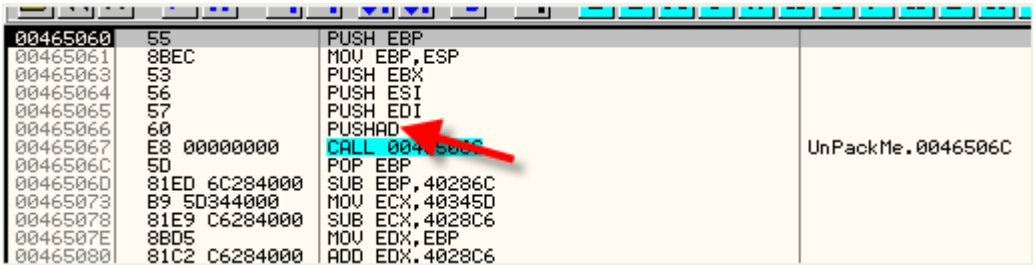
第三十八章-手脱 Yoda's Protector v1.3(Yoda's Crypter)

上一章节给大家介绍了 IAT 重定向以及修复方法。本章我们稍微增加一点难度。

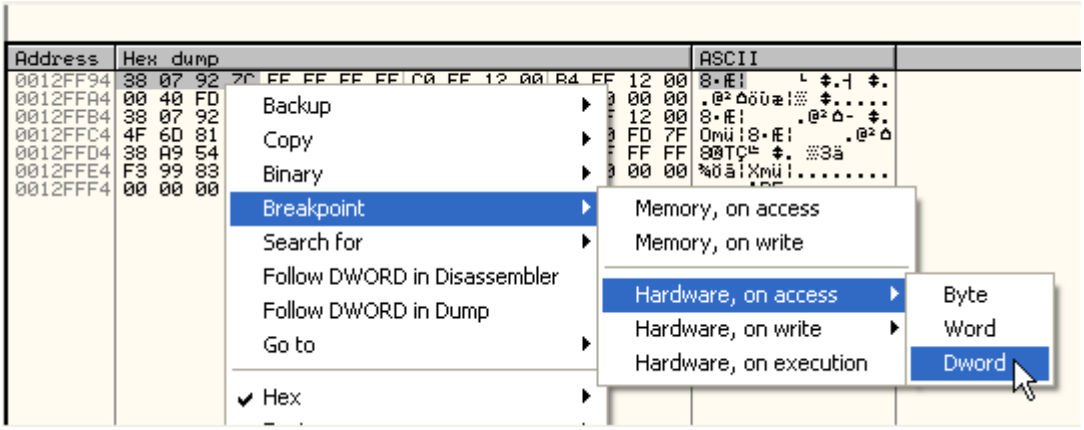
我们一起来脱 Yoda Crypter v1.3,首先大家配置好 OD 的反反调试插件,加载它。

断在了入口点处。

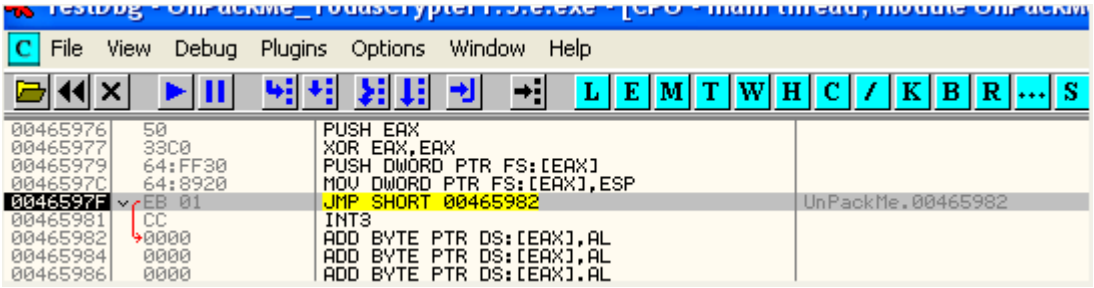


我们可以看到这里有个 PUSHAD 指令,嘿嘿,我们可以使用 ESP 定律来定位 OEP,我们单步到 PUSHAD 指令处,按 F7 键单步执行 PUSHAD 指令。

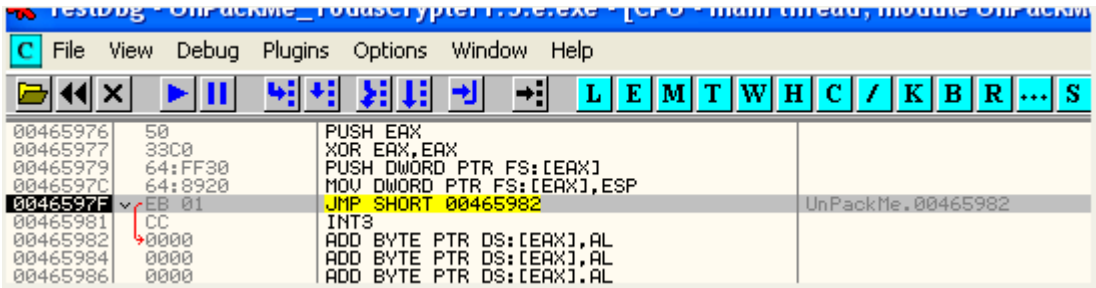
然后在 ESP 寄存器的值上面单击鼠标右键选择-Follow in Dump,这样就能在数据窗口中定位到刚刚保存的寄存器环境了,我们选中前 4 个字节,单击鼠标右键选择-Breakpoint-Hardware,on access-Dword,对其设置硬件访问断点。



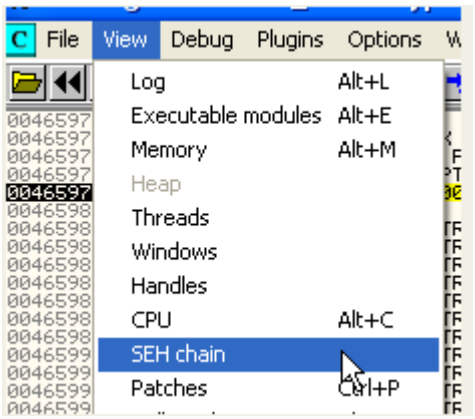
运行起来。



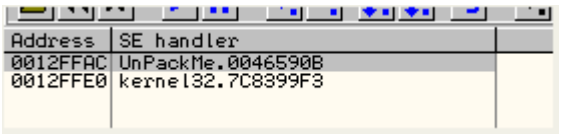
断在了这里,我们可以看到这里将给 SEH 链添加一个新的节点,接着通过 JMP 指令跳转到下面引发一个异常,我们跟到 JMP 指令处。



