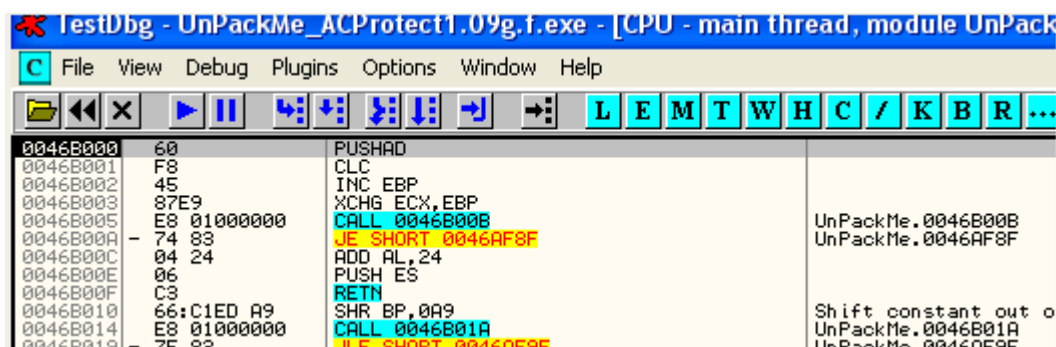


## 第四十二章-ACProtect V1.09 脱壳(寻找 OEP,绕过硬件断点检测,修复 Stolen code)

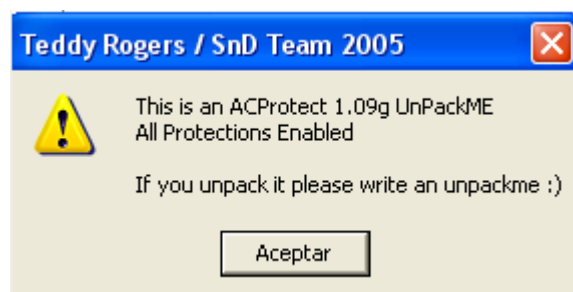
本章开始,我们将讨论更加复杂的壳-ACProtect 1.09。

我们一步步的来剖析这个壳的保护机制。

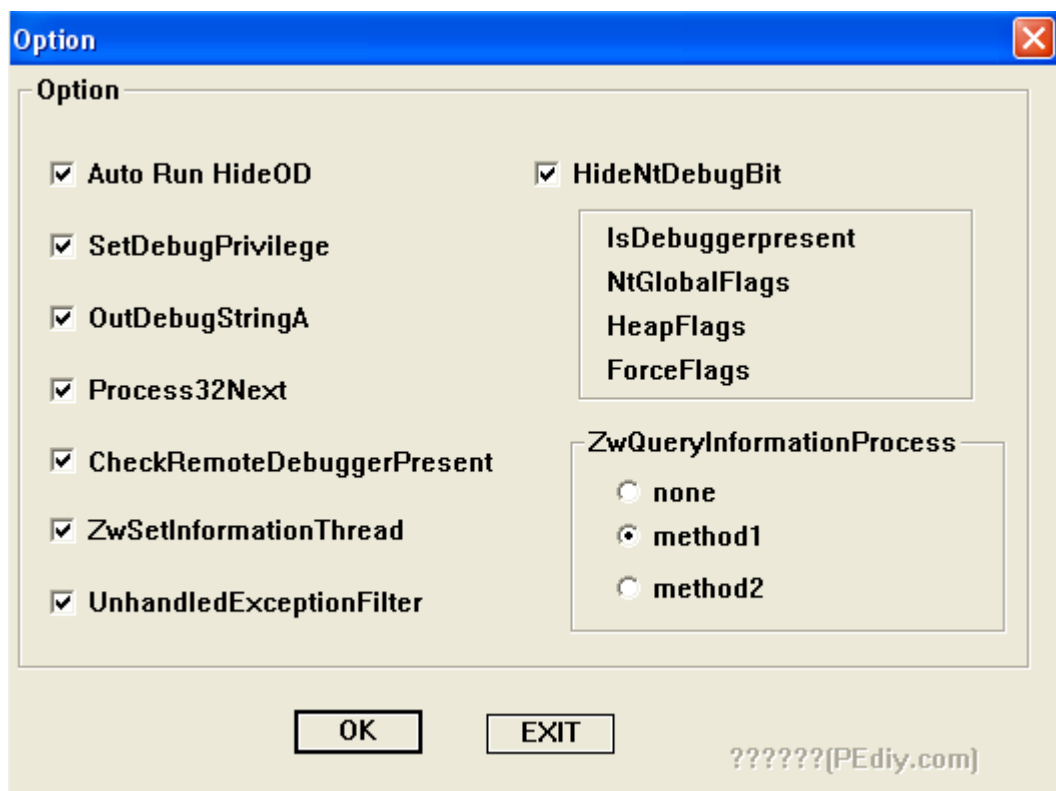
该例子的名称为 UnPackMe\_ACProtect1.09g.f。配合好反反调试插件,用 OD 加载之。



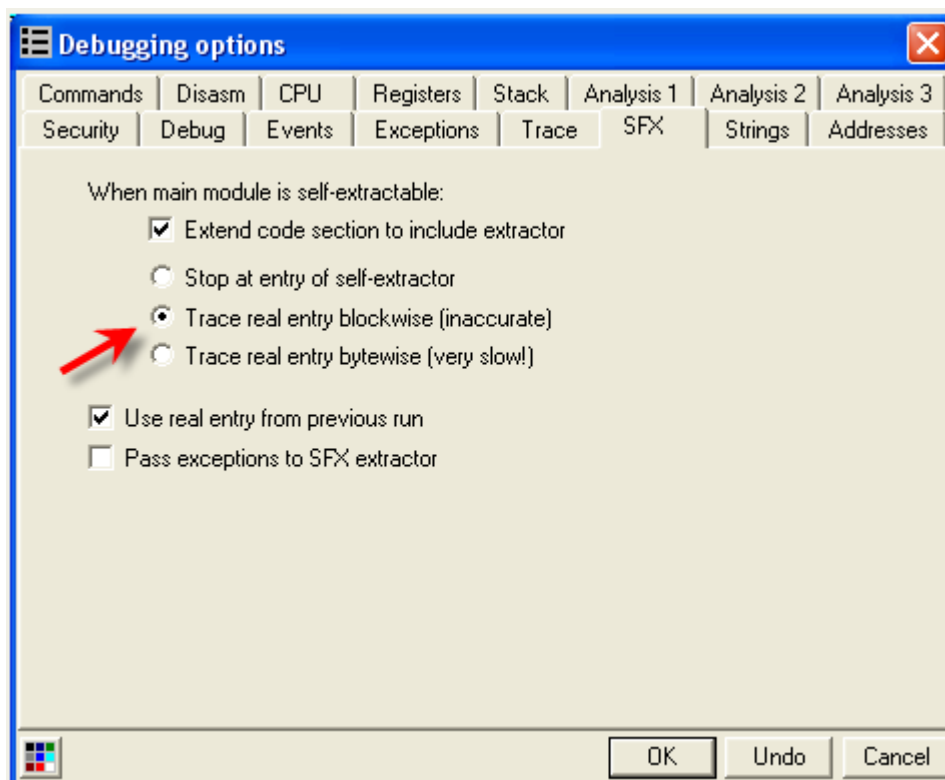
我们直接运行起来,看看会发生什么。



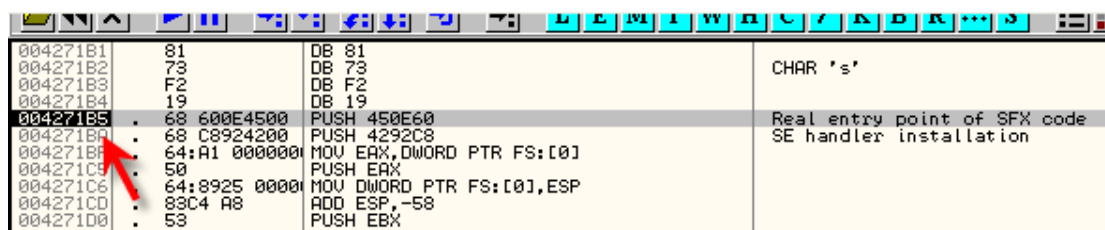
这里我用的是 Parcheado\_4(西班牙语:Patched\_4)这个 OD,接着按照下图来配置 HideOD 0.12 反反调试插件,这样就可以正常运行了,该壳会使用 Process32Next 这类 API 函数来检测 OD 进程,反反调试插件可以绕过。



