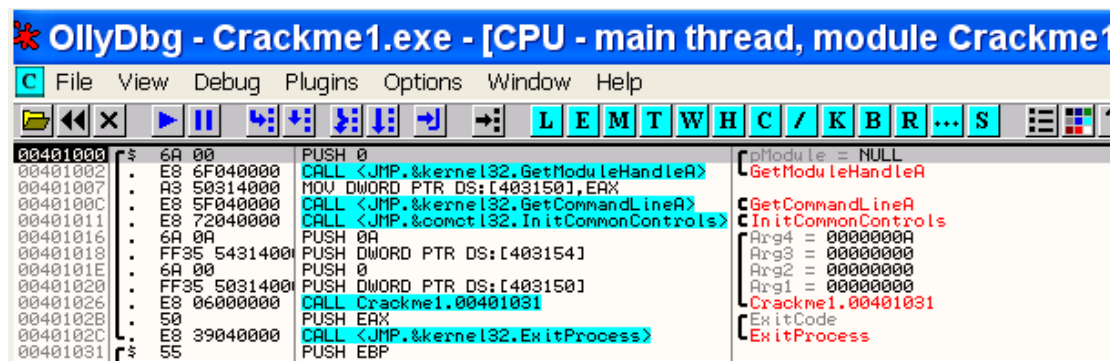


第十九章-OllyDbg 反调试之 IsDebuggerPresent

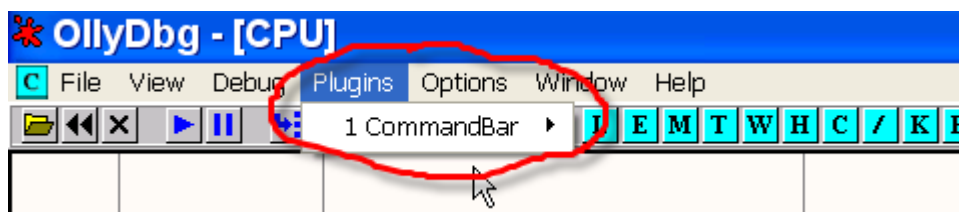
本章开始,我们将讨论反调试的相关话题,包括手工以及通过 OD 插件来绕过对应的反调试的技巧。很多程序会检测自身是否正在被调试,如果检测到正在被调试的话,就会结束自身进程或者不按常规流程运行。所以绕过程序对 OD 的检测是很有必要的。

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。

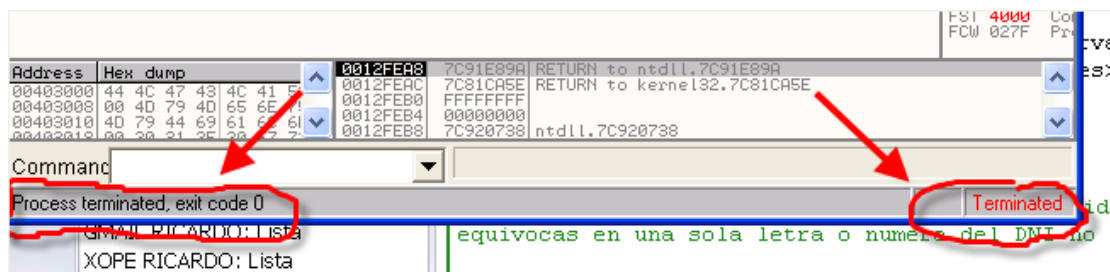
这里,我们使用 Crackme1.exe 来讲解,用 OD 加载它。



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?

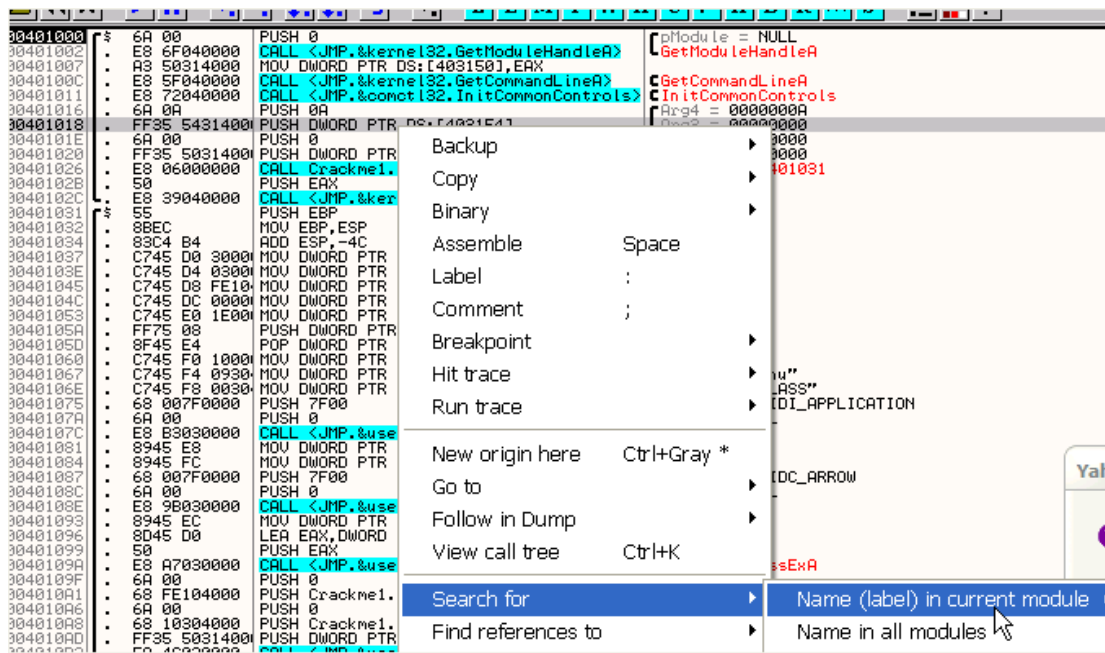


如果我们按 F9 键让程序运行起来,我们会发现并没有弹出 CrackMe 窗口,程序直接终止了。



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。

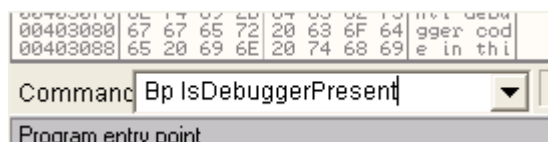


我们看看 API 函数列表

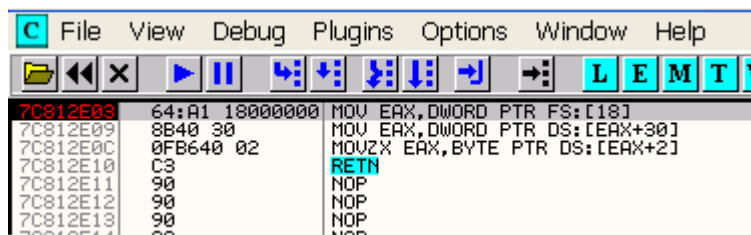
Address	Section	Type	Name	Comment
00402050	.rdata	Import	user32.CreateDialogParamA	
00402058	.rdata	Import	user32.DefWindowProcA	
0040205A	.rdata	Import	user32.DestroyWindow	
0040205C	.rdata	Import	user32.DispatchMessageA	
0040205E	.rdata	Import	kernel32.ExitProcess	
00402060	.rdata	Import	kernel32.GetCommandLineA	
00402062	.rdata	Import	user32.GetDlgItem	
00402064	.rdata	Import	user32.GetMessageA	
00402066	.rdata	Import	kernel32.GetModuleHandleA	
00402068	.rdata	Import	kernel32.GetVolumeInformationA	
0040206A	.rdata	Import	user32.GetWindowTextA	
0040206C	.rdata	Import	comctl32.InitCommonControls	
0040206E	.rdata	Import	kernel32.IsDebuggerPresent	
00402070	.rdata	Import	user32.LoadCursorA	
00402072	.rdata	Import	user32.LoadIconA	
00402074	.rdata	Import	user32.MessageBoxA	
00401000	.text	Export	<ModuleEntryPoint>	
00402078	.rdata	Import	user32.PostQuitMessage	
0040207A	.rdata	Import	user32.RegisterClassExA	
0040207C	.rdata	Import	user32.SendMessageA	
0040207E	.rdata	Import	user32.SetWindowTextA	
00402080	.rdata	Import	user32.ShowWindow	
00402082	.rdata	Import	user32.TranslateMessage	
00402084	.rdata	Import	user32.UpdateWindow	
00402086	.rdata	Import	user32.wsprintfA	

