第四十六章-Patrick 的 CrackMe-Part1

本章我们继续加深难度,实验程序名称为 Patrick.exe,该程序的保护较强。大家会发现之前介绍的一些方法针对于该保护并不奏效。所有很多时候我们不得不适当变通,特定的保护特定解。

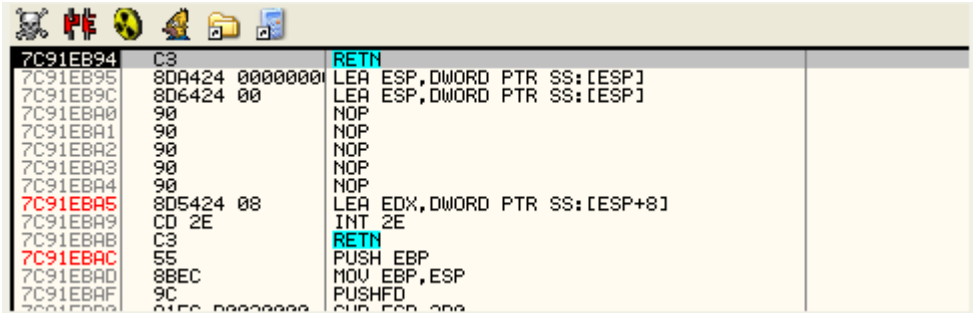
这里要提一句,一些商业壳不可能将所有的保护手段都运用到,因为商业壳的宗旨是要保证目标程序为任意软件。但是恰恰有些保护手段只针对于特定的程序。

好了,这里我们打开 OD,还记得之前介绍那款修改过的 Patched 4 这款 OD 吧,就它了,配置好反反调试插件。

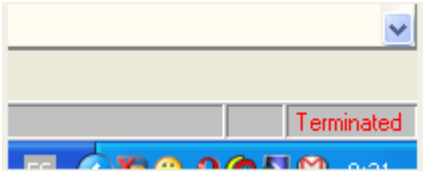
这个 CrackMe,作者的要求不仅仅是脱壳,还必须将 DLL 剥离,让 EXE 单独正常运行。

那么就是说 Patrick 主程序目录下的 AntiDebugDll.dll 这是一个核心 DLL,如果将这个 DLL 删除掉的话,主程序将无法正常运行。

好了,现在我们用 Patched4 这款 OD 加载 Patrick.exe。

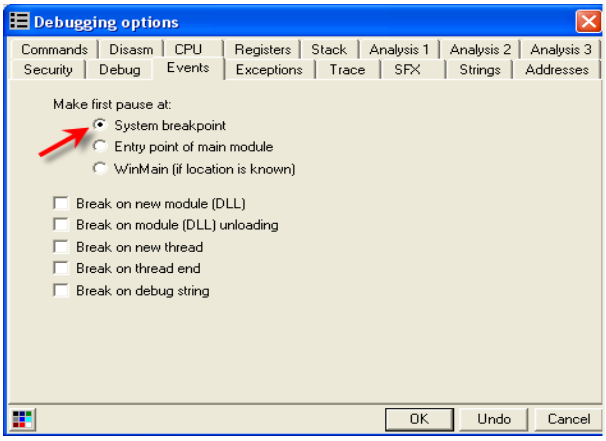


我们可以看到还没有到达入口点,程序就终止了。



好,下面我们来尝试将入口点修改为系统断点处,系统断点会在到达主程序入口点之前断下。

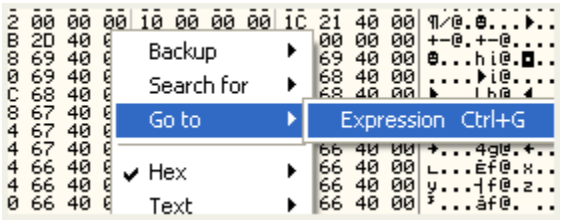
打开主菜单中的 Debugging options-Events,选中 System breakpoint。

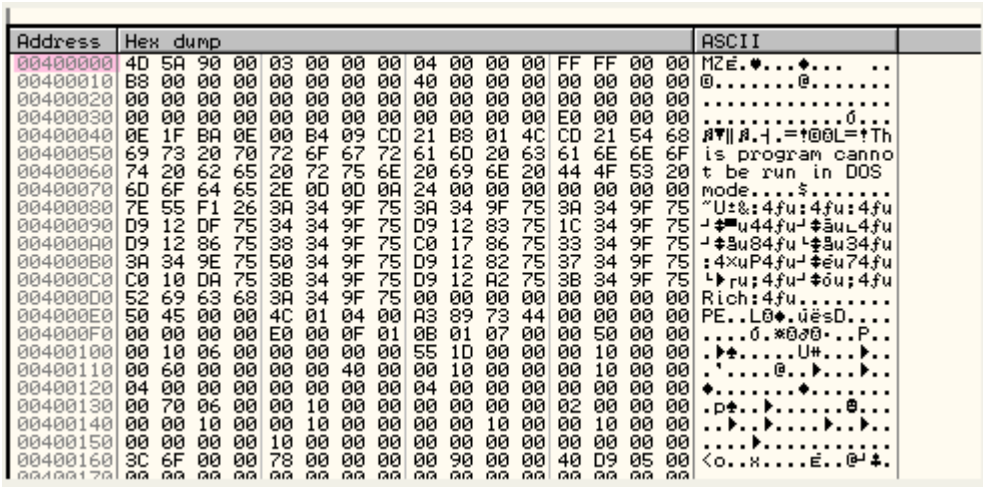


这样就可以让程序首次中断在系统断点处,这里要提一句,有些 DLL 会在到达入口点之前被加载,而这些 DLL 会检测主程序是否正在被调试,如果正在被调试的话就立即结束进程。

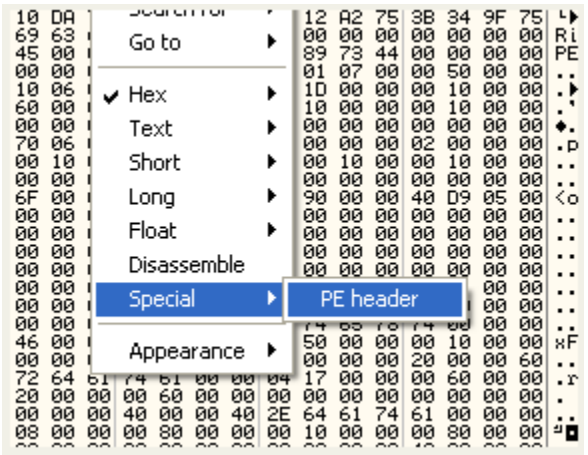
有一点必须明确,就是系统动态库并不会对 OD 进行检测。

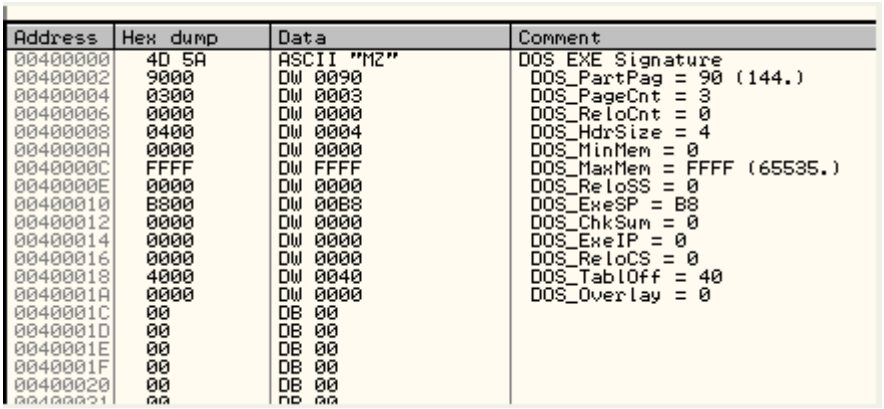
下面我们在数据窗口中单击鼠标右键选择 Goto Expression,输入 400000 定位到 PE 头。



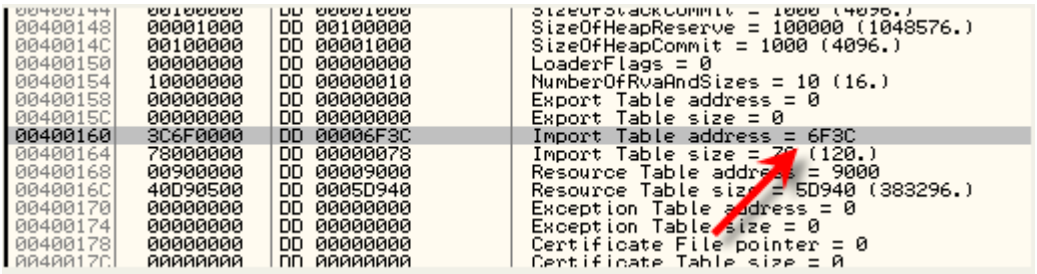


接着在数据窗口中单击鼠标右键选择 Special-PE Header,切换到 PE 结构解析模式。





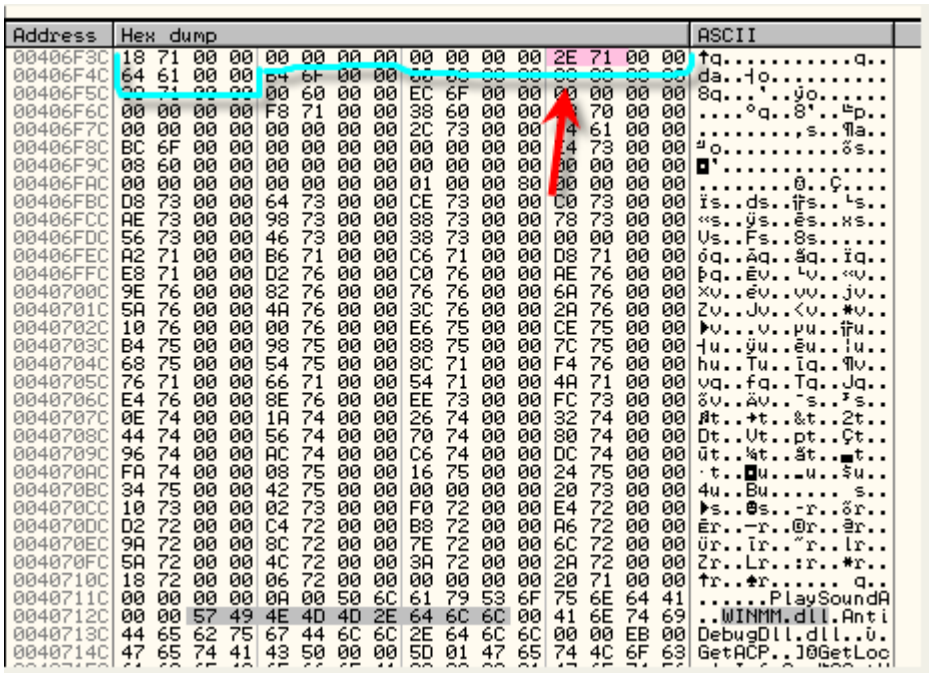
往下拉。



这里我们可以看到 Import Table(缩写:IT,俗称:导入表)的 RVA(相对虚拟地址)(PS:不要将 Import Table 与 IAT 搞混淆了,IAT 是 Import Address Table(输入函数地址表)的缩写,嘿嘿).

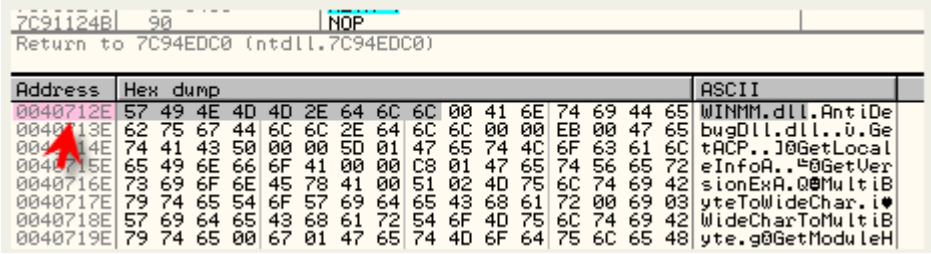
这里 6F3C + 映像基地址(400000)就可以定位到 Import Table 了。

将数据窗口的显示模式由 PE 解析模式切换回十六进制模式。



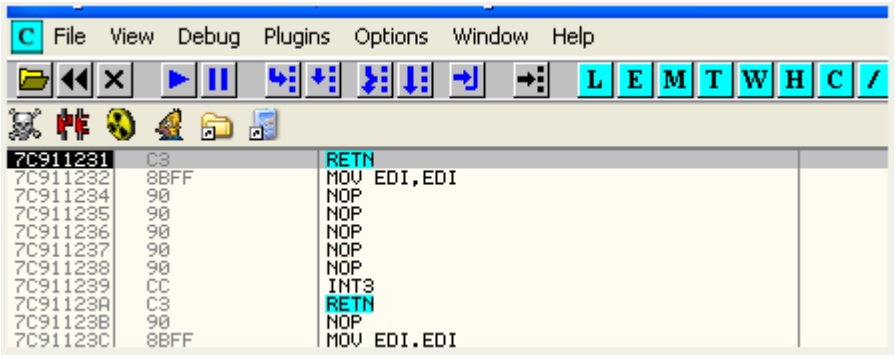
不知道大家是否还记得导入表的格式,每个 DLL 项占 5 个 DWORD。(PS:关于 PE 结构不了解的童鞋,可以参看传说中的小黄书,你懂得!嘿嘿 Windows PE 权威指南)

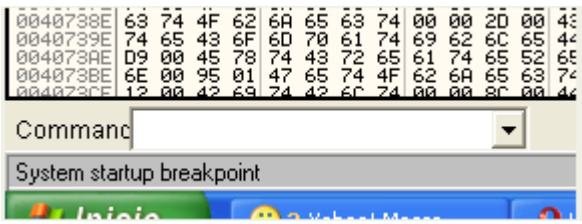
每个 DLL 项中的第 4 个 DWORD 指向了 DLL 的名称字符串,那么 40712E 这个地址就指向了第一个 DLL 的名称字符串,我们一起来看一看。



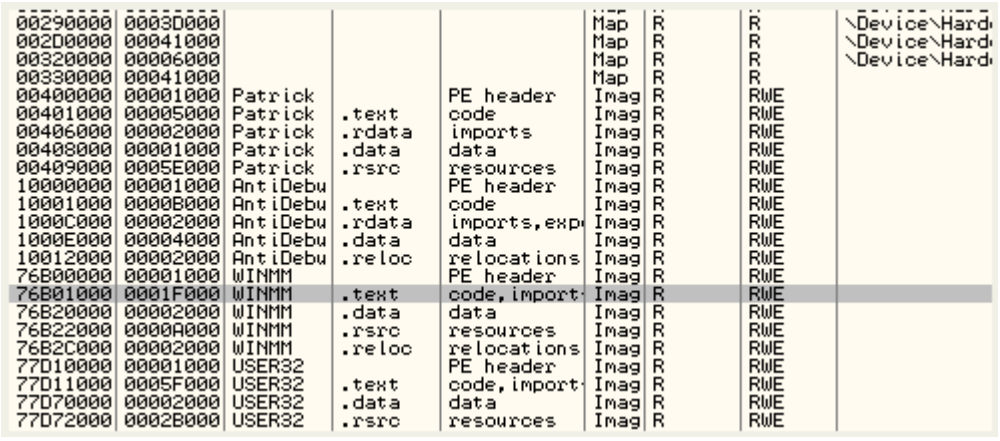
这里我们可以看到第一个 DLL 为 WINMM.DLL,我们假设在到达入口点之前如果该 DLL 被执行了的话,OD 就会终止。(PS:这是个系统 DLL,并不会进行反调试的处理,这里只是举个例子)

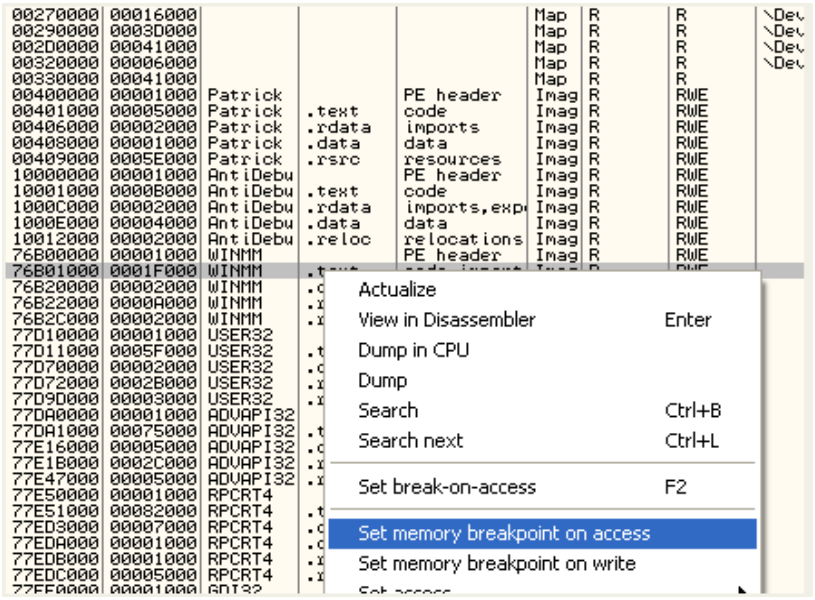
我们来一探究竟,首先 OD 断在系统断点处。



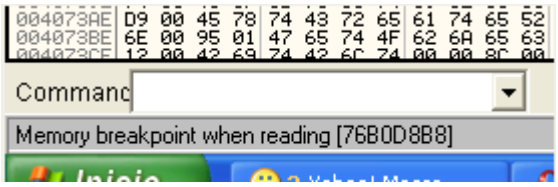


断在了这里,此时该 CrackMe 还没有到达入口点处,因为当某个 DLL 加载并执行以后就会终止进程,而当前进程并没有终止。也就是说此时存在反调试的 DLL 还没有得到执行,我们单击工具栏中的 M 按钮打开区段列表窗口,对该 DLL 的代码段设置内存访问断点。



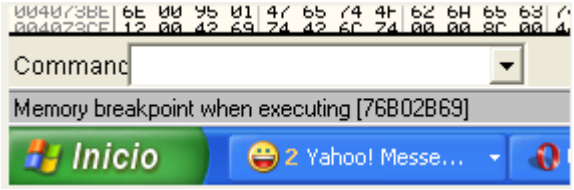


这里断点就设置好了,我们直接运行起来。

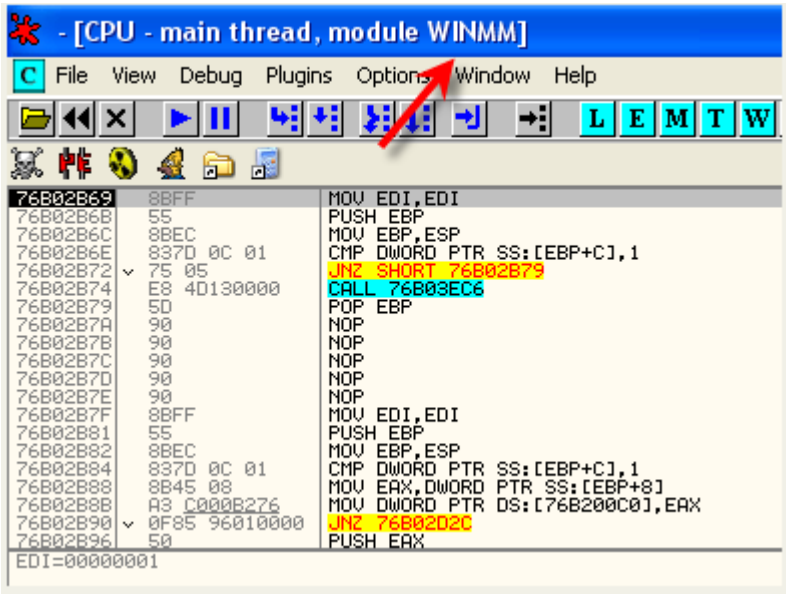


我们看到断了下来,这里是由于读取 76B0D8B8 地址处的内容导致的中断。

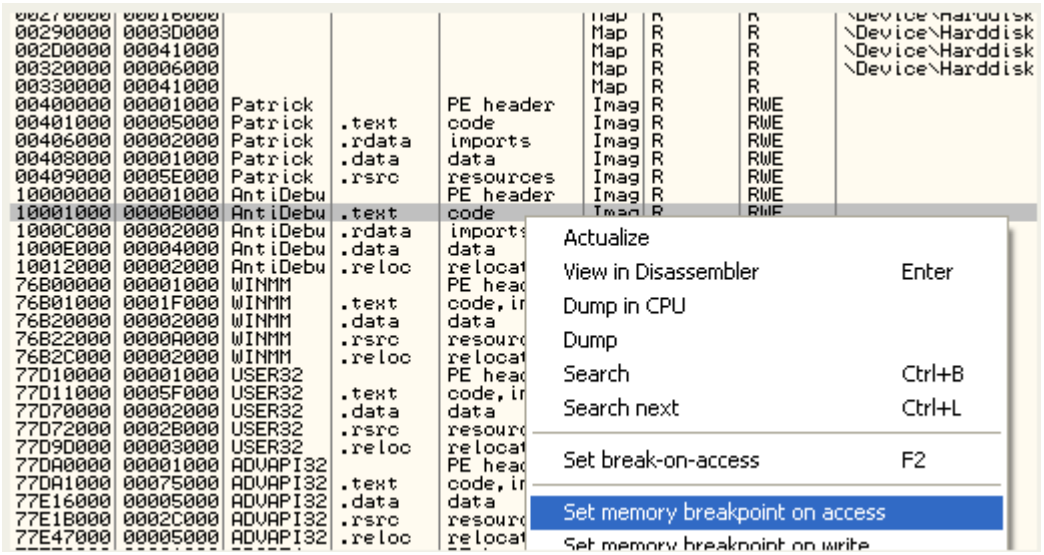
我们继续运行,直到触发内存执行断点为止。



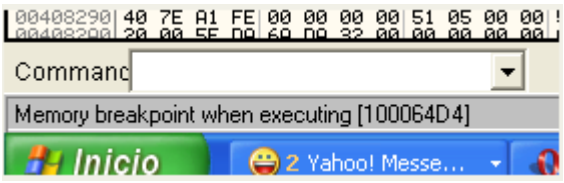
断在了这里,这里就要开始执行该 DLL 的代码了。

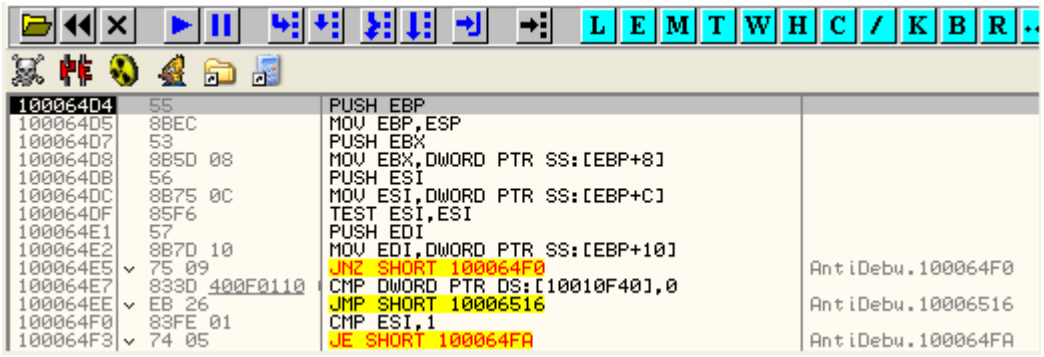


跟踪反调试 DLL 的流程就是这样的,接下来,我们重启 OD,再次重复一遍上面的步骤,这次我们直接对 AntiDebugdll.dll 的代码段设置内存访问断点。

