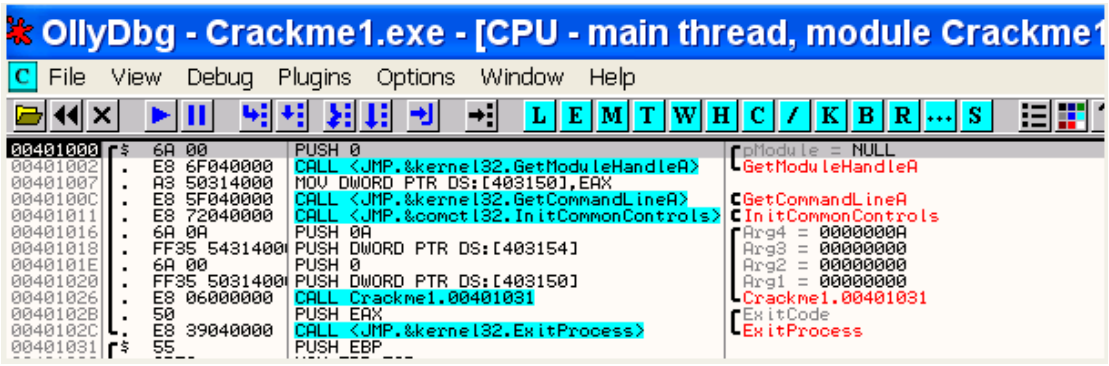
第十九章-OllyDbg 反调试之 IsDebuggerPresent

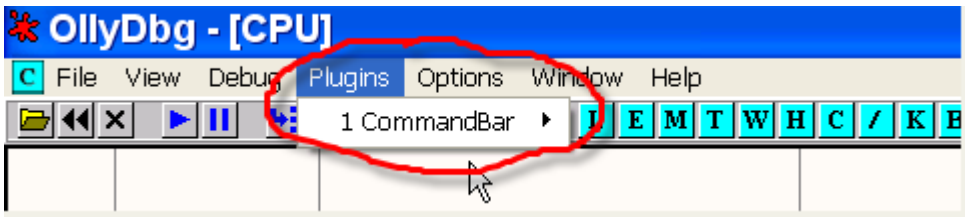
本章开始,我们将讨论反调试的相关话题,包括手工以及通过 OD 插件来绕过对应的反调试的技巧。很多程序会检测自身是否正在被调试,如果检测到正在被调试的话,就会结束自身进程或者不按常规流程运行。所以绕过程序对 OD 的检测是很有必要的。

本章就介绍使用 API 函数-IsDebuggerPresent 检测 OD,这也是最常用的检测调试器的方法。

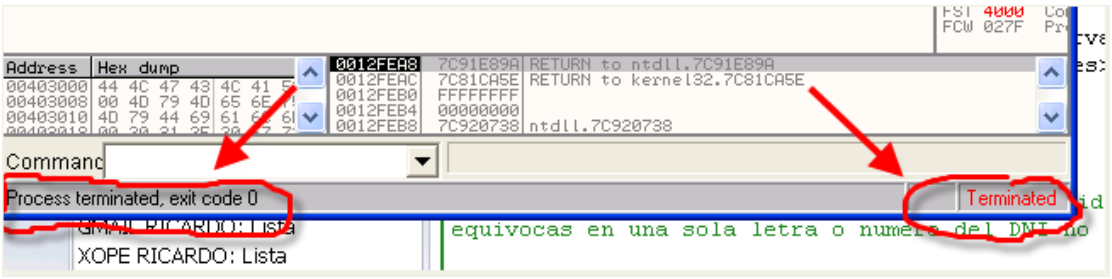
这里,我们使用 Crackme1.exe 来讲解,用 OD 加载它。



我们记得我们的 OD 只安装了命令栏插件,并没有安装绕过 IsDebuggerPresent 检测的插件,那么是如何使用 IsDebuggerPresent 来检测 OD 的呢?

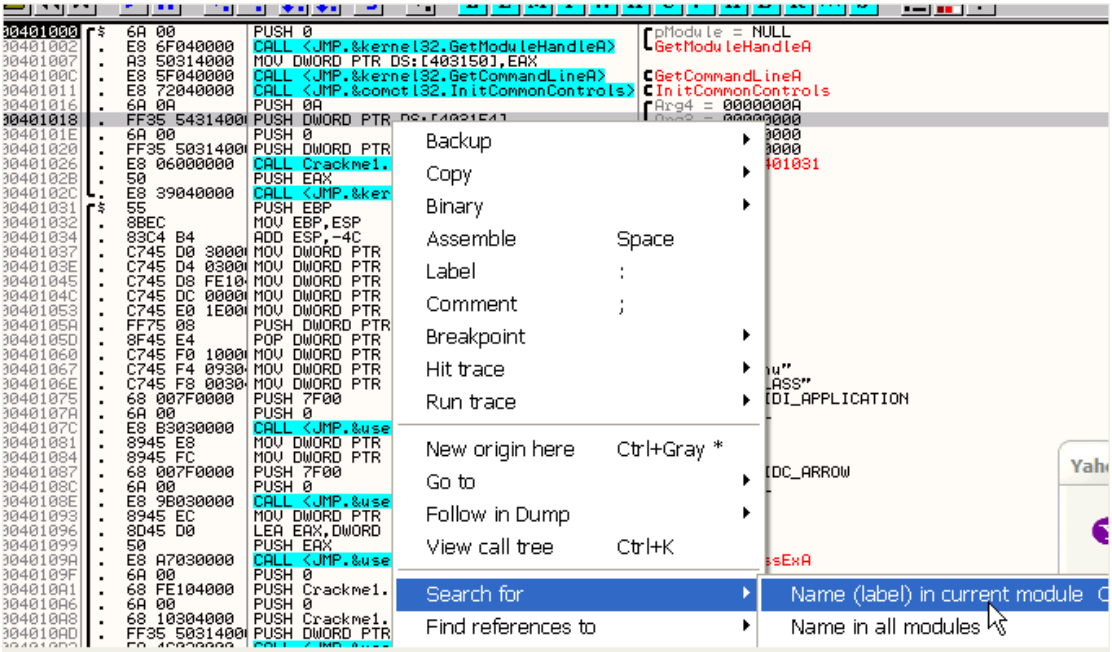


如果我们按 F9 键让程序运行起来,我们会发现并没有弹出 CrackMe 窗口,程序直接终止了。

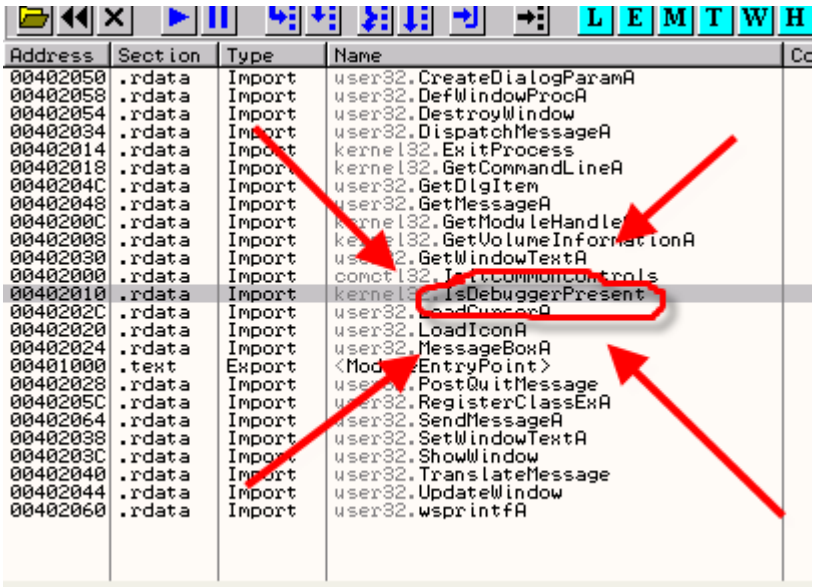


OD 的左下方显示程序已经终止,所以,我们看不到窗口出现,嘿嘿。该 CrackMe 可能使用的是最常见的 API 函数 IsDebuggerPresent 来检测是否被调试的。

重启该 CrackMe,通过单击鼠标右键选择 Search for-Name(label) in current module 查看 API 函数列表,看看是否使用了 IsDebuggerPresent。

