: the first is a filling loop and second is a printing loop. The `shl ecx, 1` instruction is used for value multiplication by 2 in ECX, more about below 15.2.1 on page 92.

80 bytes are allocated on the stack for the array, 20 elements of 4 bytes.

16.2 Buffer overflow

16.2.1 Reading outside array bounds

So, array indexing is just *array[index]*. If you study the generated code closely, you'll probably note the missing index bounds checking, which could check *if it is less than 20*. What if the index is 20 or greater? That's the one C/C++ feature it is often blamed for.

Here is a code that successfully compiles and works:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[20]=%d\n", a[20]);

        return 0;
};
```

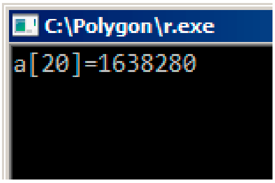
Compilation results (MSVC 2008):

Listing 16.2: Non-optimizing MSVC 2008

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
```

```
_a$ = -80 ; size = 80
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    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
    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+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

The code produced this result:



```
a[20]=1638280
```

Figure 16.1: OllyDbg: console output

It is just *something* that was lying in the stack near to the array, 80 bytes away from its first element.

16.2.2 Writing beyond array bounds

OK, we read some values from the stack *illegally*, but what if we could write something to it?

Here is what we have got:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

MSVC

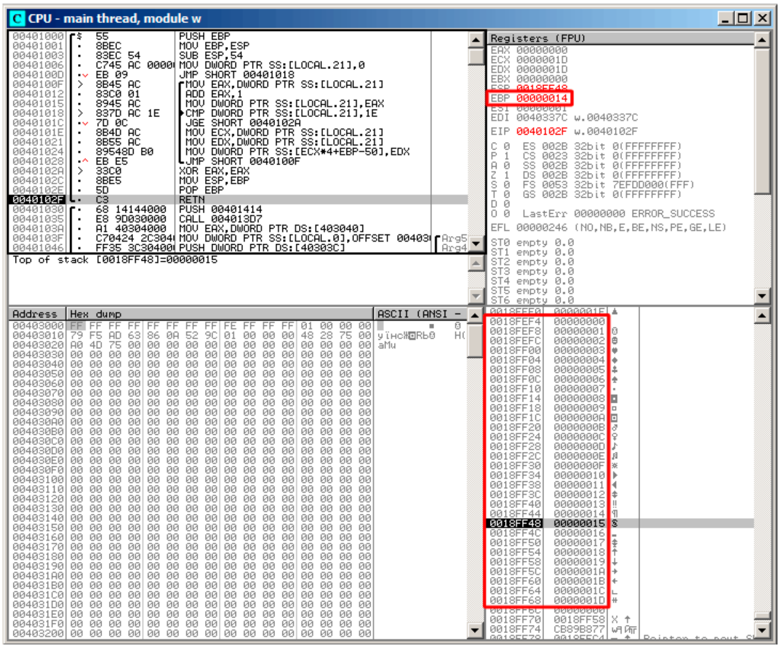
And what we get:

Listing 16.3: Non-optimizing MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 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$[ebp]      ; that instruction is
    obviously redundant
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as
    second operand here instead
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

The compiled program crashes after running. No wonder. Let’s see where exactly does it is crash.

Let’s load it into OllyDbg, and trace until all 30 elements are written:


CPU - main thread, module w

```
00401000  $  55            PUSH EBP
00401001  .  8BEC          MOV EBP,ESP
00401003  .  83EC 54       SUB ESP,54
00401006  .  C745 AC 0000  MOV DWORD PTR SS:[LOCAL.21],0
0040100D  .v EB 09         JMP SHORT 00401018
0040100F  >  8B45 AC       MOV EAX,DWORD PTR SS:[LOCAL.21]
00401012  .  83C0 01       ADD EAX,1
00401015  .  8945 AC       MOV DWORD PTR SS:[LOCAL.21],EAX
00401018  >  837D AC 1E    CMP DWORD PTR SS:[LOCAL.21],1E
0040101C  .v 7D 0C         JGE SHORT 0040102A
0040101E  .  8B4D AC       MOV ECX,DWORD PTR SS:[LOCAL.21]
00401021  .  8B55 AC       MOV EDX,DWORD PTR SS:[LOCAL.21]
00401024  .  89548D B0     MOV DWORD PTR SS:[ECX*4+EBP-50],EDX
00401028  .^ EB E5         JMP SHORT 0040100F
0040102A  >  33C0          XOR EAX,EAX
0040102C  .  8BE5          MOV ESP,EBP
0040102E  .  5D            POP EBP
0040102F  .  C3            RETN
00401030  .  68 14144000   PUSH 00401414
00401035  .  E8 9D030000   CALL 004013D7
0040103A  .  A1 40304000   MOV EAX,DWORD PTR DS:[403040]
0040103F  .  C70424 2C304  MOV DWORD PTR SS:[LOCAL.0],OFFSET 00403   Arg5
00401046  .  FF35 3C30400  PUSH DWORD PTR DS:[40303C]                Arg4
Top of stack [0018FF48]=00000015
```

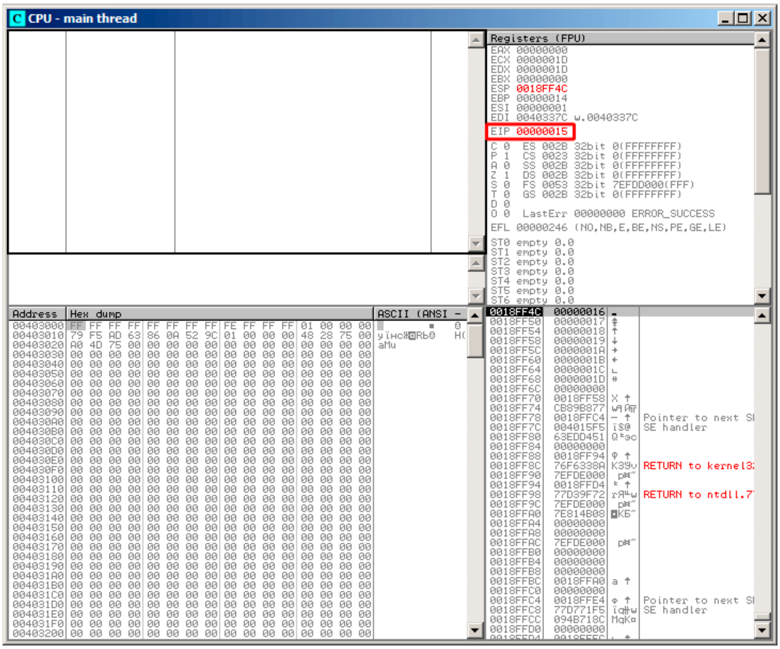
```
Registers (FPU)
EAX 00000000
ECX 0000001D
EDX 0000001D
EBX 00000000
ESP 0018FF48
EBP 00000014
ESI 00000001
EDI 0040337C w.0040337C

EIP 0040102F w.0040102F

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
```

```
Address   Hex dump                                         ASCII (ANSI -
00403000  FF FF FF FF FF FF FF FF FE FF FF FF 01 00 00 00             0
00403010  79 F5 AD 63 86 0A 52 9C 01 00 00 00 48 28 75 00  yїнcЖ◘Rb0   H(
00403020  A0 4D 75 00 00 00 00 00 00 00 00 00 00 00 00 00  aMu
00403030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0018FEF0  0000001F  ▲
0018FEF4  00000000
0018FEF8  00000001  ☺
0018FEFC  00000002  ☻
0018FF00  00000003  ♥
0018FF04  00000004  ♦
0018FF08  00000005  ♣
0018FF0C  00000006  ♠
0018FF10  00000007  •
0018FF14  00000008  ◘
0018FF18  00000009  ○
0018FF1C  0000000A  ◙
0018FF20  0000000B  ♂
0018FF24  0000000C  ♀
0018FF28  0000000D  ♪
0018FF2C  0000000E  ♫
0018FF30  0000000F  ☼
0018FF34  00000010  ►
0018FF38  00000011  ◄
0018FF3C  00000012  ↕
0018FF40  00000013  ‼
0018FF44  00000014  ¶
0018FF48  00000015  §
0018FF4C  00000016  ▬
0018FF50  00000017  ↨
0018FF54  00000018  ↑
0018FF58  00000019  ↓
0018FF5C  0000001A  →
0018FF60  0000001B  ←
0018FF64  0000001C  ∟
0018FF68  0000001D  ↔
0018FF6C  00000000
0018FF70  0018FF58  X ↑
0018FF74  CB89B877  wЯ ЙЛ
0018FF78  0018FFC4  — ↑   Pointer to next SE
```

Figure 16.2: OllyDbg: after restoring the value of EBP

Trace until the function end:



```
Registers (FPU)
EAX 00000000
ECX 0000001D
EDX 0000001D
EBX 00000000
ESP 0018FF4C
EBP 00000014
ESI 00000001
EDI 0040337C w.0040337C
EIP 00000015
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
```

```
Address  Hex dump                                         ASCII (ANSI -
00403000 FF FF FF FF FF FF FF FF FE FF FF FF 01 00 00 00           0
00403010 79 F5 AD 63 86 0A 52 9C 01 00 00 00 48 28 75 00 yïнc※◘Rb0    H(
00403020 A0 4D 75 00 00 00 00 00 00 00 00 00 00 00 00 00 aMu
00403030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004030F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0018FF4C  00000016
0018FF50  00000017
0018FF54  00000018
0018FF58  00000019
0018FF5C  0000001A
0018FF60  0000001B
0018FF64  0000001C
0018FF68  0000001D
0018FF6C  00000000
0018FF70  0018FF58  X ↑
0018FF74  CB89B877
0018FF78  0018FFC4  ─ ↑   Pointer to next SE
0018FF7C  004015F5  ïS@   SE handler
0018FF80  63EDD451
0018FF84  00000000
0018FF88  0018FF94  φ ↑
0018FF8C  76F6338A  K3Ўv  RETURN to kernel3
0018FF90  7EFDE000
0018FF94  0018FFD4    ↑
0018FF98  77D39F72  rЯ╨w  RETURN to ntdll.7
0018FF9C  7EFDE000
0018FFA0  7E814B08  ◘KБ~
0018FFA4  00000000
0018FFA8  00000000
0018FFAC  7EFDE000
0018FFB0  00000000
0018FFB4  00000000
0018FFB8  00000000
0018FFBC  0018FFA0  a ↑
0018FFC0  00000000
0018FFC4  0018FFE4  φ ↑   Pointer to next SE
0018FFC8  77D771F5  ïq╫w  SE handler
0018FFCC  094B718C  MqK
0018FFD0  00000000
```

Figure 16.3: OllyDbg: EIP was restored, but OllyDbg can't disassemble at 0x15

Now please keep your eyes on the registers.

EIP is 0x15 now. It is not a legal address for code—at least for win32 code! We got there somehow against our will. It is also interesting that the EBP register contain 0x14, ECX and EDX—0x1D.

Let's study stack layout a bit more.

After the control flow was passed to main(), the value in the EBP register was saved on the stack. Then, 84 bytes were allocated for the array and the i variable. That's (20+1)*sizeof(int). ESP now points to the _i variable in the local stack and after the execution of the next PUSH something, *something* is appearing next to _i.

That's the stack layout while the control is in main():

ESP	4 bytes allocated for i variable
ESP+4	80 bytes allocated for a[20] array
ESP+84	saved EBP value
ESP+88	return address

`a[19]=something` statement writes the last *int* in the bounds of the array (in bounds so far!)

`a[20]=something` statement writes *something* to the place where the value of EBP is saved.

Please take a look at the register state at the moment of the crash. In our case, 20 was written in the 20th element. At the function end, the function epilogue restores the original EBP value. (20 in decimal is `0x14` in hexadecimal). Then RET gets executed, which is effectively equivalent to `POP EIP` instruction.

The RET instruction takes the return address from the stack (that is the address in CRT), which was called `main()`), and 21 iss stored there (`0x15` in hexadecimal). The CPU traps at address `0x15`, but there is no executable code there, so exception gets raised.

Welcome! It is called a *buffer overflow*[3].

Replace the *int* array with a string (*char* array), create a long string deliberately and pass it to the program, to the function, which doesn't check the length of the string and copies it in a short buffer, and you'll able to point the program to an address to which it must jump. It's not that simple in reality, but that is how it emerged[4]

16.3 One more word about arrays

Now we understand why it is impossible to write something like this in C/C++ code:

```
void f(int size)
{
    int a[size];
...
};
```

That's just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.

If you need an array of arbitrary size, allocate it by using `malloc()`, then access the allocated memory block as an array of variables of the type you need.

Or use the C99 standard feature[ISO07, pp. 6.7.5/2], and it works like alloca() (5.2.4 on page 17) internally.

It's also possible to use garbage collecting libraries for C. And there are also libraries supporting smart pointers for C++.

[3]wikipedia
[4]Classic article about it: [One96].

16.4 Array of pointers to strings

Here is an example for an array of pointers.

Listing 16.4: Get month name

```
#include <stdio.h>

const char* month1[]=
{
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
        return month1[month];
};
```

16.4.1 x64

Listing 16.5: Optimizing MSVC 2013 x64

```
_DATA   SEGMENT
month1  DQ      FLAT:$SG3122
        DQ      FLAT:$SG3123
        DQ      FLAT:$SG3124
        DQ      FLAT:$SG3125
        DQ      FLAT:$SG3126
        DQ      FLAT:$SG3127
        DQ      FLAT:$SG3128
        DQ      FLAT:$SG3129
        DQ      FLAT:$SG3130
        DQ      FLAT:$SG3131
        DQ      FLAT:$SG3132
        DQ      FLAT:$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

The code is very simple:

- The first MOVSXD instruction copies a 32-bit value from ECX (where $month$ argument is passed) to RAX with sign-extension (because the $month$ argument is of type *int*). The reason for the sign extension is that this 32-bit value is to be used in calculations with other 64-bit values. Hence, it has to be promoted to 64-bit[5].
- Then the address of the pointer table is loaded into RCX.
- Finally, the input value ($month$) is multiplied by 8 and added to the address. Indeed: we are in a 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element is 8 bytes wide. And that's why to pick a specific element, $month * 8$ bytes has to be skipped from the start. That's what MOV does. In addition, this instruction also loads the element at this address. For 1, an element would be a pointer to a string that contains "February", etc.

