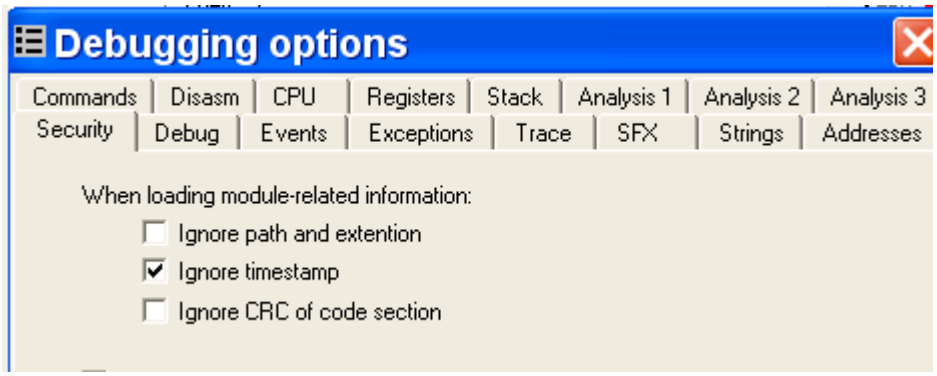
第二十章-OllyDbg 反调试之检测 OD 进程名

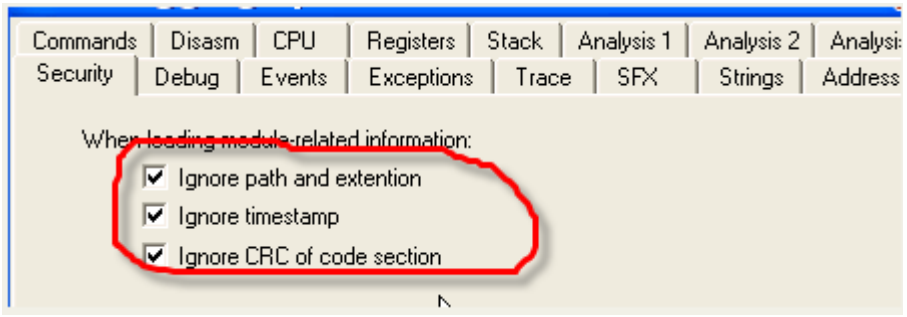
本章我们介绍通过查找 OD 进程名来进行反调试技巧,首先我们有必要对 OD 进行相应的设置。

我们选择主菜单项 Option-Debugging options-Security。



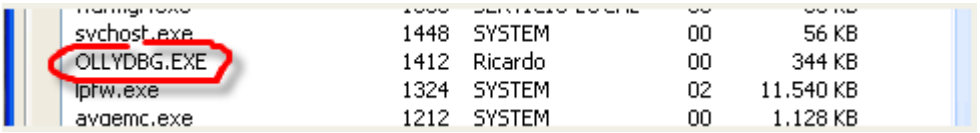
我们可以看到这里有 3 个复选框,我们知道通常情况下,我们给某个 API 函数设置断点,当我们重启 OD 后,刚刚给 API 函数设置的断点就被清除了。但是勾选上这 3 个复选框,再给 API 函数设置断点,重启 OD 后我们会发现刚刚设置的断点依然存在,这就避免了重启 OD 后需要重新给 API 设置断点的麻烦。

实际上,我也不知道 OD 关于这个功能的实现原理,我们只需要知道这 3 个复选框可以让我们设置的 API 断点在 OD 重启后仍然有效。



这个设置我觉得很有必要,避免了很多麻烦,现在我们开始讨论通过查找 OD 进程名来进行反调试的话题吧。

打开 OD,接着通过 Ctrl+ Alt + Delete 快捷键打开任务管理器。

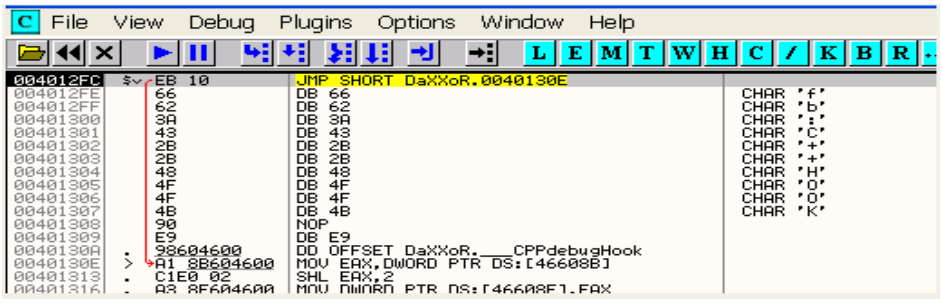


我们可以看到进程名称列表,我们可以通过检测进程名称是否为 OllyDbg,如果是就结束进程,哈哈。

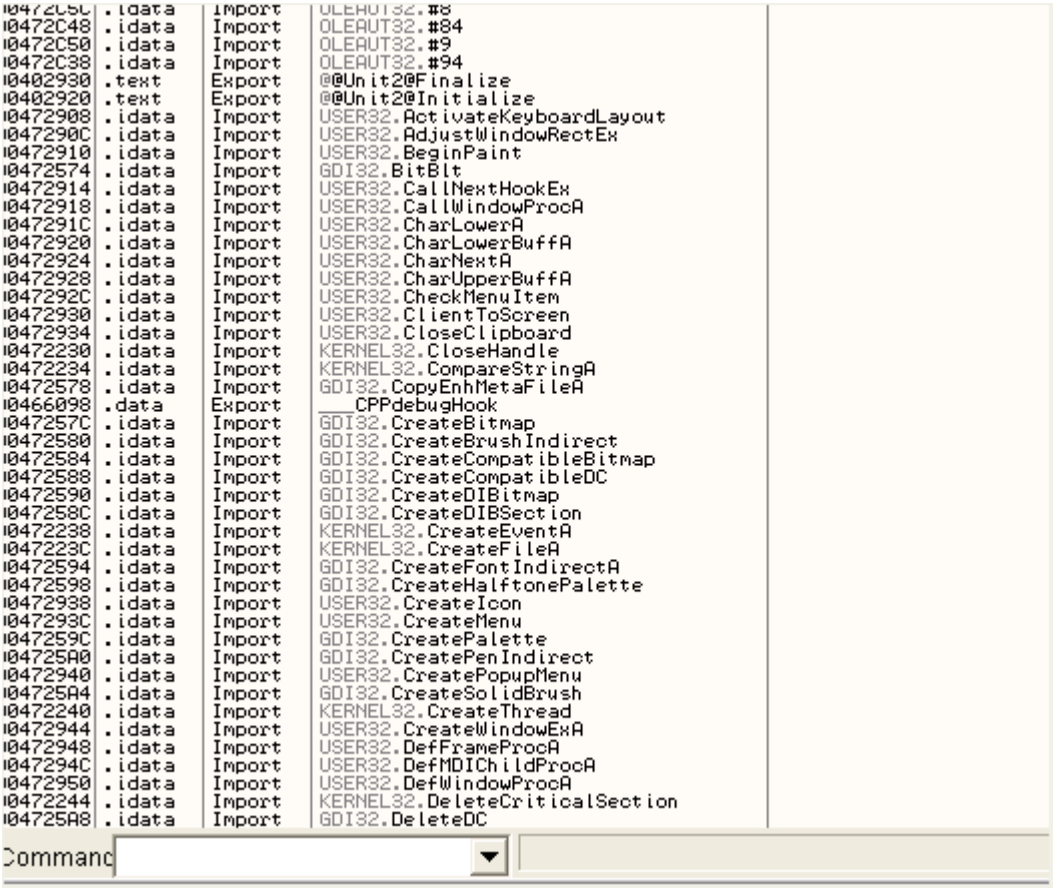
我们来看一个 CrackMe,这个 CrackMe 运用我们前面介绍的知识暂时还解决不了,我们只是来看看它是如何检测 OllyDbg 的进程名,以及我们如何来绕过它的检测,嘿嘿。

这个 CrackMe 的名字叫做 daxxor,如果我们先运行这个 CrackMe,然后打开 OD,会发现 OD 马上就退出了。如果我们用 OD 加载该 CrackMe 然后运行起来,会发生两者一起退出了。

