第五十七章-ExeCryptor v2.2.50.c/d/e/f/g 脱壳

UnPackMe C:

本章我们继续加强 ExeCryptor UnPackMe 的难度。UnPackMe C 与 UnPackMe B 的难度比较接近,只不过 UnPackMe C 在运行的时候会检测是否存在注册表以及文件监视工具,如果检测到了会将它们关闭。

由于这里我并没有开启注册表以及文件的监视工具,所以 UnPackMe C 与 UnPackMe B 的脱壳方法是一样的,我的机器上 MOV EAX,DWORD PTR SS:[EBP - C]这条指令的地址为 486DF7。

00486DF7 8B45 F4 MOV EAX,DWORD PTR SS:[EBP-C]

00486DFA E8 61670100 CALL 0049D560 ; UnPackMe.0049D560

00486DFF 5B POP EBX

00486E00 8B0424 MOV EAX,DWORD PTR SS:[ESP]

00486E03 52 PUSH EDX

我们将脚本中硬件执行断点的地址修改为 486DFA。

脚本如下:

```
var table
var content
    mov table,460818

start:
    cmp table,460F28
    ja final
    cmp [table],50000000
    ja ToSkip


    mov content,[table]
    cmp content,0
    je ToSkip
    log content
    log table

    mov eip,content
    bphws 486DFA,"x"
    mov [486DFA],0
    mov [486DFB],0
    cob ToRepair
```

run

ToRepair:

cmp eip,486DFA

jne ToSkip

log eax

mov [table],eax

run

ToSkip:

add table,4

jmp start

final:

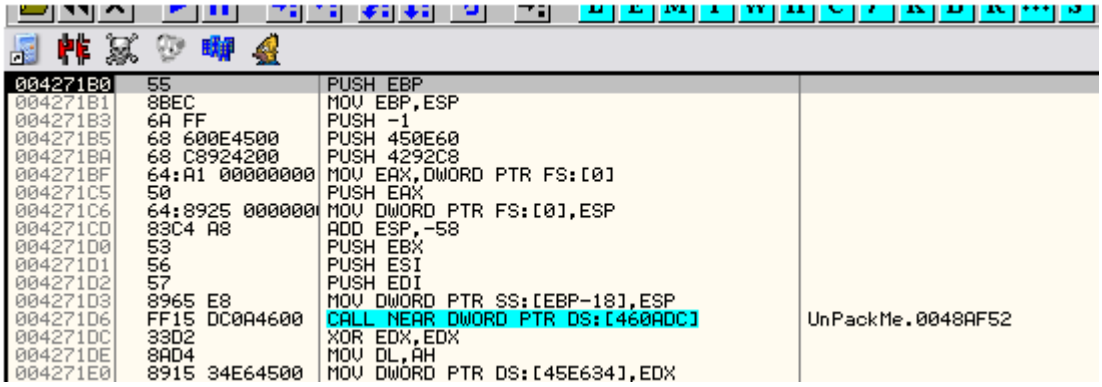
ret

到达 OEP 处以后别忘了删除掉 break-on-execute 断点(PS:使用 OllyBone 插件设置了 break-on-execute 断点的话,记得要删除,没有用到 OllyBone 插件的话,就不用管了),接着将监控线程挂起,然后在 ZwTerminateProcess 这个 API 函数的入口处设置一个硬件执行断点,接着执行该脚本修复 IAT,IAT 修复完毕以后就可以进行 dump 了,然后打开 IMP REC 修复 dump 文件,这样 UnPackMe C 就搞定了。

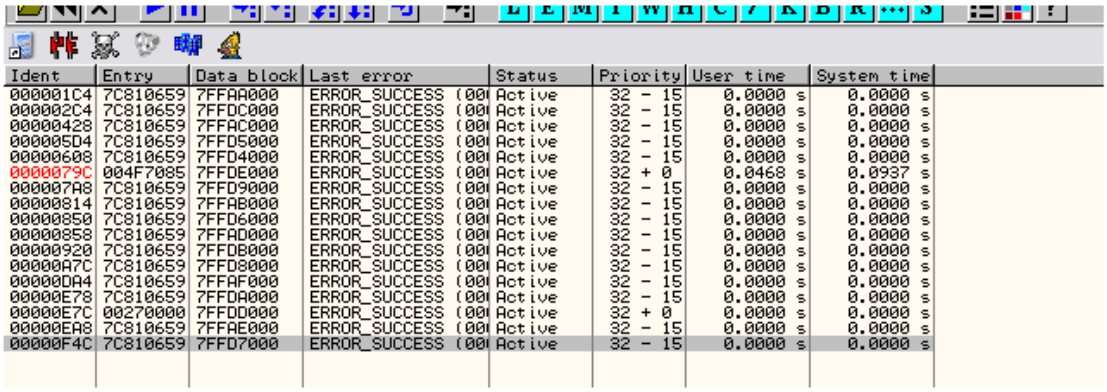
UnPackMe D:

接下来我们来看看 UnPackMe D,双击运行,看看等级 D 的保护措施:

我们可以看到调试消息这个选项开启了,也就是说会检测调试消息。不知道对我们有没有影响,我们用 OD 加载 UnPackMe D,还是跟之前一样断在了系统断点处,我们删除掉断点列表窗口中一次性断点,接着给代码段设置 break-on-execute 断点,运行起来就可以到达 OEP 处了。