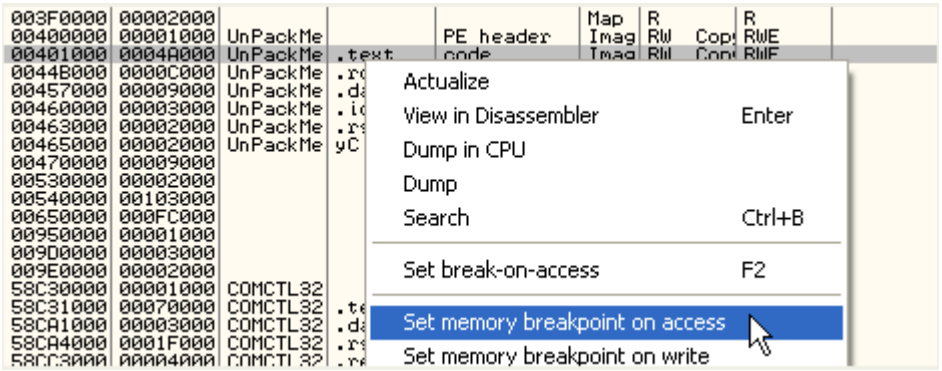
这里将跳往下面,将产生异常。我们来看看异常处理程序在哪里。



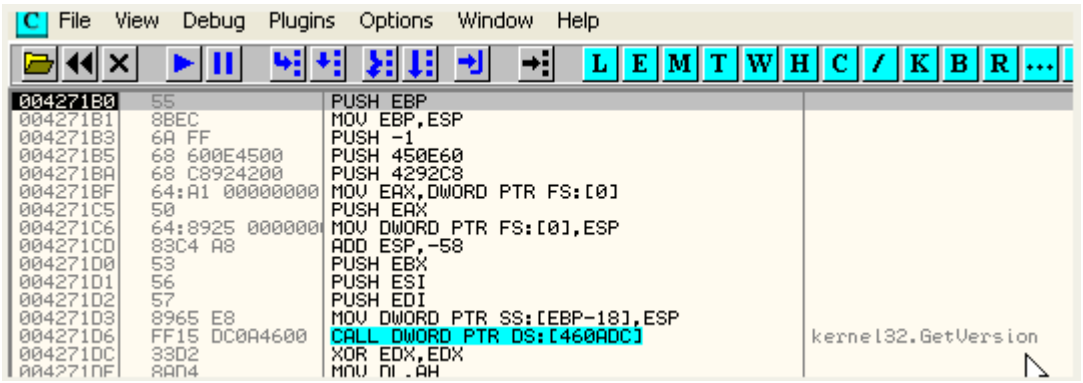
选择主菜单项中的 View-SEH chain。



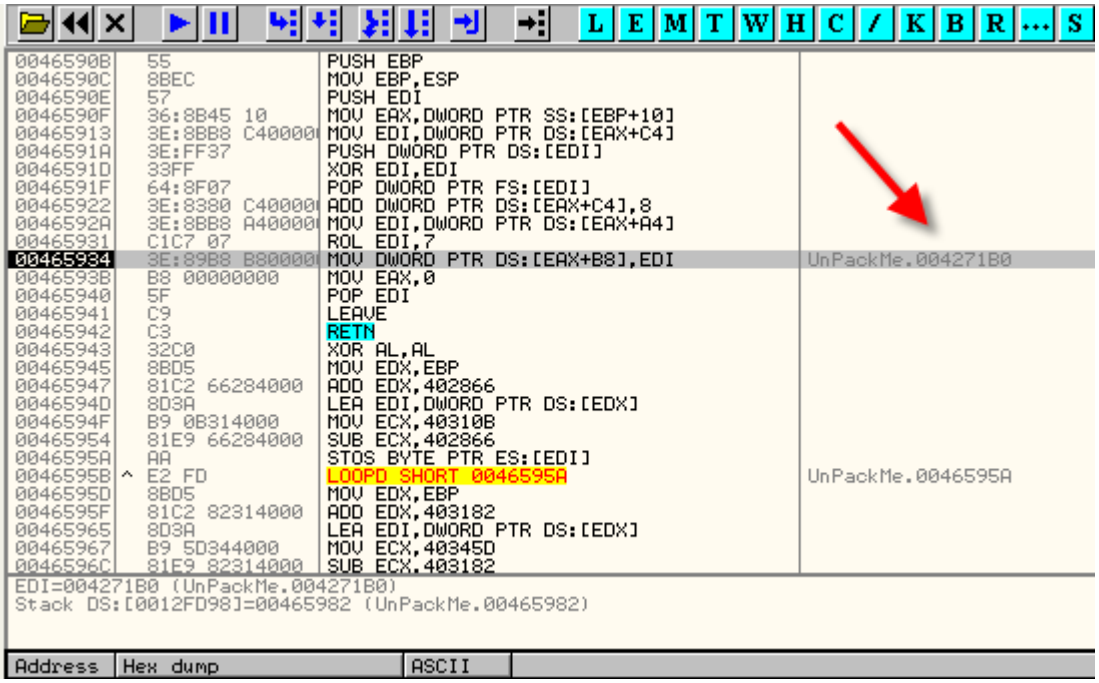
我们可以看到刚刚安装的异常处理程序入口地址为 46590B,不出意外的话到达该异常处理程序,随后就会到达 OEP,我们给第一个区段设置内存访问断点,触发异常后,随即我们就会到达 OEP 处。



直接运行起来。

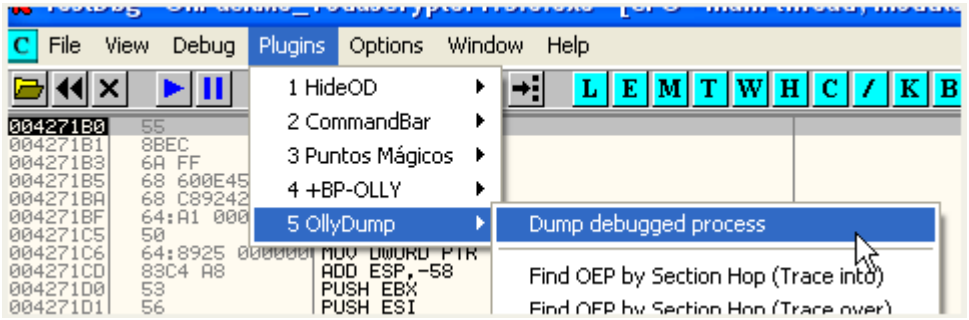


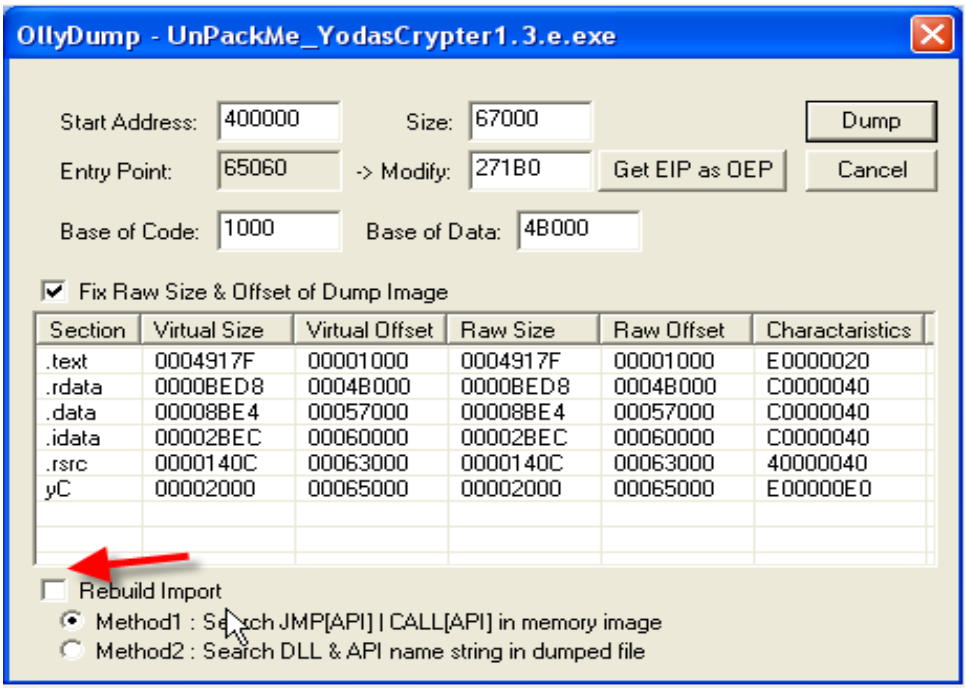
断在了 OEP 处,这里 OEP 为 4271B0,大家应该很熟悉了吧,我们加壳的目标程序都是同一个 CrackMe,嘿嘿。



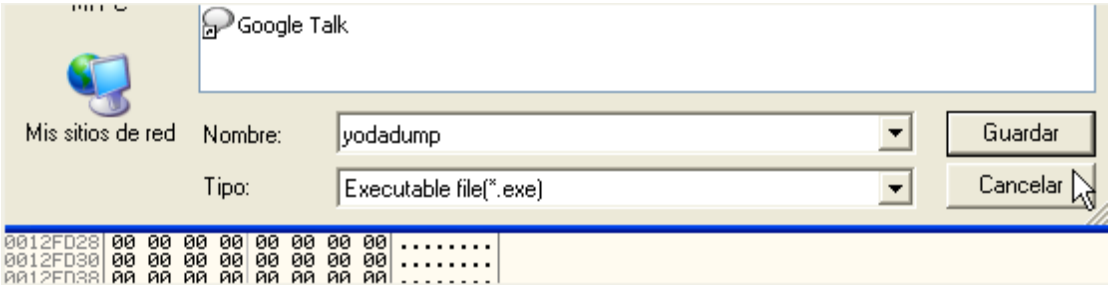
如果要深究的话,大家会发现异常处理函数中 Context 结构的 EIP 寄存器的值由 465982 修改为了 4271B0,即 OEP,关于 CONTEXT 结构体的详细介绍我们后面章节再说。

好,现在我们到了 OEP 处,可以 dump 了。



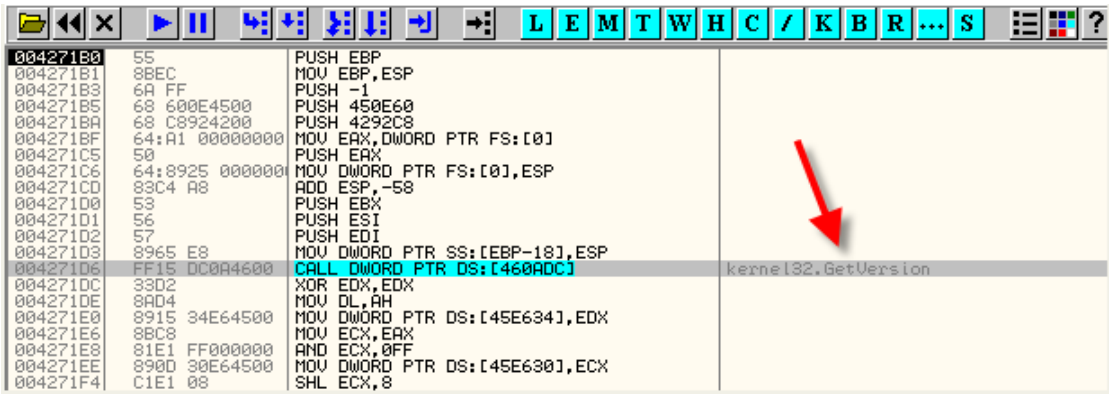


这里我们不勾选 Rebuild Import 选项。

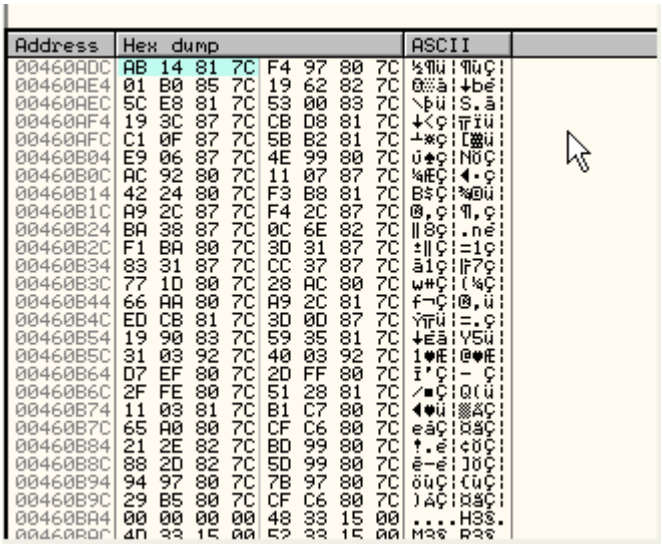


这里我们将其重命名为 yodadump.exe,我们会发现程序无法正常运行。我们需要修复 IAT。

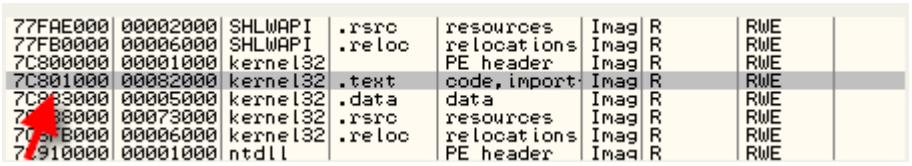
下面我们来分析一下 IAT,随便找一个 API 函数的调用处,这里我们还是跟之前一样拿 CALL GetVersion 开刀,嘿嘿。



