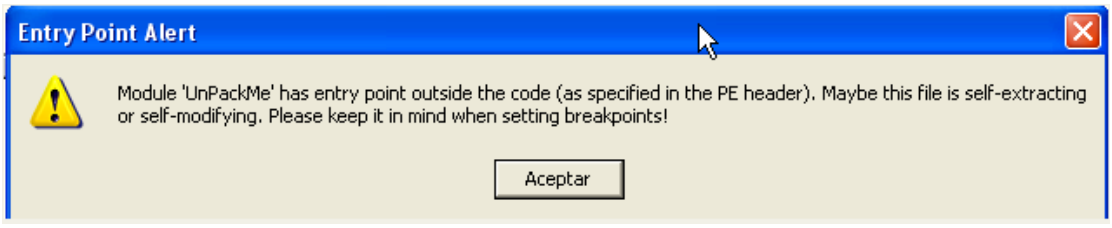
第三十五章-手脱 ASPack V2.12

本章我们继续介绍脱壳,稍微增加一点难度。

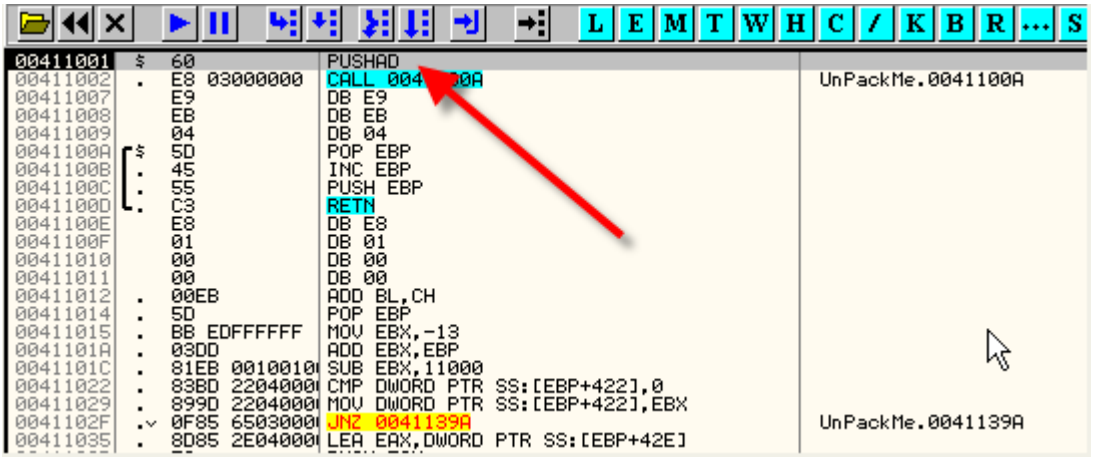
我们要脱的壳是 ASPack,比 UPX 稍微复杂那么一点点,拿 UnPackMe_ASPack2.12.exe 作为实验对象,这个程序我们在第三十二章介绍 OEP 的时候遇到过,大家应该还记得吧。

Dump 的话我们用 OD 的插件 OllyDump 来完成,大家将其放到 OD 的插件目录下。

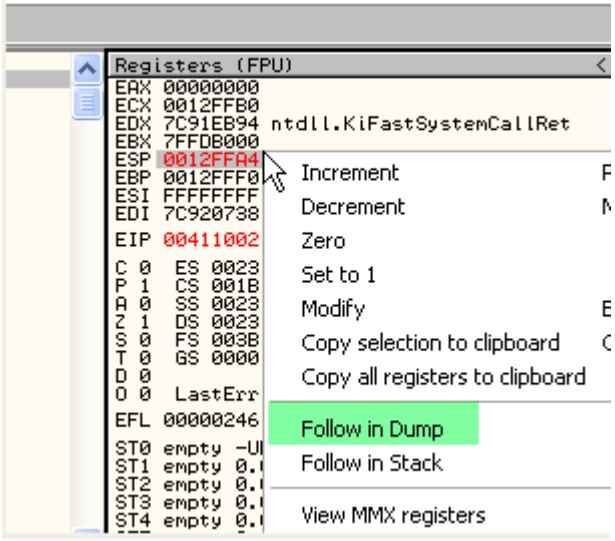
将 OD 的反反调试插件配置好,然后加载。



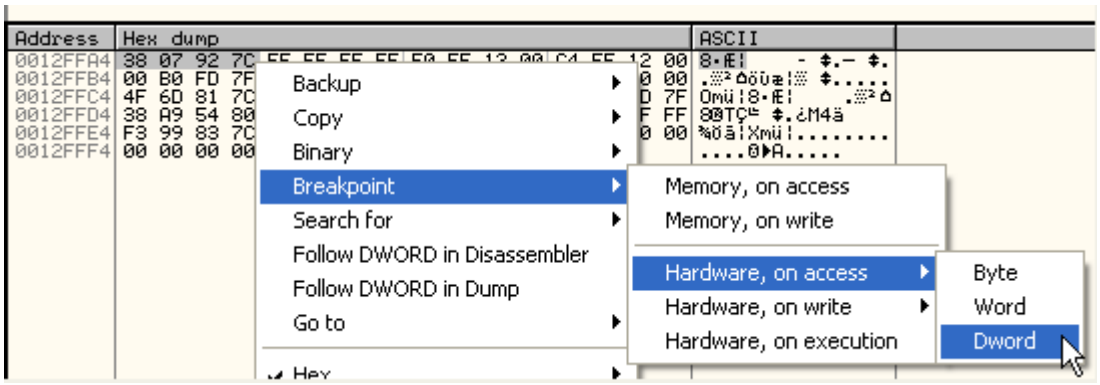
OD 提示该程序入口点位于代码段之外,对于大部分加壳程序 OD 都会弹出此警告窗口,大家不必大惊小怪。



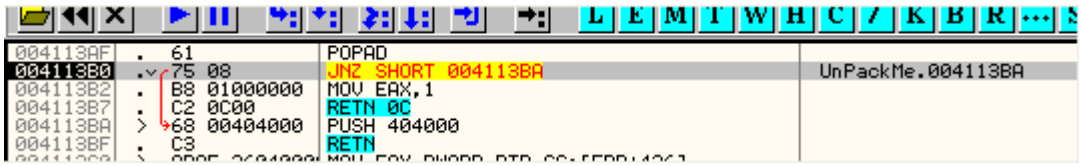
我们可以看到第一条指令是 PUSHAD,按 F7 键执行 PUSHAD。



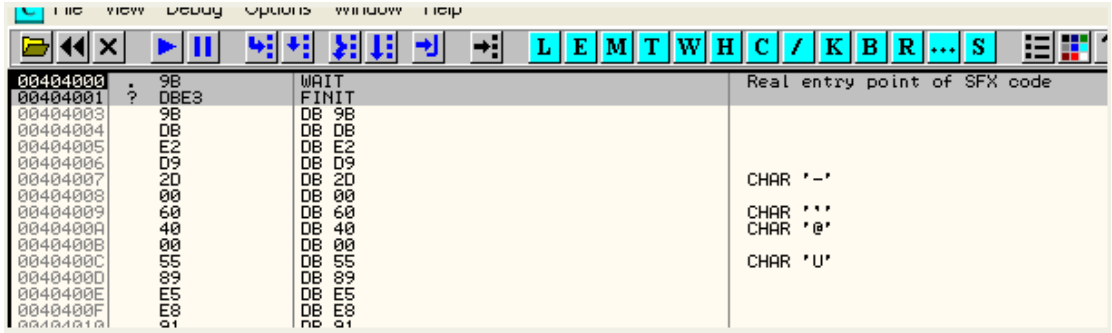
在 ESP 寄存器值上面单击鼠标右键选择-Follow in Dump,就可以在数据窗口中定位到刚刚通过 PUSHAD 指令保存的寄存器环境了,选中前 4 个字节,单击鼠标右键选择-Breakpoint-Hardware,on access-Dword,这样就可以给这 4 个字节设置硬件访问断点了。