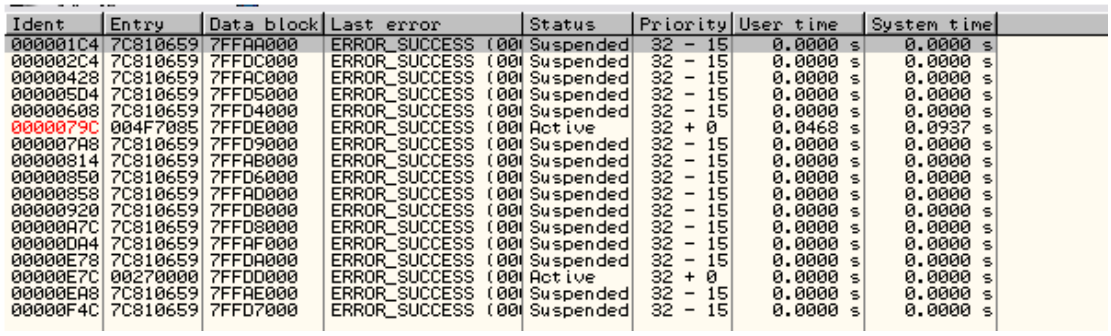


到目前为止,我们还没有看到 Debug Messages 这个选项开启了对我们有什么实质上的影响。下面我们来查看一下线程的情况。



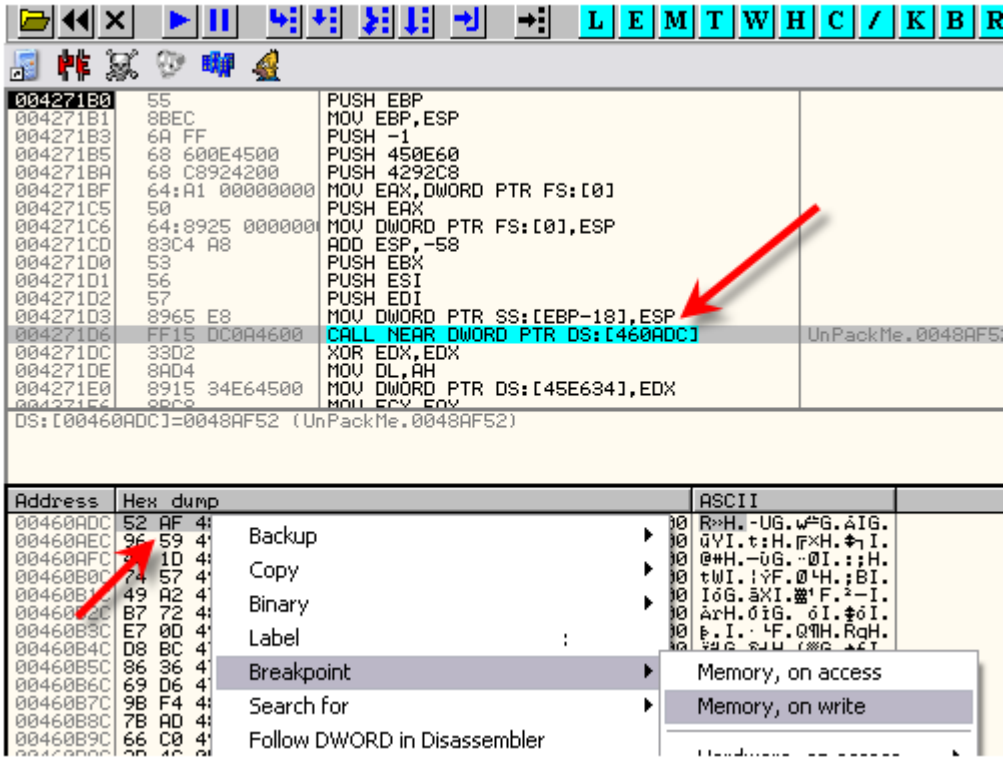
Ident	Entry	Data block	Last error	Status	Priority	User time	System time
000001C4	7C810659	7FFAA000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
000002C4	7C810659	7FFDC000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000428	7C810659	7FFAC000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
000005D4	7C810659	7FFD5000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000608	7C810659	7FFD4000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
0000079C	004F7085	7FFDE000	ERROR_SUCCESS (00	Active	32 + 0	0.0468 s	0.0937 s
000007A8	7C810659	7FFD9000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000814	7C810659	7FFAB000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000850	7C810659	7FFD6000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000858	7C810659	7FFAD000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000920	7C810659	7FFDB000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000A7C	7C810659	7FFD8000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000DA4	7C810659	7FFAF000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000E78	7C810659	7FFDA000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000E7C	00270000	7FFDD000	ERROR_SUCCESS (00	Active	32 + 0	0.0000 s	0.0000 s
00000EA8	7C810659	7FFAE000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s
00000F4C	7C810659	7FFD7000	ERROR_SUCCESS (00	Active	32 - 15	0.0000 s	0.0000 s

跟之前的一样,我们还是将除了主线程以及线程函数入口地址为 270000 的这两个线程以外的其他线程都挂起。

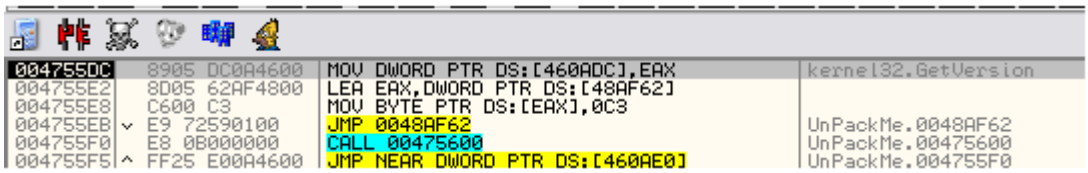


Ident	Entry	Data block	Last error	Status	Priority	User time	System time
000001C4	7C810659	7FFAA000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
000002C4	7C810659	7FFDC000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000428	7C810659	7FFAC000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
000005D4	7C810659	7FFD5000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000608	7C810659	7FFD4000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
0000079C	004F7085	7FFDE000	ERROR_SUCCESS (00	Active	32 + 0	0.0468 s	0.0937 s
000007A8	7C810659	7FFD9000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000814	7C810659	7FFAB000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000850	7C810659	7FFD6000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000858	7C810659	7FFAD000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000920	7C810659	7FFDB000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000A7C	7C810659	7FFD8000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000DA4	7C810659	7FFAF000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000E78	7C810659	7FFDA000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000E7C	00270000	7FFDD000	ERROR_SUCCESS (00	Active	32 + 0	0.0000 s	0.0000 s
00000EA8	7C810659	7FFAE000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s
00000F4C	7C810659	7FFD7000	ERROR_SUCCESS (00	Suspended	32 - 15	0.0000 s	0.0000 s

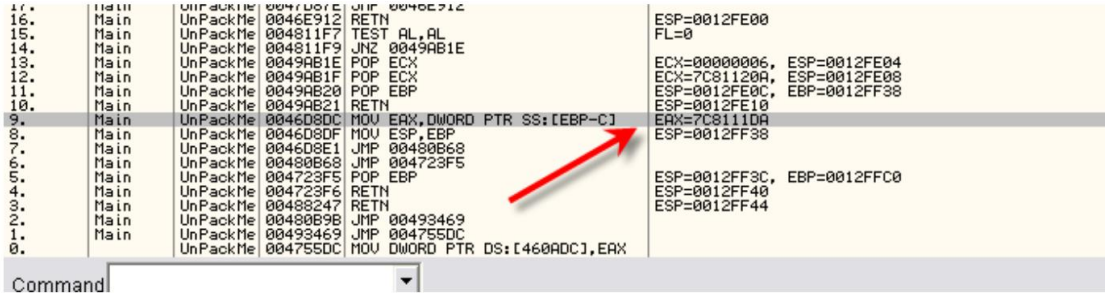
好,下面我们定位到 OEP 下方调用的第一个 API 函数指令处,对其 IAT 项设置内存写入断点,接着利用 OD 的 Trace into 进行自动跟踪。