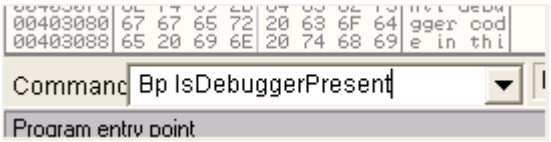


我们看看 API 函数列表

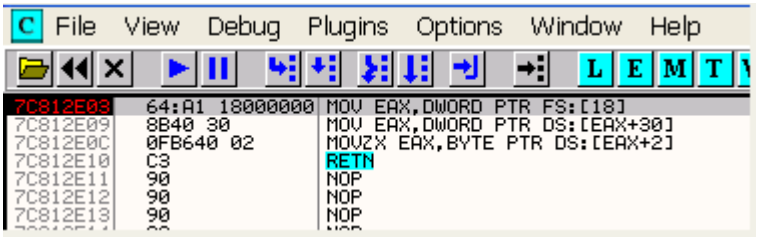


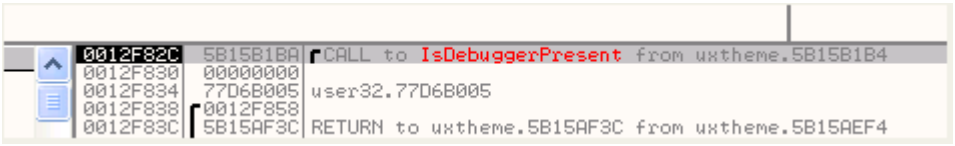
使用了 IsDebuggerPresent,嘿嘿。

我们对该函数设置一个断点,看看该 CrackMe 哪里使用了这个函数。

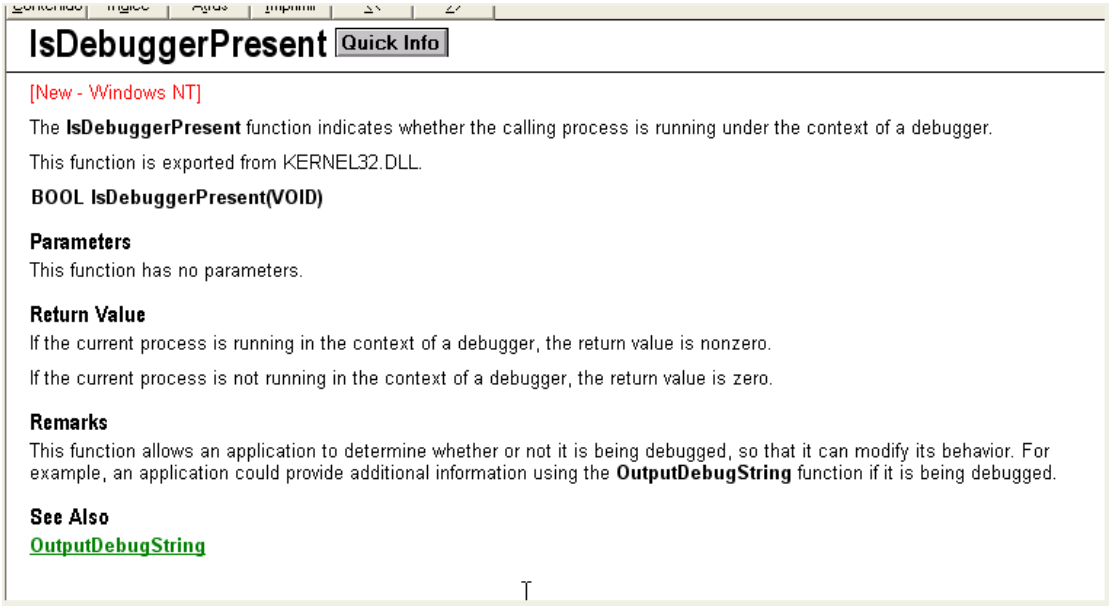


我们运行起来,马上就断在该函数的入口处了。





根据堆栈窗口中的信息来看,该 API 函数没有参数,它干的唯一的事情就是检测当前程序是否正在被调试,如果你对该函数还有什么疑问,可以查看 MSDN。



IsDebuggerPresent Quick Info

[New - Windows NT]

The **IsDebuggerPresent** function indicates whether the calling process is running under the context of a debugger.

This function is exported from KERNEL32.DLL.

BOOL IsDebuggerPresent(VOID)

Parameters

This function has no parameters.

Return Value

If the current process is running in the context of a debugger, the return value is nonzero.

If the current process is not running in the context of a debugger, the return value is zero.

Remarks

This function allows an application to determine whether or not it is being debugged, so that it can modify its behavior. For example, an application could provide additional information using the **OutputDebugString** function if it is being debugged.

See Also

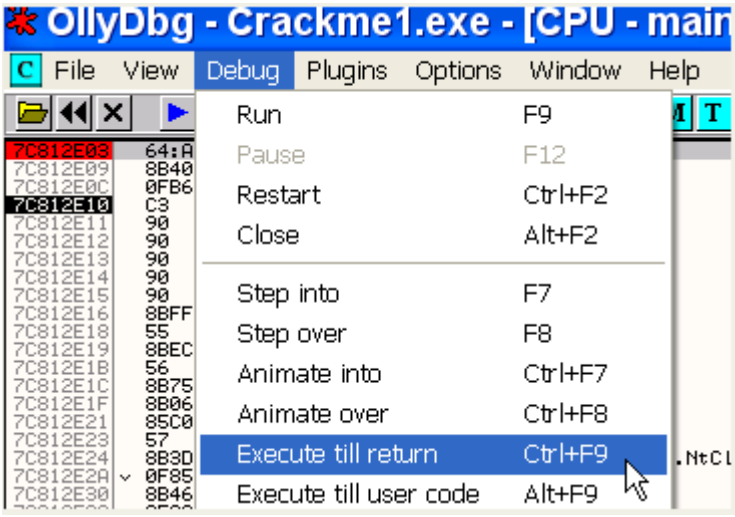
OutputDebugString

这里解释了该函数的功能,我们来翻译一下。

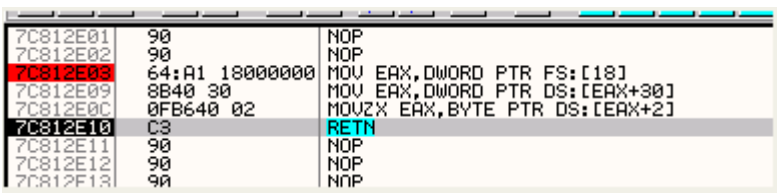
IsDebuggerPresent 表示在被调试器调试情况下,调用该函数会返回正在被调试。

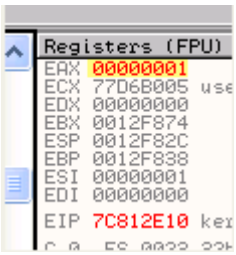
并且该函数是被 Kernel32.dll 导出的,该函数没有参数,如果当前程序正在被调试的话,返回值为 1,没有被调试的话,返回值为 0。

这是非常重要的信息,我们执行到返回,看看返回值是多少。



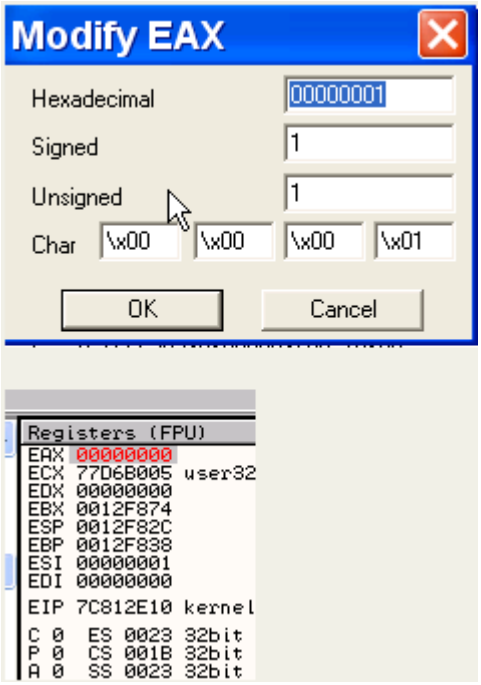
我们停在了 RET 指令处,看看寄存器的情况。



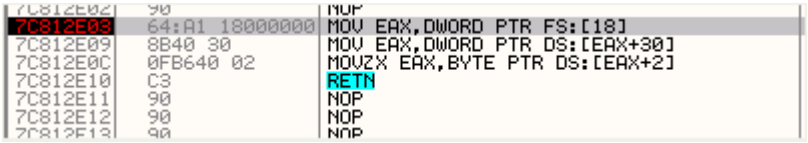


EAX 的值变成了粉红色,表示 EAX 的值被修改过,跟其他 API 函数一样,IsDebuggerPersent 的返回值也保存在 EAX 中,这里 EAX 为 1,表示当前程序正在被调试。

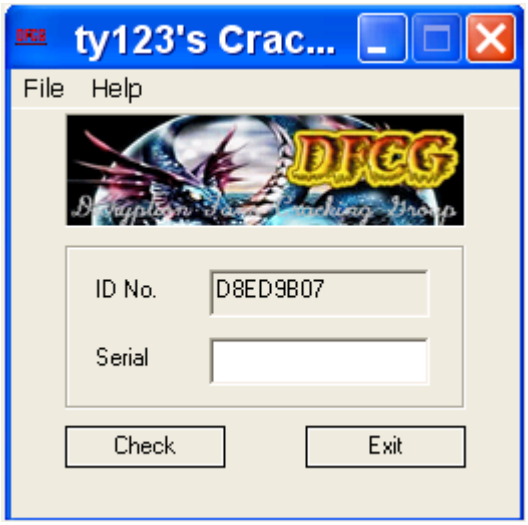
我们尝试手动将 EAX 修改为 0,表示当前程序没有被调试,看看会发生什么。



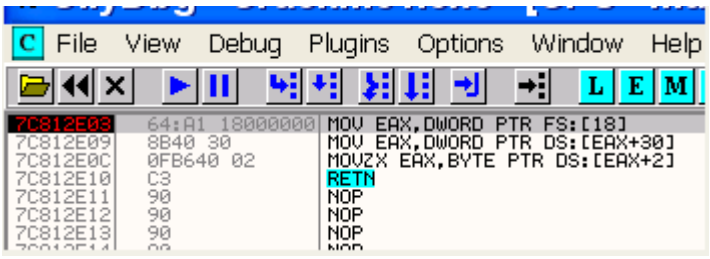
运行起来。



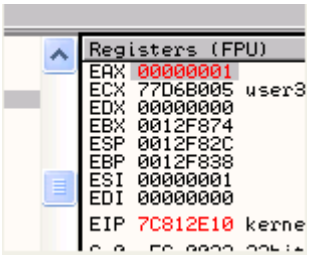
再次断在了该 API 函数入口处,我们执行到返回,然后将 EAX 的值修改为 0。



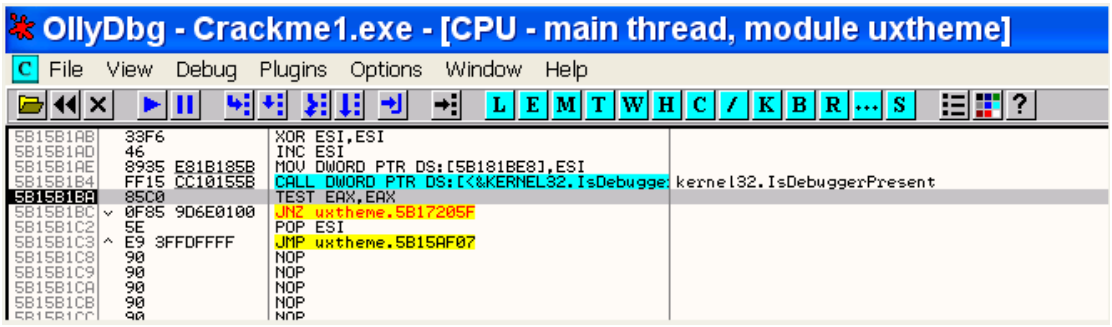
我们可以看到该程序启动时的检测是基于 IsDebuggerPresent 这个 API 函数的,我们重新运行 CrackMe,会断在 IsDebuggerPersent 处,相应返回值的解释可以参考 MSDN。



