# X86/X64软件逆向分析入门

## 课前准备&工具安装与配置

1. 课程回顾：《PC微信探秘》
2. Visual Studio 2019开发套件：<http://www.microsoft.com>
3. x64dbg：动态调试工具：<https://x64dbg.com>
4. IDA：静态分析工具
   1. <https://www.hex-rays.com/>
   2. <https://github.com/AngelKitty/IDA7.0>
5. 课程源代码：<https://github.com/zmrbak/ReverseAnalysis>
6. 如何下载课程源代码

## 软件中的逆向分析与非官方功能扩展（1）

1. PPT来源于本人在成都理工大学的一次公开讲座
2. 《PC微信探秘》

## 软件中的逆向分析与非官方功能扩展（2）

## CPU指令集&最简单的函数

1. 指令码
2. 机器码
3. 汇编语言
4. CPU寄存器
5. 机器码和编程语言的区别
6. 指令集架构
7. 一个最简单的函数，了解一下返回值的处理。

## 剖析C语言中的经典程序-Hello World

1. 把C代码转换成汇编语言代码
2. 汇编语言的两种主流语体：Intel、AT&T
3. X64寄存器、X86寄存器、16位寄存器

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **X64** | **RAX** | | | | | | | |
| **X86** |  | | | | **EAX** | | | |
| **16位** |  | | | | | | **AX** | |
|  |  | | | | | | **AH** | **AL** |

RCX、RDX、R8、R9

## 函数的序言、函数的尾声

1. 函数序言

01172400 55 **push ebp**

01172401 8B EC  **mov ebp,esp /rsp**

01172403 81 EC C0 00 00 00 sub esp,0C0h （为局部变量申请存储空间）

1. 函数尾声

0117243A 8B E5 mov esp,ebp

0117243C 5D  **pop ebp**

0117243D C3  **ret**

1. 递归调用的问题

int f()

{

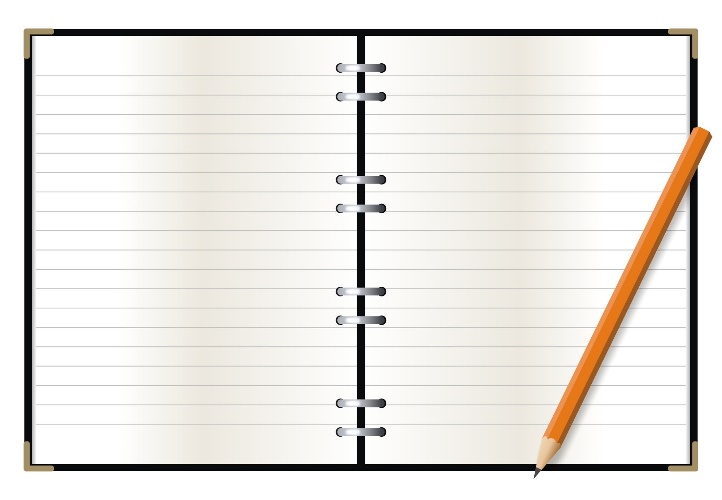
f();

}

## 逆向生长的堆栈

1. 栈是CPU寄存器中的某个指针指向的一片内存区域。
2. PUSH，减法操作
3. POP，加法操作
4. 栈顶：ESP/RSP
5. 栈底：EBP
6. 为什么栈会逆向生长

|  |  |  |
| --- | --- | --- |
| 低地址 高地址 | | |
| 堆的起点 | **-----> <-----** | 栈的起点 |



## 栈的用途

1. 保存函数结束时的返回地址

|  |  |  |  |
| --- | --- | --- | --- |
| 010B1025 | 33C0 | xor eax,eax |  |

00AFF838 010B1025 返回到 l008a.010B1025 自 l008a.010B1000

00AFF83C 00AFF884

00AFF83C 00AFF884

00AFF840 010B1269 返回到 l008a.010B1269 自 l008a.010B1010

1. 传递参数
2. 存储局部变量

$-10 00000001 //第1个局部变变量

$-C 00000002 //第2个局部变变量

$-8 00000000 //第3个局部变变量

$-4 008FFD4C //EBP

$ ==> 00251055 //返回地址：到 l008a.00251055 自 l008a.00251000

$+4 00000001 //arg1

$+8 00000002 //arg2

1. Windows的SEH结构化异常处理
2. 缓冲区溢出保护

ESP: 00F5FD08-00F5FAB0

EBP: 00F5FD14

## 栈的噪音

CALL等价于：PUSH+JMP

RET 等价于：POP+JMP

堆栈的溢出

Release版本的优化

堆栈的噪音：噪音数据、脏数据

00DAFB24 01381058 返回到 l009.01381058 自 l009.01381000

00DAFB28 00DAFB70

## 函数与参数

## 整数型数据的指针传递（1）

指针：描述某个内存地址的数据

X86指针：32位，4个字节

X64指针：64位，8个字节

Void\*

多个返回值的传递处理，scanf()