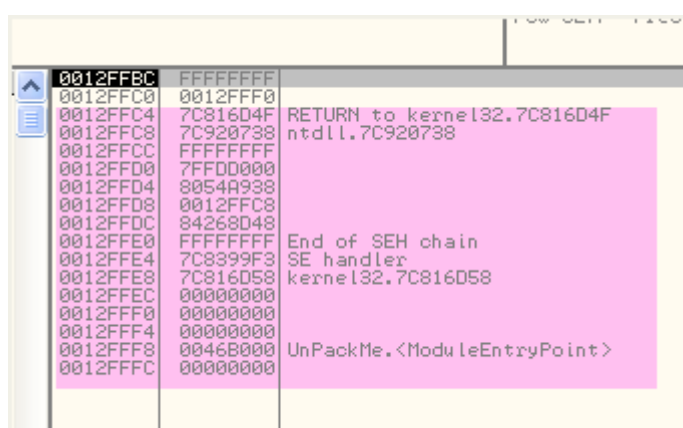
下面我们来定位 OEP。



前面章节我们介绍了很多定位 OEP 的方法,比如说可以使用 OD 自带的跟踪功能就能定位到 OEP,打开主菜单项 Debugging options-SFX,选择 Trace real entry blockwise(inaccurate)。接着重启 OD,不一会儿断在了这里。



现在的问题是这里是否是真正的 OEP,或者说是不是存在 stolen bytes。我们来看一看堆栈的情况。



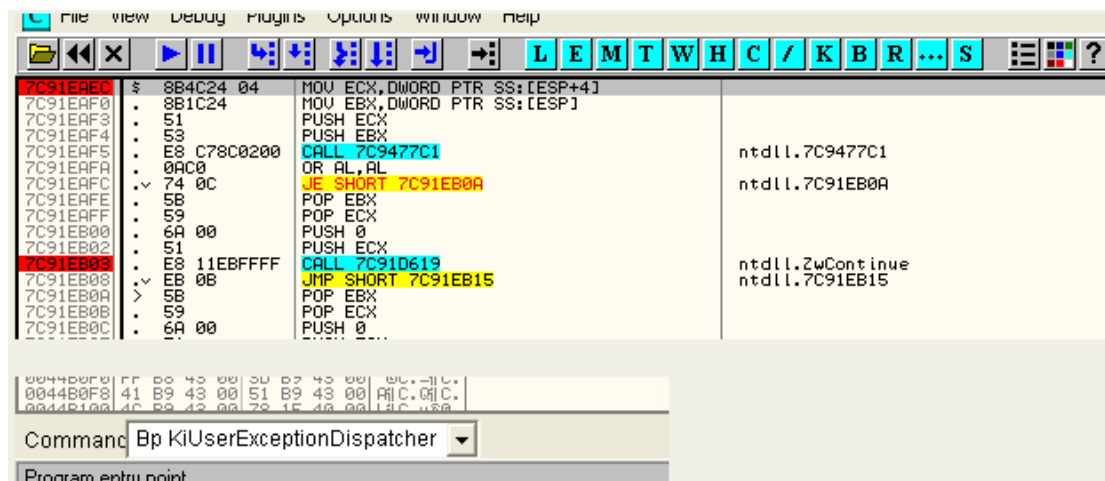
现在我们不勾选自动定位 OEP 的选项,重启 OD。

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDA000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84266B40	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0046B000	UnPackMe.<ModuleEntryPoint>
0012FFFC	00000000	

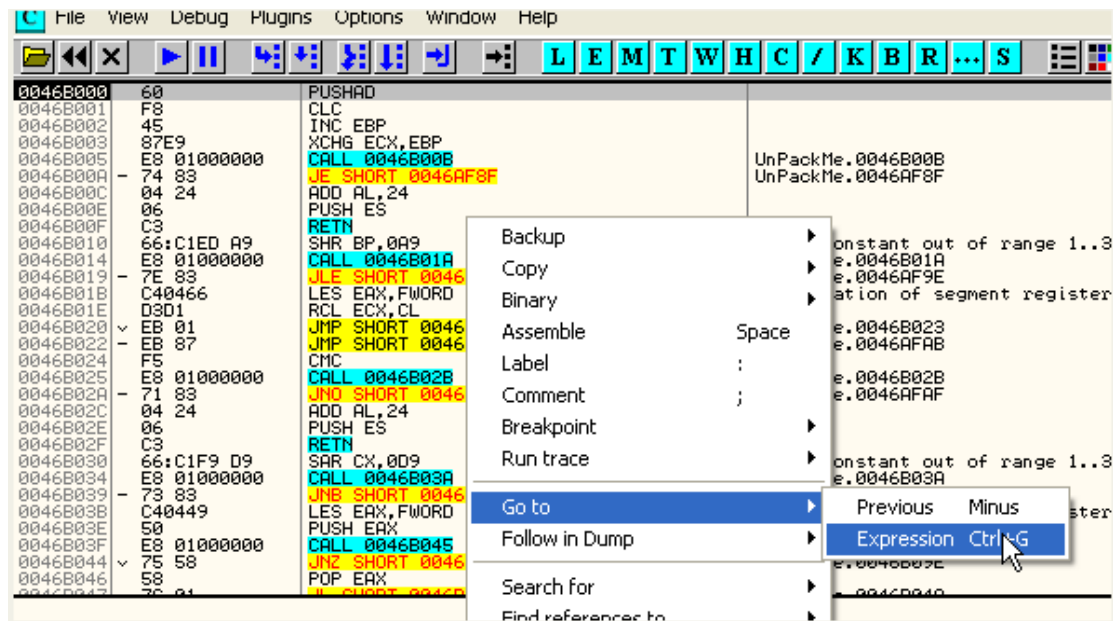
我们可以看到比开始的时候多了两个值。一般来说,这样情况存在 stolen bytes。下面我们来定位 stolen bytes。

在定位 stolen bytes 之前,先给大家介绍一下如何利用 HBP.txt 这个脚本来设置硬件断点,这个方法我们在脚本的编写那一章中介绍过了,大家应该还记得吧!通过脚本来设置硬件断点可以绕过很多壳对于硬件断点的检测。假如我们现在想通过设置硬件断点让程序断在 OEP 处的话,我们会发现程序并不会断下来,我们可以通过 HPB.txt 这个脚本来解决这个问题。下面就来给大家详细讲解硬件断点为什么不触发的原因,以及 HPB.txt 脚本如何解决这个问题的。

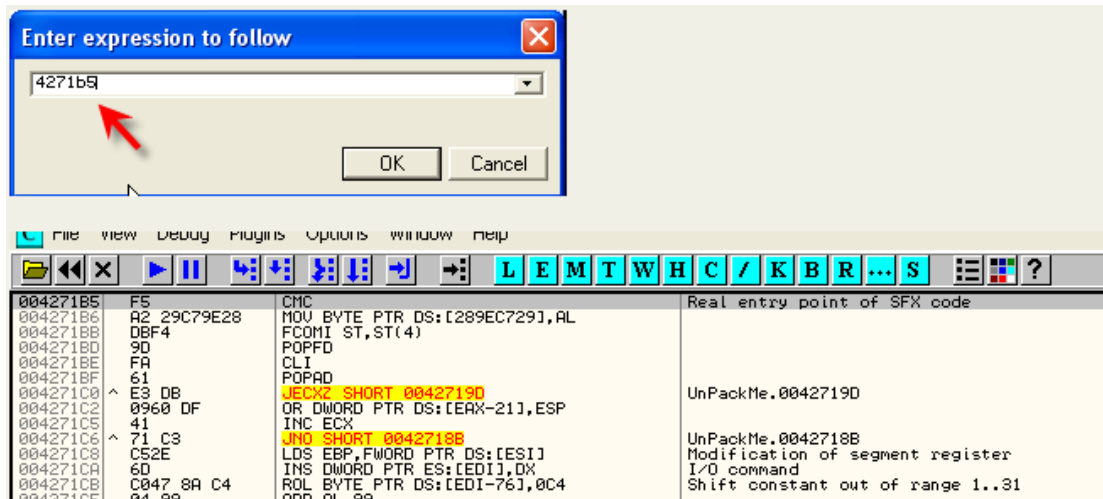
下面我们再次重启 OD,分别给 KiUserExceptionDispatcher 以及其内部调用的 ZwContinue 处设置断点。



