

第三十四章-手脱 UPX,修复 IAT

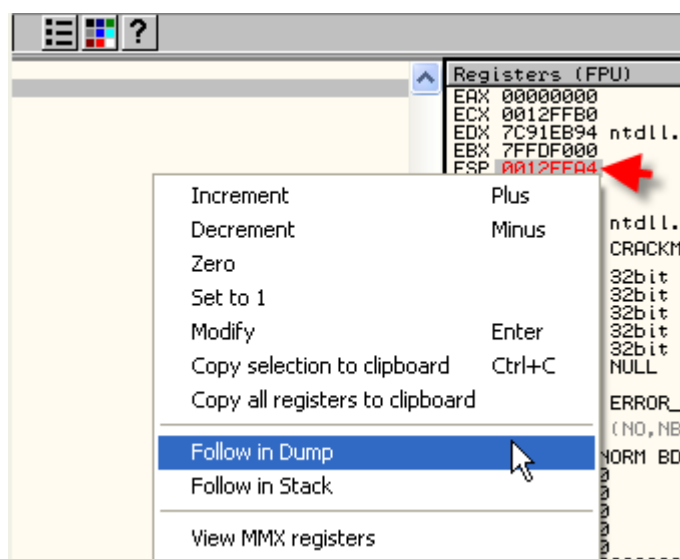
我们在上一章中给大家介绍了 IT(导入表),IAT(输入函数地址表)的相关概念以及原理。有的人可能会认为,我们只是想修复 IAT 呀,并不需要知道 IAT 的具体原理以及它是如何被填充的吧?不是有现成的 IAT 自动修复工具吗?可以很明确的告诉大家,了解操作系统是如何填充 IAT 的过程非常有必要的。因为很多壳会检测这些常用的 IAT 修复工具,致其不能正常运行,在这种情况下,我们就需要自己进行相应的手工修复。

本章我们还是用最简单的 CRACKME UPX 作为例子,我们将对其进行脱壳以及修复 IAT,让其能正常运行。

首先第一步我们来定位 OEP,我们用 OD 加载 CRACKME UPX。

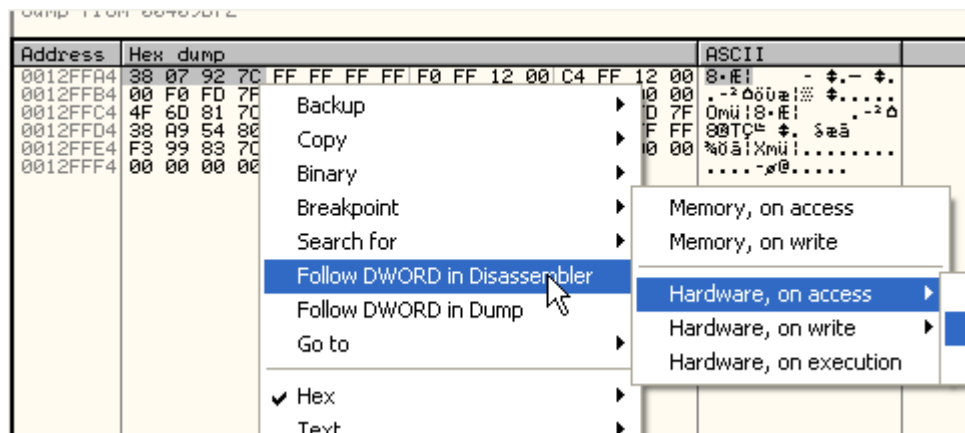


这里我们用 ESP 定律来定位 OEP,现在我们停在了入口点处,单击 F7 键执行 PUSHAD。

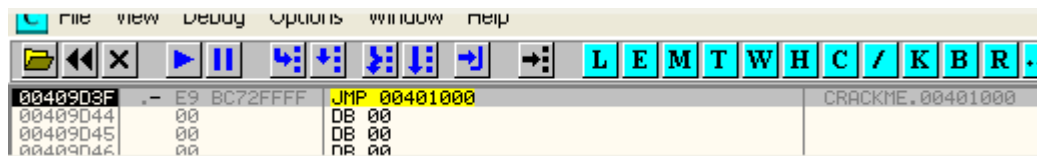


在 ESP 寄存器值上面单击鼠标右键选择-Follow in Dump。

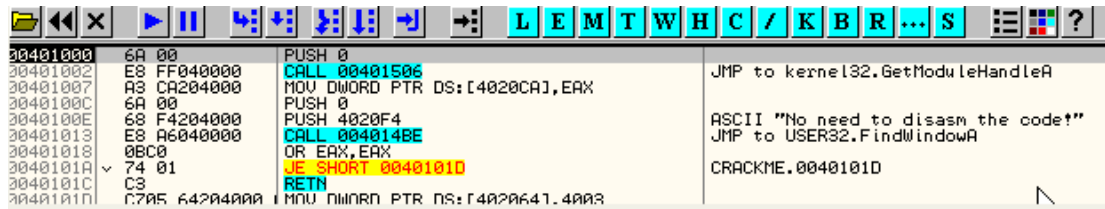
就可以在数据窗口中定位到刚刚 PUSHAD 指令保存到堆栈中的寄存器环境了,我们选中前 4 个字节,我们通过单击鼠标右键选择 Breakpoint-Hardware,on access-Dword 给这 4 个字节设置硬件访问断点。



运行起来,马上就断在了 JMP OEP 指令处。



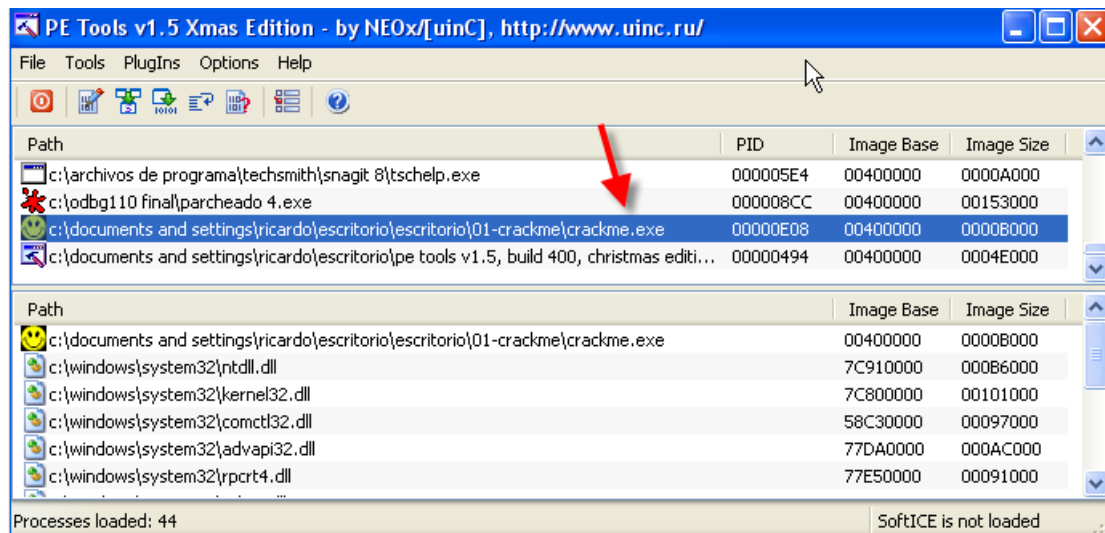
我们直接按 F7 键单步到 OEP 处。



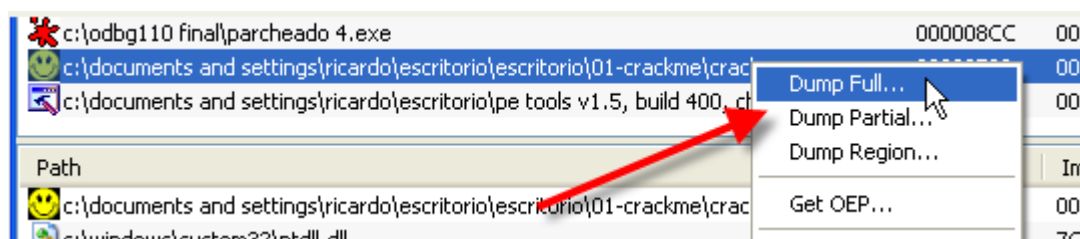
好了,现在我们处于 OEP 处,原程序区段已经解密完毕,我们现在可以进行 dump 了。

前面已经提到过,有很多 dump 的工具,OD 有一个款插件 OllyDump 的 dump 效果也不错。上一章中我们使用 LordPE 来进行 dump 的,这里我们来使用另外一个工具 PE TOOLS 来进行 dump。

