

第五十三章-TPPpack 脱壳

上一章中最后留的那个小比赛最后的获胜者是 Ulateck 童鞋。下面我们就用 Ulateck 童鞋编写的第一个脚本来定位 OEP 以及修复 stolen bytes。脚本如下:

```
/*
#####

CracksLatinoS - 2006

作者: Ulateck.

描述: 该脚本的功能的定位 TPPack 的 OEP 以及修复其 stolen bytes
目标程序: UnPackMe_TPPack.exe
配置要求: ODBGScript 1.48 , HideDebugger 1.24 , HideOD, 停在入口点处,忽略 Kernel32 的异常,其他异常均不忽略.
        因为我们这里要用到最后一次异常法,所以在执行该脚本之前首先要知道最后一次异常的地址,然后再执行该脚本

以下关于该脚本的详细注释

#####
*/

var dir_excep
var Newwoep
var dir_JMP

var dir_CALL
var oep
var StartScan
var Opcodes
var temp
var temp2
var temp3

Data:
    mov Newwoep, eip           // 将入口点保存到变量 Newwoep 中

    ask "最后一次异常的地址是多少?" // 弹出一个对话框让用户输入最后一次异常的地址
    cmp $RESULT,0             // 判断用户是否输入了地址
    je warning                // 如果用户没有输入地址则跳转到 warning 标签处
    mov dir_excep, $RESULT    // 将用户输入的地址保存到变量 dir_excep 中
    jmp Initiation            // 跳转到 Initiation 标签处

warning:
    msg "请重新执行该脚本,再次输入一个有效的地址!"
```

jmp final

Initiation:

run // 运行起来
eoe check // 如果发生异常断了下来,就跳转到 check 标签处

check:

cmp eip,dir_excep // 判断断下来的地方是不是最后一次异常处
je last // 断下来的地方刚好是最后一次异常处,则跳转到 last 标签处
esto // 忽略掉异常继续执行,相当于在 OD 中按 SHIFT+F9
jmp Initiation: // 跳转 Initiation 标签处继续定位最后一次异常处

last:

findop eip,#FFE0# // 从最后一次异常处开始搜索 JMP EAX 指令,以便下面定位 stolen bytes
mov dir_JMP,\$RESULT // 将 JMP EAX 指令的地址保存到变量 dir_JMP 中
bp dir_JMP // 对 JMP EAX 指令设置断点
esto // 忽略掉异常继续执行,相当于在 OD 中按了 SHIFT+F9
bc dir_JMP // 删除掉 JMP EAX 指令处的断点
sti // 单步步入,相当于在 OD 中按 F7,单步以后就到了 stolen bytes 处

mov oep,eip // 将 stolen bytes 的起始地址保存到变量 oep 中
mov StartScan,eip // 将 stolen bytes 的起始地址保存到变量 StartScan 中

LookForCall: // 开始搜索 Stolen bytes 中需要修正偏移量的 CALL

findop StartScan,#E8# // 搜索以机器码 E8 开头的 CALL 指令,即待修正偏移量的 CALL 指令
cmp \$RESULT, 0 // 判断是否搜索到了待修正偏移量的 CALL 指令

je final // 没有搜索到的话,则跳转到 final 标签处
mov dir_CALL, \$RESULT // 将待修正偏移量 CALL 的地址保存到变量 dir_CALL 中
mov StartScan, \$RESULT // 将待修正偏移量的 CALL 指令的地址赋值给变量 StartScan
add dir_CALL,1 // 指向偏移量
mov Opcodes, [dir_CALL] // 获取待修正的偏移量并保存到变量 Opcodes 中
add Opcodes,StartScan // 将偏移量加上 CALL 指令所在的地址
add Opcodes,5 // 加上 CALL 指令的长度
// 这样就得到了 CALL 指令的目标地址

//修正 CALL 指令的偏移量

mov temp, StartScan // 将 CALL 指令的地址保存到临时变量 temp 中
sub temp, oep // 计算 CALL 指令距离 stolen bytes 起始地址的长度,并保存到临时变量 temp 中
mov temp2, Newoep // 将入口点的值保存到临时变量 temp2 中
add temp,temp2 // 计算 CALL 指令新的地址,并保存到变量 temp 中

```

sub Opcodes, temp          // 将目标地址减去 CALL 指令新的地址
sub Opcodes, 5             // 然后减去 5,就得到了 CALL 指令修正后的偏移量

edit:                      // 将 CALL 指令的偏移量修正
mov temp3, StartScan       // 将 CALL 指令所在的地址保存到临时变量 temp3 中
add temp3,1               // 指向待修正的偏移量
mov [temp3], Opcodes       // 修正偏移量
jmp LookForCall

final:
ret

```

附脚本的截图:

```

0000 /*
0001 ****
0002
0003                                     CracksLatinoS - 2006
0004
0005     作者: Ulaterck.
0006
0007     描述: 该脚本的功能的定位TPPpack的OEP以及修复其stolen bytes
0008
0009     目标程序: UnPackMe_TPPpack.exe
0010     配置要求: ODBGScript 1.48 , HideDebugger 1.24 , HideOO, 停在入口点处,忽略Kerne132的异常,其他异常均不忽略.
0011             因为我们这里要用到最后一次异常发,所以在执行该脚本之前首先定位到最后一次异常的地址,然后再执行该脚本
0012
0013     以下关于该脚本的详细注释
0014     ****
0015 */
0016
0017 var dir_excep
0018 var Newoep
0019 var dir_JMP
0020
0021 var dir_CALL
0022 var oep
0023 var StartScan
0024 var Opcodes
0025 var temp
0026 var temp2
0027 var temp3
0028
0029 Data:
0030 mov Newoep, eip          // 将入口点保存到变量Newoep中
0031
0032 ask "最后一次异常的地址是多少?" // 弹出一个对话框让用户输入最后一次异常的地址
0033 cmp $RESULT,0           // 判断用户是否输入了地址
0034 je warning              // 如果用户没有输入地址则跳转到warning标签处
0035 mov dir_excep, $RESULT  // 将用户输入的地址保存到变量dir_excep中
0036 jmp Initiation          // 跳转到Initiation标签处
0037
0038 warning:
0039 msg "请重新执行该脚本,再次输入一个有效的地址!"
0040 jmp final
0041
0042 Initiation:
0043 run                      // 运行起来
0044 eoe check               // 如果发生异常断了下来,就跳转到check标签处

```

```

0045
0046 check:
0047     cmp eip,dir_excep          // 判断断下来的地方是不是最后一次异常处
0048     je last                    // 断下来的地方刚好是最后一次异常处,则跳转到last标签处
0049     esto                       // 忽略掉异常继续执行,相当于在00中按了SHIFT+F9
0050     jmp Initiation:           // 跳转Initiation标签处继续定位最后一次异常处
0051
0052 last:
0053     findop eip,#FFE0#          // 从最后一次异常处开始搜索JMP EAX指令,以便下面定位stolen bytes
0054     mov dir_JMP,$RESULT        // 将JMP EAX指令的地址保存到变量dir_JMP中
0055     bp dir_JMP                // 对JMP EAX指令设置一个断点
0056     esto                       // 忽略掉异常继续执行,相当于在00中按了SHIFT+F9
0057     bc dir_JMP                // 删除掉JMP EAX指令处的断点
0058     sti                       // 单步步入,相当于在00中按F7,单步以后就到了stolen bytes处
0059
0060     mov oep,eip                // 将stolen bytes的起始地址保存到变量oep中
0061     mov StartScan,eip         // 将stolen bytes的起始地址保存到变量StartScan中
0062
0063 LookForCall:                  // 开始搜索Stolen bytes中需要修正偏移量的CALL,修正完CALL以后,将stolen bytes拷贝到入口点处
0064
0065
0066     findop StartScan,#E8#      // 搜索以机器码E8开头的CALL指令,即待修正偏移量的CALL指令
0067     cmp $RESULT, 0            // 判断是否搜索到了待修正偏移量的CALL指令
0068
0069     je final                  // 没有搜索的话,则跳转到final标签处
0070     mov dir_CALL, $RESULT      // 将待修正偏移量CALL的地址保存到变量dir_CALL中
0071     mov StartScan, $RESULT     // 将待修正偏移量的CALL指令的地址赋值给变量StartScan
0072     add dir_CALL,1             // 指向偏移量
0073     mov Opcodes,[dir_CALL]     // 获取待修正的偏移量并保存到变量Opcodes中
0074     add Opcodes,StartScan      // 将偏移量加上CALL指令所在的地址
0075     add Opcodes,5              // 加上CALL指令的长度
0076                                // 这样就得到了CALL指令的目标地址
0077
0078
0079                                //修正CALL指令的偏移量
0080
0081     mov temp, StartScan        // 将CALL指令的地址(即stolen bytes中的最后一条指令)保存到临时变量temp中
0082     sub temp, oep              // 计算stolen bytes所有指令占的总长度,将该总长度保存到临时变量temp中
0083     mov temp2, Newoep          // 将入口点的值保存到临时变量temp2中
0084     add temp,temp2             // 计算CALL指令新的地址,并保存到变量temp中
0085     sub Opcodes, temp          // 将目标地址减去CALL指令新的地址
0086     sub Opcodes, 5             // 然后减去5,就得到了CALL指令的修正后的偏移量
0087
0088     edit:                     // 将CALL指令的偏移量修正
0089     mov temp3, StartScan      // 将CALL指令所在的地址保存到临时变量temp3中
0090
0091     mov temp3, StartScan      // 将CALL指令所在的地址保存到临时变量temp3中
0092     add temp3,1               // 指向待修正的偏移量
0093     mov [temp3], Opcodes      // 修正偏移量
0094     jmp LookForCall
0095
0096 final:
0097     ret

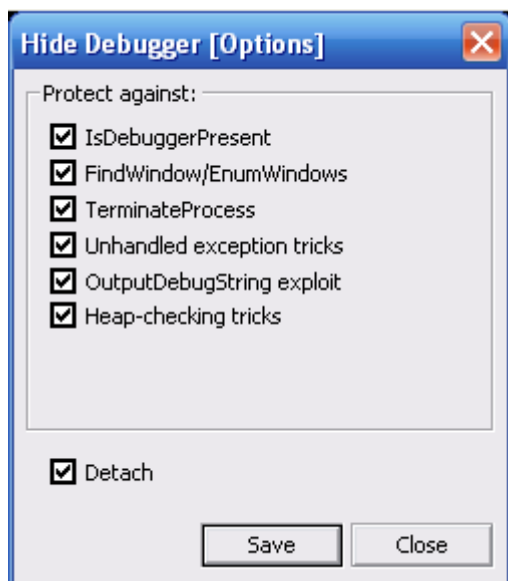
```

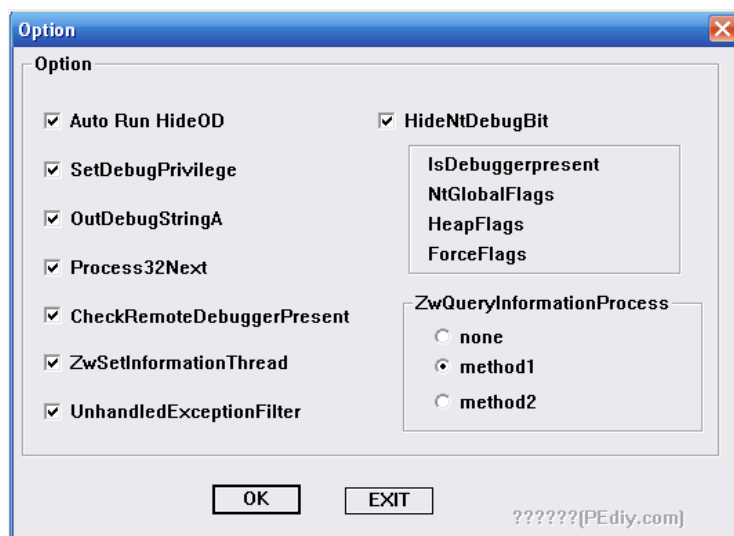
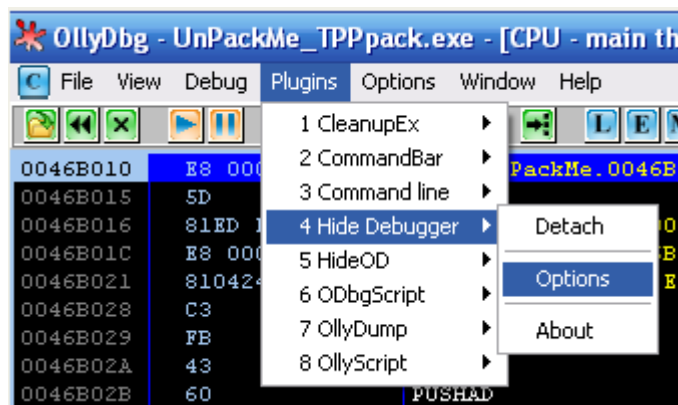
下面我来给大家详细讲解这个脚本的作用。

