# X86/X64软件逆向分析【提高篇】

## 课前准备&工具安装与配置（1）

1. 课程回顾：《x86/x64软件逆向分析入门》
2. Visual Studio 2019开发套件：<http://www.microsoft.com>
3. 动态调试工具：

* x64dbg：<https://x64dbg.com>

1. 静态分析工具

* IDA Pro：<https://www.hex-rays.com/>
* Ghidra：<https://github.com/NationalSecurityAgency/ghidra>

1. 文本查看比较工具

* Notepad++: <https://notepad-plus-plus.org/>
* Beyond Compare: <https://www.scootersoftware.com/>

1. 参考文档：

* 英特尔64和IA-32体系结构软件开发人员手册(Intel® 64 and IA-32 Architectures Software Developer Manuals) <https://software.intel.com/content/www/cn/zh/develop/articles/intel-sdm.html>

1. 课程源代码：<https://github.com/zmrbak/ReverseAnalysis2>
2. 如何下载课程源代码

## 课前准备&工具安装与配置（2）

## 如何将机器码转成汇编代码（1）

0x12345678

低->高

12,34,56,78

77551000 **12345678** 7755C150 001E001C 7755C130

77551000 **78 56 34 12** 50 C1 55 77 1C 00 1E 00 30 C1 55 77 xV4.PÁUw....0ÁUw

|  |  |  |  |
| --- | --- | --- | --- |
| 77600E0B | 890D B4576777 | mov dword ptr ds:[776757B4],ecx |  |
|  |  |  |  |
| 77600DFD | 59 | pop ecx |  |

59

POPd64 into general register rCX/r9

POP ECX

|  |  |  |  |
| --- | --- | --- | --- |
| 77600E09 | 33C9 | xor ecx,ecx |  |
| 33C9  XOR Gv, Ev  . |  |  |  |

G:

The reg field of the ModR/M byte selects a general register (for example, AX (000))

v:

Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.

E:

A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

ECX/CX/CL/MM/XMM1

ECX

XOR ECX, ECX

//532页

77600E1C | 74 05 | je ntdll.77600E23 |

**74 05**

Jccf64, Jb - Short-displacement jump on condition Z/E

**J**

The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP(0E9), LOOP).

**b:**

Byte, regardless of operand-size attribute.

**jcc offset 05**

**77600E29 | 8BEC | mov ebp,esp |**

**8BEC**

MOV Gb, Eb

**MOV EBP,ESP**

|  |  |  |  |
| --- | --- | --- | --- |
| 77600DF3 | 8B4D F0 | mov ecx,dword ptr ss:[ebp-10] |  |

8B4D F0

MOV Gb, Eb

**MOV ECX,** [EBP] F0

**MOV ECX,** [EBP]

**MOV ECX, dowrd prt ss:** [EBP

**MOV ECX,** [EBP

|  |  |  |  |
| --- | --- | --- | --- |
| 77600E03 | 64:A1 30000000 | mov eax,dword ptr fs:[30] |  |

64:A1 30 00 00 00

64

SEG=FS (Prefix)

**MOV** EAX, Ov

**O**

The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

**MOV EAX,dword ptr FS:[30]**

**MOV EAX,FS:[30]**

**第二卷，**

**APPENDIX A OPCODE MAP**

**A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS ，P2672**

**CHAPTER 2 INSTRUCTION FORMAT**

**2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE**

**2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes,P532**

## 如何将机器码转成汇编代码（2）

## 程序的优化编译与SIMD指令（1）

MOVUPS

Move four **unaligned packed** **single-precision floating-point values** between XMM registers or between and XMM register and memory.

32位\*4=128位

**SSE Data Transfer Instructions**

**SSE SIMD Single-Precision Floating-Point Instructions**

**SSE：Streaming SIMD Extensions.**

**SIMD：Single Instruction, Multiple Data**

**一条指令，处理多个数据**

**FPU，SIMD**

**图像，音频，视频，加密解密**

PADDD

Add packed doubleword integers.

**Dword 32位**

**16字节，每一个字节8位，一共16\*8=128位**

VMOVDQU32/64

V**MOV**DQU with 32/64-bit granular conditional update.