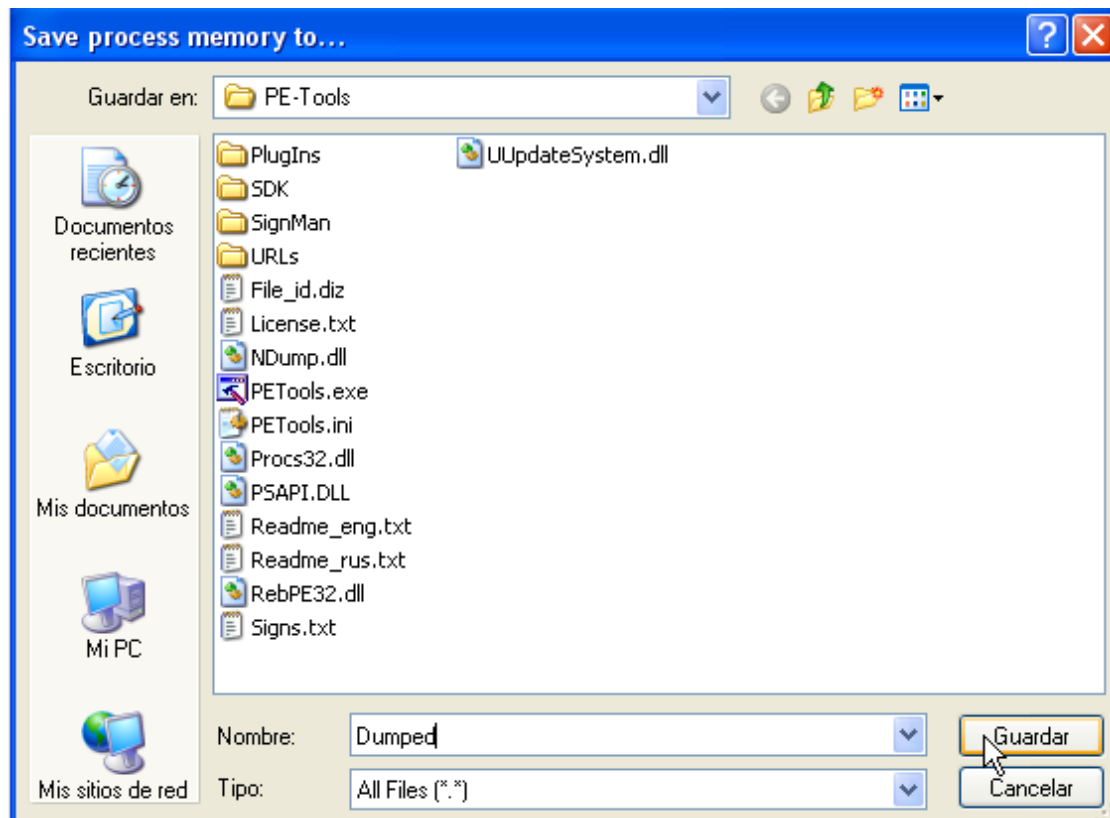
我们用 PE-TOOLS 定位到 CRACKME UPX 所在的进程。



这里这个 crackme.exe 就是,因为我忘了把那个重命名的 CRACKME UPX 放到哪里去了,所以我又重新弄了一个新的,忘了改名字,直接命名为原来的名字 crackme.exe 了,当前这个 crackme.exe 进程停在了 OEP 处。



单击鼠标右键选择-Dump Full。



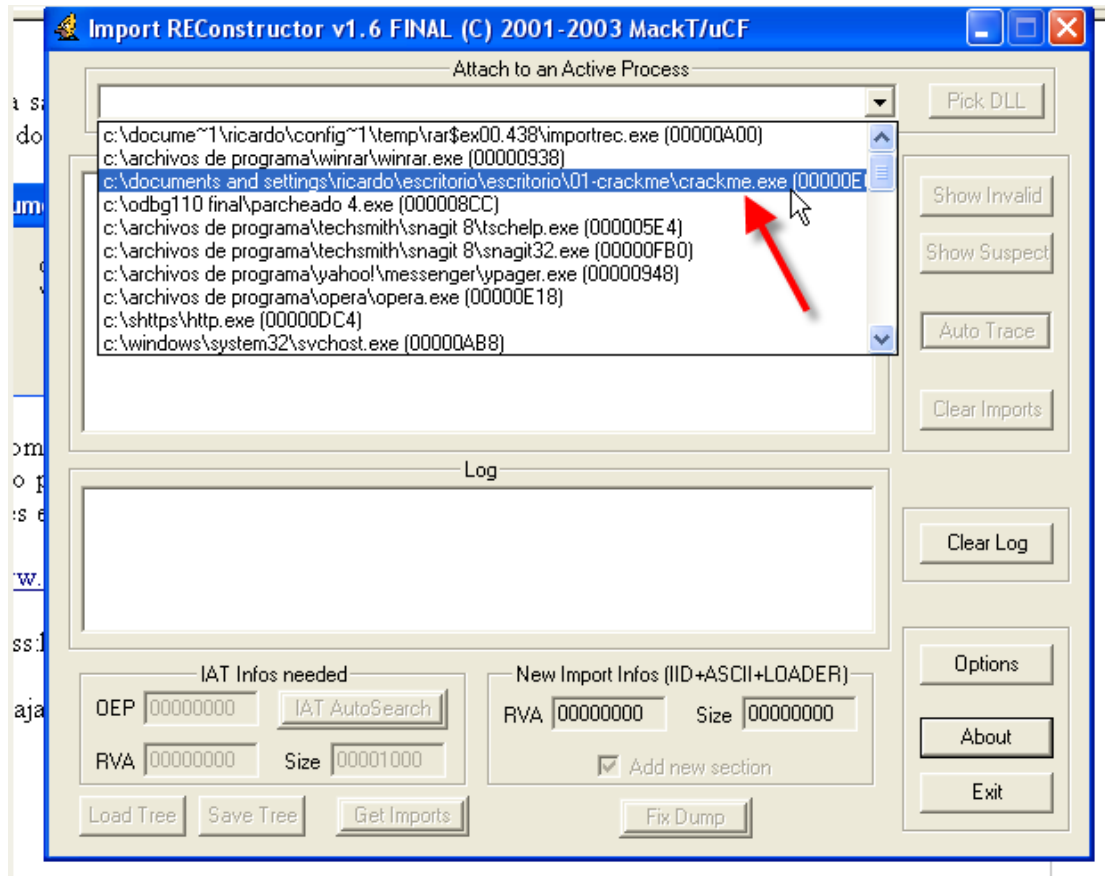
这里,我们就 dump 完毕了,接下来我们来修复 IAT,关闭 PE TOOLS,将 PE TOOLS 目录下的 dumped.exe 拷贝一份到 CRACKME UPX 所在的目录中。



我们知道没有修复 IAT,是不能运行的,我们双击 Dumped.exe 看看会发生什么。



提示无效的 win32 程序,我们需要修复 IAT,我们需要用到一个工具,名字叫做 Import REConstructor,不要关闭 OD,将让其断在 OEP 处,Import REConstructor 需要用到它。



运行 Import REConstructor,定位到 CRACKME UPX 所在的进程。

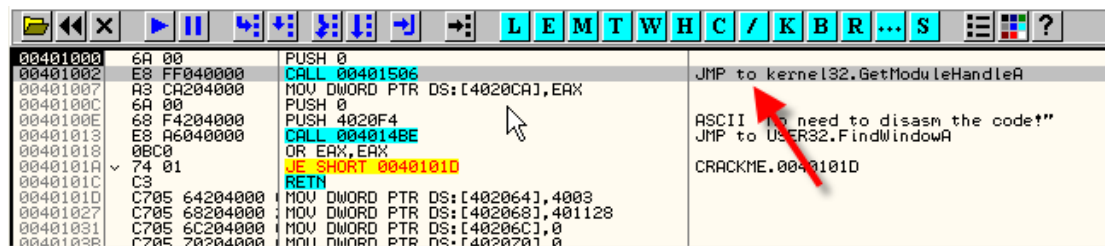
这里很多初学者可能会有疑问-如何定位 IAT 的起始位置和结束位置呢?我们知道当前该 CRACKME UPX 进程停在了 OEP 处,此时壳已经将导入表破坏了,我们知道 IID 项的第四个字段为动态库名称字符串的指针,第五个字段为其对应 IAT 项第一个元素的地址,这些已经被壳破坏了,我们需要通过其他方式来定位。

我们知道 API 函数的调用通常是通过间接跳转或者间接 CALL 来实现的。

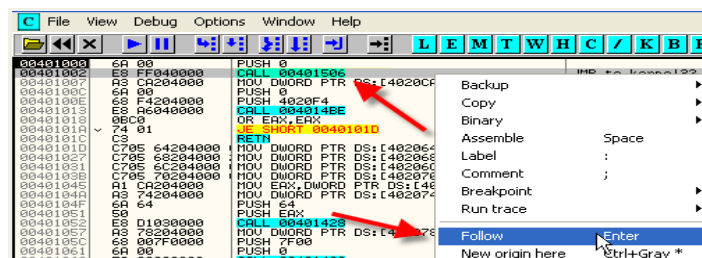
即

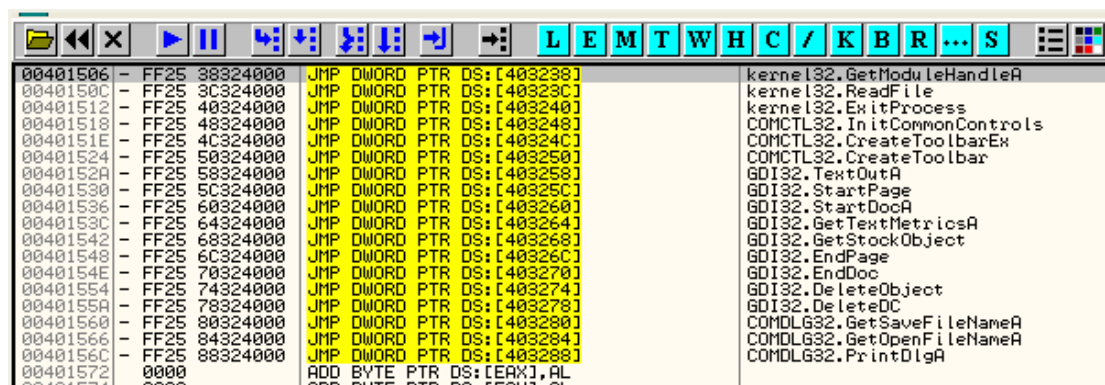
JMP [XXXXXXX] or CALL [XXXXXXX]

上一章我们已经介绍过了,这样是直接调用 IAT 中保存的 API 函数地址。我们来看看 CRACKME UPX。



这里我们看到第二行的 CALL 指令,OD 提示调用的是 Kernel32.dll 中的 GetModuleHandleA,是个间接跳转,我们选中这一行,单击鼠标右键选择-Follow。