编译器在类型检查阶段——检查指针数据类型

局部变量：函数内部可以访问，函数外部应该访问不到。

本地变量：ebp+偏移

外部传来的参数：ebp+偏移

**X86栈结构**

$-C 00000001 //第1个局部变变量

$-8 00000002 //第2个局部变变量

$-4 00000000 //第3个局部变变量

$==> 008FFD4C //EBP

$+4 00251055 //返回地址：到 l008a.00251055 自 l008a.00251000

$+8 00000001 //arg1

$+C 00000002 //arg2

**lea : load effective address**

lea eax, DWORD PTR \_x$[ebp]

lea eax, [ebp+\_x]

lea eax, [ebp-4]

lea eax, [local.1]

## 整数型数据的指针传递（2）

## 整数型数据的指针传递（3）

局部变量VS全局变量

全局变量

未初始化

已初始化

## 整数型数据的指针传递（4）

函数返回值

cmp eax, 1

jne SHORT $LN2@main

**jne**: Jump Not Equal

jmp SHORT $LN3@main

cmp eax, 1

jne SHORT $LN2@main

cmp->jcc: Jump condition code

cmp：本质是减法操作

1-1=0；ZF置1

JNE：ZF

JNZ:

SUB—CMP

HACK

补丁

**注意：**

无论是x86还是x64，函数的返回值都放在eax中

EAX 的值不等于 RAX

Int 是32位，无论在x86还是x64都是

64位寄存器，以R开头

32位寄存器，以E开头

## 函数如何获取参数？

调用方：calller

被调用方：callee

X64

sub rsp, 40

Shadow Space：阴影空间

1. 可以避免浪费寄存器资源
2. 便于调试器在程序中断时找到函数参数。

## 如何接收函数的返回值？

1. int 型函数的返回值
2. float型函数的返回值
3. void 型函数的返回值
4. 函数返回值不被使用的情况
5. 结构体型函数的返回值

X86：

EAX：返回调用结果

Byte、char：EAX，低8位->AL

Float：FPU->ST0

exit(main\_result);

int const main\_result = invoke\_main();

static int \_\_cdecl invoke\_main()

return main(\_\_argc, \_\_argv, \_get\_initial\_narrow\_environment());

fld1 // FLD类似于   PUSH指令

fstp dword ptr [a] // FSTP类似于  POP指令

**cvtss2sd**XMM,XMM/m32

把源存储器低32位1个单精度浮点数变成1个双精度浮点数,结果送入目的寄存器的低64位,高64位不变.

cvtss2sd xmm0,dword ptr [a]

//在俩个xmm指令之间或者内存位置和xmm寄存器之间复制浮点数

movsd mmword ptr [esp],xmm0

如果函数返回的是大型结构的数据，那么由调用方函数（Caller）负责分配空间，给结构体分配指针，再把指针作为第一个参数传递给被调用方函数。然后由被调用方函数填充数据。

## 用指针作函数的参数

**push OFFSET ?multi\_result@@3HA ; multi\_result**

**push OFFSET ?sum\_result@@3HA ; sum\_result**

push 456 ; 000001c8H

push 123 ; 0000007bH

call ?f@@YAXHHPAH0@Z ; f

add esp, 16 ; 00000010H

; Line 19

**mov eax, DWORD PTR ?multi\_result@@3HA ; multi\_result**

push eax

**mov ecx, DWORD PTR ?sum\_result@@3HA ; sum\_result**

push ecx

push OFFSET $SG5562

call \_printf

add esp, 12 ; 0000000cH

;

mov eax, DWORD PTR \_x$[ebp]

add eax, DWORD PTR \_y$[ebp]

mov ecx, DWORD PTR \_sum$[ebp]

**mov DWORD PTR [ecx], eax**

?f@@YAXHHPAH0@Z PROC ; f

; File C:\Users\libit\source\repos\L017\L017\L017.cpp

; Line 7

push ebp

mov ebp, esp

; Line 8

mov eax, DWORD PTR \_x$[ebp]

add eax, DWORD PTR \_y$[ebp]

mov ecx, DWORD PTR \_sum$[ebp]

mov DWORD PTR [ecx], eax

; Line 9

mov edx, DWORD PTR \_x$[ebp]

imul edx, DWORD PTR \_y$[ebp]

mov eax, DWORD PTR \_multi$[ebp]

mov DWORD PTR [eax], edx

; Line 10

pop ebp

ret 0

?f@@YAXHHPAH0@Z ENDP ; f

**//优化**

?f@@YAXHHPAH0@Z PROC ; f

; File C:\Users\libit\source\repos\L017\L017\L017.cpp

; Line 8

mov edx, DWORD PTR \_y$[esp-4]

mov eax, DWORD PTR \_sum$[esp-4]

push esi

mov esi, DWORD PTR \_x$[esp]