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。



0012F82C	5B15B1BA	CALL to IsDebuggerPresent from uxtheme.5B15B1B4
0012F830	00000000	
0012F834	77D6B005	user32.77D6B005
0012F838	0012F858	
0012F83C	5B15AF3C	RETURN to uxtheme.5B15AF3C from uxtheme.5B15AEF4

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

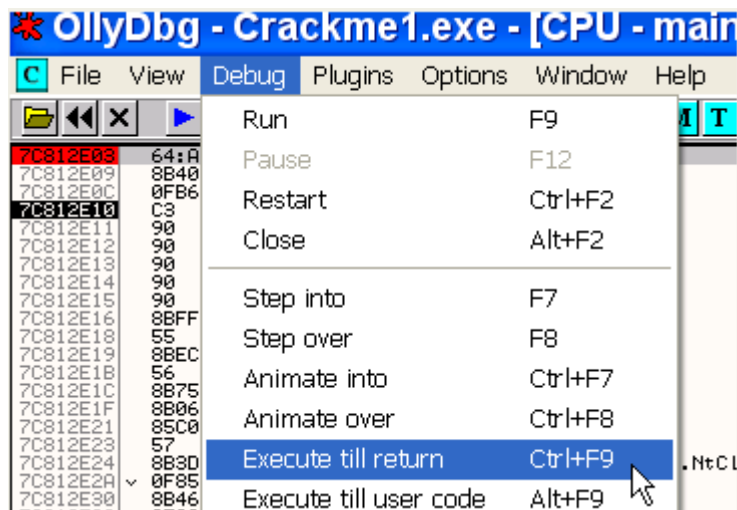
[OutputDebugString](#)

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。

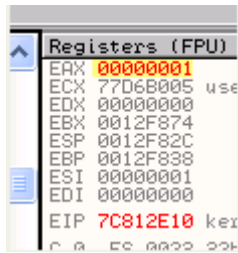
并且该函数是被 Kernel32.dll 导出的,该函数没有参数,如果当前程序正在被调试的话,返回值为 1,没有被调试的话,返回值为 0。

这是非常重要的信息,我们执行到返回,看看返回值是多少。



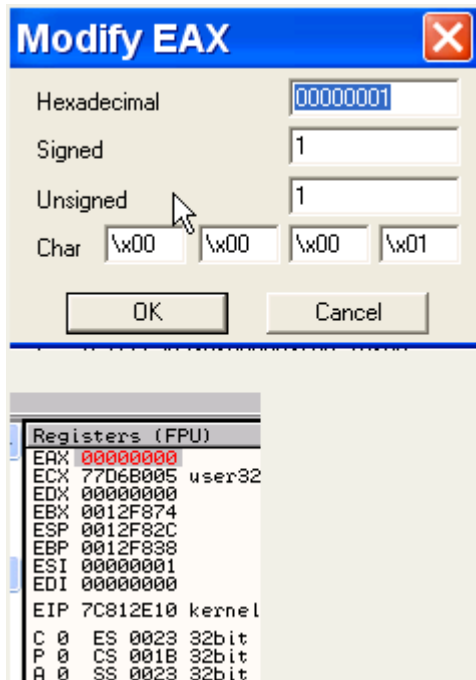
我们停在了 RET 指令处,看看寄存器的情况。

7C812E01	90	NOP
7C812E02	90	NOP
7C812E03	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]
7C812E09	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
7C812E0C	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
7C812E10	C3	RETN
7C812E11	90	NOP
7C812E12	90	NOP
7C812E13	90	NOP

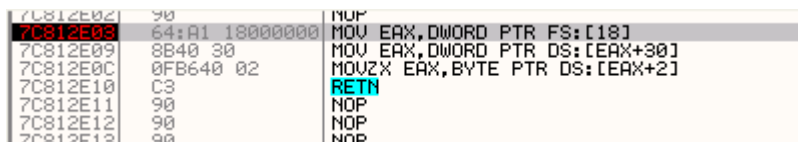


EAX 的值变成了粉红色,表示 EAX 的值被修改过,跟其他 API 函数一样,IsDebuggerPresent 的返回值也保存在 EAX 中,这里 EAX 为 1,表示当前程序正在被调试。

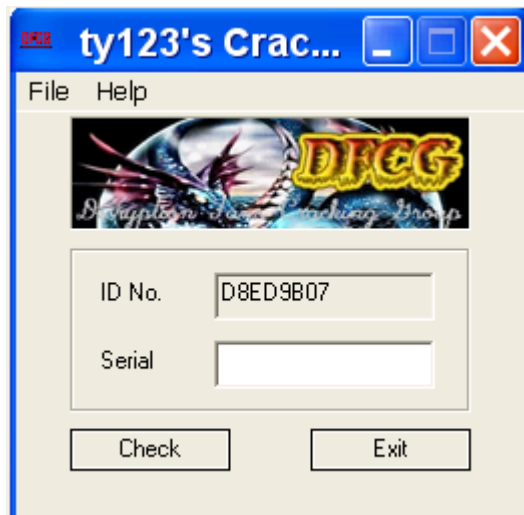
我们尝试手动将 EAX 修改为 0,表示当前程序没有被调试,看看会发生什么。



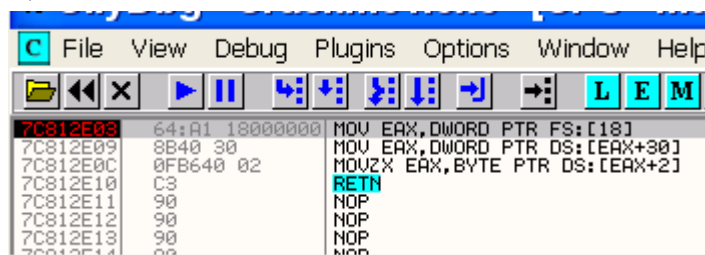
运行起来。



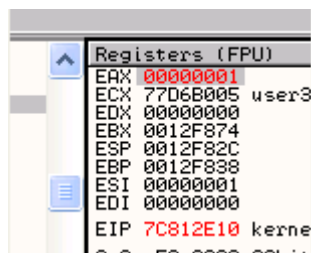
再次断在了该 API 函数入口处,我们执行到返回,然后将 EAX 的值修改为 0。



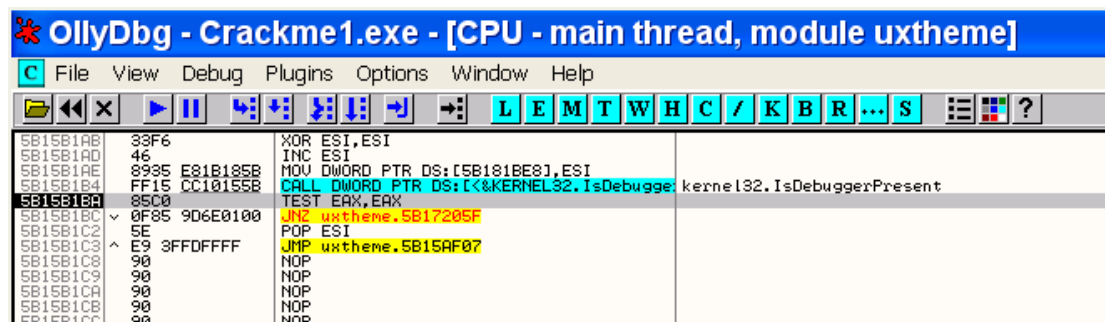
我们可以看到该程序启动时的检测是基于 IsDebuggerPresent 这个 API 函数的,我们重新运行 CrackMe,会断在 IsDebuggerPresent 处,相应返回值的解释可以参考 MSDN。



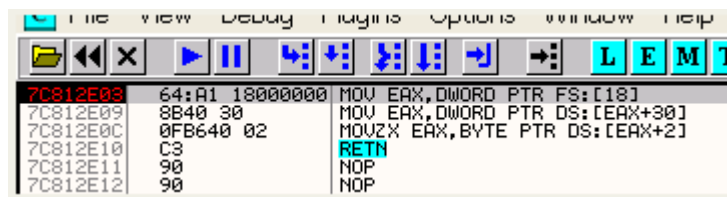
我们可以执行到返回,也可以直接 F8 单步到 RET 指令,因为这个函数很简短,就几条指令。



