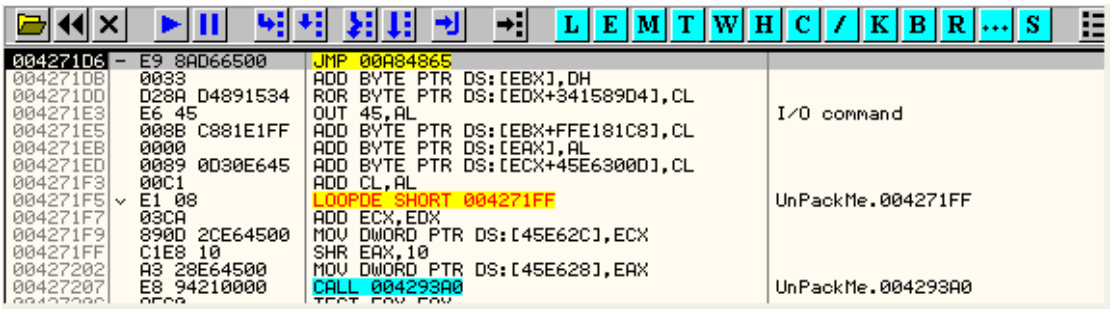
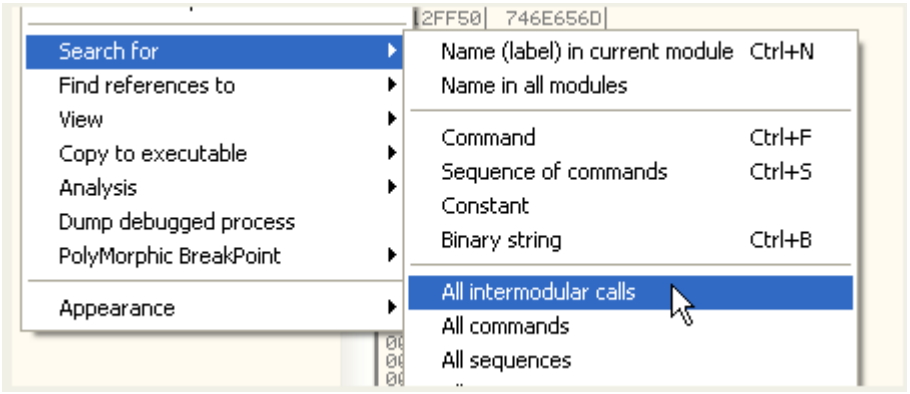
第四十一章-神马是 AntiDump

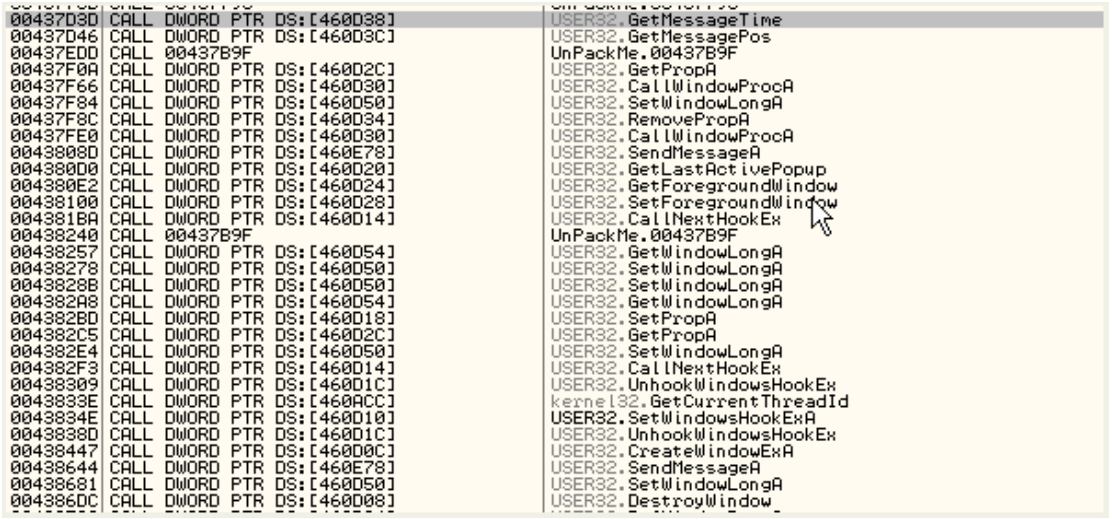
本章我们继续脱 PELock,修复其 IAT。

这里我们仍然采用最后一次异常法定位到 OEP 处。

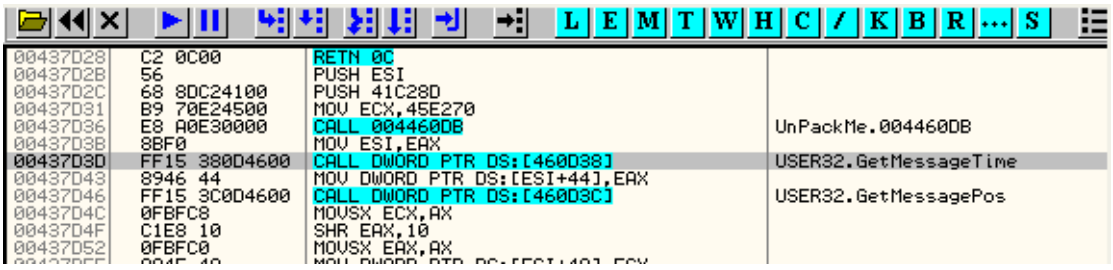


好了,现在我们到达了假的 OEP 处,接着我们单击鼠标右键选择 Search for-All intermodular calls,查看 API 函数的调用列表。



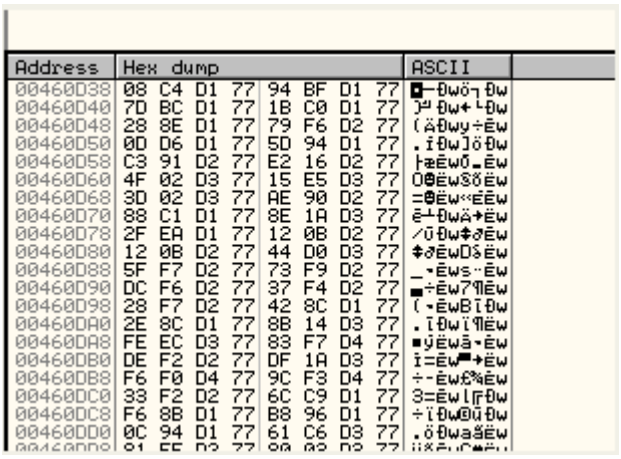


这里我们随便选择一个,选中并双击之。



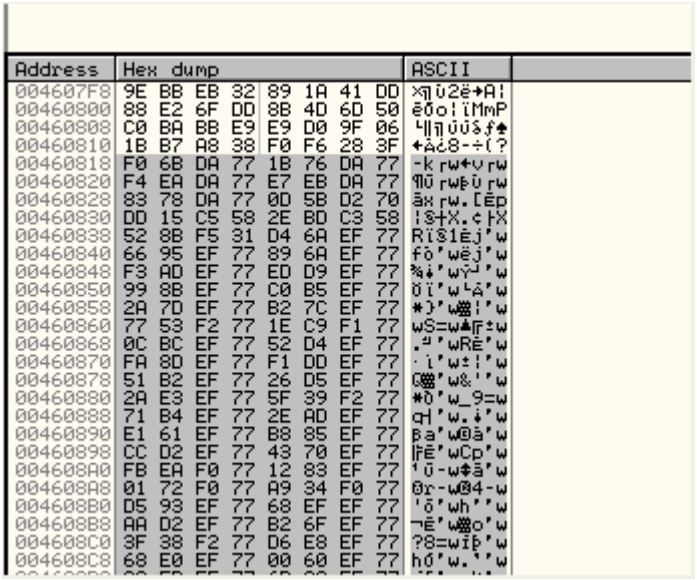
这里我们可以看到该 API 函数的入口地址位于 460D38 这个内存单元中的,也就是说 460D38 为 IAT 的其中一项。好,在数据窗口

中定位到该地址。



Address	Hex dump	ASCII
00460D38	08 C4 D1 77 94 BF D1 77	
00460D40	7D BC D1 77 1B C0 D1 77	
00460D48	28 8E D1 77 79 F6 D2 77	
00460D50	0D D6 D1 77 5D 94 D1 77	
00460D58	C3 91 D2 77 E2 16 D2 77	
00460D60	4F 02 D3 77 15 E5 D3 77	
00460D68	3D 02 D3 77 AE 90 D2 77	
00460D70	88 C1 D1 77 8E 1A D3 77	
00460D78	2F EA D1 77 12 0B D2 77	
00460D80	12 0B D2 77 44 D0 D3 77	
00460D88	5F F7 D2 77 73 F9 D2 77	
00460D90	DC F6 D2 77 37 F4 D2 77	
00460D98	28 F7 D2 77 42 8C D1 77	
00460DA0	2E 8C D1 77 8B 14 D3 77	
00460DA8	FE EC D3 77 83 F7 D4 77	
00460DB0	DE F2 D2 77 DF 1A D3 77	
00460DB8	F6 F0 D4 77 9C F3 D4 77	
00460DC0	33 F2 D2 77 6C C9 D1 77	
00460DC8	F6 8B D1 77 B8 96 D1 77	
00460DD0	0C 94 D1 77 61 C6 D3 77	

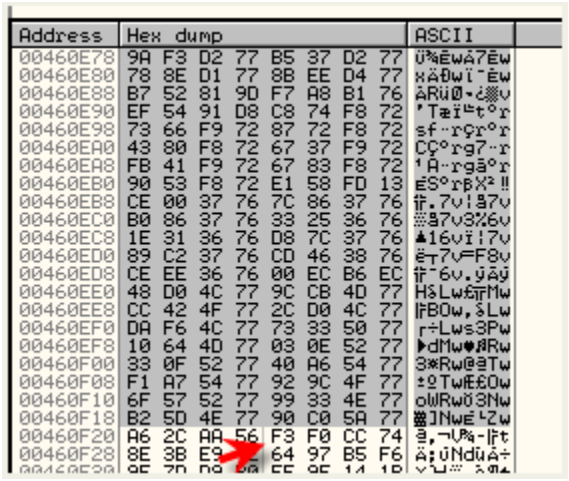
我们可以看到这部分 IAT 项是正确的,我们往上看,定位 IAT 的起始位置。



Address	Hex dump	ASCII
004607F8	9E BB EB 32 89 1A 41 DD	
00460800	88 E2 6F DD 8B 4D 6D 50	
00460808	C0 BA BB E9 E9 D0 9F 06	
00460810	1B B7 A8 38 F0 F6 28 3F	
00460818	F0 6B DA 77 1B 76 DA 77	
00460820	F4 EA DA 77 E7 EB DA 77	
00460828	83 78 DA 77 0D 5B D2 70	
00460830	DD 15 C5 58 2E BD C3 58	
00460838	52 8B F5 31 D4 6A EF 77	
00460840	66 95 EF 77 89 6A EF 77	
00460848	F3 AD EF 77 ED D9 EF 77	
00460850	99 8B EF 77 C0 B5 EF 77	
00460858	2A 7D EF 77 B2 7C EF 77	
00460860	77 53 F2 77 1E C9 F1 77	
00460868	0C BC EF 77 52 D4 EF 77	
00460870	FA 8D EF 77 F1 DD EF 77	
00460878	51 B2 EF 77 26 D5 EF 77	
00460880	2A E3 EF 77 5F 39 F2 77	
00460888	71 B4 EF 77 2E AD EF 77	
00460890	E1 61 EF 77 B8 85 EF 77	
00460898	CC D2 EF 77 43 70 EF 77	
004608A0	FB EA F0 77 12 83 EF 77	
004608A8	01 72 F0 77 A9 34 F0 77	
004608B0	D5 93 EF 77 68 EF EF 77	
004608B8	AA D2 EF 77 B2 6F EF 77	
004608C0	3F 38 F2 77 D6 E8 EF 77	
004608C8	68 E0 EF 77 00 60 EF 77	