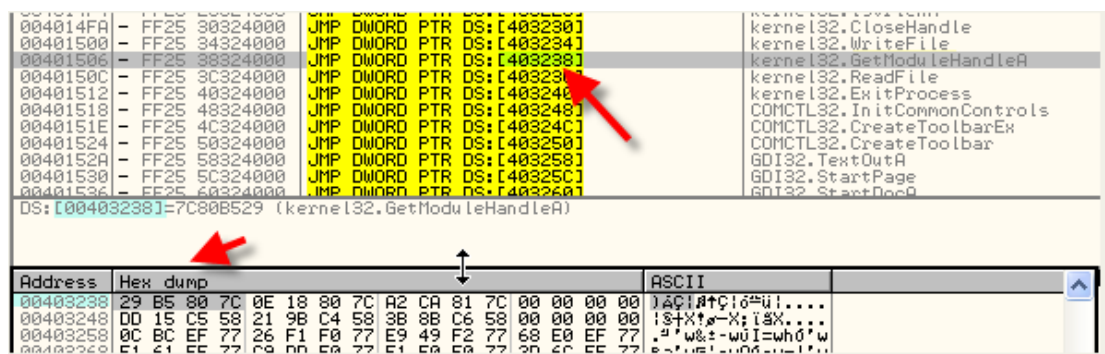




这里我们定位到获取 IAT 中函数地址的跳转表,这里就是该程序将要调用到的一些 API 函数,我们可以看到这些跳转指令的都是以机器码 FF 25 开头的,有些教程里面说直接搜索二进制 FF 25 就可以快速的定位该跳表。

其实,并不是所有的程序都通过这种间接跳转来调用 API 函数,所以直接搜索 FF 25 这种方法有时候会失败,而通过定位某个 API 函数的调用处,然后 Follow 到跳表是这种方式才是一直有效的。

这里我们看到 JMP [403238]:



403238 是 IAT 的其中一个元素,里面保存的是 GetModuleHandleA 这个 API 函数的入口地址,我们 dump 出来的程序的 IAT 部分跟这里是一样的,我们来看看 IAT 的起始位置和结束位置分别在哪儿。

其实,大家可以通过跳转表中最小的地址和最大的地址算出 IAT 的起始位置和结束位置,但是比较慢,比较好的做法是,直接将数据窗口往上滚动,我们知道 IAT 的每个元素中都保存了一个 API 函数的入口地址,比如这里的 7C80B529,在数据窗口中显示的形式为 29 B5 80 7C (小端存储),我们将显示列数调整为两列,这样看起来更方便一些。

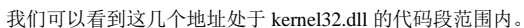
Address	Hex dump	ASCII
00403238	29 B5 80 7C 0E 18 80 7C	!AÇ!A+C!ô#ü!....
00403240	A2 CA 81 7C 00 00 00 00	g#ü!....
00403248	DD 15 C5 58 21 9B C4 58	!\$+X!ø-X;isX....
00403250	3B 88 C6 58 00 00 00 00	;isX....
00403258	0C BC EF 77 26 F1 F0 77	.#w&:-w
00403260	E9 49 F2 77 68 E0 EF 77	üI=whö'w
00403268	E1 61 EF 77 C9 DD F0 77	ßa'wfi'-w
00403270	51 E0 F0 77 2D 6C EF 77	00-w-l'w
00403278	98 6E EF 77 00 00 00 00	yn'w....
00403280	D8 7C 37 76 1E 31 36 76	ii?v▲16v
00403288	CD 46 38 76 00 00 00 00	=F8v....
00403290	00 00 00 00 00 00 00 00
00403298	00 00 00 00 00 00 00 00

这里我们看到的就是整个 IAT 了,我们直接下拉到 IAT 的尾部,我们知道属于同一个动态库的 API 函数地址都是连续存放的,不同的动态库函数地址列表是用零隔开的。

有一些壳会将这部分全部填零,使得我们的 IAT 重建工作变得异常困难,这里由于原程序还需要调用这些 API 函数,所以该壳没有将这部分填零,我们现在来看看其中一个动态库的 IAT 项,如下:

这里显示了该 DLL 中的三个 API 函数,入口地址都是 7C XXXXXX 的形式,然后紧接着是一个零。

我们单击工具栏中的 **M** 按钮看看 7C 开头的地址是属于哪个 DLL 的。



当然在你们的机器上 kernel32.dll 可能在别的地址处,但在我的机器上,这几个函数地址是属于 kernel32.dll 的。

属于 kernel32.dll 中的函数地址这里我用粉红色标注出来了,我们再来看看 77DXXXXX 这类地址是属于哪个 DLL 的。

这里我们可以看到 77DXXXXX 这类地址是属于 user32.dll 的,我也用粉红色标注出来了。

00403168	00 00 00 00 00 00 00 00
00403170	00 00 00 00 00 00 00 00
00403178	00 00 00 00 00 00 00 00
00403180	00 00 00 00 42 8C D1 77B!Dw
00403188	9D 8F D1 77 3E 0B D2 77	0ADw>8Ew
00403190	24 15 D3 77 4C 1F D3 77	83EwLwEw
00403198	04 B6 D1 77 E8 0F D2 77	EADw8Ew
004031A0	24 13 D2 77 DA 5E D2 77	8Ew^Ew
004031A8	60 DA D1 77 EA 04 D5 77	rDwUw
004031B0	11 12 D2 77 35 EE D3 77	4Ew5Ew
004031B8	F5 B5 D1 77 9C FA D2 77	8ADw8Ew
004031C0	EC 0B D1 77 F6 8B D1 77	UADw+Dw
004031C8	83 F7 D4 77 A4 D8 D1 77	8EwADw
004031D0	9A F3 D2 77 2E 8C D1 77	U8Ew.Dw
004031D8	1B C0 D1 77 F9 D7 D1 77	+Dw.Dw
004031E0	8C 14 D2 77 09 B6 D1 77	iEw.Dw
004031E8	5E 02 D2 77 EE D4 D1 77	^EwEw
004031F0	1C B1 D3 77 B8 96 D1 77	L8EwUADw
004031F8	9C F3 D4 77 50 62 D2 77	8EwPBEw
00403200	1D B6 D1 77 81 E5 D2 77	#ADwU8Ew
00403208	C7 86 D1 77 16 48 D2 77	8ADwHEw
00403210	1E AC D6 77 42 10 D2 77	AEwB8Ew
00403218	00 00 00 00 C1 C9 80 7C	...-fC!
00403220	99 6B 82 7C 2F FE 80 7C	0ke!-C!
00403228	2D FF 80 7C E0 C6 80 7C	-C!08C!

我们可以看到user32.dll的这些函数地址项上面是全零的,表示IAT的起始地址为403184,403184中存放的了IAT中的第一个元素。

Address	Hex dump	ASCII
00403174	00 00 00 00 00 00 00 00
0040317C	00 00 00 00 00 00 00 00
00403184	42 8C D1 77 9D 8F D1 77	B!Dw0ADw
0040318C	3E 0B D2 77 24 15 D3 77	>8Ew83Ew
00403194	4C 1F D3 77 04 B6 D1 77	LwEwEADw
0040319C	E8 0F D2 77 24 13 D2 77	8Ew8Ew
004031A4	DA 5E D2 77 60 DA D1 77	r^Ew'rDw
004031AC	EA 04 D5 77 11 12 D2 77	U+^w4Ew
004031B4	35 EE D3 77 F5 B5 D1 77	5Ew8ADw
004031BC	9C FA D2 77 EC 0B D1 77	8EwUADw
004031C4	F6 8B D1 77 83 F7 D4 77	+Dw8Ew
004031CC	A4 D8 D1 77 9A F3 D2 77	8EwU8Ew
004031D4	2E 8C D1 77 1B C0 D1 77	.Dw+Dw
004031DC	F9 D7 D1 77 8C 14 D2 77	.DwiEw
004031E4	09 B6 D1 77 5E 02 D2 77	.Dw^Ew
004031EC	EE D4 D1 77 1C B1 D3 77	EwL8Ew
004031F4	B8 96 D1 77 9C F3 D4 77	UADw8Ew
004031FC	50 62 D2 77 1D B6 D1 77	PBEw#ADw
00403204	81 E5 D2 77 C7 86 D1 77	U8Ew8ADw
0040320C	16 48 D2 77 1E AC D6 77	HEwAEw
00403214	42 10 D2 77 00 00 00 00	B8Ew...
0040321C	C1 C9 80 7C 99 6B 82 7C	-fC!0ke!
00403224	2F FE 80 7C 2D FF 80 7C	-C!-C!
0040322C	E0 C6 80 7C 77 9B 80 7C	08C!w8C!
00403234	9F 0F 81 7C 29 B5 80 7C	8Ew!8C!
0040323C	0E 18 80 7C A2 CA 81 7C	8C!8Ew!
00403244	00 00 00 00 DD 15 C5 58	...!8+X
0040324C	21 9B C4 58 3B 8B C6 58	8Ew-X;8Ew
00403254	00 00 00 00 0C BC EF 778Ew
0040325C	26 F1 F0 77 E9 49 F2 77	8Ew-wU8Ew
00403264	68 E0 EF 77 E1 61 EF 77	h0'w8Ew
0040326C	C9 D0 F0 77 51 E0 F0 77	f!-wU0-w
00403274	2D 6C EF 77 98 6E EF 77	-l'wU8Ew
0040327C	00 00 00 00 08 7C 37 76	...!8+X
00403284	1E 31 36 76 CD 46 38 76	AEwF8Ew
0040328C	00 00 00 00 00 00 00 00
00403294	00 00 00 00 00 00 00 00
0040329C	00 00 00 00 00 00 00 00
004032A4	00 00 00 00 00 00 00 00

这里我们用红线标注出来了,403184 是 IAT 的起始地址。有些强壳可能会将 IAT 前后都填充上垃圾数据,让我们定位 IAT 的起始位置和结束位置更加困难。但是我们知道 IAT 中的数值都是属于某个动态库代码段范围内的,如果我们发现某数值不属于任何一个动态库的代码段的话,就说明该数值是垃圾数据。