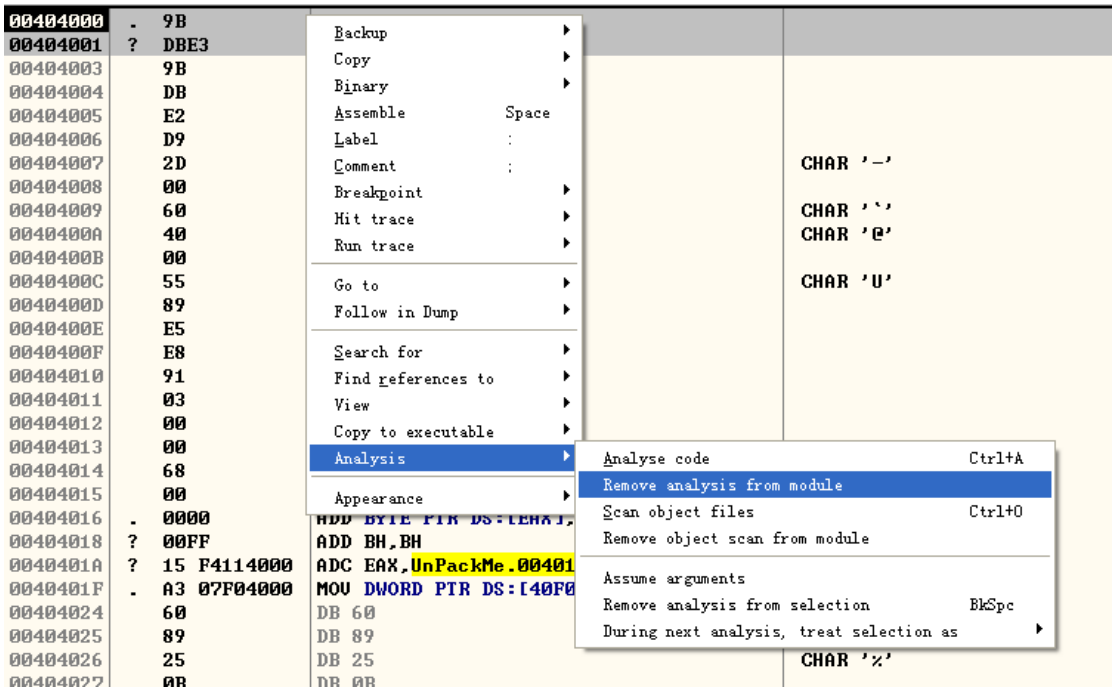
按 F9 键运行起来。

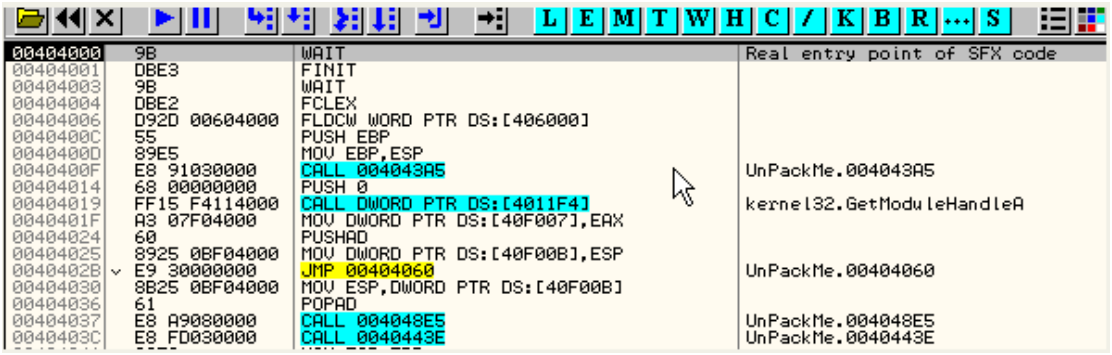


断在了 POPAD 指令的下一行,我们直接按 F7 键单步跟踪到 OEP 处。

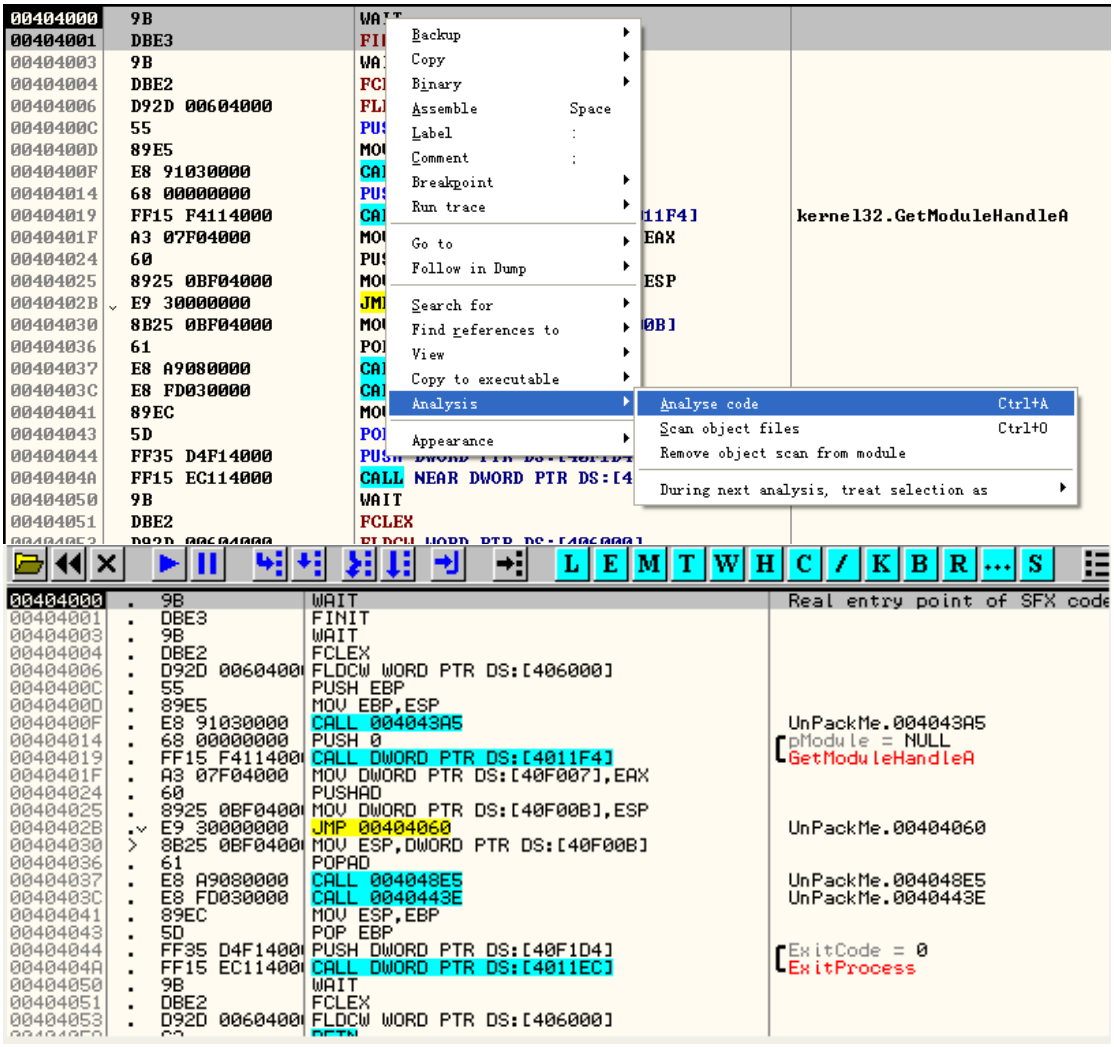


这里我们到了 OEP 处,OD 这里解析有误,将代码解析为数据了,我们在反汇编窗口中单击鼠标右键选择-Analysis-Remove analysis from module,删除掉 OD 的分析结果,这样就能正常解析了。

