

## 《软安》笔记&考点

考试类型：

1、名词解释

2、问答题（至少 5 道）

(1)给出一段汇编代码 大家能够理解代码意思么 以及能不能用高级语言（C、C++、JAVA、PYTHON）把它写出来

(2)PE 文件这一块的知识

复习知识点：

(1)我们讲过的概念、内容

(2)我们做过的实践操作

(3)具体看视频



## 第 1 章 反汇编与逆向分析

### 1、引例

好多终端都存在很多木马样本，360 安全技术人员得到这些样本以后他们要做逆向分析(反汇编)目的是**提取样本的特征**。

利用特征配合流量分析(工具有 wireshark)监测木马，从而抓到数据包，获取 IP 地址。

(1)TCP/IP 数据包里面包含哪些信息？

源 IP 地址，目标 IP 地址等

(2)这个 IP 真的就是美国 NSA 的 IP 地址么？

不一定，54 个跳板(欧洲、南非、亚洲)

### 2、汇编语言形式——程序运行以后，在内存里面如何表现/行为

(1)结构

if 语句、for 语句、while 语句、do while 语句、switch 语句、数组、函数、结构体

(2)所用到的工具

- OllyDbg(动态逆向——将目标代码变换为易读形式的逆向分析过程，但不是仅仅静态阅读变换之后的程序，而是在一个调试器或调试工具中加载程序，然后一边运行程序一边对程序的行为进行观察和分析)

- IDA Pro(静态逆向——不执行代码而是使用反汇编工具，把程序的二进制代码翻译成汇编语言进行手工分析或者借助工具自动化分析)

- WinDbg(内核调试)

- SoftICE

(3)领空——某一时刻，一条要执行的代码所在的内存中的地址

- 地址：778F01B8 系统的领空——在内存里面的一段地址(栈)

- 地址：00401220 应用程序的领空——所分析的应用程序在内存里面的一段地址(栈)

### 3、OD 快捷键的使用

(1)F2：在所选代码指令下断点，再按一次 F2 键则会删除断点

(2)F4：运行到选定位置。作用就是直接运行到光标所在位置处暂停

(3)F7：单步步入。功能同单步步过(F8)类似，区别是遇到 CALL 等子程序时会进入其中，进入后首先会停留在子程序的第一条指令上，遇到函数就进入函数的领空(代码空间)

(4)F8：单步步过。每按一次这个键执行一条反汇编窗口中的一条指令，遇到 CALL 等子程序不进入其代码，遇到函数不进入函数的领空(代码空间)

(5)F9：运行。按下这个键如果没有设置相应断点的话，被调试的程序将直接开始运行。

(6)CTR+F9：执行到返回。此命令在执行到一个 ret(返回指令)指令时暂停，常用于从系统领空返回到我们调试的程序领空。

(7)ALT+F9：执行到用户代码。可用于从系统领空快速返回到我们调试的程序领空。

#### 4、具体程序的逆向分析

##### (1)程序

```
#include <stdio.h>
#include <stdlib.h>
int ldf(int a,int b)
{int c;c=a+b;return c;}
int main(int argc, char *argv[])
{
    int a,b,c;
    c=ldf(a,b);
    printf("c=%d\n",c);}
```

##### (2)操作步骤

- 用 OD 打开 project2.exe，一路按 F8，一直到 project2.exe 程序的领空，到这条代码指令以后，通过字符串参考功能可以直接快速定位到我们的程序代码。

- 按 F2 下断点，可以快速运行到我们的代码。

##### (3)变量表现形式：

- 在一个函数内部，对传递过来的参数的引用是 `ebp+xxxx` 的形式，一般情况下，`ebp+4` 是返回地址，`ebp+8` 是第一个参数，`ebp+c` 是第二个参数，以此类推，第 `n` 个参数是 `ebp+4*n+4`；

- 在一个函数的内部，对局部变量的引用是 `ebp-xxx` 的形式，按照声明的顺序，第一个是 `ebp-4`，第二个是 `ebp-8`，以此类推，第 `n` 个局部变量是 `ebp-n*4`。

- 一个程序运行以后，都会在我们系统内存中申请一段栈（内存空间），就用到了两个寄存器：EBP(栈底)、ESP(栈顶)。

**注：**栈底是第一个进栈的数据，栈顶是最后一个进栈的数据。栈顶是低地址，栈底是高地址。

EBP+4	
EBP	栈底（基址）
EBP-4	存放程序的第一个局部变量，8
EBP-8	存放程序的第二个局部变量，9
EBP-C	存放程序的第三个局部变量
INT 3	中断指令
INT 3	中断指令
INT 3	中断指令
INT 3	中断指令
ESP	栈顶（栈顶地址）

## 5、常见的汇编指令

```
{
00B017B0  push      ebp
00B017B1  mov       ebp,esp
00B017B3  sub       esp,0C0h
00B017B9  push      ebx
00B017BA  push      esi
00B017BB  push      edi
00B017BC  lea       edi,[ebp-0C0h]
00B017C2  mov       ecx,30h
00B017C7  mov       eax,0CCCCCCCCh
00B017CC  rep stos  dword ptr es:[edi]
    printf("Hello World!\n");
00B017CE  push      http:offset string "Hello2World!\n" (0B06B30h)
00B017D3  call      _printf (0B01320h)
00B017D8  add       esp,4
    return;
}
00B017DB  xor       eax,eax
00B017DD  pop       edi
00B017DE  pop       esi
00B017DF  pop       ebx
```

### (1)add

加法指令，第一个是目标操作数，第二个是源操作数，格式为：目标操作数+源操作数

### (2)sub

减法指令，格式同 add

### (3)call

调用函数，一般函数的参数放在寄存器中

### (4)ret

跳转会调用函数的地方。对应于 call，返回到对应的 call 调用的下一条指令，若有返回值，则放入 eax 中

### (5)push

把一个 32 位的操作数压入堆栈中，这个操作在 32 位机中会使得 esp 被减 4（字节），esp 通常是指向栈顶的，压入堆栈的数据越多，esp 也就越来越小

### (6)pop

与 push 相反，esp 每次加 4（字节），一个数据出栈。pop 的参数一般是一个寄存器，栈顶的数据被弹出到这个寄存器中

### (7)mov

数据传送。第一个参数是目的操作数，第二个参数是源操作数，就是把源操作数拷贝一份到目的操作数

### (8)xor

异或指令，通常用来实现清零功能

### (9)jmp

无条件跳转指令，对应于大量的条件跳转指令

### (10)jg/jl/je

条件跳转，大于/小于/相等时成立，通常条件跳转前会有一条比较指令用于设置标志位

### (11)jge/jle

大于/小于等于时跳转

### (12)cmp

比较大小指令，结果用来设置标志位

## 6、汇编语言转换成高级语言(以 C 语言为例)

### (1)if 语句

```
#include<stdio.h>
main()
{
    int a,b;
    int c;
    scanf("%d,%d",&a,&b);
    if(a>b) ← JG
    {c=a+b;}
    else ← JLE
    {c=a-b;}
    printf("c=%d",c);
}
```

### if 语句的反汇编代码形式(部分):

0040103D	.	8B55 FC	MOV EDX,DWORD PTR SS:[EBP-4]	a
00401040	.	3B55 F8	CMP EDX,DWORD PTR SS:[EBP-8]	a与b比较
00401043	.	7E 0B	JLE SHORT ldf.00401050	满足JLE即a<=b就跳转
00401045	.	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]	否则, 相加ADD
00401048	.	0345 F8	ADD EAX,DWORD PTR SS:[EBP-8]	
0040104B	.	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX	
0040104E	.	EB 09	JMP SHORT ldf.00401059	

### (2)for 语句(while 与之类似)

00401044	.	C745 E4 00000	MOV DWORD PTR SS:[EBP-1C],0	对i赋值
0040104B	.	EB 09	JMP SHORT array1.00401056	
0040104D	>.	8B45 E4	/MOV EAX,DWORD PTR SS:[EBP-1C]	
00401050	.	83C0 01	ADD EAX,1	i += 1
00401053	.	8945 E4	MOV DWORD PTR SS:[EBP-1C],EAX	
00401056	>.	837D E4 02	CMP DWORD PTR SS:[EBP-1C],2	i与2比较
0040105A	.	7D 31	JGE SHORT array1.0040108D	若i>=2, 则跳转到0040108D
0040105C	.	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
0040105F	.	8B55 E4	MOV EDX,DWORD PTR SS:[EBP-1C]	
00401062	.	8B448D F8	MOV EAX,DWORD PTR SS:[EBP+ECX*4-8]	
00401066	.	0FAF4495 F0	IMUL EAX,DWORD PTR SS:[EBP+EDX*4-10]	
0040106B	.	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
0040106E	.	89448D E8	MOV DWORD PTR SS:[EBP+ECX*4-18],EAX	
00401072	.	8B55 F4	MOV EDX,DWORD PTR SS:[EBP-1C]	
00401075	.	8B4495 E8	MOV EAX,DWORD PTR SS:[EBP+EDX*4-18]	
00401079	.	50	PUSH EAX	
0040107A	.	8B4D E4	MOV ECX,DWORD PTR SS:[EBP-1C]	
0040107D	.	51	PUSH ECX	
0040107E	.	68 802E4200	PUSH OFFSET array1.??_C@_0P@BAJC@array_>	
00401083	.	E8 28C60000	CALL array1.printf	
00401088	.	83C4 0C	ADD ESP,0C	
0040108B	.	EB C0	JMP SHORT array1.0040104D	
0040108D	>.	33C0	XOR EAX,EAX	return 0

### (3)数组

```
#include<stdio.h>
int main()
{
    int array_1[2],array_2[2],array_3[2];
    array_1={3,4};
    array_2={8,9};
    int i;
    for(i=0;i<2;i++)
    {
        array_3[i]=array_1[i]*array_2[i];
        printf("array_3[%d]=%d\n",i,array_3[i]);
    }
    return 0;
}
```

数组的反汇编代码形式(部分):

00401028		C745 F8 03000>MOV DWORD PTR SS:[EBP-8],3	
0040102F		C745 FC 04000>MOV DWORD PTR SS:[EBP-4],4	
00401036		C745 F0 08000>MOV DWORD PTR SS:[EBP-10],8	
0040103D		C745 F4 09000>MOV DWORD PTR SS:[EBP-C],9	
00401044		C745 E4 00000>MOV DWORD PTR SS:[EBP-1C],0	
0040104B		EB 09 JMP SHORT array1.00401056	
0040104D	>	8B45 E4 /MOV EAX,DWORD PTR SS:[EBP-1C]	
00401050		83C0 01  ADD EAX,1	
00401053		8945 E4  MOV DWORD PTR SS:[EBP-1C],EAX	
00401056	>	837D E4 02 CMP DWORD PTR SS:[EBP-1C],2	数组1
0040105A		7D 31  JGE SHORT array1.0040108D	
0040105C		8B4D E4  MOV ECX,DWORD PTR SS:[EBP-1C]	
0040105F		8B55 E4  MOV EDX,DWORD PTR SS:[EBP-1C]	
00401062		8B448D F8  MOV EAX,DWORD PTR SS:[EBP+ECX*4-8]	
00401066		0FAF4495 F0  IMUL EAX,DWORD PTR SS:[EBP+EDX*4-10]	数组2
0040106B		8B4D E4  MOV ECX,DWORD PTR SS:[EBP-1C]	数组3=数组1*数组2
0040106E		89448D E8  MOV DWORD PTR SS:[EBP+ECX*4-18],EAX	
00401072		8B55 E4  MOV EDX,DWORD PTR SS:[EBP-1C]	
00401075		8B4495 E8  MOV EAX,DWORD PTR SS:[EBP+EDX*4-18]	
00401079		50  PUSH EAX	输出数组3
0040107A		8B4D E4  MOV ECX,DWORD PTR SS:[EBP-1C]	
0040107D		51  PUSH ECX	输出i
0040107E		68 80 2E4200  PUSH OFFSET array1.??_C@@_0P@BAJC@array_>	
00401083		E8 28C60000  CALL array1.printf	
00401088		83C4 0C  ADD ESP,0C	堆栈平衡
0040108B		EB C0  JMP SHORT array1.0040104D	
0040108D	>	33C0 XOR EAX,EAX	return 0
0040108F		5F POP EDI	
00401090		5E POP ESI	
00401091		5B POP EBX	
00401092		83C4 5C ADD ESP,5C	
00401095		3BEC CMP EBP,ESP	
00401097		E8 D4C50000 CALL array1.__chkesp	