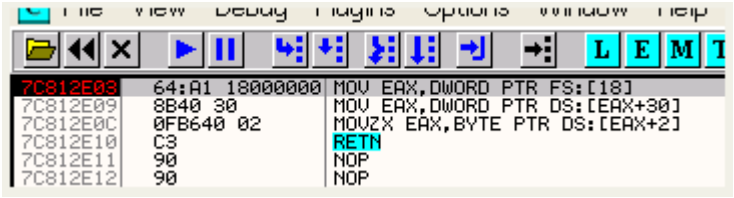
我们可以执行到返回,也可以直接 F8 单步到 RET 指令,因为这个函数很简短,就几条指令。



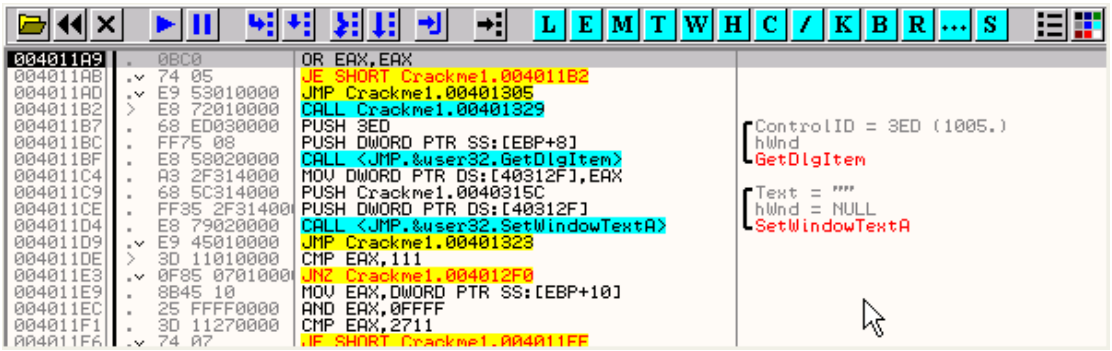
这里到了 RET 指令处,EAX 的值为 1,按 F8 键返回。



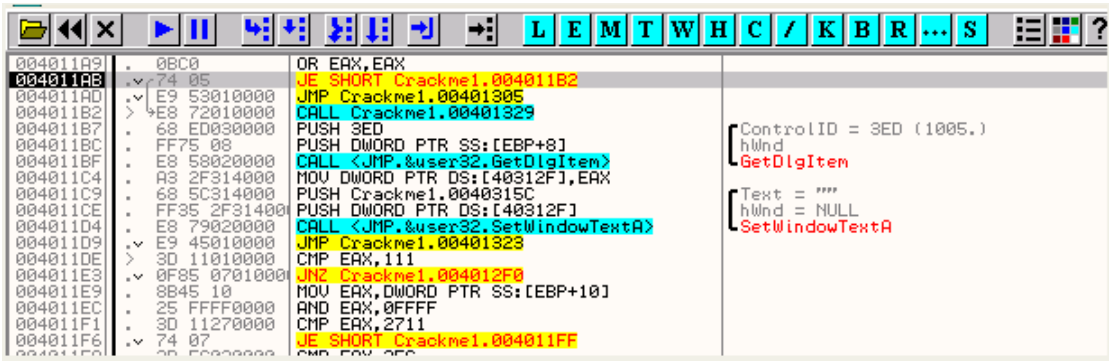
这里我们可以看到,IsDebuggerPersent 的调用是位于 uxtheme.dll 模块中的,我们运行起来,会第二次断在 IsDebuggerPresent 的入口处。



这是第二次断在这里了,我们执行到返回,接着 F8 键返回到上层调用处。

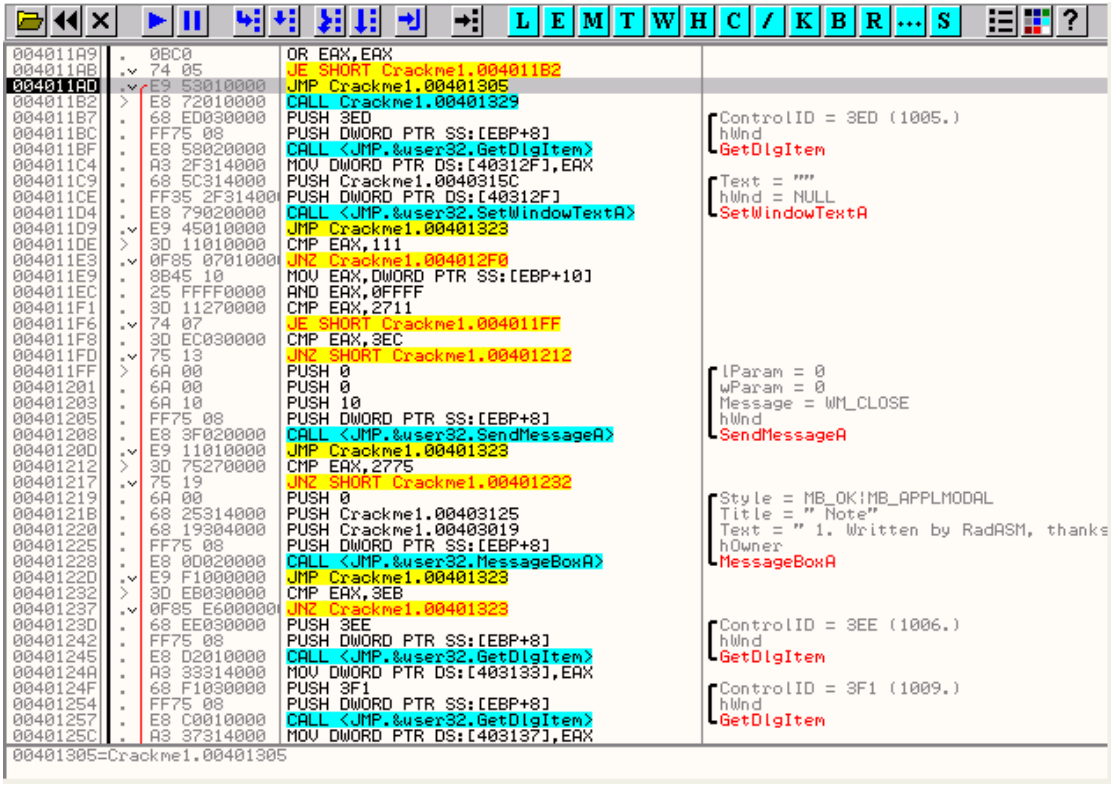


我们可以看到返回到了 4011A9 地址处,这里有一个条件跳转 JE 指令,判断 EAX 的值是否为 0。

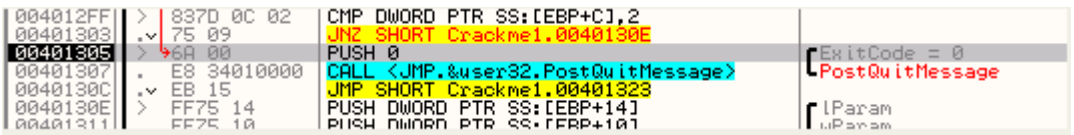


可以看到如果 EAX 不等于 0,跳转将不会发生。如果 EAX 为 0,条件跳转发生,程序将继续执行 GetDlgItem 函数,读取 CrackMe 窗口的信息。

在这里条件跳转不会发生,接着执行后面的无条件跳转 JMP 指令。



继续跟,我们到了这里。



这里调用了 PostQuitMessage 这里 API 函数,我们看下 MSDN 中关于该函数的说明。

PostQuitMessage Quick Info

The **PostQuitMessage** function indicates to Windows that a thread has made a request to terminate (quit). It is typically used in response to a WM_DESTROY message.

```
VOID PostQuitMessage(
  int nExitCode        // exit code
);
```

Parameters

nExitCode

Specifies an application exit code. This value is used as the *wParam* parameter of the WM_QUIT message.

Return Values

This function does not return a value.

Remarks

The **PostQuitMessage** function posts a WM_QUIT message to the thread's message queue and returns immediately; the function simply indicates to the system that the thread is requesting to quit at some time in the future.

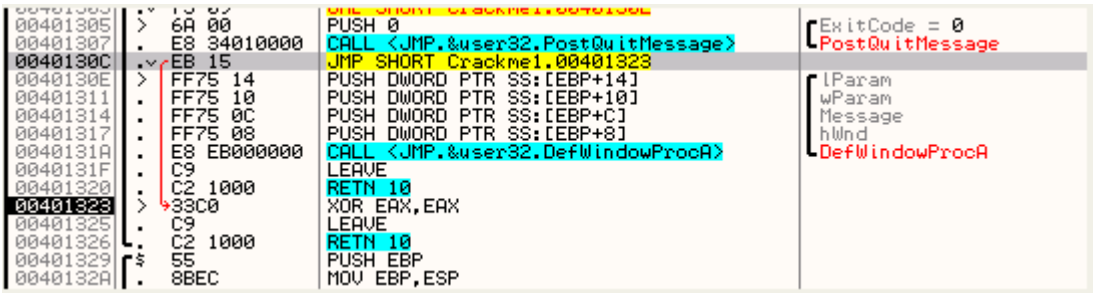
When the thread retrieves the WM_QUIT message from its message queue, it should exit its message loop and return control to Windows. The exit value returned to Windows must be the *wParam* parameter of the WM_QUIT message.