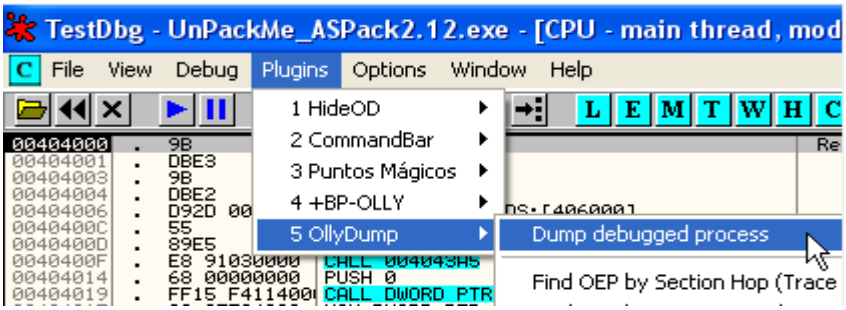


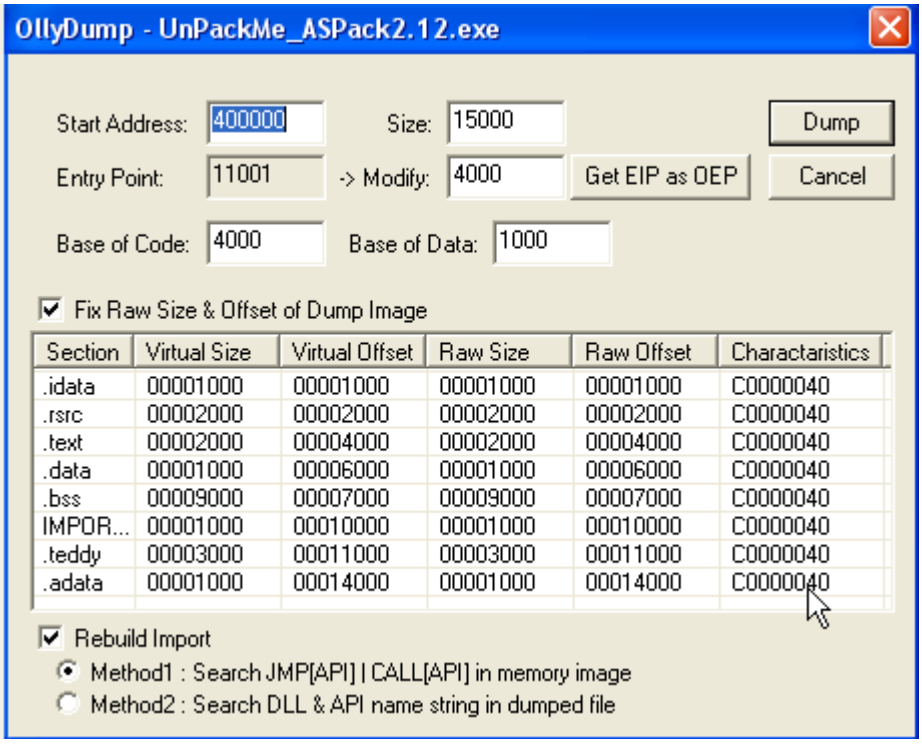


我们可以看到虽然已经被解析成代码了,但是解析的还不够完整,我们还需要解析一次,继续单击鼠标右键选择-Analysis-Analyse code。



现在我们可以对该进程进行 dump 了,在菜单栏中找到 OllyDump 插件。

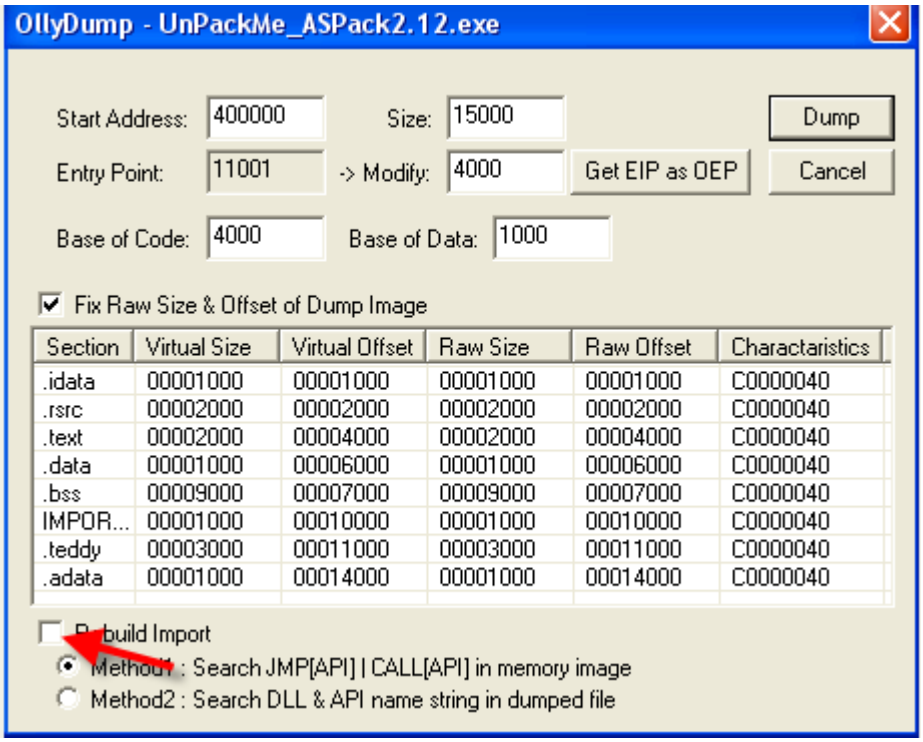




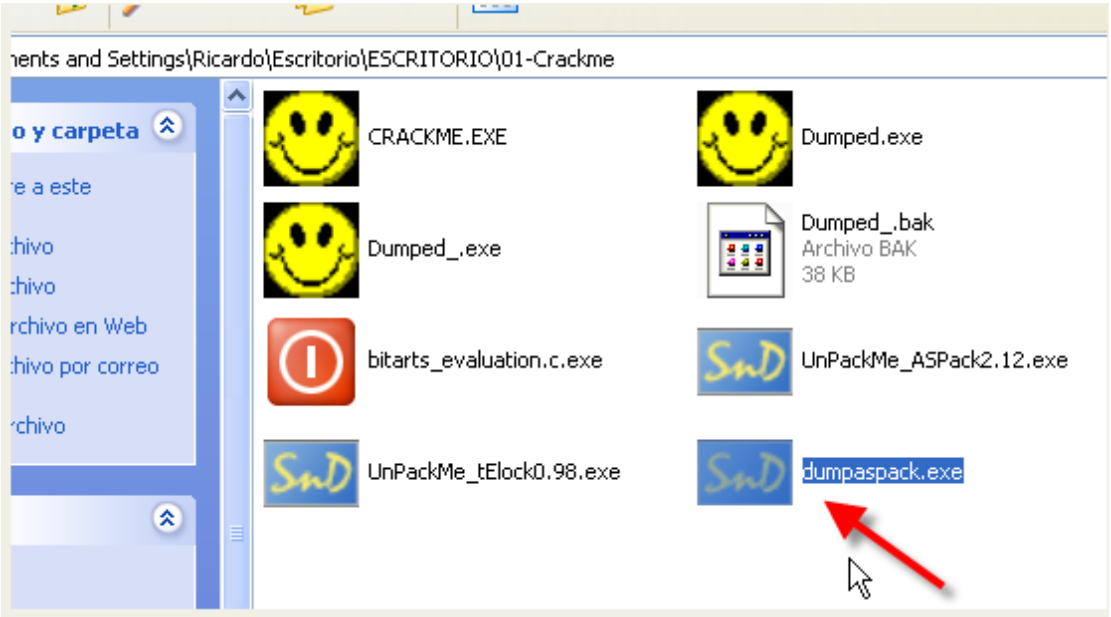
该插件的窗口的弹了出来,有一些选项可供我们修改,我们可以对 Base of Code 进行修改,这里 Base of Code = 4000(RVA),该选项相当于对代码段进行了指定,不需要像上一章那样在数据窗口中的 PE 头中去修改。我们应该还记得 ASPack 加壳程序的原程序代码段并不是第一个区段,而是第三个区段,4000(RVA),即 404000(VA),OEP 也是 404000,刚好在代码段中,所以 Base of Code 这一项我们不需要修改。

我们可以看到下面还有一个选项 Rebuild Import(重建 IAT),跟 IMP REC 的功能类似,提供了两种方式来重建 IAT,这个选项对于一些简单的壳还是有效的,大家可以试一试。

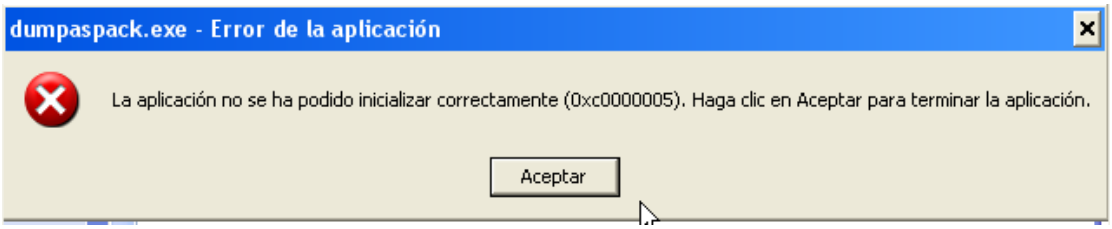
我们这里就不使用这个功能了,去掉 Rebuild Import 的对勾,为了保险起见,我们还是使用 IMP REC 来修复 IAT。