现在我们知道了 IAT 开始于 403184,我们现在来看一看 IAT 的结束位置在哪里。

后面我们会遇到有些壳会将 IAT 重定向到壳的例程中去,然后再跳转到 API 函数的入口处,这样的话,上面这样定位 IAT 的起始和结束位置的做法就行不通了。该如何应对这样情况,我们后面再来介绍。

0040323C	0E 18 80 7C H2 CH 81 7C	8Ew!8Ew
00403244	00 00 00 00 DD 15 C5 58	...!8+X
0040324C	21 9B C4 58 3B 8B C6 58	8Ew-X;8Ew
00403254	00 00 00 00 0C BC EF 778Ew
0040325C	26 F1 F0 77 E9 49 F2 77	8Ew-wU8Ew
00403264	68 E0 EF 77 E1 61 EF 77	h0'w8Ew
0040326C	C9 D0 F0 77 51 E0 F0 77	f!-wU0-w
00403274	2D 6C EF 77 98 6E EF 77	-l'wU8Ew
0040327C	00 00 00 00 08 7C 37 76	...!8+X
00403284	1E 31 36 76 CD 46 38 76	AEwF8Ew
0040328C	00 00 00 00 00 00 00 00
00403294	00 00 00 00 00 00 00 00
0040329C	00 00 00 00 00 00 00 00
004032A4	00 00 00 00 00 00 00 00

这里我们可以看到 IAT 的最后一个元素的地址形式为 76XXXXXX,我们来看看它是属于哪个 DLL 的。

58CA4000	0001F000	COMCTL32	.rsrc	resources	Image	R	RWE	
58CC3000	00004000	COMCTL32	.reloc	relocations	Image	R	RWE	
76360000	00001000	COMDLG32		PE header	Image	R	RWE	
76361000	00030000	COMDLG32	.text	code,import	Image	R	RWE	
76391000	00004000	COMDLG32	.data	data	Image	R	RWE	
76395000	00012000	COMDLG32	.rsrc	resources	Image	R	RWE	
763A7000	00003000	COMDLG32	.reloc	relocations	Image	R	RWE	
773A0000	00001000	comctl_1		PE header	Image	R	RWE	
773A1000	00090000	comctl_1	.text	code,import	Image	R	RWE	

这里我们可以看到其是属于 COMDLG32.DLL 的,后面全是零了,所以 40328C 是 IAT 的结束位置。

好了,现在我们知道了 IAT 的起始位置和结束位置。

begin:403184

end:40328c

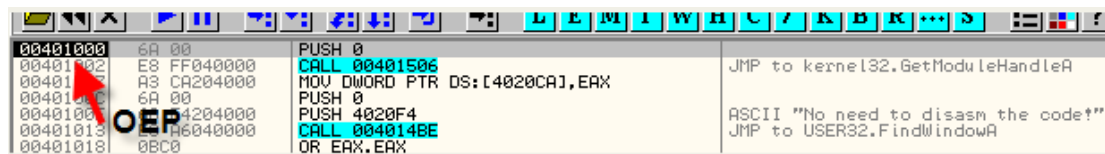
Import REConstructor 重建 IAT 需要三项指标:

1)IAT 的起始地址,这里是 403184,减去映像基址 400000 就得到了 3184(RVA:相对虚拟地址)。

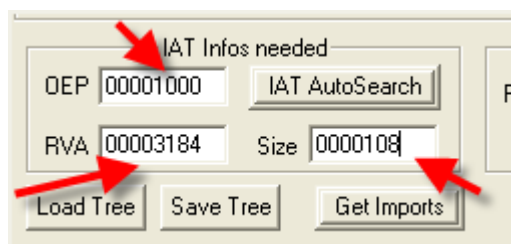
2)IAT 的大小

IAT 的大小 = 40328C - 403184 = 108(十六进制)

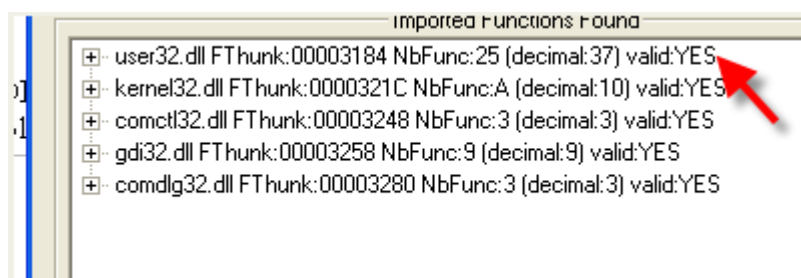
3)OEP = 401000(虚拟地址)- 映像基址 400000 = 1000(OEP 的 RVA)。



我们将这三条数据输入到 Import REConstructor 中。

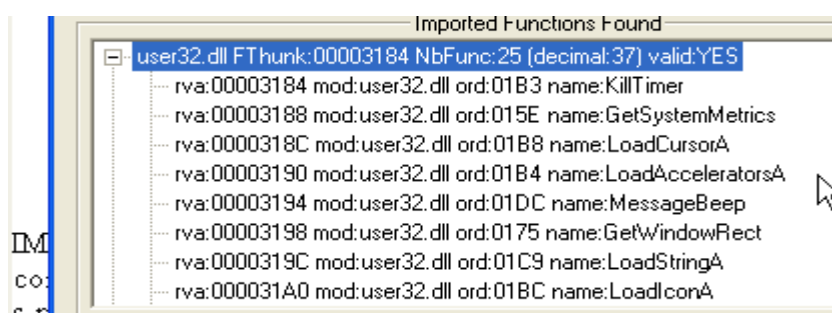


我们单击 Get Imports。

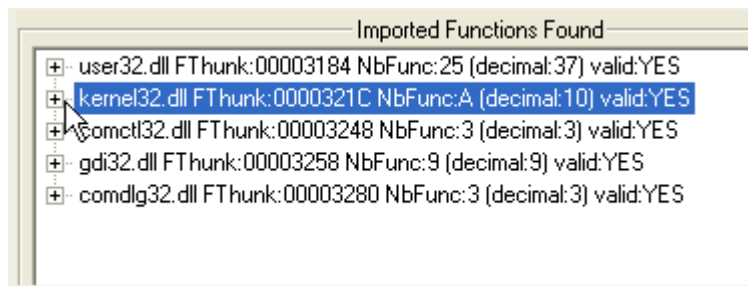


我们看到 Import REConstructor 找到了 IAT 中的每项元素,并且 valid 显示的都是 YES,表示这些项都是有效的,那么无效的是什么情况呢,有些壳会将这些值进行重定向,并不是直接调用 API 函数,就会导致 Import REConstructor 定位出来的项都是无效的,好了现在在这些项都是有效的,我们就可以进行 dump 了。

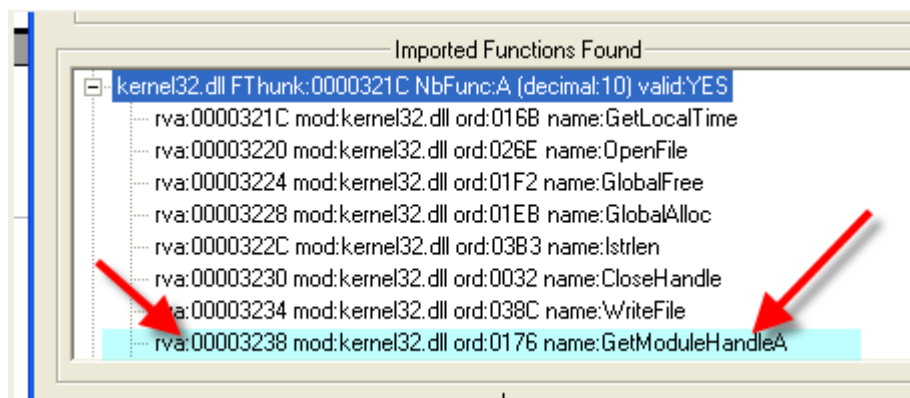
在 dump 之前我们先来看看 User32.dll 这个项中内容是什么,选中该项,单击左边的+号就可以展开。



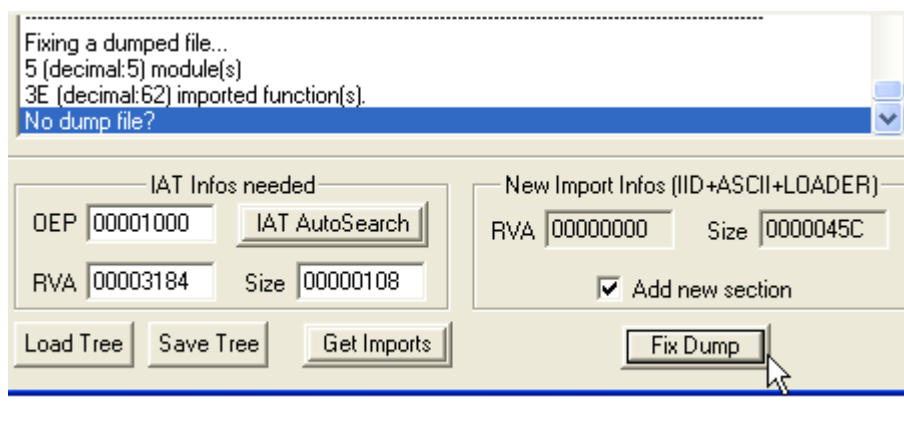
好了,我们可以看到每个 API 函数名称字符串的指针都显示出来了,该程序调用的第一个 API 函数 GetModuleHandleA 这一项在哪里呢?