自动跟踪需要一段时间。



好了,自动跟踪结束了,断在了这里,我们在跟踪日志中定位到 MOV EAX,DWORD PTR SS:[EBP -C]这条语句,我们可以看到它在这里。



也就是说跟 UnPackMe C 并没有什么区别。

0046D8DC 8B45 F4 MOV EAX,DWORD PTR SS:[EBP-C] ; kernel32.GetVersion

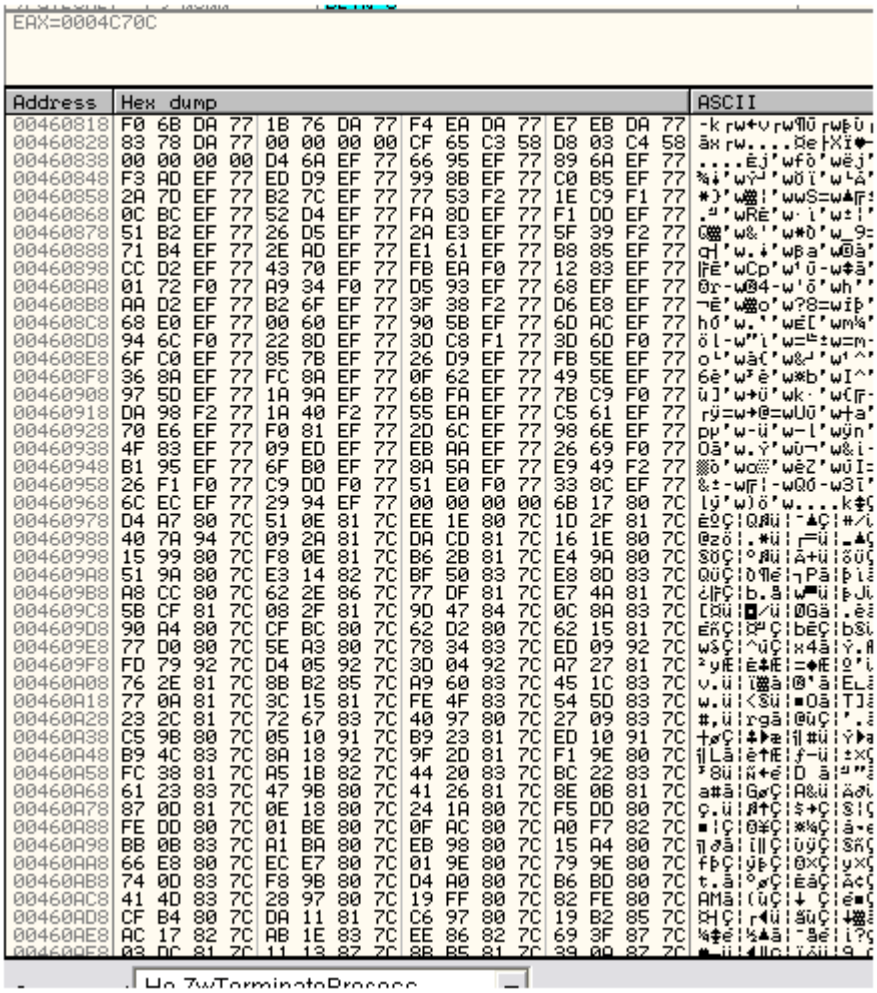
0046D8DF 8BE5 MOV ESP,EBP

我们只需要将硬件执行断点的地址修改为 46D8DF 即可,修改后的脚本如下:

```
var table
var content
    mov table,460818

start:
    cmp table,460F28
    ja final
    cmp [table],50000000
    ja ToSkip


    mov content,[table]
    cmp content,0
    je ToSkip
    log content
    log table

    mov eip,content
    bphws 46D8DF,"x"
    mov [46D8DF],0
    mov [46D8DF],0
    cob ToRepair
    run

ToRepair:
```

```
    cmp eip,46D8DF
    jne ToSkip
    log eax
    mov [table],eax
    run

ToSkip:
    add table,4
    jmp start

final:
    ret
```

好了,现在我们重启 OD,再次断到 OEP 处,接着删除掉 break-on-execute 断点,然后在 ZwTerminateProcess 这个 API 函数的入口处设置一个硬件执行断点,接着执行该脚本。



我们可以看到 IAT 项都被修复了,也就是说的确跟 UnPackMe C 没有区别。下面我们继续来看 UnPackMe E,看看有没有什么不同的地方。

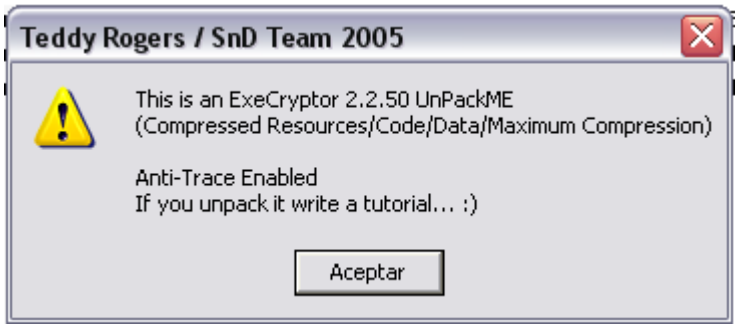
UnPackMe E:

我们运行 UnPackMe E,看看跟 UnPackMe D 相比有什么区别。我们可以看到 UnPackMe E 的 Active Watch(这个单词我们可以理解为动态监视)这个选项开启了。

我们会发现脱 UnPackMe E,UnPackMe F 与脱 UnPackMe D 的步骤基本上是一样的,这里我就不再赘述了。

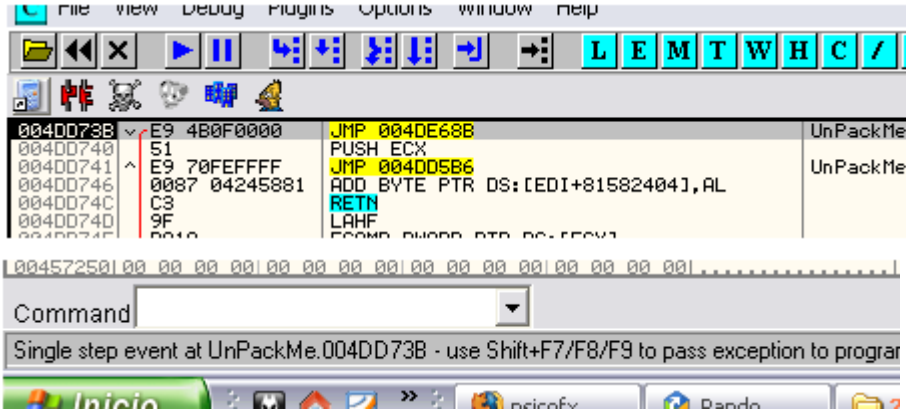
我们直接来看 UnPackMe G,UnPackMe G 的话就会玩一些新花样了。