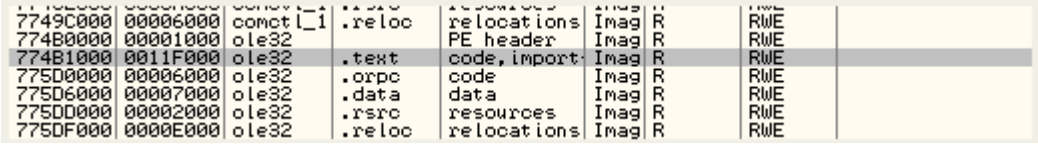
这里很明显 460818 为 IAT 的起始位置,因为上面的项,查看参考引用的话,会发现没有参考引用处,并且也不属于系统 DLL 的区段。

这里很幸运,IAT 项都是正确的,没有被重定向,所以我们并不需要修复重定向,接下来,我们来定位 IAT 的结束位置。

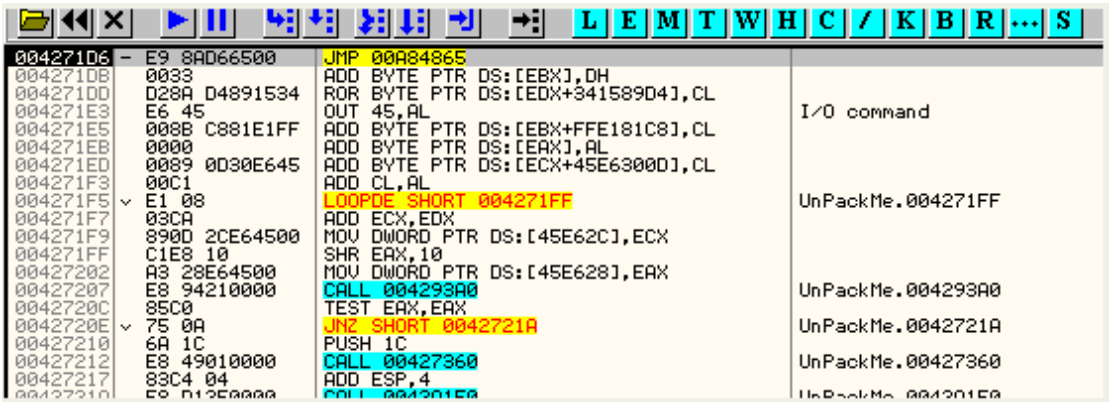


Address	Hex dump	ASCII
00460E78	9A F3 D2 77 B5 37 D2 77	
00460E80	78 8E D1 77 8B EE D4 77	
00460E88	B7 52 81 9D F7 A8 B1 76	
00460E90	EF 54 91 D8 C8 74 F8 72	
00460E98	73 66 F9 72 87 72 F8 72	
00460EA0	43 80 F8 72 67 37 F9 72	
00460EA8	FB 41 F9 72 67 83 F8 72	
00460EB0	90 53 F8 72 E1 58 FD 13	
00460EB8	CE 00 37 76 7C 86 37 76	
00460EC0	B0 86 37 76 33 25 36 76	
00460EC8	1E 31 36 76 D8 7C 37 76	
00460ED0	89 C2 37 76 CD 46 38 76	
00460ED8	CE EE 36 76 00 EC B6 EC	
00460EE0	48 D0 4C 77 9C CB 4D 77	
00460EE8	CC 42 4F 77 2C D0 4C 77	
00460EF0	DA F6 4C 77 73 33 50 77	
00460EF8	10 64 4D 77 03 0E 52 77	
00460F00	33 0F 52 77 40 A6 54 77	
00460F08	F1 A7 54 77 92 9C 4F 77	
00460F10	6F 57 52 77 99 33 4E 77	
00460F18	B2 5D 4E 77 90 C0 5A 77	
00460F20	A6 2C AA 56 F3 F0 CC 74	
00460F28	8E 3B E9 [redacted] 64 97 B5 F6	

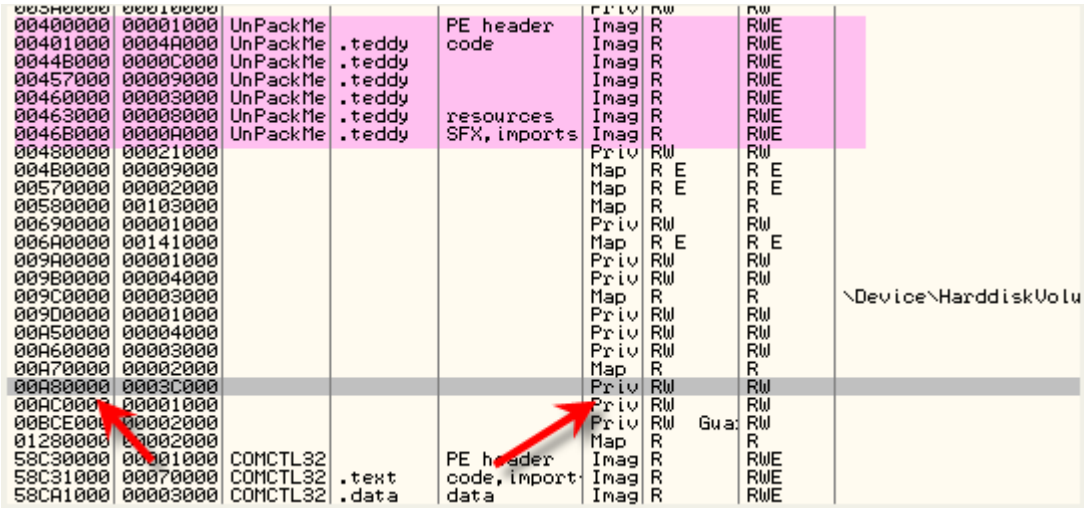
这里灰色的部分没有问题,但是 460F24 这一项就是比较可疑了,这一项没有参考引用。灰色的这部分是属于 ole32.dll 的代码段的。



这里我们选中属于该 DLL 的某些项,会发现也没有参考引用。

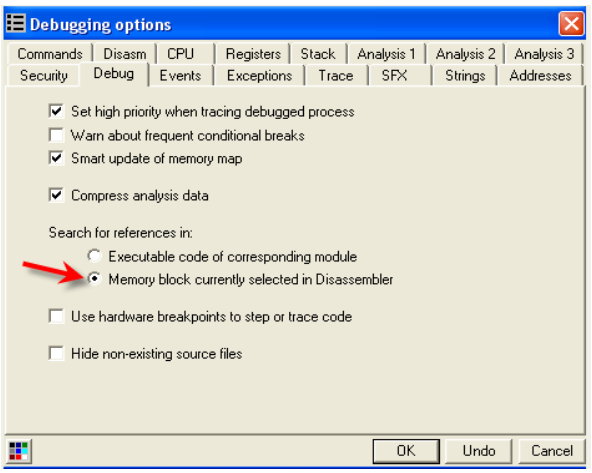


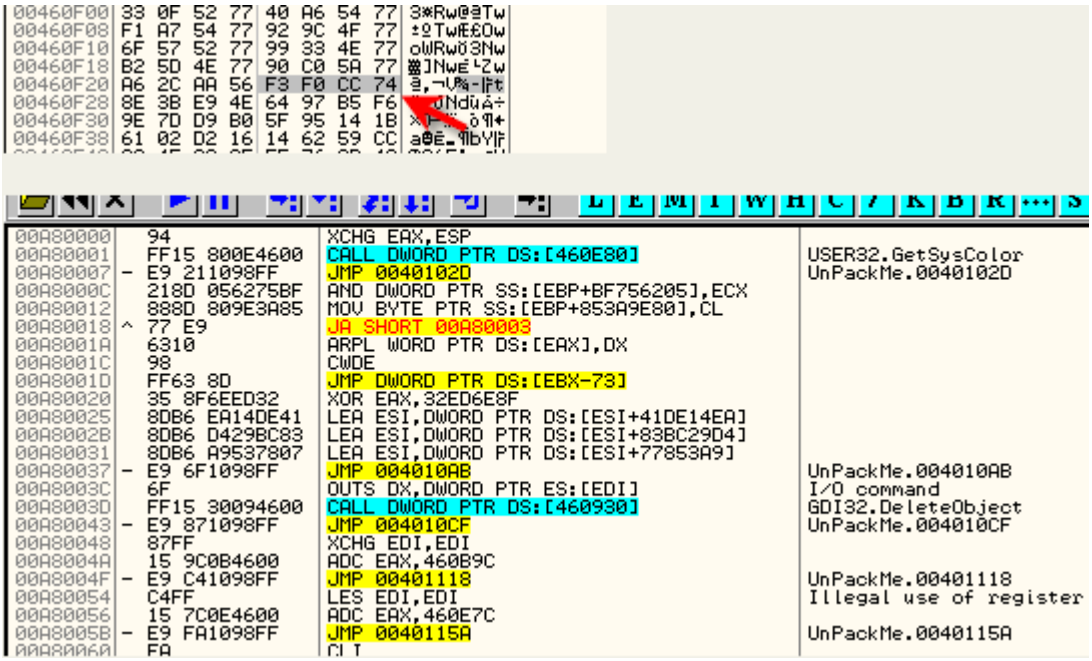
我们回到 OEP 处,可以看到该 JMP 指令将要跳转的一个地址为 Axxxxx(在我的机器上为这种形式,在其他的机器上可能会不一样)的形式,该地址不属于 exe 所在的区段,我们来看一看区段列表。