Optimizing GCC 4.9 can do the job even better[6]:

[5] It is somewhat weird, but negative array index could be passed here as $month$. And if this happens, the negative input value of *int* type is sign-extended correctly and the corresponding element before table is picked. It is not going to work correctly without sign-extension.

[6] "0+" was left in the listing because GCC assembler output is not tidy enough to eliminate it. It's *displacement*, and it's zero here.

Listing 16.6: Optimizing GCC 4.9 x64

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

32-bit MSVC

Let's also compile it in the 32-bit MSVC compiler:

Listing 16.7: Optimizing MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

The input value does not need to be extended to 64-bit value, so it is used as is. And it's multiplied by 4, because the table elements are 32-bit (or 4 bytes) wide.

16.5 Multidimensional arrays

Internally, a multidimensional array is essentially the same thing as a linear array. Since the computer memory is linear, it is an one-dimensional array. For convenience, this multi-dimensional array can be easily represented as one-dimensional.

For example, this is how the elements of the 3x4 array are placed in one-dimensional array of 12 cells:

Offset in memory	array element
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Table 16.1: Two-dimensional array represented in memory as one-dimensional

Here is how each cell of 3*4 array are placed in memory:

0	1	2	3
4	5	6	7
8	9	10	11

Table 16.2: Memory addresses of each cell of two-dimensional array

So, in order to calculate the address of the element we need, we first multiply the first index by 4 (array width) and then add the second index. That's called *row-major order*, and this method of array and matrix representation is used in at least C/C++ and Python. The term *row-major order* in plain English language means: "first, write the elements of the first row, then the second row …and finally the elements of the last row".

Another method for representation is called *column-major order* (the array indices are used in reverse order) and it is used at least in FORTRAN, MATLAB and R. *column-major order* term in plain English language means: "first, write the elements of the first column, then the second column …and finally the elements of the last column".

Which method is better? In general, in terms of performance and cache memory, the best scheme for data organization is the one, in which the elements are accessed sequentially. So if your function accesses data per row, *row-major order* is better, and vice versa.

16.5.1 Two-dimensional array example

We are going to work with an array of type *char*, which implies that each element requires only one byte in memory.

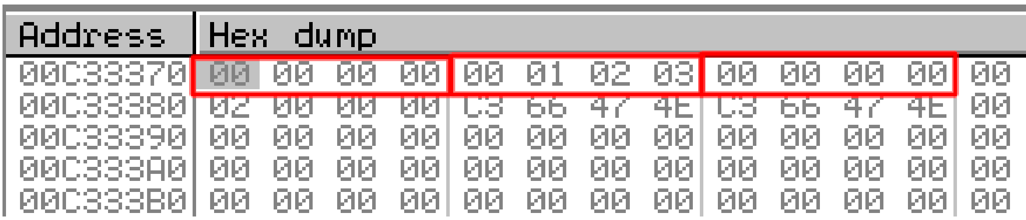
Row filling example

Let's fill the second row with these values 0..3:

Listing 16.8: Row filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // clear array
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // fill second row by 0..3:
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

All three rows are marked with red. We see that second row now has values 0, 1, 2 and 3:



```
Address   Hex dump
00C33370  00 00 00 00 00 01 02 03 00 00 00 00 00
00C33380  02 00 00 00 C3 66 47 4E C3 66 47 4E 00
00C33390  00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0  00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0  00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 16.4: OllyDbg: array is filled

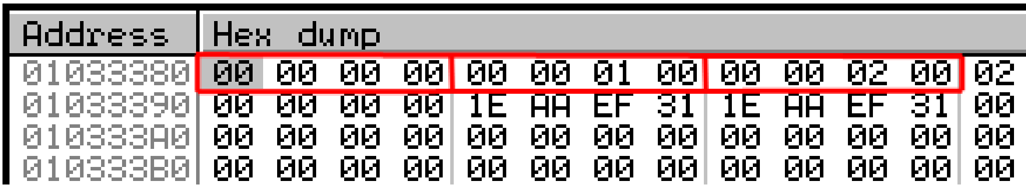
Column filling example

Let's fill the third column with values: 0..2:

Listing 16.9: Column filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // clear array
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // fill third column by 0..2:
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

The three rows are also marked in red here. We see that in each row, at third position these values are written: 0, 1 and 2.



```
Address   Hex dump
01033380  00 00 00 00 00 00 01 00 00 00 02 00 02
01033390  00 00 00 00 1E AA EF 31 1E AA EF 31 00
010333A0  00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0  00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 16.5: OllyDbg: array is filled

16.5.2 Access two-dimensional array as one-dimensional

We can be easily assured that it's possible to access a two-dimensional array as one-dimensional array in at least two ways:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // treat input array as one-dimensional
        // 4 is array width here
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
        // treat input array as pointer,
        // calculate address, get value at it
        // 4 is array width here
        return *(array+a*4+b);
};

int main()
{
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Compile and run it: it shows correct values.

What MSVC 2013 did is fascinating, all three routines are just the same!

Listing 16.10: Optimizing MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=address of array
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+address of array
        movzx   eax, BYTE PTR [rax+r9*4]
```

```
; AL=load byte at address RAX+R9*4=b+address of
    array+a*4=address of array+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

16.5.3 Three-dimensional array example

It's thing in multidimensional arrays. Now we are going to work with an array of type *int*: each element requires 4 bytes in memory.

Let's see:

Listing 16.11: simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

We get (MSVC 2010):

Listing 16.12: 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
```

Nothing special. For index calculation, three input arguments are used in the formula $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$, to represent the array as multidimensional. Do not forget that the *int* type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

Listing 16.13: 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
```

The GCC compiler does it differently. For one of the operations in the calculation ($30y$), GCC produces code without multiplication instructions. This is how it done: $(y+y) \ll 4 - (y+y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Thus, for the $30y$ calculation, only one addition operation, one bitwise shift operation and one subtraction operation are used. This works faster.

16.6 Conclusion

An array is a pack of values in memory located adjacently. It's true for any element type, including structures. Access to a specific array element is just a calculation of its address.

Chapter 17

Manipulating specific bit(s)

A lot of functions define their input arguments as flags in bit fields. Of course, they could be substituted by a set of *bool*-typed variables, but it is not frugally.

17.1 Specific bit checking

17.1.1 x86

Win32 API example:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ⤦
    ↳ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL⤦
    ↳ , NULL);
```

We get (MSVC 2010):

Listing 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
```

Let's take a look in WinNT.h:

Listing 17.2: WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Everything is clear, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, and that value is used as the second argument for the `CreateFile()`[1]function.

How would `CreateFile()` check these flags? If we look in KERNEL32.DLL in Windows XP SP3 x86, we'll find this fragment of code in `CreateFileW`:

Listing 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
```

Here we see the TEST instruction, however it doesn't take the whole second argument, but only the most significant byte (`ebp+dwDesiredAccess+3`) and checks it for flag `0x40` (which implies the `GENERIC_WRITE` flag here) TEST is basically the same instruction as AND, but without saving the result (recall the fact CMP is merely the same as SUB, but without saving the result (7.3.1 on page 35)).

The logic of this code fragment is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If AND instruction leaves this bit, the ZF flag is to be cleared and the JZ conditional jump is not to be triggered. The conditional jump is triggered only if the `0x40000000` bit is absent in `dwDesiredAccess` variable —then the result of AND is 0, ZF is to be set and the conditional jump is to be triggered.

17.2 Setting and clearing specific bits

For example:

[1] msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

```
#include <stdio.h>

#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;
};

int main()
{
    f(0x12340678);
};
```

17.2.1 x86

Non-optimizing MSVC

We get (MSVC 2010):

Listing 17.4: 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
```

The OR instruction sets one bit to value while ignoring the rest.

AND resets one bit. It can be said that AND just copies all bits except one. Indeed, in the second AND operand only the bits that need to be saved are set, just the one do not want to copy is not (which is 0 in the bitmask). It is the easier way to memorize the logic.

Optimizing MSVC

If we compile it in MSVC with optimization turned on (/Ox), the code is even shorter:

Listing 17.5: Optimizing 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
```

17.3 Shifts

Bit shifts in C/C++ are implemented using ≪ and ≫ operators.

The x86 ISA has the SHL (SHift Left) and SHR (SHift Right) instructions for this.

Shift instructions are often used in division and multiplications by powers of two: 2^n (e.g., 1, 2, 4, 8, etc): 15.1.2 on page 88, 15.2.1 on page 91.

Shifting operations are also so important because they are often used for specific bit isolation or for constructing a value of several scattered bits.

17.4 Counting bits set to 1

Here is a simple example of a function that calculates the number of bits set in the input value.

This operation is also called “population count”[2].

```
#include <stdio.h>

#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;
};

int main()
{
    f(0x12345678); // test
};
```

In this loop, the iteration count value i is counting from 0 to 31, so the $1 \ll i$ statement is counting from 1 to 0x80000000. Describing this operation in natural language, we would say *shift 1 by n bits left*. In other words, $1 \ll i$ statement consequently produces all possible bit positions in a 32-bit number. The freed bit at right is always cleared.

Here is a table of all possible $1 \ll i$ for $i = 0 \ldots 31$:

[2]modern x86 CPUs (supporting SSE4) even have a POPCNT instruction for it

C/C++ expression	Power of two	Decimal form	Hexadecimal form
$1 \ll 0$	1	1	1
$1 \ll 1$	2^1	2	2
$1 \ll 2$	2^2	4	4
$1 \ll 3$	2^3	8	8
$1 \ll 4$	2^4	16	0x10
$1 \ll 5$	2^5	32	0x20
$1 \ll 6$	2^6	64	0x40
$1 \ll 7$	2^7	128	0x80
$1 \ll 8$	2^8	256	0x100
$1 \ll 9$	2^9	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

These constant numbers (bit masks) very often appear in code and a practicing reverse engineer must be able to spot them quickly. You probably haven't to memorize the decimal numbers, but the hexadecimal ones are very easy to remember.

These constants are very often used for mapping flags to specific bits. For example, here is excerpt from `ssl_private.h` from Apache 2.4.6 source code:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE           (0)
#define SSL_OPT_RELSET         (1<<0)
#define SSL_OPT_STDENVVARS     (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH  (1<<4)
#define SSL_OPT_STRICTREQUIRE  (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Let's get back to our example.

The IS_SET macro checks bit presence in a. The IS_SET macro is in fact the logical AND operation (*AND*) and it returns 0 if the specific bit is absent there, or the bit mask, if the bit is present. *The if()* operator in C/C++ triggers if the expression in it is not zero, it might be even 123456, that is why it always works correctly.

17.4.1 x86

MSVC

Let's compile (MSVC 2010):

Listing 17.6: 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]   ; increment of i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; loop finished?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je      SHORT $LN1@f                  ; result of AND instruction
    was 0?
                                          ; then skip next
    instructions
    mov     eax, DWORD PTR _rt$[ebp]  ; no, not zero
    add     eax, 1                        ; increment 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
```

17.4.2 x64

Let's modify the example slightly to extend it to 64-bit:

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

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

Optimizing MSVC 2010

Listing 17.7: MSVC 2010

```
a$ = 8
f       PROC
; RCX = input value
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f:
        test    rdx, rcx
; there are no such bit in input value?
; skip the next INC instruction then.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Here the ROL instruction is used instead of SHL, which is in fact "rotate left" instead of "shift left", but in this example it works just as SHL.

R8 here is counting from 64 to 0. It's just like an inverted i.

Here is a table of some registers during the execution:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

Optimizing MSVC 2012

Listing 17.8: MSVC 2012

```
a$ = 8
f       PROC
; RCX = input value
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f:
; pass 1 ------------------------------------
```

```
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
; ---------------------------------------------
; pass 2 --------------------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax     ; rt++
$LN11@f:
        rol     rdx, 1  ; RDX=RDX<<1
; ---------------------------------------------
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Optimizing MSVC 2012 does almost the same job as optimizing MSVC 2010, but somehow, it generates two identical loop bodies and the loop count is now 32 instead of 64. To be honest, it's not possible to say why. Some optimization trick? Maybe it's better for the loop body to be slightly longer? Anyway, such code is relevant here to show that sometimes the compiler output may be really weird and illogical, but perfectly working.