如何来应对这种情况呢,我们先用 OD 加载这个 CrackMe。

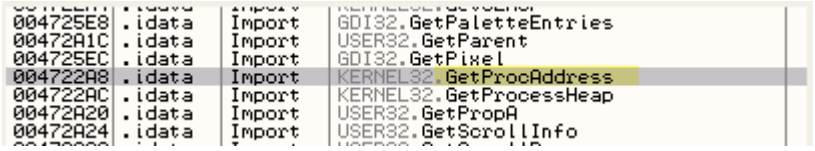


停在了入口点处,我们先看看该程序使用了哪些 API 函数。

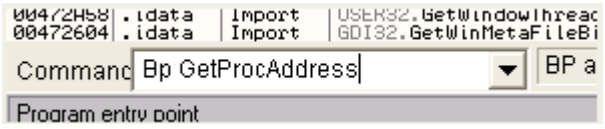


我们可以看到有很多 API 函数,但都不是用于检测进程名的,可能这些重要的 API 函数被隐藏起来了,没有出现该列表中。

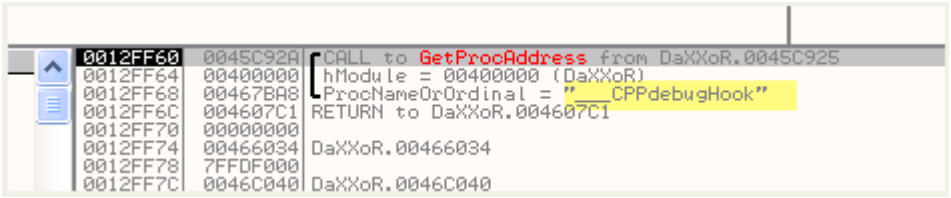
很显然如果程序不直接导入某些 API 的话,会使用 GetProcAddress 这个 API 函数来获取这些 API 函数的地址进行间接调用。



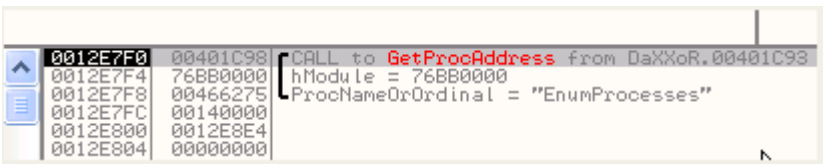
使用 GetProcAddress 函数加载的一些 API 函数并不会出现该 API 函数列表中,我们给 GetProcAddress 设置一个断点。



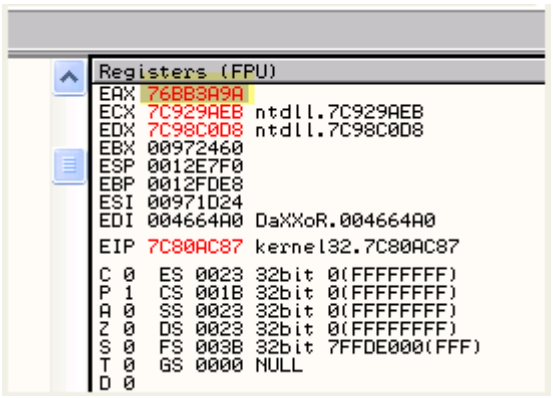
运行起来。



断在了 GetProcAddress 的入口点处,当前待获取的函数是__CPPdebugHook,该函数检测进程名没有关系,继续运行。

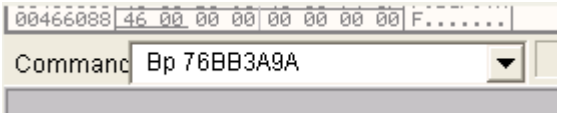


我们继续按 F9 键运行直到待获取的 API 是与检测进程名相关的为止,这里待获取的 API 函数是 EnumProcesses,这里通过选择主菜单项 Debug-Execute till return 执行到返回,这个时候 EAX 寄存器中保存的就是 EnumProcesses 这个 API 函数的地址了。我们接着就可以给该地址设置断点了。

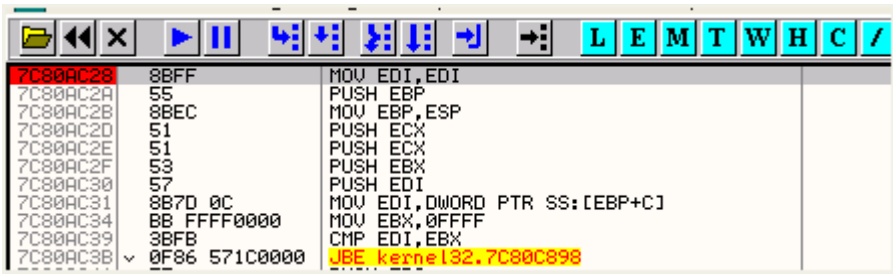


这里 EAX 就保存了 EnumProcesses 的函数地址,我的机器上是 76BB3A9A(可能与你机器上的不一样)。

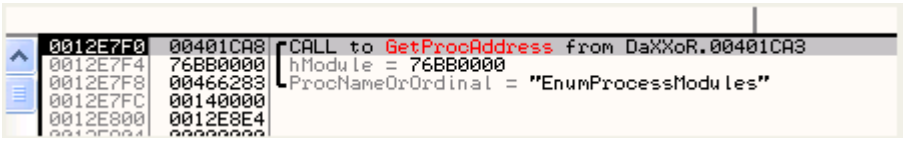
另外,OD 的 API 的函数列表中并没有列出 EnumProcesses 这个名称,所以我们直接 bp EnumProcesses 是设置不了断点的,我们可以给 EnumProcesses 函数的地址设置断点。



设置成功了。

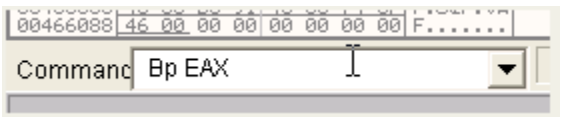


好了,我们已经给 EnumProcesses 设置了断点,继续我们刚才的步骤,看看还有什么 API 函数被加载。

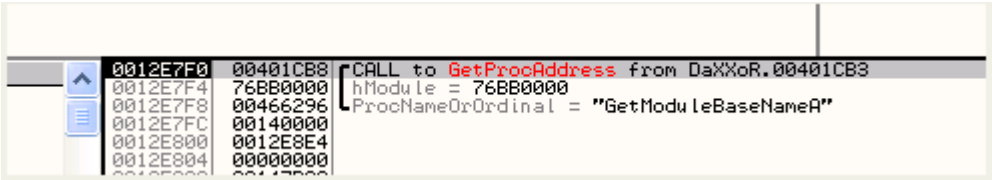


这里是获取枚举进程模块函数的地址,我们还是执行到返回,接着给 EAX 中保存的地址设置断点。

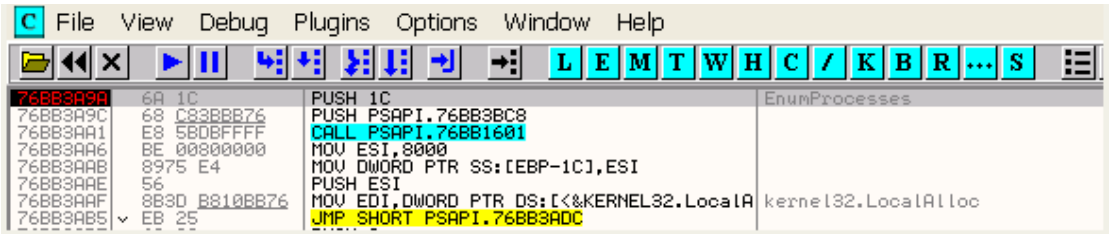
我们直接在命令栏中输入 bp EAX。



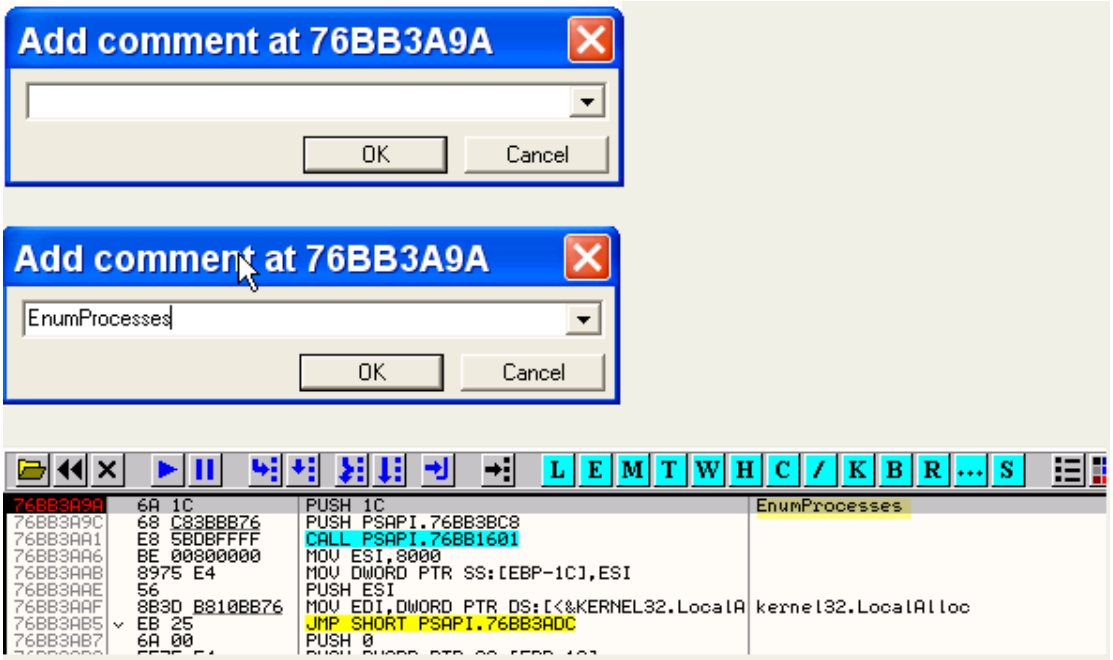
同理,我们给所有与检测进程名相关的 API 函数设置断点。



这里是另外一个可疑的 API 函数 GetModuleBaseNameA,我们跟之前一样给该函数设置断点,然后我们运行起来,断在了 EnumProcesses 函数的入口处。

