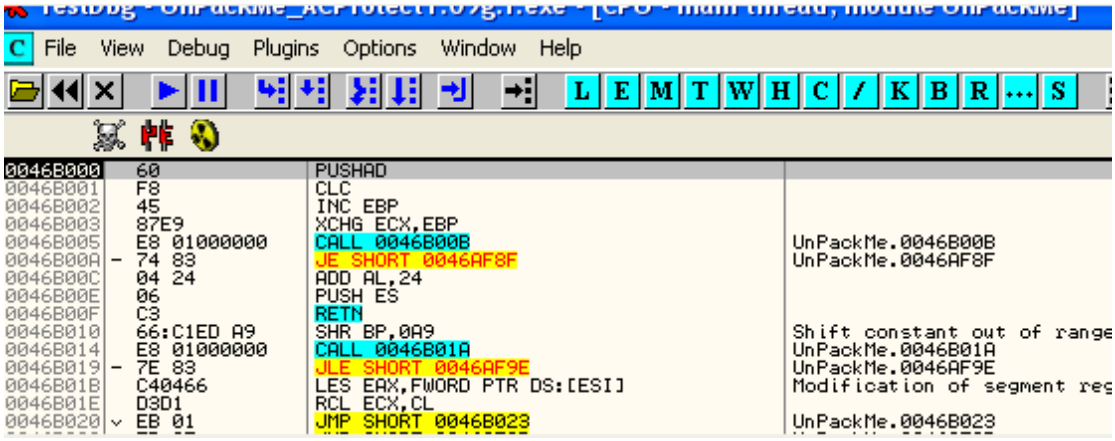
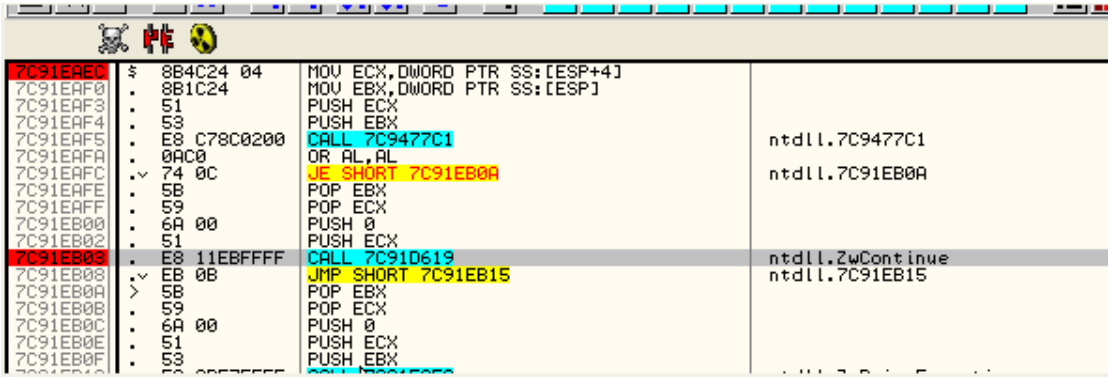
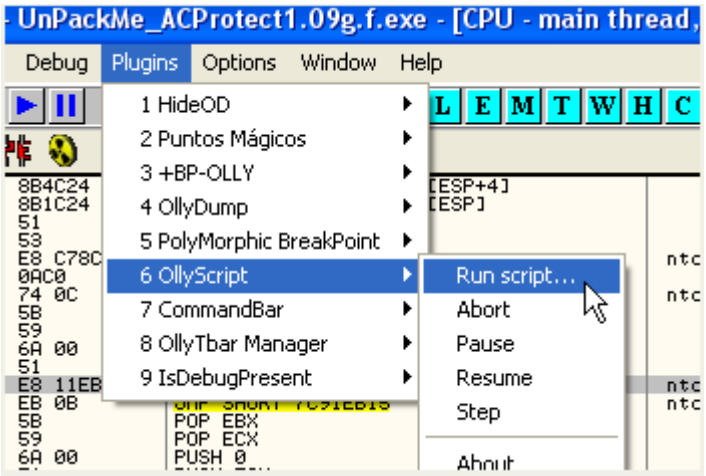
第四十三章-ACProtect V1.09(编写脚本修复 IAT)

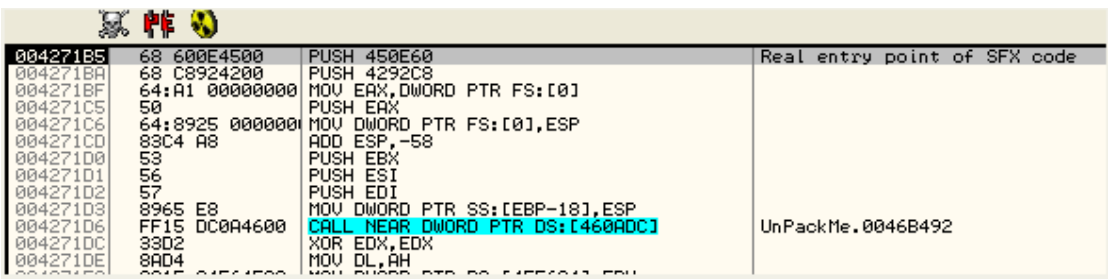
上一章节,我们介绍了如何定位 stolen bytes,本章我们的任务是修复 IAT,再次用 OD 加载 UnPackMe_ACProtect1.09,我们可以通过并执行 HBP.TXT 脚本到达假的 OEP 处。



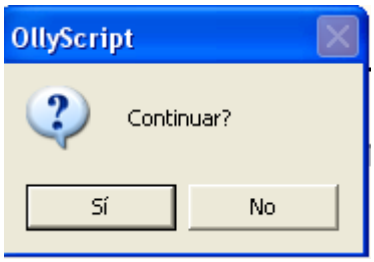
现在我们在壳的入口处,现在我们来利用上一章节编写的脚本定位到 OEP 处,首先,需要需要给 KiUserExceptionDispatcher 入口处以及其下方的 ZwContinue 调用处分别设置断点。