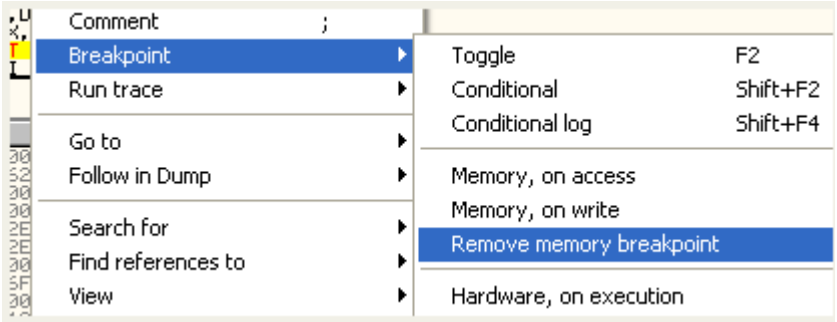


我们运行起来,会发现首次就触发了执行断点。

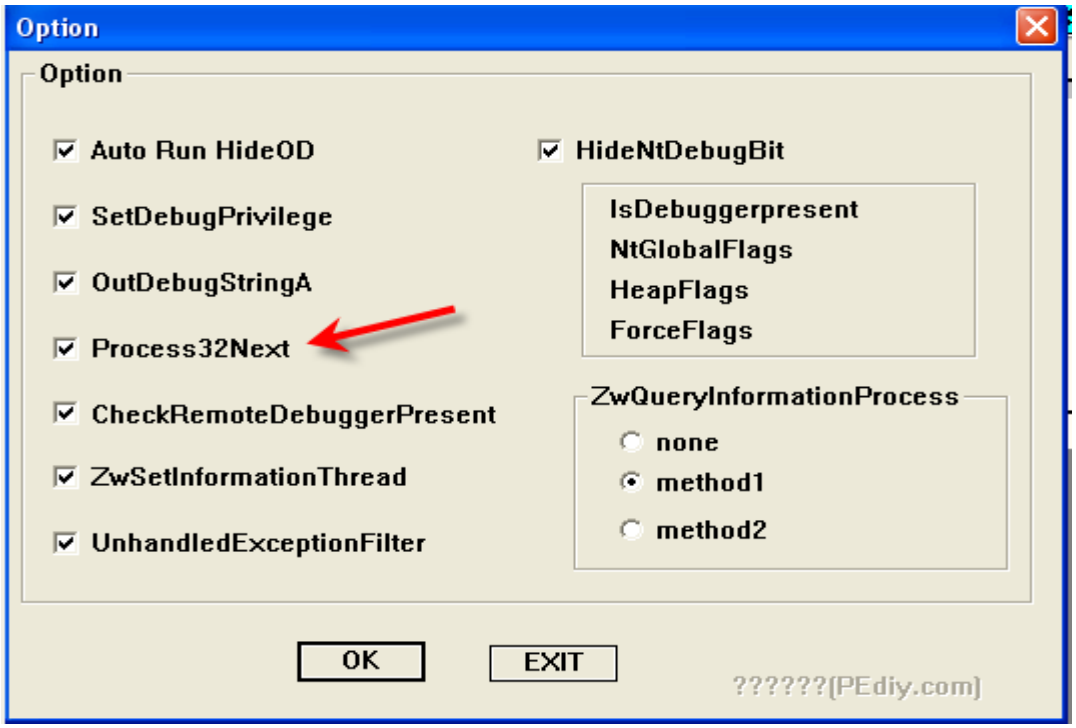




在运行之前我们首先删除掉之前设置的内存访问断点:

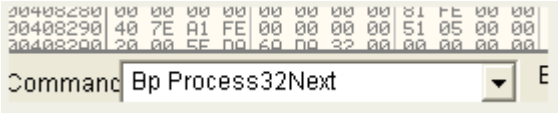


这里提示一下该程序既不是检测 OD 的进程名,也是不是检测 OD 窗口名,也不是 HideOD 插件中的涉及的那些检测方法,它是检测该进程是由谁创建的,通过调用 Process32Next 等 API 函数来遍历进程,判断当前进程的父进程是否为桌面进程来达到反调试的目的。

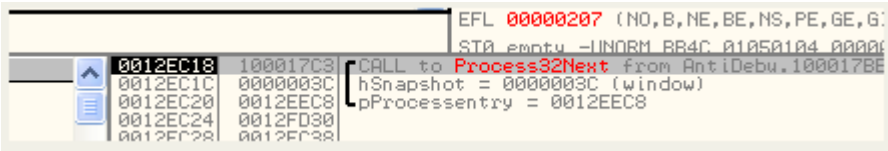


也就说该 CrackMe 会判断当前进程是否由 explorer.exe 启动,如果是的话,那么就说明是用户双击运行的。如果不是的话,那么就说明正在被调试,直接终止进程。

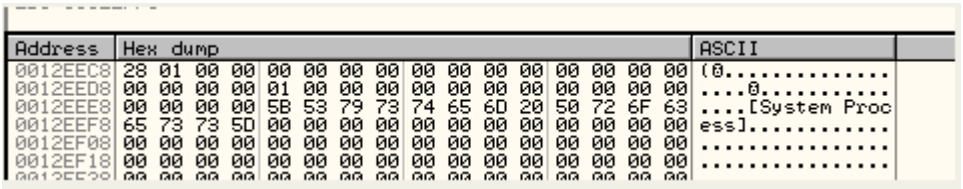
这里我们给 Process32Next 这个 API 函数设置一个断点。



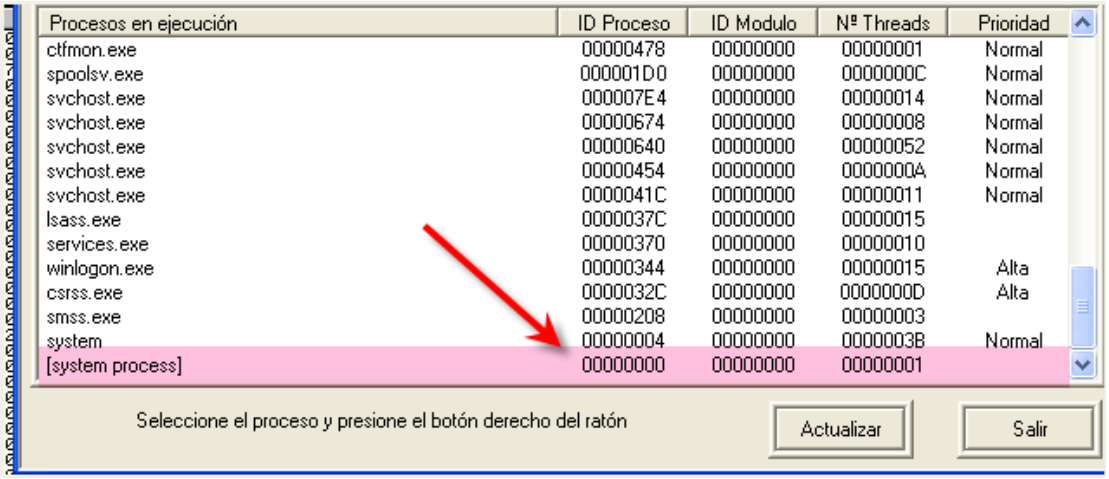
接着运行起来。



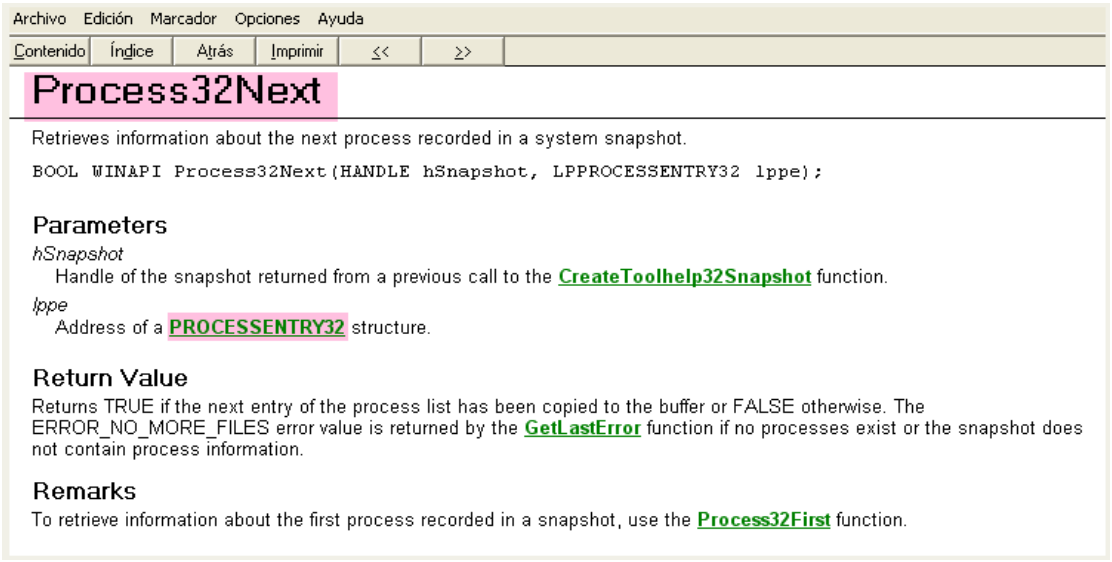
断了下来,这里可以看到第二个参数 pProcessentry 的值为 12EEC8。



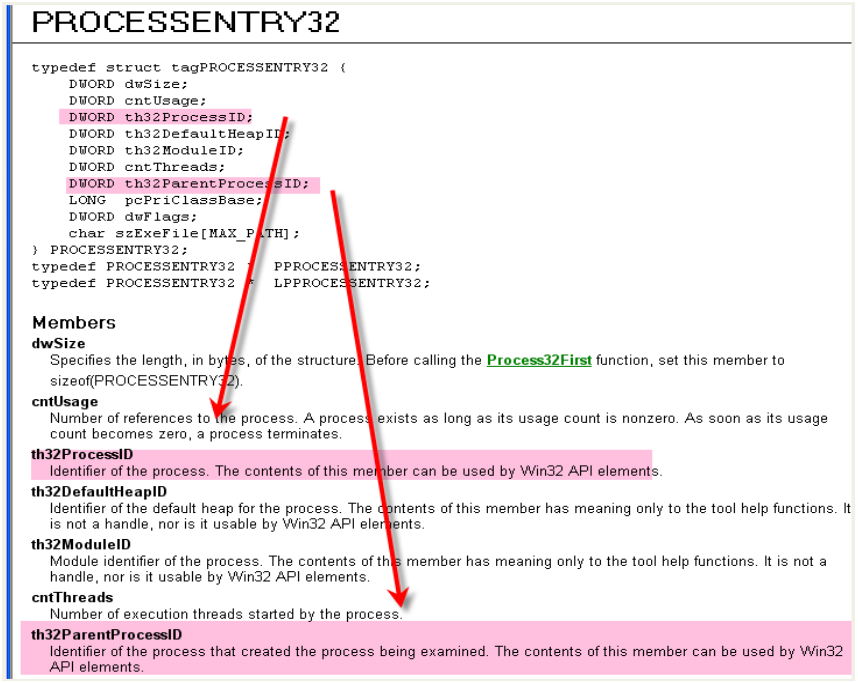
这里我们用 PUPE 这个小工具查看一下进程的 PID。



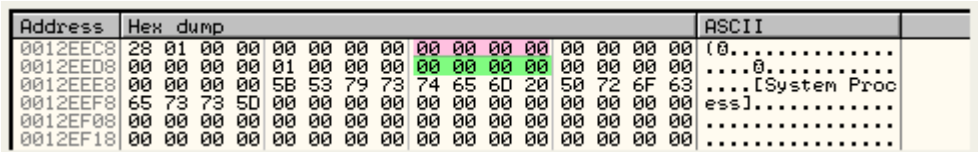
这里我们可以看到第一个进程的 PID 为零,我们结合 MSDN 来看。



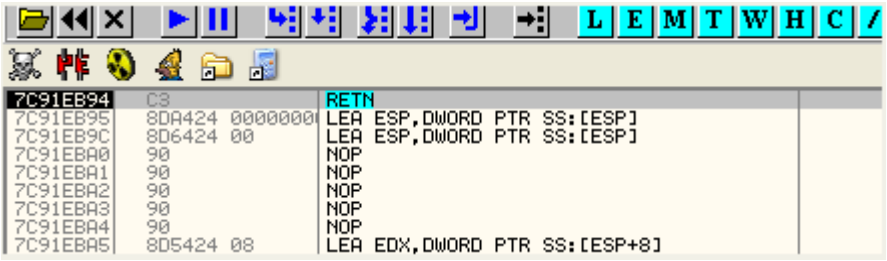
这里我们可以看到关于 Process32Next 这个 API 函数的说明。我们看下关于 PROCESSENTRY32 这个结构体的解释。

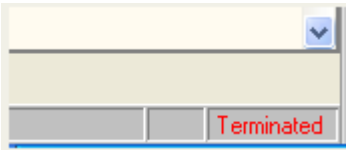


这里我们可以看到第三个字段为 th32ProcessID 即进程 PID,第七个字段为 th32ParentProcessID 即父进程 ID。

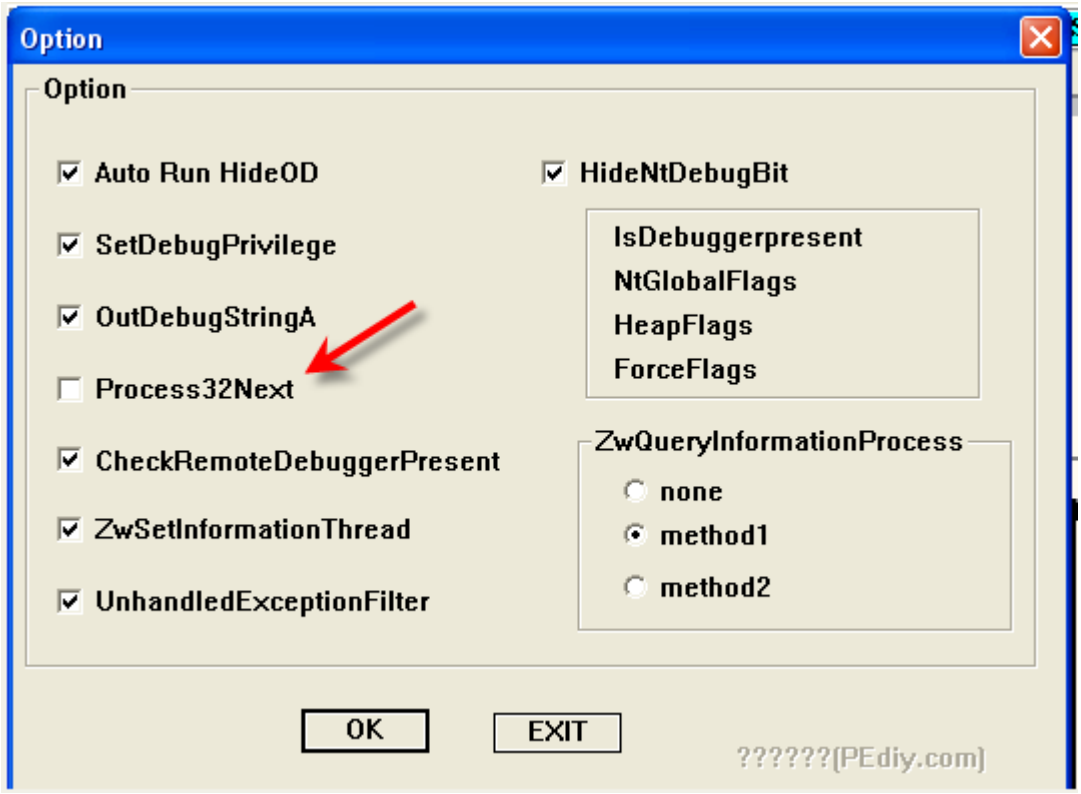


这里我用粉红色标注出了 PID,绿色标注出了 PPID。这里我们可以看到当前遍历到的这个进程的 PID(进程 ID)和 PPID(父进程 ID)都为零,代表是[System Process]这个进程,这个进程并不是我们要定位的,我们直接运行起来。

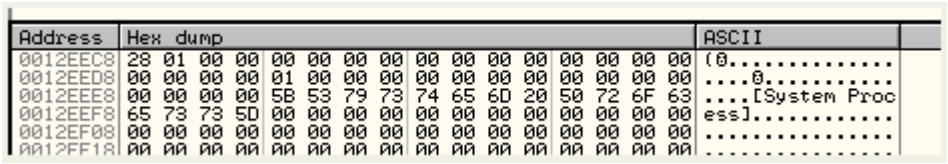




嘿嘿,程序终止了。怎么这样就终止了呢?难道[System Process]这个系统进程有问题?不太可能吧,那么会不会是 HideOD 插件冲突了的原因呢？我们去掉 HideOD 中 Process32Next 这一项的对勾试试。

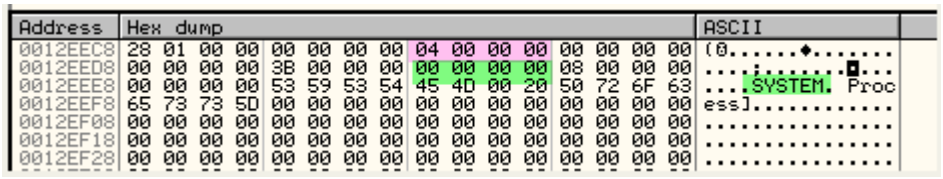


现在我们重启 OD,这里我们不需要再给该 DLL 的代码段设置内存访问断点了,我们直接给 Process32Next 下断。

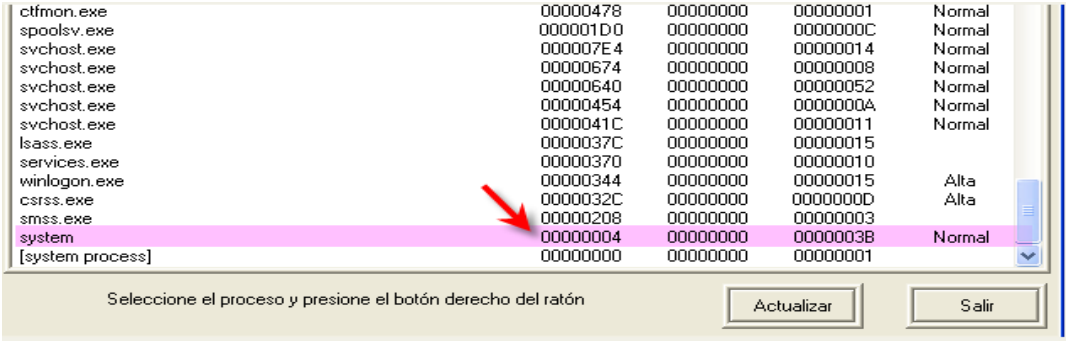


断了下来,我们可以看到 PID 和 PPID 都为零。

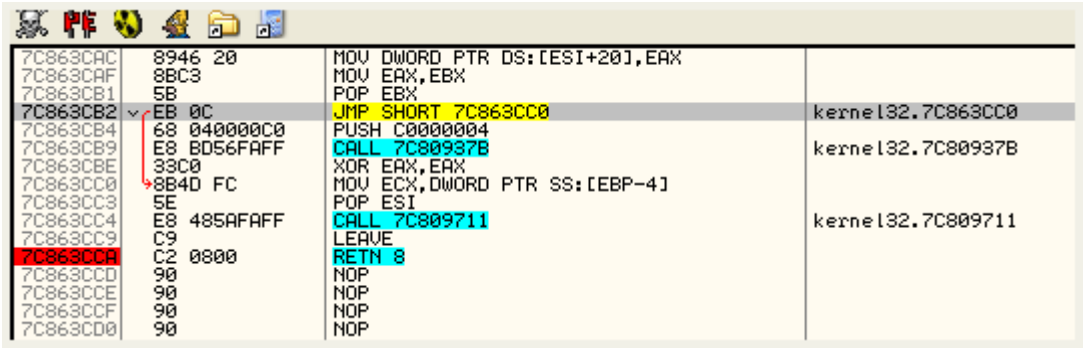
这里就是刚刚出问题的地方,如果勾选了 HideOD 上的 Process32Next 这个选项的话,当该函数执行完毕,EAX 等于零,说明出错了,程序就直接退出了。这里我们不勾选这一项,EAX 非零。函数执行成功,这样程序就不会直接退出了。



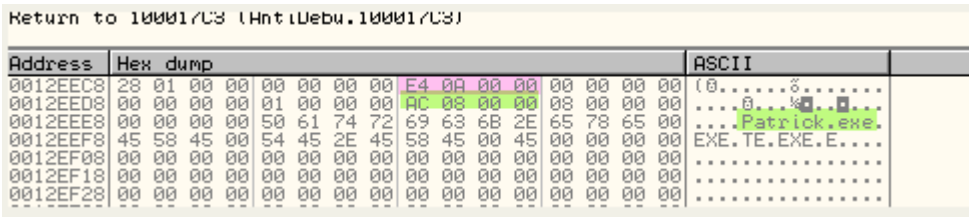
我们运行起来,这次程序并没有退出,这里遍历到了进程快照中的第二个进程。即 SYSTEM 进程,PID 为 4,我用粉红色标注出来了。PPID 为 0,我用绿色标注出来了。