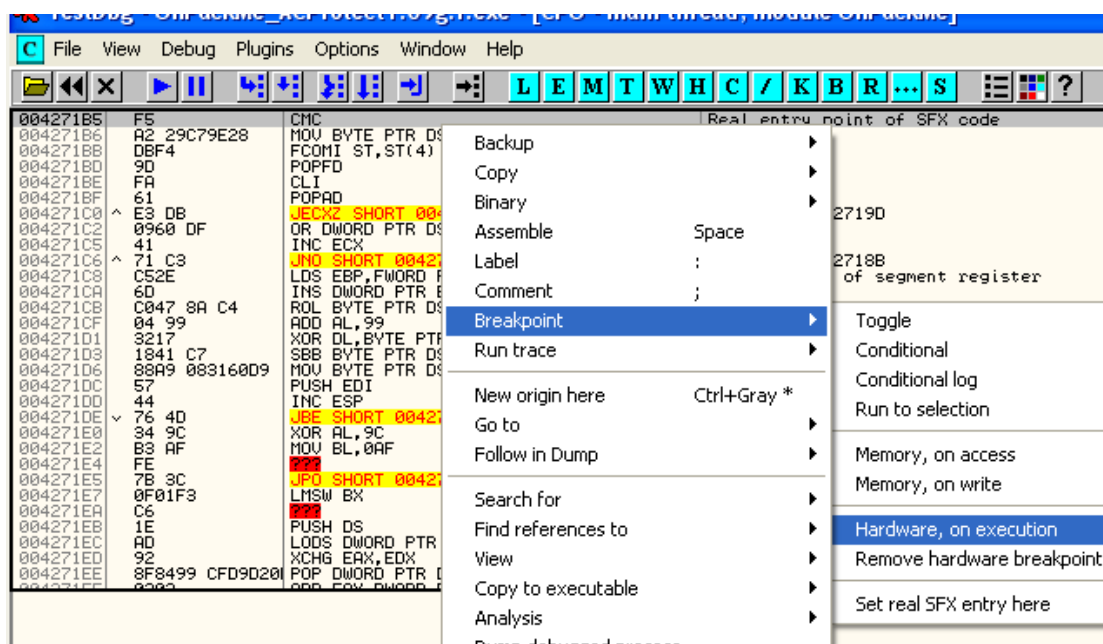
下面我们来给 OEP 处设置一个硬件执行断点。



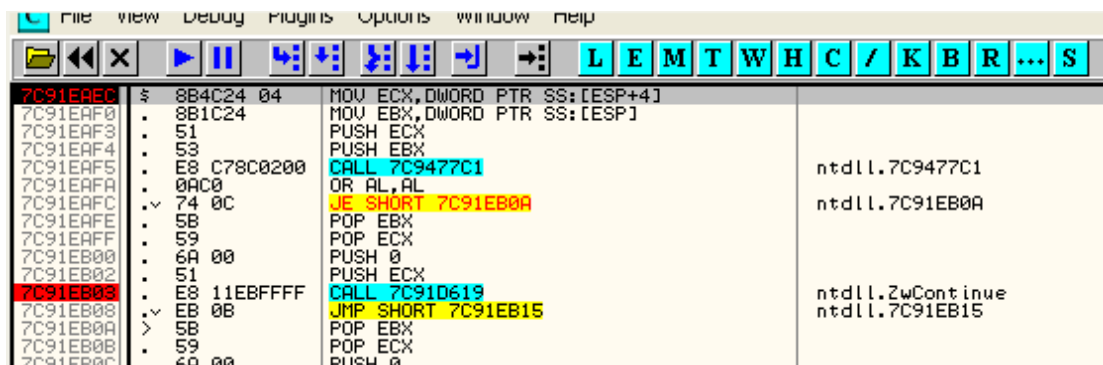
此时在壳的入口处,单击鼠标右键选择-Goto-Expression,输入地址 4271B5。