lea ecx, DWORD PTR [esi+edx] //x+y

; Line 9

imul esi, edx //x\*y

mov DWORD PTR [eax], ecx //为sum赋值

mov eax, DWORD PTR \_multi$[esp]

mov DWORD PTR [eax], esi //multi赋值

pop esi

; Line 10

ret 0

?f@@YAXHHPAH0@Z ENDP ; f

//x64未优化

main PROC

; File C:\Users\libit\source\repos\L017\L017\L017.cpp

; Line 16

$LN3:

sub rsp, 40 ; 00000028H

; Line 17

lea rcx, OFFSET FLAT:$SG5099

call printf

; Line 18

lea r9, OFFSET FLAT:?multi\_result@@3HA ; multi\_result

lea r8, OFFSET FLAT:?sum\_result@@3HA ; sum\_result

mov edx, 456 ; 000001c8H

mov ecx, 123 ; 0000007bH

call ?f@@YAXHHPEAH0@Z ; f

; Line 19

mov r8d, DWORD PTR ?multi\_result@@3HA ; multi\_result

mov edx, DWORD PTR ?sum\_result@@3HA ; sum\_result

lea rcx, OFFSET FLAT:$SG5100

call printf

; Line 20

xor eax, eax

add rsp, 40 ; 00000028H

ret 0

main ENDP

//取地址

**lea** eax, DWORD PTR \_multi\_result$[ebp]

//取值

**mov** edx, DWORD PTR \_multi\_result$[ebp]

lea ecx, DWORD PTR [esi+edx]

==》

Mov ecx, esi+edi

借助指针，函数可以返回位于任意地址的任意值。

C++中的引用： Reference

C中的指针: Pointer

## 被废弃的GOTO语句

## 条件跳转指令(1)

数值比较

**有符号比较**

**JLE**/JNE/**JGE**

Jump **Less** Equal

Jump Not Equal

Jump **Greater** Equal

**无符号比较：**

**JBE**/JNE/**JAE**

Jump **Below** Equal/Not Above

Jump Not Equal /Not Zero

Jump **Above** Equal

可以从条件跳转指令来判断CMP比较的变量的数据类型。

## 条件跳转指令(2)

计算绝对值

NEG是汇编指令中的求补指令，NEG[指令](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4/3225201)对操作数执行求补运算：**用零减去操作数**，然后结果返回操作数。求补运算也可以表达成：将操作数按位取反后加1。

neg eax

test eax,eax 基本上和 And eax,eax 是一样的，不同的是test 不改变eax的结果，只是**改变FLAG寄存器**的状态，也就是改变进位标志，零标志，溢出标志等 等。

test eax, eax

如果符号位S不为1，就跳转。

//符号位为1：负数

//符号位为0: 正数

jns SHORT $LN3@myAbs

## 条件跳转指令(3)

条件运算

表达式？表达式：表达式

*//*不等于*0*时传送

c**mov**ne eax, ecx

c**mov**e rax, rdx

je 表示等于就跳转，jne是不等于就跳转，完全相反的意思。

je = jmp equal    jne = jmp not equal 。

jz 表示当zf =1 时跳转，即结果为0跳转。

jnz 即 zf=0 时跳转，即结果不为0 跳转。

jz是零标志位置位（1－－即为0）时跳转。

jnz与上面相反，是当零标志位置0（0－－即不为0）时跳转。

je SHORT $LN3@f

## 条件跳转指令(4)

比较最大值和最小值

*//*有符号小于时传送

c**mov**l eax, DWORD PTR \_a$[esp-4]

*//*有符号大于时传送

c**mov**g eax, DWORD PTR \_a$[esp-4]

总结：

CMP 寄存器,寄存器/值

JCC TRUE:

FALSE:

….

JMP EXIT:

TRUE:

…

EXIT:

## SWITCH CASE语句(1)

**Case语句比较少的情况**

## SWITCH CASE语句(2)

**Case语句比较多的情况**

//Move with Sign-Extension

movsxd rax, DWORD PTR tv64[rsp]

总结：

当case语句比较多的时候，采用查表的方式进行跳转

首先判断是不是超出表：default语句

接下来，查表并跳转执行

执行完毕后，跳转到返回

## SWITCH CASE语句(3)

多对一的Case语句

减1指令

dec eax

## SWITCH CASE语句(4)

Fall-through

落空、滑梯

## 一个简单的FOR循环

X86:

LOOP

检测ECX寄存器的值是否为0，如果它不为0，则将ECX递减，并跳转到指定的标签处，继续执行。

编译器几乎不生成LOOP指令。

LOOP手工编写的汇编代码。

C/C++循环：

For、while、do/while

for(初始化；循环条件判断；循环控制)

{

执行循环体；

}

## 复制内存的For循环

SIMD

ECX：32

CX：·16

CL：8

## 计算C语言字符串的长度

字符串结束符 ‘\0’

