

第四十九章-PeSpin V1.3.04 脱壳-Part2

本章我们来修改 PeSpin 的 IAT。

当我们到达伪造的 OEP 处时,我们随便定位一个 API 函数调用处,接着定位到 IAT,通过观察很容易得知 IAT 的起始地址为 460818,结束位置为 460F28。但是我们会发现一个问题:有些 CALL 并没有调用 IAT 中的项。我们一起来看看个例子。

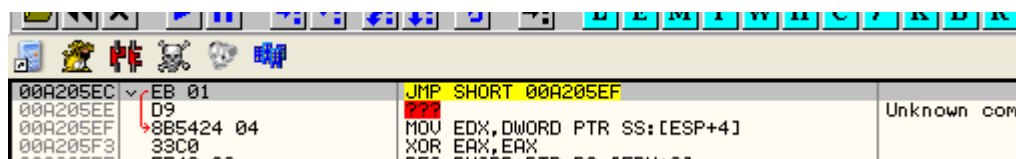
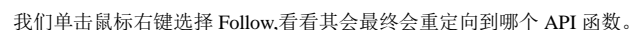
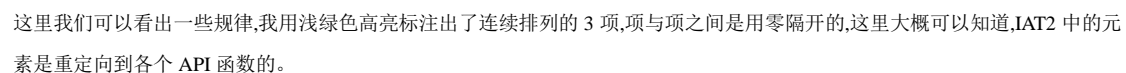
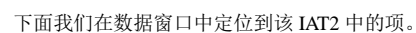
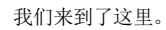
Address	Hex dump	ASCII
004607F4	00 00 00 00 00 00 00 00
004607FC	00 00 00 00 00 00 00 00
00460804	00 00 00 00 00 00 00 00
0046080C	00 00 00 00 00 00 00 00
00460814	00 00 00 00 F0 68 DA 77-k r w
0046081C	1B 76 DA 77 F4 EA DA 77	+U r w 9 0 r w
00460824	E7 EB DA 77 83 78 DA 77	8 0 r w 8 k r w
0046082C	00 00 00 00 DD 15 C5 58	...!S+X
00460834	2E BD C3 58 00 00 00 00	.c t X
0046083C	56 27 06 00 64 27 06 00	U' . . d' . .
00460844	76 27 06 00 8A 27 06 00	U' . . e' . .
0046084C	96 27 06 00 A0 27 06 00	Q' . . a' . .
00460854	E0 25 06 00 B0 27 06 00	0' . . 3' . .
0046085C	C4 27 06 00 D6 27 06 00	-' . . i' . .
00460864	42 27 06 00 F0 27 06 00	B' . . -' . .
0046086C	FC 27 06 00 0A 28 06 00	' . . (. .
00460874	18 28 06 00 22 28 06 00	+ (. .)' (. .
0046087C	2E 28 06 00 38 28 06 00	(. . 8 (. .
00460884	48 28 06 00 04 25 06 00	H (. . e' (. .
0046088C	C8 25 06 00 B6 25 06 00	8' . . a' (. .
00460894	AC 25 06 00 9C 25 06 00	%' . . e' (. .
0046089C	82 25 06 00 6A 25 06 00	e' . . j' (. .
004608A4	56 25 06 00 42 25 06 00	U' . . B' (. .
004608AC	36 25 06 00 28 25 06 00	6' . . (' (. .
004608B4	1E 25 06 00 0E 25 06 00	A' . . 8' (. .
004608BC	04 25 06 00 F8 24 06 00	4' . . 8' (. .
004608C4	E2 24 06 00 D0 24 06 00	0' . . 3' (. .
004608CC	8A 24 06 00 AA 24 06 00	' . . 7' (. .
004608D4	96 24 06 00 30 27 06 00	Q' . . 0' (. .
004608DC	1E 27 06 00 08 27 06 00	A' . . 0' (. .
004608E4	F4 27 06 00 00 27 06 00	m . . 0' (. .

这里整个 IAT 我都用浅绿色高亮标注出来了,但是我们会发现 IAT 中的有些项找不到参考引用的地方(比如说这些 0006 开头的值),这些值并不指向任何 API 函数,而且也不在任意一个 DLL 的地址空间范围之内,那么也就是说这些值不是重定向过的就是垃圾数据了。

现在我们来查看所有的 API 函数调用处,在反汇编窗口中单击鼠标右键选择 -Search for-All intermodular calls。

Address	Disassembly	Destination
00425306	CALL NEAR DWORD PTR DS:[46F42A]	DS:[0046F42A]=00A20000
0042535E	CALL NEAR DWORD PTR DS:[46F475]	DS:[0046F475]=00A201B2
004259D1	CALL NEAR DWORD PTR DS:[46F542]	DS:[0046F542]=00A20642
004259E8	CALL NEAR DWORD PTR DS:[46F524]	DS:[0046F524]=00A205BA
004259F5	CALL NEAR DWORD PTR DS:[46F52E]	DS:[0046F52E]=00A205EC
00425B2B	CALL NEAR DWORD PTR DS:[46F60F]	DS:[0046F60F]=00A20A9C
00425B3F	CALL NEAR DWORD PTR DS:[46F60F]	DS:[0046F60F]=00A20A9C
00425B53	CALL NEAR DWORD PTR DS:[46F60F]	DS:[0046F60F]=00A20A9C
00425BF1	CALL NEAR DWORD PTR DS:[46F60F]	DS:[0046F60F]=00A20A9C
00425C9C	CALL NEAR DWORD PTR DS:[46F434]	DS:[0046F434]=00A20049
00425CA6	CALL NEAR DWORD PTR DS:[46F68C]	DS:[0046F68C]=00A20D70
00425D71	CALL NEAR DWORD PTR DS:[46F542]	DS:[0046F542]=00A20642
00425D8B	CALL NEAR DWORD PTR DS:[46F524]	DS:[0046F524]=00A205BA
00425DB8	CALL NEAR DWORD PTR DS:[46F52E]	DS:[0046F52E]=00A205EC
00425E13	CALL NEAR DWORD PTR DS:[46F6D7]	DS:[0046F6D7]=00A20EE7
00425E27	CALL NEAR DWORD PTR DS:[46F6D2]	DS:[0046F6D2]=00A20EC5
00425E68	CALL NEAR DWORD PTR DS:[46F6D2]	DS:[0046F6D2]=00A20EC5
00425F34	CALL NEAR DWORD PTR DS:[46F6D2]	DS:[0046F6D2]=00A20EC5
004271F7	ADD ECX,EDX	(Initial CPU selection)
00427200	CALL E66B1515	
0042723E	CALL NEAR DWORD PTR DS:[46F43E]	DS:[0046F43E]=00A2007E
004272D5	CALL NEAR DWORD PTR DS:[46F439]	DS:[0046F439]=00A20069
004272F6	CALL NEAR DWORD PTR DS:[46F6DC]	DS:[0046F6DC]=00A20F09
00427458	CALL 00400236	UnPackIt.e.004001D8
004274F7	CALL 0040023C	UnPackIt.e.0040023C
004277F3	CALL 00400270	UnPackIt.e.00400270
00427929	CALL NEAR DWORD PTR DS:[46F448]	DS:[0046F448]=00A200C1
00427E07	CALL NEAR DWORD PTR DS:[46F57E]	DS:[0046F57E]=00A20793
00427E0E	CALL NEAR DWORD PTR DS:[46F452]	DS:[0046F452]=00A200FE
00427E98	CALL NEAR DWORD PTR DS:[46F44D]	DS:[0046F44D]=00A200E1
00428829	CALL NEAR DWORD PTR DS:[46F4CF]	DS:[0046F4CF]=00A203E9
00428842	CALL 004003ED	UnPackIt.e.004001D8
004288B8	CALL NEAR DWORD PTR DS:[46F4E3]	DS:[0046F4E3]=00A20451
004288D5	CALL 00427360	UnPackIt.e.004001D8
004288E3	CALL NEAR DWORD PTR DS:[46F542]	DS:[0046F542]=00A20642
0042884D	CALL NEAR DWORD PTR DS:[46F524]	DS:[0046F524]=00A205BA
0042886E	CALL NEAR DWORD PTR DS:[46F52E]	DS:[0046F52E]=00A205EC
004288AE	CALL NEAR DWORD PTR DS:[46F524]	DS:[0046F524]=00A205BA
004288E0	CALL NEAR DWORD PTR DS:[46F524]	DS:[0046F524]=00A205BA
0042891E	CALL NEAR DWORD PTR DS:[46F52E]	DS:[0046F52E]=00A205EC
00428950	CALL NEAR DWORD PTR DS:[46F52E]	DS:[0046F52E]=00A205EC
00428983	CALL NEAR DWORD PTR DS:[46F6D7]	DS:[0046F6D7]=00A20EE7
00428C15	CALL NEAR DWORD PTR DS:[46F6DC]	DS:[0046F6DC]=00A20F09
00428C25	CALL NEAR DWORD PTR DS:[46F669]	DS:[0046F669]=00A20C98
004293A9	CALL NEAR DWORD PTR DS:[46F461]	DS:[0046F461]=00A2014D
004293C8	CALL NEAR DWORD PTR DS:[46F45C]	DS:[0046F45C]=00A20137
00429401	CALL NEAR DWORD PTR DS:[46F4D4]	DS:[0046F4D4]=00A20401
0042951F	CALL NEAR DWORD PTR DS:[46F466]	DS:[0046F466]=00A2016F
00429536	CALL NEAR DWORD PTR DS:[46F4D9]	DS:[0046F4D9]=00A20419
00429560	CALL NEAR DWORD PTR DS:[46F466]	DS:[0046F466]=00A2016F
00429597	CALL NEAR DWORD PTR DS:[46F4D9]	DS:[0046F4D9]=00A20419
004295E9	CALL NEAR DWORD PTR DS:[46F466]	DS:[0046F466]=00A2016F
00429892	CALL NEAR DWORD PTR DS:[46F46B]	DS:[0046F46B]=00A2018D
00429CB0	CALL NEAR DWORD PTR DS:[46F547]	DS:[0046F547]=00A20658
00429CFD	CALL NEAR DWORD PTR DS:[46F591]	DS:[0046F591]=00A204F8
00429D08	CALL NEAR DWORD PTR DS:[46F470]	DS:[0046F470]=00A2019B
00429D0E	CALL 00427360	UnPackIt.e.004001D8
00429E2A	CALL NEAR DWORD PTR DS:[46F439]	DS:[0046F439]=00A20069

下面我们就在数据窗口中来定位到 IAT2,我们随便选中一个 CALL,单击鼠标右键选择 Follow in Disassembler,在反汇编窗口中定位到该 CALL。



00A205EC	EB 01	JMP SHORT 00A205EF	
00A205EE	D9		Unknown command
00A205EF	8B5424 04	MOV EDX,DWORD PTR SS:[ESP+4]	
00A205F3	33C0	XOR EAX,EAX	
00A205F5	FF4A 08	DEC DWORD PTR DS:[EDX+8]	
00A205F8	EB 07	JMP SHORT 00A20601	
00A205FA	FFE9	JMP FAR ECX	Illegal use of

经过几次跳转之后来到了这里。

00A205FB	E9 F60AEF7B	JMP 7C9110F6	ntdll.7C9110F6
00A20600	EA EBF8EB01 D96A	JMP FAR 6AD9:01EBF8EB	Far jump
00A20607	14 68	ADC AL,68	
00A20609	A8 8F	TEST AL,8F	

又经过几次跳转以后我们来到了这里:

7C9110E4	90	NOP	
7C9110E8	90	NOP	
7C9110EC	90	NOP	
7C9110ED	8B5424 04	MOV EDX,DWORD PTR SS:[ESP+4]	
7C9110F1	33C0	XOR EAX,EAX	
7C9110F3	FF4A 08	DEC DWORD PTR DS:[EDX+8]	
7C9110F6	75 26	JNZ SHORT 7C91111E	ntdll.7C91111E
7C9110F8	8942 0C	MOV DWORD PTR DS:[EDX+C],EAX	
7C9110FB	F0:FF4A 04	LOCK DEC DWORD PTR DS:[EDX+4]	LOCK prefix
7C9110FF	7D 03	JGE SHORT 7C911104	ntdll.7C911104
7C911101	C2 0400	RETN 4	
7C911104	52	PUSH EDX	
7C911105	E8 4D7F0100	CALL 7C929057	ntdll.RtlUnWaitCriticalSection
7C91110A	33C0	XOR EAX,EAX	
7C91110C	C2 0400	RETN 4	
7C91110F	90	NOP	
7C911110	8DA424 00000000	LEA ESP,DWORD PTR SS:[ESP]	
7C911117	8DA424 00000000	LEA ESP,DWORD PTR SS:[ESP]	
7C91111E	F0:FF4A 04	LOCK DEC DWORD PTR DS:[EDX+4]	LOCK prefix
7C911122	C2 0400	RETN 4	
7C911125	90	NOP	

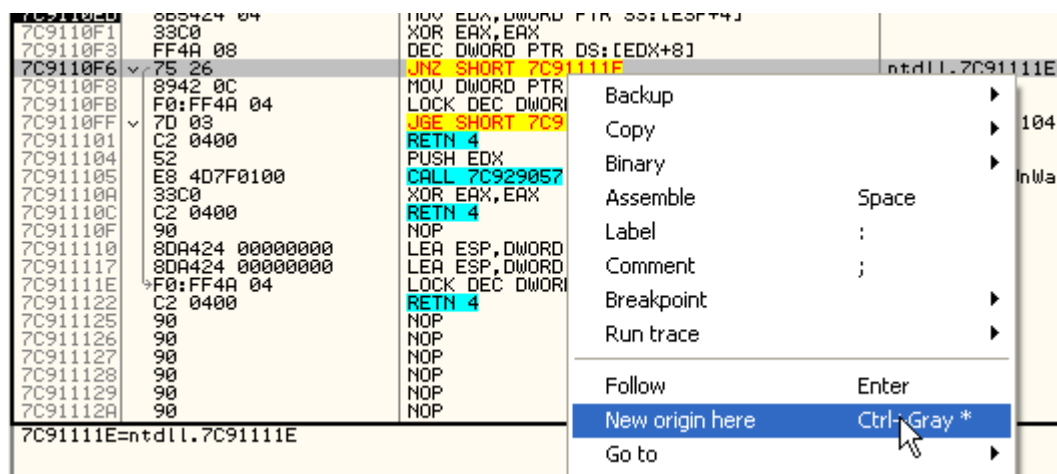
大家细心观察的话,会发现这里我用浅绿色标注出来的这三条指令在前面壳创建的区段中已经执行过了(不记得的话可以按减号键往回翻看),这样做的目的很可能是防止破解者通过在 API 的入口点处设置断点以此来判断是不是被重定向到了该函数。这里如果我们想知道该 API 函数是什么的话,直接在选中 MOV EDX,DWORD PTR SS:[ESP + 4]这条指令,单击鼠标右键选择 New origin here,将 EIP 指针指向该指令。(PS:其实不需要这样做,我们可以直接将反汇编窗口中内存地址与机器码中间的那一条线往右边拖动,就可以看到 API 函数的名称为 RtlLeaveCriticalSection 了)