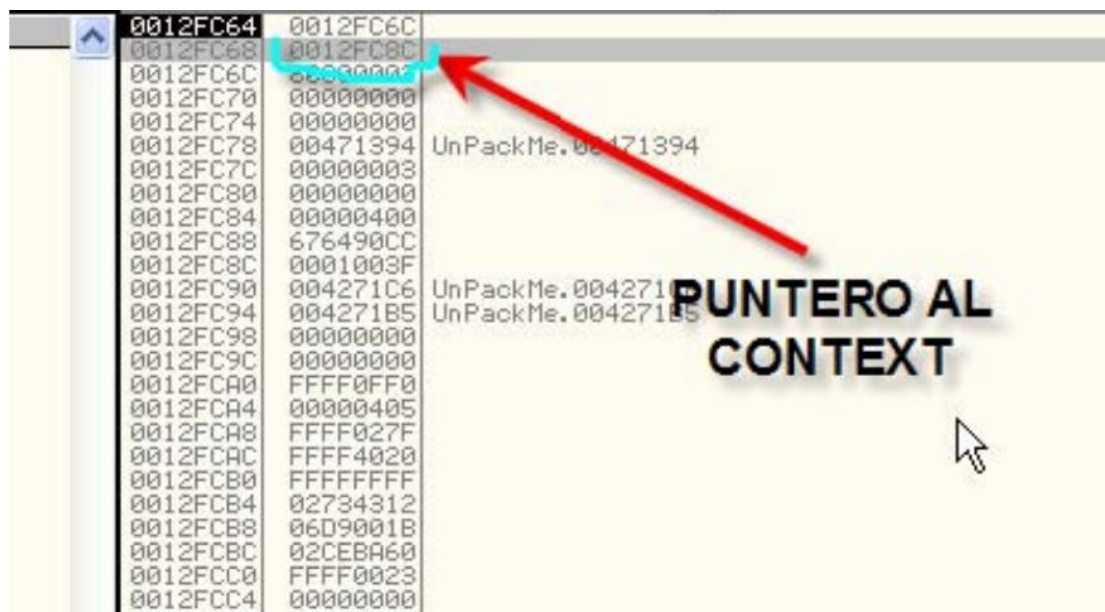
在这里我们设置一个硬件执行断点。



单击鼠标右键选择 Breakpoint-Hardware,on execution,接着运行起来。



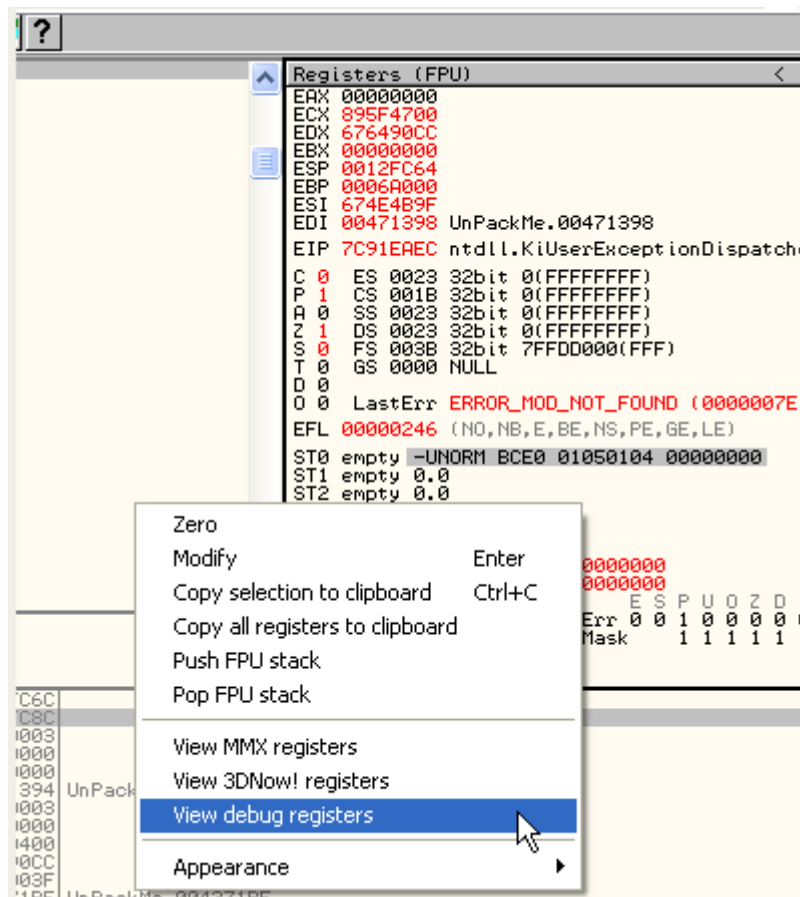
断了下来,我们来看下堆栈情况。



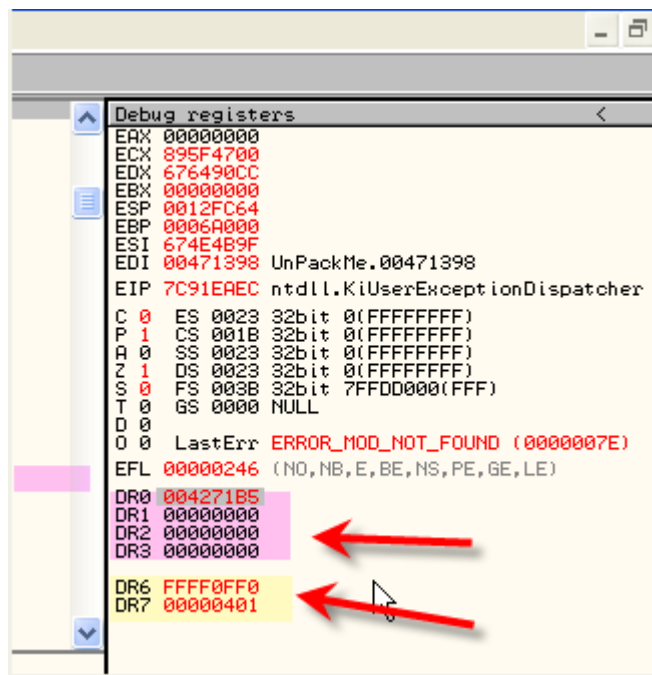
此时断在了 KiUserExceptionDispatcher 的入口处,堆栈中偏移 4 的单元中存放了 context 的指针,下面我们到数据窗口中定位该 context。

ntdll.KiUserApcDispatcher+2C									
Address	Hex	dump	ASCII						
0012FC8C	3F 00 01 00	B5 71 42 00	?.0.ÁqB.						
0012FC94	00 00 00 00	00 00 00 00	.....						
0012FC9C	00 00 00 00	F0 0F FF FF	...-*						
0012FCA4	01 04 00 00	7F 02 FF FF	0+.00						
0012FCAC	20 40 FF FF	FF FF FF FF	@						
0012FCB4	12 43 73 02	18 00 09 06	+Cs0+.'+						
0012FCBC	60 BA CE 02	23 00 FF FF	'llf0#.						
0012FCC4	00 00 00 00	04 01 05 01	...0+0						
0012FCCC	E0 BC 00 00	00 00 00 00	0².....						
0012FCD4	00 00 00 00	00 00 00 00	.....						
0012FCD4	00 00 00 00	00 00 00 00	.....						
0012FCE4	00 00 00 00	00 00 00 00	.....						
0012FCEC	00 00 00 00	00 00 00 00	.....						
0012FCF4	00 00 00 00	00 00 00 00	.....						
0012FCFC	00 00 00 00	00 00 00 00	.....						
0012FD04	00 00 00 00	FF 3F 00 00	...Ç ?.						
0012FD0C	00 00 00 00	00 00 FF 3F	....Ç ?						
0012FD14	00 00 00 00	00 00 00 00	.....						
0012FD1C	38 00 00 00	23 00 00 00	;...#...						
0012FD24	23 00 00 00	98 13 47 00	#...ÿ!!G.						
0012FD2C	9F 48 4E 67	00 00 00 00	fKNg....						
0012FD34	CC 90 64 67	00 47 5F 89	lfedg.G_ë						
0012FD3C	00 00 00 00	00 A0 06 00	....â+.						
0012FD44	94 13 47 00	18 00 00 00	ð!!G.+...						
0012FD4C	46 02 00 00	58 FF 12 00	F0...X+.						
0012FD54	23 00 00 00	7F 02 20 40	#...00 @						
0012FD5C	00 00 00 00	00 00 00 00	.....						
0012FD64	00 00 00 00	00 00 00 00	.....						
0012FD6C	00 00 FF FF	80 1F 00 00	.. Ç▼..						
0012FD74	00 00 00 00	00 00 00 00	.....						
0012FD7C	04 01 05 01	E0 BC 00 00	0+000²..						
0012FD84	00 00 00 00	00 00 00 00	.....						
0012FD8C	00 00 00 00	00 00 00 00	.....						

这里我暂时不解释 CONTEXT 结构体各个字段的含义,等后面用到的时候再解释。



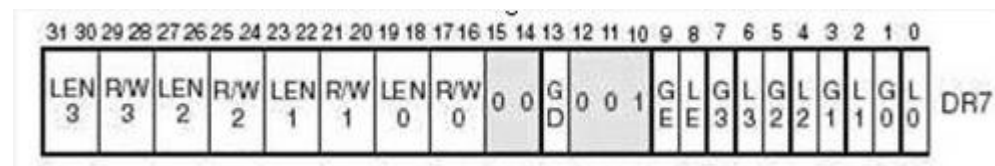
在寄存器窗口中,单击鼠标右键选择 View debug registers,切换到调试寄存器显示模式,如果没有该选项的话,说明当前已经是调试寄存器显示模式了。



这些就是调试寄存器组,Dr0 ~ Dr7。Dr0,Dr1,Dr2,Dr3 是用于设置硬件断点的,由于只有 4 个硬件断点寄存器,所以同时最多只能设置 4 个硬件断点。产生的异常是 STATUS\_SINGLE\_STEP(单步异常)。Dr4,Dr5 是系统保留的。Dr7 是一些控制位,用于控制断点的方式,Dr6 是用于显示哪个硬件调试寄存器引发的断点,如果是 Dr0 ~ Dr3 的话,相应位会被置 1。即如果是 Dr0 引发的断点,则 Dr6



调试控制寄存器 Dr7 比较重要,其 32 位结构如下:



LE 和 GE: P6 family 和之后的 IA32 处理器都不支持这两位。当设置时, 使得处理器会检测触发数据断点的精确的指令。当其中一个被设置的时候, 处理器会放慢执行速度, 这样当命令执行的时候可以通知这些数据断点。建议在设置数据断点时需要设置其中一个。切换任务时 LE 会被清除而 GE 不会被清除。为了兼容性, Intel 建议使用精确断点时把 LE 和 GE 都设置为 1。

- (1) 00 1 字节
- (2) 01 2 字节
- (3) 10 保留
- (4) 11 4 字节

- (1) 00 只执行
- (2) 01 写入数据断点
- (3) 10 I/O 端口断点 (只用于 pentium+, 需设置 CR4 的 DE 位, DE 是 CR4 的第 3 位)
- (4) 11 读或写数据断点