VZEROUPPER

Zero upper 128 bits of all YMM registers

VZEROALL

Zero all YMM registers

## 程序的优化编译与SIMD指令（1）

## 编译器内置的SIMD指令支持（1）

typedef union **\_\_declspec(intrin\_type)** **\_\_declspec(align(16))** \_\_m128i {

\_\_int8 m128i\_i8[16];

\_\_int16 m128i\_i16[8];

\_\_int32 m128i\_i32[4];

\_\_int64 m128i\_i64[2];

unsigned \_\_int8 m128i\_u8[16];

unsigned \_\_int16 m128i\_u16[8];

unsigned \_\_int32 m128i\_u32[4];

unsigned \_\_int64 m128i\_u64[2];

} \_\_m128i;

<https://docs.microsoft.com/zh-cn/cpp/cpp/declspec?view=msvc-160>

\_\_declspec

Microsoft 专用

用于指定存储类信息的扩展特性语法使用 \_\_declspec 关键字，该关键字指定给定类型的实例将与下面列出的 Microsoft 特定存储类特性一起存储。 其他存储类修饰符的示例包括 static 和 extern 关键字。 但是，这些关键字是 C 和 C++ 语言的 ANSI 规范的一部分，并且本身不包含在扩展特性语法中。 扩展特性语法简化并标准化了 Microsoft 专用的 C 和 C ++ 语言扩展。

//内存16字节对齐

\_mm\_load\_si128

\_\_m128i \_mm\_load\_si128 (\_\_m128i \*p);

MOV**DQA**

//不需要16字节对齐

\_mm\_loadu\_si128.

\_\_m128i \_mm\_load**u**\_si128 (\_\_m128i \*p);

MOV**DQU**

MOVUPS \_\_m128 \_mm\_loadu\_ps ( float \* p);

**x86 内部函数列表**

<https://docs.microsoft.com/zh-cn/cpp/intrinsics/x86-intrinsics-list?view=msvc-160>

Intel 内部函数指南

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

\_\_m128i **\_mm\_lddqu\_si128** (\_\_m128i const\* mem\_addr)  
#include <pmmintrin.h>  
Instruction: lddqu xmm, m128  
CPUID Flags: SSE3

\_\_m128i **\_mm\_loadu\_si128** (\_\_m128i const\* mem\_addr)  
#include <emmintrin.h>  
Instruction: movdqu xmm, m128  
CPUID Flags: SSE2

\_\_m128i \_mm\_add\_epi32 (\_\_m128i a, \_\_m128i b)  
#include <emmintrin.h>  
Instruction: paddd xmm, xmm  
CPUID Flags: SSE2

Add packed 32-bit integers in a and b, and store the results in dst.

FOR j := 0 to 3

i := j\*32

dst[i+31:i] := a[i+31:i] + b[i+31:i]

ENDFOR

## 编译器内置的SIMD指令支持（2）

## X86/X64平台中寄存器的使用（1）

16位，指针16位，0-2^16, 65,536

32位，指针32位，0-2^32, 4,294,967,296

64位，指针64位，0-2^64, 18,446,744,073,709,551,616

X86-64

mov eax, 0x11111111

mov ax, 0x2222

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RAX | | | | | | | |
|  | | | | EAX | | | |
|  | | | | | | AX | |
|  | | | | | | AH | AL |

|  |  |  |  |
| --- | --- | --- | --- |
| 00007FFF615FF781 | 48:B8 1111111111111111 | mov rax,1111111111111111 |  |
| 00007FFF615FF78B | B8 22222222 | mov eax,22222222 |  |
| 00007FFF615FF790 | 66:B8 3333 | mov ax,3333 |  |
| 00007FFF615FF794 | B0 44 | mov al,44 | 44:'D' |
| 00007FFF615FF796 | B4 55 | mov ah,55 | 55:'U' |
|  |  |  |  |
| 00007FFF615FF798 | 49:B9 1111111111111111 | mov r9,1111111111111111 |  |
| 00007FFF615FF7A2 | 41:B9 22222222 | mov r9d,22222222 |  |
| 00007FFF615FF7A8 | 6641:B9 3333 | mov r9w,3333 |  |

