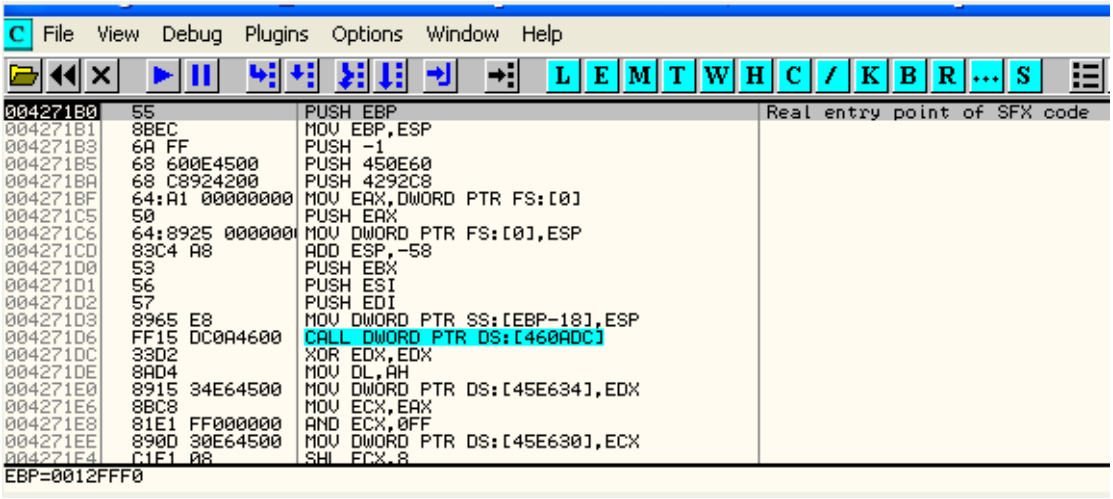
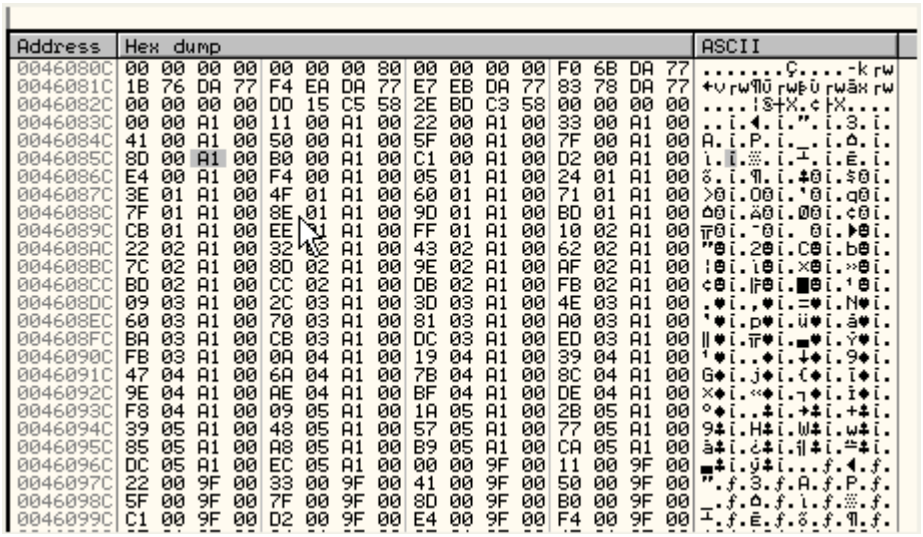
论 IAT 重定向之修复

本章我将给大家介绍几种常见的修复 IAT 重定向的方法。

我们只是介绍修复 IAT 重定向的基本思路,有些方法可能对有个壳有效,对有的壳无效,这个就需要大家多加练习,举一反三,触类旁通了。

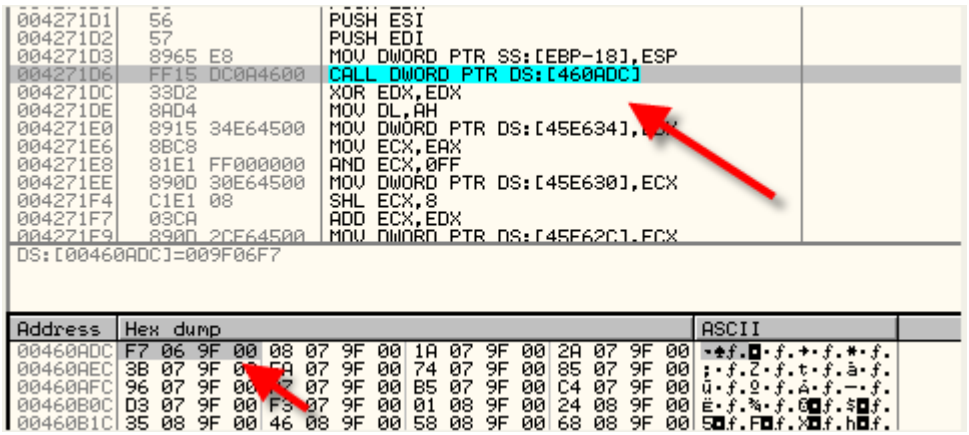


我们用 OD 加载 UnPackMe_tElock0.98,按照上一章节介绍的方法定位到其 OEP,这里就不再赘述了。IAT 中有部分元素被重定向到壳创建的区段中去了。

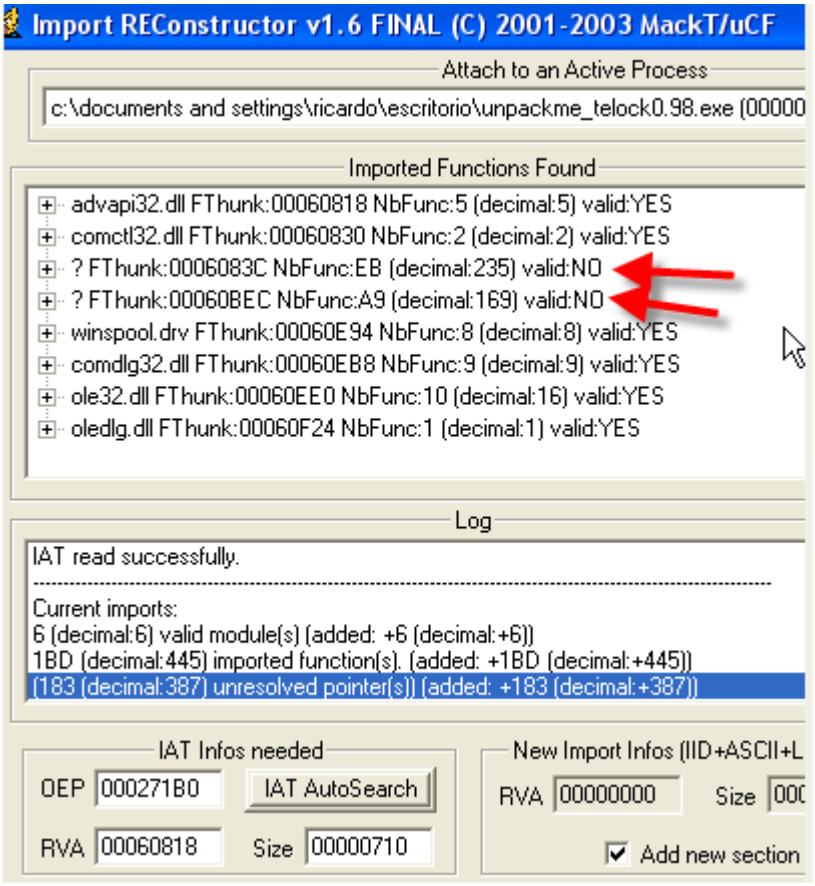


这些重定向过的 IAT 项我们需要修复。

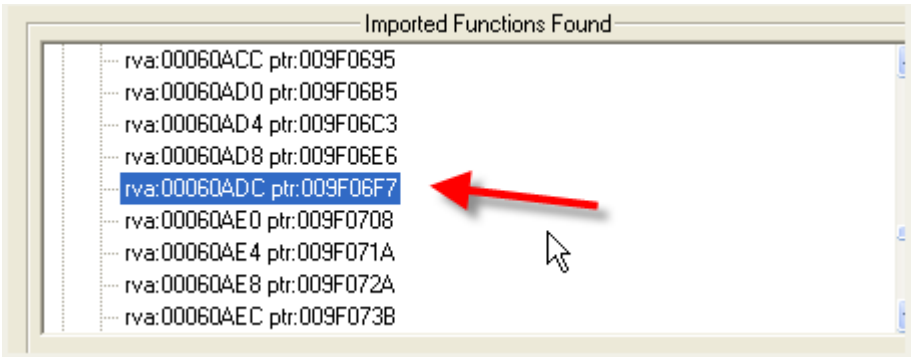
我们首先单步到第一个 API 函数 GetVersion 的调用处。



在数据窗口中定位到该 IAT 项 460ADC,这里 460ADC 保存的内容是 9F06F7,我们打开 IMP REC,填充上 IAT 起始地址的 RVA,大小以及 OEP 的 RVA 后,看一看无效的项。

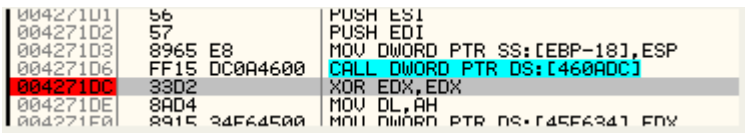


我们可以看到 9F06F7 这一项是无效的。

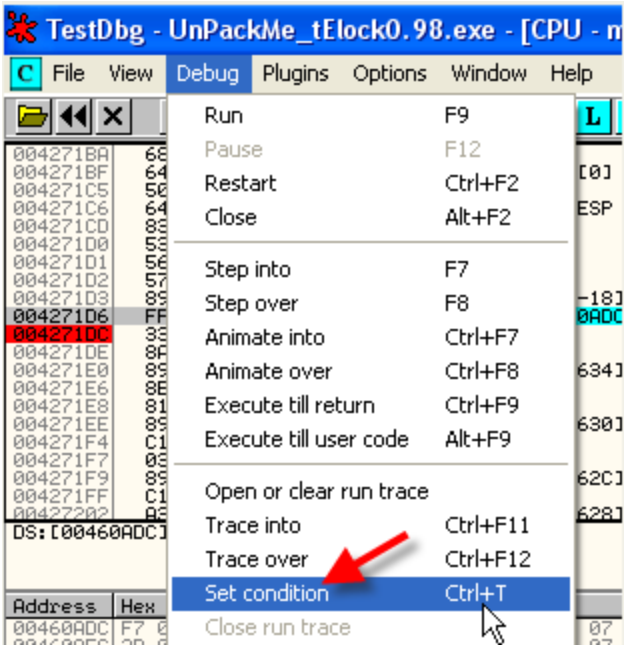


RVA 60ADC 对应的 IAT 项正好是 460ADC。

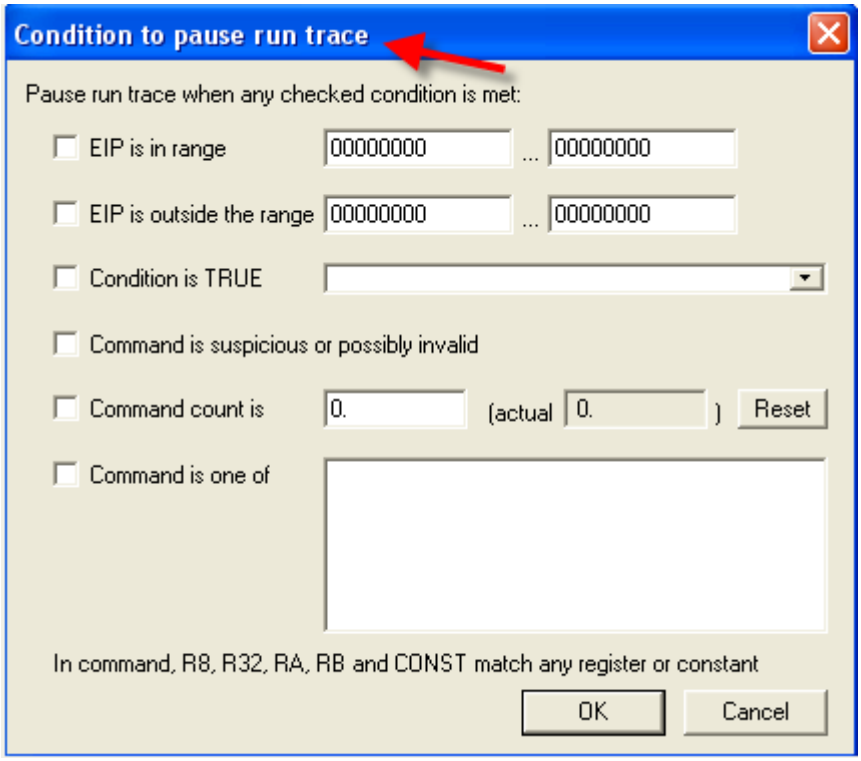
我们回到 OD 中,上一章我们已经介绍过了如何定位 GetVersion 这个 API 函数的入口地址,其实 OD 有一个自动单步跟踪的功能,我们用这个自动单步跟踪的功能可以方便定位到重定向过的 IAT 项对应的 API 函数入口地址,重定向过的 IAT 项单步步入 5,6 行就可以定位到其对应的 API 函数。



我们在 4271D6 这条 CALL 指令的返回地址处设置一个断点,这里我们缺少一步,由于 OD 的自动跟踪功能是不会主动停止的,我们应该设置自动跟踪停止的条件。



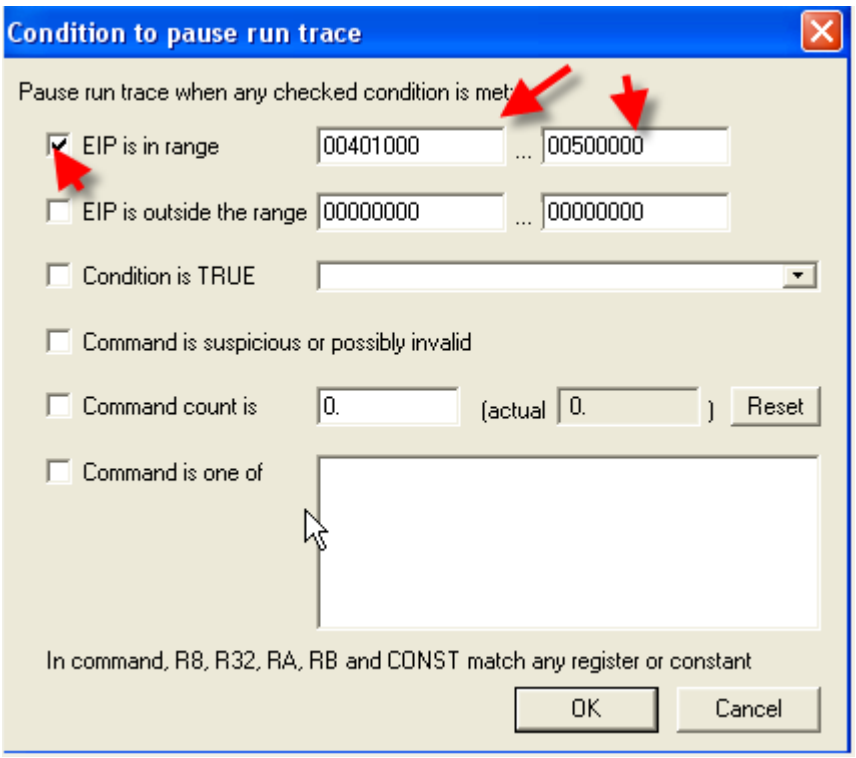
我们选择主菜单中 Debug-Set condition,设置跟踪停止的条件。



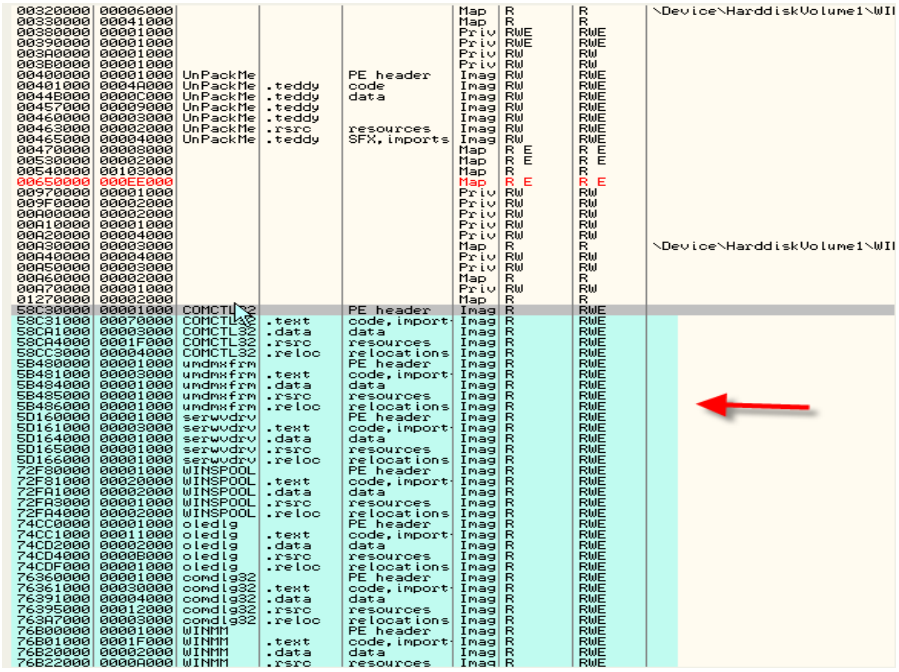
我们可以看到弹出了一个对话框,对话框的标题为 Condition to pause run trace(设置自动跟踪终止的条件)。

