

## 第三十二章-OEP 寻踪

在上一章中我们提到了 OEP(Original EntryPoint)的概念,也就是应用程序原本要执行的第一行代码,OEP 99% 的情况位于第一个区段中(本章中有一个例子,OEP 就不在第一个区段,我是特意举的这个例子,嘿嘿)。

我们知道当到达 OEP 后,各个区段在内存中的分布跟原始程序很接近,这个时候我们就可以尝试将其转储到(dump)文件中,完成程序的重建工作(PS:脱壳)。

通常脱壳的基本步骤如下:

1:寻找 OEP

2:转储(PS:传说中的 dump)

3:修复 IAT(修复导入表)

4:检查目标程序是否存在 AntiDump 等阻止程序被转储的保护措施,并尝试修复这些问题。

以上是脱壳的经典步骤,可能具体到不同的壳的话会有细微的差别。本章我们主要介绍定位 OEP 的方法。

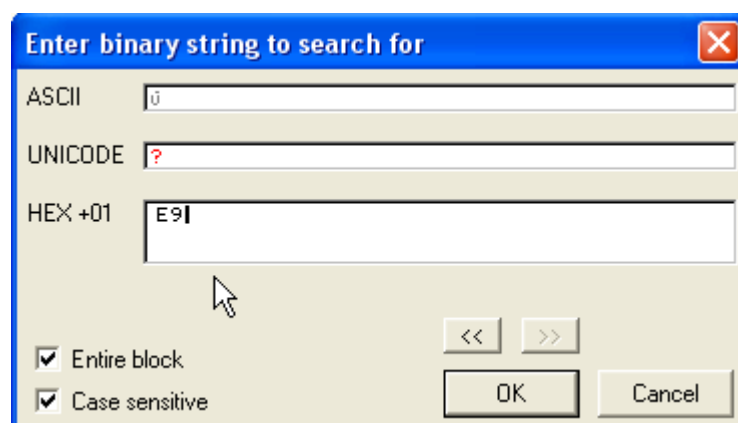
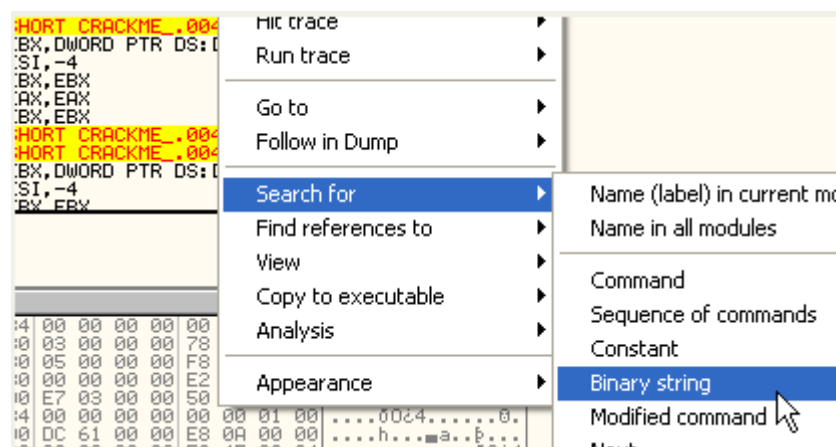
很多时候我们遇到的壳会想方设法的隐藏原程序的 OEP,要定位 OEP 的话就需要我们尝试各种各样的方法了。

首先我们来看看上一章 CRACKME UPX,然后再来看其他的壳。

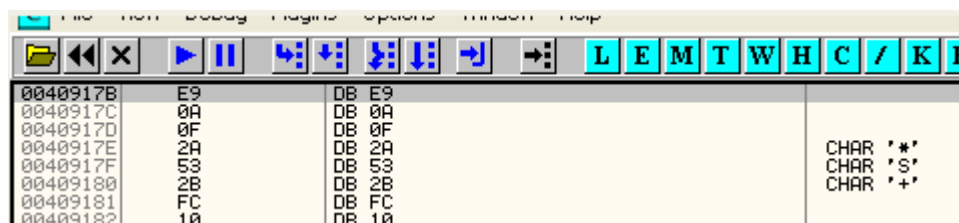
### 1)搜索 JMP 或者 CALL 指令的机器码(即一步直达法,只适用于少数壳,包括 UPX,ASPACK 壳)

对于一些简单的壳可以用这种方式来定位 OEP,但是对于像 AsProtect 这类强壳(PS:AsProtect 在 04 年算是强壳了,嘿嘿)就不适用了,我们可以直接搜索长跳转 JMP(0E9)或者 CALL(0E8)这类长转移的机器码,一般情况下(理想情况)壳在解密完原程序各个区段以后,需要一个长 JMP 或者 CALL 跳转到原程序代码段中的 OEP 处开始执行原程序代码。

下面我们将上一章中加了 UPX 壳的那个 CrueHead 的 CrackMe 加载到 OD 中:



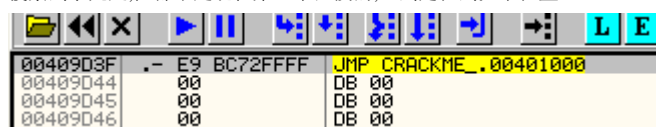
这里我们按 CTRL+B 组合键搜索一下 JMP 的机器码 E9,看看有没有这样一个 JMP 跳转到原程序的代码段。



多按几次 CTRL+L。

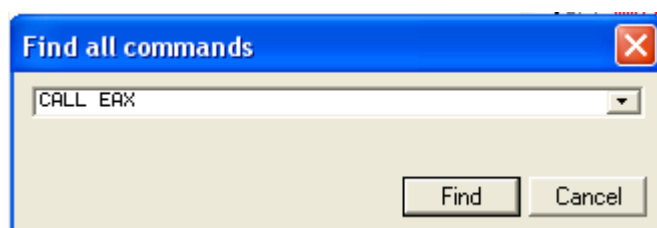
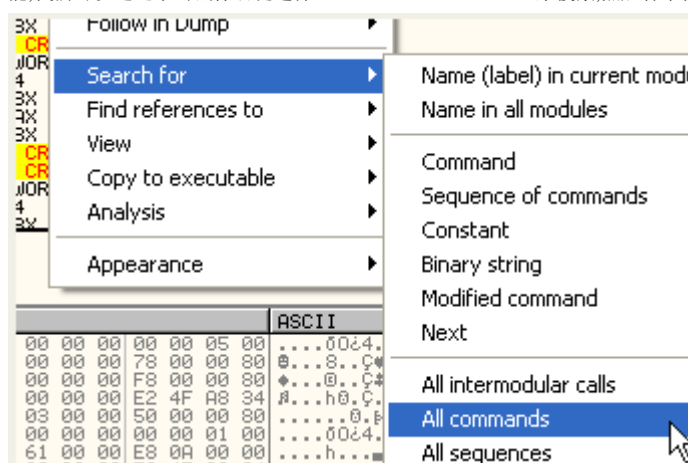


搜索到了几处,但都不是跳往第一个区段的,直到定位到如下位置:



这个 JMP 是跳转到第一个区段的,我们在这条指令处设置一个断点,断在这里时,我们按 F7 键就可以单步跳转到 OEP 处。

除了搜索 JMP 指令的机器码以外,大家还可以尝试搜索 CALL EAX, CALL EBX, JMP EAX 等指令的机器码,因为很多壳是将 OEP 的值存放在寄存器中,然后通过 CALL 某寄存器或者 JMP 某寄存器来跳往 OEP 的。OllyDbg 提供了搜索 ALL COMMANDS 的功能,我们可以通过单击鼠标右键选择-Search for-All Commands 来搜索,然后各个指令处依次设置断点,下面我们来看个例子。

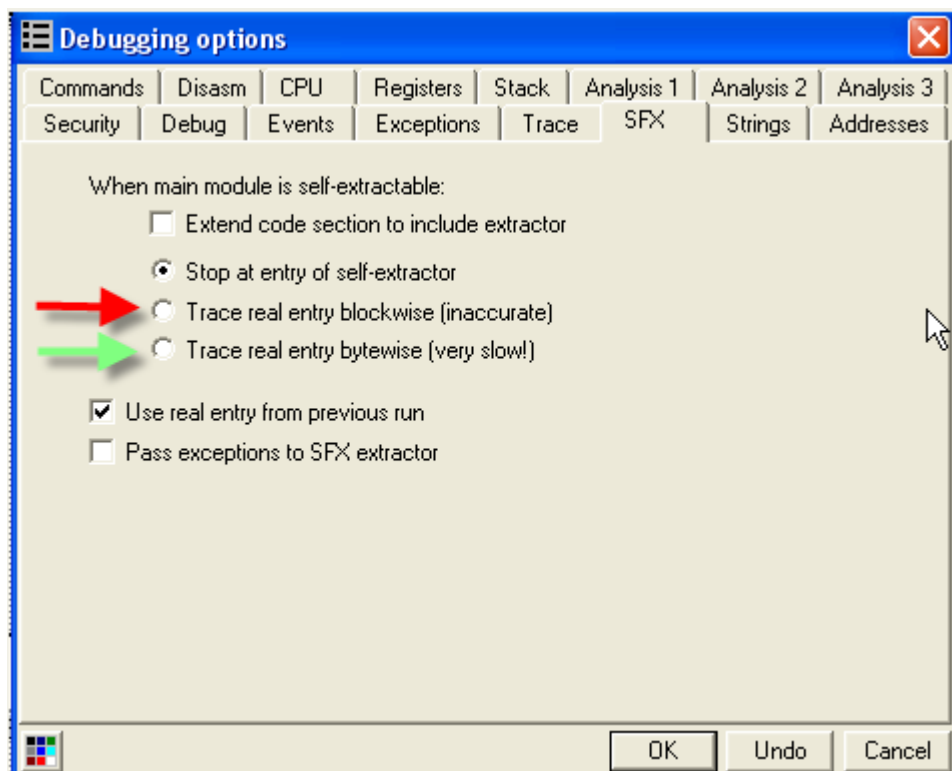


这里没有搜到任何记录,但是如果存在这样的指令的话,那么会在一个列表里面全部显示出来,我们就可以依次在每条记录上单击鼠标右键选择设置断点,当断下来的时候,我们可以看 EAX 的值为多少,根据 EAX 的值来确实是不是 CALL 或者 JMP 转移到第一个区段中。

这种搜索机器码的方式大家不是很常用,因为现在大部分的壳的解密例程都具有自修改功能,特别对于跳转到 OEP 这样的指令通常都是做了隐藏处理的,就是为了防止 Cracker 搜索机器码,但是,该方法对于一些简单的壳还是挺管用的。

## 2)使用 OllyDbg 自带的功能定位 OEP(SFX 法)

演示这种方法目标程序我们还是选择 CRACKME UPX.EXE,用 OD 加载该程序,然后选择菜单项 Options-Debugging options-SFX。



上图中 SFX 选项中用箭头标注出来的两个选项就是 OllyDbg 用来辅助定位 OEP 的,红色箭头标注的选项定位速度快,绿色箭头标注的选项定位速度慢,但是定位更加精确一些(PS:因为是按字节来定位的),我们来实验一下,选择红色箭头的选项。

重新加载该程序,会发现该选项并没有起作用,是因为如下原因:

## Self-extracting (SFX) files

Self-extracting file consists of extracting routine and packed original program. When troubleshooting SFX, you usually want to skip extractor and stop on the entry point of original program ("real entry"). OllyDbg contains several functions that facilitate this task.