Part1:定位 OEP 并修复 stolen bytes(by Ulartek)。

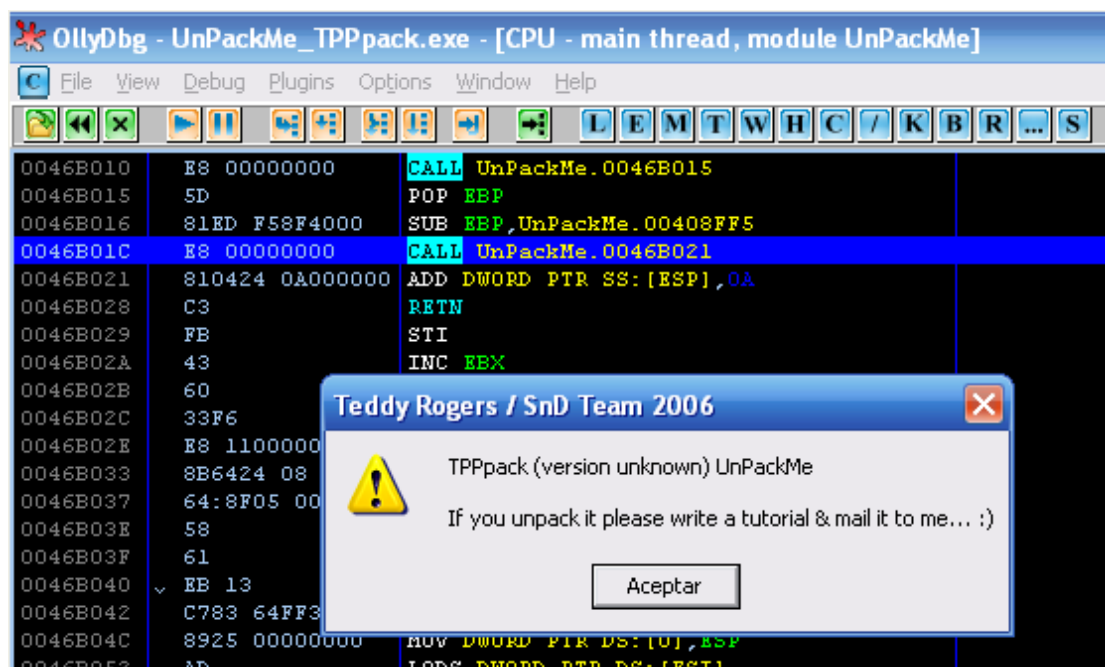
详情:

首先我们需要配置一下反反调试插件。



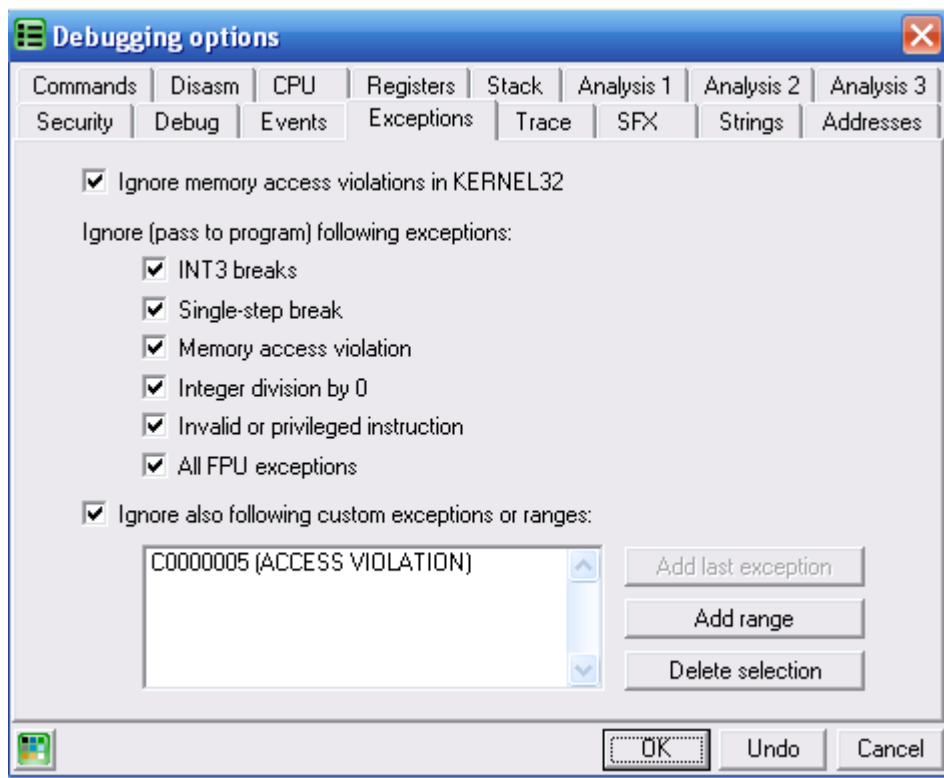


OD 加载目标程序以后直接运行起来,可以看到完美运行。



下面我们利用最后一次异常法来定位 OEP,对于最后一次异常法大家应该很熟练了吧。我们经常会用到它。此法同样适用于 ASProtect 2.1 SKE,2.2 SKE,2.3SKE 以及带 VM 的版本。

这里我们先将所有忽略的异常选项都勾选上。



接着将程序运行起来,然后打开日志窗口,看看最后一次异常发生指令所在的地址是哪里。

```

740C0000 Module C:\WINDOWS\system32\oledlg.dll
774B0000 Module C:\WINDOWS\system32\ole32.dll
100064D8 Access violation when reading [00000000]
100065A2 Access violation when reading [00000000]
1000812F Access violation when reading [00000000]
10006B28 Access violation when reading [00000000]
100081D1 Access violation when reading [00000000]
10005F71 Access violation when reading [00000000]
10006014 Access violation when reading [00000000]
10008278 Access violation when reading [00000000]
10006D8E Access violation when reading [00000000]
10006E40 Access violation when reading [00000000]
100083D9 Access violation when reading [00000000]
1000847B Access violation when reading [00000000]
10008535 Access violation when reading [00000000]
0046D2CF Access violation when reading [00000000]
0046D36B Access violation when reading [00000000]
770F0000 Module C:\WINDOWS\system32\oleaut32.dll
5B150000 Module C:\WINDOWS\system32\uxtheme.dll
746B0000 Module C:\WINDOWS\system32\MSCTF.dll
73260000 Module C:\WINDOWS\system32\RICHED32.DLL

```

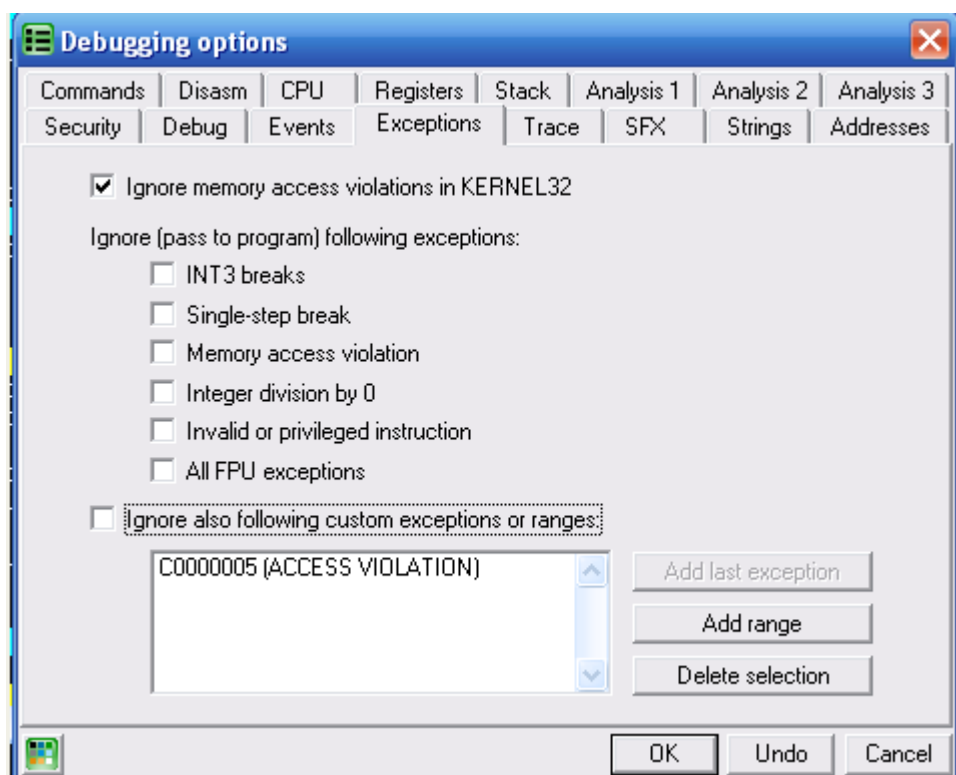
0046D36B

我这里最后一次异常指令所在的地址为 0046D36B,下面我们就可以利用脚本来定位 OEP。

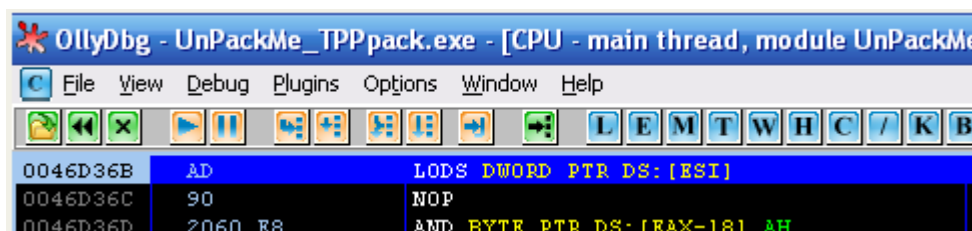
在使用脚本之前,我先演示一下如何手工定位 OEP。

我们重启 OD。

接着将忽略的异常选项的对勾都去掉。

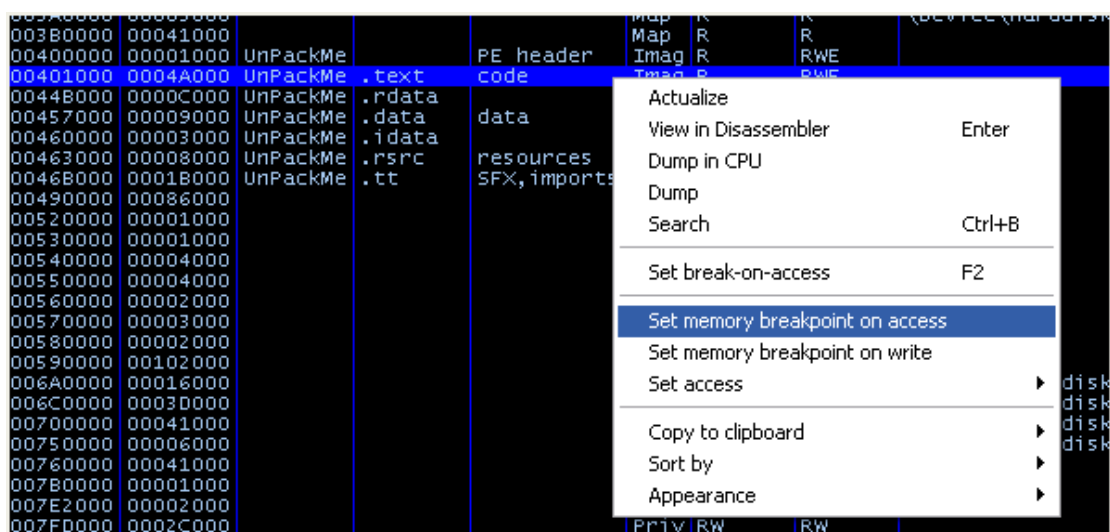


接着直接按 F9 键运行起来。如果断在了不是 0046D36B 的异常处的话,就直接按 SHIFT+F9 忽略掉异常继续执行。



这里我们就断在了最后一次异常处。接下来按 ALT+M 打开区段列表窗口。

对代码段设置内存访问断点。



按 SHIFT+F9 忽略掉异常运行起来,断在了这里。

```

004293A0 6A 00 PUSH 0
004293A2 68 00100000 PUSH 1000
004293A7 6A 00 PUSH 0
004293A9 FF15 A0094600 CALL DWORD PTR DS:[4609A0]
004293AF 85C0 TEST EAX,EAX
004293B1 A3 C4EB4500 MOV DWORD PTR DS:[45EBC4],EAX
004293B6 75 01 JNZ SHORT UnPackMe.004293B9
004293B8 C3 RETN
004293B9 E8 22000000 CALL UnPackMe.004293E0
004293BE 85C0 TEST EAX,EAX
004293C0 75 0F JNZ SHORT UnPackMe.004293D1
004293C2 A1 C4EB4500 MOV EAX,DWORD PTR DS:[45EBC4]
004293C7 50 PUSH EAX
004293C8 FF15 9C094600 CALL DWORD PTR DS:[46099C]
004293CE 33C0 XOR EAX,EAX
004293D0 C3 RETN
004293D1 B8 01000000 MOV EAX,1
004293D6 C3 RETN
004293D7 90 NOP
004293D8 90 NOP