如果设置了 EIP is in range(EIP 的范围)这个选项的话,当 EIP 位于这个范围的时候,OD 就会停止自动跟踪。

例如:

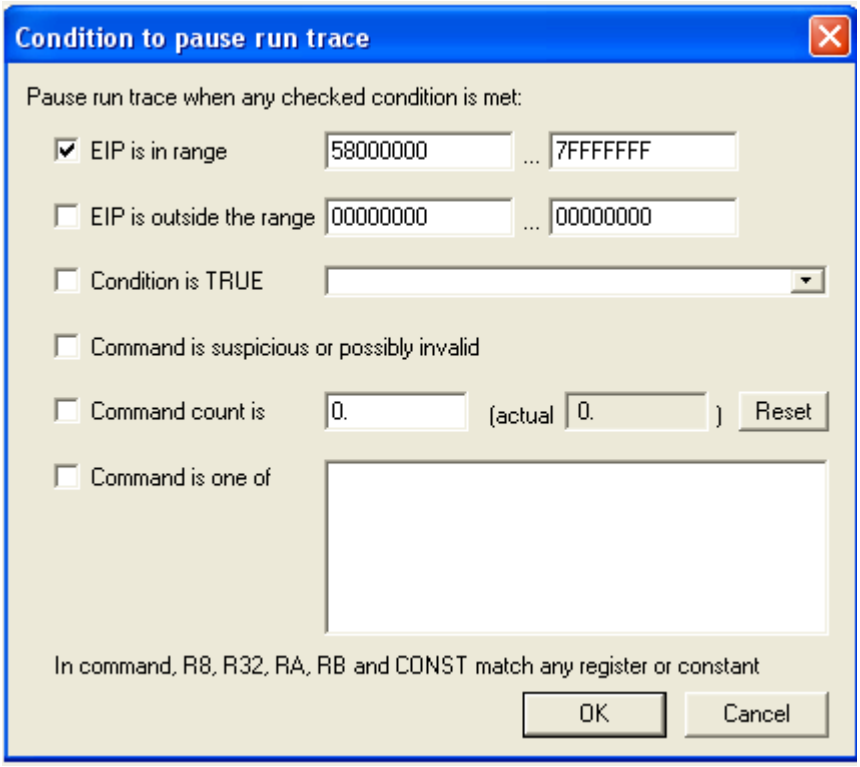


例如,这里我们将这个范围指定为 401000 ~ 500000,那么 OD 自动跟踪,当 EIP 在 401000 ~ 500000 这个范围的时候就会停止自动跟踪,这里我们只是举个例子,并不是真的要将范围指定为 401000 ~ 500000，因为我们需要 OD 自动跟踪停在 API 函数的入口处,所以我们打开区段列表窗口看看将这个范围指定为什么比较合适。

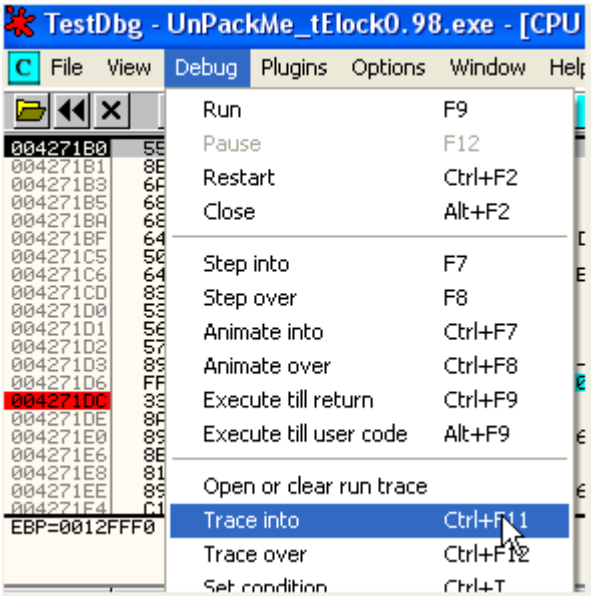


00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WI
00330000	00041000				Map	R	R	
00380000	00001000				Priv	RWE	RWE	
00390000	00001000				Priv	RWE	RWE	
003A0000	00001000				Priv	RW	RW	
003B0000	00001000				Priv	RW	RW	
00400000	00001000	UnPackMe		PE header	Imag	RW	RWE	
00401000	0004A000	UnPackMe	.teddy	code	Imag	RW	RWE	
0044B000	00009000	UnPackMe	.teddy	data	Imag	RW	RWE	
00457000	00009000	UnPackMe	.teddy		Imag	RW	RWE	
00460000	00003000	UnPackMe	.teddy		Imag	RW	RWE	
00463000	00002000	UnPackMe	.rsrc	resources	Imag	RW	RWE	
00465000	00004000	UnPackMe	.teddy	SFX,imports	Imag	RW	RWE	
00470000	00008000				Map	R E	R E	
00530000	00002000				Map	R E	R E	
00540000	00103000				Map	R	R	
00650000	006EE000				Map	R E	R E	
00970000	00001000				Priv	RW	RW	
009F0000	00002000				Priv	RW	RW	
00A00000	00002000				Priv	RW	RW	
00A10000	00001000				Priv	RW	RW	
00A20000	00004000				Priv	RW	RW	
00A30000	00003000				Map	R	R	\Device\HarddiskVolume1\WI
00A40000	00004000				Priv	RW	RW	
00A50000	00003000				Priv	RW	RW	
00A60000	00002000				Map	R	R	
00A70000	00001000				Priv	RW	RW	
01270000	00002000				Map	R	R	
58C30000	00001000	COMCTL32		PE header	Imag	R	RWE	
58C31000	00070000	COMCTL32	.text	code,import	Imag	R	RWE	
58CA1000	00003000	COMCTL32	.data	data	Imag	R	RWE	
58CA4000	0001F000	COMCTL32	.rsrc	resources	Imag	R	RWE	
58CC3000	00004000	COMCTL32	.reloc	relocations	Imag	R	RWE	
5B480000	00001000	umdmxfrm		PE header	Imag	R	RWE	
5B481000	00003000	umdmxfrm	.text	code,import	Imag	R	RWE	
5B484000	00001000	umdmxfrm	.data	data	Imag	R	RWE	
5B485000	00001000	umdmxfrm	.rsrc	resources	Imag	R	RWE	
5B486000	00001000	umdmxfrm	.reloc	relocations	Imag	R	RWE	
5D160000	00001000	serwvdrv		PE header	Imag	R	RWE	
5D161000	00003000	serwvdrv	.text	code,import	Imag	R	RWE	
5D164000	00001000	serwvdrv	.data	data	Imag	R	RWE	
5D165000	00001000	serwvdrv	.rsrc	resources	Imag	R	RWE	
5D166000	00001000	serwvdrv	.reloc	relocations	Imag	R	RWE	
72F80000	00001000	WINSPOOL		PE header	Imag	R	RWE	
72F81000	00020000	WINSPOOL	.text	code,import	Imag	R	RWE	
72FA1000	00002000	WINSPOOL	.data	data	Imag	R	RWE	
72FA3000	00001000	WINSPOOL	.rsrc	resources	Imag	R	RWE	
72FA4000	00002000	WINSPOOL	.reloc	relocations	Imag	R	RWE	
74CC0000	00001000	oledlg		PE header	Imag	R	RWE	
74CC1000	00011000	oledlg	.text	code,import	Imag	R	RWE	
74CD2000	00002000	oledlg	.data	data	Imag	R	RWE	
74CD4000	0000B000	oledlg	.rsrc	resources	Imag	R	RWE	
74CDF000	00001000	oledlg	.reloc	relocations	Imag	R	RWE	
763C0000	00001000	comdlg32		PE header	Imag	R	RWE	
763C1000	00030000	comdlg32	.text	code,import	Imag	R	RWE	
76391000	00004000	comdlg32	.data	data	Imag	R	RWE	
76395000	00012000	comdlg32	.rsrc	resources	Imag	R	RWE	
763A7000	00003000	comdlg32	.reloc	relocations	Imag	R	RWE	
76B80000	00001000	WINMM		PE header	Imag	R	RWE	
76B81000	0001F000	WINMM	.text	code,import	Imag	R	RWE	
76B20000	00002000	WINMM	.data	data	Imag	R	RWE	
76B22000	0000A000	WINMM	.rsrc	resources	Imag	R	RWE	

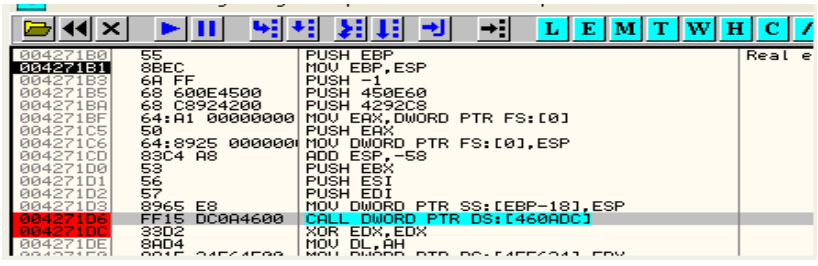
我们看到区段列表窗口中用浅蓝色标注出来的部分,这些是系统 DLL 的区段,当 EIP 位于主程序的区段或者壳创建的区段中,我们让 OD 自动继续跟踪,OD 跟进到 DLL 中的时候,我们需要 OD 停止跟踪,所以这里我们将该范围设置为 5800000~7FFFFFFF,这样就能确保 OD 跟进到任意一个系统 DLL 中的时候就会停止跟踪,这里我并没有精确的设置为 58C30000 ~ 7FFFFFFF,大家想设置的精确一点也是可以的。



也就是说 OD 自动单步跟踪,当 EIP 位于任意一个系统 DLL 的区段中的时候就会停止跟踪。

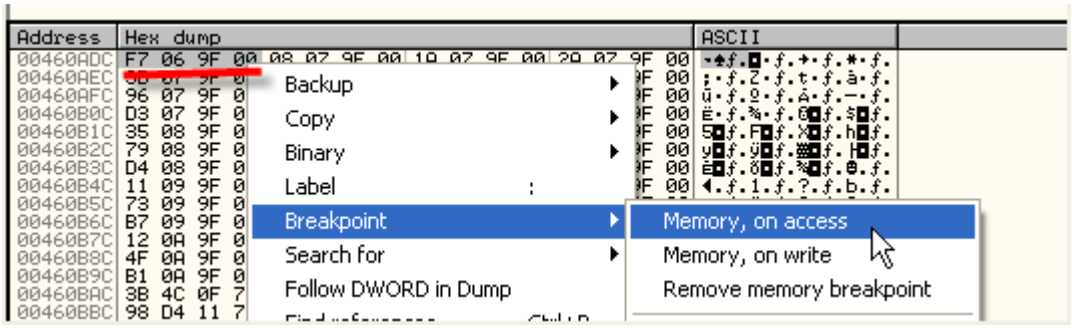


这里我们给 4271DC 处 CALL 指令以及其返回地址处分别设置一个断点。

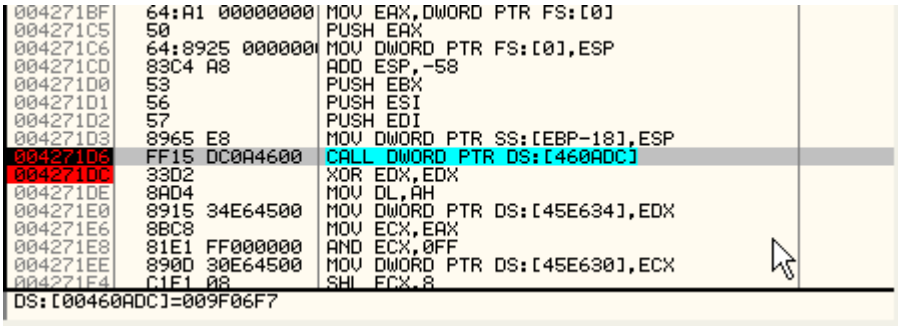


我们将之前定位 OEP 时设置的内存访问断点清除掉,接着给 460ADC 处的 IAT 项设置 4 字节的内存访问断点,运行起来,当该值被

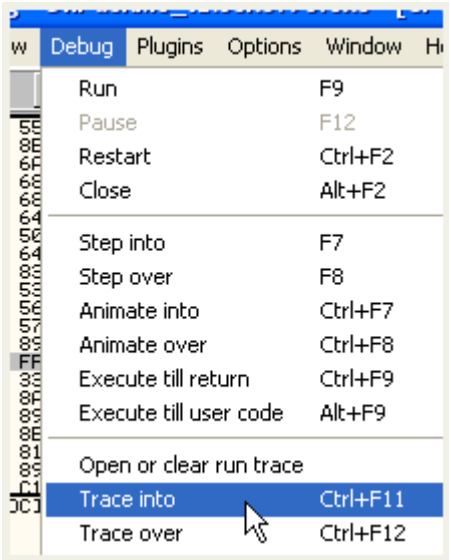
读取的时候,OD 就会断下来。



我们现在跟到了 4271D6 的 CALL 指令处。

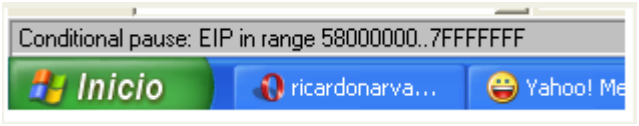


现在可以让 OD 自动跟踪了。

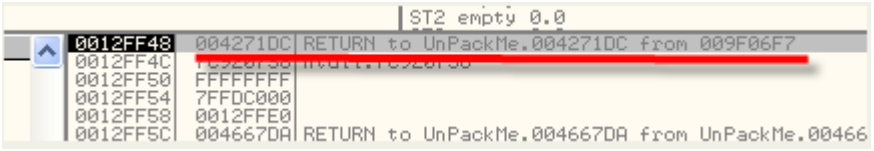


这里我们在反汇编窗口中单击鼠标右键,有两个自动跟踪的选项,第一个是 Trace into(自动单步步入),第二个是 Trace over(自动单步步过),这里我们选择-Trace into。

这里满足了停止跟踪的条件,OD 停了下来。



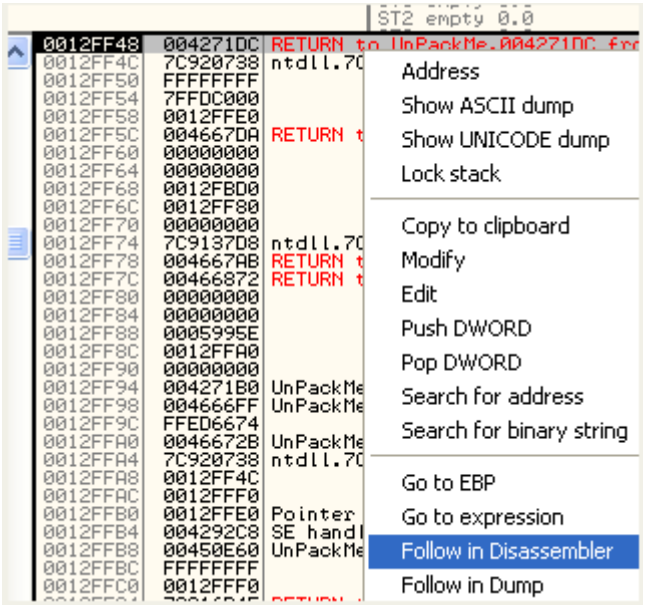
当 EIP 位于 58000000~7FFFFFFF 这个范围的时候,OD 就会断下来,为了保险起见,我们还是需要看一下 OD 的下方的提示信息,看看是不是由于其他异常原因导致断下来的。OD 下面的提示信息显示是由于 EIP 位于 58000000~7FFFFFFF 的范围内,满足了停止自动跟踪的条件,所以断了下来。