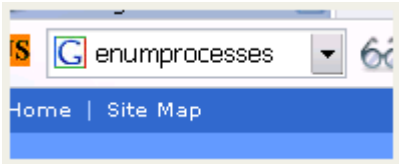


我们通过双击反汇编窗口的注释区域来给该函数添加注释,注释上该函数的名称 EnumProcesses。



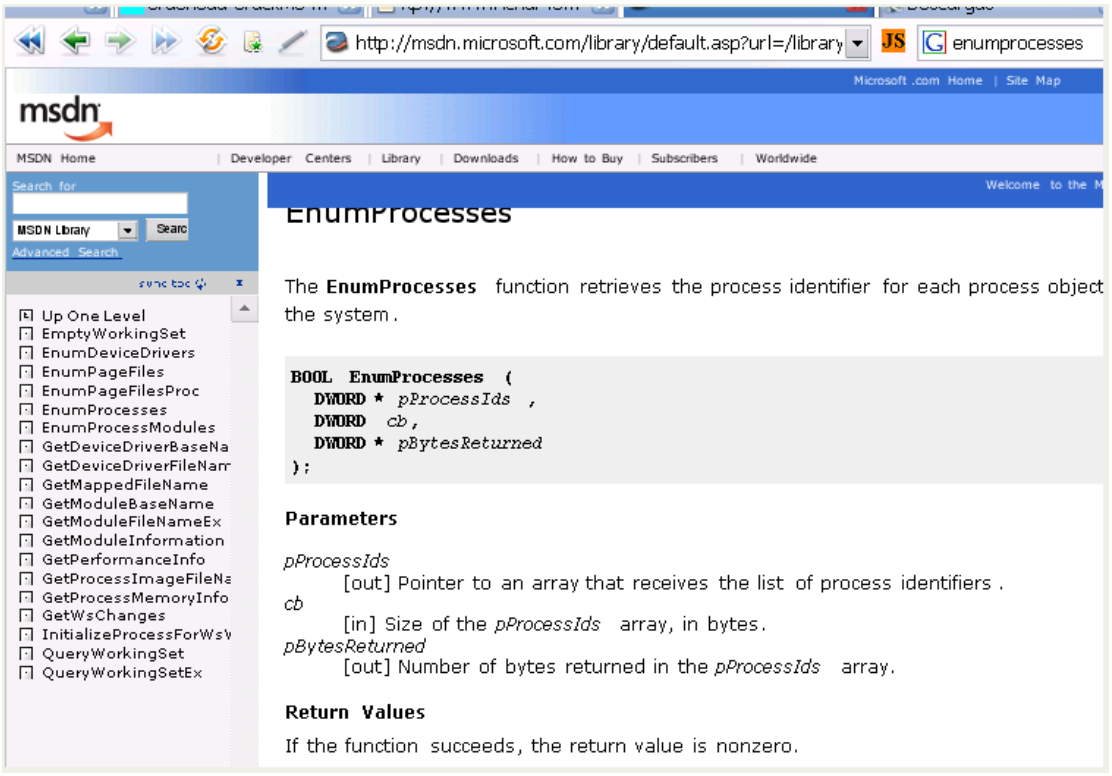
我们给这些通过 GetProcAddress 加载的 API 设置断点,并且断下来了,但是 OD 并没有在 API 函数列表中找到与之对应的项,所以 OD 也没有提示该函数相关信息。

我们谷歌一下“EnumProcesses”。



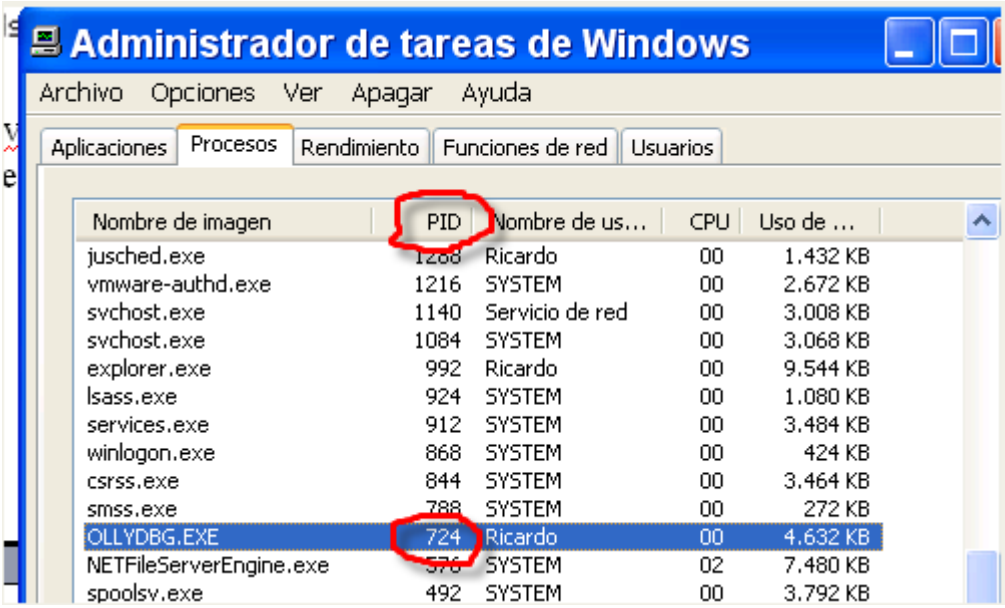
找到一个微软的网站页面:

http://msdn2.microsoft.com/en-us/library/ms682629.aspx

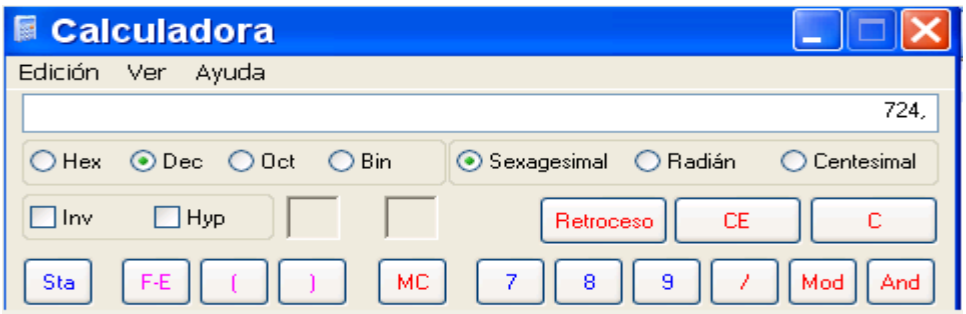


上面的解释是该函数返回正在运行的每个进程的标识即 PID。好了,接下来我们看看神马是 PID,嘿嘿。

PID 是系统分配给每个正在运行的进程的标识符-每次启动的时候都会发生变化。我们来看看进程列表。

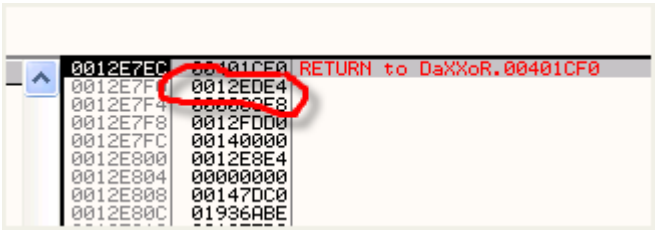


我们可以看到,这里,OD 的 PID 是 724-十进制。我们用 windows 自带的计算器将该 PID 值转化为十六进制。

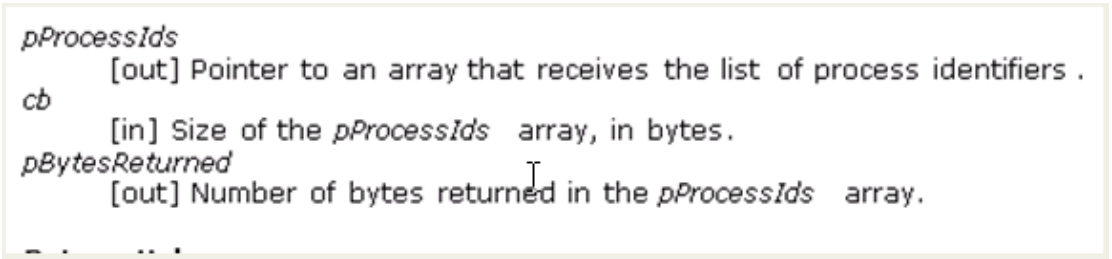


单击 Hex 按钮,就以十六进制显示了。

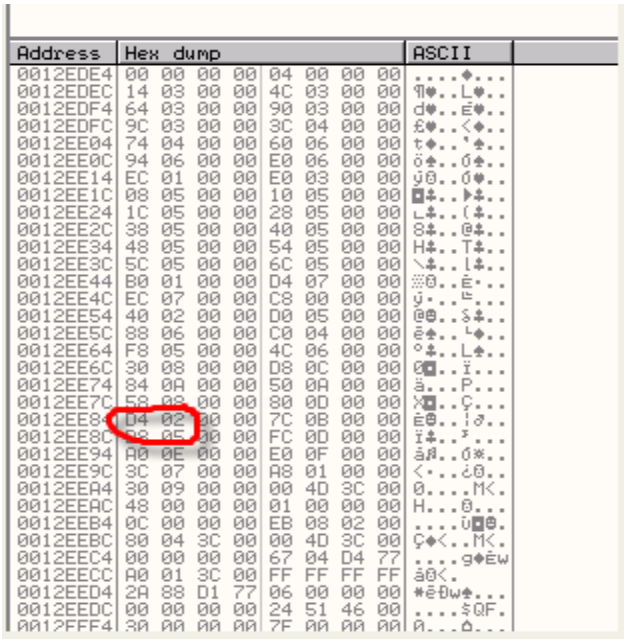
PID 对应的十六进制为 2D4。可以关闭 OD 然后重新打开一个 OD,会发现 PID 发生了变化。进程每次重新启动,PID 都改变了。很不走运,API 的参数提示也没有了。



根据 MSDN 我们知道该函数有 3 个参数。

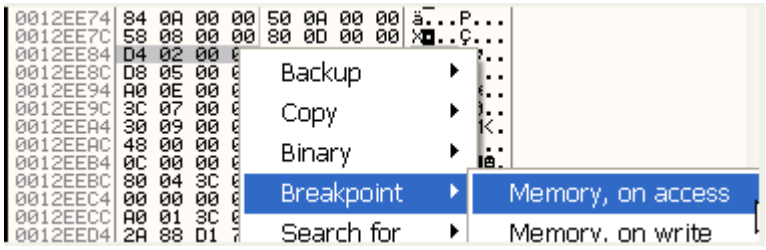


在我们的机器上 12EDE4 该地址指向了保存所有进程 PID 的数组,我们执行到返回看看该 API 函数返回了什么,我们通过数据窗口查看一下。

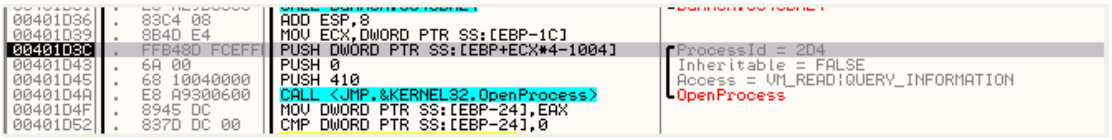


在 PID 列表中我们找到了 OD 的 PID 的值,嘿嘿。

我们给该 PID 设置内存访问断点,看看哪里使用了它。



运行起来。



可以看到这里将要调用 OpenProcess,如果调用成功,就会获取 OD 进程的句柄。

PID 跟句柄有什么区别呢?很简单。PID 是用于区别不同进程的标示符,一个进程被创建后这个进程的 PID 就是不变的,除非进程重新启动。当前,OD 的进程 PID 是 2D4。而句柄实际上是一个指针,它指向一个包含具体数据结构的内存,可能当做索引,所以句柄是你每次访问该进程的时候获取的,使用完毕后要释放,然后通过该句柄可以对该进程进行相应操作。

下面是 OpenProcess 其他参数参数解释,我们并不感兴趣。

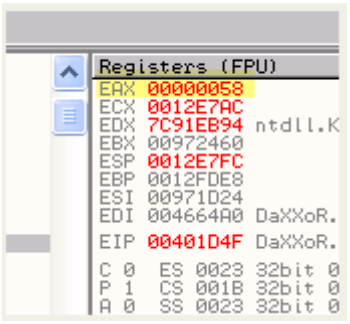
Return Values

If the function succeeds, the return value is an open handle of the specified process.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