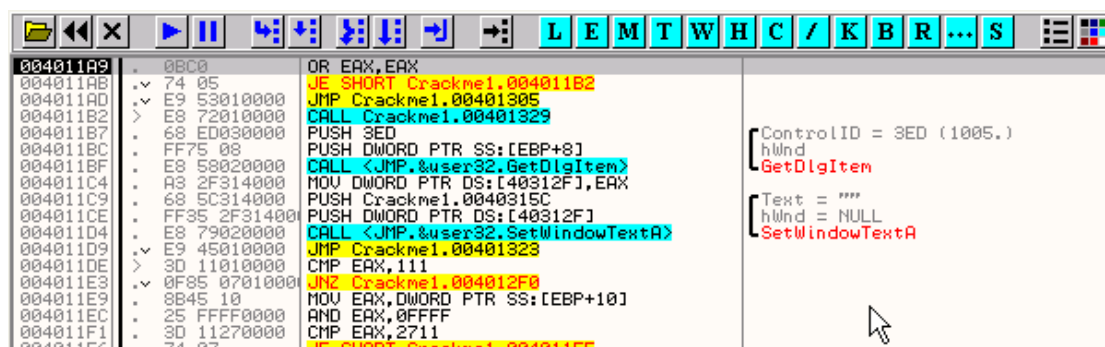
这里到了 RET 指令处,EAX 的值为 1,按 F8 键返回。



这里我们可以看到,IsDebuggerPresent 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。

004011A9	0BC0	OR EAX,EAX	
004011AB	74 05	JE SHORT Crackme1.004011B2	
004011AD	E9 53010000	JMP Crackme1.00401305	
004011B2	E8 72010000	CALL Crackme1.00401329	
004011B7	68 ED030000	PUSH 3ED	ControlID = 3ED (1005.)
004011BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011BF	E8 58020000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
004011C4	A3 2F314000	MOV DWORD PTR DS:[40312F],EAX	
004011C9	68 5C314000	PUSH Crackme1.0040315C	Text = ""
004011CE	FF35 2F314000	PUSH DWORD PTR DS:[40312F]	hWnd = NULL
004011D4	E8 79020000	CALL <JMP.&user32.SetWindowTextA>	SetWindowTextA
004011D9	E9 45010000	JMP Crackme1.00401323	
004011DE	3D 11010000	CMP EAX,111	
004011E3	0F85 07010000	JNZ Crackme1.004012F0	
004011E9	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	
004011EC	25 FFFF0000	AND EAX,0FFFF	
004011F1	3D 11270000	CMP EAX,2711	
004011F6	74 07	JE SHORT Crackme1.004011FF	
004011FA	3D EC030000	CMP EAX,3EC	

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。

004011A9	0BC0	OR EAX,EAX	
004011AB	74 05	JE SHORT Crackme1.004011B2	
004011AD	E9 53010000	JMP Crackme1.00401305	
004011B2	E8 72010000	CALL Crackme1.00401329	
004011B7	68 ED030000	PUSH 3ED	ControlID = 3ED (1005.)
004011BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011BF	E8 58020000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
004011C4	A3 2F314000	MOV DWORD PTR DS:[40312F],EAX	
004011C9	68 5C314000	PUSH Crackme1.0040315C	Text = ""
004011CE	FF35 2F314000	PUSH DWORD PTR DS:[40312F]	hWnd = NULL
004011D4	E8 79020000	CALL <JMP.&user32.SetWindowTextA>	SetWindowTextA
004011D9	E9 45010000	JMP Crackme1.00401323	
004011DE	3D 11010000	CMP EAX,111	
004011E3	0F85 07010000	JNZ Crackme1.004012F0	
004011E9	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	
004011EC	25 FFFF0000	AND EAX,0FFFF	
004011F1	3D 11270000	CMP EAX,2711	
004011F6	74 07	JE SHORT Crackme1.004011FF	
004011F8	3D EC030000	CMP EAX,3EC	
004011FD	75 13	JNZ SHORT Crackme1.00401212	
004011FF	6A 00	PUSH 0	lParam = 0
00401201	6A 00	PUSH 0	wParam = 0
00401203	6A 10	PUSH 10	Message = WM_CLOSE
00401205	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401208	E8 3F020000	CALL <JMP.&user32.SendMessageA>	SendMessageA
0040120D	E9 11010000	JMP Crackme1.00401323	
00401212	3D 75270000	CMP EAX,2775	
00401217	75 19	JNZ SHORT Crackme1.00401232	
00401219	6A 00	PUSH 0	Style = MB_OK MB_APPLMODAL
0040121B	68 25314000	PUSH Crackme1.00403125	Title = " Note"
00401220	68 19304000	PUSH Crackme1.00403019	Text = " 1. Written by RadASM, thanks
00401225	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401228	E8 0D020000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
0040122D	E9 F1000000	JMP Crackme1.00401323	
00401232	3D EB030000	CMP EAX,3EB	
00401237	0F85 E6000000	JNZ Crackme1.00401323	
0040123D	68 EE030000	PUSH 3EE	ControlID = 3EE (1006.)
00401242	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401245	E8 D2010000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
0040124A	A3 33314000	MOV DWORD PTR DS:[403133],EAX	
0040124F	68 F1030000	PUSH 3F1	ControlID = 3F1 (1009.)
00401254	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401257	E8 C0010000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
0040125C	A3 37314000	MOV DWORD PTR DS:[403137],EAX	
00401305=Crackme1.00401305			

继续跟,我们到了这里。

004012FF	937D 0C 02	CMP DWORD PTR SS:[EBP+C],2	
00401303	75 09	JNZ SHORT Crackme1.0040130E	
00401305	6A 00	PUSH 0	ExitCode = 0
00401307	E8 34010000	CALL <JMP.&user32.PostQuitMessage>	PostQuitMessage
0040130C	EB 15	JMP SHORT Crackme1.00401323	
0040130E	FF75 14	PUSH DWORD PTR SS:[EBP+14]	lParam
00401311	FF75 10	PUSH DWORD PTR SS:[EBP+10]	wParam

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:

00401305	>	6A 00	PUSH 0	ExitCode = 0
00401307	.	E8 34010000	CALL <JMP.&user32.PostQuitMessage>	PostQuitMessage
0040130C	>	EB 15	JMP SHORT Crackme1.00401323	
0040130E	>	FF75 14	PUSH DWORD PTR SS:[EBP+14]	lParam
00401311	.	FF75 10	PUSH DWORD PTR SS:[EBP+10]	wParam
00401314	.	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	Message
00401317	.	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
0040131A	.	E8 EB000000	CALL <JMP.&user32.DefWindowProcA>	DefWindowProcA
0040131F	.	C9	LEAVE	
00401320	.	C2 1000	RETN 10	
00401323	>	33C0	XOR EAX,EAX	
00401325	.	C9	LEAVE	
00401326	.	C2 1000	RETN 10	
00401329	⌵	55	PUSH EBP	
0040132A	.	8BEC	MOV EBP,ESP	

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)

0040101E	.	5A 00	PUSH 0	hrgz = 00000000
00401020	.	FF35 50314000	PUSH DWORD PTR DS:[403150]	Arg1 = 00400000
00401026	.	E8 06000000	CALL Crackme1.00401031	Crackme1.00401031
0040102B	.	50	PUSH EAX	ExitCode
0040102C	>	E8 39040000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
00401031	⌵	55	PUSH EBP	
00401032	.	8BEC	MOV EBP,ESP	
00401034	.	83C4 B4	ADD ESP,-4C	
00401037	.	C745 D0 3000	MOV DWORD PTR SS:[EBP-30],30	