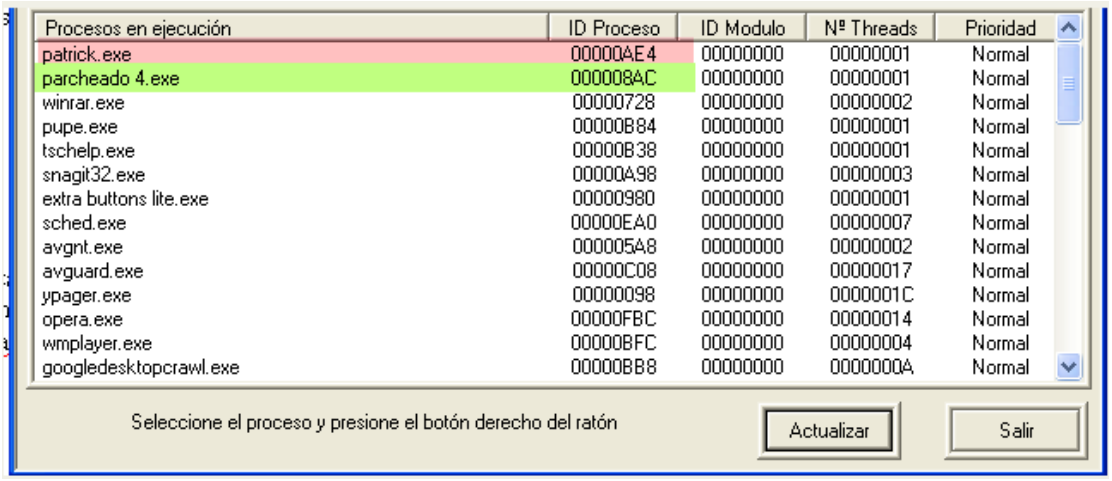
其实我们还可以对删除掉 Process32Next 入口处的断点,将断点下在该函数的返回处。



我们继续运行直到出现 Patrick.exe 为止。

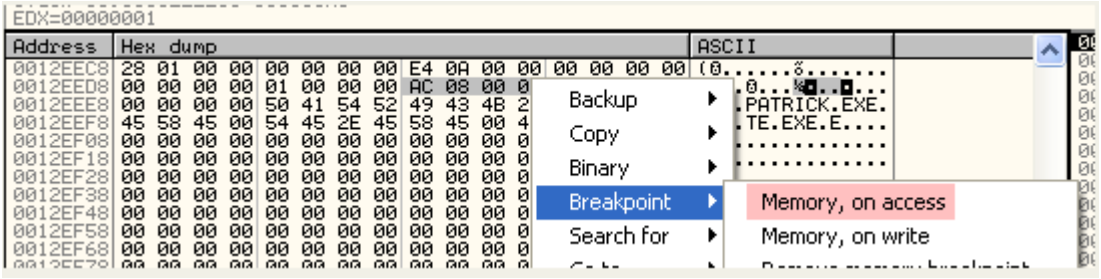


这里我们可以看到此时定位到了 Patrick.exe 这个进程。PID 为 0x0AE4,PPID 为 0x8AC,这里我们对比着 PUPE 里面的进程信息来看。我们单击 PUPE 中的 Actualizar(刷新)按钮。

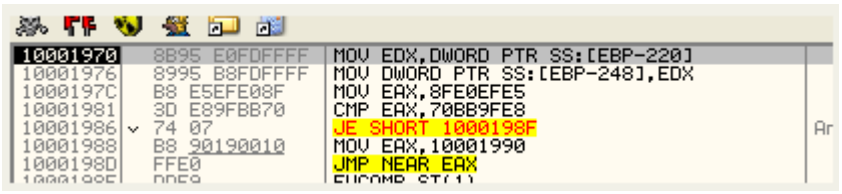


Patrick.exe 的 PID 为 0xAE4,而 Parcheado4.exe(Patched 4)的 PID 为 0x8AC。

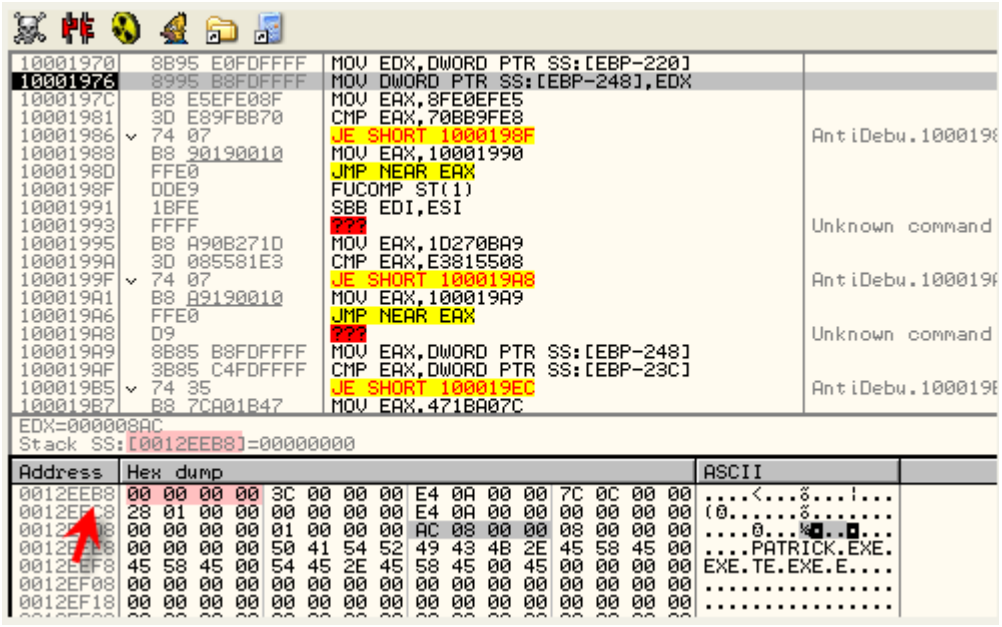
也就是说 Patrick.exe 的父进程并非 explorer.exe,而是 OD。所以 Patrick.exe 会直接退出。这里我们直接对 0x8AC 这 4 个字节设置内存访问断点,看什么地方会获取该 PPID。



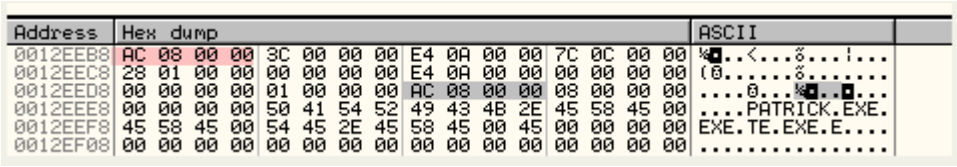
我们运行起来,断在了这里。



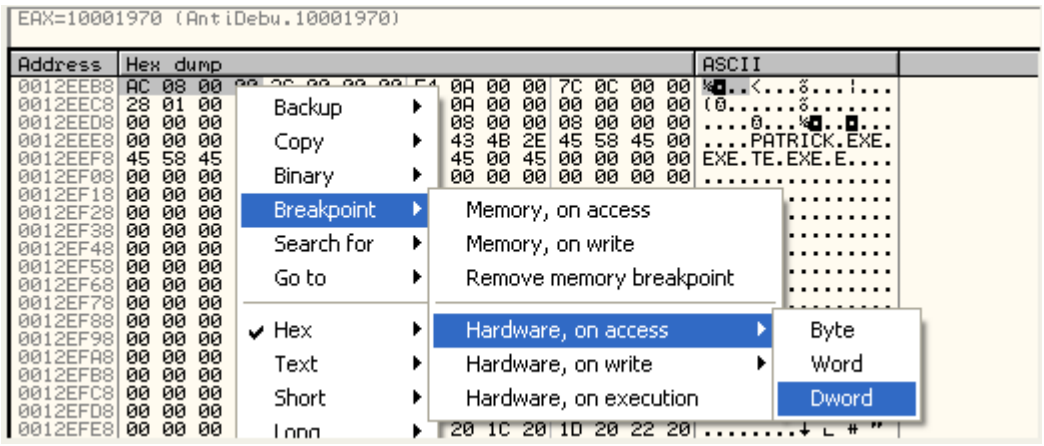
这里我们可以看到该指令将 PPID 读取出来存放到了另外一个地方。



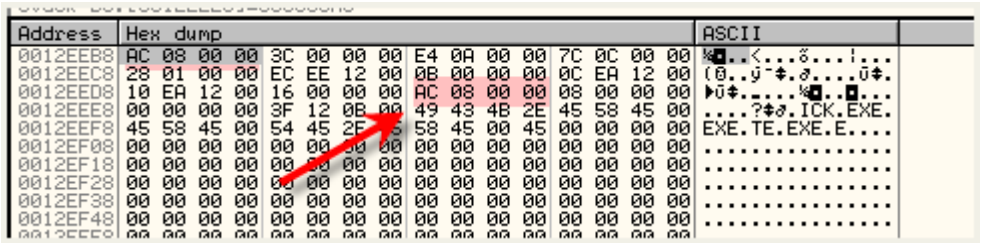
我们按 F7 单步,可以看到 PPID 被保存到了 12EEB8 地址处。



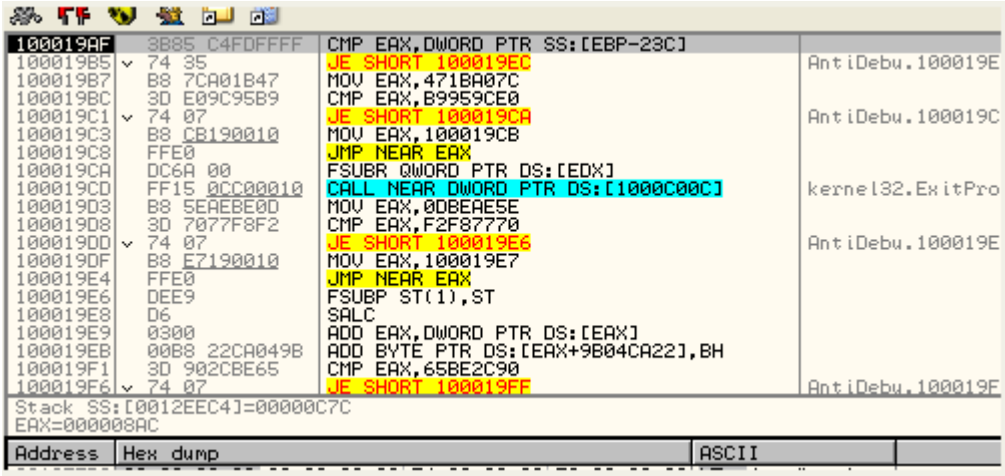
我们对 12EEB8 地址处的内容设置硬件访问断点,来定位何处会获取 PPID 的值。



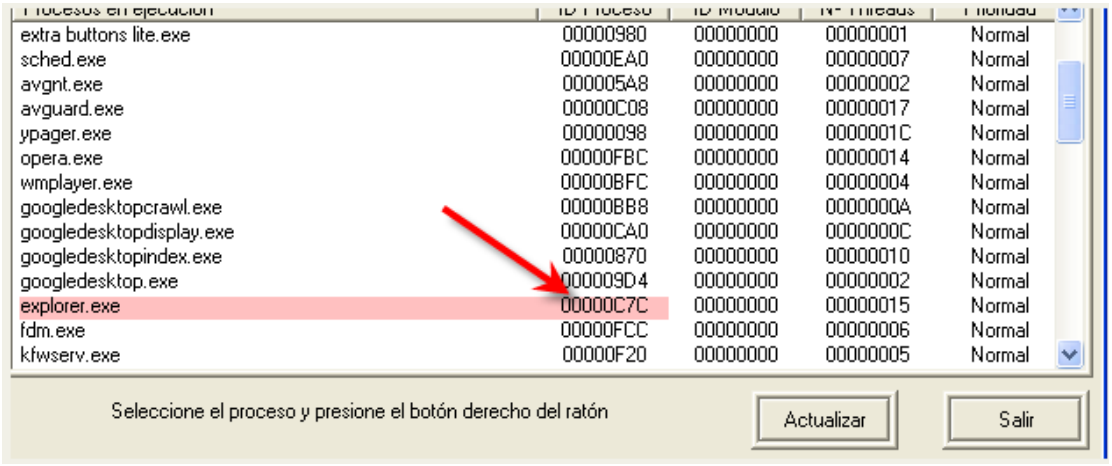
我们运行起来,会发现 PPID 最初保存的地址处的值已经被覆盖掉了。



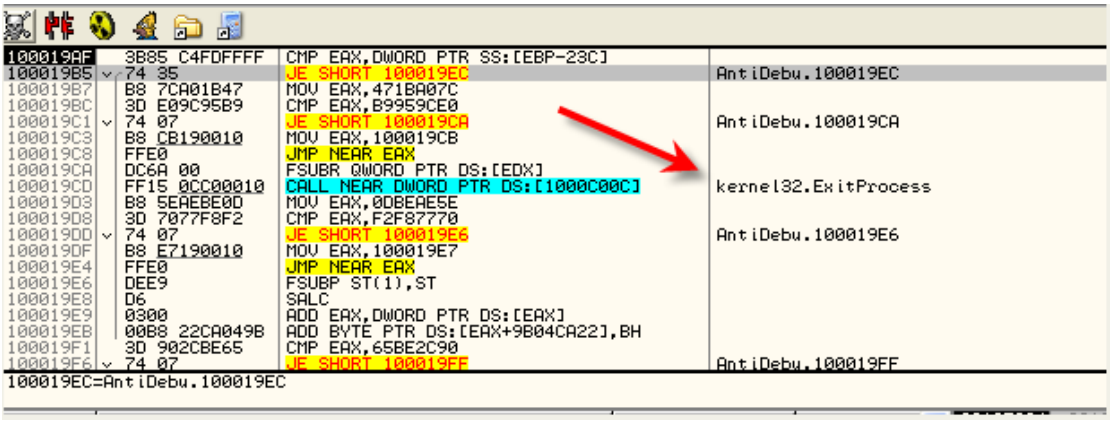
我们删除掉之前设置的内存访问断点。



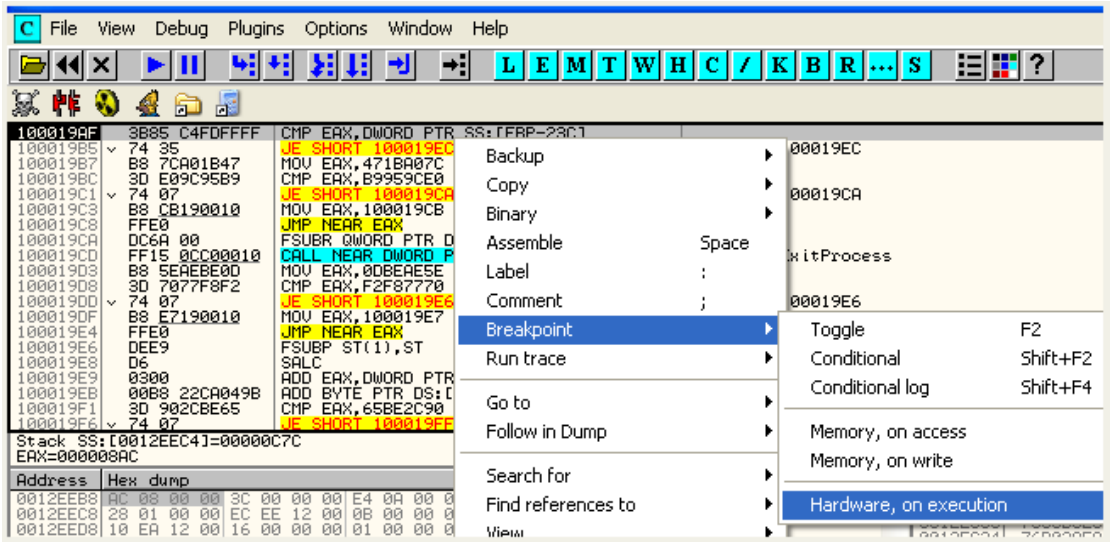
再次运行,断了下来,这里就是进行比较的地方。这个将 parcheado4(Patched 4)的 PID 与另一个 PID 0xC74 进行比较。



正如所料,是 explorer.exe 的 PID。

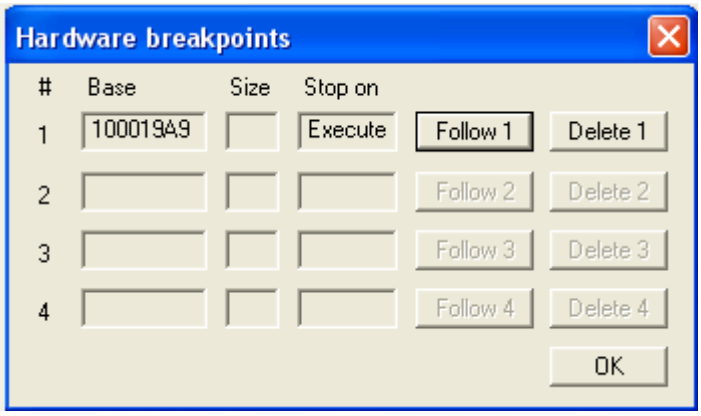


这里我们可以看到,判断 Patrick.exe 父进程的 PID 是否与 explorer.exe 的 PID 相同,相同的话,继续往下运行,不同的话则调用 ExitProcess 退出进程。这是其中一处反调试,如果我们将此处修改并保存到文件的话,那么如果还其他其他反调试的话,程序还是无法正常运行。所以这里我们不进行修改,我们删除掉之前设置的硬件访问断点。给进行比较的这一行指令设置硬件执行断点。

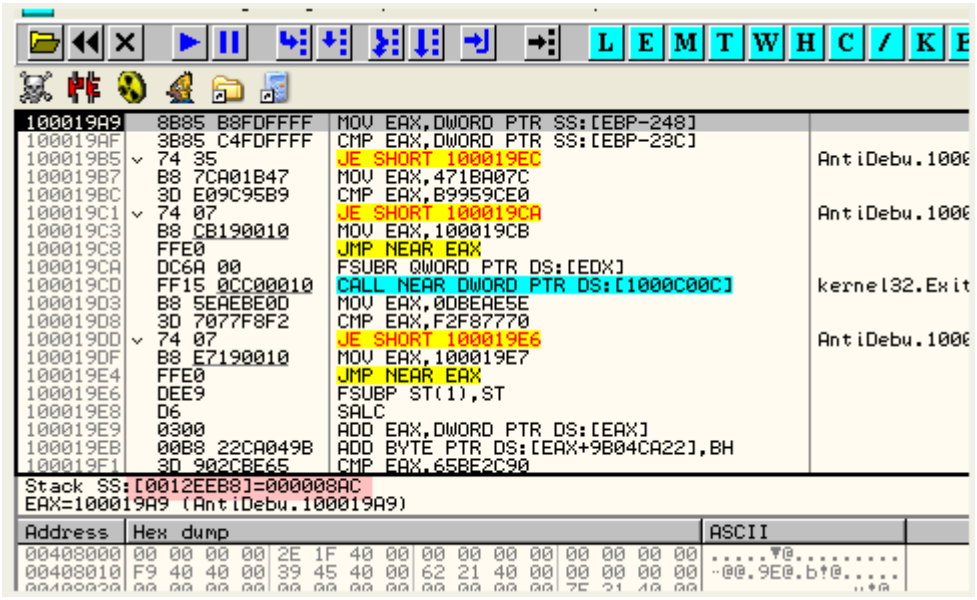


当程序断在这里的时候,我们可以手动修改 Patrick.exe 父进程的 PID,将其修改为 explorer.exe 的 PID 值。

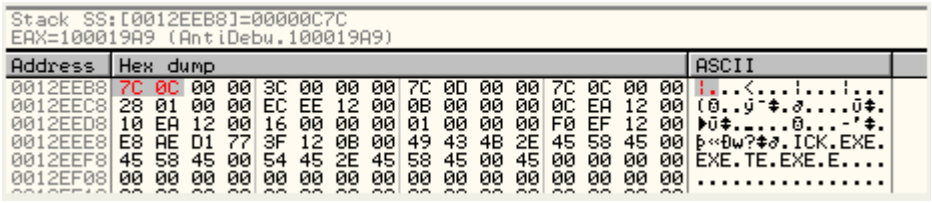
更加方便的方法是断在这一行的时候修改 EAX 的值。



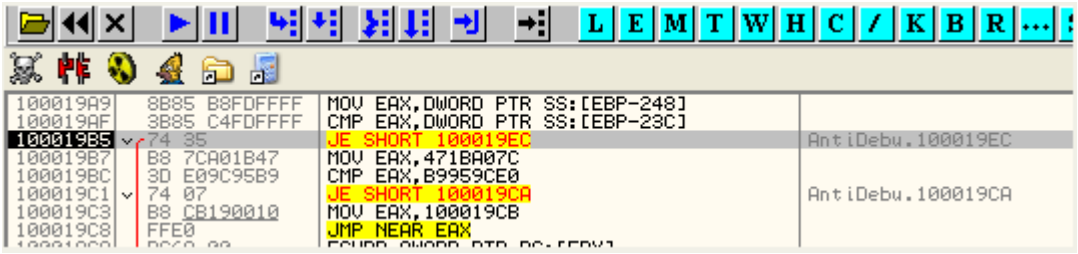
现在我们删除之前设置的 Process32Next 这个函数的断点。重启 OD。



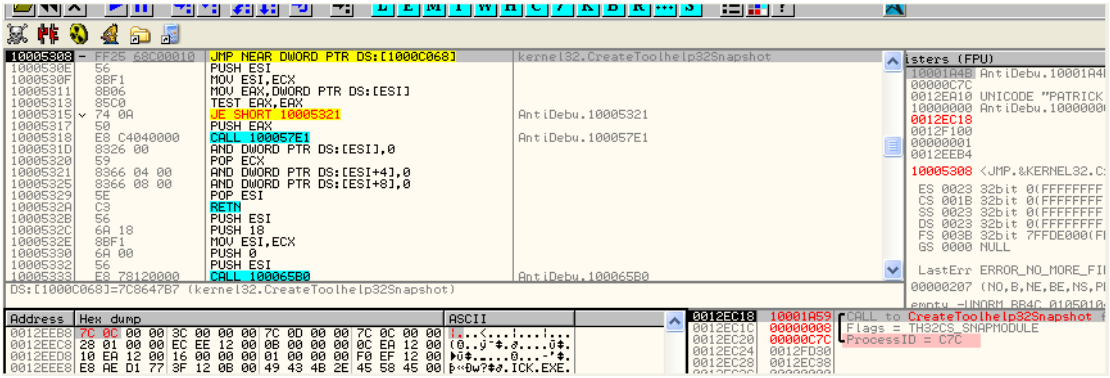
断在了这里,12EEB8 地址处保存的 PID 为 0x8AC。而另一个与之比较的 PID 为 0xC7C。



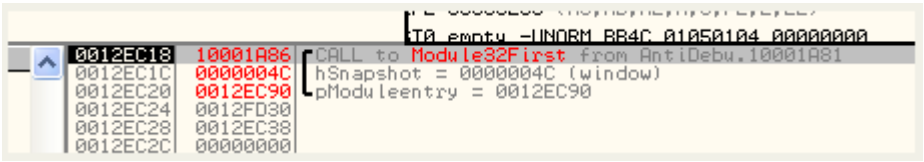
这里我们将 12EEB8 地址处的值修改为 0xC7C。这样两个 PID 就相等了,程序也就不会直接退出了。我们按 F7 键单步。



这样第一处反调试就绕过了。我们继续。



这里往下跟一点,就可以看到该 CrackMe 会获取 PID 为 0xC7C 进程(即 explorer.exe)的模块快照。进而检查模块的一些字段信息,用于判断 explorer 是不是重命名过的。



我们继续往下跟,就会到达 Module32First 函数调用处。这个 API 函数可以配合 Module32Next 这个 API 函数来遍历进程模块信息。

```
typedef MODULEENTRY32 *  LPMODULEENTRY32;
```

Members

dwSize
Specifies the length, in bytes, of the structure. Before calling the **Module32First** function, set this member to sizeof(MODULEENTRY32).

th32ModuleID
Module identifier in the context of the owning process. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

th32ProcessID
Identifier of the process being examined. The contents of this member can be used by Win32 API elements.