#### (4)函数

```
#include <stdio.h>
#include <stdlib.h>
int ldf(int a,int b)
{
    int c;
    c=a+b;
    return c;
}
int main(int argc, char *argv[])
{
    int a,b,c;
    c=ldf(a,b);  先把参数(变量)压入栈(PUSH), 然后使用 call 来调用函数
    printf("c=%d\n",c);
}
```

#### 函数的反汇编代码形式(部分):

```
MOV EAX,DWORD PTR SS:[EBP-8]
MOV DWORD PTR SS:[ESP+4],EAX
MOV EAX,DWORD PTR SS:[EBP-4]
MOV DWORD PTR SS:[ESP],EAX
CALL Project2.00401290

PUSH EBP-8
PUSH EBP-4
PUSH EAX
CALL 004
CALL Project2.00401290 调用函数
```



#### (5)结构体

```
#include <stdio.h>
#include <stdlib.h>
struct student
{
    long num;
    char name[20];
    char sex;
    float score;
};
void main()
{
    struct student stu_1;
    struct student *p;  //定义一个相同类型的指针
    p=&stu_1;
    stu_1.num=89101;
    strcpy(stu_1.name, "Li Lin&");
    stu_1.sex='M';
    stu_1.score=89.5;
    printf("NO. :%d\nname:  %s\nsex:%c\nscore:%f\n", stu_1.num,  stu_1.name,  stu_1.sex,
stu_1.score);
    printf("NO. :%d\nname:  %s\nsex:%c\nscore:%f\n", (*p).num,  (*p).name,  (*p).sex,
(*p).score);
}
```



结构体的反汇编代码形式(部分):

00401026	.	F3:AB	REP STOS DWORD PTR ES:[EDI]	
00401028	.	8D45 E0	LEA EAX,DWORD PTR SS:[EBP-20]	
0040102B	.	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	
0040102E	.	C745 E0 0D5C0>	MOV DWORD PTR SS:[EBP-20],15C0D	
00401035	.	68 48604200	PUSH OFFSET struct.??_C@_07HFNK@Li?5Lin?>; /li lin&	
0040103A	.	8D4D E4	LEA ECX,DWORD PTR SS:[EBP-1C]	复制的内容
0040103D	.	51	PUSH ECX	Li Lin
0040103E	.	E8 1D010000	CALL struct.strcpy	结构体复制(strcpy)
00401043	.	83C4 08	ADD ESP,8	; \strcpy
00401046	.	C645 F8 4D	MOV BYTE PTR SS:[EBP-8],4D	
0040104A	.	C745 FC 0000B>	MOV DWORD PTR SS:[EBP-4],42B30000	
00401051	.	D945 FC	FLD DWORD PTR SS:[EBP-4]	用连续局部变量来
00401054	.	83EC 08	SUB ESP,8	存放结构体元素
00401057	.	DD1C24	FSTP QWORD PTR SS:[ESP]	与变量定义顺序相反
0040105A	.	0FBE55 F8	MOVSX EDX,BYTE PTR SS:[EBP-8]	
0040105E	.	52	PUSH EDX	
0040105F	.	8D45 E4	LEA EAX,DWORD PTR SS:[EBP-1C]	
00401062	.	50	PUSH EAX	
00401063	.	8B4D E0	MOV ECX,DWORD PTR SS:[EBP-20]	
00401066	.	51	PUSH ECX	
00401067	.	68 1C604200	PUSH OFFSET struct.??_C@_0CD@MFNL@NO?4?5>;	
no. :%ld\nname: %s\nsex: %c\nscore: %f\n				第1次printf
0040106C	.	E8 6F000000	CALL struct.printf	
00401071	.	83C4 18	ADD ESP,18	
00401074	.	8B55 DC	MOV EDX,DWORD PTR SS:[EBP-24]	
00401077	.	D942 1C	FLD DWORD PTR DS:[EDX+1C]	用于把浮点数字传送入和传
0040107A	.	83EC 08	SUB ESP,8	送出FPU寄存器
0040107D	.	DD1C24	FSTP QWORD PTR SS:[ESP]	
00401080	.	8B45 DC	MOV EAX,DWORD PTR SS:[EBP-24]	
00401083	.	0FBE48 18	MOVSX ECX,BYTE PTR DS:[EAX+18]	
00401087	.	51	PUSH ECX	
00401088	.	8B55 DC	MOV EDX,DWORD PTR SS:[EBP-24]	
0040108B	.	83C2 04	ADD EDX,4	
0040108E	.	52	PUSH EDX	
0040108F	.	8B45 DC	MOV EAX,DWORD PTR SS:[EBP-24]	
00401092	.	8B08	MOV ECX,DWORD PTR DS:[EAX]	
00401094	.	51	PUSH ECX	
00401095	.	68 1C604200	PUSH OFFSET struct.??_C@_0CD@MFNL@NO?4?5>;	
no. :%ld\nname: %s\nsex: %c\nscore: %f\n				第2次printf
0040109A	.	E8 41000000	CALL struct.printf	; \printf

注: 协议都自己定义自己的数据结构, 很多是使用结构体, 比如木马, 特点是: 无进程 (hook 技术)、无端口 (http 协议, 80)、无代码、能穿墙无端口 (http 协议, 80), 木马的内部还有自己的一个协议, 比如心跳机制。

(6)switch(相当于多个 if 语句联立)

```

switch(x){
    case 0: printf("1");break;
    case 1: printf("2");break;
    case 64: printf("3"); break;
default:   printf("default");break;
}

```

JE x,0&1&64 JUMP

CALL printf

00401042	.	83C4 04	ADD ESP,4	
00401045	.	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	
00401048	.	50	PUSH EAX	
00401049	.	68 38504200	PUSH OFFSET ldf4.??_C@_02MECO@?SCFd?\$AA@ ; /%d	
0040104E	.	E8 CD000000	CALL ldf4.scanf ; \scanf	
00401053	.	83C4 08	ADD ESP,8	堆栈平衡
00401056	.	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]	
00401059	.	894D F8	MOV DWORD PTR SS:[EBP-8],ECX	
0040105C	.	837D F8 00	CMP DWORD PTR SS:[EBP-8],0	
00401060	.	74 0E	JE SHORT ldf4.00401070	case
00401062	.	837D F8 01	CMP DWORD PTR SS:[EBP-8],1	
00401066	.	74 15	JE SHORT ldf4.0040107D	
00401068	.	837D F8 64	CMP DWORD PTR SS:[EBP-8],64	
0040106C	.	74 1E	JE SHORT ldf4.0040108C	
0040106E	.	EB 29	JMP SHORT ldf4.00401099	
00401070	> .	68 34504200	PUSH OFFSET ldf4.??_C@_03IMHL@c?\$DN0?\$AA>; /c=0	
00401075	.	E8 06010000	CALL ldf4.printf ; \printf	
0040107A	.	83C4 04	ADD ESP,4	
0040107D	> .	68 30504200	PUSH OFFSET ldf4.??_C@_03CGOM@c?\$DN1?\$AA>; /c=1	
00401082	.	E8 F9000000	CALL ldf4.printf ; \printf	
00401087	.	83C4 04	ADD ESP,4	
0040108A	.	EB 1A	JMP SHORT ldf4.004010A6	
0040108C	> .	68 28504200	PUSH OFFSET ldf4.??_C@_05LNPA@a?\$DN100?\$>; /a=100	
00401091	.	E8 EA000000	CALL ldf4.printf ; \printf	
00401096	.	83C4 04	ADD ESP,4	
00401099	> .	68 1C504200	PUSH OFFSET ldf4.??_C@_07FMEP@default?\$A>; /default	
0040109E	.	E8 DD000000	CALL ldf4.printf ; \printf	
004010A3	.	83C4 04	ADD ESP,4	
004010A6	> .	33C0	XOR EAX,EAX	printf
004010A8	.	5F	POP EDI	
004010A9	.	5E	POP ESI	
004010AA	.	5B	POP EBX	
004010AB	.	83C4 48	ADD ESP,48	
004010AE	.	3BEC	CMP EBP,ESP	
004010B0	.	E8 4B010000	CALL ldf4.__chkexp	
004010B5	.	8BE5	MOV ESP,EBP	
004010B7	.	5D	POP EBP	
004010B8	\.	C3	RETN	

## 7、名词解释汇总 1

### (1)汇编语言形式:

程序运行以后,在内存里面如何表现/行为

### (2)动态逆向:

将目标代码变换为易读形式的逆向分析过程,但不是仅仅静态阅读变换之后的程序,而是在一个调试器或调试工具中加载程序,然后一边运行程序一边对程序的行为进行观察和分析(在严格控制的环境中执行恶意软件,并用系统检测实用工具记录其所有行为)

### (3)静态逆向:

不执行代码而是使用反编译、反汇编工具,把程序的二进制代码翻译成汇编语言,之后,分析者可以手工分析,也可以借助工具自动化分析(通过浏览程序代码来理解程序的行为)

### (4)内核调试:

内核是是基于硬件的第一层软件扩充,提供操作系统的最基本的功能。内核是操作系统的核心,直接与硬件交互的。调试内核的目的是使系统正常运行

### (5)领空: 某一时刻,一条要执行的代码所在的内存中的地址

- 系统的领空: 在内存里面的一段地址(栈)
- 程序的领空: 所分析的应用程序在内存里面的一段地址(栈)

### (6)程序跑飞:

系统受到某种干扰后,程序计数器 PC 的值偏离了给定的唯一变化历程,导致程序运行偏

离正常的运行路径，简单来说就是程序跑得跟设计想法完全不一样，PC 指针没有按照预定的程序变更

(7)ntdll.dll:

重要的 Windows NT 内核级文件。当 Windows 启动时，ntdll.dll 就驻留在内存中特定的写保护区域，使别的程序无法占用这个内存区域。缺少这个文件,电脑上的程序运行就没有办法正常进行,会出现强行中断

(8)堆栈平衡:

如果通过堆栈传递了参数，那么在程序执行完毕后，要平衡因参数导致的堆栈变化。即当我们使用堆栈传参的时候，在程序执行完毕后这些参数应该一起被清理掉，也就是加了几条参数，就要在堆栈中加几个地址

地址	HEX 数据	反汇编	注释
0040101E	CC	INT3	
0040101F	CC	INT3	
00401020	55	<del>CALL EBP</del>	
00401021	8BEC	MOV EBP,ESP	
00401023	83EC 48	SUB ESP,48	
00401026	53	PUSH EBX	
00401027	56	PUSH ESI	
00401028	57	PUSH EDI	
00401029	8D7D B8	LEA EDI,DWORD PTR SS:[EBP-48]	
0040102C	B9 12000000	MOV ECX,12	
00401031	B8 CCCCCCCC	MOV EAX,CCCCCCCC	
00401036	F3:AB	REP STOS DWORD PTR ES:[EDI]	
00401038	C745 FC 0000	MOV DWORD PTR SS:[EBP-4],0	
0040103F	C745 F8 0000	MOV DWORD PTR SS:[EBP-8],0	
00401046	EB 09	JMP SHORT ldf3.00401051	
00401048	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
0040104E	83C0 01	ADD EAX,1	
0040104E	8B45 F8	MOV DWORD PTR SS:[EBP-8],EAX	
00401051	837D F8 0A	CMP DWORD PTR SS:[EBP-8],0A	
00401055	7D 0B	JGE SHORT ldf3.00401062	
00401057	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]	
0040105A	034D F8	ADD ECX,DWORD PTR SS:[EBP-8]	
0040105D	894D FC	MOV DWORD PTR SS:[EBP-4],ECX	
00401060	EB E6	JMP SHORT ldf3.00401048	
00401062	8B55 FC	MOV EDX,DWORD PTR SS:[EBP-4]	
00401065	52	PUSH EDX	
00401068	68 50604200	PUSH OFFSET ldf3.??_C@_OBJ@LTA@the?5va	<<Xd> the value of sum is %d
0040106B	E8 00010000	CALL ldf3.printf	printf
00401070	83C4 08	ADD ESP,8	
00401073	5F	<del>POP EDI</del>	
00401074	5E	POP ESI	
00401075	5B	POP EBX	
00401078	83C4 48	ADD ESP,48	
00401079	3BEC	CMP EBP,ESP	
0040107B	E8 70010000	CALL ldf3.chkesp	
00401080	8B55 FC	MOV EBP,ESP	

堆栈平衡

57	PUSH EDI	
8D7D B8	LEA EDI,DWORD PTR SS:[EBP-48]	
B9 12000000	MOV ECX,12	
B8 CCCCCCCC	MOV EAX,CCCCCCCC	
F3:AB	REP STOS DWORD PTR ES:[EDI]	
68 3C504200	PUSH OFFSET ldf4.??_C@_OBJ@PTEE@please?	please input a number
E8 3E010000	CALL ldf4.printf	printf
83C4 04	ADD ESP,4	
8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	
50	PUSH EAX	
68 38504200	PUSH OFFSET ldf4.??_C@_OBJ@MECO@?CFd?SAA	%d
E8 CD000000	CALL ldf4 scanf	scanf
83C4 08	ADD ESP,8	