See Also

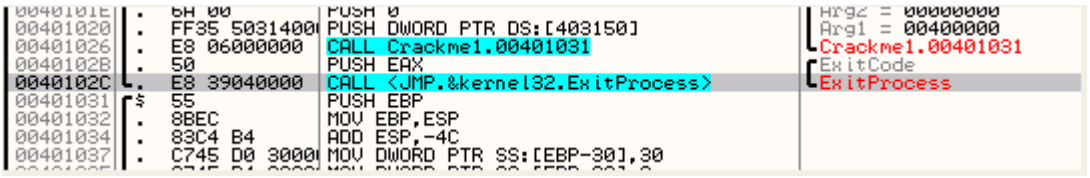
GetMessage, **PeekMessage**, **PostMessage**, WM_DESTROY, WM_QUIT

MSDN 上说该 API 函数给线程消息队列发送一个 WM_QUIT 消息来关闭窗口。

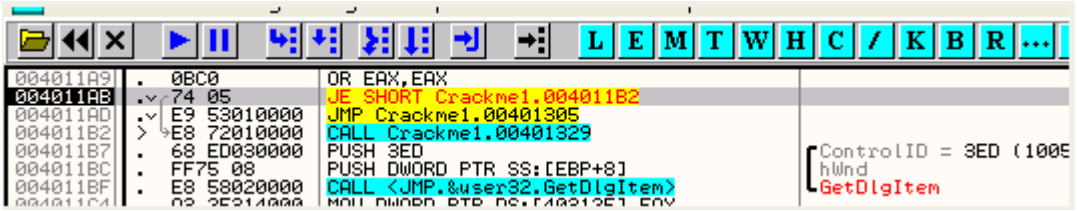
执行完该函数后返回:



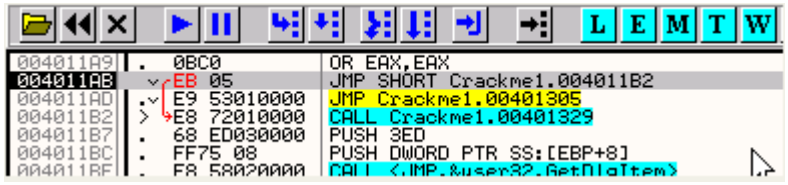
如果大家耐心的继续跟的话,就会发现该程序会调用 ExitProcess 函数来结束进程。(如果大家不想一步步的跟的话,可以直接给 ExitProcess 函数设置一个断点,然后运行起来,会发现断在了 ExitProcess 函数处)



所以,我们可以看出 IsDebuggerPresent 的返回值决定了程序是继续运行还是结束。重新运行程序又到了该条件跳转指令 JE 处,一种方案是我们可以给该程序打补丁。