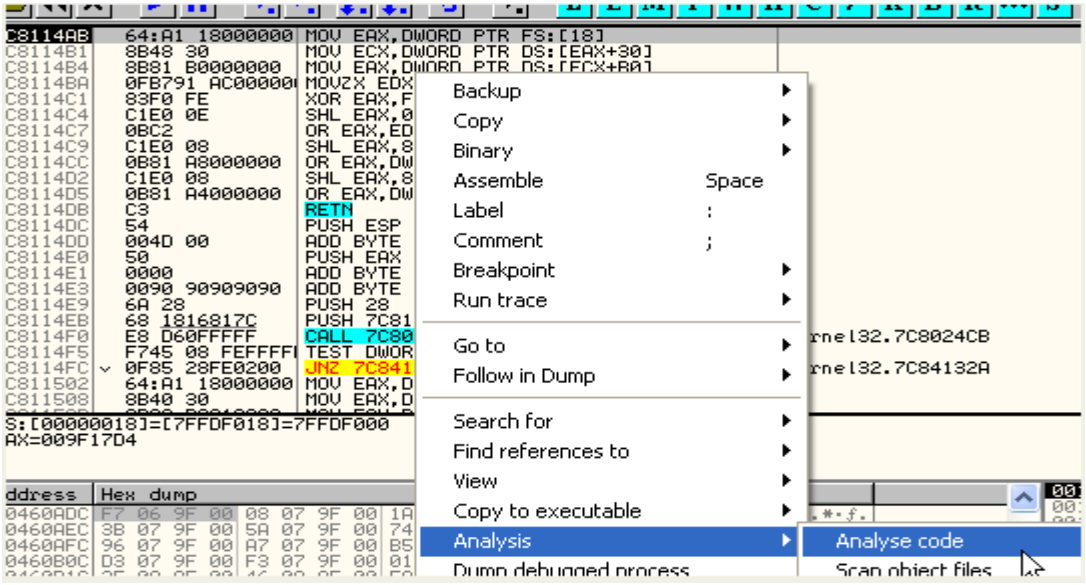
接下来的工作就是检查调用完该 API 函数以后是不是返回到我们刚刚设置了断点的返回地址处,因为有些壳会调用多个 API 函数来混淆视听,前面调用的几个 API 函数都是用来迷惑我们的,最后一个 API 函数才是其真正要调用的。

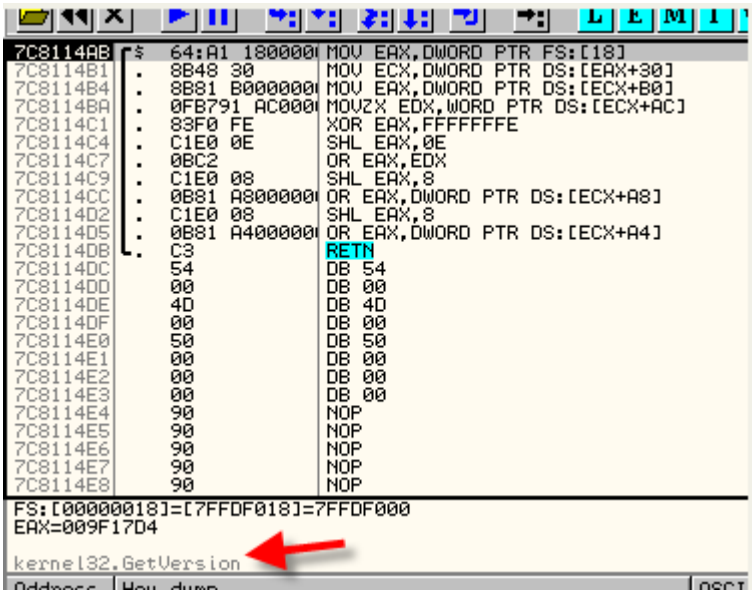
这里我们可以看到堆栈窗口显示的返回地址正好是我们刚刚设置了断点的那个返回地址 4271DC。

我们也可以在该返回地址上面单击鼠标右键选择 Follow in Disassembler 定位反汇编窗口中,也可以看到正好是我们设置了断点的返回地址处。

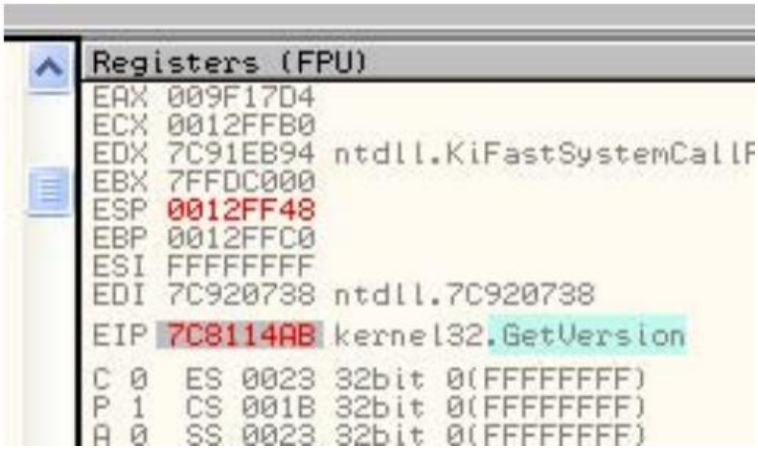


好了,现在停在了 API 函数的入口处,但是 OD 这里并没有提示 API 函数的名称,我们需要确定调用的是哪个 API 函数,我们来分析一下 DLL 的代码,在反汇编窗口中单击鼠标右键选择-Analysis-Analyse code。

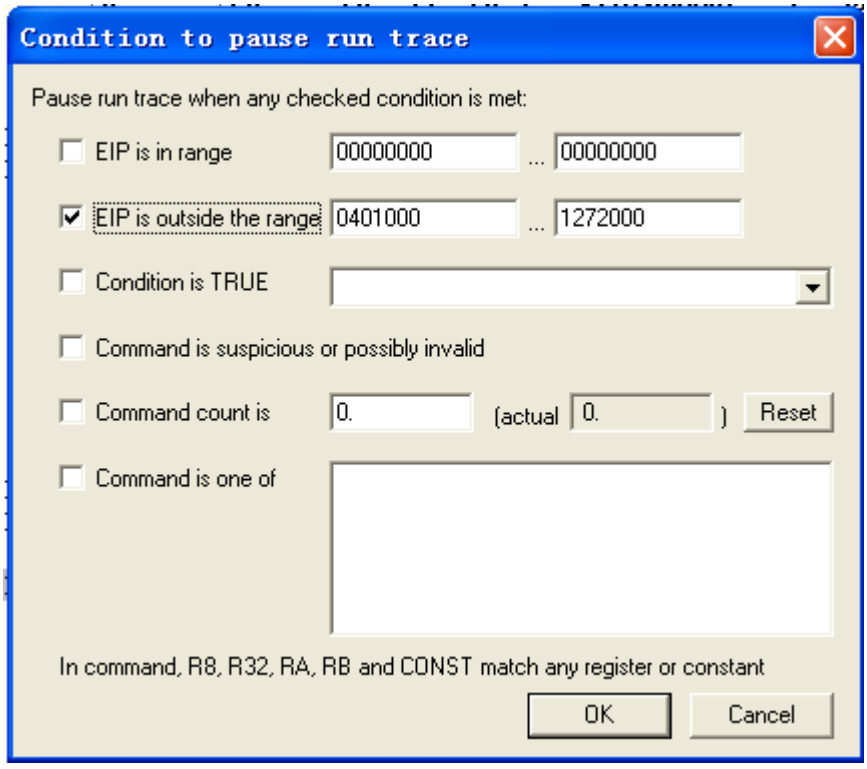




这里我们可以 OD 左下方的解释窗口中看到该 API 函数的名称为 Kernel32.GetVersion,EIP 寄存器中也显示了 API 函数的名称。



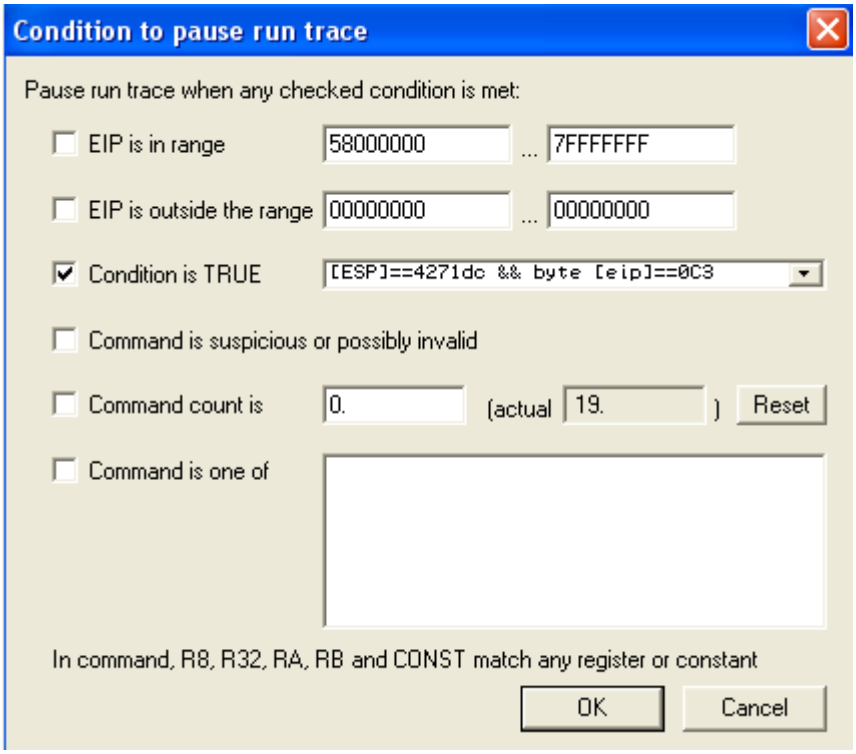
这里我们再来看下一个自动跟踪终止的条件选项 EIP is outside the range,通过设置这个选项我们也可以定位到其要调用的 API 函数:



(PS:这里原文笔误,作者写错了,作者写成了 401000 ~ 129000,因为是 401000 ~ 1272000 才对)

这里将 EIP is outside the range 这个选项的为设置为 401000~1272000,即当 EIP 超出了主程序所在区段的范围就会停止自动跟踪。

大家可以自行尝试。接下来还有一种情况。



有少数壳会将区段创建在系统 DLL 的区段之间,这样的话,我们需要用到 Condition is TRUE(当条件成立时)这个选项了,这里我们搜索 EIP 指向的指令为 RET,并且栈顶指针指向的是 4271DC 的指令。

这两个条件同时满足我们可以使用&&,表示这两个条件要同时满足,相当于 AND。

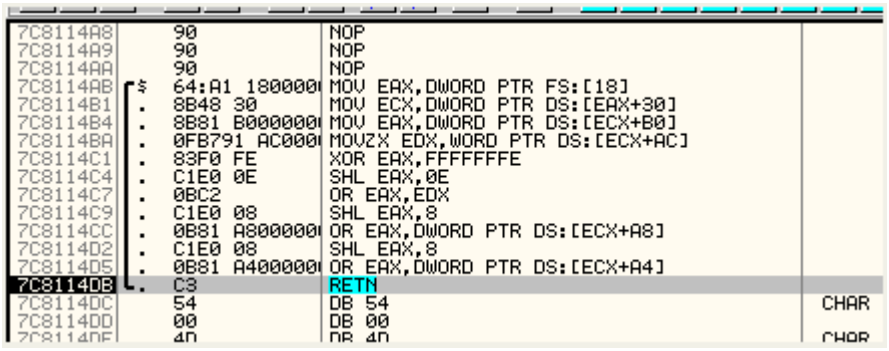
如果两个条件满足任意一个的话,我们可以使用||,相当于 OR。

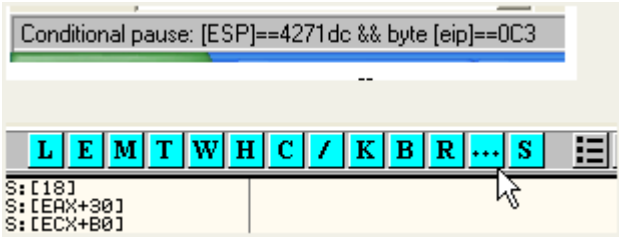
这里我们将自动跟踪终止条件设置为[ESP] == 4271DC && byte [EIP] == 0C3

[ESP] == 4271DC 即栈顶指针指向了返回地址 4271DC。

byte [EIP] == 0C3 即 EIP 指向了 API 函数的返回指令 RET。

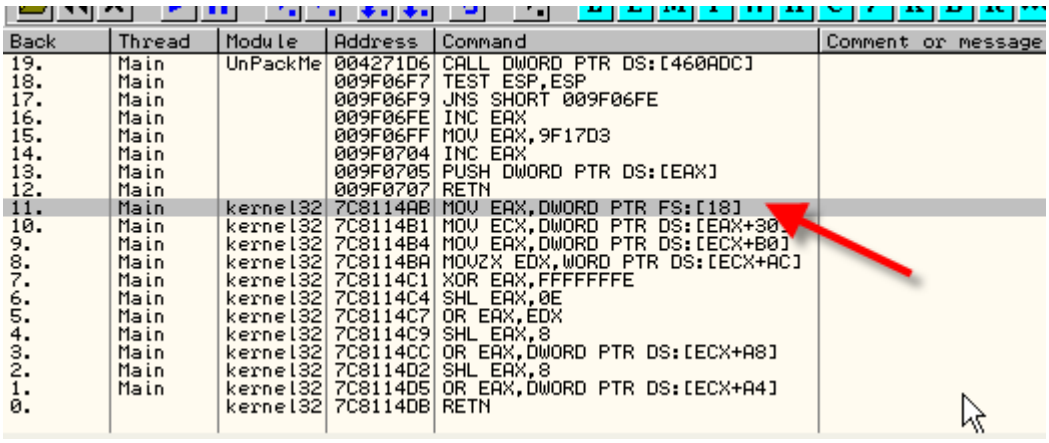
接着让 OD 自动跟踪,我们可以看到断在了 API 函数的返回指令 RET 处。



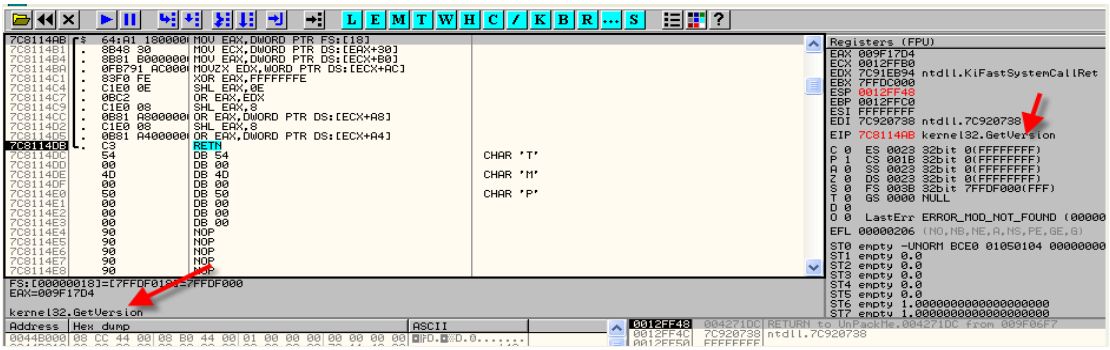


由于[ESP] == 4271DC,byte [EIP] == 0C3 这两个条件都满足了,所以 OD 停止了自动跟踪,现在我们位于 API 函数的返回指令 RET 处,这里由于我们并不在该 API 函数的入口地址处,所以 OD 并不会提示该 API 函数的名称,因此有些壳会自己模拟执行 API 函数开头的两,三条指令,然后从 API 函数的第四,五行开始指令,这样 OD 也不会提示该 API 函数的名称,尽管如此,我们还是有办法可以定位该 API 函数的名称。

我们单击工具栏中的...按钮查看自动跟踪的日志信息。



我们看到红色箭头标注的这一行也没有显示 API 函数名称,我们分析一下代码,然后在该行上双击鼠标左键。



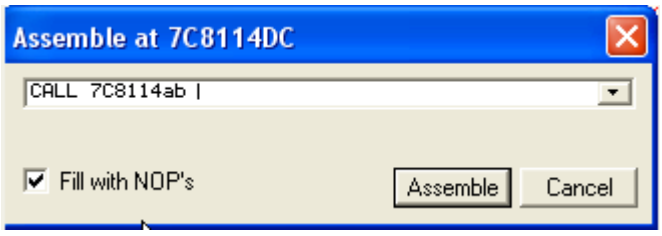
这里我们可以看到 OD 中 EIP 寄存器显示了该 API 函数的名称,当前 EIP 的值显示为红色,表示 EIP 寄存器被修改了,如果我们按减号键的话,EIP 会变为前一条执行过的指令的地址。

有些壳会去模拟执行 API 函数前几条指令,然后再去指令 API 函数后面的代码,这样 OD 也不会提示该 API 函数的名称。

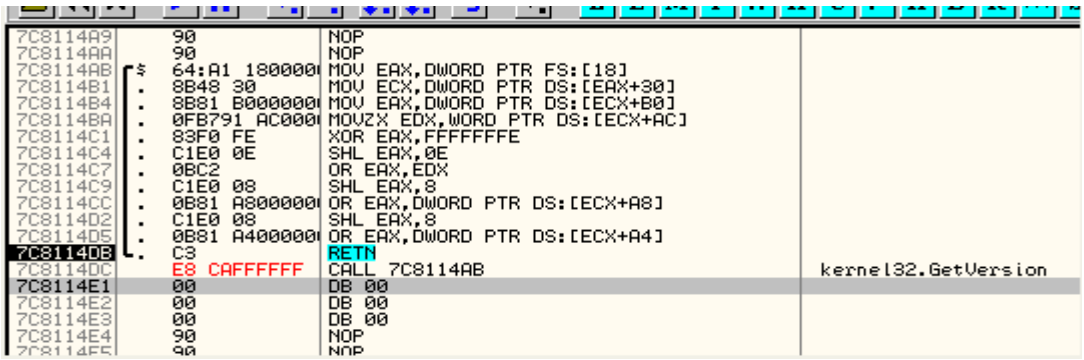
我们来看看如何应对这种情况:

我们在 API 函数返回指令 RET 的下一行指令空白处单击空格键,CALL 我们怀疑的 API 函数入口地址,这里我们输入:

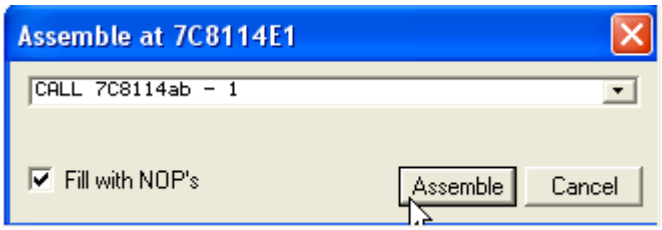
CALL 7C8114AB。

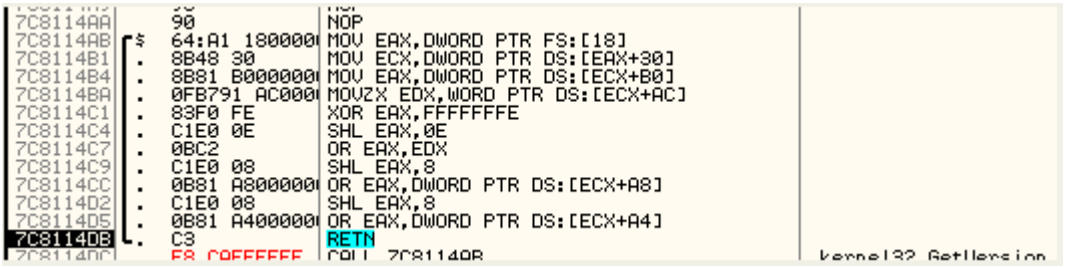


我们可以看到此时 OD 提示 API 函数的名称为 Kernel32.GetVersion。

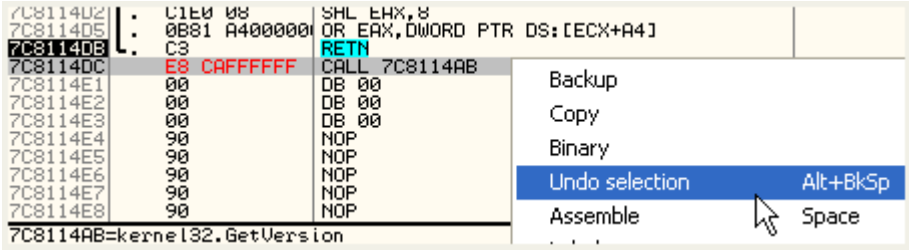


如果我们 CALL 不是 API 函数的入口地址的话,OD 就不会显示函数名称了。



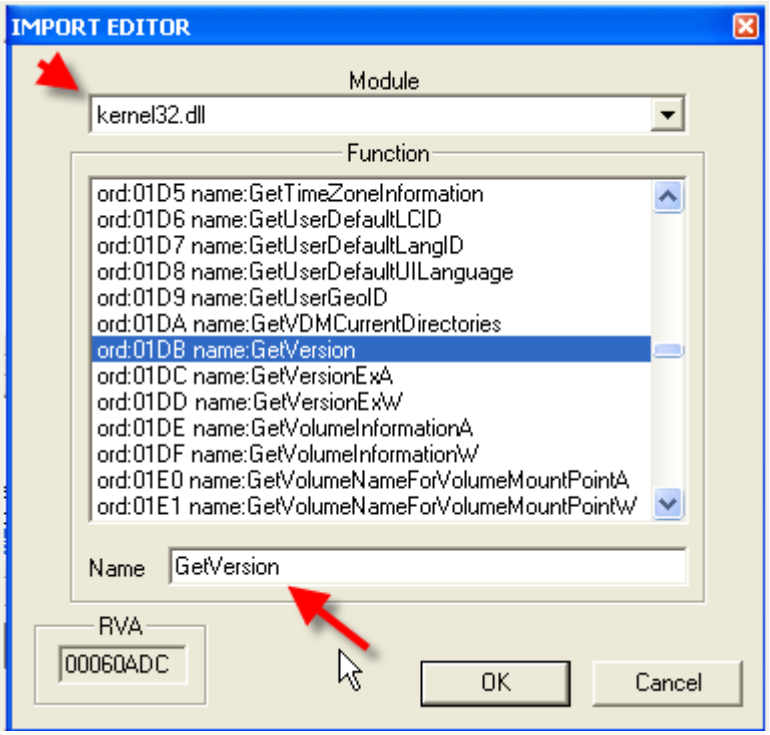


我们单击鼠标右键选择-undo selection(撤销刚刚所做的修改)。

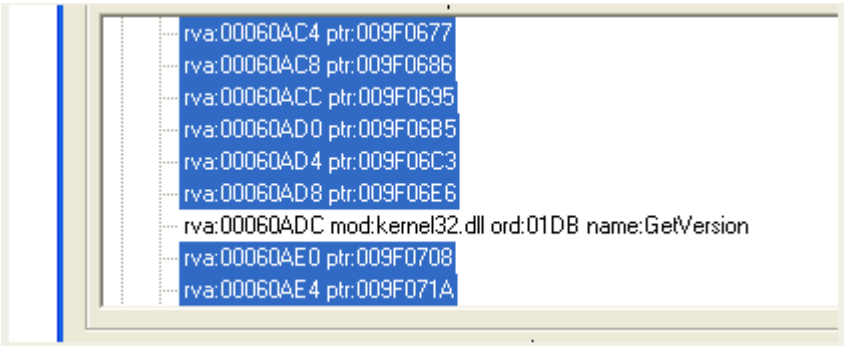


好了,我们现在知道如何手工定位 API 函数的名称了,打开 IMP REC,我们需要对重定位过的 IAT 项进行修复,在 60ADC 这个无效

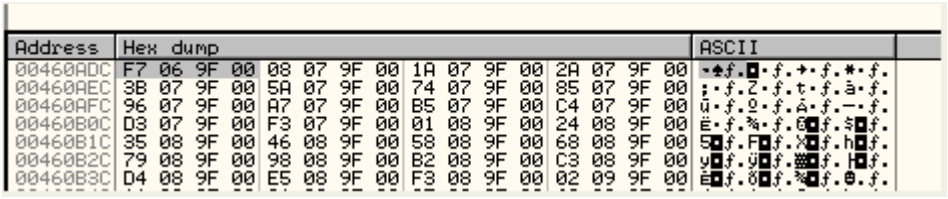
的 IAT 项上面双击鼠标左键,在弹出的窗口中选中 API 函数所在的模块以及其函数名称。



单击确定。

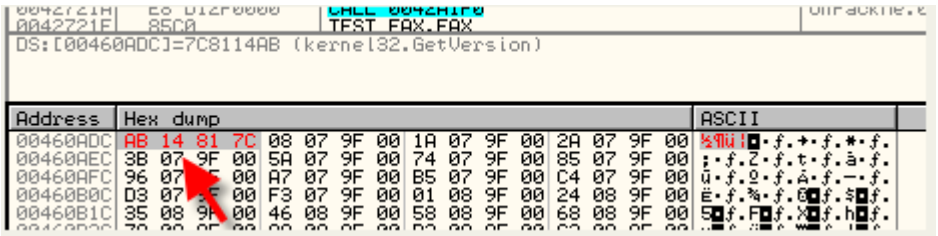


现在我们单击 Show Invaild 按钮,可以看到 60ADC 这一项已经被标记为有效了,我们要的就是这个效果。



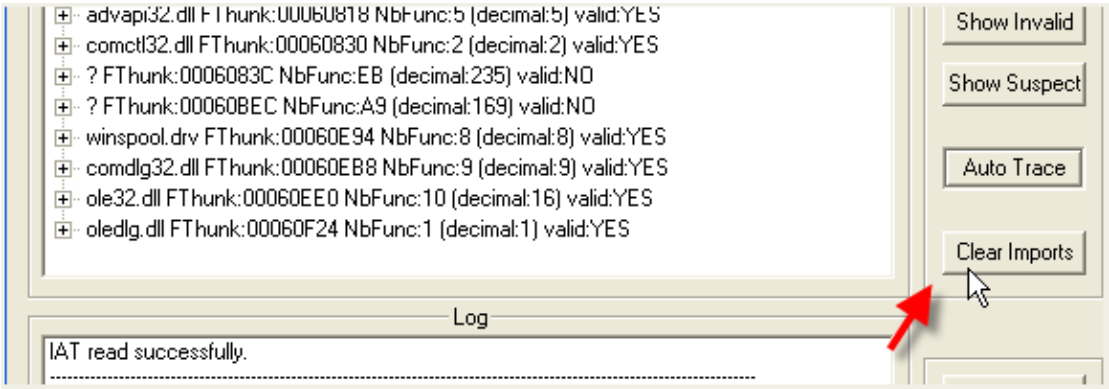
另一种修复的方式就是直接在重定向过的 IAT 项中填充上正确的 API 函数地址。

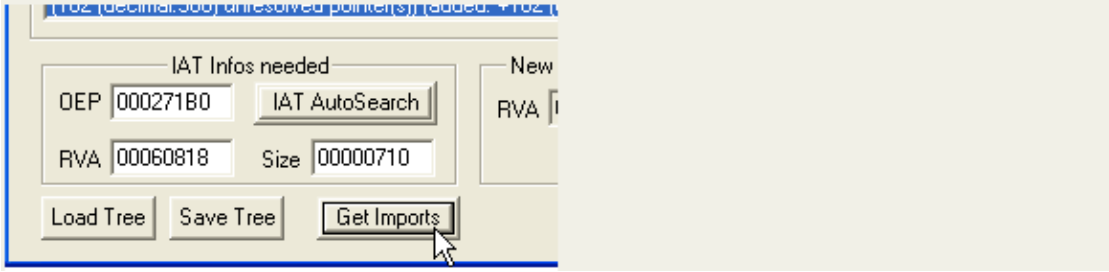
这里我们在 460ADC 这一项上填入 GetVersion 函数的地址,我的机器上该函数的入口地址为 7C8114AB。

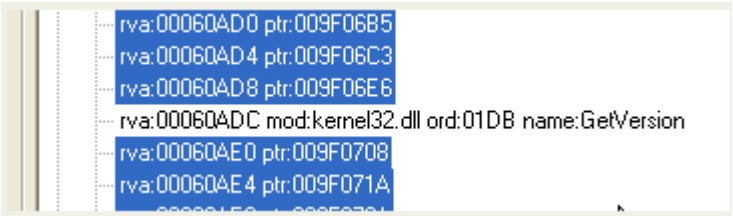


现在我们来到 IMP REC 窗口中,单击 Clear Imports 按钮。然后再单击 Get Imports 按钮,可以看到 60ADC(RVA)这一项也被标记为

了有效。



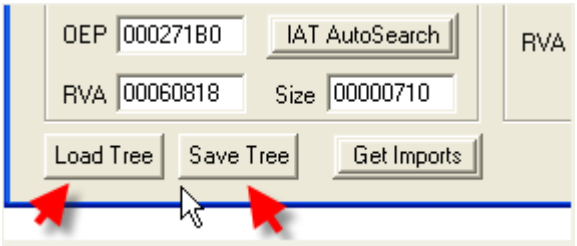




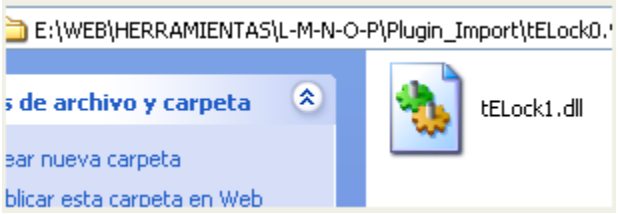
对于其他的重定向过的 IAT 项我们也可以按照上述方法来进行修复,接着就可以修复 dump 文件了。这种方法显得比较枯燥,纯属体力活。

(PS:大家想干这种体力活吗,不管你们想不想,反正我不想,嘿嘿)

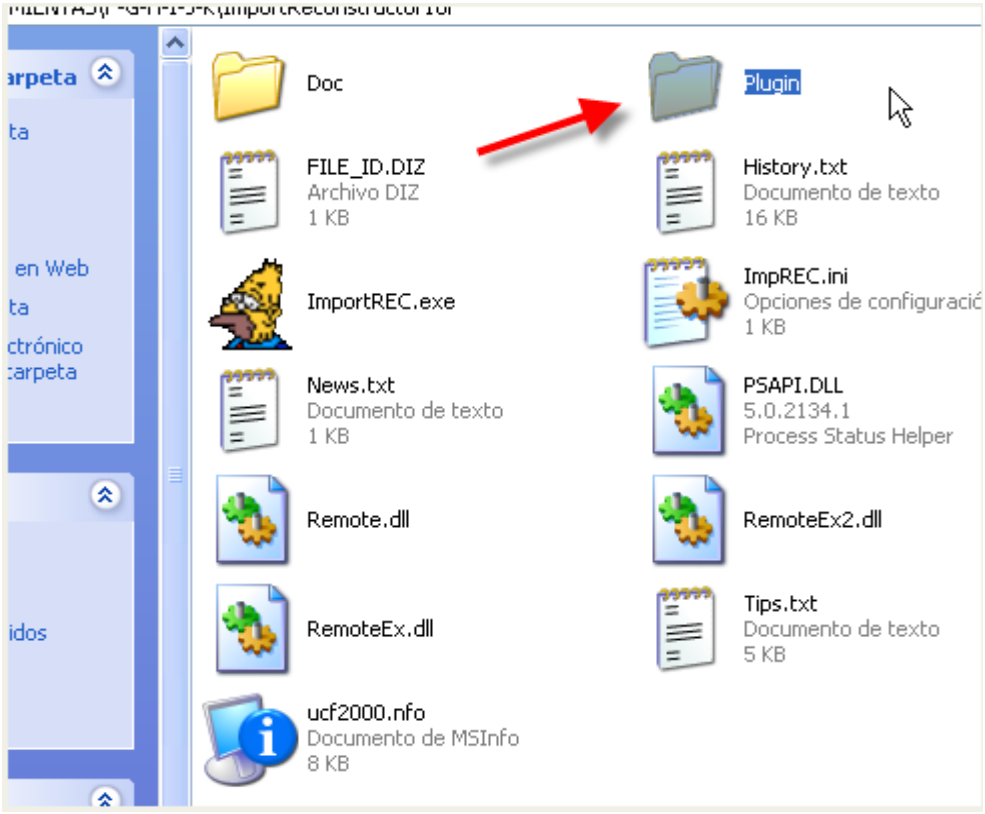
IMP REC 提示了一个功能可以协助我们完成这个手工修复工作。通过单击 Save Tree 按钮可以保存我们修复过的 IAT 项,当我们下次想继续修复其他项的时候可以通过单击 Load Tree 按钮导入我们之前保存的 IAT 信息,这样就可以继续之前没有完成的修复工作了。



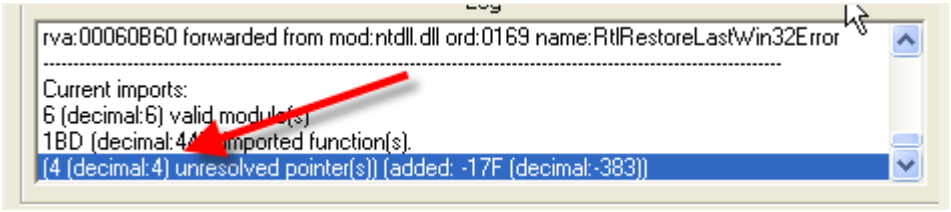
接下来的一种修复重定向的 IAT 项的方法就是借助于 IMP REC 的插件 tELock1,我们将其放到 IMP REC 目录下的 Plugin 文件下。下来我们来看看这个插件怎么用。



将插件的 DLL 拷贝到 IMP REC 中 Plugin 文件夹中。

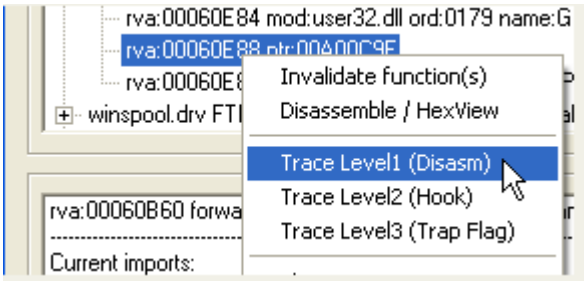


现在我们重启 IMP REC,输入 OEP,RVA,SIZE 等数据,单击 Get Imports,接着单击 Show Invaild,在无效的项上面单击鼠标右键选择-Plugin Tracers-tElock1。这样该插件就会帮我们修复这些重定向的 IAT 项,我们可以看到日志窗口提示除了 4 项以外,其他重定向的IAT项都被修复了,这个插件不是很爽,帮我们完成大部分的重定向项的修复工作,剩下少数几个重定向过IAT项我们可以自己手工修复。