汇编语言数据传送指令MOV的变体。带符号扩展，并传送。

mov**s**x edx, BYTE PTR [ecx]

负数—>高位填充1

正数—>高位填充0

S: sign

X: Extend

汇编语言数据传送指令MOV的变体。无符号扩展，并传送。

mov**z**x edx, BYTE PTR [rax]

负数—>高位填充0

正数—>高位填充0

Z: Zero

X: Extend

RDX

EDX

DX

dl

## 插播：使用VC++ 编译汇编源码文件

1. 创建一个空项目
2. 生成依赖项：
   1. 生成自定义
   2. 勾选MASM
3. 设置项目属性：
   1. 连接器\输入\附加依赖项:添加ucrtd.lib
   2. 链接器\高级\入口点：main
4. 汇编代码
   1. 项目类型：Microsoft Macro Assembler
5. 汇编高亮插件
   1. AsmHighlighter

需要根据自己的编译选项来确定链接哪些库

Release DLLs (/MD ): msvcrt.lib vcruntime.lib ucrt.lib

Debug DLLs (/MDd): msvcrtd.lib vcruntimed.lib ucrtd.lib

Release Static (/MT ): libcmt.lib libvcruntime.lib libucrt.lib

Debug Static (/MTd): libcmtd.lib libvcruntimed.lib libucrtd.lib

## 数学计算指令的替换（1）

乘法

除法

LEA

ADD SUB

SHL是一个汇编指令，作用是逻辑左移指令，将目的操作数顺序左移1位或CL寄存器中指定的位数。左移一位时，操作数的最高位移入进位标志位CF，最低位补零。

shl eax, 3

1B<<1

10B==2

1B<<2

100B==4

1B<<3

1000B<8

return a \* 8;

mov eax, DWORD PTR \_a$[ebp]

shl eax, 3

mov eax, DWORD PTR a$[rsp]

shl eax, 3

lea eax, DWORD PTR [rcx\*8]

IMUL是一个计算机函数，功能是将被乘数与乘数均作为有符号数， 它按照符号扩展方式扩展到目标操作数格式的长度。

**i**mul 有符号乘法，将被乘数与乘数均作为有符号数。

mul 无符号乘法，将被乘数及乘数均作为无符号数。

可以有三个操作数：imul eax,**eax**,**0Ch**

第3操作数是乘数，

第2操作数是被乘数，

运算结果放入第1操作数。

imul eax, DWORD PTR \_a$[ebp], 7

imul eax, DWORD PTR a$[rsp], 7

imul eax, ecx, 7

**imul eax, DWORD PTR \_a$[ebp], 28**

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

lea eax, DWORD PTR [ecx\*8] ;//eax=a\*8

sub eax, ecx ;//eax=a\*8-a=7\*a

shl eax, 2 ;//eax=eax\*4=7\*a\*4=28\*a

imul eax, DWORD PTR a$[rsp], 28

imul eax, ecx, 28

imul eax, DWORD PTR \_a$[ebp], 17

mov eax, DWORD PTR \_a$[esp-4]

shl eax, 4 //a\*2^4=a\*16

add eax, DWORD PTR \_a$[esp-4] //a\*16+a=a\*17

imul eax, DWORD PTR a$[rsp], 17

imul eax, ecx, 17

双精度左移

SHLD（双精度左移）指令将目的操作数向左移动指定位数。移动形成的空位由源操作数的高位填充。源操作数不变，但是符号标志位、零标志位、辅助进位标志位、奇偶标志位和进位标志位会受影响：

shld edx, eax, 3

SBB：带借位减法，

格式：SBB DST,SRC,

执行的操作：（DST)←(DST)-(SRC)-**CF**,其中CF为进位的值。

SBB Compents ：他们组成了Application .

他们以何种方式组合是由SLEE来确定的。

sbb edx, DWORD PTR \_a$[esp]

ADC指令的引入主要是为了实现多字节的运算。当进行32位以上运算时，要求低位字节相加，而高位字节再相加时就要考虑低位相加的进位，即CF，这时就要用到ADC指令。

adc edx, DWORD PTR \_a$[esp]

cdq的作用无非就是将一个32位有符合数扩展为64位有符合数，数据能表示的数不变

CDQ—Convert Double to Quad (386+)，该指令把edx扩展为eax的高位，也就是说变为64位。

该指令先把edx的每一位置成eax的最高位(符号位)，再把edx扩展为eax的高位。

cdq

edx:eax

SAR是算数右移指令

而 SHR是逻辑右移指令（shift logical right）。

SAR右移时保留操作数的符号，即用符号位来补足，而SHR右移时总是用0来补足。

sar eax, 1

//带符号除法指令

双字操作:64位被除数在EDX,EAX中，其中EDX为高位双字，32位除数为源操作数，结果的32位商在EAX中，32位余数在EDX中，表示为

　　(EAX)<-(EDX,EAX)/(SRC) 的商

　　(EDX)<-(EDX,EAX)/(SRC) 的余数。

idiv ecx

默认被乘数eax, \_a是乘数，结果默认在edx（high）eax中

imul DWORD PTR \_a$[esp-4]