17.5 Conclusion

Analogous to the C/C++ shifting operators $\ll$ and $\gg$, the shift instructions in x86 are SHR/SHL (for unsigned values) and SAR/SHL (for signed values).

The shift instructions in ARM are LSR/LSL (for unsigned values) and ASR/LSL (for signed values). It's also possible to add shift suffix to some instructions (which are called "data processing instructions").

17.5.1 Check for specific bit (known at compile stage)

Test if the 1000000 bit (0x40) is present in the register's value:

Listing 17.9: C/C++

```
if (input&0x40)
    ...
```

Listing 17.10: x86

```
TEST REG, 40h
JNZ is_set
; bit is not set
```

Listing 17.11: x86

```
TEST REG, 40h
JZ is_cleared
; bit is set
```

Sometimes, AND is used instead of TEST, but the flags that are set are the same.

17.5.2 Check for specific bit (specified at runtime)

This is usually done by this C/C++ code snippet (shift value by n bits right, then cut off lowest bit):

Listing 17.12: C/C++

```
if ((value>>n)&1)
    ....
```

This is usually implemented in x86 code as:

Listing 17.13: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Or (shift 1 bit n times left, isolate this bit in input value and check if it's not zero):

Listing 17.14: C/C++

```
if (value & (1<<n))
    ....
```

This is usually implemented in x86 code as:

Listing 17.15: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

17.5.3 Set specific bit (known at compile stage)

Listing 17.16: C/C++

```
value=value|0x40;
```

Listing 17.17: x86

```
OR REG, 40h
```

17.5.4 Set specific bit (specified at runtime)

Listing 17.18: C/C++

```
value=value|(1<<n);
```

This is usually implemented in x86 code as:

Listing 17.19: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

17.5.5 Clear specific bit (known at compile stage)

Just apply AND operation with the inverted value:

Listing 17.20: C/C++

```
value=value&(~0x40);
```

Listing 17.21: x86

```
AND REG, 0FFFFFFBFh
```

Listing 17.22: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

This is actually leaving all bits set except one.

17.5.6 Clear specific bit (specified at runtime)

Listing 17.23: C/C++

```
value=value&(~(1<<n));
```

Listing 17.24: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

Chapter 18

Linear congruential generator as pseudorandom number generator

The linear congruential generator is probably the simplest possible way to generate random numbers. It's not in favour in modern times[1], but it's so simple (just one multiplication, one addition and one AND operation), we can use it as an example.

```
#include <stdint.h>

// constants from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

[1]Mersenne twister is better

There are two functions: the first one is used to initialize the internal state, and the second one is called to generate pseudorandom numbers.

We see that two constants are used in the algorithm. They are taken from [Pre+07]. Let's define them using a `#define` C/C++ statement. It's a macro. The difference between a C/C++ macro and a constant is that all macros are replaced with their value by C/C++ preprocessor, and they don't take any memory, unlike variables. In contrast, a constant is a read-only variable. It's possible to take a pointer (or address) of a constant variable, but impossible to do so with a macro.

The last AND operation is needed because by C-standard `my_rand()` has to return a value in the 0..32767 range. If you want to get 32-bit pseudorandom values, just omit the last AND operation.

18.1 x86

Listing 18.1: Optimizing MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767              ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

Here we see it: both constants are embedded into the code. There is no memory allocated for them. The `my_srand()` function just copies its input value into the internal `rand_state` variable.

`my_rand()` takes it, calculates the next `rand_state`, cuts it and leaves it in the EAX register.

The non-optimized version is more verbose:

Listing 18.2: Non-optimizing MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223          ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767               ; 00007fffH
        pop     ebp
        ret     0
_rand   ENDP

_TEXT   ENDS
```

18.2 x64

The x64 version is mostly the same and uses 32-bit registers instead of 64-bit ones (because we are working with *int* values here). But `my_srand()` takes its input argument from the ECX register rather than from stack:

Listing 18.3: Optimizing MSVC 2013 x64

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS
```

```
init$ = 8
my_srand PROC
; ECX = input argument
        mov     DWORD PTR rand_state, ecx
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525      ; ↙
    ↘ 0019660dH
        add     eax, 1013904223                         ; 3↙
    ↘ c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767                              ; 00007↙
    ↘ fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Chapter 19

Structures

A C/C++ structure, with some assumptions, is just a set of variables, always stored in memory together, not necessary of the same type [1].

19.1 MSVC: SYSTEMTIME example

Let's take the SYSTEMTIME[2] win32 structure that describes time.

This is how it's defined:

Listing 19.1: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Let's write a C function to get the current time:

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

[1] AKA "heterogeneous container"

[2] MSDN: SYSTEMTIME structure

```

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;
};
```

We get (MSVC 2010):

Listing 19.2: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    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', 0↙
    ↳ aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

16 bytes are allocated for this structure in the local stack —that is exactly `sizeof(WORD)*8`
(there are 8 WORD variables in the structure).

Pay attention to the fact that the structure begins with the `wYear` field. It can be said that a pointer to the SYSTEMTIME structure is passed to the `GetSystemTime()`[3], but it is also can be said that a pointer to the `wYear` field is passed, and that is the same! `GetSystemTime()` writes the current year to the WORD pointer pointing to, then shifts 2 bytes ahead, writes current month, etc, etc.

19.1.1 Replacing the structure with array

The fact that the structure fields are just variables located side-by-side, can be easily demonstrated by doing the following. Keeping in mind the `SYSTEMTIME` structure description, it's possible to rewrite this simple example like this:

```
#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;
};
```

The compiler grumbles a bit:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible ↙
    ↳ types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

But nevertheless, it produces this code:

Listing 19.3: Non-optimizing MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16   ; size = 16
_main   PROC
```

[3]MSDN: SYSTEMTIME structure

```
        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
```

And it works just as the same!

It is very interesting that the result in assembly form cannot be distinguished from the result of the previous compilation. So by looking at this code, one cannot say for sure if there was a structure declared, or an array.

Nevertheless, no sane person would do it, as it is not convenient. Also the structure fields may be changed by developers, swapped, etc.

19.2 Let's allocate space for a structure using malloc()

Sometimes it is simpler to place structures not the in local stack, but in the heap:

```
#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;
};
```

Let's compile it now with optimization (/Ox) so it would be easy see what we need.

Listing 19.4: Optimizing 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
```

So, `sizeof(SYSTEMTIME) = 16` and that is exact number of bytes to be allocated by `malloc()`. It returns a pointer to a freshly allocated memory block in the EAX register, which is then moved into the `ESI` register. `GetSystemTime()` win32 function takes care of saving value in `ESI`, and that is why it is not saved here and continues to be used after the `GetSystemTime()` call.

New instruction —MOVZX (*Move with Zero eXtend*). It may be used in most cases as MOVSX, but it sets the remaining bits to 0. That's because `printf()` requires a 32-bit *int*, but we got a WORD in the structure —that is 16-bit unsigned type. That's why by copying the value from a WORD into *int*, bits from 16 to 31 must be cleared, because a random noise may be there, which is left from the previous operations on the register(s).

In this example, it's possible to represent the structure as an array of 8 WORDs:

```
#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;
};
```

We get:

Listing 19.5: Optimizing 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
```

Again, we got the code cannot be distinguished from the previous one. And again it should be noted, you haven't to do this in practice, unless you really know what you are doing.

19.3 Fields packing in structure

One important thing is fields packing in structures[4].

Let's take a simple example:

```
#include <stdio.h>

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

[4]See also: Wikipedia: Data structure alignment

```
void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

As we see, we have two *char* fields (each is exactly one byte) and two more —*int* (each — 4 bytes).

19.3.1 x86

This compiles to:

Listing 19.6: MSVC 2012 /GS- /Ob0

```
_tmp$ = -16
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    mov    BYTE PTR _tmp$[ebp], 1     ; set field a
    mov    DWORD PTR _tmp$[ebp+4], 2  ; set field b
    mov    BYTE PTR _tmp$[ebp+8], 3   ; set field c
    mov    DWORD PTR _tmp$[ebp+12], 4 ; set field d
    sub    esp, 16                    ; allocate place for
    temporary structure
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; copy our structure to
    the temporary one
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
    mov    ecx, DWORD PTR _tmp$[ebp+8]
    mov    DWORD PTR [eax+8], ecx
    mov    edx, DWORD PTR _tmp$[ebp+12]
    mov    DWORD PTR [eax+12], edx
    call    _f
    add    esp, 16
```

```
22      xor    eax, eax
23      mov    esp, ebp
24      pop    ebp
25      ret    0
26  _main    ENDP
27
28  _s$ = 8 ; size = 16
29  ?f@@YAXUs@@@Z PROC ; f
30      push   ebp
31      mov    ebp, esp
32      mov    eax, DWORD PTR _s$[ebp+12]
33      push   eax
34      movsx  ecx, BYTE PTR _s$[ebp+8]
35      push   ecx
36      mov    edx, DWORD PTR _s$[ebp+4]
37      push   edx
38      movsx  eax, BYTE PTR _s$[ebp]
39      push   eax
40      push   OFFSET $SG3842
41      call   _printf
42      add    esp, 20
43      pop    ebp
44      ret    0
45  ?f@@YAXUs@@@Z ENDP ; f
46  _TEXT    ENDS
```

We pass the structure as a whole, but in fact, as we can see, the structure is being copied to a temporary one (a place in stack is allocated in line 10 for it, and then all 4 fields, one by one, are copied in lines 12 … 19), and then its pointer (address) is to be passed. The structure is copied because it's not known whether the `f()` function going to modify the structure or not. If it gets changed, then the structure in `main()` has to remain as it was. We could use C/C++ pointers, and the resulting code will be almost the same, but without the copying.

As we can see, each field's address is aligned on a 4-byte boundary. That's why each *char* occupies 4 bytes here (like *int*). Why? Because it is easier for the CPU to access memory at aligned addresses and to cache data from it.

However, it is not very economical.

Let's try to compile it with option (`/Zp1`) (*/Zp[n] pack structures on n-byte boundary*).

Listing 19.7: MSVC 2012 /GS- /Zp1

```
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    mov    BYTE PTR _tmp$[ebp], 1     ; set field a
    mov    DWORD PTR _tmp$[ebp+1], 2  ; set field b
    mov    BYTE PTR _tmp$[ebp+5], 3   ; set field c
    mov    DWORD PTR _tmp$[ebp+6], 4  ; set field d
    sub    esp, 12                    ; allocate place for temporary structure
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; copy 10 bytes
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
    mov    cx, WORD PTR _tmp$[ebp+8]
    mov    WORD PTR [eax+8], cx
    call    _f
    add    esp, 12
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_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
```

Now the structure takes only 10 bytes and each *char* value takes 1 byte. What does it give to us? Size economy. And as drawback —the CPU accessing these fields slower than it could.

The structure is also copied in `main()`. Not field-by-field, but directly 10 bytes, using three pairs of MOV. Why not 4? The compiler decided that it's better to copy 10 bytes using 3 MOV pairs than to copy two 32-bit words and two bytes using 4

MOV pairs.

As it can be easily guessed, if the structure is used in many source and object files, all these must be compiled with the same convention about structures packing.

Aside from MSVC /Zp option which sets how to align each structure field, there is also the #pragma pack compiler option, which can be defined right in the source code. It is available in both MSVC[5]and GCC[6].

Let's get back to the SYSTEMTIME structure that consists of 16-bit fields. How does our compiler know to pack them on 1-byte alignment boundary?

WinNT.h file has this:

Listing 19.8: WinNT.h

```
#include "pshpack1.h"
```

And this:

Listing 19.9: WinNT.h

```
#include "pshpack4.h"                     // 4 byte packing is ↙
    ↳ the default
```

The file PshPack1.h looks like:

Listing 19.10: 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)) */
```

This tell the compiler how to pack the structures defined after #pragma pack.

[5] MSDN: Working with Packing Structures
[6] Structure-Packing Pragmas

19.3.2 One more word

Passing a structure as a function argument (instead of a passing pointer to structure) is the same as passing all structure fields one by one. If the structure fields are packed by default, the f() function can be rewritten as:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

And that leads to the same code.