//fast call

cl RE009.cpp /Fa1.2.asm /Ox /Gr

\_TEXT SEGMENT

\_a3$ = 8 ; size = 4

\_a4$ = 12 ; size = 4

\_a5$ = 16 ; size = 4

\_a6$ = 20 ; size = 4

\_a7$ = 24 ; size = 4

\_a8$ = 28 ; size = 4

\_a9$ = 32 ; size = 4

\_a10$ = 36 ; size = 4

?f@@YIHHHHHHHHHHH@Z PROC ; f

**; \_a1$ = ecx**

**; \_a2$ = edx**

; File C:\Users\Zmrbak\source\repos\RE009\RE009\RE009.cpp

; Line 8

**lea eax, DWORD PTR [ecx+edx]**

add eax, DWORD PTR \_a3$[esp-4]

add eax, DWORD PTR \_a4$[esp-4]

add eax, DWORD PTR \_a5$[esp-4]

add eax, DWORD PTR \_a6$[esp-4]

add eax, DWORD PTR \_a7$[esp-4]

add eax, DWORD PTR \_a8$[esp-4]

add eax, DWORD PTR \_a9$[esp-4]

add eax, DWORD PTR \_a10$[esp-4]

; Line 9

ret 32 ; 00000020H

?f@@YIHHHHHHHHHHH@Z ENDP ; f

\_TEXT ENDS

//x86

**/Gd \_\_cdecl 调用约定 /Gr \_\_fastcall 调用约定**

**/Gz \_\_stdcall 调用约定** /GZ 启用堆栈检查(/RTCs)

/Gv \_\_vectorcall 调用约定 **/QIfist[-] 使用 FIST 而不是 ftol()**

//X64

/GZ 启用堆栈检查(/RTCs) /Gv \_\_vectorcall 调用约定

//X64

mov DWORD PTR [rsp+72], 10

mov DWORD PTR [rsp+64], 9

mov DWORD PTR [rsp+56], 8

mov DWORD PTR [rsp+48], 7

mov DWORD PTR [rsp+40], 6

mov DWORD PTR [rsp+32], 5

mov r9d, 4

mov r8d, 3

mov edx, 2

mov ecx, 1

call ?f@@YAHHHHHHHHHHH@Z ; f

## X86/X64平台中寄存器的使用（1）

## X86/X64平台中的浮点数计算（1）

SIMD指令相比于FPU指令来说，使用更简单，可读性也更强。

FPU、SIMD，浮点数，IEEE754格式

目前编译器针对的X86-64CPU，越来越多使用SIMD指令。

**mov**ss xmm0, DWORD PTR \_\_real@4407cd71

movss DWORD PTR [esp], xmm0

Scalar Single: 标量单精度浮点数

MOVSS

Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory.

cvtss2sd xmm0, DWORD PTR \_a$[ebp]

CVTSS2SD

Convert scalar single-precision floating-point values to scalar double-precision floating point values.

\_\_real@408428f6 DD **0408428f6**r ; 4.13

\_\_real@4010851eb851eb85 DQ **04010851eb851eb85**r ; 4.13

DIVSS

Divide scalar single-precision floating-point values.

DIVSD

Divide scalar double-precision floating-point values.

MULSD

Multiply scalar double-precision floating-point values.

MULSS

Multiply scalar single-precision floating-point values.

ADDSS

Add scalar single-precision floating-point values.

COMISD

Perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.

vcmpltsd xmm2, xmm1, xmm0

vblendvpd xmm0, xmm1, xmm0, xmm2

## X86/X64平台中的浮点数计算（2）

## 数字的存储与内容解析（1）

WCHAR

UNICODE

UTF-16LE

UTF-8,

ASCII,0-255,0x00-0xFF

#define CHAR\_BIT 8

#define SCHAR\_MIN (-128)