GlblcntUsage
Global usage count on the module.

ProccntUsage
Module usage count in the context of the owning process.

modBaseAddr
Base address of the module in the context of the owning process.

modBaseSize
Size, in bytes, of the module.

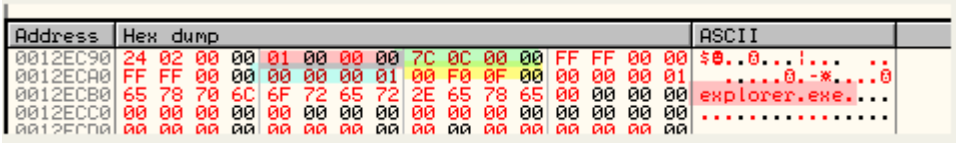
hModule
Handle of the module in the context of the owning process.

szModule
String containing the module name.

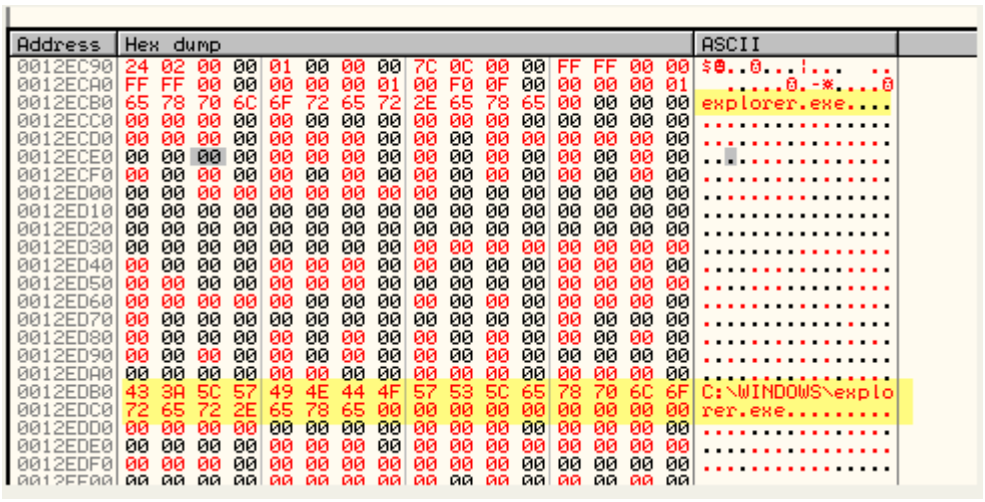
szExePath
String containing the location (path) of the module.

Note **modBaseAddr** and **hModule** are valid *only* in the context of the process specified by **th32ProcessID**.

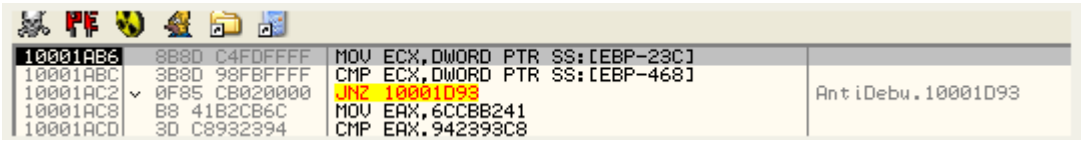
我们来看看 MSDN 中的关于该函数的说明。



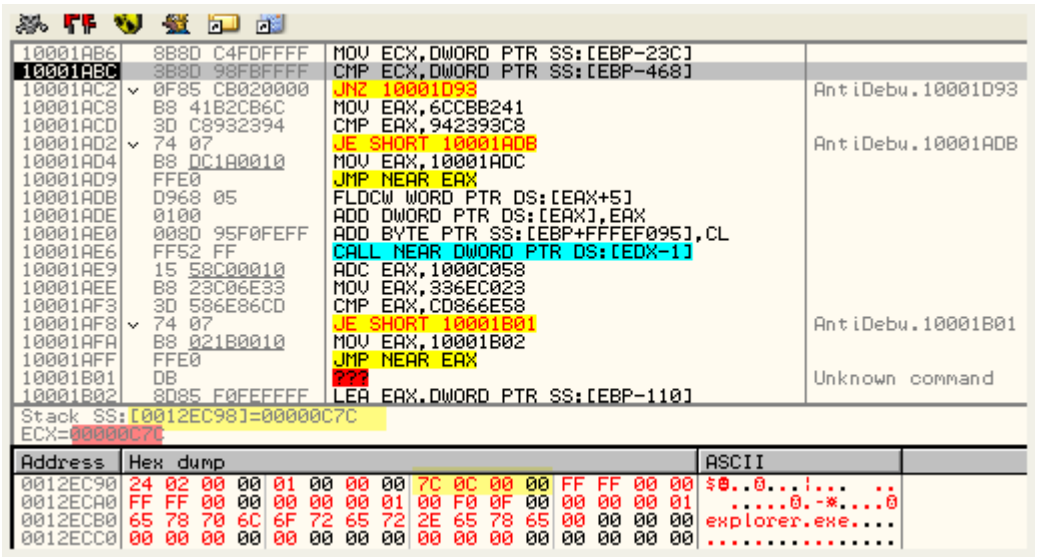
这里模块信息依次是:MODULEENTRY32 结构大小,模块标示符,进程 ID,全局模块引用计数,所在进程范围内模块引用计数,模块基地址,模块大小,模块句柄,模块名称,模块全路径,预留标志。这里我们可以看到模块基地址为 0x10000000。



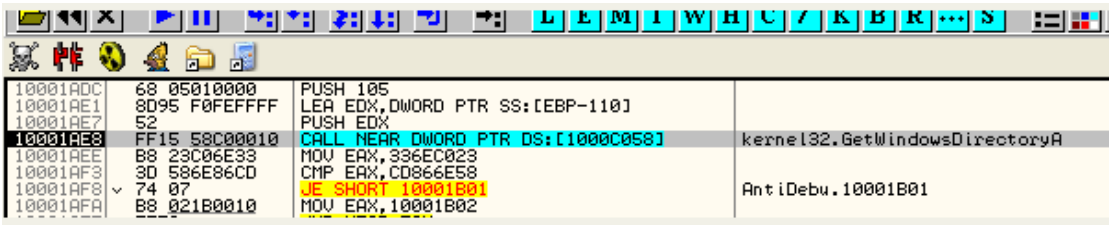
这里我们可以看到关于模块的相关信息,我们来一起看看通过该 CrackMe 通过模块信息会干什么事情。



往下跟一点就可以看到读取 0xC7C 与模块信息中的进程 PID 进行比较。判断该模块是不是 explorer.exe。



我们继续往下跟。



这里我们可以看到调用 GetWindowsDirectory 这个 API 函数获取 Windows 的目录,也就是 explorer.exe 所在目录。也就是说该 CrackMe 会检测父进程的路劲。

```
0012EC1C  0012EFF0  Buffer = 0012EFF0
0012EC20  00000105  BufSize = 105 (261.)
0012EC24  0012FD30
0012EC28  0012EC38
0012EC2C  00000000
```

这里我们可以看到 Windows 所在目录字符串将要保存的这个缓冲区中。

```
Address   Hex dump                                          ASCII
0012EFF0  43 3A 5C 57 49 4E 44 4F 57 53 00 20 DC 02 22 21  C:\WINDOWS. ▄@"!
0012F000  60 01 3A 20 52 01 9D 00 7D 01 78 01 A0 00 A1 00  `: R...}.x.á.í.
0012F010  A2 00 A3 00 A4 00 A5 00 A6 00 A7 00 A8 00 A9 00  ó.ú.ñ.Ñ.ª.º.¿.⌐.
0012F020  AA 00 AB 00 AC 00 AD 00 AE 00 AF 00 B0 00 B1 00  ¬.½.¼.¡.«.».░.▒.
0012F030  B2 00 B3 00 B4 00 B5 00 B6 00 B7 00 B8 00 B9 00  ▓.│.┤.╡.╢.╖.╕.╣.
```

这里我们可以看到保存到了这里。下面就进行比较路径是否一致了。这里路径是一致的,我们继续跟。

```
10001B2D  ^ 75 E2          JNZ SHORT 10001B11                    AntiDebu.10001B11
10001B2F    8BBD 68FBFFFF  MOV EDI,DWORD PTR SS:[EBP-498]
10001B35    66:A1 04C20010 MOV AX,WORD PTR DS:[1000C204]
10001B3B    66:8907        MOV WORD PTR DS:[EDI],AX
10001B3E    B8 05CE11FA    MOV EAX,FA11CE05
10001B43    3D E848E906    CMP EAX,6E948E8
10001B48  v 74 07          JE SHORT 10001B51                     AntiDebu.10001B51
10001B4A    B8 521B0010    MOV EAX,10001B52
10001B4F    FFE0           JMP NEAR EAX
AX=005C
Stack DS:[0012EFFA]=2000

Address   Hex dump                                          ASCII
0012EFEA  00 00 00 00 00 00 43 3A 5C 57 49 4E 44 4F 57 53  ......C:\WINDOWS
0012EFFA  00 20 DC 02 22 21 60 01 3A 20 52 01 9D 00 7D 01  . ▄@"!`: R...}.
0012F00A  78 01 A0 00 A1 00 A2 00 A3 00 A4 00 A5 00 A6 00  x.á.í.ó.ú.ñ.Ñ.ª.
0012F01A  A7 00 A8 00 A9 00 AA 00 AB 00 AC 00 AD 00 AE 00  º.¿.⌐.¬.½.¼.¡.«.
```

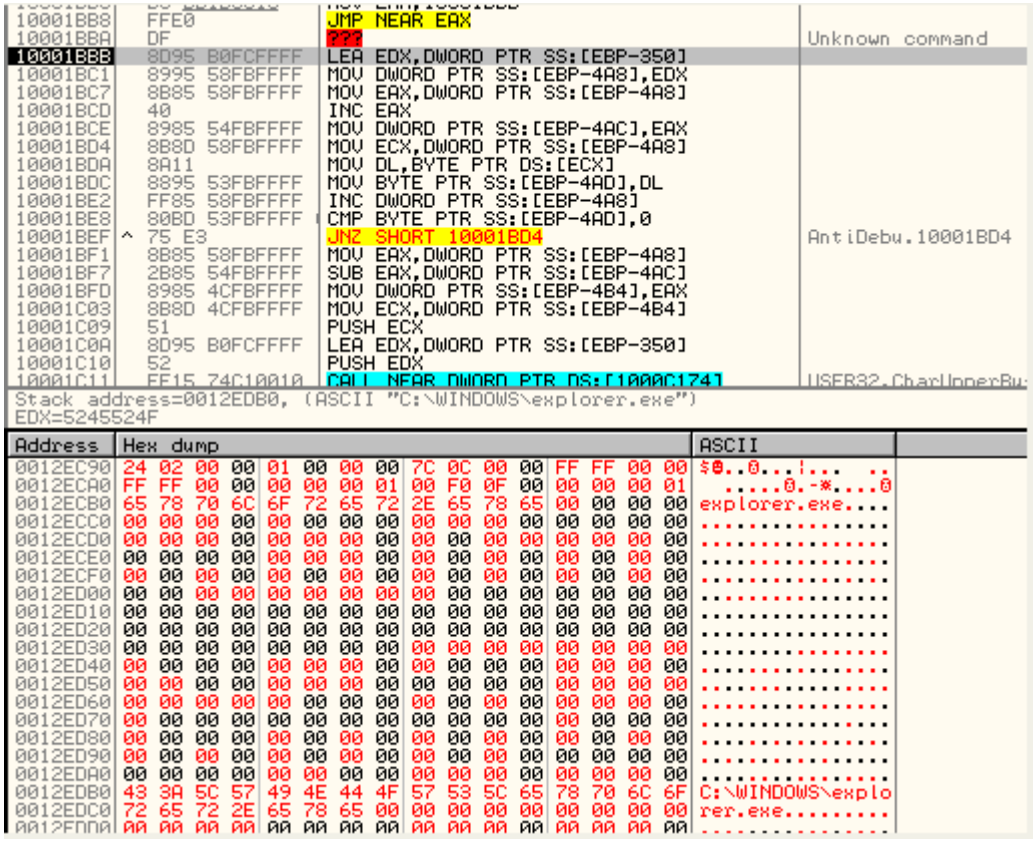
我们继续往下,可以看到该程序将 C:\WINDOWS 与 explorer.exe 进行字符串拼接。

```
10001B7F    8BBD 60FBFFFF  MOV EDI,DWORD PTR SS:[EBP-4A0]
10001B85    8B0D 08C20010  MOV ECX,DWORD PTR DS:[1000C208]
10001B8B    890F           MOV DWORD PTR DS:[EDI],ECX
10001B8D    8B15 0CC20010  MOV EDX,DWORD PTR DS:[1000C20C]
10001B93    8957 04        MOV DWORD PTR DS:[EDI+4],EDX
10001B96    A1 10C20010    MOV EAX,DWORD PTR DS:[1000C210]
10001B9B    8947 08        MOV DWORD PTR DS:[EDI+8],EAX
10001B9E    8A0D 14C20010  MOV CL,BYTE PTR DS:[1000C214]
10001BA4    884F 0C        MOV BYTE PTR DS:[EDI+C],CL
10001BA7    B8 E7DBB4C0    MOV EAX,C0B4DBE7
10001BAC    3D 78234C40    CMP EAX,404C2378
10001BB1  v 74 07          JE SHORT 10001BBA                     AntiDebu.10001BBA
10001BB3    B8 BB1B0010    MOV EAX,10001BBB
10001BB8    FFE0           JMP NEAR EAX
DS:[1000C208]=4C505845

Address   Hex dump                                          ASCII
1000C208  45 58 50 4C 4F 52 45 52 2E 45 58 45 00 00 00 00  EXPLORER.EXE....
1000C218  52 45 41 4C 49 47 4E 2E 44 4C 4C 00 50 52 4F 43  REALIGN.DLL.PROC
1000C228  53 2E 44 4C 4C 00 00 00 43 41 44 54 2E 44 4C 4C  S.DLL...CADT.DLL
1000C238  00 00 00 00 55 55 50 44 41 54 45 53 59 53 54 45  ....UUPDATESYSTE
```

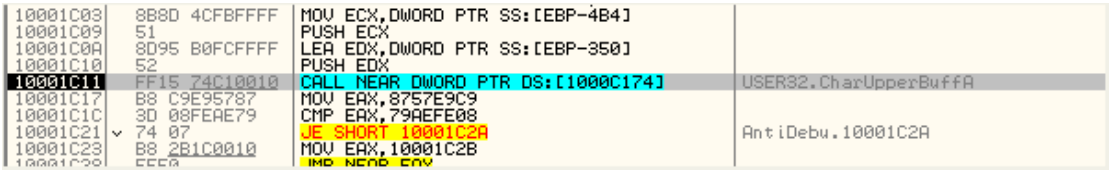
这里就得到了 explorer.exe 的全路径。

```
10001B7D  ^ 75 E2          JNZ SHORT 10001B61                    AntiDebu.10001B6
10001B7F    8BBD 60FBFFFF  MOV EDI,DWORD PTR SS:[EBP-4A0]
10001B85    8B0D 08C20010  MOV ECX,DWORD PTR DS:[1000C208]
10001B8B    890F           MOV DWORD PTR DS:[EDI],ECX
10001B8D    8B15 0CC20010  MOV EDX,DWORD PTR DS:[1000C20C]
10001B93    8957 04        MOV DWORD PTR DS:[EDI+4],EDX
10001B96    A1 10C20010    MOV EAX,DWORD PTR DS:[1000C210]
10001B9B    8947 08        MOV DWORD PTR DS:[EDI+8],EAX
10001B9E    8A0D 14C20010  MOV CL,BYTE PTR DS:[1000C214]
10001BA4    884F 0C        MOV BYTE PTR DS:[EDI+C],CL
10001BA7    B8 E7DBB4C0    MOV EAX,C0B4DBE7
10001BAC    3D 78234C40    CMP EAX,404C2378
10001BB1  v 74 07          JE SHORT 10001BBA                     AntiDebu.10001BB
10001BB3    B8 BB1B0010    MOV EAX,10001BBB
10001BB8    FFE0           JMP NEAR EAX
EAX=C0B4DBE7

Address   Hex dump                                          ASCII
0012EFEA  00 00 00 00 00 00 43 3A 5C 57 49 4E 44 4F 57 53  ......C:\WINDOWS
0012EFFA  5C 45 58 50 4C 4F 52 45 52 2E 45 58 45 00 7D 01  \EXPLORER.EXE.}.
0012F00A  78 01 A0 00 A1 00 A2 00 A3 00 A4 00 A5 00 A6 00  x.á.í.ó.ú.ñ.Ñ.ª.
0012F01A  A7 00 A8 00 A9 00 AA 00 AB 00 AC 00 AD 00 AE 00  º.¿.⌐.¬.½.¼.¡.«.
```

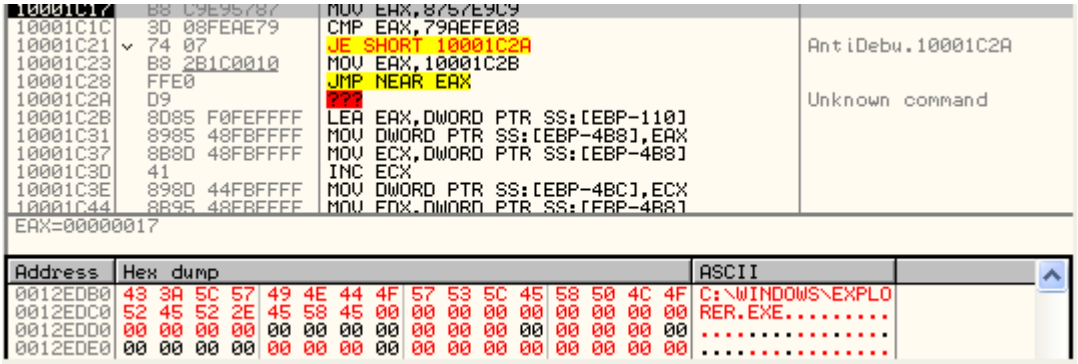
好了,我们来看看接下来要干什么。



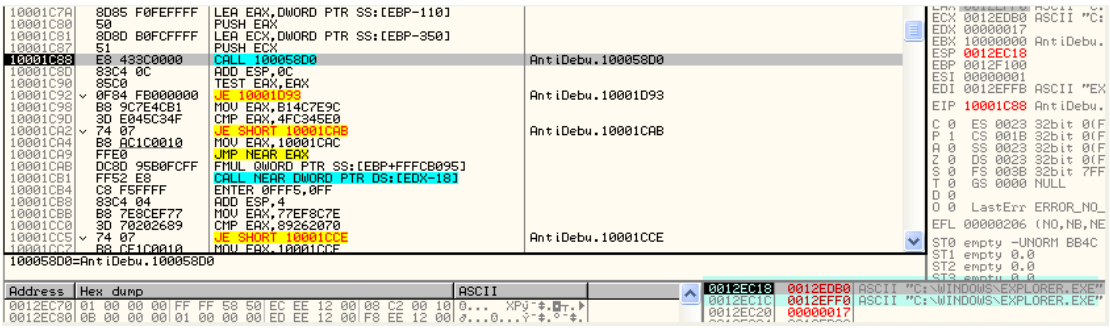
下面将开始与快照中的进程路径进行比较。



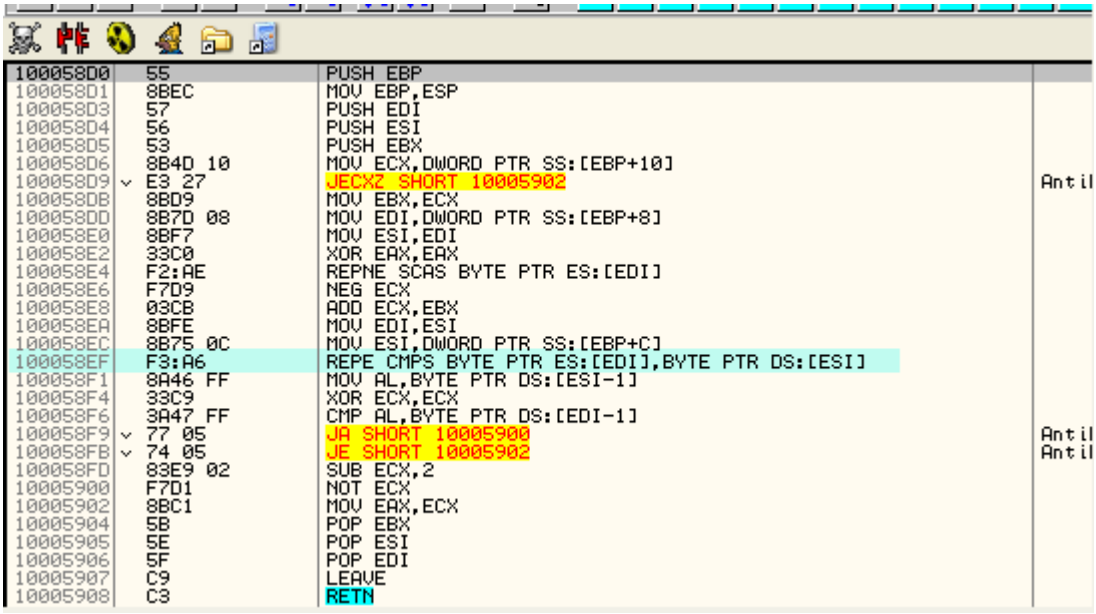
这里我们到了 CharUpperBuffA 这个 API 函数的调用处。这里是将目标缓冲区中的字符串由小写转大写。



这里没看出什么特别的,就是将遍历到的进程全路径小写转大写。然后通过 GetWindowsDirectory 获取 windows 目录,接着与 explorer.exe 进行字符串连接,然后进行比较。