Usually extractor loads to address that is outside the executable section of the original program. In this case OllyDbg recognizes file as SFX.

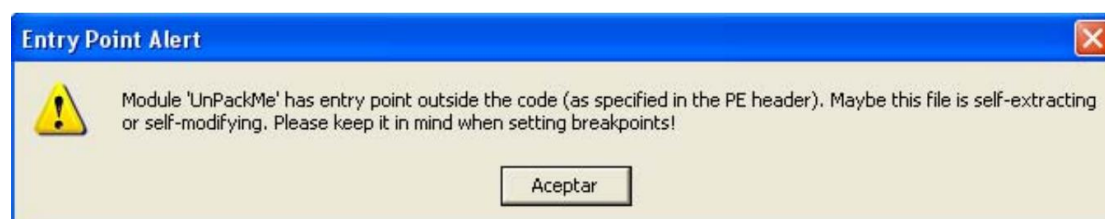
When [SFX options](#) request tracing of real entry, OllyDbg sets memory breakpoint on the whole code section. Initially this is empty or contains compressed data. When program attempts to execute some command within protected area which is neither `RET` nor `JMP`, OllyDbg reports real entry. This is how byte-wise extraction works.

This method is very slow. There is another, much faster method. Each time exception on data read occurs, OllyDbg enables reading from this 4-K memory block and disables previous read window. On each data write exception it enables writing to this block and disables previous write window. When program executes command within remaining protected area, OllyDbg reports real entry. However, when real entry is inside read or write window, its location will be reported incorrectly.

我们可以看到 OllyDbg 帮助文档中的解释是,该选项只有当 OllyDbg 发现入口点位于代码段之外的时候才会起作用,而当前这个程序的入口点恰恰是位于代码段中的,所以 OllyDbg 的该选项就不起作用了。壳的入口点位于代码段中的情况还是比较少见的。为了演示如何该选项在什么情况下能够起作用,这里给大家提供了一个名为 `UnPackMe_ASPack2.12` 的小程序,从名称上来看大家就应该可以猜到使用了 ASPack 壳。

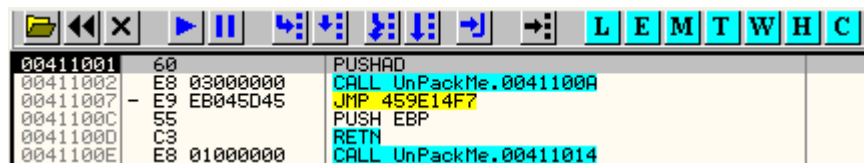
首先我们使用 Options-Debugging options-SFX 中默认的选项,看看 OllyDbg 会不会检测到入口点位于代码段之外。

我们可以看到 OllyDbg 弹出了一个消息框显示入口点位于代码段之外。

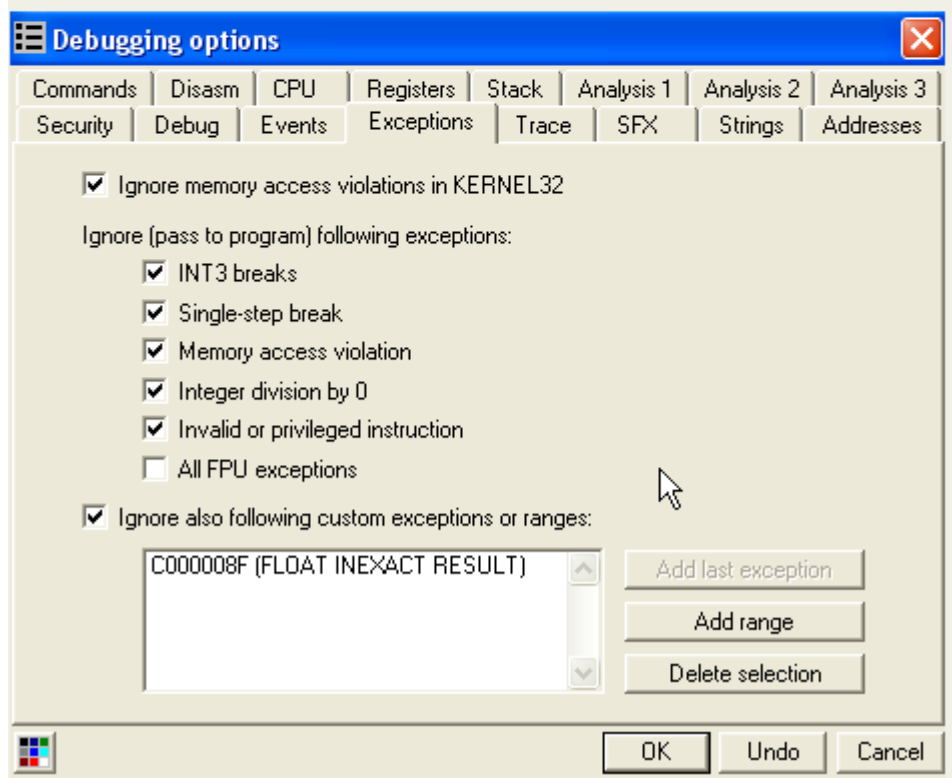
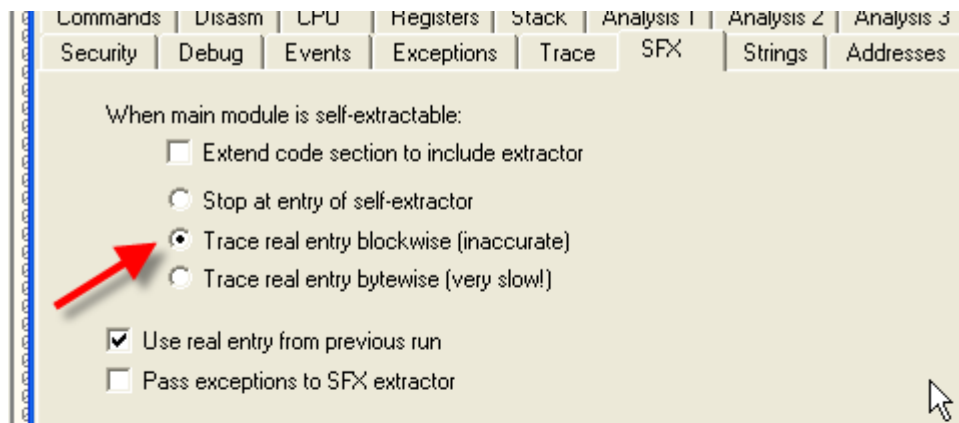


我们单击 Acceptar(确定)按钮。

到了壳的入口点处。

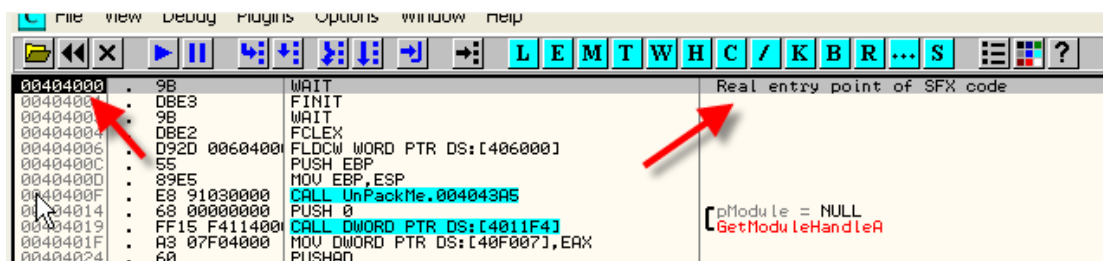


现在我们勾选上红色箭头标注的选项,并确保 EXCEPTIONS 菜单项中的忽略异常的选项都被勾选上了,重启 OD。



运行起来。

我们可以看到停在了 404000 处,OllyDbg 显示《Real entry point of SFX code》,即“真正的自解压代码的入口点”。(该程序是一个特例,OEP 并不位于第一个区段,是位于第三个区段)

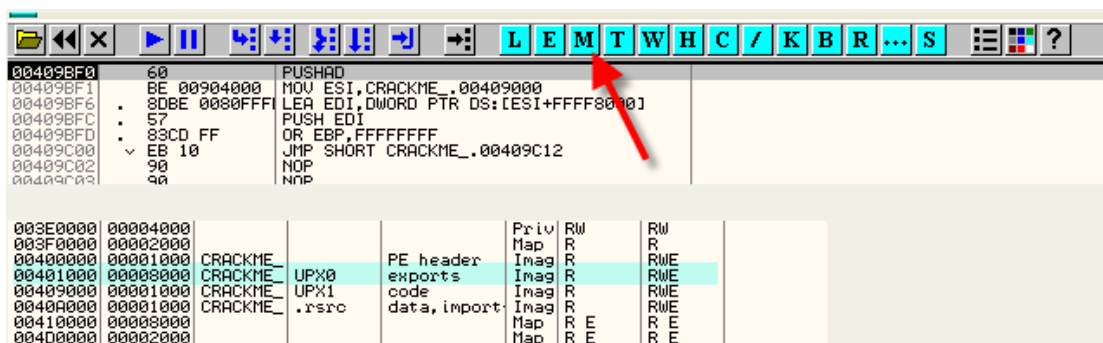


这里我们可以看到该选项起作用了,定位到了 OEP,现在我们来试试绿色箭头标注的选项,OllyDbg 显示该选项定位 OEP 会很慢。我们重启 OllyDbg 发现不一会儿就又停在了 OEP 处(PS:不是说很慢吗?),这是因为壳的解密例程太简短了,如果壳的解密例程很长的话,这两个选项的速度对比就能体现出来了。

