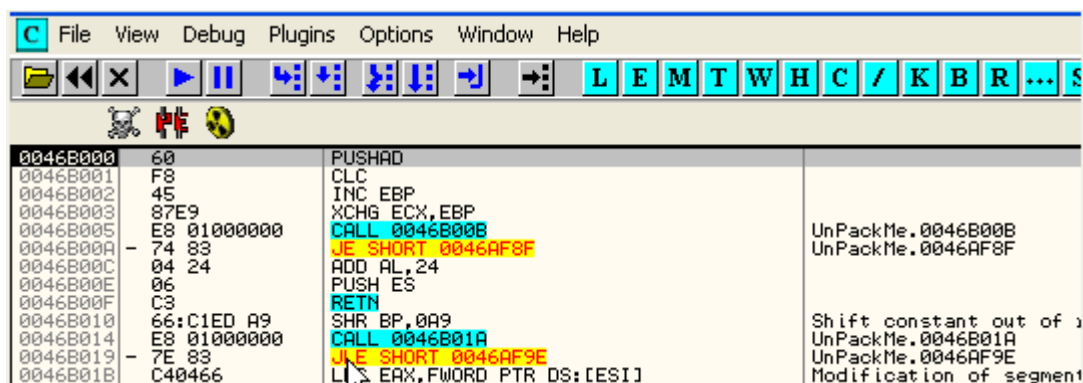


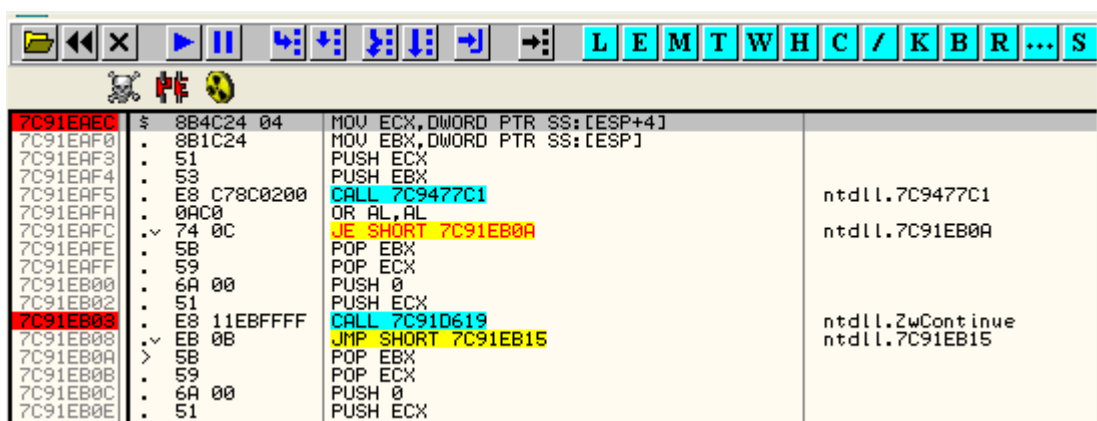
第四十四章-ACProtect V1.09 脱壳(修复 AntiDump)

我们上一章节介绍了如何定位 stolen bytes,以及 IAT 的修复。我们利用上一章编写的脚本可以很方便的修复 IAT 并且定位到 OEP, 接下来我们的任务就是 dump。



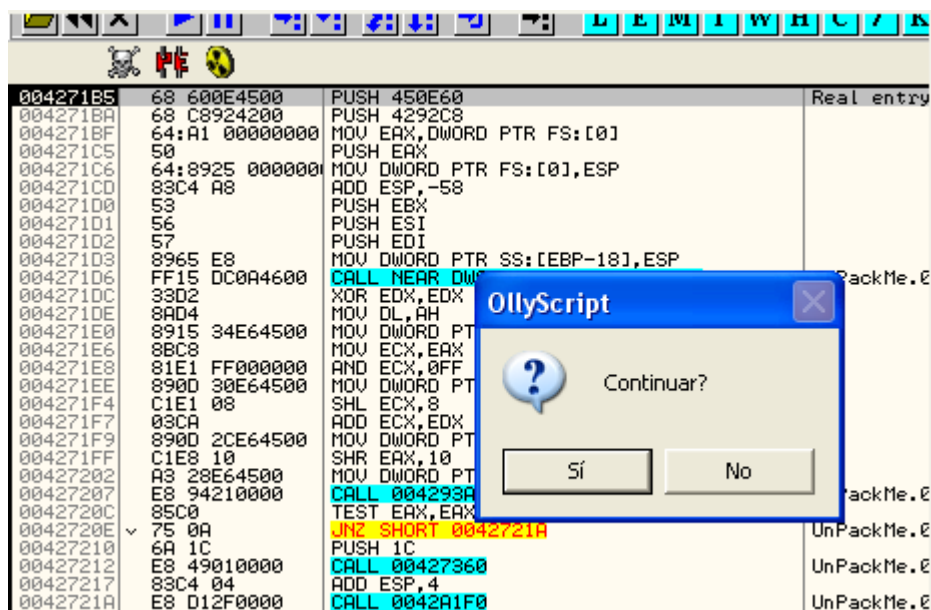
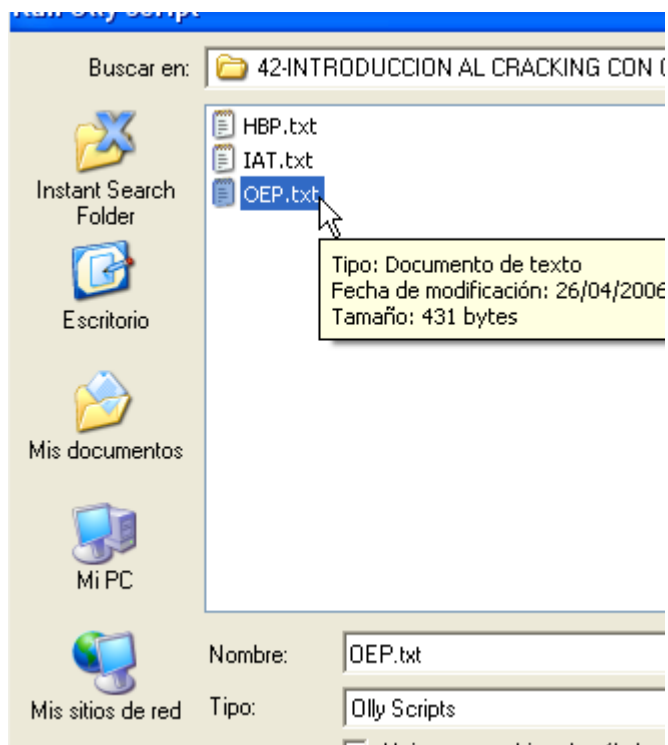
Address	Disassembly	Comment
0046B000	PUSHAD	
0046B001	CLC	
0046B002	INC EBP	
0046B003	XCHG ECX,EBP	
0046B005	CALL 0046B00B	UnPackMe.0046B00B
0046B00A	JE SHORT 0046AF8F	UnPackMe.0046AF8F
0046B00C	ADD AL,24	
0046B00E	PUSH ES	
0046B00F	RETN	
0046B010	SHR BP,0A9	Shift constant out of
0046B014	CALL 0046B01A	UnPackMe.0046B01A
0046B019	JLE SHORT 0046AF9E	UnPackMe.0046AF9E
0046B01B	LFS EAX,FWORD PTR DS:[ESI]	Modification of segment

我们依然用 OD 加载 UnPackMe_ACProtect1.09,并对 KiUserExceptionDispatcher 入口以及下方的 ZwContinue 调用处分别设置断点, 并且清除之前设置的硬件断点。