我们定位到 4271D6 处 CALL DWORD PTR DS:[460ADC]指令,460ADC 是 IAT 中的一项,我们在数据窗口中定位到这个地址。

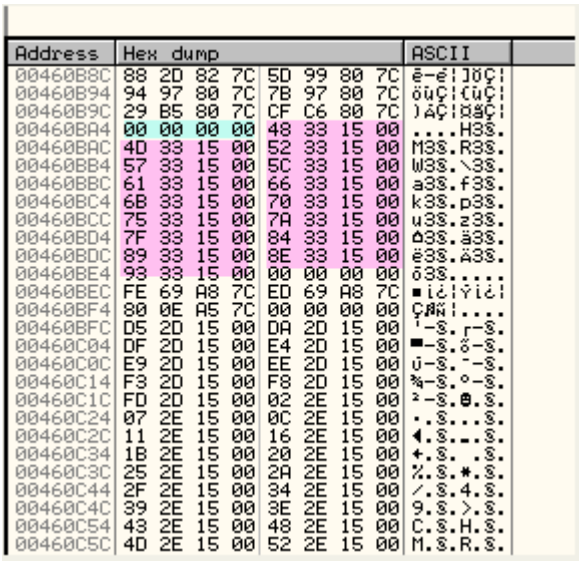


这部分 IAT 项看起来像是正常的,我们打开区段列表窗口,看看这部分 IAT 项属于哪个系统 DLL,这里我们可以看到这部分 IAT 项命中在 Kernel32.dll 的代码段,即这部分 IAT 项是正确的。



再随便选一项单击鼠标右键选择-Find references(查看一下参考引用),这里我选择 460B34 这一项。

继续往下:

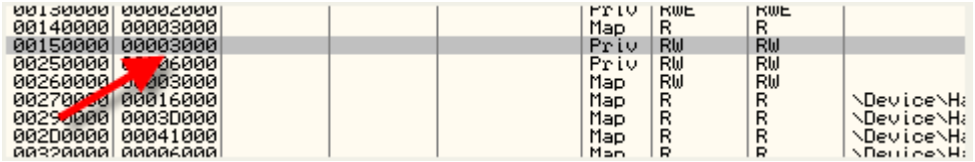


Address	Hex dump	ASCII
00460B8C	88 2D 82 7C 5D 99 80 7C	ê-é¦]öÇ¦
00460B94	94 97 80 7C 7B 97 80 7C	öùÇ¦{ùÇ¦
00460B9C	29 B5 80 7C CF C6 80 7C	)µÇ¦ΦãÇ¦
00460BA4	00 00 00 00 48 33 15 00	....H3§.
00460BAC	4D 33 15 00 52 33 15 00	M3§.R3§.
00460BB4	57 33 15 00 5C 33 15 00	W3§.\3§.
00460BBC	61 33 15 00 66 33 15 00	a3§.f3§.
00460BC4	6B 33 15 00 70 33 15 00	k3§.p3§.
00460BCC	75 33 15 00 7A 33 15 00	u3§.z3§.
00460BD4	7F 33 15 00 84 33 15 00	⌂3§.ä3§.
00460BDC	89 33 15 00 8E 33 15 00	ë3§.Ä3§.
00460BE4	93 33 15 00 00 00 00 00	ô3§.....
00460BEC	FE 69 A8 7C ED 69 A8 7C	■i¿¦Ýi¿¦
00460BF4	80 0E A5 7C 00 00 00 00	Ç♫Ñ¦....
00460BFC	D5 2D 15 00 DA 2D 15 00	╒-§.┌-§.
00460C04	DF 2D 15 00 E4 2D 15 00	▀-§.σ-§.
00460C0C	E9 2D 15 00 EE 2D 15 00	ú-§.⁻-§.
00460C14	F3 2D 15 00 F8 2D 15 00	¾-§.°-§.
00460C1C	FD 2D 15 00 02 2E 15 00	²-§.☻.§.
00460C24	07 2E 15 00 0C 2E 15 00	•.§...§.
00460C2C	11 2E 15 00 16 2E 15 00	◄.§...§.
00460C34	1B 2E 15 00 20 2E 15 00	←.§. .§.
00460C3C	25 2E 15 00 2A 2E 15 00	%.§.*.§.
00460C44	2F 2E 15 00 34 2E 15 00	/.§.4.§.
00460C4C	39 2E 15 00 3E 2E 15 00	9.§.>.§.
00460C54	43 2E 15 00 48 2E 15 00	C.§.H.§.
00460C5C	4D 2E 15 00 52 2E 15 00	M.§.R.§.

这里我们可以看到粉红色标注出来的区域,我们在区段列表窗口中看看这部分属于哪个区段。

00130000	00002000			Priv	RWE	RWE	
00140000	00003000			Map	R	R	
00150000	00029000			Priv	RW	RW	
00250000	00006000			Priv	RW	RW	
00[illegible]000	00003000			Map	RW	RW	

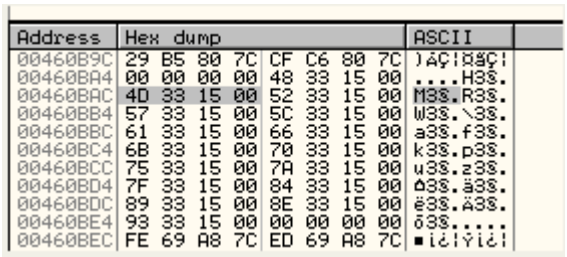
我们可以看到不属于系统 DLL,该区段应该是由壳创建的,我们重启 OD,来验证一下。



00130000	00002000			Priv	RWE	RWE	
00140000	00003000			Map	R	R	
00150000	00003000			Priv	RW	RW	
00250000	[illegible]6000			Priv	RW	RW	
00260000	[illegible]3000			Map	RW	RW	
00270000	00016000			Map	R	R	\Device\H
00290000	0003D000			Map	R	R	\Device\H
002D0000	00041000			Map	R	R	\Device\H
00320000	00006000			Map	R	R	\Device\H

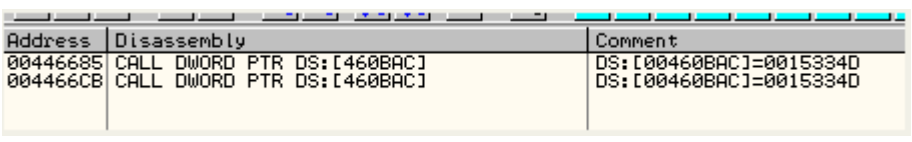
我们可以看到此时起始地址为 150000 的区段,长度为 3000 字节,但是壳执行以后,该区段变成了 29000 字节,即可壳将该区段增大了,并且增大的部分供自己使用。

该区段是用于重定向 IAT 的部分元素的,我们具体来分析一下。



Address	Hex dump	ASCII
00460B9C	29 B5 80 7C CF C6 80 7C	)µÇ¦ΦãÇ¦
00460BA4	00 00 00 00 48 33 15 00	....H3§.
00460BAC	4D 33 15 00 52 33 15 00	M3§.R3§.
00460BB4	57 33 15 00 5C 33 15 00	W3§.\3§.
00460BBC	61 33 15 00 66 33 15 00	a3§.f3§.
00460BC4	6B 33 15 00 70 33 15 00	k3§.p3§.
00460BCC	75 33 15 00 7A 33 15 00	u3§.z3§.
00460BD4	7F 33 15 00 84 33 15 00	⌂3§.ä3§.
00460BDC	89 33 15 00 8E 33 15 00	ë3§.Ä3§.
00460BE4	93 33 15 00 00 00 00 00	ô3§.....
00460BEC	FE 69 A8 7C ED 69 A8 7C	■i¿¦Ýi¿¦

随便找一个举例,这里我们将拿 460BAC 这一项来说,我们来看看其参考引用,选中该项,单击鼠标右键选择-Find references。



Address	Disassembly	Comment
00446685	CALL DWORD PTR DS:[460BAC]	DS:[00460BAC]=0015334D
004466CB	CALL DWORD PTR DS:[460BAC]	DS:[00460BAC]=0015334D