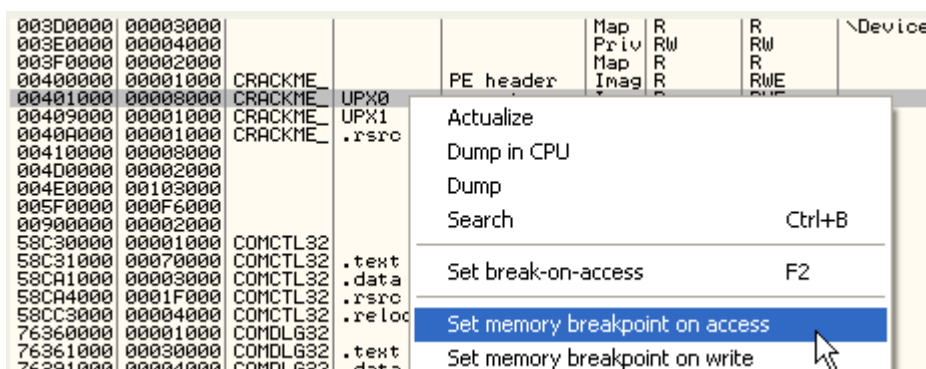
这里大家要注意,在使用完该选项以后要记得恢复其默认的选项,如果不恢复默认选项的话,OD 在分析其他正常的程序的时候就不会停在正常的入口点处了,进而影响程序的分析工作。

### 3)使用 Patch 过的 OD 来定位 OEP(即内存映像法)

就是我们在 VB 章节中使用的那个 Patch 过的 OD,即正常的内存访问断点读取,写入,执行的时候都会断下来,该 Patch 过的 OD 内存访问断点仅当执行的时候才会断下来,我们可以利用这一点来定位 OEP,我们还是来看看 CRACKME UPX,首先来看看区段列表。



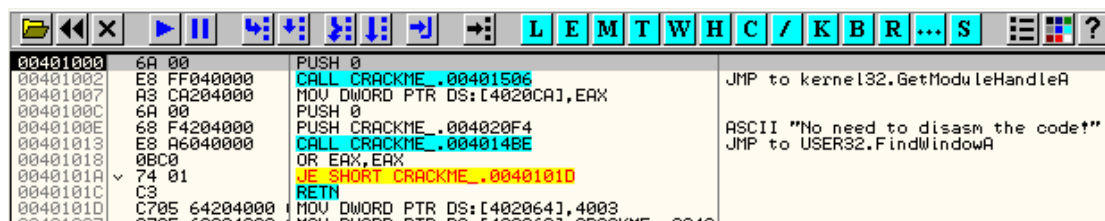
UPX 壳的解密例程会解密原程序的各个区段并将各个区段原始字节写回到原处,我们最好不要在解密区段的过程中断下来,说不定要断成千上万次才能到达 OEP,这里有了这个 Patch 过的 OD 就方便多了,其内存访问断点仅当执行的时候才会断下来,当其在执行第一个区段中的代码时,基本上就可以断定是 OEP 了。



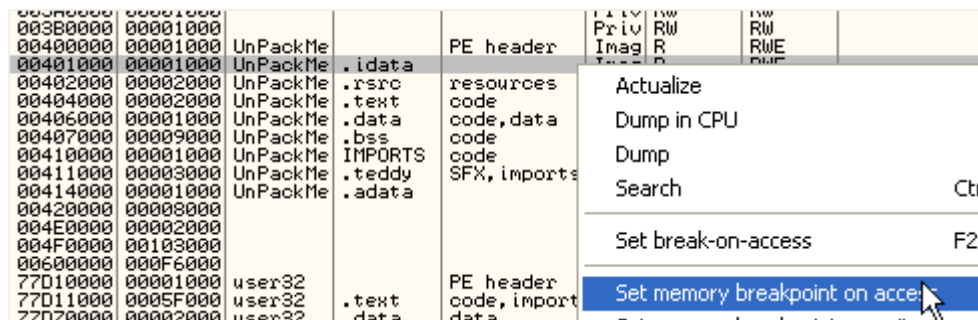
同样这里也要勾选上了忽略各种异常的选项,运行起来。

这种方法可能有点慢,因为设置的是内存访问断点(PS:写过调试器的童鞋应该知道,内存断点的机制决定了其慢的特性)。这里定位 OEP 的过程快则几秒钟,慢则几分钟不等(PS:大家可以去泡杯咖啡慢慢等,嘿嘿)。

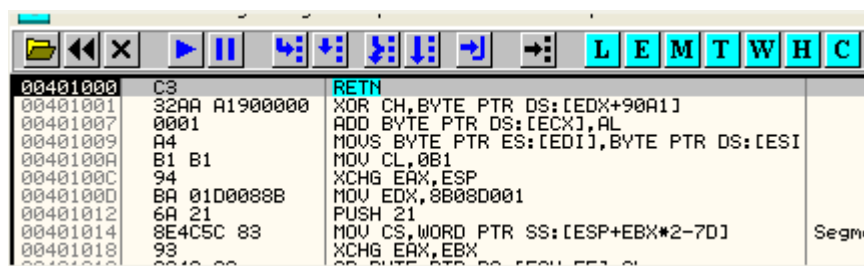
不一会儿我们会发现定位到了 OEP。



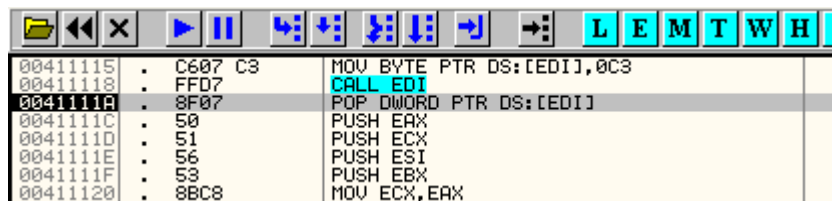
接着我们再来看看 UnPackMe\_ASPack2.12。



运行起来。



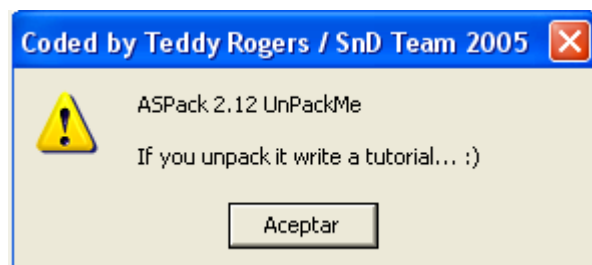
我们可以看到执行的第一行是这里,我们按下 F7 键看看会发生什么。



我们可以看到还是返回到了壳的解密例程中了,我们再次运行起来,程序直接运行起来了,为什么会这样呢?

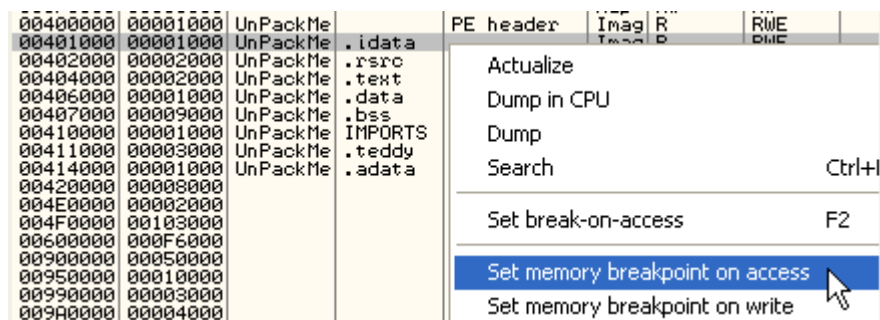
因为这里壳的解密例程并不是将原程序代码段解密到第一个区段中,所以我们可以继续给后面的区段设置内存访问断点,逐一排查。

我们再解释一下整个流程,直接运行该 CrackMe 的效果如下:

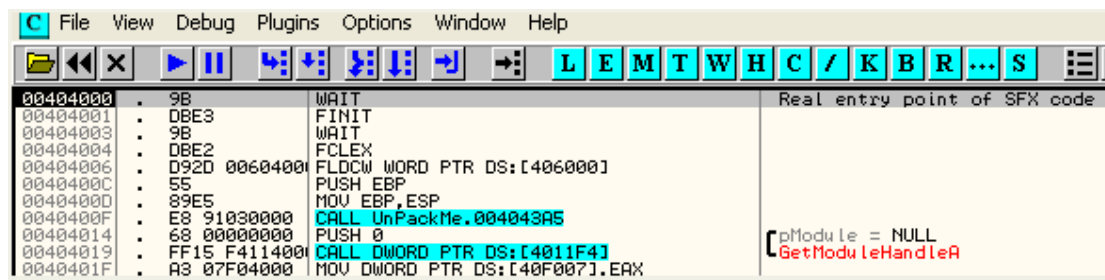
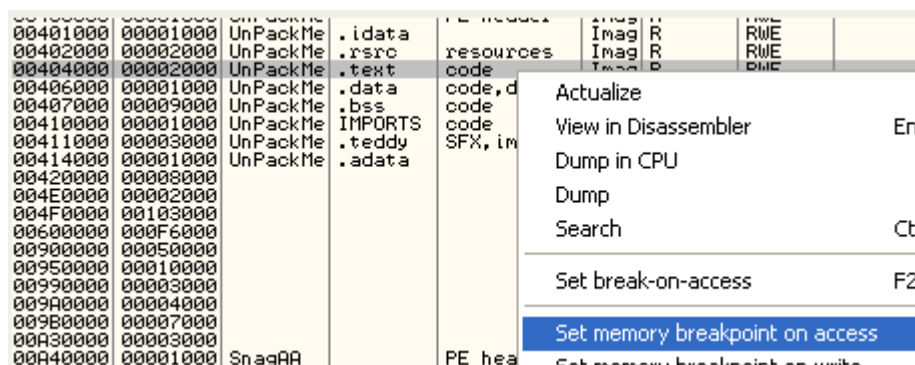


如果弹出了这个窗口就表示该程序在内存中解密区段完毕了,原程序的代码得以执行,我们依次给各个区段设置内存访问断点,然后运行起来,如果程序直接运行起来了,就证明这个区段不是我们要定位的,继续给下一个区段设置内存访问断点。





这里由于给第一个区段设置内存访问断点断了下来,但是继续执行程序就运行起来了,所以我们继续给第二个区段设置内存访问断点,当我们给第三个区段设置内存访问断点的时候其断在了 404000 地址处。