UnPackMe G:

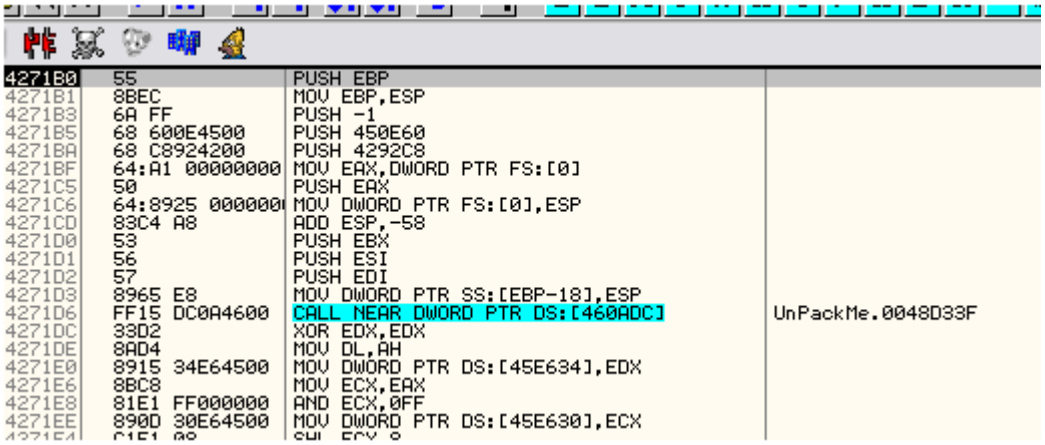


我们双击运行 UnPackMe G,可以看到这个等级反跟踪选项被开启了。我们需要利用 OD 的 trace into(自动跟踪)功能来定位 MOV EAX,DWORD PTR SS:[EBP-C]这条指令，不知道反跟踪这个选项会不会对此造成影响。

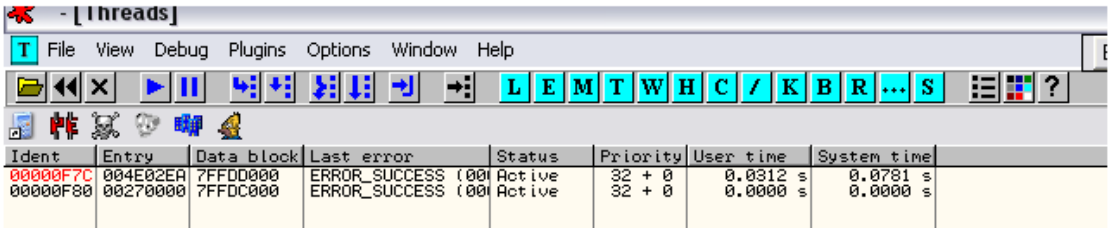
我们用 OD 加载 UnPackMe G,断在了系统断点处,接着我们删除掉断点窗口中的一次性断点,然后对代码段设置 break-on-execute 断点,接着运行起来,我们会发现在到达 OEP 之前,会断下来 5 到 6 次(由于单步异常导致的),这是反跟踪这个选项带来第一处影响。



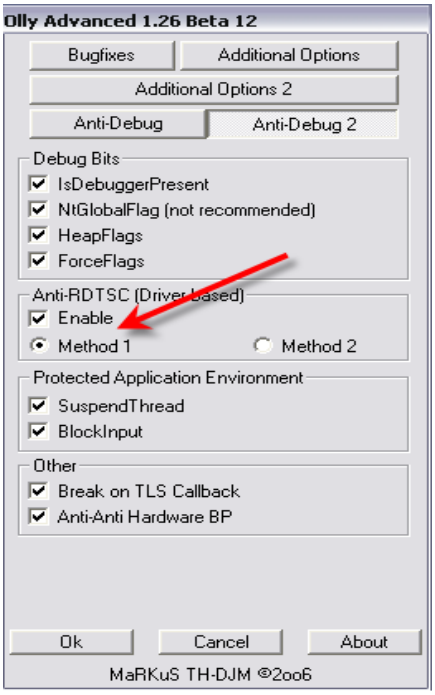
我们可以看到 OD 状态栏中提示:发生了单步异常,需要我们手动按 Shift + F7/F8/F9 忽略掉这个异常继续往下执行。这里我们不能够勾选忽略单步异常这个选项,如果我们勾选了这个选项的话,那么 OllyBone 插件就不起作用了,所以这里我们必须手动按 Shift+F9 忽略这些单步异常,大约按 5 到 6 次 Shift +F9 就可能断到 OEP 处了。



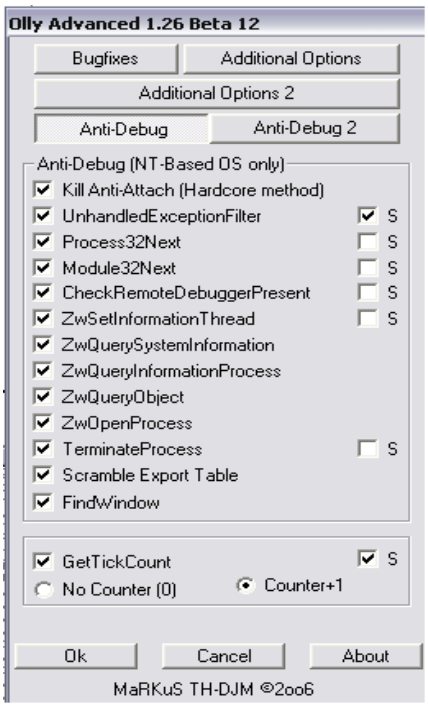
另外一个细节就是,我们会发现除了该程序运行必需的两个线程,并不存在其他的监控线程了,这也算是反跟踪选项带了的又一个不同之处吧。