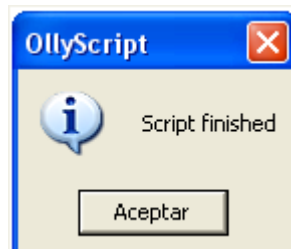


Address	Disassembly	Comment
7C91EAF0	MOV ECX,DWORD PTR SS:[ESP+4]	
7C91EAF3	MOV EBX,DWORD PTR SS:[ESP]	
7C91EAF4	PUSH ECX	
7C91EAF5	PUSH EBX	
7C91EAF5	CALL 7C9477C1	ntdll.7C9477C1
7C91EAF8	OR AL,AL	
7C91EAFB	JE SHORT 7C91EB0A	ntdll.7C91EB0A
7C91EAFE	POP EBX	
7C91EAFF	POP ECX	
7C91EB00	PUSH 0	
7C91EB02	PUSH ECX	
7C91EB03	CALL 7C91D619	ntdll.ZwContinue
7C91EB08	JMP SHORT 7C91EB15	ntdll.7C91EB15
7C91EB0A	POP EBX	
7C91EB0B	POP ECX	
7C91EB0C	PUSH 0	
7C91EB0E	PUSH ECX	

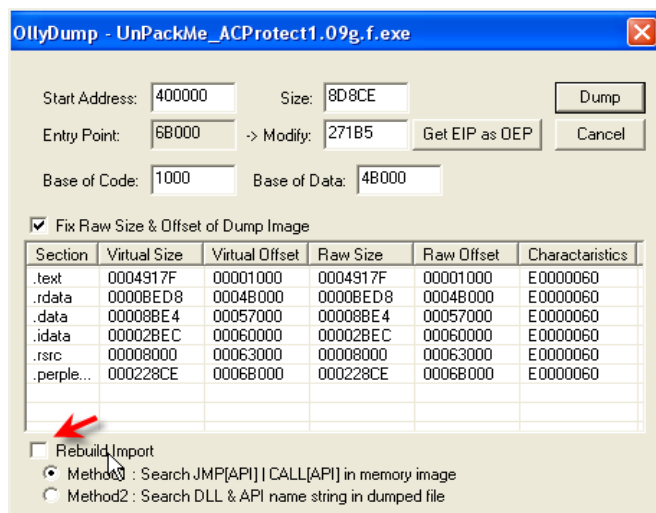
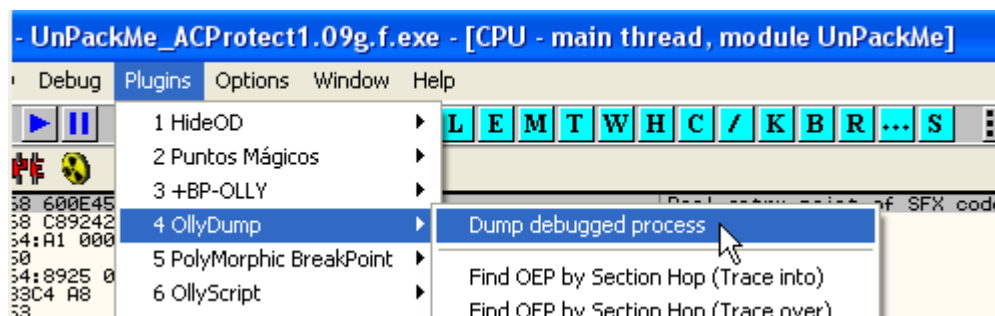
接着我们执行 OEP.txt 这个脚本定位到 OEP 处。



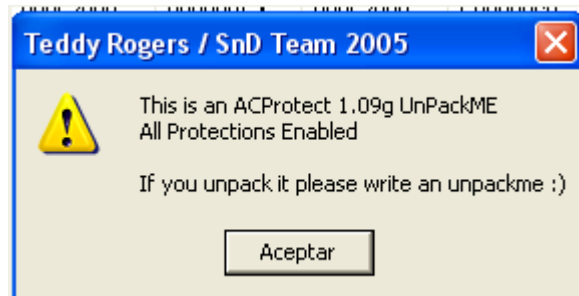
好了,现在我们到达了 OEP 处。



接下来进行 dump。



这里我们不勾选 Rebuild Import 的选项,将转储出来的文件重命名为 dumped.exe。接下来我们重启 OD,执行修复 IAT 的脚本。



接下来打开 Import Reconstructor,定位到该程序所在进程。

接着就需要填上 OEP(RVA),IAT 起始地址(RVA),IAT 的大小。

OEP = 271B5

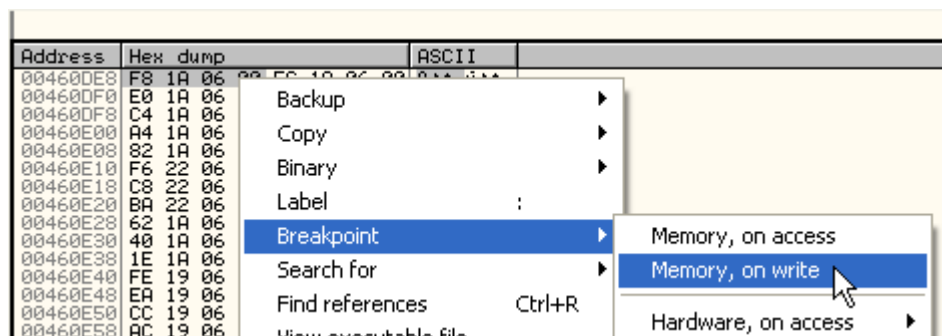
IAT 的起始地址(RVA) = 60818

IAT 的大小 = 460F28 - 460818 = 710

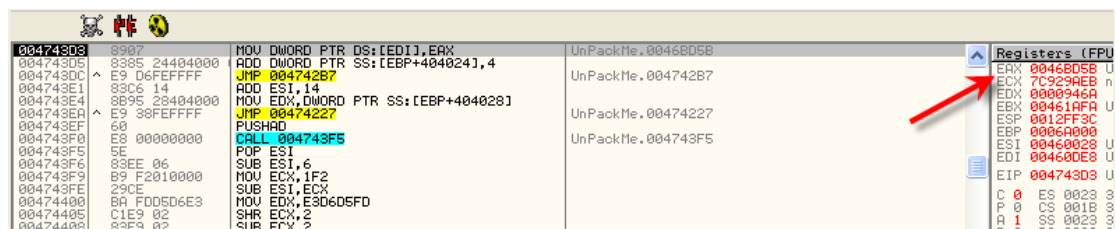
点击 Get Imports。

我们会发现有一项是无效的,其他项都是有效的,是不是我们哪里处理的有问题?

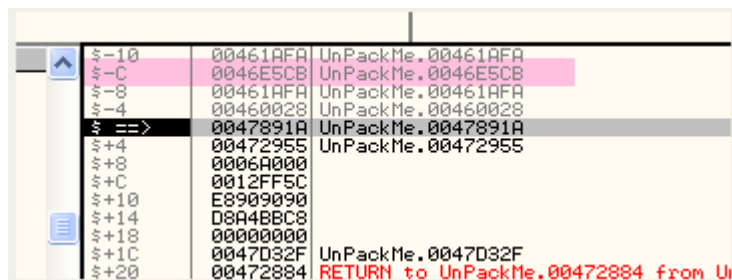
我们来跟一下,看看哪里出了问题,重启 OD,我们选中上面显示无效的 IAT 项,设置内存写入断点。



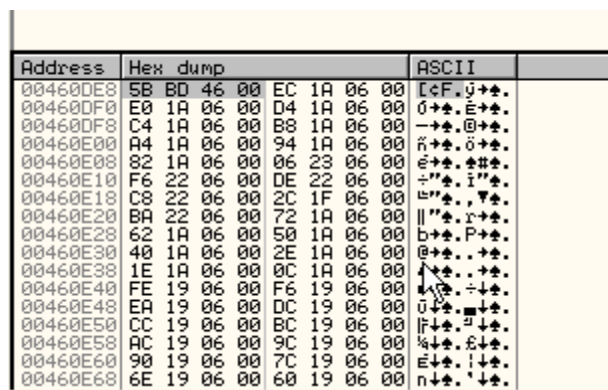
运行起来,断在了尝试对 460DE8 该内存单元进行写入的指令处。



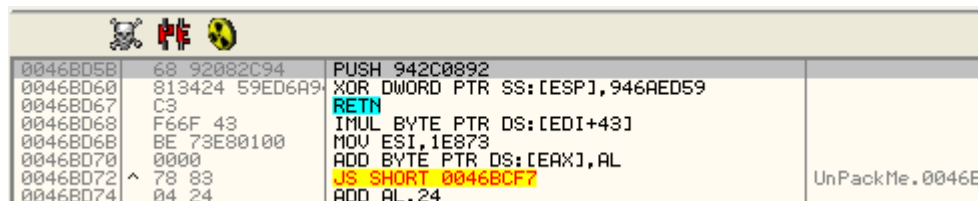
我们会发现 ESP - 0C 指向的栈空间中并没有保存 API 函数的入口地址,而是保存了 46E5CB 这个值。