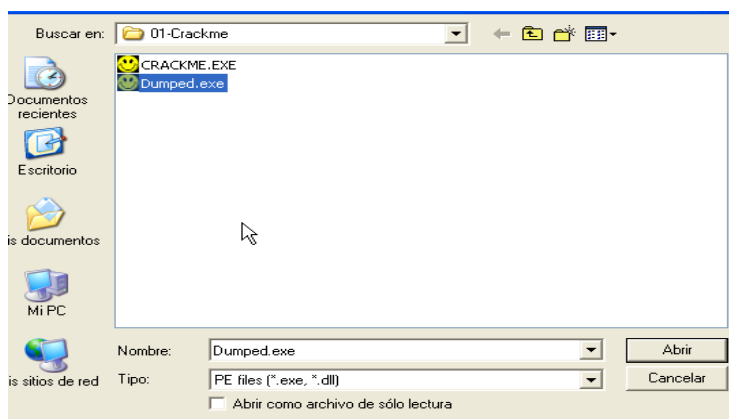
大家应该还记得,IAT 的中的第一项值为 403238,减去映像基址 400000 就等于 3238,位于 kernel32.dll 中。



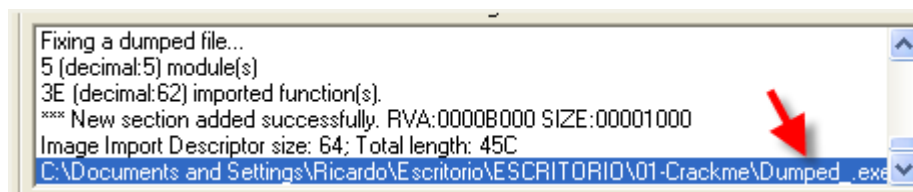
我们单击 kernel32.dll 这一项左边的+号。



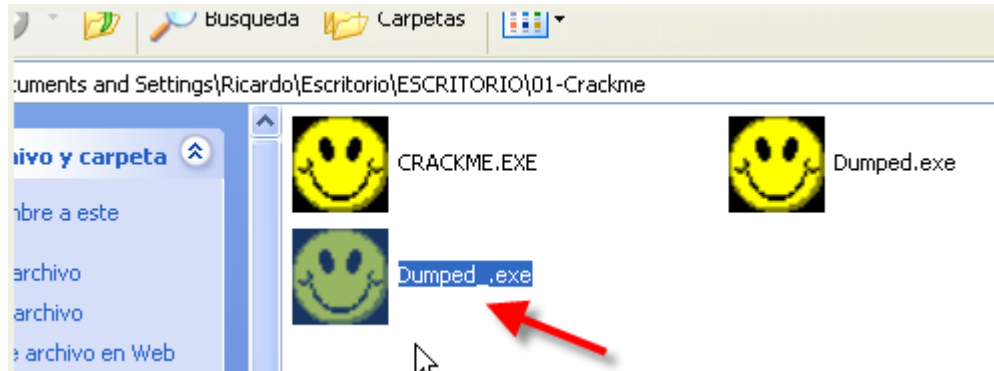
可以看到 3238 对应的正好是 GetModuleHandleA。好了,现在我们可以对之前 dump 出来的程序的 IAT 进行修复了,我们单击 Fix Dump 按钮。



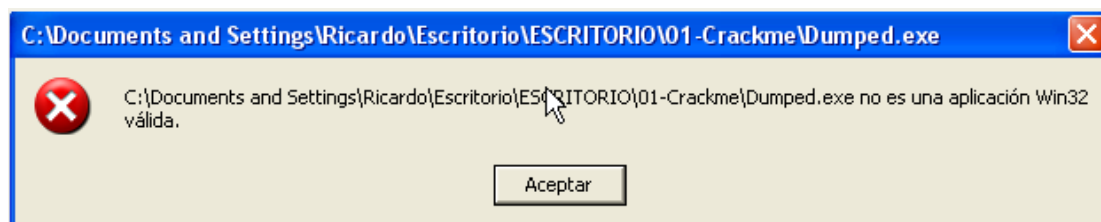
选中之前 dump 出来的文件。



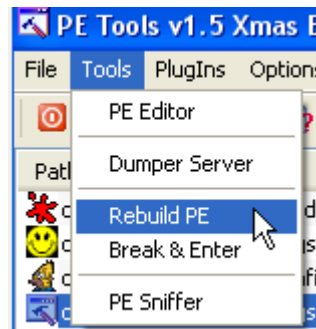
我们可以看到修复完毕了,修复过的文件被重命名为了 dumped_.exe。我们来看一看。



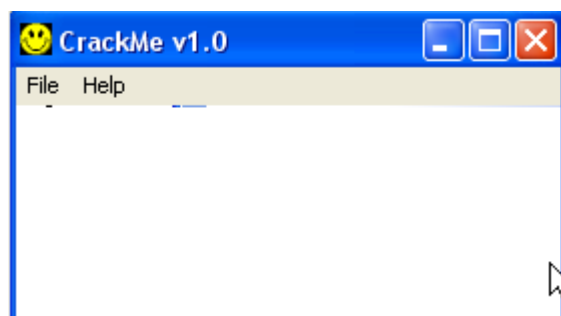
我们双击它,看看能不能正常运行。



我们可以看到还是提示无效的 win32 程序,嘿嘿,为什么呢?别担心,通常修复了 IAT 以后,都会出现这种状况。我们现在来打开 PE TOOLS。

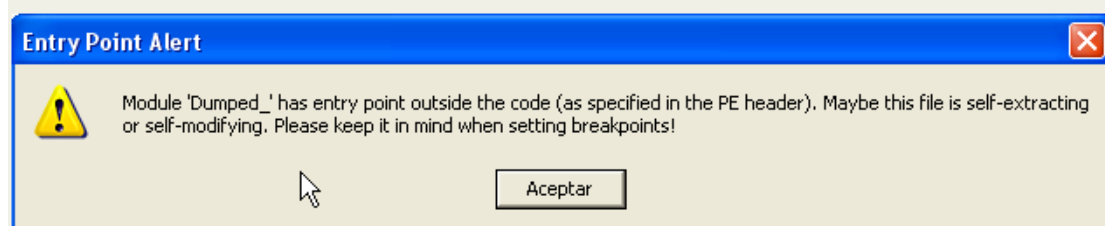
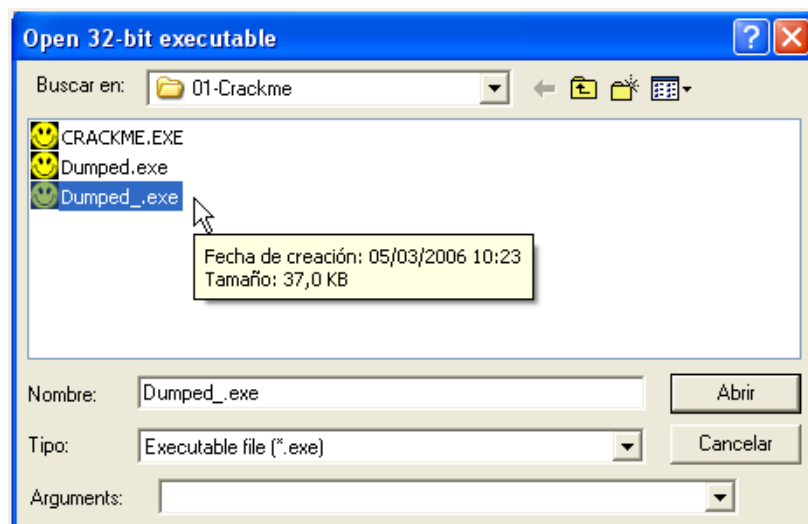