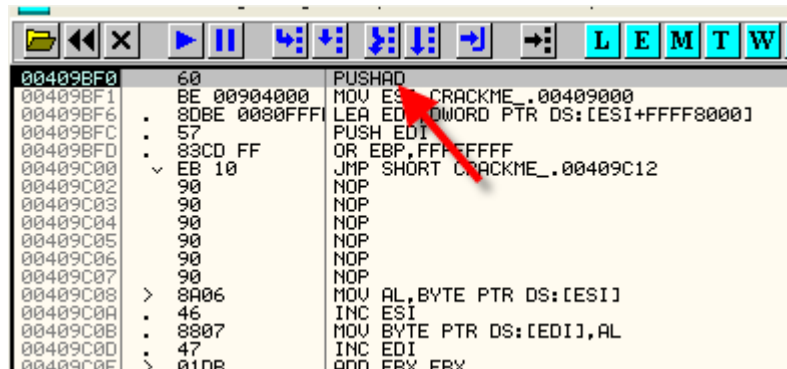
这里我们可以看到断在了 OEP 处,如果大家还不放心的话,继续运行,会发现继续断在下一条指令处,可以进一步断定这里是 OEP 了。

这种方法对很多壳都有效,嘿嘿。

#### 4)堆栈平衡法(即 ESP 定律法)

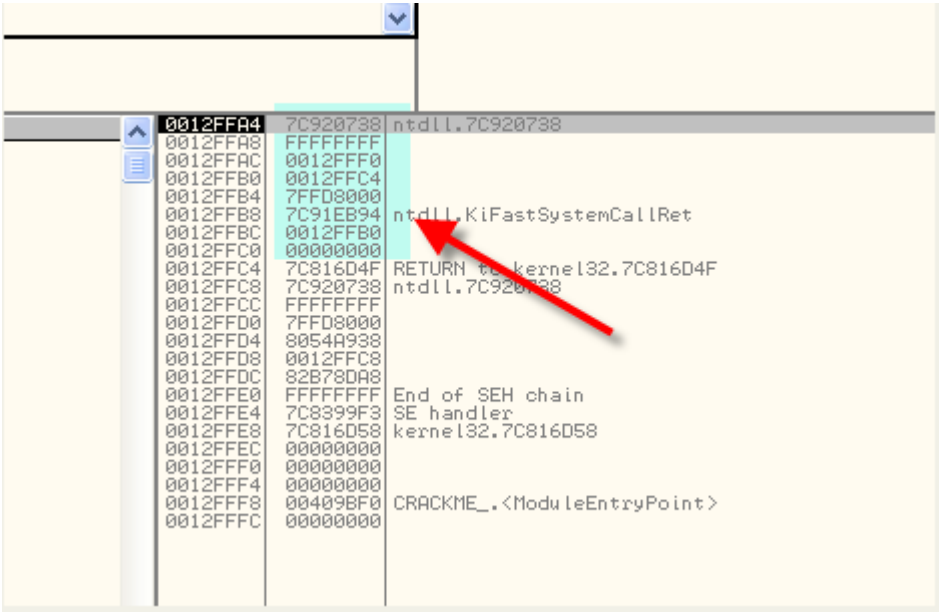
这种方法适用于一些古老的壳。这些壳首先会使用 PUSHAD 指令保存寄存器环境,在解密各个区段完毕,跳往 OEP 之前,会使用 POPAD 指令恢复寄存器环境。

这里我们来看看 CRACKME UPX。

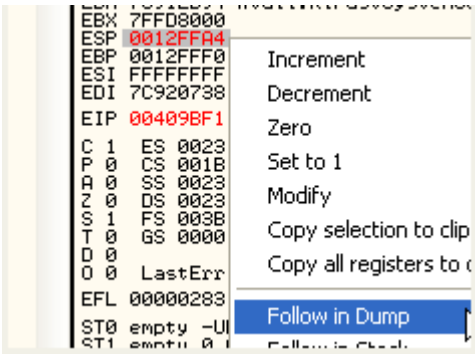


我们可以看到第一条指令就是 PUAHAD,有的情况下保存寄存器环境可能不是第一条指令,但也在附近了,还有些情况下,有些壳不使用 PUSHAD,而是逐一 PUSH 各个寄存器(例如:PUSH EAX,PUSH EBX 等等),总而言之,在解密完区段,跳往 OEP 之前会恢复寄存

器环境。  
这里我们按 F7 键执行 PUSHAD:



可以看到各个寄存器的初始值被压入到堆栈中了,这里我们可以对这些初始值设置内存或者硬件访问断点,当解密例程读取这些初始值的时候就会断下来,断下来处基本上就在 OEP 附近了。  
这里我们可以通过在 ESP 寄存器值上面单击鼠标右键选择-Follow in dump 在数据窗口中定位到这些寄存器的初始值。

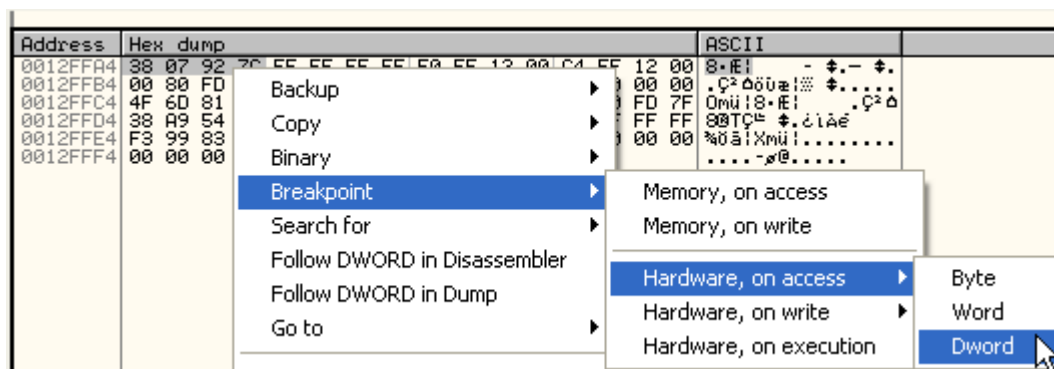


数据窗口中显示的堆栈内容如下:

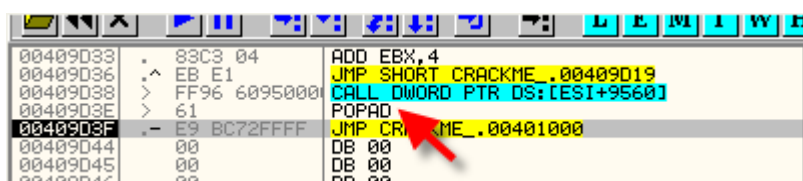
Address	Hex dump	ASCII
0012FFA4	38 07 92 7C FF FF FF FF F0 FF 12 00 C4 FF 12 00	8·Æ! - - -
0012FFB4	00 80 FD 7F 94 EB 91 7C B0 FF 12 00 00 00 00 00	.C²Äöü!... ..
0012FFC4	4F 6D 81 7C 38 07 92 7C FF FF FF FF 00 80 FD 7F	Omü!8·Æ! .C²Ä
0012FFD4	38 A9 54 80 C8 FF 12 00 A8 8D B7 82 FF FF FF FF	88TÇ ½.äiaë
0012FFE4	F3 99 83 7C 58 6D 81 7C 00 00 00 00 00 00 00 00	%0ä!Xmü!.....
0012FFF4	00 00 00 00 F0 9B 40 00 00 00 00 00 00 00 00 00	....-g@.....

这里我们可以对这些初始值的第一个字节或者前 4 个字节设置硬件访问断点。



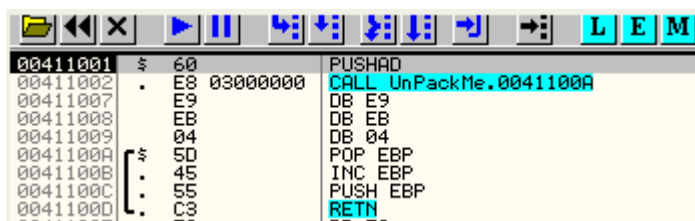


选择字节,字,双字都可以,只要解密例程在读取这些值的时候断下来就 OK,运行起来。

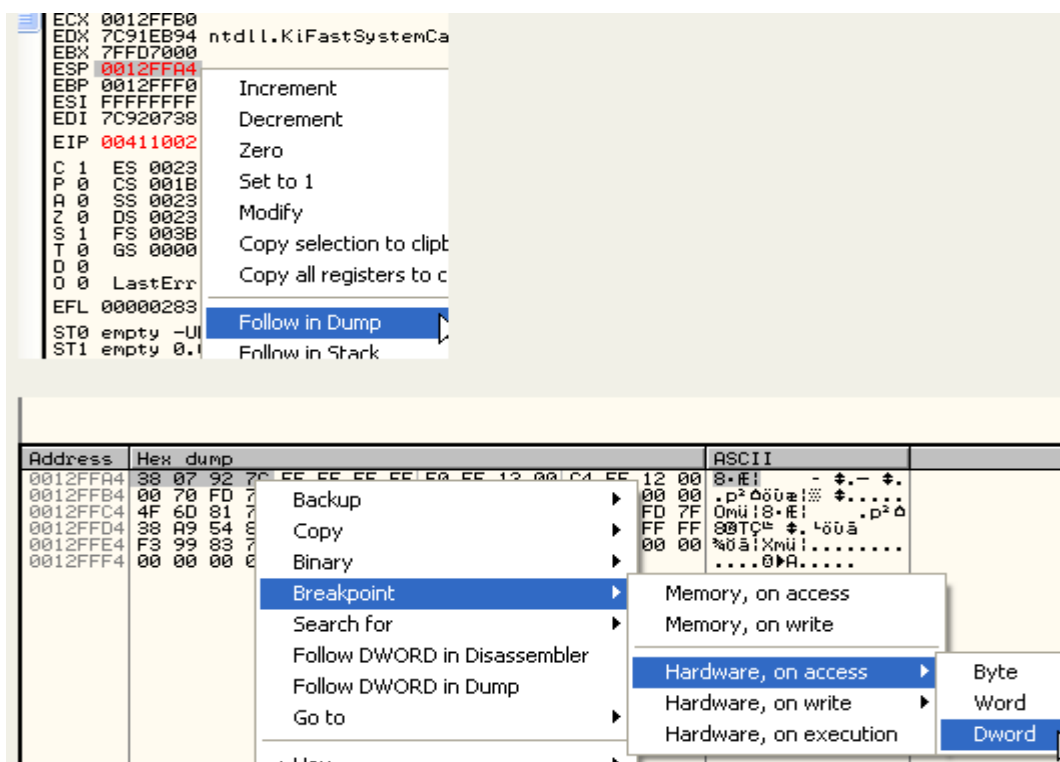


我们可以断在了 POPAD 指令的下一行,当壳的解密例程读取该值的时候断了下来,紧接着下面就是跳往 OEP 处,说明这个方法起作用了。

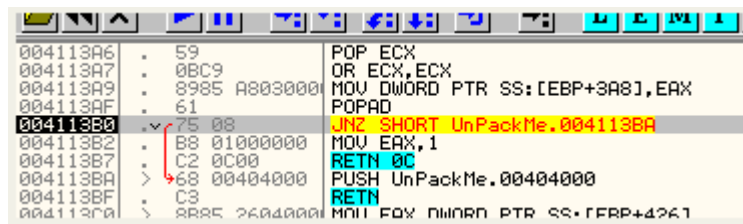
我们再来看看 UnPackMe\_ASPack2.12。



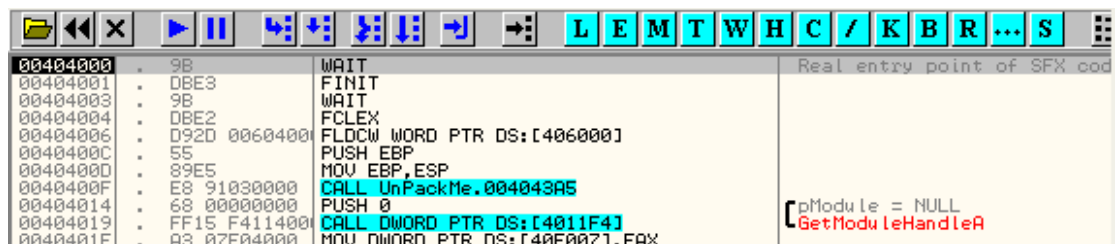
我们可以看到第一行也是 PUSHAD,我们依然按 F7 键执行 PUSHAD,然后还是通过在 ESP 寄存器值上面单击鼠标右键选择-Follow in dump 在数据窗口中定位到这些寄存器的初始值。