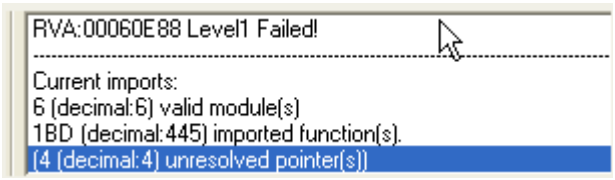


这里 IMP REC 提示 4 项未修复的,大家可以按照前面介绍的方法进行手工修复,这种插件的修复方式有明显的局限性,只能针对于特定的壳。

IMP REC 还自带有常规的 Tracers,我们可以试试看其能不能修复这些重定向过的 IAT 项,大多数情况,通常它也可以替代我们完成手工修复任务。



这里我们在第一个修复失败的项上面单击鼠标右键,我们可以看到 3 个等级,这里我们选择-Trace Leve1(Disasm),看看等级 1 能不能起作用。



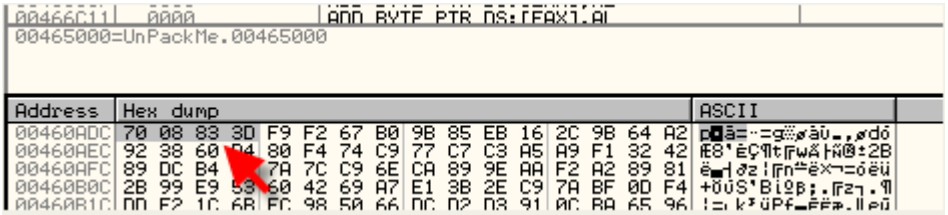
失败了,我们继续尝试 Level2,Level3,我们会发现这两项也不起作用,IMP REC 死掉了,看来这几个选项只能对比较简单的壳起作用。

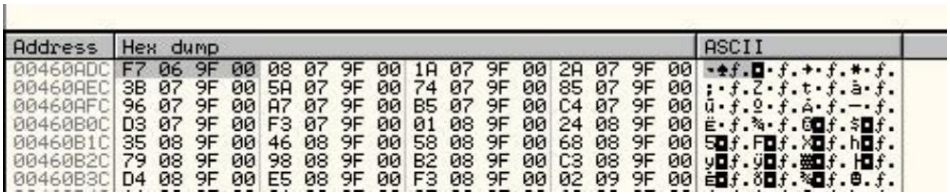
接下来的这种修复重定向的方法叫做关键跳法,这种方法在很多脱壳教程中都有用到。

这种方法就是定位壳填充 IAT 的时机,看看何时填充正确 IAT 项,何时填充重定向过的 IAT 项。

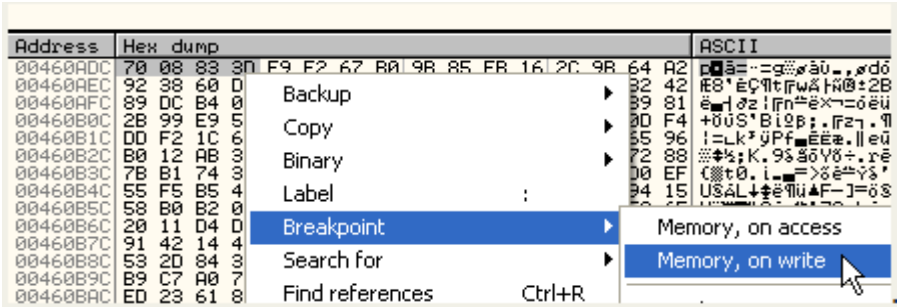
举个例子:就拿 GetVersion 这一项来说吧,我们重启 UnPackMe-tElock0.98。

此时我们还没有到达 OEP 处,460ADC 中的值为 3D830870,在壳的解密例程执行过程中会将重定向过的值 9F06F7 填充到 460ADC 这个内存单元中。

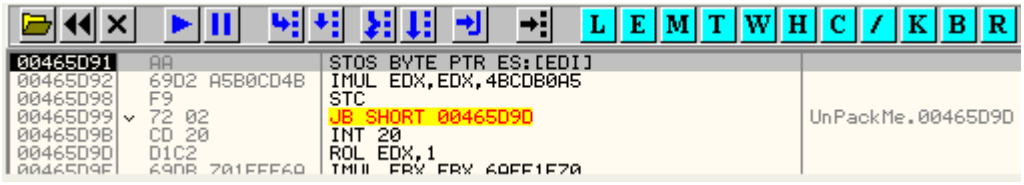




为了能够定位何时该 IAT 项被写入,我们可以对 460ADC 设置硬件写入断点,但是由于该壳会检测硬件断点,所以这里我们设置内存写入断点,让壳在此处写入重定向过的 IAT 值的时候断下来。

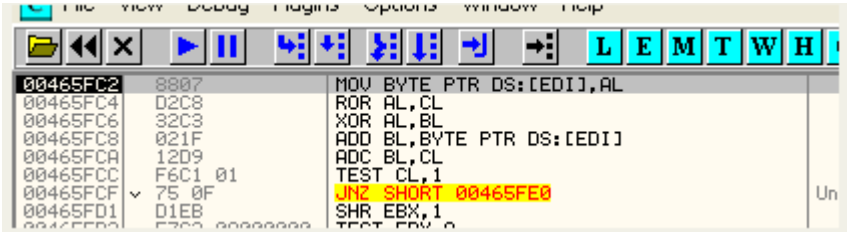


运行起来,我们可以看到断在了这里。

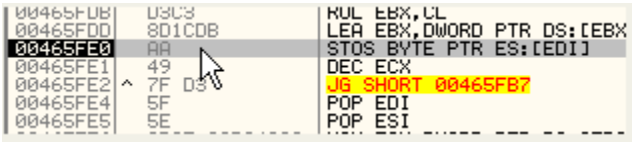


这里并不是我们要定位的点,因为我们按 F8 键执行这一行会发现重定向过 IAT 值并没有被写进来,我们继续运行。

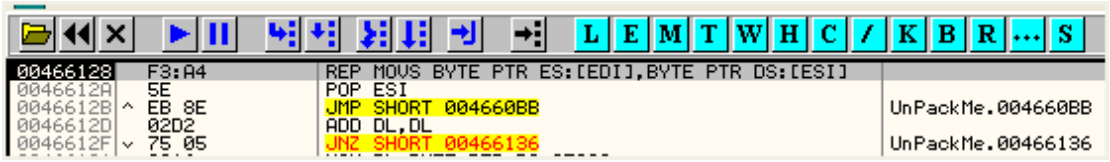
我们多运行几次,断在了这里:



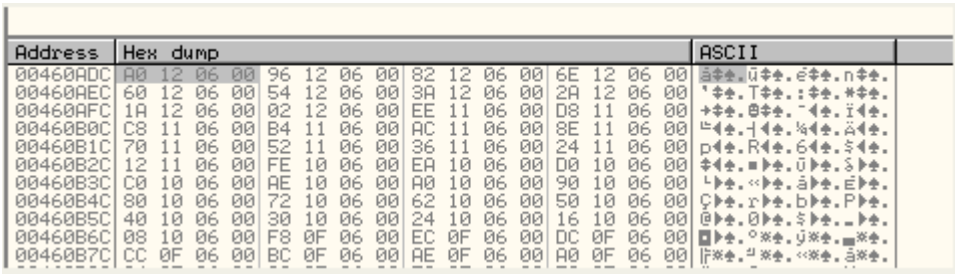
这里也不是写入重定向过的 IAT 值,我们继续运行。



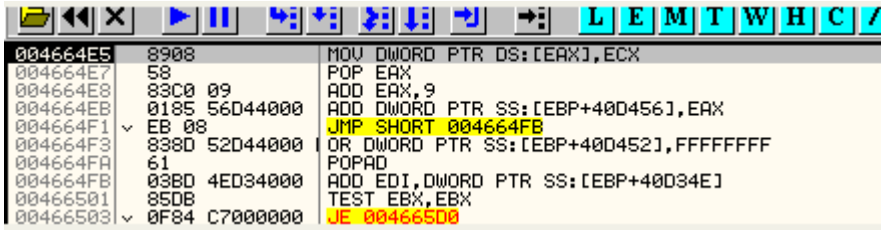
我们按 F8 键执行这行指令,发现也不是保存重定向过的 IAT 值,继续运行。



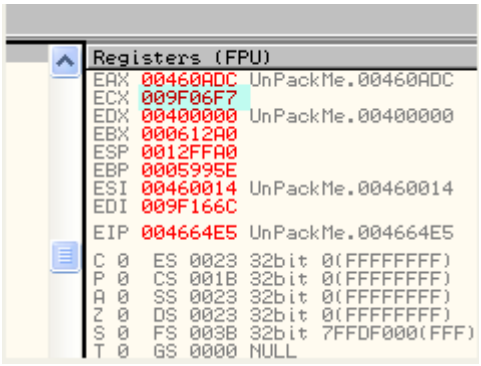
断在了这里,这里将写入一个无效的值,但是并不是我们要定位的重定向过的 IAT 值 9F06F7。



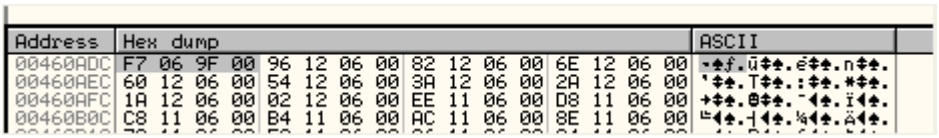
我们执行 REP MOVS 指令,该无效的值被写入到该 IAT 项中,我们继续运行。



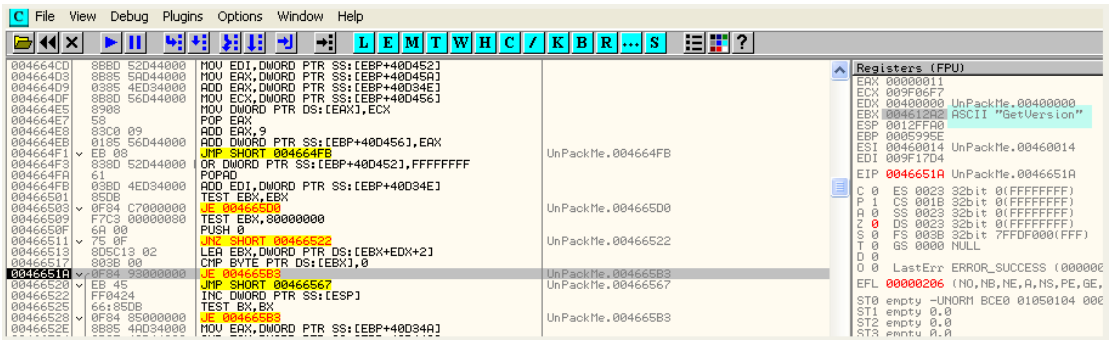
我们断在了这里,我们可以看到当前 ECX 寄存器的值正好等于重定向过的 IAT 值 9F06F7。



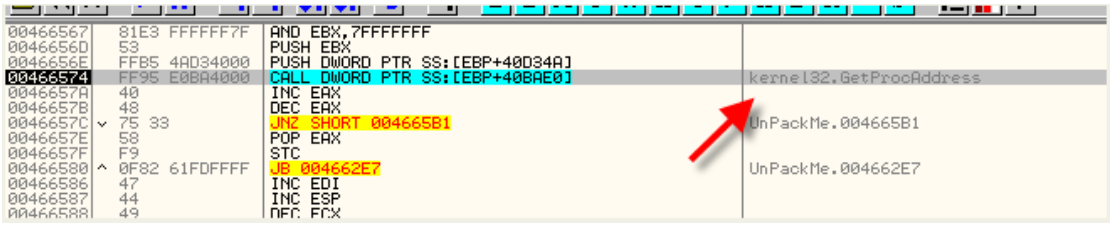
ECX = 9F06F7,将被保存到 EAX = 460ADC 指向的 IAT 中,这里就是我们要定位的地方。



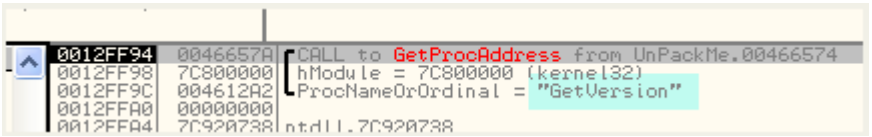
我们按下 F8 键,9F06F7 就会被保存到 460ADC 中,这只是第一步,接下来我们需要定位关键跳转。



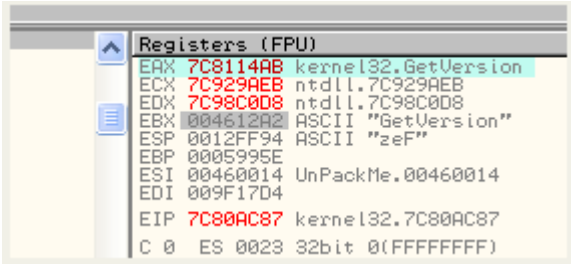
我们往下跟几步,达到了 46651A 处 JE 指令处,我们看看寄存器窗口中 EBX 寄存器的值,当前 EBX 正好指向了 GetVersion 的函数名称字符串。



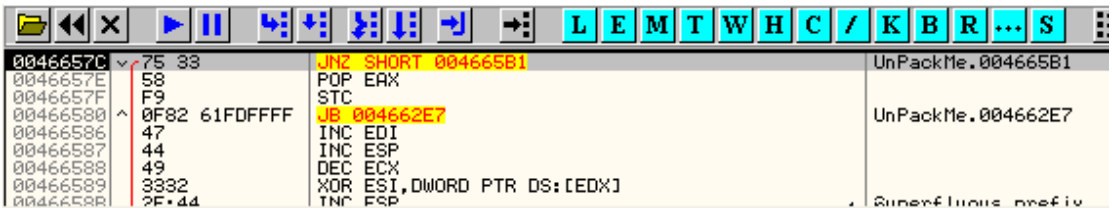
继续往下我们可以看到会调用 GetProcAddress 获取 GetVersion 这个 API 函数的地址。



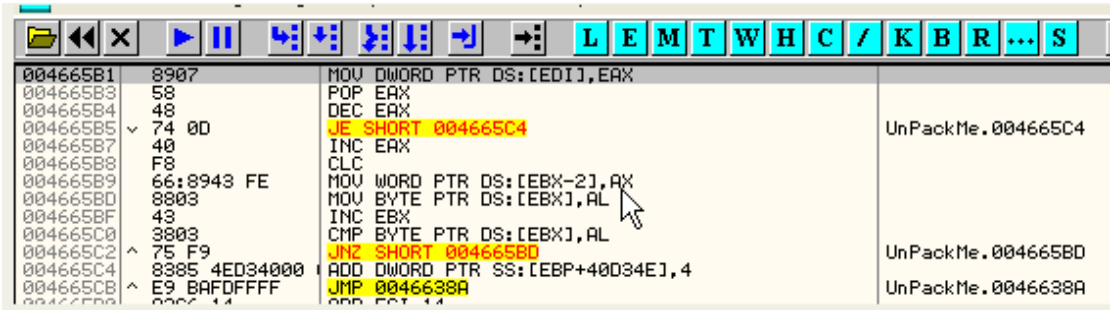
我们可以在堆栈窗口中看到参数情况,按 F8 键执行这个 CALL。



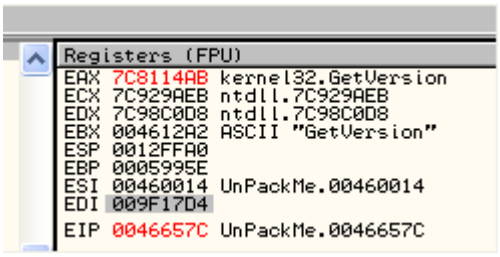
我们可以看到此时 EAX 保存了 GetVersion 这个 API 函数的地址,我们继续往下跟。



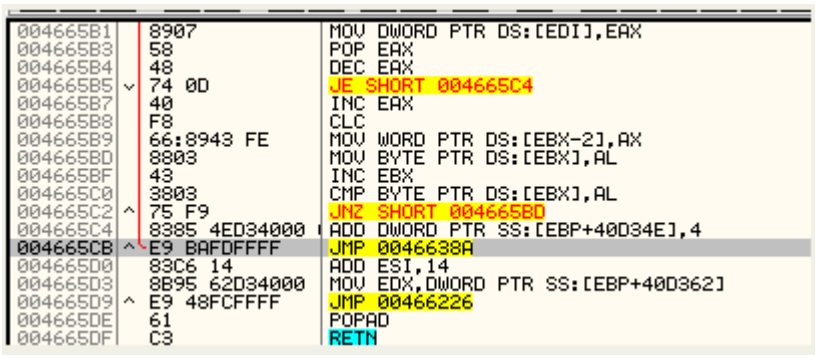
这里有一个条件跳转,这个条件跳转是成立的,我们不跟过去,直接单击鼠标右键选择-Follow。