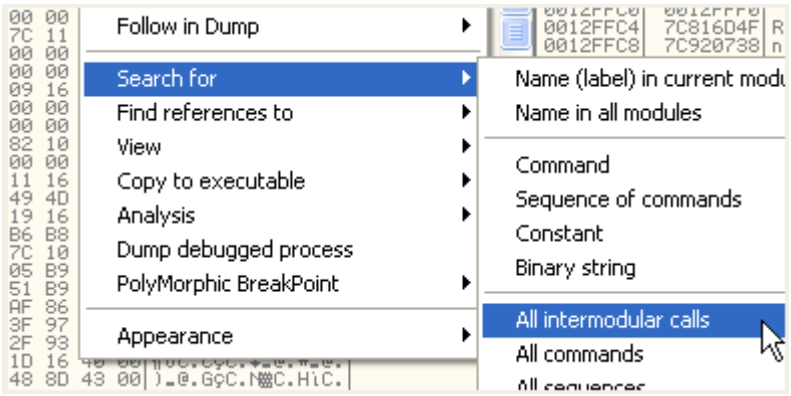




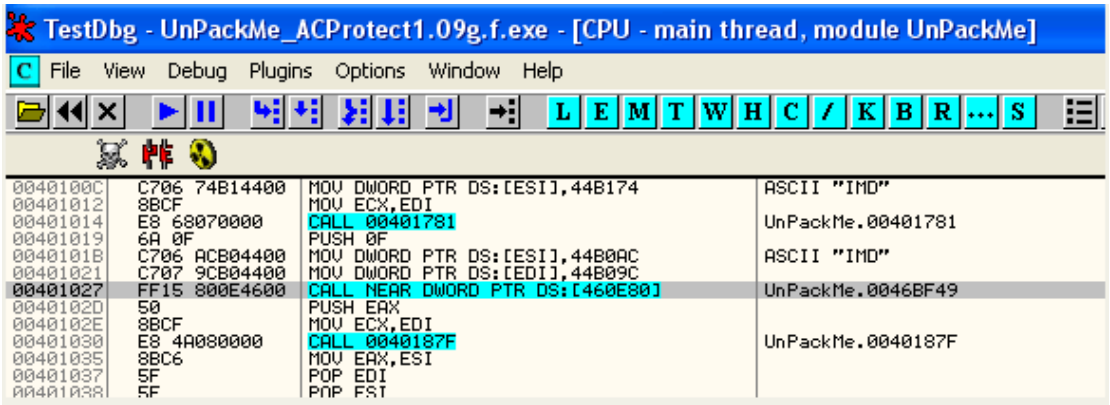
现在我们到了假的 OEP 处,大家可以修改一下这个脚本让其自动给 KiUserExceptionDispatcher 入口以及 ZwContinue 调用处设置断点,并且让其支持输入需要设置硬件断点的地址,但这里我们暂时没有必须修改,这个脚本目前来说已经够用了。

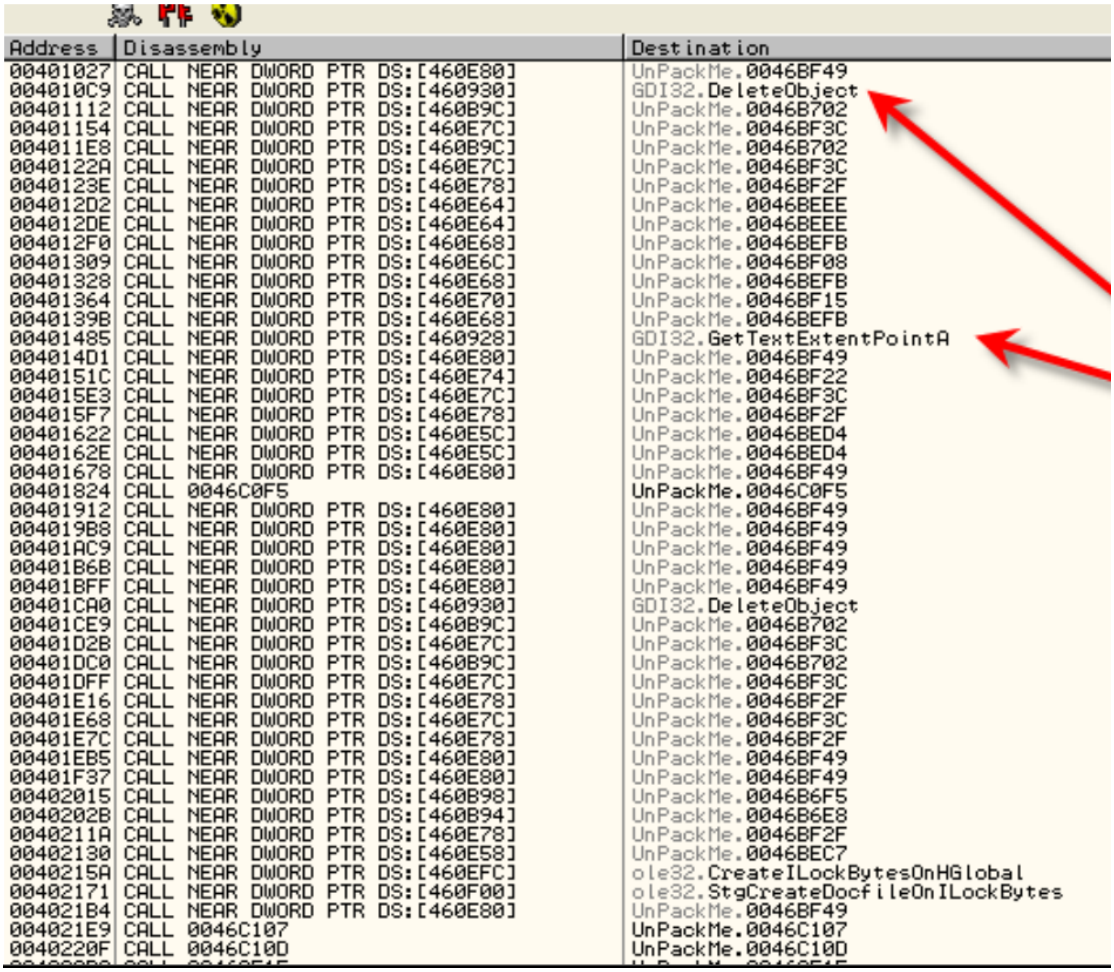


下面我们来随便定位一个 API 函数的调用处,单击鼠标右键选择 Search for - All intermodular calls 选项搜索。



我们可以看到很多 API 函数的调用,其中有些 IAT 项是正常的,显示出了函数名称,但是大部分的 IAT 项都是经过重定位的,并没有显示函数名称,我们随便选中一个重定向的 IAT 项,双击鼠标左键。