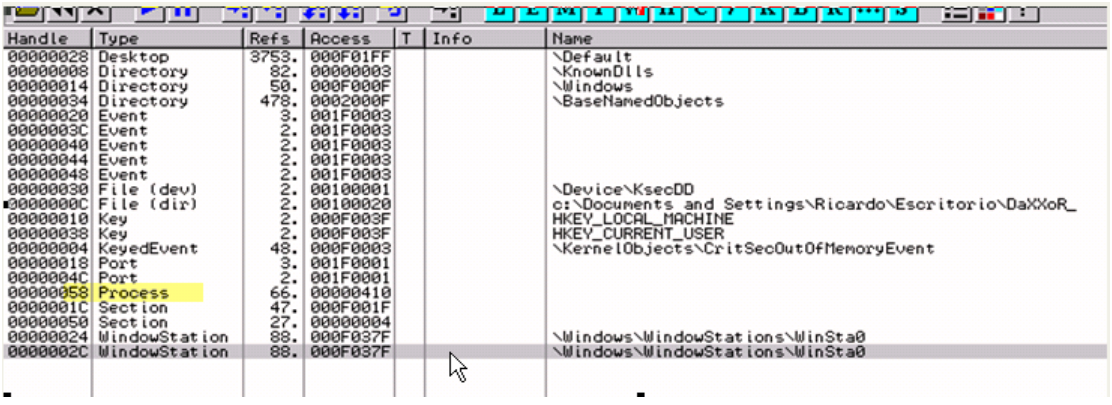
返回值为指定进程的句柄。

我们 F8 单步直到调用该函数。



EAX 中返回了 OllyDbg 进程的句柄,我这里是 58。

我们也可以通过单击工具栏上面的 H 按钮查看 OD 的句柄列表。

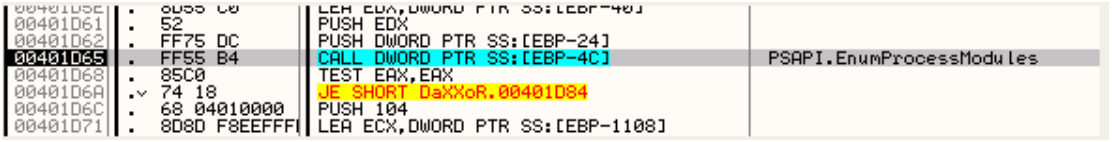


我们可以看到 58 对应的类型是 Process,它标识的是 OllyDbg 的进程句柄。

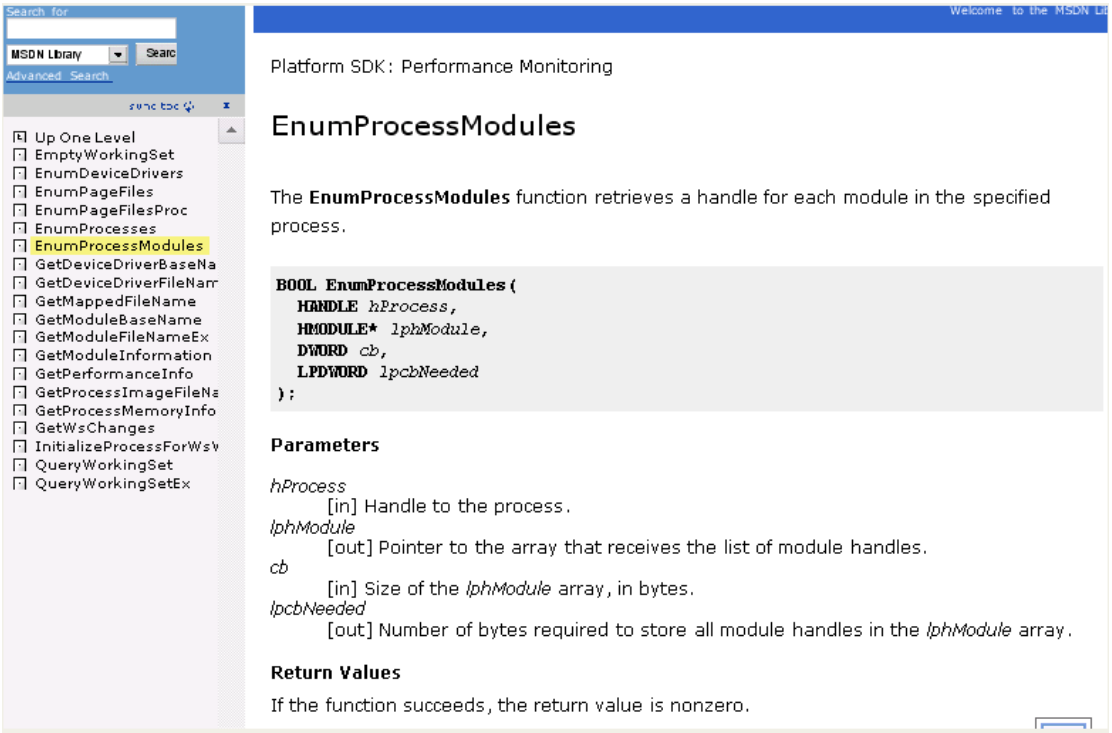
如果另一个进程调用 EnumProcesses 获取 OD 的进程 PID 的话,仍然会是 2D4,但是获取进程句柄的话,会是另外一个值,因为句柄在系统中是独一无二的。

这里,OllyDbg 进程就比较危险了,有了 OD 的进程句柄,该程序就可以做很多事情了(比如结束 OD 进程),以上只是检测 OllyDbg 步骤的一部分,现在还需要来验证一下进程名称是否为 OllyDbg,很明显该程序会对进程列表的所有进程做以上操作,我们跳过这些步骤直接给对应的 PID 设置内存访问断点。

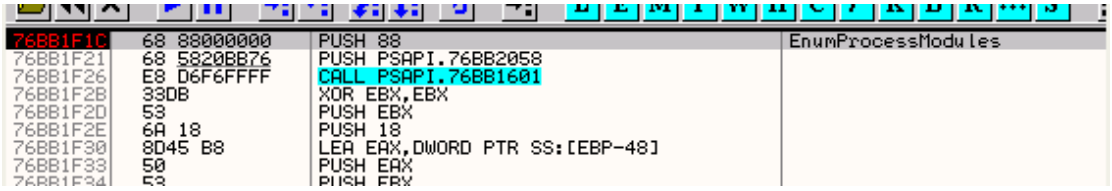
我们继续 F8 键单步。



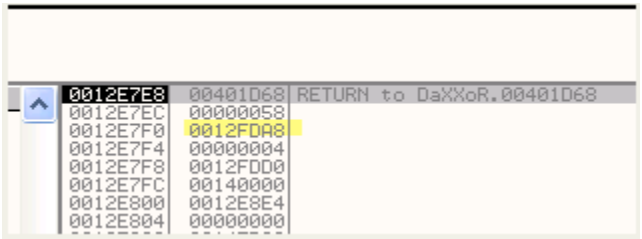
我们看到了 EnumProcessModules 这个函数,我们看看 MSDN 上面关于这个函数的说明。



该函数是枚举指定进程的模块,我们 F7 键跟进到该函数的入口处。

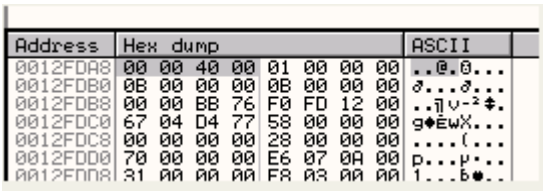


我们在堆栈窗口中来看看参数情况:

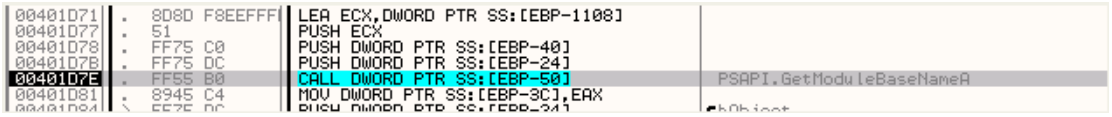


58 是 OD 的进程句柄作为第一个参数。

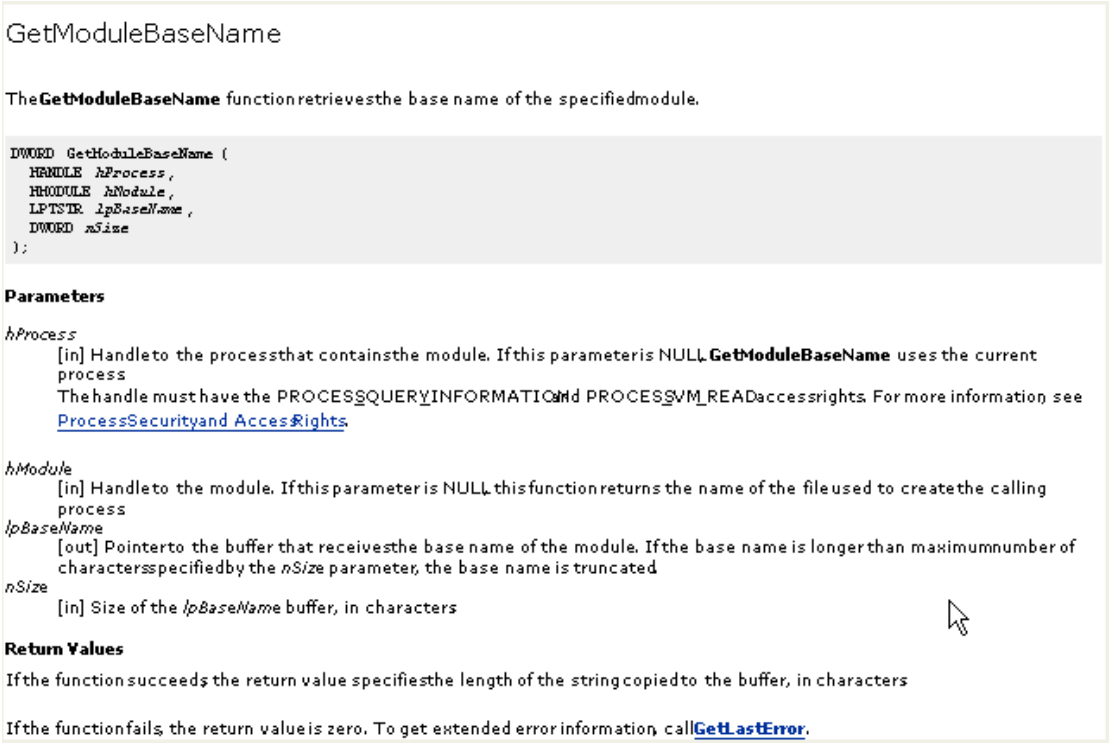
这里,你需要清楚一点:当我们请求获取进程中模块句柄的时候,系统会返回该模块在进程内存中基地址(模块起始地址),这里,系统给我返回的是 400000,该基地址对应的 OllyDbg 进程中的。



继续跟,我们可以看到另一个 API 函数。



GetModuleBaseNameA



GetModuleBaseName

The **GetModuleBaseName** function retrieves the base name of the specified module.

```
DWORD GetModuleBaseName(
  HANDLE hProcess,
  HMODULE hModule,
  LPTSTR lpBaseName,
  DWORD nSize
);
```

Parameters

hProcess
[in] Handle to the process that contains the module. If this parameter is NULL, **GetModuleBaseName** uses the current process.
The handle must have the PROCESS_QUERY_INFORMATION and PROCESS_VM_READ access rights. For more information, see Process Security and Access Rights.

hModule
[in] Handle to the module. If this parameter is NULL, this function returns the name of the file used to create the calling process.

lpBaseName
[out] Pointer to the buffer that receives the base name of the module. If the base name is longer than maximum number of characters specified by the *nSize* parameter, the base name is truncated.

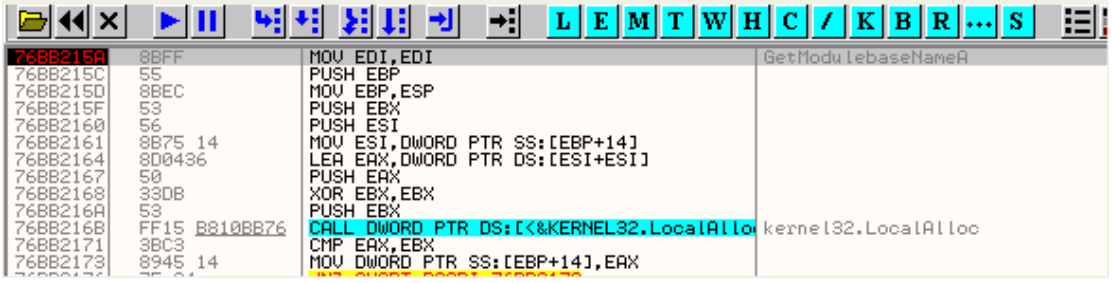
nSize
[in] Size of the *lpBaseName* buffer, in characters.