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。

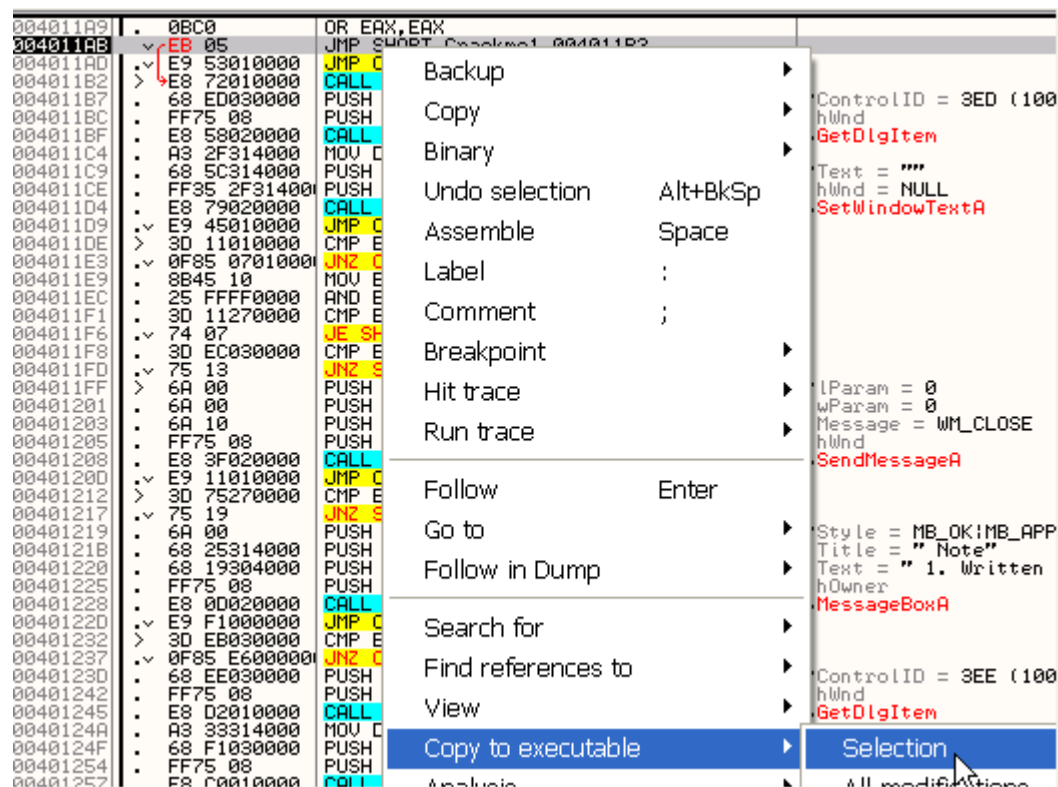
004011A9	.	0BC0	OR EAX,EAX	
004011AB	>	74 05	JE SHORT Crackme1.004011B2	
004011AD	>	E9 53010000	JMP Crackme1.00401305	
004011B2	>	E8 72010000	CALL Crackme1.00401329	
004011B7	.	68 ED030000	PUSH 3ED	ControlID = 3ED (1005)
004011BC	.	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011BF	.	E8 58020000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem

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)。

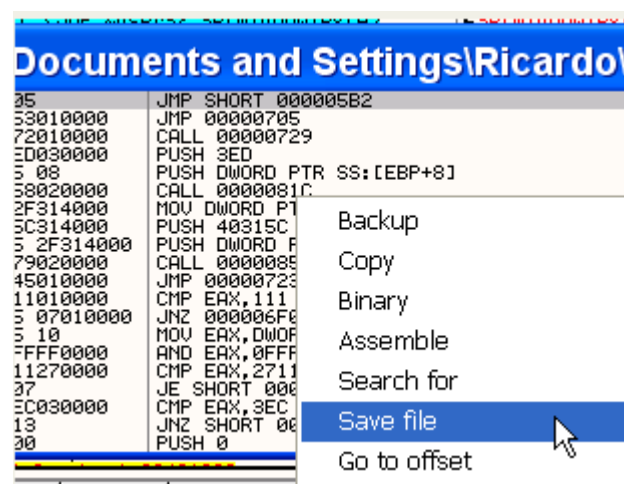


这里,我们可以看到 JMP 指令跳过了关闭程序的代码,程序将继续运行。

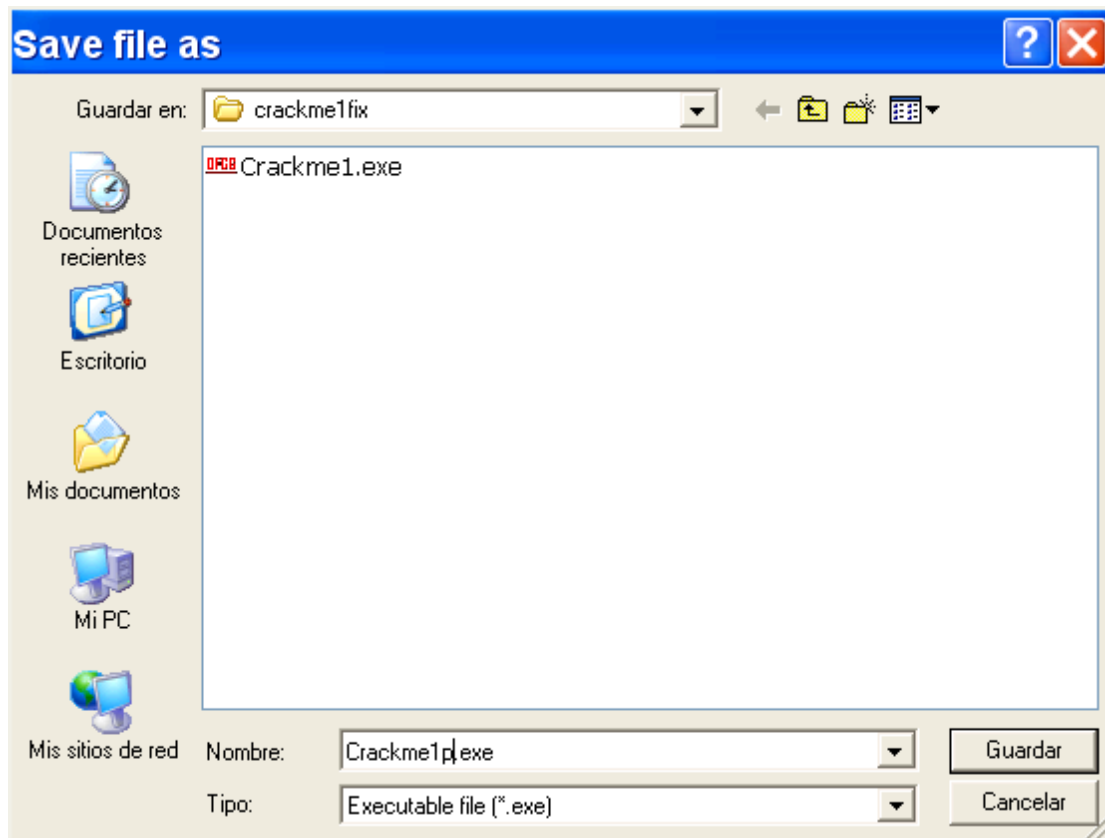
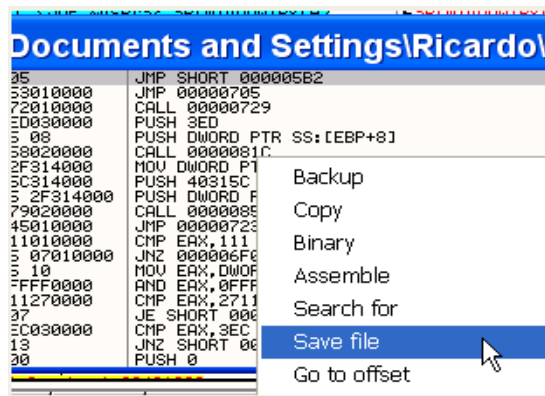
我们将该修改保存到文件并且重命名一下,我们单击鼠标右键。