这里我们可以看到有两处参考引用,我们直接在第一条记录上面双击,定位到了这里。

```
00446683   57              PUSH EDI
00446684   53              PUSH EBX
00446685   FF15 AC0B4600   CALL DWORD PTR DS:[460BAC]
0044668B   33F6            XOR ESI,ESI
0044668D   8B3D C00A4600   MOV EDI,DWORD PTR DS:[460AC0]   kernel32.WideCharToMultiByte
00446693   56              PUSH ESI
00446694   56              PUSH ESI
00446695   8BE8            MOV EBP,EAX
00446697   56              PUSH ESI
00446698   56              PUSH ESI
00446699   55              PUSH EBP
0044669A   53              PUSH EBX
0044669B   56              PUSH ESI
0044669C   56              PUSH ESI
0044669D   FFD7            CALL EDI
0044669F   50              PUSH EAX
004466A0   8B4C24 18      MOV ECX,DWORD PTR SS:[ESP+18]
004466A4   894424 1C      MOV DWORD PTR SS:[ESP+1C],EAX
004466A8   E8 B259FFFF    CALL 0043C05F                   UnPackMe.0043C05F
004466AD   56              PUSH ESI
```

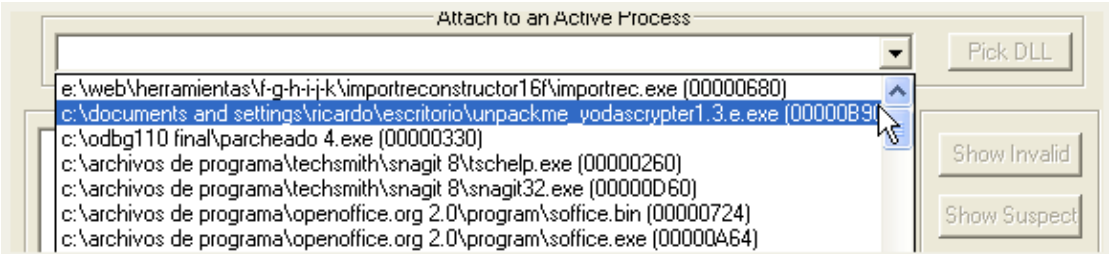
定位到这个 CALL 处,我们在其上面单击鼠标右键选择-Follow,看看被重定向到了哪里。

```
00153352 - E9 E918FA76   JMP 770F4C3B   OLEAUT32.SysStringLen
00153352 - E9 3D72FC76   JMP 7711A594   OLEAUT32.OleCreateFontIndirect
00153357 - E9 FD17FA76   JMP 770F4B59   OLEAUT32.SysAllocStringLen
0015335C - E9 211BFA76   JMP 770F4E82   OLEAUT32.SafeArrayGetDim
00153361 - E9 32A1FC76   JMP 7711D498   OLEAUT32.SafeArrayGetElemsize
00153366 - E9 301DFA76   JMP 770F509B   OLEAUT32.SafeArrayGetLBound
0015336B - E9 DF1CFA76   JMP 770F504F   OLEAUT32.SafeArrayGetUBound
00153370 - E9 9B1CFA76   JMP 770F5010   OLEAUT32.SafeArrayAccessData
00153375 - E9 C51CFA76   JMP 770F503F   OLEAUT32.SafeArrayUnaccessData
0015337A - E9 5A33FA76   JMP 770F66D9   OLEAUT32.VariantChangeType
0015337F - E9 CC14FA76   JMP 770F4850   OLEAUT32.SysFreeString
00153384 - E9 CC18FA76   JMP 770F4C55   OLEAUT32.SysAllocStringByteLen
00153389 - E9 3418FA76   JMP 770F4BC2   OLEAUT32.SysAllocString
0015338E - E9 029FFC76   JMP 7711D295   OLEAUT32.VariantCopy
00153393 - E9 E8290077   JMP 77155D80   OLEAUT32.OleLoadPicture
00153398   0000          ADD BYTE PTR DS:[EAX],AL
0015339A   0000          ADD BYTE PTR DS:[EAX],AL
0015339C   0000          ADD BYTE PTR DS:[EAX],AL
```

我们可以看到重定向到这里,并没有过多的弯弯绕就直接 JMP 去调用 SysStringLen 这个 API 函数了,这样我们就知道其真正要调用的 API 函数了。

这个重定向比较简单,我们打开 IMP REC,看看能不能用 IMP REC 自带的 Trace 功能进行修复,不过修复不成功,我们再来手动分析。

打开 IMP REC。



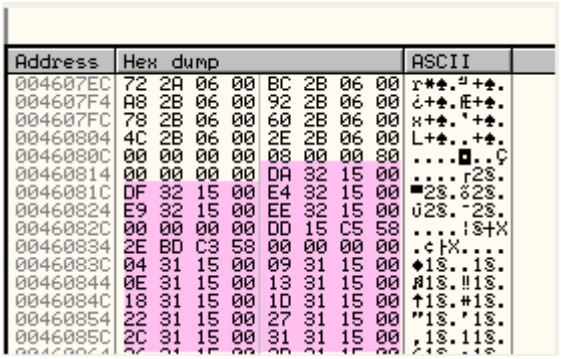
定位到 unpackme_yoda's crypter1.3 所在进程,当前其断在了 OEP 处,填写上 OEP,IAT 起始地址的 RVA,IAT 长度。

OEP = 4271B0 - 映像基址 400000 = 271B0 (RVA)。

我们现在来定位 IAT 的起始位置,往上拉。

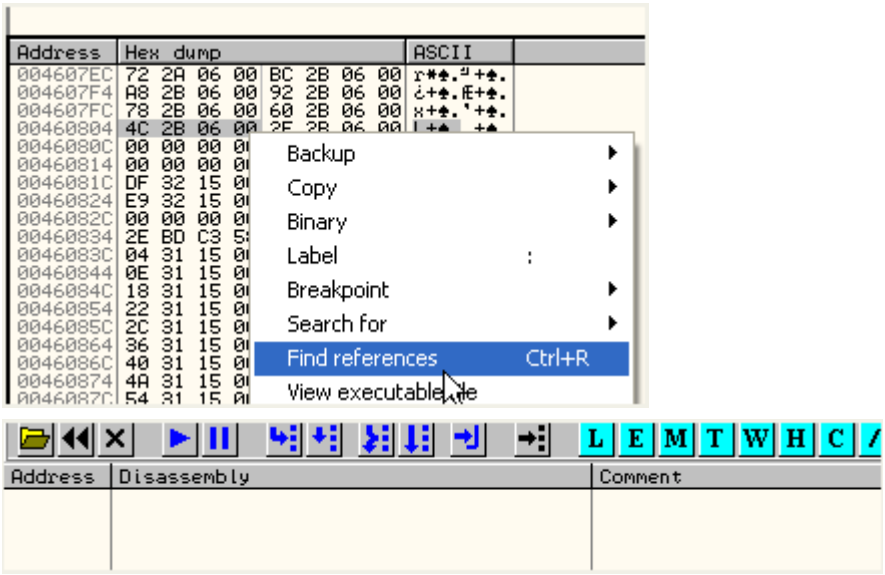
```
Address    Hex dump                  ASCII
004608FC   F4 31 15 00 F9 31 15 00   ¶1§.·1§.
00460904   FE 31 15 00 03 32 15 00   ■1§.♥2§.
0046090C   08 32 15 00 0D 32 15 00   ◘2§..2§.
00460914   12 32 15 00 17 32 15 00   ↕2§.↨2§.
0046091C   1C 32 15 00 21 32 15 00   ∟2§.!2§.
00460924   26 32 15 00 2B 32 15 00   &2§.+2§.
0046092C   30 32 15 00 35 32 15 00   02§.52§.
00460934   3A 32 15 00 3F 32 15 00   :2§.?2§.
0046093C   44 32 15 00 49 32 15 00   D2§.I2§.
00460944   4E 32 15 00 53 32 15 00   N2§.S2§.
0046094C   58 32 15 00 5D 32 15 00   X2§.]2§.
00460954   62 32 15 00 67 32 15 00   b2§.g2§.
0046095C   6C 32 15 00 71 32 15 00   l2§.q2§.
00460964   76 32 15 00 7B 32 15 00   v2§.{2§.
0046096C   80 32 15 00 00 00 00 00   Ç2§.....
00460974   6B 17 80 7C C1 C9 80 7C   k↨Ç|┴╔Ç|
0046097C   69 10 81 7C EE 1E 80 7C   i►ü|¯▲Ç|
00460984   8D 2C 81 7C 40 7A 94 7C   ì,ü|@zö|
0046098C   E1 EA 81 7C A2 CA 81 7C   ßΩü|¢╩ü|
00460994   16 1E 80 7C 43 99 80 7C   ▬▲Ç|CÖÇ|
0046099C   10 11 81 7C 29 29 81 7C   ►◄ü|))ü|
004609A4   14 9B 80 7C 81 9A 80 7C   ¶¢Ç|üÜÇ|
004609AC   FB 2C 82 7C AE 94 83 7C   ¹,é|«öâ|
```