## 数学计算指令的替换（2）

## FPU：浮点数的乘法和除法

FPU是CPU的一个组件，是一个专门处理浮点数的运算单元。

80486之前，FPU叫做协处理器（辅助处理器），是一个独立的芯片，需要在主板上单独安装FPU，而且还得单独购买。

80486DX之后，CPU集成了FPU的功能。

FWAIT：让CPU等待FPU运算结束。

FPU：自带一个由8个80位寄存器构成的循环栈，存储IEEE 754格式的浮点数。

**ST(0)---ST(7)**

IEEE 754：符号位+尾数（有效数字、小数位）+指数

32位：单精度浮点数，float

64位：双精度浮点数，double

Int，32位

MOVSB, MOVSW, MOVSD

最后一个字母也就表示的是我们每次移动的数据字节大小。第一个MOVSB是每次一个字节，其次是一个字、接着一个双字。

我们从它的'B','W','D'就可以看出来。

B代表Byte，

W代表WORD,

D代表DWORD。

mov**s**d 是符号扩展

mov**z**d 是0扩展

什么意思呢，比如 一个8位的数 是 1

movsd eax, 1 //因为1是正数，所以 8位变32位的时候，前面的24位全部用1的**符号位**来填充，1的符号位为0

movsd eax,-1 //就是用1来填充了，因为-1的符号为1

在计算机中，整数的符号位0表示正数，1表示负数

mov**s**d xmm0, QWORD PTR \_\_real@400b333333333333

这是一个浮点指令，也就是用协处理器执行的指令。

FST 指令将 ST(0) 寄存器中的值复制到目标操作数，目标操作数可以是内存位置或 FPU 寄存器堆栈中的另一个寄存器。将值存储到内存时，值会转换成单精度或双精度实数格式。

FSTP 指令先执行同 FST 指令相同的操作，然后弹出寄存器堆栈。为了弹出寄存器堆栈，处理器将 ST(0) 寄存器标记为空，并使堆栈指针 (TOP) 递增 1。FSTP 指令还可以按扩展的实数格式在内存中存储值。

fstp QWORD PTR [esp]

FLD是Intel的指令集协处理器的汇编指令，FLD 指令用于把浮点数字传送入和传送出FPU寄存器。

fld QWORD PTR tv70[ebp]

mov**ss**和mov**aps**：

movss是将一个单精度数传输到xmm寄存器的低32位，而movaps则是一次性向寄存器中写

入四个单精度数。

movaps xmm1, xmm0

movb #完成1个字节的复制

movw #完成2个字节的复制

movl #完成4个字节的复制

movq #完成8个字节的复制

movq rdx, xmm1

## FPU：浮点数的参数传递

fstp qword ptr ss:[esp], st0

fstp QWORD PTR [esp]

## FPU：浮点数大小比较

## FPU：为何FPU指令用的这么少？

x87： FPU指令

//浮点比较出栈

fcomp QWORD PTR \_a$[ebp]

fnstsw ax， c3 / c2 / c0 分别放到AH的0 / 2 / 6位，

test ah, 5，只看c0 / c2，c0 / c2 都为1 或都为0， PF =1 , JP跳转

fnstsw ax

## 数组：数组的存储结构

在内存中，数组就是**按次序排列**的、**相同数据类型**的一组数据。

**i\*2**

shl ecx, 1

**数组**

mov DWORD PTR \_a$[**ebp+edx\*4**], ecx

mov edx, DWORD PTR \_a$[**ebp+ecx\*4**]

**i++;**

mov eax, DWORD PTR \_i$[ebp]

add eax, 1

mov DWORD PTR \_i$[ebp], eax

inc esi

**一次将4个整数放到堆栈中**

movaps xmm0, XMMWORD PTR \_\_xmm@0000000e0000000c0000000a00000008

movups XMMWORD PTR \_a$[esp+100], xmm0

**开始**

mov eax, DWORD PTR \_\_\_security\_cookie

xor **eax**, esp

mov DWORD PTR \_\_$ArrayPad$[esp+84], **eax**

mov **ecx**, DWORD PTR \_\_$ArrayPad$[esp+88]

xor **ecx**, esp

call @\_\_security\_check\_cookie@4

**加密/解密**

1. xor，加密
2. xor, 解密

## 数组：缓冲区溢出

编译器借助索引index，以array[index]的形式表示数组。

1. **读取**数组边界以外的内容
2. 向数组边界以外**写入**内容

EAX:5EBE70ED

EBP:012FF8FC

结果：5F918811

|  |  |  |  |
| --- | --- | --- | --- |
| 00E51006 | A1 0490E600 | mov eax,dword ptr ds:[E69004] |  |