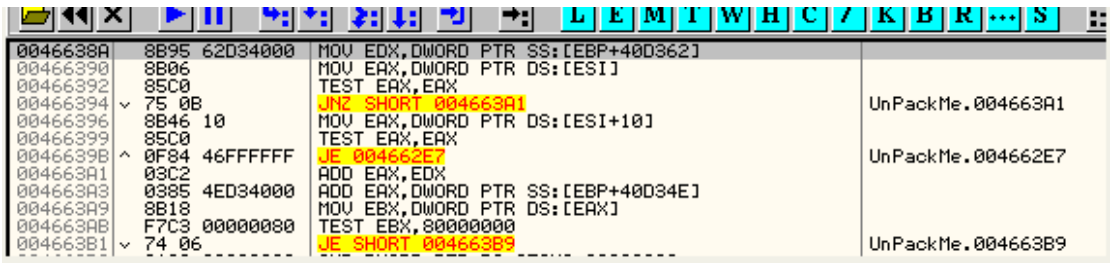
这里我们可以看到 GetVersion 的函数地址并不是被填充到 IAT 中,而是被写入到 EDI 指向的内存单元中,当前 EDI 为:



保存了该 API 函数的地址以后,紧接着我们到了 JMP 指令处。



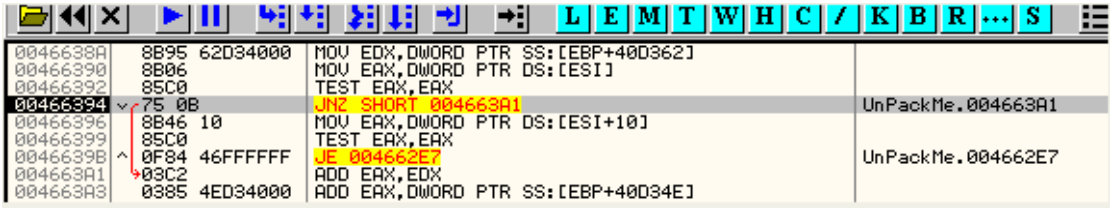
这里通过这个 JMP 我们又将回到这个过程的开始处,这里是一个循环,我们单击鼠标右键选择-Follow。

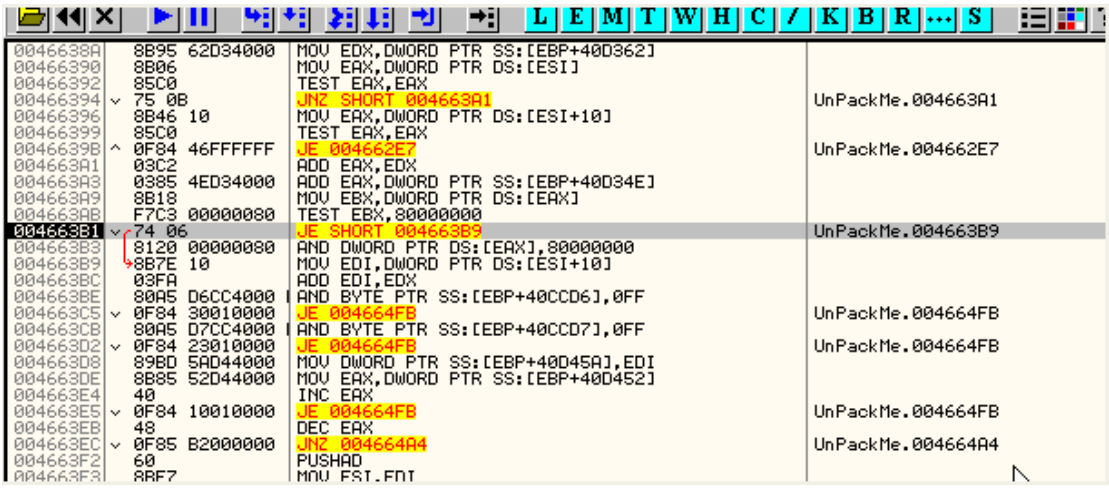


这里接下来又要将第二个重定向过的 IAT 值填充的下一个 IAT 项中,然后通过 GetProcAddress 获取对应的 API 函数地址,并将获取到的 API 函数地址保存到某处,接着又是第三个重定向过的 IAT 值,循环往复,直到所有 IAT 项都被处理完毕为止。

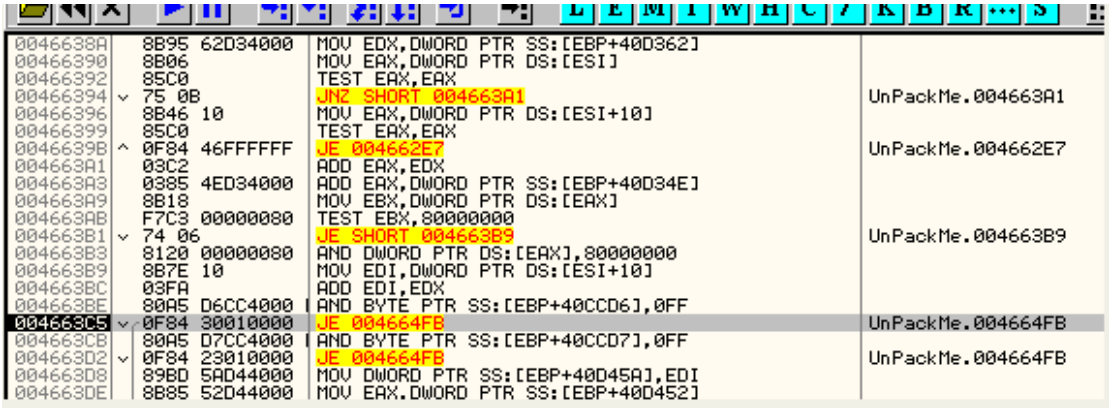
由于壳只是对一部分 IAT 项进行重定向,所以下面我们来看看壳对 IAT 项重定向以及不重定向流程的差异。

IAT 是如何被重定向的

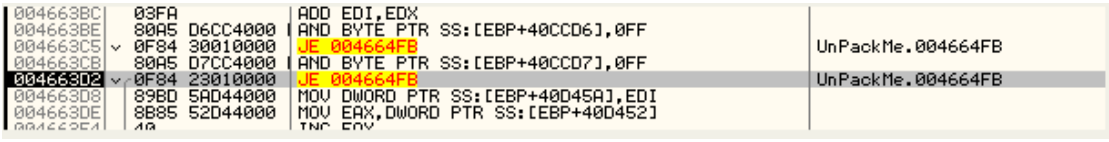




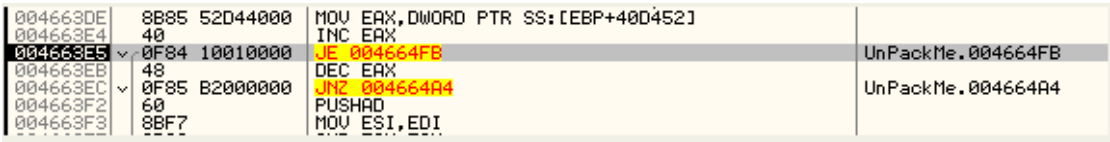
```
0046638A   8B95 62D34000   MOV EDX,DWORD PTR SS:[EBP+40D362]
00466390   8B06            MOV EAX,DWORD PTR DS:[ESI]
00466392   85C0            TEST EAX,EAX
00466394   75 0B           JNZ SHORT 004663A1                  UnPackMe.004663A1
00466396   8B46 10         MOV EAX,DWORD PTR DS:[ESI+10]
00466399   85C0            TEST EAX,EAX
0046639B   0F84 46FFFFFF   JE 004662E7                         UnPackMe.004662E7
004663A1   03C2            ADD EAX,EDX
004663A3   0385 4ED34000   ADD EAX,DWORD PTR SS:[EBP+40D34E]
004663A9   8B18            MOV EBX,DWORD PTR DS:[EAX]
004663AB   F7C3 00000080   TEST EBX,80000000
004663B1   74 06           JE SHORT 004663B9                   UnPackMe.004663B9
004663B3   8120 00000080   AND DWORD PTR DS:[EAX],80000000
004663B9   8B7E 10         MOV EDI,DWORD PTR DS:[ESI+10]
004663BC   03FA            ADD EDI,EDX
004663BE   80A5 D6CC4000   AND BYTE PTR SS:[EBP+40CCD6],0FF
004663C5   0F84 30010000   JE 004664FB                         UnPackMe.004664FB
004663CB   80A5 D7CC4000   AND BYTE PTR SS:[EBP+40CCD7],0FF
004663D2   0F84 23010000   JE 004664FB                         UnPackMe.004664FB
004663D8   89BD 5AD44000   MOV DWORD PTR SS:[EBP+40D45A],EDI
004663DE   8B85 52D44000   MOV EAX,DWORD PTR SS:[EBP+40D452]
004663E4   40              INC EAX
004663E5   0F84 10010000   JE 004664FB                         UnPackMe.004664FB
004663EB   48              DEC EAX
004663EC   0F85 B2000000   JNZ 004664A4                        UnPackMe.004664A4
004663F2   60              PUSHAD
004663F3   8BF7            MOV ESI,EDI
```



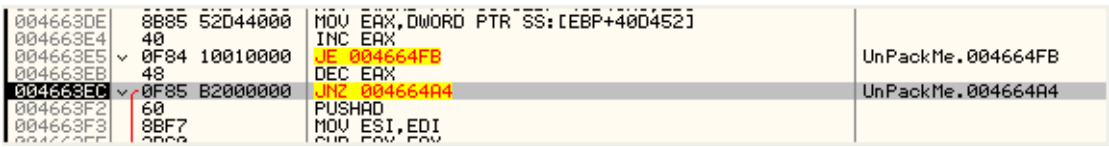
```
0046638A   8B95 62D34000   MOV EDX,DWORD PTR SS:[EBP+40D362]
00466390   8B06            MOV EAX,DWORD PTR DS:[ESI]
00466392   85C0            TEST EAX,EAX
00466394   75 0B           JNZ SHORT 004663A1                  UnPackMe.004663A1
00466396   8B46 10         MOV EAX,DWORD PTR DS:[ESI+10]
00466399   85C0            TEST EAX,EAX
0046639B   0F84 46FFFFFF   JE 004662E7                         UnPackMe.004662E7
004663A1   03C2            ADD EAX,EDX
004663A3   0385 4ED34000   ADD EAX,DWORD PTR SS:[EBP+40D34E]
004663A9   8B18            MOV EBX,DWORD PTR DS:[EAX]
004663AB   F7C3 00000080   TEST EBX,80000000
004663B1   74 06           JE SHORT 004663B9                   UnPackMe.004663B9
004663B3   8120 00000080   AND DWORD PTR DS:[EAX],80000000
004663B9   8B7E 10         MOV EDI,DWORD PTR DS:[ESI+10]
004663BC   03FA            ADD EDI,EDX
004663BE   80A5 D6CC4000   AND BYTE PTR SS:[EBP+40CCD6],0FF
004663C5   0F84 30010000   JE 004664FB                         UnPackMe.004664FB
004663CB   80A5 D7CC4000   AND BYTE PTR SS:[EBP+40CCD7],0FF
004663D2   0F84 23010000   JE 004664FB                         UnPackMe.004664FB
004663D8   89BD 5AD44000   MOV DWORD PTR SS:[EBP+40D45A],EDI
004663DE   8B85 52D44000   MOV EAX,DWORD PTR SS:[EBP+40D452]
```



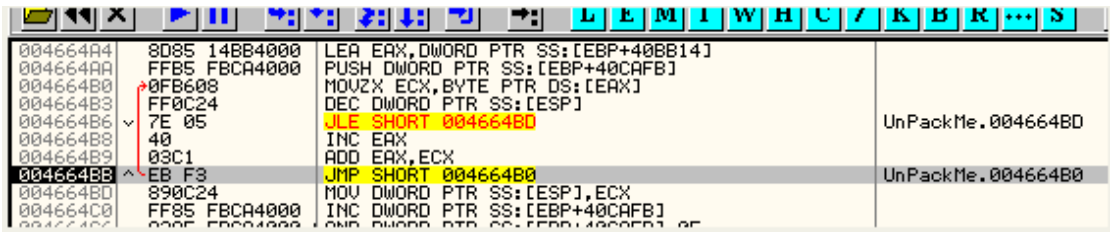
```
004663BC   03FA            ADD EDI,EDX
004663BE   80A5 D6CC4000   AND BYTE PTR SS:[EBP+40CCD6],0FF
004663C5   0F84 30010000   JE 004664FB                         UnPackMe.004664FB
004663CB   80A5 D7CC4000   AND BYTE PTR SS:[EBP+40CCD7],0FF
004663D2   0F84 23010000   JE 004664FB                         UnPackMe.004664FB
004663D8   89BD 5AD44000   MOV DWORD PTR SS:[EBP+40D45A],EDI
004663DE   8B85 52D44000   MOV EAX,DWORD PTR SS:[EBP+40D452]
```



```
004663DE   8B85 52D44000   MOV EAX,DWORD PTR SS:[EBP+40D452]
004663E4   40              INC EAX
004663E5   0F84 10010000   JE 004664FB                         UnPackMe.004664FB
004663EB   48              DEC EAX
004663EC   0F85 B2000000   JNZ 004664A4                        UnPackMe.004664A4
004663F2   60              PUSHAD
004663F3   8BF7            MOV ESI,EDI
```

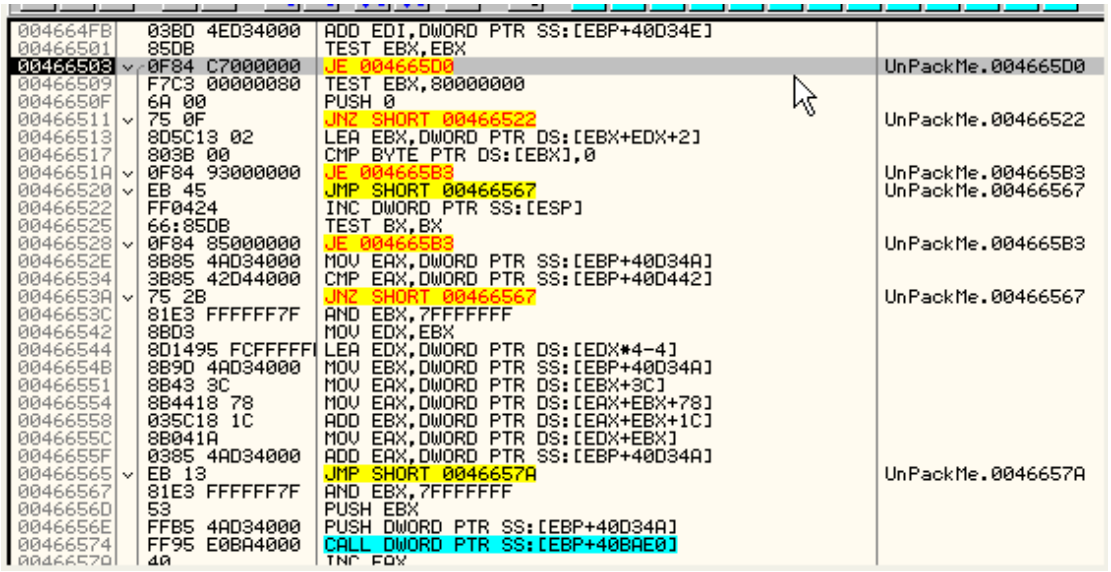


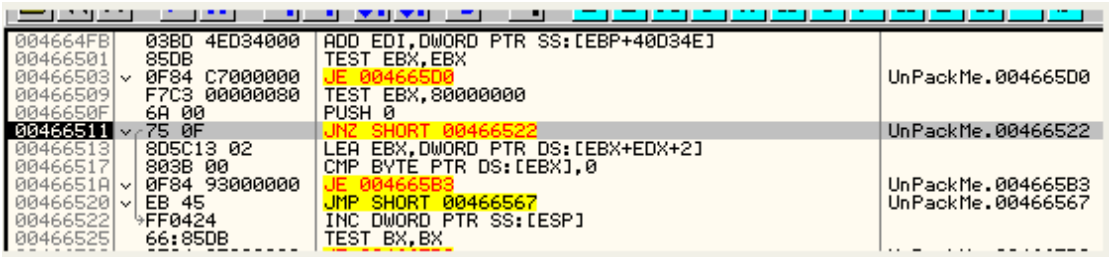
```
004663DE   8B85 52D44000   MOV EAX,DWORD PTR SS:[EBP+40D452]
004663E4   40              INC EAX
004663E5   0F84 10010000   JE 004664FB                         UnPackMe.004664FB
004663EB   48              DEC EAX
004663EC   0F85 B2000000   JNZ 004664A4                        UnPackMe.004664A4
004663F2   60              PUSHAD
004663F3   8BF7            MOV ESI,EDI
```