```

如果我们观察一下堆栈的话就会发现这里并不是真正的 OEP,明显存在 stolen bytes。

```

0012FF48 008B0EA4 RETURN to 008B0EA4 from UnPackMe.004293A0
0012FF4C 00000002
0012FF50 00460613 ASCII "erm"
0012FF54 0040B5EE UnPackMe.0040B5EE
0012FF58 818B9258
0012FF5C 817CD020
0012FF60 00000202

```

我们可以看到之前已经执行过 stolen bytes 了。返回地址为 8B0EA4,该地址属于起始地址为 8B0000 的区段。

00460000	00003000	UnPackMe	.idata		Imag	R	RWE	
00463000	00008000	UnPackMe	.rsrc	resources	Imag	R	RWE	
0046B000	0001B000	UnPackMe	.tt	SFX,imports	Imag	R	RWE	
00490000	00086000				Priv	RW	RW	
00520000	00001000				Priv	RW	RW	
00530000	00001000				Priv	RW	RW	
00540000	00004000				Priv	RW	RW	
00550000	00004000				Priv	RW	RW	
00560000	00002000				Map	R	R	
00570000	00003000				Priv	RW	RW	
00580000	00002000				Map	R	R	
00590000	00102000				Map	RW	RW	
006A0000	00016000				Map	R	R	
006C0000	0003D000				Map	R	R	
00700000	00041000				Map	R	R	
00750000	00006000				Map	R	R	
00760000	00041000				Map	R	R	
007B0000	00001000				Priv	RW	RW	
007E2000	00002000				Priv	RW	RW	
007FD000	0002C000				Priv	RW	RW	
008B0000	00001000				Priv	RW	RW	
00AB0000	00004000				Map	R	E	
00B70000	00002000				Map	R	E	
00B80000	00103000				Map	R		
00C90000	00092000				Map	R	E	
00F90000	00001000				Priv	RW	RW	
10000000	00028000				Priv	RWE	RWE	
58C30000	00001000	COMCTL32		PE header	Imag	R	RWE	
58C31000	00070000	COMCTL32	.text	code,imports	Imag	R	RWE	

好,下面我们重启 OD。

我们现在将脚本修改一下,让其自动定位到最后一次异常处。

将脚本修改成如下:


```

var dir_excep
var Newoep

Data:
mov Newoep, eip          // 将入口点保存到变量Newoep中

ask "最后一次异常的地址是多少?" // 弹出一个对话框让用户输入最后一次异常的地址
cmp $RESULT,0           // 判断用户是否输入了地址
je warning              // 如果用户没有输入地址则跳转到warning标签处
mov dir_excep, $RESULT  // 将用户输入的地址保存到变量dir_excep中
jmp Initiation          // 跳转到Initiation标签处

warning:
msg "请重新执行该脚本,再次输入一个有效的地址!"
jmp final

Initiation:
run                      // 运行起来
eoe check               // 如果发生异常断了下来,就跳转到check标签处

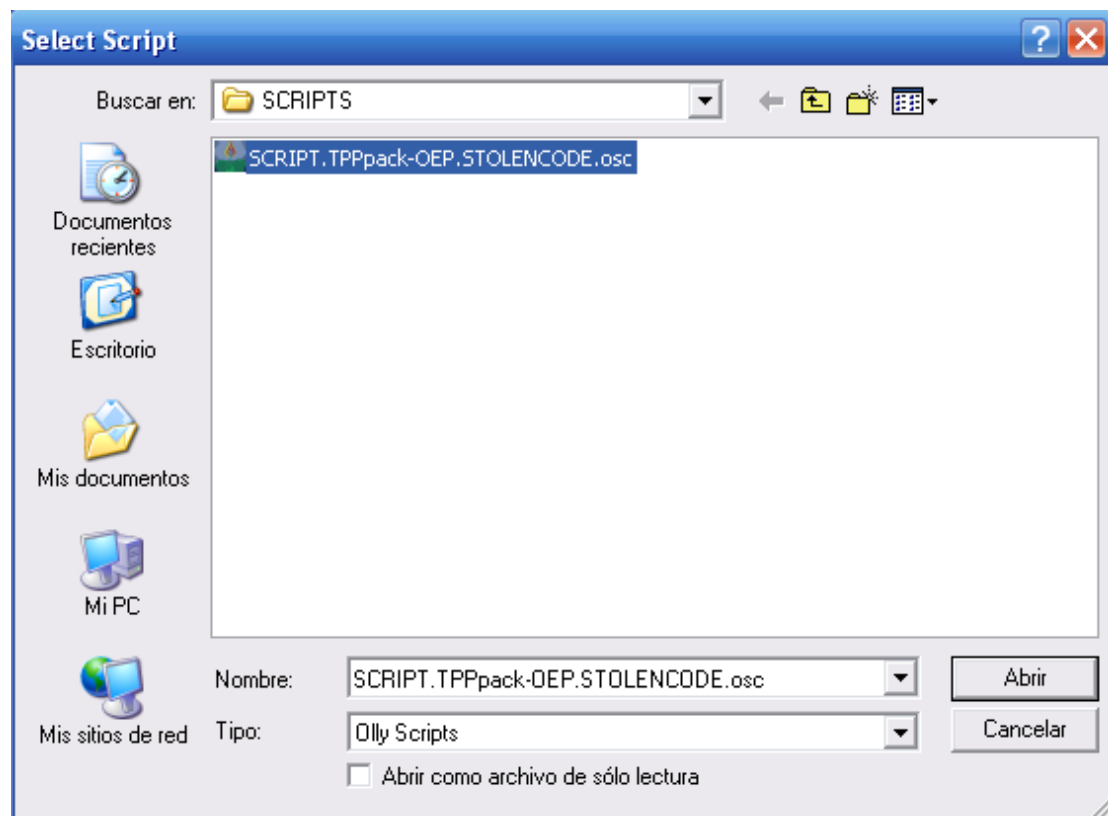
check:
cmp eip,dir_excep       // 判断断下来的地方是不是最后一次异常处
je last                 // 断下来的地方刚好是最后一次异常处,则跳转到last标签处
esto                    // 忽略掉异常继续执行,相当于在OD中按了SHIFT+F9
jmp Initiation          // 跳转Initiation标签处继续定位最后一次异常处

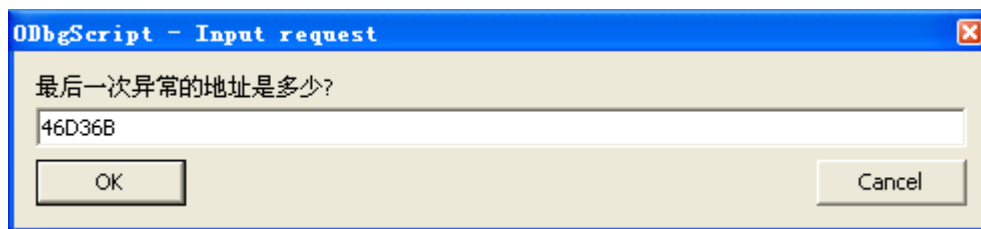
last:
final:
ret

```

好,修改完毕以后。

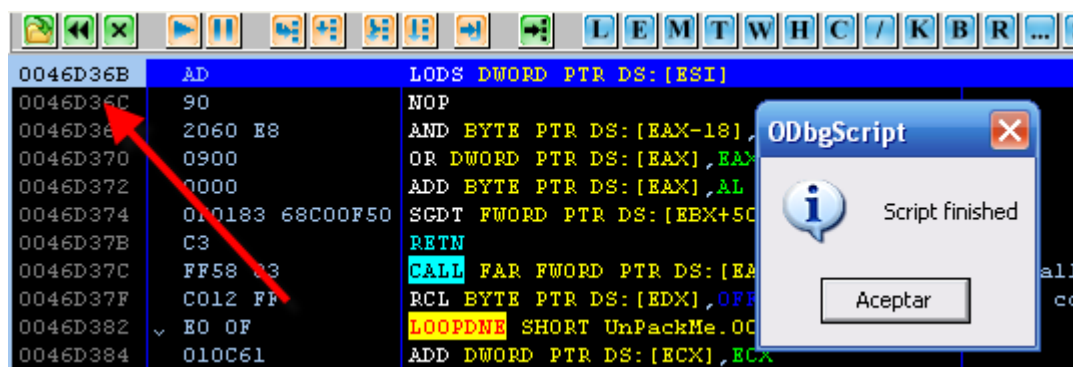
我们重启 OD 以后,执行该脚本。





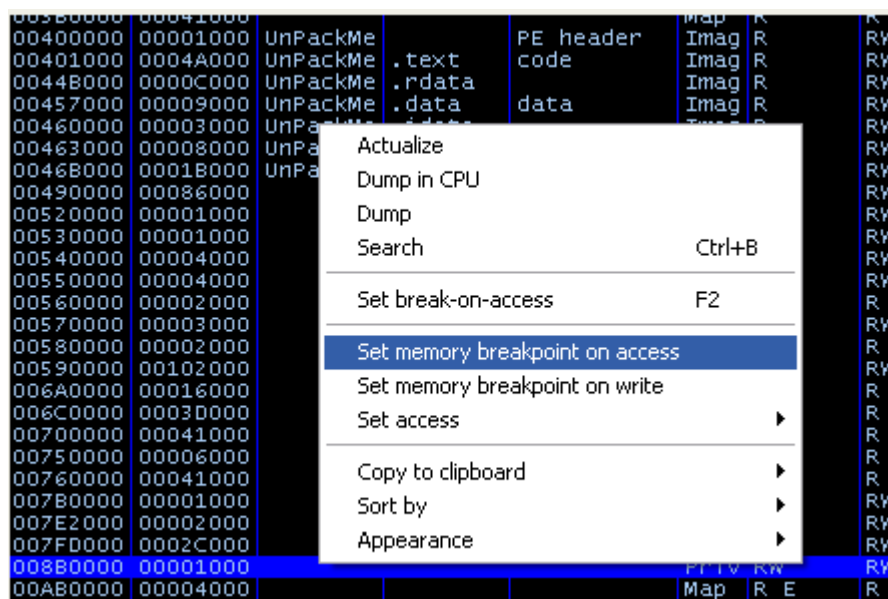
这里弹出了一个对话框要求我们输入最后一次异常指令所在的地址,这里我输入 46D36B。