19.4 Nested structures

Now what about situations when one structure is defined inside of another?

```
#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);
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

... in this case, both `inner_struct` fields are to be placed between the a,b and d,e fields of the `outer_struct`.

Let's compile (MSVC 2010):

Listing 19.11: Optimizing MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00↙
    ↘ H

_TEXT    SEGMENT
_s$ = 8
_f    PROC
    mov    eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; ↙
    ↘ e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f    ENDP

_s$ = -24
_main    PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
    mov    BYTE PTR _s$[esp+60], 1
    mov    ebx, DWORD PTR _s$[esp+60]
    mov    DWORD PTR [eax], ebx
    mov    DWORD PTR [eax+4], ecx
    lea    edx, DWORD PTR [ecx+98]
    lea    esi, DWORD PTR [ecx+99]
    lea    edi, DWORD PTR [ecx+2]
    mov    DWORD PTR [eax+8], edx
    mov    BYTE PTR _s$[esp+76], 3
    mov    ecx, DWORD PTR _s$[esp+76]
    mov    DWORD PTR [eax+12], esi
    mov    DWORD PTR [eax+16], ecx
    mov    DWORD PTR [eax+20], edi
    call   _f
    add    esp, 24
    pop    edi
    pop    esi
    xor    eax, eax
    pop    ebx
    add    esp, 24
    ret    0
_main    ENDP
```

One curious thing here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are unfolded into *linear* or *one-dimensional* structure.

Of course, if we replace the `struct inner_struct c;` declaration with `struct inner_struct *c;` (thus making a pointer here) the situation will be quite different.

19.5 Bit fields in a structure

19.5.1 CPUID example

The C/C++ language allows to define the exact number of bits for each structure field. It is very useful if one needs to save memory space. For example, one bit is enough for a *bool* variable. But of course, it is not rational if speed is important.

Let's consider the CPUID[7] instruction example. This instruction returns information about the current CPU and its features.

If the EAX is set to 1 before the instruction's execution, CPUID returning this information packed into the EAX register:

[7] wikipedia

3:0 (4 bits)	Stepping
7:4 (4 bits)	Model
11:8 (4 bits)	Family
13:12 (2 bits)	Processor Type
19:16 (4 bits)	Extended Model
27:20 (8 bits)	Extended Family

MSVC 2010 has CPUID macro, but GCC 4.4.1 does not. So let's make this function by ourselves for GCC with the help of its built-in assembler[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));
}
#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__
```

[8] More about internal GCC assembler

```
    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;
};
```

After CPUID fills EAX/EBX/ECX/EDX, these registers are to be written in the b[] array. Then, we have a pointer to the CPUID_1_EAX structure and we point it to the value in EAX from the b[] array.

In other words, we treat a 32-bit *int* value as a structure. Then we read specific bits from the structure.

MSVC

Let's compile it in MSVC 2008 with /Ox option:

Listing 19.12: Optimizing 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
```

The SHR instruction shifting the value in EAX by the number of bits that must be *skipped*, e.g., we ignore some bits *at the right side*.

The AND instruction clears the unneeded bits *on the left*, or, in other words, leaves only those bits in the EAX register we need.

Chapter 20

64-bit values in 32-bit environment

20.1 Returning of 64-bit value

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

20.1.1 x86

In a 32-bit environment, 64-bit values are returned from functions in the EDX:EAX register pair.

Listing 20.1: Optimizing MSVC 2010

```
_f      PROC
        mov     eax, -1867788817        ; 90abcdefH
        mov     edx, 305419896          ; 12345678H
        ret     0
_f      ENDP
```

20.2 Arguments passing, addition, subtraction

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345)↙
    ↳ );
#else
        printf ("%I64d\n", f_add(12345678901234, ↙
    ↳ 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

20.2.1 x86

Listing 20.2: Optimizing MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  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
_f_add  ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff_subH
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  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
_f_sub  ENDP
```

We can see in the `f_add_test()` function that each 64-bit value is passed using two 32-bit values, high part first, then low part.

Addition and subtraction occur in pairs as well.

In addition, the low 32-bit part are added first. If carry was occurred while adding, the CF flag is set. The following ADC instruction adds the high parts of the values, and also adds 1 if $CF = 1$.

Subtraction also occurs in pairs. The first SUB may also turn on the CF flag, which is to be checked in the subsequent SBB instruction: if the carry flag is on, then 1 is also to be subtracted from the result.

It is easy to see how the `f_add()` function result is then passed to `printf()`.

20.3 Multiplication, division

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};
```

```

uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
};
```

20.3.1 x86

Listing 20.3: Optimizing MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; long long multiplication
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; unsigned long long division
        pop     ebp
        ret     0
_f_div  ENDP

```

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; unsigned long long remainder
        pop     ebp
        ret     0
_f_rem  ENDP
```

Multiplication and division are more complex operations, so usually the compiler embeds calls to a library functions doing that.

20.4 Shifting right

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

20.4.1 x86

Listing 20.4: Optimizing MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

Shifting also occurs in two passes: first the lower part is shifted, then the higher part. But the lower part is shifted with the help of the SHRD instruction, it shifts the value of EDX by 7 bits, but pulls new bits from EAX, i.e., from the higher part. The higher part is shifted using the more popular SHR instruction: indeed, the freed bits in the higher part must be filled with zeroes.

20.5 Converting 32-bit value into 64-bit one

```
#include <stdint.h>

int64_t f (int32_t a)
{
        return a;
};
```

20.5.1 x86

Listing 20.5: Optimizing MSVC 2012

```
_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP
```

Here we also run into necessity to extend a 32-bit signed value into a 64-bit signed one. Unsigned values are converted straightforwardly: all bits in the higher part must be set to 0. But this is not appropriate for signed data types: the sign has to be copied into the higher part of the resulting number. The CDQ instruction does that here, it takes its input value in EAX, extends it to 64-bit and leaves it in the EDX:EAX register pair. In other words, CDQ gets the number sign from EAX (by getting the most significant bit in EAX), and depending of it, sets all 32 bits in EDX to 0 or 1. Its operation is somewhat similar to the MOVSX instruction.

Chapter 21

64 bits

21.1 x86-64

It is a 64-bit extension to the x86 architecture.

From the reverse engineer's perspective, the most important changes are:

- Almost all registers (except FPU and SIMD) were extended to 64 bits and got a R- prefix. 8 additional registers wer added. Now GPR's are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

  It is still possible to access the *older* register parts as usual. For example, it is possible to access the lower 32-bit part of the RAX register using EAX:

7th $^{\text{(byte number)}}$	6th	5th	4th	3rd	2nd	1st	0th
RAX^{x64}							
				EAX			
						AX	
						AH	AL

  The new R8-R15 registers also have their *lower parts*: R8D-R15D (lower 32-bit parts), R8W-R15W (lower 16-bit parts), R8L-R15L (lower 8-bit parts).

7th $^{\text{(byte number)}}$	6th	5th	4th	3rd	2nd	1st	0th
R8							
				R8D			
						R8W	
							R8L

  The number of SIMD registers was doubled from 8 to 16: XMM0-XMM15.

- In Win64, the function calling convention is slightly different, somewhat resembling fastcall . The first 4 arguments are stored in the RCX, RDX, R8, R9 registers, the rest —in the stack. The caller function must also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for its own needs. Short functions may use arguments just from registers, but larger ones may save their values on the stack.

  System V AMD64 ABI (Linux, *BSD, Mac OS X)[Mit13] also somewhat resembles fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed via the stack.

- The C/C++ *int* type is still 32-bit for compatibility.

- All pointers are 64-bit now.

  This provokes irritation sometimes: now one needs twice as much memory for storing pointers, including cache memory, despite the fact that x64 CPUs can address only 48 bits of external RAM[1].

Since now the number of registers is doubled, the compilers have more space for maneuvering called register allocation. For us this implies that the emitted code containing less number of local variables.

By the way, there are CPUs with much more GPR's, e.g. Itanium (128 registers).

[1]Random-access memory

Part II

Important fundamentals

Chapter 22

Signed number representations

There are several methods for representing signed numbers[1], but “two’s complement” is the most popular one in computers.

Here is a table for some byte values:

binary	hexadecimal	unsigned	signed (2’s complement)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000110	0x6	6	6
00000101	0x5	5	5
00000100	0x4	4	4
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
11111100	0xfc	252	-4
11111011	0xfb	251	-5
11111010	0xfa	250	-6
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

[1] wikipedia

CHAPTER 22. SIGNED NUMBER REPRESENTATIONS

The difference between signed and unsigned numbers is that if we represent `0xFFFFFFFE` and `0x0000002` as unsigned, then the first number (4294967294) is bigger than the second one (2). If we represent them both as signed, the first one is to be -2, and it is smaller than the second (2). That is the reason why conditional jumps (11 on page 50) are present both for signed (e.g. `JG`, `JL`) and unsigned (`JA`, `JB`) operations.

For the sake of simplicity, that is what one need to know:

- Numbers can be signed or unsigned.
- C/C++ signed types:
  - `int64_t` (-9,223,372,036,854,775,808..9,223,372,036,854,775,807) (- 9.2.. 9.2 quintillions) or `0x8000000000000000..0x7FFFFFFFFFFFFFFF`),
  - *int* (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) or `0x80000000..0x7FFFF`
  - *char* (-128..127 or `0x80..0x7F`),
  - `ssize_t`.

  Unsigned:
  - `uint64_t` (0..18,446,744,073,709,551,615 (18 quintillions) or `0..0xFFFFFFF`
  - `unsigned int` (0..4,294,967,295 (4.3Gb) or `0..0xFFFFFFFF`),
  - `unsigned char` (0..255 or `0..0xFF`),
  - `size_t`.
- Signed types have the sign in the most significant bit: 1 mean "minus", 0 mean "plus".
- Promoting to a larger data types is simple: 20.5 on page 152.
- Negation is simple: just invert all bits and add 1. We can remember that a number of inverse sign is located on the opposite side at the same proximity from zero. The addition of one is needed because zero is present in the middle.
- The addition and subtraction operations work well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: `IDIV`/`IMUL` for signed and `DIV`/`MUL` for unsigned.

Chapter 23

Memory

There are 3 main types of memory:

- Global memory. AKA "static memory allocation". No need to allocate explicitly, the allocation is done just by declaring variables/arrays globally. These are global variables, residing in the data or constant segments. The are available globally (hence, considered as an anti-pattern). Not convenient for buffers/arrays, because they must have a fixed size. Buffer overflows that occur here usually overwrite variables or buffers reside next to them in memory. There's an example in this book: 7.2 on page 30.
- Stack. AKA "allocate on stack". The allocation is done just by declaring variables/arrays locally in the function. These are usually local variables for the function. Sometimes these local variable are also available to descending functions (to callee functions, if caller passes a pointer to a variable to the callee to be executed). Allocation and deallocation are very fast, it's just SP needs to be shifted. But they're also not convenient for buffers/arrays, because the buffer size has to be fixed, unless `alloca()` (5.2.4 on page 17) (or a variable-length array) is used. Buffer overflows usually overwrite important stack structures: 16.2 on page 95.
- Heap. AKA "dynamic memory allocation". Allocation is done by calling `malloc()/free()` or `new/delete` in C++. This is the most convenient method: the block size may be set at runtime. Resizing is possible (using `realloc()`), but can be slow. This is the slowest way to allocate memory: the memory allocator must support and update all control structures while allocating and deallocating. Buffer overflows usually overwrite these structures. Heap allocations are also source of memory leak problems: each memory block has to be deallocated explicitly, but one may forget about it, or do it incorrectly. Another problem is the "use after free"—using a memory block after `free()`

was called on it, which is very dangerous. Example in this book: 19.2 on page 132.

Part III

Finding important/interesting stuff in the code

Minimalism it is not a prominent feature of modern software.

But not because the programmers are writing a lot, but because a lot of libraries are commonly linked statically to executable files. If all external libraries were shifted into an external DLL files, the world would be different. (Another reason for C++ are the STL and other template libraries.)

Thus, it is very important to determine the origin of a function, if it is from standard library or well-known library (like Boost[1], libpng[2]), or if it is related to what we are trying to find in the code.

It is just absurd to rewrite all code in C/C++ to find what we're looking for.

One of the primary tasks of a reverse engineer is to find quickly the code he/she needs.

The IDA disassembler allow us to search among text strings, byte sequences and constants. It is even possible to export the code to .lst or .asm text files and then use `grep`, awk, etc.

When you try to understand what some code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings which look familiar, it is always worth to *google* them. And if you find the open-source project where they are used, then it's enough just to compare the functions. It may solve some part of the problem.

For example, if a program uses XML files, the first step may be determining which XML library is used for processing, since the standard (or well-known) libraries are usually used instead of self-made one.

For example, author of these lines once tried to understand how the compression/decompression of network packets worked in SAP 6.0. It is a huge software, but a detailed .PDB with debugging information is present, and that is convenient. He finally came to the idea that one of the functions, that was called CsDecomprLZC, was doing the decompression of network packets. Immediately he tried to google its name and he quickly found the function was used in MaxDB (it is an open-source SAP project) .

`http://www.google.com/search?q=CsDecomprLZC`

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for the compression/decompression of network packets.

[1] `http://go.yurichev.com/17036`
[2] `http://go.yurichev.com/17037`

Chapter 24

Communication with the outer world (win32)

Sometimes it's enough to observe some function's inputs and outputs in order to understand what it does. That way you can save time.