(9)EBP-4:

内部地址，第一个局部变量；EBP-8 是第二个局部变量所在地址

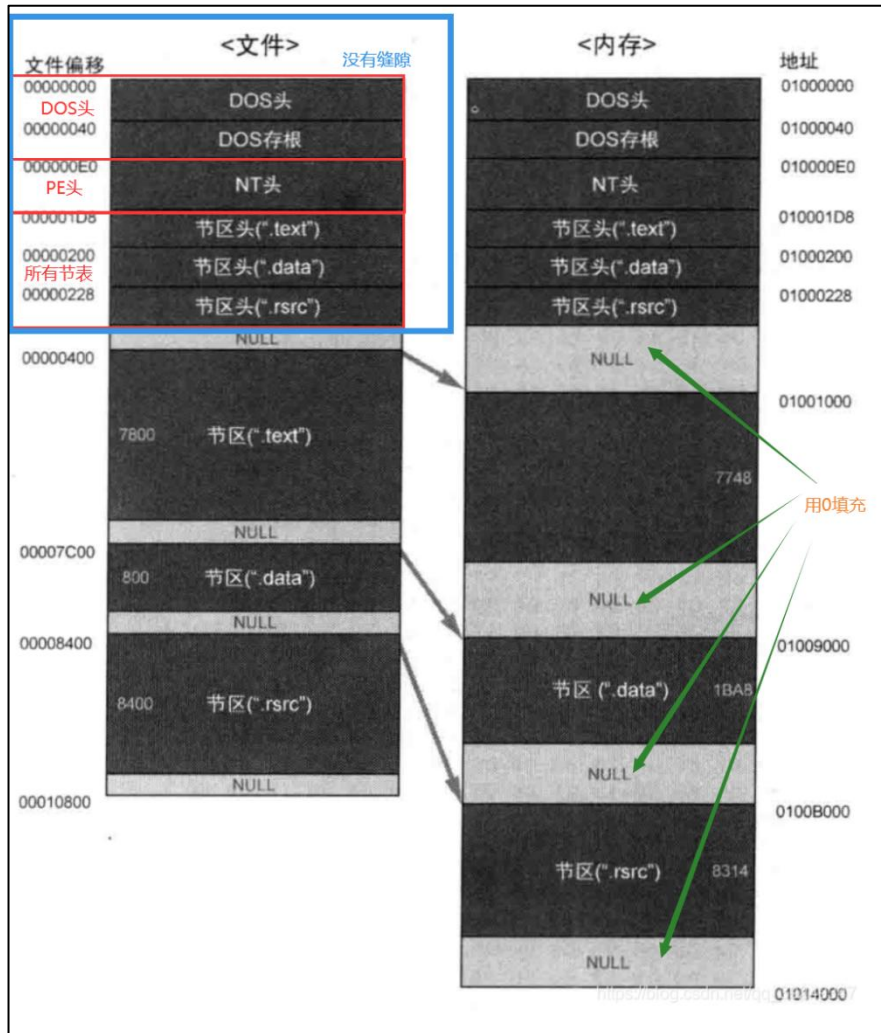
(10)汇编:

C 语言的 for(while)、switch、if、函数、结构体、局部变量和函数的形参的存放规律



## 第2章 PE 文件格式及加壳(加壳)

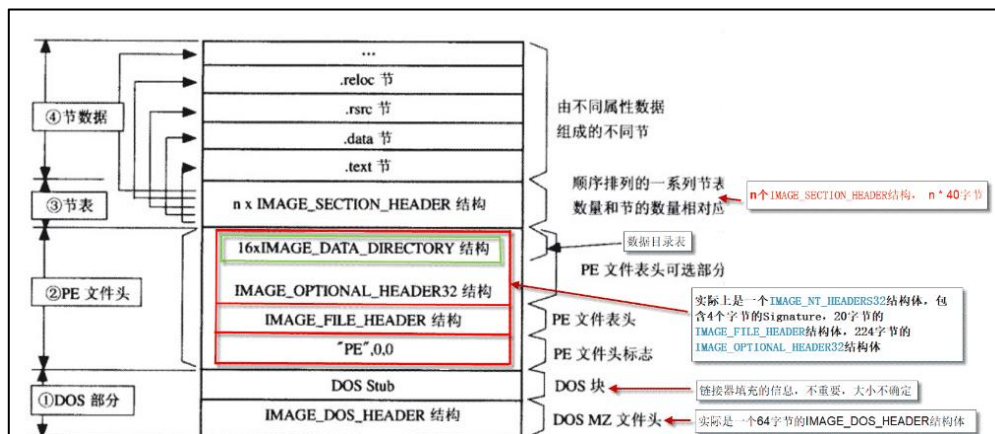
### 1、PE 文件结构



• 从 DOS 头到节区头是 PE 头部分，其下的节区合称 PE 体，文件中使用偏移，内存中使用 VA 来表示位置

• 文件加载到内存时，情况就会发生变化（节区的大小、位置等），文件的内容一般可分为代码、数据、资源节，分别保存。

• PE 头与各节区的尾部都存在一个区域，称为 NULL 填充，为了提高处理文件、内存、网络包的效率，文件/内存中节区的起始位置应该在各文件/内存最小单位的倍数上。



	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	;MZ?.....
00000010h:	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	;?.....@.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030h:	00	00	00	00	00	00	00	00	00	00	00	80	00	00	00	00	.....€...
00000040h:	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	;...???L?Th
00000050h:	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	;is program canno
00000060h:	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	;t be run in DOS
00000070h:	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	;mode...\$.....
00000080h:	50	45	00	00	4C	01	07	00	00	00	00	00	00	00	00	00	;PE..L.....
00000090h:	00	00	00	00	E0	00	0F	03	0B	01	03	05	80	1C	03	00	;.....€...
000000a0h:	24	21	00	00	F4	2A	00	00	40	2C	03	00	00	10	00	00	;\$!...?..@...
000000b0h:	00	30	03	00	00	00	40	00	00	10	00	00	00	02	00	00	;.0.....@.....
000000c0h:	04	00	00	00	01	00	00	00	04	00	00	00	00	00	00	00	.....
000000d0h:	00	50	05	00	00	04	00	00	78	BF	05	00	02	00	00	00	;.P.....x?.....
000000e0h:	00	00	00	01	00	10	00	00	00	00	10	00	00	10	00	00	.....
000000f0h:	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	.....
00000100h:	00	20	04	00	64	00	00	00	00	30	04	00	34	12	01	00	.....d...0..4...
00000110h:	00	00	00	00	00	00	00	00	00	E6	04	00	60	3A	00	00	.....?..^:...
00000120h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000130h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000140h:	D0	50	03	00	18	00	00	00	00	00	00	00	00	00	00	00	.....端
00000150h:	00	00	00	00	00	00	00	00	14	22	04	00	B0	01	00	00	....."....?..
00000160h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000170h:	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	00	00	.....text...
00000180h:	80	1C	03	00	00	10	00	00	00	1E	03	00	00	04	00	00	.....€.....
00000190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	.....
000001a0h:	2E	64	61	74	61	00	00	00	24	21	00	00	00	30	03	00	.....data...\$!..0..
000001b0h:	00	22	00	00	00	22	03	00	00	00	00	00	00	00	00	00	.....
000001c0h:	00	00	00	00	40	00	00	C0	2E	72	64	61	74	61	00	00	.....@...?rdata...
000001d0h:	30	7E	00	00	00	60	03	00	00	80	00	00	00	44	03	00	.....0~...€...D...
000001e0h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	.....@...@
000001f0h:	2E	62	73	73	00	00	00	00	F4	2A	00	00	00	E0	03	00	.....hcs...?....?
00000200h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000210h:	00	00	00	00	80	00	00	C0	2E	43	52	54	00	00	00	00	.....€...?CRT....
00000220h:	0C	00	00	00	00	10	04	00	00	02	00	00	00	C4	03	00	.....?.....?
00000230h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	.....@...?
00000240h:	2E	69	64	61	74	61	00	00	3D	0B	00	00	00	20	04	00	.....idata..=....
00000250h:	00	0C	00	00	00	C6	03	00	00	00	00	00	00	00	00	00	.....?.....
00000260h:	00	00	00	00	40	00	00	C0	2E	72	73	72	63	00	00	00	.....@...?rsrc...
00000270h:	34	12	01	00	00	30	04	00	00	14	01	00	00	D2	03	00	.....4...0.....?..
00000280h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	.....@...?
00000290h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000002a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000002b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000002c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000002d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

IMAGE\_DOS\_HEADER 64B

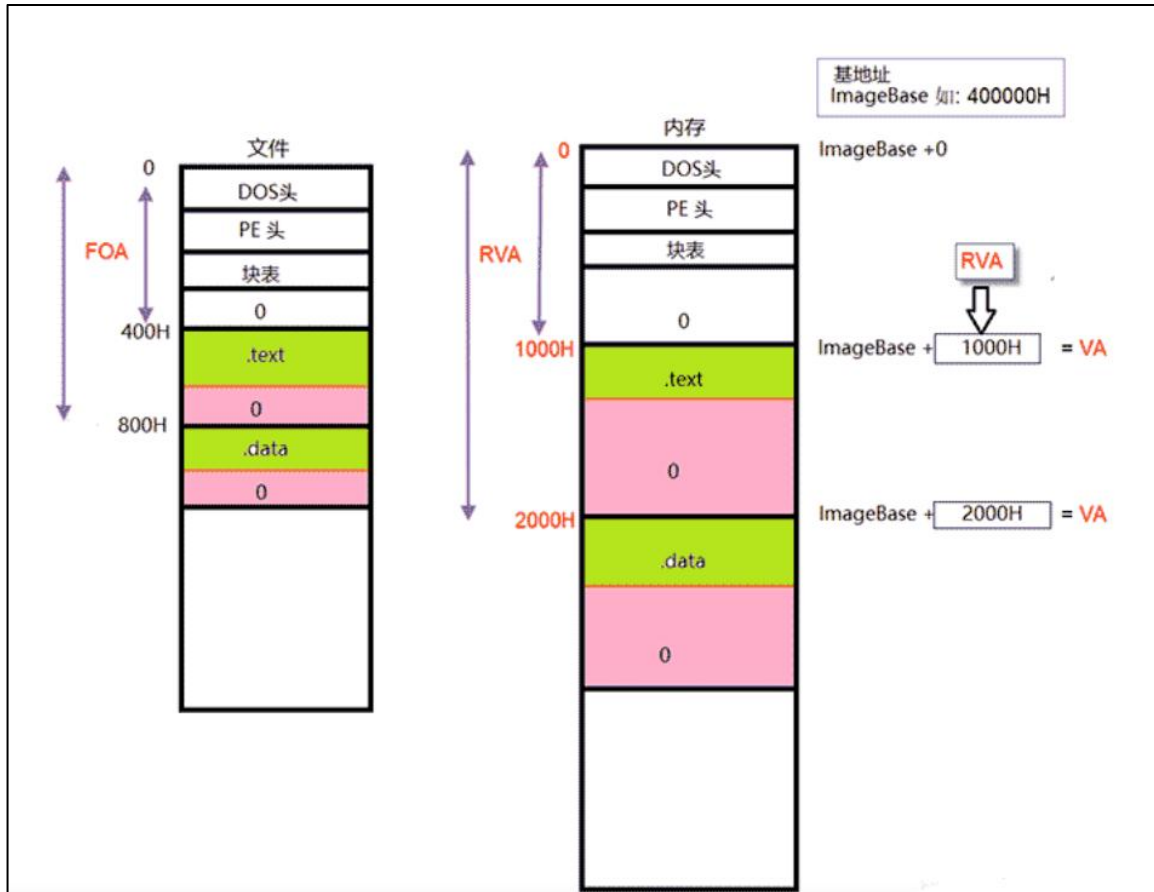
DOS stub

IMAGE\_NT\_HEADERS32 4B+20B+224B

N \* IMAGE\_SECTION\_HEADER n \* 40B

## 2、文件到内存的映射

- 文件中块的大小为 200H 的整数倍，内存中块的大小为 1000H 的整数倍，映射后实际数据的大小不变，多余部分可用 0 填充
- PE 文件头部（DOS 头+PE 头）到块表之间没有间隙，然而他们却和块之间有间隙，大小取决于对齐参数