单击 dump 按钮。



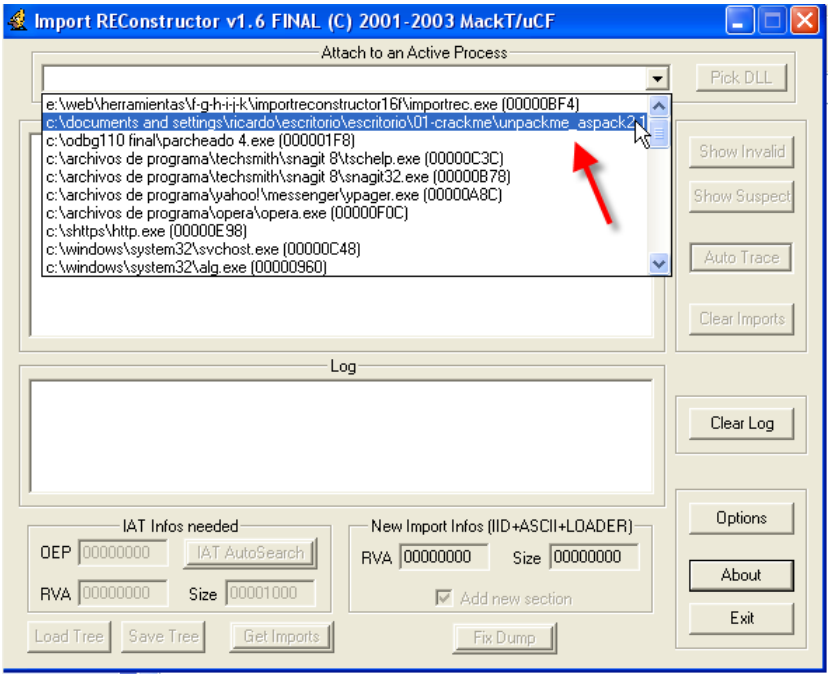
这里我们将 dump 出来的文件命名为 dumpaspack.exe。这里我们没有修复 IAT 直接运行起来,看看会不会报错。



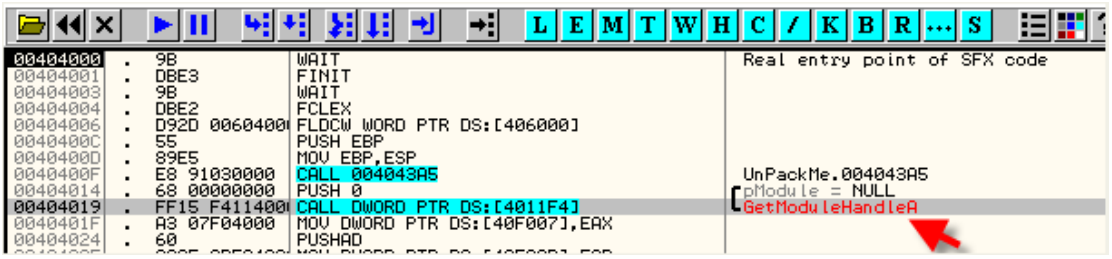
提示无效的 win32 程序,我们需要修复 IAT,打开 IMP REC,定位到 UnPackMe_ASPack2.12 所在的进程,当前该进程停在了 OEP 处。我们回到 OD 中,我们需要定位 IAT 的起始地址以及大小,还有 OEP 的值。

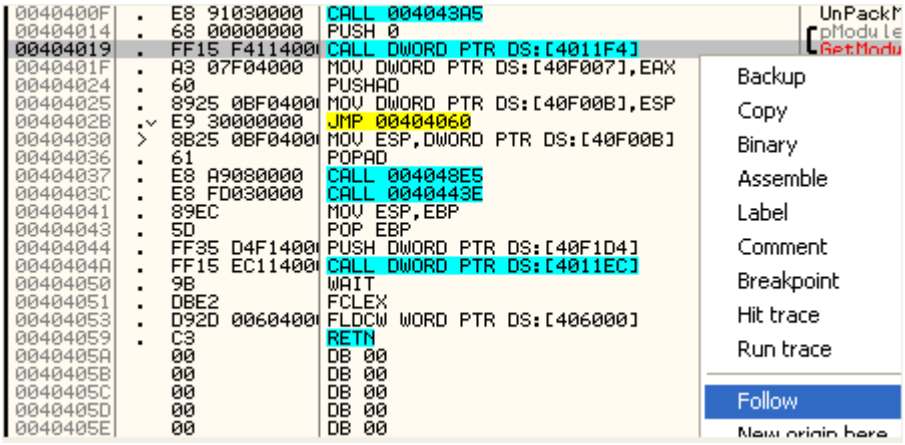
现在 OEP 的值我们知道了,等于 404000,IMP REC 要求的是 RVA(相对虚拟地址),404000(VA:虚拟地址) - 400000(映像基址) = 4000(RVA)。

接下来我们需要定位 IAT 的起始地址以及大小,我们随便找一个 API 函数的调用处,好,OEP 的下面正好有一处 CALL GetModuleHandleA。



选中 CALL GetModuleHandleA 这一行,单击鼠标右键选择-Follow。

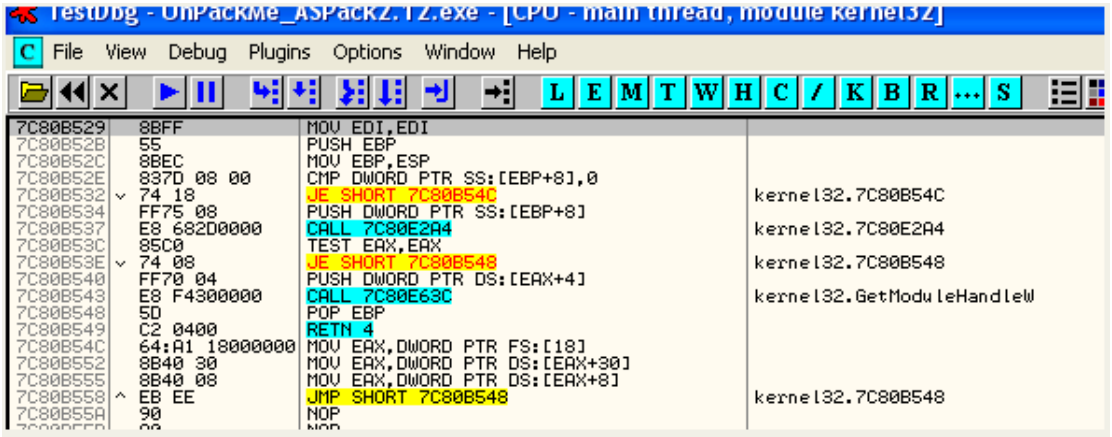




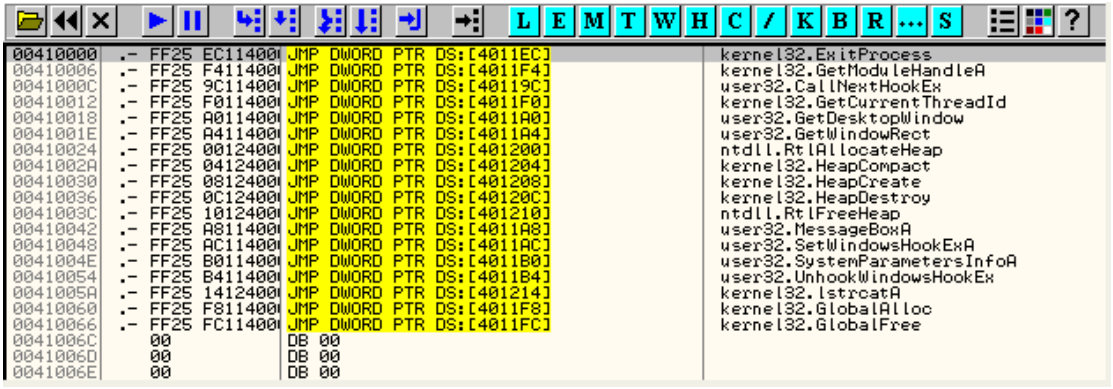
这里我们可以看到直接来到了 GetModuleHandleA 的入口点处,并没有间接跳转(并不是所有的程序调用 API 函数都是通过间接跳转来实现的,该程序就没有使用间接跳转)。