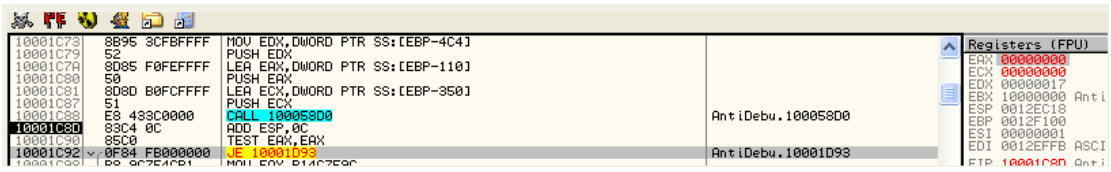


这里我们就到了字符串比较函数处,这里待比较的两个字符串明显是相等的,所以返回值会为零。我们单击回车键进去看看这个字符串比较函数的实现。

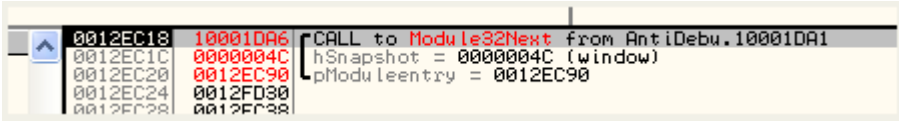


这里我们就可以看到该字符串比较函数的实现了,我们单击减号键返回。

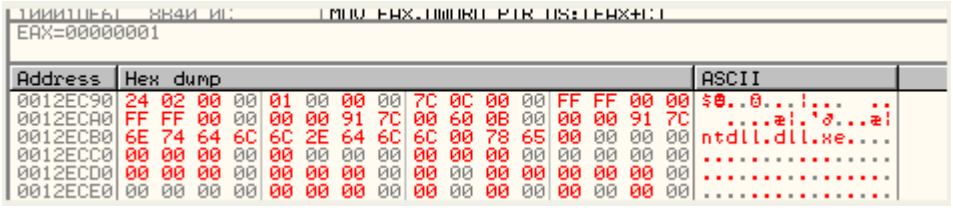
我们直接按 F8 键执行这个字符串比较函数。



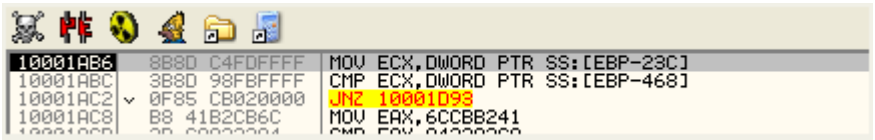
这里我们可以看到 EAX 的值为零,下面的 JE 跳转就成立。

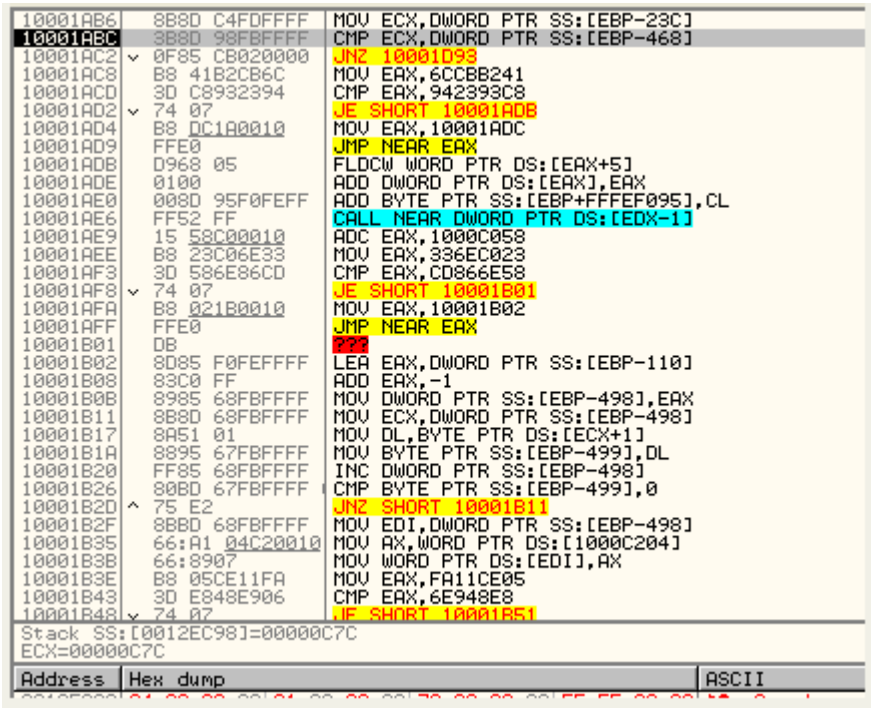


继续往下跟,这里可以看到该 CrackMe 还要进行进一步的判断,继续遍历下一个模块。

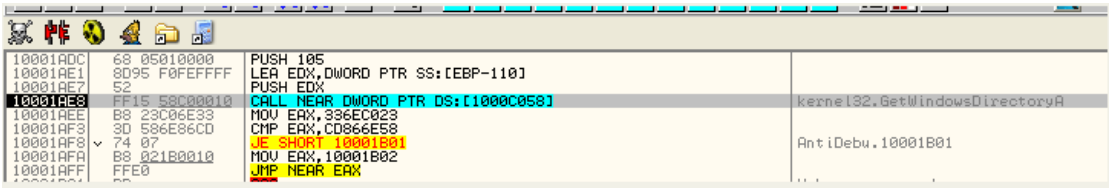


这里我们可以看到接下来这个模块是 ntdll.dll。其模块标示符为 0x01，PID 为 0xC7C,以及其他相关的模块信息。

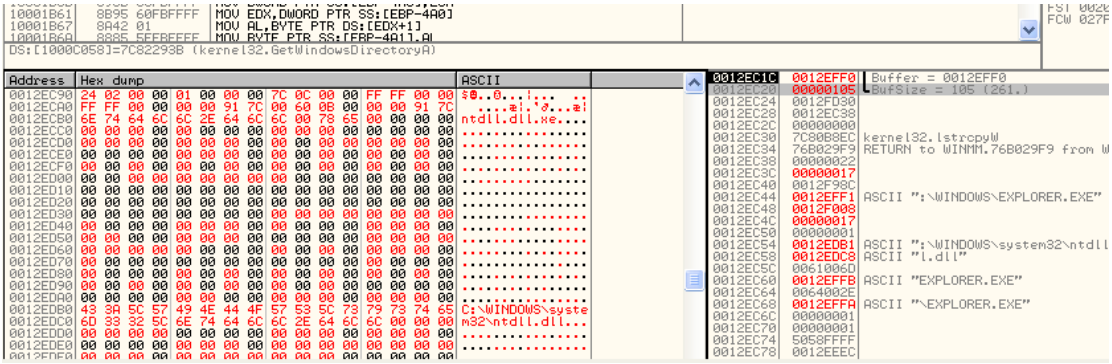




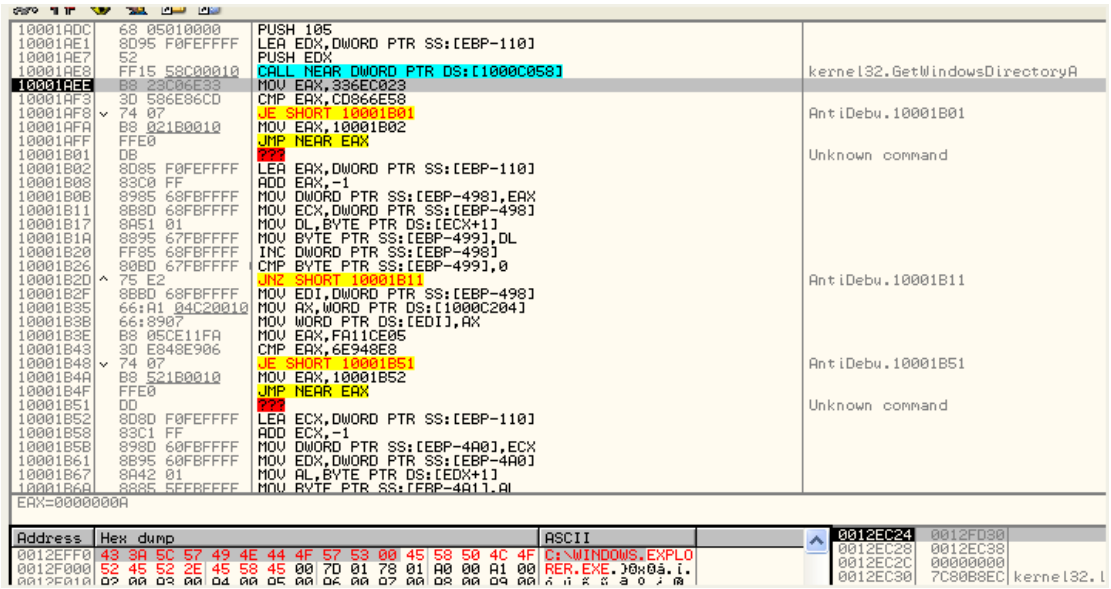
接下来继续进行比较,依次判断 explorer.exe 的所有模块。



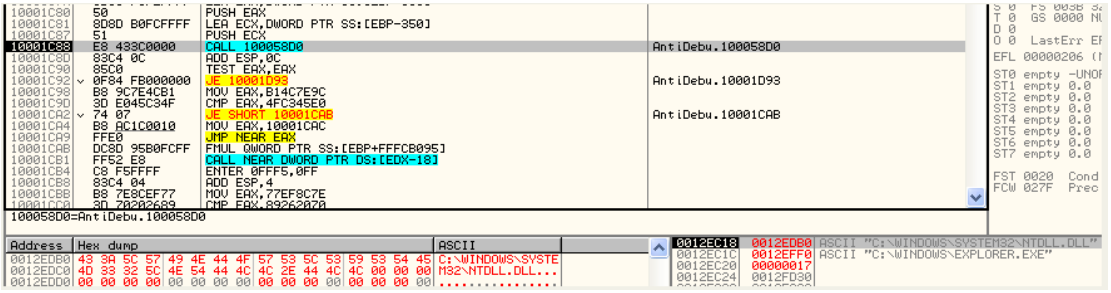
这里再次到了 GetWindowsDirectoryA 的调用处。



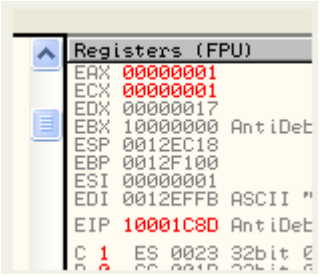
我们继续往下。



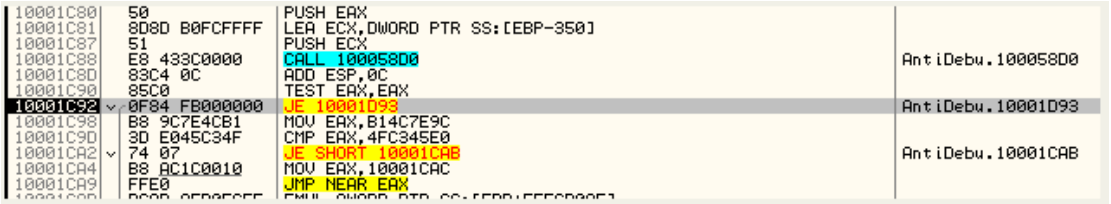
这里又是获取 Windows 的所在目录,重复前面的比较步骤,这里我们就不再赘述了,继续跟踪。



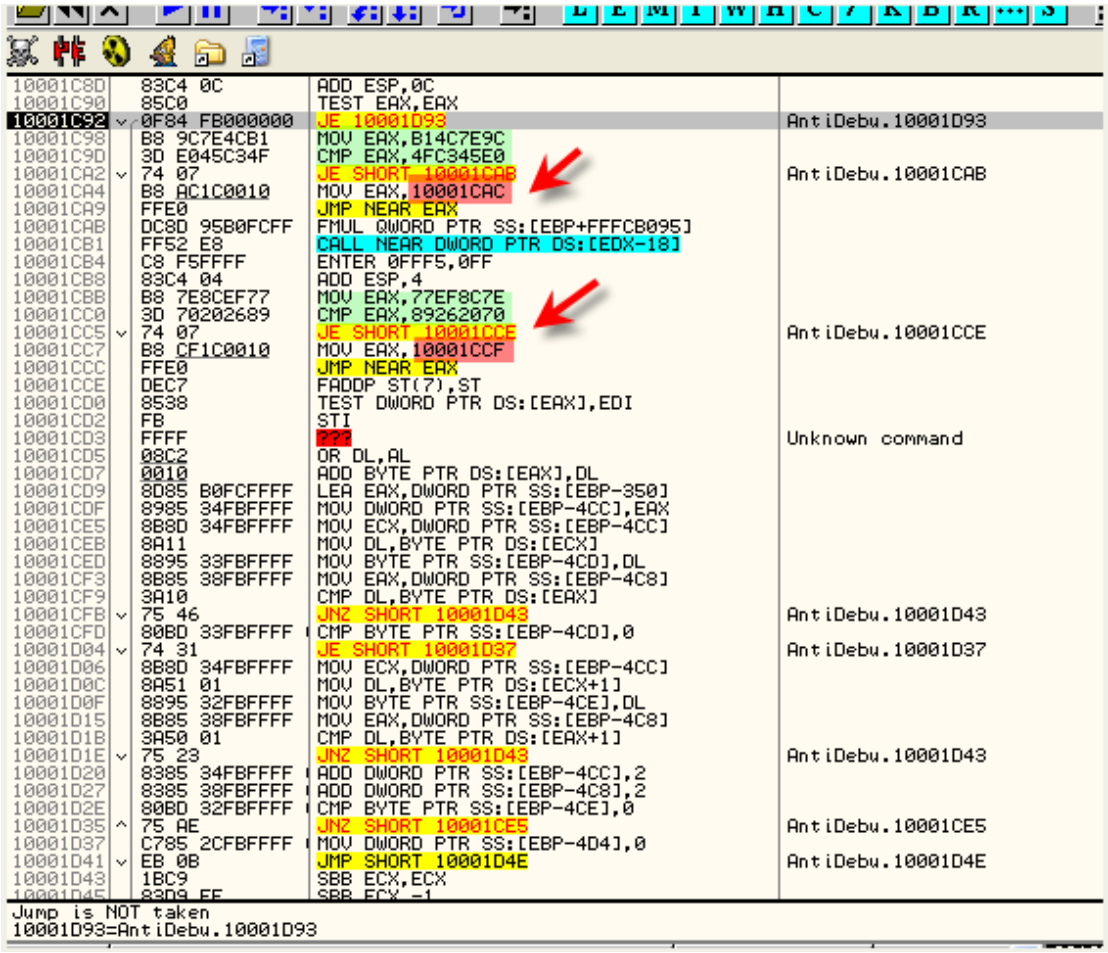
我们再次到了字符串比较函数处。这里两个字符串明显不相等。



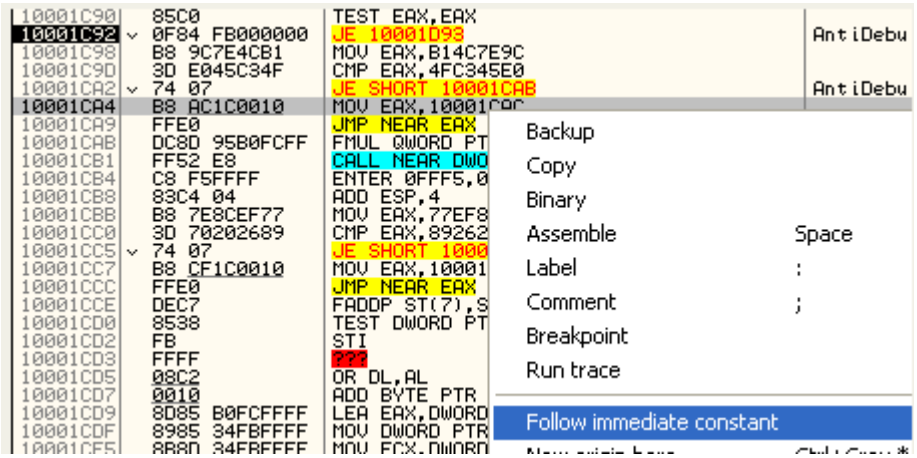
EAX 返回 1,说明比较的两个字符串不相等。下面的跳转不成立。



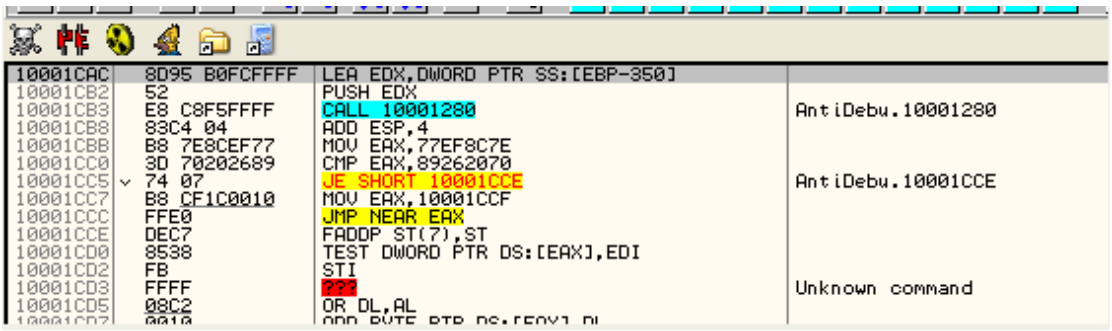
下面的代码是经过混淆了的。(PS:也就是常说的花指令)



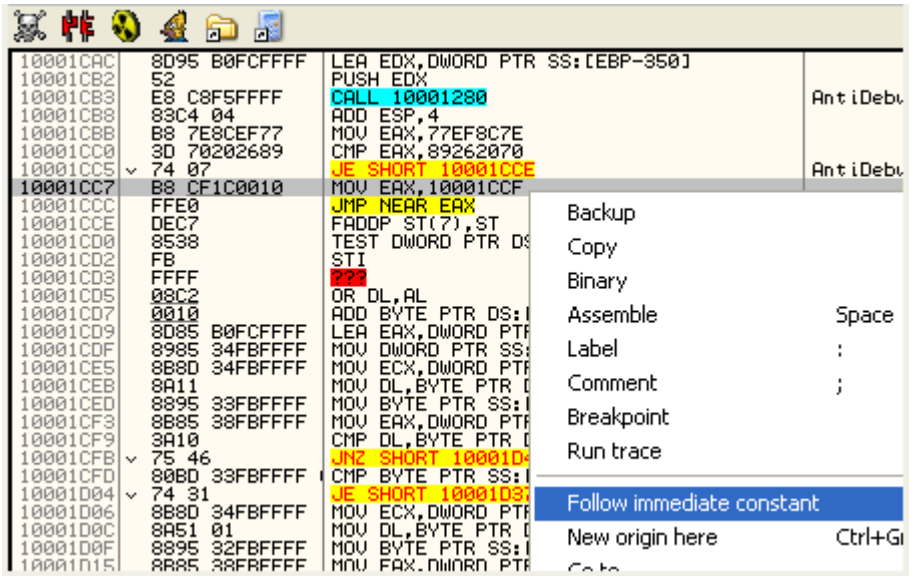
这里我们可以看到,首先将一个常量赋值给 EAX,然后与另一个常量进行比较,这里由于两个常量并不相等,所有下面 JE 条件跳转永远不会成立。接下来又是将一个常量赋值给 EAX,大家注意到了我标红了的地址没?紧接着下面通过 JMP NEAR EAX 来跳转的刚刚标红的常量所表示地址处。这里可以看到这部分代码被混淆了,我们如何看到定位其真正要执行的代码处呢?我们选中标红了地址,单击鼠标右键选择 Follow immediate constant(跟随立即数)。

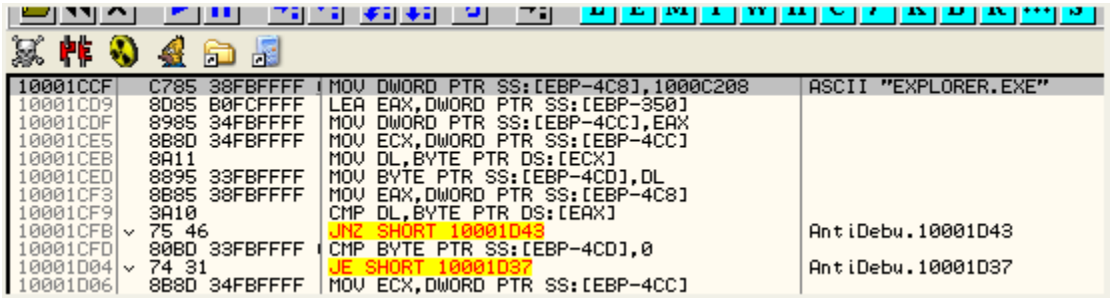


这样我们将能够不执行混淆过的代码,直接定位到实际的功能代码了。



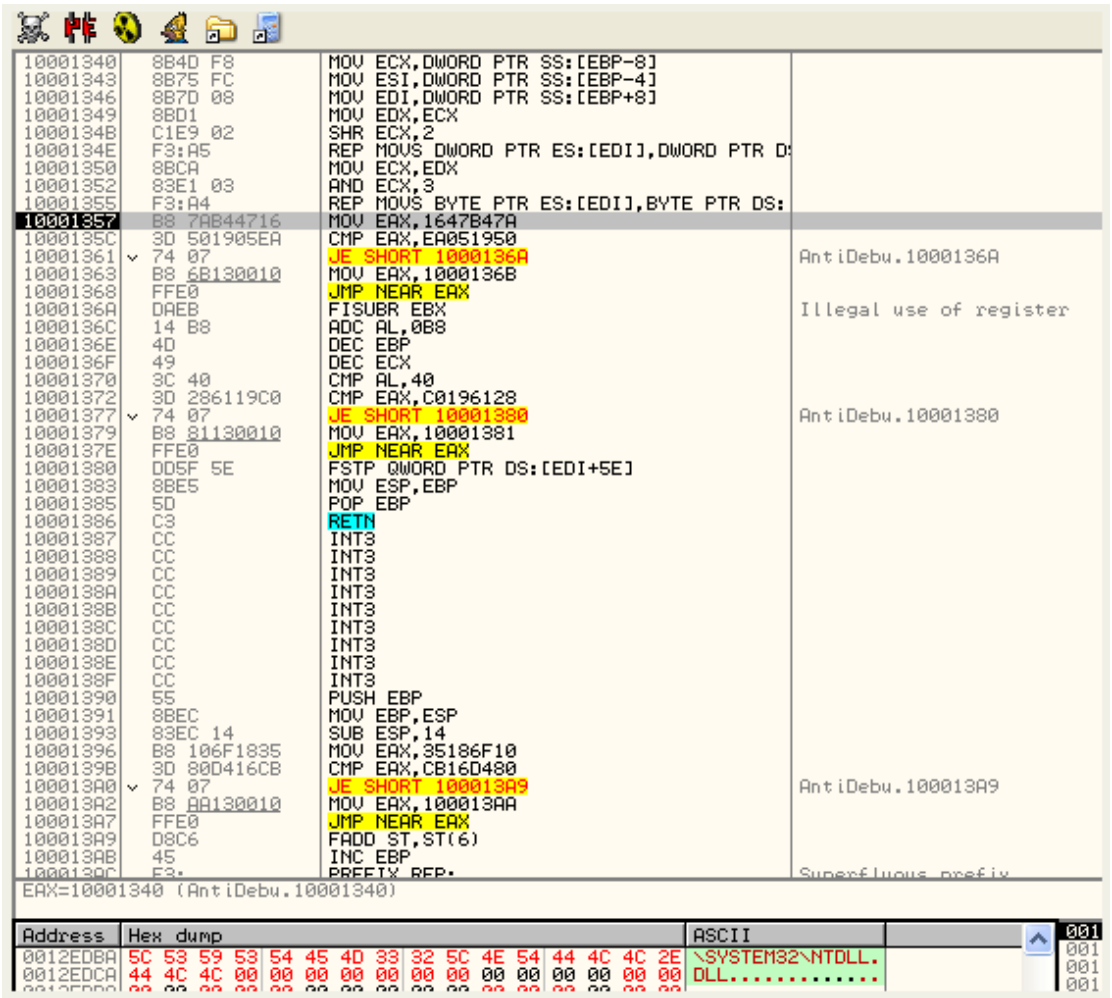
这里我们将就可以看到其实际要执行的代码了。下一个 MOV 指令处同样通过这样方式来查看。