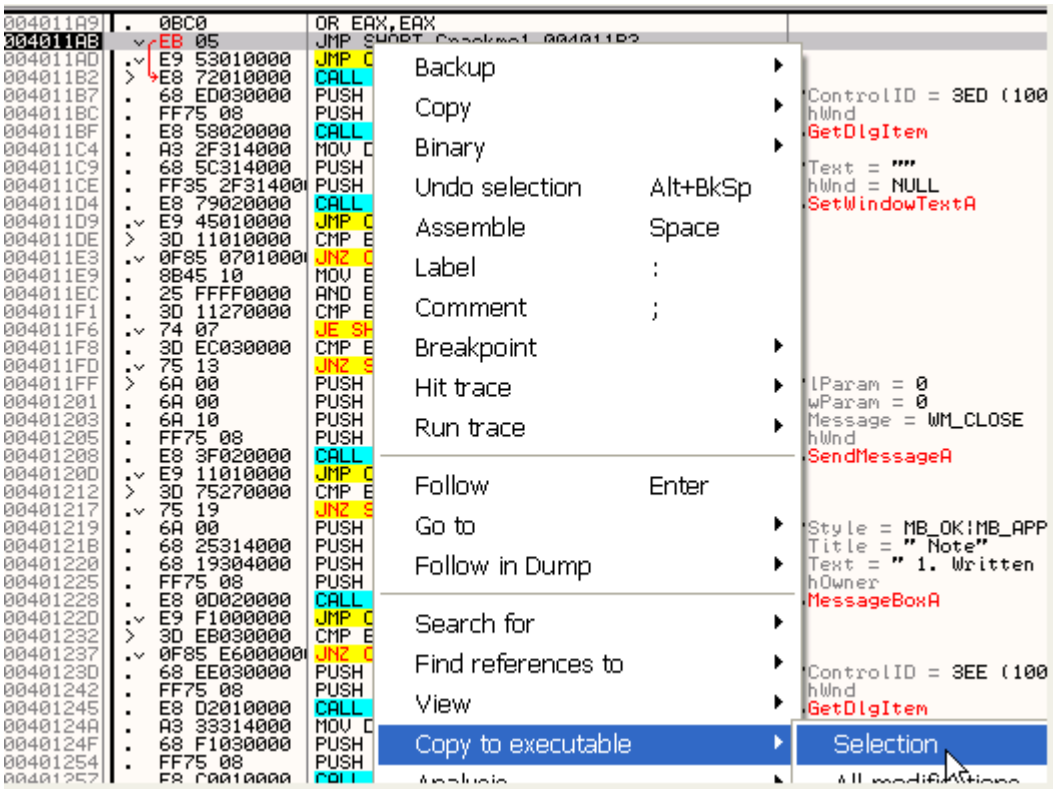


这里,我们可以将条件跳转 JE 指令改为无条件跳转 JMP 指令,即 JMP 4011B2(在该指令上单击空格键)。

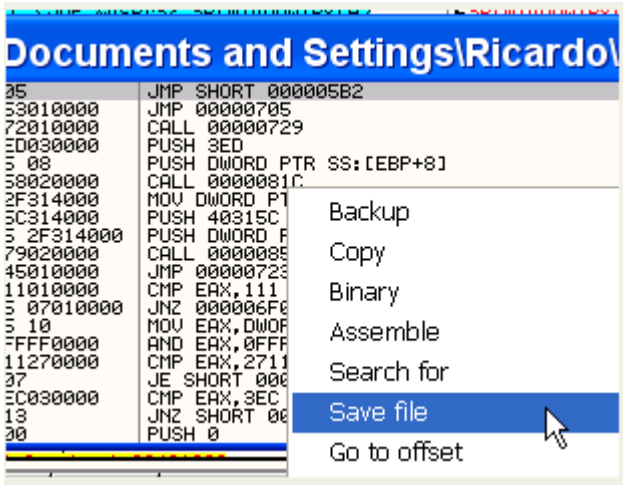


这里,我们可以看到 JMP 指令跳过了关闭程序的代码,程序将继续运行。

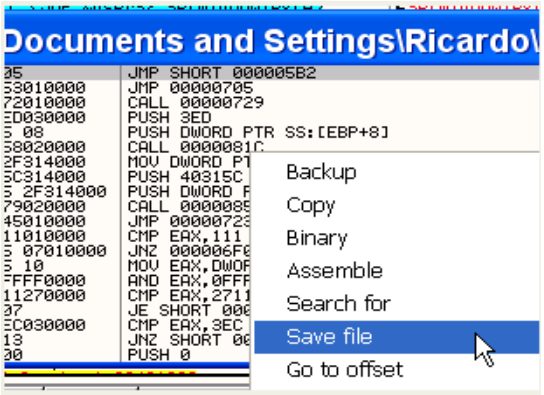
我们将该修改保存到文件并且重命名一下,我们单击鼠标右键。

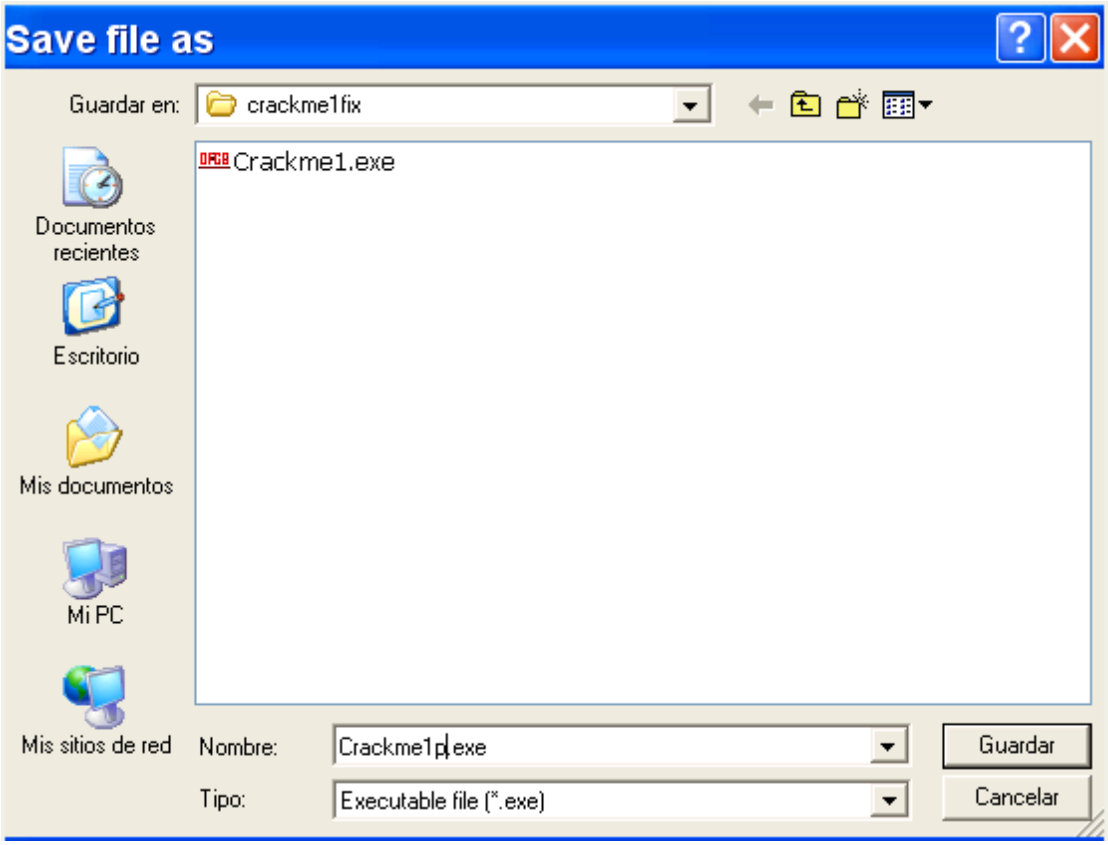


选择 Copy to executable-Selection。



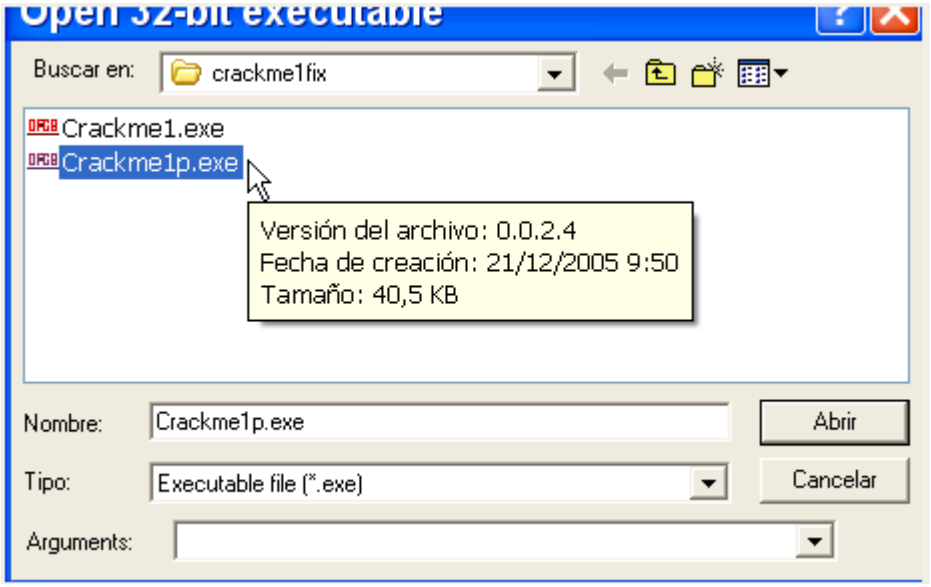
在新窗口中再次单击鼠标右键选择-Save file。

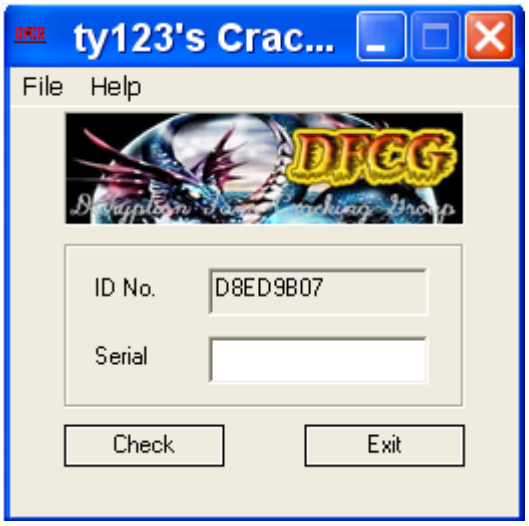




我们将该文件命名为 Crackme1p 并保存,现在我们就有了原始版本和修改版本了。

现在我们用 OD 加载刚刚修改过的程序,不设置任何断点,直接运行起来。

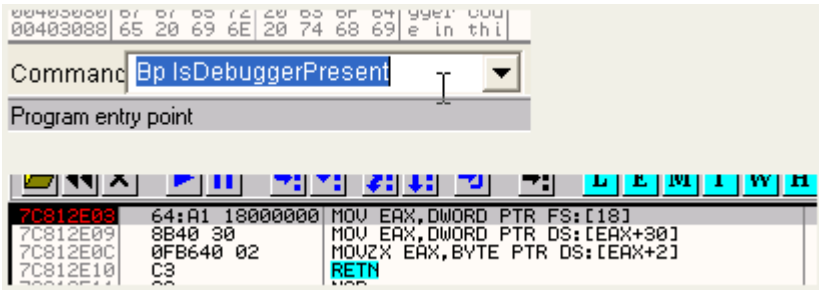




程序完美运行,所以我们知道了可以通过修改 IsDebuggerPersent 的返回值 EAX 来让程序运行,也知道了该如何给程序打补丁而不必每次都手工修改。

对于该 CrackMe 我们可以打补丁的方法让程序运行起来,但是,还有更简单的方法,直接使用插件,我们就可以避免这些麻烦。不过,我们首先还是有必要知道该插件的原理是什么。

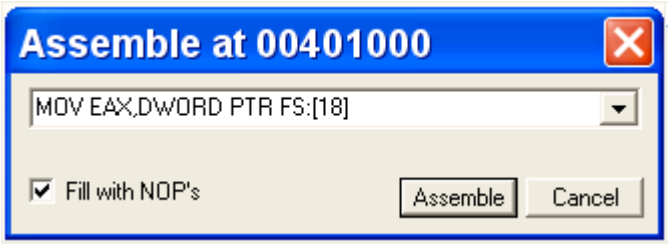
我们依然是给 IsDebuggerPersent 设置断点。

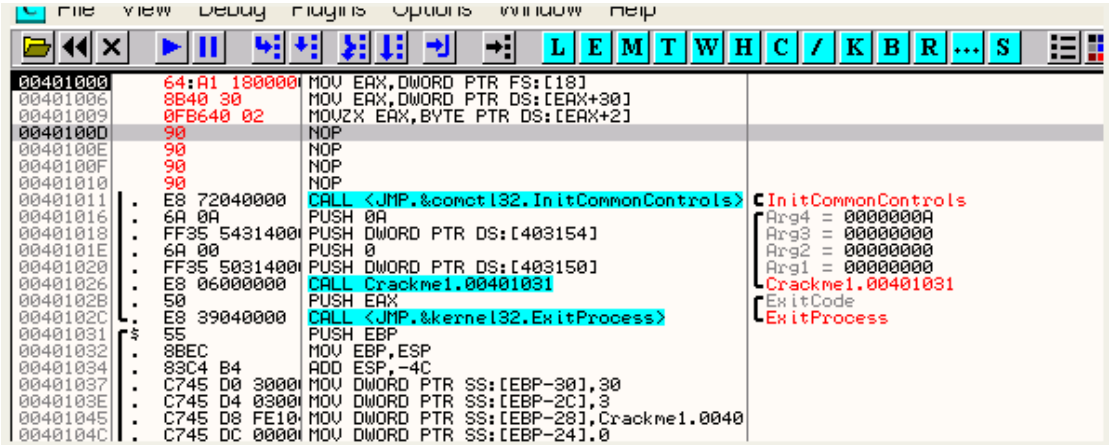


该函数由 3 条不伦不类的 MOV 指令组成,由该函数判断当前程序是否正在被调试。我们可以想到的第一点就是,当前正在被调试程序不能执行这 3 条指令,如果被调试的程序执行了这 3 条指令并且 EAX 返回 1 表示当前程序正在被调试。我们现在重新运行该 CrackMe,并且在入口点处输入这 3 条指令。

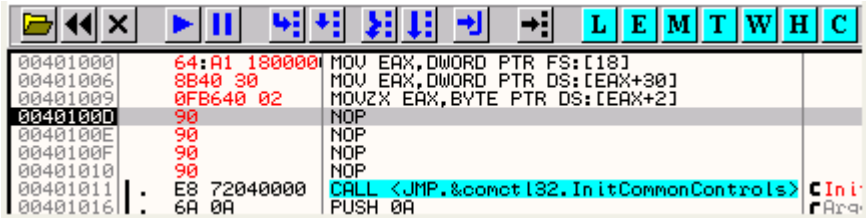
```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
```

我们可以一行一行的复制过去。

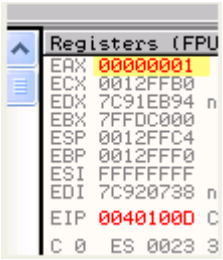




3条指令都写入了。



可以看到单步执行这 3 条指令后,跟 IsDebuggerPresent 函数一样 EAX 的值被置为了 1。



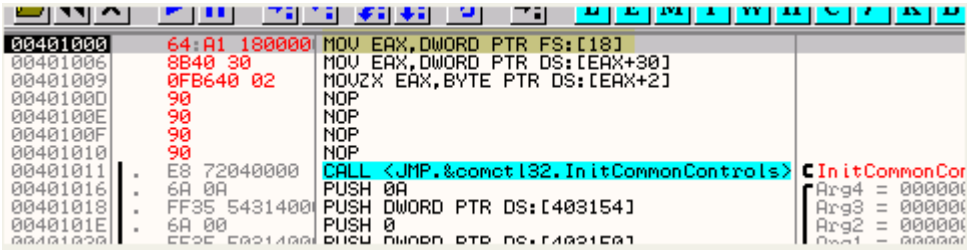
尽管当前 IsDebuggerPresent 函数并没有被调用,程序也没有中断下来,我们依然检测到了当前程序正在被调试。

因此,真正关键的不是 IsDebuggerPresent 这个 API 函数,而是该函数所包含的这 3 条指令。

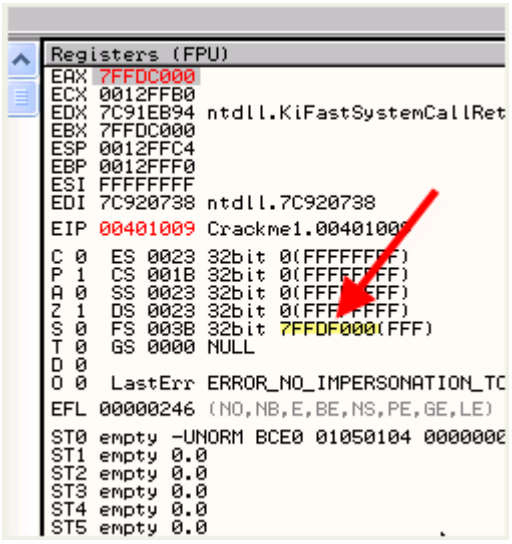
当前程序开始运行的时候,在内存的某处存放在一个特殊的标志,通过该标志来检测当前程序是否正在被调试,如果该标志为 1-当前程序正在被调试,如果该标志为 0-当前程序没有被调试。这个特殊的字节的值可以通过 IsDebuggerPresent 或者上面 3 条指令读取出来。

我们怎样才能找到这个字节呢?让我们来分析一下这 3 条指令都干了些什么。

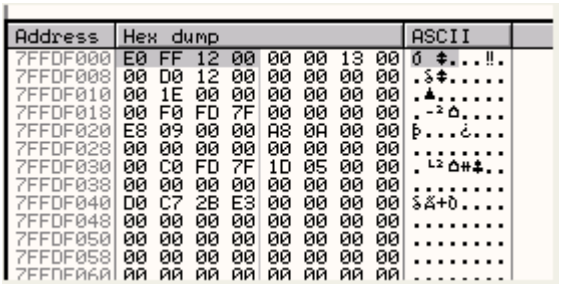
第 1 条指令:



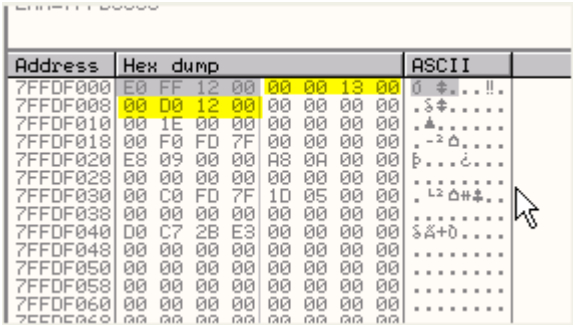
我们简单介绍一下有关的理论知识,我们首先在 OD 中看到这里。



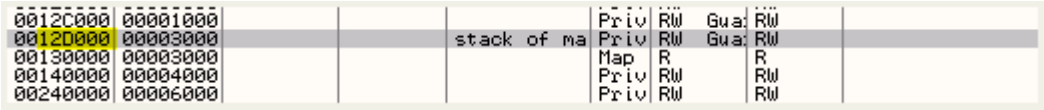
在这个窗口中我们可以看到一个非常重要的寄存器 FS 的值,不要把 FS 想的很复杂,其实该 FS 的值就是指向了一个结构体,该结构体包含了有关正在运行的程序的一些非常重要的信息。我们在数据窗口中定位到这个地址(你机器上的地址可能与我机器上的地址不同,以你机器上的值为准)。



该结构被称为 TEB 或者 TIB(线程环境块),该结构保存了有关当前程序的非常重要的信息。例如,TIB 被存储在哪里,程序堆栈从哪里开始以及到哪里结束。

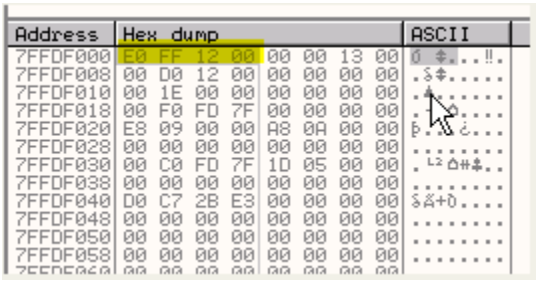


我们单击工具栏中的 M 按钮打开内存列表窗口,看看堆栈所在的区段。

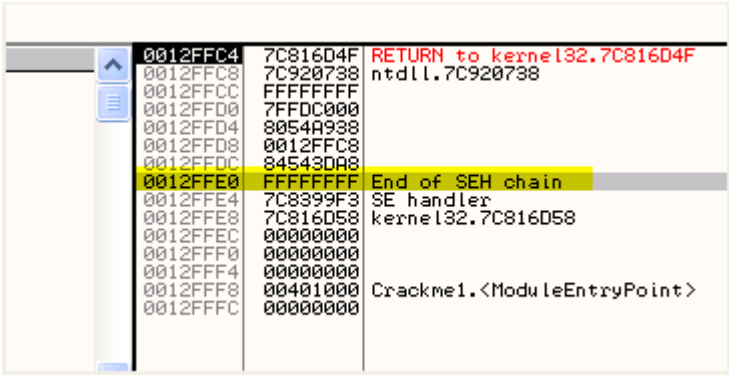


可以看到堆栈开始于 12d000,下一个区段开始于 130000。

该结构里面还有一些有趣的值值得我们研究,比如异常处理相关的东西,这里我们可以看到 12FFE0 这个地址(我的机器上是 12FFE0)。

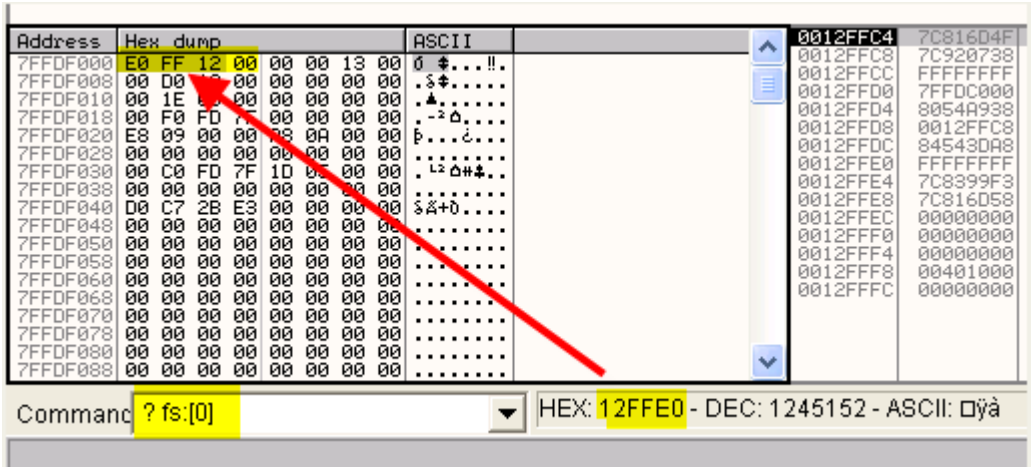


在堆栈中定位到该地址。

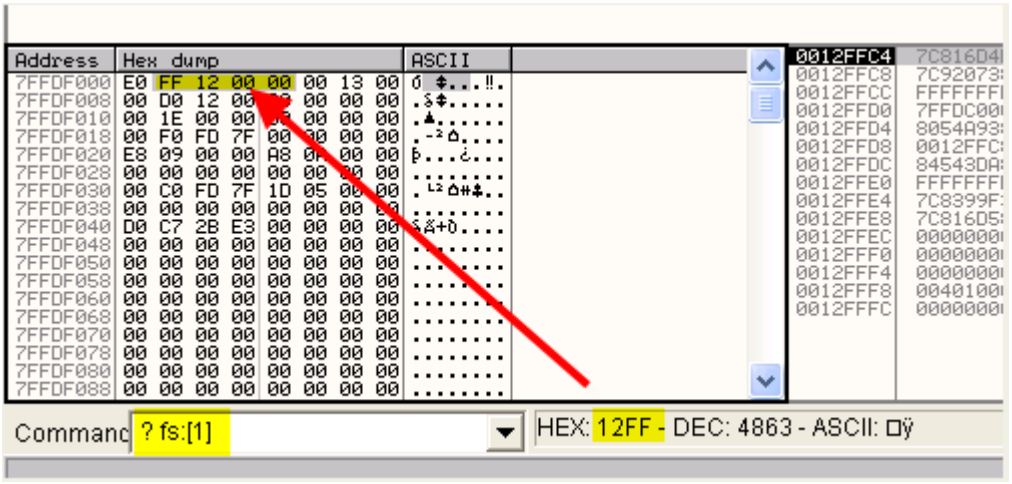


可以看到该地址被标识为 End of SEH chain。所以我们看不出什么来,但是我们可以确定该地址是与异常相关的。

对于 TIB 比较有意思的一点是可以通过相应的方式获取自身的值,例如:可以通过 FS:[0]的方式获取。



我们在命令栏中输入? FS:[0]可以获取到该值,如果我们输入? FS:[1]可以得到:



因此:

FS:[0] 我的机器上对应的值为 7FFDF000

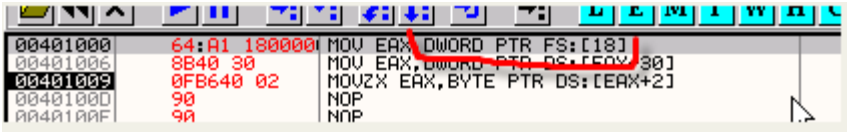
FS:[1] 我的机器上对应的值为 7FFDF001

.....

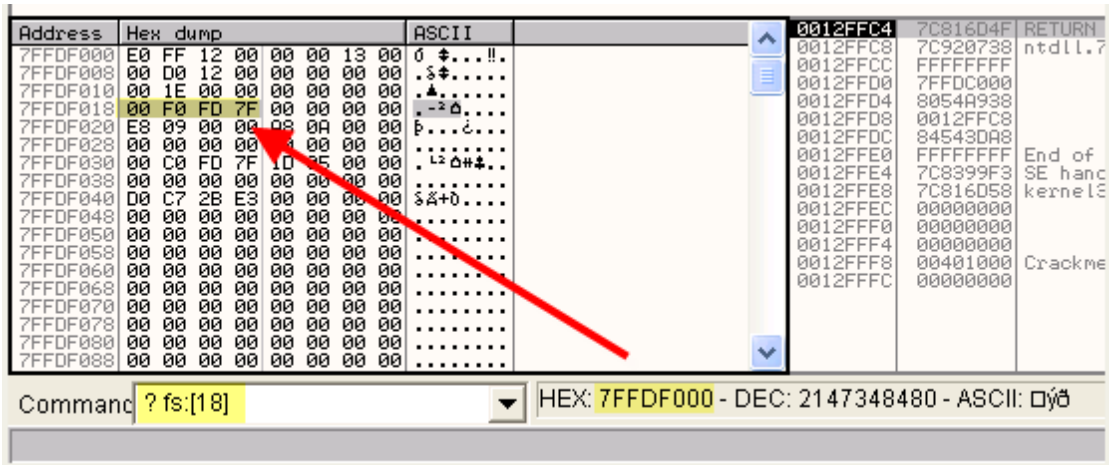
.....

