第九章-基本概念

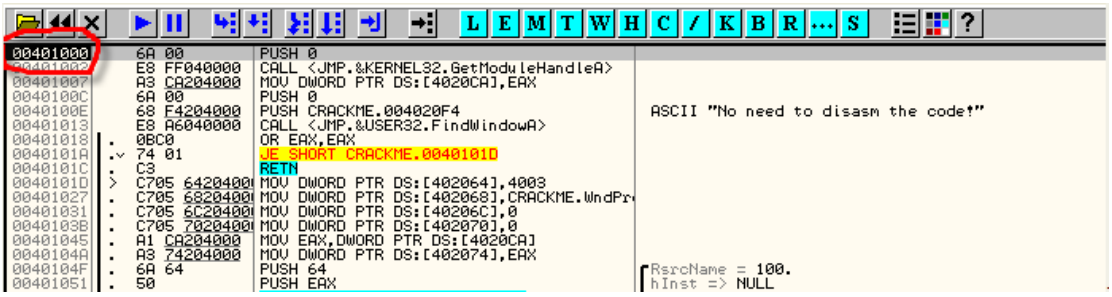
本章我们将开始破解。我们先从基本的概念开始一步步的来介绍破解所需要的步骤。

本次实验依然是我们熟悉的 Cruehead`a 的 CrackMe,但是不要把自己局限于破解这个简单的 CrackMe 的不同方式-在此过程中,我们将介绍适用于更加复杂的软件的一些标准方法。

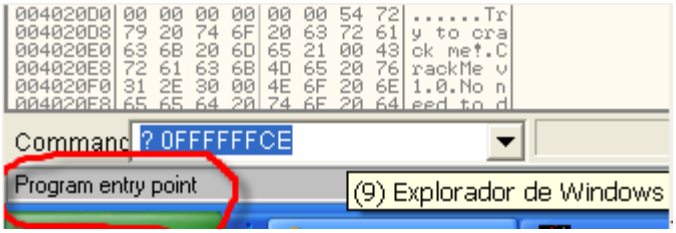
让我们用调试器开始破解之旅吧。

通过这个 CrackMe 我们可以掌握一些基本的概念。

入口点:程序刚刚被加载第一条指令的地址。为了不和 OEP(原始入口点)相混淆,我们稍后再来介绍 OEP 的概念。当用 OD 加载应用程序后,调试器就会停在入口点处,分析代码并且等待用户的进一步提示。

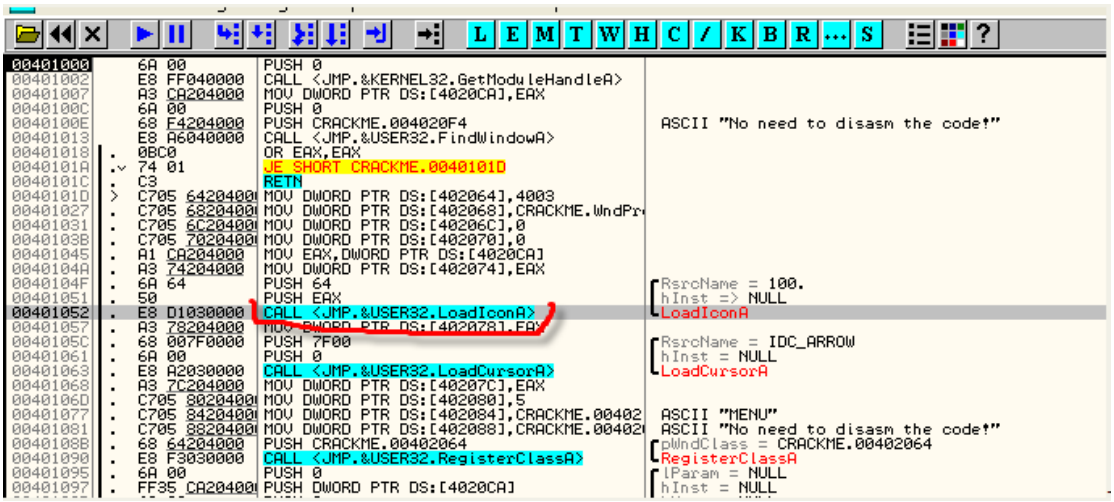


对于 Cruehead`a 的 CrackMe 这个程序来说,入口点为 401000。通常情况下,状态栏会显示调试器暂停的原因。现在就提示我们,当前是入口点:



大部分(99%)的程序启动的时候都会停在入口点处。还有一些程序通过一些修改方式让其在启动的时候不停在入口点处。这些方法我们将在后面讨论。这些方法就是我们常说的反调试。

接下来,让我们来了解一下 DLL(动态链接库)的概念以及 DLL 导出函数的功能。



注意一下突出显示的部分,举个例子,比如我们要调用 401020 或者要跳转到 421367,这里会是外部函数的名称替代了这个绝对地址。

CALL LoadIconA

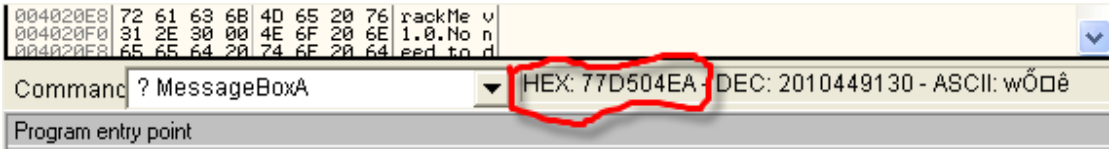
在最右边的列中显示了一些额外的信息,调用的是 LoadIconA。

Windows 操作系统支持的所谓的动态链接库(扩展名为 DLL 文件),它们与正常的可执行文件 EXE 具有相同的格式。动态链接库可

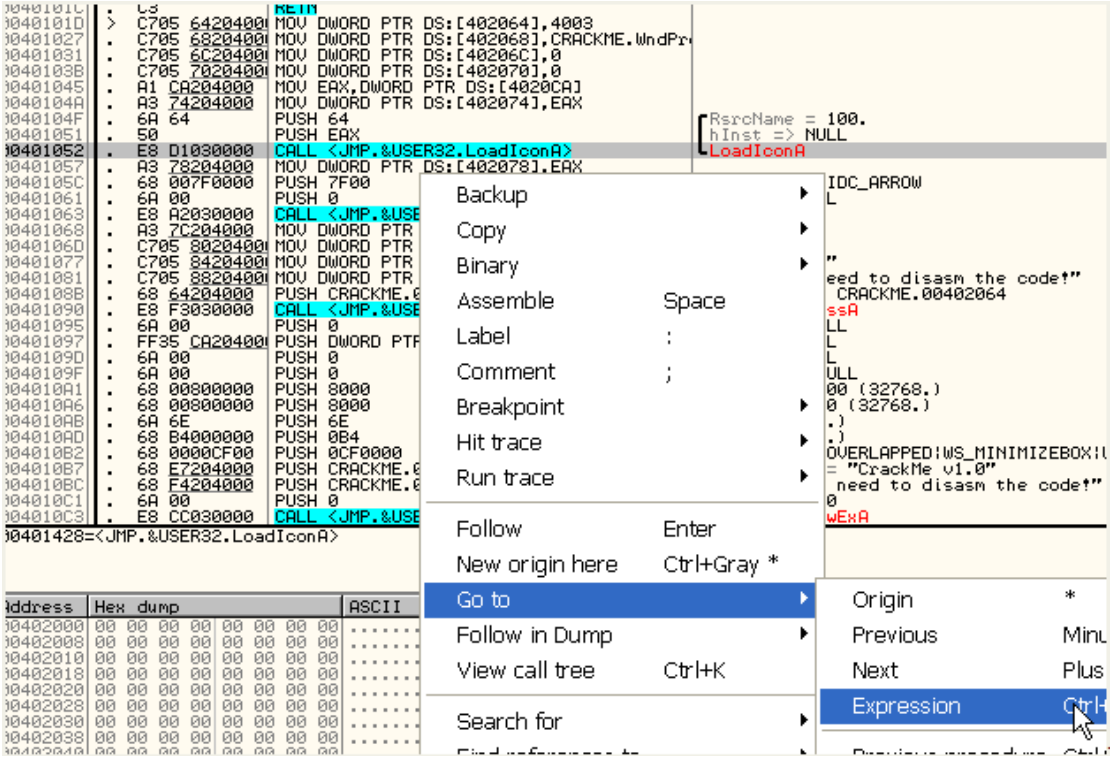
导出函数供其他可执行文件(EXE 和 DLL)调用。不是在多个可执行文件中有相同的静态副本,而是把功能放置在 DLL 中。如果一个功能的代码量很大,那么这样就可以缩减可执行文件的大小,更重要的是可以节省内存。Windows 的基本功能:文件,内存，进程,线程,图形,声音,网络等都是在标准的动态链接库中实现的。LoadIconA 是在 User32.dll 中实现的一个加载位图的应用程序接口。应用程序接口也称之为 API。