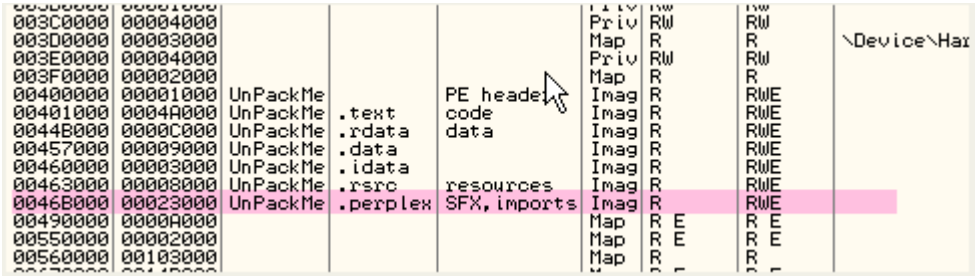




Address	Disassembly	Destination
00401027	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
004010C9	CALL NEAR DWORD PTR DS:[460930]	GDI32.DeleteObject
00401112	CALL NEAR DWORD PTR DS:[460B9C]	UnPackMe.0046B702
00401154	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
004011E8	CALL NEAR DWORD PTR DS:[460B9C]	UnPackMe.0046B702
0040122A	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
0040123E	CALL NEAR DWORD PTR DS:[460E78]	UnPackMe.0046BF2F
004012D2	CALL NEAR DWORD PTR DS:[460E64]	UnPackMe.0046BEEE
004012DE	CALL NEAR DWORD PTR DS:[460E64]	UnPackMe.0046BEEE
004012F0	CALL NEAR DWORD PTR DS:[460E68]	UnPackMe.0046BEFB
00401309	CALL NEAR DWORD PTR DS:[460E6C]	UnPackMe.0046BF08
00401328	CALL NEAR DWORD PTR DS:[460E68]	UnPackMe.0046BEFB
00401364	CALL NEAR DWORD PTR DS:[460E70]	UnPackMe.0046BF15
0040139B	CALL NEAR DWORD PTR DS:[460E68]	UnPackMe.0046BEFB
00401485	CALL NEAR DWORD PTR DS:[460928]	GDI32.GetTextExtentPointA
004014D1	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
0040151C	CALL NEAR DWORD PTR DS:[460E74]	UnPackMe.0046BF22
004015E3	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
004015F7	CALL NEAR DWORD PTR DS:[460E78]	UnPackMe.0046BF2F
00401622	CALL NEAR DWORD PTR DS:[460E5C]	UnPackMe.0046BED4
0040162E	CALL NEAR DWORD PTR DS:[460E5C]	UnPackMe.0046BED4
00401678	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401824	CALL 0046C0F5	UnPackMe.0046C0F5
00401912	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
004019B8	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401AC9	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401B6B	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401BFF	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401CA0	CALL NEAR DWORD PTR DS:[460930]	GDI32.DeleteObject
00401CE9	CALL NEAR DWORD PTR DS:[460B9C]	UnPackMe.0046B702
00401D2B	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
00401DC0	CALL NEAR DWORD PTR DS:[460B9C]	UnPackMe.0046B702
00401DFF	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
00401E16	CALL NEAR DWORD PTR DS:[460E78]	UnPackMe.0046BF2F
00401E68	CALL NEAR DWORD PTR DS:[460E7C]	UnPackMe.0046BF3C
00401E7C	CALL NEAR DWORD PTR DS:[460E78]	UnPackMe.0046BF2F
00401EB5	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00401F37	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
00402015	CALL NEAR DWORD PTR DS:[460B98]	UnPackMe.0046B6F5
0040202B	CALL NEAR DWORD PTR DS:[460B94]	UnPackMe.0046B6E8
0040211A	CALL NEAR DWORD PTR DS:[460E78]	UnPackMe.0046BF2F
00402130	CALL NEAR DWORD PTR DS:[460E58]	UnPackMe.0046BEC7
0040215A	CALL NEAR DWORD PTR DS:[460EFC]	ole32.CreateILockBytesOnHGlobal
00402171	CALL NEAR DWORD PTR DS:[460F00]	ole32.StgCreateDocfileOnILockBytes
004021B4	CALL NEAR DWORD PTR DS:[460E80]	UnPackMe.0046BF49
004021E9	CALL 0046C107	UnPackMe.0046C107
0040220F	CALL 0046C10D	UnPackMe.0046C10D

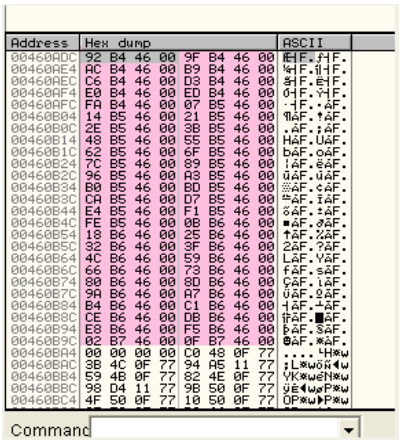
这里我们可以看到 CALL 的是 460E80 内存单元中对应的 IAT 项,我们在数据窗口中定位到该项。

这里我们可以看到很多经过重定向的项,这些项直接就位于壳的区段中,我们来看看区段列表窗口。



003C0000	00004000				Priv	RW	RW	
003D0000	00003000				Map	R	R	\Device\Har
003E0000	00004000				Priv	RW	RW	
003F0000	00002000				Map	R	R	
00400000	00001000	UnPackMe		PE header	Imag	R	RWE	
00401000	0004A000	UnPackMe	.text	code	Imag	R	RWE	
0044B000	0000C000	UnPackMe	.rdata	data	Imag	R	RWE	
00457000	00009000	UnPackMe	.data		Imag	R	RWE	
00460000	00003000	UnPackMe	.idata		Imag	R	RWE	
00463000	00008000	UnPackMe	.rsrc	resources	Imag	R	RWE	
0046B000	00023000	UnPackMe	.perplex	SFX, imports	Imag	R	RWE	
00490000	0000A000				Map	R E	R E	
00550000	00002000				Map	R E	R E	
00560000	00103000				Map	R	R	