Files and registry access: for the very basic analysis, Process Monitor[1] utility from SysInternals can help.

For the basic analysis of network accesses, Wireshark[2] can be useful.

But then you will have to to look inside anyway.

The first thing to look for is which functions from the OS's API[3]s and standard libraries are used.

If the program is divided into a main executable file and a group of DLL files, sometimes the names of the functions in these DLLs can help.

If we are interested in exactly what can lead to a call to `MessageBox()` with specific text, we can try to find this text in the data segment, find the references to it and find the points from which the control may be passed to the `MessageBox()` call we're interested in.

If we are talking about a video game and we're interested in which events are more or less random in it, we may try to find the `rand()` function or its replacements (like the Mersenne twister algorithm) and find the places from which those functions are called, and more importantly, how are the results used.

[1] http://go.yurichev.com/17301
[2] http://go.yurichev.com/17303
[3] Application programming interface

But if it is not a game, and `rand()` is still used, it is also interesting to know why. There are cases of unexpected `rand()` usage in data compression algorithms (for encryption imitation): blog.yurichev.com.

24.1 Often used functions in the Windows API

These functions may be among the imported. It is worth to note that not every function might be used in the code that was written by the programmer. A lot of functions might be called from library functions and CRT code.

- Registry access (advapi32.dll): RegEnumKeyEx[4] [5], RegEnumValue[6] [5], RegGetValue[7] [5], RegOpenKeyEx[8] [5], RegQueryValueEx[9] [5].
- Access to text .ini-files (kernel32.dll): GetPrivateProfileString [10] [5].
- Dialog boxes (user32.dll): MessageBox [11] [5], MessageBoxEx [12] [5], SetDlgItemText [13] [5], GetDlgItemText [14] [5].
- Resources access : (user32.dll): LoadMenu [15] [5].
- TCP/IP networking (ws2_32.dll): WSARecv [16], WSASend [17].
- File access (kernel32.dll): CreateFile [18] [5], ReadFile [19], ReadFileEx [20], WriteFile [21], WriteFileEx [22].
- High-level access to the Internet (wininet.dll): WinHttpOpen [23].

[4]MSDN
[5]May have the -A suffix for the ASCII version and -W for the Unicode version
[6]MSDN
[7]MSDN
[8]MSDN
[9]MSDN
[10]MSDN
[11]MSDN
[12]MSDN
[13]MSDN
[14]MSDN
[15]MSDN
[16]MSDN
[17]MSDN
[18]MSDN
[19]MSDN
[20]MSDN
[21]MSDN
[22]MSDN
[23]MSDN

- Checking the digital signature of an executable file (wintrust.dll): WinVerifyTrust [24].
- The standard MSVC library (if it's linked dynamically) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

24.2 tracer: Intercepting all functions in specific module

There are INT3 breakpoints in the tracer, that are triggered only once, however, they can be set for all functions in a specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let's set INT3 breakpoints on all functions with the `xml` prefix in their name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of the coin, such breakpoints are triggered only once.

Tracer will show the call of a function, if it happens, but only once. Another drawback—it is impossible to see the function's arguments.

Nevertheless, this feature is very useful when you know that the program uses a DLL, but you do not know which functions are actually used. And there are a lot of functions.

For example, let's see, what does the uptime utility from cygwin use:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all that cygwin1.dll library functions that were called at least once, and where from:

```
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))
```

[24] MSDN

```
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))
```

Chapter 25

Strings

25.1 Text strings

25.1.1 C/C++

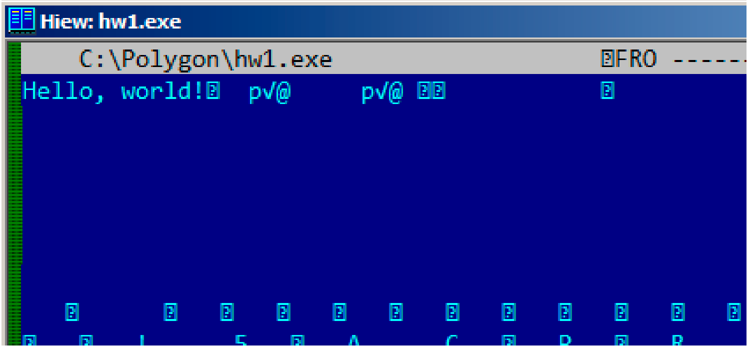
The normal C strings are zero-terminated (ASCIIZ-strings).

The reason why the C string format is as it is (zero-terminated) is apparently historical. In [Rit79] we read:

> 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.

In Hiew or FAR Manager these strings looks like this:

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


Hiew: hw1.exe

```
    C:\Polygon\hw1.exe                          FRO -----
Hello, world!   pv@       pv@
```

Figure 25.1: Hiew

25.1.2 Borland Delphi

The string in Pascal and Borland Delphi is preceded by an 8-bit or 32-bit string length.

For example:

Listing 25.1: 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
```

25.1.3 Unicode

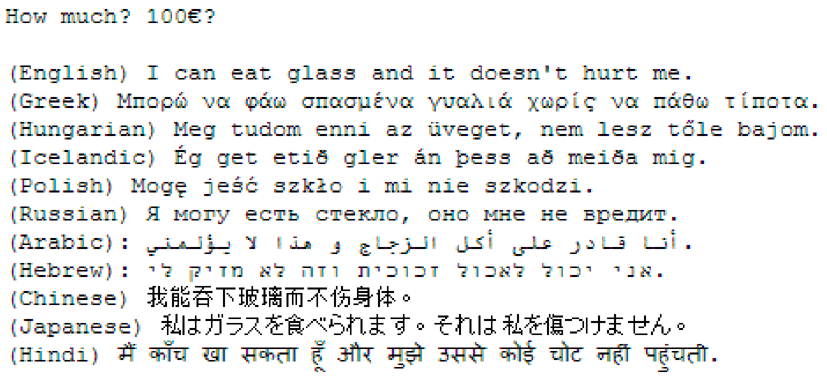
Often, what is called Unicode is a methods for encoding strings where each character occupies 2 bytes or 16 bits. This is a common terminological mistake. Unicode is a standard for assigning a number to each character in the many writing systems of the world, but does not describe the encoding method.

The most popular encoding methods are: UTF-8 (is widespread in Internet and *NIX systems) and UTF-16LE (is used in Windows).

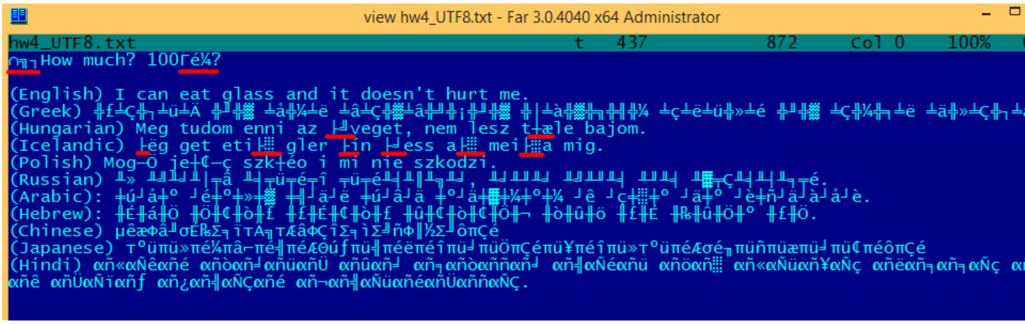
UTF-8

UTF-8 is one of the most successful methods for encoding characters. All Latin symbols are encoded just like in ASCII, and the symbols beyond the ASCII table are encoded using several bytes. 0 is encoded as before, so all standard C string functions work with UTF-8 strings just like any other string.

Let's see how the symbols in various languages are encoded in UTF-8 and how it looks like in FAR, using the 437 codepage [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.
```



view hw4_UTF8.txt - Far 3.0.4040 x64 Administrator

```
hw4_UTF8.txt                                        t    437          872      Col 0     100%
∩╗┐How much? 100Γé¼?

(English) I can eat glass and it doesn't hurt me.
(Greek) ╬£╬┐╧ü╬┐╧Ä ╬╜╬▒ ╧å╬¼╧ë ╧â╧Ç╬▒╧â╬╝╬¡╬╜╬▒ ╬│╧à╬▒╬╗╬╣╬¼ ╧ç╧ë╧ü╬»╧é ╬╜╬▒ ╧Ç╬¼╬╕╧ë ╧ä╬»╧Ç╬┐╧ä╬▒
(Hungarian) Meg tudom enni az ├╝veget, nem lesz t┼æle bajom.
(Icelandic) ├ëg get eti├░ gler ├ín ├╛ess a├░ mei├░a mig.
(Polish) Mog─Ö je┼¢─ç szk┼éo i mi nie szkodzi.
(Russian) ╨» ╨╝╨╛╨│╤â ╨╡╤ü╤é╤î ╤ü╤é╨╡╨║╨╗╨╛, ╨╛╨╜╨╛ ╨╝╨╜╨╡ ╨╜╨╡ ╨▓╤Ç╨╡╨┤╨╕╤é.
(Arabic): ╪ú┘å╪º ┘é╪º╪»╪▒ ╪╣┘ä┘ë ╪ú┘â┘ä ╪º┘ä╪▓╪¼╪º╪¼ ┘ê ┘ç╪░╪º ┘ä╪º ┘è╪ñ┘ä┘à┘å┘è.
(Hebrew): ╫É╫á╫Ö ╫Ö╫¢╫ò╫£ ╫£╫É╫¢╫ò╫£ ╫û╫¢╫ò╫¢╫Ö╫¬ ╫ò╫û╫ö ╫£╫É ╫₧╫û╫Ö╫º ╫£╫Ö.
(Chinese) µêæΦâ╜σÉ₧Σ╕ïτÄ╗τÆâΦÇîΣ╕ìΣ╝ñΦ║½Σ╜ôπÇé
(Japanese) τº üπü»πé¼πâ⌐πé╣πéÆΘúƒπü╣πéëπéîπü╛πüÖπÇéπü¥πéîπü»τºüπéÆσé╖πüñπüæπü╛πü¢πéôπÇé
(Hindi) αñ«αÑêαñé αñòαñ╛αñüαñÜ αñûαñ╛ αñ╕αñòαññαñ╛ αñ╣αÑéαñü αñöαñ░ αñ«αÑüαñ¥αÑç αñëαñ╕αñ╕αÑç αñ
αñê αñÜαÑïαñƒ αñ¿αñ╣αÑÇαñé αñ¬αñ╣αÑüαñéαñÜαññαÑÇ.
```

Figure 25.2: FAR: UTF-8

As you can see, the English language string looks the same as it is in ASCII. The Hungarian language uses some Latin symbols plus symbols with diacritic marks. These symbols are encoded using several bytes, these are underscored with red. It's the same story with the Icelandic and Polish languages. There is also the "Euro"

[1] The example and translations was taken from here: http://go.yurichev.com/17304

currency symbol at the start, which is encoded with 3 bytes. The rest of the writing systems here have no connection with Latin. At least in Russian, Arabic, Hebrew and Hindi we can see some recurring bytes, and that is not surprise: all symbols from a writing system are usually located in the same Unicode table, so their code begins with the same numbers.

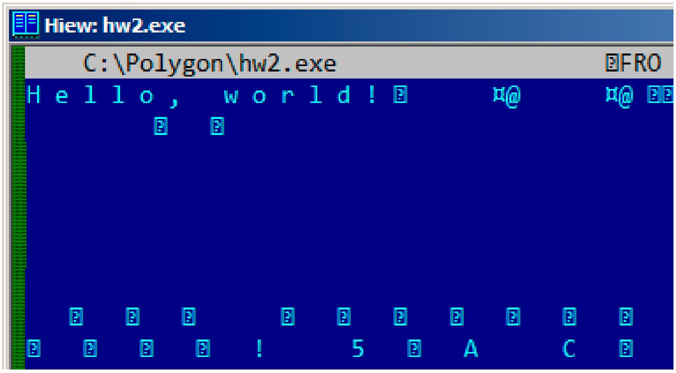
At the beginning, before the “How much?” string we see 3 bytes, which are in fact the BOM[2]. The BOM defines the encoding system to be used.

UTF-16LE

Many win32 functions in Windows have the suffixes -A and -W. The first type of functions works with normal strings, the other with UTF-16LE strings (*wide*). In the second case, each symbol is usually stored in a 16-bit value of type *short*.