我们选择菜单项中的 Rebuild PE(重建 PE),我们找到刚刚的 dumped_.exe,重建之,运行,发现可以正常运行了,嘿嘿。

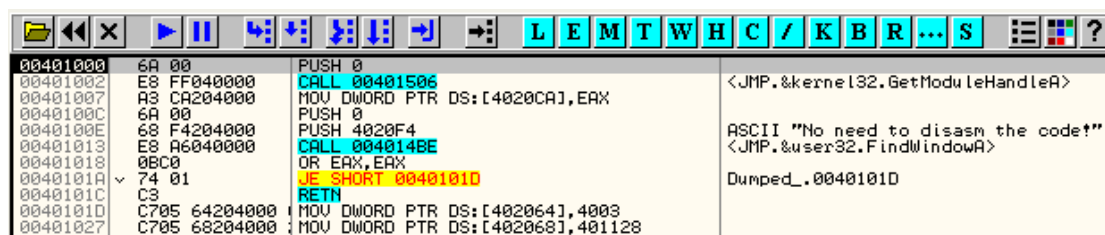


大家将这个程序拿到其它机器上运行也是没有问题的。

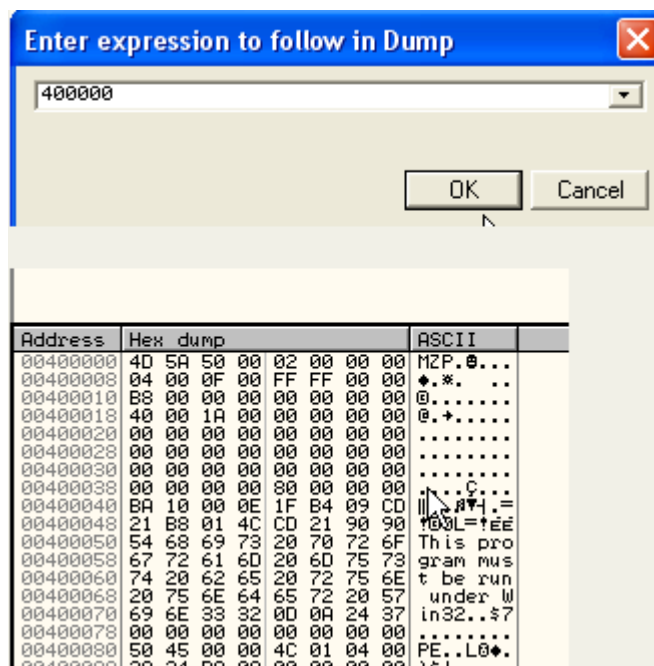
我们将 dumped_.exe 加载到 OD 中。



这里 OD 提示入口点位于代码段之外,因为 UPX 壳将代码段指定到了第三个区段。这个问题可以修复。我们下面来修复它。



我们单击 OK,停在了入口点处,我们在数据窗口中定位到 400000 地址处。



将数据窗口的显示模式切换为 PE 头模式。

Address	Hex dump	ASCII
00400000	4D 5A 50 00 02 00 00 00	MZP.0...
00400008	04 00 0F 00 FF FF 00 00	♦.*. ..
00400010	B8 00 00 00 00 00 00 00	@.....
00400018	40 00 1A 00 00 00 00 00	@.+.....
00400020	00 00 00 00 00 00 00 00
00400028	00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00
00400038	00 00 00 00 00 00 00 00
00400040	BA 00 00 00 00 00 00 00	.87+. =
00400048	21 00 00 00 00 00 00 00	!00L=+EE
00400050	54 00 00 00 00 00 00 00	2 6F This pro
00400058	67 00 00 00 00 00 00 00	5 73 gram mus
00400060	74 00 00 00 00 00 00 00	5 6E t be run
00400068	20 00 00 00 00 00 00 00	0 57 under W
00400070	69 00 00 00 00 00 00 00	4 37 in32..\$7
00400078	00 00 00 00 00 00 00 00
00400080	50 00 00 00 00 00 00 00	4 00 PE..L0+
00400088	29 00 00 00 00 00 00 00	0 00)\$J.....
00400090	00 00 00 00 00 00 00 00	F 810.Au
00400098	0B 00 00 00 00 00 00 00	0 00 000+. ...
004000A0	00 00 00 00 00 00 00 00	0 00 000+. ...
004000A8	00 00 00 00 00 00 00 00	0 00 000+. ...
004000B0	00 00 00 00 00 00 00 00	0 00 000+. ...
004000B8	00 00 00 00 00 00 00 00	0 00 000+. ...
004000C0	01 00 00 00 00 00 00 00	0 00 000+. ...
004000C8	03 00 00 00 00 00 00 00	0 00 000+. ...
004000D0	00 00 00 00 00 00 00 00	0 00 000+. ...

Address	Hex dump	Data	Comment
00400000	4D 5A	ASCII "MZ"	DOS EXE Signature
00400002	5000	DW 0050	DOS_PartPag = 50 (80.)
00400004	0200	DW 0002	DOS_PageCnt = 2
00400006	0000	DW 0000	DOS_ReloCnt = 0
00400008	0400	DW 0004	DOS_HdrSize = 4
0040000A	0F00	DW 000F	DOS_MinMem = F (15.)
0040000C	FFFF	DW FFFF	DOS_MaxMem = FFFF (65535.)
0040000E	0000	DW 0000	DOS_ReloSS = 0
00400010	B800	DW 00B8	DOS_EweSP = B8
00400012	0000	DW 0000	DOS_ChkSum = 0
00400014	0000	DW 0000	DOS_EweIP = 0
00400016	0000	DW 0000	DOS_ReloCS = 0
00400018	4000	DW 0040	DOS_Tabloff = 40
0040001A	1A00	DW 001A	DOS_Overlay = 1A
0040001C	00	DB 00	
0040001D	00	DB 00	

往下拉:

Address	Hex dump	Data	Comment
00400032	00	DB 00	
00400033	00	DB 00	
00400034	00	DB 00	
00400035	00	DB 00	
00400036	00	DB 00	
00400037	00	DB 00	
00400038	00	DB 00	
00400039	00	DB 00	
0040003A	00	DB 00	
0040003B	00	DB 00	
0040003C	80000000	DD 00000080	Offset to PE signature
00400040	BA	DB BA	
00400041	10	DB 10	
00400042	00	DB 00	
00400043	0E	DB 0E	
00400044	1C	DB 1C	

可以看到 PE 头的相对虚拟地址(RVA)为 80,即虚拟地址(VA)为 400080。

Address	Hex dump	Data	Comment
0040007F	00	DB 00	
00400080	50 45 00 00	ASCII "PE"	PE signature (PE)
00400084	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400086	0400	DW 0004	NumberOfSections = 4
00400088	2924D90A	DD 0AD92429	TimeDateStamp = AD92429
0040008C	00000000	DD 00000000	PointerToSymbolTable = 0
00400090	00000000	DD 00000000	NumberOfSymbols = 0
00400094	0000	DW 0000	SizeOfOptionalHeader = E0 (224.)
00400096	8F81	DW 818F	Characteristics = EXECUTABLE_IMAGE 32BIT_MACHINE RE...
00400098	0B01	DW 010B	MagicNumber = PE32
0040009A	02	DB 02	MajorLinkerVersion = 2
0040009B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040009C	00100000	DD 00001000	SizeOfCode = 1000 (4096.)
004000A0	00100000	DD 00001000	SizeOfInitializedData = 1000 (4096.)
004000A4	00000000	DD 00000000	SizeOfUninitializedData = 8000 (32768.)
004000A8	00100000	DD 00001000	AddressOfEntryPoint = 1000
004000AC	00900000	DD 00009000	BaseOfCode = 9000
004000B0	00A00000	DD 0000A000	BaseOfData = A000
004000B4	00004000	DD 00004000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000
004000BC	00020000	DD 00000200	FileAlignment = 200

这里我们看到 Base of Code = 9000, 表示代码段的相对虚拟地址为 9000, 我们需要将代码段的相对虚拟地址修改为 1000。

Address	Hex dump	Data	Comment
0040007F	00	DB 00	
00400080	50 45 00 00	ASCII "PE"	PE signature (PE)
00400084	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400086	0400	DW 0004	NumberOfSections = 4
00400088	2924D90A	DD 0AD92429	TimeDateStamp = AD92429
0040008C	00000000	DD 00000000	PointerToSymbolTable = 0
00400090	00000000	DD 00000000	NumberOfSymbols = 0
00400094	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
00400096	8F81	DW 818F	Characteristics = EXECUTABLE_IMAGE
00400098	0B01	DW 010B	MagicNumber = PE32
0040009A	02	DB 02	MajorLinkerVersion = 2
0040009B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040009C	00100000	DD 00001000	SizeOfCode = 1000 (4096.)
004000A0	00100000	DD 00001000	SizeOfInitializedData = 1000 (4096.)
004000A4	00800000	DD 00008000	SizeOfUninitializedData = 8000 (32768.)
004000A8	00100000	DD 00001000	AddressOfEntryPoint = 1000
004000AC	00900000	DD 00009000	BaseOfCode = 9000
004000B0	00A00000	DD 0000A000	BaseOfData = A000
004000B4	00004000	DD 00400000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000

我们选中 BaseOfCode 这一行, 单击鼠标右键选择-Modify integer。

Modify dword at 00400...

Hexadecimal

00001000

Signed

4096

Unsigned

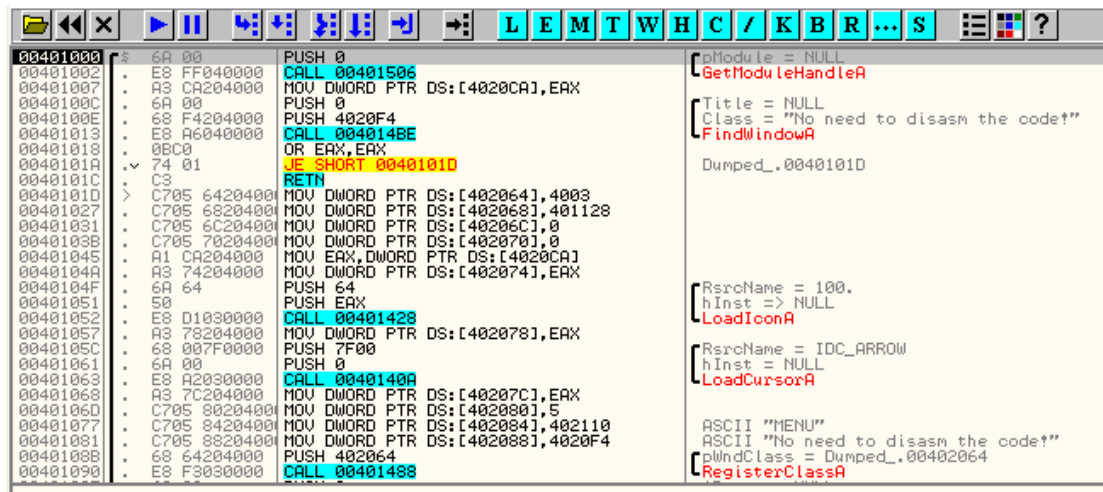
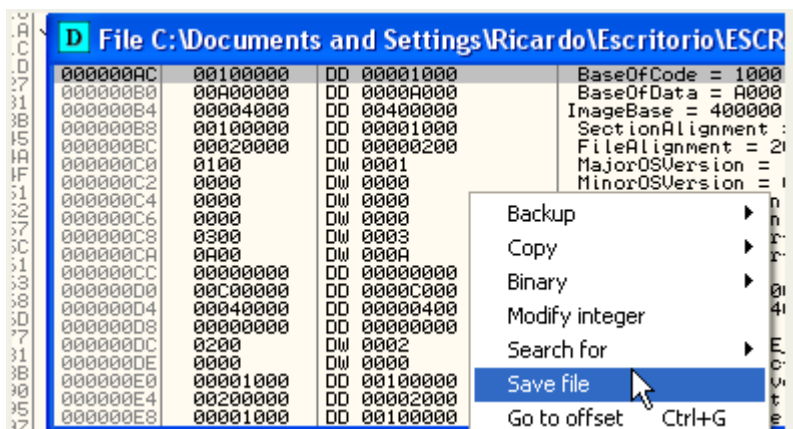
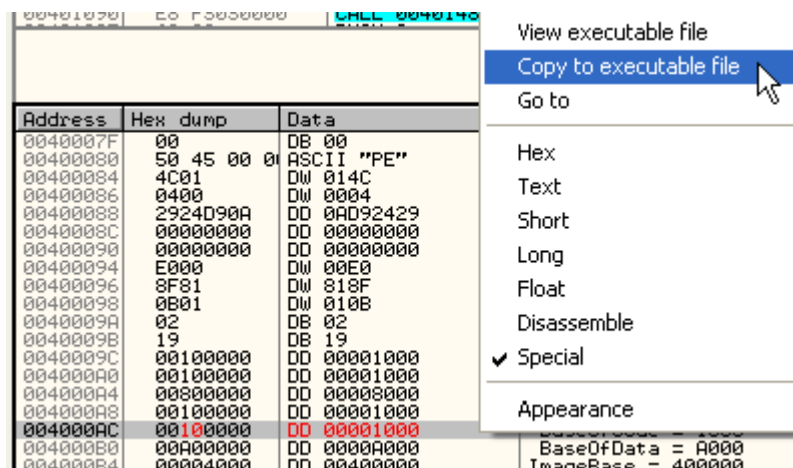
4096