这里是通过一个间接 CALL 来调用 API 函数的。

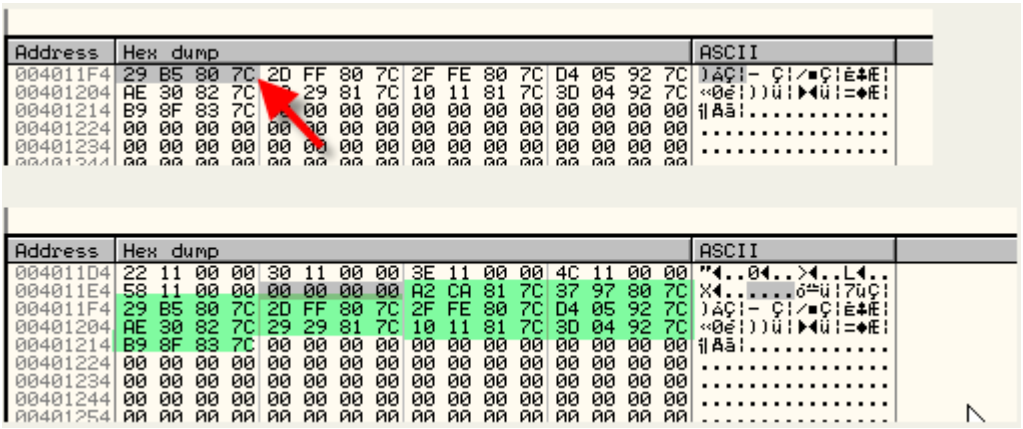
显然,4011F4 是 IAT 其中的一项,该内存单元中保存了 GetModuleHandleA 的入口地址。



我们直接搜索二进制串 FF 25,看看能不能定位到跳转表。



我们可以看到定位到了跳转表,接着我们在数据窗口中定位到 IAT。



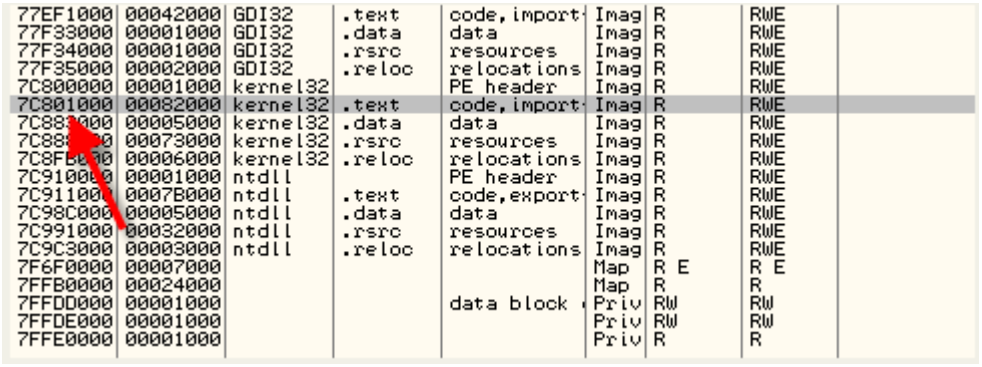
Address	Hex dump	ASCII
004011F4	29 B5 80 7C 2D FF 80 7C 2F FE 80 7C D4 05 92 7C	)µÇ¦- ÿ¦/■Ç¦Ô♣Æ¦
00401204	AE 30 82 7C 29 29 81 7C 10 11 81 7C 3D 04 92 7C	«0é¦))ü¦►◄ü¦=♦Æ¦
00401214	B9 8F 83 7C 00 00 00 00 00 00 00 00 00 00 00 00	╣Åâ¦............
00401224	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401234	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401244	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Address	Hex dump	ASCII
004011D4	22 11 00 00 30 11 00 00 3E 11 00 00 4C 11 00 00	"◄..0◄..>◄..L◄..
004011E4	58 11 00 00 00 00 00 00 A2 CA 81 7C 37 97 80 7C	X◄.....ó╩ü¦7ùÇ¦
004011F4	29 B5 80 7C 2D FF 80 7C 2F FE 80 7C D4 05 92 7C	)µÇ¦- ÿ¦/■Ç¦Ô♣Æ¦
00401204	AE 30 82 7C 29 29 81 7C 10 11 81 7C 3D 04 92 7C	«0é¦))ü¦►◄ü¦=♦Æ¦
00401214	B9 8F 83 7C 00 00 00 00 00 00 00 00 00 00 00 00	╣Åâ¦............
00401224	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401234	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401244	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401254	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

我们将最后一个 DLL 中的 IAT 项用绿色标注出来了,地址是 7C8XXXXX 的形式,单击工具栏中的 M 按钮,在区段列表窗口中看看这类地址是属于哪个 DLL。

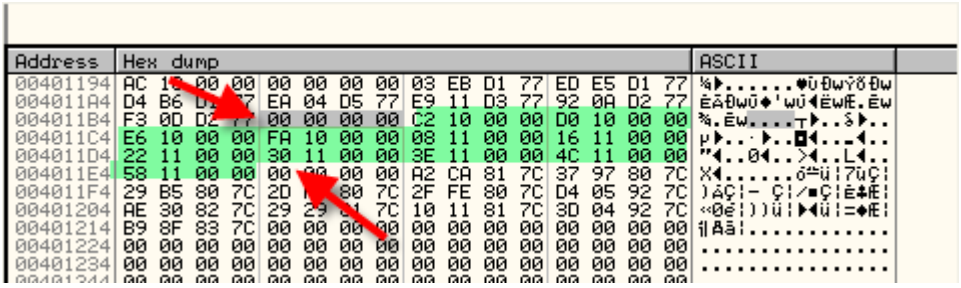
这里我们可以看到这些地址是属于 kernel32.dll 的代码段。

这里 IAT 中的最后一个元素起始地址为 401214,即 401218 是 IAT 的结束位置,现在我们再来看看 IAT 的起始地址是多少。



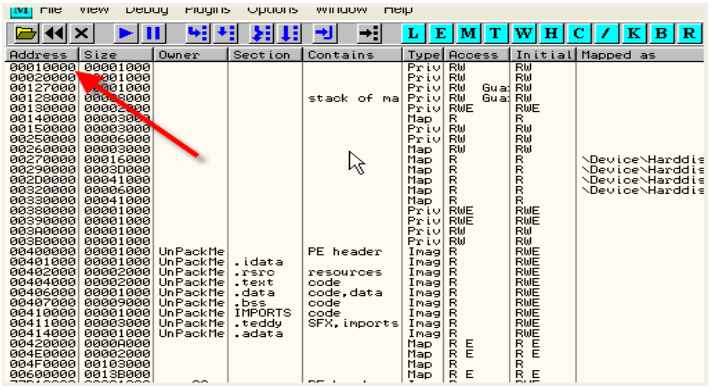
77EF1000	00042000	GDI32	.text	code,import	Imag	R	RWE
77F33000	00001000	GDI32	.data	data	Imag	R	RWE
77F34000	00001000	GDI32	.rsrc	resources	Imag	R	RWE
77F35000	00002000	GDI32	.reloc	relocations	Imag	R	RWE
7C800000	00001000	kernel32		PE header	Imag	R	RWE
7C801000	00082000	kernel32	.text	code,import	Imag	R	RWE
7C883000	00005000	kernel32	.data	data	Imag	R	RWE
7C888000	00073000	kernel32	.rsrc	resources	Imag	R	RWE
7C8FB000	00006000	kernel32	.reloc	relocations	Imag	R	RWE
7C910000	00001000	ntdll		PE header	Imag	R	RWE
7C911000	0007B000	ntdll	.text	code,export	Imag	R	RWE
7C98C000	00005000	ntdll	.data	data	Imag	R	RWE
7C991000	00032000	ntdll	.rsrc	resources	Imag	R	RWE
7C9C3000	00003000	ntdll	.reloc	relocations	Imag	R	RWE
7F6F0000	00007000				Map	R E	R E
7FFB0000	00024000				Map	R	R
7FFDD000	00001000			data block	Priv	RW	RW
7FFDE000	00001000				Priv	RW	RW
7FFE0000	00001000				Priv	R	R