The Latin symbols in UTF-16 strings look in Hiew or FAR like they are interleaved with zero byte:

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

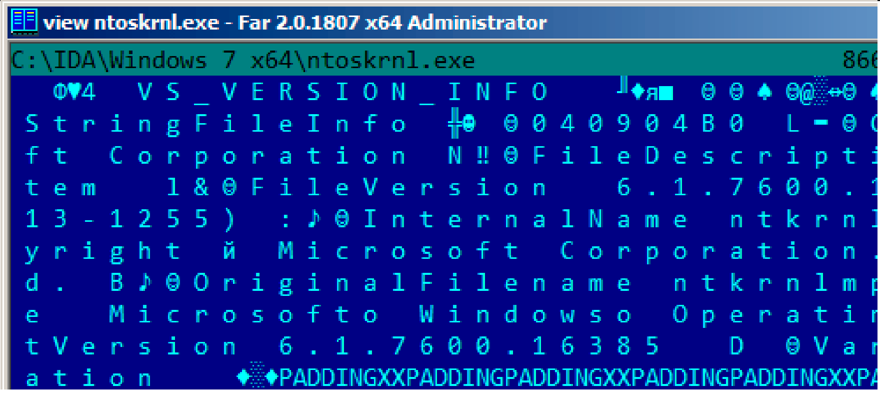


```
    C:\Polygon\hw2.exe                        FRO
H e l l o ,   w o r l d !
```

Figure 25.3: Hiew

We can see this often in Windows NT system files:

[2] Byte order mark



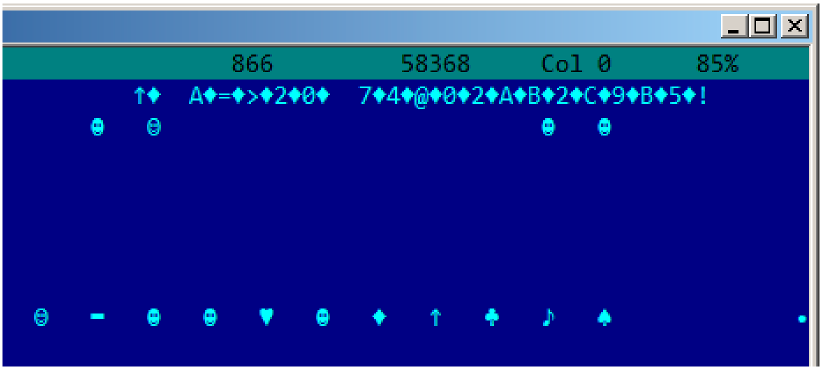
```
view ntoskrnl.exe - Far 2.0.1807 x64 Administrator
C:\IDA\Windows 7 x64\ntoskrnl.exe                                    866
   4   V S _ V E R S I O N _ I N F O      я
 S t r i n g F i l e I n f o          0 4 0 9 0 4 B 0   L
 f t   C o r p o r a t i o n   N ‼   F i l e D e s c r i p t
 t e m     1 &   F i l e V e r s i o n     6 . 1 . 7 6 0 0 .
 1 3 - 1 2 5 5 )   : ♪   I n t e r n a l N a m e   n t k r n
 y r i g h t   й   M i c r o s o f t   C o r p o r a t i o n
 d .   B ♪   O r i g i n a l F i l e n a m e   n t k r n l m
 e   M i c r o s o f t o   W i n d o w s o   O p e r a t i
 t V e r s i o n   6 . 1 . 7 6 0 0 . 1 6 3 8 5     D   V a
 a t i o n         PADDINGXXPADDINGPADDINGXXPADDINGPADDINGXXPA
```

Figure 25.4: Hiew

Strings with characters that occupy exactly 2 bytes are called "Unicode" in IDA:

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Here is how the Russian language string is encoded in UTF-16LE:



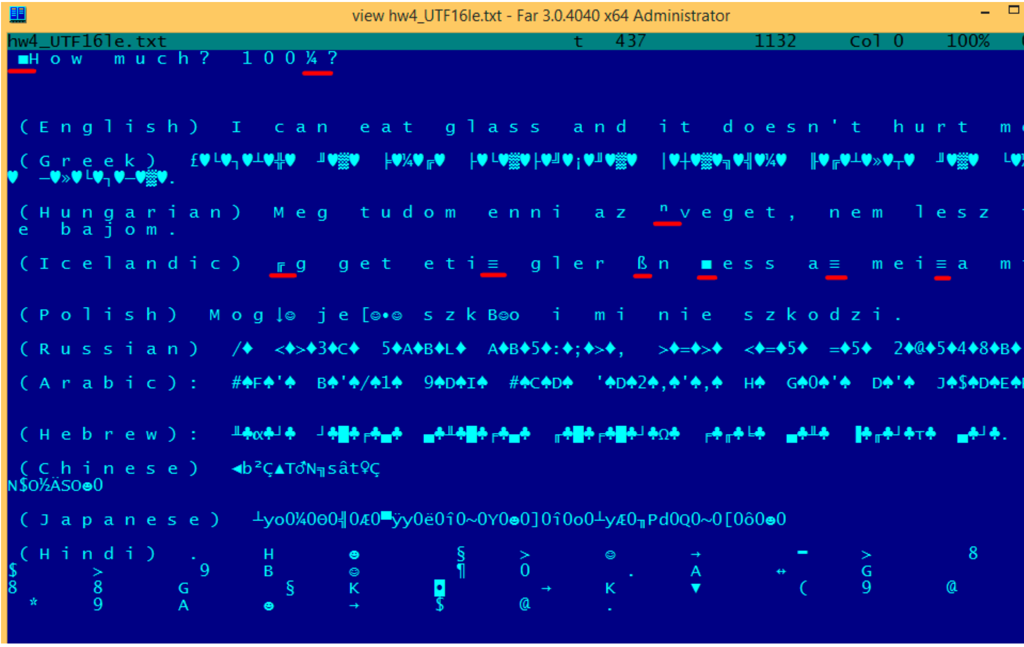
```
                866          58368        Col 0        85%
        ↑♦  A♦=♦>♦2♦0♦  7♦4♦@♦0♦2♦A♦B♦2♦C♦9♦B♦5♦!
```

Figure 25.5: Hiew: UTF-16LE

What we can easily spot is that the symbols are interleaved by the diamond character (which has the ASCII code of 4). Indeed, the Cyrillic symbols are located in the fourth Unicode plane [3]. Hence, all Cyrillic symbols in UTF-16LE are located in the 0x400-0x4FF range.

[3]wikipedia

Let's go back to the example with the string written in multiple languages. Here is how it looks like in UTF-16LE.



```
view hw4_UTF16le.txt - Far 3.0.4040 x64 Administrator
hw4_UTF16le.txt                          t    437         1132      Col 0     100%
 ■H o w   m u c h ?   1 0 0 ¼ ?


 ( E n g l i s h )   I   c a n   e a t   g l a s s   a n d   i t   d o e s n ' t   h u r t   m
 ( G r e e k )   £♥└♥┐♥┴♥╬♥  ╝♥▒♥  ├♥¼♥╔♥  ├♥└♥▒♥├♥╝♥¡♥╝♥▒♥  │♥┼♥▒♥┐♥╣♥¼♥  ║♥╔♥┴♥»♥┬♥  ╝♥▒♥  └♥
♥  ─♥»♥└♥┐♥─♥▒♥.
 ( H u n g a r i a n )   M e g   t u d o m   e n n i   a z   ⁿ v e g e t ,   n e m   l e s z
 e   b a j o m .
 ( I c e l a n d i c )   ╔ g   g e t   e t i ≡   g l e r   ß n   ■ e s s   a ≡   m e i ≡ a   m

 ( P o l i s h )   M o g ↓☺  j e [☺•☺  s z k B☺o   i   m i   n i e   s z k o d z i .
 ( R u s s i a n )   /♦  <♦>♦3♦C♦  5♦A♦B♦L♦  A♦B♦5♦:♦;♦>♦,   >♦=♦>♦  <♦=♦5♦  =♦5♦  2♦@♦5♦4♦8♦B♦
 ( A r a b i c ) :   #♠F♠'♠  B♠'♠/♠1♠  9♠D♠I♠  #♠C♠D♠  '♠D♠2♠,♠'♠,♠  H♠  G♠0♠'♠  D♠'♠  J♠$♠D♠E♠

 ( H e b r e w ) :   ╨♣α♣╜♣  ┘♣█♣╒♣▄♣  ▄♣╨♣█♣╒♣▄♣  ╓♣█♣╒♣█♣╜♣Ω♣  ╒♣╓♣╘♣  ▄♣╨♣  ▐♣╓♣╜♣╥♣  ▄♣╜♣.
 ( C h i n e s e )   ◄b²Ç▲T♂N╗sât♀Ç
N$O½ÄSO☻0
 ( J a p a n e s e )   ┴yo0¼0Θ0╡0Æ0▀ÿy0ë0î0~0Y0☻0]0î0o0┴yÆ0╗Pd0Q0~0[0ô0☻0
 ( H i n d i )   .     H     ☻      §    >     ☺     →       ▬     >      8
$       >       9     B     ☺      ¶    0        .  A     ↔       G
8       8     G        §    K     ◘            →     K     ▼     (    9      @
  *     9     A     ☻      →     $      @       .
```

Figure 25.6: FAR: UTF-16LE

Here we can also see the BOM in the beginning. All Latin characters are interleaved with a zero byte. Some characters with diacritic marks (Hungarian and Icelandic languages) are also underscored in red.

25.1.4 Base64

The base64 encoding is highly popular for the cases when you need to transfer binary data as a text string. In essence, this algorithm encodes 3 binary bytes into 4 printable characters: all 26 Latin letters (both lower and upper case), digits, plus sign ("+") and slash sign ("/"), 64 characters in total.

One distinctive feature of base64 strings is that they often (but not always) ends with 1 or 2 padding equality symbol(s) ("="), for example:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+⤦
    ↳ qEJAp9lAOuWs=
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+↙
    ↳ qEJAp9lA0uQ==
```

The equality sign ("=") is never encounter in the middle of base64-encoded strings.

25.2 Error/debug messages

Debugging messages are very helpful if present. In some sense, the debugging messages are reporting what's going on in the program right now. Often these are `printf()`-like functions, which write to log-files, or sometimes do not writing anything but the calls are still present since the build is not a debug one but *release* one. If local or global variables are dumped in debug messages, it might be helpful as well since it is possible to get at least the variable names. For example, one of such function in Oracle RDBMS is `ksdwrt()`.

Meaningful text strings are often helpful. The IDA disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen[4].

The error messages may help us as well. In Oracle RDBMS, errors are reported using a group of functions.
You can read more about them here: blog.yurichev.com.

It is possible to find quickly which functions report errors and in which conditions. By the way, this is often the reason for copy-protection systems to inarticulate cryptic error messages or just error numbers. No one is happy when the software cracker quickly understand why the copy-protection is triggered just by the error message.

25.3 Suspicious magic strings

Some magic strings which are usually used in backdoors looks pretty suspicious. For example, there was a backdoor in the TP-Link WR740 home router[5]. The backdoor was activated using the following URL:
`http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html`.
Indeed, the "userRpmNatDebugRpm26525557" string is present in the firmware. This string was not googleable until the wide disclosure of information about the backdoor. You would not find this in any RFC[6]. You would not find any computer

[4] blog.yurichev.com
[5] `http://sekurak.pl/tp-link-httptftp-backdoor/`
[6] Request for Comments

science algorithm which uses such strange byte sequences. And it doesn't look like an error or debugging message. So it's a good idea to inspect the usage of such weird strings.

Sometimes, such strings are encoded using base64. So it's a good idea to decode them all and to scan them visually, even a glance should be enough.

More precise, this method of hiding backdoors is called "security through obscurity".

Chapter 26

Calls to assert()

Sometimes the presence of the `assert()` macro is useful too: commonly this macro leaves source file name, line number and condition in the code.

The most useful information is contained in the assert's condition, we can deduce variable names or structure field names from it. Another useful piece of information are the file names—we can try to deduce what type of code is there. Also it is possible to recognize well-known open-source libraries by the file names.

Listing 26.1: Example of informative assert() calls

```
.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
```

It is advisable to "google" both the conditions and file names, which can lead us to an open-source library. For example, if we "google" "sp->lzw_nbits <= BITS_MAX", this predictably gives us some open-source code that's related to the LZW compression.

Chapter 27

Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, in real life as well as in the code.

The practicing reverse engineer usually know them well in hexadecimal representation: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

The constants 0xAAAAAAAA (10101010101010101010101010101010) and 0x55555555 (01010101010101010101010101010101) are also popular—those are composed of alternating bits. That may help to distinguish some signal from the signal where all bits are turned on (1111 ...) or off (0000 ...). For example, the 0x55AA constant is used at least in the boot sector, MBR[1], and in the ROM[2] of IBM-compatible extension cards.

Some algorithms, especially cryptographical ones use distinct constants, which are easy to find in code using IDA.

For example, the MD5[3] algorithm initializes its own internal variables like this:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants used in the code in a row, it is very highly probable that this function is related to MD5.

Another example are the CRC16/CRC32 algorithms, whose calculation algorithms often use precomputed tables like this one:

[1] Master Boot Record
[2] Read-only memory
[3] wikipedia

Listing 27.1: 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,
        ...
```

27.1 Magic numbers

A lot of file formats define a standard file header where a *magic number(s)*[4] is used, single one or even several.

For example, all Win32 and MS-DOS executables start with the two characters "MZ"[5].