我们可以看到通过 IAT.txt 脚本,460DE8 这个 IAT 项被保存的是 46E5CB 这个值,那么不通过脚本呢?我们按 F7 执行该指令。



我们可以看到不通过脚本 460DE8 这个内存单元中保存的值是 46BD5B。

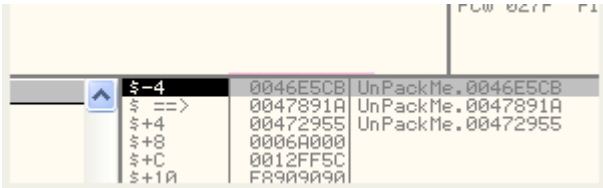
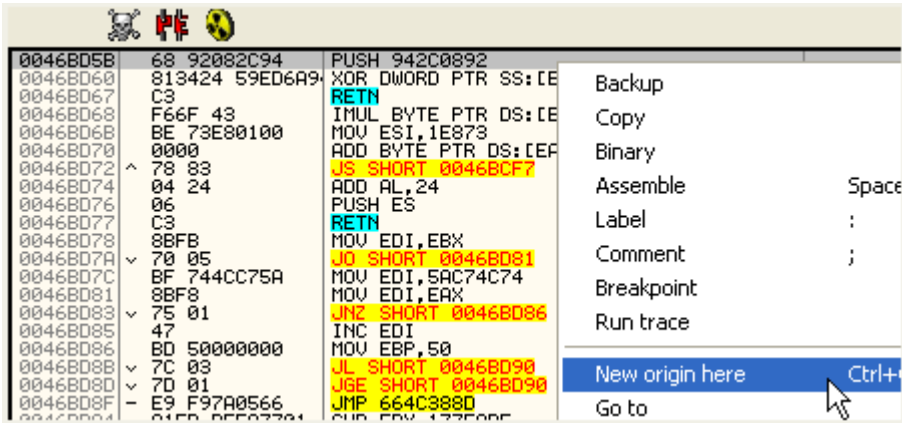


我们转到 46BD5B 地址处看看是什么。

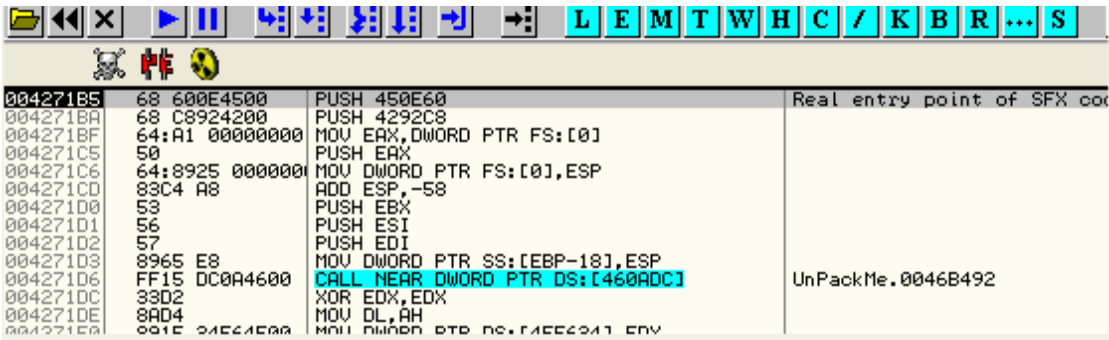
这里我们可以看到首先是将一个常量压入堆栈,接着与另一个常量进行异或就可以得到某个 API 函数的入口地址,我们一起来计算一下。

942C0892 xor 946aed59

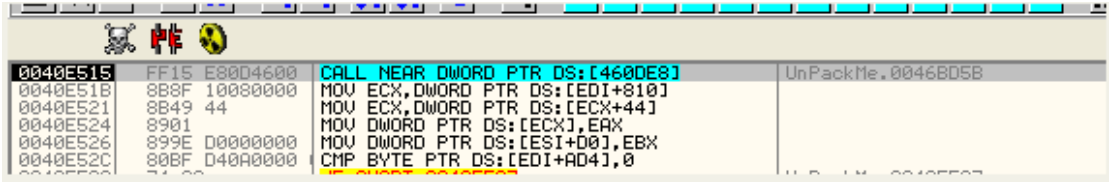
如果你不想通过计算器来算的话,那么也可以通过 OD 执行这几条指令来得到结果。



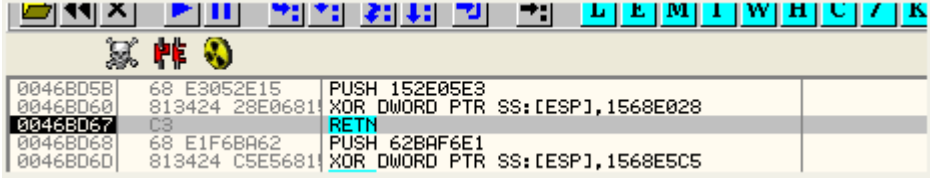
这里我们可以看到异或得到的结果是 46E5CB,跟执行 IAT.txt 脚本得到的结果是一致的。也就是问题并不是出在脚本上。应该是其他的某个环节出了问题。我们重启 OD,随便找一个 API 函数跟一下。



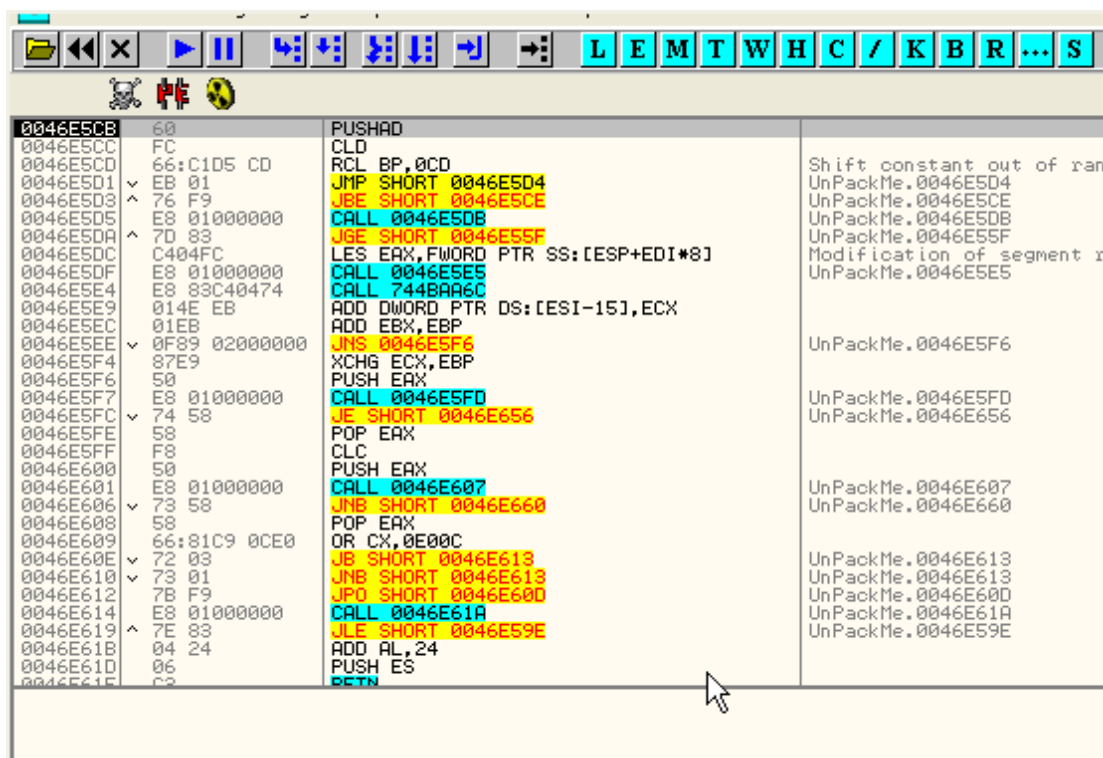
我们执行 OEP.txt 脚本到达了 OEP 处,接着依然对刚刚的 460DE8 这个无效的项设置内存访问断点,看看会断在哪里。