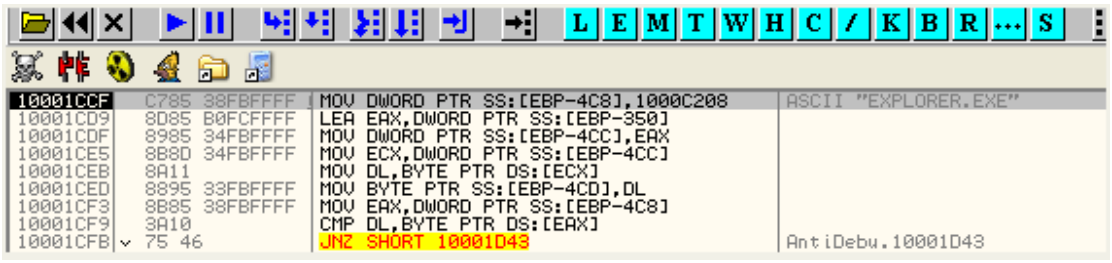


我们再次定位到了实际代码处,通过这样方式就可以不执行混淆过的代码直接定位实际代码了,比较方便。

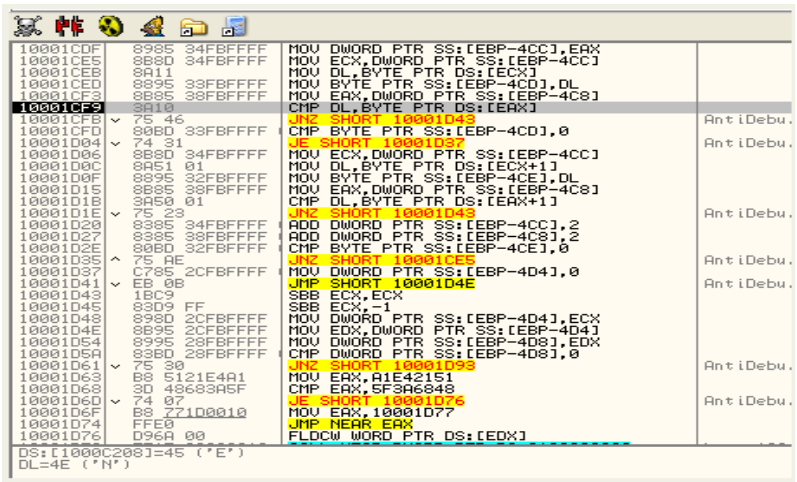
我们继续跟踪。



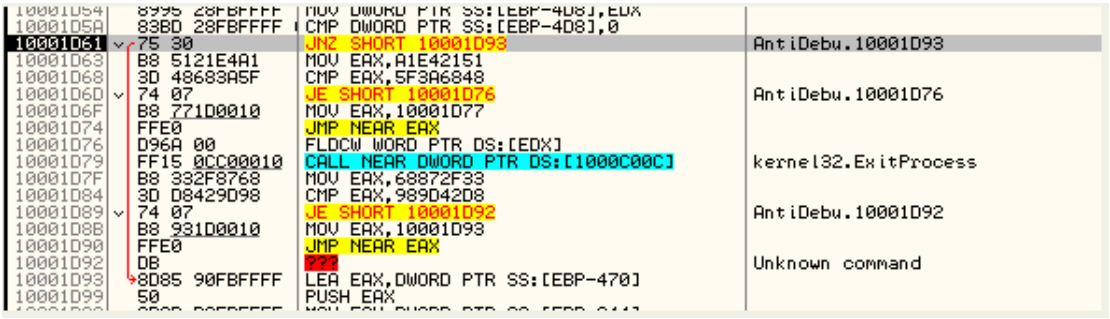
这里我们可以看到拼接字符串得到 C:\WINDOWS\SYSTEM32\NTDLL.DLL,我们继续。



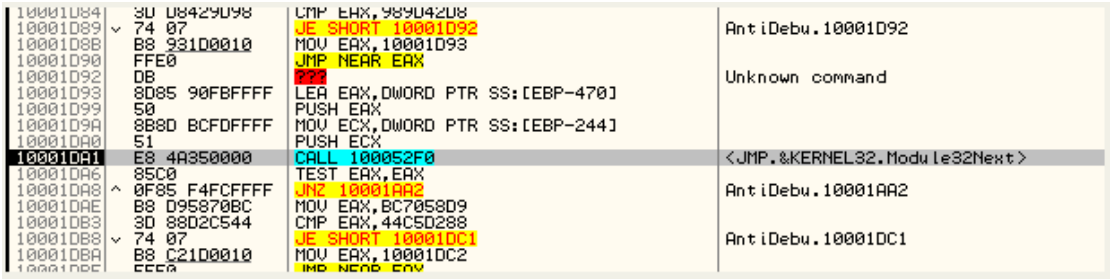
这里再次看到了 explorer.exe 这个名称。



下面就将 EXPLORER.EXE 的全路径与 NTDLL.DLL 的全路径进行比较。

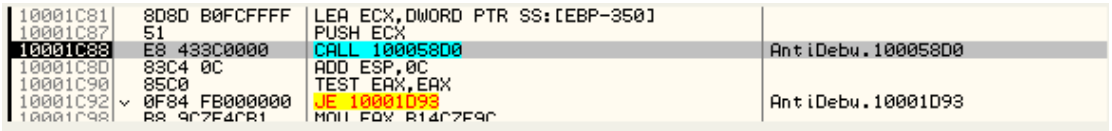


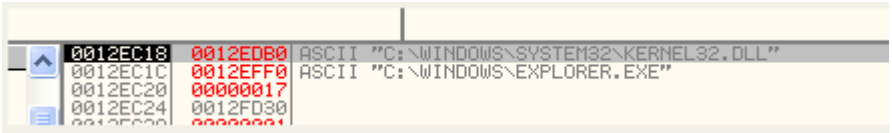
两者并不相等,并且会跳过了中间的 ExitProcess,继续。



我们看到这里继续遍历其他模块。说实话这部分我没看出有多大意义。

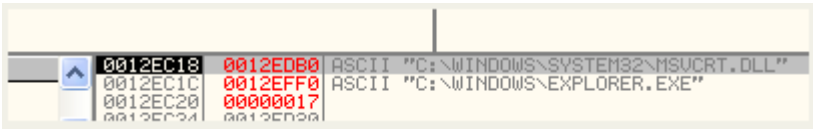
我们继续耐心跟吧。



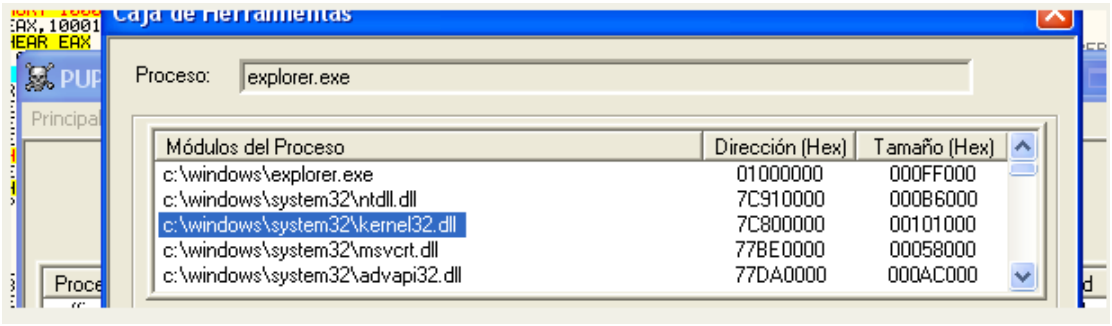


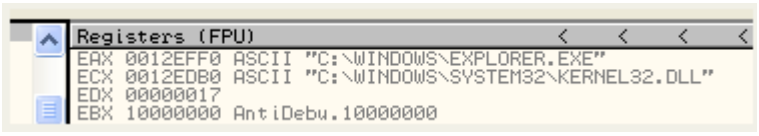
我们可以看到继续一个模块一个模块的遍历,都与 EXPLORER.EXE 的全路径进行比较。只有首次比较是相等的,其余的比较都不相等。

那么为了解决时间,我们直接给进行字符串比较的 CALL 处设置硬件执行断点。

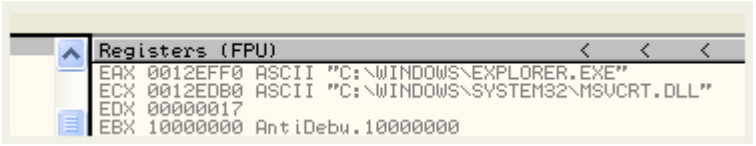


这里我们可以通过 PUPE 查看 explorer.exe 进程的模块列表信息对照的看。



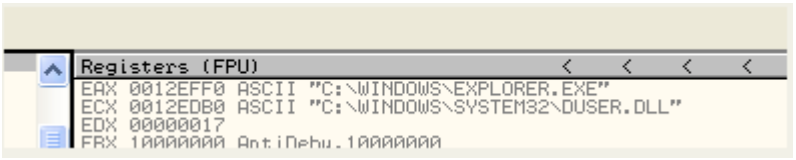


我们只需要定位最后一个模块即可。

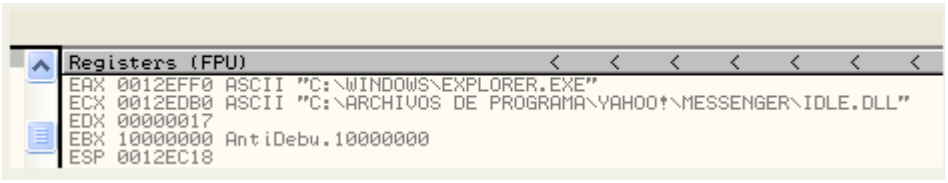


我这里最后一个模块是 idle.dll。

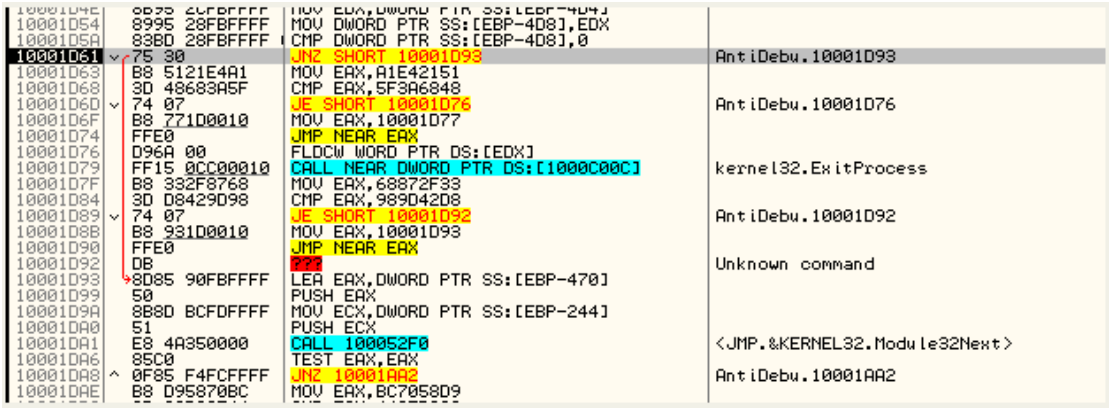
不符合的我们直接放过。



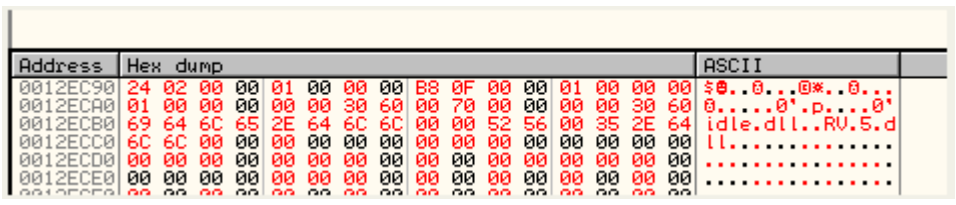
还没有到,我们再继续。

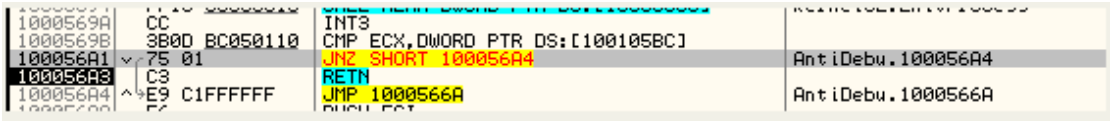


这里我们到了最后一个 DLL 了,我们继续跟踪看看会发生什么。

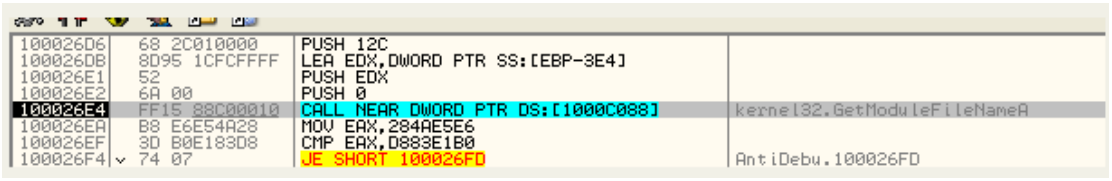


这里依然跳过了 ExitProcess,再次调用 Module32Next。

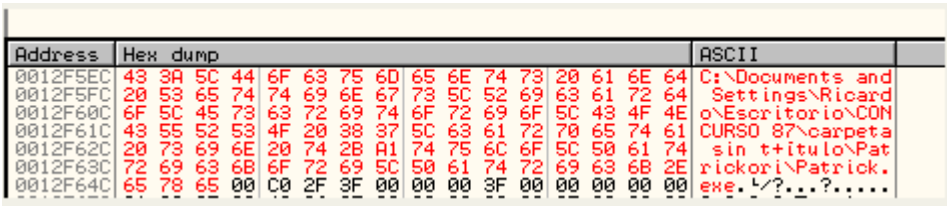




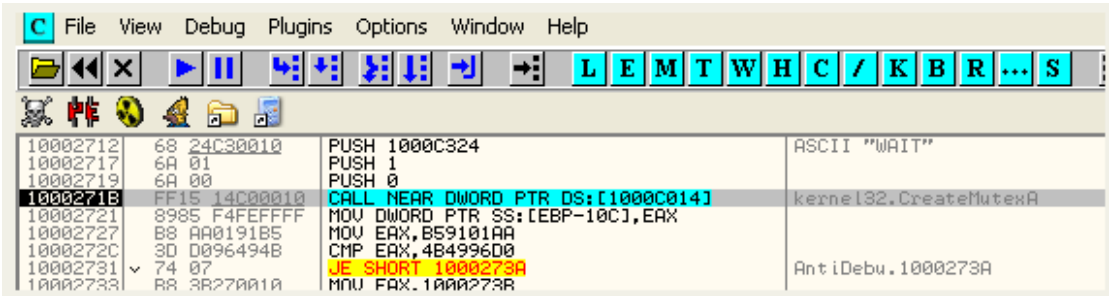
我们可以看到这里 ExitProcess 又被跳过了。继续往下跟。



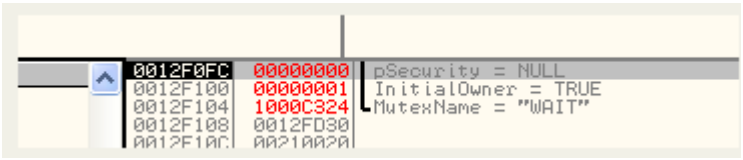
这里调用 GetModuleFileNameA。



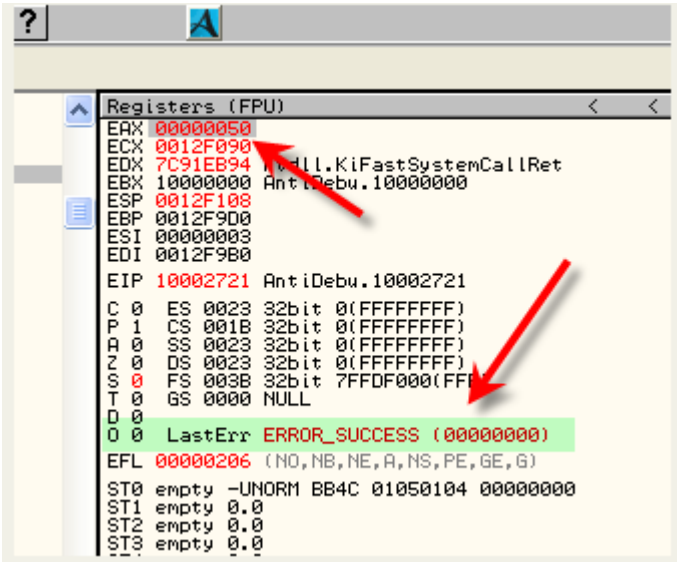
这里就得到了该 CrackMe 的全路径。



接下来就到了 CreateMutexA 的调用处,该函数用于创建互斥体,可用于防止多个实例运行。(PS:最简单的防多开)

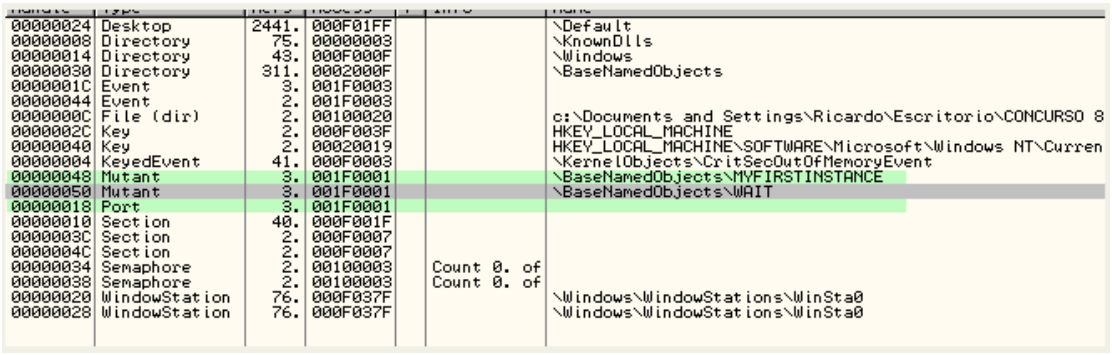


这里是该函数的参数。

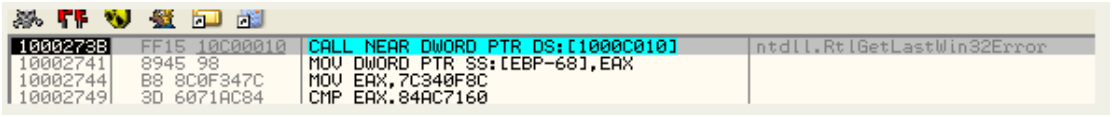


这里我们直接按 F8 执行该函数就得到了句柄 50,OD 中 LastErr(PS:也就是调用 GetLastError 得到的值)的返回值为 ERROR_SUCCESS,也就是创建互斥体成功。

我们单击工具栏中的 H 按钮查看句柄列表窗口,可以看到其中有两个互斥体的句柄,一个叫做 MYFIRSTINSTANCE,另一个叫做 WAIT。

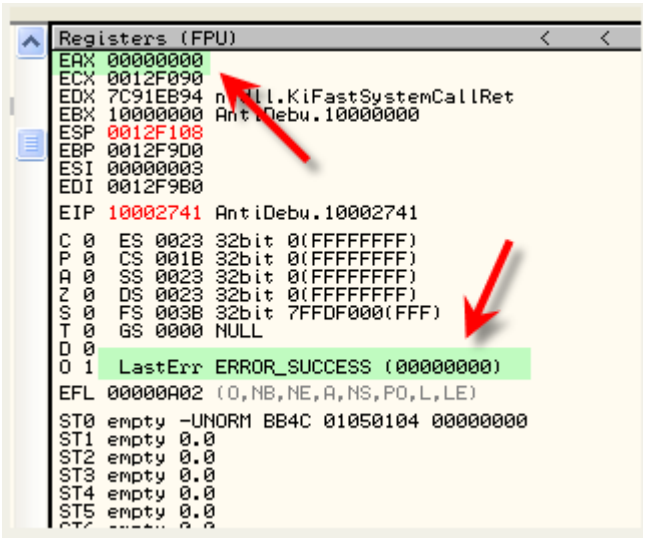


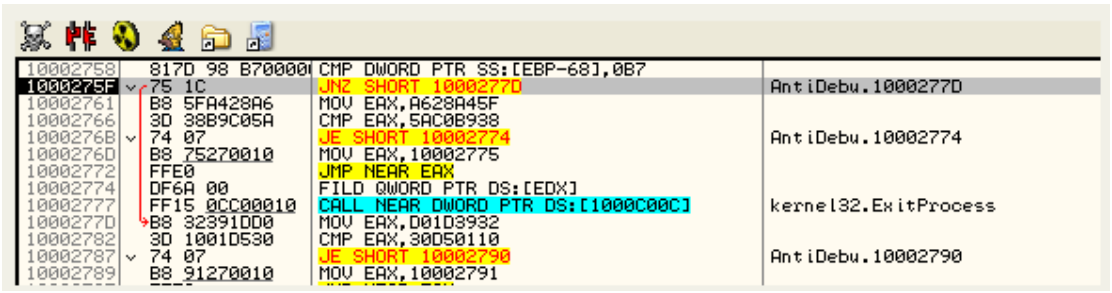
MYFIRSTINSTANCE 这个互斥体是之前创建的,这里我就不跟踪了。



接下来调用 RtlGetLastWin32Error 这个函数获取 API 函数执行的错误码,这里我们可以对照着 OD 来看。

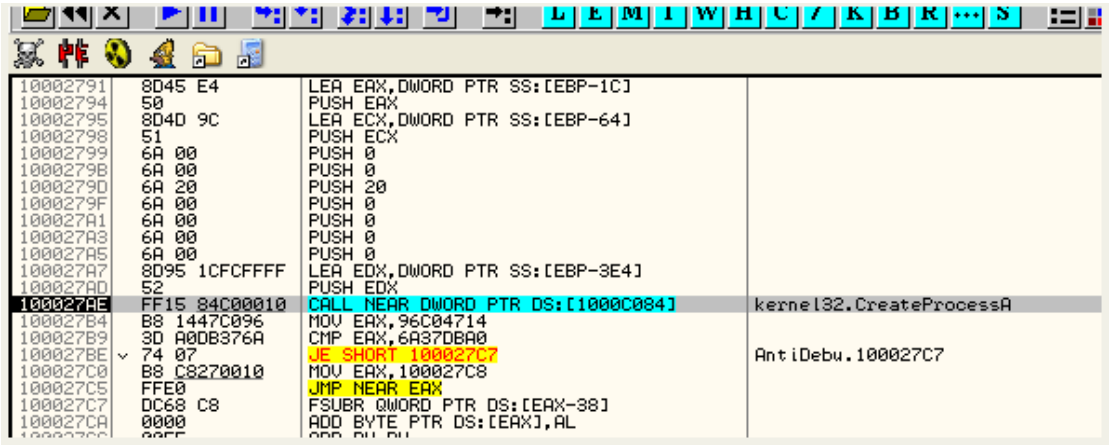
这里 EAX 的值为 0,也就是 OD 中显示的 LastErr 的值。



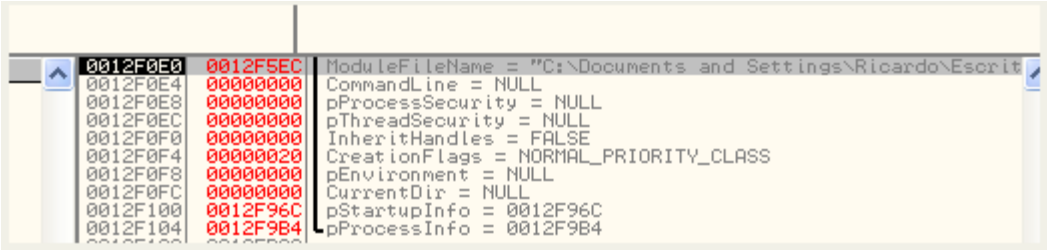


这里我们可以看到这个 WAIT 互斥体是第一次创建,所以跳过了 ExitProcess,如果 WAIT 不是第一次创建的话,那么就会调用 ExitProcess 结束进程。这样就可以防止多个程序实例运行。