At the beginning of a MIDI file the "MThd" signature must be present. If we have a program which uses MIDI files for something, it's very likely that it must check the file for validity by checking at least the first 4 bytes.

This could be done like this:

(*buf* points to the beginning of the loaded file in memory)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling a function for comparing memory blocks like `memcmp()` or any other equivalent code up to a CMPSB instruction.

When you find such point you already can say where the loading of the MIDI file starts, also, we could see the location of the buffer with the contents of the MIDI file, what is used from the buffer, and how.

27.1.1 DHCP

This applies to network protocols as well. For example, the DHCP protocol's network packets contains the so-called *magic cookie*: `0x63538263`. Any code that

[4]wikipedia
[5]wikipedia

generates DHCP packets somewhere must embed this constant into the packet. If we find it in the code we may find where this happens and, not only that. Any program which can receive DHCP packet must verify the *magic cookie*, comparing it with the constant.

For example, let's take the dhcpcore.dll file from Windows 7 x64 and search for the constant. And we can find it, twice: it seems that the constant is used in two functions with descriptive names like `DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptions()`:

Listing 27.2: 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
```

And here are the places where these constants are accessed:

Listing 27.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

And:

Listing 27.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

27.2 Searching for constants

It is easy in IDA: Alt-B or Alt-I. And for searching for a constant in a big pile of files, or for searching in non-executable files, there is a small utility called *binary grep*[6].

[6] GitHub

Chapter 28

Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in the code, one can try to check each one manually with a debugger.

For example, we may be interested how Microsoft Excel calculates the formulae entered by user. For example, the division operation.

If we load excel.exe (from Office 2010) version 14.0.4756.1000 into IDA, make a full listing and to find every FDIV instruction (except the ones which use constants as a second operand—obviously, they do not suit us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we see that there are 144 of them.

We can enter a string like =(1/3) in Excel and check each instruction.

By checking each instruction in a debugger or tracer (one may check 4 instruction at a time), we get lucky and the sought-for instruction is just the 14th:

```
.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
```

ST(0) holds the first argument (1) and second one is in [EBX].

The instruction after FDIV (FSTP) writes the result in memory:

```
.text:3011E91B DD 1E                                      fstp    ↙
    ↘ qword ptr [esi]
```

If we set a breakpoint on it, we can see the result:

```
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
```

Also as a practical joke, we can modify it on the fly:

```
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 shows 666 in the cell, finally convincing us that we have found the right point.

CHAPTER 28. FINDING THE RIGHT INSTRUCTIONS

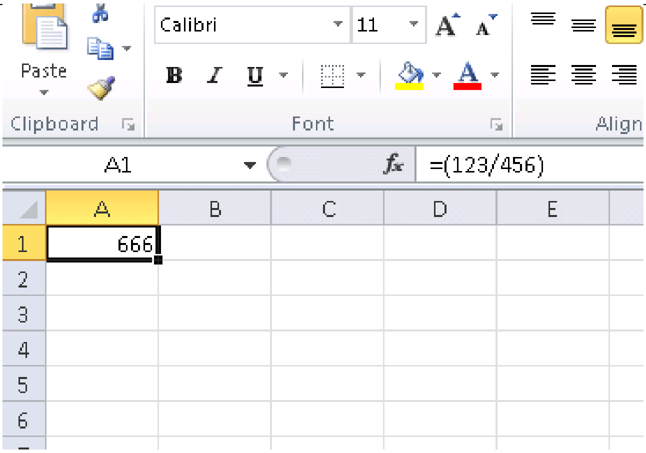


Figure 28.1: The practical joke worked

If we try the same Excel version, but in x64, we will find only 12 FDIV instructions there, and the one we looking for is the third one.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0⤦
    ⤥ ,666)
```

It seems that a lot of division operations of *float* and *double* types, were replaced by the compiler with SSE instructions like DIVSD (DIVSD is present 268 times in total).

Chapter 29

Suspicious code patterns

29.1 XOR instructions

Instructions like `XOR op, op` (for example, `XOR EAX, EAX`) are usually used for setting the register value to zero, but if the operands are different, the "exclusive or" operation is executed. This operation is rare in common programming, but widespread in cryptography, including amateur one. It's especially suspicious if the second operand is a big number. This may point to encrypting/decrypting, checksum computing,etc.

This AWK script can be used for processing IDA listing (.lst) files:

```
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
```

29.2 Hand-written assembly code

Modern compilers do not emit the `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code directly in assembly language. If you spot these, it can be said that there is a high probability that this fragment of code was hand-written.

Also the function prologue/epilogue are not commonly present in hand-written assembly.

Commonly there is no fixed system for passing arguments to functions in the hand-written code.

Example from the Windows 2003 kernel (ntoskrnl.exe file):

```
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
```

Indeed, if we look in the WRK[1] v1.2 source code, this code can be found easily in file *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

[1]Windows Research Kernel

Chapter 30

Using magic numbers while tracing

Often, our main goal is to understand how the program uses a value that was either read from file or received via network. The manual tracing of a value is often a very labour-intensive task. One of the simplest techniques for this (although not 100% reliable) is to use your own *magic number*.

This resembles X-ray computed tomography is some sense: a radiocontrast agent is injected into the patient's blood, which is then used to improve the visibility of the patient's internal structure in to the X-rays. It is well known how the blood of healthy humans percolates in the kidneys and if the agent is in the blood, it can be easily seen on tomography, how blood is percolating, and are there any stones or tumors.

We can take a 32-bit number like `0x0badf00d`, or someone's birth date like `0x11101979` and write this 4-byte number to some point in a file used by the program we investigate.

Then, while tracing this program with tracer in *code coverage* mode, with the help of *grep* or just by searching in the text file (of tracing results), we can easily see where the value was used and how.

Example of *grepable* tracer results in *cc* mode:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP↙
    ↳ +8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69↙
    ↳ AEB08h]=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=0↙
    ↳ xf1ac360
```

This can be used for network packets as well. It is important for the *magic number* to be unique and not to be present in the program's code.

Aside of the tracer, DosBox (MS-DOS emulator) in heavydebug mode is able to write information about all registers' states for each executed instruction of the program to a plain text file[1], so this technique may be useful for DOS programs as well.

[1]See also my blog post about this DosBox feature: blog.yurichev.com

Chapter 31

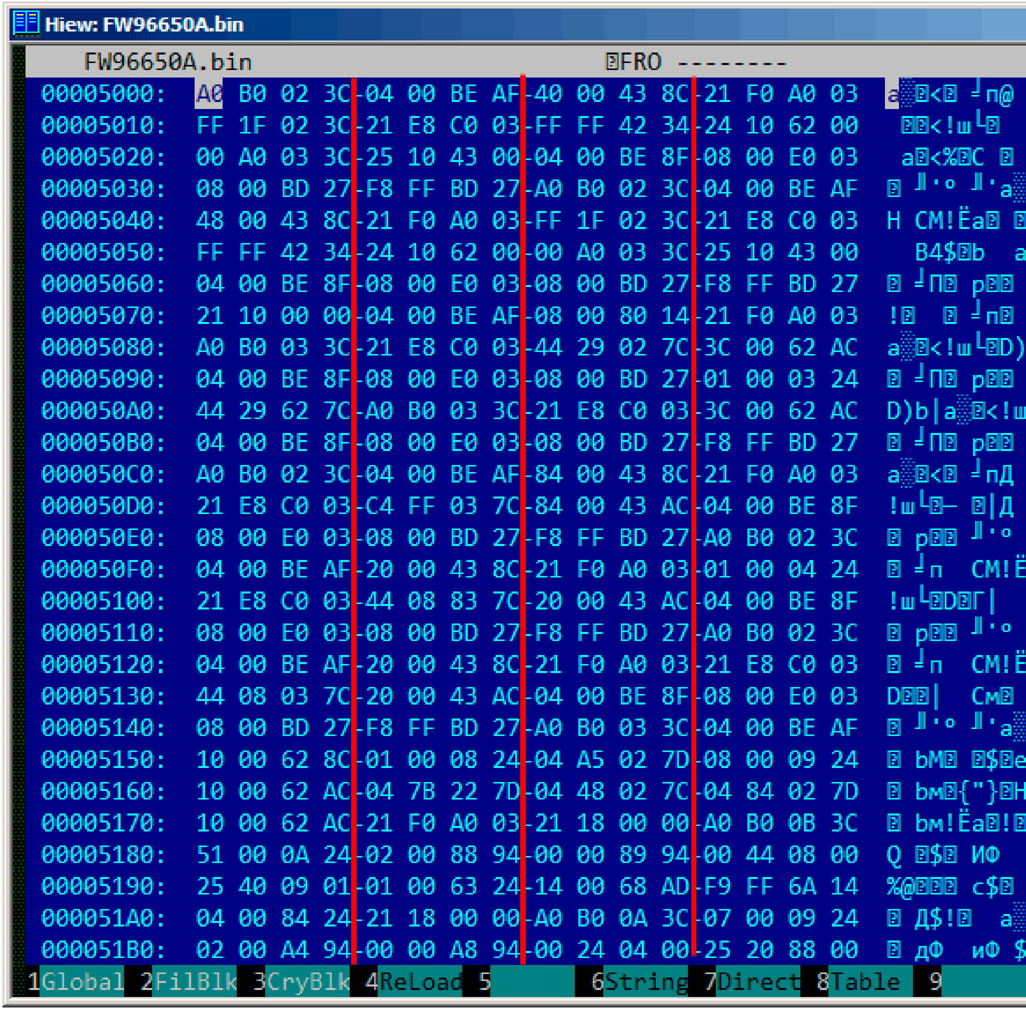
Other things

31.1 General idea

A reverse engineer should try to be in programmer's shoes as often as possible. To take his/her viewpoint and ask himself, how would one solve some task the specific case.

31.2 Some binary file patterns

Sometimes, we can clearly spot an array of 16/32/64-bit values visually, in hex editor. Here is an example of very typical MIPS code. As we may remember, every MIPS (and also ARM in ARM mode or ARM64) instruction has size of 32 bits (or 4 bytes), so such code is array of 32-bit values. By looking at this screenshot, we may see some kind of pattern. Vertical red lines are added for clarity:



```
Hiew: FW96650A.bin
    FW96650A.bin                                   FRO --------
 00005000:  A0 B0 02 3C-04 00 BE AF-40 00 43 8C-21 F0 A0 03
 00005010:  FF 1F 02 3C-21 E8 C0 03-FF FF 42 34-24 10 62 00
 00005020:  00 A0 03 3C-25 10 43 00-04 00 BE 8F-08 00 E0 03
 00005030:  08 00 BD 27-F8 FF BD 27-A0 B0 02 3C-04 00 BE AF
 00005040:  48 00 43 8C-21 F0 A0 03-FF 1F 02 3C-21 E8 C0 03
 00005050:  FF FF 42 34-24 10 62 00-00 A0 03 3C-25 10 43 00
 00005060:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
 00005070:  21 10 00 00-04 00 BE AF-08 00 80 14-21 F0 A0 03
 00005080:  A0 B0 03 3C-21 E8 C0 03-44 29 02 7C-3C 00 62 AC
 00005090:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-01 00 03 24
 000050A0:  44 29 62 7C-A0 B0 03 3C-21 E8 C0 03-3C 00 62 AC
 000050B0:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
 000050C0:  A0 B0 02 3C-04 00 BE AF-84 00 43 8C-21 F0 A0 03
 000050D0:  21 E8 C0 03-C4 FF 03 7C-84 00 43 AC-04 00 BE 8F
 000050E0:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
 000050F0:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-01 00 04 24
 00005100:  21 E8 C0 03-44 08 83 7C-20 00 43 AC-04 00 BE 8F
 00005110:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
 00005120:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-21 E8 C0 03
 00005130:  44 08 03 7C-20 00 43 AC-04 00 BE 8F-08 00 E0 03
 00005140:  08 00 BD 27-F8 FF BD 27-A0 B0 03 3C-04 00 BE AF
 00005150:  10 00 62 8C-01 00 08 24-04 A5 02 7D-08 00 09 24
 00005160:  10 00 62 AC-04 7B 22 7D-04 48 02 7C-04 84 02 7D
 00005170:  10 00 62 AC-21 F0 A0 03-21 18 00 00-A0 B0 0B 3C
 00005180:  51 00 0A 24-02 00 88 94-00 00 89 94-00 44 08 00
 00005190:  25 40 09 01-01 00 63 24-14 00 68 AD-F9 FF 6A 14
 000051A0:  04 00 84 24-21 18 00 00-A0 B0 0A 3C-07 00 09 24
 000051B0:  02 00 A4 94-00 00 A8 94-00 24 04 00-25 20 88 00
1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table  9
```

Figure 31.1: Hiew: very typical MIPS code

31.3 Memory "snapshots" comparing

The technique of the straightforward comparison of two memory snapshots in order to see changes was often used to hack 8-bit computer games and for hacking "high score" files.