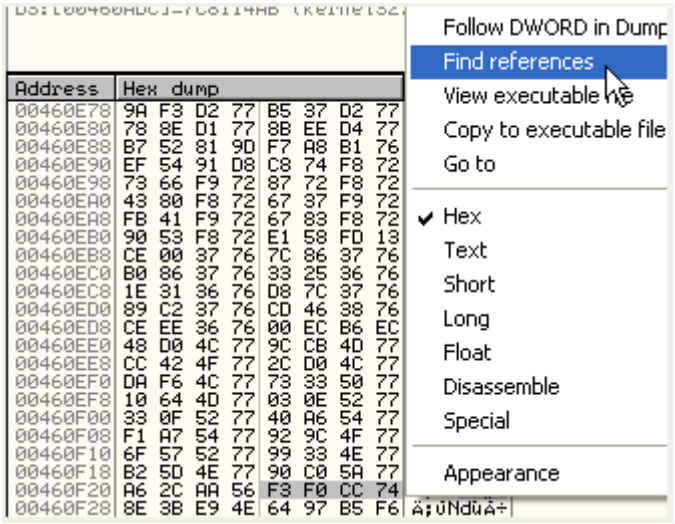
我们知道原始程序在 OEP 附近是太可能跳转到 exe 之外的区段的,那么 Axxxxx 所在的这个区段很可能是通过 VirtualAlloc 或者类似的函数创建的,所以这个区段很可能是壳创建的。

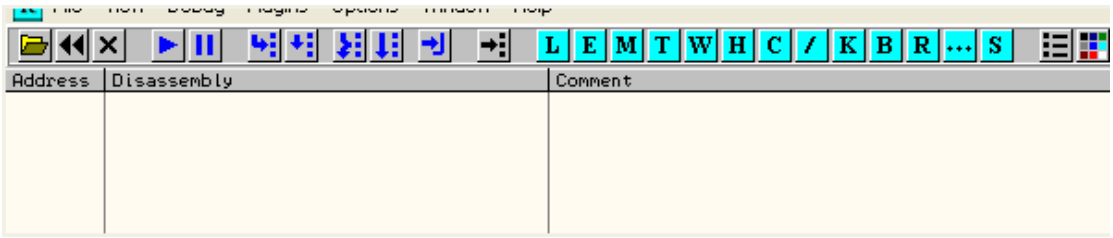
在这个区段中壳可能会将 IAT 的中某些值取出来加以变换就转化为 API 函数的调用了。下面我们来对 OD 的调试选项加以配置。



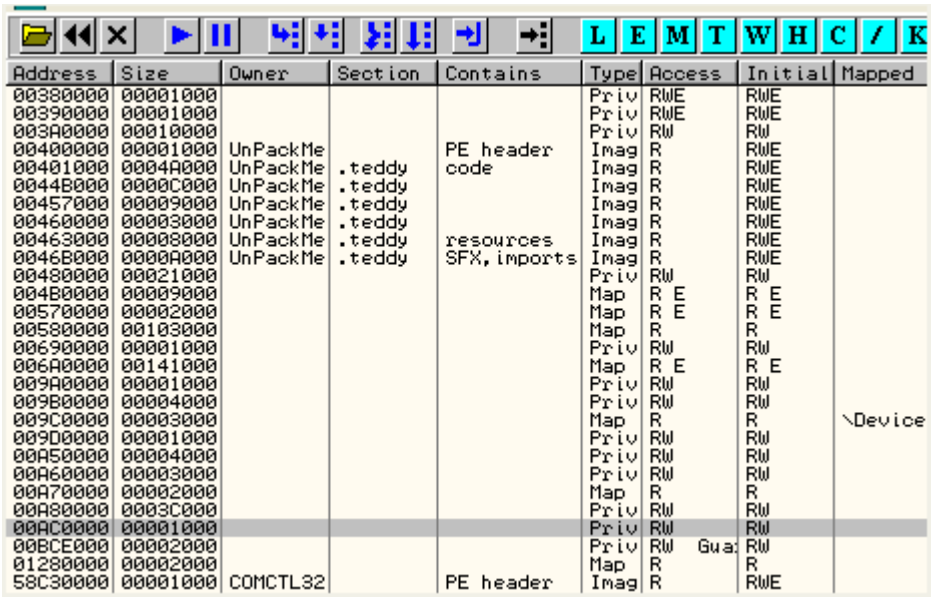


以上就是壳创建的一个区段,我们来查看一下 460F24 这一项的参考引用。

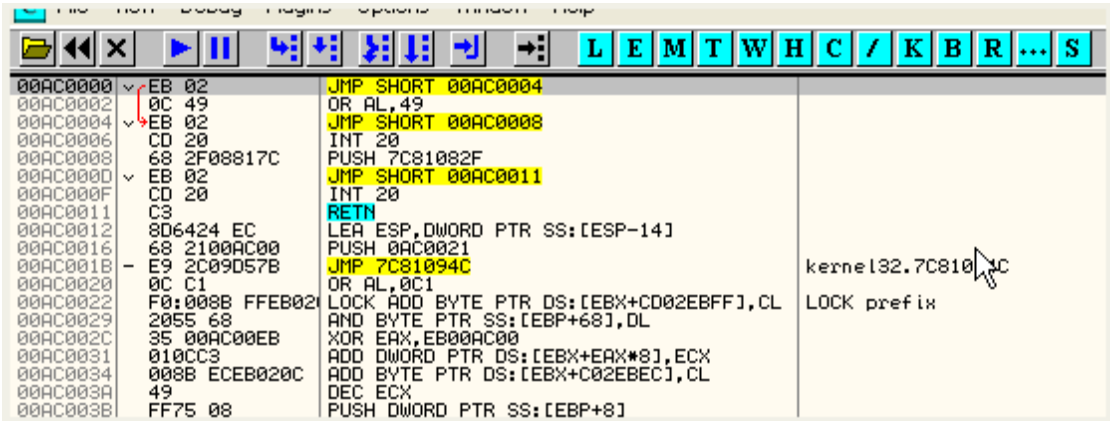




依然是什么也没有,别灰心。我们继续看下面一个区段。



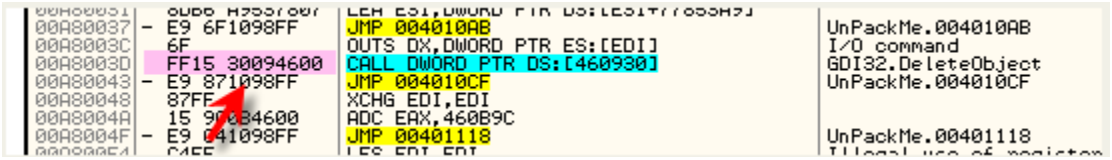
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped
00380000	00001000				Priv	RWE	RWE	
00390000	00001000				Priv	RWE	RWE	
003A0000	00010000				Priv	RW	RW	
00400000	00001000	UnPackMe		PE header	Imag	R	RWE	
00401000	0004A000	UnPackMe	.teddy	code	Imag	R	RWE	
0044B000	0000C000	UnPackMe	.teddy		Imag	R	RWE	
00457000	00009000	UnPackMe	.teddy		Imag	R	RWE	
00460000	00003000	UnPackMe	.teddy		Imag	R	RWE	
00463000	00008000	UnPackMe	.teddy	resources	Imag	R	RWE	
0046B000	0000A000	UnPackMe	.teddy	SFX,imports	Imag	R	RWE	
00480000	00021000				Priv	RW	RW	
004B0000	00009000				Map	R E	R E	
00570000	00002000				Map	R E	R E	
00580000	00103000				Map	R	R	
00690000	00001000				Priv	RW	RW	
006A0000	00141000				Map	R E	R E	
009A0000	00001000				Priv	RW	RW	
009B0000	00004000				Priv	RW	RW	
009C0000	00003000				Map	R	R	\Device
009D0000	00001000				Priv	RW	RW	
00A50000	00004000				Priv	RW	RW	
00A60000	00003000				Priv	RW	RW	
00A70000	00002000				Map	R	R	
00A80000	0003C000				Priv	RW	RW	
00AC0000	00001000				Priv	RW	RW	
00BCE000	00002000				Priv	RW Gua	RW	
01280000	00002000				Map	R	R	
58C30000	00001000	COMCTL32		PE header	Imag	R	RWE	



```
00AC0000  EB 02          JMP SHORT 00AC0004
00AC0002  0C 49          OR AL,49
00AC0004  EB 02          JMP SHORT 00AC0008
00AC0006  CD 20          INT 20
00AC0008  68 2F08817C    PUSH 7C81082F
00AC000D  EB 02          JMP SHORT 00AC0011
00AC000F  CD 20          INT 20
00AC0011  C3             RETN
00AC0012  8D6424 EC      LEA ESP,DWORD PTR SS:[ESP-14]
00AC0016  68 2100AC00    PUSH 0AC0021
00AC001B  E9 2C09D57B    JMP 7C81094C                    kernel32.7C810
00AC0020  0C C1          OR AL,0C1
00AC0022  F0:008B FFEB02 LOCK ADD BYTE PTR DS:[EBX+CD02EBFF],CL  LOCK prefix
00AC0029  2055 68        AND BYTE PTR SS:[EBP+68],DL
00AC002C  35 00AC00EB    XOR EAX,EB00AC00
00AC0031  010CC3         ADD DWORD PTR DS:[EBX+EAX*8],ECX
00AC0034  008B ECEB020C  ADD BYTE PTR DS:[EBX+C02EBEC],CL
00AC003A  49             DEC ECX
00AC003B  FF75 08        PUSH DWORD PTR SS:[EBP+8]
```

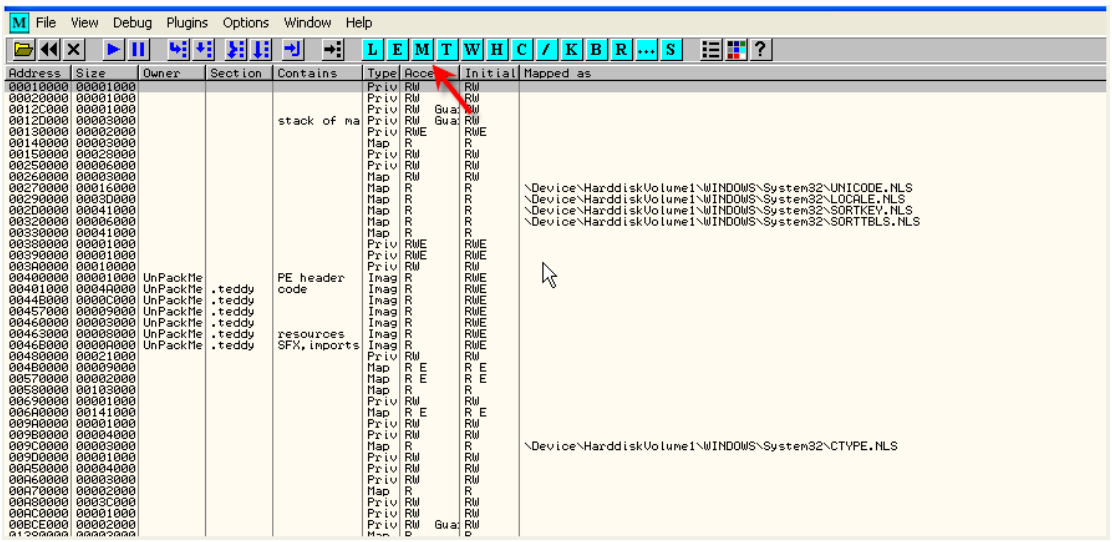
也看不出什么端倪。

我们回头再看看 A80000 这个区段,我们注意到 A8003D 这处 CALL [460930],其对应的机器码为 FF15 30094600,也就是说如果 CALL [460F24]的话,其对应的机器码应该是 FF15 240F4600,也就是说可以通过这种方式来调用 API。