VA: 虚拟内存地址

RVA: 相对虚拟地址即相对于基地址的偏移地址

FOA: 文件偏移地址

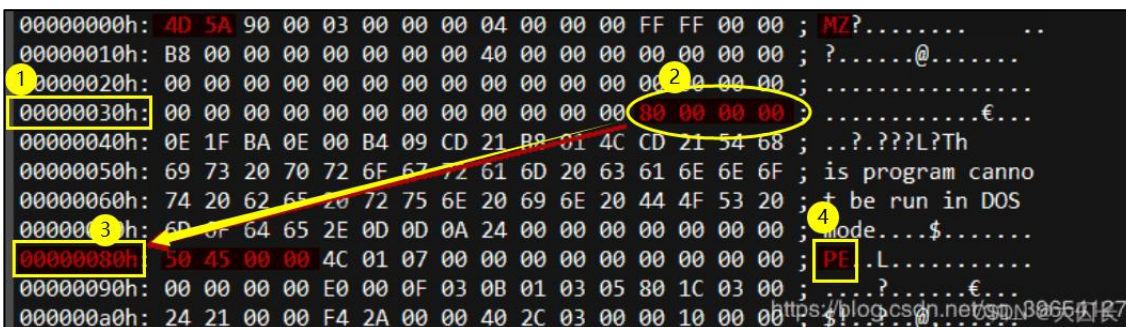


### 3、DOS 部分

(1)DOS MZ 文件头实际是一个结构体 (IMAGE\_DOS\_HEADER)，占 64 字节

- DOS 头用于 16 位系统中，在 32 位系统中 DOS 头成为冗余数据，但还存在两个重要成员 e\_magic 字段（偏移 0x0）和 e\_lfanew 字段（偏移 0x3C）

- e\_magic 保存“MZ”字符，e\_lfanew 保存 PE 文件头地址，通过这个地址找到 PE 文件头，得到 PE 文件标识“PE”。



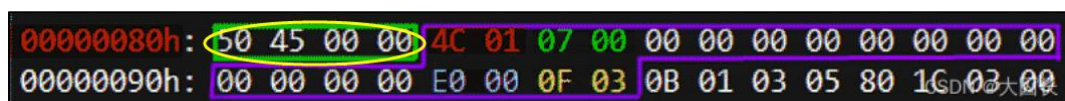
(2) “DOS Stub”区域的数据由链接器填充（可自己填充如意数据），是一段可以在 DOS 下运行的一小段代码，大小不固定。

这段代码的唯一作用是向终端输出一行字：“This program cannot be run in DOS”（“e\_cs”和“e\_ip”指向）然后退出程序，表示该程序不能在 DOS 下运行。

### 4、PE 文件头(PE Header)

PE 文件头是一个结构体(IMAGE\_NT\_HEADERS32)，里面还包含两个其它结构体，占用 4B + 20B + 224B

- 4B: PE 文件标识(Signature)
- 20B: 标准 PE 头(IMAGE\_FILE\_HEADER FileHeader)
- 224B: 扩展 PE 头(IMAGE\_OPTIONAL\_HEADER32 OptionalHeader)

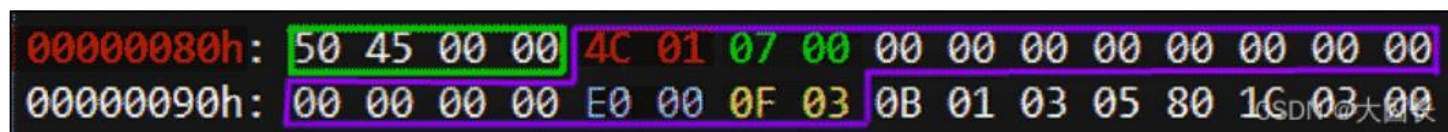


Signature 字段设置为 0x00004550，ANSI 码字符是“PE00”，标识 PE 文件头的开始，PE 标识不能破坏。

(1)IMAGE\_FILE\_HEADER(映像文件头或标准 PE 头)

重要字段：（对应结构为下图紫色部分）

- Machine: 可运行在什么样的 CPU 上，0x014C 说明运行于 x86 CPU
- NumberOfSections: 文件的区块(节)数，0x0007 说明当前 exe 有 7 个节
- TimeDateStamp: 文件创建时间，不重要
- PointerToSymbolTable: 指向符号表(用于调试)
- NumberOfSymbols: 符号表中符号的个数(用于调试)
- SizeOfOptionalHeader: 扩展 PE 头的结构大小，0x00E0 说明为 224B
- Characteristics: 文件属性，0x030F (0000 0011 0000 1111) 代表文件属性



注：其中，Machine、NumberOfSections、SizeOfOptionalHeader、Characteristics 字段各占 2B

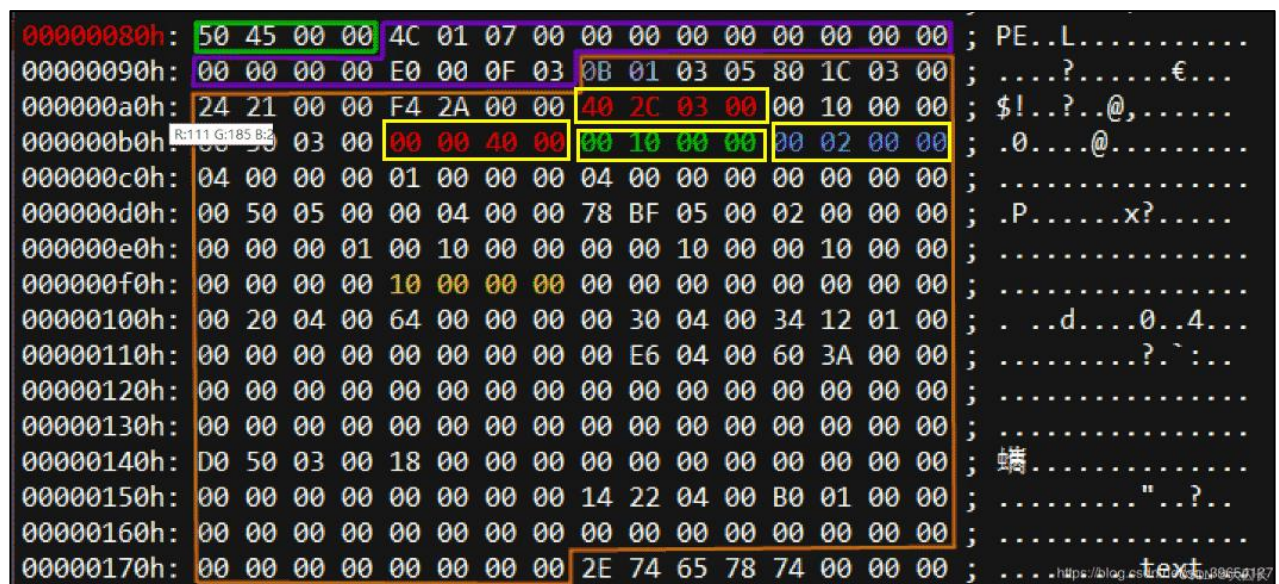


## (2)IMAGE\_OPTIONAL\_HEADER(可选映像头或扩展 PE 头)

IMAGE\_OPTIONAL\_HEADER 是一个可选的结构，是 IMAGE\_FILE\_HEADER 结构的扩展，大小由 IMAGE\_FILE\_HEADER 结构的 SizeOfOptionalHeader 字段记录。

### 重要字段：

- AddressOfEntryPoint: 程序入口地址(RVA)，下图为 32C40H
- ImageBase: 内存镜像基地址，下图为 400000H
- SectionAlignment: 内存对齐，下图为 1000H
- FileAlignment: 文件对齐，下图为 200H



标准 PE 头中 SizeOfOptionalHeader 字段:0x00E0 说明 IMAGE\_OPTIONAL\_HEADER32 为 224 字节，为上图橙色框部分。

### 计算：

ImageBase + AddressOfEntryPoint = 程序实际运行入口地址（实际加载地址等于 ImageBase）  
0x400000 + 0x32C40 = 0x432C40 (使用 OD 运行程序发现就是从这个地址开始运行)

## 5、块表

块表是 IMAGE\_SECTION\_HEADER 的结构数组，每个 IMAGE\_SECTION\_HEADER 结构 40 字节，每个 IMAGE\_SECTION\_HEADER 结构包含了它所关联的区块的信息，例如位置、长度、属性。

重要字段：

- Name[8]：占 8B
- VirtualSize：内存代码块对齐前大小，下图中第一节为 31C80H
- VirtualAddress：代码块装载到内存 RVA，下图中第一节为 1000H
- SizeOfRawData：文件对齐后代码块大小，下图中第一节为 31E00H
- PointerToRawData：代码块在文件中的偏移，下图中第一节为 400H
- Characteristics：代码块属性，下图中第一节为 60000020H

00000160h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; .....
00000170h:	00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00	; .....text...
00000180h:	00 1C 03 00 00 10 00 00 00 1E 03 00 00 04 00 00	; .....f.....
00000190h:	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 00	; .....
000001a0h:	2E 64 61 74 61 00 00 00 24 21 00 00 00 30 03 00	; .data...\$!...0..
000001b0h:	00 22 00 00 00 22 03 00 00 00 00 00 00 00 00 00	; ....."....
000001c0h:	00 00 00 00 40 00 00 C0 2E 72 64 61 74 61 00 00	; .....@...?rdata...
000001d0h:	30 7E 00 00 00 60 03 00 00 80 00 00 00 44 03 00	; 0~...€...D..
000001e0h:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	; .....@..@
000001f0h:	2E 62 73 73 00 00 00 F4 2A 00 00 00 E0 03 00	; .hps.....?...?
00000200h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; .....
00000210h:	00 00 00 00 80 00 00 C0 2E 43 R:22 G:22 B:22 00 00	; ...€...?CRT...
00000220h:	0C 00 00 00 00 10 04 00 00 02 00 00 00 C4 03 00	; .....?...
00000230h:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	; .....@...?
00000240h:	2E 69 64 61 74 61 00 00 3D 08 00 00 00 20 04 00	; .idata...=... ..
00000250h:	00 0C 00 00 00 C6 03 00 00 00 00 00 00 00 00 00	; .....?.....
00000260h:	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00	; .....@...?rsrc...
00000270h:	34 12 01 00 00 30 04 00 00 14 01 00 00 D2 03 00	; 4....0.....?..
00000280h:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	; .....@...?

N \* IMAGE\_SECTION\_HEADER n \* 40B

https://blog.csdn.net/s0138654127

## 6、节的操作——扩大节：对文件的节表进行扩大

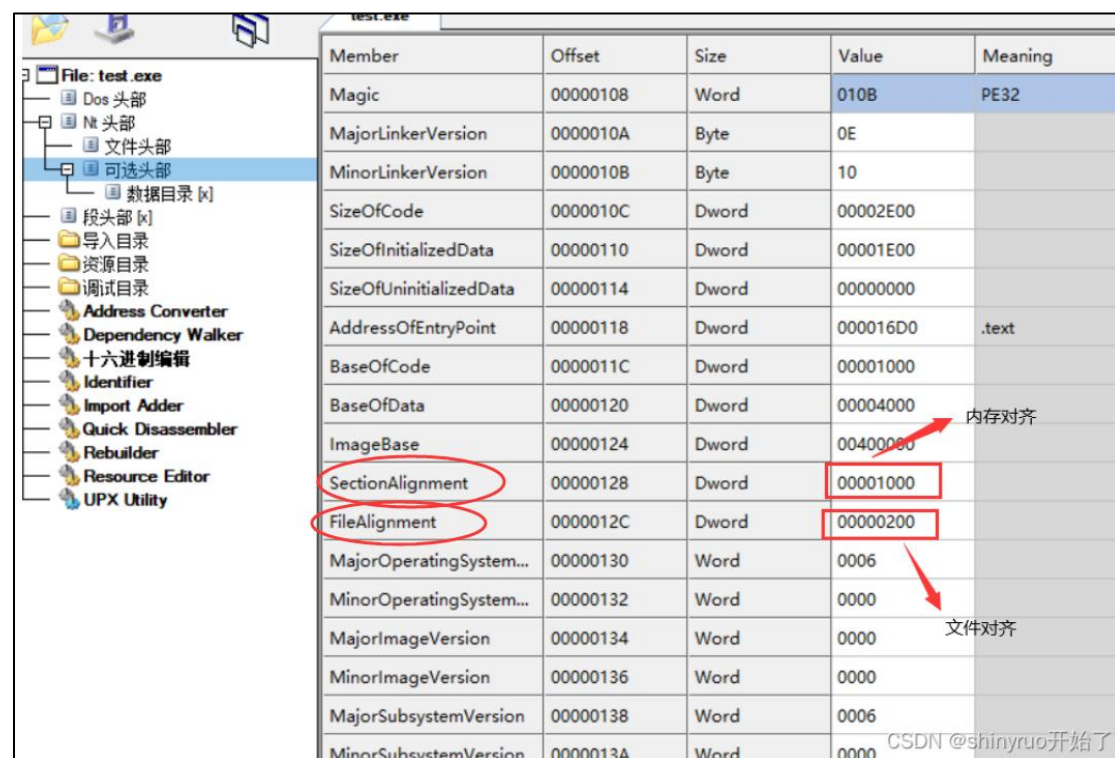
若要对 PE 文件填充自己的代码，然后发现字节空间不足，这时候就需要对节表进行操作——扩大节、新增节、合并节