此时 Dr0 为 4271B5, 表示 4271B5 地址处被设置了硬件断点。现在我们来看看 CONTEXT 结构。

[illegible]

这里我们可以看到 context 结构中 Dr0~Dr3 寄存器的内容。黄色标注的 4271B5 是我们设置了硬件断点的地址。其他三个粉红色标注的位零,因为我们只设置了一个硬件断点,所以它们是空的。

当异常处理完毕以后,context 中的值将被清零,我们一起来看看,直接运行起来,断在第二个断点处。

ntdll.KiUserApcDispatcher+43			
Address	Hex	dump	ASCII
0012FC8C	3F 00 01 00	00 00 00 00	? . 0 . . . . .
0012FC94	00 00 00 00	00 00 00 00	. . . . .
0012FC9C	00 00 00 00	F0 0F FF FF	. . . . *
0012FCA4	01 04 00 00	7F 02 FF FF	0 . . . 0
0012FCAC	20 40 FF FF	FF FF FF FF	@ . . . .
0012FCB4	12 43 73 02	1B 00 D9 06	+ C s @ + . ' +
0012FCBC	60 BA CE 02	23 00 FF FF	'    行 @ # .
0012FCC4	00 00 00 00	04 01 05 01	. . . . 0 0 0 0
0012FCCC	E0 BC 00 00	00 00 00 00	0 . . . . .
0012FCD4	00 00 00 00	00 00 00 00	. . . . .
0012FCDC	00 00 00 00	00 00 00 00	. . . . .
0012FCE4	00 00 00 00	00 00 00 00	. . . . .
0012FCEC	00 00 00 00	00 00 00 00	. . . . .
0012FCF4	00 00 00 00	00 00 00 00	. . . . .
0012FCFC	00 00 00 00	00 00 00 00	. . . . .
0012FD04	00 00 00 80	FF 3F 00 00	. . . . ? .
0012FD0C	00 00 00 00	00 80 FF 3F	. . . . ? ?
0012FD14	00 00 00 00	00 00 00 00	. . . . .
0012FD1C	3B 00 00 00	23 00 00 00	. . . . # .
0012FD24	23 00 00 00	98 13 47 00	# . . . 0 ! G .
0012FD2C	9F 48 4E 67	00 00 00 00	f K N g . . . .
0012FD34	CC 90 64 67	00 47 5F 89	l f e d g . G _ e
0012FD3C	00 00 00 00	00 A0 06 00	. . . . 0 0 0 0
0012FD44	95 13 47 00	1B 00 00 00	0 ! G . + . . .
0012FD4C	46 02 00 00	58 FF 12 00	F @ . . X + . .
0012FD54	23 00 00 00	7F 02 20 40	# . . . 0 0 @
0012FD5C	00 00 00 00	00 00 00 00	. . . . .
0012FD64	00 00 00 00	00 00 00 00	. . . . .
0012FD6C	00 00 00 00	00 00 00 00	. . . . .

我们可以看到经过了异常处理以后,Dr0 被清零了,我们可以编写脚本来恢复它。这里我们有两种方案,第一种方案:在断在 KiUserExceptionDispatcher 入口处时将 context 中调试寄存器的值保存一份。当断在下面的 ZwContinue 的调用处时将调试寄存器的值恢复。第二种方案:在 context 中定位到程序的返回地址,在返回地址处设置一个断点,当断在返回地址处时恢复硬件断点,相当于重新设置了一次硬件断点。

好,这里我们采用第二种方案,但是现在的问题是程序的返回地址保存在 context 结构中的哪里呢?

ntdll.KiUserApcDispatcher+43			
Address	Hex	dump	ASCII
0012FC8C	3F 00 01 00	00 00 00 00	? . 0 . . . .
0012FC94	00 00 00 00	00 00 00 00	. . . . .
0012FC9C	00 00 00 00	F0 0F FF FF	. . . . *
0012FCA4	01 04 00 00	7F 02 FF FF	0 . . . 0
0012FCAC	20 40 FF FF	FF FF FF FF	@ . . . .
0012FCB4	12 43 73 02	1B 00 D9 06	+ C s @ + . ' +
0012FCBC	60 BA CE 02	23 00 FF FF	'    行 @ # .
0012FCC4	00 00 00 00	04 01 05 01	. . . . 0 0 0 0
0012FCCC	E0 BC 00 00	00 00 00 00	0 . . . . .
0012FCD4	00 00 00 00	00 00 00 00	. . . . .
0012FCDC	00 00 00 00	00 00 00 00	. . . . .
0012FCE4	00 00 00 00	00 00 00 00	. . . . .
0012FCEC	00 00 00 00	00 00 00 00	. . . . .
0012FCF4	00 00 00 00	00 00 00 00	. . . . .
0012FCFC	00 00 00 00	00 00 00 00	. . . . .
0012FD04	00 00 00 80	FF 3F 00 00	. . . . ? .
0012FD0C	00 00 00 00	00 80 FF 3F	. . . . ? ?
0012FD14	00 00 00 00	00 00 00 00	. . . . .
0012FD1C	3B 00 00 00	23 00 00 00	. . . . # .
0012FD24	23 00 00 00	98 13 47 00	# . . . 0 ! G .
0012FD2C	9F 48 4E 67	00 00 00 00	f K N g . . . .
0012FD34	CC 90 64 67	00 47 5F 89	l f e d g . G _ e
0012FD3C	00 00 00 00	00 A0 06 00	. . . . 0 0 0 0
0012FD44	95 13 47 00	1B 00 00 00	0 ! G . + . . .
0012FD4C	46 02 00 00	58 FF 12 00	F @ . . X + . .
0012FD54	23 00 00 00	7F 02 20 40	# . . . 0 0 @
0012FD5C	00 00 00 00	00 00 00 00	. . . . .
0012FD64	00 00 00 00	00 00 00 00	. . . . .
0012FD6C	00 00 FF FF	80 1F 00 00	. . . . C ? . .

Context 的起始地址偏移 0B8 地址处就保存了返回地址,我的机器上是:

$12FC8C + 0B8 = 12FD44$

0012FD84	00 00 00 00	00 00 00 00	. . . . .
0012FD8C	00 00 00 00	00 00 00 00	. . . . .
Command		12FC8c + 0b8	0012FD44

12FD44 内存单元中就保存着程序的返回地址,在脚本中可以对返回地址设置一个断点,当程序返回以后,再次设置硬件断点,我们来看一看脚本是怎么写的。



```

0000 var RetAddr
0001 Beginning:
0002
0003 bphws 4271b5,"x"
0004
0005 Work:
0006
0007 eob ToProcess
0008 run
0009
0010 ToProcess:
0011 log eip
0012 cmp eip,7c91eaec
0013 je ToClear
0014 cmp eip,7c91eb03
0015 je ToRecover
0016 cmp eip,RetAddr
0017 je ToReset
0018 jmp Final
0019
0020 ToClear:
0021 bphwc 4271b5
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

```

脚本的开头我们定义一个变量

```
0000 var RetAddr
```

var 命令是用来声明一个变量,这个变量 RetAddr 我们用来保存由异常返回的地址。

```

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

```

这就是恢复硬件断点的一部分,首先将 ESP 寄存器的值保存到 RetAddr 变量中。

```
0026 mov RetAddr,[RetAddr]
```

此时 RetAddr 保存 ESP 寄存器的值,接着再取其内容,也就是 context 结构的首地址,我这里 context 的首地址为 12FC8C,接着加上 0B8 等于 12FD44,12FD44 内存单元中就保存了异常的返回地址。

```
0028 mov RetAddr,[RetAddr]
```

最后再次取内容就得到了异常的返回地址,接着就对该返回地址设置断点。然后运行起来。

```

0010 ToProcess:
0011 log eip
0012 cmp eip,7c91eaec
0013 je ToClear
0014 cmp eip,7c91eb03
0015 je ToRecover
0016 cmp eip,RetAddr
0017 je ToReset
0018 jmp Final