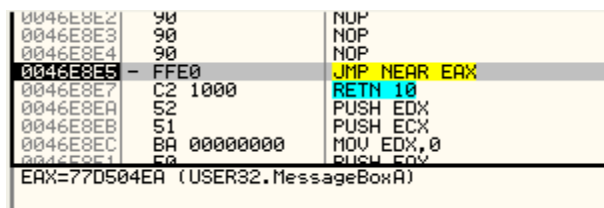
断在了这里,我们跟进去看看。



我们跟到 RET 指令处,继续跟进。

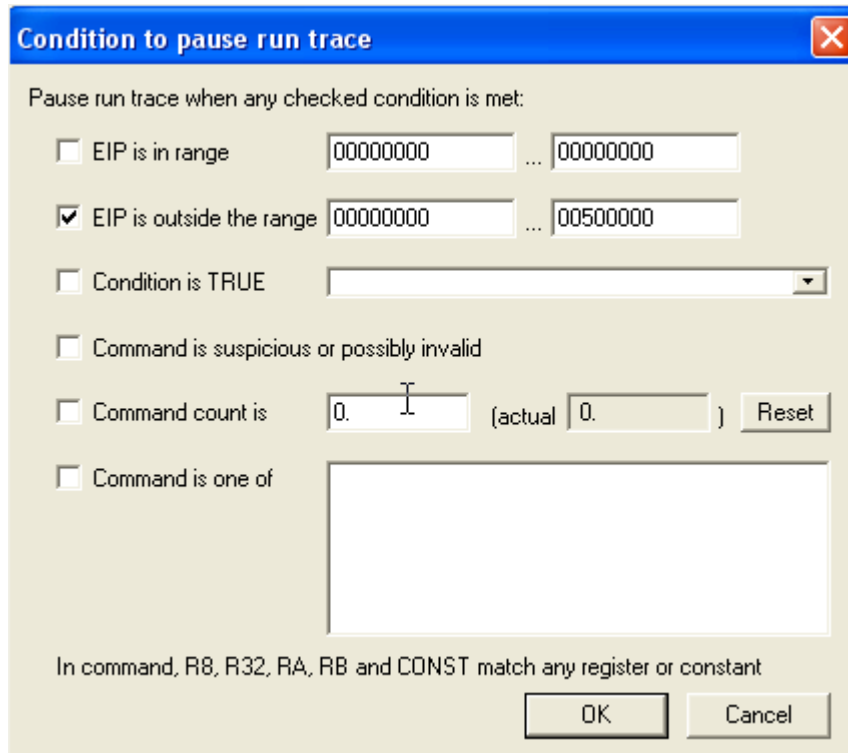


这里我们可以看到第一条指令是 PUSHAD,那么根据堆栈平衡原理,可以很自然的想到后面应该有个 POPAD 指令,我们按 F7 键往下跟若干行会到达这里。

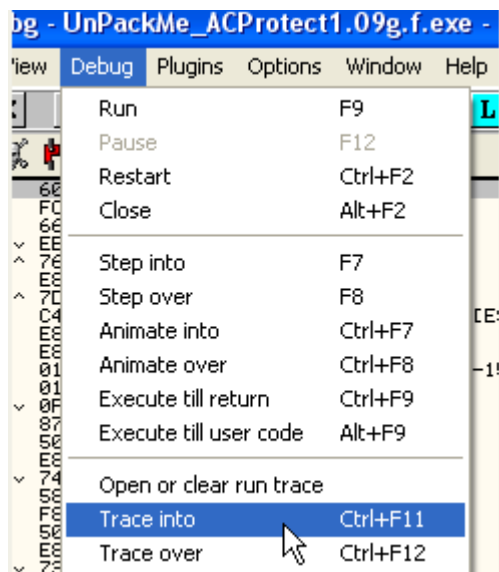


这里我们可以看到调用了 MessageBoxA 这个 API 函数。

另一种更加快捷的方法就是通过 OD 的自动跟踪功能来定位,我们设置自动跟踪的终止条件为 EIP 大于 500000,也就是说从当前区段转入到系统 DLL 中去执行 API 函数的时候就会断下来,我们一起来看一看。



这里我们选中 EIP is outside the range。范围设置为 0~500000。下面我们一起来看看效果。



77D504EA	8BFF	MOV EDI,EDI	
77D504EC	55	PUSH EBP	
77D504ED	8BEC	MOV EBP,ESP	
77D504EF	833D BC04D777	CMP DWORD PTR DS:[77D704BC],0	
77D504F6	74 24	JE SHORT 77D50510	USER32.77D5051C
77D504F8	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
77D504FE	6A 00	PUSH 0	
77D50500	FF70 24	PUSH DWORD PTR DS:[EAX+24]	
77D50503	68 240B0777	PUSH 77D70B24	
77D50508	FF15 C812D177	CALL NEAR DWORD PTR DS:[77D112C8]	kernel32.InterlockedCompareExchange
77D5050E	85C0	TEST EAX,EAX	
77D50510	75 0A	JNZ SHORT 77D5051C	USER32.77D5051C
77D50512	C705 200B0777	MOV DWORD PTR DS:[77D70B20],1	
77D5051C	6A 00	PUSH 0	
77D5051E	FF75 14	PUSH DWORD PTR SS:[EBP+14]	
77D50521	FF75 10	PUSH DWORD PTR SS:[EBP+10]	
77D50524	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
77D50527	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
77D5052A	E8 2D000000	CALL 77D5055C	USER32.MessageBoxExA
77D5052F	5D	POP EBP	
77D50530	C2 1000	RETN 10	
77D50533	90	NOP	
77D50534	90	NOP	
77D50535	90	NOP	
77D50536	90	NOP	

我们可以看到断在了 MessageBoxA 的入口处,为了验证 API 函数的正确性,我们来看看该函数的调用处是哪里,看看堆栈中的返回地址。

0012FCA0	0040E51B	CALL to MessageBoxA from UnPackMe.0040E515
0012FCB0	00000000	hOwner = NULL
0012FCB3	00A21CC0	Text = "This is an ACPProtect 1.09g UnPackME/All Protectio
0012FCB6	00A21D90	Title = "Teddy Rogers / SnD Team 2005"
0012FCB9	00000030	Style = MB_OK MB_ICONEXCLAMATION MB_APPLMODAL
0012FCB4	00000000	
0012FCB7	00010000	

这里我们可以看到返回地址是 40E51B,我们直接转到这个地址。

0040E512	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040E515	FF15 E80D4600	CALL NEAR DWORD PTR DS:[460DE8]	UnPackMe.00468D5B
0040E51B	8B8F 10080000	MOV ECX,DWORD PTR DS:[EDI+810]	
0040E521	8B49 44	MOV ECX,DWORD PTR DS:[ECX+44]	
0040E524	8901	MOV DWORD PTR DS:[ECX],EAX	
0040E526	899E D0000000	MOV DWORD PTR DS:[ESI+D0],EBX	
0040E52C	80BF D40A0000	CMP BYTE PTR DS:[EDI+AD4],0	
0040E533	74 02	JE SHORT 0040E537	UnPackMe.0040E537
0040E535	90	NOP	