单击 OK。



好了,我们可以看到脚本执行完毕了,我们可以看到刚好断在了最后一次异常处。

我们按 ALT+M 打开区段列表窗口。



接下来我们并不是跟刚才一样对代码段设置内存访问断点,这次我们对起始地址为 8B0000 的区段设置内存访问断点。

按 SHIFT+F9 忽略掉运行起来,断在了 stolen bytes 处。

008B0E48	55	PUSH EBP
008B0E49	8BEC	MOV EBP,ESP
008B0E4B	6A FF	PUSH -1
008B0E4D	68 600E4500	PUSH 450E60
008B0E52	68 C8924200	PUSH 4292C8
008B0E57	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
008B0E5D	50	PUSH EAX
008B0E5E	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
008B0E65	83C4 A8	ADD ESP,-58
008B0E68	53	PUSH EBX
008B0E69	56	PUSH ESI
008B0E6A	57	PUSH EDI
008B0E6B	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
008B0E6E	FF15 DCOA4600	CALL DWORD PTR DS:[460ADC]
008B0E74	33D2	XOR EDX,EDX
008B0E76	8AD4	MOV DL,AH
008B0E78	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
008B0E7E	8BC8	MOV ECX,EAX
008B0E80	81E1 FF000000	AND ECX,0FF
008B0E86	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX
008B0E8C	C1E1 08	SHL ECX,8

如果我们按减号键可以看到回到了最后一次异常指令处。

0046D36B	AD	LODS DWORD PTR DS:[ESI]	
0046D36C	90	NOP	
0046D36D	2060 E8	AND BYTE PTR DS:[EAX-18],AH	
0046D370	0900	OR DWORD PTR DS:[EAX],EAX	
0046D372	0000	ADD BYTE PTR DS:[EAX],AL	
0046D374	0F0183 68C00F50	SGDT FWORD PTR DS:[EBX+500FC068]	
0046D37B	C3	RETN	
0046D37C	FF58 83	CALL FAR FWORD PTR DS:[EAX-7D]	Far
0046D37F	C012 FF	RCL BYTE PTR DS:[EDX],0FF	Shi
0046D382	✓ E0 0F	LOOPDNE SHORT UnPackMe.0046D393	
0046D384	010C61	ADD DWORD PTR DS:[ECX],ECX	
0046D387	✓ 74 04	JE SHORT UnPackMe.0046D38D	
0046D389	✓ 75 02	JNZ SHORT UnPackMe.0046D38D	
0046D38B	F66E EB	IMUL BYTE PTR DS:[ESI-15]	
0046D38E	0BF2	OR ESI,EDX	
0046D390	06	PUSH ES	
0046D391	C6	???	Unl
0046D392	FD	STD	
0046D393	^ E2 C3	LOOPD SHORT UnPackMe.0046D358	
0046D395	E4 E5	IN AL,0E5	I/O
0046D397	A1 62B360E8	MOV EAX,DWORD PTR DS:[E860B362]	
0046D39C	0900	OR DWORD PTR DS:[EAX],EAX	
0046D39E	0000	ADD BYTE PTR DS:[EAX],AL	
0046D3A0	0F0183 68C00F50	SGDT FWORD PTR DS:[EBX+500FC068]	
0046D3A7	C3	RETN	
0046D3A8	FF58 83	CALL FAR FWORD PTR DS:[EAX-7D]	Far
0046D3AB	C012 FF	RCL BYTE PTR DS:[EDX],0FF	Shi
0046D3AE	✓ E0 0F	LOOPDNE SHORT UnPackMe.0046D3BF	
0046D3B0	010C61	ADD DWORD PTR DS:[ECX],ECX	
0046D3B3	E8 03000000	CALL UnPackMe.0046D3BB	
0046D3B8	- E9 EBE483C4	JMP C4CAB8A8	
0046D3BD	04 E8	ADD AL,0E8	
0046D3BF	0000	ADD BYTE PTR DS:[EAX],AL	
0046D3C1	0000	ADD BYTE PTR DS:[EAX],AL	
0046D3C3	810424 08000000	ADD DWORD PTR SS:[ESP],8	
0046D3CA	C3	RETN	
0046D3CB	FFEO	JMP EAX	
0046D3CD	8B5C24 04	MOV EBX,DWORD PTR SS:[ESP+4]	

以下脚本是根据 Martian 先生在他的教程中介绍的定位 stolen bytes 的思路编写的,定位 stolen bytes 的思路如下:首先定位到最后一次异常处,接着往下搜索机器码为 FFE0 的 JMP EAX 指令,搜到该指令以后,对其设置断点,接着运行起来,断到了 JMPEAX 处,然后

按 F7 键单步一下,就可以到达 stolen bytes 处。

我们在脚本中添加一个变量。

```
var dir_excep
var Newoep
var dir_JMP |
```

Data:

```
mov Newoep, eip
```

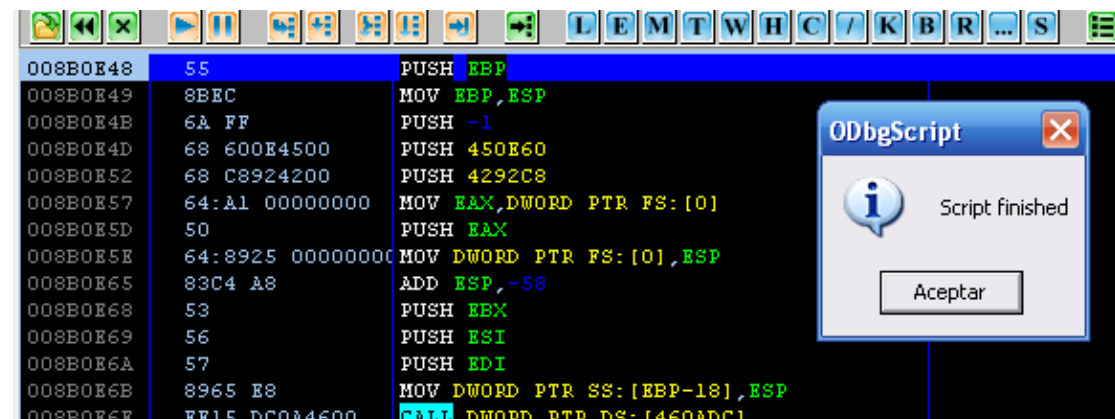
// 将入口点保存到变量Newoep中

变量 dir_JMP 用于保存 JMP EAX 指令的地址。

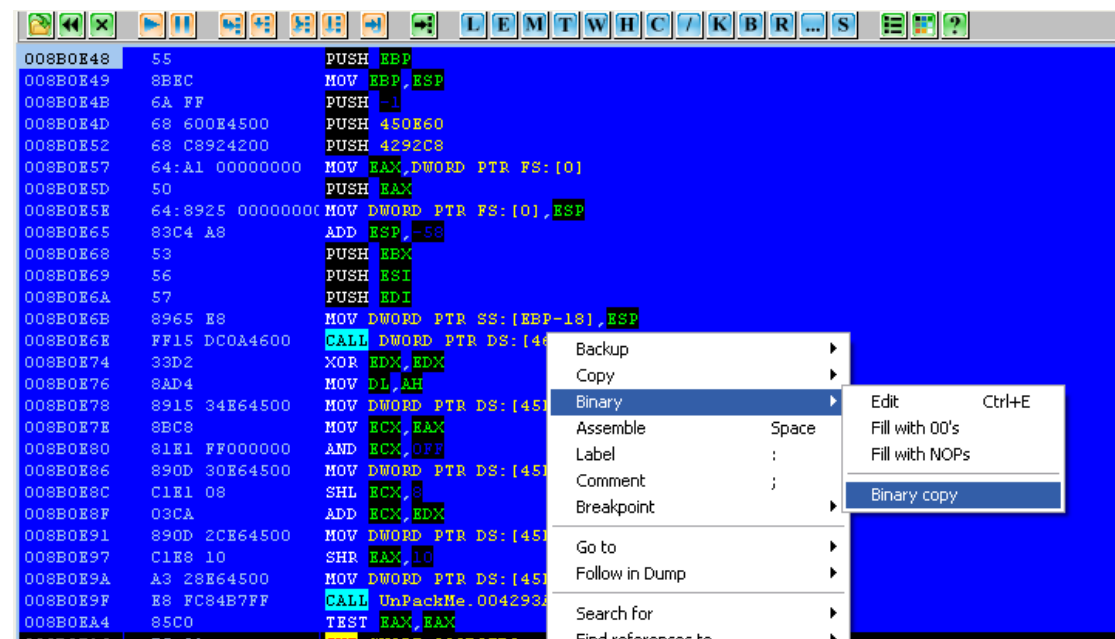
接下来在 last 标签处添加以下内容:

```
last:
findop eip,#FFE0H // 从最后一次异常处开始搜索 JMP EAX指令,以便下面定位stolen bytes
mov dir_JMP,$RESULT // 将JMP EAX指令的地址保存到变量dir_JMP中
bp dir_JMP // 对JMP EAX指令设置一个断点
esto // 忽略掉异常继续执行,相当于在OD中按了SHIFT+F9
bc dir_JMP // 删除掉JMP EAX指令处的断点
sti // 单步步入,相当于在OD中按F7,单步以后就到了stolen bytes处
```

我们执行该脚本看看效果。

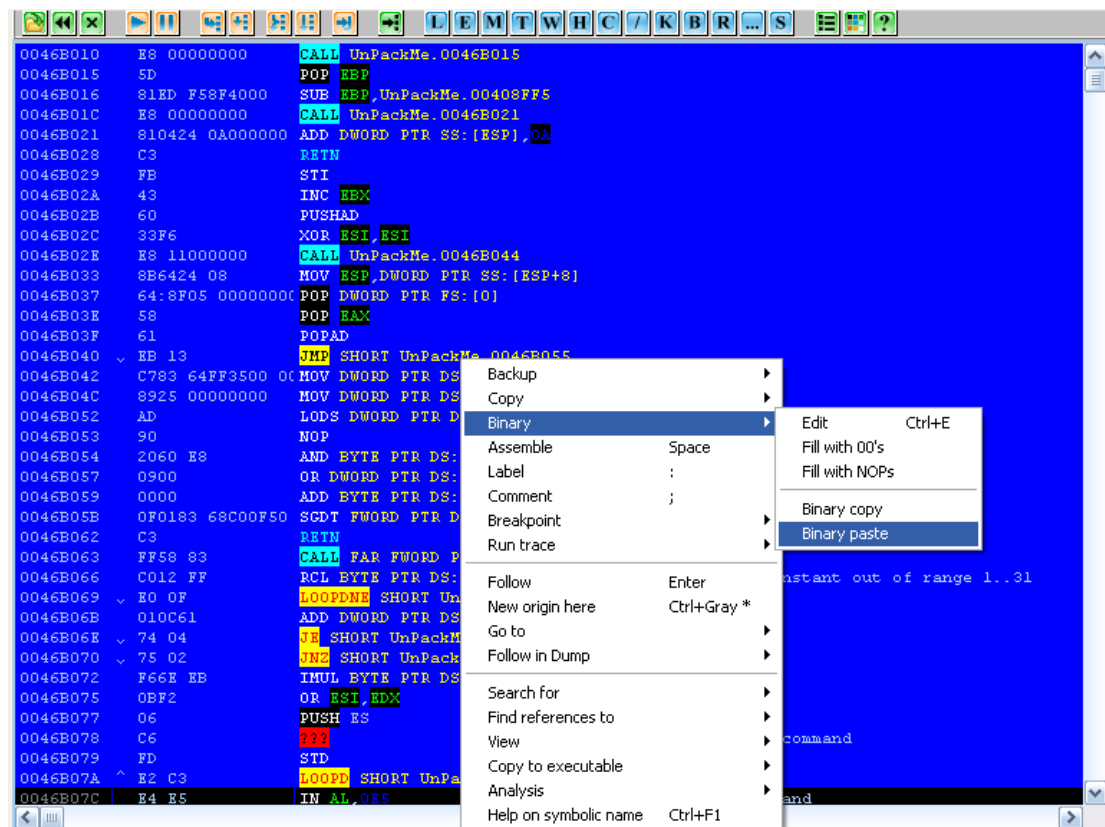


我们可以看到成功定位到了 stolen bytes 处。下面我们要做的就是将 stolen bytes 拷贝到入口点处。



这里我们从 8B0E48 开始拷贝,一直到 8B0EA4 为止,注意是二进制复制。

粘贴到入口点处。



Address	Hex	Assembly
0046B010	55	PUSH EBP
0046B011	8BEC	MOV EBP,ESP
0046B013	6A FF	PUSH -1
0046B015	68 60E4500	PUSH UnPackMe.00450E60
0046B01A	68 C8924200	PUSH UnPackMe.004292C8
0046B01F	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
0046B025	50	PUSH EAX
0046B026	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
0046B02D	83C4 A8	ADD ESP,-58
0046B030	53	PUSH EBX
0046B031	56	PUSH ESI
0046B032	57	PUSH EDI
0046B033	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
0046B036	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
0046B03C	33D2	XOR EDX,EDX
0046B03E	8AD4	MOV DL,AH
0046B040	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
0046B046	8BC8	MOV ECX,EAX
0046B048	81E1 FF000000	AND ECX,OFF
0046B04E	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX
0046B054	C1E1 08	SHL ECX,8
0046B057	03CA	ADD ECX,EDX
0046B059	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX
0046B05F	C1E8 10	SHR EAX,10
0046B062	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
0046B067	E8 FC84B7FF	CALL FFFE3568
0046B06C	85C0	TEST EAX,EAX
0046B06E	75 0A	JNZ SHORT UnPackMe.0046B07A
0046B070	75 02	JNZ SHORT UnPackMe.0046B074
0046B072	F66E EB	IMUL BYTE PTR DS:[ESI-15]
0046B075	0BF2	OR ESI,EDX
0046B077	06	PUSH ES
0046B078	C6	???