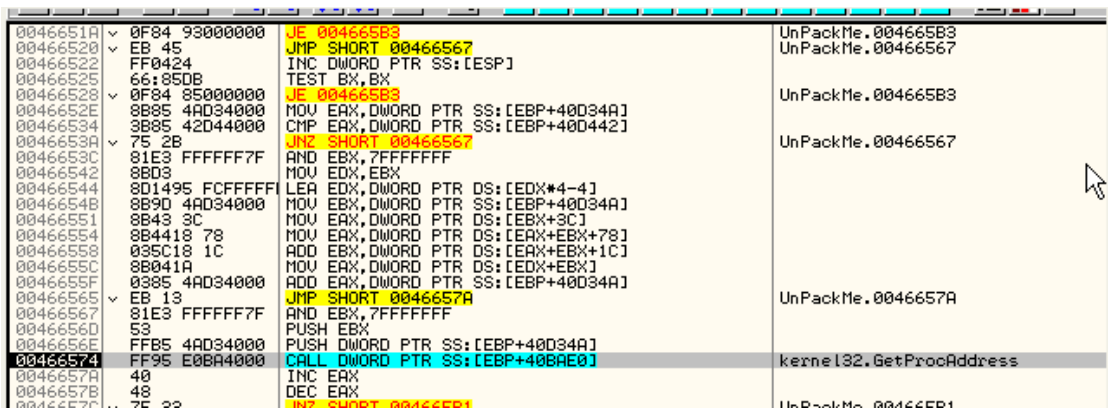


```
004664A4   8D85 14BB4000   LEA EAX,DWORD PTR SS:[EBP+40BB14]
004664AA   FFB5 FBCA4000   PUSH DWORD PTR SS:[EBP+40CAFB]
004664B0   0FB608          MOVZX ECX,BYTE PTR DS:[EAX]
004664B3   FF0C24          DEC DWORD PTR SS:[ESP]
004664B6   7E 05           JLE SHORT 004664BD                  UnPackMe.004664BD
004664B8   40              INC EAX
004664B9   03C1            ADD EAX,ECX
004664BB   EB F3           JMP SHORT 004664B0                  UnPackMe.004664B0
004664BD   890C24          MOV DWORD PTR SS:[ESP],ECX
004664C0   FF85 FBCA4000   INC DWORD PTR SS:[EBP+40CAFB]
```

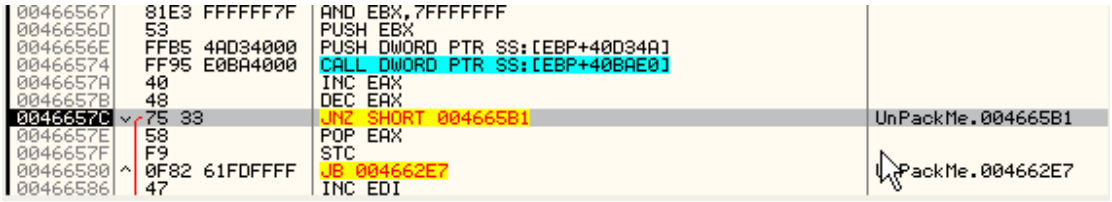
这里是一个小循环,继续往下:

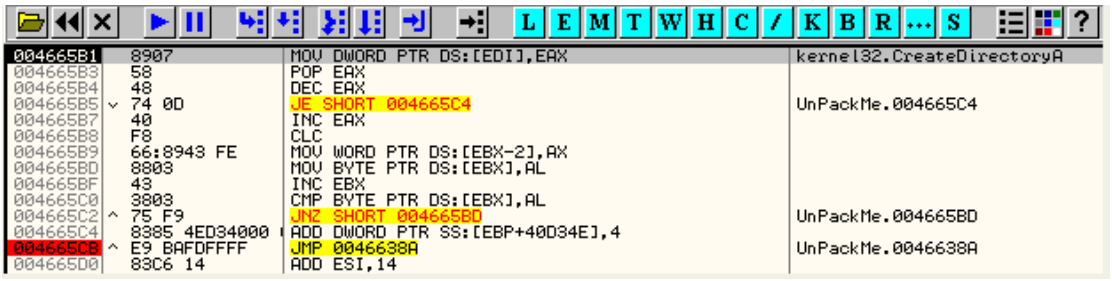






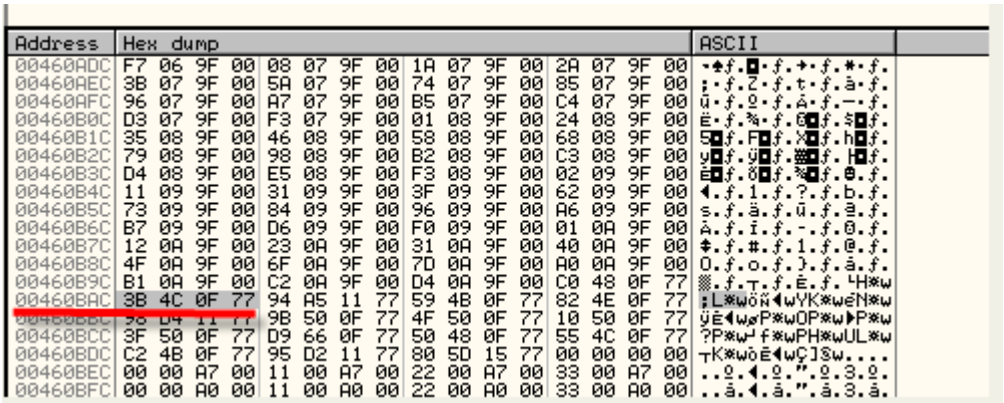
这里是调用 GetProcAddress,接着就会到保存 GetProcAddress 获取到的 API 函数地址处。

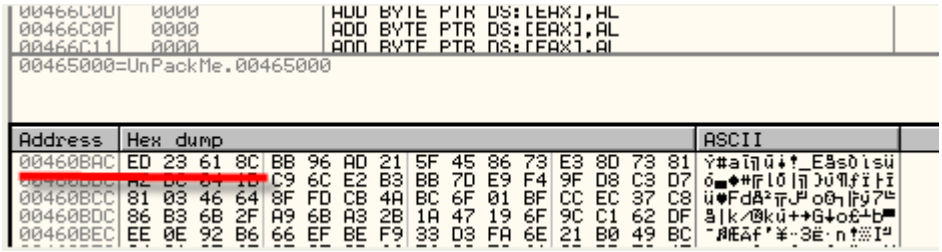




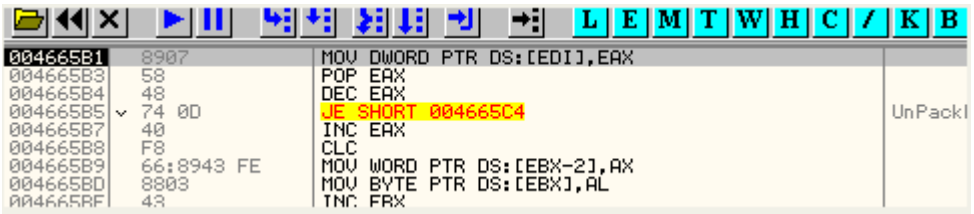
以上就是跟踪一个 IAT 项被重定向的完成流程,其中有很多的条件跳转。现在我们跟到 OEP 处,定位到一个正确的 IAT 项,比较一下两者有什么差别。

这里我们选择 460BAC 这一项。重新启动程序。

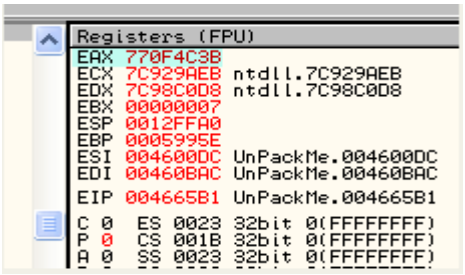




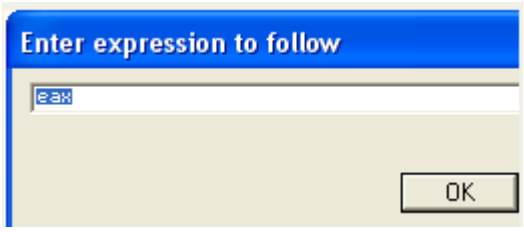
对 460BAC 这 4 个字节设置内存写入断点,运行起来,看看壳何时会将正确的 IAT 项填充进来。

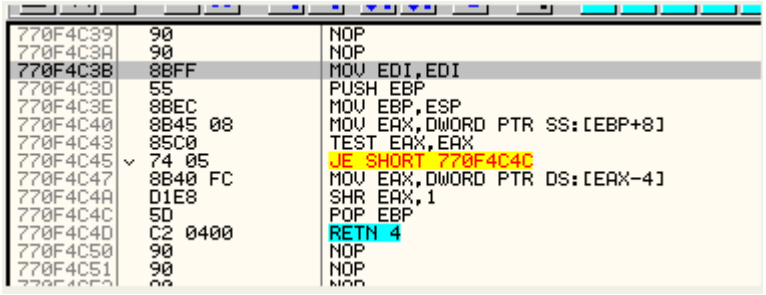


断在了这里,正确的 IAT 值将被写入。

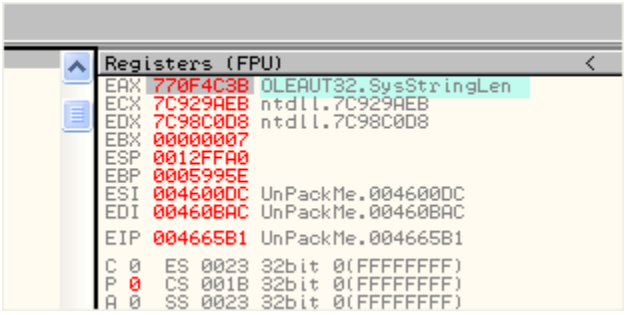


我们可以看到此时 EAX 寄存器中可能包含的是一个 API 函数的入口地址,虽然这里 OD 没有提示该 API 函数的名称,但是我们可以 Goto 过去看一看,我们 CTRL + G 输入 EAX。





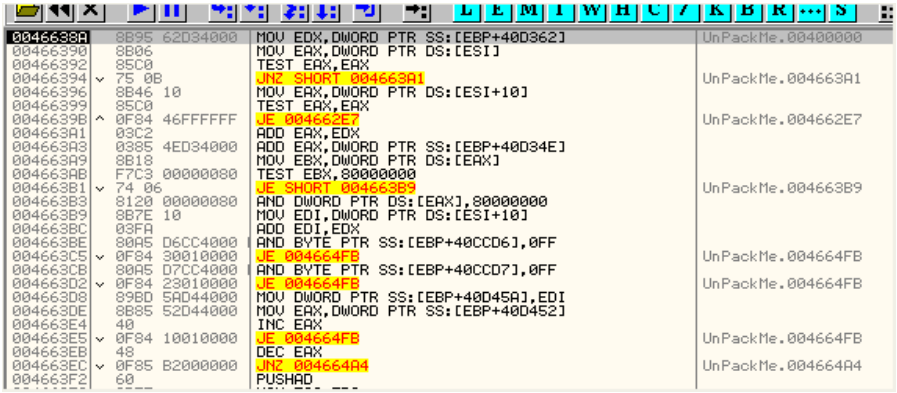
我们到了这里,我们可以看到寄存器窗口中 EAX 显示处了该 API 函数的名称,嘿嘿。

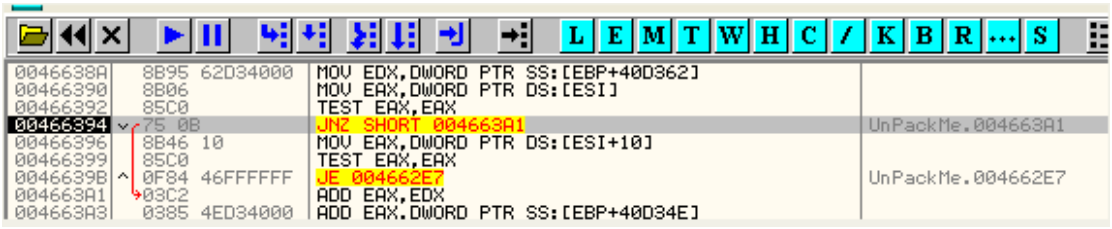


接下来的任务就是需要将 IAT 项被重定向的流程修改为正确 IAT 项的处理流程。

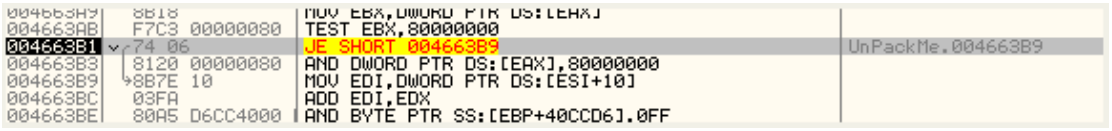
让所有 IAT 项走正常流程

我们依然从循环的起始位置开始跟。

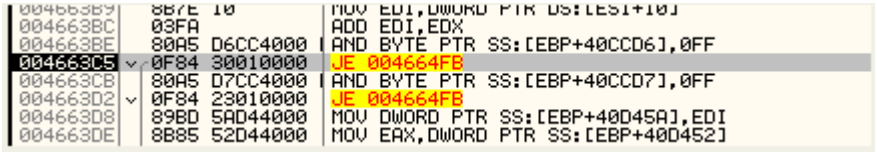




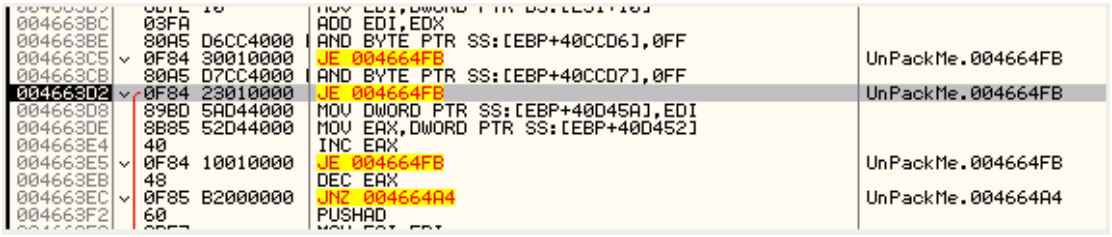
这里跟 IAT 项重定向的流程是一致的。



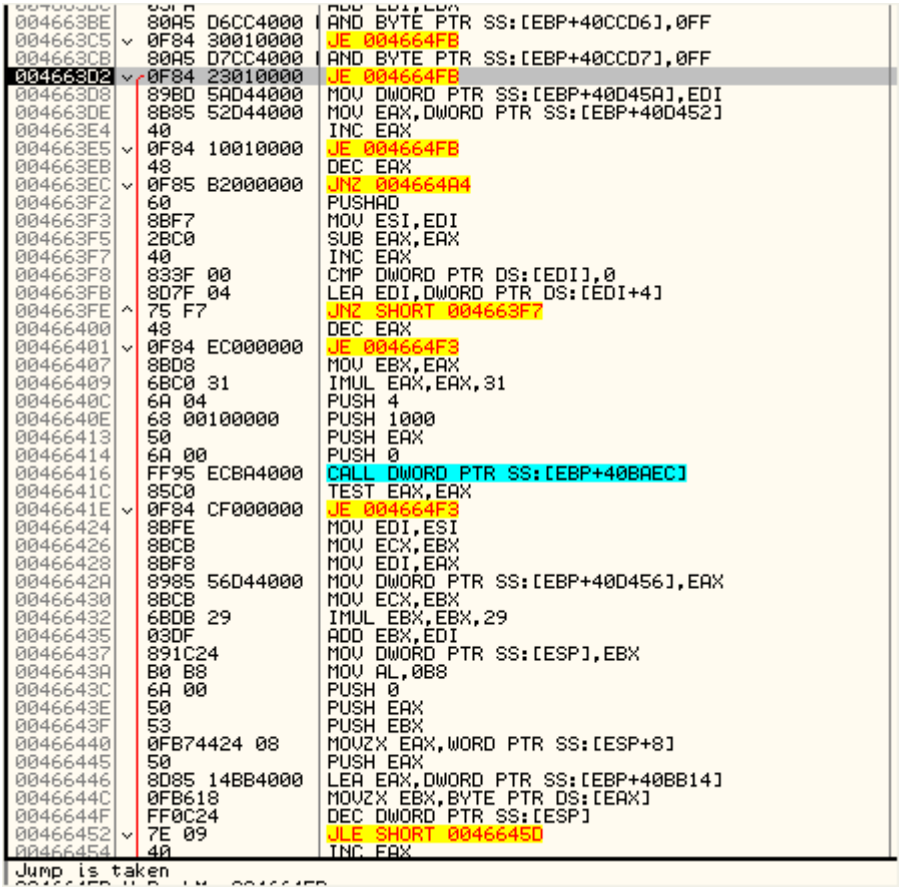
这里只是一个小跳转,我们继续跟。



这个跳转不成立。

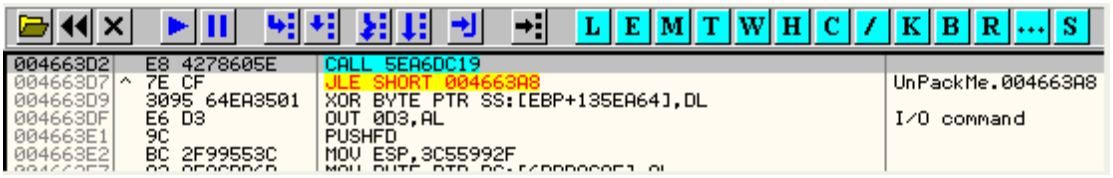


这里跟之前 IAT 项被重定位的处理有明显区别,跳过的中间的大片代码。

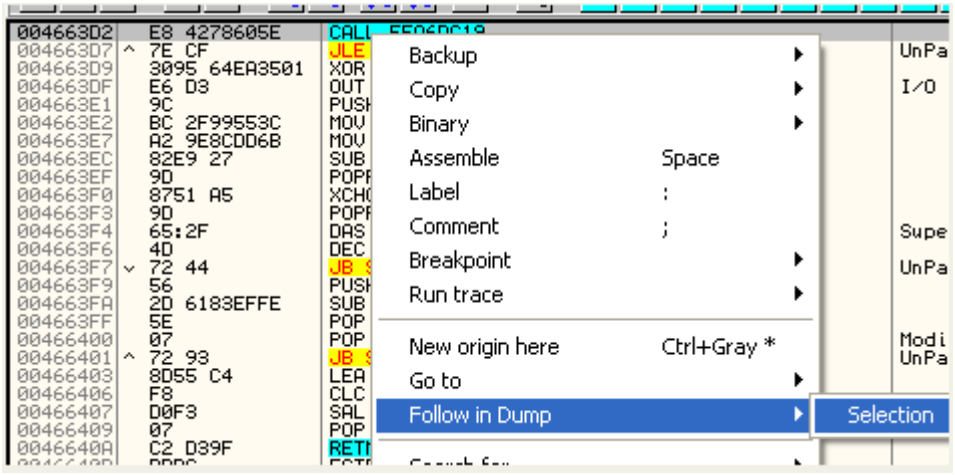


我们可以看到这里是一个长跳转,跳转与否取决于是 IAT 项是否被重定向,也就是说这里是决定是 IAT 项是否被重定向的关键跳,我们可以将这个条件跳转替换为 JMP。