7C9110E8	90	NOP	
7C9110EC	90	NOP	
7C9110ED	8B5424 04	MOV EDX,DWORD PTR SS:[ESP+4]	
7C9110F1	33C0	XOR EAX,EAX	
7C9110F3	FF4A 08	DEC DWORD PTR DS:[ED	
7C9110F6	75 26	JNZ SHORT 7C91111E	
7C9110F8	8942 0C	MOV DWORD PTR DS:[ED	
7C9110FB	F0:FF4A 04	LOCK DEC DWORD PTR D	
7C9110FF	7D 03	JGE SHORT 7C911104	
7C911101	C2 0400	RETN 4	
7C911104	52	PUSH EDX	
7C911105	E8 4D7F0100	CALL 7C929057	
7C91110A	33C0	XOR EAX,EAX	
7C91110C	C2 0400	RETN 4	
7C91110F	90	NOP	
7C911110	8DA424 00000000	LEA ESP,DWORD PTR SS	
7C911117	8DA424 00000000	LEA ESP,DWORD PTR SS	
7C91111E	F0:FF4A 04	LOCK DEC DWORD PTR D	
7C911122	C2 0400	RETN 4	
7C911125	90	NOP	
7C911126	90	NOP	
7C911127	90	NOP	
7C911128	90	NOP	

这里我们可以看到 EIP 寄存器的右边显示出了该 API 函数的名称为 RtlLeaveCriticalSection。

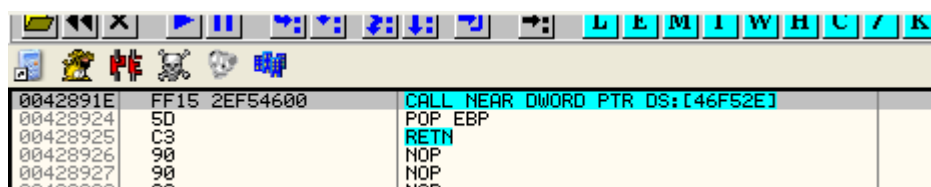
Registers (FPU)	
EAX	0A280105
ECX	00000501
EDX	00000001
EBX	7FFD8000
ESP	0012FF4C
EBP	0012FFC0
ESI	FFFFFFFF
EDI	0046D8EA UnPackMe.0046D8EA
EIP	7C9110ED ntdll.RtlLeaveCriticalSection
C 0	ES 0023 32bit 0(FFFFFFFF)
P 0	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)

好了,这里我们还是将 EIP 的值恢复吧,以免出错。

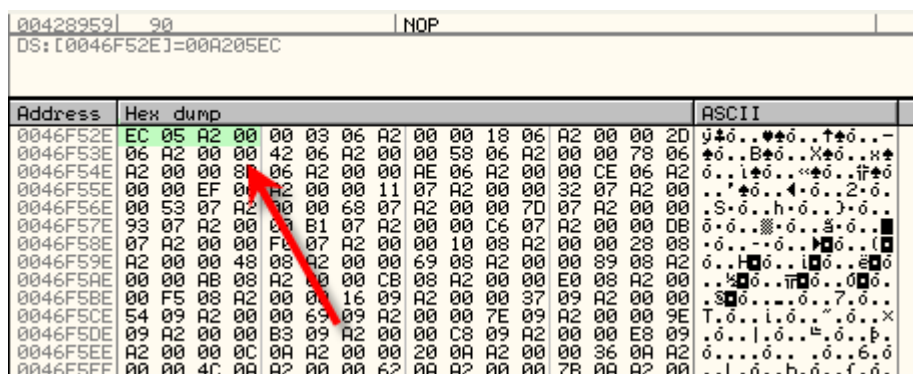


好了,这里我们就知道 IAT2 中的项的确是重定向的 IAT 项。也就是说该壳不会去调用 IAT 中的那些 0006 开头的值(这些值是壳可以构造的,也可以说是混淆视听吧),那么 0006 开头的值占据的 IAT 项怎么办呢?对于这些项该壳会调用 IAT2 中相应的项,这么做的目的无疑是为了增加我们修复 IAT 的难度。我们该如何来修复 IAT 呢?

首先还是来看到 42891E 地址处的这个 CALL(PS:这里该地址大家可能各不相同,以自己机器上的为准)。



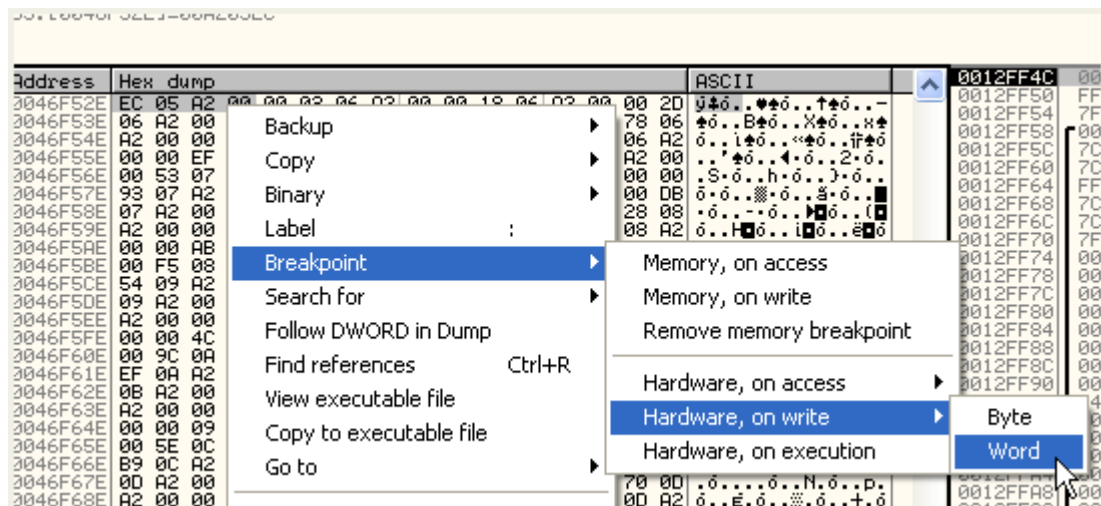
我们已经知道了该 CALL 实际调用的 API 函数是 RtlLeaveCriticalSection。我们可以利用 RtlLeaveCriticalSection 这个函数入口地址来做做文章,首先第一步,我们需要定位到 46F525 这一项的值 00A205EC 是哪写入的。



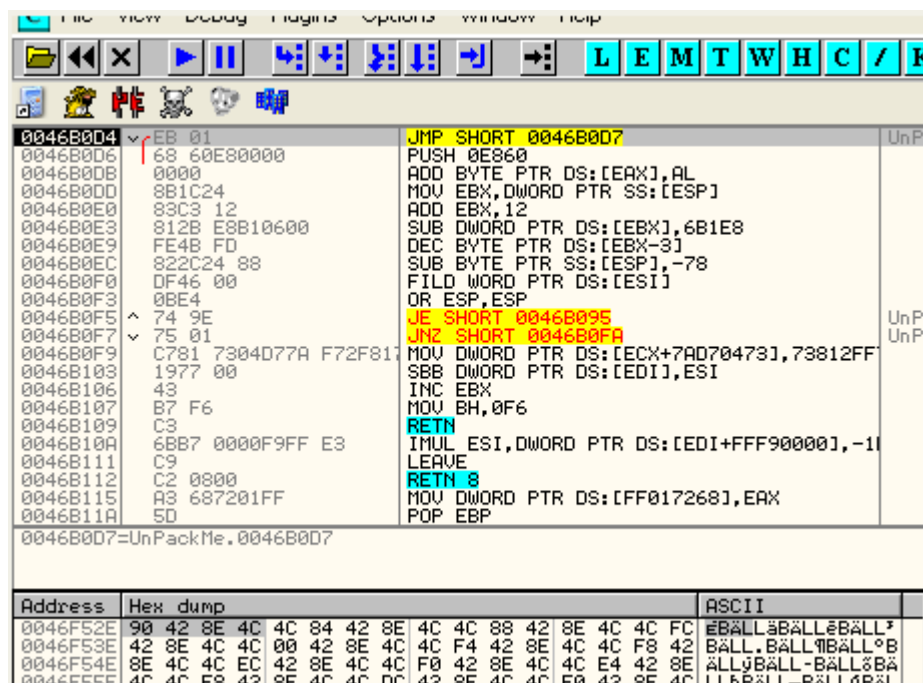
我们定位到了 46F525 这一项写入指令处以后,直接将写入指令中的 00A205EC 替换成 7C9110ED(RtlLeaveCriticalSection 的地址)即可。

这里我们需要想个办法让其直接调用正确的 API 函数,也就说我们需要将该 IAT 项中的内容 00A205EC 替换成 7C9110ED。

首先我们要知道该项是由哪里写入的话,我们需要给该项设置一个硬件写入断点。



下面我们来看看 00A205EC 这个值是哪写入到 46F525 中的,现在我们重启 OD。



断在了入口点处,我们运行起来。

Address	Hex dump	ASCII
0046F52E	EC 05 A2 00 00 48 0A 46 00 00 4C 0A 46 00 00 50	046.H.F..L.F..P
0046F53E	0A 46 00 00 54 0A 46 00 00 58 0A 46 00 00 5C 0A	.F..T.F..X.F..
0046F54E	46 00 00 60 0A 46 00 00 64 0A 46 00 00 68 0A 46	F..'.F..d.F..h.F
0046F55E	00 00 6C 0A 46 00 00 70 0A 46 00 00 74 0A 46 00	..l.F..p.F..t.F.
0046F56E	00 70 00 4C 00 00 7C 00 4C 00 00 0A 00 4C 00 00	..F..F..C..F..

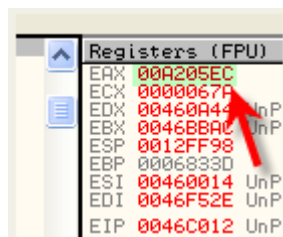
中间会由于写入 46F525 断几次,但是因为写入的值并不是 00A205EC,所以我们继续运行,运行几次以后,我们发现这里写入的是 00A205EC,由于硬件断点是断在下一条指令处,所以我们来看看前一条指令是什么。

Address	Hex dump	ASCII
0046C000	C3	
0046C001	EB 44	
0046C003	9A EB079AEB 01E6	
0046C00A	EB 04	
0046C00C	A1 EBF8D789	
0046C011	07	
0046C012	EB 02	
0046C014	02F5	
0046C016	F9	
0046C017	72 08	

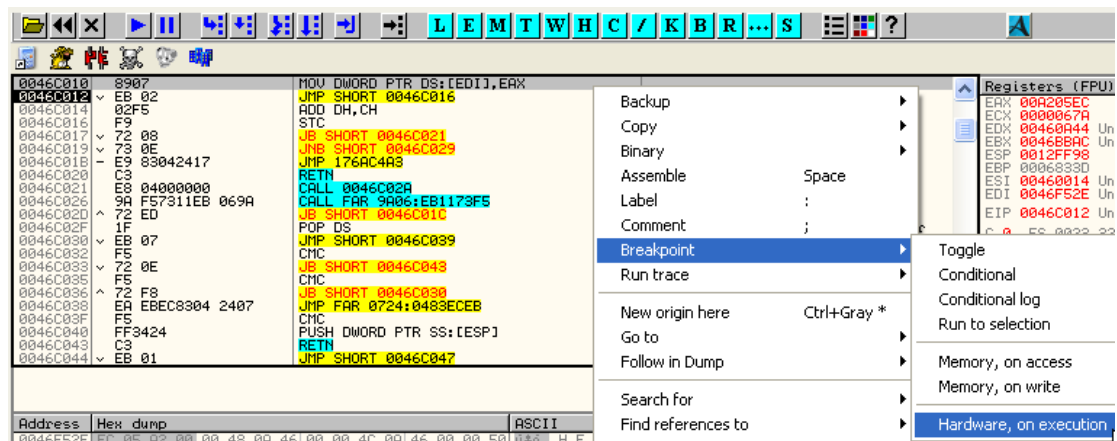
这里的代码被混淆了,看不出个所以来。如果想清楚的看到前一条指令到底是什么的话,我们可以选中前面的一个 JMP 指令,然后单击鼠标右键选择 Follow。

Address	Hex dump	ASCII
0046C000	C3	
0046C001	EB 44	
0046C003	9A EB079AEB 01E6	
0046C00A	EB 04	
0046C00C	A1 EBF8D789	
0046C011	07	
0046C012	EB 02	
0046C014	02F5	
0046C016	F9	

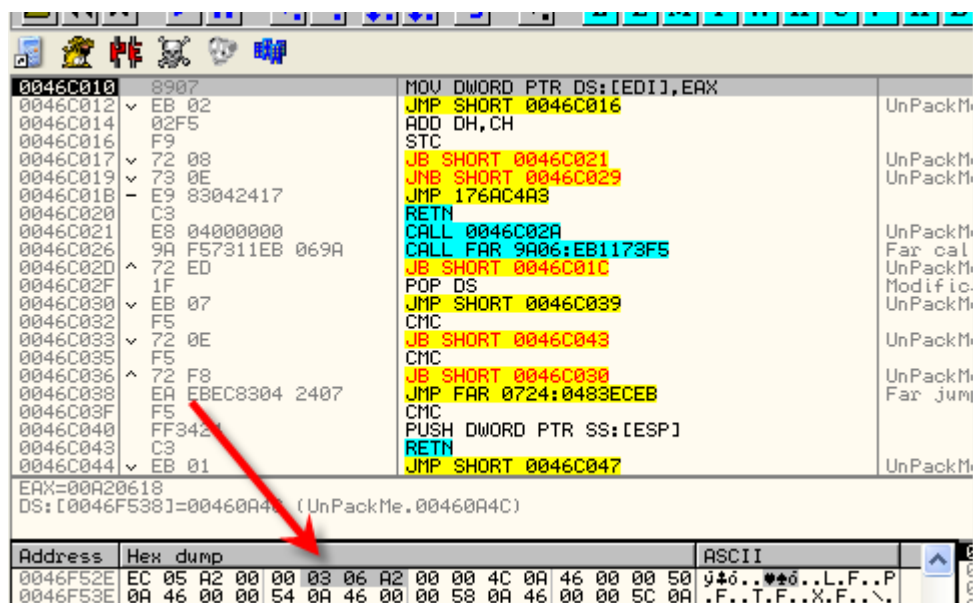
这样就可以清楚的看到前一条指令是什么了,重定向到 IAT2 中值来至于 EAX 寄存器,我们来看看 EAX 此时的值。