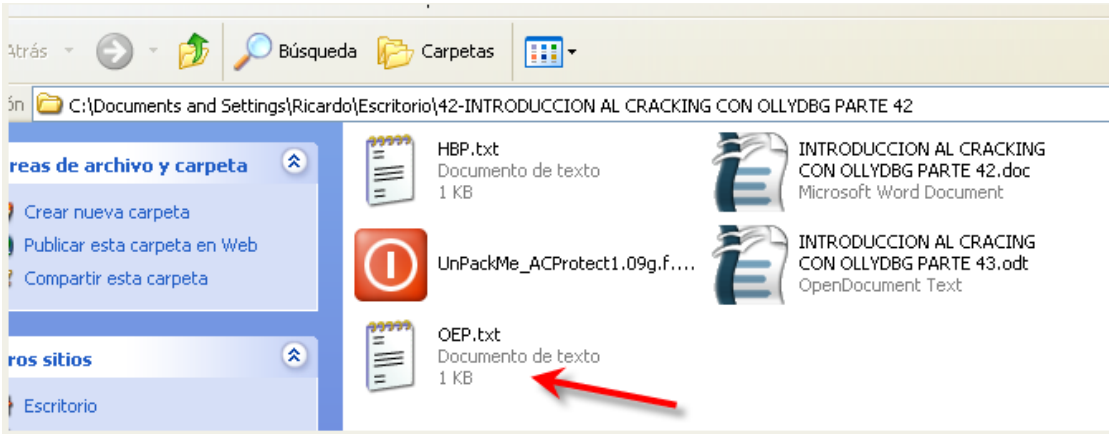
大家应该还记得该壳的入口点也是位于这个区段的,壳的入口点为 46B000,也就是说该壳并没有重新创建一个区段用来处理重定向的 API 函数。

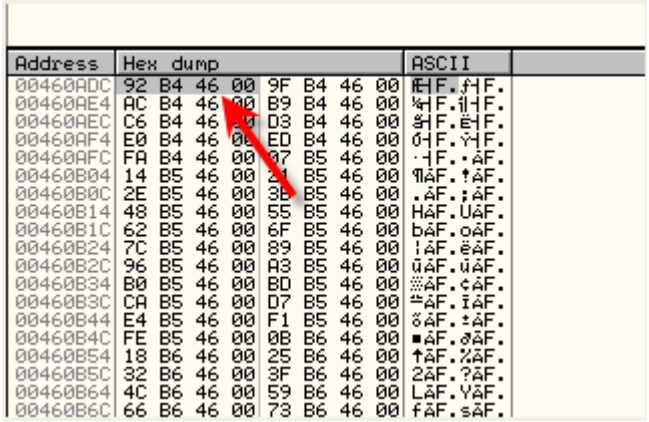


接下来任务就需要定位修复 IAT 项的关键跳转(magical jump)了。

下面我们就来通过 460ADC 这一项来定位关键跳。

首先我们将定位 OEP 的脚本备份一下。





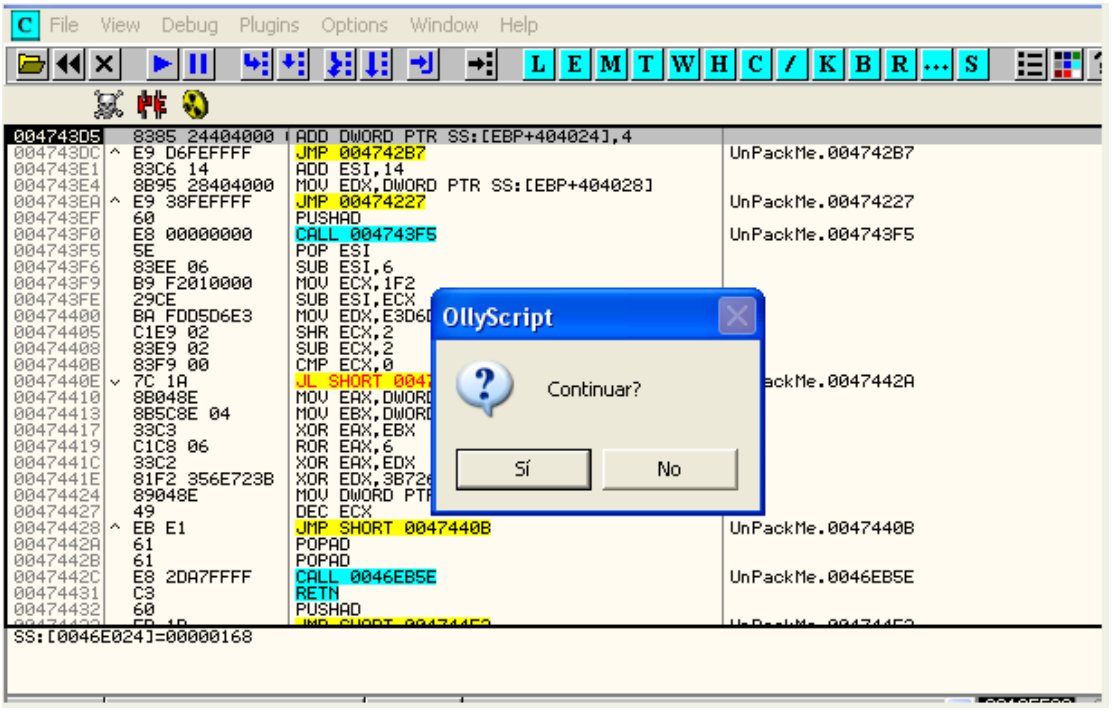
我们将备份过的脚本重命名为 OEP.txt,接下来我们通过修改 HBP.txt 脚本来定位修复 IAT 的关键跳转。