每个 DLL 的 IAT 项都是以零隔开的。



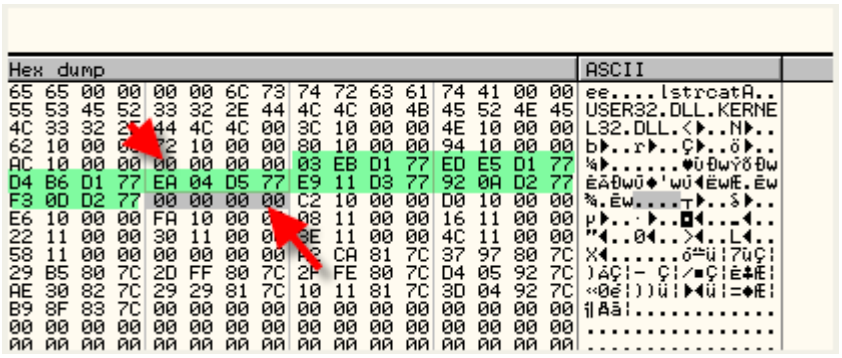
Address	Hex dump	ASCII
00401194	AC 10 00 00 00 00 00 00 03 EB D1 77 ED E5 D1 77	¼►.....♦ûÐwýõÐw
004011A4	D4 B6 D1 77 EA 04 D5 77 E9 11 D3 77 92 0A D2 77	èÃÐwÜ♦'wÚ◄ÉwÆ.Èw
004011B4	F3 0D D2 77 00 00 00 00 C2 10 00 00 D0 10 00 00	¾.ÈwÂ...┬►..ð►..
004011C4	E6 10 00 00 FA 10 00 00 08 11 00 00 16 11 00 00	µ►..·►..◘◄.▬◄..
004011D4	22 11 00 00 30 11 00 00 3E 11 00 00 4C 11 00 00	"◄..0◄..>◄..L◄..
004011E4	58 11 00 00 00 00 00 00 A2 CA 81 7C 37 97 80 7C	X◄.....ó╩ü¦7ùÇ¦
004011F4	29 B5 80 7C 2D FF 80 7C 2F FE 80 7C D4 05 92 7C	)µÇ¦- ÿ¦/■Ç¦Ô♣Æ¦
00401204	AE 30 82 7C 29 29 81 7C 10 11 81 7C 3D 04 92 7C	«0é¦))ü¦►◄ü¦=♦Æ¦
00401214	B9 8F 83 7C 00 00 00 00 00 00 00 00 00 00 00 00	╣Åâ¦............
00401224	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401234	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00401244	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

这里我们可以看到这部分绿色标注出来的数值为 10XX 或者 11XX 的形式,明显不属于任何一个 DLL,而且这些数值比当前进程空间中分配内存单元中的最小地址还要小。

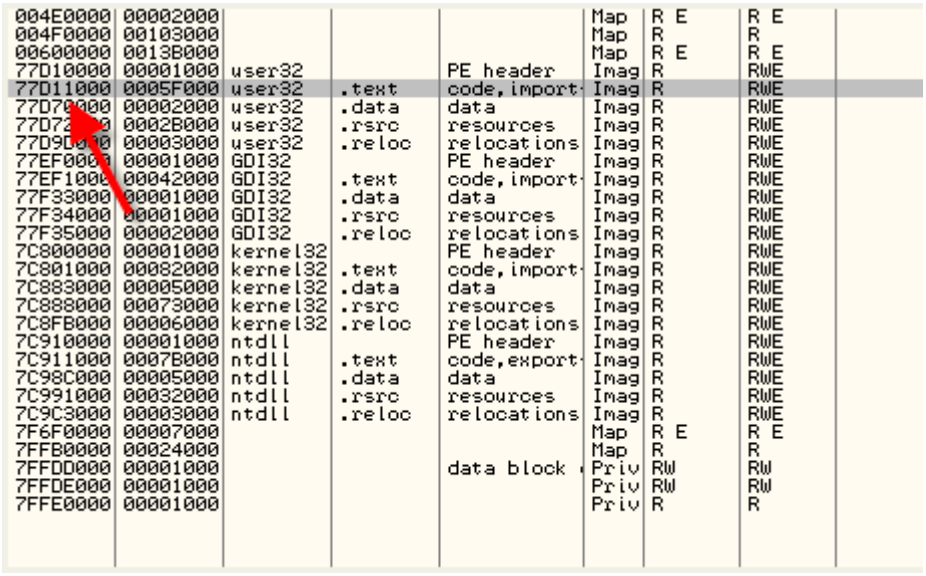


Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000				Priv	RW	RW	
00020000	00001000				Priv	RW	RW	
00127000	00001000				Priv	RW Guar	RW	
00128000	00008000			stack of ma	Priv	RW Guar	RW	
00130000	00002000				Priv	RWE	RWE	
00140000	0000300				Map	R	R	
00150000	00003000				Priv	RW	RW	
00250000	00006000				Priv	RW	RW	
00260000	00003000				Map	RW	RW	
00270000	00016000				Map	R	R	\Device\Harddis
00290000	0003D000				Map	R	R	\Device\Harddis
002D0000	00041000				Map	R	R	\Device\Harddis
00320000	00006000				Map	R	R	\Device\Harddis
00330000	00041000				Map	R	R	
00380000	00001000				Priv	RWE	RWE	
00390000	00001000				Priv	RWE	RWE	
003A0000	00001000				Priv	RW	RW	
003B0000	00001000				Priv	RW	RW	
00400000	00001000	UnPackMe		PE header	Imag	R	RWE	
00401000	00001000	UnPackMe	.idata		Imag	R	RWE	
00402000	00002000	UnPackMe	.rsrc	resources	Imag	R	RWE	
00404000	00002000	UnPackMe	.text	code	Imag	R	RWE	
00406000	00001000	UnPackMe	.data	code,data	Imag	R	RWE	
00407000	00009000	UnPackMe	.bss	code	Imag	R	RWE	
00410000	00001000	UnPackMe	IMPORTS	code	Imag	R	RWE	
00411000	00003000	UnPackMe	.teddy	SFX,imports	Imag	R	RWE	
00414000	00001000	UnPackMe	.adata		Imag	R	RWE	
00420000	0000A000				Map	R E	R E	
004E0000	00002000				Map	R E	R E	
004F0000	00103000				Map	R	R	
00600000	0013B000				Map	R E	R E	

所以这些数值不属于任何一个 DLL,也不属于任何一个区段,有可能是壳存放的一些垃圾数据,我们继续往下拉。

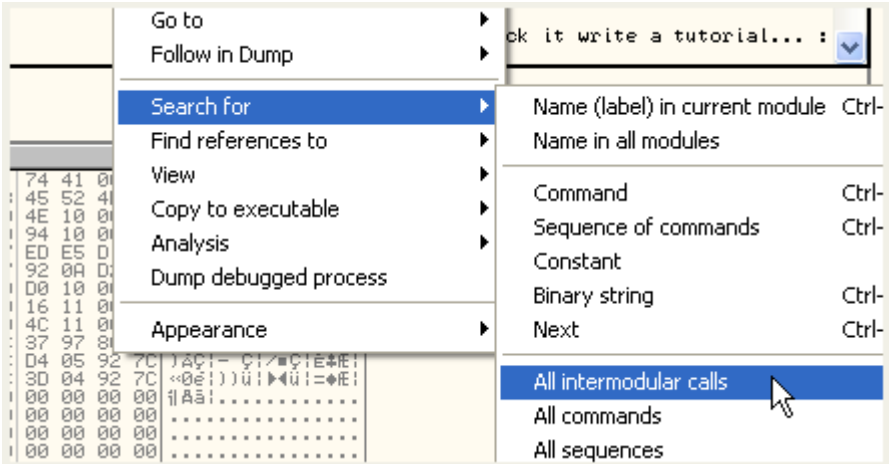


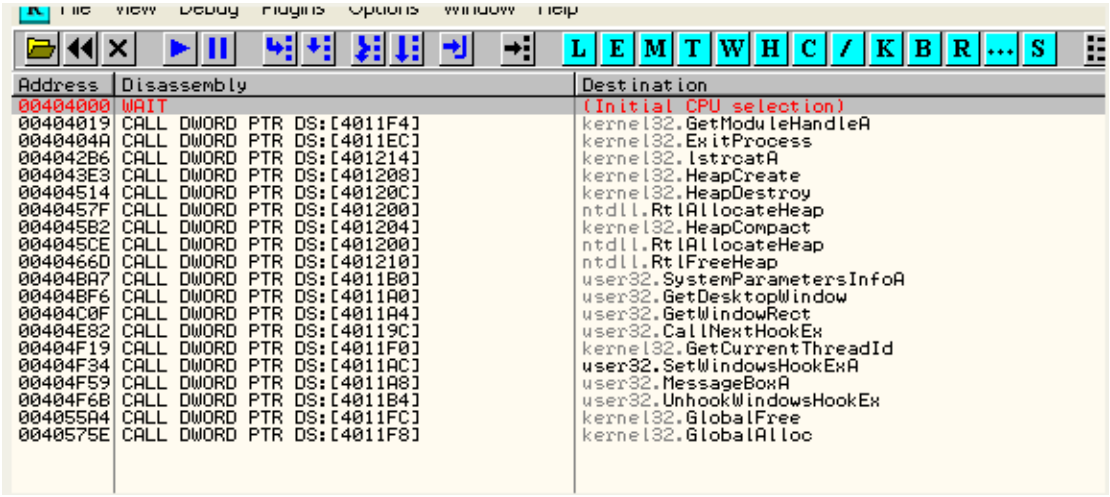
这部分地址是 77DXXXXX 的形式,我们在区段列表窗口中看看这些地址属于哪个 DLL。