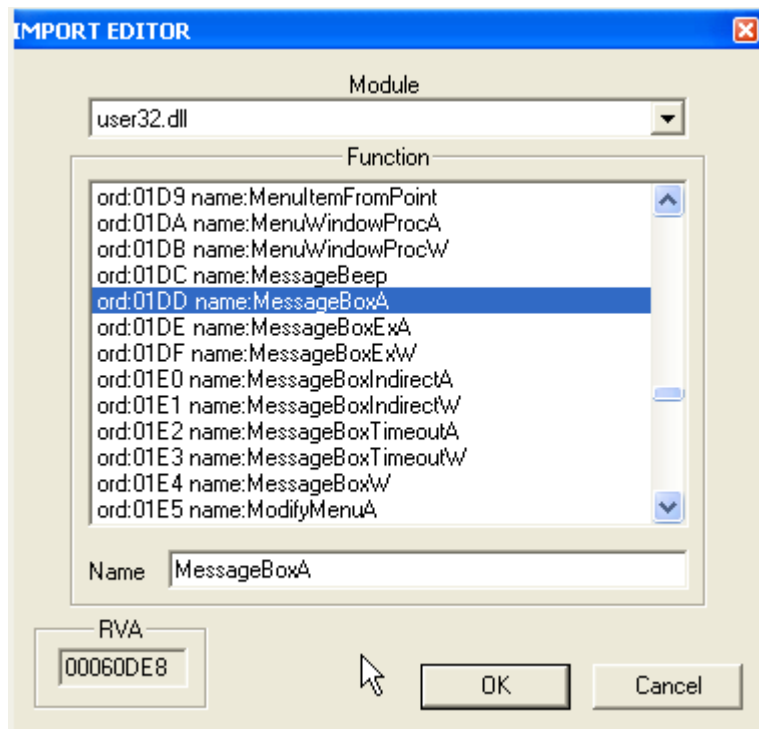
这里返回到的是调用处的下一行,好了,这里我们就得到了正确的 API 函数,我们重启 OD,执行 IAT.txt 脚本,就 OEP,IAT 起始地址,IAT 大小等数据都填入到 IMP REC 中。

```

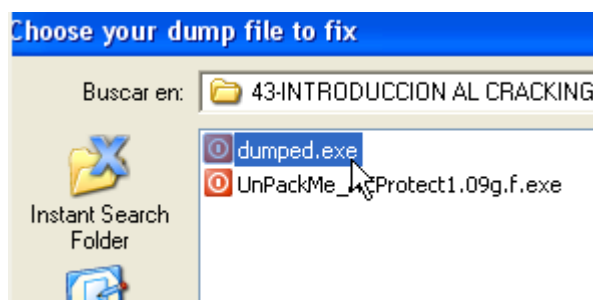
rva:00060DDC mod:user32.dll ord:0010 name:BringWindowToTop
rva:00060DE0 mod:user32.dll ord:00F3 name:GetAsyncKeyState
rva:00060DE4 mod:user32.dll ord:02D9 name:wsprintfA
rva:00060DE8 ptr:0046E5CB
rva:00060DEC mod:user32.dll ord:01BC name:LoadIconA
rva:00060DF0 mod:user32.dll ord:024E name:SetCursor
rva:00060DF4 mod:user32.dll ord:01AC name:IsWindow

```

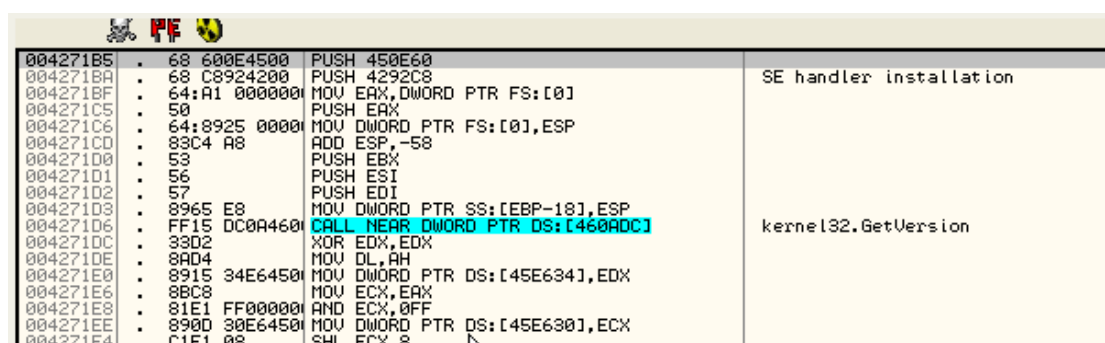
我们定位到这个无效的项,双击之,将其修改为 MessageBoxA。



好了,现在所有的项都有效了。接下来修复 dump 文件,选中 dumped.exe, 单击 Fix dump。



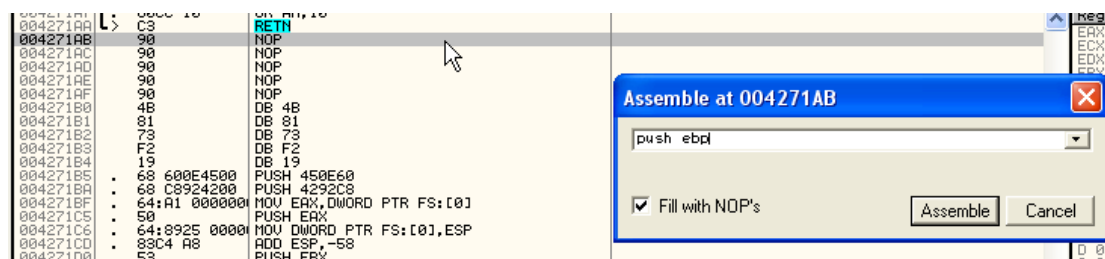
修复后的 dump 文件被重命名为为了 dumped_exe,别忘了 stolen bytes 我们还没有填充回去,我们用 OD 打开 dumped_exe。



停在了假的 OEP 处,我们准备好 stolen bytes。

00485AF3	Main	PUSH EBP	
00485AF4	Main	MOV EBP,ESP	; EBP=0012FFC0
00485AF6	Main	PUSH -1	

我们将 stolen bytes 以汇编指令的形式填充上。



004271AA	C3	RETN	
004271AB	55	PUSH EBP	
004271AC	8BEC	MOV EBP,ESP	
004271AD	6A FF	PUSH -1	
004271B0	4B	DB 4B	CHAR 'K'
004271B1	81	DB 81	CHAR 's'
004271B2	73	DB 73	
004271B3	F2	DB F2	
004271B4	19	DB 19	
004271B5	68 600E4500	PUSH 450E60	
004271B6	68 C8924200	PUSH 4292C8	
004271B7	64:A1 000000	MOV EAX,DWORD PTR FS:[0]	
004271B8	50	PUSH EAX	
004271B9	64:8925 0000	MOV DWORD PTR FS:[0],ESP	
004271BA	83C4 A8	ADD ESP,-58	
004271BB	53	PUSH EAX	

我们通过计算可知 stolen bytes 总共占 5 个字节长度,4271b5 往上 5 个字节就是 4271b0,也就是说正确的 OEP 应该是 4271b0。

我们从 4271b0 地址处开始写入 stolen bytes。

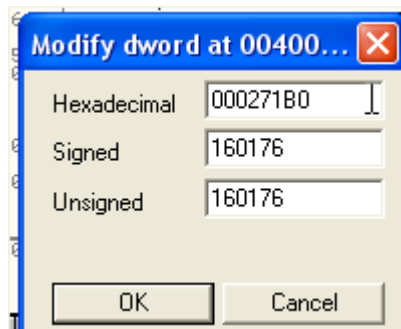
004271AE	90	NOP	
004271AF	90	NOP	
004271B0	55	PUSH EBP	
004271B1	8BEC	MOV EBP,ESP	
004271B2	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH 450E60	
004271B6	68 C8924200	PUSH 4292C8	
004271B7	64:A1 000000	MOV EAX,DWORD PTR FS:[0]	SE handler installation