### (1)所需工具

- Winhex
- CFF Explorer

### (2)实例演示

• 将程序拖入到 CFF Explorer 工具，查看 PE 属性文件对齐与内存对齐的值如下，此处分别为 200、1000



Member	Offset	Size	Value	Meaning
Magic	00000108	Word	010B	PE32
MajorLinkerVersion	0000010A	Byte	0E	
MinorLinkerVersion	0000010B	Byte	10	
SizeOfCode	0000010C	Dword	00002E00	
SizeOfInitializedData	00000110	Dword	00001E00	
SizeOfUninitializedData	00000114	Dword	00000000	
AddressOfEntryPoint	00000118	Dword	000016D0	.text
BaseOfCode	0000011C	Dword	00001000	
BaseOfData	00000120	Dword	00004000	
ImageBase	00000124	Dword	00400000	
SectionAlignment	00000128	Dword	00001000	
FileAlignment	0000012C	Dword	00000200	
MajorOperatingSystem...	00000130	Word	0006	
MinorOperatingSystem...	00000132	Word	0000	
MajorImageVersion	00000134	Word	0000	
MinorImageVersion	00000136	Word	0000	
MajorSubsystemVersion	00000138	Word	0006	
MinorSubsystemVersion	0000013A	Word	0000	

- 查看最后一个节 rsrc 的 Virtual Size(节的实际大小)和 Raw Size(节文件对齐后的大小)

Name	Virtual Size	Virtual Addr...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
000001E8	000001F0	000001F4	000001F8	000001FC	00000200	00000204	00000208	0000020A	0000020C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00002C27	00001000	00002E00	00000400	00000000	00000000	0000	0000	60000020
.rdata	000014CE	00004000	00001600	00003200	00000000	00000000	0000	0000	40000040
.data	000003C0	00006000	00000200	00004800	00000000	00000000	0000	0000	C0000040
.msvcjmc	00000016	00007000	00000200	00004A00	00000000	00000000	0000	0000	C0000040
.rsrc	000001E0	00008000	00000200	00004C00	00000000	00000000	0000	0000	40000040

此处 rsrc 节的 Virtual Size 值为 1E0，Raw Size 的值为 200，两者取最大值 200

再取 200 与 1000(内存对齐)的最小公倍数，结果为 1000；1000 再加上 1000(需扩大的字节数)，结果为 2000；随后将 Virtual Size 和 Raw Size 的值都修改为 2000

节内存对齐所增加的字节 = 节内存对齐后的大小(1000) - 节文件对齐后的大小(200)，计算结果为 E00

- 转到 PE 扩展头查看 SizeOfImage(整个 PE 文件内存对齐后的大小), 此处值为 9000, 将其修改成 9000+1000(扩大的字节数), 即修改成 10000

文件头	AddressOfEntryPoint	00000118	Dword	000016D0	.text
可选头部	BaseOfCode	0000011C	Dword	00001000	
数据目录 [x]	BaseOfData	00000120	Dword	00004000	
段头部 [x]	ImageBase	00000124	Dword	00400000	
导入目录	SectionAlignment	00000128	Dword	00001000	
资源目录	FileAlignment	0000012C	Dword	00000200	
调试目录	MajorOperatingSystem...	00000130	Word	0006	
Address Converter	MinorOperatingSystem...	00000132	Word	0000	
Dependency Walker	MajorImageVersion	00000134	Word	0000	
十六进制编辑	MinorImageVersion	00000136	Word	0000	
Identifier	MajorSubsystemVersion	00000138	Word	0006	
Import Adder	MinorSubsystemVersion	0000013A	Word	0000	
Quick Disassembler	Win32VersionValue	0000013C	Dword	00000000	
Rebuilder	SizeOfImage	00000140	Dword	00009000	
Resource Editor	SizeOfHeaders	00000144	Dword	00000400	
UPX Utility	Checksum	00000148	Dword	00000000	
	Subsystem	0000014C	Word	0003	Windows Conso.
	DllCharacteristics	0000014E	Word	8100	Click here

- 将文件保存后拖入到 winhex 中打开, 将光标拖到最后一个字节, 执行编辑插入 0 字节操作。要插入的字节数 = E00(节内存对齐增加的字节) + 1000(需扩大的字节) = 1E00, 换算成 10 进制, 即为 7680。



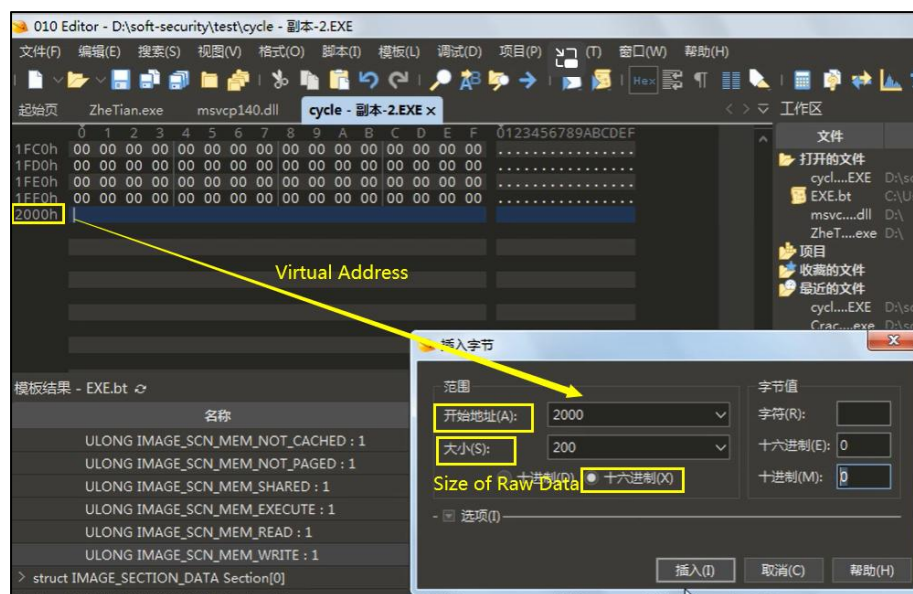


(3)老土的步骤:

- 先看我们的 PE 文件的空间是否可以足以增加一节;
- 可以的话, 我们修改 `numberofsection` 的值, 然后保存, 重新加载 PE 文件;
- 使用 010editor 设置新增加的这个节的成员值, 到  
`struct IMAGE_SECTION_HEADER SectionHeaders[6]`设置;

Visual Size(节的实际大小)=200h

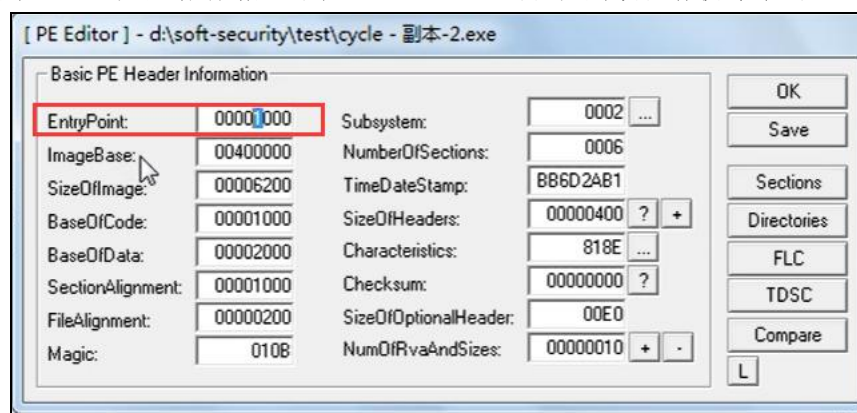
Visual Address=上一节的.Visual Address+上一节的.Sizeofrawdata 的内存对齐后的大小  
=2000h+内存对齐(200h, 1000h)=3000h



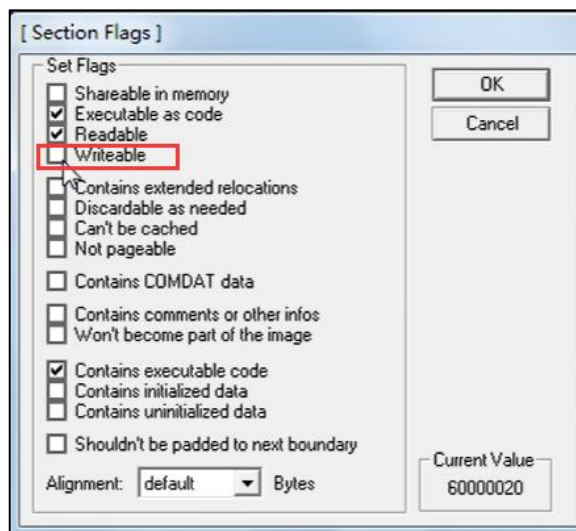
Sizeofrawdata=要增加的节的大小

PointertoRawdata=上一节的.PointertoRawdata + 上一节的.Sizeofrawdata 的内存对齐后的大小

- 在 010editor 里面, 到 PE 文件的尾部, 点击-编辑-插入/覆盖-插入字节;
- 在 010editor 里面, 修改 `Sizeofimage`, 它等于最后一节的 `Visual adresss`+最后一节的 `Sizeofrawdata` 的内存对齐后的大小;
- 保存;
- 使用 loadPE, 查看新增加的节 `Visual Offset` 的值, 用该值替换原来的 `EntryPoint`;



- 再进入 section，修改.text 节(代码 CODE)的属性为可写(writeable);



- 保存退出;
- 尝试运行修改的 PE 文件，不能运行;
- 打开 OD，加载修改好的 PE 文件，然后在第一条指令，修改为“JMP 004010000”，右键-复制到可执行文件-保存文件;

(4)如何内存对齐:

.rsrc	0000043C	0001E000	00000600	00009600	00000000	00000000	0000	0000	40000040
.reloc	00000584	0001F000	00000600	00009C00	00000000	00000000	0000	0000	42000040

.rsrc 的 Virtual Address 是 0001E000，Raw Size 是 00000600

(这里的 Raw Size 是 Virtual Size 文件对齐的结果)

Virtual Size 是 0000043c，文件对齐后必须扩展到 200 的倍数的空间，所以加到 00000600

注：这个 43c 到 600 怎么体现 200 倍的呢，400 装不下(小于 34c)，文件对齐，就得再加 200，即为 600。

内存对齐需要再大到 1000 的倍数空间，这里 600 没超过 1000，说明没有超出空间，一个 1000 的空间就够了

Visual Address=上一节的 Visual Address + 上一节的.Sizeofrawdata 内存对齐后的大小

所以这里对应的就是上一节 Visual Address+1000

假设，.rsrc 的 Virtual Size(节的实际大小)是 00001001，那么 Raw Size(节文件对齐后的大小)是 00001200，这时候一个 1000 的空间装不下了( $1000 < \max\{1001, 1200\}$ )，就需要再开一个 1000 的空间才能满足内存对齐，那就要+2000

## 7、名词解释汇总 2

### (1)PE 文件

PE 文件全称 **Portable Executable**，意为可移植可执行文件即可以由操作系统进行加载执行的文件。常见的 EXE、DLL、OCX、SYS、COM 都是 PE 文件。PE 文件以段的形式存储代码和相关资源数据，其中数据段和代码段是必不可少的两个段

### (2)模块

PE 文件通过 Windows 加载器载入内存后，内存中的版本称为模块

### (3)重要字段

- **Virtual Size**: 节的实际大小
- **Raw Size**: 节文件对齐后的大小
- **IMAGE\_FILE\_HEADER**: 映像文件头或标准 PE 头
  - **VA**: 虚拟内存地址
  - **RVA**: 相对虚拟地址即相对于基地址的偏移地址
  - **FOA**: 文件偏移地址
- **IMAGE\_OPTIONAL\_HEADER**: 可选映像头或扩展 PE 头
  - **AddressOfEntryPoint**: 程序入口地址 (RVA)
  - **ImageBase**: 内存镜像基地址
  - **FileAlignment**: 文件对齐
  - **SectionAlignment**: 内存对齐