OK

Cancel

Address	Hex dump	Data	Comment
0040007F	00	DB 00	
00400080	50 45 00 00	ASCII "PE"	PE signature (PE)
00400084	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400086	0400	DW 0004	NumberOfSections = 4
00400088	2924D90A	DD 0AD92429	TimeDateStamp = AD92429
0040008C	00000000	DD 00000000	PointerToSymbolTable = 0
00400090	00000000	DD 00000000	NumberOfSymbols = 0
00400094	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
00400096	8F81	DW 818F	Characteristics = EXECUTABLE_IMAGE
00400098	0B01	DW 010B	MagicNumber = PE32
0040009A	02	DB 02	MajorLinkerVersion = 2
0040009B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040009C	00100000	DD 00001000	SizeOfCode = 1000 (4096.)
004000A0	00100000	DD 00001000	SizeOfInitializedData = 1000 (4096.)
004000A4	00800000	DD 00008000	SizeOfUninitializedData = 8000 (32768.)
004000A8	00100000	DD 00001000	AddressOfEntryPoint = 1000
004000AC	00100000	DD 00001000	BaseOfCode = 1000
004000B0	00A00000	DD 0000A000	BaseOfData = A000
004000B4	00004000	DD 00400000	ImageBase = 400000
004000B8	00100000	DD 00001000	SectionAlignment = 1000

修改完毕以后我们将所做的修改保存到文件。



我们再次加载修复过的程序,可以看到 OD 没有弹出警告窗口了。我们单击工具栏中 M 按钮看看各个区段的描述信息。

Address	Size	Section	Attributes	Permissions	Device\Harddisk
003B0000	00003000		Map	R	R
003C0000	00004000		Priv	RW	RW
003D0000	00002000		Map	R	R
003F0000	00002000		Map	R	R
00400000	00001000	Dumped_	Image	R	RWE
00401000	00008000	Dumped_	Image	R	RWE
00409000	00001000	Dumped_	Image	R	RWE
0040A000	00001000	Dumped_	Image	R	RWE
0040B000	00001000	Dumped_	Image	R	RWE
00410000	0000A000	Dumped_	Map	R	R
004D0000	00002000	Dumped_	Map	R	R
004E0000	00103000	Dumped_	Map	R	R

我们可以看到这里多出了一个区段,名字叫做 mackt,这是 Import REConstructor 给该程序添加的一个新的区段,专门用来存放新的导入表。我们在数据窗口中看一看导入表的情况(PE 头显示模式)。

Address	Hex dump	Data	Comment
00400008	00000000	DD 00000000	Checksum = 0
0040000C	0200	DW 0002	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_GUI
0040000E	0000	DW 0000	DLLCharacteristics = 0
00400010	00001000	DD 00100000	SizeOfStackReserve = 100000 (1048576.)
00400014	00200000	DD 00002000	SizeOfStackCommit = 2000 (8192.)
00400018	00001000	DD 00100000	SizeOfHeapReserve = 100000 (1048576.)
0040001C	00100000	DD 00001000	SizeOfHeapCommit = 1000 (4096.)
00400020	00000000	DD 00000000	LoaderFlags = 0
00400024	10000000	DD 00000010	NumberOfRvaAndSizes = 10 (16.)
00400028	00400000	DD 00004000	Export Table address = 4000
0040002C	46000000	DD 00000046	Export Table size = 46 (70.)
00400100	00000000	DD 0000B000	Import Table address = B000
00400104	64000000	DD 00000064	Import Table size = 64 (100.)
00400108	00A00000	DD 0000A000	Resource Table address = A000
0040010C	E0040000	DD 000004E0	Resource Table size = 4E0 (1248.)
00400110	00000000	DD 00000000	Exception Table address = 0
00400114	00000000	DD 00000000	Exception Table size = 0

我们可以看到导入表的相对虚拟地址为 B000,即虚拟地址为 40B000,刚好就是 Import REConstructor 添加的那个区段。

我们将数据窗口的显示模式切换为正常模式。

Dumped .<ModuleEntryPoint>			
Address	Hex dump	ASCII	
0040B000	00 00 00 00 00 00 00 00 00 00 00 00 78 B0 00 00x.....	
0040B010	84 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ä1.....	
0040B020	BA B2 00 00 1C 32 00 00 00 00 00 00 00 00 00 00	...L2.....	
0040B030	00 00 00 00 52 B3 00 00 48 32 00 00 00 00 00 00	...R ..H2.....	
0040B040	00 00 00 00 00 00 00 00 98 B3 00 00 58 32 00 00g ..X2..	
0040B050	00 00 00 00 00 00 00 00 00 00 00 00 1A B4 00 00+.....	
0040B060	80 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Ç2.....	
0040B070	00 00 00 00 00 00 00 00 75 73 65 72 33 32 2E 64user32.d	
0040B080	6C 6C 00 00 B3 01 48 69 6C 6C 54 69 6D 65 72 00	ll.. 0KillTimer.	
0040B090	5E 01 47 65 74 53 79 73 74 65 6D 4D 65 74 72 69	^0GetSystemMetri	
0040B0A0	63 73 00 00 B8 01 4C 6F 61 64 43 75 72 73 6F 72	cs..00LoadCursor	
0040B0B0	41 00 B4 01 4C 6F 61 64 41 63 63 65 6C 65 72 61	A..f0LoadAccelera	
0040B0C0	74 6F 72 73 41 00 DC 01 4D 65 73 73 61 67 65 42	torsA..0MessageB	
0040B0D0	65 65 70 00 75 01 47 65 74 57 69 6E 64 6F 77 52	eeep.u0GetWindowR	
0040B0E0	65 63 74 00 C9 01 4C 6F 61 64 53 74 72 69 6E 67	ect..f0LoadString	
0040B0F0	41 00 BC 01 4C 6F 61 64 49 63 6F 6E 41 00 B6 01	A..0LoadIconA.00	
0040B100	4C 6F 61 64 42 69 74 6D 61 70 41 00 57 02 53 65	LoadBitmapA.00Se	
0040B110	74 46 6F 63 75 73 00 00 DD 01 4D 65 73 73 61 67	tFocus..!0Messag	
0040B120	65 42 6F 78 41 00 02 02 50 6F 73 74 51 75 69 74	eBoxA.00PostQuit	
0040B130	4D 65 73 73 61 67 65 00 D3 02 57 69 6E 48 65 6C	Message..0WinHel	
0040B140	70 41 00 00 94 01 49 6E 76 61 6C 69 64 61 74 65	pA..00Invalidate	
0040B150	52 65 63 74 00 00 07 02 54 72 61 6E 73 6C 61 74	Rect..00Translat	
0040B160	65 41 63 63 65 6C 65 72 61 74 6F 72 00 00 EA 01	eAccelerator..00	
0040B170	4D 6F 76 65 57 69 6E 64 6F 77 00 00 AB 02 54 72	MoveWindow..%0Tr	
0040B180	61 6E 73 6C 61 74 65 4D 65 73 73 61 67 65 00 00	anslateMessage..	
0040B190	C4 01 4C 6F 61 64 4D 65 6E 75 41 00 93 02 53 68	-0LoadMenuA.00Sh	
0040B1A0	6F 77 57 69 6E 64 6F 77 00 00 3C 02 53 65 6E 64	owWindow..<0Send	
0040B1B0	4D 65 73 73 61 67 65 41 00 00 7B 02 53 65 74 54	MessageA..<0SetT	
0040B1C0	69 6D 65 72 00 00 84 02 53 65 74 57 69 6E 64 6F	imer..00SetWindo	
0040B1D0	77 50 6F 73 00 00 BC 02 55 70 64 61 74 65 57 69	wPos..00UpdateWi	
0040B1E0	6E 64 6F 77 00 00 17 02 52 65 67 69 73 74 65 72	ndow..00Register	
0040B1F0	43 6C 61 73 73 41 00 00 0E 00 42 65 67 69 6E 50	ClassA..A.BeginP	
0040B200	61 69 6E 74 00 00 61 00 43 72 65 61 74 65 57 69	aint..a.CreateWi	
0040B210	6E 64 6F 77 45 78 41 00 8F 00 44 65 66 57 69 6E	ndowExA.A.DefWin	
0040B220	64 6F 77 50 72 6F 63 41 00 00 9F 00 44 69 61 6C	dowProcA..f.Dial	
0040B230	6F 67 42 6F 78 50 61 72 61 6D 41 00 A2 00 44 69	ogBoxParamA.0.Di	
0040B240	73 70 61 74 63 68 4D 65 73 73 61 67 65 41 00 00	spatchMessageA..	
0040B250	89 00 44 73 61 72 4D 65 65 75 42 61 72 00 C7 00	#_00DefaultMenuBox	