这里我们将脚本修改为对 460ADC 这个重定向的 IAT 项设置硬件写入断点,也就是说断点类型修改为 W。

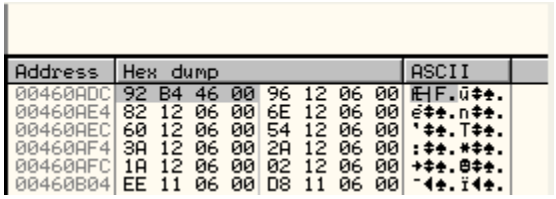
现在我们清除之前设置的硬件断点,重新启动 OD。

```
0000 var RetAddr
0001 Beginning:
0002
0003 bphws 460adc,"w"
0004
0005 Work:
0006
0007 eob ToProcess
0008 run
0009
0010 ToProcess:
0011 log eip
0012 cmp eip,7c92e47c
0013 je ToClear
0014 cmp eip,7c92e493
0015 je ToRecover
0016 cmp eip,RetAddr
0017 je ToReset
0018 jmp Final
0019
0020 ToClear:
0021 bphwc 460adc
0022 jmp Work
0023
0024 ToRecover:
0025 mov RetAddr,esp
0026 mov RetAddr,[RetAddr]
0027 add RetAddr,0b8
0028 mov RetAddr,[RetAddr]
0029 log RetAddr
0030 bp RetAddr
0031 jmp Beginning
0032
0033 ToReset:
0034 bc RetAddr
0035 jmp Beginning
0036
0037 Final:
0038 msgyn "Continue?"
0039 cmp $RESULT,1
0040 je Beginning
0041 ret
0042
```

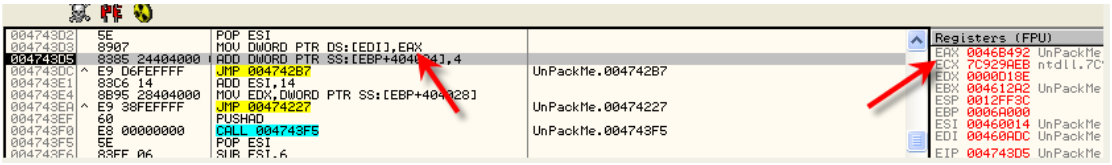
执行上面的脚本,不一会儿弹出了是否继续执行脚本的消息框。



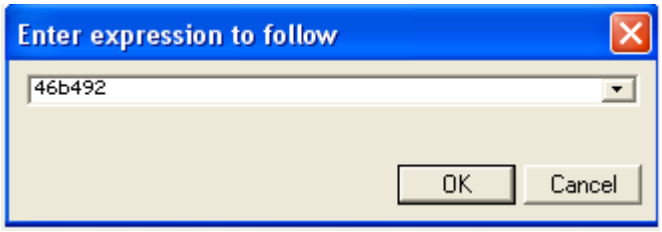
我们可以看到这里对460ADC写入的时候触发了硬件写入断点,我们知道硬件断点是断在下一条指令处,我们来看看前一条指令是什么。