接下来我们通过单击鼠标右键选择-Copy to executable-All modifications 将刚所做的修改保存到文件。接着我们来将其 OEP 更正。

用 OD 加载刚刚保存的文件,我们可以通过在数据窗口中单击鼠标右键选择-Go to-Expression,输入 400000 定位到 PE 头。

Address	Hex dump	ASCII
00400000	4D 5A 90 00 03 00 00 00	MZ... ..
00400008	04 00 00 00 FF FF 00 00
00400010	B8 00 00 00 00 00 00 00	@... ..
00400018	40 00 00 00 00 00 00 00	@... ..
00400020	00 00 00 00 00 00 00 00
00400028	00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00
00400038	00 00 00 00 80 00 00 00
00400040	0E 1F BA 0E 00 B4 09 CD
00400048	21 B8 01 4C CD 21 54 68
00400050	69 73 20 70 72 6F 67 72	is progr
00400058	61 6D 20 63 61 6E 6E 6F	am canno
00400060	74 20 62 65 20 72 75 6E	t be run
00400068	20 69 6E 20 44 4F 53 20	in DOS
00400070	6D 6F 64 65 2E 0D 0D 0A	mode....
00400078	24 00 00 00 00 00 00 00	\$.
00400080	50 45 00 00 4C 01 07 00	PE..LO..
00400088	C5 91 FC 41 00 00 00 00	+...A....
00400090	00 00 00 00 E0 00 0F 01
00400098	0B 01 05 00 00 92 04 00
004000A0	00 7A 01 00 00 00 00 00
004000A8	00 B0 06 00 00 10 00 00
004000B0	00 B0 04 00 00 00 40 00
004000B8	00 10 00 00 00 02 00 00
004000C0	04 00 00 00 00 00 00 00

单击鼠标右键选择 Special-PE header 将显示模式切换为 PE 头视图,接下来定位到 AddressOfEntryPoint,将其修改为 271b0 并保存。



Address	Hex dump	Data	Comment
0040009C	00920400	DD 00049200	SizeOfCode = 49200 (299520.)
004000A0	007A0100	DD 00017A00	SizeOfInitializedData = 17A00 (96768.)
004000A4	00000000	DD 00000000	SizeOfUninitializedData = 0
004000A8	B0710200	DD 000271B0	AddressOfEntryPoint = 271B0
004000AC	00100000	DD 00001000	BaseOfCode = 1000
004000B0	00B00400	DD 0004B000	BaseOfData = 4B000
004000B4	00004000	DD 00004000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000

好了,现在我们就修复完毕了,但是我们运行修复后的程序会发现无法正常运行,程序崩溃了。好,那么我们不勾选忽略异常的选项,加载刚刚修复过的程序,运行起来,看看会提示哪些异常。

```

77350000 Module C:\WINDOWS\winSxS\x86_microsoft.window
50160000 Module C:\WINDOWS\system32\serwvdrv.dll
004271B0 Program entry point
Access violation when executing [00000000]

```

我们可以看到日志窗口中显示了一个非法访问异常。我们通过单击工具栏中的 K 按钮查看调用堆栈。

Address	Stack	Procedure / arguments	Called from
0012FF08	0042980B	dumped_.0046C5F3	dumped_.00429806
0012FF20	004245C0	dumped_.00429740	dumped_.004245C8
0012FF38	0042A20F	dumped_.00424580	dumped_.0042A20A
0012FF48	0042721F	dumped_.0042A1F0	dumped_.0042721A

这里我们可以看到最后一次调用来至于 429806,我们通过单击鼠标右键选择-Show call 定位到该调用处。

Called from	Frame
dumped_.00429806	
dumped_.004245C8	
dumped_.0042A20A	
dumped_.0042721A	

Actualize	
Hide arguments	
Follow address in stack	
Show procedure	
Show call	
Execute to return	
Copy to clipboard	
Appearance	

00429802	8B17	MOV EDX,DWORD PTR DS:[EDI]	
00429804	2BD3	SUB EDX,EBX	
00429806	E8 E82D0400	CALL 0046C5F3	dumped_.00429806
00429808	5F	POP EDI	
0042980C	5E	POP ESI	
00429810	5D	POP EBP	
00429814	5B	POP EBX	
00429818	59	POP ECX	
0042981C	C3	RET	
00429820	8B4C24 10	MOV ECX,DWORD PTR SS:[ESP+10]	

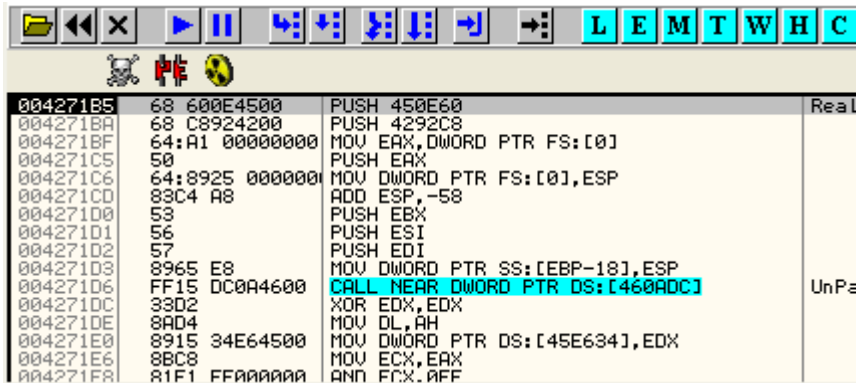
单击鼠标右键选择-Follow 看看这个 CALL 里面是什么。

0046C5D8	FF25 F4751700	JMP NEAR DWORD PTR DS:[1775F4]	
0046C5E1	FF25 F3751700	JMP NEAR DWORD PTR DS:[1775F3]	
0046C5E7	FF25 FC751700	JMP NEAR DWORD PTR DS:[1775FC]	
0046C5ED	FF25 00761700	JMP NEAR DWORD PTR DS:[177600]	
0046C5F3	FF25 04761700	JMP NEAR DWORD PTR DS:[177604]	
0046C5F9	FF25 05761700	JMP NEAR DWORD PTR DS:[177608]	
0046C5FF	FF25 0C761700	JMP NEAR DWORD PTR DS:[17760C]	
0046C605	FF25 10761700	JMP NEAR DWORD PTR DS:[177610]	
0046C60B	FF25 14761700	JMP NEAR DWORD PTR DS:[177614]	
0046C611	FF25 18761700	JMP NEAR DWORD PTR DS:[177618]	
0046C617	FF25 1C761700	JMP NEAR DWORD PTR DS:[17761C]	
0046C61D	FF25 20761700	JMP NEAR DWORD PTR DS:[177620]	
0046C623	FF25 24761700	JMP NEAR DWORD PTR DS:[177624]	
0046C629	FF25 28761700	JMP NEAR DWORD PTR DS:[177628]	
0046C62F	FF25 2C761700	JMP NEAR DWORD PTR DS:[17762C]	
0046C635	FF25 30761700	JMP NEAR DWORD PTR DS:[177630]	
0046C63B	FF25 34761700	JMP NEAR DWORD PTR DS:[177634]	
0046C641	FF25 38761700	JMP NEAR DWORD PTR DS:[177638]	
0046C647	FF25 3C761700	JMP NEAR DWORD PTR DS:[17763C]	
0046C64D	FF25 40761700	JMP NEAR DWORD PTR DS:[177640]	
0046C653	FF25 44761700	JMP NEAR DWORD PTR DS:[177644]	
0046C659	FF25 48761700	JMP NEAR DWORD PTR DS:[177648]	
0046C65F	FF25 4C761700	JMP NEAR DWORD PTR DS:[17764C]	
0046C665	FF25 50761700	JMP NEAR DWORD PTR DS:[177650]	
0046C66B	FF25 54761700	JMP NEAR DWORD PTR DS:[177654]	
0046C671	FF25 58761700	JMP NEAR DWORD PTR DS:[177658]	
0046C677	FF25 5C761700	JMP NEAR DWORD PTR DS:[17765C]	
0046C67D	FF25 60761700	JMP NEAR DWORD PTR DS:[177660]	
0046C683	FF25 64761700	JMP NEAR DWORD PTR DS:[177664]	
0046C689	FF25 68761700	JMP NEAR DWORD PTR DS:[177668]	
0046C68F	FF25 6C761700	JMP NEAR DWORD PTR DS:[17766C]	
0046C695	FF25 70761700	JMP NEAR DWORD PTR DS:[177670]	