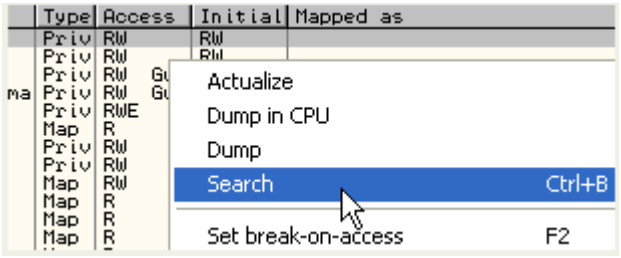


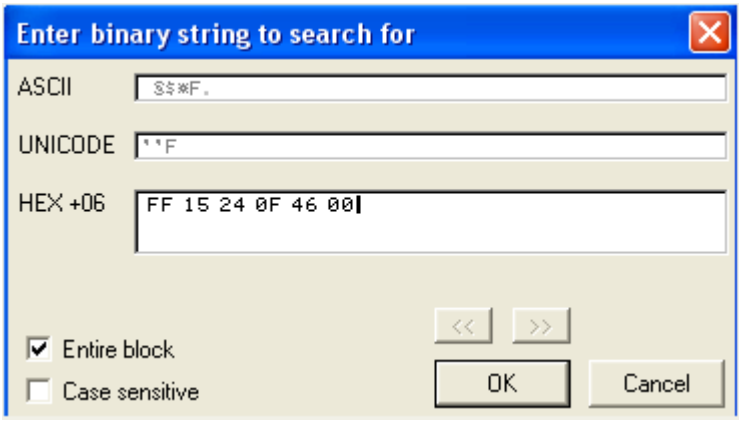
```
00A80037  E9 6F1098FF    JMP 004010AB                    UnPackMe.004010AB
00A8003C  6F             OUTS DX,DWORD PTR ES:[EDI]      I/O command
00A8003D  FF15 30094600  CALL DWORD PTR DS:[460930]      GDI32.DeleteObject
00A80043  E9 871098FF    JMP 004010CF                    UnPackMe.004010CF
00A80048  87FF           XCHG EDI,EDI
00A8004A  15 9C0B4600    ADC EAX,460B9C
00A8004F  E9 C41098FF    JMP 00401118                    UnPackMe.00401118
```

好了,下面我们搜索一下 FF15 240F4600 这串二进制串。

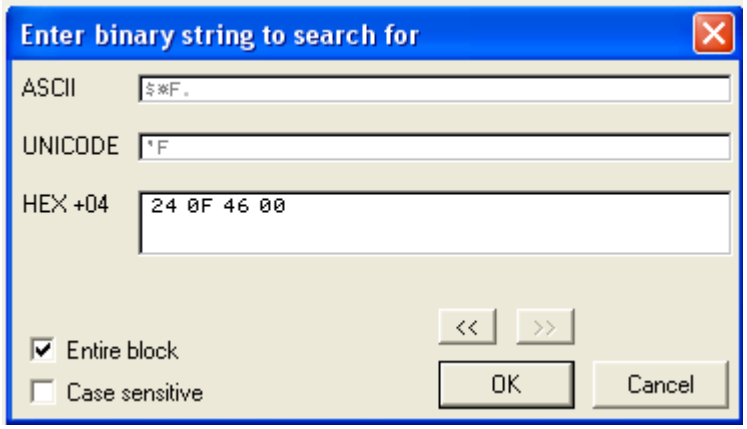


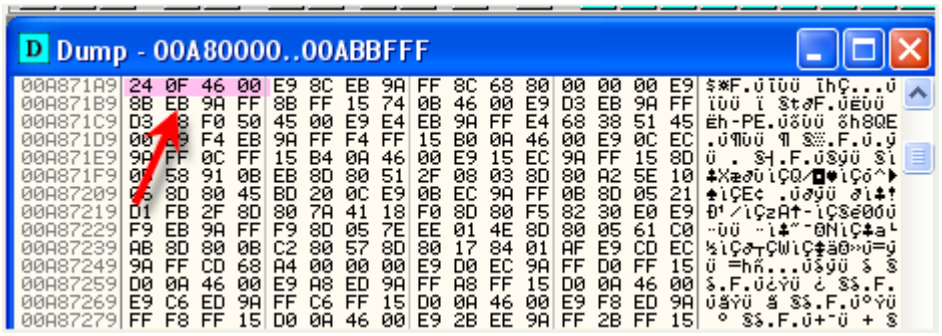
单击工具栏上面的 M 按钮打开区段列表窗口,选中第一个区段,单击鼠标右键选择-Search。



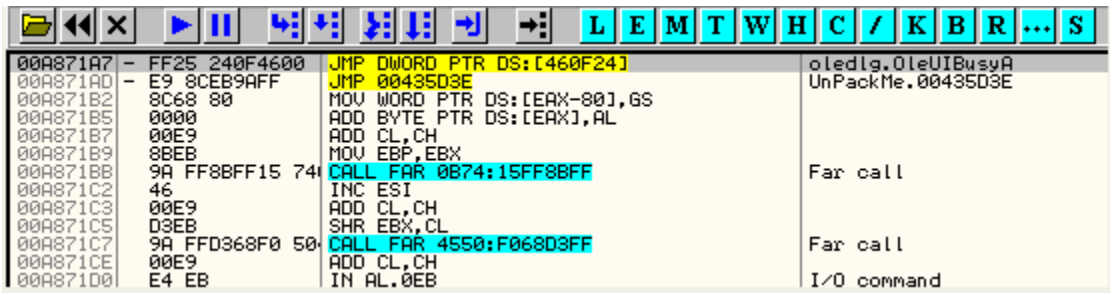


在 16 进制编辑框中写入 FF15240F4600 字节序列,单击 OK,会发现,什么也没找到,说明有可能不是通过间接 CALL,我们去掉前面的 FF15,再次搜索。

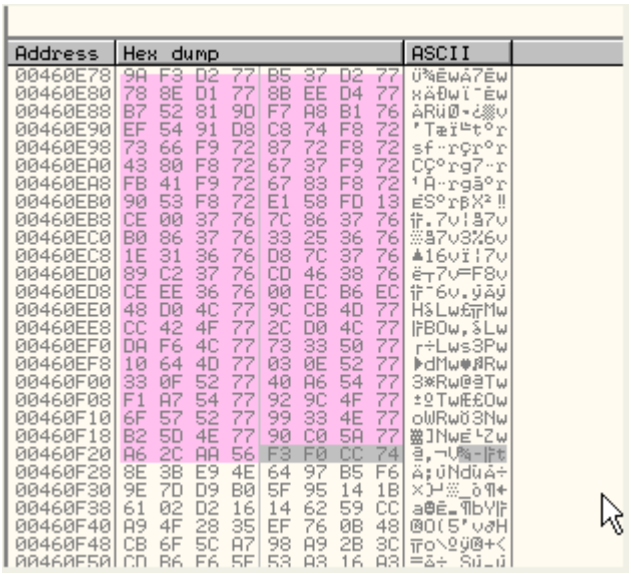




正好定位到了壳创建的这个区段,我们其反汇编代码是什么。



这里我们可以看到是通过 JMP 来调用的,而并非 CALL,所以机器码为 FF25,并非 FF15。也就是说 460F24 也是 IAT 中的一项。



所以说 IAT 的结束位置为 460F28。

IAT 的长度 = 结束位置 - 起始位置 = 460F28 - 460818 = 710

因此:

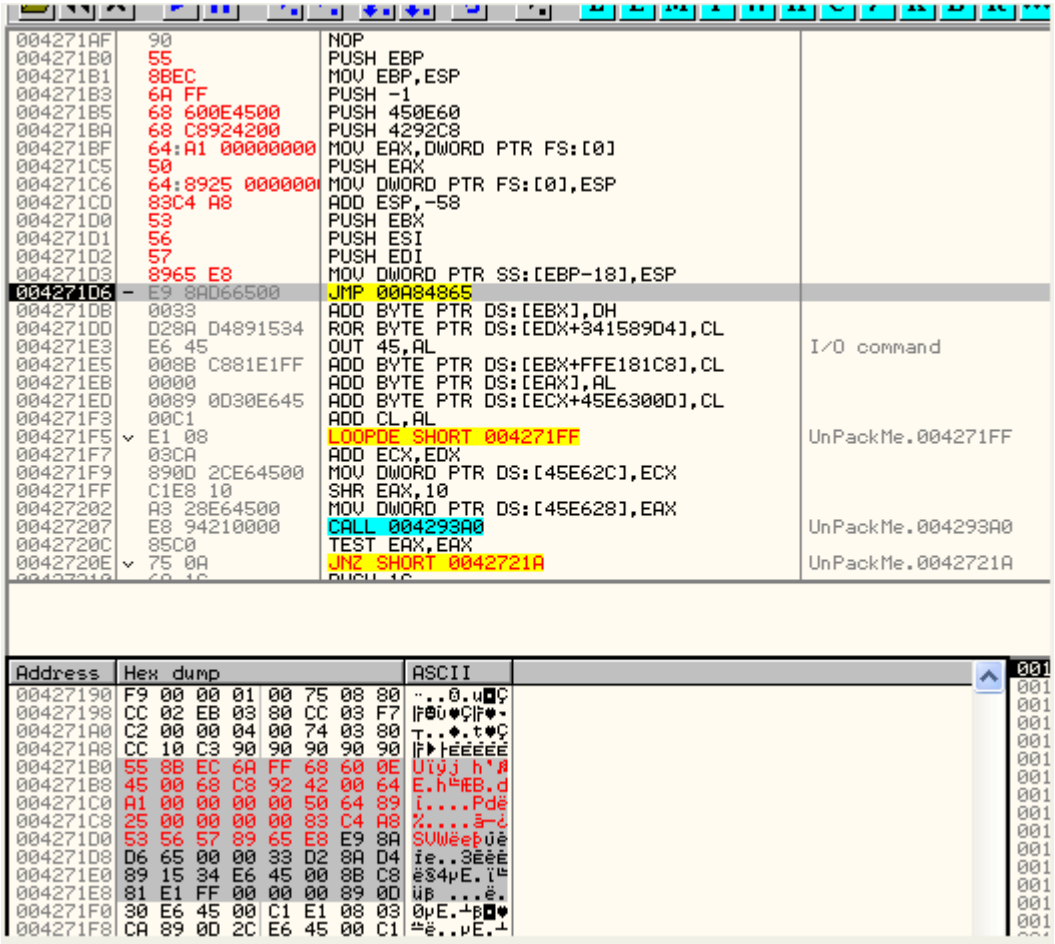
OEP 的 RVA = 271B0

IAT 起始地址的 RVA = 60818

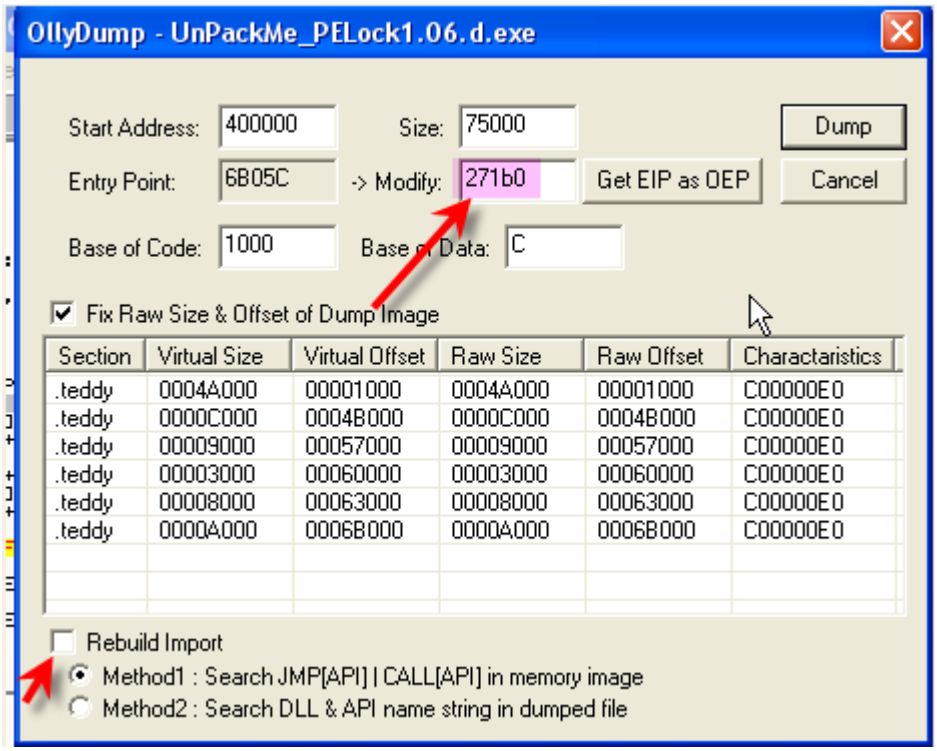
现在我们可以看到 IAT 项都是正确的了,我们回到假的 OEP 处,将 stolen bytes 填充上。

stolen bytes 是 55 8B EC 6A FF 68 60 0E 45 00 68 C8 92 42 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 C4 A8 53 56 57 89 65 E8 。

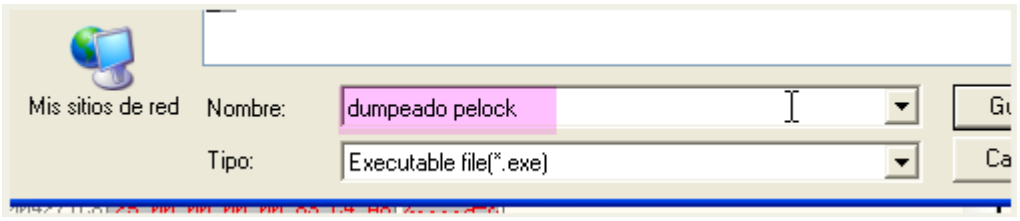
我们在数据窗口中定位到 4271B0 处, 按照第 39 章介绍的方式将 stolen bytes 填充好。



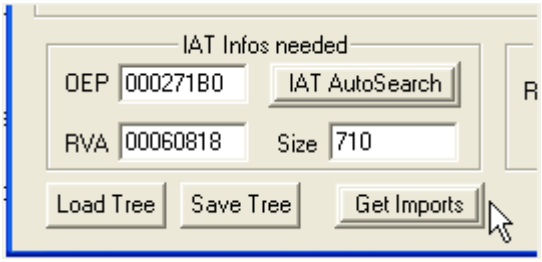
现在我们来 dump,将 OEP 指定为 4271B0。



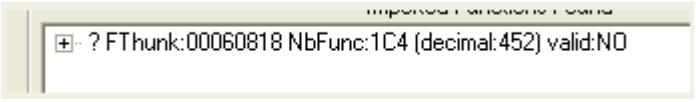
这里我们把 OEP 修正了。去掉 Rebuild Import 的对勾,我们不使用 OllyDump 的重建输入表的功能。



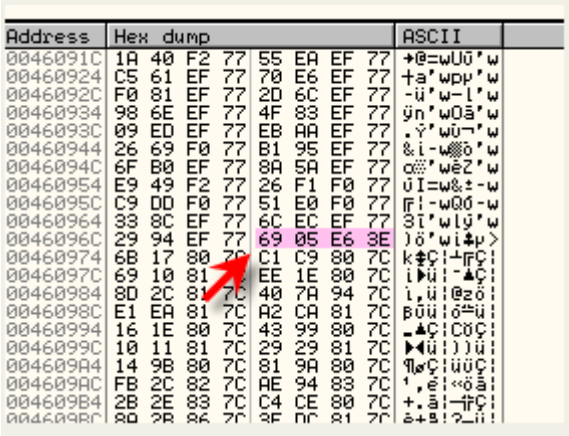
将 dump 出来的文件命名为 dumppeado pelock,现在我们来修复 IAT,打开 IMP REC,将 OEP 修正。



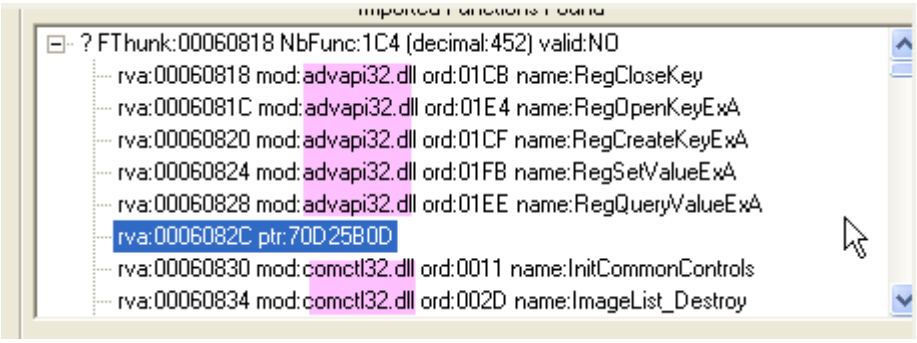
将 IAT 的信息填好,接着单击 Get Imports 按钮。



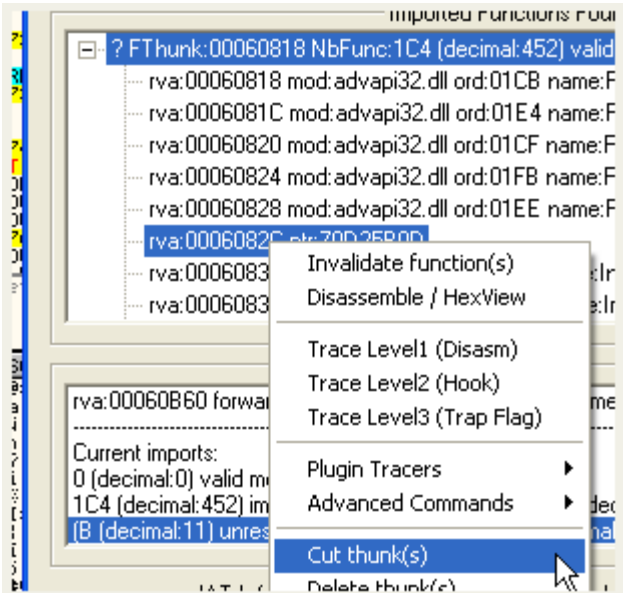
我们可以看到提示有无效的项,但是并没有经过重定向,只是每个 DLL 的 IAT 项原本应该是用零间隔开的,这里壳使用垃圾数据将零覆盖掉了。我们来看一看。

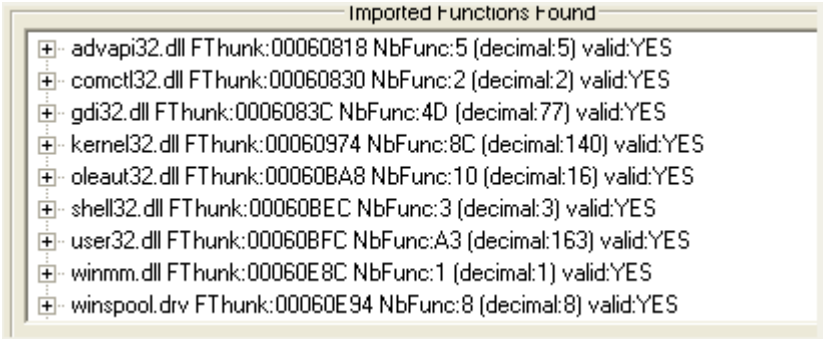


这里应该为零,用 IMP REC 修复很容易,我们单击 Show Invalid 即可。



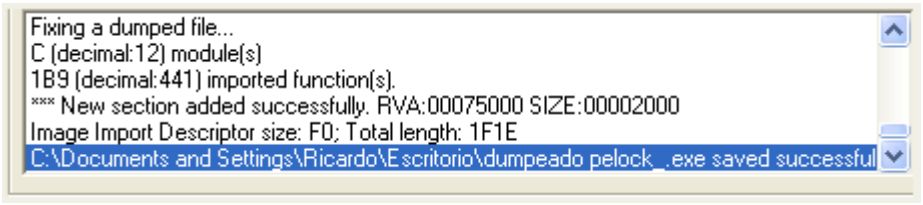
这里我们可以看到 advapi32.dll 与 comctl32.dll 的项之间被垃圾数据隔开了,选中这些无效的项,单击鼠标右键选择 Cut thunk(s),剪切掉这些无效的项。