这里我们可以看到 46B067 处的这个 CALL 是一个间接 CALL。

Address	Hex	Assembly
008B0E91	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX
008B0E97	C1E8 10	SHR EAX,10
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
008B0E9F	E8 FC84B7FF	CALL UnPackMe.004293A0
008B0EA4	85C0	TEST EAX,EAX
008B0EA6	75 0A	JNZ SHORT 008B0EB2

这里原 stolen bytes 应该是 CALL 004293A0,目标地址是 004293A0。

但是由于这是一个间接 CALL,所以我们这里直接将其二进制复制到别的地方的话,目标地址就变了。

所以这个 CALL 被复制到别处的话,首先需要修正偏移量,我们来看看如何修正偏移量,首先我们在数据窗口中定位到该指令。

Address	Hex	Assembly
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
008B0E9F	E8 FC84B7FF	CALL UnPackMe.004293A0
008B0EA4	85C0	TEST EAX,EAX
008B0EA6	75 0A	JNZ SHORT 008B0EB2
008B0EA8	6A 1C	PUSH 1C
008B0EAA	E8 B164B7FF	CALL UnPackMe.00427360
008B0EAF	83C4 04	ADD ESP,4
008B0EB2	E8 3993B7FF	CALL UnPackMe.0042A1F0
008B0EB7	85C0	TEST EAX,EAX
008B0EB9	75 0A	JNZ SHORT 008B0EC5
008B0EBB	6A 10	PUSH 10
008B0EBD	E8 9E64B7FF	CALL UnPackMe.00427360
008B0EC2	83C4 04	ADD ESP,4
008B0EC5	C745 FC 00000000	MOV DWORD PTR SS:[EBP-4]

Backup
 Copy
 Binary
 Assemble Space
 Label :
 Comment ;
 Breakpoint
 Follow Enter
 New origin here Ctrl+Gray *
 Go to
 Follow in Dump
 Selection

Address	Hex dump
008B0E97	C1 E8 10 A3 28 E6
008B0E9F	E8 FC 84 B7 FF 85
008B0EA7	0A 6A 1C E8 B1 64
008B0EAF	83 C4 04 E8 39 93
008B0EF7	85 C8 75 8A 6A 10

这里我们不用考虑前面的操作码,直接看后面的 4 个字节的偏移量。

FF B7 84 FC,为了下面列公式方便,这里我们将其命名为 OPCODES。

008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
008B0E9F	E8 FC84B7FF	CALL UnPackMe.004293A0
008B0EA4	85C0	TEST EAX,EAX

这里我们将 008B0E9F,即这个 CALL 指令所在的地址命名为 DIR_CALL。

我们来算一下目标地址 004293A0 是如何得到的:

目标地址 = OPCODES + DIR_CALL + 5

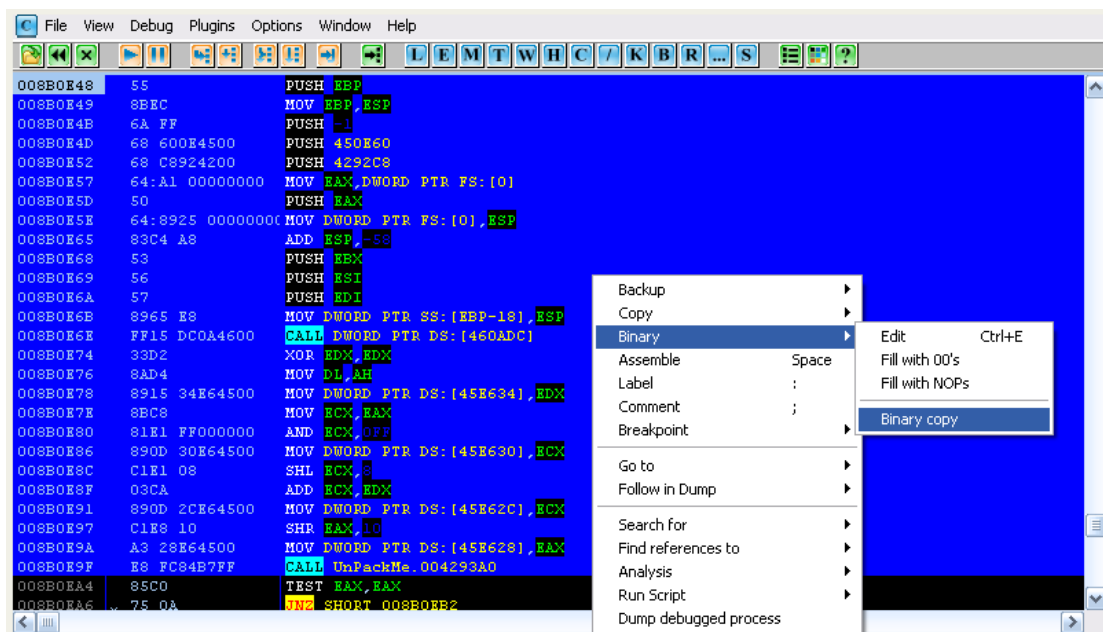
目标地址 = FFB784FC + 008B0E9F + 5 = 004293A0

好了,现在我们已经知道 004293A0 这个目标地址是如何得来的了。下面我们来计算新的 OPCODES。

目标地址 - CALL 指令新的地址 - 5 = 新的 OPCODES

何谓 CALL 指令新的地址:即该 CALL 指令被拷贝到的新的地址。

例如:



如果我们将 Stolen bytes 拷贝到入口点处。

0046B05F	C1E8 10	SHR EAX,10
0046B062	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
0046B067	E8 FC84B7FF	CALL FFFB3568
0046B06C	0C 61	OR AL,61

这里我们看到 46B067 这个地址。这个地址的计算公式如下:

原地址 - Stolen bytes 的起始地址 + 入口点

这里原地址为 008B0E9F。

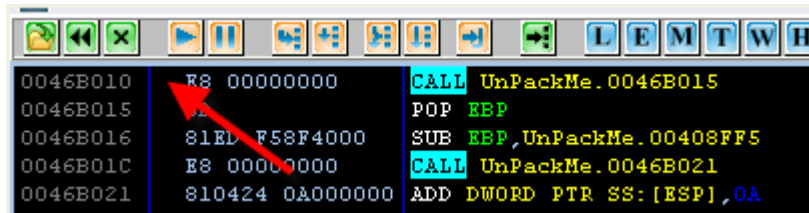
008B0E97	C1E8 10	SHR EAX,10
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
008B0E9F	E8 FC84B7FF	CALL UnPackMe.004293A0
008B0EA4	85C0	TEST EAX,EAX

stolen bytes 的起始地址 = 008B0E48



008B0E48	55	PUSH EBP
008B0E49	8BEC	MOV EBP, ESP
008B0E4B	6A FF	PUSH -1

入口点为 0046B010。



0046B010	E8 00000000	CALL UnPackMe.0046B015
0046B015		POP EBP
0046B016	81ED F58F4000	SUB EBP, UnPackMe.00408FF5
0046B01C	E8 00000000	CALL UnPackMe.0046B021
0046B021	810424 0A000000	ADD DWORD PTR SS:[ESP], 0A

$8B0E9F - 8B0E48 + 46B010 = 0046B067$

所以说新地址为 0046B067

好了,现在我们来计算新的 OPCODES。

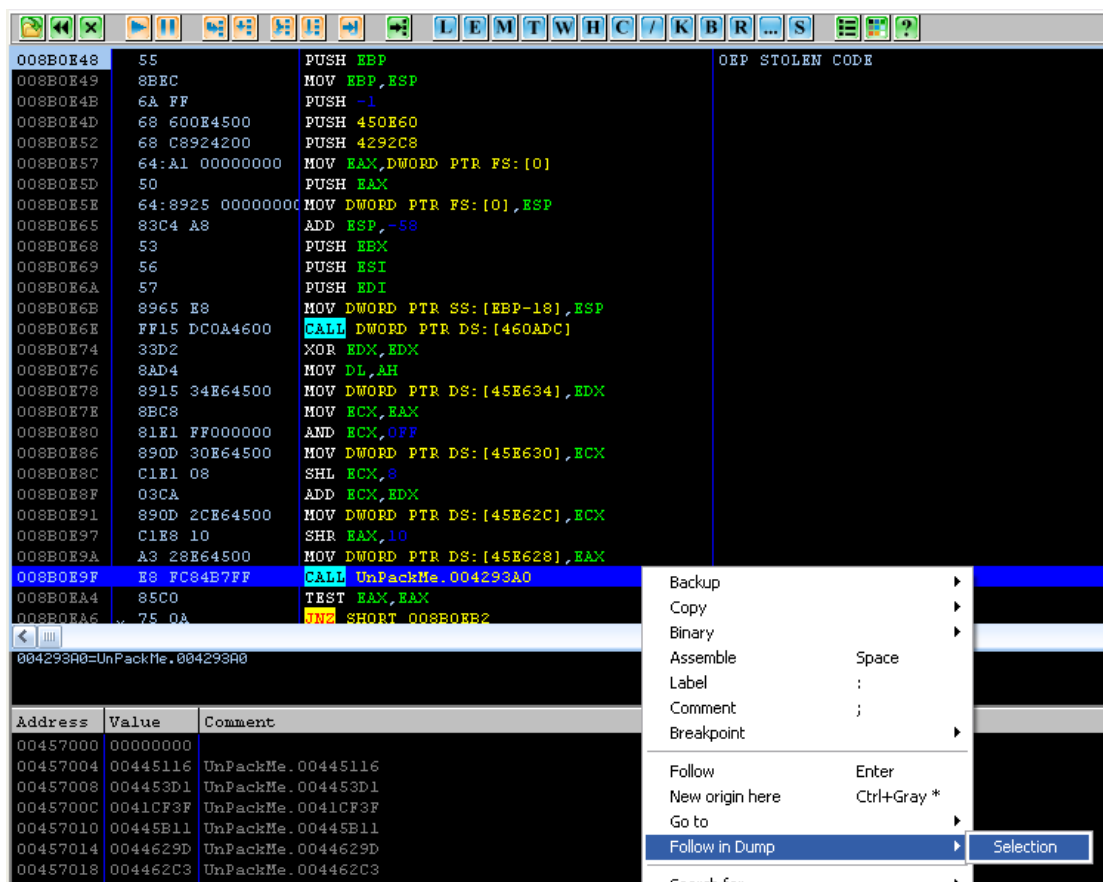
目标地址 - 新地址 - 5 = FFFBE334

$004293A0 - 0046B067 - 5 = FFFBE334$

这里新的 OPCODES 我们有了。

现在我们来手动编辑它。

我们定位到 stolen bytes 处。



008B0E48	55	PUSH EBP
008B0E49	8BEC	MOV EBP, ESP
008B0E4B	6A FF	PUSH -1
008B0E4D	68 600E4500	PUSH 450E60
008B0E52	68 C8924200	PUSH 4292C8
008B0E57	64:A1 00000000	MOV EAX, DWORD PTR FS:[0]
008B0E5D	50	PUSH EAX
008B0E5E	64:8925 00000000	MOV DWORD PTR FS:[0], ESP
008B0E65	83C4 A8	ADD ESP, -58
008B0E68	53	PUSH EBK
008B0E69	56	PUSH ESI
008B0E6A	57	PUSH EDI
008B0E6E	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP
008B0E6E	FF15 DCOA4600	CALL DWORD PTR DS:[460ADC]
008B0E74	33D2	XOR EDX, EDX
008B0E76	8AD4	MOV DL, AH
008B0E78	8915 34E64500	MOV DWORD PTR DS:[45E634], EDX
008B0E7E	8EC8	MOV ECX, EAX
008B0E80	81E1 FF000000	AND ECX, 0FF
008B0E86	890D 30E64500	MOV DWORD PTR DS:[45E630], ECX
008B0E8C	C1E1 08	SHL ECX, 8
008B0E8F	03CA	ADD ECX, EDX
008B0E91	890D 2CE64500	MOV DWORD PTR DS:[45E62C], ECX
008B0E97	C1E8 10	SHR EAX, 10
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628], EAX
008B0E9F	E8 FC84B7FF	CALL UnPackMe.004293A0
008B0EA4	85C0	TEST EAX, EAX
008B0EA6	75 0A	JNZ SHORT 008B0EB2