这一部分是重定向到起始地址为 150000 的区段的,我们继续往上。



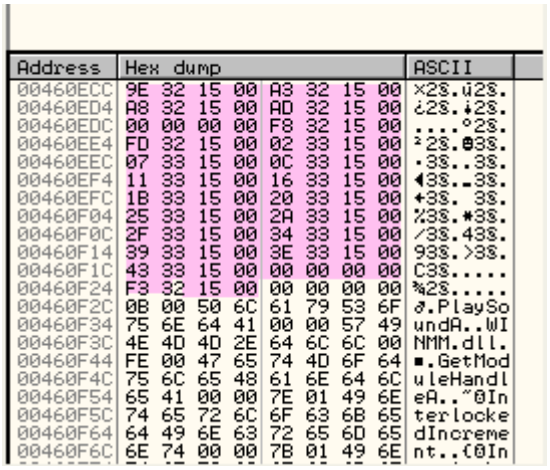
Address	Hex dump	ASCII
004607EC	72 2A 06 00 BC 2B 06 00	r*♠.╝+♠.
004607F4	A8 2B 06 00 92 2B 06 00	¿+♠.Æ+♠.
004607FC	78 2B 06 00 60 2B 06 00	x+♠.`+♠.
00460804	4C 2B 06 00 2E 2B 06 00	L+♠..+♠.
0046080C	00 00 00 00 08 00 00 80	....◘..Ç
00460814	00 00 00 00 DA 32 15 00	....┌2§.
0046081C	DF 32 15 00 E4 32 15 00	■2§.ä2§.
00460824	E9 32 15 00 EE 32 15 00	ú2§.ˆ2§.
0046082C	00 00 00 00 DD 15 C5 58	....¦§+X
00460834	2E BD C3 58 00 00 00 00	.¢├X....
0046083C	04 31 15 00 09 31 15 00	♦1§..1§.
00460844	0E 31 15 00 13 31 15 00	♫1§.‼1§.
0046084C	18 31 15 00 1D 31 15 00	↑1§.∟1§.
00460854	22 31 15 00 27 31 15 00	"1§.'1§.
0046085C	2C 31 15 00 31 31 15 00	,1§.11§.

这里我们可以看到 460810 这一项为 80000008,明显不是 IAT 项,继续往上的一部分是以 6XXXX 的形式,我们看看 460804 这一项的参考引用情况,会发现没有引用该项的地方。



因此,IAT 的起始地址为 460818,IAT 起始地址的 RVA = 460818 - 400000(映像基址) = 60818。

下面我们再来看看哪里是 IAT 的结束位置。



Address	Hex dump	ASCII
00460ECC	9E 32 15 00 A3 32 15 00	×2§.ú2§.
00460ED4	A8 32 15 00 AD 32 15 00	¿2§.¡2§.
00460EDC	00 00 00 00 F8 32 15 00	....°2§.
00460EE4	FD 32 15 00 02 33 15 00	²2§.☻3§.
00460EEC	07 33 15 00 0C 33 15 00	•3§..3§.
00460EF4	11 33 15 00 16 33 15 00	◄3§.▬3§.
00460EFC	1B 33 15 00 20 33 15 00	←3§. 3§.
00460F04	25 33 15 00 2A 33 15 00	%3§.*3§.
00460F0C	2F 33 15 00 34 33 15 00	/3§.43§.
00460F14	39 33 15 00 3E 33 15 00	93§.>3§.
00460F1C	43 33 15 00 00 00 00 00	C3§.....
00460F24	F3 32 15 00 00 00 00 00	≤2§.....
00460F2C	0B 00 50 6C 61 79 53 6F	♂.PlaySo
00460F34	75 6E 64 41 00 00 57 49	undA..WI
00460F3C	4E 4D 4D 2E 64 6C 6C 00	NMM.dll.
00460F44	FE 00 47 65 74 4D 6F 64	■.GetMod
00460F4C	75 6C 65 48 61 6E 64 6C	uleHandl
00460F54	65 41 00 00 7E 01 49 6E	eA..~☺In
00460F5C	74 65 72 6C 6F 63 6B 65	terlocke
00460F64	64 49 6E 63 72 65 6D 65	dIncreme
00460F6C	6E 74 00 00 7B 01 49 6E	nt..{☺In

这里 460f24 是 IAT 的最后一项的地址,因为后面 460F2C 这一项定位不到参考引用处。所以 460f28 是 IAT 的结束位置。现在我们来计算一下 IAT 的大小。