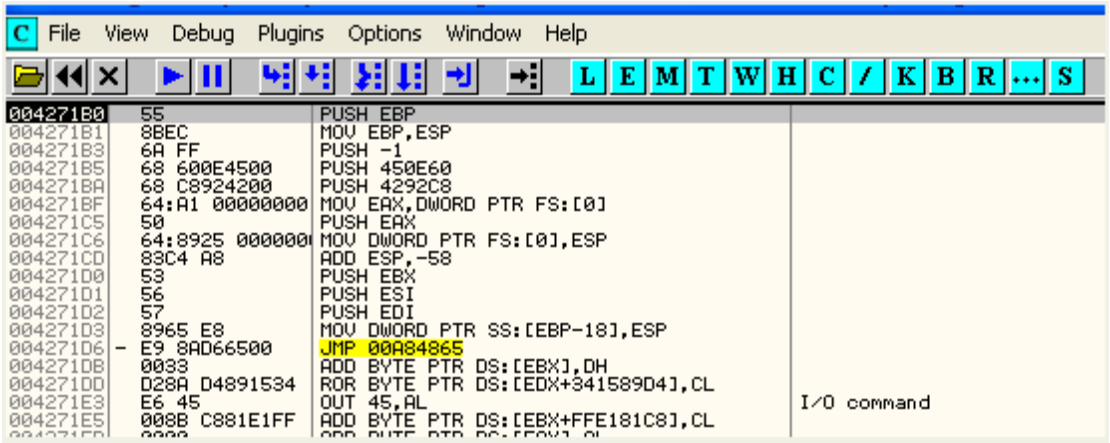
我们可以看到剪切掉这些无效的项后,显示的项都标识为有效的了。

接着我们单击 Fix Dump 按钮来修复刚刚 dump 出来的文件。

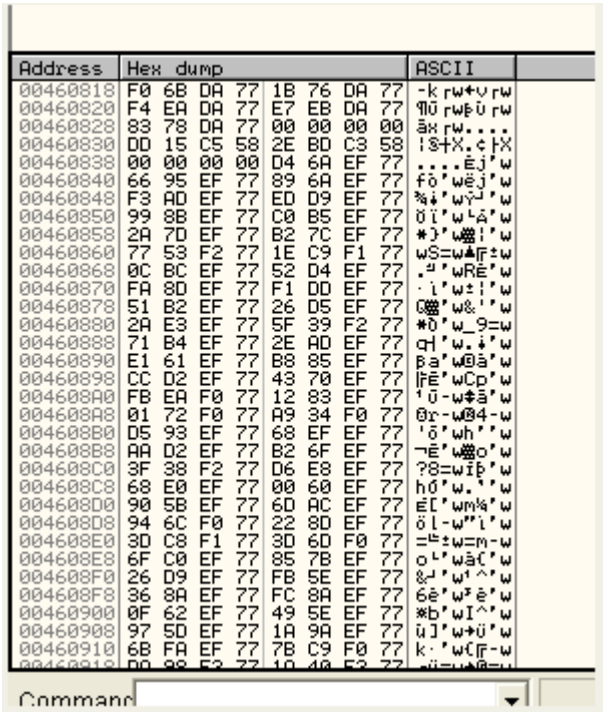


修复后的文件被重命名为了 dumpeado pelock_.exe。如果我们运行它的话,会提示错误。

下面我们需要用到 LordPe 来修复。



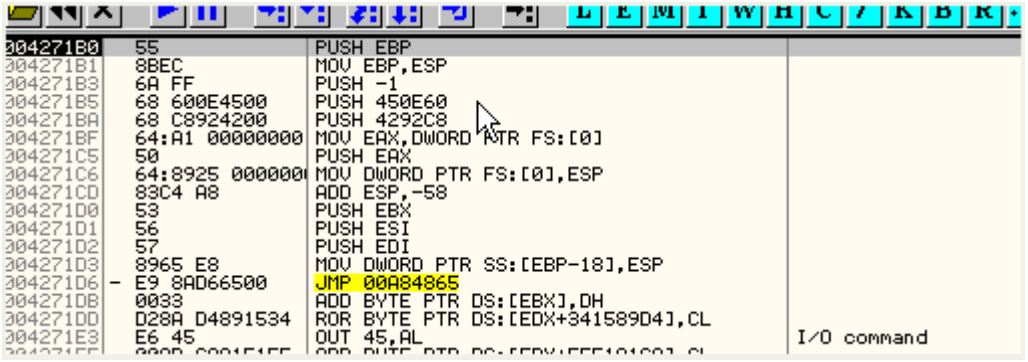
此时我们处在正确的入口点处,我们来看看 IAT。



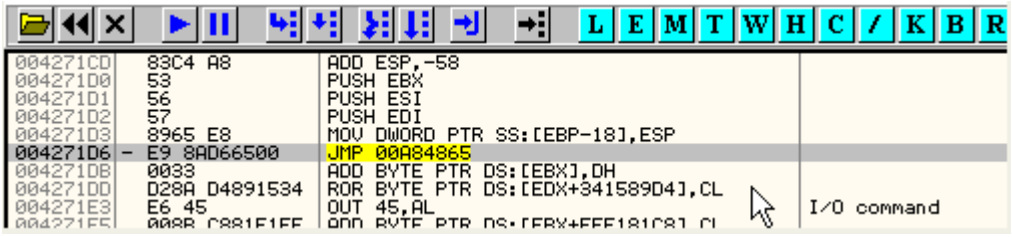
我们可以看到这里 IMP REC 已经把 IAT 都修复了,各个 DLL 中的 IAT 项中间的间隔也由垃圾数据替换为了零。但是为什么运行起来还报错呢?

这就是我们接下来要讨论的一个话题,在修复了 IAT,以及 stolen bytes 以后,很多壳还会有 AntiDump。

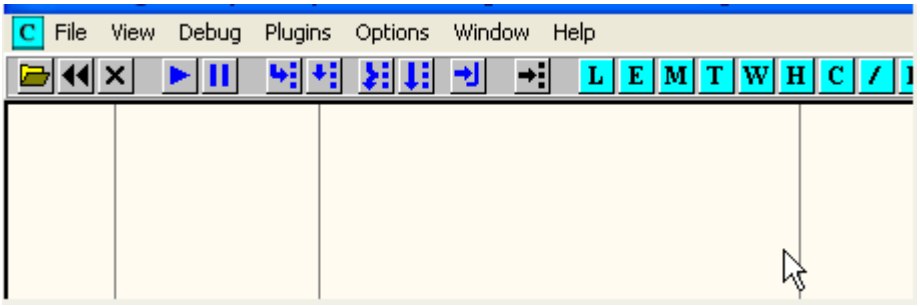
有很多类型的 AntiDump,简单一点的 AntiDump 会校验映像的大小或者区段的个数,我们这个例子的 AntiDump 是壳创建了几个区段,在运行时会对原程序做一些处理,如果 dump 出来的程序不包含这些区段,那么程序就不能正常运行,我们来到入口点处。