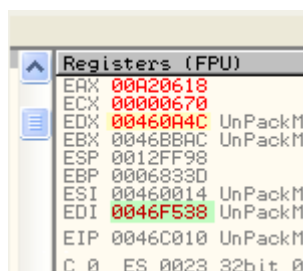
我们可以看到的的确是 00A205EC(PS:每个人机器上可能该值不同,以自己机器的为准),好了,我们已经定位到了关键点了。



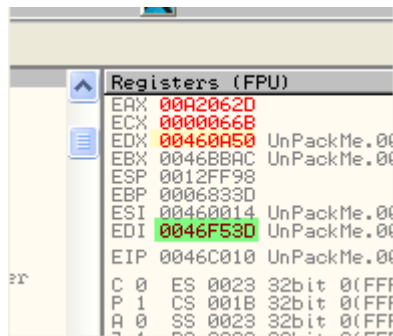
接下来我们给 MOV DWORD PTR DS:[EDI],EAX 这一条指令设置一个硬件执行断点。



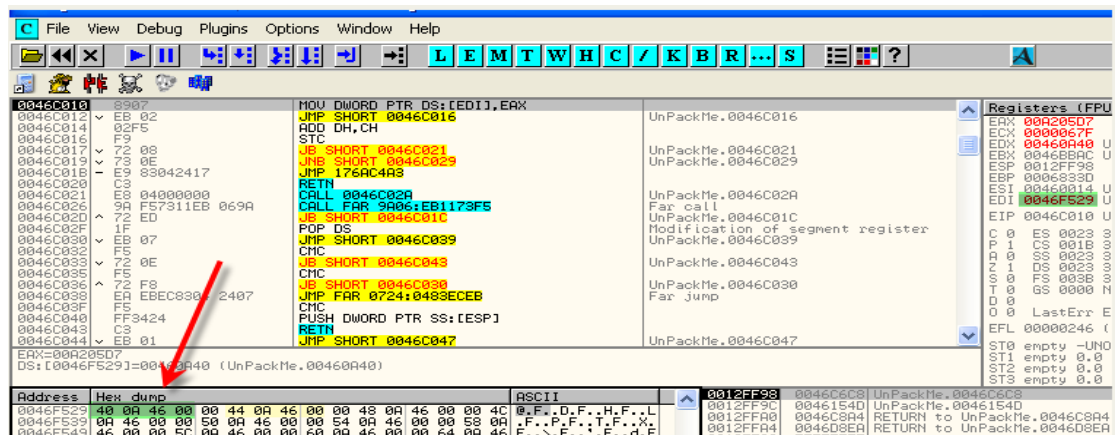
我们运行起来,再次断了下来,会发现继续在写入 46F52E 后面的 IAT2 中的项。我们观察一下寄存器的情况,会发现 EAX 中正好保存的是另一个重定向的值。



我们继续运行,会继续写入下一个重定向的值。



这里我们注意观察 EDI 值的变化,我们会发现 EDI 的值是递增的,指向的都是 IAT2 表中的每一个元素,再看看 EDX 的值,EDX 指向的正好是 IAT 中的项,好,现在我们重启 OD,一直运行,直到断到写入 46F52E 的前一项为止。



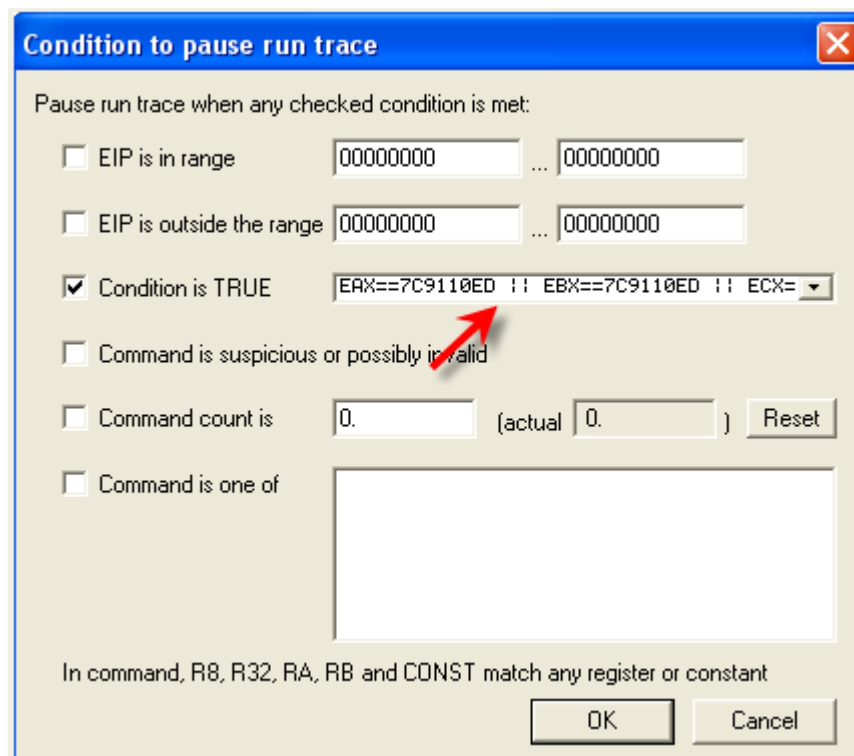
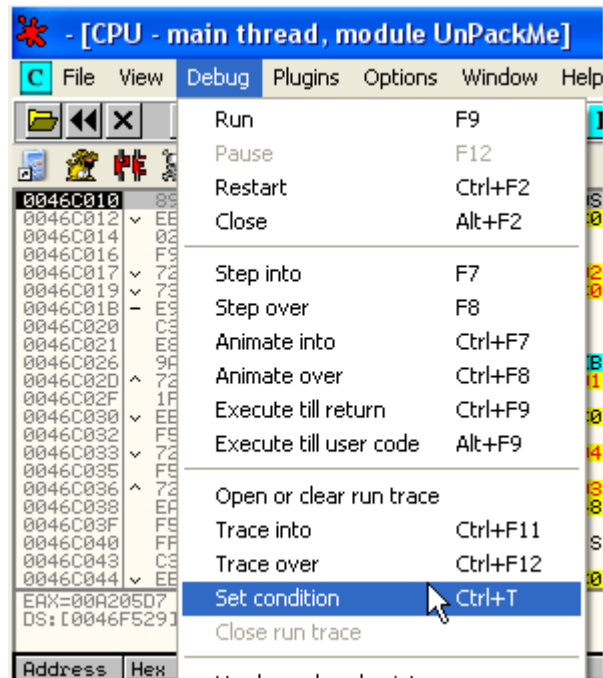
好了,这里由于写入 46F529 所在内存单元的时候断了下来,即 46F52E 的前一项,这里我们知道下一项要写入的是 RtlLeaveCriticalSection 这个 API 函数重定向过的值,这里我们不继续按 F9 运行了,我们利用 OD 自带的跟踪功能来协助我们分析,下一项的正确的 IAT 项值应该 7C9110ED(RtlLeaveCriticalSection 这个 API 函数的首地址)(PS:以自己机器上的地址为准),这里我们来看看哪个寄存器中会出现 7C9110ED 这个值,我们利用 OD 的自动跟踪功能来定位这个值。

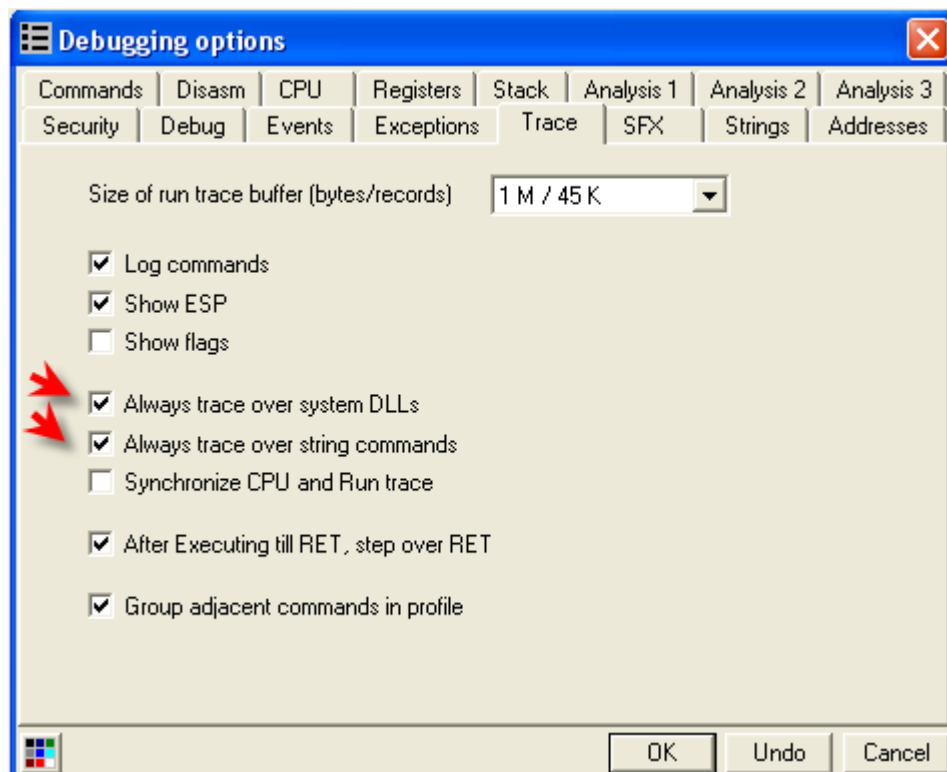
我们设置一个自动跟踪的条件:

EAX == 7C9110ED || EBX == 7C9110ED || ECX == 7C9110ED || EDX == 7C9110ED || ESI == 7C9110ED || EDI == 7C9110ED

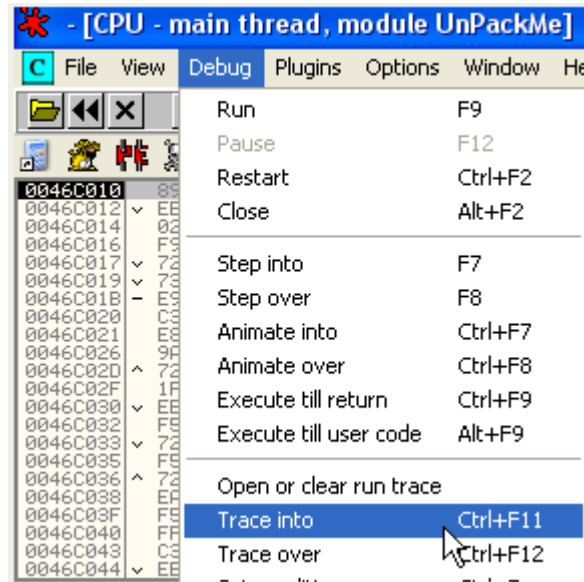
||表示任意一个条件成立即为真。

也就是说当以上寄存器组中任意一个的值等于 7C9110ED 的时候就会停下来。

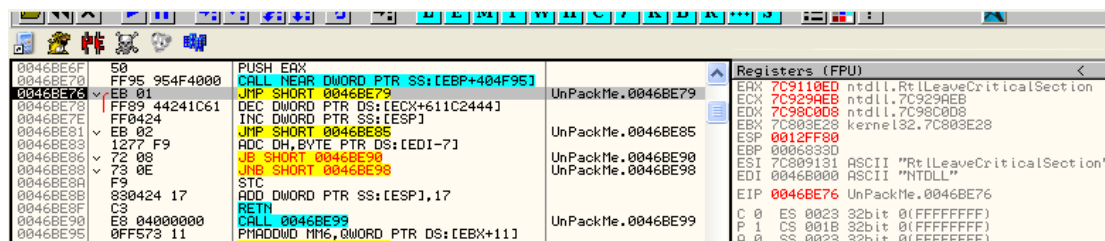




这里我们被忘了勾选上 Debugging options-Trace 菜单项中的 Always trace over system DLLs, Always trace over string commands(这个选项的意思就是遇到诸如 REPS 这样的字符串循环操作指令的时候直接跳过)这两项,因为如果让 OD 自动跟踪系统 DLL 或者字符串操作指令的话耗费的时间可能会非常长,接下来我们选中 Debug-Trace into 菜单项开始自动跟踪。



稍等片刻就会停在这里。



我们可以看到这是一个 CALL 的返回地址,我们来看看跟踪的日志信息。

3.	Main	UnPackMe.0046BE69	CALL NEAR DWORD PTR SS:[EBP+404F90]	EAX=7C910000, ECX=7C801BF6, EDX=00120001, ESP=0012FF7C
2.	Main	UnPackMe.0046BE6F	PUSH EAX	ESP=0012FF78
1.	Main	UnPackMe.0046BE70	CALL NEAR DWORD PTR SS:[EBP+404F95]	EAX=7C9110ED, ECX=7C929AEB, EDX=7C98C0D8, ESP=0012FF80
0.	Main	UnPackMe.0046BE76	JMP SHORT 0046BE79	

我们双击日志中的这一项。

0046BE6F	50	PUSH EAX	
0046BE70	FF95 954F4000	CALL NEAR DWORD PTR SS:[EBP+404F95]	kernel32.GetProcAddress
0046BE76	EB 01	JMP SHORT 0046BE79	UnPackMe.0046BE79
0046BE78	FF89 44241C61	DEC DWORD PTR DS:[ECX+611C2444]	
0046BE7E	FF0424	INC DWORD PTR SS:[ESP]	
0046BE81	EB 02	JMP SHORT 0046BE85	UnPackMe.0046BE85
0046BE83	1277 F9	ADC DH, BYTE PTR DS:[EDI-7]	
0046BE86	72 00	JB SHORT 0046BE88	UnPackMe.0046BE88

(PS:这里我的日志信息中并没有记录 CALL NEAR DWORD PTR SS:[EBP + 404F95]这一项,这里由于我的日志跟踪信息中没有记录这条指令的缘故,所以我这里也显示不出 GetProcAddress 这个 API 函数名称,但是影响并不大)

这里明显可以看出壳在获取 API 函数的地址,接着会重定向到自己创建的区段中。

接下来我们来定位哪里会写入 00A205EC 这个值。

我们将跟踪的条件修改为:

EAX == 00A205EC || EBX == 00A205EC || ECX == 00A205EC || EDX == 00A205EC || ESI == 00A205EC || EDI == 00A205EC

Condition to pause run trace

Pause run trace when any checked condition is met:

☐ EIP is in range 00000000 ... 00000000

☐ EIP is outside the range 00000000 ... 00000000

☒ Condition is TRUE EAX== 00A205EC || EBX== 00A205EC || E

☐ Command is suspicious or possibly invalid

☐ Command count is 0. (actual 16166.) Reset

☐ Command is one of

In command, R8, R32, RA, RB and CONST match any register or constant

OK Cancel

我们再次选择 Debug-trace into 菜单项进行自动跟踪。

0046B8C4	897424 1C	MOV DWORD PTR SS:[ESP+1C],ESI	
0046B8C8	8307 03	ADD DWORD PTR DS:[EDI],3	
0046B8CC	EB 01	JMP SHORT 0046B8CE	UnPackMe.0046B8CE
0046B8D0	D166 C7	SHL DWORD PTR DS:[ESI-39],1	
0046B8D6	06	PUSH ES	
0046B8D1	EB 01	JMP SHORT 0046B8D4	UnPackMe.0046B8D4
0046B8D3	C646 02 D9	MOV BYTE PTR DS:[ESI+2],0D9	
0046B8D7	83C6 03	ADD ESI,3	
0046B8DA	2B02	SUB EDI,EDI	
0046B8DC	EB 01	JMP SHORT 0046B8DF	UnPackMe.0046B8DF
0046B8DE	EA EB04E9EB 0400	JMP FAR 0004:EB04E9EB	Far jump
0046B8E5	EB FB	JMP SHORT 0046B8E2	UnPackMe.0046B8E2
0046B8E7	FF83 FA12737B	INC DWORD PTR DS:[EBX+7B7312FA]	

Registers (FPU)

EAX 7C9110ED ntdll.RtlLeaveCriticalSection

ECX 0046C6C8 UnPackMe.0046C6C8

EDX 00460A44 UnPackMe.00460A44

EBX 0046BBAC UnPackMe.0046BBAC

ESP 0012FF74

EBP 0012FF94

ESI 00A205EC

EDI 0046D218 UnPackMe.0046D218

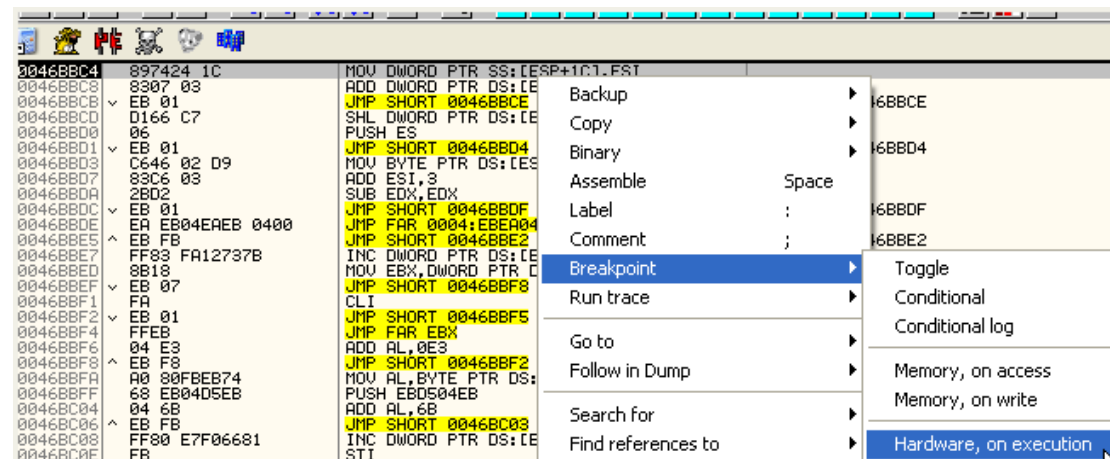
EIP 0046B8C4 UnPackMe.0046B8C4

C 0 ES 0023 32bit 0(FFFFFFFF)

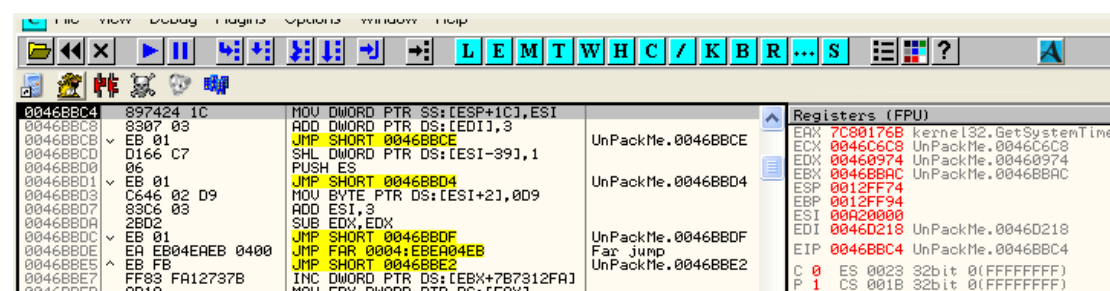
P 0 CS 001B 32bit 0(FFFFFFFF)

停在了这里,此时 EAX 指向的是正确的 API 函数地址,而 ESI 指向了重定向过的值,嘿嘿。这个点非常完美。我们删除之前创建的

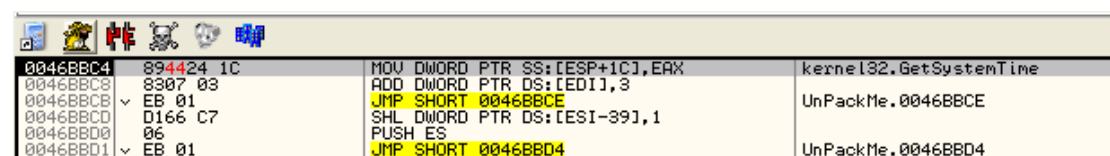
硬件断点。接着给 MOV DWORD PTR SS:[ESP+1C],ESI 这一条指令处设置硬件执行断点。



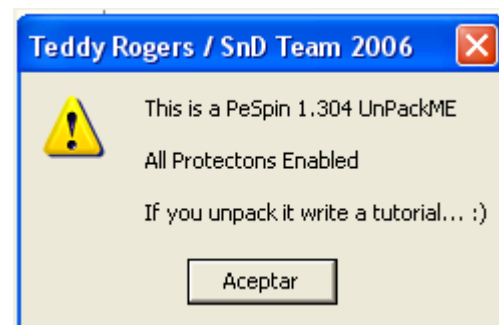
我们重启 OD,然后直接运行起来,可以看到第一次就断在了这里。



我们可以看到此时 EAX 同样指向了一个正确的 API 函数-GetSystemTime,而 ESI 寄存器也同样指向了一个重定向过的值,所以我们来 Patch 这条指令以此来修复 IAT2。



这里我们将 MOV DWORD PTR SS:[ESP+1C],ESI 这条指令修改为 MOV DWORD PTR SS:[ESP+1C],EAX,这样就用正确的 API 函数地址替代了重定向过的值,接着我们删除之前设置的硬件断点,运行起来。



我们来看看 IAT2 发生了什么变化。

0046BC04	04 6B	ADD HL,6B	UnPackMe.00
0046BC06	EB FB	JMP SHORT 0046BC03	

Address	Hex dump	ASCII	
0046F529	C9 25 81 7C 00 ED 10 91 7C 00 36 8F 83 7C 00 8A	fZu!..y!..6A!..e	0012F
0046F539	18 92 7C 00 0F 2B 81 7C 00 A1 9F 80 7C 00 7C 36	tE!..*+u!..ifC!..16	0012F
0046F549	81 7C 00 52 70 82 7C 00 50 F8 81 7C 00 00 FD 81	u!..Rp!..P!..!u	0012F
0046F559	7C 00 92 FE 81 7C 00 77 98 80 7C 00 58 CD 80 7C	!..E!..u!..w!..X=C!	0012F
0046F569	00 A6 0D 81 7C 00 9F 0F 81 7C 00 0E 18 80 7C 00	..z!..f!..u!..#tC!	0012F
0046F579	24 1A 80 7C 00 00 0D E0 80 7C 00 16 E0 80 7C 00	z+C!..oC!..oC!..)	0012F
0046F589	C7 80 7C 00 97 AA 80 7C 00 62 5F 82 7C 00 9A E1	AC!..u-C!..b..e!..0B	0012F
0046F599	81 7C 00 29 89 80 7C 00 19 99 80 7C 00 05 A4 80	u!..)jC!..+oC!..+AC	0012F
0046F5A9	7C 00 66 EA 80 7C 00 EC E9 80 7C 00 83 9E 80 7C	!..foC!..u!..jxC!	0012F
0046F5B9	00 29 9F 80 7C 00 79 EE 81 7C 00 AD 9C 80 7C 00	..)fC!..u-u!..!eC!	0012F
0046F5C9	C7 A0 80 7C 00 E0 C6 80 7C 00 B9 8F 83 7C 00 37	AC!..oAC!..j!..A!..7	0012F
0046F5D9	97 80 7C 00 19 01 81 7C 00 82 00 81 7C 00 57 B3	u!..+0u!..e..u!..u!	0012F
0046F5E9	80 7C 00 AB 14 81 7C 00 F4 97 80 7C 00 01 B0 85	C!..%u!..u!..C!..0!..a	0012F
0046F5F9	7C 00 19 62 82 7C 00 5C E8 81 7C 00 53 00 83 7C	!..+be!..p!..S..A!	0012F
0046F609	00 19 3C 87 7C 00 CB 08 81 7C 00 C1 0F 87 7C 00	..+C!..pi!..+*C!	0012F
0046F619	5B 82 81 7C 00 E9 06 87 7C 00 4E 99 80 7C 00 AC	[u!..o+C!..N!..C!..%	0012F
0046F629	92 80 7C 00 11 07 87 7C 00 42 24 80 7C 00 F3 B8	EC!..+C!..B!..C!..%0	0012F
0046F639	81 7C 00 A9 2C 87 7C 00 F4 2C 87 7C 00 BA 38 87	u!..@..C!..u!..C!.. 0C	0012F
0046F649	7C 00 0C 6E 82 7C 00 F1 BA 80 7C 00 30 31 87 7C	!..ne!..+jC!..=1C!	0012F
0046F659	00 83 31 87 7C 00 CC 37 87 7C 00 77 1D 80 7C 00	..a!..C!..f7C!..w+C!	0012F
0046F669	28 AC 80 7C 00 66 AA 80 7C 00 A9 2C 81 7C 00 ED	(%C!..f-C!..@..u!..y	0012F
0046F679	CB 81 7C 00 3D 00 87 7C 00 19 90 83 7C 00 59 35	pu!..=..C!..+E!..V5	0012F
0046F689	81 7C 00 31 03 92 7C 00 40 03 92 7C 00 D7 EF 80	u!..1!..E!..@!..E!..i!..C	0012F
0046F699	7C 00 2D FF 80 7C 00 2F FE 80 7C 00 51 28 81 7C	!..-C!..!..C!..Q(u!	0012F
0046F6A9	00 11 03 81 7C 00 B1 C7 80 7C 00 65 A0 80 7C 00	..+u!..%AC!..e!..C!	0012F
0046F6B9	CF C6 80 7C 00 21 2E 82 7C 00 BD 99 80 7C 00 88	AC!..+..e!..c!..C!..e	0012F
0046F6C9	2D 82 7C 00 5D 99 80 7C 00 94 97 80 7C 00 7B 97	-e!..j!..C!..o!..C!..t!..u	0012F
0046F6D9	80 7C 00 29 B5 80 7C 00 CF C6 80 7C 00 00 EA D6	C!..)AC!..oAC!..u!	0012F
0046F6E9	D1 77 00 93 B6 05 77 00 7D B5 05 77 00 27 BE D1	0w..oA!..w..oA!..w..*#0	0012F
0046F6F9	77 00 28 8E D1 77 00 DC 10 D2 77 00 3E F1 D2 77	w..(A!..w..>+E!..w	0012F
0046F709	00 59 A2 D5 77 00 C6 B5 D1 77 00 E8 0F D2 77 00	..Vo!..w..oA!..w..*#E!..w	0012F
0046F719	7E 5E D5 77 00 AB 8E D1 77 00 ED E5 D1 77 00 41	..^!..w..%A!..w..y!..0!..w..A	0012F
0046F729	8D 01 77 00 42 10 D2 77 00 95 FB D2 77 00 76 BD	c!..w..B!..E!..w..o!..E!..w..v!..c	0012F
0046F739	D1 77 00 28 8D 01 77 00 11 12 D2 77 00 90 0F D2	0w..+!..0!..w..+!..E!..w..E!..w	0012F
0046F749	77 00 9D 0B D2 77 00 84 BA D5 77 00 90 57 D5 77	w..o!..E!..w..+!..j!..w..o!..w	0012F
0046F759	00 DA C6 D3 77 00 9D A1 D5 77 00 1D B6 D1 77 00	..r!..E!..w..o!..w..+A!..w	0012F
0046F769	09 B6 D1 77 00 21 90 D1 77 00 F9 FF D4 77 00 DA	..A!..w..t!..E!..w..+..E!..w..r	0012F

好,我们可以看到现在 IAT2 中保存的都是正确的 API 函数地址了。

我们看下 46F52E 调用指令的情况试试:

0042890F	5D	PUSH EBX			
00428910	E8 4BFFFFFF	CALL 00428860	UnPackMe.00428860	ESI 000	
00428915	83C4 04	ADD ESP,4		EDI 000	
00428918	5D	POP EBP		EIP 7C9	
00428919	C3	RETN		C 0 ES	
0042891A	83C0 20	ADD EAX,20		P 1 CS	
0042891D	59	PUSH EAX		A 0 SS	
0042891E	FF15 2EF54600	CALL NEAR DWORD PTR DS:[46F52E]	ntdll.RtlLeaveCriticalSection	Z 1 DS	
00428924	5D	POP EBP		T 0 FS	
00428925	C3	RETN		O 0 GS	
00428926	90	NOP		O 0 La	
00428927	90	NOP		EFL 000	
00428928	90	NOP		ST0 emp	
00428929	90	NOP		ST1 emp	
0042892A	90	NOP		ST2 emp	
0042892B	90	NOP		ST3 emp	
0042892C	90	NOP		ST4 emp	
0042892D	90	NOP		ST5 emp	

DS:[0046F52E]=7C9110ED (ntdll.RtlLeaveCriticalSection)					
--	--	--	--	--	--

Address	Hex dump	ASCII			
0046F52E	ED 10 91 7C 00 36 8F 83 7C 00 8A 18 92 7C 00 0F	fZu!..y!..6A!..e	0012E03C	01130000	
0046F53E	2B 81 7C 00 A1 9F 80 7C 00 7C 36 81 7C 00 52 70	+u!..ifC!..!6u!..Rp	0012EA40	7C920732	RETURN to ntdll.7C92
0046F54E	82 7C 00 50 F8 81 7C 00 00 32 FE 81 e!..P!..!u!..f!..u		0012EA44	00000011	
0046F55E	7C 00 77 98 80 7C 00 58 CD 80 7C 00 A6 80 81 7C	!..w!..X=C!..a!..u!	0012EA48	011309B8	
0046F56E	00 9F 0F 81 7C 00 0E 18 80 7C 00 24 1A 80 7C 00	..f!..u!..#tC!..z+C!	0012EA4C	01130000	
0046F57E	24 1A 80 7C 00 00 0D E0 80 7C 00 16 E0 80 7C 00	..z+C!..oC!..oC!..)	0012EA50	00000000	
0046F58E	C7 80 7C 00 97 AA 80 7C 00 62 5F 82 7C 00 9A E1	AC!..u-C!..b..e!..0B	0012EA54	0012EA44	