For example, if you had a loaded game on an 8-bit computer (there isn't much memory on these, but the game usually consumes even less memory) and you know that you have now, let's say, 100 bullets, you can do a "snapshot" of all memory and back it up to some place. Then shoot once, the bullet count goes to 99, do a second "snapshot" and then compare both: the must be must be a byte somewhere which was 100 in the beginning, and now it is 99. Considering the fact that these 8-bit games were often written in assembly language and such variables were global, it can be said for sure which address in memory was holding the bullet count. If you searched for all references to the address in the disassembled game code, it was not very hard to find a piece of code decrementing the bullet count, then to write a NOP instruction there, or a couple of NOP-s, and then have a game with 100 bullets forever. Games on these 8-bit computers were commonly loaded at the constant address, also, there were not much different versions of each game (commonly just one version was popular for a long span of time), so enthusiastic gamers knew which bytes must be overwritten (using the BASIC's instruction POKE) at which address in order to hack it. This led to "cheat" lists that contained POKE instructions, published in magazines related to 8-bit games. See also: wikipedia.

Likewise, it is easy to modify "high score" files, this does not work with just 8-bit games. Notice your score count and back up the file somewhere. When the "high score" count gets different, just compare the two files, it can even be done with the DOS utility FC[1] ("high score" files are often in binary form). There will be a point where a couple of bytes are different and it is easy to see which ones are holding the score number. However, game developers are fully aware of such tricks and may defend the program against it.

31.3.1 Windows registry

It is also possible to compare the Windows registry before and after a program installation. It is a very popular method of finding which registry elements are used by the program. Probably, this is the reason why the "windows registry cleaner" shareware is so popular.

31.3.2 Blink-comparator

Comparison of files or memory snapshots remind us blink-comparator [2]: a device used by astronomers in past, intended to find moving celestial objects. Blink-comparator allows to switch quickly between two photographies shot in different time, so astronomer would spot the difference visually. By the way, Pluto was discovered by blink-comparator in 1930.

[1]MS-DOS utility for comparing binary files

[2]http://go.yurichev.com/17348

Part IV

Tools

Chapter 32

Disassembler

32.1 IDA

An older freeware version is available for download [1].

[1] hex-rays.com/products/ida/support/download_freeware.shtml

Chapter 33

Debugger

33.1 tracer

The author often use *tracer*[1] instead of a debugger.

The author of these lines stopped using a debugger eventually, since all he need from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called *tracer* was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

However, for learning purposes it is highly advisable to trace code in a debugger manually, watch how the registers state changes (e.g. classic SoftICE, OllyDbg, WinDbg highlight changed registers), flags, data, change them manually, watch the reaction, etc.

[1] yurichev.com

Chapter 34

Decompilers

There is only one known, publicly available, high-quality decompiler to C code: Hex-Rays:
hex-rays.com/products/decompiler/

Chapter 35

Other tools

- Microsoft Visual Studio Express[1]: Stripped-down free version of Visual Studio, convenient for simple experiments.
- Hiew[2] for small modifications of code in binary files.
- binary grep: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: GitHub.

[1]visualstudio.com/en-US/products/visual-studio-express-vs
[2]hiew.ru

Part V

Books/blogs worth reading

Chapter 36

Books

36.1 Windows

[RA09].

36.2 C/C++

[ISO13].

36.3 x86 / x86-64

[Int13], [AMD13]

36.4 ARM

ARM manuals: `http://go.yurichev.com/17024`

36.5 Cryptography

[Sch94]

Chapter 37

Blogs

37.1 Windows

- Microsoft: Raymond Chen
- nynaeve.net

Chapter 38

Other

There are two excellent RE[1]-related subreddits on reddit.com: reddit.com/r/ReverseEngineer
and reddit.com/r/remath (on the topics for the intersection of RE and mathematics).

There is also a RE part of the Stack Exchange website:
reverseengineering.stackexchange.com.

On IRC there a ##re channel on FreeNode[2].

[1] Reverse Engineering
[2] freenode.net

Afterword

Chapter 39

Questions?

Do not hesitate to mail any questions to the author: `<dennis(a)yurichev.com>`

Any suggestions what also should be added to my book?

Please, do not hesitate to send me any corrections (including grammar (you see how horrible my English is?)),etc.

The author is working on the book a lot, so the page and listing numbers, etc. are changing very rapidly. Please, do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to me. He'll fix it much faster. And if you familiar with git and LaTeX you can fix the error right in the source code:
GitHub.

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.

Warning: this is a shortened LITE-version!

It is approximately 6 times shorter than full version (~150 pages) and intended to those who wants for very quick introduction to reverse engineering basics. There are nothing about MIPS, ARM, OllyDBG, GCC, GDB, IDA, there are no exercises, examples, etc.

If you still interesting in reverse engineering, full version of the book is always available on my website: beginners.re.

Acronyms used

OS	Operating System xi
PL	Programming language 3
ROM	Read-only memory 177
RA	Return Address 66
SP	stack pointer. SP/ESP/RSP in x86/x64. SP in ARM. 13
PC	Program Counter. IP/EIP/RIP in x86/64. PC in ARM. 205
IDA	Interactive Disassembler and debugger developed by Hex-Rays
MSVC	Microsoft Visual C++
AKA	Also Known As
CRT	C runtime library 9
CPU	Central processing unit xi
SIMD	Single instruction, multiple data 80
DBMS	Database management systems ix
ISA	Instruction Set Architecture 3
SEH	Structured Exception Handling 19
NOP	No OPeration 38
RAM	Random-access memory 154
API	Application programming interface 163

ASCIIZ ASCII Zero (null-terminated ASCII string) 37

VM Virtual Memory

WRK Windows Research Kernel 184

GPR General Purpose Registers 3

RE Reverse Engineering 198

BOM Byte order mark 170

MBR Master Boot Record 177

RFC Request for Comments 173

EOF End of file 34

Glossary

real number numbers which may contain a dot. this is *float* and *double* in C/C++. 92

decrement Decrease by 1. 77, 86, 189

increment Increase by 1. 77, 86

product Multiplication result. 41

stack pointer A register pointing to a place in the stack. 9, 13, 17, 22, 203

quotient Division result. 92

anti-pattern Generally considered as bad practice. 30, 159

callee A function being called by another. 11, 16, 26, 27, 35, 40, 42, 66, 154, 159

caller A function calling another. 5, 9, 35, 40, 41, 44, 66, 154

heap usually, a big chunk of memory provided by the OS so that applications can divide it by themselves as they wish. malloc()/free() work with the heap. 14, 16, 132

jump offset a part of the JMP or Jcc instruction's opcode, to be added to the address of the next instruction, and this is how the new PC[1] is calculated. May be negative as well. 38, 54

NOP "no operation", idle instruction. 189

PDB (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names. 162

POKE BASIC language instruction for writing a byte at a specific address. 189

[1]Program Counter. IP/EIP/RIP in x86/64. PC in ARM.

register allocator The part of the compiler that assigns CPU registers to local variables. 86, 154

reverse engineering act of understanding how the thing works, sometimes in order to clone it. iv

stack frame A part of the stack that contains information specific to the current function: local variables, function arguments, RA, etc. 28, 41

stdout standard output. 18, 66

tracer My own simple debugging tool. You can read more about it here: 33.1 on page 192. 165, 180, 185, 186

Windows NT Windows NT, 2000, XP, Vista, 7, 8. 170

Index

Buffer Overflow, 95
C language elements
  Pointers, 26, 153
  C99
    bool, 112
    variable length arrays, 101
  const, 8
  for, 77
  if, 50, 65
  return, 9, 35
  switch, 63, 65
  while, 84
C standard library
  alloca(), 17, 101, 159
  assert(), 175
  free(), 159
  longjmp(), 66
  malloc(), 133, 159
  memcmp(), 178
  memcpy(), 27
  rand(), 125, 163
  realloc(), 159
  scanf(), 26
  strlen(), 84
Compiler's anomalies, 61, 121
C++
  STL, 162
grep usage, 162, 180, 185
Global variables, 30
Buffer overflow, 101
Recursion, 11, 15
Stack, 13, 40, 66
  Stack overflow, 15
  Stack frame, 28
Syntactic Sugar, 65
OllyDbg, 99, 106, 107
Oracle RDBMS, 9, 173

ARM
  Instructions
    ASR, 121
    CSEL, 62
    LSL, 121
    LSR, 121
    MOV, 5
    MOVcc, 62
    POP, 13
    PUSH, 13
    TEST, 86
AWK, 183

Base64, 172
base64, 174
bash, 46
BASIC
  POKE, 189
binary grep, 179, 194
Borland Delphi, 168

cdecl, 22
column-major order, 105
Compiler intrinsic, 19
Cygwin, 165

DosBox, 186

Error messages, 173

fastcall, 10, 25
FORTRAN, 105
Function epilogue, 11, 183
Function prologue, 11, 183

Hiew, 37, 54, 167

IDA, 171
Intel C++, 9

jumptable, 69

MD5, 177
MIDI, 178
MIPS, 187
MS-DOS, 178, 186, 189

Pascal, 168
puts() instead of printf(), 46

Register allocation, 154
row-major order, 105

SAP, 162
Security through obscurity, 174
Shadow space, 43
Signed numbers, 52, 157

tracer, 165, 180, 185, 192

Unicode, 168
UTF-16LE, 168, 170
UTF-8, 168, 169

Windows
  KERNEL32.DLL, 113
  PDB, 162
  Structured Exception Handling, 19
  Win32, 112, 170

x86
  Instructions
    ADC, 149
    ADD, 9, 22, 41
    AND, 113, 115, 118, 122, 146
    CALL, 8, 14
    CBW, 158
    CDQ, 152, 158
    CDQE, 158
    CMOVcc, 59, 62
    CMP, 35
    CMPSB, 178
    CPUID, 142
    CWD, 158
    CWDE, 158
    DEC, 86
    DIV, 158
    DIVSD, 182
    FDIV, 180, 181
    IDIV, 158
    IMUL, 41, 158
    INC, 86
    INT3, 165
    JA, 52, 158
    JAE, 52
    JB, 52, 158
    JBE, 52
    Jcc, 61
    JE, 66
    JG, 52, 158
    JGE, 52
    JL, 52, 158
    JLE, 51
    JMP, 14
    JNE, 35, 36, 52
    JZ, 66
    LEA, 29, 42
    LOOP, 77, 82, 183
    MOV, 5, 9
    MOVSX, 85, 158
    MOVSXD, 103
    MOVZX, 134
    MUL, 158
    OR, 115
    POP, 9, 13, 15
    PUSH, 8, 13, 14, 28
    RCL, 183
    RET, 5, 9, 15
    ROL, 120

SAR, 121, 158
SBB, 149
SHL, 88, 95, 121
SHR, 92, 121, 145
SHRD, 152
SUB, 9, 35, 66
TEST, 85, 113, 122
XOR, 9, 35, 183
Registers
Flags, 35
EAX, 35, 45
EBP, 28, 41
ESP, 22, 28
JMP, 71
ZF, 36, 113
x86-64, 9, 23, 27, 29, 38, 42, 153

Bibliography

[AMD13] AMD. AMD64 Architecture Programmer's Manual. Also available as http://go.yurichev.com/17284. 2013.

[Dij68] Edsger W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: Commun. ACM 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: http://go.yurichev.com/17299.

[Fog13] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs / An optimization
http://go.yurichev.com/17278. 2013.

[Int13] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined
Also available as http://go.yurichev.com/17283. 2013.

[ISO07] ISO. ISO/IEC 9899:TC3 (C C99 standard). Also available as http://go.yurichev.com/17274. 2007.

[ISO13] ISO. ISO/IEC 14882:2011 (C++ 11 standard). Also available as http://go.yurichev.com/17275. 2013.

[Ker88] Brian W. Kernighan. The C Programming Language. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[Knu74] Donald E. Knuth. "Structured Programming with go to Statements". In: ACM Comput. Surv. 6.4 (Dec. 1974). Also available as http://go.yurichev.com/17271, pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640. URL: http://go.yurichev.com/17300.

[Knu98] Donald E. Knuth. The Art of Computer Programming Volumes 1-3 Boxed Set. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201485419.

[Mit13] Michael Matz / Jan Hubicka / Andreas Jaeger / Mark Mitchell. System V Applicatio
Also available as http://go.yurichev.com/17295. 2013.

[One96] Aleph One. "Smashing The Stack For Fun And Profit". In: Phrack (1996). Also available as http://go.yurichev.com/17266.

[Pre+07] William H. Press et al. Numerical Recipes. 2007.

[RA09] Mark E. Russinovich and David A. Solomon with Alex Ionescu. Windows® Internal
2009.

[Rit79] Dennis M. Ritchie. “The Evolution of the Unix Time-sharing System”. In: (1979).

[RT74] D. M. Ritchie and K. Thompson. “The UNIX Time Sharing System”. In: (1974). Also available as http://go.yurichev.com/17270.

[Sch94] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code ir
1994.

[Str13] Bjarne Stroustrup. The C++ Programming Language, 4th Edition. 2013.

[Yur13] Dennis Yurichev. C/C++ programming language notes. Also available as http://go.yurichev.com/17289. 2013.