IAT 的大小 = 结束地址 - 起始地址 = 460F28 - 460818 = 710。

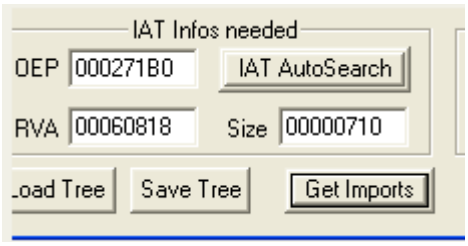
因此

OEP 的 RVA = 271B0

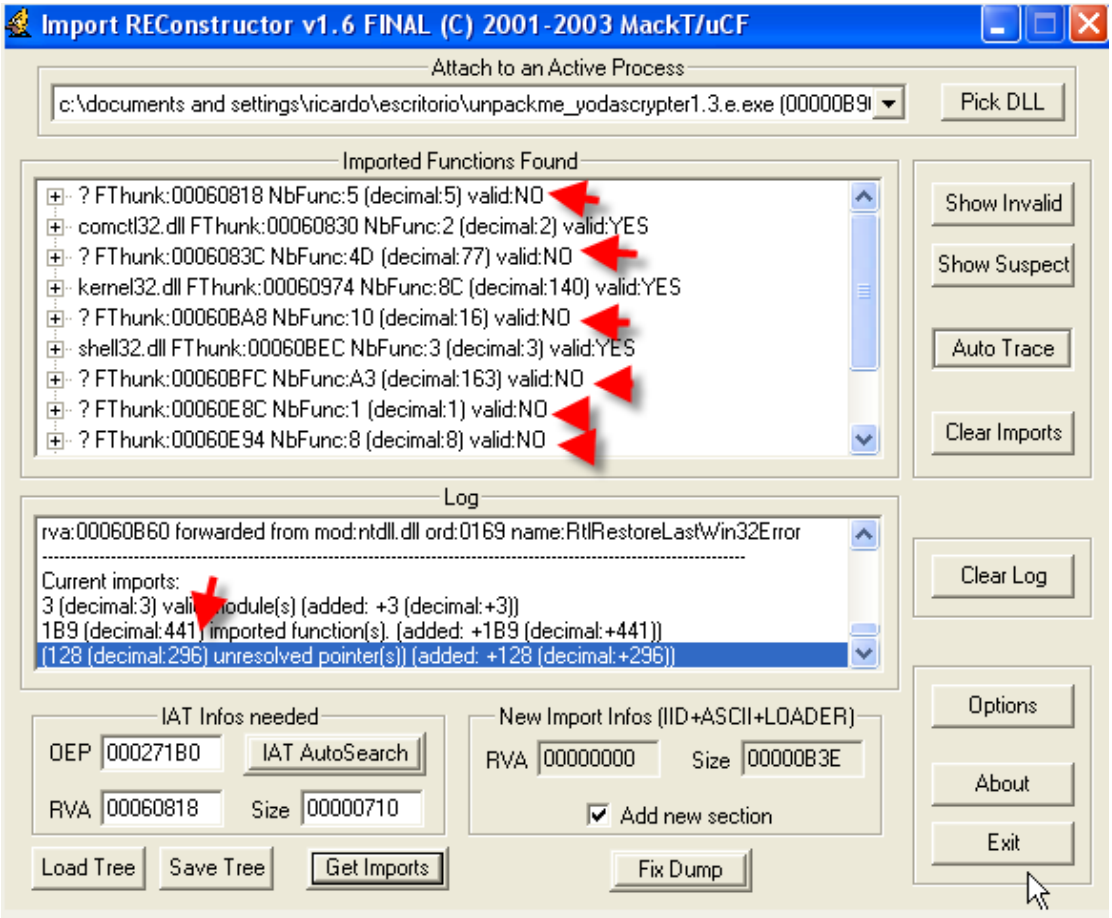
IAT 起始地址的 RVA = 60818

IAT 的长度 = 710

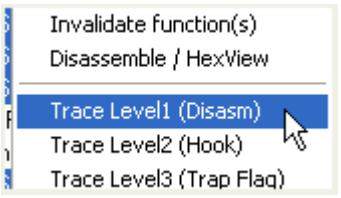
将这个三个值填到 IMP REC 中去。

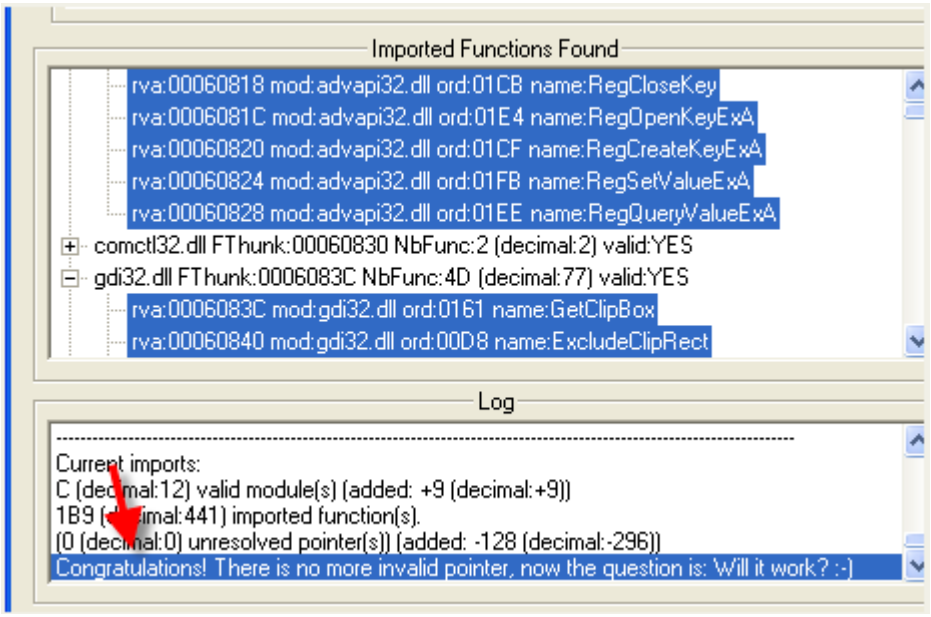


单击 Get Imports 按钮。

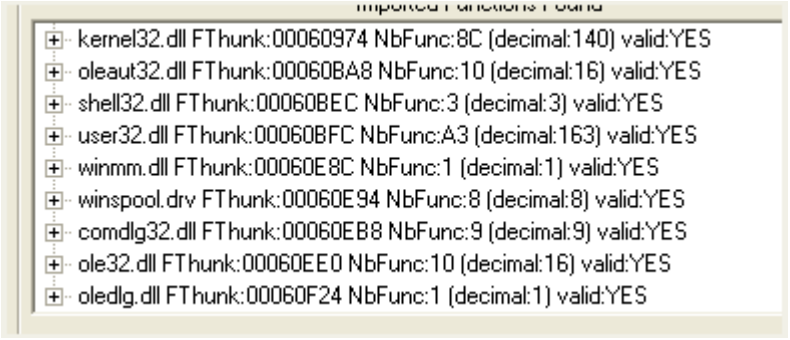


这里我们可以看到有 296 项被 IMP REC 标识为了无效,我们来试试 IMP REC 内置的 Tracer 能不能修复这些项。如果单击右边的 Auto Trace(自动跟踪)按钮的话,IMP REC 将被挂起(像卡死了一样)。我们再来试试其他的 Tracer,首先单击右边的 Show Invalid,接着在显示出来的无效项上面单击鼠标右键选择-Trace Level1(Disasm)。

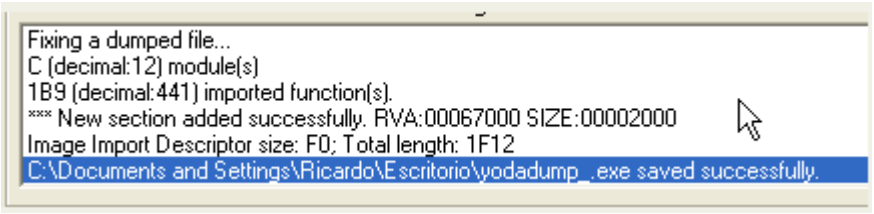




可以看到这里提示所有无效项均被修复,嘿嘿,我们再次单击 Show Invalid,可以看到所有的项都被标记为有效。



我们单击 Fix Dump 对刚刚 dump 出来的文件 yodadump.exe 进行修复。

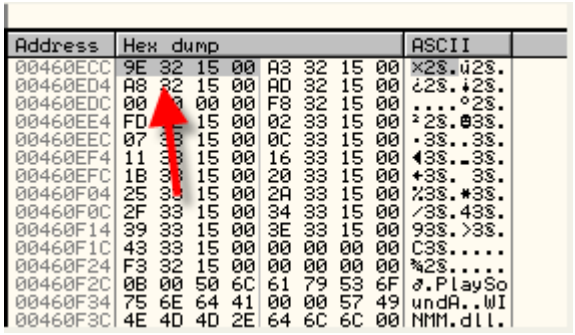


修复后的文件被重命名为了 yodadump_.exe,我们运行试试。

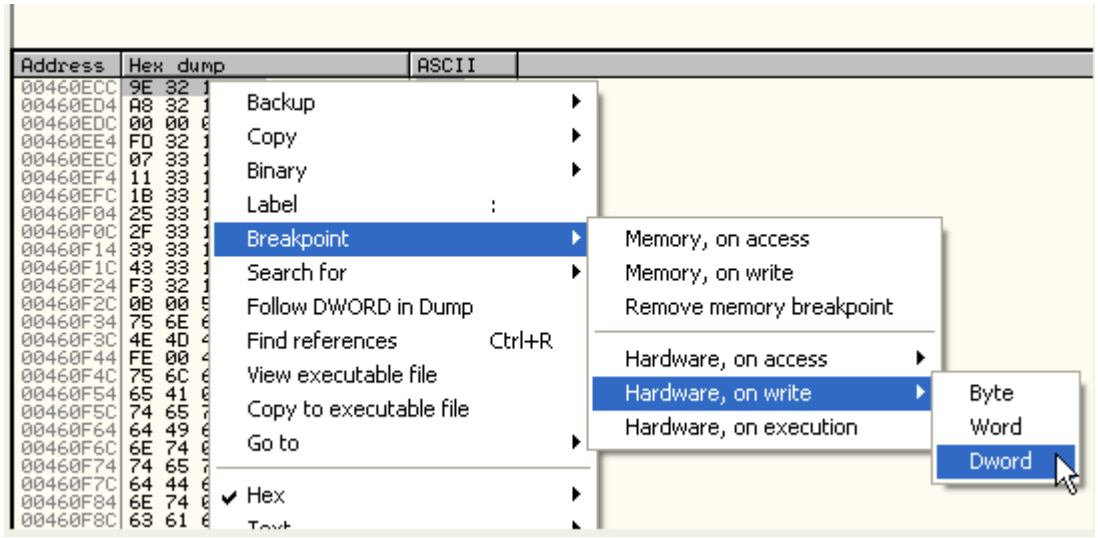
嘿嘿,正常运行。

下面我们来看看如何定位该壳的关键跳。

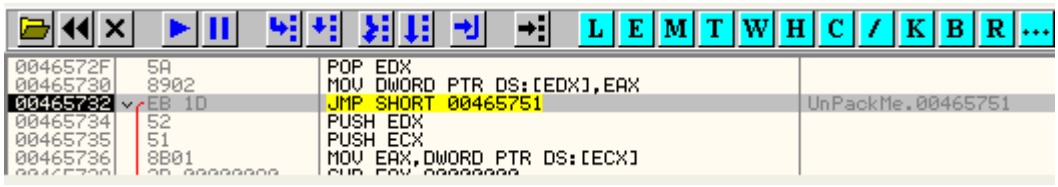
首先让程序停在 OEP 处,接着随便定位一个重定向过的 IAT 项。



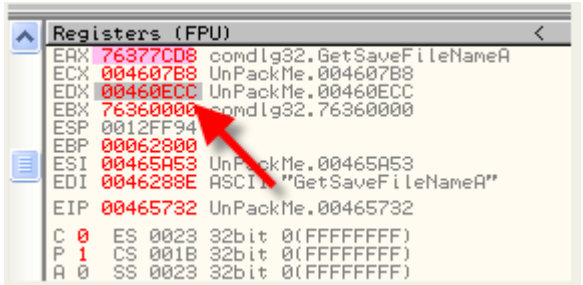
我们给 460ECC 这一项设置硬件写入断点,接着如果我们重启 OD 的话,硬件断点依然存在。



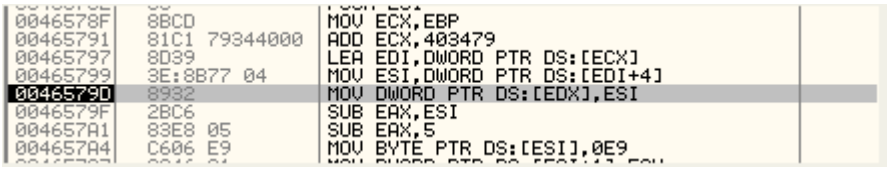
当 460ECC 地址处被写入的时候 OD 就会断下来,我们直接运行起来。



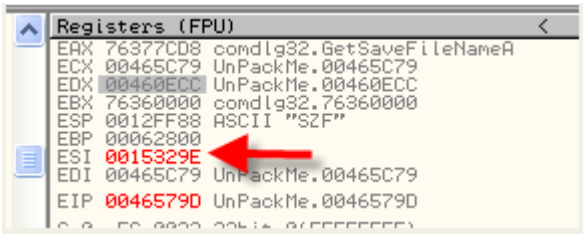
断在了这里,这里向 460ECC 中写入的是一个正常的值,此时,EAX 包含了一个 API 函数的地址。



EDX 的值 460ECC,即指向了之前我们设置了硬件断点的那个重定向的 IAT 项,这里其被写入的是正确的值,随后会被修改为重定向过的值,我们继续往下跟,看看会发生什么。