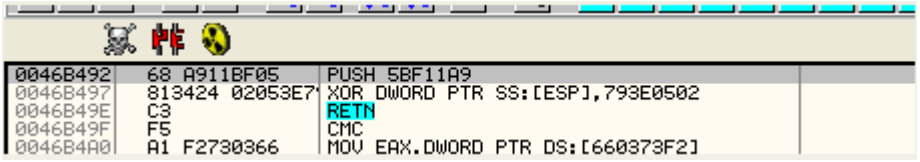


这里我们可以看到是这条指令将对 460ADC 地址进行写入,我们来看看 EAX 的值是多少。



这里 EAX 的值为 46B492。





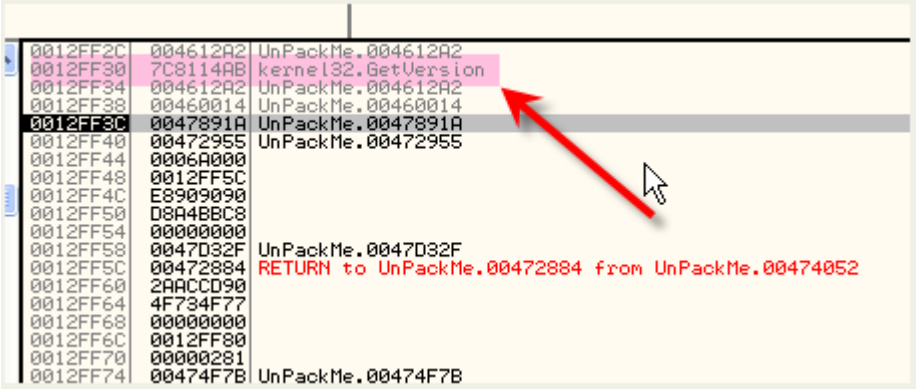
这里我们在反汇编窗口中定位到 46B492 这个地址,看看该壳做什么处理。

这里我们看到这个重定向的处理非常简单,首先将常量 5BF11A9 压入堆栈,接着将该值与 793E0502 进行异或,接着就 RET 返回。

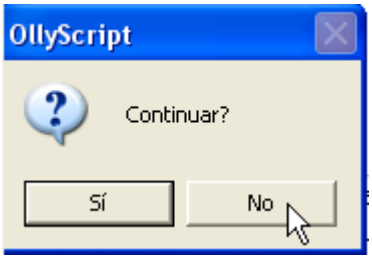
异或得到的结果就是 API 函数的入口地址,我们一起来计算一下。

5BF11A9 XOR 793E0502 = 7C8114AB

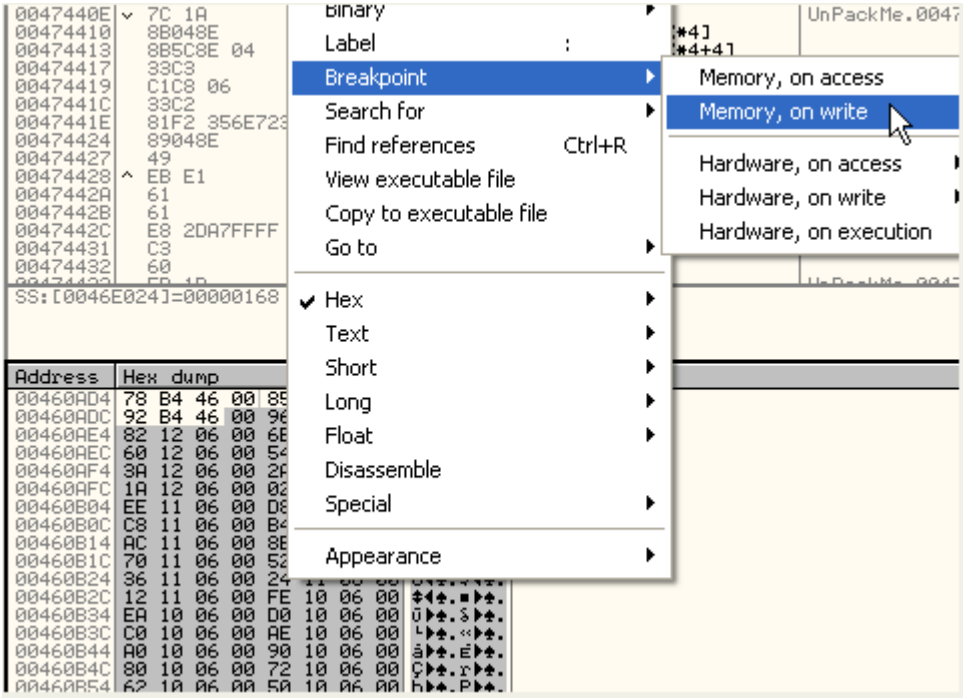
计算出来的这个地址就是 API 函数的入口地址,我们返回到 OD 中看看。



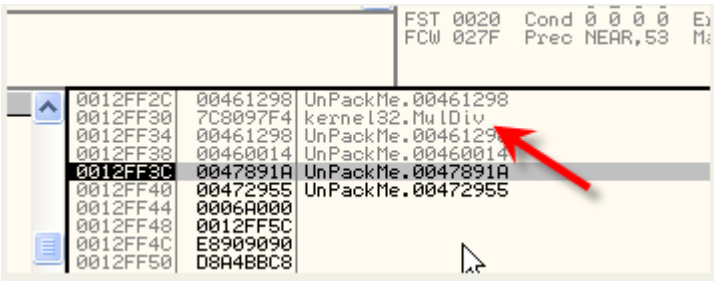
我们将堆栈窗口往上拉一点就能看到正确的 API 函数地址了,下面我们就来看看是不是所有的重定向的 IAT 项都是这样处理的,先停止脚本。



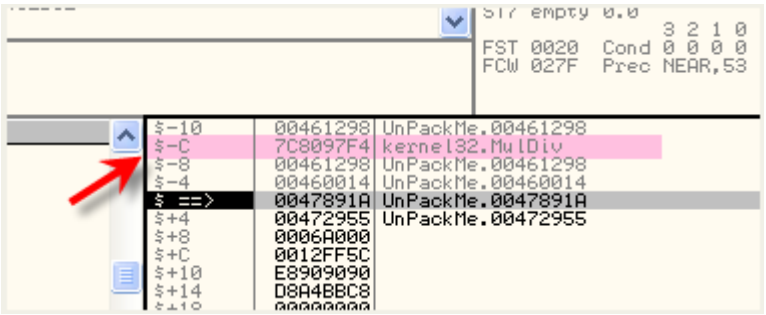
我们对部分重定向的 IAT 项设置内存写入断点,看看会发生什么。



我们运行起来,当下一个重定向的 IAT 项被写入的时候断了下来。

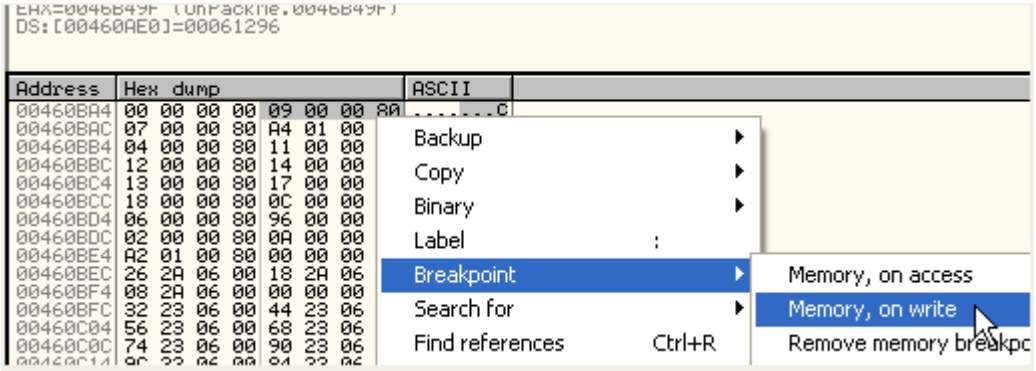


我们定位到重定向的部分,执行异或操作以后可以看到得到的 API 函数是 MulDiv,这么看来修复是有可能的,我们可以看到正常的 API 地址被保存在了堆栈中,确切点来说是[ESP - 0C]中。

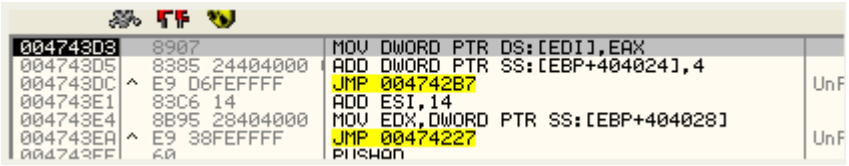


这里我们可以看到往下都是需要重定向的,到了 460BA8,就是正常的 IAT 项了。

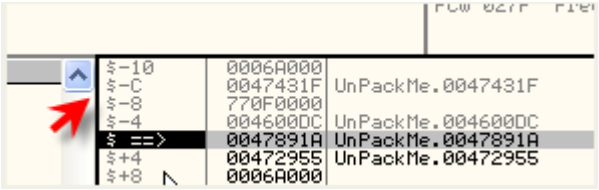
所以我们对 460BA8 这一项设置内存写入断点。



运行起来,到了这里。



我们可以看到对于正常的 IAT 项,[EBP - 0C]处并不会保存正确的 API 函数入口地址,所以说有点遗憾,不然我们可以使用一个简单的脚本轻松的修复 IAT。