我们在数据窗口中定位到导入表 40B000 地址处。

Dumped .<ModuleEntryPoint>			
Address	Hex dump	ASCII	
0040B000	00 00 00 00 00 00 00 00 00 00 00 00 78 B0 00 00x.....	
0040B010	84 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ä1.....	
0040B020	BA B2 00 00 1C 32 00 00 00 00 00 00 00 00 00 00	...L2.....	
0040B030	00 00 00 00 52 B3 00 00 48 32 00 00 00 00 00 00	...R ..H2.....	
0040B040	00 00 00 00 00 00 00 00 98 B3 00 00 58 32 00 00g ..X2..	
0040B050	00 00 00 00 00 00 00 00 00 00 00 00 1A B4 00 00+.....	
0040B060	80 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Ç2.....	
0040B070	00 00 00 00 00 00 00 00 75 73 65 72 33 32 2E 64user32.d	
0040B080	6C 6C 00 00 B3 01 48 69 6C 6C 54 69 6D 65 72 00	ll.. 0KillTimer.	
0040B090	5E 01 47 65 74 53 79 73 74 65 6D 4D 65 74 72 69	^0GetSystemMetri	

这里我们可以看到第一个 IID, 第四个字段为 DLL 名称字符串的指针, 这里是 B078(RVA), 即 40B078(VA)。

Dumped .<ModuleEntryPoint>			
Address	Hex dump	ASCII	
0040B078	75 73 65 72 33 32 2E 64 6C 6C 00 00 B3 01 48 69	user32.dll.. 0Ki	
0040B088	6C 6C 54 69 6D 65 72 00 5E 01 47 65 74 53 79 73	llTimer.^0GetSys	
0040B098	74 65 6D 4D 65 74 72 69 63 73 00 00 B8 01 4C 6F	temMetrics..00Lo	
0040B0A8	61 64 43 75 72 73 6F 72 41 00 B4 01 4C 6F 61 64	adCursorA..f0Load	
0040B0B8	41 63 63 65 6C 65 72 61 74 6F 72 73 41 00 DC 01	AcceleratorsA..0	
0040B0C8	4D 65 73 73 61 67 65 42 65 65 70 00 75 01 47 65	MessageBeep.u0Ge	
0040B0D8	74 57 69 6E 64 6F 77 52 65 63 74 00 C9 01 4C 6F	tWindowRect..f0Lo	
0040B0E8	61 64 53 74 72 69 6E 67 41 00 BC 01 4C 6F 61 64	adStringA..0Load	
0040B0F8	49 63 6F 6E 41 00 B6 01 4C 6F 61 64 42 69 74 6D	IconA..00LoadBitm	
0040B108	61 70 41 00 57 02 53 65 74 46 6F 63 75 73 00 00	anA..00SetFocus..	

指向的是 user32.dll。

第五个字段为该 DLL IAT 项的起始地址的 RVA,这里是 3184,即 IAT 中的第一个元素的地址为 403184。

Dumped .<ModuleEntryPoint>																		
Address	Hex dump															ASCII		
00403164	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00403174	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00403184	42	8C	01	77	9D	8F	01	77	3E	0B	02	77	24	15	03	77	B i d w @ A d w > s e w s s e w	
00403194	4C	1F	03	77	04	B6	01	77	E8	0F	02	77	24	13	02	77	L ^ e w e A d w b * e w s ! l e w	
004031A4	0A	5E	02	77	60	0A	01	77	EA	04	05	77	11	12	02	77	r ^ e w ' r d w u d w i d w	
004031B4	35	EE	03	77	F5	B5	01	77	9C	FA	02	77	EC	0B	01	77	S ^ e w s A d w e e w u d w	
004031C4	F6	8B	01	77	83	F7	04	77	A4	08	01	77	9A	F3	02	77	+ i d w s e w n i d w u e w	
004031D4	2E	8C	01	77	1B	C0	01	77	F9	07	01	77	8C	14	02	77	. i d w + l d w i d w i e w	
004031E4	09	B6	01	77	5E	02	02	77	EE	04	01	77	1C	B1	03	77	. A d w ^ e w e d w l e w	
004031F4	B8	96	01	77	9C	F3	04	77	50	62	02	77	1D	B6	01	77	@ d w e e w P b e w # A d w	
00403204	81	E5	02	77	C7	86	01	77	16	48	02	77	1E	AC	06	77	u d e w s A d w l H e w A e w	
00403214	42	10	02	77	00	00	00	00	C1	C9	80	7C	99	6B	82	7C	B b e w + f c i d k e i	
00403224	2F	FE	80	7C	2D	FF	80	7C	E0	C6	80	7C	77	9B	80	7C	/ ^ c i - c i d c i w c c i	
00403234	9F	0F	81	7C	29	B5	80	7C	0E	18	80	7C	A2	CA	81	7C	f * u i l A c i d t c i d u i	
00403244	00	00	00	00	DD	15	C5	58	21	9B	C4	58	3B	8B	C6	58 ! s + x ! s - x i s x	
00403254	00	00	00	00	0C	BC	EF	77	26	F1	F0	77	E9	49	F2	77 ' u w s t - w d u e w	
00403264	68	E0	EF	77	E1	61	EF	77	C9	D0	F0	77	51	E0	F0	77	h o ' w b a ' w f i - w d o o w	
00403274	2D	6C	EF	77	98	6E	EF	77	00	00	00	00	08	7C	37	76	- l ' w y n ' w i i l v	
00403284	1E	31	36	76	CD	46	38	76	00	00	00	00	00	00	00	00	A 1 6 v = F 8 v	
00403294	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
004032A4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
004032B4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
004032C4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

这里我们可以看到 IAT 中保存的各个 API 函数的入口地址。我们定位到可执行文件中对应 IAT 的偏移处,看看各个 API 函数名称字符串指针的情况如何。

Dumped .<ModuleEntryPoint>									
Address	Hex dump								
00403164	00	00	00	00	00	00	00	00	00
00403174	00	00	00	00	00	00	00	00	00
00403184	42	8C	01	77	9D	8F	01	77	3E
00403194	4C	1F	03	77					
004031A4	0A	5E	02	77	Backup				
004031B4	35	EE	03	77					
004031C4	F6	8B	01	77	Copy				
004031D4	2E	8C	01	77					
004031E4	09	B6	01	77	Binary				
004031F4	B8	96	01	77					
00403204	81	E5	02	77	Label				
00403214	42	10	02	77					
00403224	2F	FE	80	7C	Breakpoint				
00403234	9F	0F	81	7C					
00403244	00	00	00	00	Search for				
00403254	00	00	00	00					
00403264	68	E0	EF	77	Follow DWORD in Disassemble				
00403274	2D	6C	EF	77	Follow DWORD in Dump				
00403284	1E	31	36	76					
00403294	00	00	00	00	Find references				
004032A4	00	00	00	00					
004032B4	00	00	00	00	View executable file				
004032C4	00	00	00	00	Compute executable file				

Dumped .<ModuleEntryPoint>									
Address	Hex dump								
00002584	84	B0	00	00	90	B0	00	00	A4
00002594	C6	B0	00	00	04	B0	00	00	E4
000025A4	FE	B0	00	00	0C	B1	00	00	18
000025B4	38	B1	00	00	44	B1	00	00	56
000025C4	7C	B1	00	00	90	B1	00	00	9C
000025D4	BA	B1	00	00	C6	B1	00	00	D6
000025E4	F8	B1	00	00	06	B2	00	00	18
000025F4	3C	B2	00	00	00	B2	00	00	5E
00002604	76	B2	00	00	84	B2	00	00	8C
00002614	AC	B2	00	00	00	00	00	00	C8
00002624	E4	B2	00	00	F2	B2	00	00	0A
00002634	18	B3	00	00	24	B3	00	00	38
00002644	00	00	00	00	60	B3	00	00	76
00002654	00	00	00	00	A2	B3	00	00	AE
00002664	C6	B3	00	00	D8	B3	00	00	EA
00002674	FE	B3	00	00	0E	B4	00	00	28
00002684	3C	B4	00	00	50	B4	00	00	00
00002694	00	00	00	00	00	00	00	00	00

我们可以看到 IAT 项中的值为 B084,即 40B084,指向的刚好是 KillTimer 这个 API 函数名称字符串。

Address	Hex dump																ASCII
0040B084	B3	01	4B	69	6C	6C	54	69	6D	65	72	00	5E	01	47	65	!0KillTimer.^0Ge
0040B094	74	53	79	73	74	65	6D	40	65	74	72	69	63	73	00	00	tSystemMetrics..
0040B0A4	B8	01	4C	6F	61	64	43	75	72	73	6F	72	41	00	B4	01	@LoadCursorA.l0
0040B0B4	4C	6F	61	64	41	63	63	65	6C	65	72	61	74	6F	72	73	LoadAccelerators
0040B0C4	41	00	DC	01	40	65	73	73	61	67	65	42	65	65	70	00	A.#0MessageBeep.
0040B0D4	75	01	47	65	74	57	69	6E	64	6F	77	52	65	63	74	00	u0GetWindowRect.
0040B0E4	C9	01	4C	6F	61	64	53	74	72	69	6E	67	41	00	BC	01	fLoadStringA.#0

这样我们就通过 Import REConstructor 完成 IAT 的重建工作,操作系统在程序启动的时候就可以根据 IAT 中 API 函数名称字符串通过调用 GetProcAddress 获取到相应 API 函数地址,然后重新填充到 IAT 中。

本章我们就完成了第一个 IAT 的修复工作,虽然不是很难,但是也算是对我们前面章节知识点的一个汇总吧。大家要好好理解本章的内容。

本章我们没有介绍 AntiDump,等我们后面遇到了再介绍。