运行起来。



我们可以看到断在了跳往 OEP PUSH 404000 指令之前,我们继续按 F7 键单步。



可以看到到了 OEP 处。

这里要给大家说明一点,现在很多壳都能检测这种方法,所以说大家可以多多汲取一些方法和经验,尝试不同的方法,才能知道那种方法最合适。

我们继续来看其他定位 OEP 的方法。

#### 5)VB 应用程序定位 OEP 法(Native 或者 P-CODE)

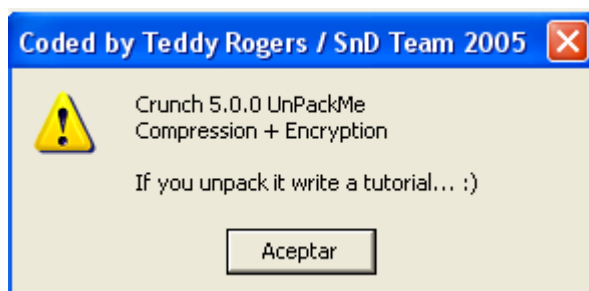
定位 VB 程序的 OEP 比较容易,因为 VB 应用程序都有一个特点-开始都是一个 PUSH 指令,紧接着一个 CALL 指令调用一个 VB API 函数。我们可以使用 Patch 过的 OD,首先定位到 VB 的动态库,接着给该动态库的代码段设置内存访问断点,当壳的解密例程解密完原程序各个区段,接着就会断在 VB DLL 的第一条指令处,接着我们可以在堆栈中定位到返回地址,就可以来到 OEP 的下一条指令处。这里我们也可以使用前面介绍的方法-跟逐一给各个区段设置内存访问断点(使用 Patch 过的 OD),但是很多壳会检测这种方法,所以大家可能根据需要不同的情况来尝试这不同的方法。这种方法很容易理解,我就不举例子了,以后大家如果遇到 VB 程序可以试试这种方法。

#### 6)最后一次异常法

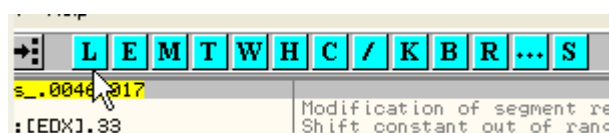
如果我们在脱壳的过程中发现目标程序产生大量异常的话,就可以使用最后一次异常法,我们来看一个例子,名字叫做“bitarts\_evaluations.c”。

我们还是使用 Patch 过的 OD 来加载它,并且配置好反调试插件。

然后将 EXCEPTIONS 菜单项中的忽略各个异常的选项都勾选上,运行起来。



我们可以看到程序运行起来了,我们单击工具栏中 L 按钮打开日志窗口。



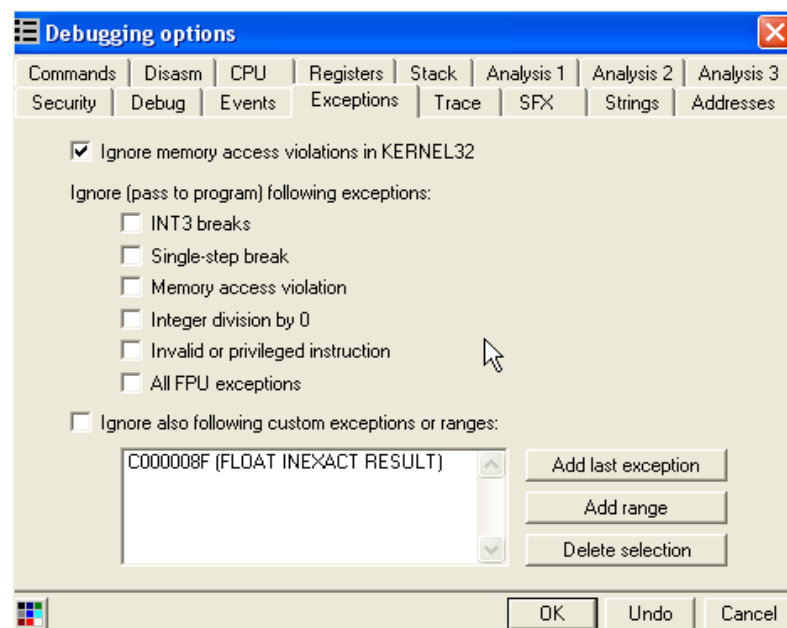
```

0046F3F0 Access violation when reading [00000000]
7C810856 New thread with ID 00000CEC created
0046EF14 Access violation when reading [00000000]
0046ECA1 Integer division by zero
00B8008E INT3 command at 00B8008E
0046EF14 Access violation when reading [00000000]
0046ECA1 Integer division by zero
00BA008E INT3 command at 00BA008E
0046EF14 Access violation when reading [00000000]
0046ECA1 Integer division by zero
00BC008E INT3 command at 00BC008E
0046EF14 Access violation when reading [00000000]
0046ECA1 Integer division by zero
00BE008E INT3 command at 00BE008E
0046EF14 Access violation when reading [00000000]
0046ECA1 Integer division by zero
Thread 00000CEC terminated, exit code 46FE79 (4652665.)
0046E88F INT3 command at bitarts_.0046E88F
5B480000 Module C:\WINDOWS\system32\umdmxfrm.dll
5B1F0000 Module C:\WINDOWS\system32\uxtheme.dll
746B0000 Module C:\WINDOWS\system32\MSCTF.dll
73260000 Module C:\WINDOWS\system32\RICHED32.DLL
74DC0000 Module C:\WINDOWS\system32\RICHED20.dll
60300000 Module C:\Archivos de programa\Yahoo!\Messenger\idle.dll
7C340000 Module C:\Archivos de programa\Yahoo!\Messenger\MSUCR71.dll
10000000 Module C:\Archivos de programa\Sunbelt Software\Personal Firewall 4\gk
76F10000 Module C:\WINDOWS\system32\wtsapi32.dll
76310000 Module C:\WINDOWS\system32\WINSTA.dll
76310000 Unload C:\WINDOWS\system32\WINSTA.dll
76F10000 Unload C:\WINDOWS\system32\wtsapi32.dll
00B60000 Module C:\Archivos de programa\TechSmith\SnagIt 8\SnagAA.dll
74C10000 Module C:\WINDOWS\system32\OLEACC.dll
76030000 Module C:\WINDOWS\system32\MSUCP60.dll
77730000 Module C:\WINDOWS\system32\shdocvw.dll
77A50000 Module C:\WINDOWS\system32\CRYPT32.dll
77AF0000 Module C:\WINDOWS\system32\MSASN1.dll
76890000 Module C:\WINDOWS\system32\CRYPTUI.dll
76BF0000 Module C:\WINDOWS\system32\WINTRUST.dll
76C50000 Module C:\WINDOWS\system32\IMAGEHLP.dll
597F0000 Module C:\WINDOWS\system32\NETAPI32.dll
77180000 Module C:\WINDOWS\system32\WININET.dll
76F20000 Module C:\WINDOWS\system32\LDAP32.dll
597F0000 Unload C:\WINDOWS\system32\NETAPI32.dll
76890000 Unload C:\WINDOWS\system32\CRYPTUI.dll
76BF0000 Unload C:\WINDOWS\system32\WINTRUST.dll
76C50000 Unload C:\WINDOWS\system32\IMAGEHLP.dll
76F20000 Unload C:\WINDOWS\system32\LDAP32.dll
77180000 Unload C:\WINDOWS\system32\WININET.dll
77730000 Unload C:\WINDOWS\system32\shdocvw.dll
77A50000 Unload C:\WINDOWS\system32\CRYPT32.dll
77AF0000 Unload C:\WINDOWS\system32\MSASN1.dll
00B60000 Unload C:\Archivos de programa\TechSmith\SnagIt 8\SnagAA.dll
74C10000 Unload C:\WINDOWS\system32\OLEACC.dll
76030000 Unload C:\WINDOWS\system32\MSUCP60.dll

```

这里我们可以看到产生了好几处异常,但是都不是位于第一个区段,说明这些异常不是在原程序运行期间发生的,是在壳的解密例程执行期间产生的异常,最后一次是 46e88f 处的这个异常。

好,现在我们重新启动 OD,将 EXCEPTIONS 菜单项中忽略的异常选项的对勾都去掉,仅保留 Ignore memory access violations in KERNEL32 这个选项的对勾。