Return Values

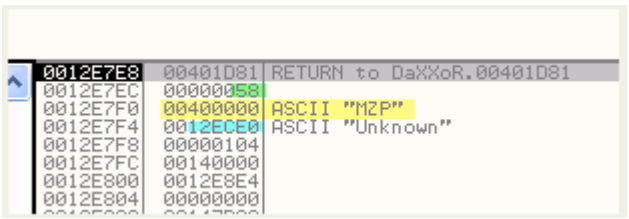
If the function succeeds, the return value specifies the length of the string copied to the buffer, in characters.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

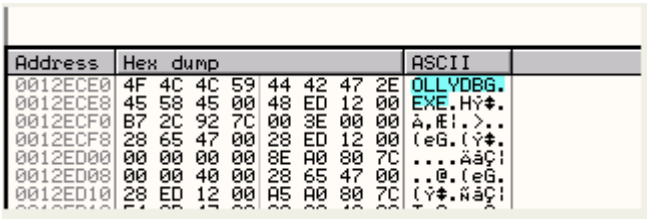
该函数是获取指定进程模块的名称,lpBaseName 这个参数是用来保存模块名称的缓冲区首地址,我们执行到返回。



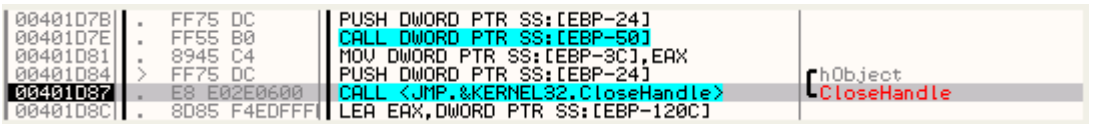
堆栈窗口中参数情况如下:



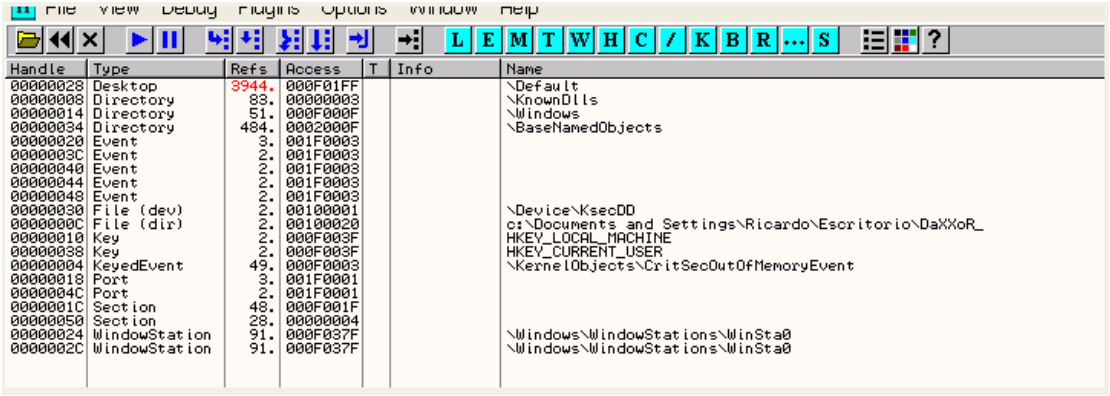
58 是 OllyDbg 的进程句柄,400000 是 OD 主模块的基地址,用于保存模块名称的缓冲区首地址为 12ECE0,我们在数据窗口中定位到这个地址。



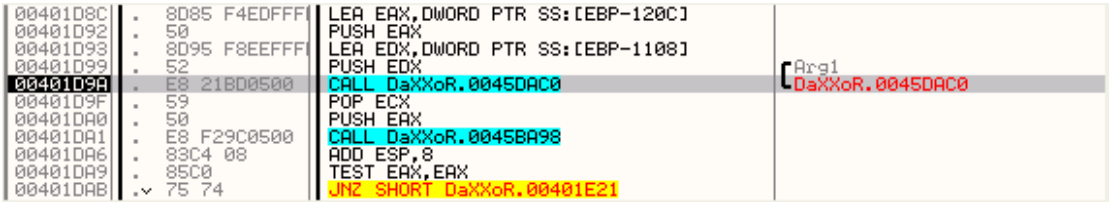
呵呵,很显然,我们执行到返回以后获取到了进程的模块名称,同理,该程序会依次判断每个进程的名称是否为”OLLYDBG.exe”。



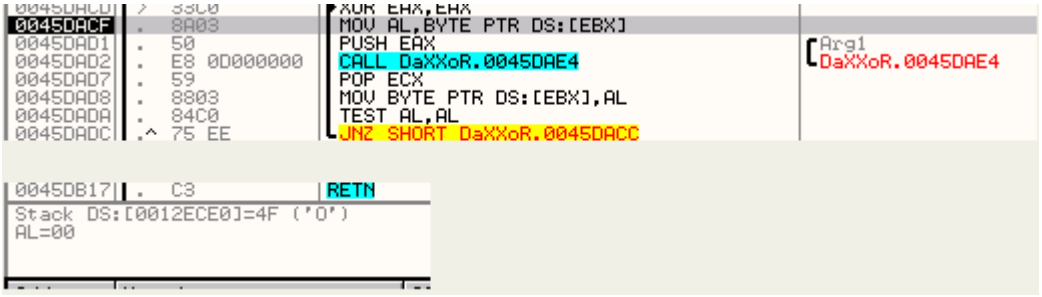
好了,现在我们看到了 CloseHandle 这个 API 函数,该函数用来关闭指定的句柄,即 58 将从句柄列表中消失。



好了,现在进程句柄被关闭了,我们不能再依赖它了。

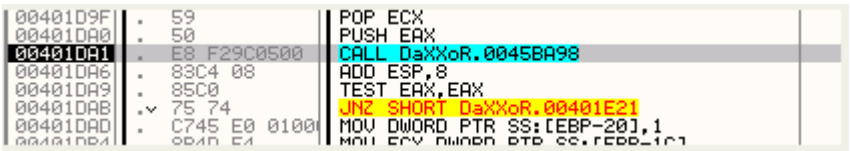


这里有一个 CALL 指令,我们按 F7 键跟进。



这里,我们可以看到将要压入堆栈的是“OLLYDBG”字符串的第一个字母(4F),然后调用一个 CALL 指令,我们继续跟进。

我们可以看到这个 CALL 里面没有什么特别的,所以我们跟出来,然后跟到下一个 CALL 指令处。



跟进。

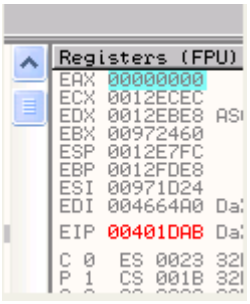
```
0045BA98  $  8B4C24 04    MOV ECX,DWORD PTR SS:[ESP+4]
0045BA9C  .  8B5424 08    MOV EDX,DWORD PTR SS:[ESP+8]
0045BAA0  .  53           PUSH EBX
0045BAA1  .  33C0         XOR EAX,EAX
0045BAA3  .  33DB         XOR EBX,EBX
0045BAA5  >  8A01         MOV AL,BYTE PTR DS:[ECX]
0045BAA7  .  8A1A         MOV BL,BYTE PTR DS:[EDX]
0045BAA9  .  2BC3         SUB EAX,EBX
0045BAAB  .v 75 34        JNZ SHORT DaXXoR.0045BAE1
0045BAAD  .  84DB         TEST BL,BL
0045BAAF  .v 74 30        JE SHORT DaXXoR.0045BAE1
0045BAB1  .  8A41 01      MOV AL,BYTE PTR DS:[ECX+1]
0045BAB4  .  8A5A 01      MOV BL,BYTE PTR DS:[EDX+1]
0045BAB7  .  2BC3         SUB EAX,EBX
0045BAB9  .v 75 26        JNZ SHORT DaXXoR.0045BAE1
0045BABB  .  84DB         TEST BL,BL
0045BABD  .v 74 22        JE SHORT DaXXoR.0045BAE1
0045BABF  .  8A41 02      MOV AL,BYTE PTR DS:[ECX+2]
0045BAC2  .  8A5A 02      MOV BL,BYTE PTR DS:[EDX+2]
0045BAC5  .  2BC3         SUB EAX,EBX
0045BAC7  .v 75 18        JNZ SHORT DaXXoR.0045BAE1
0045BAC9  .  84DB         TEST BL,BL
0045BACB  .v 74 14        JE SHORT DaXXoR.0045BAE1
0045BACD  .  8A41 03      MOV AL,BYTE PTR DS:[ECX+3]
0045BAD0  .  8A5A 03      MOV BL,BYTE PTR DS:[EDX+3]
0045BAD3  .  2BC3         SUB EAX,EBX
0045BAD5  .v 75 0A        JNZ SHORT DaXXoR.0045BAE1
0045BAD7  .  83C1 04      ADD ECX,4
0045BADA  .  83C2 04      ADD EDX,4
0045BADD  .  84DB         TEST BL,BL
0045BADF  .^ 75 C4        JNZ SHORT DaXXoR.0045BAA5
0045BAE1  >  5B           POP EBX
0045BAE2  L. C3           RETN
```

```
Registers (FPU)
EAX 00000000
ECX 0012ECE0 ASCII "OLLYDBG.EXE"
EDX 0012EBDC ASCII "OLLYDBG.EXE"
EBX 00000000
ESP 0012E7EC
EBP 0012FDE8
ESI 00971D24
EDI 004664A0 DaXXoR.004664A0

EIP 0045BAA5 DaXXoR.0045BAA5

C 0  ES 0023 32bit 0(FFFFFFFF)
```