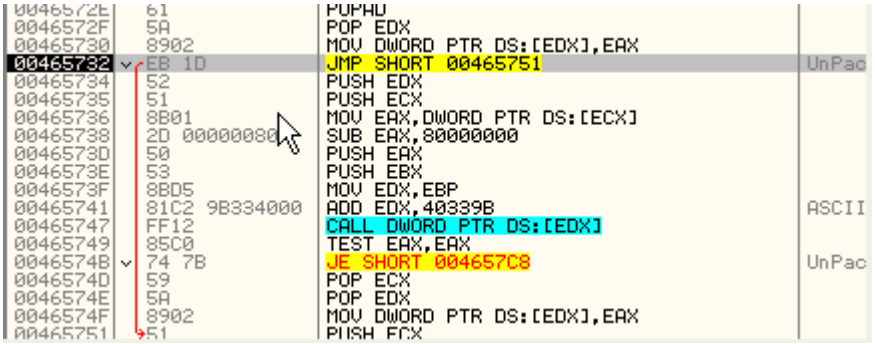


我们跟到这里,这里 460ECC 中将被写入重定向过的值,ESI 此时保存的就是重定向过的值。

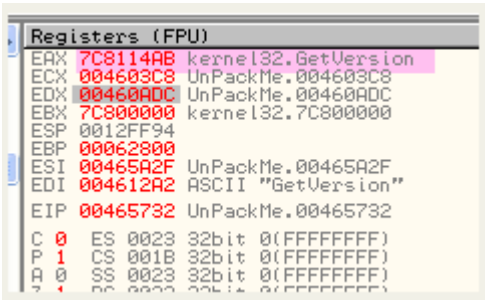


也就是说正常的 IAT 项只会被写入一次,而需要被重定向的项将被写入两次。

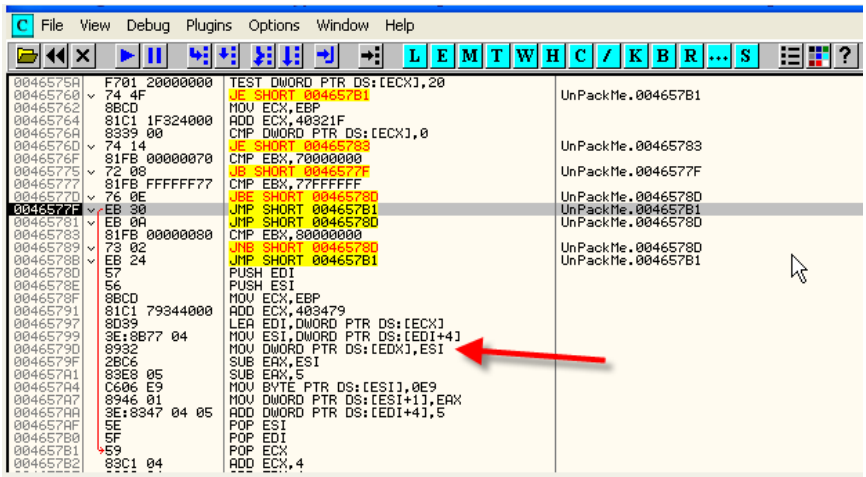
下面我们来看看正常的 IAT 的项的处理流程,就拿 OEP 下面调用的 GetVersion 这个 API 函数来说吧,即 460ADC 处将被壳填充上正常 IAT 值,我们对其设置硬件写入断点,运行起来。



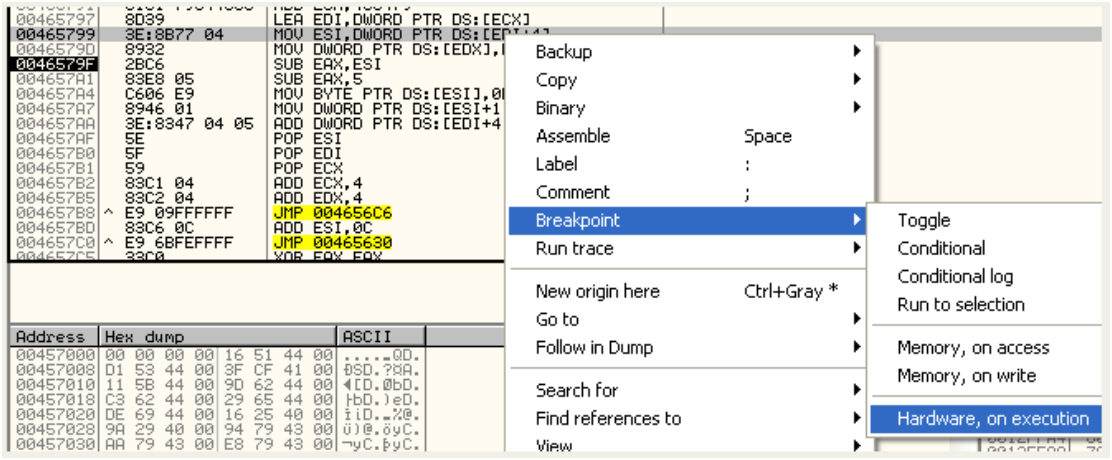
断在了这里,此时 EAX 保存正确 API 函数地址,将写入到 460ADC 中。



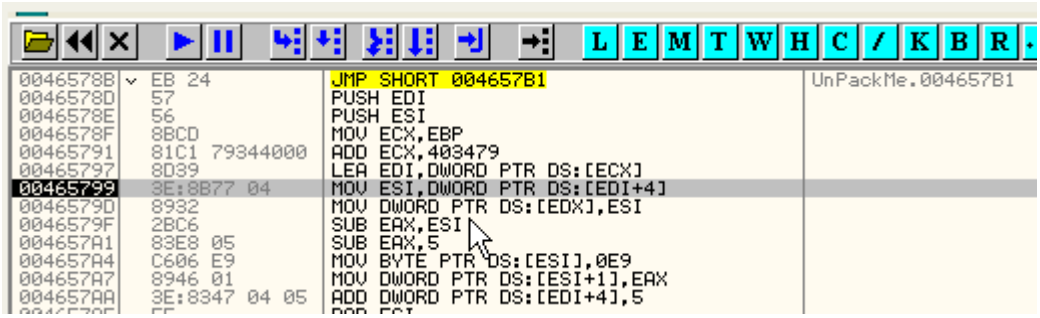
继续往下跟一点点。



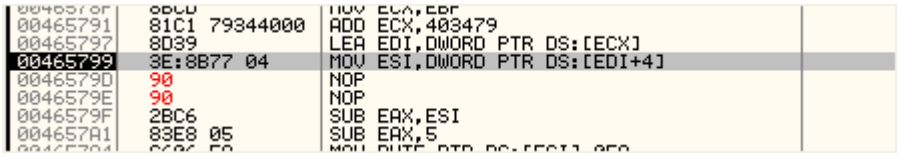
这里我们可以看到 46577F 这一处 JMP 指令,其将跳过红色箭头标注的设置为重定向值的指令,所以这里是关键跳,之前还有几处条件跳转,这里我们来尝试 NOP 掉写入重定向值的指令,看看效果如何。