EAX： **949F4F2D**

EBP： 004FF7C8

结果：94D0B8E5

94D0B8E5

ECX ：**949F4F2D**

**缓冲区溢出攻击**

试想一种情况：使用字符串替代int数组，可以构造一个超长的字符串，把字符串传递给程序；因为函数不会去检查字符串的长度，而是直接赋值给较短的缓冲区；因此，你就可以强制这个程序跳转到其他程序的地址去执行。

## 数组：缓冲区溢出保护

/GS 启用安全检查

/RTCs 堆栈帧运行时检查

不可用

//int b[size];

//可用

int\* a= (int\*)alloca(size\*4);

a[i] = i \* 2;

## 数组：字符串指针

## 数组：数组的溢出和溢出保护

## 数组：多维数组

多维数组和一维数组在本质上是相同的。

由于计算机内存是连续的线性空间，因此多维数组可以与一维数组直接对应。

CL -> CX->ECX->RCX

**mov byte ptr ds:[edx+eax\*4+0x3898B0]**, 0x0

**以行优先的顺序填充数据**

for (size\_t i = 0; i < 4; i++)

{

a[1][i] = i;

}

mov byte ptr ds:[edx+eax\*1+0x3898B0], cl

**以列优先的顺序填充数据**

for (size\_t i = 0; i < 3; i++)

{

a[i][2] = i;

}

**mov byte ptr ds:[edx+eax\*4+0x8998B0]**, cl

以一维数组的方式访问二维数组

|  |  |  |
| --- | --- | --- |
| getItem3 | getItem2 | getItem1 |
| mov eax, DWORD PTR \_a$[ebp]  mov ecx, DWORD PTR \_array$[ebp]  lea edx, DWORD PTR [ecx+eax\*4]  mov eax, DWORD PTR \_b$[ebp]  mov al, BYTE PTR [edx+eax] | mov eax, DWORD PTR \_a$[ebp]  mov ecx, DWORD PTR \_b$[ebp]  lea edx, DWORD PTR [ecx+eax\*4]  mov eax, DWORD PTR \_array$[ebp]  mov al, BYTE PTR [eax+edx] | mov eax, DWORD PTR \_a$[ebp]  mov ecx, DWORD PTR \_array$[ebp]  lea edx, DWORD PTR [ecx+eax\*4]  mov eax, DWORD PTR \_b$[ebp]  mov al, BYTE PTR [edx+eax] |

三维数组

imul eax, DWORD PTR \_x$[ebp], 2400

//eax=x\* 2400

// =600 \*4 \*x

// **=20\*30\*4\*x**

imul ecx, DWORD PTR \_y$[ebp], 120

//ecx=120\*y

// **=30\*4\*y**

lea edx, DWORD PTR ?a@@3PAY1BE@BO@HA[eax+ecx]

//edx=a+eax+ecx

// **a+20\*30\*4\*x+30\*4\*y**

mov eax, DWORD PTR \_z$[ebp]

//**eax=z**

mov ecx, DWORD PTR \_value$[ebp]

//**exc=value**

mov DWORD PTR [edx+eax\*4], ecx

//**\*(edx+eax\*4)=value**

//\*(a+20\*30\*4\*x+30\*4\*y + z\*4)

//\*(4\*(20\*30\*x+30\*y+z))

//**三维转换成一维**

//**20\*30\*x+30\*y+z**

显示器屏幕，显示的是一个2D空间，但是显存却是一个一位线性空间。

## 数组：二维字符串数组的封装

数组元素可以是任意类型的数据，甚至可以是结构体数据。

如果需要访问数组中特定的元素，则首先需要计算出其在内存中的地址。

## 位操作：特定位的操作

标识位使用BOOL型，是一种非常不理智的行为。

winnt.h

#define GENERIC\_READ (0x80000000L)

#define GENERIC\_WRITE (0x40000000L)

#define GENERIC\_EXECUTE (0x20000000L)

#define GENERIC\_ALL (0x10000000L)

#define DELETE (0x00010000L)

#define READ\_CONTROL (0x00020000L)

#define WRITE\_DAC (0x00040000L)

#define WRITE\_OWNER (0x00080000L)

#define SYNCHRONIZE (0x00100000L)

8：

4：

2：

1：

TEST

## 位操作：特定位的设置和清除

AND

OR

NOT

XOR

>> sar

<< shl

移位运算用途非常广泛，可对特定位取值或隔离。

## 位操作：FPU上特定位的设置和清除

FPU中存储的是浮点数

IEEE 754

|  |  |  |
| --- | --- | --- |
| 符号位（1位） | 指数（阶码，8位） | 小数（23位） |
| 31 | 30-23 | 22-0 |