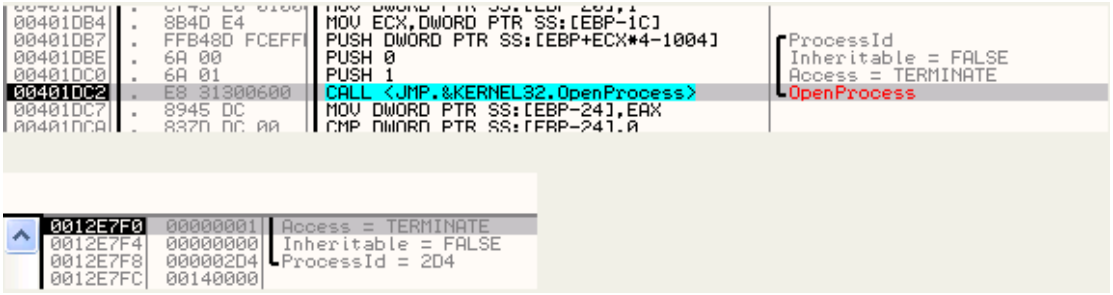
嘿嘿,这里在比较进程名称是否为”OLLYDBG.EXE”,如果是,就会...嘿嘿,我们现在执行到返回,看看会发生什么。

```
.  50              PUSH EAX
.  E8 F29C0500     CALL DaXXoR.0045BA98
.  83C4 08         ADD ESP,8
.  85C0            TEST EAX,EAX
.v 75 74           JNZ SHORT DaXXoR.00401E21
.  C745 E0 0100    MOV DWORD PTR SS:[EBP-20],1
.  8B4D E4         MOV ECX,DWORD PTR SS:[EBP-1C]
.  FFB48D FCEFF    PUSH DWORD PTR SS:[EBP+ECX*4-1004]      ProcessId
.  6A 00           PUSH 0                                  Inheritable = FALSE
.  6A 01           PUSH 1                                  Access = TERMINATE
.  E8 31300600     CALL <JMP.&KERNEL32.OpenProcess>        OpenProcess
.  8945 DC         MOV DWORD PTR SS:[EBP-24],EAX
.  837D DC 00      CMP DWORD PTR SS:[EBP-24],0
.v 74 3F           JE SHORT DaXXoR.00401E0F
.  6A 00           PUSH 0                                  ExitCode = 0
.  FF75 DC         PUSH DWORD PTR SS:[EBP-24]              hProcess
.  E8 78300600     CALL <JMP.&KERNEL32.TerminateProcess>   TerminateProcess
.  85C0            TEST EAX,EAX
.v 74 17           JE SHORT DaXXoR.00401DF5
.  FF75 DC         PUSH DWORD PTR SS:[EBP-24]              hObject
.  E8 862E0600     CALL <JMP.&KERNEL32.CloseHandle>        CloseHandle
.  FF75 D0         PUSH DWORD PTR SS:[EBP-30]              hLibModule
.  E8 CC2E0600     CALL <JMP.&KERNEL32.FreeLibrary>        FreeLibrary
.  33C0            XOR EAX,EAX
.v E9 AF020000     JMP DaXXoR.004020A4
>  FF75 DC         PUSH DWORD PTR SS:[EBP-24]              hObject
.  E8 6F2E0600     CALL <JMP.&KERNEL32.CloseHandle>        CloseHandle
.  FF75 D0         PUSH DWORD PTR SS:[EBP-30]              hLibModule
.  E8 B52E0600     CALL <JMP.&KERNEL32.FreeLibrary>        FreeLibrary
.  B8 5A020000     MOV EAX,25A
.v E9 95020000     JMP DaXXoR.004020A4
>  FF75 D0         PUSH DWORD PTR SS:[EBP-30]              hLibModule
.  E8 A32E0600     CALL <JMP.&KERNEL32.FreeLibrary>        FreeLibrary
.  B8 5C020000     MOV EAX,25C
.v E9 83020000     JMP DaXXoR.004020A4
>  FF45 E4         INC DWORD PTR SS:[EBP-1C]

T taken
aXXoR.00401E21
```

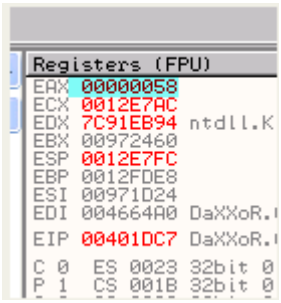
如果进程名不为”OLLYDBG.EXE”,EAX 就不等于 0,JNZ 条件跳转将会发生,如果 EAX 为 0,条件跳转不会发生,将执行后面结束 OD 进程等一系列操作。



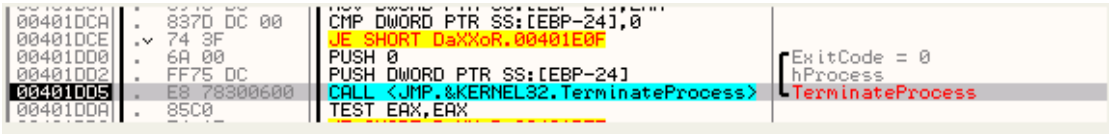
跳转不会发生,将会再次调用 OpenProcess 打开 OD 进程并且获得其句柄,然后调用 TerminateProcess 结束掉该 OD 进程。