让我们来看看另一个 API MessageBoxA 的例子。

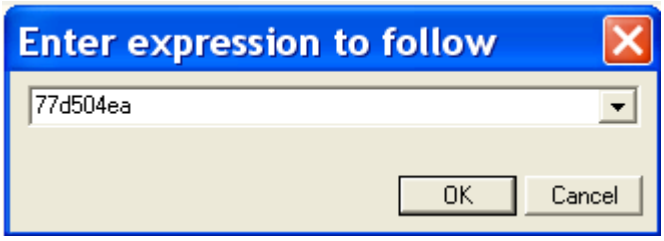
在命令栏中输入: ? MessageBoxA



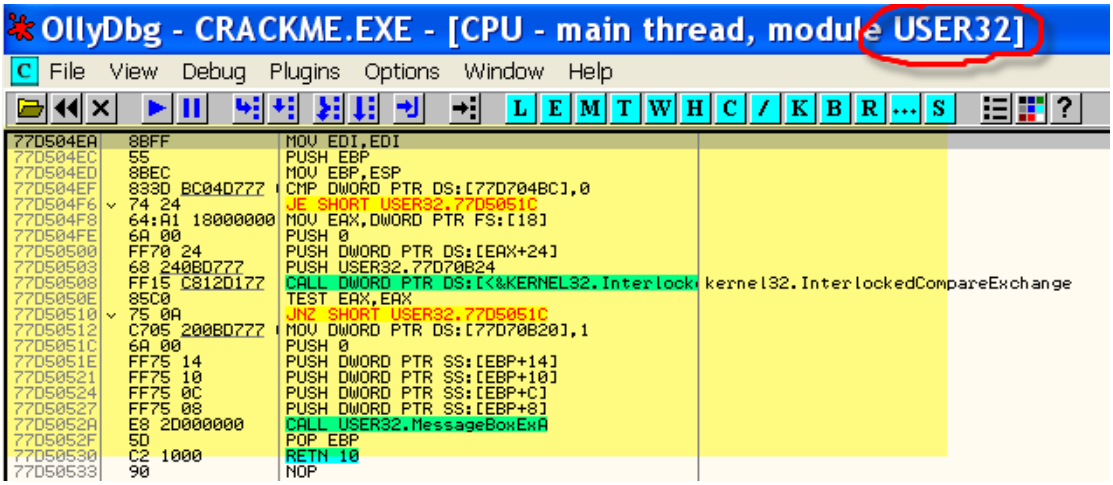
有个简单的提示,该函数的地址是 77D504EA。我们单击鼠标右键选择-Goto-Expression 输入这个地址。