#define SCHAR\_MAX 127

#define UCHAR\_MAX 0xff

775B1000 3.82047e-037 1.00825e-034 2.65846e-032 7.00365e-030

775B1010 1.84362e-027 4.84942e-025 1.27467e-022 3.34819e-020

775B1020 8.78905e-018 2.30573e-015 6.04532e-013 1.58413e-010

775B1030 4.14886e-008 1.08604e-005 0.00284155 0.743122

775B1040 194.255 50757.3 1.3257e+007 3.46117e+009

775B1050 9.03307e+011 2.35662e+014 6.14598e+016 1.60231e+019

775B1060 4.17598e+021 1.08801e+024 2.83386e+026 7.37897e+028

775B1070 1.92083e+031 4.99878e+033 1.30054e+036 3.38275e+038

775B1080 -7.67045e-037 -2.02406e-034 -5.33626e-032 -1.40568e-029

775B1090 -3.69991e-027 -9.73128e-025 -2.55764e-022 -6.71763e-020

775B10A0 -1.76325e-017 -4.62538e-015 -1.21263e-012 -3.17738e-010

775B10B0 -8.32109e-008 -2.17807e-005 -0.00569841 -1.49017

775B10C0 -389.514 -101772 -2.65799e+007 -6.93919e+009

775B10D0 -1.81093e+012 -4.72428e+014 -1.23202e+017 -3.21185e+019

775B10E0 -8.37048e+021 -2.18077e+024 -5.67986e+026 -1.4789e+029

775B10F0 -3.84962e+031 -1.00179e+034 -2.60629e+036 -1.#QNAN

775B1100 0 0 0 0

775B1000 7.94992889512736e-275 3.69191620486509e-236

775B1010 1.84632392568185e-197 8.56774561661236e-159

775B1020 4.28794340323905e-120 1.98828852527466e-081

775B1030 9.95833437889674e-043 0.000461413578911957

775B1040 2.31270850962124e+035 1.07077806593965e+074

775B1050 5.37095661886286e+112 2.48488731765823e+151

775B1060 1.24732309260667e+190 5.76649710375256e+228

775B1070 2.8966956840444e+267 1.3381833135893e+306

775B1080 -2.08157600053169e-272 -9.60917677782711e-234

775B1090 -4.83403088450085e-195 -2.22990337163316e-156

775B10A0 -1.12259529394798e-117 -5.17468694387072e-079

775B10B0 -2.6069558730967e-040 -0.120082660515186

775B10C0 -6.05397785008957e+037 -2.78660051125151e+076

775B10D0 -1.40586842870057e+115 -6.46647081108696e+153

775B10E0 -3.26471481303578e+192 -1.50057648353792e+231

775B10F0 -7.58128041190211e+269 -1.#QNAN

775B1100 0 0

## 数字的存储与内容解析（1）

## 不可逆算法与哈希函数（1）

设计一种加密算法（不可逆算法）：

1. 从0到9，随机抽取10个数字，按顺序记录下来，作为密码。
2. 取第一个数字，作为参数1。
3. 取第二个数字，作为参数2。
4. 加密：
   1. 找出“参数1”所在位置的值1
   2. 找出“参数2”所在位置的值2
   3. 值1与值2下相互交换

密码

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 8 | 4 | 5 | 3 | 4 | 9 | 0 | 1 | 5 |

加密：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 8 | 4 | 5 | 3 | 4 | 9 | 0 | 1 | 5 |

第一个数字，3

第二个数字，8

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 8 | 4 | 1 | 3 | 4 | 9 | 0 | 5 | 5 |

密码：3845349015

密文：3841349055

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 1 | 4 | 5 | 3 | 4 | 9 | 0 | 1 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 1 | 4 | 5 | 3 | 4 | 9 | 0 | 1 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 5 | 4 | 1 | 3 | 4 | 9 | 0 | 1 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| 3 | 5 | 4 | 4 | 3 | 1 | 9 | 0 | 1 | 5 |

哈希函数

CRC32。

哈希值，一种**固定长度**的信息摘要，不可根据哈希值逆向推导出原文。

防止伪造。多种不同的加密方式。

网站密码、数据库密码。

https://www.cmd5.com/

01B307ACBA4F54F55AAFC33BB06BBBF6CA803E9A

1234567890

SHA1

Y=X+X^2+X^3+4X^(1/2);