我们运行起来,产生异常断了下来,我们直接按 SHIFT + F9 忽略异常继续运行。直到停在了 46E88F 处为止。

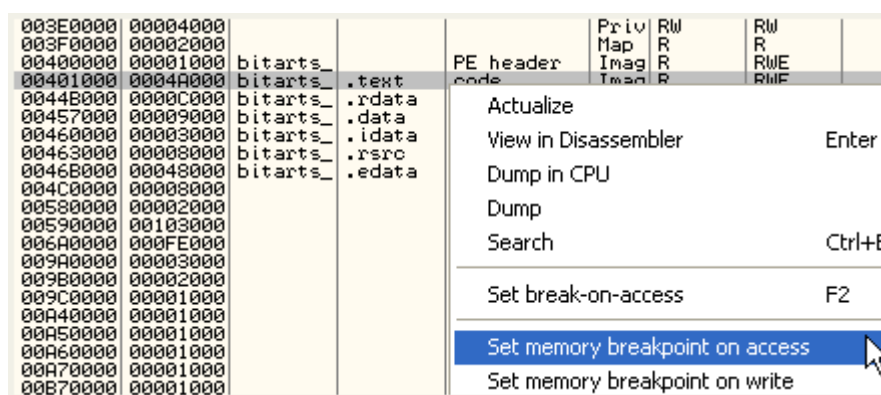
0046F3F0	8B00	MOV EAX,DWORD PTR DS:[EAX]
0046F3F2	64:8F05 000000	POP DWORD PTR FS:[0]
0046F3F9	83C4 04	ADD ESP,4
0046F3FC	C3	RETN
0046F3FD	4D	DEC EBP

这里不是,我们按 SHIFT + F9 忽略异常继续运行,我们知道最后一次异常是 46E88F 处的 INT3 指令引发的。

0046E88F	CC	INT3
0046E890	E8 28000000	CALL bitarts_.0046E8B0
0046E895	44	INC ESP
0046E896	12C3	ADC AL,BL
0046E898	0321	ADD ESP,DWORD PTR DS:[ECX]
0046E89A	64:8F05 000000	POP DWORD PTR FS:[0]
0046E89A	E8 17000000	CALL bitarts_.0046E8B0
0046E89C	DF12	FIST WORD PTR DS:[EDX]

这里是壳的解密例程执行过程中产生的最后一次异常,接着就是执行原程序的代码了。

接着我们可以对代码段设置内存访问断点,可能有人会问,为什么不在一开始设置内存访问断点呢?原因是很多壳会检测程序在开始是否自身被设置内存访问断点,如果执行到了最后一次异常处的话,很可能已经绕过了壳的检测时机,我们来试一试。



我们按 SHIFT + F9 忽略该异常运行起来。

004271B0	55	PUSH EBP
004271B1	8BEC	MOV EBP,ESP
004271B3	6A FF	PUSH -1
004271B5	68 600E4500	PUSH bitarts_.00450E60
004271BA	68 C8924200	PUSH bitarts_.004292C8
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
004271C5	50	PUSH EAX
004271C6	64:8925 000000	MOV DWORD PTR FS:[0],ESP
004271CD	83C4 A8	ADD ESP,-58
004271D0	53	PUSH EBX
004271D1	56	PUSH ESI
004271D2	57	PUSH EDI
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
004271D6	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
004271DC	33D2	XOR EDX,EDX
004271DE	8AD4	MOV DL,AH
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
004271E6	8BC8	MOV ECX,EAX

我们可以看到断在了 OEP 处,下面我们来看看该壳在开始的时候是否有检测内存访问断点。

我们重新加载该程序,将忽略的异常选项都勾选上,接着打开区段列表窗口,给第一个区段设置内存访问断点,过了很久断在了 OEP 处。

004271B0	55	PUSH EBP
004271B1	8BEC	MOV EBP,ESP
004271B3	6A FF	PUSH -1
004271B5	68 600E4500	PUSH bitarts_.00450E60
004271BA	68 C8924200	PUSH bitarts_.004292C8
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
004271C5	50	PUSH EAX
004271C6	64:8925 000000	MOV DWORD PTR FS:[0],ESP
004271CD	83C4 A8	ADD ESP,-58
004271D0	53	PUSH EBX
004271D1	56	PUSH ESI
004271D2	57	PUSH EDI
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
004271D6	FF15 DC0A4600	CALL DWORD PTR DS:[460ADC]
004271DC	33D2	XOR EDX,EDX
004271DE	8AD4	MOV DL,AH
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX
004271E6	8BC8	MOV ECX,EAX
004271E8	81E1 FF000000	AND ECX,0FF

虽然这里我们直接给第一个区段设置内存访问断点直接定位到了 OEP,但是了解某些壳会检测内存访问断点还是非常有必要的,如果我们在离 OEP 越近的地方设置内存访问断点,就越不容易被壳检测到。

好了,现在我们来试试第二种方法中介绍的 OD 自带的功能选项是否能够定位到 OEP。

```

004271B0 . 55      PUSH EBP
004271B1 . 8BEC    MOV EBP,ESP
004271B3 . 6A FF   PUSH -1
004271B5 . 68 60E45000 PUSH bitarts_.00450E60
004271B8 . 68 C8924200 PUSH bitarts_.004292C8
004271BF . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
004271C5 . 50      PUSH EAX
004271C6 . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
004271CD . 83C4 A8   ADD ESP,-58
004271D0 . 53      PUSH EBX
004271D1 . F7       PUSH ECX
  
```

同样定位到了 OEP。

现在我们来试试第四种方法中介绍的 ESP 定律,我们观察一下该壳的入口点:

```

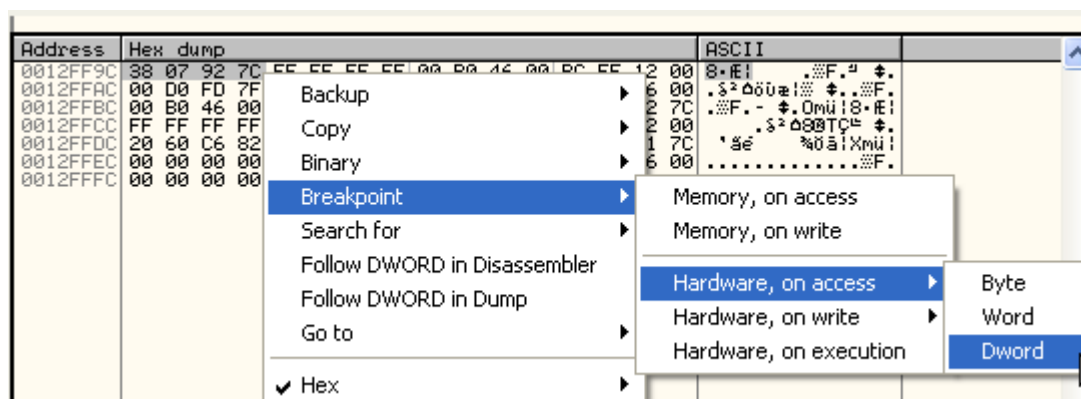
0046B000 $v EB 15  JMP SHORT bitarts_.0046B017
0046B002 . 03      DB 03
0046B003 . 00      DB 00
0046B004 . 00      DB 00
0046B005 . 0006    ADD BYTE PTR DS:[ESI],AL
0046B007 . 0000    ADD BYTE PTR DS:[EAX],AL
0046B009 . 0000    ADD BYTE PTR DS:[EAX],AL
0046B00B . 0000    ADD BYTE PTR DS:[EAX],AL
0046B00D . 0000    ADD BYTE PTR DS:[EAX],AL
0046B00F . 0000    ADD BYTE PTR DS:[EAX],AL
0046B011 . 0068 00  ADD BYTE PTR DS:[EAX],CH
0046B014 . 0000    ADD BYTE PTR DS:[EAX],AL
0046B016 . 0055 E8  ADD BYTE PTR SS:[EBP-18],DL
0046B019 . 0000    ADD BYTE PTR DS:[EAX],AL
0046B01B . 0000    ADD BYTE PTR DS:[EAX],AL
0046B01D . 5D      POP EBP
0046B01E . 81ED 1D000000 SUB EBP,1D
0046B024 . 8BC5    MOV EAX,EBP
0046B026 . 55      PUSH EBP
0046B027 . 60      PUSHAD
0046B028 . 9C      PUSHFD
  
```

我们按 F7 键单步跟踪几行就能到达 PUSHAD 指令处。

```

0046B017 > 55      PUSH EBP
0046B018 . E8 00000000 CALL bitarts_.0046B01D
0046B01D . 5D      POP EBP
0046B01E . 81ED 1D000000 SUB EBP,1D
0046B024 . 8BC5    MOV EAX,EBP
0046B026 . 55      PUSH EBP
0046B027 . 60      PUSHAD
0046B028 . 9C      PUSHFD
0046B029 . 2B85 FC070000 SUB EAX,DWORD PTR SS:[EBP+7FC]
0046B02F . 8985 E8070000 MOV DWORD PTR SS:[EBP+7E8],EAX
0046B035 . FF7424 2C   PUSH DWORD PTR SS:[ESP+2C]
0046B039 . E8 20020000 CALL bitarts_.0046B25E
0046B03E . 0F82 94060000 JB bitarts_.0046B6D8
0046B044 . F0 50040000 CALL bitarts_.0046B500
  
```

我们按 F7 键执行 PUSHAD 指令,接着在 ESP 寄存器值上面单击鼠标右键选择-Follow in dump 在数据窗口中定位到寄存器的初始值。



给前 4 个字节设置硬件访问断点,运行起来。

0046BBFD	64:8F05 0000	POP DWORD PTR FS:[0]	0012F
0046BC04	83C4 04	ADD ESP,4	
0046BC07	5D	POP EBP	
0046BC08	61	POPAD	
0046BC09	C3	RETN	
0046BC0A	57	PUSH EDI	

这里就断在了恢复寄存器环境的指令的下一行,我们按 F7 键单步执行到 RETN 处,接着再单步一下就能到达 OEP 处。

004271B0	55	PUSH EBP	Real entry point of SFX code
004271B1	8BEC	MOV EBP,ESP	
004271B3	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH bitarts_.00450E60	
004271B8	68 C8924200	PUSH bitarts_.004292C8	SE handler installation
004271BF	64:A1 000000	MOV EAX,DWORD PTR FS:[0]	
004271C5	50	PUSH EAX	
004271C6	64:8925 0000	MOV DWORD PTR FS:[0],ESP	
004271CD	83C4 A8	ADD ESP,-58	
004271D0	53	PUSH EBX	
004271D1	56	PUSH ESI	
004271D2	57	PUSH EDI	
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
004271D6	FF15 DC0A460	CALL DWORD PTR DS:[460ADC]	kernel32.GetVersion
004271DC	33D2	XOR EDX,EDX	

## 7)利用壳最常用的 API 函数来定位 OEP

我们还是用 Patch 过的 OD 加载 bitarts\_evaluations.c。将忽略的异常选项都勾选上,我们来定位一下壳最常用的 API 函数,比如 GetProcAddress,LoadLibrary。ExitThread 有些壳会用。我们首先来看看 GetProcAddress。

Command	? GetProcAddress	HEX: 7C80AC28
---------	------------------	---------------

我们可以看到该壳使用了 GetProcAddress,接着使用 bp GetProcAddress 命令给该 API 函数设置一个断点。

Command	Bp GetProcAddress
Program entry point	

如果在命令栏中使用 bp 命令设置断点失败的话,可以尝试手工设置断点。

运行起来。

7C80AC28	8BFF	MOV EDI,EDI	
7C80AC2A	55	PUSH EBP	
7C80AC2B	8BEC	MOV EBP,ESP	
7C80AC2D	51	PUSH ECX	
7C80AC2E	51	PUSH ECX	
7C80AC2F	53	PUSH EBX	
7C80AC30	57	PUSH EDI	
7C80AC31	8B7D 0C	MOV EDI,DWORD PTR SS:[EBP+C]	
7C80AC34	BB FFF0000	MOV EBX,0FFFF	

这里我们并不需要知道壳在哪些地方调用 GetProcAddress,所以我们在断下来的这一行上面单击鼠标右键选择-Breakpoint-Conditional log,来设置条件记录。

Modify conditional log breakpoint at kernel32.GetProcAddress

Condition:

Explanation: Expression:

Decode value of expression as: Assumed by expression

Never

On condition

Always

Pass count (dec.)