-123.456

AND

OR

XOR

;**BT**(Bit **Test**):        位测试

;BTS(Bit Test and Set):    位测试并置位

;BTR(Bit Test and Reset):  位测试并复位

;BTC(Bit Test and Complement): 位测试并取反

btc eax, 31

/arch:<IA32|SSE|SSE2|AVX|AVX2|AVX512> 最小 CPU 架构要求，以下之一:

**IA32** - 不使用增强指令并将 x87 用于浮点

(按<回车键>继续)

**SSE** - 允许使用支持 SSE 的 CPU 可用的指令

**SSE2** - (默认)允许使用支持 SSE2 的 CPU 可用的指令

AVX - 允许使用支持 AVX 的 CPU 可用的指令

AVX2 - 允许使用支持 AVX2 的 CPU 可用的指令

AVX512 - 允许使用支持 AVX-512 的 CPU 可用的指令

/arch:<AVX|AVX2|AVX512> 最小 CPU 架构要求，以下之一:

AVX - 允许使用支持 AVX 的 CPU 可用的指令

(按<回车键>继续)

AVX2 - 允许使用支持 AVX2 的 CPU 可用的指令

AVX512 - 允许使用支持 AVX-512 的 CPU 可用的指令

**SSE**

Streaming **SIMD** Extensions，单指令多数据流扩展

## 位操作：位校验

CL -> CX -> ECX 🡪RCX

1<<1

1<<2

1<<3

…..

1<<31

#define \_CRT\_INTERNAL\_PRINTF\_LEGACY\_VSPRINTF\_NULL\_TERMINATION (1ULL << 0)

#define \_CRT\_INTERNAL\_PRINTF\_STANDARD\_SNPRINTF\_BEHAVIOR (1ULL << 1)

#define \_CRT\_INTERNAL\_PRINTF\_LEGACY\_WIDE\_SPECIFIERS (1ULL << 2)

#define \_CRT\_INTERNAL\_PRINTF\_LEGACY\_MSVCRT\_COMPATIBILITY (1ULL << 3)

#define \_CRT\_INTERNAL\_PRINTF\_LEGACY\_THREE\_DIGIT\_EXPONENTS (1ULL << 4)

## 线性同余法的伪随机数函数

**伪**随机数

随机数：电磁辐射、原子核裂变、掷硬币

线性同余法：乘法、加法、与运算

## 结构体：SYSTEMTIME解析

在C/C++中的结构体，是一系列数据简单的堆积而成的一种数据类型。

结构体中的数据类型，可以是同一种类型，也可以是不同的数据类型。

C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\um\minwinbase.h

typedef struct \_SYSTEMTIME {

WORD wYear;

WORD wMonth;

WORD wDayOfWeek;

WORD wDay;

WORD wHour;

WORD wMinute;

WORD wSecond;

WORD wMilliseconds;

} SYSTEMTIME, \*PSYSTEMTIME, \*LPSYSTEMTIME;

## 结构体：以数组代替结构体

## 结构体：用malloc来分配结构体的空间

在某些情况下，使用堆（Heap）来存储结构体要比栈（stack）容易一些。

Malloc（）

SYSTEMTIME **\***pst;

**pst->wHour,**

SYSTEMTIME pst;

**pst.wHour,**

## 结构体：结构体的字段封装

#pragma pack(2)

|  |  |
| --- | --- |
| f1 | f |
| mov eax, DWORD PTR \_d$[ebp]  push eax  movsx ecx, BYTE PTR \_c$[ebp]  push ecx  mov edx, DWORD PTR \_b$[ebp]  push edx  movsx eax, BYTE PTR \_a$[ebp]  push eax  push OFFSET $SG5576  call \_printf  add esp, 20 ; 00000014H | mov eax, DWORD PTR \_s1$[ebp+12]  push eax  movsx ecx, BYTE PTR \_s1$[ebp+8]  push ecx  mov edx, DWORD PTR \_s1$[ebp+4]  push edx  movsx eax, BYTE PTR \_s1$[ebp]  push eax  push OFFSET $SG5569  call \_printf  add esp, 20 ; 00000014H |

**#pragma pack( show )**  
**#pragma pack( push** [ **,** identifier ] [ **,** n ] **)**  
**#pragma pack( pop** [ **,** { identifier | n } ] **)**  
**#pragma pack(** [ n ] **)**

## 结构体：结构体的嵌套

struct in\_s

{

int a;

int b;

};

struct ou\_s1

{

char a;

int b;

**in\_s c;**

char d;

int e;

};

struct ou\_s2

{

char a;

int b;

**in\_s\* c;**

char d;

int e;

};

## 结构体：结构体中的位操作

C/C++可以精确地操作结构体中的**位域**。

CPU**ID**：获取CPU及其特征信息。

cupid

EAX=1

cupid

**EAX改变**

EBX改变

ECX改变

EDX改变

**EAX改变**

|  |  |  |
| --- | --- | --- |
| **位域** | **位数** | **特征** |
| 3:0 | 4 | Stepping |
| 7:4 | 4 | Model |
| 11:8 | 4 | Family |
| 13:12 | 2 | Processor Type |
| 15:14 | 2 | Reserved1 |
| 19:16 | 4 | Extended Model |
| 27:20 | 8 | Extended Family |
| 31:28 | 4 | Reserved2 |