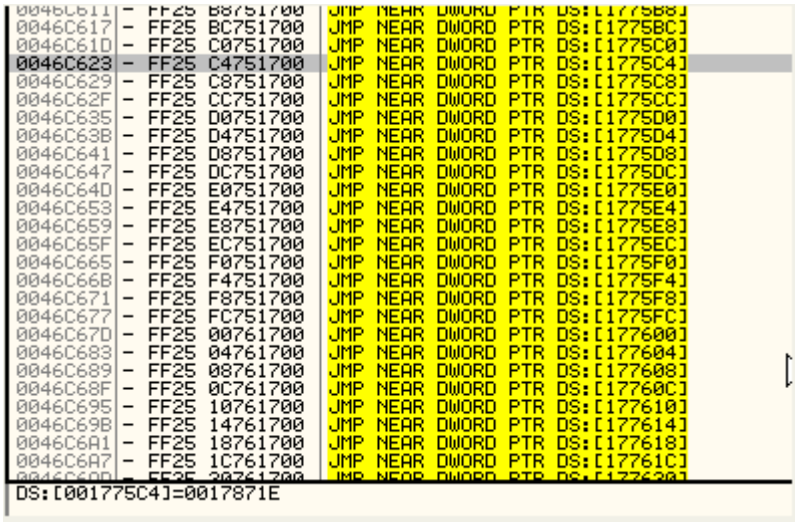
嘿嘿,这里我们可以看到一个跳转表,正是由于这些跳转表的目标地址诸如 177658 这类地址才导致程序无法正常运行报错的。

该地址所属的区段是由壳创建的,但是我们并没有 dump 出来。下面我们就来解决这个问题。

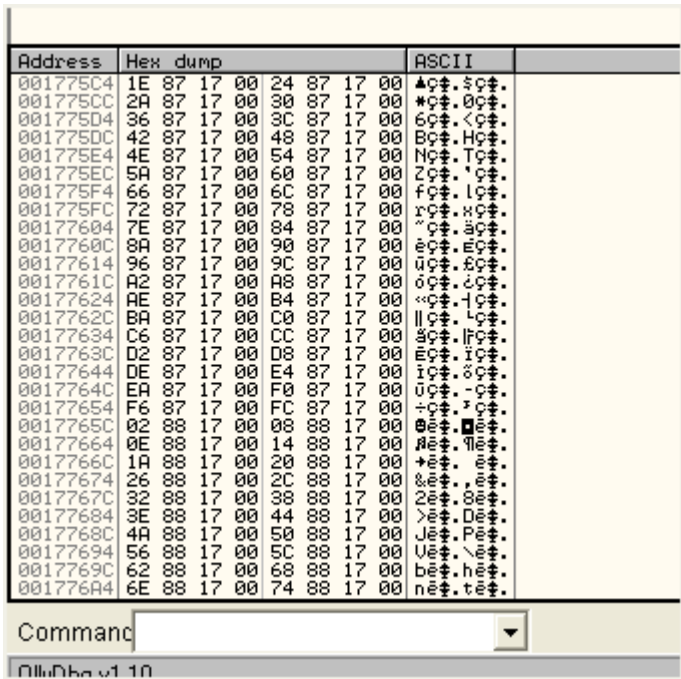
我们再开一个 OD,加载原始的程序,执行 OEP.txt 脚本到达 OEP 处。



我们到了这里,我们定位到上面的跳转表。



我们在数据窗口中定位跳转表中唯一一个目标地址,这里我们定位 1775C4。



我们往上定位到第一个跳转,我这里是 46C0F5。

0046C0F5	- FF25 B0721700	JMP NEAR DWORD PTR DS:[1772B0]
0046C0FB	- FF25 B4721700	JMP NEAR DWORD PTR DS:[1772B4]
0046C101	- FF25 B8721700	JMP NEAR DWORD PTR DS:[1772B8]
0046C107	- FF25 BC721700	JMP NEAR DWORD PTR DS:[1772BC]
0046C10D	- FF25 C0721700	JMP NEAR DWORD PTR DS:[1772C0]
0046C113	- FF25 C4721700	JMP NEAR DWORD PTR DS:[1772C4]
0046C119	- FF25 C8721700	JMP NEAR DWORD PTR DS:[1772C8]
0046C11F	- FF25 CC721700	JMP NEAR DWORD PTR DS:[1772CC]
0046C125	- FF25 D0721700	JMP NEAR DWORD PTR DS:[1772D0]
0046C12B	- FF25 D4721700	JMP NEAR DWORD PTR DS:[1772D4]
0046C131	- FF25 D8721700	JMP NEAR DWORD PTR DS:[1772D8]
0046C137	- FF25 DC721700	JMP NEAR DWORD PTR DS:[1772DC]
0046C13D	- FF25 E0721700	JMP NEAR DWORD PTR DS:[1772E0]
0046C143	- FF25 E4721700	JMP NEAR DWORD PTR DS:[1772E4]
0046C149	- FF25 E8721700	JMP NEAR DWORD PTR DS:[1772E8]
0046C14F	- FF25 EC721700	JMP NEAR DWORD PTR DS:[1772EC]
0046C155	- FF25 F0721700	JMP NEAR DWORD PTR DS:[1772F0]
0046C15B	- FF25 F4721700	JMP NEAR DWORD PTR DS:[1772F4]

这些跳转指令都占 6 个字节长度,我们随便挑一个目标地址,这里我们选择目标地址为 178250,转到这个地址处,我们会发现其执行会 6 个字节长度的指令,然后返回。

00178250	8BF1	MOV ESI,ECX
00178252	8B4E 34	MOV ECX,DWORD PTR DS:[ESI+34]
00178255	C3	RETN
00178256	8BF1	MOV ESI,ECX
00178258	8B4E 38	MOV ECX,DWORD PTR DS:[ESI+38]
0017825B	C3	RETN
0017825D	8BF1	MOV ESI,ECX

这里我的想法是用这 6 个字节替换掉那跳转表中的 6 个字节。

00178250	8BF1	MOV ESI,ECX
00178252	8B4E 34	MOV ECX,DWORD PTR DS:[ESI+34]
00178255	C3	RETN
00178256	8BF1	MOV ESI,ECX
00178258	8B4E 38	MOV ECX,DWORD PTR DS:[ESI+38]
0017825B	C3	RETN
0017825C	8BF1	MOV ESI,ECX
0017825E	8B4D 0C	MOV ECX,DWORD PTR DS:[ESI+0C]
00178261	C3	RETN
00178262	8BF1	MOV ESI,ECX
00178264	8B46 78	MOV ECX,DWORD PTR DS:[ESI+78]
00178267	C3	RETN
00178268	897E 78	MOV EDI,DWORD PTR DS:[ESI+78]
0017826B	8B3F	MOV EDI,DWORD PTR DS:[ESI+3F]
0017826D	C3	RETN
0017826E	8945 F4	MOV EDI,DWORD PTR DS:[ESI+F4]
00178271	8B07	MOV ECX,DWORD PTR DS:[ESI+07]