Pause program:

Log value of expression:

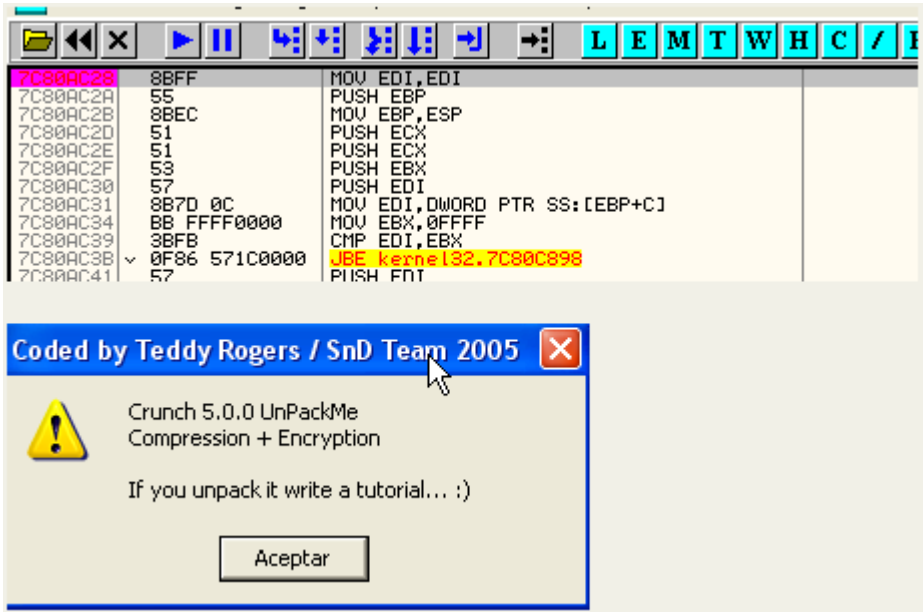
Log function arguments:

If program pauses, pass following commands to plugins:

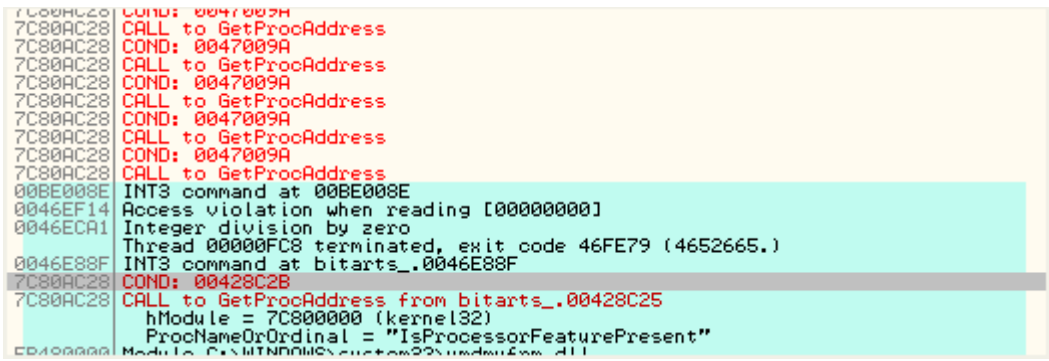
OK Cancel



这里我们将 Pause program 这一项勾选上 Never,记录的表达式设置为[ESP],也就是记录返回地址,这样我们就能知道哪些地方调用 GetProcAddress。接着在日志窗口中单击鼠标右键选择-Clear Log(清空日志)。



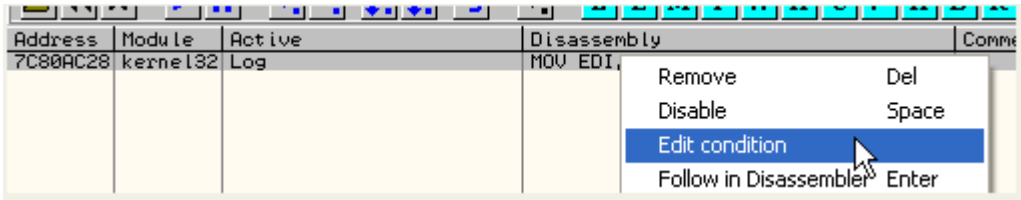
运行起来,我们可以看到程序的主窗口弹了出来,打开日志窗口,看看最后一次 GetProcAddress(排除掉第一个区段中调用的位置)是在哪里被调用的。



我们可以看到基本上 GetProcAddress 都是解密例程中调用的,除了 428C2B 这一处以外(这里是第一个区段中调用的,也就是原程序本身调用的)。所以我们要定位的应该是 47009A 这一处。接下来我们重新来编辑一下条件断点中断的条件,将中断条件设置为[ESP] == 47009A。

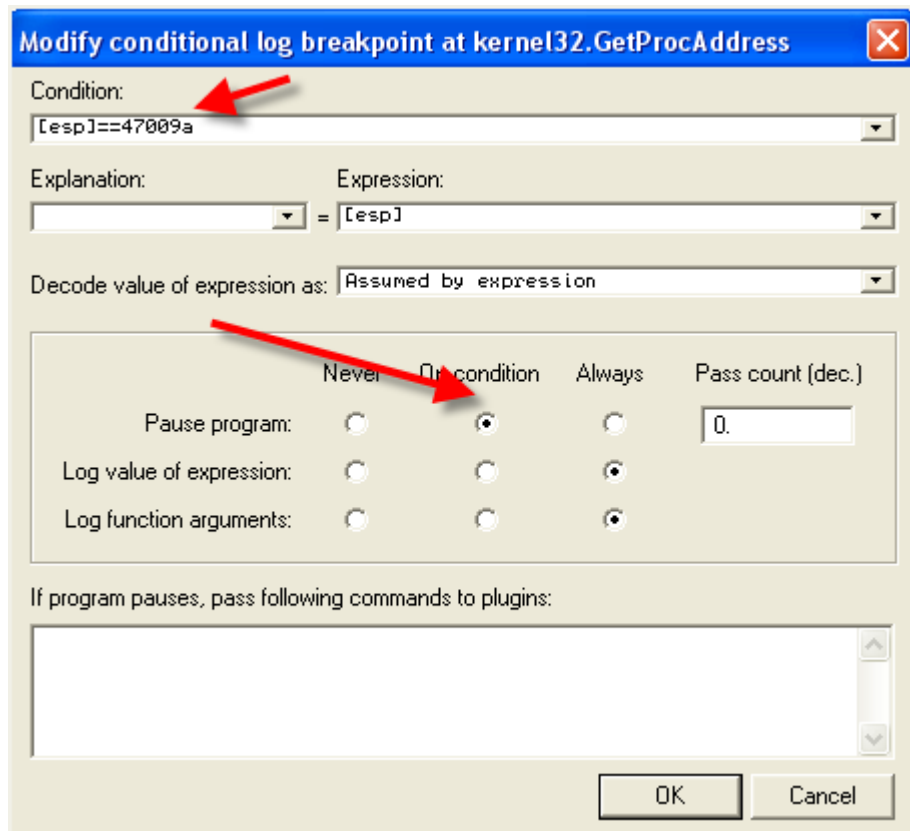
并且将 Pause program 这一项勾选上 On condition。

重新启动 OllyDbg。



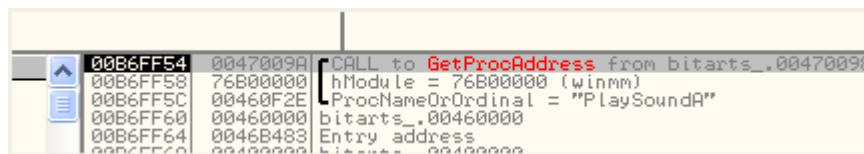
编辑条件断点。





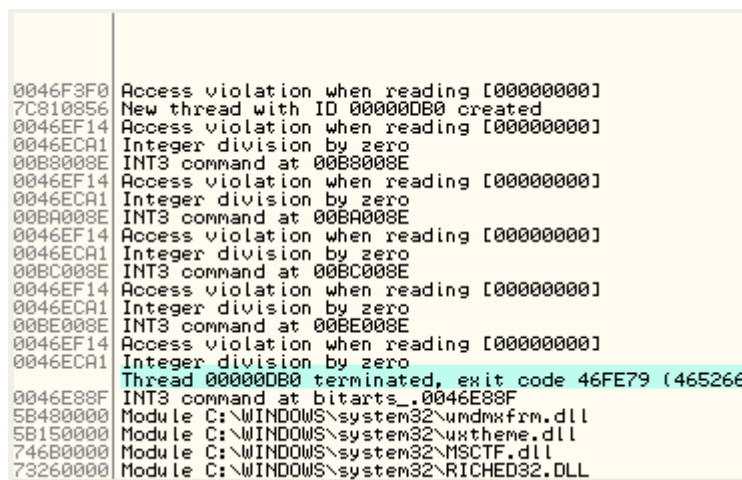
设置 Condition 为[ESP] == 47009A,接着将 Pause program 这一项勾选上 On condition。