选择 Copy to executable-Selection。

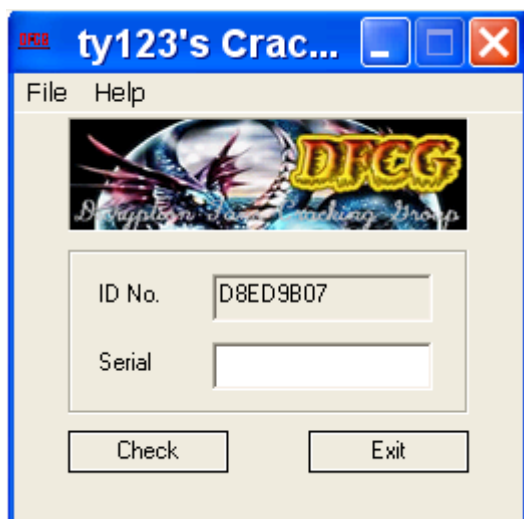
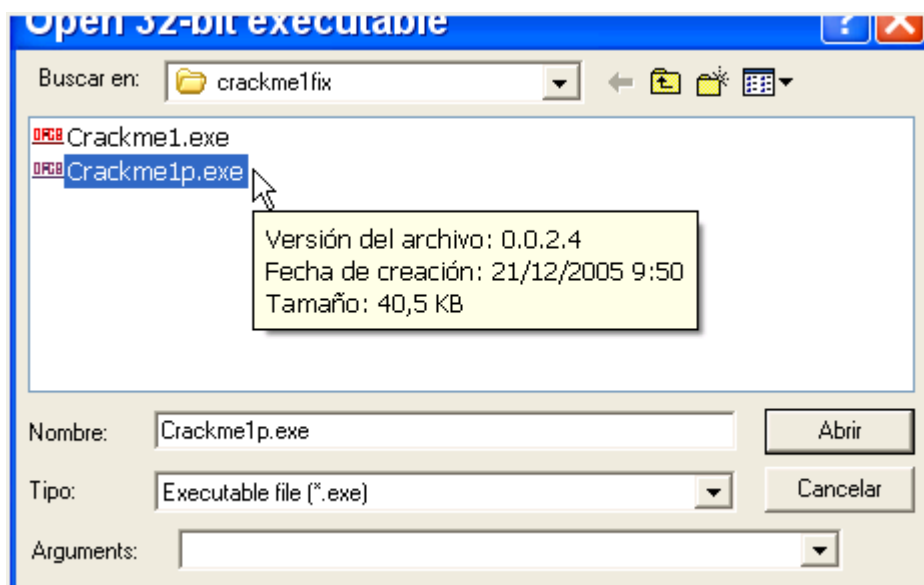


在新窗口中再次单击鼠标右键选择-Save file。



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。

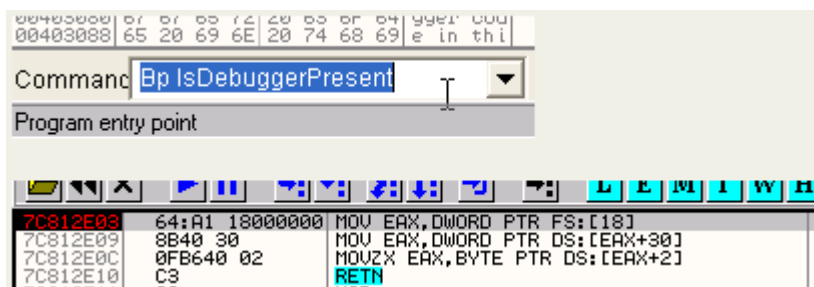
现在我们用 OD 加载刚刚修改过的程序,不设置任何断点,直接运行起来。



程序完美运行,所以我们知道了可以通过修改 `IsDebuggerPresent` 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。

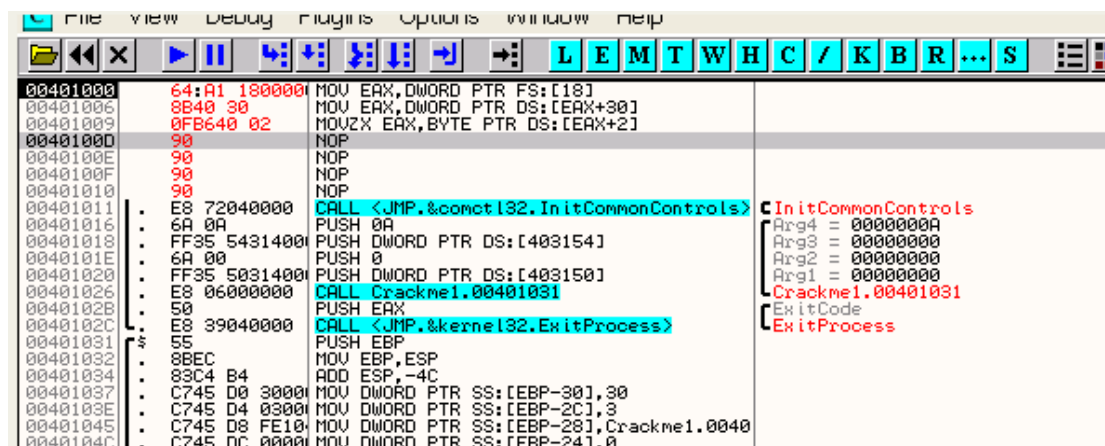
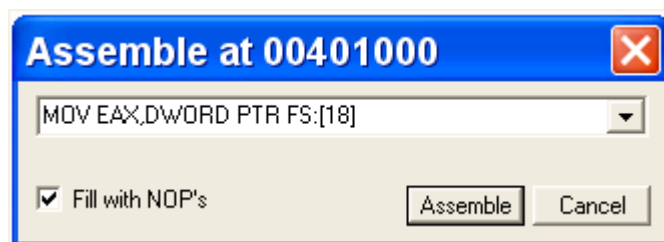
我们依然是给 `IsDebuggerPresent` 设置断点。



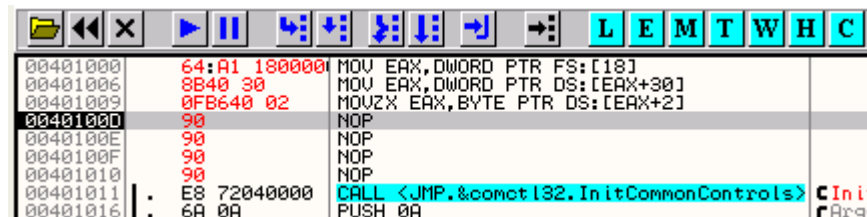
该函数由 3 条不伦不类的 `MOV` 指令组成,由该函数判断当前程序是否正在被调试。我们可以想到的第一点就是,当前正在被调试程序不能执行这 3 条指令,如果被调试的程序执行了这 3 条指令并且 `EAX` 返回 1 表示当前程序正在被调试。我们现在重新运行该 `CrackMe`,并且在入口点处输入这 3 条指令。

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

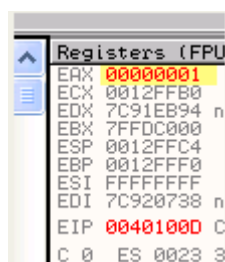
我们可以一行一行的复制过去。



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

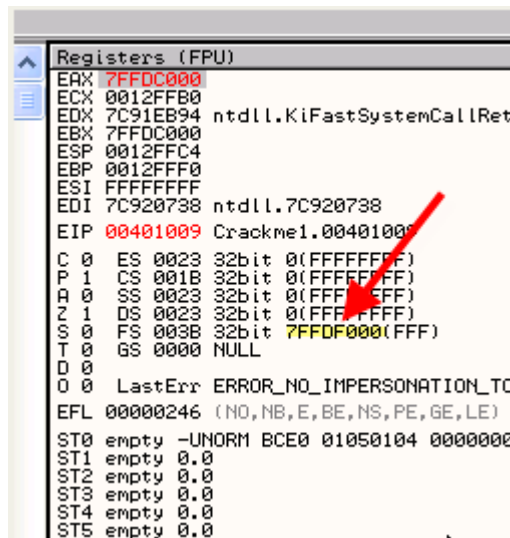
当前程序开始运行的时候,在内存的某处存放在一个特殊的标志,通过该标志来检测当前程序是否正在被调试,如果该标志为 1-当前程序正在被调试,如果该标志为 0-当前程序没有被调试。这个特殊的字节的值可以通过 IsDebuggerPresent 或者上面 3 条指令读取出来。

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:

00401000	64:01 180000	MOV EAX,DWORD PTR FS:[18]	
00401006	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	
00401009	0FB640 02	MOUZX EAX,BYTE PTR DS:[EAX+2]	
0040100D	90	NOP	
0040100E	90	NOP	
0040100F	90	NOP	
00401010	90	NOP	
00401011	E8 72040000	CALL <JMP.&comet132.InitCommonControls>	C InitCommonCor
00401016	6A 0A	PUSH 0A	Arg4 = 000000
00401018	FF35 5431400	PUSH DWORD PTR DS:[403154]	Arg3 = 000000
0040101E	6A 00	PUSH 0	Arg2 = 000000
00401020	CC25 5021400	PUSH DWORD PTR DS:[402150]	Arg1 = 000000

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)。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 #...!!
7FFDF008	00 00 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...c....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	...L2Δ#Δ..
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.SΔ+0....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00

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。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 #...!!
7FFDF008	00 00 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...c....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	...L2Δ#Δ..
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.SΔ+0....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00

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。

0012C000	00001000				Priv	RW	Gua	RW	
0012D000	00003000			stack of ma	Priv	RW	Gua	RW	
00130000	00003000				Map	R		R	
00140000	00004000				Priv	RW		RW	
00240000	00006000				Priv	RW		RW	

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)。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!.
7FFDF008	00 D0 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	..A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ.....
7FFDF020	E8 09 00 00 A8 0A 00 00	p...è.....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2Δ#Δ.....
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.SΔ+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00

在堆栈中定位到该地址。

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDC000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84543DA8	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	Crackme1.<ModuleEntryPoint>
0012FFFC	00000000	

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。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!.
7FFDF008	00 D0 12 00 00 00 00 00	.S#.....
7FFDF010	00 1E 00 00 00 00 00 00	..A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ.....
7FFDF020	E8 09 00 00 A8 0A 00 00	p...è.....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2Δ#Δ.....
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	00 C7 2B E3 00 00 00 00	.SΔ+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00
7FFDF068	00 00 00 00 00 00 00 00
7FFDF070	00 00 00 00 00 00 00 00
7FFDF078	00 00 00 00 00 00 00 00
7FFDF080	00 00 00 00 00 00 00 00
7FFDF088	00 00 00 00 00 00 00 00

0012FFC4	7C816D4F
0012FFC8	7C920738
0012FFCC	FFFFFFFF
0012FFD0	7FFDC000
0012FFD4	8054A938
0012FFD8	0012FFC8
0012FFDC	84543DA8
0012FFE0	FFFFFFFF
0012FFE4	7C8399F3
0012FFE8	7C816D58
0012FFEC	00000000
0012FFF0	00000000
0012FFF4	00000000
0012FFF8	00401000
0012FFFC	00000000

Command	? fs:[0]	HEX: 12FFE0 - DEC: 1245152 - ASCII: 0ÿà
---------	----------	---

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:

Address	Hex dump	ASCII	
7FFDF000	E0 FF 12 00 00 00 13 00	0 0 0 0 0 0 0 0	
7FFDF008	00 00 12 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF010	00 1E 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF018	00 F0 FD 7F 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF020	E8 09 00 00 A8 00 00 00	0 0 0 0 0 0 0 0	
7FFDF028	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF030	00 C0 FD 7F 10 05 00 00	. 0 0 0 0 0 0 0 0	
7FFDF038	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF040	00 C7 2B E3 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF048	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF050	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF058	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF060	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF068	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF070	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF078	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF080	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF088	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	

Command: ? fs:[1] HEX: 12FF - DEC: 4863 - ASCII: 0ÿ

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。

00401000	64:A1 180000	MOV EAX, DWORD PTR FS:[18]
00401006	8B40 30	MOV EAX, DWORD PTR DS:[EAX+30]
00401009	0FB640 02	MOVZX EAX, BYTE PTR DS:[EAX+2]
0040100D	90	NOP
0040100F	9A	NOP

我们可以看到 FS:[18]的值为 77FFDF018。

Address	Hex dump	ASCII	
7FFDF000	E0 FF 12 00 00 00 13 00	0 0 0 0 0 0 0 0	
7FFDF008	00 00 12 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF010	00 1E 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF018	00 F0 FD 7F 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF020	E8 09 00 00 A8 00 00 00	0 0 0 0 0 0 0 0	
7FFDF028	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF030	00 C0 FD 7F 10 05 00 00	. 0 0 0 0 0 0 0 0	
7FFDF038	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF040	00 C7 2B E3 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF048	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF050	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF058	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF060	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF068	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF070	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF078	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF080	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	
7FFDF088	00 00 00 00 00 00 00 00	. 0 0 0 0 0 0 0 0	

Command: ? fs:[18] HEX: 7FFDF000 - DEC: 2147348480 - ASCII: 0ÿ0

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。

Registers (FPU)	
EAX	7FFDF000
ECX	0012FFB0
EDX	7C91EB94 ntdll.h
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.h
EIP	00401006 Crackme
C 0	ES 0023 32bit 0
P 1	CS 001B 32bit 0

好了,现在 EAX 就保存了 TIB 的指令。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!.
7FFDF008	00 D0 12 00 00 00 00 00	.S+.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...¿....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2Δ#Δ..
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	D0 C7 2B E3 00 00 00 00	\$Δ+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00

现在我们看下一条指令。

00401000	64:A1 180000	MOV EAX,DWORD PTR FS:[18]
00401006	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]
00401009	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
0040100D	90	NOP
0040100E	90	NOP

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

对应为 FS:[30]。

Address	Hex dump	ASCII
7FFDF000	E0 FF 12 00 00 00 13 00	0 0...!!.
7FFDF008	00 D0 12 00 00 00 00 00	.S+.....
7FFDF010	00 1E 00 00 00 00 00 00	.A.....
7FFDF018	00 F0 FD 7F 00 00 00 00	..-2Δ....
7FFDF020	E8 09 00 00 A8 0A 00 00	b...¿....
7FFDF028	00 00 00 00 00 00 00 00
7FFDF030	00 C0 FD 7F 1D 05 00 00	..L2Δ#Δ..
7FFDF038	00 00 00 00 00 00 00 00
7FFDF040	D0 C7 2B E3 00 00 00 00	\$Δ+0.....
7FFDF048	00 00 00 00 00 00 00 00
7FFDF050	00 00 00 00 00 00 00 00
7FFDF058	00 00 00 00 00 00 00 00
7FFDF060	00 00 00 00 00 00 00 00
7FFDF068	00 00 00 00 00 00 00 00
7FFDF070	00 00 00 00 00 00 00 00
7FFDF078	00 00 00 00 00 00 00 00
7FFDF080	00 00 00 00 00 00 00 00
7FFDF088	00 00 00 00 00 00 00 00

Command	? fs:[30]	HEX: 7FFDC000 - DEC: 2147336
---------	-----------	------------------------------

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。

Registers (FPU)	
EAX	7FFDC000
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFDC000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00401009 Crac
C 0	ES 0023 32bi
P 1	CS 001B 32bi
A 0	SS 0023 32bi
7 1	DS 0023 32bi

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。

Address	Hex dump	ASCII
7FFDC000	00 00 01 00 FF FF FF FF	..0.
7FFDC008	00 00 40 00 A0 1E 24 00	..0.â€š.
7FFDC010	00 00 02 00 00 00 00 00	..0....
7FFDC018	00 00 14 00 C0 E4 98 7C	..Œ.ÿ!
7FFDC020	05 10 91 7C ED 10 91 7C	ÿ!ÿ!ÿ!
7FFDC028	01 00 00 00 80 29 01 77	0...ÿ!0w
7FFDC030	00 00 00 00 00 00 00 00
7FFDC038	00 00 00 00 00 00 00 00
7FFDC040	80 E4 98 7C 03 00 00 00	ÿ!ÿ!...
7FFDC048	00 00 00 00 00 00 00 00

最后一条指令:

00401009	0FB640 02	MOVZX EAX, BYTE PTR DS:[EAX+2]
0040100D	90	NOP
0040100F	90	NOP

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。

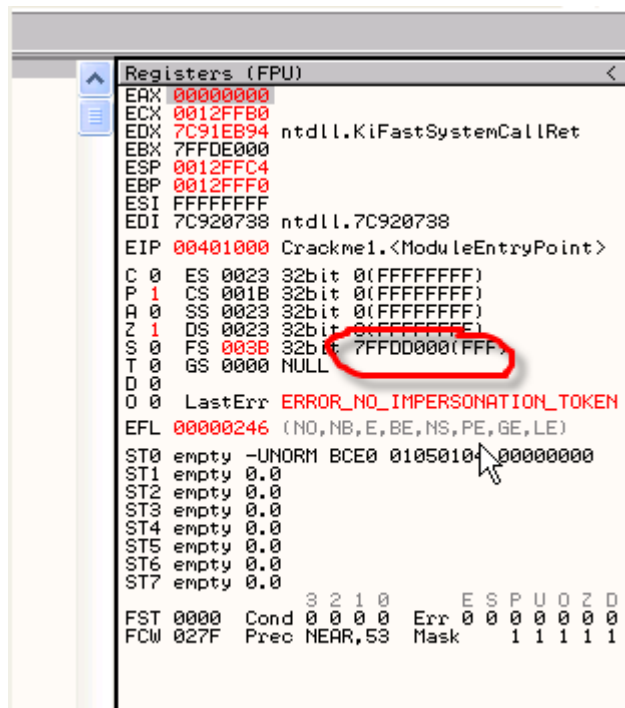
Address	Hex dump	ASCII
7FFDC000	00 00 01 00 FF FF FF FF	..0.
7FFDC008	00 00 40 00 A0 1E 24 00	..0.â€š.
7FFDC010	00 00 02 00 00 00 00 00	..0....
7FFDC018	00 00 14 00 C0 E4 98 7C	..Œ.ÿ!
7FFDC020	05 10 91 7C ED 10 91 7C	ÿ!ÿ!ÿ!
7FFDC028	01 00 00 00 80 29 01 77	0...ÿ!0w
7FFDC030	00 00 00 00 00 00 00 00
7FFDC038	00 00 00 00 00 00 00 00
7FFDC040	80 E4 98 7C 03 00 00 00	ÿ!ÿ!...
7FFDC048	00 00 00 00 00 00 00 00

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,但是您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。

00401000	6A 00	PUSH 0	
00401002	E8 6F040000	CALL <JMP.&kernel32.GetModuleHandleA>	
00401007	A3 50314000	MOV DWORD PTR DS:[403150],EAX	
0040100C	E8 5F040000	CALL <JMP.&kernel32.GetCommandLineA>	
00401011	E8 72040000	CALL <JMP.&comctl32.InitCommonControls>	InitCommonCo
00401016	6A 0A	PUSH 0A	Arg4 = 00000
00401018	FF35 54314000	PUSH DWORD PTR DS:[403154]	Arg3 = 00000
0040101E	6A 00	PUSH 0	Arg2 = 00000
00401020	FF35 50314000	PUSH DWORD PTR DS:[403150]	Arg1 = 00000
00401026	E8 06000000	CALL Crackme1.00401031	Crackme1.004
0040102B	50	PUSH EAX	ExitCode
0040102C	E8 39040000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess

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。

Address	Hex dump	ASCII
7FFDD000	E0 FF 12 00 00 00 13 00	0 +...!!.
7FFDD008	00 D0 12 00 00 00 00 00	.s+....
7FFDD010	00 1E 00 00 00 00 00 00	.A.....
7FFDD018	00 D0 FD 7F 00 00 00 00	.S^Δ....
7FFDD020	8C 0F 00 00 34 0B 00 00	i*.4δ...
7FFDD028	00 00 00 00 00 00 00 00
7FFDD030	00 E0 FD 7F 1D 05 00 00	.0^Δ#Δ..
7FFDD038	00 00 00 00 00 00 00 00
7FFDD040	D0 C7 2B E3 00 00 00 00	\$Δ+0....
7FFDD048	00 00 00 00 00 00 00 00
7FFDD050	00 00 00 00 00 00 00 00
7FFDD058	00 00 00 00 00 00 00 00

我们找到 FS:[30]即 TIB 的指针偏移 30,我们找到的内容如下:

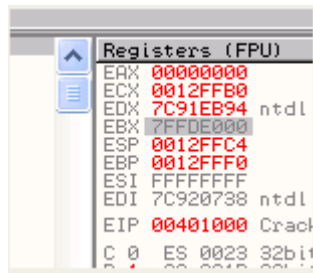
Address	Hex dump	ASCII
7FFDD000	E0 FF 12 00 00 00 13 00	0 +...!!.
7FFDD008	00 D0 12 00 00 00 00 00	.s+....
7FFDD010	00 1E 00 00 00 00 00 00	.A.....
7FFDD018	00 D0 FD 7F 00 00 00 00	.S^Δ....
7FFDD020	8C 0F 00 00 34 0B 00 00	i*.4δ...
7FFDD028	00 00 00 00 00 00 00 00
7FFDD030	00 E0 FD 7F 1D 05 00 00	.0^Δ#Δ..
7FFDD038	00 00 00 00 00 00 00 00
7FFDD040	D0 C7 2B E3 00 00 00 00	\$Δ+0....
7FFDD048	00 00 00 00 00 00 00 00
7FFDD050	00 00 00 00 00 00 00 00
7FFDD058	00 00 00 00 00 00 00 00

该内容为 7FFDE000,我们继续在数据窗口中定位到该地址。

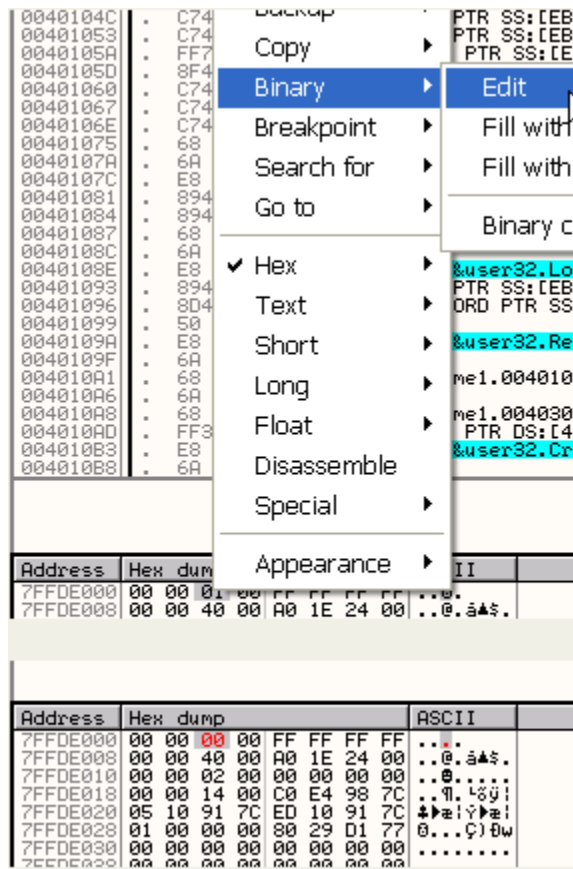
Address	Hex dump	ASCII
7FFDE000	00 00 01 00 FF FF FF FF	..0.
7FFDE008	00 00 40 00 A0 1E 24 00	..0.ΔΔ\$.
7FFDE010	00 00 02 00 00 00 00 00	..0.
7FFDE018	00 00 14 00 C0 E4 98 7C	..0. L\$y!
7FFDE020	05 10 91 7C ED 10 91 7C	ΔΔΔ!ΔΔΔ!
7FFDE028	01 00 00 00 80 29 D1 77	0...C)0w
7FFDE030	00 00 00 00 00 00 00 00
7FFDE038	00 00 00 00 00 00 00 00
7FFDE040	80 E4 98 7C 03 00 00 00	C\$y!Δ....
7FFDE048	00 00 00 00 00 00 6F 7FoΔ
7FFDE050	00 00 6F 7F 88 06 6F 7F	..oΔΔΔoΔ
7FFDE058	00 00 FB 7F 00 10 FC 7F	..Δ.ΔΔΔ

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

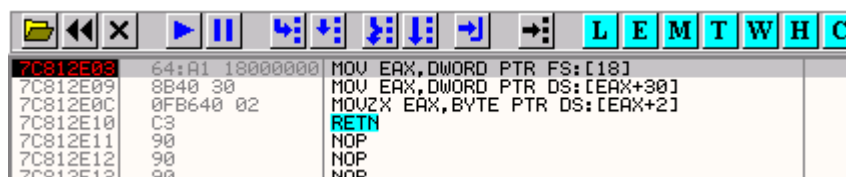
该区域的话,那么 $EBX = FS:[30]$ 。



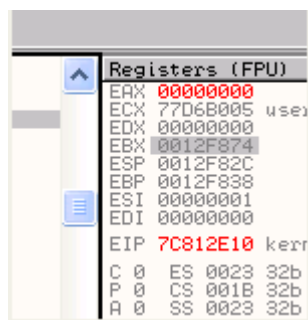
好了,该方法在程序刚刚启动的时候就可以定位到该特殊字节。现在我们将该字节修改为0。