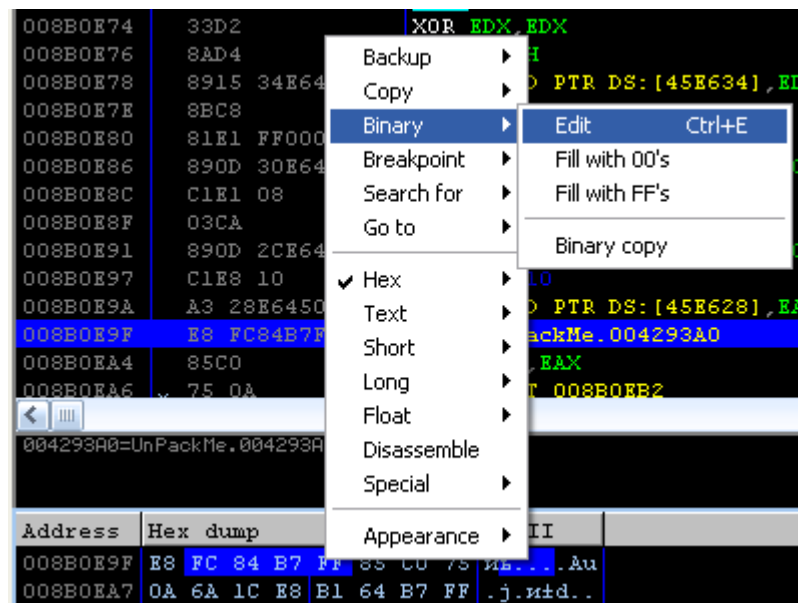
004293A0=UnPackMe.004293A0

Address	Value	Comment
00457000	00000000	
00457004	00445116	UnPackMe.00445116
00457008	004453D1	UnPackMe.004453D1
0045700C	0041CF3F	UnPackMe.0041CF3F
00457010	00445B11	UnPackMe.00445B11
00457014	0044629D	UnPackMe.0044629D
00457018	004462C3	UnPackMe.004462C3

我们在数据窗口中定位到这个 CALL:

Address	Hex dump	ASCII
008B0E9F	E8 FC 84 B7 FF 85 C0 75	иъ...Au
008B0EA7	0A 6A 1C E8 B1 64 B7 FF	.j.иtd..
008B0EAF	83 C4 04 E8 39 93 B7 FF	.Д.и9...

这里我们跳过 E8 这个机器码,直接修改后面的 4 个字节的偏移量:



Edit data at 008B0EA0

ASCII: ù . . .

UNICODE: □□

HEX +00: FC 84 B7 FF

☒ Keep size

OK Cancel

将其替换成新的 OPCODES.

Edit data at 008B0EA0

ASCII: 4ãû.

UNICODE: □□

HEX +04: 34 E3 FB FF

☒ Keep size

OK Cancel

008B0E91	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX
008B0E97	C1E8 10	SHR EAX,10
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
008B0E9F	E8 34E3FBFF	CALL 0086F1D8
008B0EA4	85C0	TEST EAX,EAX

我们可以看到该 CALL 的 OPCODE 已经改变了,现在我们将 stolen bytes 拷贝到入口点处。

008B0E48	55	PUSH EBP	ORP STOLEN CODE
008B0E49	8BEC	MOV EBP,ESP	
008B0E4B	6A FF	PUSH -1	
008B0E4D	68 600E4500	PUSH 450E60	
008B0E52	68 C8924200	PUSH 4292C8	
008B0E57	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
008B0E5D	50	PUSH EAX	
008B0E5E	64:8925 00000000	MOV DWORD PTR FS:[0],ESP	
008B0E65	83C4 A8	ADD ESP,-58	
008B0E68	53	PUSH EBX	
008B0E69	56	PUSH ESI	
008B0E6A	57	PUSH EDI	
008B0E6B	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
008B0E6E	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]	
008B0E74	33D2	XOR EDX,EDX	
008B0E76	8AD4	MOV DL,AH	
008B0E78	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
008B0E7E	8BC8	MOV ECX,EAX	
008B0E80	81E1 FF000000	AND ECX,OFF	
008B0E86	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX	
008B0E8C	C1E1 08	SHL ECX,8	
008B0E8F	03CA	ADD ECX,EDX	
008B0E91	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX	
008B0E97	C1E8 10	SHR EAX,10	
008B0E9A	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX	
008B0E9F	E8 34E3FBFF	CALL 0086F1D8	
008B0EA4	85C0	TEST EAX,EAX	

0046B010	55	PUSH EBP
0046B011	8BEC	MOV EBP,ESP
0046B013	6A FF	PUSH -1
0046B015	68 600E4500	PUSH UnPackMe.00450E60
0046B01A	68 C8924200	PUSH UnPackMe.004292C8
0046B01F	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
0046B025	50	PUSH EAX
0046B026	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
0046B02D	83C4 A8	ADD ESP,-58
0046B030	53	PUSH EBX
0046B031	56	PUSH ESI
0046B032	57	PUSH EDI
0046B033	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
0046B036	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
0046B03C	33D2	XOR EDX,EDX
0046B03E	8AD4	MOV DL,AH
0046B040	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
0046B046	8BC8	MOV ECX,EAX
0046B048	81E1 FF000000	AND ECX,OFF
0046B04E	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX
0046B054	C1E1 08	SHL ECX,8
0046B057	03CA	ADD ECX,EDX
0046B059	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX
0046B05F	C1E8 10	SHR EAX,10
0046B062	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
0046B067	E8 34E3FBFF	CALL UnPackMe.004293A0
0046B06C	0C 61	OR AL,61
0046B06E	74 04	JF SHORT UnPackMe.0046B074
0046B070	75 02	JNZ SHORT UnPackMe.0046B074
0046B073	E65E FD	MOV DWORD PTR DS:[45E61F],ESI

我们可以看到 46B067 处的 CALL 的目标地址这次正确了。

下面我们给脚本添加一些内容让其自动完成上述操作。

```
var dir_CALL
var oep
var StartScan
var Opcodes
var temp
var temp2
var temp3
```

然后

```
last:
findop eip, #FFE0h // 从最后一次异常处开始搜索 JMP EAX指令, 以便下面定位 stolen bytes
mov dir_JMP, $RESULT // 将 JMP EAX指令的地址保存到变量 dir_JMP 中
bp dir_JMP // 对 JMP EAX指令设置一个断点
esto // 忽略掉异常继续执行, 相当于在 0D 中按了 SHIFT+F9
bc dir_JMP // 删除掉 JMP EAX指令处的断点
sti // 单步入入, 相当于在 0D 中按 F7, 单步以后就到了 stolen bytes 处

mov oep, eip // 将 stolen bytes 的起始地址保存到变量 oep 中
mov StartScan, eip // 将 stolen bytes 的起始地址保存到变量 StartScan 中

LookForCall: // 开始搜索 Stolen bytes 中需要修正偏移量的 CALL, 修正完 CALL 以后, 将 stolen bytes 拷贝到入口点处

findop StartScan, #E8h // 搜索以机器码 E8 开头的 CALL 指令, 即待修正偏移量的 CALL 指令
cmp $RESULT, 0 // 判断是否搜索到了待修正偏移量的 CALL 指令

je final // 没有搜索的话, 则跳转到 final 标签处
mov dir_CALL, $RESULT // 将待修正偏移量 CALL 的地址保存到变量 dir_CALL 中
mov StartScan, $RESULT // 将待修正偏移量的 CALL 指令的地址赋值给变量 StartScan
add dir_CALL, 1 // 指向偏移量
mov Opcodes, [dir_CALL] // 获取待修正的偏移量并保存到变量 Opcodes 中
add Opcodes, StartScan // 将偏移量加上 CALL 指令所在的地址
add Opcodes, 5 // 加上 CALL 指令的长度
// 这样就得到了 CALL 指令的目标地址

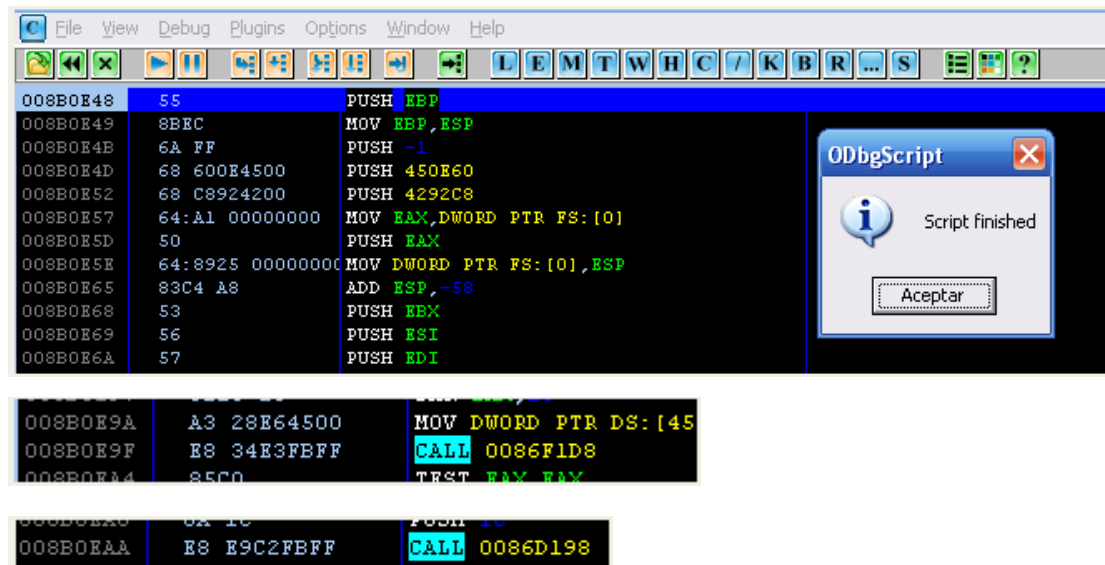
// 修正 CALL 指令的偏移量

mov temp, StartScan // 将 CALL 指令的地址 (即 stolen bytes 中的最后一条指令) 保存到临时变量 temp 中
sub temp, oep // 计算 stolen bytes 所有指令占的总长度, 将该总长度保存到临时变量 temp 中
mov temp2, Newoep // 将入口点的值保存到临时变量 temp2 中
add temp, temp2 // 计算 CALL 指令新的地址, 并保存到变量 temp 中
sub Opcodes, temp // 将目标地址减去 CALL 指令新的地址
sub Opcodes, 5 // 然后减去 5, 就得到了 CALL 指令的修正后的偏移量

edit: // 将 CALL 指令的偏移量修正
mov temp3, StartScan // 将 CALL 指令所在的地址保存到临时变量 temp3 中
add temp3, 1 // 指向待修正的偏移量
mov [temp3], Opcodes // 修正偏移量
jmp LookForCall

final:
ret
```