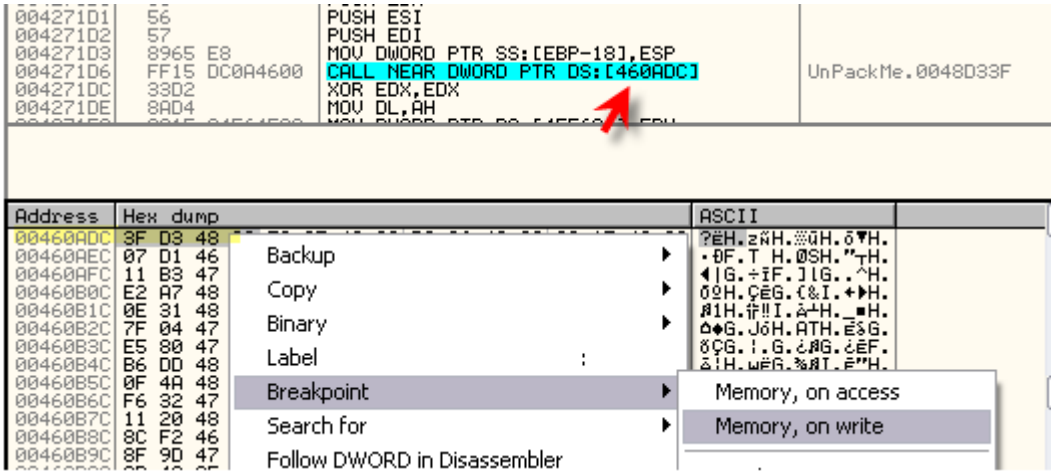
我们可以看到这里只有该程序运行必需的两个线程,如果还存在其他线程(这里我们姑且称这些线程为监控线程)的话,我们需要将这些监控线程挂起。这里由于反跟踪模式开启了,为了以防万一,我们将 OllyAdvanced 插件里面的 Anti-RDTSC 以及 GetTickCount 这两个选项勾选上。



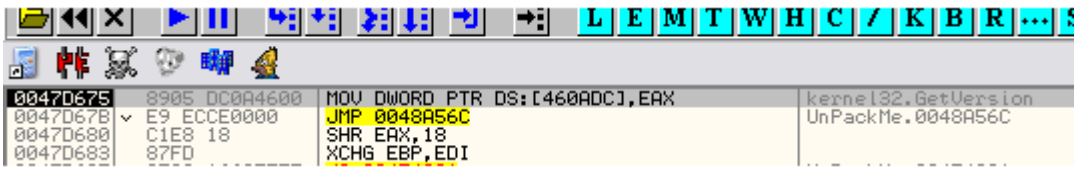
和



大家在使用 Anti-RDTSC 这个反反调试选项的时候,要稍微留意一下 OD 右下角显示的状态,如果显示为中断,过一会儿弹出一个消息框提示驱动无法正常启动的话,说明 Anti-RDTSC 这个选项附带的驱动程序无法正常工作,遇到情况的话,大家可以重启一下电脑试试看。

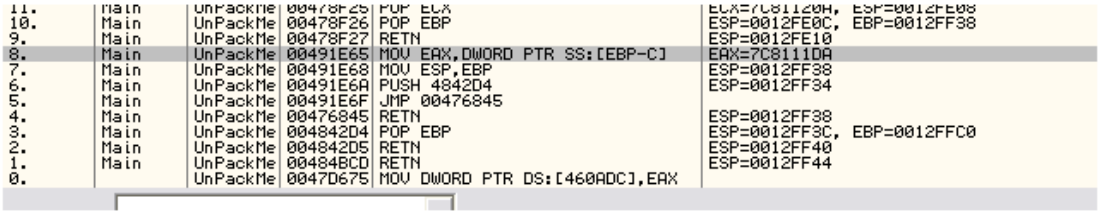


现在我们跟之前一样对 460ADC 这个 IAT 项设置内存写入断点,删除掉 break-on-execute 断点,接下来利用 OD 的 trace into 功能进行自动跟踪,看看会发生什么。



我们断到了这里。

下面我们在跟踪日志中定位 MOV EAX,DWORD PTR SS:[EBP-C]这条指令。

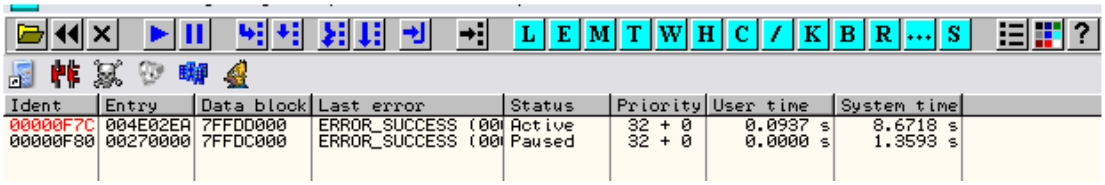


这里我们可以看到 MOV EAX,DWORD PTR SS:[EBP-C]这条指令的地址为 491E65。

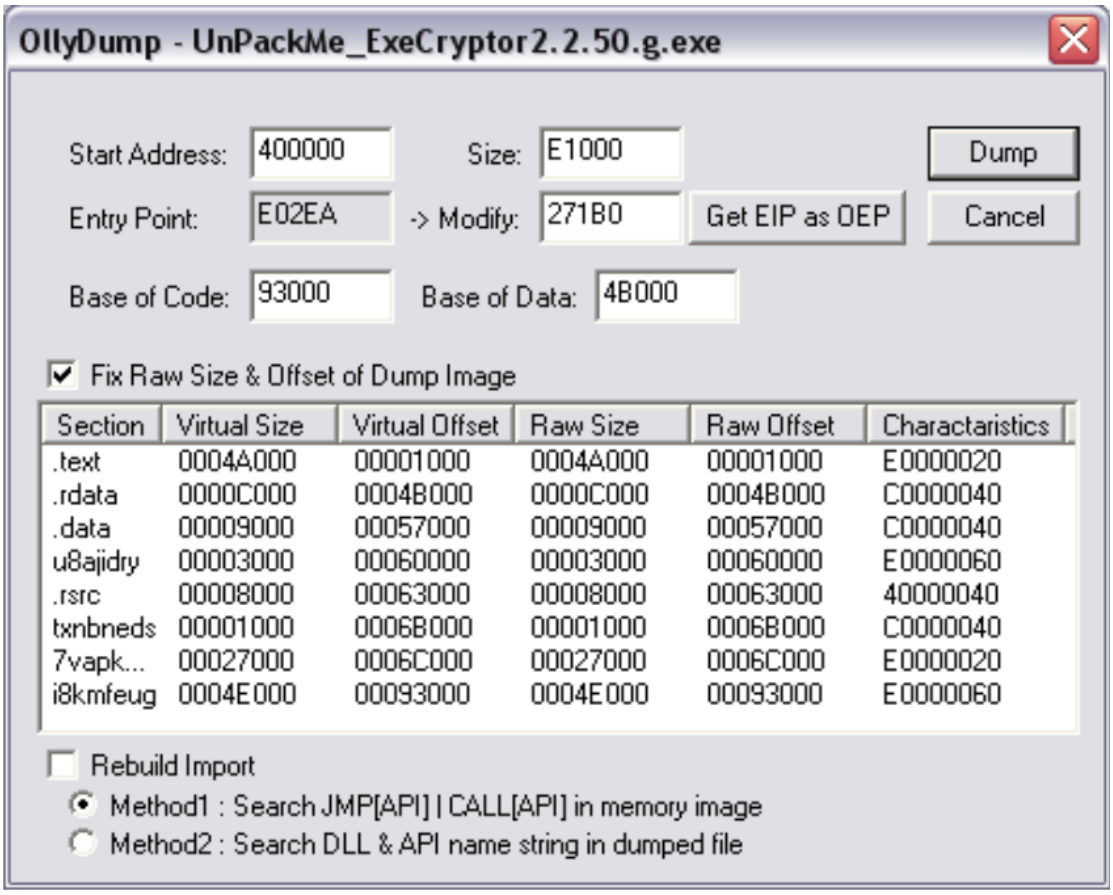
00491E65 8B45 F4 MOV EAX,DWORD PTR SS:[EBP-C] ; kernel32.GetVersion