我们按 F8 键单步。



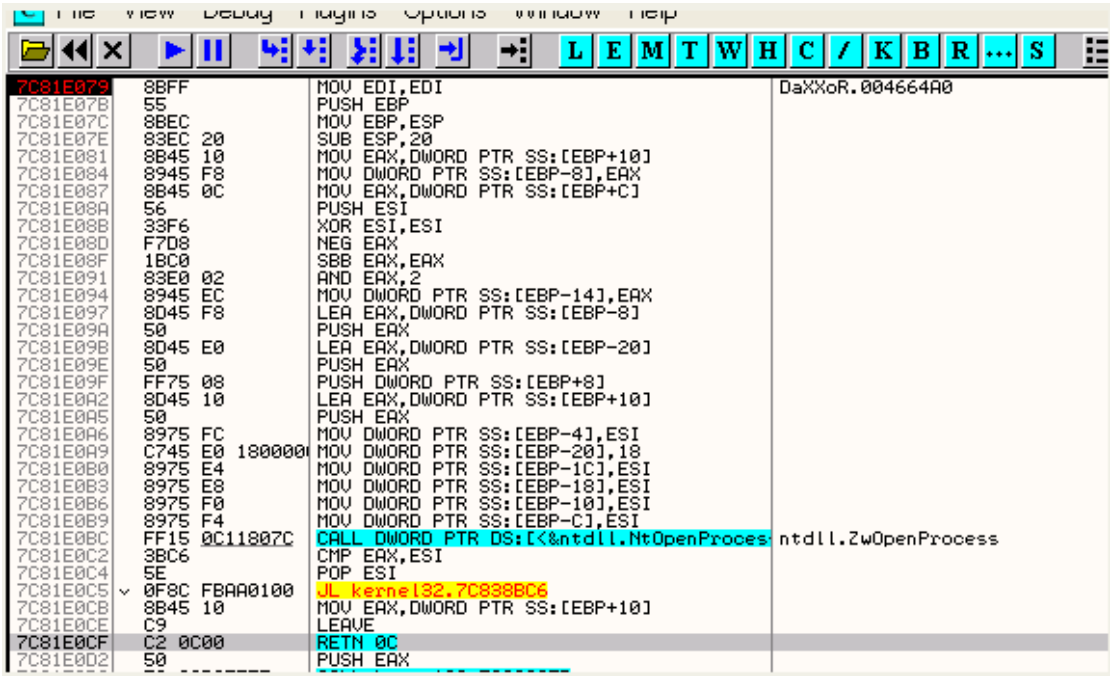
返回的是 58,我们继续。



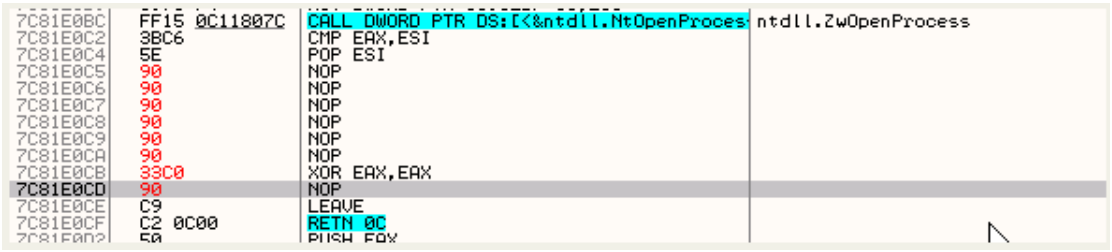
正如你所看到的,调用 TerminateProcess 这个 API 函数,指定进程的句柄,结束调用 OllyDbg 进程。

我们按 F8 键,OD 被关闭了。这就是我们研究的如何检测 OllyDbg 进程名的原理。

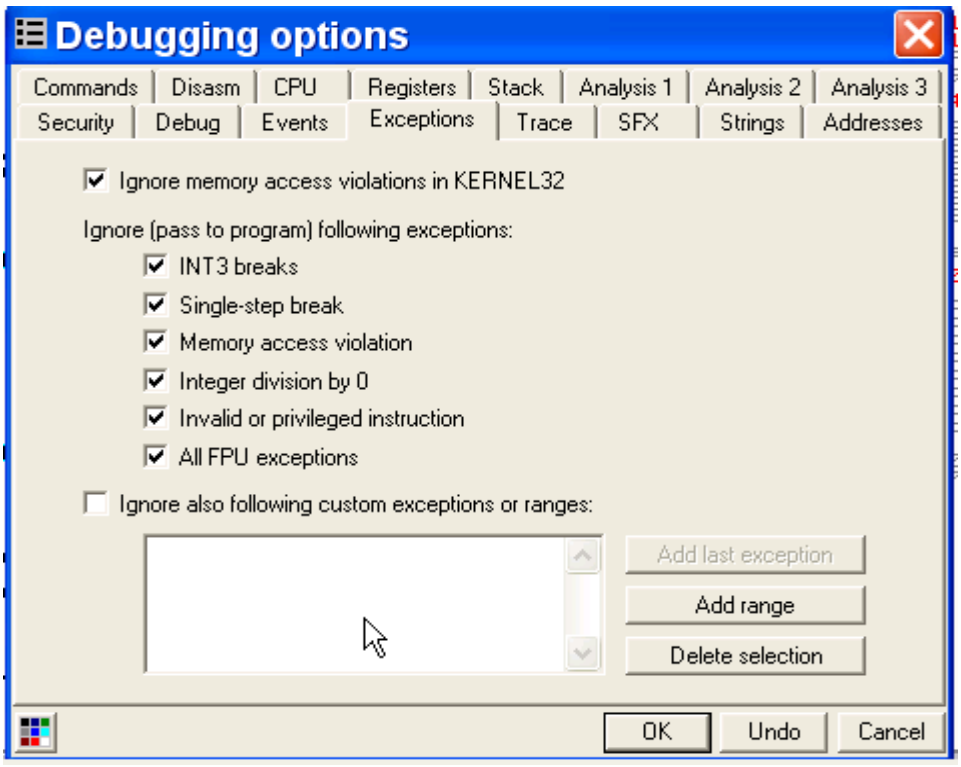
好吧,我们重新启动 OD,然后在 OpenProcess 入口处设置断点。



断了下来,我们可以修改该 API 函数,让它总是返回 0,程序就会认为没有其他程序正在运行,获取不到任何进程句柄,要做到这一点,我们修改最后一行。

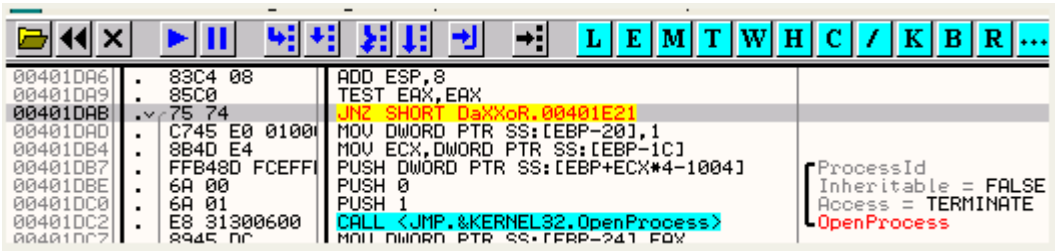


这样,这个 API 函数就总是返回 0 了。现在我们删除之前设置的所有断点。接着设置一下主菜单项中的调试选项中的异常,如下:



这里,我们勾选上忽略所以异常,因为这些异常中(有的必须按 Shift + F9 键,才能忽略异常继续执行,这里我们直接忽略掉)运行起来。

正常运行,其实你也可以这么做:

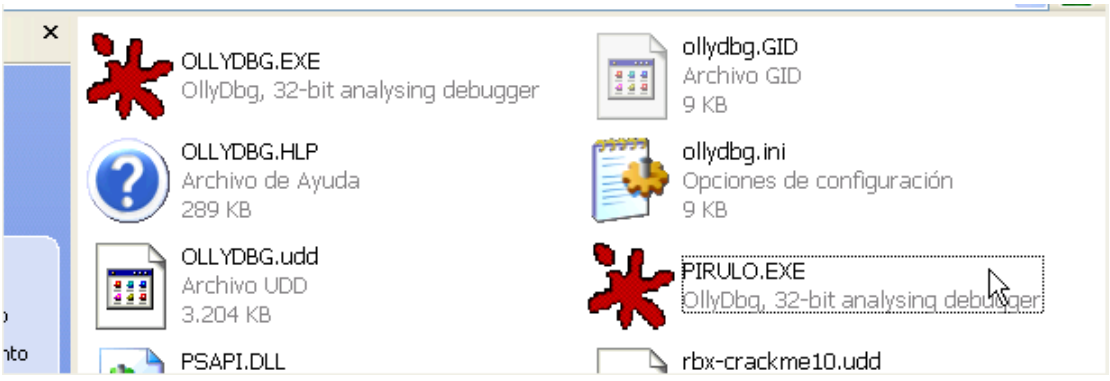


将 JNZ 修改为 JMP,让其直接跳过下面的关闭 OllyDbg 进程的代码。

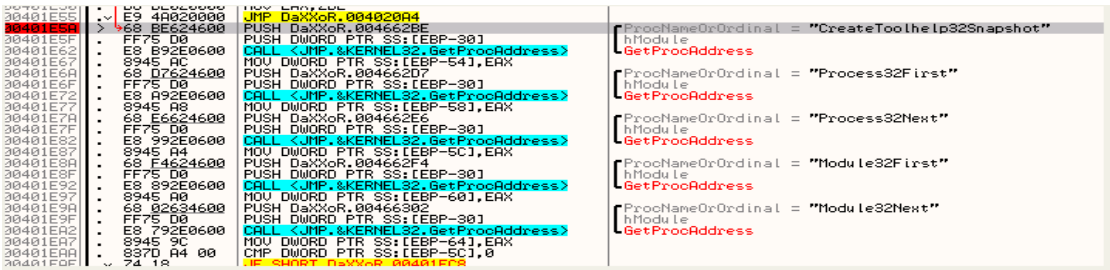
运行起来,主窗口显示出来了,我们单击 Try 按钮。

会弹出一个错误信息框。显示您已经成功绕过该反调试。

以上的方法并不一定能绕过这个反调试,其实我们可以通过将 OLLYDBG.EXE 创建一个副本,将其重命名为 PIRULO.EXE,然后运行该程序,被其调试的程序将无法找到一个进程名为 OLLYDBG 的进程了。



有一点很重要就是在 OD 的文件夹下面留一个原始的 OD,避免 OD 的插件出现问题。



以上只是该程序检测 OllyDbg 的一部分,如果我们输入正确的序列号,该程序还会调用其他的一些 API 函数来检测是否被 OD 调用,我们下一章再来详细讨论。