```

这里是判断断下来的地址是 KiUserExceptionDispatcher 的入口处,还是 ZwContinue 的调用处,还是异常的返回地址。

```

0033 ToReset:
0034 bc RetAddr
0035 jmp Beginning

```

这里是删除异常返回地址处的断点,接着重新设置硬件断点。


好了,下面我们来看看脚本执行的效果,重启程序。

我们通过这个脚本来断到 OEP 处。

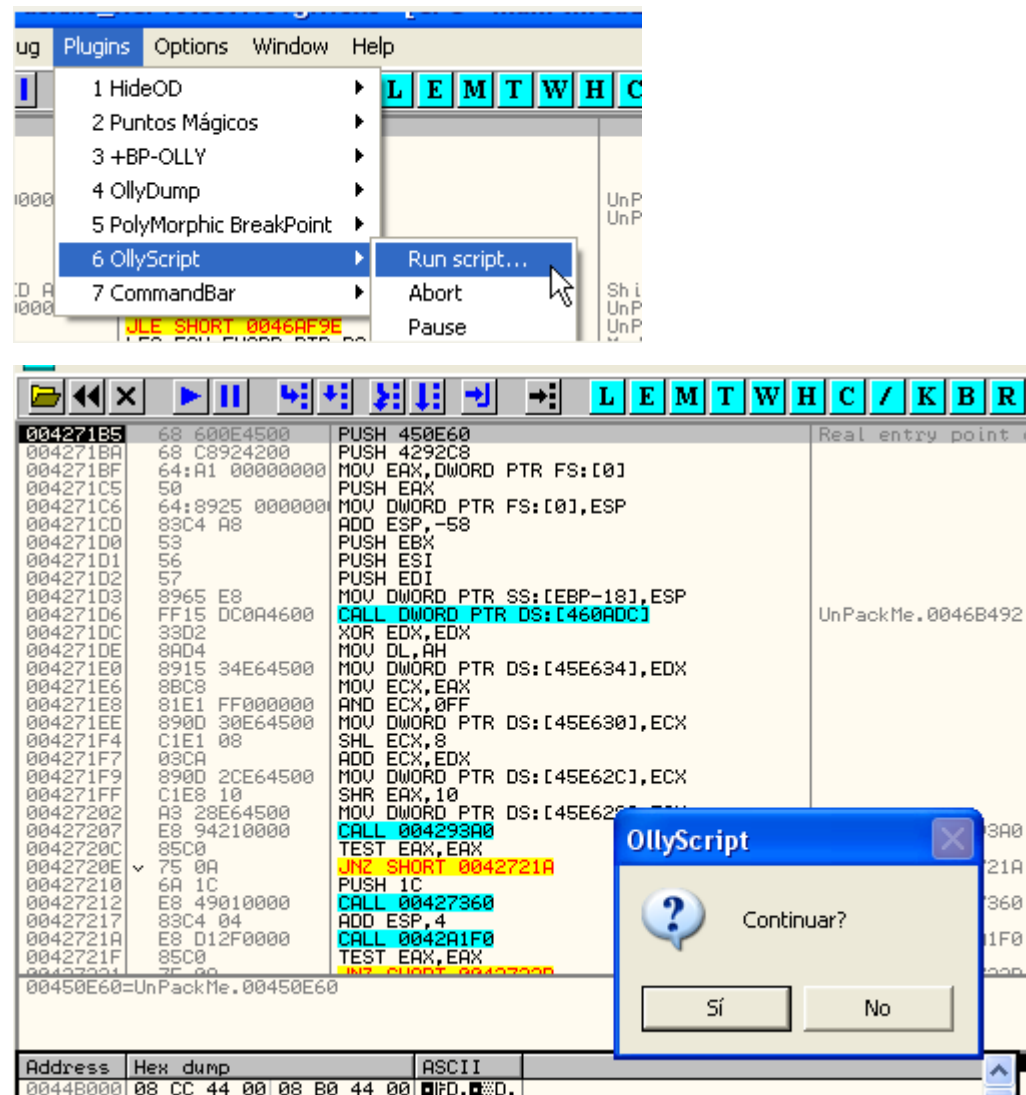
```

0000 var RetAddr
0001 Beginning:
0002
0003 bphws 4271b5,"x"
0004
0005 Work:
0006
0007 eob ToProcess
0008 run
0009
0010 ToProcess:
0011 log eip
0012 cmp eip,7c91eaec
0013 je ToClear
0014 cmp eip,7c91eb03
0015 je ToRecover
0016 cmp eip,RetAddr
0017 je ToReset
0018 jmp Final
0019
0020 ToClear:
0021 bphwc 4271b5
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

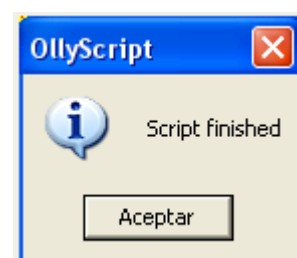
```



我们现在来运行脚本,不要忘了设置 KiUserExceptionDispatcher 入口处以及下面 ZwContinue 调用处的断点,还要记得将忽略异常的选项都勾选上。



这里我们可以看到断在了硬件断点处,我们选择 NO,不继续。



好,我们现在断在了 OEP 处。

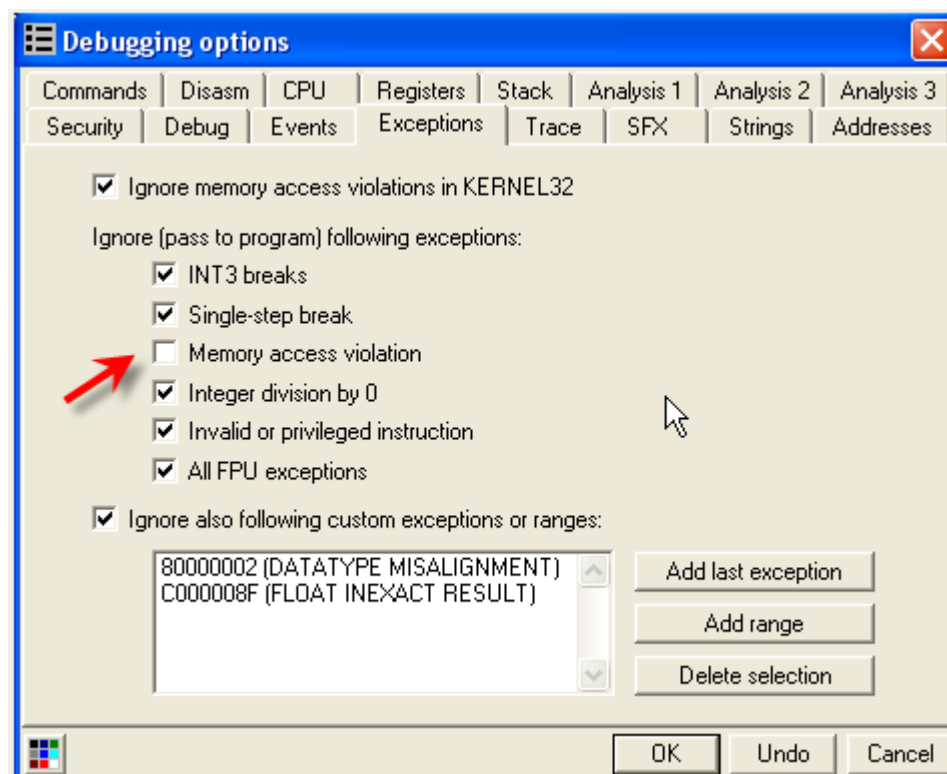
有些壳会通过一些方法来清除硬件断点,现在我们可以很完美的给 stolen bytes 设置硬件断点了。