这里我们可以看到调用的确是正确的 API 函数了,不再是重定向的值了。但是这里仍然是 IAT2 中的值被修复了,IAT 中哪些 0006 开头的无效的项仍然没有被修复,所以下一步我们要将 IAT 中无效的值修复。

现在我们有二个切入点,一个就是 46BBC4 这一条指令,我们需要将 ESI 修改为 EAX,这样就可以确保正确的 API 函数地址被填充到 IAT2 中。

另一个切入点是 46C010 这一条指令,我们分别给该处指令设置硬件执行断点,接着重启 OD。

0046BBC4	897424 1C	MOV DWORD PTR SS:[ESP+1C],ESI		
0046BBC8	8307 03	ADD DWORD PTR DS:[EDI],3		
0046BBCB	EB 01	JMP SHORT 0046BBCE	UnPackMe.0046BBCE	
0046BBCD	D166 C7	SHL DWORD PTR DS:[ESI-39],1		
0046BBDB	06	PUSH ES		
0046BBDD	EB 01	JMP SHORT 0046BBDF	UnPackMe.0046BBDF	
0046BBDE	C646 02 D9	MOV BYTE PTR DS:[ESI+2],0D9		
0046BBE0	83C6 03	ADD ESI,3		
0046BBE2	2BD2	SUB EDX,EDX		
0046BBE4	EB 01	JMP SHORT 0046BBDF	UnPackMe.0046BBDF	

断在了这里,我们将 ESI 修改为 EAX。

0046B8C4	89 42 4 1C	MOV DWORD PTR SS:[ESP+1C],EAX	kernel32.GetSystemTime
0046B8C8	83 07 03	ADD DWORD PTR DS:[EDI],3	UnPackMe.0046BBCE
0046B8CB	EB 01	JMP SHORT 0046BBCE	UnPackMe.0046BBCE
0046B8CD	D1 66 C7	SHL DWORD PTR DS:[ESI-39],1	UnPackMe.0046BBCE
0046B8D0	06	PUSH ES	UnPackMe.0046BBCE
0046B8D1	FR 01	JMP SHORT 0046BBCE	UnPackMe.0046BBCE

我们运行起来。

0046C010	89 07	MOV DWORD PTR DS:[EDI],EAX	kernel32.GetSystemTime
0046C012	EB 02	JMP SHORT 0046C016	UnPackMe.0046C016
0046C014	02 F5	ADD DH,CH	UnPackMe.0046C016
0046C016	F9	STC	UnPackMe.0046C016
0046C017	72 08	JB SHORT 0046C021	UnPackMe.0046C021
0046C019	73 0E	JNB SHORT 0046C029	UnPackMe.0046C029
0046C01B	E9 83 04 24 17	JMP 176AC4A3	UnPackMe.0046C02A
0046C01D	C3	RETN	UnPackMe.0046C02A
0046C01E	E8 04 00 00 00	CALL 0046C02A	UnPackMe.0046C02A
0046C020	9A F5 73 11 E8 06 9A	CALL FAR 9A06:EB1173F5	UnPackMe.0046C01C
0046C022	72 ED	JB SHORT 0046C01C	UnPackMe.0046C01C
0046C024	1F	POP DS	UnPackMe.0046C039
0046C026	EB 07	JMP SHORT 0046C039	UnPackMe.0046C039
0046C028	F5	CMC	UnPackMe.0046C039

断在了这里,这里是将正确的 API 函数地址填充到 IAT2 中,此时 EDX 指向了对应的 IAT 项,而 EDI 指向了 IAT2 中对应的项,现在是我们修复 IAT 的绝佳时机,我们做如下操作即可:

0046C00A	EB 04	JMP SHORT 0046C010	UnPackMe.0046C010
0046C00C	A1 EBF8D789	MOV EAX,DWORD PTR DS:[89D7F8EB]	Modification of segment register
0046C00E	07	POP ES	UnPackMe.0046C016
0046C010	EB 02	JMP SHORT 0046C016	UnPackMe.0046C016
0046C012	02 F5	ADD DH,CH	UnPackMe.0046C021
0046C014	F9	STC	UnPackMe.0046C021
0046C016	72 08	JB SHORT 0046C021	UnPackMe.0046C021

我们知道上面这条 JMP 指令是直接跳转到 MOV DWORD PTR DS:[EDI],EAX 这条指令的,这里压根我们就不需要其无条件跳转,这条指令相当于是废的,我们将其 NOP 掉。

Edit data at 0046C00A

ASCII

éééééé

UNICODE

HEX +06

90 90 90 90 90 90

☒ Keep size

OK

Cancel

如下:

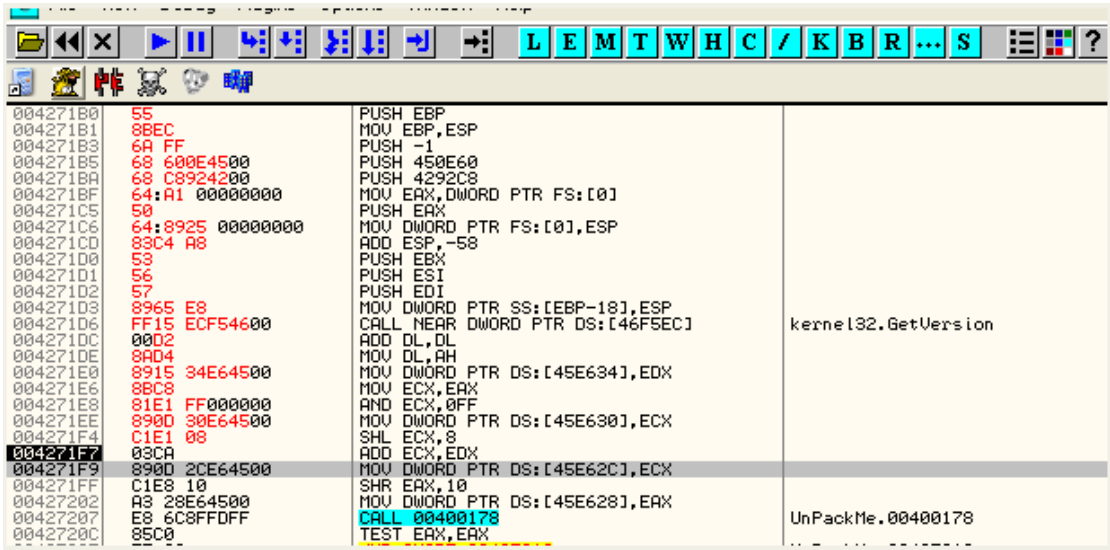
0046C00A	90	NOP	
0046C00B	90	NOP	
0046C00C	90	NOP	
0046C00D	90	NOP	
0046C00E	90	NOP	
0046C00F	90	NOP	
0046C010	89 07	MOV DWORD PTR DS:[EDI],EAX	kernel32.GetSystemTime
0046C012	EB 02	JMP SHORT 0046C016	UnPackMe.0046C016
0046C014	02 F5	ADD DH,CH	UnPackMe.0046C016

这里我们在上面添加一条指令 MOV DWORD PTR DS:[EDX],EAX,这样可以将正确 API 函数地址就被写入到了 IAT 中了。

Address	Hex dump	ASCII
004607E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004607F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00460804	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00460814	00 00 00 00 F0 6B DA 77 1B 76 DA 77 F4 EA DA 77-k r w + v r w 0 u r w
00460824	E7 EB DA 77 33 78 DA 77 00 00 00 00 DD 15 C5 58	é U r w Å k r w! \$ + X
00460834	2E B0 C3 58 00 00 00 00 04 6A EF 77 66 95 EF 77	. c [Xé j w f 0 - w
00460844	89 6A EF 77 F3 AD EF 77 ED D9 EF 77 99 8B EF 77	é j w Å s w + w j w 0 i w
00460854	0C B5 EF 77 2A 7D EF 77 B2 7C EF 77 77 53 F2 77	! Å w * w Å j w w S = w
00460864	1E C9 F1 77 0C BC EF 77 52 04 EF 77 FA 8D EF 77	Å F r w . w Å R E w . i w
00460874	F1 D0 EF 77 51 B2 EF 77 26 05 EF 77 2A E3 EF 77	z : w Å C Å w Å w 0 - w
00460884	5F 39 F2 77 11 84 EF 77 2E AD EF 77 E1 61 EF 77	- 9 w Å C w . i w Å B a w
00460894	B8 85 EF 77 CC D2 EF 77 43 70 EF 77 FB EA F0 77	0 Å w Å F r w Å C p w Å 0 - w
004608A4	12 83 EF 77 01 72 F0 77 A9 34 F0 77 D5 93 EF 77	0 Å w Å 0 z - w Å 0 - w Å 0 w
004608B4	68 EF EF 77 AA D2 EF 77 B2 6F EF 77 3F 38 F2 77	h w Å R E w Å 0 0 w Å 78 = w
004608C4	D6 E8 EF 77 6A E0 EF 77 00 60 EF 77 90 5B EF 77	i B w Å 0 - w . w Å E L w
004608D4	6D AC EF 77 94 6C F0 77 22 8D EF 77 3D C8 F1 77	n Å w Å 0 - w Å L w Å = z w
004608E4	3D 60 F0 77 6F 0C EF 77 85 78 EF 77 26 D9 EF 77	= n w Å 0 w Å Å C w Å 0 - w
004608F4	FB 5E EF 77 36 8A EF 77 FC 8A EF 77 0F 62 EF 77	! Å w Å 0 w Å E w Å w B w
00460904	49 5E EF 77 97 5D EF 77 1A 9A EF 77 68 FA EF 77	I Å w Å j w w Å u Å k w
00460914	78 C9 F0 77 DA 98 F2 77 1A 40 F2 77 55 EA EF 77	C (F w Å r y = w Å 0 = w Å 0 w
00460924	C5 61 EF 77 70 E6 EF 77 F0 81 EF 77 2D 6C EF 77	+ Å w Å w p w w Å u w Å l w
00460934	98 6E EF 77 4F 83 EF 77 09 E0 EF 77 EB AA EF 77	g n w Å 0 Å w Å u Å w Å w
00460944	26 69 F0 77 B1 95 EF 77 6F 80 EF 77 8A 5A EF 77	& i w Å 0 0 w Å C w Å w Å Z w
00460954	E9 49 F2 77 26 F1 F0 77 C9 D0 F0 77 51 E0 F0 77	0 i w Å = w Å - w F r - w Å 0 - w
00460964	33 8C EF 77 6C EC EF 77 29 94 EF 77 00 00 00 00	3 i w Å j w Å 0 w Å w Å . . .
00460974	B6 16 06 00 C1 C9 80 7C 69 10 81 7C EE 1E 80 7C	Å Å . - Å F C i Å u i Å C Å i
00460984	80 2C 81 7C 40 7A 94 7C E1 EA 81 7C A2 CA 81 7C	i Å u Å 0 0 Å 0 u Å 0 Å u
00460994	16 1E 80 7C 43 99 80 7C 10 11 81 7C 29 29 81 7C	- Å C Å C Å C Å Å u Å u Å u
004609A4	14 98 80 7C 81 9A 80 7C FB 2C 82 7C AE 94 83 7C	Å Å C Å u Å u Å u . Å Å Å Å
004609B4	5F 2E 83 7C 4A CE 80 7C 8A 28 86 7C 3F DC 81 7C	+ Å Å - Å F C Å Å Å i Å u Å u
004609C4	25 48 81 7C 23 CC 81 7C 78 2C 81 7C 86 03 81 7C	- Å u Å Å Å u Å u Å Å u Å u
004609D4	B9 8C 83 7C 80 AA 80 7C 57 B8 80 7C 7E DA 80 7C	Å Å Å Å C Å C Å u Å C Å C Å C
004609E4	72 17 81 7C 93 D2 80 7C 4E A3 80 7C A9 26 82 7C	r Å u Å 0 Å C Å u Å C Å 0 Å E
004609F4	ED 09 92 7C 7B 79 92 7C DA 05 92 7C 30 04 92 7C	Y Å E Å Y Å E Å Å Å Å Å Å
00460A04	2A E8 81 7C E6 2B 81 7C 73 B0 85 7C 39 30 82 7C	* Å u Å p Å u Å Å Å Å Å 0 Å E
00460A14	E2 F8 81 7C 8F 0C 81 7C 4C 17 81 7C A1 97 83 7C	0 Å u Å Å u Å L Å u Å u Å Å
00460A24	B7 2B 82 7C 96 29 81 7C F0 78 82 7C 50 97 80 7C	A Å E Å u Å u Å Å E Å F Å C Å i

这里我们可以看到 IAT 中之前那些 0006 开头的值也已经被修复了。好了,下面我们要做的就是将代码段中的这些间接 CALL IAT2 中的项转化为 CALL IAT 中的项,下面我们就通过一个简单的脚本来完成。

我会给大家逐一介绍每一条语句是干什么用的,但是在解释脚本之前我们先来修复一下 stolen bytes。



这里脚本我已经写好了,可能不是最优的,但是已经足够用了,名称叫做 Spin.txt。

```

0000 var var_call
0001 var var_table
0002 var var_api
0003 var var_iat
0004 var var_program
0005 var var_end
0006 mov var_program,401000
0007 mov var_iat,460818
0008 TheStart:
0009 findop var_program,#FF15??#
0010 log $RESULT
0011 mov var_call,$RESULT
0012 cmp var_call,0
0013 je TheFinal
0014 cmp var_call,449048
0015 jae TheFinal
0016 TheFollow:
0017 add var_call,2
0018 log var_call
0019 mov var_table,[var_call]
0020 log var_table
0021 mov var_api,[table]
0022 log var_api
0023 cmp var_api,50000000
0024 jb TheJump
0025 TheLoop:
0026 cmp var_api,[var_iat]
0027 je TheSolve
0028 add var_iat,4
0029 cmp var_iat,460F28
0030 jae TheJump
0031 jmp TheLoop
0032 TheSolve:
0033 log var_iat
0034 log var_call
0035 mov [var_call],var_iat
0036 cmp var_call,449068
0037 jae TheFinal
0038 TheJump:
0039 sti
0040 mov eip,4271F7
0041 mov var_program,var_call
0042 log var_program
0043 mov var_iat,460818
0044 jmp TheStart
0045 TheFinal:
0046 ret