我们选中这三行指令,单击鼠标右键选择-Binary-Binary copy。

然后定位到相应的跳转指令处,单击鼠标右键选择-Binary-Binary paste。

0046C0F5	- FF25 B0721700	JMP NEAR DWORD PTR DS:[1772B0]
0046C0FB	- FF25 B4721700	JMP NEAR DWORD PTR DS:[1772B4]
0046C101	- FF25 B8721700	JMP NEAR DWORD PTR DS:[1772B8]
0046C107	- FF25 BC721700	JMP NEAR DWORD PTR DS:[1772BC]
0046C10D	- FF25 C0721700	JMP NEAR DWORD PTR DS:[1772C0]
0046C113	- FF25 C4721700	JMP NEAR DWORD PTR DS:[1772C4]
0046C119	- FF25 C8721700	JMP NEAR DWORD PTR DS:[1772C8]
0046C11F	- FF25 CC721700	JMP NEAR DWORD PTR DS:[1772CC]
0046C125	- FF25 D0721700	JMP NEAR DWORD PTR DS:[1772D0]
0046C12B	- FF25 D4721700	JMP NEAR DWORD PTR DS:[1772D4]
0046C131	- FF25 D8721700	JMP NEAR DWORD PTR DS:[1772D8]
0046C13D	- FF25 DC721700	JMP NEAR DWORD PTR DS:[1772DC]
0046C137	- FF25 E0721700	JMP NEAR DWORD PTR DS:[1772E0]
0046C143	- FF25 E4721700	JMP NEAR DWORD PTR DS:[1772E4]
0046C149	- FF25 E8721700	JMP NEAR DWORD PTR DS:[1772E8]
0046C14F	- FF25 EC721700	JMP NEAR DWORD PTR DS:[1772EC]
0046C155	- FF25 F0721700	JMP NEAR DWORD PTR DS:[1772F0]

这里我们可以看到覆盖后的效果。

0046C0F5	8BF1	MOV ESI,ECX
0046C0F7	8B4E 34	MOV ECX,DWORD PTR DS:[ESI+34]
0046C0FA	C3	RETN
0046C0FB	- FF25 B4721700	JMP NEAR DWORD PTR DS:[1772B4]
0046C101	- FF25 B8721700	JMP NEAR DWORD PTR DS:[1772B8]
0046C107	- FF25 BC721700	JMP NEAR DWORD PTR DS:[1772BC]
0046C10D	- FF25 C0721700	JMP NEAR DWORD PTR DS:[1772C0]
0046C113	- FF25 C4721700	JMP NEAR DWORD PTR DS:[1772C4]
0046C119	- FF25 C8721700	JMP NEAR DWORD PTR DS:[1772C8]

下面我们将整个跳转表都替换掉。

我们可以看到跳转表开始于 1772B4。

Address	Hex dump	ASCII
001772B4	56 82 17 00 5C 82 17 00	Ue%. \e%.
001772BC	62 82 17 00 68 82 17 00	be%. he%.
001772C4	7E 82 17 00 74 82 17 00	ne%. te%.
001772CC	7A 82 17 00 80 82 17 00	ze%. ce%.
001772D0	86 82 17 00 8C 82 17 00	ae%. ie%.
001772D8	92 82 17 00 98 82 17 00	fe%. oe%.
001772E4	9E 82 17 00 A4 82 17 00	xe%. ne%.
001772EC	AA 82 17 00 B0 82 17 00	me%. re%.
001772F4	B6 82 17 00 BC 82 17 00	ae%. ue%.
001772FC	C2 82 17 00 C8 82 17 00	me%. ve%.

该跳转表的最后一项为 1782B4,其中的值为 1799BA。(PS:该跳转表的地址在你们的机器上可能不一样)

Address	Hex dump	ASCII
001781D4	06 99 17 00 0C 99 17 00	*0%. .0%.
001781DC	12 99 17 00 18 99 17 00	*0%. 10%.
001781E4	1E 99 17 00 24 99 17 00	*0%. 20%.
001781EC	2A 99 17 00 30 99 17 00	*0%. 30%.
001781F4	36 99 17 00 3C 99 17 00	60%. <0%.
001781FC	42 99 17 00 48 99 17 00	B0%. H0%.
00178204	4E 99 17 00 54 99 17 00	N0%. T0%.
0017820C	5A 99 17 00 60 99 17 00	Z0%. '0%.
00178214	66 99 17 00 6C 99 17 00	f0%. l0%.
0017821C	72 99 17 00 78 99 17 00	r0%. w0%.
00178224	7E 99 17 00 84 99 17 00	~0%. a0%.
0017822C	8A 99 17 00 90 99 17 00	e0%. e0%.
00178234	96 99 17 00 9C 99 17 00	00%. 60%.
0017823C	A2 99 17 00 A8 99 17 00	60%. 20%.
00178244	AE 99 17 00 B4 99 17 00	<0%. j0%.
0017824C	BA 99 17 00 8B F1 8B 4E	ll0%. i: iN
00178254	34 C3 8B F1 8B 4E 38 C3	4ti: iNSt
0017825C	8B F1 8B 4D 0C C3 8B F1	i: iM. ti:

这里将所有的代码都以二进制的形式拷贝出来,接着以二进制的形式粘贴到跳转表所在区域。

Address	Hex dump	Assembly
0046C0F5	8BF1	MOV ESI,ECX
0046C0F7	8B4E 34	MOV ECX,DWORD PTR DS:[ESI+34]
0046C0FA	C3	RETN
0046C0FB	8BF1	MOV ESI,ECX
0046C0FD	8B4E 38	MOV ECX,DWORD PTR DS:[ESI+38]
0046C100	C3	RETN
0046C101	8BF1	MOV ESI,ECX
0046C103	8B4D 0C	MOV ECX,DWORD PTR SS:[EBP+C]
0046C106	C3	RETN
0046C107	8BF1	MOV ESI,ECX
0046C109	8B46 78	MOV EAX,DWORD PTR DS:[ESI+78]
0046C10C	C3	RETN
0046C10D	897E 78	MOV DWORD PTR DS:[ESI+78],EDI
0046C110	8B3F	MOV EDI,DWORD PTR DS:[EDI]
0046C112	C3	RETN
0046C113	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
0046C116	8B07	MOV EAX,DWORD PTR DS:[EDI]
0046C118	C3	RETN
0046C119	8BCF	MOV ECX,EDI
0046C11B	8975 F8	MOV DWORD PTR SS:[EBP-8],ESI
0046C11E	C3	RETN

这里我们会发现没有覆盖完全。

0046C9B5	8B40 0C	MOV EAX,DWORD PTR DS:[EAX+C]
0046C9B8	C3	RETN
0046C9B9	8948 0C	MOV DWORD PTR DS:[EAX+C],ECX
0046C9BC	8B0F	MOV ECX,DWORD PTR DS:[EDI]
0046C9BE	C3	RETN
0046C9BF	8BF1	MOV ESI,ECX
0046C9C1	8B46 5C	MOV EAX,DWORD PTR DS:[ESI+5C]
0046C9C4	C3	RETN
0046C9C5	03C1	ADD EAX,ECX
0046C9C7	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]
0046C9CA	C3	RETN
0046C9CB	B6 B6	MOV DH,0B6
0046C9CD	B6 B6	MOV DH,0B6
0046C9CF	B6 B6	MOV DH,0B6
0046C9D1	B6 B6	MOV DH,0B6
0046C9D3	B6 B6	MOV DH,0B6
0046C9D5	B6 B6	MOV DH,0B6
0046C9D7	B6 B6	MOV DH,0B6
0046C9D9	B6 B6	MOV DH,0B6
0046C9DB	B6 B6	MOV DH,0B6
0046C9DD	B6 B6	MOV DH,0B6
0046C9DF	B6 B6	MOV DH,0B6
0046C9E1	B6 B6	MOV DH,0B6
0046C9E3	B6 B6	MOV DH,0B6
0046C9E5	B6 B6	MOV DH,0B6

可能该壳做了一些干扰吧,没有关系,到目前为止,我们的思路是正确的。好,那么我们直接定位 dump 文件所在的 OD,就这部分二进制数据粘贴上去。效果同上。

(PS:作者处理有点冗余,这里我稍微修改了下,让大家更好理解)

接着保存所做的修改到文件,运行修复过的文件看看效果。



我们可以看到程序可以正常运行了,至此关于 ACProtect V1.09 的 OEP 定位,stolen bytes,IAT,AntiDump 的修复我们就介绍完毕了。