X ->Y ? 容易

Y- >X ? 非常难

## 不可逆算法与哈希函数（2）

## 经典程序剖析：“温度转换”（1）

C=5\*(F-32)/9

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 |
| 错误 | C4996 | 'scanf': This function or variable may be unsafe. Consider using scanf\_s instead. To disable deprecation, use \_CRT\_SECURE\_NO\_WARNINGS. See online help for details. | RE017 | C:\Users\Zmrbak\source\repos\RE017\RE017\RE017.cpp | 12 |  |

#define \_CRT\_SECURE\_NO\_WARNINGS

if (scanf\_s("%d", &fahrenheit) != 1)

{

printf("Error while parsing your input!\n");

exit(0);

};

%**c 字符**

%**C**  宽字符

%**s**  字符数组

%**S 宽字符数组**

sub ecx, 32

add eax, -32

imul eax, ecx, 5

lea ecx, DWORD PTR [eax+eax\*4]

mov ecx, 9

idiv ecx

mov eax, 954437177 ; 38e38e39H

imul ecx

sar edx, 1

mov eax, edx

shr eax, 31 ; 0000001fH

add eax, edx

## 经典程序剖析：“温度转换”（2）

## 经典程序剖析：“斐波那契数列”

斐波那契数列：

0,1,1,2,3,5,8,13,21,33，。。。。。。

从第三项开始，每一项都是前两项之和。

0 -> 0

1 -> 1

2 ->1

3 ->2

4 -> 3

Fn=F(n-2)+F(n-1) ,n>=2

Fn = 0 ,n=0

Fn = 1 ,n=1

递归函数：

1. 简单、明了，与数学函数保持一致。
2. 对堆栈压力比较大，太多的递归次数，会导致堆栈溢出，从而导致程序崩溃。
3. 对堆栈有大量得到操作，导致程序性能低下。
4. 在注重性能的环境下，应避免使用递归。
5. 所有的递归，都可以使用等效的循环来实现。

## 经典程序剖析：“循环冗余校验码”

校验码

奇偶校验：根据被传输的一组二进制代码的数位中“1”的个数是奇数或偶数来进行校验。

奇校验：101010101000001111，9个1，奇数，加一个校验位，1010101010000011110

偶校验：101010101000001111，9个1，奇数，加一个校验位，1010101010000011111

1010101010**1**00011111，错误数据

10101**1**1010**1**00011111，偶数，认为传输的数据是正确的，但实际是错误的。

只能监测出数据中出现一位错误的状况，超过1位出现错误，无法监测到。

循环冗余校验，Cyclic Redundancy Check，CRC

**CRC校验（循环冗余校验）小知识**

CRC即循环冗余校验码（Cyclic Redundancy Check）：是数据通信领域中最常用的一种**查错**校验码，其特征是**信息字段**和**校验字段**的长度可以任意选定。循环冗余检查（CRC）是一种数据传输**检错**功能，对数据进行多项式计算，并将得到的结果附在帧的后面，接收设备也执行类似的算法，以保证数据传输的正确性和完整性。

CRC算法参数模型解释：

NAME：参数模型名称。

WIDTH：宽度，即CRC比特数。

POLY：生成项的简写，以16进制表示。例如：CRC-32即是0x04C11DB7，忽略了最高位的"1"，即完整的生成项是0x1’04C11DB7。

INIT：这是算法开始时寄存器（crc）的初始化预置值，十六进制表示。

*XOROUT：计算结果与此参数异或后得到最终的CRC值。*

REFIN：待测数据的**每个字节**是否按位反转，True或False。

*REFOUT：在计算之后，异或输出之前，****整个数据****是否按位反转，True或False。*

检错VS纠错

传输的数据：有效数据 +crc32(有效数据)