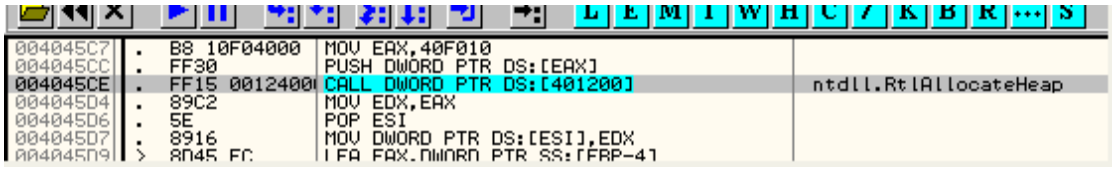
属于 user32.dll。

区段列表窗口中还有些其他的 DLL,如:GDI32.dll,NTDLL.dll,原程序并没有用到,这几个 DLL 有可能是壳加载的,我们在反汇编窗口中单击鼠标右键选择-Search for-All intermodular calls 看看该程序调用了哪些 DLL 中的 API 函数。

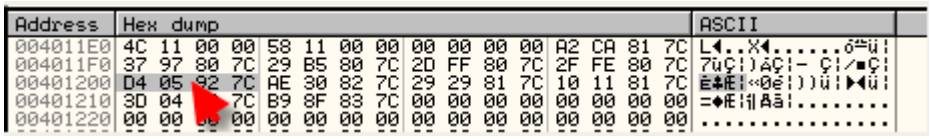




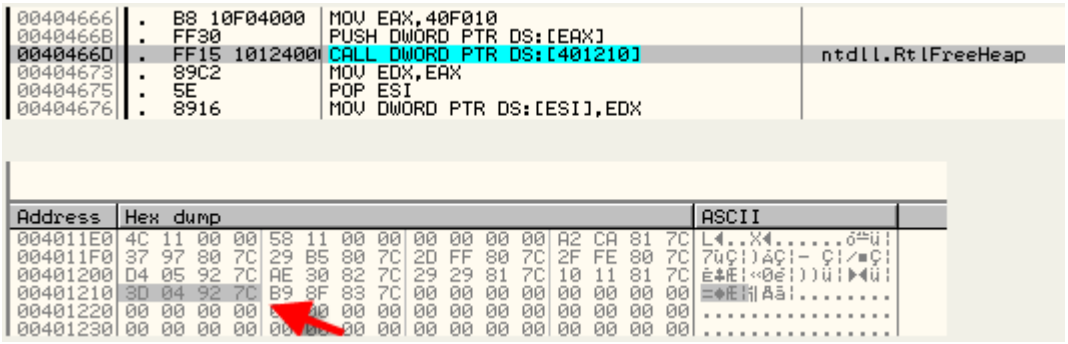
这里我们可以看到原程序总共调用了 3 个 DLL 的 API 函数,我们只找到了两个 DLL 的 IAT 项,NTDLL 的 IAT 项呢?我们随便找一个 NTDLL 的 API 函数调用处。



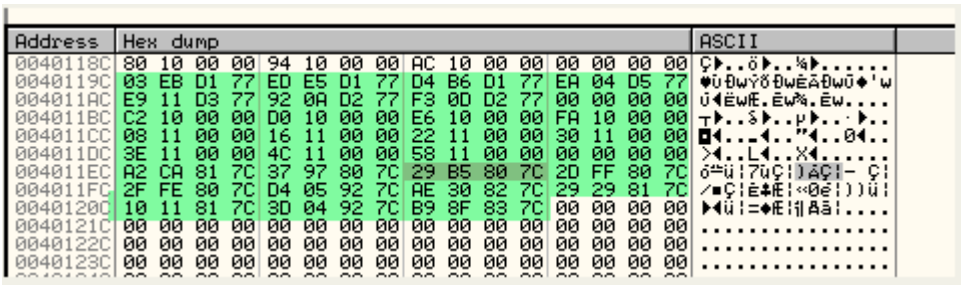
这一项的地址为 401200,和 Kernel32.dll 的 IAT 项混在了一起。



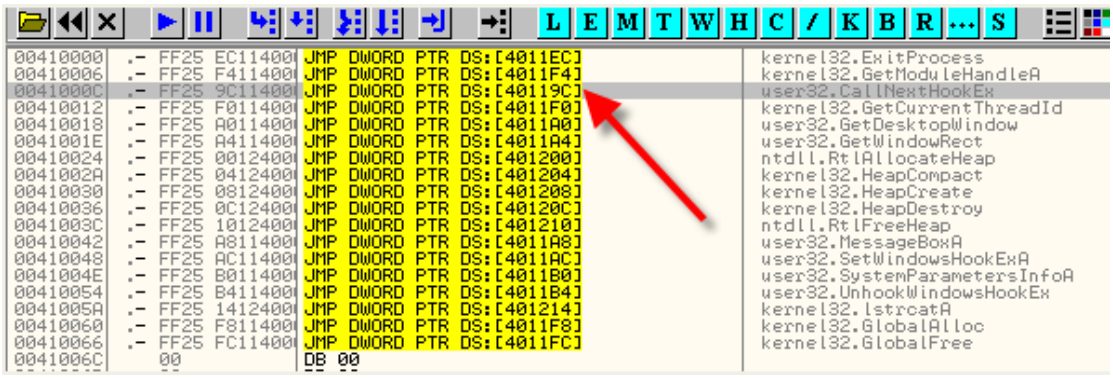
401210 这一项也被混在了 Kernel32.dll 的 IAT 项中。



好,现在我们来看看 IAT 的起始位置在哪里,IAT 的所有元素这里我们用绿色标注出来了。



IAT 的起始地址为 40119C,跟跳转表中的最小地址一致。



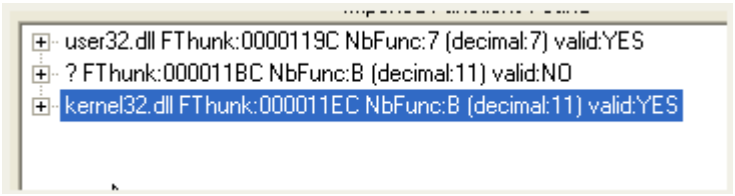
好了,现在我们有了以下三条数据:

OEP = 4000 (RVA)

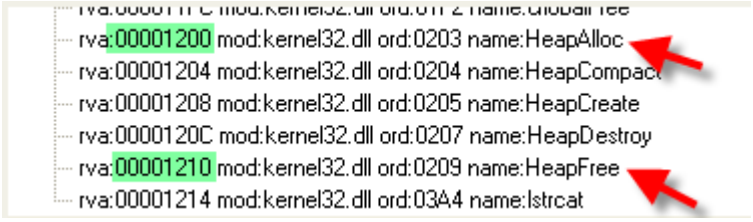
IAT 的起始地址 = 119C (RVA)

IAT 的大小 = 401218 - 40119C = 7C。

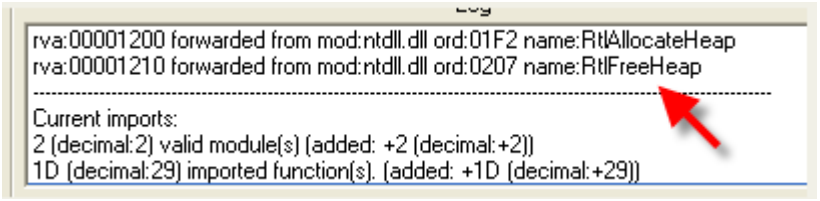
我们将这三个值填到 IMP REC 中去,单击 Get Imports,获取 IAT 项。