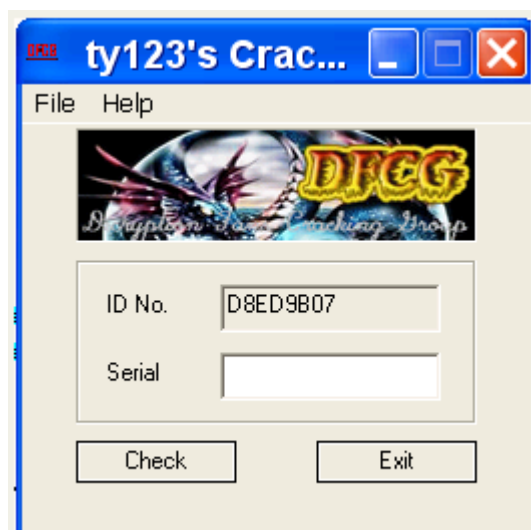
现在我们给 `IsDebuggerPresent` 设置一个断点。运行起来看看接下来会发生什么。



执行到返回。



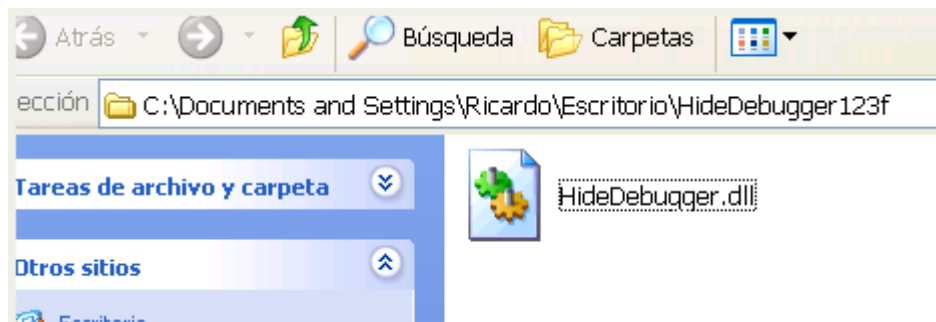
可以看到 EAX 的值为 0,因为我们修改了那个特殊字节的值,所以当前程序认为它没有被调试。



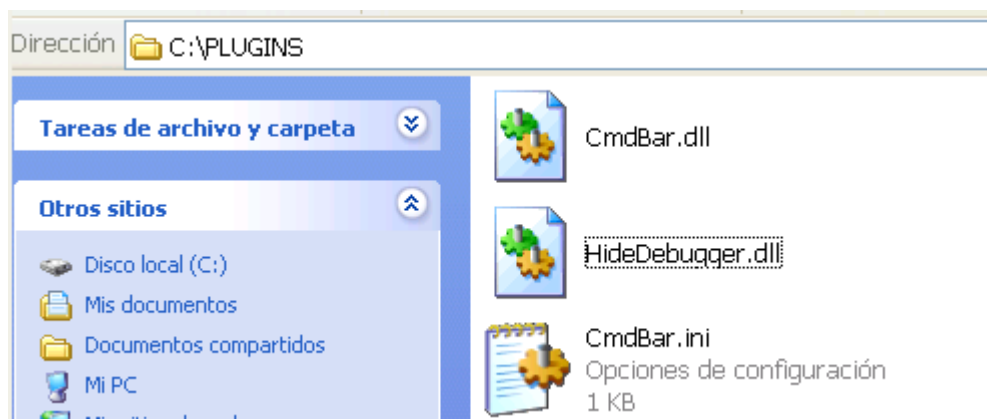
可以看到,程序正常运行起来了。

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。

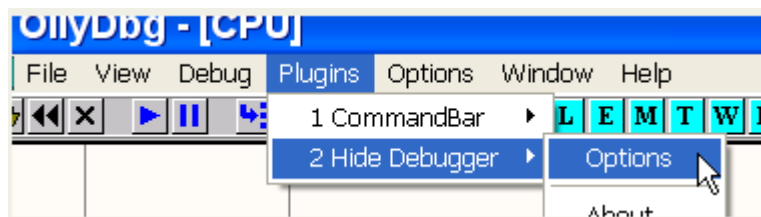
下载该插件,然后将 DLL 文件复制到 OD 的插件目录下。



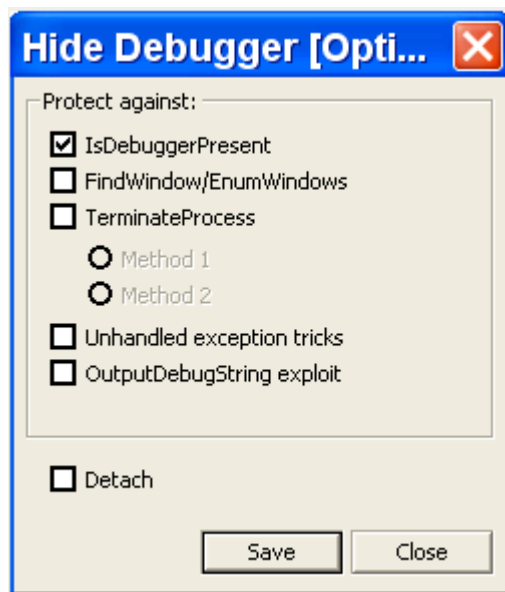
复制,嘿嘿。



重新启动 OD。

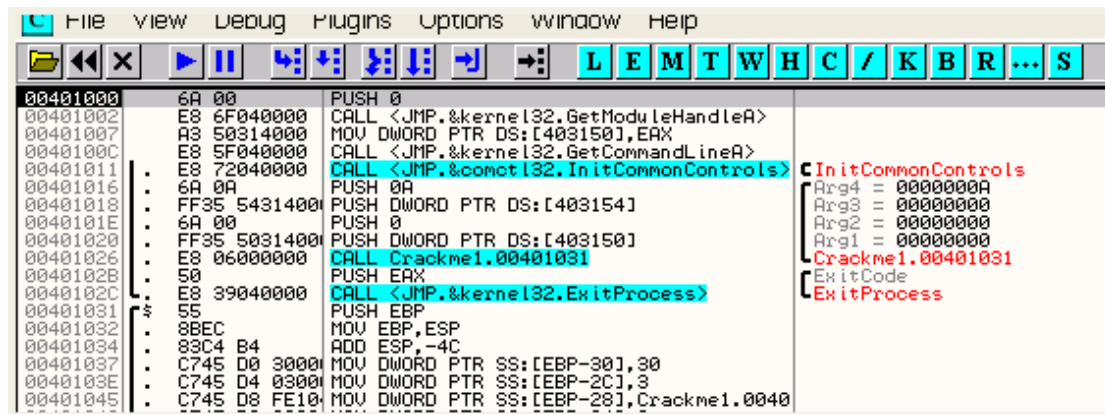


选择该插件并进行设置。

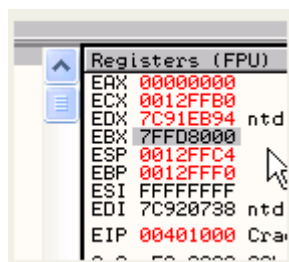


勾选中 IsDebuggerPresent,其他的选项我们后面详细介绍了再勾选。

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。

Address	Hex dump	ASCII
7FFD8000	00 00 00 00 FF FF FF FF
7FFD8008	00 00 40 00 A0 1E 24 00	..@.á\$.
7FFD8010	00 00 02 00 00 00 00 00	..@.....
7FFD8018	00 00 14 00 C0 E4 98 7C	..@.L\$!
7FFD8020	05 10 91 00 ED 10 91 7C	5!ÿ!
7FFD8028	01 00 00 00 80 29 01 77	@...Ç)0w
7FFD8030	00 00 00 00 00 00 00 00
7FFD8038	00 00 00 00 00 00 00 00

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。