POLY

x8+x2+x+1

2^8+2^2+2^1+2^0

256+4+2+1=263= 0x107

0x1’ 07

1’00000111

## 经典程序剖析：“循环冗余校验码”原理以及手算（1）

<http://www.ip33.com/crc.html>

模2除法

传输的数据+校验码

我们直观想到的算法：

1. 加法
2. 减法
3. 乘法
4. 除法
5. 异或
6. 查表

算法的性能？

乘法/除法 ->加法，移位

移位，布尔运算，加法

“模2除法”与“算术除法”类似，但它既不向上位借位，也不比较除数和被除数的相同位数值的大小，只要以相同位数进行相除即可。

除法：乘法 + 减法

异或运算：

相同为0，不同为1  
 //61 62 63

011000010110001001100011

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |

//107

000100000111

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |

## 经典程序剖析：“循环冗余校验码”原理以及手算（2）

## 经典程序剖析：CRC-8软件编码

异或运算满足交换律

A异或B异或C= A异或（B异或C）= A异或（C异或B）= A异或C异或B

## 经典程序剖析：CRC-32/MPEG-2软件编码

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |  |

0xE66C6494

0xE66C6494

0x9B73448C

0x9B73448C

0x9B73448C

## 经典程序剖析：CRC-32软件编码

* 1. 这是算法开始时寄存器（crc）的初始化预置值
  2. 待测数据的每个字节按位反转
  3. 计算
  4. 在计算后之后，整个数据按位反转
  5. 计算结果与此参数异或后得到最终的CRC值

0xE8B7BE43

0xE8B7BE43

0x352441C2

0x352441C2

0x352441C2

0x4EB86AEA

0x4EB86AEA

压缩文件，会用CRC32做校验

游戏程序，对代码进行CRC32校验

32数字，2^32。碰撞

【原创】验证：CRC-32循环冗余校验

<https://blog.csdn.net/u013667796/article/details/117780420>

## 经典程序剖析：网络地址计算（1）

<https://www.sojson.com/>

IP地址范围：

1.0.0.0 -223.255.255.255

0x01000000 – 0xDFFFFFFF, 32位的无符号整数



192,168,240,140

0xc0a8f08c

## 经典程序剖析：网络地址计算（2）

## 循环与迭代器的优化

imul edx, DWORD PTR \_i$1[ebp], 7

imul eax, DWORD PTR \_i$1[ebp], 3

lea edx, DWORD PTR [edx+28]

lea ecx, DWORD PTR [ecx+12]

mov eax, DWORD PTR [edx]

lea edx, DWORD PTR [edx+28]

mov DWORD PTR [ecx], eax

lea ecx, DWORD PTR [ecx+12]

sub esi, 1

jne SHORT $LL4@f

mov eax, DWORD PTR [edx]

lea ecx, DWORD PTR [ecx+12]

mov DWORD PTR [ecx-12], eax

lea edx, DWORD PTR [edx+28]

sub esi, 1

jne SHORT $LL4@f1

count/7:

mov ecx, DWORD PTR \_count$[esp-4]

mov eax, 613566757 ; 24924925H

mul ecx

sub ecx, edx

shr ecx, 1

add ecx, edx

shr ecx, 2

## 经典程序剖析：神奇的达夫装置

达夫装置，Duff’s Device，它是一种综合了多种控制语句的循环展开技术，它可以大幅度地减少代码的分支数量。这种**循环展开**技术，巧妙地利用了switch语句的fallthrough（贯穿，滑梯）效应。

<https://blog.csdn.net/kingmax26/article/details/5252657>

TC++PL ：The C++ Programming Language，Bjarne

## 经典程序剖析：除法变乘法

cdq -> EDX:EAX (EDX，符号位)

CWD/CDQ Convert word to doubleword/Convert doubleword to quadword.

?f\_unsigned@@YAII@Z PROC ; f\_unsigned

; File C:\Users\Zmrbak\source\repos\RE030\RE030\RE030.cpp

; Line 13

// 2863311531

mov eax, -1431655765 ; aaaaaaabH

//EDX:EAX = 0xaaaaaaab \* a

//EDX= EDX:EAX/2^32 = EDX:EAX>>32

//EDX =0xaaaaaaab \* a /2^32

mul DWORD PTR \_a$[esp-4]

//EDX =0xaaaaaaab \* a /2^32/2

=0xaaaaaaab \* a /2^33

shr edx, 1

//结果：eax

mov eax, edx

; Line 14

ret 0

?f\_unsigned@@YAII@Z ENDP ; f\_unsigned

result=input/divisor

=

= // M=

=

M==1,431,655,765.3333333333333333333333 //有符号

M== 2,863,311,530.6666666666666666666667 //无符号 + 无符号

M:magic number,魔法数字，魔术因子

2863311531

2863311531

result=a/3

=a\*2863311531/(2^33)

IMUL Signed multiply.