继续。



这里可以看到该程序将调用 CreateProcessA 创建第二个进程。



这里我们可以看到该函数的参数,如果我们按 F8 键执行该函数的话,那么第二个进程将会被创建,但是 WAIT 互斥体已经存在了,所以被创建的进程会检测是否存在 WAIT 这个互斥体,显示这个互斥体是存在的,所有程序将调用 ExitProcess 退出。

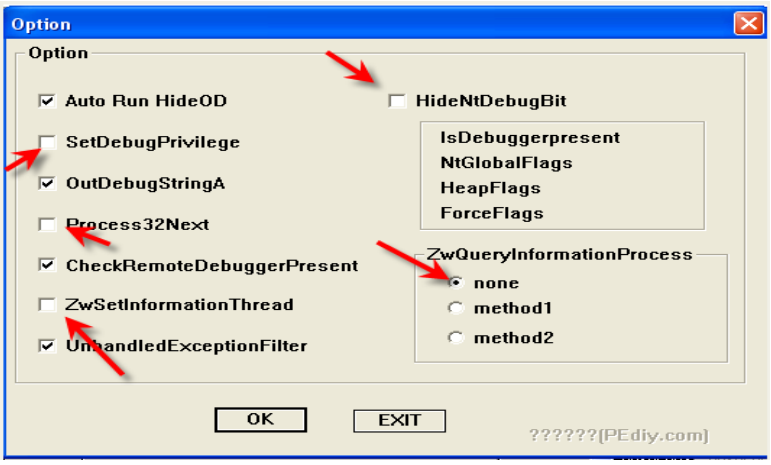
这里如果我们按 F8 键创建该进程的话,就算被创建了,该进程也会立即结束掉,所以这里我们可以尝试将进程挂起,我们可以通过修改 CreateFlags 这个参数将进程挂起,这里我们将 CreateFlags 的值修改 0x4。

我们来看看会发生什么。

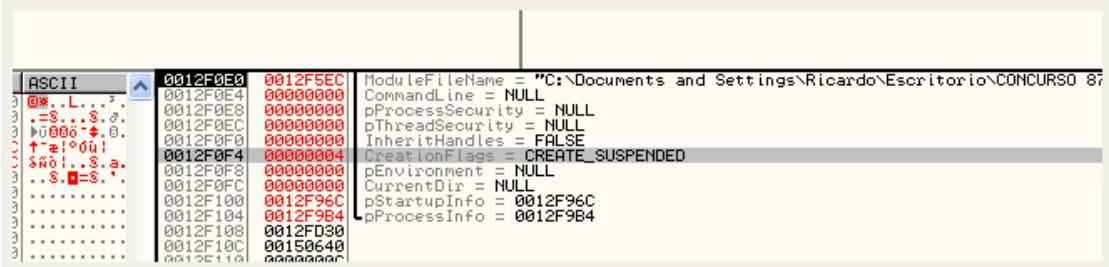
这里我们按 F8 键执行该函数,EAX 返回的是 0,也就是说创建进程失败,这是为什么呢?

好,我们直接给 CreateProcessA 这个函数调用处设置硬件执行断点,来看看为什么创建进程失败。也就是说此时有两个硬件访问断点,一个是父进程 PID 与 explorer.exe PID 进行比较处,另一个就是该 CreateProcessA 的调用处。

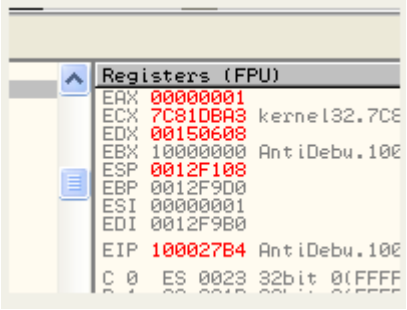
这里有可能是 HideOD 插件的影响,可能是该插件某些与权限相关的选项导致不能创建其他进程,我们尝试去掉这些选项。



这里去掉上图中的这些与权限有关的选项以后,接着将 CreatetionFlags 标志修改为 0x4。

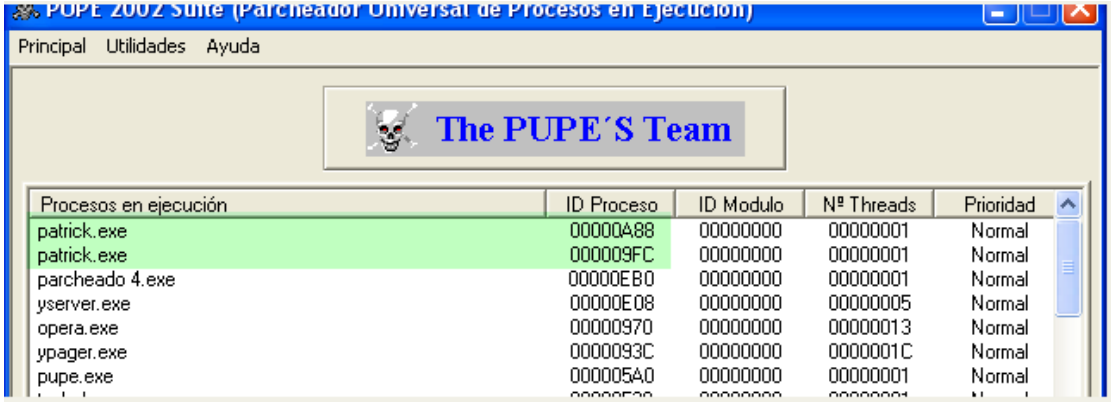


我们按 F8 键创建进程,我们可以看到 EAX 的值为 1,说明创建进程成功,证明我们的猜想是正确的。



好,现在我们再将 HideOD 的反反调试选项都勾选上。

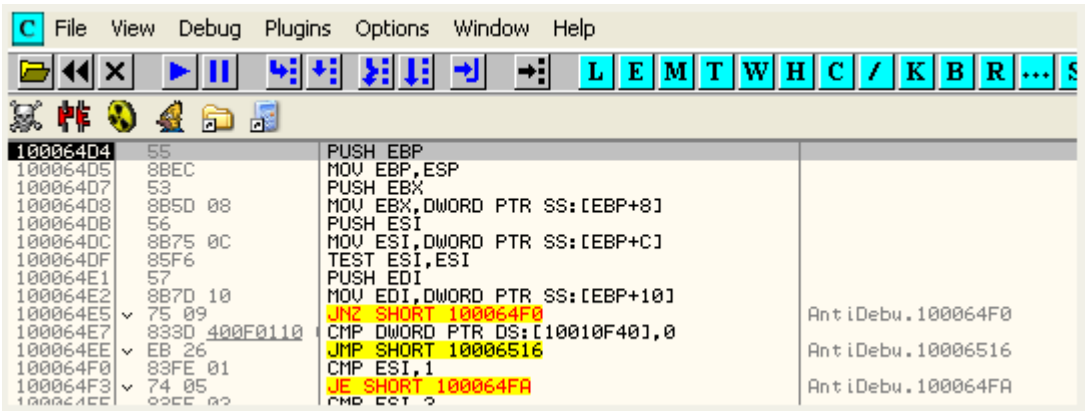
我们用 PUPE 查看一下进程。



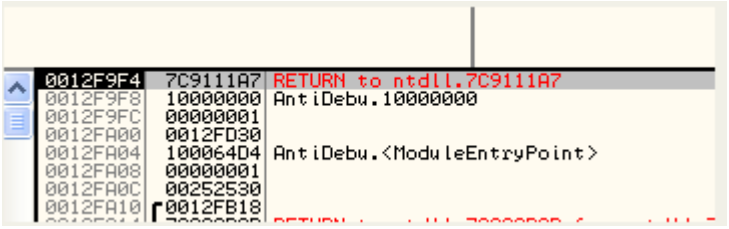
这里我们可以看到有两个 patrick.exe,第二个 patrick.exe 是挂起的,嘿嘿。

现在有个比较棘手的问题,就是如何附加以及中断新创建的进程呢?只有附加了新的进程,我们才能进一步跟踪新进程中是如何检测多开的。

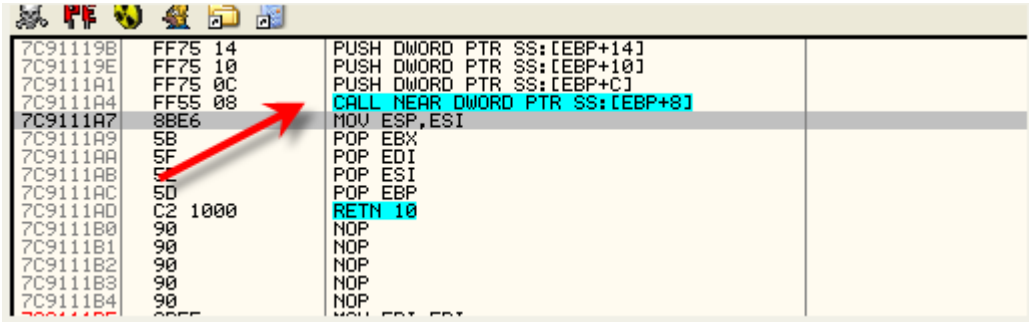
我们可以通过以下方式:重启 OD,其断在系统断点处,我们给 AntiDebugDll.dll 这个模块的代码段设置内存访问断点,我们运行起来,就会断在该 DLL 的入口点处。



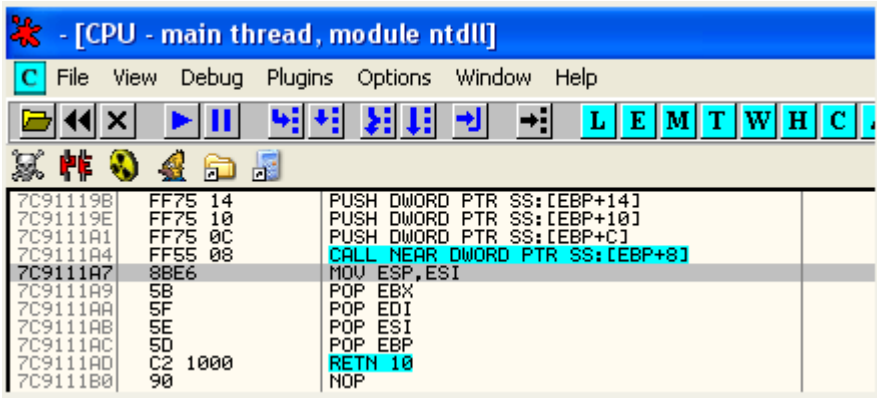
现在我们需要寻找一个点,这个点需要满足如下条件:新进程被创建,AntiDebugDll.dll 的代码还未执行。我们可以选一个调用 AntiDebugDll.dll 入口点之前的地址,然后让其一直在该地址处循环,不往下执行。我们来看一看此时的堆栈情况。



这里我们可以看到返回地址为 7C9111A7,我们定位到该返回地址处。

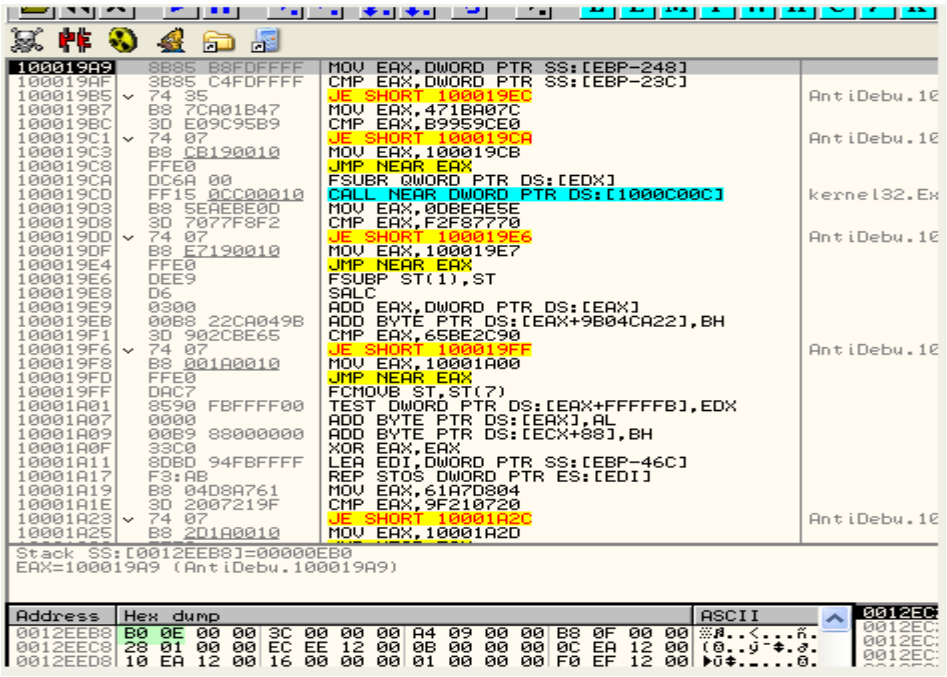


这里 7C9111A4 这个 CALL 就是跳往 AntiDebugDll.dll 的入口点,当前程序的领空属于 NTDLL.DLL,也就是说当新创建的进程由挂起状态恢复为运行状态后,就会运行到该 CALL 处,我们要想办法让其在这里一直循环而不进去。



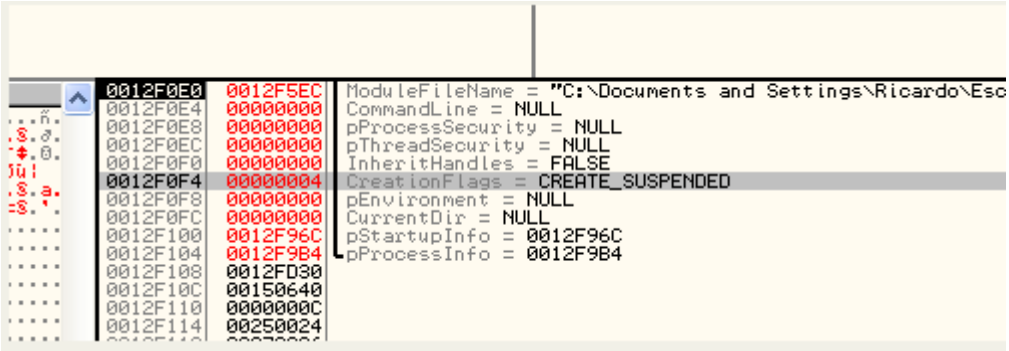
这里我们需要做的就是替换 7C9111A4 地址处的原始字节,该地址处的原始字节为 FF55。

好了思路就是这样的啊,我们之前的两个硬件断点还没有删除吧?我们删除掉内存访问断点,接着直接运行起来,断在了 PID 的比较处。



这里我们将父进程 PID 修改为 explorer.exe 的 PID,继续运行

随即就断在了 CreateProcessA 的调用处。

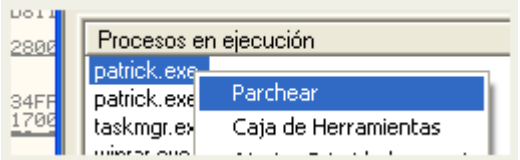


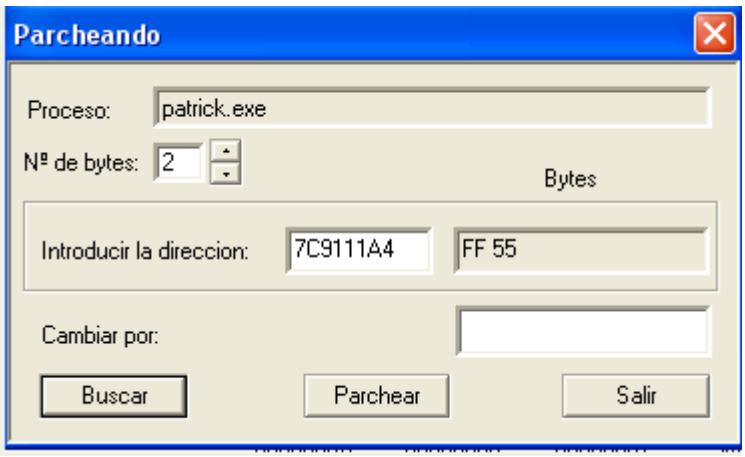
这里我们将 CreateFlags 的值修改为 0x4。

按 F8 键,这样新进程就创建了。

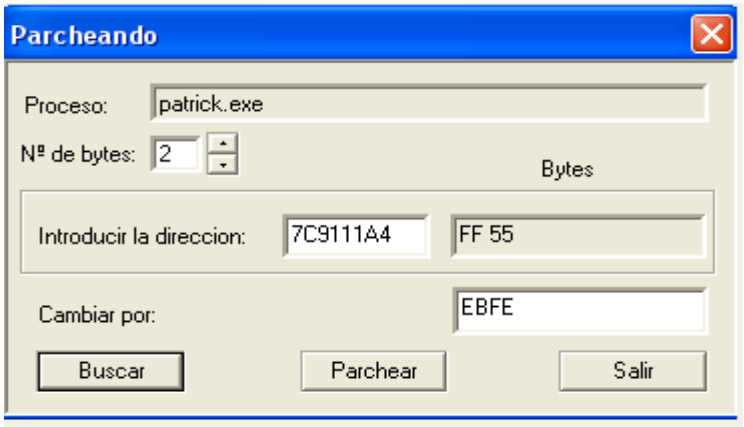
这里我们可以在 PUPE 中看到这两个进程。

这里我们选择新创建的进程进行 Patch,我们选中新创建的进程,单击鼠标右键选择 Parchear(Patch)按钮。



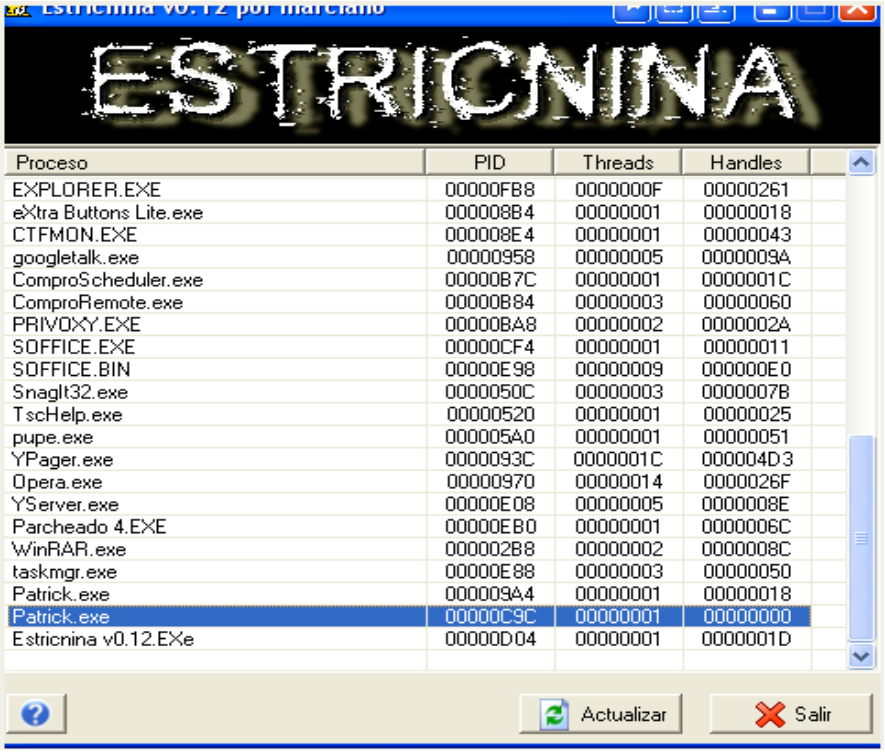


这里我们将 7C9111A4 地址处的前两个字节修改为 EB FE。

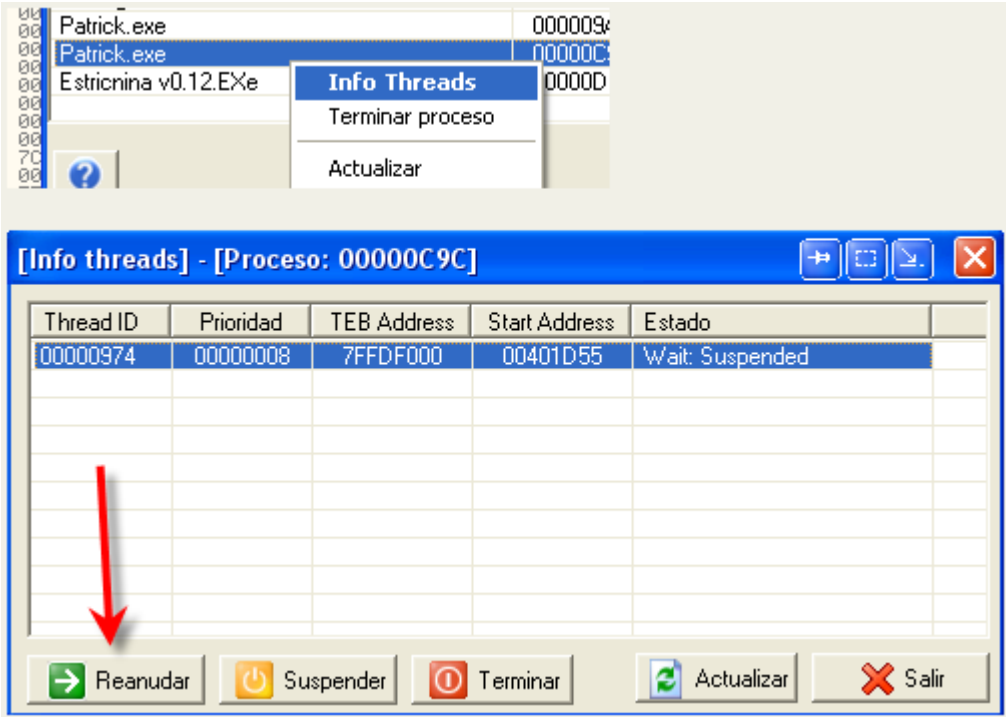


这里我们填写上 7C9111A4(PS:注意你本机的地址),单击 Buscar(获取原始字节)按钮,接着填写上要替换的字节码 EBFE,最后单击 PARCHEAR(Patch)按钮。这样就 Patch 完毕了。现在我们需要将该进程由挂起状态恢复到运行状态。