### (4)扩大节

对文件的节表进行扩大



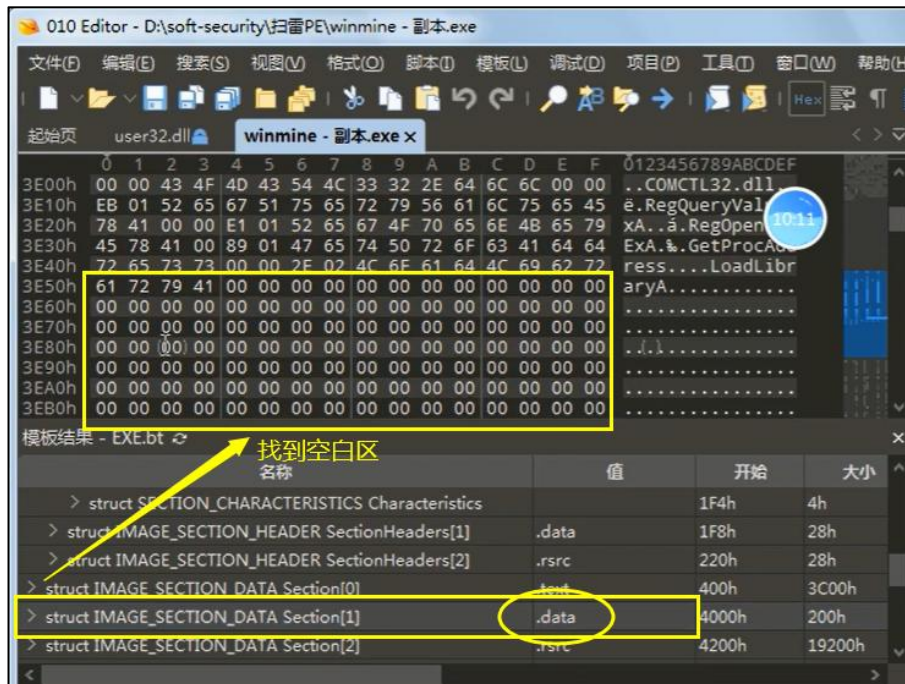
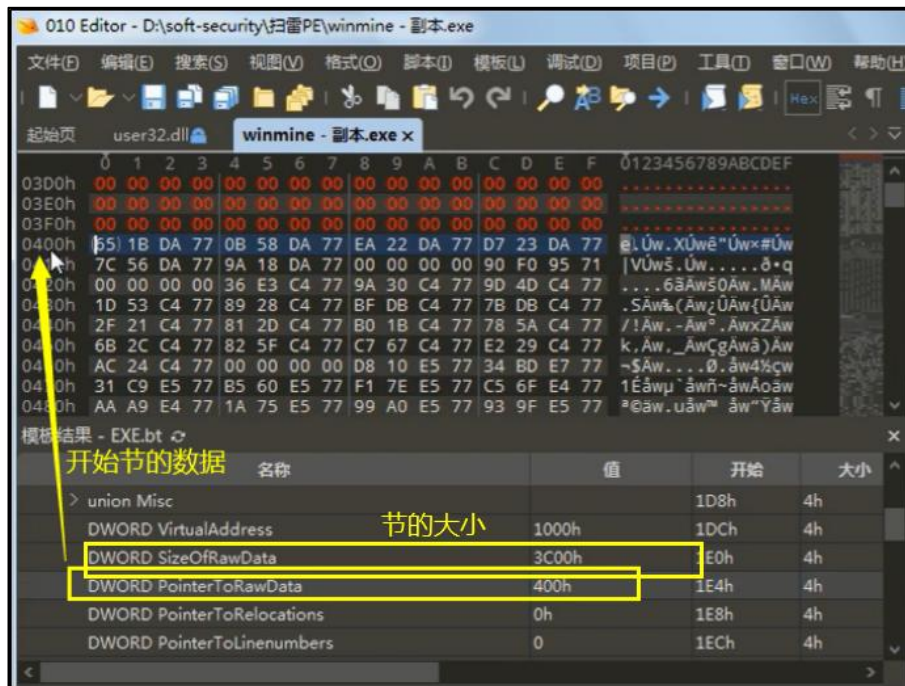
### 第 3 章 MessageBox 函数

#### 1、MessageBox 函数功能

用于显示一个消息对话框

#### 2、在给定的 PE 文件插入 MessageBox 函数 1:

(1)打开 PE 文件，在.text/code 节寻找空白的地方



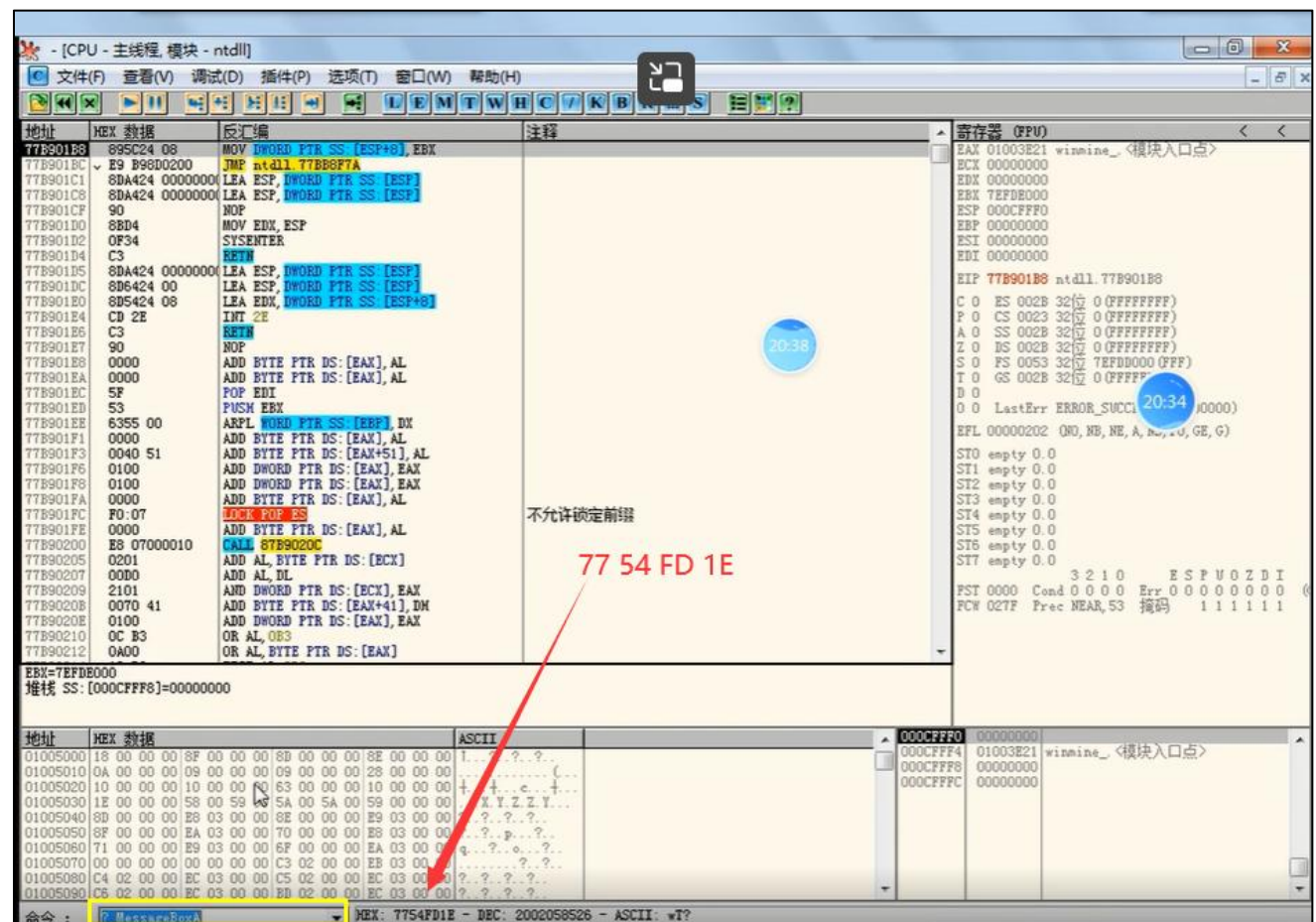
(2) 百度了解 MessageBoxA 函数的用法等

MessageBox() 函数，它的功能是弹出一个标准的 Windows 对话框。返回值是一个 int 型的整数，用于判断用户点击了对话框中的哪一个按钮

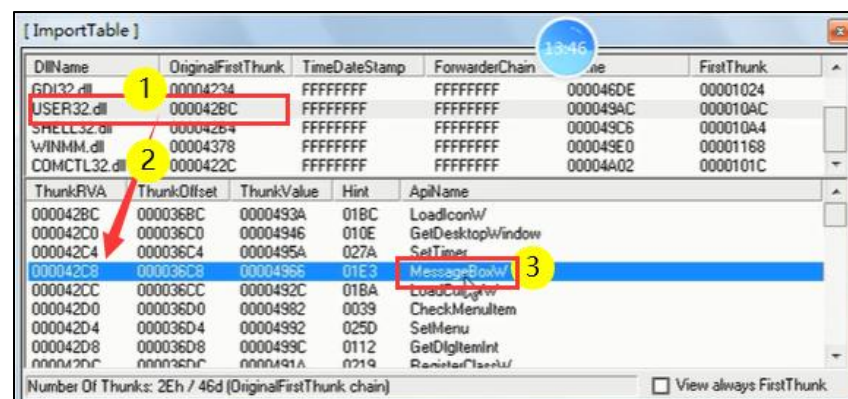
显示一个模式对话框，其中包含系统图标、一组按钮和一条简短的应用程序特定消息，例如状态或错误信息。消息框返回一个整数值，该值指示用户单击的按钮。

(3) 打开 OD 或 x32dbg, 获取 MessageBoxA 函数的地址(需要跳转的虚拟地址)——77 54 FD 1E

• 通过导入表调入函数地址——“?” + “函数名”：

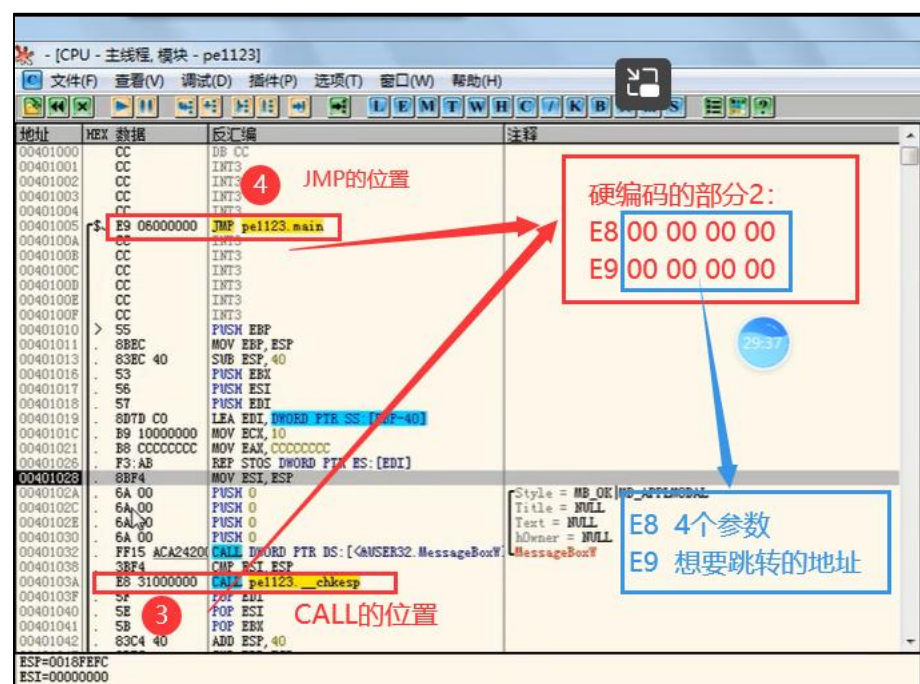
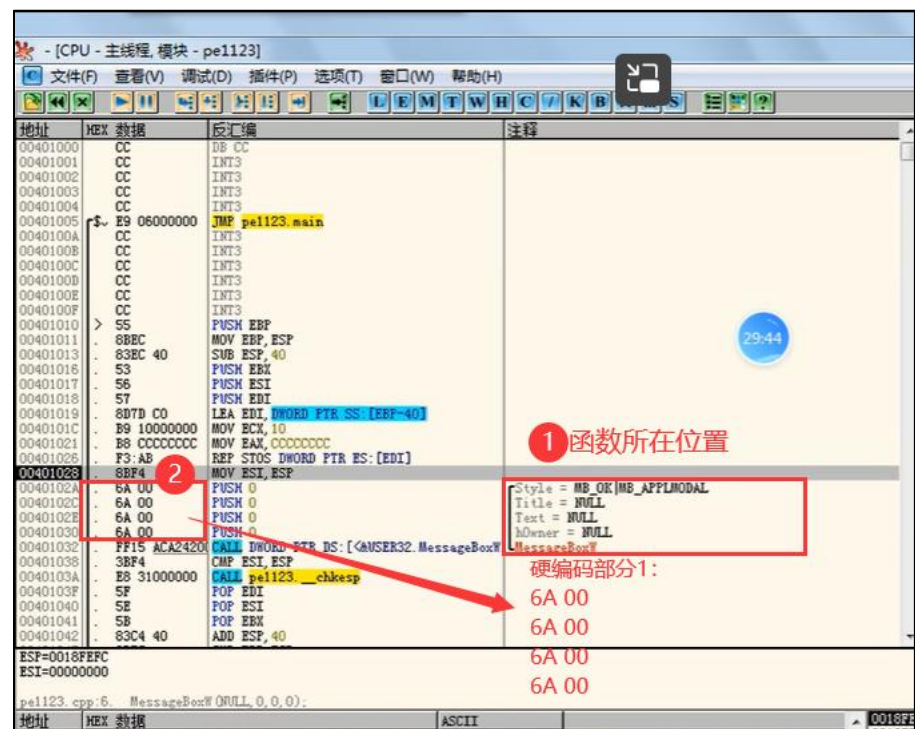