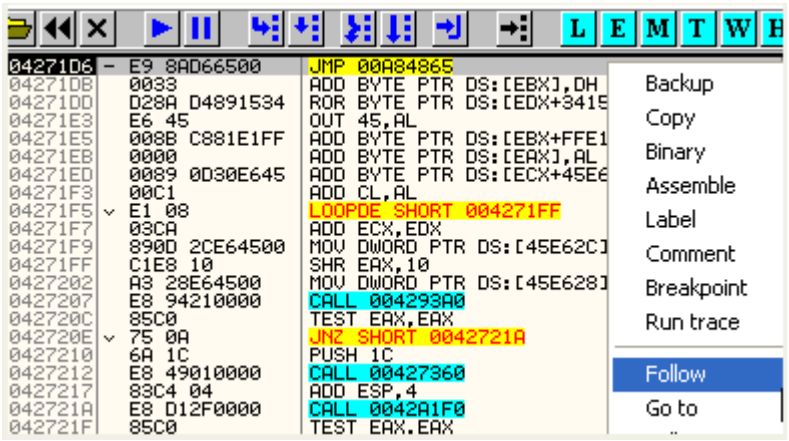
当程序运行到 4271D6 处的 JMP 指令处的时候就会报错。



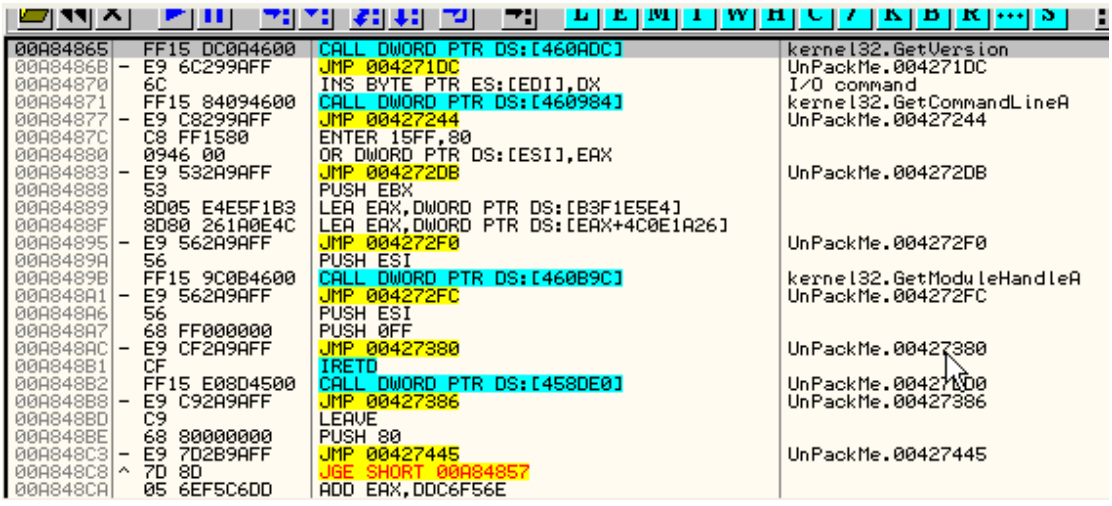
如果我们按 F7 键,会跳转到一个不存在的区段。



我们来看一看未脱壳的程序,会发现将跳转到壳运行时创建的一个区段中。



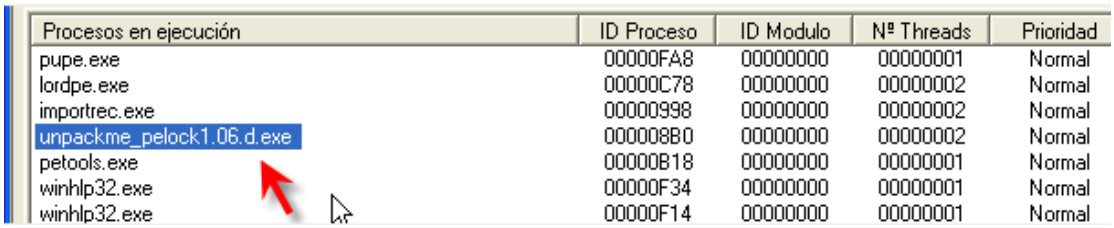
单击鼠标右键选择 Follow,就能够跟随到壳创建的区段中。



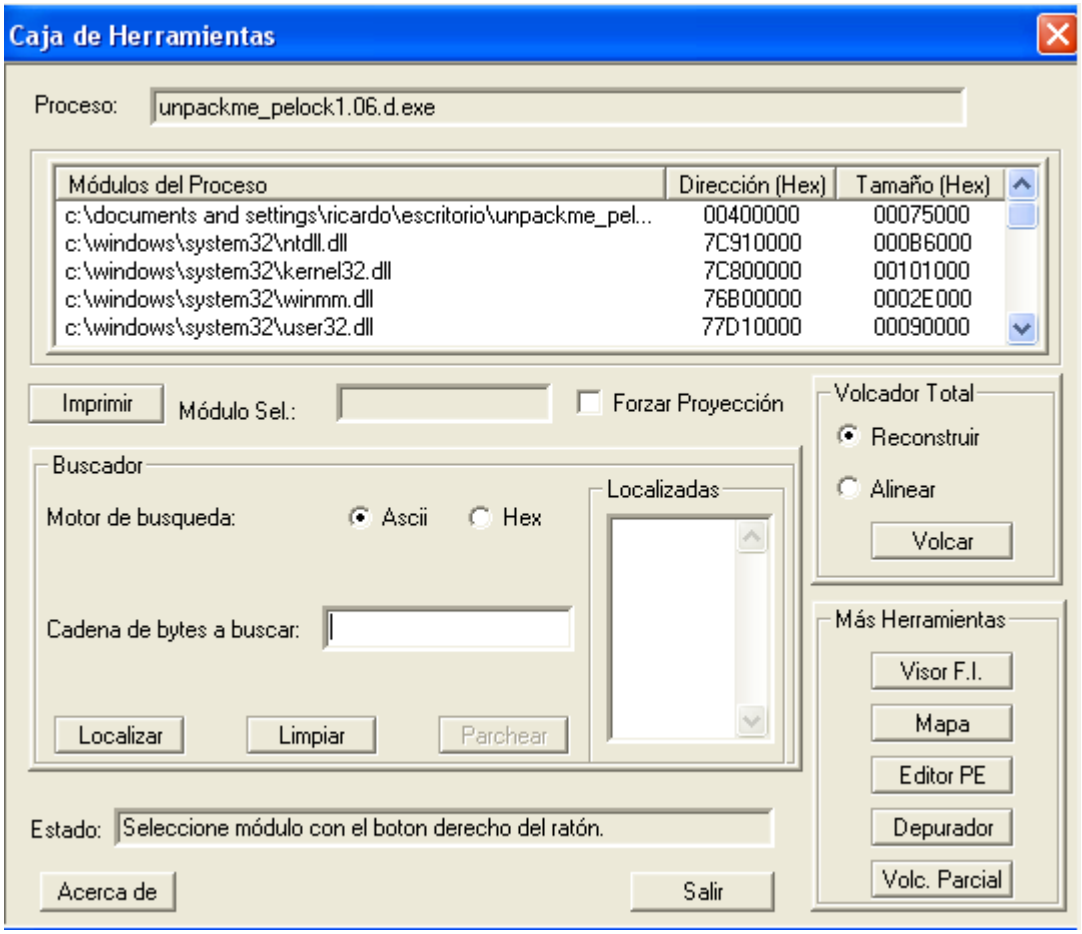
这里我们可以看到是调用了一个 API 函数,接着返回到主程序代码段,有很多方法可以解决这个问题,最简单的方法就是给 dump 出来的文件添加一个相同的区段,我们来看看。

首先添加报错的这个区段,如果添加了以后还报错,再次添加其他缺失的区段。

这里我们需要用到 PUPE 这款工具来 dump 缺失的区段。



选中 Pelock 所在的进程,单击鼠标右键选择 BOX OF TOOLS。

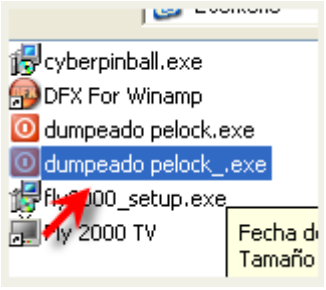


单击 Mapa(即 MAP)按钮打开区段列表窗口,定位到报错的区段。

我们还记得报错的区段的起始地址为 A80000,我们一起来看一看。

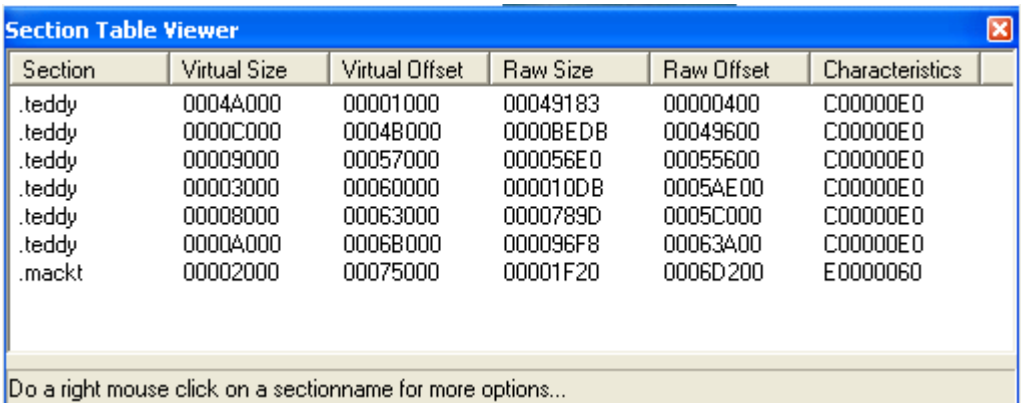
我们单击 Volcar(PS:西班牙语译为 DUMP)按钮,将其命名为 seccion(PS:西班牙语译为 section)。

现在我们打开 PEditor,将该区段添加到 dump 文件中。



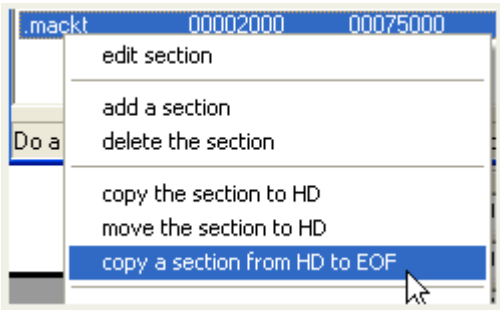
通过 PEditor 打开 dump 出来的文件。

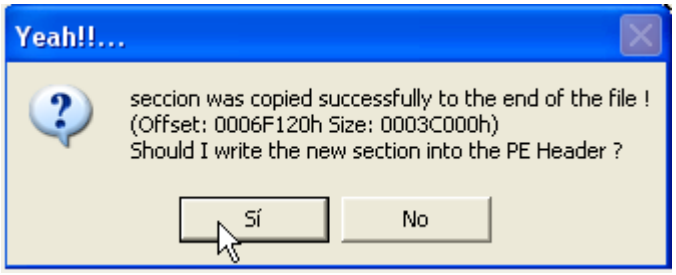
单击 sections 按钮。

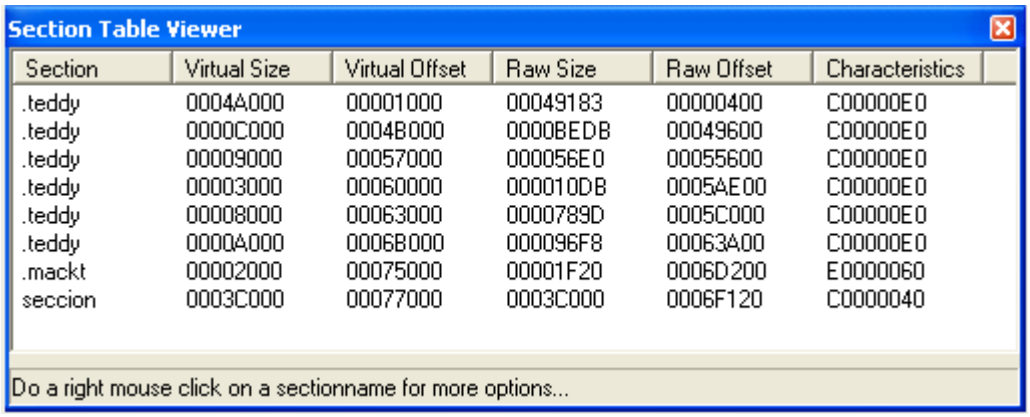


这里我们可以看到 IMP REC 在为了修复 IAT 也给该程序添加了一个区段,名称为.mackt,我们将缺失的区段添加到最后。

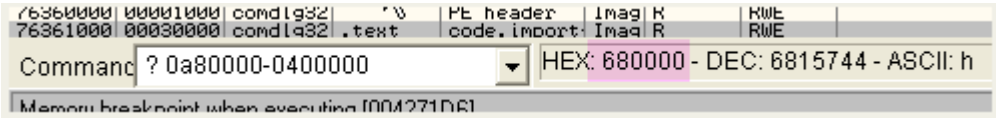
在.mackt 这个区段上面单击鼠标右键选择 copy a section from HD to EOF。接着我们将新添加区段的起始地址修改为 A80000。