我们该怎么做呢?我们需要对写入 IAT 项的指令设置硬件执行断点,当脚本执行到这里的时候,我们判断 EAX 的值是正常的还是重定向的,如果是重定向的,那么我们就将[ESP + 0C]的值填充到 IAT 对应的条目中,如果是正常的 IAT 项的话,我们就不予处理,我们来看看该脚本如何编写。

```
0000 var aux
0001 var aux2
0002
0003 Beginning:
0004 bphws 4743d5,"x"
0005 Work:
0006 eob ToProcess
0007 run
0008
0009 ToProcess:
0010 log eip
0011 cmp eip,7c92e47c
0012 je ToClear
0013 cmp eip, 7c92e493
0014 je ToRecover
0015 cmp eip, aux
0016 je ToRecover2
0017 cmp eip,4743d5
0018 je ToRepair
0019 jmp final
0020
0021 ToClear:
0022 bphwc 4743d5
0023 jmp Work
0024 ToRecover:
0025 mov aux,esp
0026 mov aux,[aux]
0027 add aux,0b8
0028 mov aux,[aux]
0029 log aux
0030 jmp Beginning
0031
0032 ToReset2:
0033 bc aux
0034 jmp Beginning
0035
0036 ToRepair:
0037 CMP eax,500000
0038 ja  Beginning
0039 mov aux2,esp
0040 sub aux2,0c
0041 mov aux2,[aux2]
0042 log aux2
0043 mov [edi],aux2
0044 jmp Beginning
0045 End:
0046 MSGYN "To continue?"
0047 cmp $RESULT,1
0048 je  Beginning
0049 ret
```

以上就是完整的脚本,这里我是在下一行的 4743d5 处设置的硬件执行断点,因为硬件执行断点当将要执行该条指令的时候就断下来,不同于硬件写入或者访问断点。

这个脚本是基于 HBP.txt 改写的,首先对 IAT 项重定向的下一条指令 4743d5 处设置硬件执行断点。

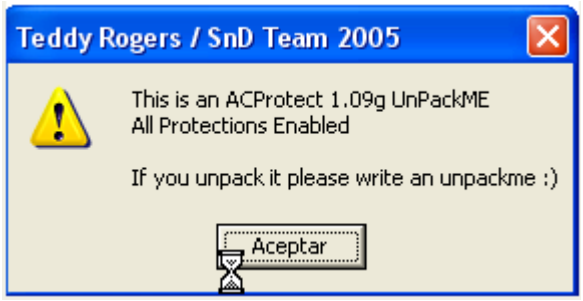
```
0017 cmp eip,4743d5
0018 je ToRepair
```

这里是 ToProcess 这个分支,当脚本检测到中断异常,将 EIP 与 4743d5 进行比较,如果相等就跳转到 ToRepair 分支中。

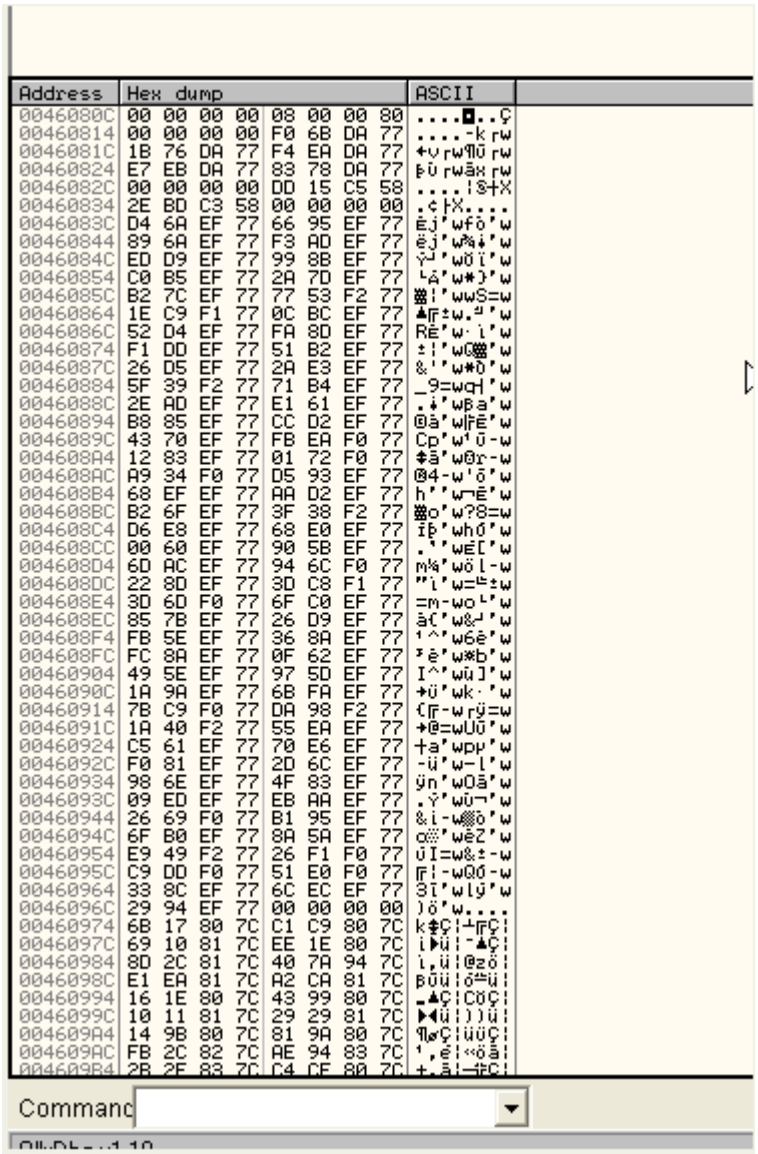
```
0036 ToRepair:
0037 cmp eax,500000
0038 ja  Beginning
0039 mov aux2,esp
0040 sub aux2,0c
0041 mov aux2,[aux2]
0042 log aux2
0043 mov [edi],aux2
0044 jmp Beginning
```

ToRepair 分支,我们首先判断 EAX 是否为正常,如果被重定向到壳所在区段的话,那么地址是小于 500000 的,反之,如果 EAX 的值大于 500000,就表示该 IAT 项是正常的,我们就无须进行修复,回到开始处继续捕获下一次中断。如果 EAX 是重定向的值的话,我们就使用变量 aux2 来保存正确的 API 函数地址,正确的 API 函数地址保存在[ESP - 0C]中。我们将其写入到 EDI 指向的 IAT 项中即可。下面我们就来执行一下该脚本,试试效果。

重启程序,将之前设置的硬件断点全部清空,接着对 KiUserExceptionDispatcher 的入口处以及 ZwContinue 的调用处分别设置断点,然后运行脚本。