执行该脚本。



008B0EBB	6A 10	PUSH 10
008B0EBD	E8 D6C2FBFF	CALL 0086D198
008B0EC2	83C4 04	ADD ESP,4
008B0EC5	C745 FC 00000000	MOV DWORD PTR SS:[EBP-4],0
008B0ECC	E8 27EDFBFF	CALL 0086FBF8
008B0ED1	E8 B2D2FBFF	CALL 0086E188
008B0ED6	FF15 84094600	CALL DWORD PTR DS:[460984]
008B0EDC	A3 D8EB4500	MOV DWORD PTR DS:[45EBD8],EAX
008B0EE1	E8 D255FCFF	CALL 008764B8

这里我们可以看到执行了该脚本后,下面 CALL 的偏移量都被修正了。

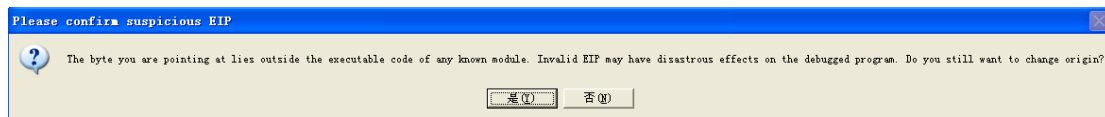
下面我们要做的就是将 stolen bytes 二进制复制到入口点处。

0046B010	55	PUSH EBP
0046B011	8BEC	MOV EBP,ESP
0046B013	6A FF	PUSH -1
0046B015	68 600E4500	PUSH UnPackMe.00450E60
0046B01A	68 C8924200	PUSH UnPackMe.004292C8
0046B01F	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
0046B025	50	PUSH EAX
0046B026	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
0046B02D	83C4 A8	ADD ESP,-58
0046B030	53	PUSH EBX
0046B031	56	PUSH ESI
0046B032	57	PUSH EDI
0046B033	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
0046B036	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
0046B03C	33D2	XOR EDX,EDX
0046B03E	8AD4	MOV DL,AH
0046B040	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
0046B046	8BC8	MOV ECX,EAX
0046B048	81E1 FF000000	AND ECX,OFF
0046B04E	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX
0046B054	C1E1 08	SHL ECX,8
0046B057	03CA	ADD ECX,EDX
0046B059	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX
0046B05F	C1E8 10	SHR EAX,10
0046B062	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX
0046B067	E8 34E3FBFF	CALL UnPackMe.004293A0
0046B06C	85C0	TEST EAX,EAX
0046B06E	75 0A	JNZ SHORT UnPackMe.0046B07A
0046B070	6A 1C	PUSH 1C
0046B072	E8 E9C2FBFF	CALL UnPackMe.00427360
0046B077	83C4 04	ADD ESP,4
0046B07A	E8 71F1FBFF	CALL UnPackMe.0042A1F0
0046B07F	85C0	TEST EAX,EAX
0046B081	75 0A	JNZ SHORT UnPackMe.0046B08D

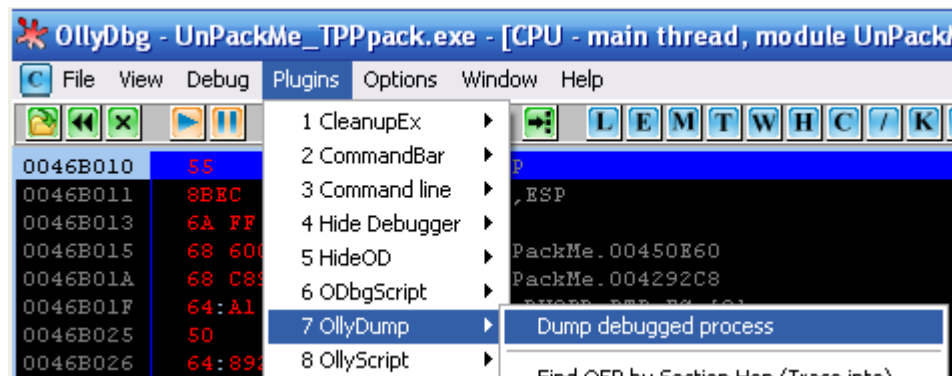
好,现在我们定位到了入口点处,我们将 EIP 修改到入口点处。

0046B010	55	PUSH EBP
0046B011	8BEC	MOV EBP,ESP
0046B013	6A FF	PUSH -1
0046B015	68 600E4500	PUSH UnPackMe.00450E60
0046B01A	68 C8924200	PUSH UnPackMe.004292C8
0046B01F	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
0046B025	50	PUSH EAX
0046B026	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
0046B02D	83C4 A8	ADD ESP,-58
0046B030	53	PUSH EBX
0046B031	56	PUSH ESI
0046B032	57	PUSH EDI
0046B033	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
0046B036	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
0046B03C	33D2	XOR EDX,EDX

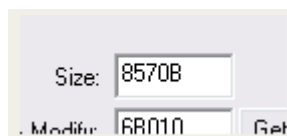
Backup ▶
 Copy ▶
 Binary ▶
 Undo selection Alt+BkSp
 Assemble Space
 Label ;
 Comment ;
 Breakpoint ▶
 Run trace ▶
 New origin here Ctrl+Gray *
 Go to ▶



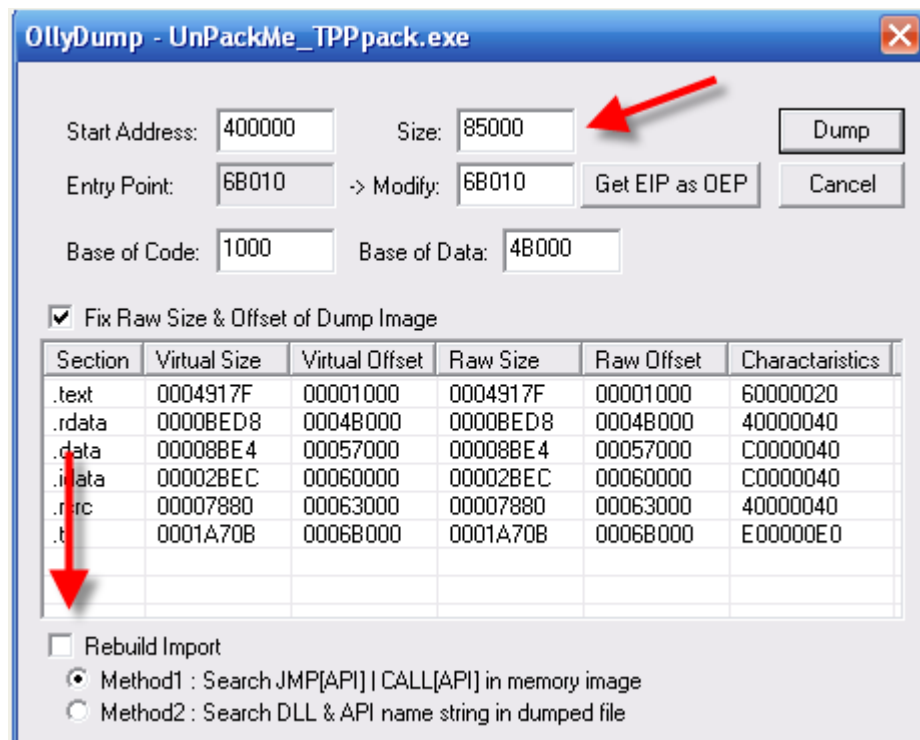
我们选择是,下面来进行 dump。



Martian 先生的教程中提到了,这个大小也得修改,不然单击 Dump 按钮,会报错。

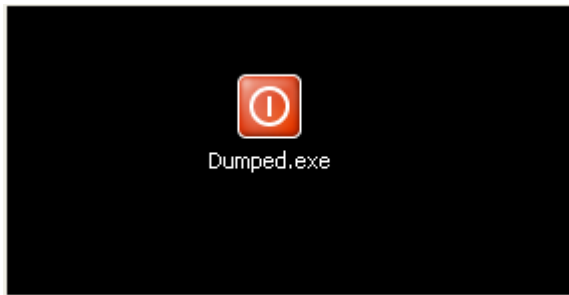


修改为:



这里我不使用 OllyDump 来修复 IAT,所以我去掉了 Rebuild Import 的对勾。

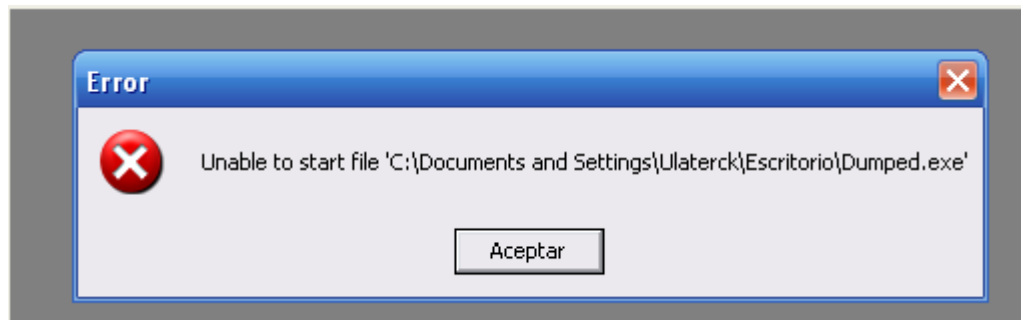
然后按 dump 按钮进行 dump。



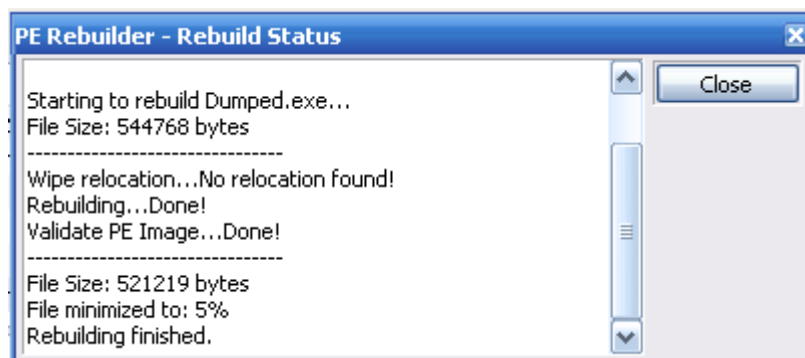
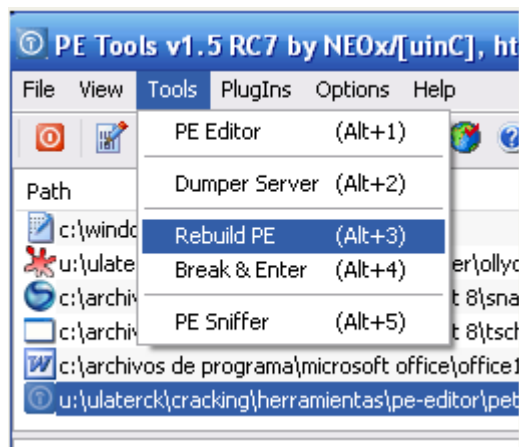
好了,这里我们就 dump 完成了,但是肯定是无法正常运行的,因为 IAT 还没有修复。

下面我们来修复 IAT。

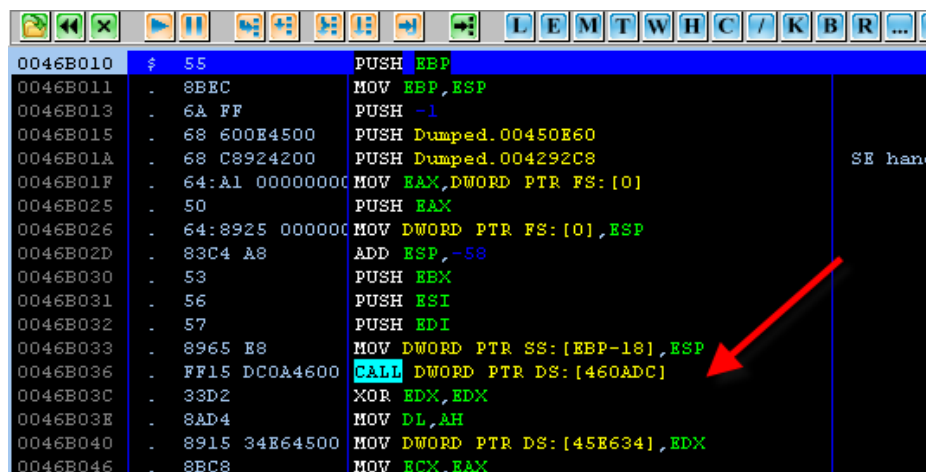
如果我们直接用 OD 加载 dump 文件的话,直接就会报错。



我们尝试用 PE 编辑工具重建 PE 看看。



好,重建 PE 完毕了,我们再次用 OD 加载它。



这里提一句,HideOD 这款插件有时候会出错导致程序正常运行,所以最好将其用 HideDebugger 和 OllyAdvanced 代替。

好了,第一个脚本已经给大家介绍完了,接下来给大家介绍第二个脚本。

```
var base
```

```
var dir_VirtualAlloc
```

```
var dir_VirtualProtect
```

```
var dir_mov
```