首先我们来看看有没有异常,我们从最后一次异常处开始跟踪。

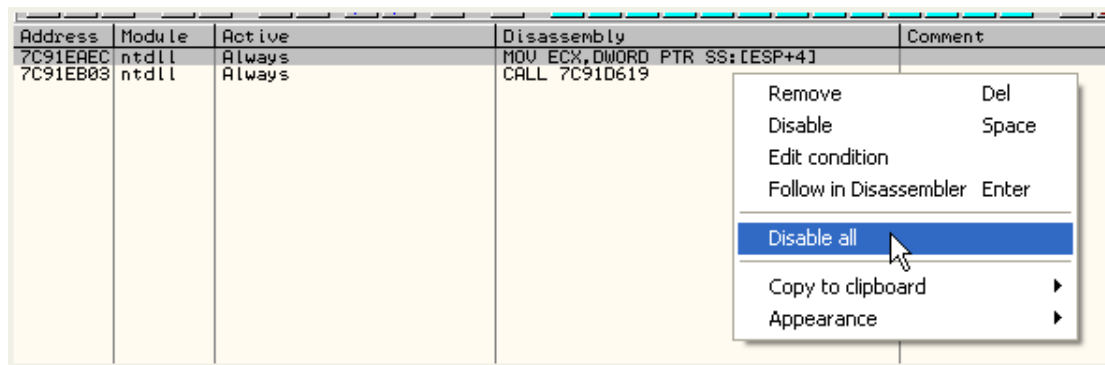
```
76360000 Module C:\WINDOWS\system32\comctl32.dll
77F40000 Module C:\WINDOWS\system32\SHLWAPI.dll
77BE0000 Module C:\WINDOWS\system32\msvcrt.dll
58C30000 Module C:\WINDOWS\system32\COMCTL32.dll
7C9D0000 Module C:\WINDOWS\system32\SHELL32.dll
773A0000 Module C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.26
72F80000 Module C:\WINDOWS\system32\WINSPOOL.DRV
74CC0000 Module C:\WINDOWS\system32\oledlg.dll
774B0000 Module C:\WINDOWS\system32\ole32.dll
00471394 INT3 command at UnPackMe.00471394
7C91E9EC Breakpoint at ntdll.KiUserExceptionDispatcher (KiUserApcDispatcher+2C)
eip = 7C91E9EC ; ntdll.KiUserExceptionDispatcher
7C91EB03 Breakpoint at ntdll.7C91EB03 (KiUserApcDispatcher+43)
eip = 7C91EB03
aux = 0012FC64
aux = 0012FC8C
aux = 0012FD44
aux = 00471395
00471395 Breakpoint at UnPackMe.00471395
eip = 00471395
0047108E Access violation when reading [FFFFFFFF]
7C91E9EC Breakpoint at ntdll.KiUserExceptionDispatcher (KiUserApcDispatcher+2C)
eip = 7C91E9EC ; ntdll.KiUserExceptionDispatcher
7C91EB03 Breakpoint at ntdll.7C91EB03 (KiUserApcDispatcher+43)
eip = 7C91EB03
aux = 0012FC68
aux = 0012FC8C
aux = 0012FD44
aux = 00471090
00471090 Breakpoint at UnPackMe.00471090
eip = 00471090
004271B5 Hardware breakpoint 1 at UnPackMe.004271B5
eip = 004271B5

Command
Hardware breakpoint 1 at UnPackMe.004271B5
```

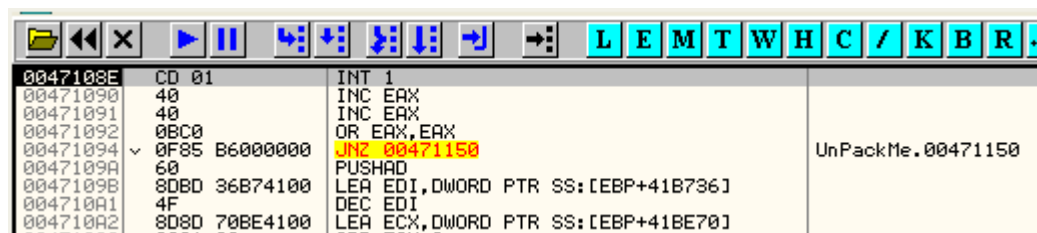
在日志窗口中我们可以看到在到达 OEP 之前产生了一次异常,我们可以看到类型是 Access violation when reading。我们来清空掉相应的异常忽略选项。