中间一条记录是垃圾数据,我们单击 kernel32.dll 这条记录左边的+号,可以看到 401200 和 401210 这两项和 kernel32.dll 中的 IAT 项混在了一起。

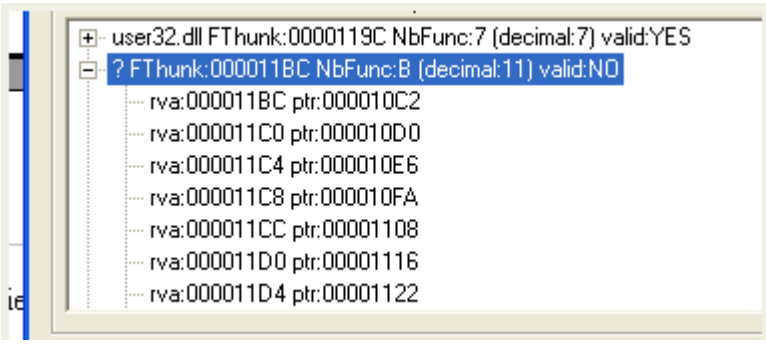


可以看到这里 ntdll.dll 中分配内存空间的两个函数用 kernel32.dll 中两个类似的函数 HeapAlloc,HeapFree 替换掉了。

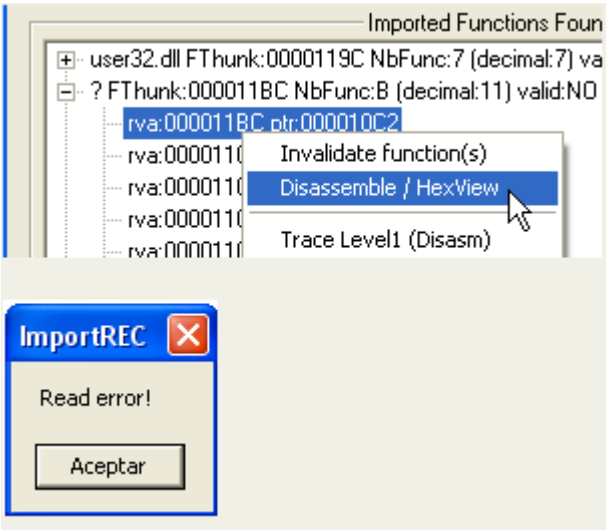


这里日志信息中也提示说这两个函数跟 ntdll.dll 中的 RtlAllocateHeap,RtlFreeHeap 完成的功能类似,壳也可以对这些 IAT 项进行修改和混淆。

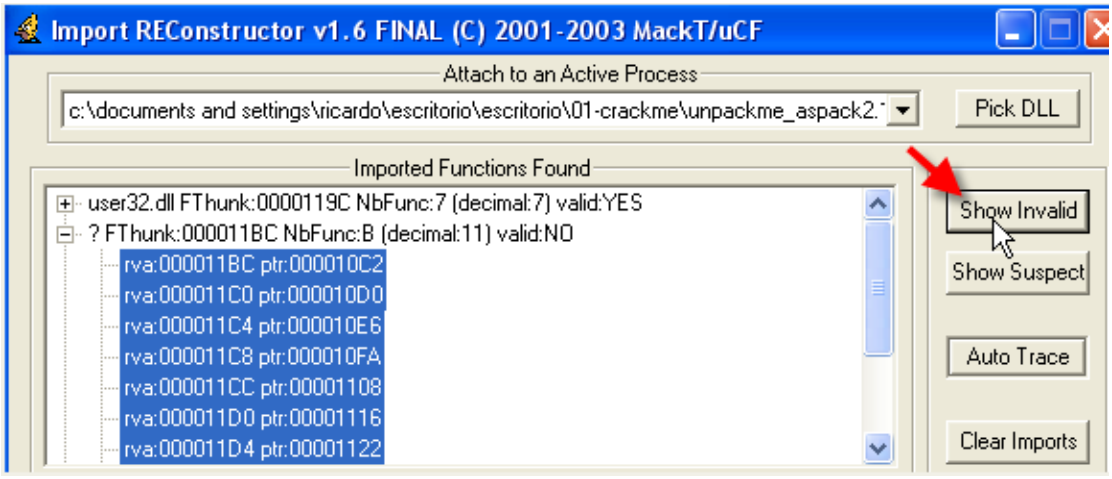
好了,现在我们需要剔除掉垃圾数据,即 valid 显示为 NO 的项(无效数据),我们单击左边的+号将其展开。



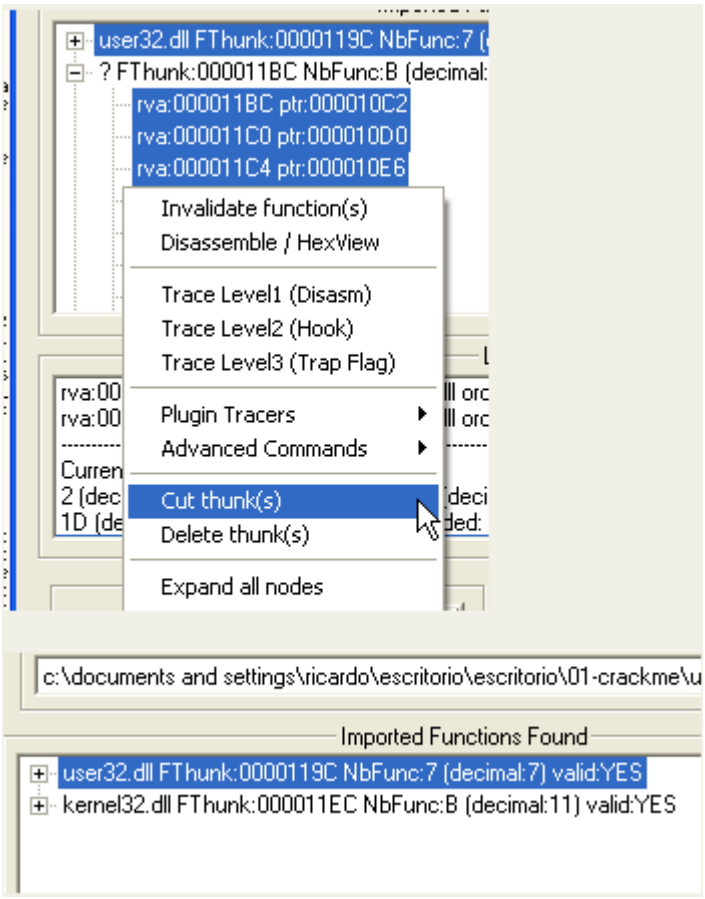
我们选中第一项,单击鼠标右键选择-Disassemble/HexView(以反汇编或者 16 进制方式显示)。



提示读取失败,下面我们来将这些垃圾数据删除掉。

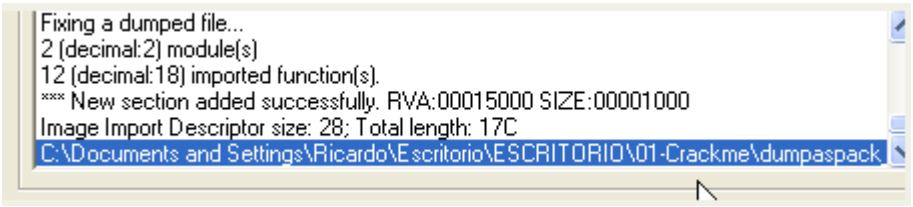


单击 Show Invalid,显示无效的数据,然后单击鼠标右键选择-Cut thunk(s)(剪切掉)。

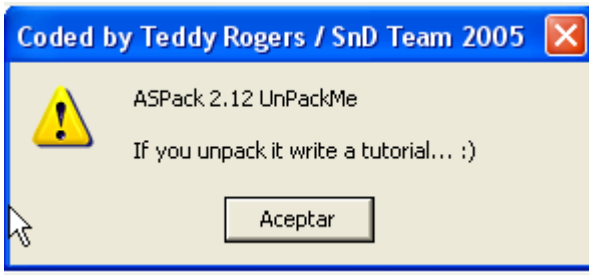


这里这些无效数据就被剪切掉了,程序运行的时候就不会提示尝试加载不存在的 API 函数了。

现在,我们单击 Fix Dump,选择我们之前 dump 出来的文件。



好了,修复完毕,我们双击运行修复完毕的程序 dumpaspack_.exe。



运行很正常,本章的内容就结束了。