```

首先:

```

var var_call
var var_table
var var_api
var var_iat
var var_program
var var_end

```

这里是声明该脚本中需要用到的一些变量。下面我会给大家介绍每个变量的功能。

```

mov var_program,401000
mov var_iat,460818

```

这里是初始化变量,将主模块代码段的起始地址赋值给变量 `var_program`,因为我们要从主模块的代码段起始地址处开始定位 CALL。接着将 IAT 的起始地址赋值给变量 `var_iat`。

TheStart:

```

findop var_program,#FF15??#
log $RESULT

```

这里才是我们脚本真正开始执行的地方。`TheStart` 是一个标签,顾名思义:开始。接下来是通过 `findop` 命令从 401000 地址处开

始查找以 FF15 开头的间接 CALL, ??表示通配符。如果查找成功, 地址将会保存到保留变量\$RESULT 中, 否则\$RESULT 将等于 0。

接下来通过 log 命令将\$RESULT 的记录到 OD 的日志窗口中, 这个命令不是必须的, 只是为了方便我们的查看。

```
mov var_call, $RESULT
cmp var_call, 0
je TheFinal
cmp var_call, 44904B
jae TheFinal
```

这里首先将查找到结果\$RESULT 赋值给变量 var_call, 接着判断查找的结果是否为 0, 如果为 0, 表示没有找到以 FF15 开头的间接 CALL, 那么就直接跳转到 TheFinal 标签处结束该脚本的执行, 如果找到了, 判断间接 CALL 指令的地址是否高于代码段中最后一项间接 CALL 指令所在的地址, 如果高于, 同样跳转到 TheFinal 标签处结束脚本的执行。

TheFollow:

```
add var_call, 2
log var_call
mov var_table, [var_call]
log var_table
mov var_api, [table]
log var_api
cmp var_api, 50000000
jb TheJump
```

这里将间接 CALL 指令的地址+2, 然后读取 4 字节的内容保存到变量 var_table 中, 也就是 IAT2 中的表项, 接着读取表项的值也就是读取 API 函数的地址, 判断其是否大于 50000000, 如果大于说明是正常的 API 函数地址, 如果小于就说明不是正确的 API 函数地址, 我们跳转至 TheJump 标签处继续定位下一个间接 CALL。

TheLoop:

```
cmp var_api, [var_iat]
je TheSolve
add var_iat, 4
cmp var_iat, 460F28
jae TheJump
jmp TheLoop
```

我们现在有 IAT2 中正确的 API 函数的地址了, 所以接着我们就需要查询 IAT 中跟其一致的 API 函数地址, 如果是 API 函数地址相等的话, 就可以将该间接 CALL 中的 IAT2 中的项替换成 IAT 中的项了, 但是如果查遍整个 IAT 都没有查找到与之相等 API 函数地址的话, 就跳转到 TheJump 标签处进行定位下一个间接 CALL, 如果在 IAT 中查找到了相等的 API 函数地址, 就跳转到 TheSolve 标签处进行替换工作。

TheSolve:

```
log var_iat
log var_call
mov [var_call], var_iat
```

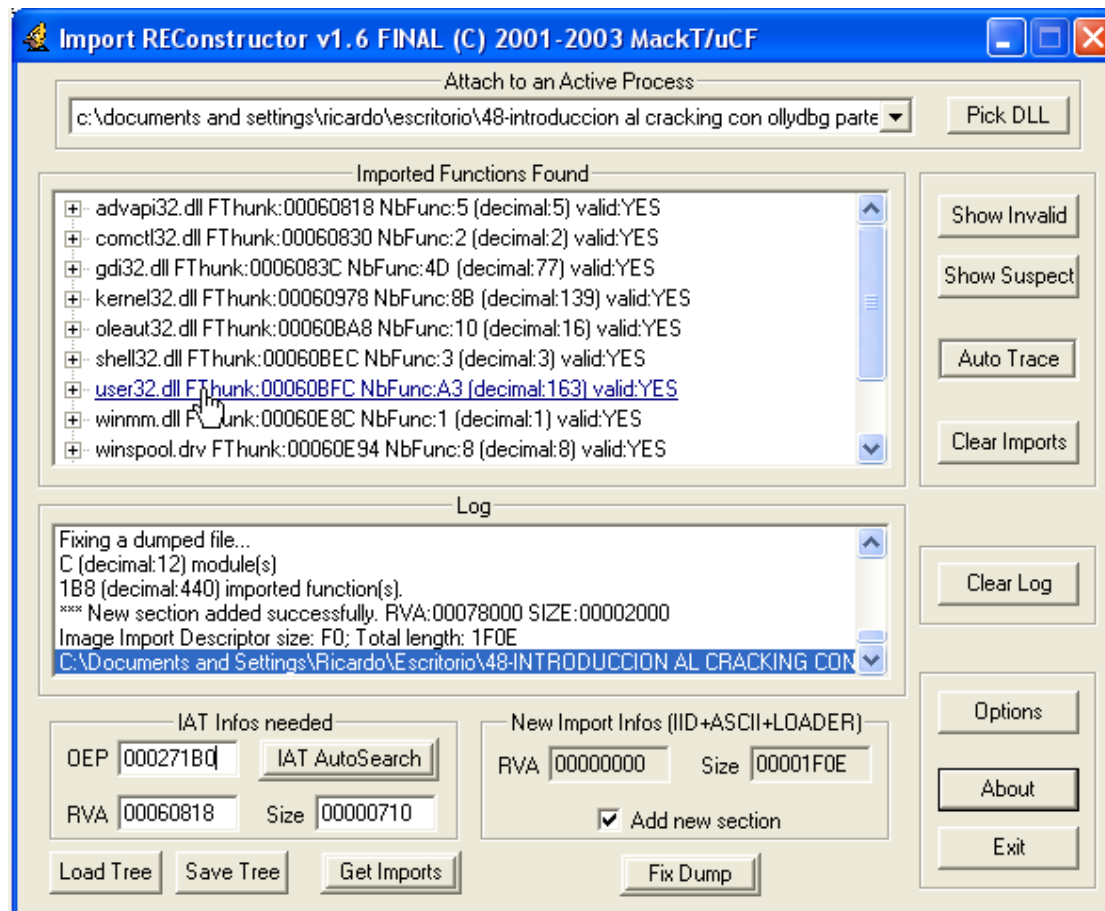
这里将 IAT2 的项值替换成对应的 IAT 的项值。

```
sti
mov eip, 4271F7
```

这两行其实不起作用, 但是由于 OllyScript 插件要求程序必须执行一些东西, 不能单单的进行搜索或者更改值, 或者挂起什么也不做, 所以我们不得不添加一个 STL 命令, 让其单步执行, 相当于 F7, 然后再将 eip 指向伪造的 OEP 处。

这里 OD 有一点不好的地方, 就是当有些脚本 Patch 过的字节超过 1000 的时候就弹出一个消息框进行提示, 我们点击几次 OK 以后脚本就执行完毕了, IAT 也就修复成功了。

好了, 现在我们来 dump, 然后用 IMP REC 修复 IAT。



这里我们发现存在一个无效的项,我们直接将其剪切掉。接着我们修复刚刚 dump 出来的文件,我们会发现无法运行,好像还缺少点什么。

004271AD	90	NOP	
004271AE	90	NOP	
004271AF	90	NOP	
004271B0	55	PUSH EBP	
004271B1	8BEC	MOV EBP,ESP	
004271B3	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH 450E60	
004271BA	68 C8924200	PUSH 4292C8	
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
004271C5	50	PUSH EAX	
004271C6	64:8925 000000	MOV DWORD PTR FS:[0],ESP	
004271CD	83C4 A8	ADD ESP,-58	
004271D0	53	PUSH EBX	
004271D1	56	PUSH ESI	
004271D2	57	PUSH EDI	
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
004271D6	FF15 DC0A4600	CALL NEAR DWORD PTR DS:[460ADC]	kernel32.GetVersion
004271DC	00D2	ADD DL,DL	
004271DE	8AD4	MOV DL,AH	
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
004271E6	8BC8	MOV ECX,EAX	
004271E8	81E1 FF000000	AND ECX,0FF	
004271EE	89D0 30E64500	MOV DWORD PTR DS:[45E630],ECX	
004271F4	C1E1 08	SHL ECX,8	
004271F7	03CA	ADD ECX,EAX	
004271F9	89D0 2CE64500	MOV DWORD PTR DS:[45E62C],ECX	
004271FF	C1E8 10	SHR EAX,10	
00427202	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX	
00427207	E8 6C8FFDFF	CALL 00400178	pepe_.00400178
0042720C	85C0	TEST EAX,EAX	
0042720E	75 0A	JNZ SHORT 0042721A	pepe_.0042721A
00427210	6A 1C	PUSH 1C	
00427212	E8 678FFDFF	CALL 0040017E	pepe_.0040017E
00427217	83C4 04	ADD ESP,4	
0042721A	E8 658FFDFF	CALL 00400184	pepe_.00400184
0042721F	85C0	TEST EAX,EAX	
00427221	75 0A	JNZ SHORT 0042722D	pepe_.0042722D
00427223	6A 10	PUSH 10	
00427225	E8 608FFDFF	CALL 0040018A	pepe_.0040018A
0042722A	83C4 04	ADD ESP,4	
00400178=pepe_.00400178			

这里存在 AntiDump,我们会发现入口点下面存在一些 CALL,这些 CALL 的地址属于 PE 头,而不是代码段。

我们单击鼠标右键选择 Search for All intermodular calls。查看所有的 API 函数的调用处。

Address	Disassembly	Destination
00401027	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
004010C9	CALL NEAR DWORD PTR DS:[460930]	GDI32.DeleteObject
00401112	CALL NEAR DWORD PTR DS:[460B9C]	kernel32.GetModuleHandleA
00401154	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
004011E8	CALL NEAR DWORD PTR DS:[460B9C]	kernel32.GetModuleHandleA
0040122A	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
0040123E	CALL NEAR DWORD PTR DS:[460E78]	USER32.SendMessageA
004012D2	CALL NEAR DWORD PTR DS:[460E64]	USER32.GetSystemMetrics
004012D6	CALL NEAR DWORD PTR DS:[460E64]	USER32.GetSystemMetrics
004012F0	CALL NEAR DWORD PTR DS:[460E68]	USER32.InflateRect
00401309	CALL NEAR DWORD PTR DS:[460E6C]	USER32.OffsetRect
00401328	CALL NEAR DWORD PTR DS:[460E68]	USER32.InflateRect
00401364	CALL NEAR DWORD PTR DS:[460E70]	USER32.DrawFrameControl
0040139B	CALL NEAR DWORD PTR DS:[460E68]	USER32.InflateRect
00401485	CALL NEAR DWORD PTR DS:[460928]	GDI32.GetTextExtentPointA
004014D1	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
0040151C	CALL NEAR DWORD PTR DS:[460E74]	USER32.DrawStateA
004015E3	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
004015F7	CALL NEAR DWORD PTR DS:[460E78]	USER32.SendMessageA
00401622	CALL NEAR DWORD PTR DS:[460E68]	USER32.EnableWindow
0040162E	CALL NEAR DWORD PTR DS:[460E6C]	USER32.EnableWindow
00401678	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401912	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401988	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401A09	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401B68	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401BFF	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401CA0	CALL NEAR DWORD PTR DS:[460930]	GDI32.DeleteObject
00401CE9	CALL NEAR DWORD PTR DS:[460B9C]	kernel32.GetModuleHandleA
00401D2B	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
00401D09	CALL NEAR DWORD PTR DS:[460B9C]	kernel32.GetModuleHandleA
00401DFF	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
00401E16	CALL NEAR DWORD PTR DS:[460E78]	USER32.SendMessageA
00401E68	CALL NEAR DWORD PTR DS:[460E7C]	USER32.LoadImageA
00401E7C	CALL NEAR DWORD PTR DS:[460E78]	USER32.SendMessageA
00401E85	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00401F37	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
00402015	CALL NEAR DWORD PTR DS:[460B98]	kernel32.InterlockedIncrement
0040202B	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
0040211A	CALL NEAR DWORD PTR DS:[460E78]	USER32.SendMessageA
00402130	CALL NEAR DWORD PTR DS:[460E78]	USER32.PostMessageA
0040215A	CALL NEAR DWORD PTR DS:[460EFC]	ole32.CreateILockBytesOnHGlobal
00402171	CALL NEAR DWORD PTR DS:[460F00]	ole32.StgCreateDocfileOnILockBytes
00402184	CALL NEAR DWORD PTR DS:[460E80]	USER32.GetSysColor
0040236C	CALL NEAR DWORD PTR DS:[460E54]	USER32.GetDlgCtrlID
00402A71	CALL NEAR DWORD PTR DS:[460B90]	kernel32.LocalFree
00402FEE	CALL NEAR DWORD PTR DS:[460B98]	kernel32.LocalAlloc
00402FFC	CALL NEAR DWORD PTR DS:[460B8C]	kernel32.LocalLock
0040300C	CALL NEAR DWORD PTR DS:[460B90]	kernel32.LocalFree
00403033	CALL NEAR DWORD PTR DS:[460B84]	kernel32.LocalUnlock
00403040	CALL NEAR DWORD PTR DS:[460B90]	kernel32.LocalFree
004032EF	CALL NEAR DWORD PTR DS:[460954]	GDI32.GetObjectA
00403374	CALL NEAR DWORD PTR DS:[460954]	GDI32.StartDocA
0040337F	CALL NEAR DWORD PTR DS:[460958]	GDI32.StartPage
00403391	CALL NEAR DWORD PTR DS:[46095C]	GDI32.EndPage
00403398	CALL NEAR DWORD PTR DS:[460960]	GDI32.EndDoc
004033B2	CALL NEAR DWORD PTR DS:[460974]	kernel32.lstrcpynA
004034EF	CALL NEAR DWORD PTR DS:[460E50]	USER32.IntersectRect
004035D4	CALL NEAR DWORD PTR DS:[460E4C]	USER32.IsRectEmpty
004035E7	CALL NEAR DWORD PTR DS:[460950]	GDI32.GetDeviceCaps
00403622	CALL NEAR DWORD PTR DS:[460944]	GDI32.Escape