• 程序自动化实现找地址：





#### (4) 获取 MessageBoxA 函数的硬编码



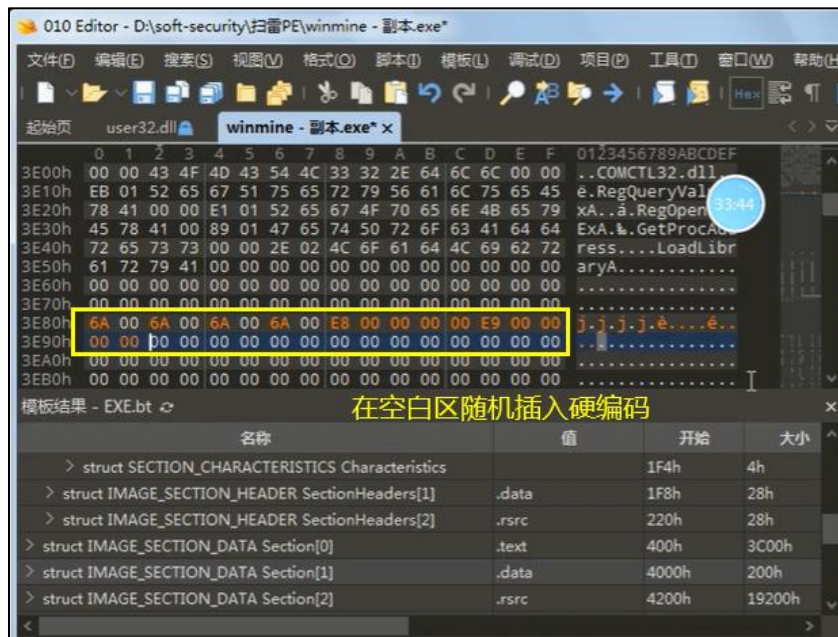
硬编码如下:

6A 00  
6A 00  
6A 00  
6A 00  
E8 00 00 00 00//7754FD1E  
E9 00 00 00 00

注: E8 调用(CALL)MessageBox 函数, E9 跳转(JMP)目标地址

(5)在空白区插入 MessageBoxA 函数的硬编码:

- 在空白处随机插入初始硬编码如下:



- 计算当前调用 MessageBoxA 函数的地址到 7754FD1E 的步长

公式:  $E8 \text{ 步长} = \text{需要跳转虚拟地址} - \text{当前指令虚拟地址} - \text{指令长度}$

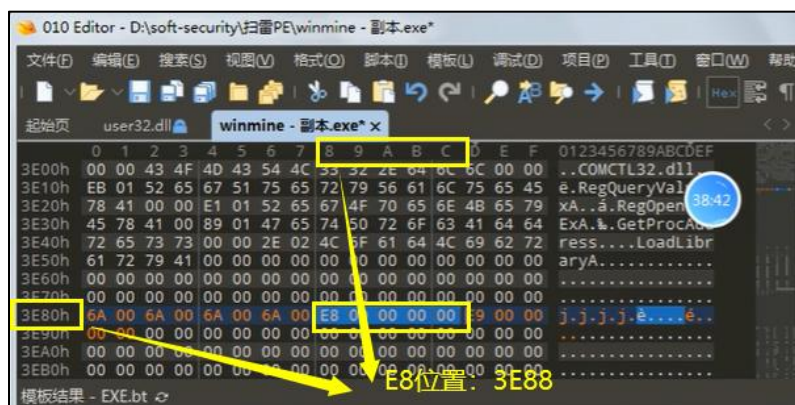
- 需要跳转虚拟地址: 7754FD1E
- 当前指令虚拟地址: 1004A88(具体过程如下框所示)
- 指令长度: 5(E8+4 个字节)

计算:  $CALL\_E8 = 7754FD1E - 1004A88 - 5 = 76\ 54\ B2\ 91$  (小端编码: 91 B2 54 76)

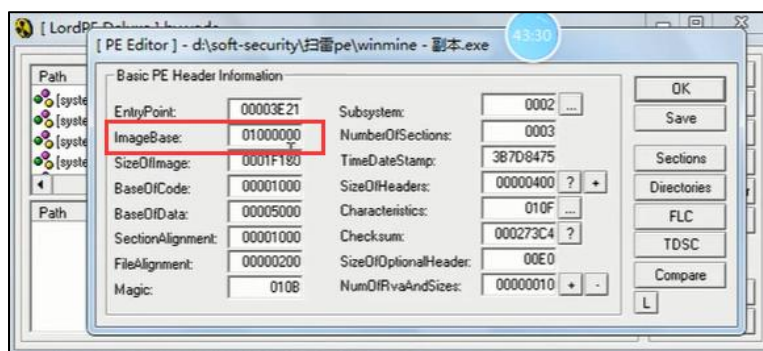
求 E8 当前指令虚拟地址:

公式: 当前虚拟地址 = E8 所在位置(FileOfSet) + 内存镜像基地址(ImageBase)  
+ (当前区块的 Virtual Address - 当前区块的 PointerToRawData)

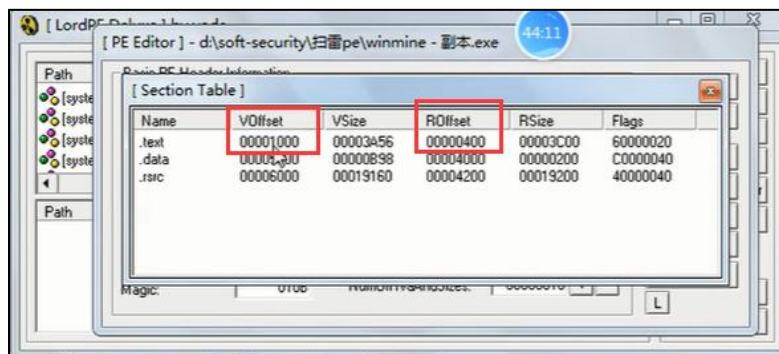
- E8 所在位置(FileOfSet): 3E88



- 内存镜像及地址(ImageBase): 01000000



- 当前区块的 VA 和 PointerToRawData



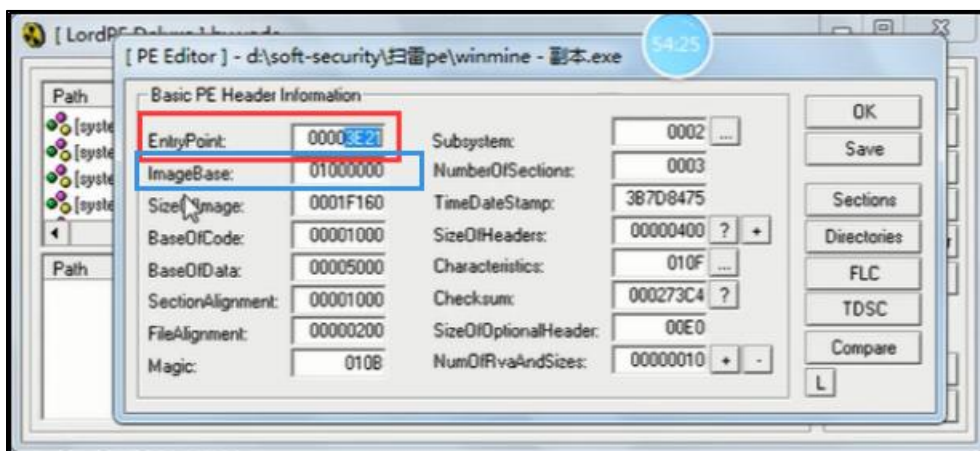
计算:  $\text{Virtual Address} = 3E88 + 01000000 + (1000 - 400) = 1004A88$



- 计算要跳回扫雷程序的原入口地址的步长：

公式：E9 步长=需要跳转虚拟地址 - 当前指令虚拟地址-指令长度

- 需要跳转虚拟地址：00003E21+01000000=01003E21

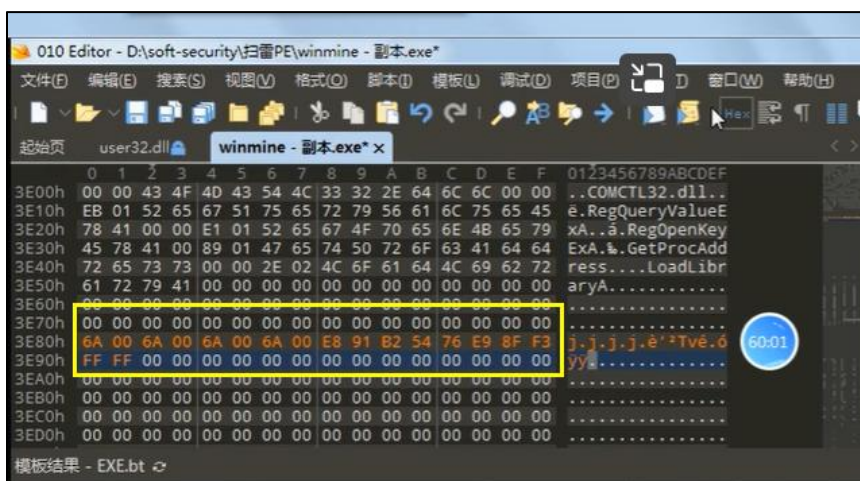


- 当前指令虚拟地址：1004A88+5=1004A8D(E8 当前虚拟地址+E8 的 5 个字节长)
- 指令长度：5(E9+4 个字节)

计算：JMP\_E9 = 01003E21-1004A8D -5 = FF FF F3 8F(小端编码：8F F3 FF FF)

#### (6)修改程序入口地址

- 将计算出的 E8 和 E9 的编码填入对应位置如下



- 计算程序入口地址：

硬编码起始位置 VA-ImageBase=1004A88-8-1000000=4A80

注：E8\_VA=1004A88，前面还有四个参数 6A 00，共占 8 个字节，所以硬编码起始位置为：1004A88-8=1004A80

#### 3、在给定的 PE 文件插入 MessageBox 函数 2：

- (1)打开 PE 文件，在.text/code 节寻找空白的地方
- (2)百度了解 MessageBoxA 函数的用法
- (3)打开 OD 或 x32dbg，获取 MessageBoxA 函数的地址——76 95 FD 3F
- (4)获取 MessageBoxA 函数的硬编码

- 看程序：

```
#include<stdio.h>
#include<Windows.h>
int main()
{
    MessageBoxW(NULL,L"软件安全分析",L"广西大学", NULL);
    system("pause");
    return 0;
}
```

- 得到硬编码：

```
PUSH 6A 00
PUSH 68 00 00 00 00 #地址
PUSH 68 00 00 00 00 #地址
PUSH 6A 00
CALL E8 00 00 00 00
JMP E9 00 00 00 00
```

(5)获取字符串“软件安全分析”、“广西大学”的十六进制编码如下：

- 广西大学：7F 5E 7F 89 27 59 66 5B
- 软件安全分析： 6F 8F F6 4E 89 5B 68 51 06 52 90 67

(6)把字符串编码填入一个空白的地址空间

(7)计算字符串的虚拟地址：

- 广西大学： 401330
- 软件安全分析： 401350

(8)计算调用 MessageBoxW 的步长：

- 计算步长：步长=需要跳转虚拟地址-当前指令虚拟地址-指令长度  
E8 的指令虚拟地址： 40130E  
CALL\_E8 = 7695FD3F-40130E-5 = 76 55 EA 2C(小端： 2C EA 55 76 )