EDX:EAX := EAX*r/m*32.

MUL Unsigned multiply

Unsigned multiply (EDX:EAX := EAX *r/m32*).

?f\_signed@@YAHH@Z PROC ; f\_signed

; File C:\Users\Zmrbak\source\repos\RE030\RE030\RE030.cpp

; Line 8

mov eax, 1431655766 ; 55555556H

imul DWORD PTR \_a$[esp-4]

//EDX:EAX >>32

//EAX=EDX= a \*0x55555556/2^32

mov eax, edx

//EAX只剩下符号位

shr eax, 31 ; 0000001fH

//edx加符号位

add eax, edx

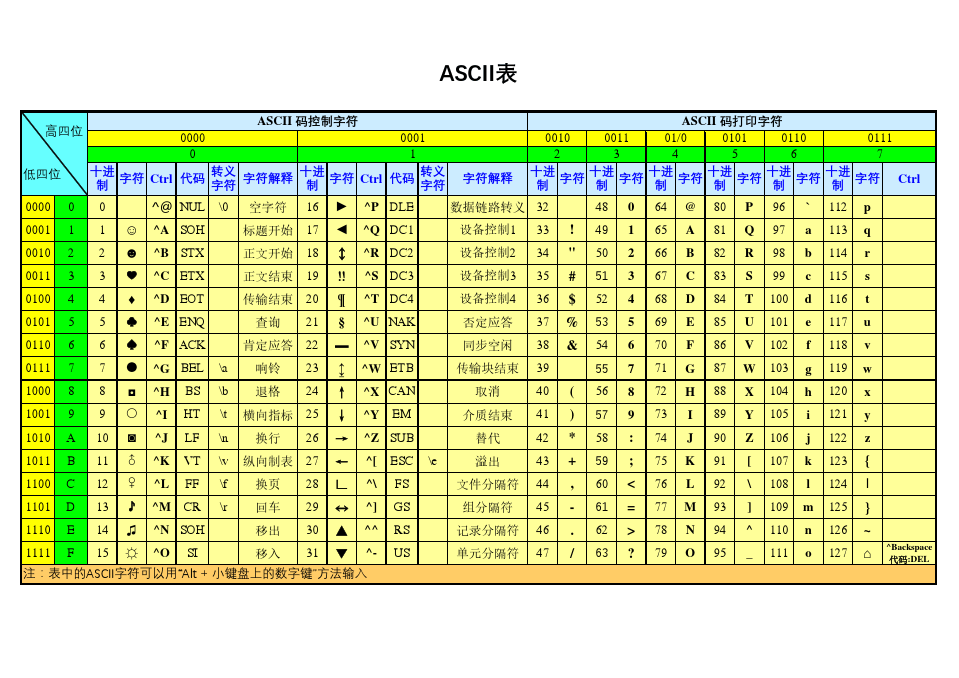
; Line 9

ret 0

?f\_signed@@YAHH@Z ENDP ; f\_signed

\_TEXT ENDS

## 经典程序剖析：字符串转数字



1=‘1’-‘0’

## 内联函数VS反编译器（1）

在编译阶段，被编译器把函数体展开，并嵌入到每一个调用点的函数，即内联函数。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 严重性 | 代码 | 说明 | 项目 | 文件 | 行 | 禁止显示状态 |
| 错误 | C4996 | 'strcpy': This function or variable may be unsafe. Consider using strcpy\_s instead. To disable deprecation, use \_CRT\_SECURE\_NO\_WARNINGS. See online help for details. | RE032 | C:\Users\Zmrbak\source\repos\RE032\RE032\RE032.cpp | 24 |  |

//IDA

int \_\_cdecl sub\_401000(int a1, int a2)

{

int v2; // eax

const char \*v3; // esi

int v4; // eax

v2 = sub\_402EF1(\*(\_DWORD \*)(a2 + 4));