首先给 465799 这条指令设置硬件执行断点,这里我们并不向想其被写入重定向的值,所以给其上一行设置硬件执行断点。

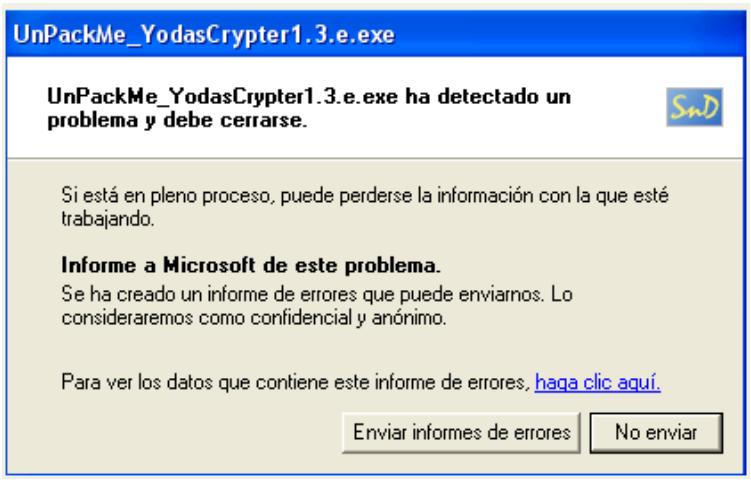


断了下来,我们将写入重定向值的指令 NOP 掉。



下面我们来看看该壳会不会检测自身代码被 NOP 掉了,我们运行到 OEP 处。

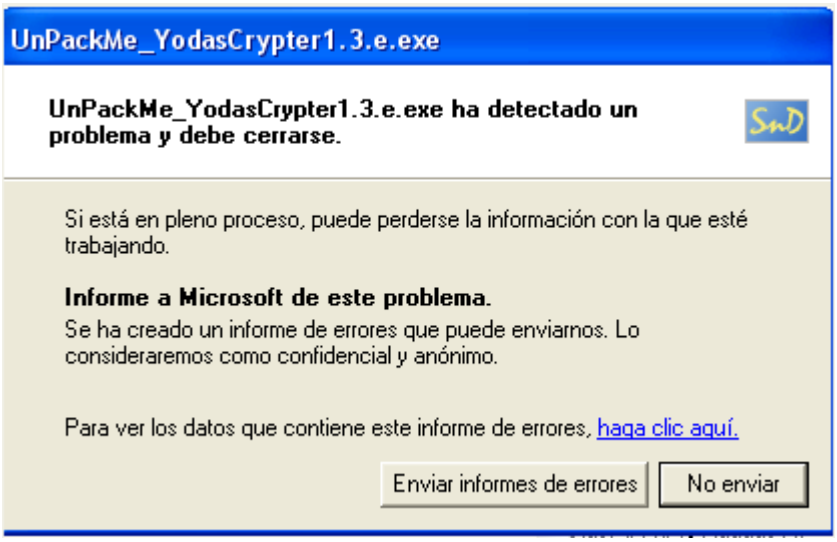
删除硬件断点,运行起来。



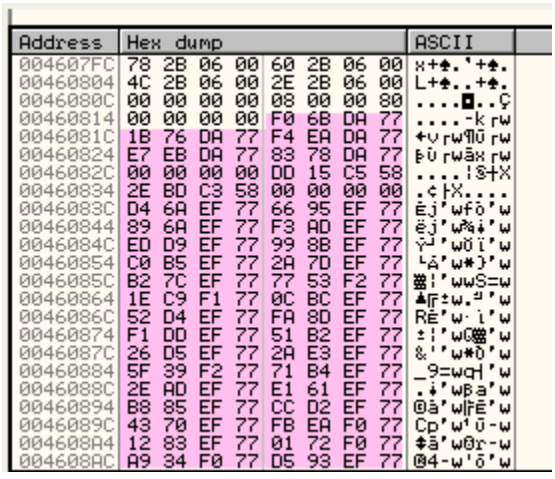
程序无法正常运行,可能是壳有自校验,检测到自身被修改了报错,所以我们不能 NOP,我们继续来到关键跳处。

```
0046576F   81FB 00000070   CMP EBX,70000000
00465775 v 72 08           JB SHORT 0046577F          UnPackMe.0046577F
00465777   81FB FFFFFF77   CMP EBX,77FFFFFF
0046577D v 76 0E           JBE SHORT 0046578D         UnPackMe.0046578D
0046577F v EB 30           JMP SHORT 004657B1         UnPackMe.004657B1
00465781 v EB 0A           JMP SHORT 0046578D         UnPackMe.0046578D
00465783   81FB 00000080   CMP EBX,80000000
00465789 v 73 02           JNB SHORT 0046578D         UnPackMe.0046578D
0046578B v EB 24           JMP SHORT 004657B1         UnPackMe.004657B1
0046578D   57              PUSH EDI
0046578E   56              PUSH ESI
```

我们随便跟一个将被重定向的项,跟到关键跳附近,可以看到 46577D 处的条件跳转将直接跳到下面,这样 46577F 处的关键跳将得不到执行,接着将被写入重定向的值,我们尝试将 46577D 这处条件跳转 NOP 掉,让其直接执行 46577F 处的 JMP 指令,我们来看看效果。

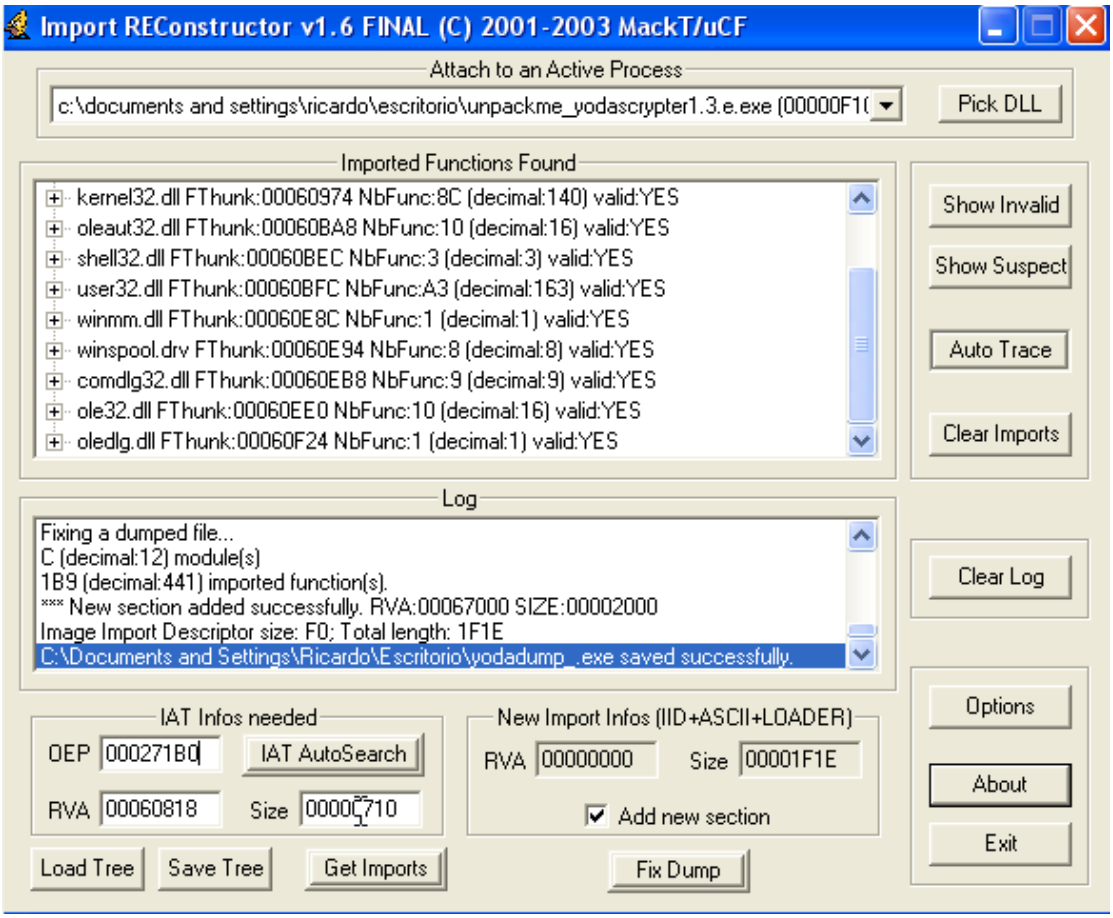


依然是报错。

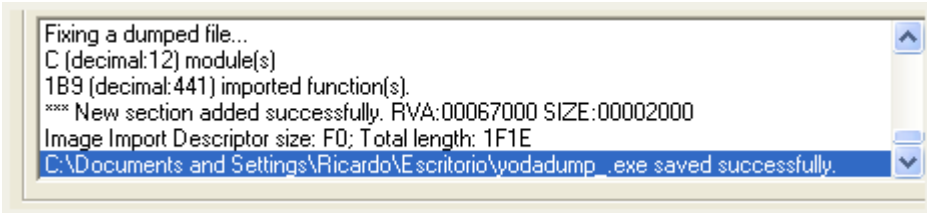


Address	Hex dump	ASCII
004607FC	78 2B 06 00 60 2B 06 00	x+♠.`+♠.
00460804	4C 2B 06 00 2E 2B 06 00	L+♠..+♠.
0046080C	00 00 00 00 08 00 00 80	....◘..Ç
00460814	00 00 00 00 F0 6B DA 77	....≡kΓw
0046081C	1B 76 DA 77 F4 EA DA 77	←vΓw¶Ωw
00460824	E7 EB DA 77 83 78 DA 77	τδΓwâxΓw
0046082C	00 00 00 00 DD 15 C5 58	....¦§+X
00460834	2E BD C3 58 00 00 00 00	.¢├X....
0046083C	D4 6A EF 77 66 95 EF 77	Ôj'wfò'w
00460844	89 6A EF 77 F3 AD EF 77	ëj'w‰i'w
0046084C	ED D9 EF 77 99 8B EF 77	ÿ┘'wöï'w
00460854	C0 B5 EF 77 2A 7D EF 77	└Á'w*}'w
0046085C	B2 7C EF 77 77 53 F2 77	▒¡'wwS=w
00460864	1E C9 F1 77 0C BC EF 77	▲╔±w.╝'w
0046086C	52 D4 EF 77 FA 8D EF 77	RÈ'w·ì'w
00460874	F1 DD EF 77 51 B2 EF 77	±¦'wQ▒'w
0046087C	26 D5 EF 77 2A E3 EF 77	&╒'w*ô'w
00460884	5F 39 F2 77 71 B4 EF 77	_9=wq┤'w
0046088C	2E AD EF 77 E1 61 EF 77	.↓'wßa'w
00460894	B8 85 EF 77 CC D2 EF 77	@à'w╠ë'w
0046089C	43 70 EF 77 FB EA F0 77	Cp'w'ü-w
004608A4	12 83 EF 77 01 72 F0 77	↕â'wΘr-w
004608AC	A9 34 F0 77 D5 93 EF 77	@4-w'õ'w

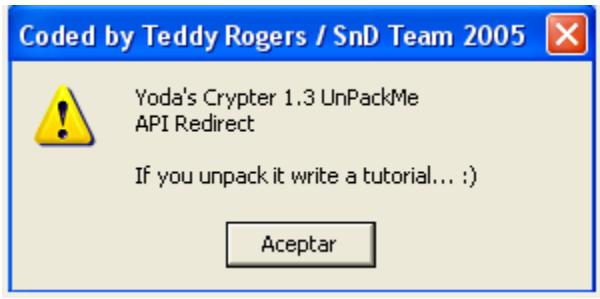
那么我们来看看 IAT 吧,我们可以看到 IAT 项都被修复了,都是正确的。这里我们有两种选择: 一,再开一个 OD,加载该 CrackMe 的另一个实例,直接跟到 OEP 处,不修改任何东西,然后将当前我们这个实例的正确的 IAT 复制出来,覆盖掉新开的这个实例的 IAT,注意是二进制复制,这个方法比较简单。二,直接用 IMP REC 定位到 CrackMe 的进程,此时 IAT 全部被写入正确的值了,但是还未到达 OEP 处,我们直接填上 OEP,RVA,SIZE 等数据,没有到达 OEP 处没有关系,我们单击 Get Imports。



我们可以看到所有的 IAT 项都是正确的,直接单击 Fix Dump 修复之前 dump 出来的文件即可。



运行修复过的 yodadump_.exe,看看效果。



我们可以看到正常运行。本章到此为止。

给大家留个练习 unpackme_FSG_1.31_dulek。比较简单,大家应该不会觉得很难。