(9)计算跳转的步长，跳转回原程序入口

- 原程序入口 =EntryPoint + ImageBase =400000+1000=401000  
E9 的指令虚拟地址： 401313  
jmp 步长 = 要跳转地址 - 当前指令地址 - 指令长度  
= 401000 - 401313 - 5 = FFFF FCE8(小端： E8 FC FF FF)

(10)修改原程序入口地址变为新的程序入口地址：

入口 = 写入硬编码开始的 VA - ImageBase =( 40130E-14) - 400000 = 1300

#### 4、名词解释汇总 3

(1)后门

绕过安全性控制而获取对程序或系统访问权的方法

(2)MessageBox 函数用法

显一个模式对话框，其中包含系统图标、一组按钮和一条简短的应用程序特定消息，例如状态或错误信息。消息框返回一个整数值，该值指示用户单击的按钮



## 第 4 章 壳与脱壳

### 1、壳与脱壳

- (1)壳是一种概念上的东西，人们为了保护软件不会被轻易的修改或者反编译，希望软件能够获得一种保护，能如同乌龟壳保护乌龟一般，能有一个东西保护自己，于是壳就出现了
- (2)壳的功能就是用来保护软件，常见的壳有压缩壳、加密壳、VM 壳
- (3)脱壳：寻找原程序(原 PE 文件)的 OEP 位置

### 2、壳的装载过程

- (1)获取壳自己所需要使用的 API 地址
- (2)解压或解密原程序的各个区块
- (3)进行必要的重定位——保证源程序能够正常运行
- (4)跳转到程序原入口点(OEP)

### 3、PE 文件脱壳

- (1)先用 PEid 来查看 PE 文件用什么壳（如 UPX 壳，那么可以找到脱壳工具来脱壳）
- (2)可以使用 OD 来加载加壳的 PE 文件
- (3)确定脱壳的方法，并实现脱壳
- (4)dump 出我们的原程序
- (5)脱壳后的 PE 文件要修复

### 4、脱壳 5 种方法

#### (1)单步跟踪法

通过 OllyDbg 的单步(F8)、单步进入(F7)和运行到(F4)功能，完整走过程序的自脱壳过程(不断使用 F8 向下运行，遇见向上跳转的就用 F4)，跳过一些循环恢复代码的片段，并用单步进入确保程序不会略过 OEP。这样可以在软件自动脱壳模块运行完毕后，到达 OEP，并 dump 程序。

#### (2)ESP 定律法：

- (1)进入壳的领空，观察 ESP 的值是否发生变化
- (2)若 ESP 的值发生了变化，则选中变化的值，右键选择“数据窗口跟随”
- (3)来到数据窗口，选中变化的值的内存地址，设置硬件断点，选择：硬件访问、字或双字、word 或 dword
- (4)按 F9 执行程序，来到断点地方，就是我们原程序的 OEP 位置
- (5)进入原程序的领空，dump 出我们的原程序；
- (6)脱壳后的 PE 文件要修复

#### (3)内存镜像法(二次断点法)

在加壳程序被加载时，通过 OD 的 ALT+M 快捷键，进入到程序虚拟内存区段。然后通过加两次内存一次性断点，到达程序正确 OEP 的位置

#### (4)一部到达 OEP

根据所脱壳的特征，寻找其距离 OEP 最近的一处汇编指令，然后下 int3 断点，在程序走到 OEP 的时候 dump 程序

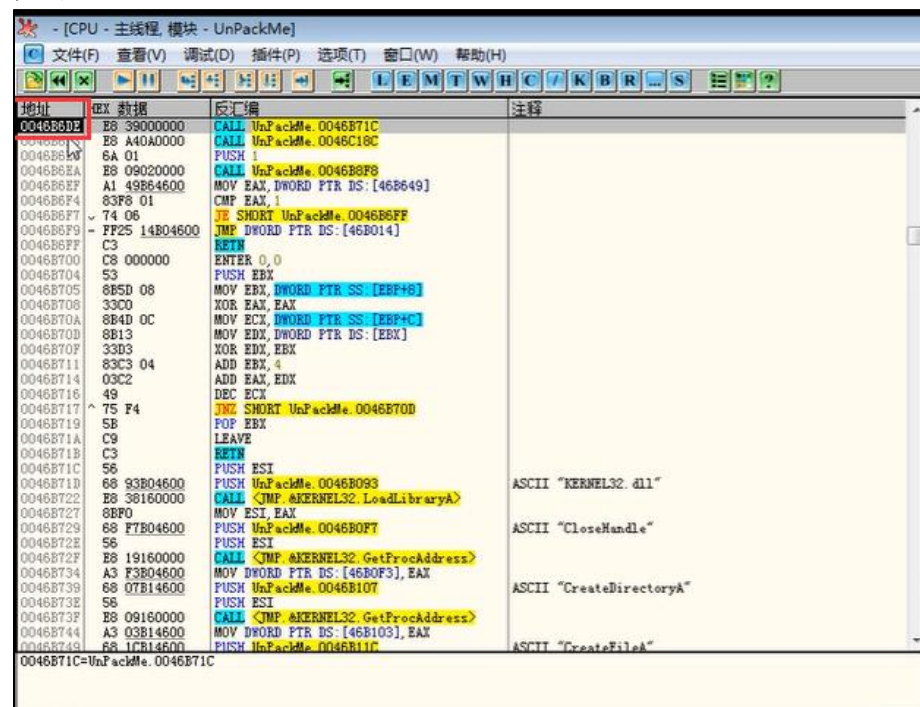
#### (5)最后一次异常法

程序在自解压或自解密过程中，可能会触发无数次的异常。如果能定位到最后一次程序异常的位置，可能就会很接近自动脱壳完成位置。

### 5、实例

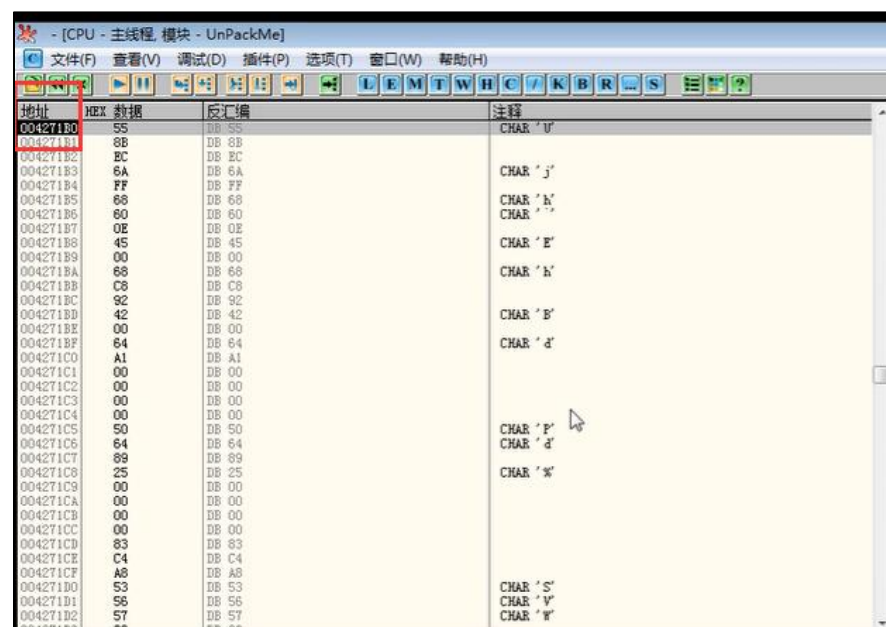
#### (1)实例 1:

壳的 OEP: 46B6DE



地址	HEX 数据	反汇编	注释
0046B6DE	E8 39000000	CALL UnPackMe.0046B71C	
0046B6E0	E8 A40A0000	CALL UnPackMe.0046C18C	
0046B6E4	6A 01	PUSH 1	
0046B6E8	E8 09020000	CALL UnPackMe.0046B8F8	
0046B6EC	A1 49B64600	MOV EAX, DWORD PTR DS:[46B649]	
0046B6F0	83F8 01	CMP EAX, 1	
0046B6F4	74 06	JE SHORT UnPackMe.0046B6FF	
0046B6F8	FF25 14B04600	JMP DWORD PTR DS:[46B014]	
0046B6FC	C3	RETN	
0046B700	C8 000000	ENTER 0,0	
0046B704	53	PUSH EBX	
0046B708	8B5D 08	MOV ESI, DWORD PTR SS:[EBP+8]	
0046B70C	33C0	XOR EAX, EAX	
0046B710	8B4D 0C	MOV ECX, DWORD PTR SS:[EBP+C]	
0046B714	8B13	MOV EDI, DWORD PTR DS:[EBX]	
0046B718	33D3	XOR EDI, EDI	
0046B71C	83C3 04	ADD EBX, 4	
0046B720	03C2	ADD EAX, EDI	
0046B724	49	DEC ECX	
0046B728	75 F4	JNZ SHORT UnPackMe.0046B70D	
0046B72C	5B	POP EBX	
0046B730	C9	LEAVE	
0046B734	C3	RETN	
0046B738	56	PUSH ESI	
0046B73C	68 93B04600	PUSH UnPackMe.0046B093	ASCII "KERNEL32.dll"
0046B740	E8 3B160000	CALL <JMP.AKERNEL32.LoadLibraryA>	
0046B744	8BFD	MOV ESI, EAX	
0046B748	68 F7B04600	PUSH UnPackMe.0046B0F7	ASCII "CloseHandle"
0046B74C	56	PUSH ESI	
0046B750	E8 19160000	CALL <JMP.AKERNEL32.GetProcAddress>	
0046B754	A3 F3B04600	MOV DWORD PTR DS:[46B0F3], EAX	
0046B758	68 07B14600	PUSH UnPackMe.0046B107	ASCII "CreateDirectoryA"
0046B75C	56	PUSH ESI	
0046B760	E8 09160000	CALL <JMP.AKERNEL32.GetProcAddress>	
0046B764	A3 03B14600	MOV DWORD PTR DS:[46B103], EAX	
0046B768	68 1CB14600	PUSH UnPackMe.0046B11C	ASCII "CreateFileA"
0046B76C			

原程序的 OEP: 4271B0



地址	HEX 数据	反汇编	注释
004271B0	55	DB 55	CHAR 'V'
004271B1	8B	DB 8B	
004271B2	EC	DB EC	
004271B3	6A	DB 6A	CHAR 'j'
004271B4	FF	DB FF	
004271B5	68	DB 68	CHAR 'N'
004271B6	60	DB 60	CHAR 'n'
004271B7	0E	DB 0E	
004271B8	45	DB 45	CHAR 'E'
004271B9	00	DB 00	
004271BA	68	DB 68	CHAR 'h'
004271BB	C8	DB C8	
004271BC	92	DB 92	
004271BD	42	DB 42	CHAR 'B'
004271BE	00	DB 00	
004271BF	64	DB 64	CHAR 'd'
004271C0	A1	DB A1	
004271C1	00	DB 00	
004271C2	00	DB 00	
004271C3	00	DB 00	
004271C4	00	DB 00	
004271C5	50	DB 50	CHAR 'P'
004271C6	64	DB 64	CHAR 'd'
004271C7	89	DB 89	
004271C8	25	DB 25	CHAR 'X'
004271C9	00	DB 00	
004271CA	00	DB 00	
004271CB	00	DB 00	
004271CC	00	DB 00	
004271CD	83	DB 83	
004271CE	C4	DB C4	
004271CF	A8	DB A8	
004271D0	53	DB 53	CHAR 'S'
004271D1	56	DB 56	CHAR 'V'
004271D2	57	DB 57	CHAR 'W'
004271D3	68	DB 68	

(2)实例 2:

壳的 OEP: 4650BE

ESP 的值是: 18FF8C

原程序的 OEP: 4271B0

## 6、名词解释汇总 4

### (1)壳

壳是一种概念上的东西,人们为了保护软件不会被轻易的修改或者反编译,希望软件能够获得一种保护,能如同乌龟壳保护乌龟一般,能有一个东西保护自己,于是壳就出现了;壳的功能就是用来保护软件,常见的壳有压缩壳、加密壳、VM 壳

### (2)加/脱壳

加壳是利用特殊的算法,对 EXE、DLL 文件里的资源进行压缩、加密,脱壳是寻找原程序(原 PE 文件)的 OEP 位置

### (3)dump 程序

dump 的本意是"倾卸垃圾"、"把(垃圾桶)倒空", dump 程序一般指将数据导出、转存成文件或静态形式