我们定位到无效的 CALL。

0042723E	CALL	NEAR	DWORD	PTR	DS:[460984]	kernel32.GetCommandLineA
004272D5	CALL	NEAR	DWORD	PTR	DS:[460980]	kernel32.GetStartupInfoA
004272F6	CALL	NEAR	DWORD	PTR	DS:[460B9C]	kernel32.GetModuleHandleA
00427458	CALL	00400236				UnPackMe.004001D8
004274F7	CALL	0040023C				ntdll.RtlUnwind
004277F3	CALL	0040027D				ntdll.RtlUnwind
00427929	CALL	NEAR	DWORD	PTR	DS:[46098C]	kernel32.RaiseException
00427E07	CALL	NEAR	DWORD	PTR	DS:[460A84]	kernel32.GetCurrentProcess
00427E0E	CALL	NEAR	DWORD	PTR	DS:[46F452]	kernel32.TerminateProcess
00427E98	CALL	NEAR	DWORD	PTR	DS:[460990]	kernel32.ExitProcess
00428029	CALL	NEAR	DWORD	PTR	DS:[4609F8]	ntdll.RtlReAllocateHeap
00428042	CALL	004003ED				UnPackMe.004001D8
004280B8	CALL	NEAR	DWORD	PTR	DS:[460A08]	kernel32.GetCPInfo
00428805	CALL	00427360				UnPackMe.004001D8
00428823	CALL	NEAR	DWORD	PTR	DS:[460A54]	kernel32.InitializeCriticalSection
0042884D	CALL	NEAR	DWORD	PTR	DS:[460A3C]	ntdll.RtlEnterCriticalSection
0042886E	CALL	NEAR	DWORD	PTR	DS:[460A44]	ntdll.RtlLeaveCriticalSection
004288AE	CALL	NEAR	DWORD	PTR	DS:[460A3C]	ntdll.RtlEnterCriticalSection
004288E0	CALL	NEAR	DWORD	PTR	DS:[460A3C]	ntdll.RtlEnterCriticalSection
0042891E	CALL	NEAR	DWORD	PTR	DS:[460A44]	ntdll.RtlLeaveCriticalSection
00428950	CALL	NEAR	DWORD	PTR	DS:[460A44]	ntdll.RtlLeaveCriticalSection
00428983	CALL	NEAR	DWORD	PTR	DS:[460B98]	kernel32.InterlockedIncrement
00428C15	CALL	NEAR	DWORD	PTR	DS:[460B9C]	kernel32.GetModuleHandleA
00428C25	CALL	NEAR	DWORD	PTR	DS:[460B40]	kernel32.GetProcAddress
004293A9	CALL	NEAR	DWORD	PTR	DS:[4609A0]	kernel32.HeapCreate
004293C8	CALL	NEAR	DWORD	PTR	DS:[46099C]	kernel32.HeapDestroy
00429401	CALL	NEAR	DWORD	PTR	DS:[4609FC]	ntdll.RtlAllocateHeap
0042951F	CALL	NEAR	DWORD	PTR	DS:[4609A4]	kernel32.VirtualFree
00429536	CALL	NEAR	DWORD	PTR	DS:[460A00]	ntdll.RtlFreeHeap
00429560	CALL	NEAR	DWORD	PTR	DS:[4609A4]	kernel32.VirtualFree
00429597	CALL	NEAR	DWORD	PTR	DS:[460A00]	ntdll.RtlFreeHeap
004295E9	CALL	NEAR	DWORD	PTR	DS:[4609A4]	kernel32.VirtualFree
00429892	CALL	NEAR	DWORD	PTR	DS:[4609A8]	kernel32.VirtualAlloc
00429CB0	CALL	NEAR	DWORD	PTR	DS:[460A58]	kernel32.GetFullPathNameA
00429CCF	CALL	NEAR	DWORD	PTR	DS:[460A20]	kernel32.GetCurrentDirectoryA
00429DA8	CALL	NEAR	DWORD	PTR	DS:[4609AC]	kernel32.GetDriveTypeA
00429DDE	CALL	00427360				UnPackMe.004001D8
00429E2A	CALL	NEAR	DWORD	PTR	DS:[460980]	kernel32.GetStartupInfoA
00429F00	CALL	NEAR	DWORD	PTR	DS:[46097C]	kernel32.GetFileType
00429F79	CALL	NEAR	DWORD	PTR	DS:[46097C]	kernel32.GetFileType
00429FBE	CALL	NEAR	DWORD	PTR	DS:[460B80]	kernel32.SetHandleCount
0042A1F6	CALL	NEAR	DWORD	PTR	DS:[460A50]	kernel32.TlsAlloc
0042A21F	CALL	NEAR	DWORD	PTR	DS:[460A38]	kernel32.TlsSetValue
0042A232	CALL	NEAR	DWORD	PTR	DS:[460ACC]	kernel32.GetCurrentThreadId
0042A272	CALL	NEAR	DWORD	PTR	DS:[460B5C]	ntdll.RtlGetLastWin32Error
0042A280	CALL	NEAR	DWORD	PTR	DS:[460A30]	kernel32.TlsGetValue
0042A2A6	CALL	NEAR	DWORD	PTR	DS:[460A38]	kernel32.TlsSetValue
0042A2B9	CALL	NEAR	DWORD	PTR	DS:[460ACC]	kernel32.GetCurrentThreadId
0042A2C9	CALL	NEAR	DWORD	PTR	DS:[460B60]	ntdll.RtlSetLastWin32Error
0042A2D6	CALL	00427360				UnPackMe.004001D8
0042A2DF	CALL	NEAR	DWORD	PTR	DS:[460B60]	ntdll.RtlSetLastWin32Error
0042AF92	CALL	NEAR	DWORD	PTR	DS:[460A6C]	kernel32.CloseHandle

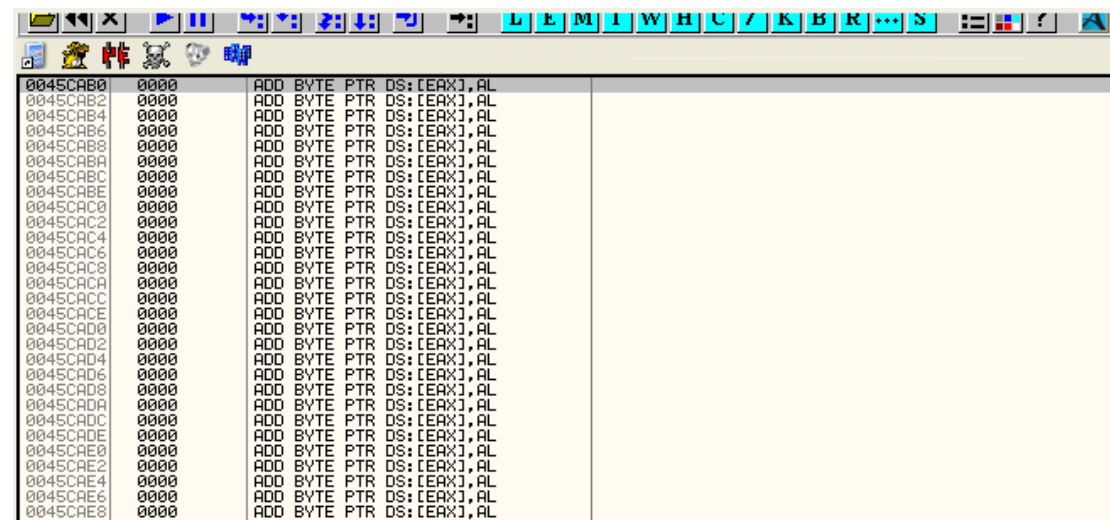
我们可以看到这些是需要我们修复的。

但愿不需要我们编写第二个脚本来进行修复。

现在我们需要在刚刚 dump 并修复了 IAT 的文件中定位 1000(十六进制)个字节的空间用于存储 PE 头中有用的信息。

我们用 OD 加载刚刚 dump 并修复了 IAT 的那个文件,随便定位一个 1000(十六进制)字节的区域,这里我找的区域起始地址为

45CAB0。



0045CAB0	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAB2	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAB4	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAB6	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAB8	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CABA	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CABC	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CABE	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAC0	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAC2	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAC4	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAC6	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAC8	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CACA	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CACB	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CACE	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAD0	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAD2	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAD4	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAD6	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAD8	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CADA	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CADC	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CADE	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAE0	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAE2	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAE4	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAE6	0000	ADD	BYTE	PTR	DS:[EAX], AL
0045CAE8	0000	ADD	BYTE	PTR	DS:[EAX], AL

这里我们需要验证一下这块虚拟地址空间是否在文件中存在,我们单击鼠标右键选择 View-Executable file:

0005CAB0	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAB2	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAB4	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAB6	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAB8	0000	ADD BYTE PTR DS:[EAX],AL	
0005CABA	0000	ADD BYTE PTR DS:[EAX],AL	
0005CABC	0000	ADD BYTE PTR DS:[EAX],AL	
0005CABE	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAC0	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAC2	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAC4	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAC6	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAC8	0000	ADD BYTE PTR DS:[EAX],AL	
0005CACA	0000	ADD BYTE PTR DS:[EAX],AL	
0005CACC	0000	ADD BYTE PTR DS:[EAX],AL	
0005CACE	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAD0	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAD2	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAD4	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAD6	0000	ADD BYTE PTR DS:[EAX],AL	
0005CAD8	0000	ADD BYTE PTR DS:[EAX],AL	
0005CADA	0000	ADD BYTE PTR DS:[EAX],AL	
0005CADC	0000	ADD BYTE PTR DS:[EAX],AL	

我们可以看到这块虚拟地址空间在文件中的确存在,现在我们需要从之前那个断在 OEP 处的 OD 中拷贝 PE 头中的 1000(16 进制)个字节出来。

File View Debug Plugins Options Window Help

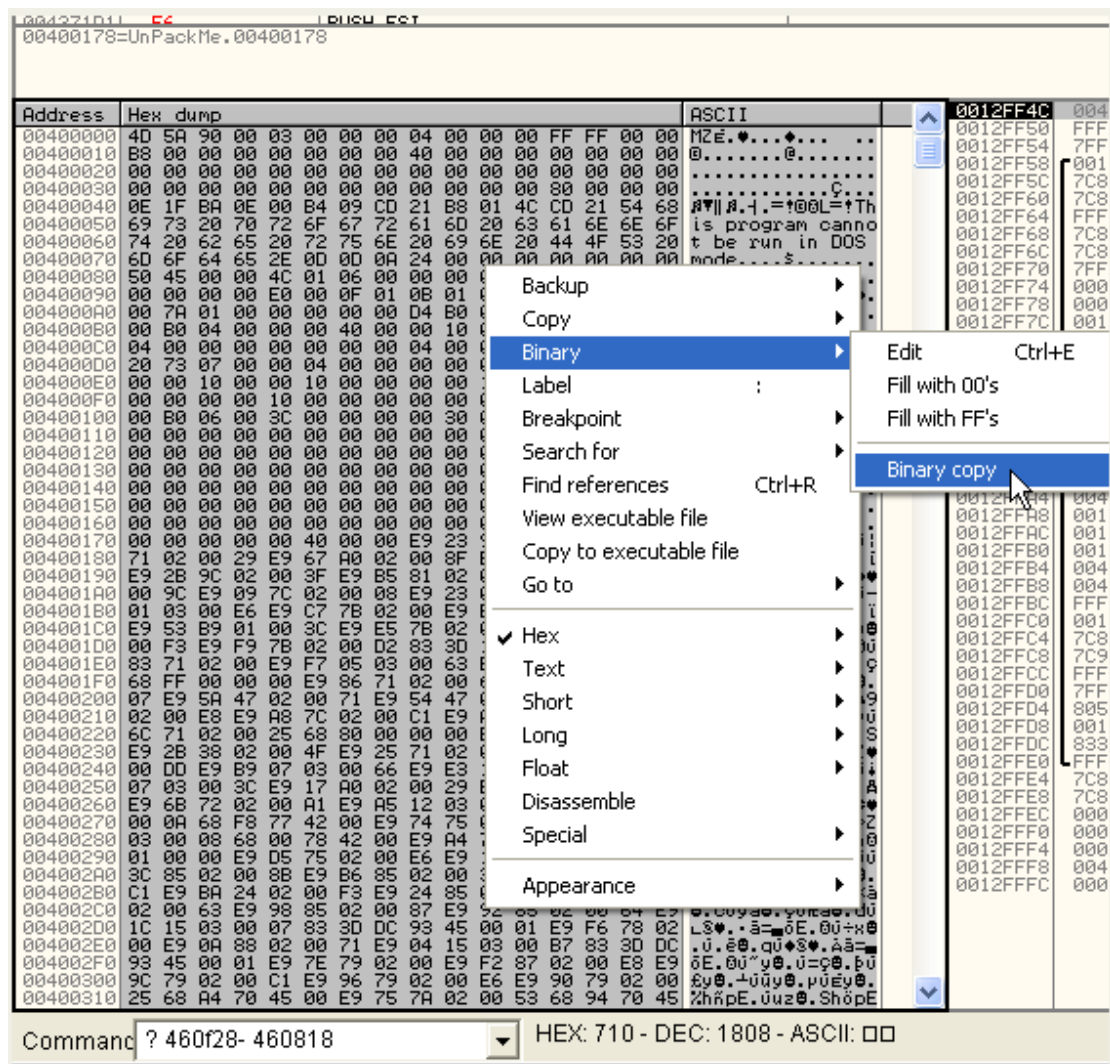
L E M T W H C / K B R

004271B0	55	PUSH EBP	
004271B1	8BEC	MOV EBP,ESP	
004271B3	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH 450E60	
004271BA	68 C8924200	PUSH 4292C8	
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
004271C5	50	PUSH EAX	
004271C6	64:8925 00000000	MOV DWORD PTR FS:[0],ESP	
004271CD	83C4 A8	ADD ESP,-58	
004271D0	53	PUSH EBX	
004271D1	56	PUSH ESI	
004271D2	57	PUSH EDI	
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
004271D6	FF15 DC0A4600	CALL NEAR DWORD PTR DS:[460ADC]	kernel32.GetVersio
004271DC	00D2	ADD DL,DL	
004271DE	8AD4	MOV DL,AH	
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
004271E6	8BC8	MOV ECX,EAX	
004271E8	81E1 FF000000	AND ECX,0FF	
004271EE	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX	
004271F4	C1E1 08	SHL ECX,8	
004271F7	03CA	ADD ECX,EDX	
004271F9	890D 2CE64500	MOV DWORD PTR DS:[45E62C],ECX	
004271FF	C1E8 10	SHR EAX,10	
00427202	A3 28E64500	MOV DWORD PTR DS:[45E628],EAX	
00427207	E8 6C8FFDFF	CALL 00400178	UnPackMe.00400178
0042720C	85C0	TEST EAX,EAX	
0042720E	75 0A	JNZ SHORT 0042721A	UnPackMe.0042721A
00427210	6A 1C	PUSH 1C	
00427212	E8 678EFDFF	CALL 0040017E	UnPackMe.0040017E