Initiation:

```
gpa "VirtualAlloc", "kernel32.dll"           //获取 VirtualAlloc 这个 API 函数的地址
```

```
mov dir_VirtualAlloc, $RESULT
```

```
log dir_VirtualAlloc
```

```
gpa "VirtualProtect", "kernel32.dll"
```

```
mov dir_VirtualProtect, $RESULT
```

```
bp dir_VirtualAlloc
```

```
run
```

```
eob info
```

info:

```
mov base, eax
```

```
log base
```

```
bc dir_VirtualAlloc
```

```
bp dir_VirtualProtect
```

Area:

```
eob Section
```

```
run
```

Section:

```
cmp esi, 00460000
```

```
je Return
```

```
jmp Area
```

Return:

```
bc dir_VirtualProtect
```

```
mov Reg_esp, [esp]
```

```
bp Reg_esp
```

```
eob Area_1
```

```
run
```

Area_1:

```
bc Reg_esp
```

```
find base, #897C24188B4424#
```

```
mov dir_mov, $RESULT
```

```
log dir_mov
```

```
jmp Nop
```

Nop:

```
bp dir_mov
```

```
eob Nop2
```

```
run
```

Nop2:

```
bc dir_mov
```

```
fill dir_mov, 4, 90
```

final:

```
msg "NOP 完毕,请按 F9 键运行."
```

```
ret
```

附第二个脚本截图:


```

0000 var base
0001 var dir_VirtualAlloc
0002 var dir_VirtualProtect
0003 VAR dir_mov
0004
0005 Initiation:
0006
0007 gpa "VirtualAlloc", "kernel32.dll" //获取VirtualAlloc这个API函数的地址
0008 mov dir_VirtualAlloc, $RESULT
0009 log dir_VirtualAlloc
0010
0011 gpa "VirtualProtect", "kernel32.dll"
0012 mov dir_VirtualProtect, $RESULT
0013
0014 bp dir_VirtualAlloc
0015 run
0016 eob info
0017
0018 info:
0019
0020 mov base, eax
0021 log base
0022 bc dir_VirtualAlloc
0023 bp dir_VirtualProtect
0024
0025 Area:
0026 eob Section
0027 run
0028
0029 Section:
0030 cmp esi, 00460000
0031 je Return
0032 jmp Area
0033
0034 Return:
0035 bc dir_VirtualProtect
0036 mov Reg_esp, [esp]
0037 bp Reg_esp
0038
0039 eob Area_1
0040 run
0041
0042 Area_1:
0043 bc Reg_esp
0044 find base, #097C24188B4424#

0045 mov dir_mov, $RESULT
0046 log dir_mov
0047 jmp Nop
0048
0049 Nop:
0050 bp dir_mov
0051 eob Nop2
0052 run
0053 Nop2:
0054 bc dir_mov
0055 fill dir_mov, 4, 90
0056
0057 final:
0058
0059 msg "按F9键运行."
0060
0061 ret

```

这是修复 IAT 其中一个比较经典的方法,Martian 先生在他的教程中详细介绍过。在到达 OEP 之前我们可以对 IAT 中重定向的项设置内存写入断点,断下来的地方就是写入重定向值的地方。但是要对 IAT 进行写入的话,首先得让 IAT 所在的内存单元具有写入权限,所以势必会调用 VirtualProtect 来修改 IAT 所在内存单元的内存访问属性,赋予其写入权限。所以我们可以先对 VirtualProtect 这个 API 函数设置一个断点,等执行完该函数赋予写入权限以后,我们再对 IAT 中重定向的项设置内存写入断点。

00127A60	10007965	CALL to VirtualProtect from 1000795F
00127A64	00460000	Address = UnPackMe.00460000
00127A68	00002BEC	Size = 2BEC (11244.)
00127A6C	00000040	NewProtect = PAGE_EXECUTE_READWRITE
00127A70	00127BE0	pOldProtect = 00127BE0
00127A74	00000002	
00127A78	00460613	ASCII "erm"

从 Martian 先生教程中这张截图我们可以看到断在了 VirtualProtect 这个 API 函数的入口处,其想修改 IAT 所在内存单元的访问属性。我们执行到返回,然后对重定向的 IAT 项设置内存写入断点,接着运行起来,断在了写入重定向值的地方。

10005CE9	894C37 02	MOV DWORD PTR DS:[EDI+ESI+2],ECX
10005CED	66:C74437 06 C	MOV WORD PTR DS:[EDI+ESI+6],EC7
10005CF4	894C37 08	MOV DWORD PTR DS:[EDI+ESI+8],ECX
10005CF8	894437 0C	MOV DWORD PTR DS:[EDI+ESI+C],EAX
10005CFC	C64437 10 C3	MOV BYTE PTR DS:[EDI+ESI+10],0C3
10005D01	897C24 18	MOV DWORD PTR SS:[ESP+18],EDI
10005D05	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]
10005D09	8945 00	MOV DWORD PTR SS:[EBP],EAX
10005D0C	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]
10005D10	8B78 04	MOV EDI,DWORD PTR DS:[EAX+4]
10005D13	83C0 04	ADD EAX,4
10005D16	83C5 04	ADD EBP,4
10005D19	85FF	TEST EDI,EDI
10005D1B	894424 24	MOV DWORD PTR SS:[ESP+24],EAX
10005D1F	896C24 48	MOV DWORD PTR SS:[ESP+48],EBP
10005D23	^ 0F85 CAFDFFFF	JNZ 10005AF3
10005D29	834424 2C 14	ADD DWORD PTR SS:[ESP+2C],14
10005D2E	8B6C24 2C	MOV EBP,DWORD PTR SS:[ESP+2C]
10005D32	8B8C24 98030000	MOV EDI,DWORD PTR SS:[ESP+398]
10005D39	^ E9 E5FCFFFF	JMP 10005A23
10005D3E	5E	POP ESI
10005D3F	5B	POP EBX
10005D40	5F	POP EDI
10005D41	33C0	XOR EAX,EAX
10005D43	5D	POP EBP
10005D44	81C4 84030000	ADD ESP,384
10005D4A	C2 0400	RETN 4

这里就断在了写入重定向值的地方,仔细观察我们可以知道此时 EAX 中保存了重定向的值,而 EBP 指向的是对应的 IAT 项。

现在我们在前面几行处设置一个断点,跟踪一下,看看是什么情况。

10005CF4	894C37 08	MOV DWORD PTR DS:[EDI+ESI+8],ECX	
10005CF8	894437 0C	MOV DWORD PTR DS:[EDI+ESI+C],EAX	
10005CFC	C64437 10 C3	MOV BYTE PTR DS:[EDI+ESI+10],0C3	
10005D01	897C24 18	MOV DWORD PTR SS:[ESP+18],EDI	;Machaca valor bueno
10005D05	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]	
10005D09	8945 00	MOV DWORD PTR SS:[EBP],EAX	;Escribe en la IAT
10005D0C	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]	
10005D10	8B78 04	MOV EDI,DWORD PTR DS:[EAX+4]	
10005D13	83C0 04	ADD EAX,4	

我们可以看到前面几行会获取正确的 IAT 值,而接下来会将正确的 IAT 值覆盖为重定向的值。所以我们要做的就是将写入重定向值的语句 NOP 掉。

所以脚本要做的事情就是定位到写入重定向值的指令,并将其 NOP 掉。

首先脚本要做的第一件事情就是查找 VirtualAlloc 这个 API 函数的地址,首次断到 VirtualAlloc 这个 API 函数时,我们就可以获取需要 NOP 掉的指令所在内存单元的首地址了,因为在不同的机器上,待 NOP 掉的指令的地址是会变的,所以我们有必要动态获取它。

```
var base
```

```
var dir_VirtualAlloc
```

```
var dir_VirtualProtect
```

```
var dir_mov
```

```
Initiation:
```

```
gpa "VirtualAlloc", "kernel32.dll" //获取 VirtualAlloc 这个 API 函数的地址
```

```
mov dir_VirtualAlloc, $RESULT
```

```
log dir_VirtualAlloc
```

这里是获取 VirtualAlloc 这个 API 函数的地址,然后将其保存到变量 dir_VirtualAlloc 中,同理 VirtualProtect 也是一样。

```
gpa "VirtualProtect", "kernel32.dll"
```

```
mov dir_VirtualProtect, $RESULT
```

获取 VirtualProtect 这个 API 函数的地址,然后将其保存到变量 dir_VirtualProtect 中。

```
bp dir_VirtualAlloc
```

```
run
```

```
eob info
```

```
info:
```

```
mov base, eax
log base
bc dir_VirtualAlloc
bp dir_VirtualProtect
```

接着对 VirtualAlloc 设置断点,运行起来,如果断下来就跳转到info 标签处,将该程序刚申请的内存单元的首地址保存到变量 base 中,下面我们需要 NOP 掉的指令将位于这块内存单元中。
接下来删除掉 VirtualAlloc 的断点,然后对 VirtualProtect 设置断点。

```
Area:
eob Section
run
```

```
Section:
cmp esi, 00460000
je Return
jmp Area
```

断下来了的话,判断 ESI 的值是否等于 460000(IAT 的起始地址),因为该程序会调用 VirtualProtect 修改 IAT 所在内存单元的访问属性。如果 ESI 等于 460000 的话,就跳转到 return 标签处。

```
Return:
bc dir_VirtualProtect
mov Reg_esp, [esp]
bp Reg_esp
```

删除掉 VirtualProtect 的断点,将 ESP 指向的内容(返回地址)保存到变量 Reg_esp 中,接着对返回地址处设置断点。

```
Area_1:
bc Reg_esp
find base, #897C24188B4424#
mov dir_mov, $RESULT
log dir_mov
jmp Nop
```

当 VirtualProtect 调用返回后,下面就可以在之前申请的内存单元中搜索需要 NOP 掉的指令了。这里我们利用特征码来搜索。
特征码为:897C24188B4424。

10005CF4	894C37 08	MOV DWORD PTR DS:[EDI+ESI+8],ECX	
10005CF8	894437 0C	MOV DWORD PTR DS:[EDI+ESI+C],EAX	
10005CFC	664437 10 C3	MOV BYTE PTR DS:[EDI+ESI+10],0C3	
10005D01	897C24 18	MOV DWORD PTR SS:[ESP+18],EDI	;Machaca valor bueno
10005D05	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]	
10005D09	8345 00	MOV DWORD PTR SS:[EBP],EAX	;Escribe en la IAT
10005D0D	8B4424 24	MOV EAX,DWORD PTR SS:[ESP+24]	
10005D11	8B78 04	MOV EDI,DWORD PTR DS:[EAX+4]	
10005D15	83C0 04	ADD EAX,4	

搜索到了的话就跳转到 Nop 标签处。

```
Nop:
bp dir_mov
eob Nop2
run
```

```
Nop2:
bc dir_mov
fill dir_mov, 4, 90
```

final:

```
msg "NOP 完毕,请按 F9 键运行."
```

```
ret
```

好了,这里第二个脚本也介绍完了。感谢 Ularteck 童鞋提供的这两个脚本以及 Martian 先生提供的教程。我从 Martian 先生的教程中截了一张图片来解释第二个脚本。

本章,大家应该对于如何编写脚本更加了解了吧。

好,本章就到这里。