**sub\_4010D0**((const char \*)&unk\_412160, 5 \* (v2 - 32) / 9);

v3 = \*(const char \*\*)(a2 + 4);

strcpy(\*(char \*\*)(a2 + 4), "12345");

sub\_4010D0("%s\n", v3);

v4 = strcmp("12345", v3);

if ( v4 )

v4 = -(v4 < 0) | 1;

**sub\_4010D0**("%d\n", v4);

**sub\_4010D0**("%d\n", strlen(v3));

return 0;

}

//Gridra

undefined4 \_\_cdecl FUN\_00401000(undefined4 param\_1,int param\_2)

{

char \*pcVar1;

char cVar2;

char \*pcVar3;

char \*pcVar4;

char \*pcVar5;

FID\_conflict:**\_atoi**(\*(char \*\*)(param\_2 + 4));

FUN\_004010d0((int)&DAT\_00412160);

pcVar5 = \*(char \*\*)(param\_2 + 4);

pcVar4 = "12345";

pcVar3 = pcVar5;

do {

cVar2 = \*pcVar4;

pcVar4 = pcVar4 + 1;

\*pcVar3 = cVar2;

pcVar3 = pcVar3 + 1;

} while (cVar2 != '\0');

FUN\_004010d0((int)&DAT\_0041216c);

pcVar3 = "12345";

pcVar4 = pcVar5;

do {

if (((\*pcVar3 != \*pcVar4) || (\*pcVar3 == '\0')) || (pcVar1 = pcVar3 + 1, \*pcVar1 != pcVar4[1]))

break;

pcVar3 = pcVar3 + 2;

pcVar4 = pcVar4 + 2;

} while (\*pcVar1 != '\0');

FUN\_004010d0((int)&DAT\_00412170);

do {

cVar2 = \*pcVar5;

pcVar5 = pcVar5 + 1;

} while (cVar2 != '\0');

FUN\_004010d0((int)&DAT\_00412174);

return 0;

}

## 内联函数VS反编译器（2）

## 受限指针的使用（1）

C99标准中提出。

编程人员可以通过受限指针向编译器声明：被该关键字（restrict）标记的指针是操作相关内存的唯一指针，没有其他指针重复指向此内存区域。

int\* \_\_restrict x

**//size**

mov QWORD PTR [rsp+48], 18

lea rax, QWORD PTR sum\_product$[rsp]

**//sum\_product**

mov QWORD PTR [rsp+40], rax

lea rax, QWORD PTR update$[rsp]

**//update**

mov QWORD PTR [rsp+32], rax

**//product**

lea r9, QWORD PTR product$[rsp]

**//sum**

lea r8, QWORD PTR sum$[rsp]

**//y**

lea rdx, QWORD PTR y$[rsp]

**//x**

lea **rcx**, QWORD PTR x$[rsp]

call ?f@@YAXPEAH00000\_K@Z

## 受限指针的使用（2）

## 变长参数函数实现机制（1）

## 变长参数函数实现机制（2）

## 经典程序剖析：字符串修剪（1）

char c;

* + 1. ② ⑤ ④⑦

for (size\_t str\_len = strlen(s); str\_len >0; str\_len--)

{

* + 1. ⑥

}

//微软C语言函数源码

C:\Program Files (x86)\Windows Kits\10\Source\10.0.19041.0\ucrt

//The GNU C Library (glibc)

[**https://www.gnu.org/software/libc/**](https://www.gnu.org/software/libc/)

**//openbsd**

[**https://github.com/openbsd/src/tree/master/lib/libc**](https://github.com/openbsd/src/tree/master/lib/libc)

## 经典程序剖析：字符串修剪（2）

## 经典程序剖析：字符大小写转换

?toupper\_1@@YADD@Z PROC

mov cl, BYTE PTR \_c$[esp-4]

//char a1 = 'a';

lea eax, DWORD PTR [ecx-97]

//char a5 = 'z' - 'a';

cmp al, 25

ja SHORT $LN2@toupper\_1

//char a4 = 'A' - 'a';

lea eax, DWORD PTR [ecx-32]

ret 0

$LN2@toupper\_1:

mov al, cl

ret 0

?toupper\_1@@YADD@Z ENDP

**A**

**01 0 00001**

**a**

**01 1 00001**

**Z**

**01 0 11010**

**z**

**01 1 11010**