00491E68 8BE5 MOV ESP,EBP

也就说到目前为止,trace into 并没有出现问题,有可能是我们勾选上了 OllyAdvanced 插件中的 Anti-RDTSC 以及 GetTickCount 这两个选项的缘故吧,我们还可以看到此时线程函数入口地址为 270000 的这个线程中止了。



好,我们需要的信息已经搜集完毕了,接下来我们先进行 dump,接着来修改脚本,然后执行该脚本修复 IAT。



这里我们将脚本中硬件执行断点的地址替换掉。

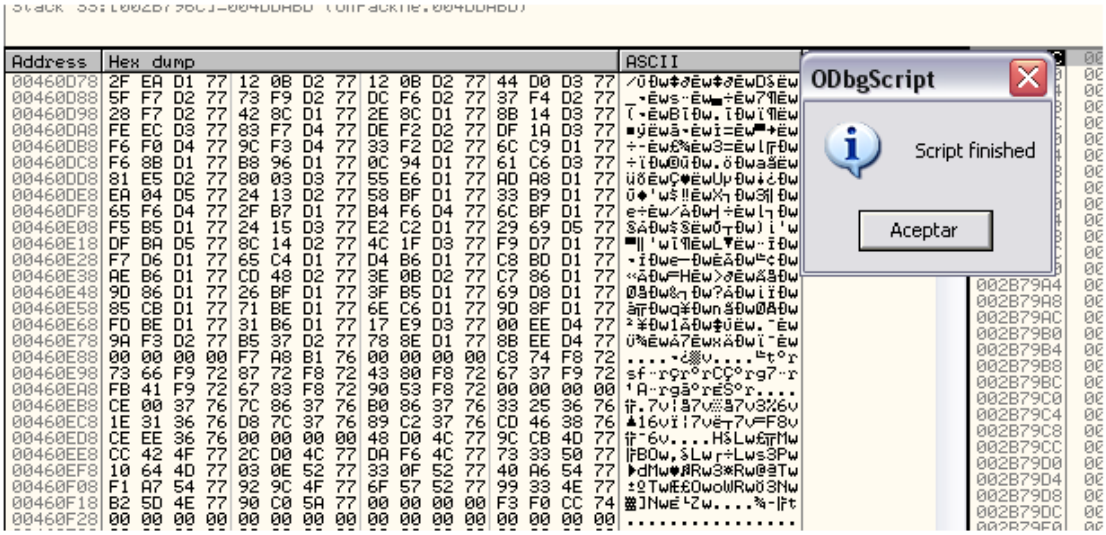
var table

var content

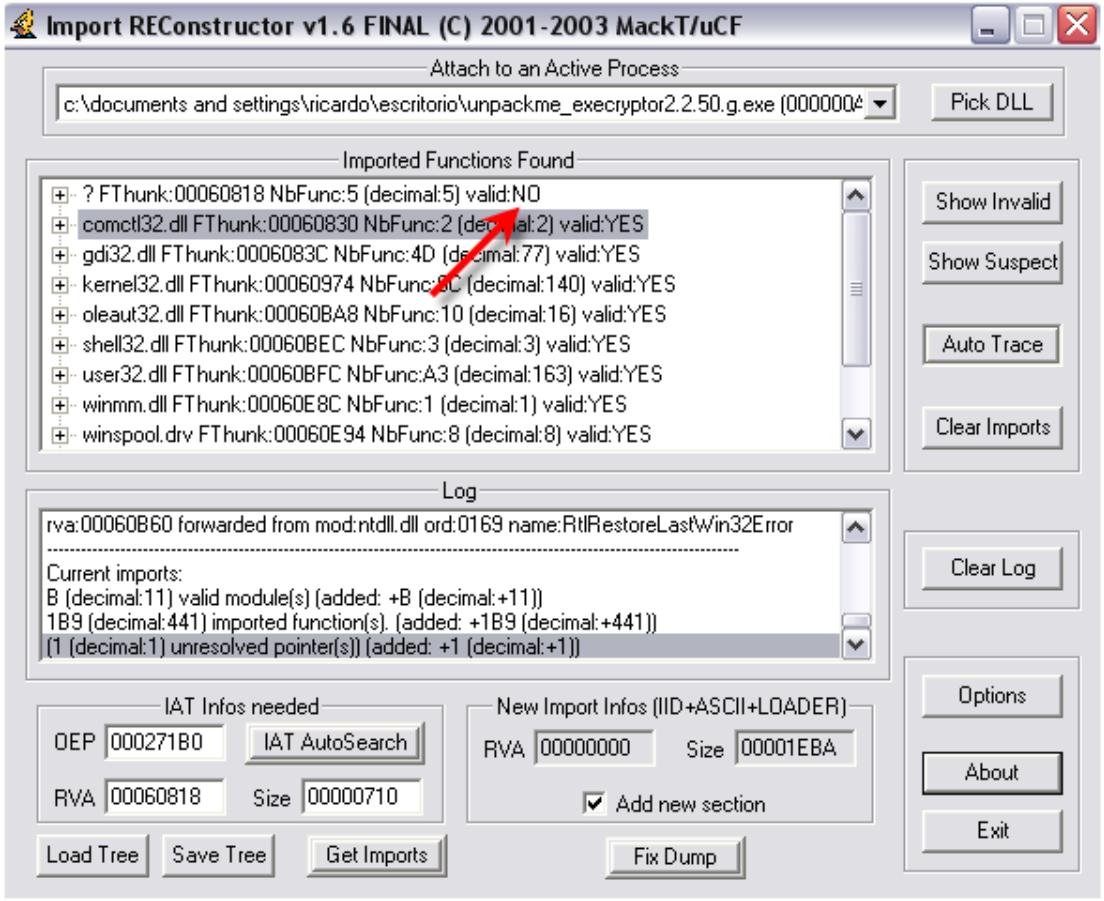
mov table,460818

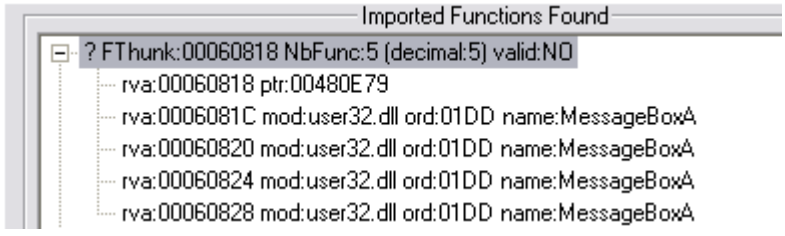
```
start:
    cmp table,460F28
    ja final
    cmp [table],50000000
    ja ToSkip


    mov content,[table]
    cmp content,0
    je ToSkip
    log content
    log table

    mov eip,content
    bphws 491E68,"x"
    mov [491E68],0
    mov [491E68],0
    cob ToRepair
    run

ToRepair:
    cmp eip,491E68
    jne ToSkip
    log eax
    mov [table],eax
    run

ToSkip:
    add table,4
    jmp start

final:
    ret
```

好,现在我们在 ZwTerminateProcess 这个 API 函数的入口地址处设置一个硬件执行断点,接着执行该脚本,我们可以看到 IAT 项都被修复了。



这里的不同之处在于,中途并没有跳转到 ZwTerminateProcess 处,但是 IAT 项也被修复了,好,下面我们打开 IMP REC。





我们可以看到 IMP REC 中显示有无效的项,我们单击该项左边的加号将其展开,可以看到这些项修复后的值是错误的。

mov table,460818

start:

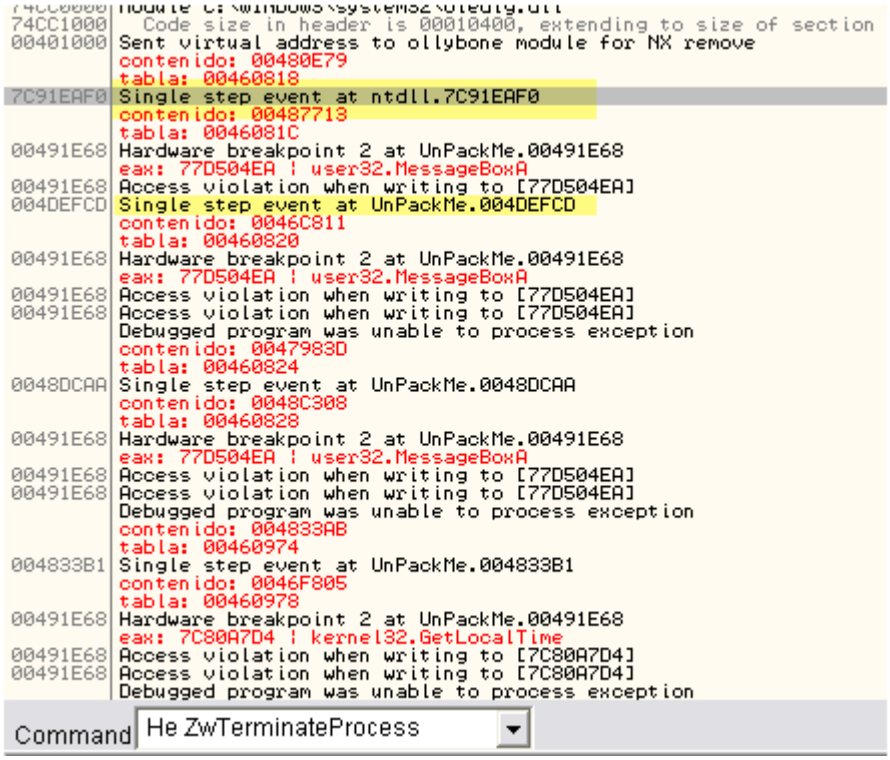
cmp table,460978

ja final

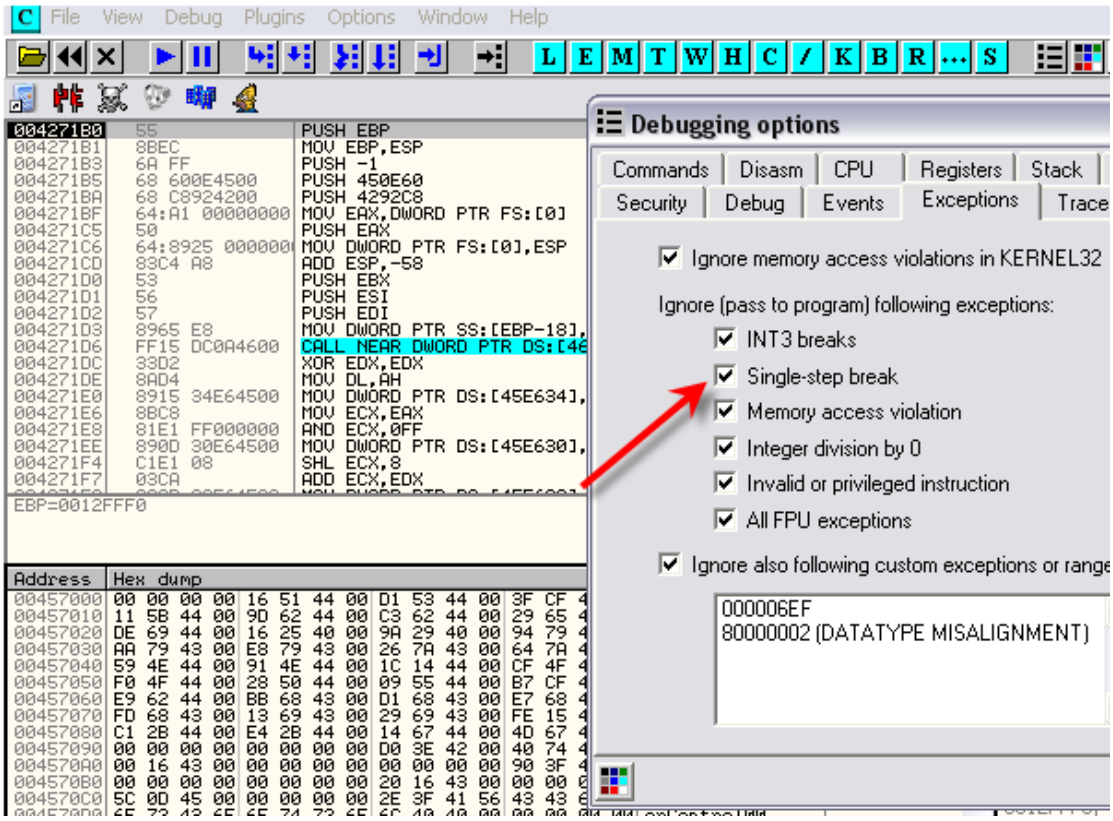
cmp [table],50000000

ja ToSkip

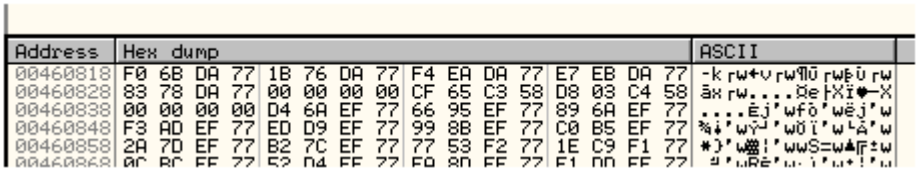
我们将脚本修改成这个样子(只让其修复第一个 DLL 中导出函数对应的 IAT 项),我们执行该脚本,接下来看看日志信息。



这里我们可以看到中间有几处单步异常(之前是没有的),好,我们重启 OD,断到 OEP 处,现在我们不需要使用 OllyBone 插件了,所以我们可以将忽略单步异常的选项勾选上。



我们再次在 ZwTerminateProcess 的入口处设置一个硬件执行断点,执行该脚本,接下来我们来看看第一个 DLL 中的 IAT 项有没有被修复。

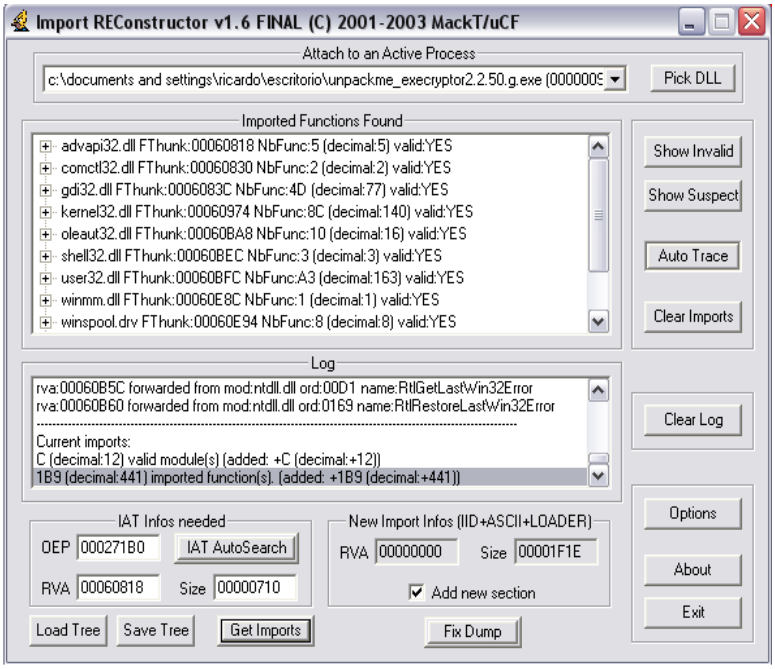


我们再来查看一下日志信息。

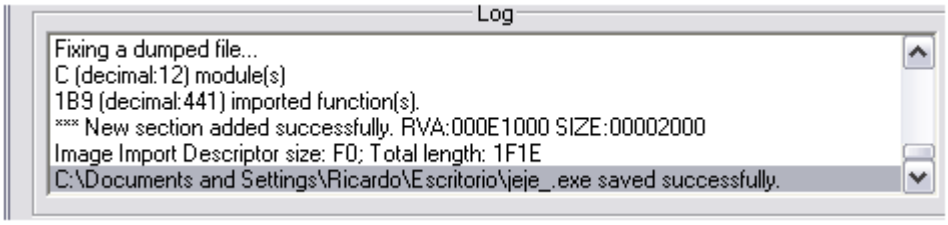
```
74CC1000   Code size in header is 00010400, extending to size of
00401000 Sent virtual address to ollybone module for NX remove
         contenido: 00480E79
         tabla: 00460818
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 77DA6BF0 ¦ ADVAPI32.RegCloseKey
00491E68 Access violation when writing to [77DA6BF0]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 00487713
         tabla: 0046081C
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 77DA761B ¦ ADVAPI32.RegOpenKeyExA
00491E68 Access violation when writing to [77DA761B]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 0046C811
         tabla: 00460820
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 77DAEAF4 ¦ ADVAPI32.RegCreateKeyExA