00400178=UnPackMe.00400178

Address	Hex dump	ASCII
00400000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ E.
00400010	00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....@.....
00400020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00C.....
00400030	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00@.....
00400040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	PE\Signature=Th
00400050	69 73 00 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00400060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00400070	6D 6F 64 65 2E 0D 0D 0A 24 0D 00 00 00 00 00 00	mode....\$.....
00400080	50 45 00 00 4C 01 06 00 00 00 00 00 00 00 00 00	PE..LO.....
00400090	00 00 00 00 E0 00 0F 01 08 01 00 00 00 92 04 000.*000.E.
004000A0	00 7A 01 00 00 00 00 00 04 B0 06 00 00 10 00 00	.z0.....E...>
004000B0	00 00 04 00 00 00 40 00 00 10 00 00 00 02 00 00	...@.....@...
004000C0	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	...@.....@...
004000D0	20 73 07 00 00 04 00 00 00 00 00 00 02 00 00 00	s.....@.....
004000E0	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	...>...>...>
004000F0	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00<.....0x..
00400100	00 B0 06 00 3C 00 00 00 00 30 06 00 94 78 00 00<.....0x..
00400110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

选中整个区段。



二进制复制并粘贴到 dump 文件所在 OD 的 45CAB0 为起始地址,长度为 1000(16 进制)字节的区域中。并且将修改保存到文件。

现在 PE 头的数据我们是拷贝过去了,下面我需要在程序开始运行起来,起始地址为 45CAB0,长度为 1000 这部分数据覆盖到 PE 头中以此来修复 AntiDump。但是现在有个问题呀,PE 头具有可写权限吗?PE 头并不一定具有可写的权限,那么我就需要给予其赋予可写权限,我们通常会调用 VirtualProtect 这个 API 函数来修改内存的访问权限。但是该程序的 IAT 中并没有 VirtualProtect 这个 API 函数对应的项。

怎么办呢?通常为了兼容性考虑会使用 LoadLibrary 和 GetProcAddress 这两个 API 函数来配合获取 VirtualProtect 的地址。但是这里我会告诉你一个更加快捷的方式。

7C801D77	8BFF	MOV EDI,EDI	
7C801D79	55	PUSH EBP	
7C801D7A	8BEC	MOV EBP,ESP	
7C801D7C	837D 08 00	CMPL DWORD PTR SS:[EBP+8],0	
7C801D80	53	PUSH EBX	
7C801D81	56	PUSH ESI	
7C801D82	74 14	JE SHORT 7C801D98	kernel32.7C801D98
7C801D84	68 F0E2007C	PUSH 7C80E2F0	ASCII "twain_32.dll"
7C801D89	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C801D8C	FF15 9C13007C	CALL NEAR DWORD PTR DS:[7C80139C]	ntdll._stricmp
7C801D92	85C0	TEST EAX,EAX	
7C801D94	59	POP EAX	
7C801D95	59	POP ECX	
7C801D96	74 12	JE SHORT 7C801DAA	kernel32.7C801DAA
7C801D98	6A 00	PUSH 0	
7C801D9A	6A 00	PUSH 0	
7C801D9C	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C801D9F	E8 ABFFFFFF	CALL 7C801D4F	kernel32.LoadLibraryExA
7C801DA4	5E	POP ESI	
7C801DA5	5B	POP EBX	
7C801DA6	5D	POP EBP	
7C801DA7	C2 0400	RETN 4	
7C801DAA	64:A1 10000000	MOV EAX,DWORD PTR FS:[10]	
7C801DB0	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	
7C801DB3	BE 04010000	MOV ESI,104	
7C801DB8	56	PUSH ESI	
7C801DB9	FF35 D436087C	PUSH DWORD PTR DS:[7C8036D4]	
7C801DBF	FF70 18	PUSH DWORD PTR DS:[EAX+18]	
7C801DC2	FF15 0C10007C	CALL NEAR DWORD PTR DS:[7C80100C]	ntdll.RtlAllocateHeap
7C801DC8	8B08	MOV EBX,EAX	
7C801DCA	85DB	TEST EBX,EBX	
7C801DCC	74 CA	JE SHORT 7C801D98	kernel32.7C801D98
7C801DCE	57	PUSH EDI	
7C801DCF	56	PUSH ESI	
7C801DD0	53	PUSH EBX	
7C801DD1	E8 650B0200	CALL 7C82293B	kernel32.GetWindowsDirectoryA
7C801DD6	8BFB	MOV EDI,EBX	
7C801DD8	4F	DEC EDI	

7C801AD0	E9 BBE59283	JMP 00130090	
7C801AD5	FF75 14	PUSH DWORD PTR SS:[EBP+14]	
7C801AD8	FF75 10	PUSH DWORD PTR SS:[EBP+10]	
7C801ADB	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
7C801ADE	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C801AE1	6A FF	PUSH -1	
7C801AE3	E8 75FFFFFF	CALL 7C801A5D	kernel32.VirtualProtect
7C801AE8	5D	POP EBP	
7C801AE9	C2 1000	RETN 10	
7C801AEC	90	NOP	

假设 LoadLibraryA 是 IAT 中的一项,这里 LoadLibraryA 的实现代码我已经用绿色高亮显示了。其实几乎所有的 XP 系统上 LoadLibraryA 与 VirtualProtect 的入口地址的差是一个定值。我们只需要通过 LoadLibraryA 的首地址减去这个定值就可以定位到 VirtualProtect 的首地址。

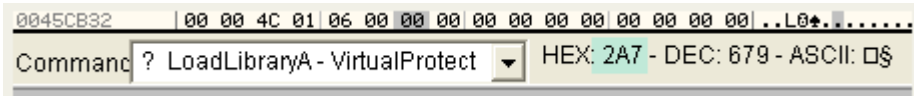
0045CA06	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA08	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA0A	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA0C	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA0E	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA10	60	PUSHAD	
0045CA11	90	NOP	
0045CA12	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA14	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA16	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA18	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA1A	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA1C	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA1E	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA20	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA22	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA24	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA26	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA28	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA2A	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA2C	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA2E	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA30	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA32	0000	ADD BYTE PTR DS:[EAX],AL	

Address	Hex dump	ASCII
00460B3C	<&kernel32.LoadLibraryA>	77 1D 80 7C 28 AC 80 7C 66 AA 80 7C A9 2C 81 7C
00460B4C	<&kernel32.WriteConsoleA>	ED CB 81 7C 3D 0D 87 7C 19 90 83 7C 59 35 81 7C
00460B5C	<&kernel32.GetLastError>	31 03 92 7C 40 03 92 7C 07 EF 80 7C 2D FF 80 7C
00460B6C	<&kernel32.GlobalFree>	2F FE 80 7C 51 28 81 7C 11 03 81 7C B1 C7 80 7C
00460B7C	<&kernel32.LoadResource>	65 00 80 7C CF C6 80 7C 21 2F 82 7C B0 99 80 7C

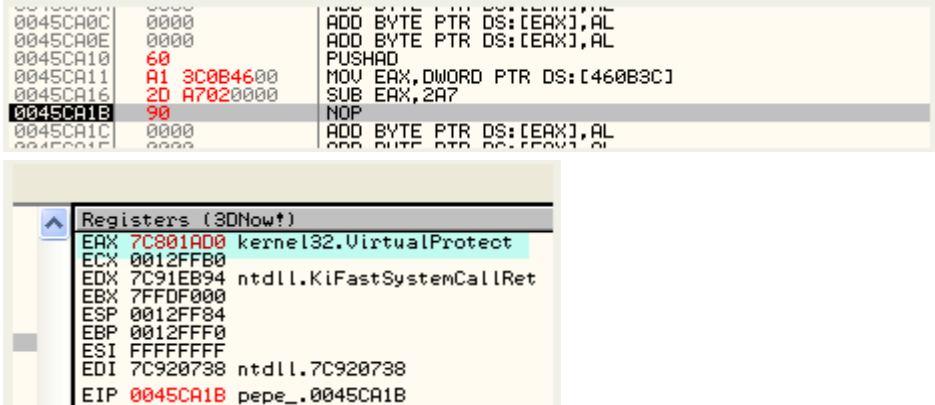
这里我们定位到 IAT 中的 LoadLibrary 这一项。

0045CA06	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA08	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA10	60	PUSHAD	
0045CA11	A1 3C0B4600	MOV EAX,DWORD PTR DS:[460B3C]	
0045CA16	2D A7020000	SUB EAX,2A7	
0045CA1B	90	NOP	
0045CA1C	0000	ADD BYTE PTR DS:[EAX],AL	
0045CA1E	0000	ADD BYTE PTR DS:[EAX],AL	

这里首先将 LoadLibraryA 的地址存放到 EAX 中。下面我们来计算一下 LoadLibraryA 与 VirtualProtect 这两个函数入口地址之间的差值。



这里我们可以看到 LoadLibraryA 与 VirtualProtect 这两个函数入口地址之间的差值为 2A7。



获取到 VirtualProtect 的入口地址以后,我们就可以对 PE 头赋予写入权限了。

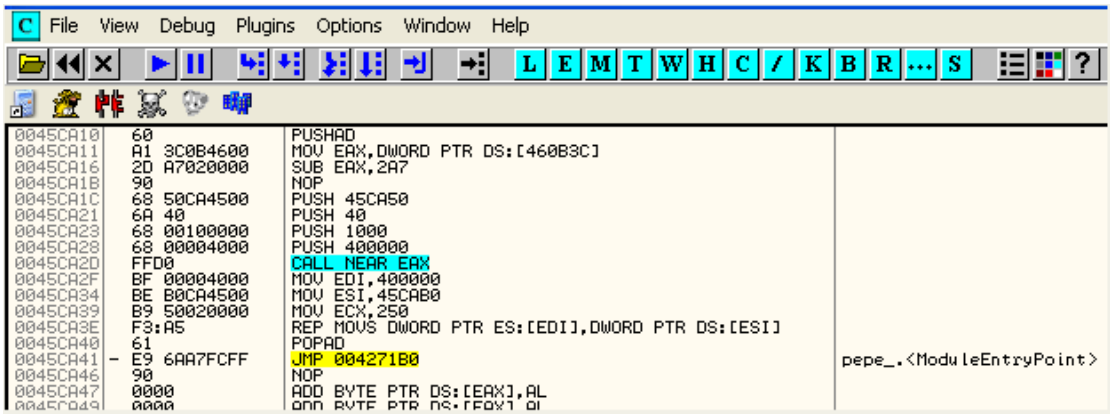
VirtualProtect

Quick Info

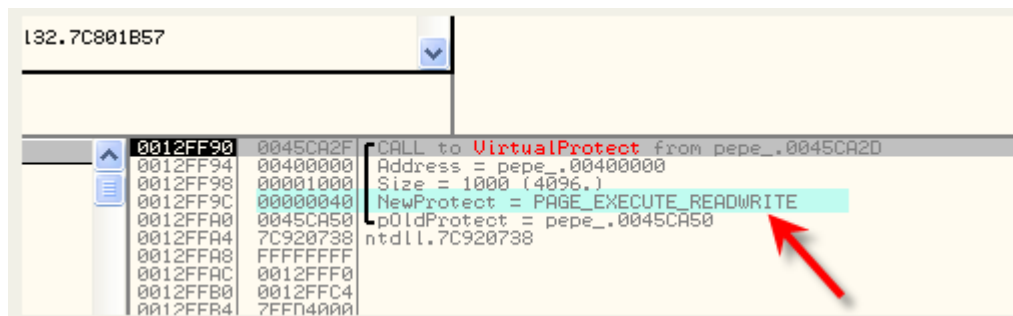
The **VirtualProtect** function changes the access protection on a region of committed pages in the virtual address space of the calling process. This function differs from **VirtualProtectEx**, which changes the access protection of any process.

BOOL VirtualProtect(
LPVOID lpAddress, // address of region of committed pages
DWORD dwSize, // size of the region
DWORD flNewProtect, // desired access protection
PDWORD lpfOldProtect // address of variable to get old protection
);

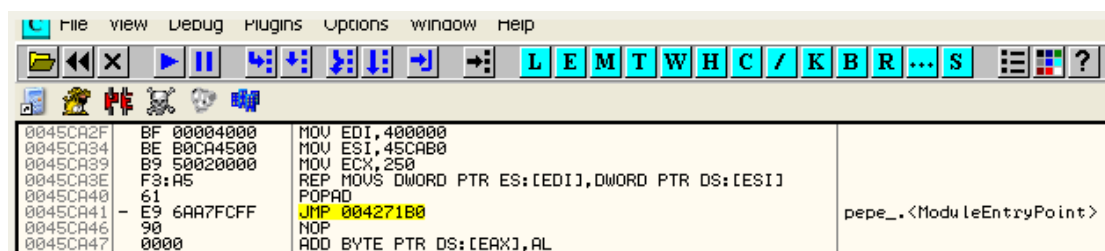
这里是 MSDN 中对于 VirtualProtect 这个函数的说明。



我们首先需要一个 PUSHAD 指令保存寄存器环境,接着将 LoadLibrary 函数的入口地址保存到 EAX 中,接着将其减去 2A7 就得到了 VirtualProtect 这个函数的入口地址。接着调用 VirtualProtect 给 PE 头赋予写入权限。该函数的参数如下:



首先是要修改的访问权限的起始地址为 400000,长度为 1000,新的访问权限为可读可写可执行,旧的访问权限保存到 45CA50 所指向的内存单元中。接着将起始地址为 45CA00,长度为 1000 的数据拷贝到起始地址为 400000,长度为 1000 的内存单元(也就是 PE 头)中。接着调用 POPAD 指令还原寄存器环境,然后跳往 OEP 处。



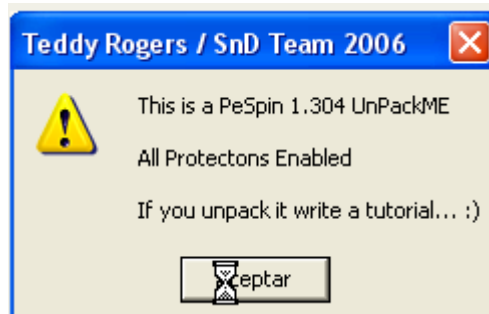
现在要做的事情就是将入口点修改为 45CA10 了。

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	00710200	DD 00027100	AddressOfEntryPoint = 27100
004000AC	00100000	DD 00001000	BaseOfCode = 1000
004000B0	00B00400	DD 0004B000	BaseOfData = 4B000
004000B4	00004000	DD 00004000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000
004000BC	00020000	DD 00002000	FileAlignment = 200

这里我们将 AddressOfEntryPoint 修改为 5CA10 即可(RVA)。

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	10CA0500	DD 0005CA10	AddressOfEntryPoint = 5CA10
004000AC	00100000	DD 00001000	BaseOfCode = 1000
004000B0	00B00400	DD 0004B000	BaseOfData = 4B000
004000B4	00004000	DD 00004000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000
004000BC	00020000	DD 00002000	FileAlignment = 200

保存修改到文件,直接运行起来。



嘿嘿,搞定。