我机器上面提示的地址可能和你机器上面的地址不一样。

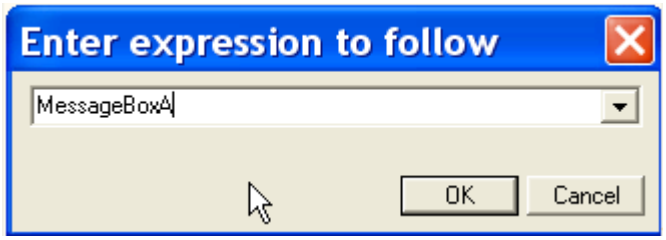


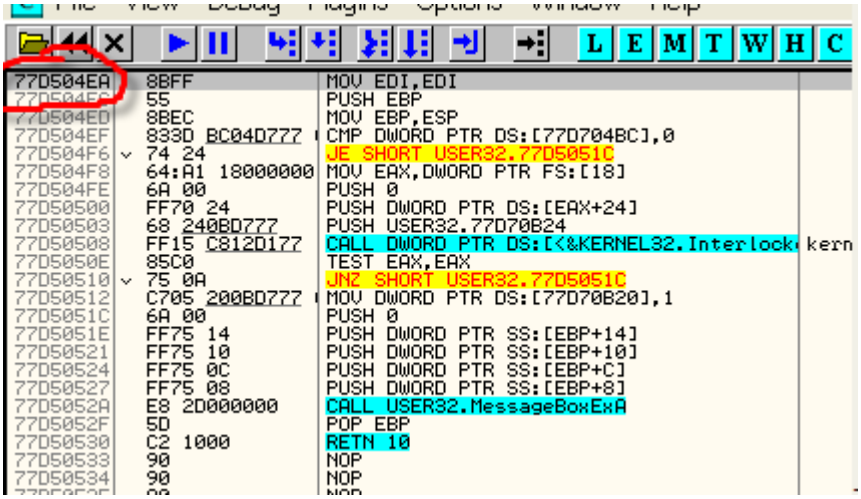
如果你是 windows 9x 的话,就不能这样做了,我们后面会讲到。



可以看到该函数属于 USER32.DLL,通常在调试器中显示的函数名称前面会有 DLL 的名称(例如 Call USER32.MessageBoxA)。从当前地址开始到函数返回实现了 MessageBoxA 的功能。MessageBoxA 便于我们的识别,但并不是机器码,给破解者提供了一定的帮助。

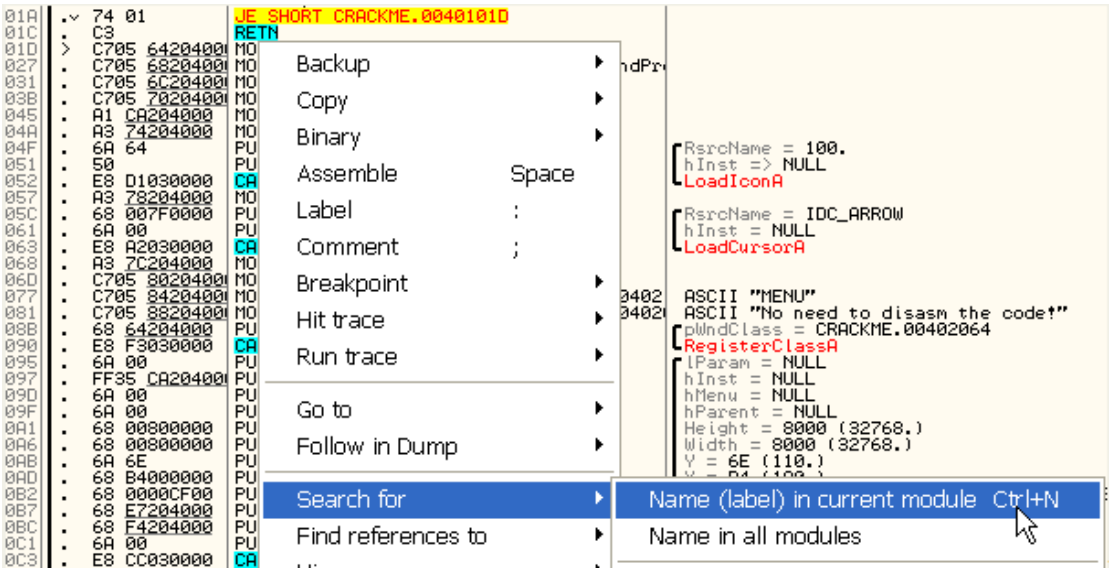
想要返回到前一条指令处的话,只需要按一下减号键。转到 MessageBoxA 的函数实现处,也可以直接单击鼠标右键选择 Goto-Expression 输入 MessageBoxA。