00491E68 Access violation when writing to [77DAEAF4]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 0047983D
         tabla: 00460824
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 77DAEBE7 ¦ ADVAPI32.RegSetValueExA
00491E68 Access violation when writing to [77DAEBE7]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 0048C308
         tabla: 00460828
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 77DA7883 ¦ ADVAPI32.RegQueryValueExA
00491E68 Access violation when writing to [77DA7883]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 004833AB
         tabla: 00460974
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 7C80176B ¦ kernel32.GetSystemTime
00491E68 Access violation when writing to [7C80176B]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
         contenido: 0046F805
         tabla: 00460978
00491E68 Hardware breakpoint 2 at UnPackMe.00491E68
         eax: 7C80A7D4 ¦ kernel32.GetLocalTime
00491E68 Access violation when writing to [7C80A7D4]
7C91E88E Hardware breakpoint 1 at ntdll.ZwTerminateProcess
```

我们可以看到第一个 DLL 中的 IAT 项这次都被修复了,而且跟之前一样中途会跳转到 ZwTerminateProcess 处,好,现在我们再次将脚本修改回去,让其修复整个 IAT。

执行了该脚本以后,我们可以看到所有的项都被修复了。



我们修复 dump 文件。



运行修复后的 dump 文件。

好了,程序完美运行,本章到此结束。