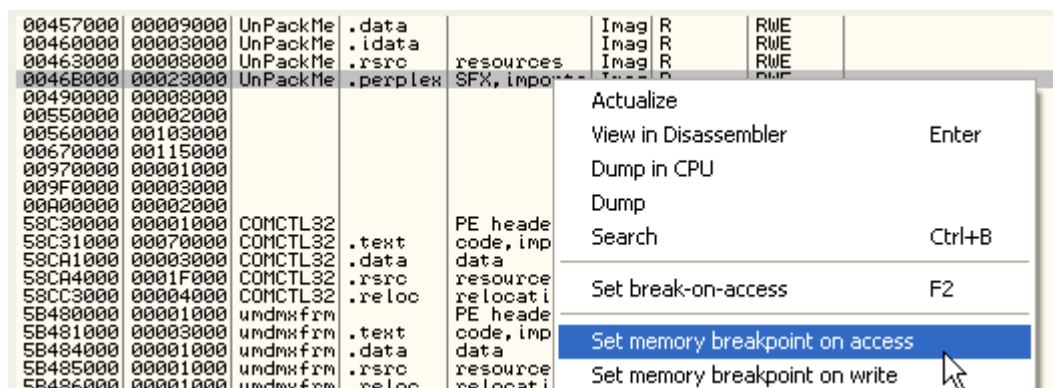
禁用掉脚本中用到的断点。



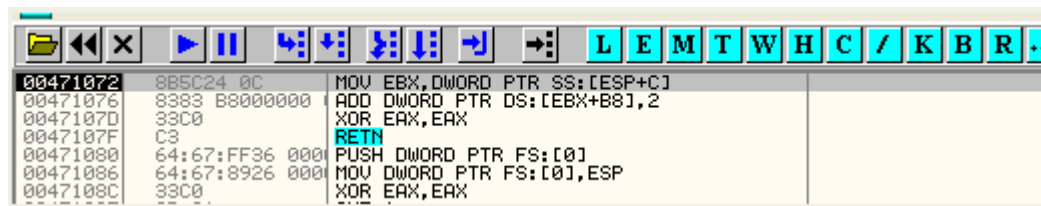
现在我们重新启动,然后运行起来。



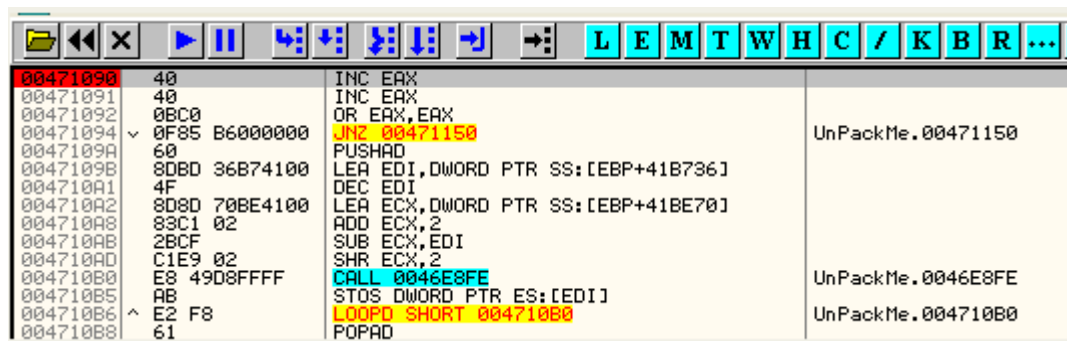
断在了最后一次异常处,给当前区段设置内存访问断点,让其断在异常处理程序中。



按 Shift + F9 忽略异常运行起来。

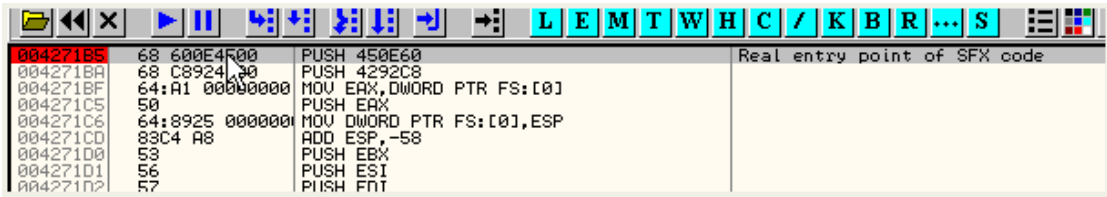


按 F7 键单步几下到达异常处理函数的 RET 处。接着运行起来,我们返回到了这里



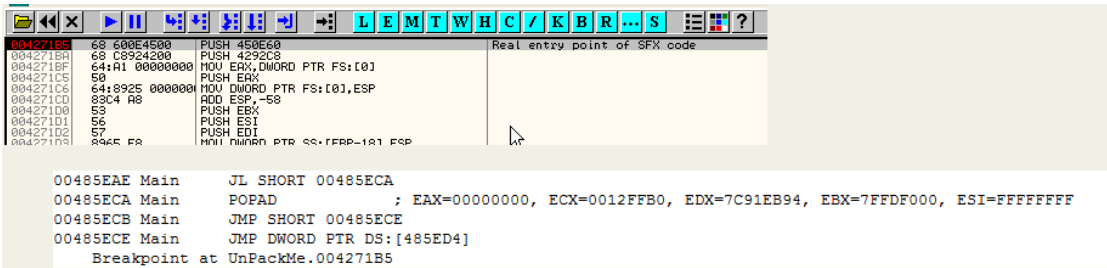
00471090 40 INC EAX

下面我们可以对 OEP 处设置断点,因为此时壳已经解密完毕了。

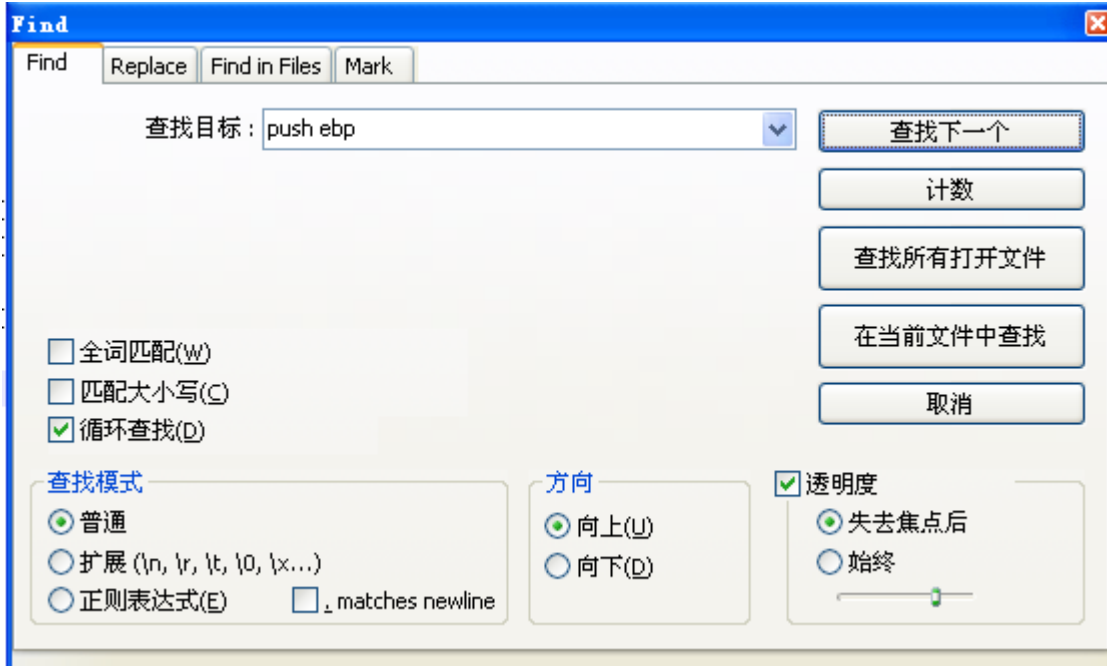


现在我们可以从 471090 处开始跟踪了,不一会儿跟踪完毕了,我们现在到达了 OEP 处。

我们来看一看跟踪的指令列表。



这里跟踪的指令的最后几行,这里我们可以搜索 PUSH EBP,一般是从上往下搜索,但这里前面位于系统 DLL 中的 PUSH EBP 过多,所以这里我们从下往上搜索。(PS:由于这里记录的 TXT 又几十 M,所以这里我使用的是 EditPlus)





```

00485AF3 Main    PUSH EBP
00485AF4 Main    MOV EBP,ESP      ; EBP=0012FFC0
00485AF6 Main    PUSH -1
00485AF8 Main    NOP
00485AF9 Main    PUSHAD
00485AFA Main    PUSHAD
00485AFB Main    CALL 00485B00
00485B00 Main    POP ESI          ; ESI=00485B00
00485B01 Main    SUB ESI,6        ; ESI=00485AFA
00485B04 Main    MOV ECX,35       ; ECX=00000035
00485B09 Main    SUB ESI,ECX      ; ESI=00485AC5
00485B0B Main    MOV EDX,E3D6D5FD ; EDX=E3D6D5FD
00485B10 Main    SHR ECX,2        ; ECX=0000000D
00485B13 Main    SUB ECX,2        ; ECX=0000000B
00485B16 Main    CMP ECX,0

```

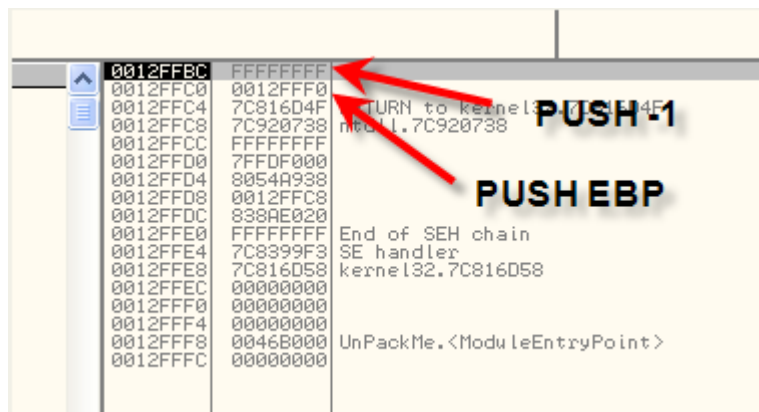
这里我们可以看到 stolen bytes,接着是 NOP 指令,然后使用 PUSHAD 指令保存寄存器环境,那么在跳往假的 OEP 之前会使用 POPAD 指令来恢复寄存器环境的。

```

00485ECA Main    POPAD           ; EAX=00000000, ECX=0012FFB0, EDX=7C91EB94, EBX=7FFDF000, ESI=FFFFFFFF
00485ECB Main    JMP SHORT 00485ECE
00485ECE Main    JMP DWORD PTR DS:[485ED4]
Breakpoint at UnPackMe.004271B5

```

这里我们可以看到 stolen bytes 的数值部分是被保存在栈中的。

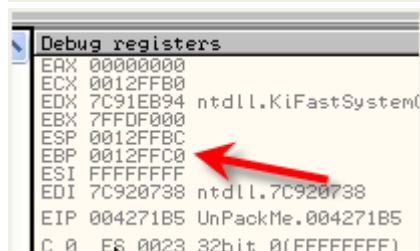


EBP 寄存器可能在其间会赋予一个随机值,但是随后通过 MOV EBP,ESP 指令又可以修正。

```

00485AF3 Main    PUSH EBP
00485AF4 Main    MOV EBP,ESP      ; EBP=0012FFC0
00485AF6 Main    PUSH -1

```



我们可以看到记录文件中显示 EBP 的值为 12FFC0,该值刚好与假 OEP 处的 EBP 的值相等。说明我们定位到的 stolen bytes 是正确的。

好了,这里我们就找到了 stolen bytes,关于前面的通过脚本设置硬件断点的技巧我们将在下一章节中使用。