这里我们又进入了 USER32.DLL 的 MessageBoxA 函数中。该函数名称是区分大小写的(这里名称是 MessageBoxA 而不是 messageboxa)。

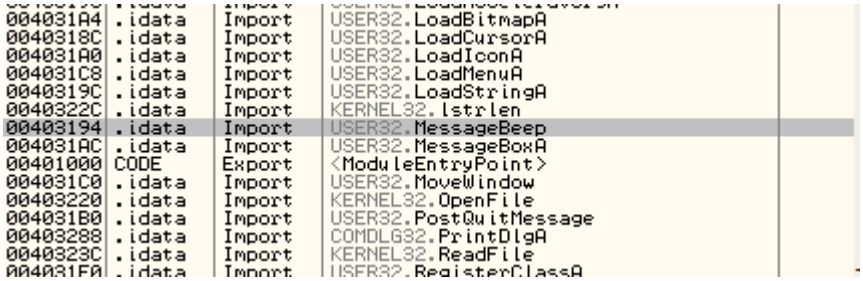
按减号-回到入口点。



单击鼠标右键:选择 Search for-Name(label)in current module Ctrl+N 菜单项。获取该 CrackMe 的 API 的名称列表。

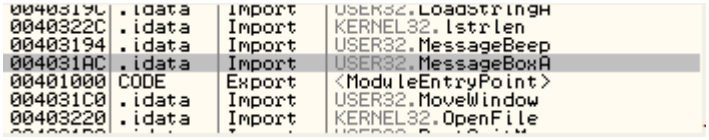
Address	Section	Type	Name	Comment
004031E4	.idata	Import	USER32.BeginPaint	
00403230	.idata	Import	KERNEL32.CloseHandle	
00403250	.idata	Import	COMCTL32.CreateToolbar	
0040324C	.idata	Import	COMCTL32.CreateToolbarEx	
004031E8	.idata	Import	USER32.CreateWindowExA	
004031EC	.idata	Import	USER32.DefWindowProcA	
00403278	.idata	Import	GDI32.DeleteDC	
00403274	.idata	Import	GDI32.DeleteObject	
004031F0	.idata	Import	USER32.DialogBoxParamA	
004031F4	.idata	Import	USER32.DispatchMessageA	
004031F8	.idata	Import	USER32.DrawMenuBar	
004031FC	.idata	Import	USER32.EndDialog	
00403270	.idata	Import	GDI32.EndDoc	
0040326C	.idata	Import	GDI32.EndPage	
00403200	.idata	Import	USER32.EndPaint	
00403240	.idata	Import	KERNEL32.ExitProcess	
00403204	.idata	Import	USER32.FindWindowA	
00403208	.idata	Import	USER32.GetDC	
0040320C	.idata	Import	USER32.GetDlgItem	
00403210	.idata	Import	USER32.GetDlgItemTextA	
0040321C	.idata	Import	KERNEL32.GetLocalTime	
00403214	.idata	Import	USER32.GetMessageA	
00403238	.idata	Import	KERNEL32.GetModuleHandleA	
00403284	.idata	Import	COMDLG32.GetOpenFileNameA	
00403280	.idata	Import	COMDLG32.GetSaveFileNameA	
00403268	.idata	Import	GDI32.GetStockObject	
00403188	.idata	Import	USER32.GetSystemMetrics	
00403264	.idata	Import	GDI32.GetTextMetricsA	
00403198	.idata	Import	USER32.GetWindowRect	
00403228	.idata	Import	KERNEL32.GlobalAlloc	
00403224	.idata	Import	KERNEL32.GlobalFree	
00403248	.idata	Import	COMCTL32.InitCommonControls	
004031B8	.idata	Import	USER32.InvalidateRect	
00403184	.idata	Import	USER32.KillTimer	
00403190	.idata	Import	USER32.LoadAcceleratorsA	
004031A4	.idata	Import	USER32.LoadBitmapA	
0040318C	.idata	Import	USER32.LoadCursorA	
004031A0	.idata	Import	USER32.LoadIconA	
004031C8	.idata	Import	USER32.LoadMenuA	
0040319C	.idata	Import	USER32.LoadStringA	
0040322C	.idata	Import	KERNEL32.lstrlen	
00403194	.idata	Import	USER32.MessageBeep	
004031AC	.idata	Import	USER32.MessageBoxA	
00401000	CODE	Export	<ModuleEntryPoint>	
004031C0	.idata	Import	USER32.MoveWindow	
00403220	.idata	Import	KERNEL32.OpenFile	
004031B0	.idata	Import	USER32.PostQuitMessage	
00403288	.idata	Import	COMDLG32.PrintDlgA	
0040323C	.idata	Import	KERNEL32.ReadFile	
004031E0	.idata	Import	USER32.RegisterClassA	
004031D0	.idata	Import	USER32.SendMessageA	
004031A8	.idata	Import	USER32.SetFocus	
004031D4	.idata	Import	USER32.SetTimer	
004031D8	.idata	Import	USER32.SetWindowPos	
004031CC	.idata	Import	USER32.ShowWindow	
00403260	.idata	Import	GDI32.StartDocA	
0040325C	.idata	Import	GDI32.StartPage	
00403258	.idata	Import	GDI32.TextOutA	
004031BC	.idata	Import	USER32.TranslateAcceleratorA	
004031C4	.idata	Import	USER32.TranslateMessage	
004031DC	.idata	Import	USER32.UpdateWindow	
004031B4	.idata	Import	USER32.WinHelpA	
00401128	CODE	Export	WndProc	
00403234	.idata	Import	KERNEL32.WriteFile	