CPUID\_1\_EAX\* temp = (CPUID\_1\_EAX\*)b; // 0x005afb44

lea eax, DWORD PTR **\_b$[ebp]**

mov DWORD PTR \_temp$[ebp], eax

CPUID\_1\_EAX\* temp1= (CPUID\_1\_EAX\*)&b; // 0x005afb44

; lea ecx, DWORD PTR **\_b$[ebp]**

mov DWORD PTR \_temp1$[ebp], ecx

;

CPUID\_1\_EAX\* temp2 = (CPUID\_1\_EAX\*)b[0];// 0x000306f2

mov edx, 4

imul eax, edx, 0

**mov ecx, DWORD PTR \_b$[ebp+eax]**

mov DWORD PTR \_temp2$[ebp], ecx

CPUID\_1\_EAX\* temp3= (CPUID\_1\_EAX\*)&b[0]; // 0x005afb44

mov edx, 4

imul eax, edx, 0

lea ecx, DWORD PTR **\_b$[ebp+eax]**

mov DWORD PTR \_temp3$[ebp], ecx

CPUID\_1\_EAX\* temp2 = (CPUID\_1\_EAX\*)b[1];

0x03020800

CPUID\_1\_EAX\* temp3= (CPUID\_1\_EAX\*)&b[1];

0x0113fa34

temp2 不等于 temp3

CPUID\_1\_EAX\* temp = (CPUID\_1\_EAX\*)b; // 0x005afb44

CPUID\_1\_EAX\* temp1= (CPUID\_1\_EAX\*)&b; // 0x005afb44

CPUID\_1\_EAX\* temp3= (CPUID\_1\_EAX\*)&b[0]; // 0x005afb44

## 结构体：用结构体构建浮点数

|  |  |  |
| --- | --- | --- |
| 符号位（31） | 指数部分（23-30） | 小数部分（0-22） |

## 共用体：生成一个伪随机数

|  |  |  |
| --- | --- | --- |
| 符号位（31） | 指数部分（23-30） | 小数部分（0-22） |
| 0 | 1 | 随机填写 |

线性同余

IEEE 754

## 共用体：浮点数的机器精度

机器精度：相对误差的上限，也就是FPU操作的最小值。

浮点数的小数位越多，误差越小，精度越高。

Float 精度：2^-23

Double精度：2^-52

## 函数指针与回调函数

函数指针，代表一个地址，函数代码段的起始地址，函数头的地址。

回调函数，callback

驱动程序

排序函数，qsrot()

CreateThread()

call dword ptr [p]

call \_\_RTC\_CheckEsp (0201235h)

## 64位数据的处理

mov DWORD PTR \_a$[ebp], eax

mov DWORD PTR \_a$[ebp+4], edx

mov eax, -1867788817 90abcdefH

mov edx, 305419896 ; 12345678H

return 0x1234567890ABCDEF;

//使用两个寄存器返回，edx高位，eax低位

edx, eax

mov DWORD PTR \_a$[ebp], eax

mov DWORD PTR \_a$[ebp+4], edx

//变量a

**90abcdef 12345678**

$-8 004FFAC8 90ABCDEF

$-4 004FFACC 12345678

//小端存储

004FFAC8 **90ABCDEF 12345678** 004FFB18 00B9144E

004FFAC8 **1234567890ABCDEF** 00B9144E004FFB18

004FFAC8 **CDEF 90AB 5678 1234** FB18 004F 144E 00B9

004FFAC8 **EF CD AB 90 78 56 34 12** 18 FB 4F 00 4E 14 B9 00 ïÍ«.xV4..ûO.N.¹.

a = f\_add(0x11111111FFFFFFFF,0x22222222EEEEEEEE);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 00911125 | 68 22222222 | push 22222222 |  |  |
| 0091112A | 68 EEEEEEEE | push EEEEEEEE |  |  |
| 0091112F | 68 11111111 | push 11111111 |  |  |
| 00911134 | 6A FF | push FFFFFFFF |  |  |
| 00911136 | E8 D5FEFFFF | call rev\_059.911010 |  |  |

00FDF9A8 **11111111FFFFFFFF** 22222222EEEEEEEE

00FDF9B8 **33333334EEEEEEED** 0091148E00FDFA08

00FDF9B8 **ED EE EE EE 34 33 33 33** 08 FA FD 00 8E 14 91 00 íîîî4333.úý.....

al->ax-EAX-RAX

0-FF

CL-CX-ECX-RCX

mov rax, QWORD PTR b$[rsp]

movzx ecx, **al**

mov rax, QWORD PTR a$[rsp]

shl rax, **cl**