FS:[18] 我的机器上对应的值为 7FFDF018

如果我们还记得,该值其实就是 IsDebuggerPresent 这个 API 函数第 1 条指令读取的值。

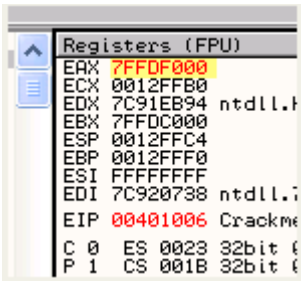


我们可以看到 FS:[18]的值为 77FFDF018。

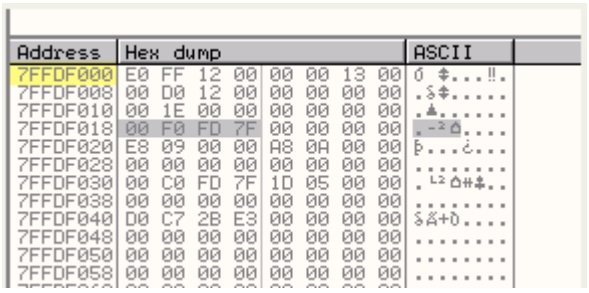


这里就保存着 TIB 的值,FS:[18]标识的就是 TIB 的指针。

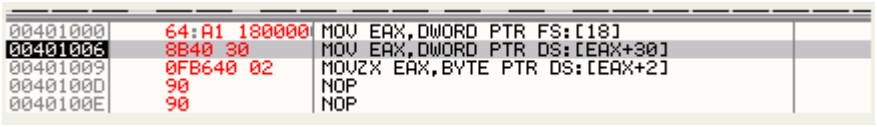
因此,第 1 条指令执行后,EAX 就保存了 TIB 的指针,该值在不同的机器上可能会不同。我们按 F8 键执行该指令。



好了,现在 EAX 就保存了 TIB 的指令。

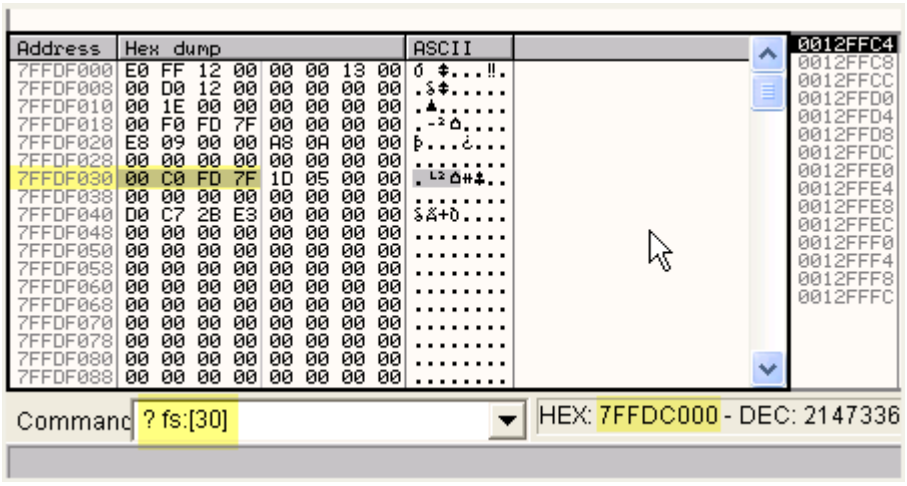


现在我们看下一条指令。

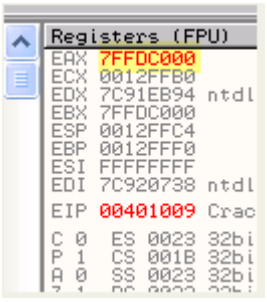


这里 EAX + 30,我的机器上计算的结果为 7FFDF000 + 30 = 7FFDF030。

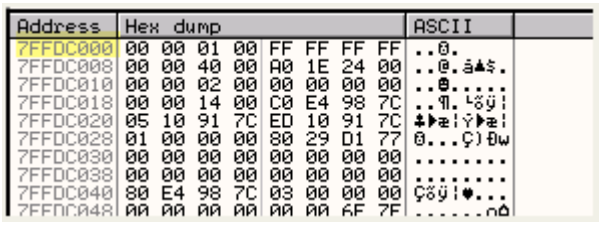
对应为 FS:[30]。



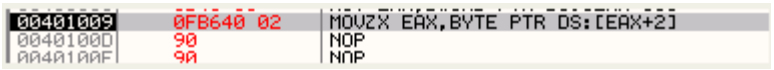
这里 EAX 将保存 7FFDF030 或者 FS:[30]内存单元中的值,在我的机器上该值为 7FFDC000,不要与前面的 TIB 的值弄混淆了。



这是一个指针,指向了别的东西,我们在数据窗口中定位该地址。



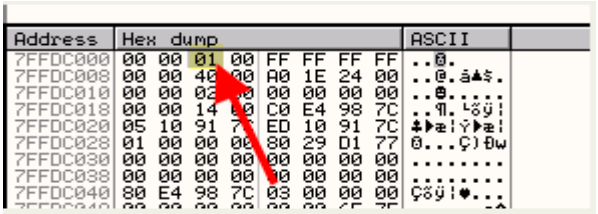
最后一条指令:



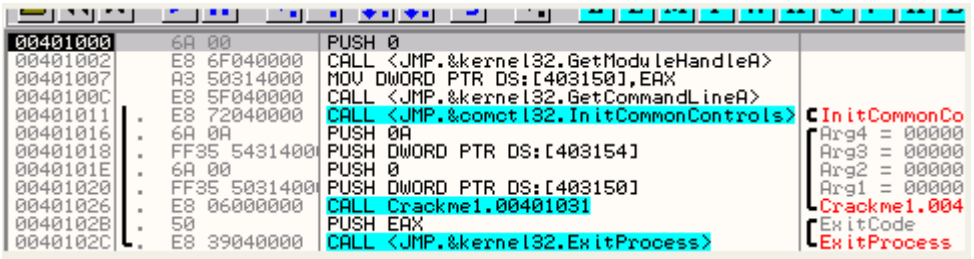
EAX + 2 即:

7FFDC000 + 2= 7FFDC002

7FFDC002(我的机器上)这个地址处的字节值被保存到了 EAX 中,这就是我们需要找的字节值。

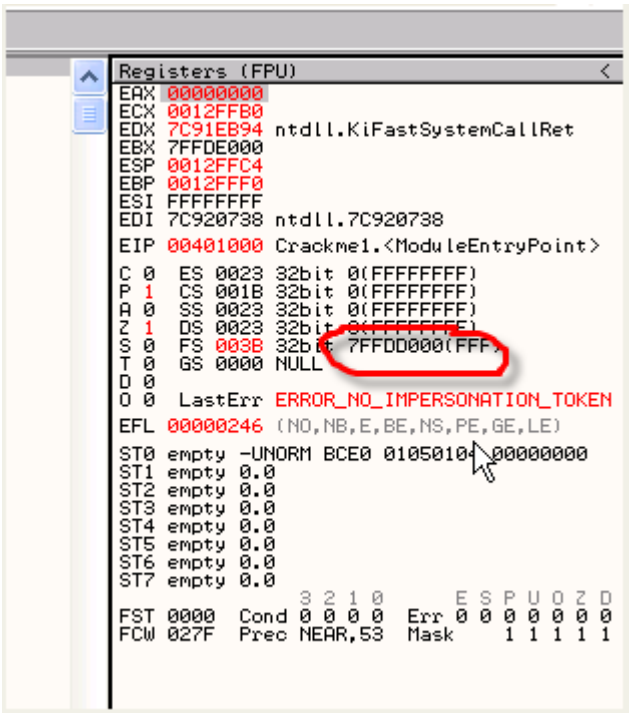


这个特殊字节值可以通过 IsDebuggerPresent 这个 API 函数读取到。可能在你的机器的该特殊字节保存的地址与我机器的地址不同,但是你可以通过上面的方法很容易的定位到该字节。用 OD 重新加载该 CrackMe。

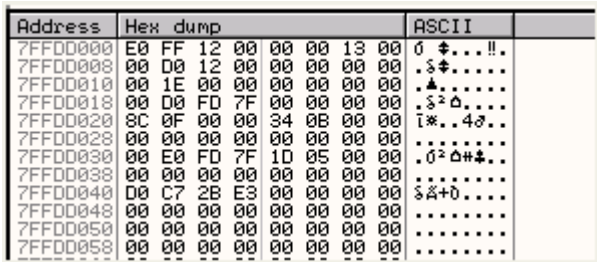


这里我们跟上面一样来手工定位到该特殊字节。

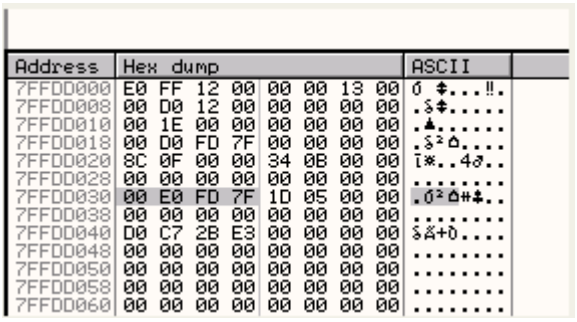
首先找到 FS 寄存器中保存的 TIB 的指针。



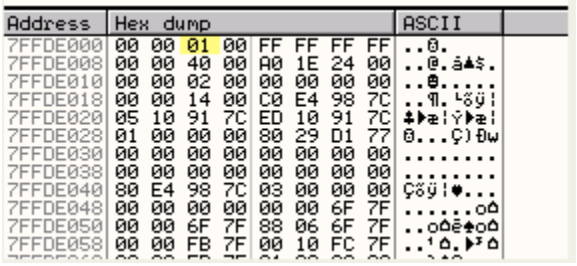
接着在数据窗口中定位到 TIB。



我们找到 FS:[30]即 TIB 的指针偏移 30,我们找到的内容如下:

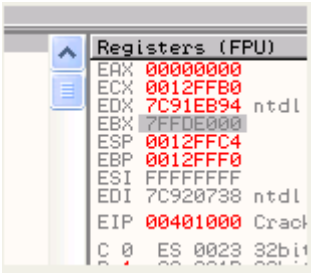


该内容为 7FFDE000,我们继续在数据窗口中定位到该地址。

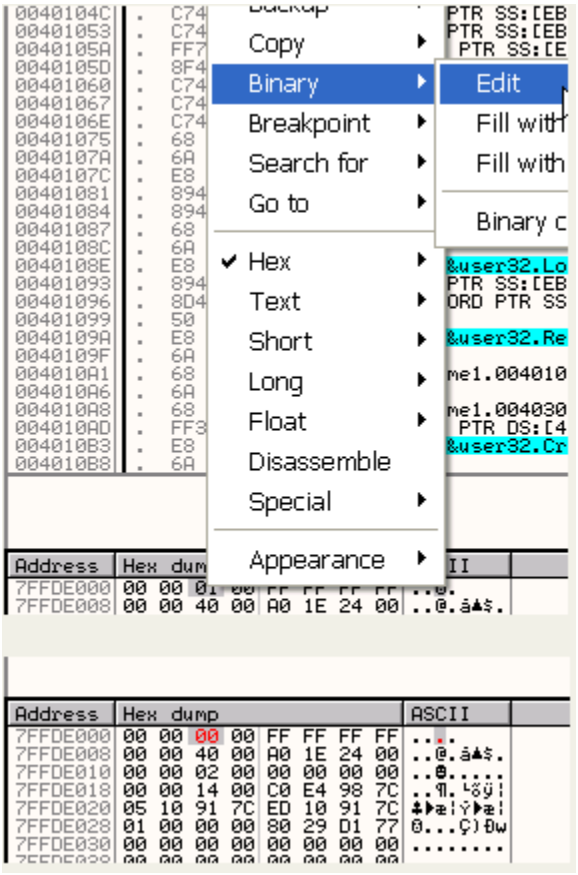


偏移 2 处的字节就是我们需要找的特殊字节。(我们可以看到 EBX 的值也指向了该区域,所以如果程序启动的时候,EBX 就指向了

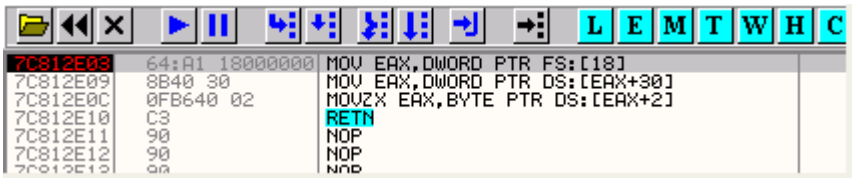
该区域的话,那么 EBX = FS:[30])。



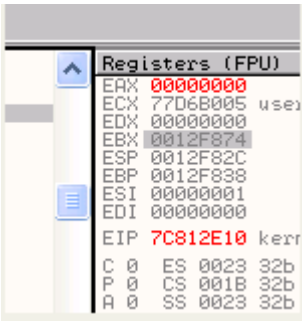
好了,该方法在程序刚刚启动的时候就可以定位到该特殊字节。现在我们将该字节修改为 0。



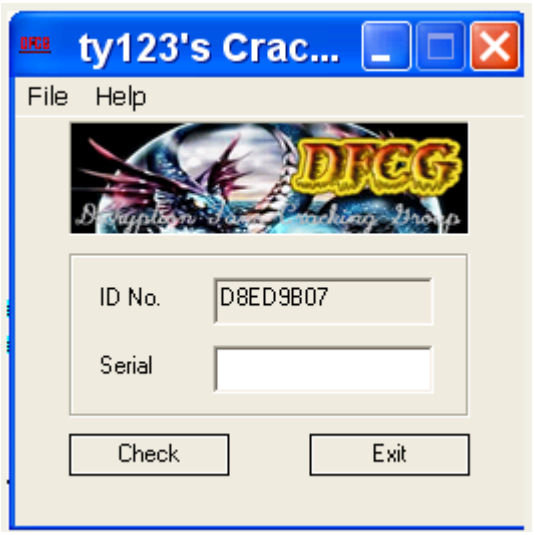
现在我们给 IsDebuggerPresent 设置一个断点。运行起来看看接下来会发生什么。



执行到返回。



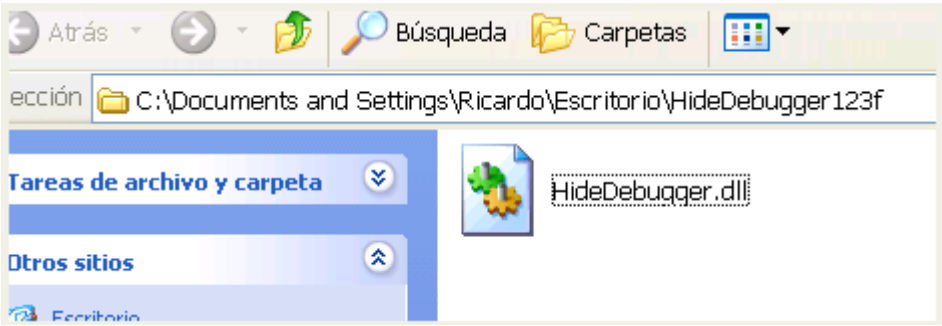
可以看到 EAX 的值为 0,因为我们修改了那个特殊字节的值,所以当前程序认为它没有被调试。



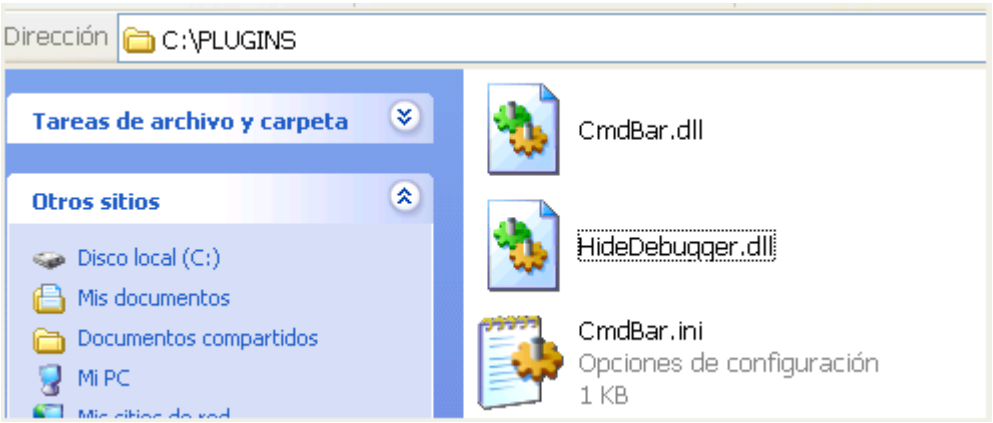
可以看到,程序正常运行起来了。

当然,有很多插件,为我们做了上面的事情,HideDebugger1.24 就是其中之一。

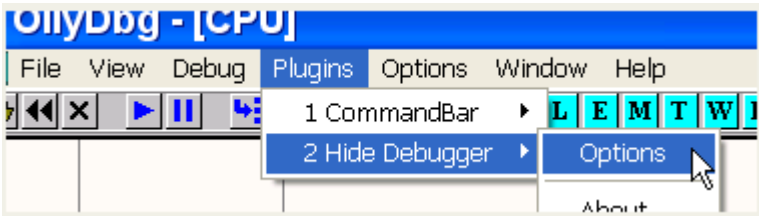
下载该插件,然后将 DLL 文件复制到 OD 的插件目录下。



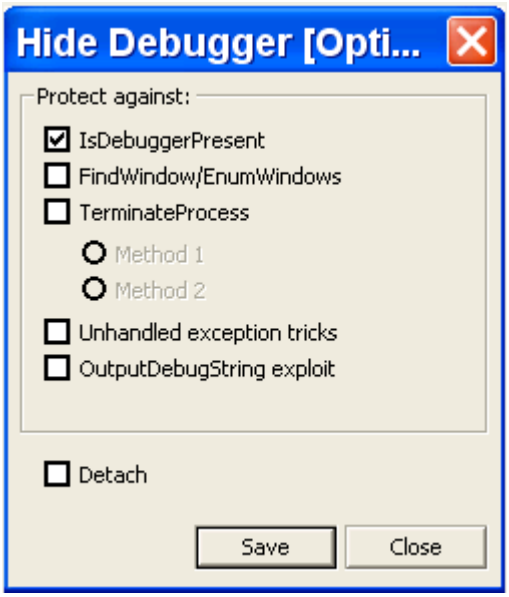
复制,嘿嘿。



重新重启 OD。

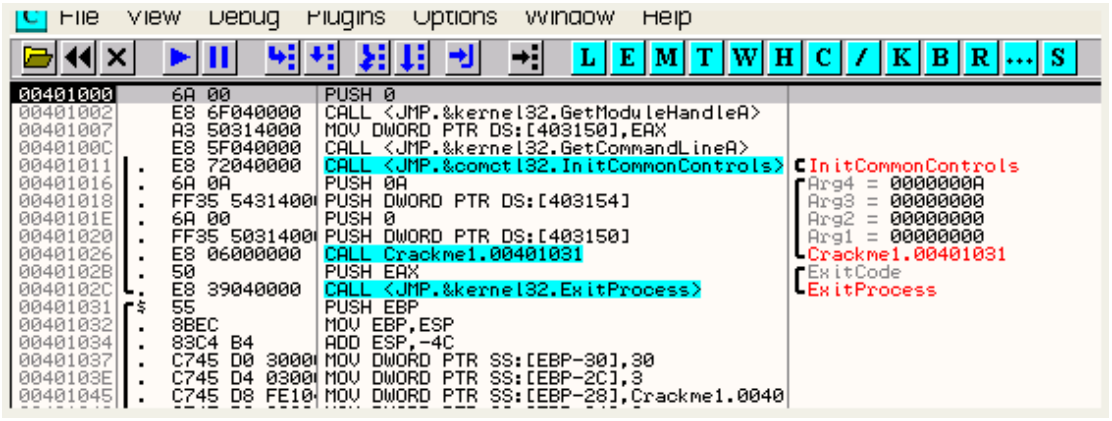


选择该插件并进行设置。

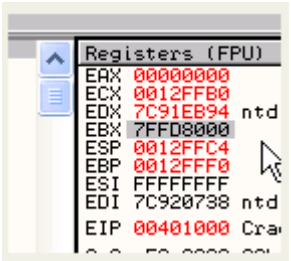


勾选中 IsDebuggerPresent,其他的选项我们后面详细介绍到了再勾选。

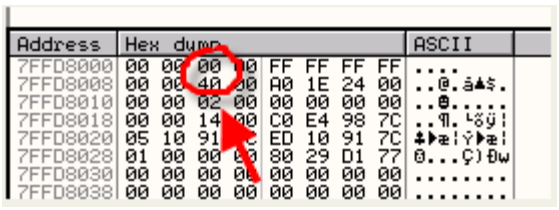
单击 Save。



我们重新加载 CrackMe,可以注意到 EBX 指向得到区域就是特殊字节的区域。



我们在 EBX 的值上面单击鼠标右键选择-Follow in Dump。



可以看到该插件将该特殊字节清零了,从而绕过这个保护。可能这个过程比较简单,但是我认为了解 IsDebuggerPresent 反调试的原理以及如何绕过这个检测的来龙去脉对大家是大有裨益的。