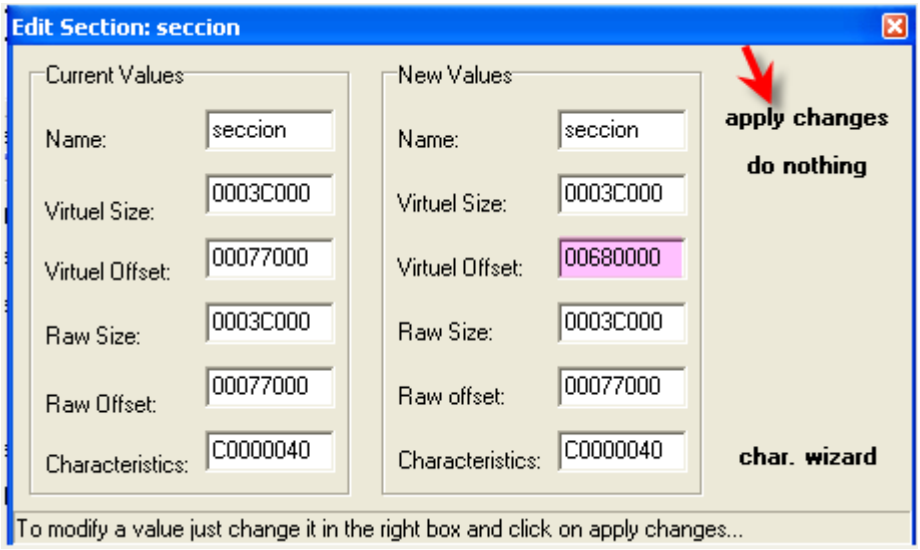




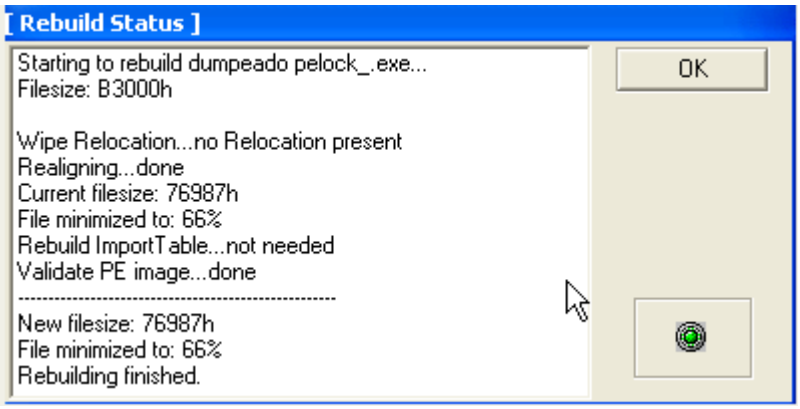
我们来计算一下该新区段的相对虚拟地址 = A80000 - 400000 = 680000。

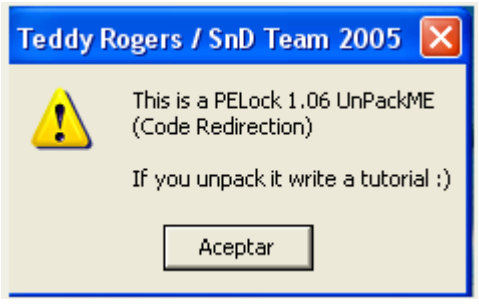


也就是说新区段的起始地址 RVA = 680000。

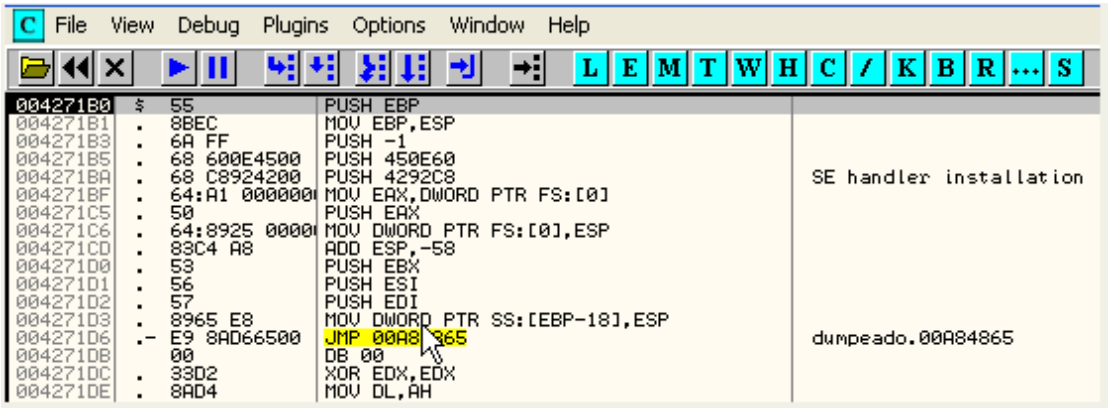


下面我们通过 LordPe 来重建该 PE 就大功告成了。

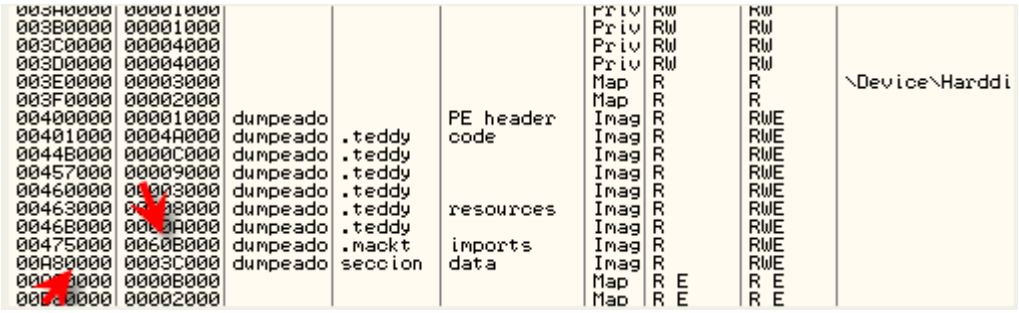




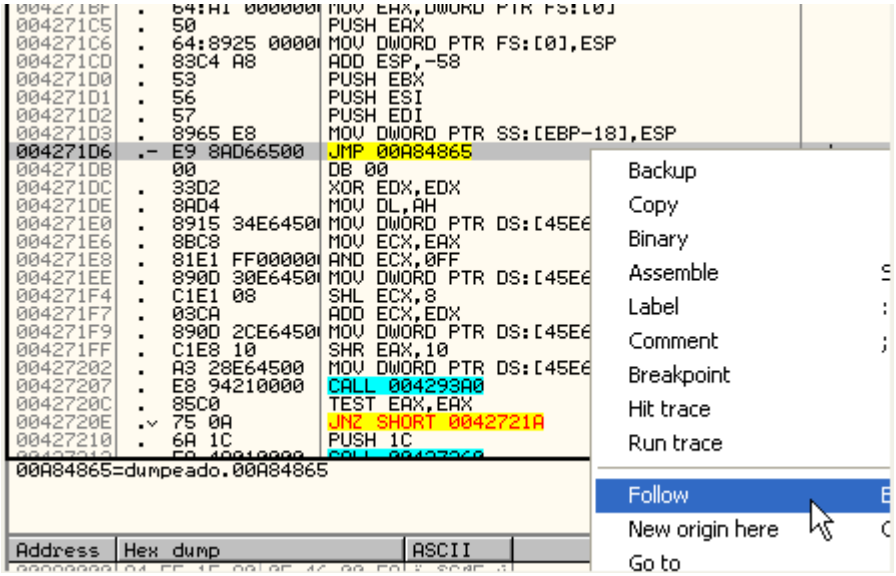
正常运行,我们用 OllyDbg 打开它。



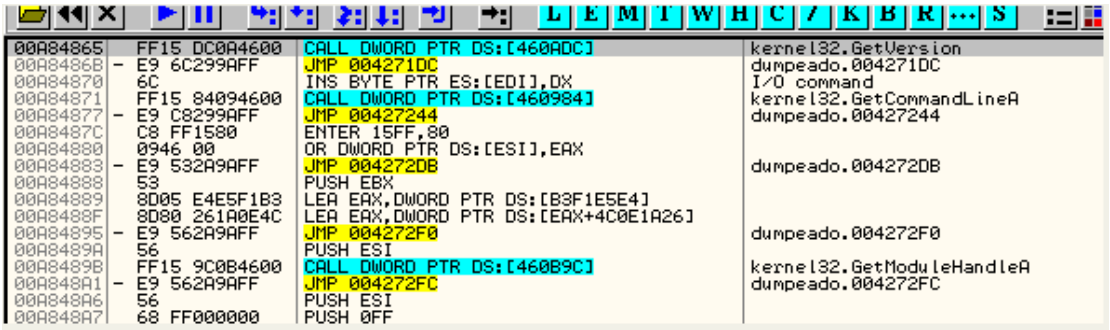
查看一下区段列表。



我们可以看到起始地址为 A80000 新添加的区段。



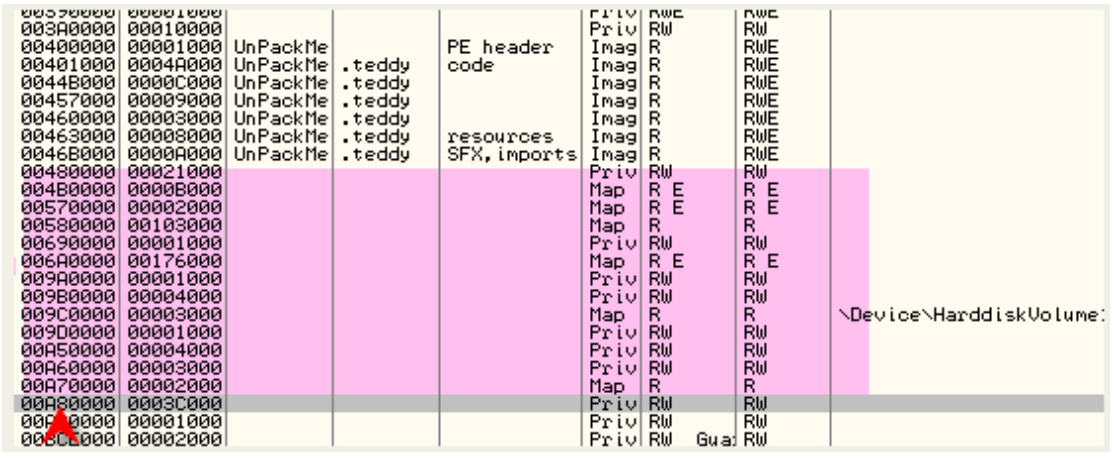
我们定位到 4271D6 地址处 JMP 指令,单击鼠标右键选择 Follow。



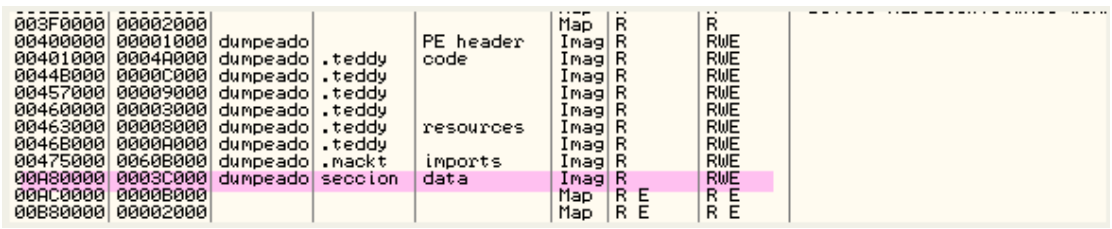
跳转到了缺失的区段中。

我们来对比一下未脱壳和脱完壳并修复后的区段。

未脱壳的区段:



脱完壳修复后的区段:



本章就到这里。(PS:后面一点总结部分,作者讲的有误,这里就略去了)