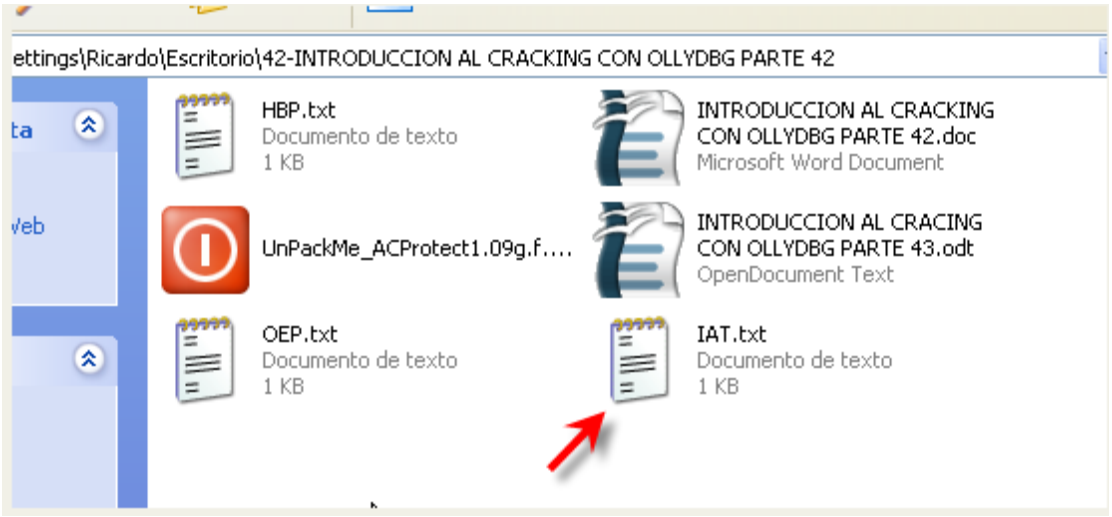


程序运行起来了,我们来看下 IAT。

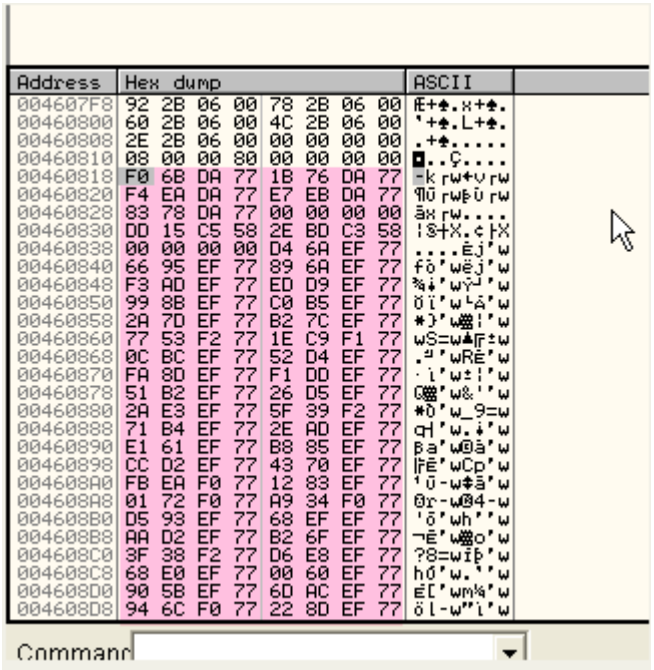


嘿嘿,我们可以看到 IAT 项都被修复了。

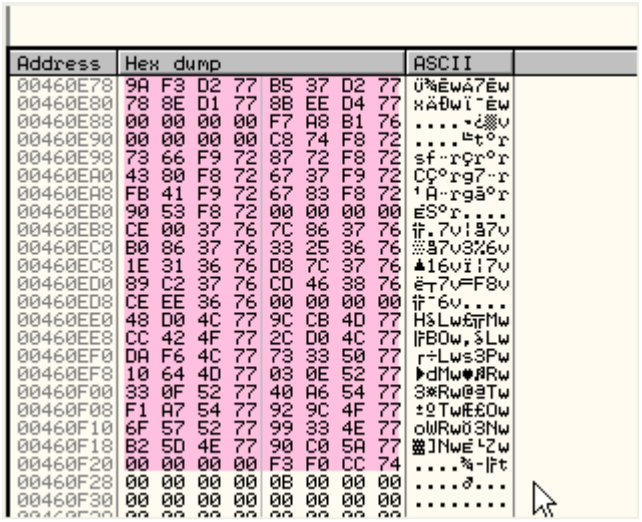
我们将脚本重命名为 IAT.txt。



下面我们需要定位 IAT 的起始地址以及 IAT 的大小。



这里我们可以看到 IAT 的起始地址为 460818,结束于 460F28。



Address	Hex dump	ASCII
00460E78	9A F3 D2 77 B5 37 D2 77	
00460E80	78 8E D1 77 8B EE D4 77	
00460E88	00 00 00 00 F7 A8 B1 76	
00460E90	00 00 00 00 C8 74 F8 72	
00460E98	73 66 F9 72 87 72 F8 72	
00460EA0	43 80 F8 72 67 37 F9 72	
00460EA8	FB 41 F9 72 67 83 F8 72	
00460EB0	90 53 F8 72 00 00 00 00	
00460EB8	CE 00 37 76 7C 86 37 76	
00460EC0	B0 86 37 76 33 25 36 76	
00460EC8	1E 31 36 76 D8 7C 37 76	
00460ED0	89 C2 37 76 CD 46 38 76	
00460ED8	CE EE 36 76 00 00 00 00	
00460EE0	48 D0 4C 77 9C CB 4D 77	
00460EE8	CC 42 4F 77 2C D0 4C 77	
00460EF0	DA F6 4C 77 73 33 50 77	
00460EF8	10 64 4D 77 03 0E 52 77	
00460F00	33 0F 52 77 40 A6 54 77	
00460F08	F1 A7 54 77 92 9C 4F 77	
00460F10	6F 57 52 77 99 33 4E 77	
00460F18	B2 5D 4E 77 90 C0 5A 77	
00460F20	00 00 00 00 F3 F0 CC 74	
00460F28	00 00 00 00 0B 00 00 00	
00460F30	00 00 00 00 00 00 00 00	

OEP(RVA) = 271b5

IAT 的起始地址(RVA) = 60818

IAT 的大小 = 710

这里我们可以用的假的 OEP,后面我们可以手工修改。

下一章节,我们来解决 AntiDump，敬请关注。