下面我要用到 Estricnina_v0.12 这款工具。



这里我们定位到新创建的进程,单击鼠标右键选择 Info Threads。

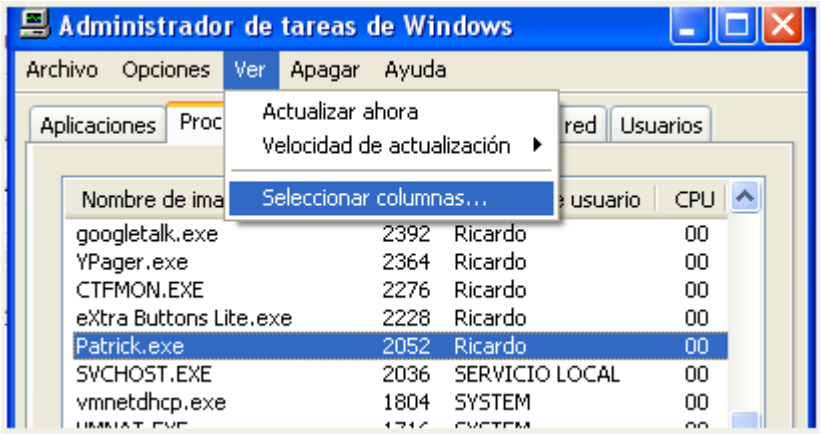


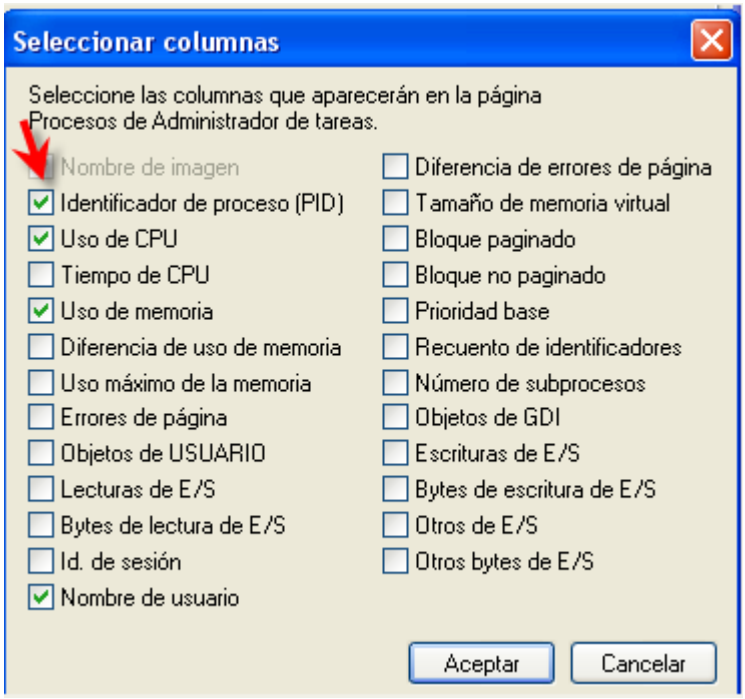
单击 Reanudar(Resume)按钮。

这样挂起的线程就被恢复了,并且一直在 7C9111A4 地址处循环。

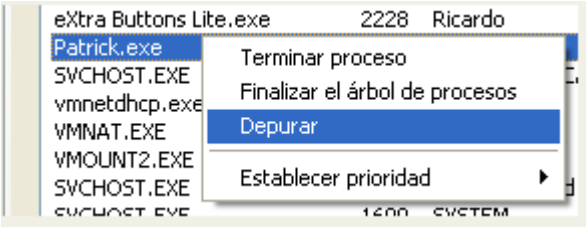
接下来我们来附加该进程。

首先我们打开任务管理器,将显示 PID 的列勾选上。

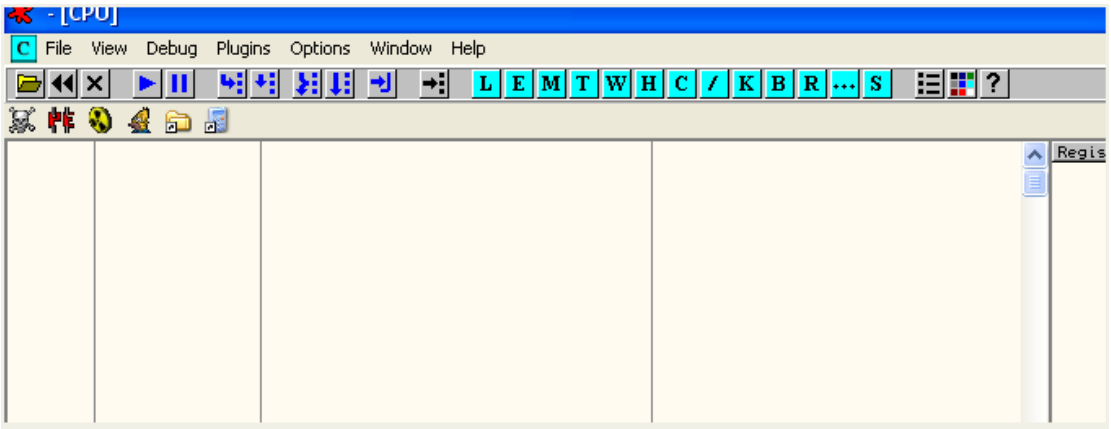




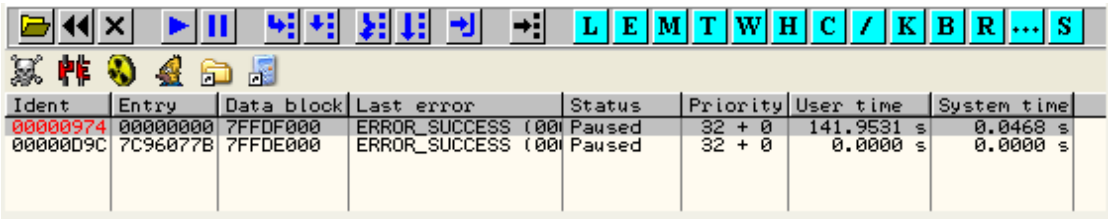
这里将任务管理器的 PID 选项勾选上,显示的是 10 进制,我们转换为 16 进制即可,这样我们就定位到新创建的进程了。



接下来我们将 OD 设置为即时调试器(JIT),我们选择新创建的 Patrick.exe 这个进程单击鼠标右键选择 Depurar(调试)。这样 OD 的即时调试功能就生效了。

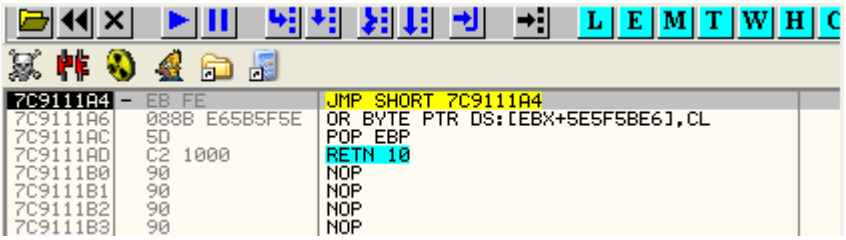


这里我们可以看到 OD 中是空白的,但是并没有发生警告,不用担心,我们直接单击工具栏中的 T 按钮查看下线程。



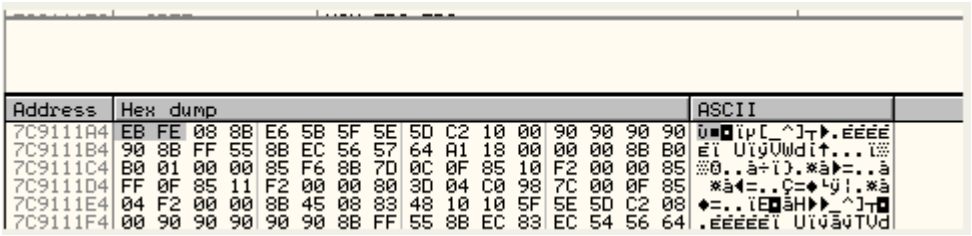
Ident	Entry	Data block	Last error	Status	Priority	User time	System time
00000974	00000000	7FFDF000	ERROR_SUCCESS (00(	Paused	32 + 0	141.9531 s	0.0468 s
00000D9C	7C96077B	7FFDE000	ERROR_SUCCESS (00(	Paused	32 + 0	0.0000 s	0.0000 s

这里我们可以看到有两个线程,其中一个是由于我们 Patch 的循环引起的,我们任选一个双击,如果不是循环,那么就证明是另一个。



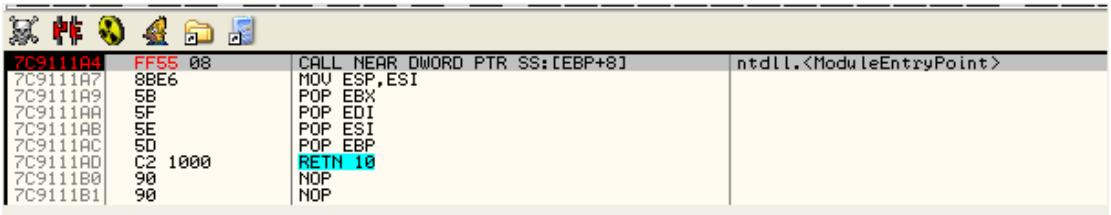
```
7C9111A4 - EB FE          JMP SHORT 7C9111A4
7C9111A6   088B E65B5F5E  OR BYTE PTR DS:[EBX+5E5F5BE6],CL
7C9111AC   5D             POP EBP
7C9111AD   C2 1000        RETN 10
7C9111B0   90             NOP
7C9111B1   90             NOP
7C9111B2   90             NOP
7C9111B3   90             NOP
```

我们可以看到该处处于循环中。



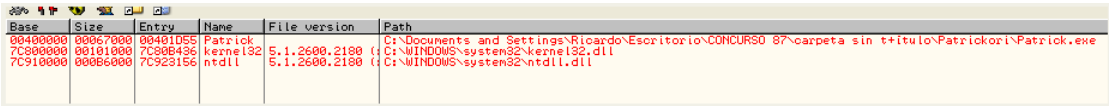
Address	Hex dump	ASCII
7C9111A4	EB FE 08 8B E6 5B 5F 5E 5D C2 10 00 90 90 90 90	
7C9111B4	90 8B FF 55 8B EC 56 57 64 A1 18 00 00 00 8B B0	
7C9111C4	B0 01 00 00 85 F6 8B 7D 0C 0F 85 10 F2 00 00 85	
7C9111D4	FF 0F 85 11 F2 00 00 80 3D 04 C0 98 7C 00 0F 85	
7C9111E4	04 F2 00 00 8B 45 08 83 48 10 10 5F 5E 5D C2 08	
7C9111F4	00 90 90 90 90 90 8B FF 55 8B EC 83 EC 54 56 64	

我们给 7C9111A4 这一行设置一个断点,接着将该地址处的前两个字节恢复为原始字节 FF55.



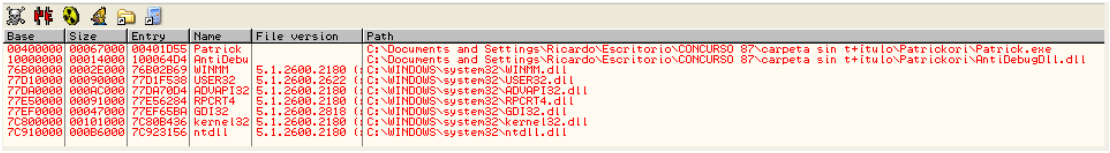
```
7C9111A4   FF55 08        CALL NEAR DWORD PTR SS:[EBP+8]   ntdll.<ModuleEntryPoint>
7C9111A7   8BE6           MOV ESP,ESI
7C9111A9   5B             POP EBX
7C9111AA   5F             POP EDI
7C9111AB   5E             POP ESI
7C9111AC   5D             POP EBP
7C9111AD   C2 1000        RETN 10
7C9111B0   90             NOP
7C9111B1   90             NOP
```

这样我们将附加成功了,我们可以查看下模块列表窗口。



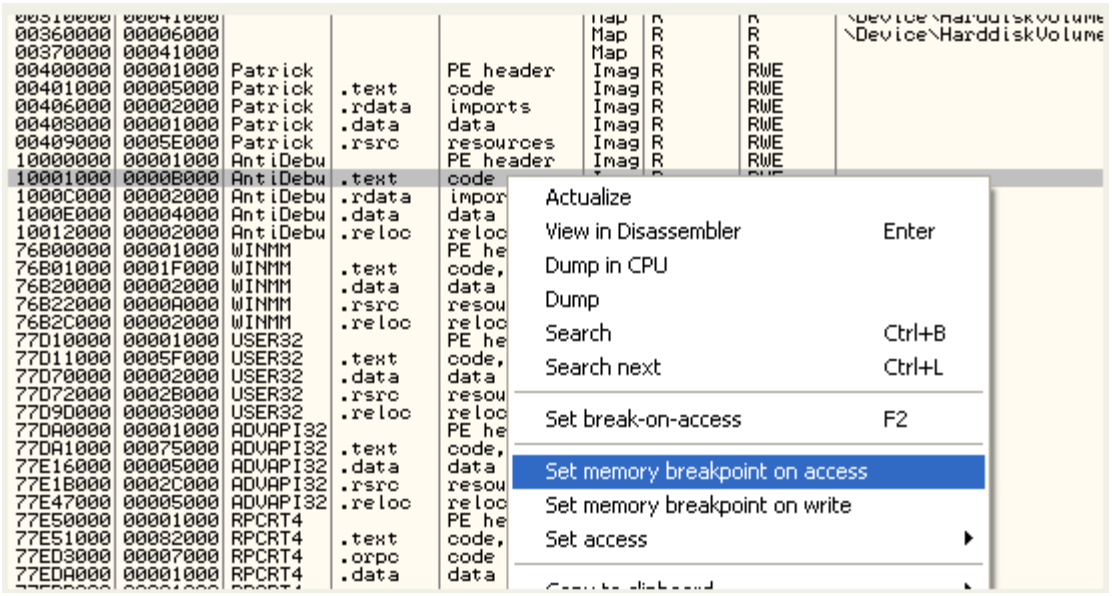
Base	Size	Entry	Name	File version	Path
00400000	00067000	00401D55	Patrick		C:\Documents and Settings\Ricardo\Escritorio\CONCURSO 87\carpeta sin t+itulo\Patrickori\Patrick.exe
7C800000	00101000	7C80B436	kernel32	5.1.2600.2180 (	C:\WINDOWS\system32\kernel32.dll
7C910000	000B6000	7C923156	ntdll	5.1.2600.2180 (	C:\WINDOWS\system32\ntdll.dll

此时并没有模块列表中并没有出现 AntiDebugDll.dll 这个模块,我们会断在 7C9111A4 处,如果还是没有出现 AntiDebugDll.dll,我继续运行,直到 AntiDebugDll.dll 这个模块为止。

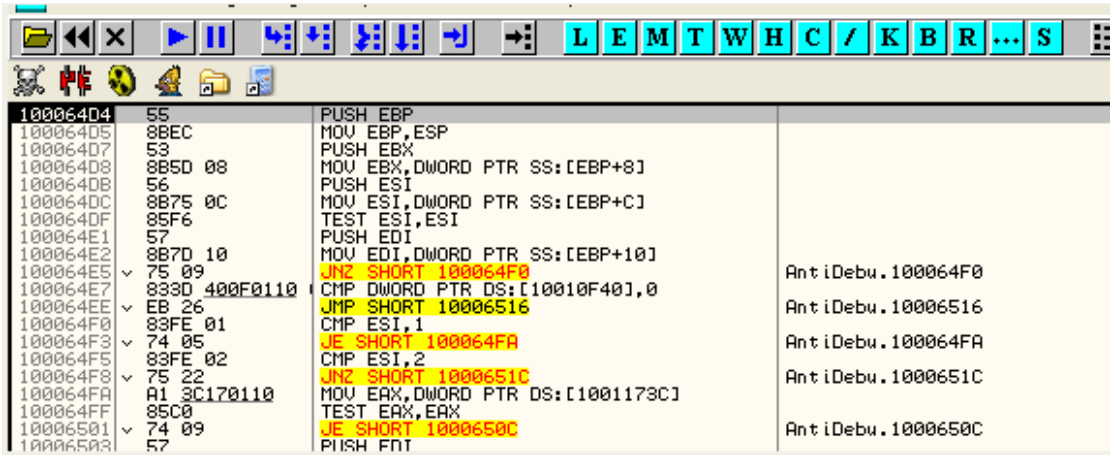


Base	Size	Entry	Name	File version	Path
00400000	00067000	00401D55	Patrick		C:\Documents and Settings\Ricardo\Escritorio\CONCURSO 87\carpeta sin t+itulo\Patrickori\Patrick.exe
10000000	00014000	100064D4	AntiDebu		C:\Documents and Settings\Ricardo\Escritorio\CONCURSO 87\carpeta sin t+itulo\Patrickori\AntiDebugDll.dll
76B00000	0002E000	76B02B69	WINMM	5.1.2600.2180 (	C:\WINDOWS\system32\WINMM.dll
77D10000	00090000	77D1F538	USER32	5.1.2600.2622 (	C:\WINDOWS\system32\USER32.dll
77DA0000	000AC000	77DA78D4	ADVAPI32	5.1.2600.2180 (	C:\WINDOWS\system32\ADVAPI32.dll
77E50000	00091000	77E56284	RPCRT4	5.1.2600.2180 (	C:\WINDOWS\system32\RPCRT4.dll
77EF0000	00047000	77EF65BA	GDI32	5.1.2600.2818 (	C:\WINDOWS\system32\GDI32.dll
7C800000	00101000	7C80B436	kernel32	5.1.2600.2180 (	C:\WINDOWS\system32\kernel32.dll
7C910000	000B6000	7C923156	ntdll	5.1.2600.2180 (	C:\WINDOWS\system32\ntdll.dll

现在我们可以看到模块列表中出现了 AntiDebugDll.dll,好,现在我们对其代码段设置内存访问断点。

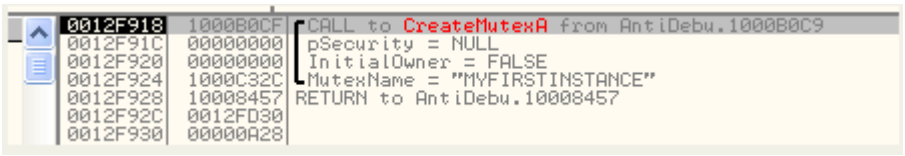


现在删除掉 7C9111A4 处的断点,运行起来。

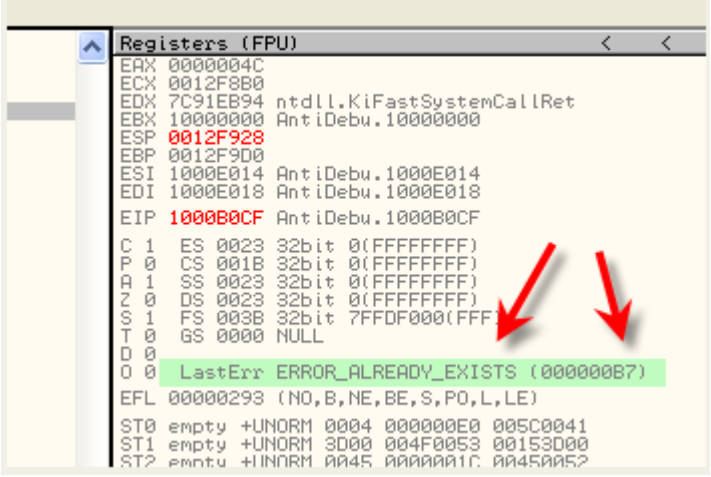


这里我们就断在了 AntiDebugDll.dll 的入口点处了。下面我们给 CreateMutexA 设置一个断点。

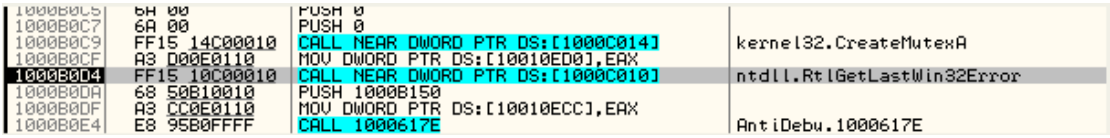
运行起来。



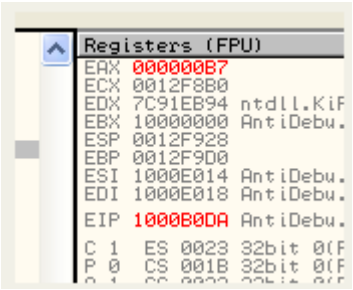
这里我们可以看到将要创建的互斥体为 MYFIRSTINSTANCE,这里由于该互斥体已经存在了,所以我们直接运行到函数返回处。



这里我们可以看到 LastErr 显示为 0xB7,即 ERROR_ALREADY_EXISTS,表示互斥体已经存在。



接下来这里就调用 GetLastError 这个 API 函数获取错误码,判断错误码是否为 0xB7。



如果为 0xB7 的话表示互斥体已经存在,该进程是并非第一次创建,如果不为 0xB7 的话,就表示该进程是首次创建。这里错误码为 0xB7,表示该进程并非首次创建。

好了,本章就这里。