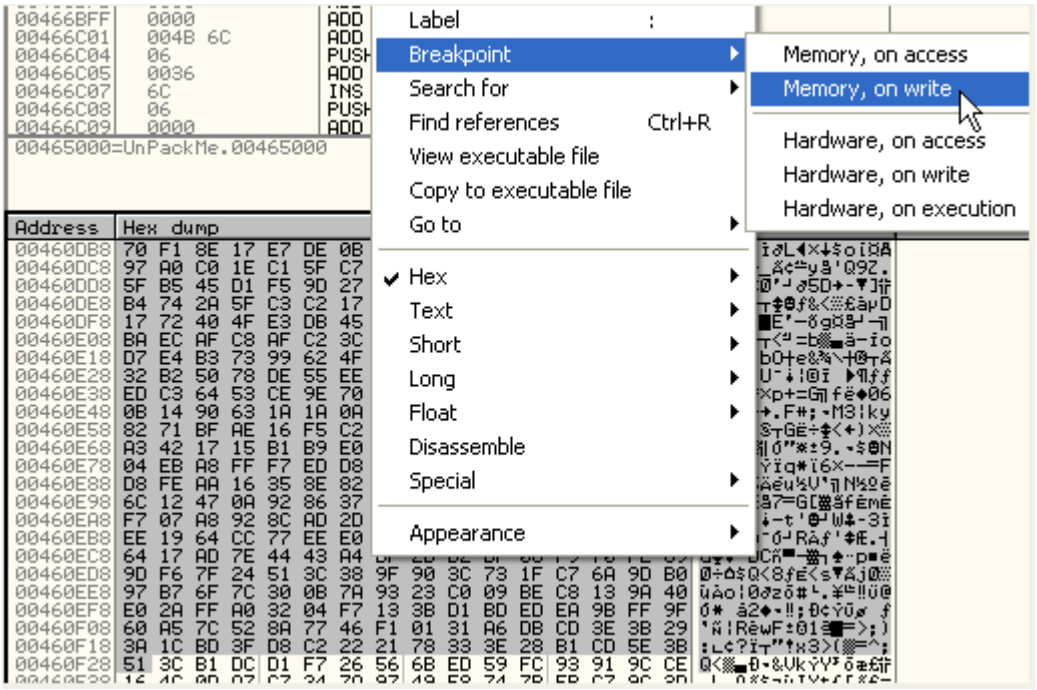
但是这个跳转在程序一开始的时候并不存在,我们重启程序。



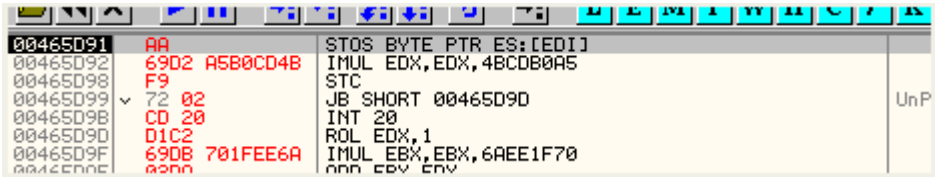
我们可以看到在程序刚开始的时候,这里都是一些垃圾指令,壳会随后的某个时间点写入实际的功能代码。



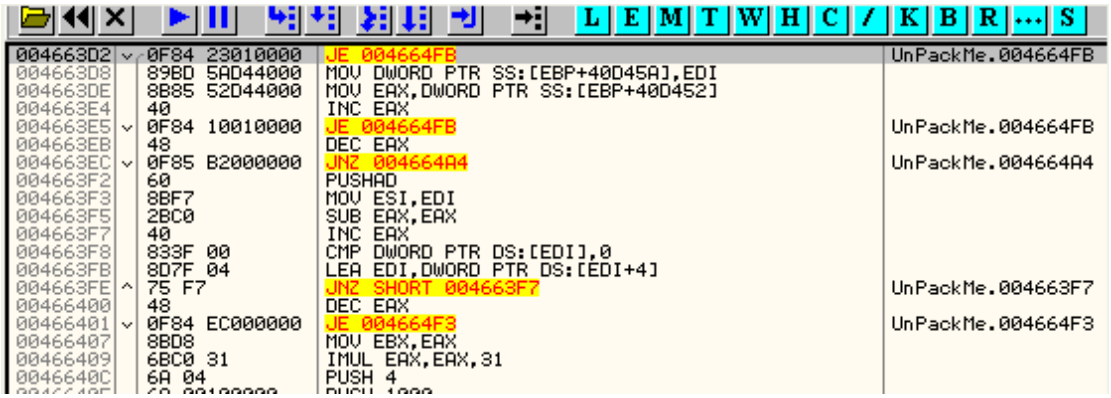
我们转到数据窗口,由于硬件断点会被检测,INT3 断点这里不能用,所以我们可以使用内存写入断点,我们对 460818~460f28 这片 IAT 区域设置内存写入断点。



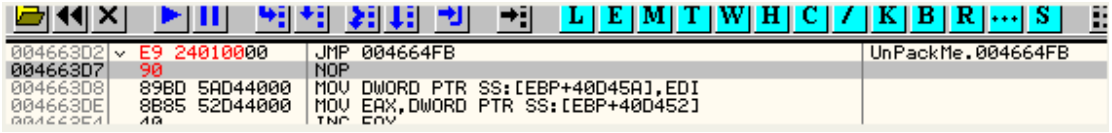
这里我们对所有的 IAT 项都设置上了内存写入断点,当第一个 IAT 项被写入的时候就会断下来。



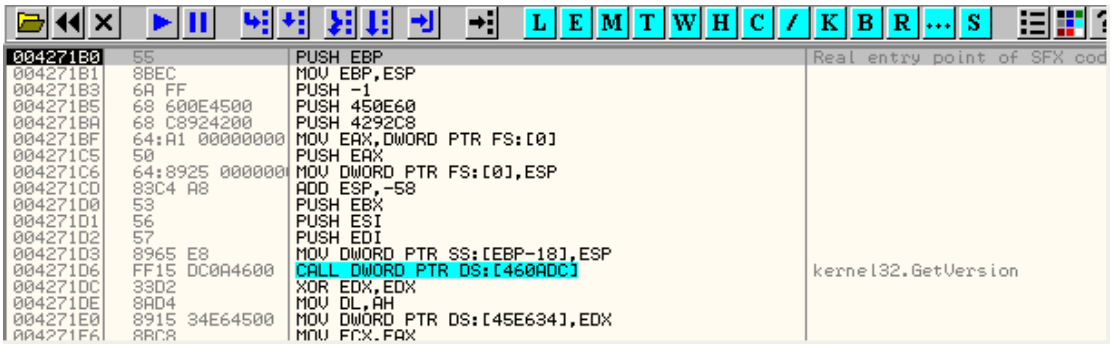
断在了这里,这里的 STOS 指令会执行多次,直到整个 IAT 都被填充完毕为止,此时关键跳已经存在了。



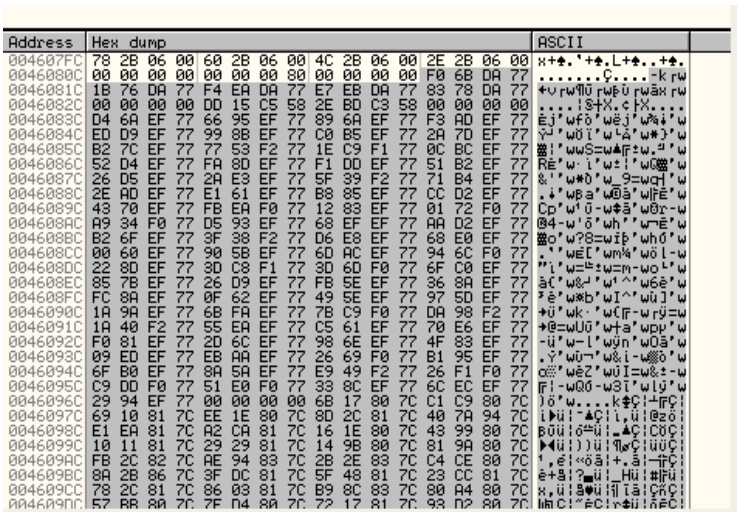
这里我们可以看到关键跳已经生成了,我们将该关键跳转 JE 替换成 JMP,接着清除之前设置的内存断点,接着跟踪到 OEP 处。



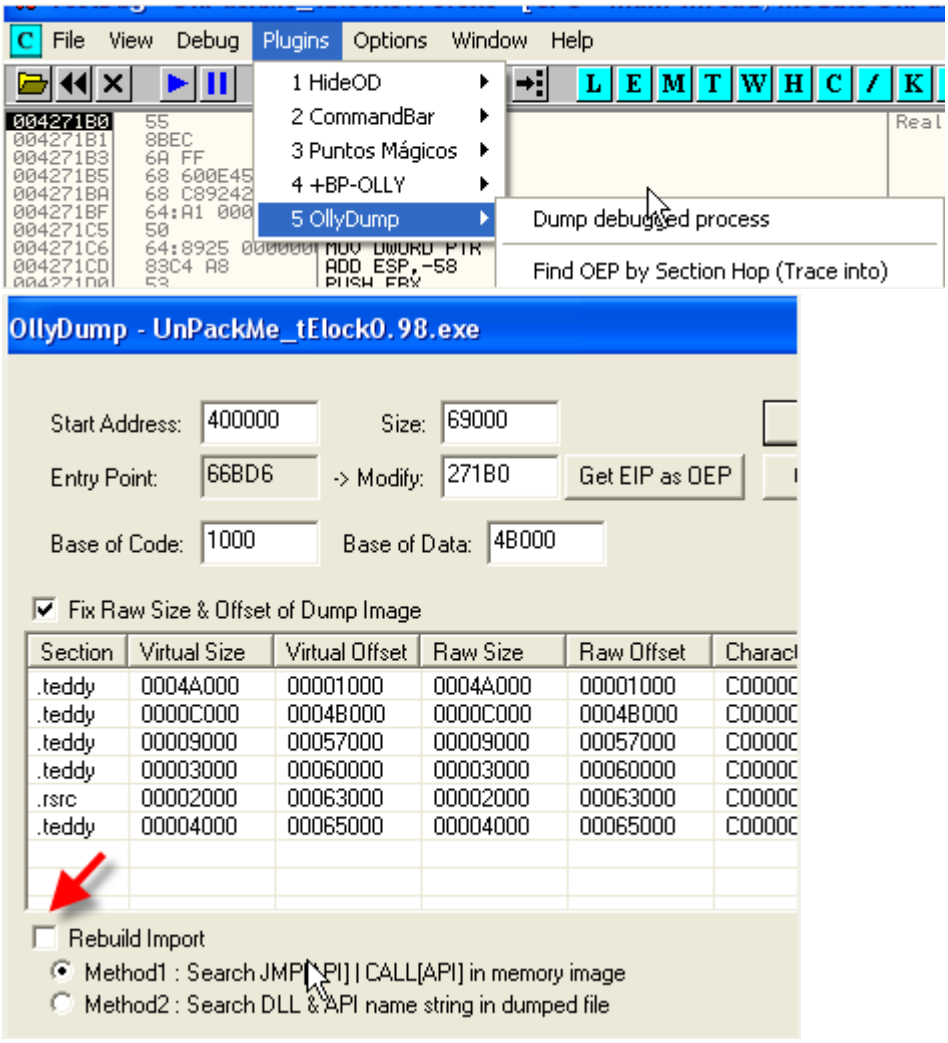
这里有两种可能,一种就是壳有自校验,运行起来会失败。另一种就是运行很正常,这里我们顺利的跟踪到了 OEP 处。



现在到了 OEP 处,我们再来看看 IAT:

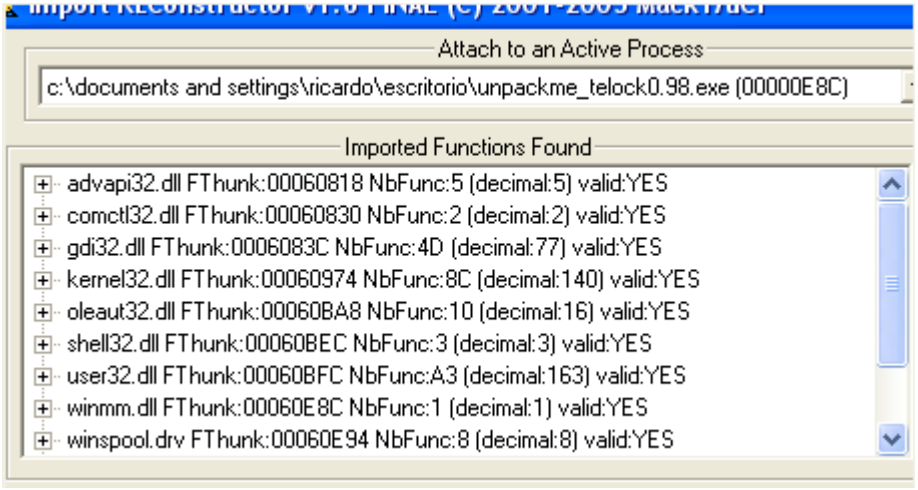


我们可以看到所有的 IAT 项都是正常的,我们成功修复了 IAT,现在我们可以进行 dump 了。

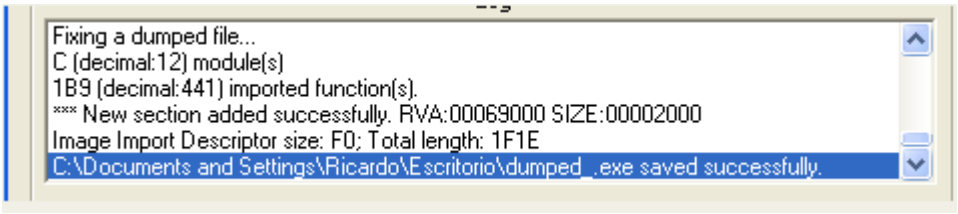


这里我们不勾选 ReBuild Import,dump 出来。

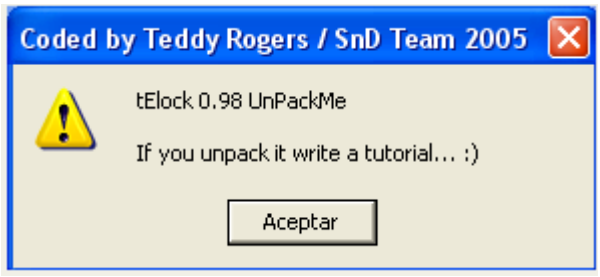
接着打开 IMP REC,填入 OEP,RVA,SIZE 等数据,单击 Get Imports。



我们可以看到所有的项都是有效的,说明刚刚修改的关键跳起作用了,接着单击 Fix dump 修复刚刚的 dump 文件。



修复过的 dump 文件被重命名为了 dumped_.exe,我们运行一下看看效果。



嘿嘿,完美运行。

不同壳关键跳的地方也不一样,但是我们细心对比正常 IAT 项与重定向过的 IAT 项的处理流程差异,总能找到一点蛛丝马迹,好了,本章就讨论到这里,下一章我们继续讨论别的壳。