运行起来。



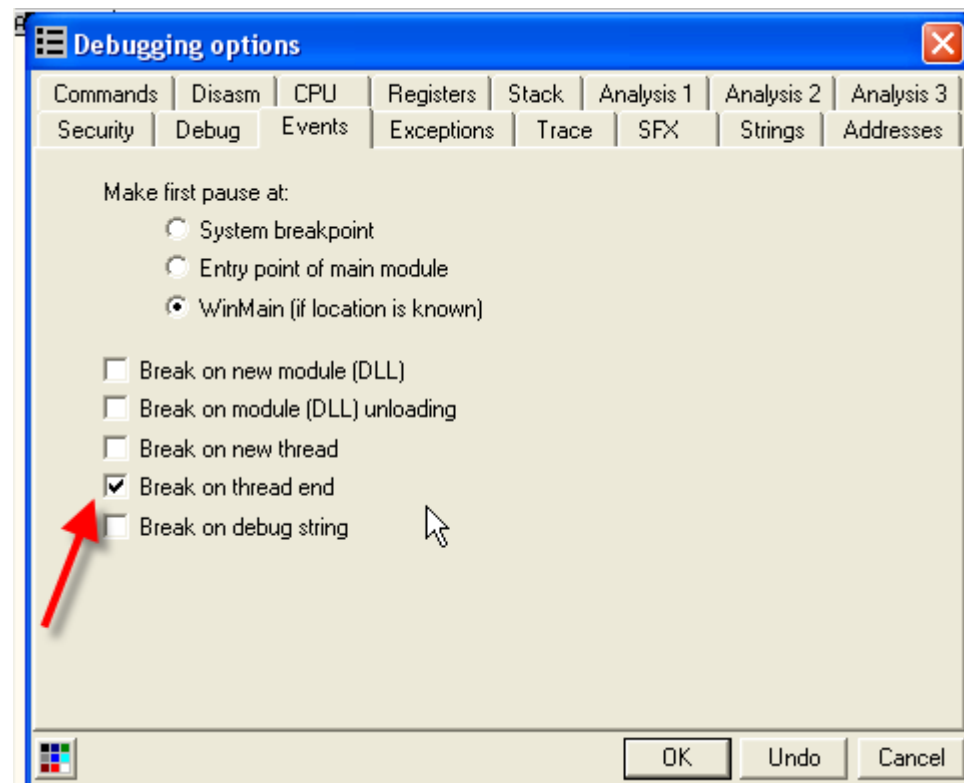
断了下来。我们可以在对代码段设置内存访问断点之前尝试一下这种方法,这样就可以绕过很多壳对内存断点的检测,但是有一些壳也会对 API 函数断点进行检测,所以说我们需要各种方式都尝试一下,找到最合适的。

对当前这个壳定位 GetProcAddress 的调用处是可行的,我们现在已经在 OEP 附近了。如果定位 GetProcAddress 的调用处失败的话,我们可以换其他的 API 函数,这里我们再来看看日志窗口,可以看到一处线程结束记录。

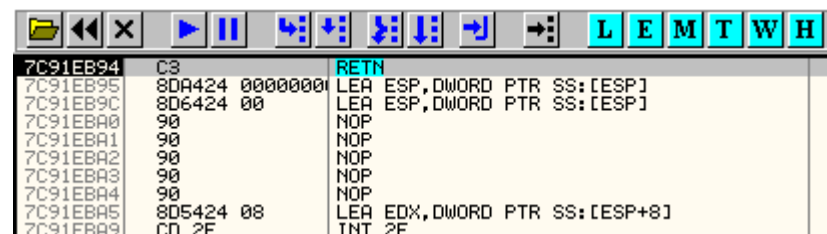


因此接下来给 ExitThread 设置断点,并且将菜单项 Debugging options-Events 中的 Break on thread end(在线程结束位置中断下来)勾

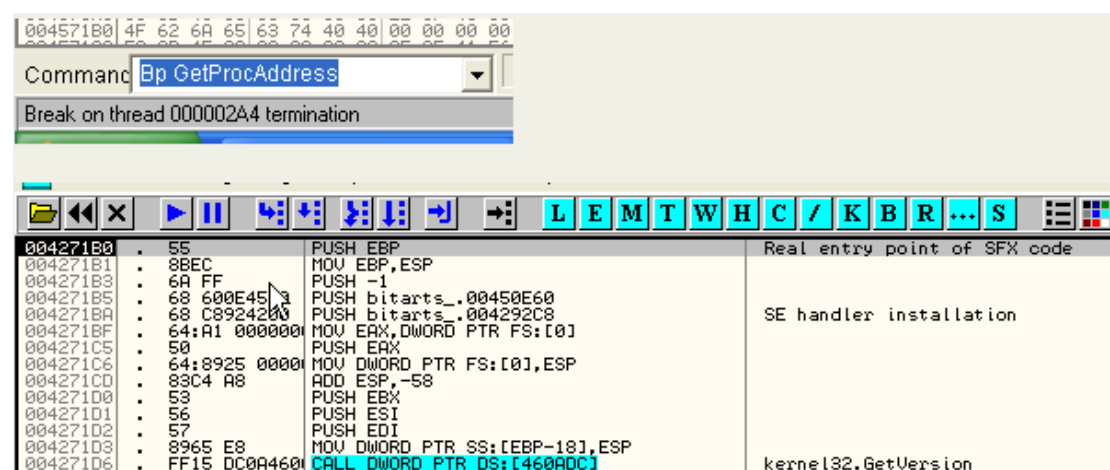
选上。



运行起来。



断在了线程结束的位置。

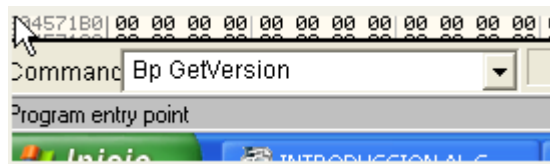


接着我们给代码段设置内存访问断点就能够马上定位到 OEP。

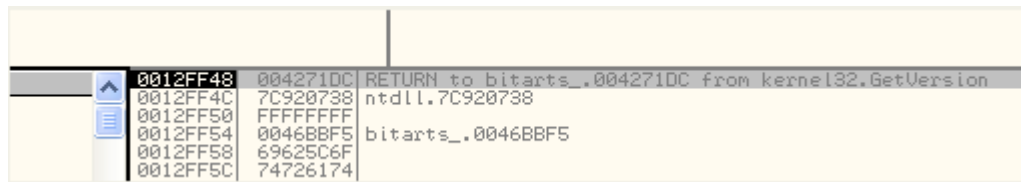
#### 8) 利用应用程序调用的第一个 API 函数来定位 OEP

这种方法就是直接给应用程序调用的第一个 API 函数设置断点,比如说,很多程序(VC++)一开始会调用 GetVersion, GetModuleHandleA, 对于 bitarts\_evaluations.c 来说我们可以断 GetVersion, 对于 CRACKME UPX 来说我们可以断

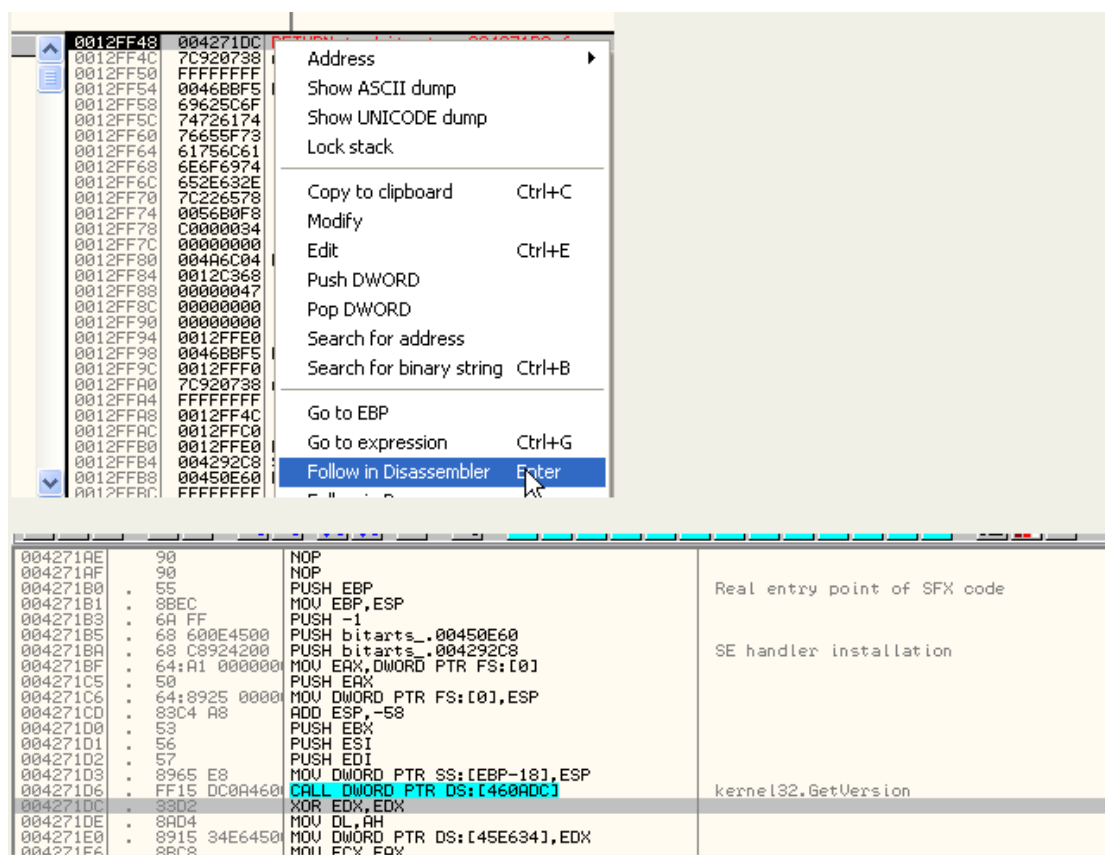
GetModuleHandleA。这里是 bitarts\_evaluations.c,所以我们给 GetVersion 设置断点。



运行起来。



这里我们断在了 GetVersion 的入口点处,从堆栈中我们可以看到返回地址位于第一个区段。我们直接在返回地址上面单击鼠标右键选择-Follow in Disassembler。



这里我们又定位到了 OEP。以上就给大家演示的如何利用应用程序调用的第一个 API 函数来定位 OEP 了。如果我们遇到有的壳检测 GetVersion 入口处的 INT 3 断点的话,我们可以尝试在该 API 函数的返回指令 RET 处下断。

其实还有很多适用于特定壳定位 OEP 的方法,这里就不给大家一一介绍了,基本上也是根据上面的这些基本方法变通来的,所以说大家掌握好上述这些基本的定位 OEP 的方法和原理就即可。

这里给大家留一个小程序练习,名字叫做 UnPackMe\_tElock0.98。大家尝试定位其 OEP。这个壳专门采取了一些技巧来干扰利用上述方法定位 OEP,所以说如果直接利用上述这些方法的话,就不能奏效了,大家可以好好琢磨一下这个壳。

大家记住,如果壳检测 INT 3 断点或者硬件断点的话,你使用 ESP 定律给堆栈中的寄存器初始值设置硬件断点也是不起作用的,只能换其他方法。

接下来的章节将介绍转储(dump),以及如何修复 IAT 等知识点。