查找我们关注的 API 函数的话,不需要一个个去看,只需要输入 API 函数的名称即可。这里我们按 M 键:

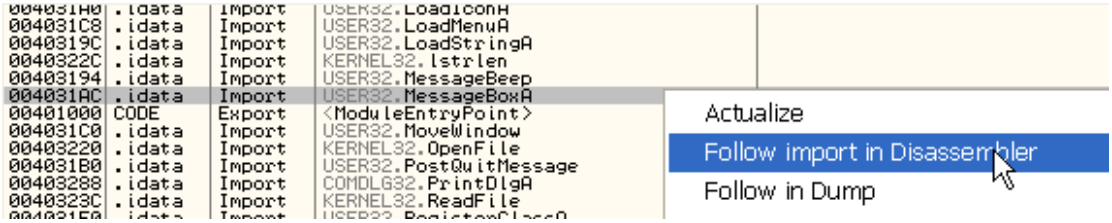


光标定位到第一个函数名称以 M 开始的 API 上。

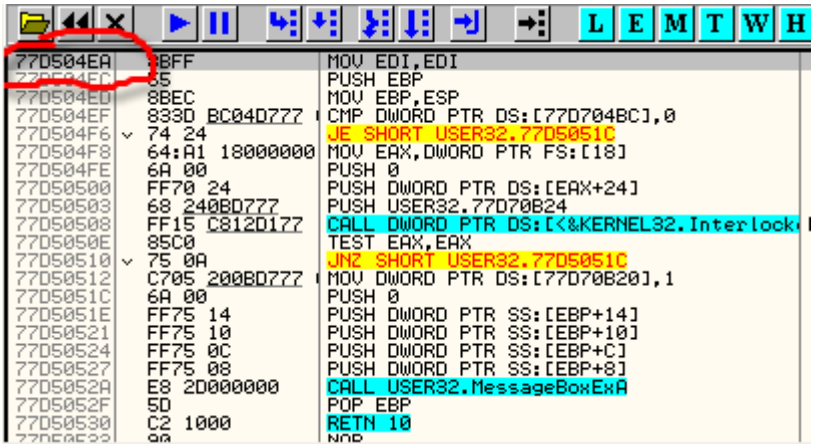
标题栏显示你正在搜索的字母。



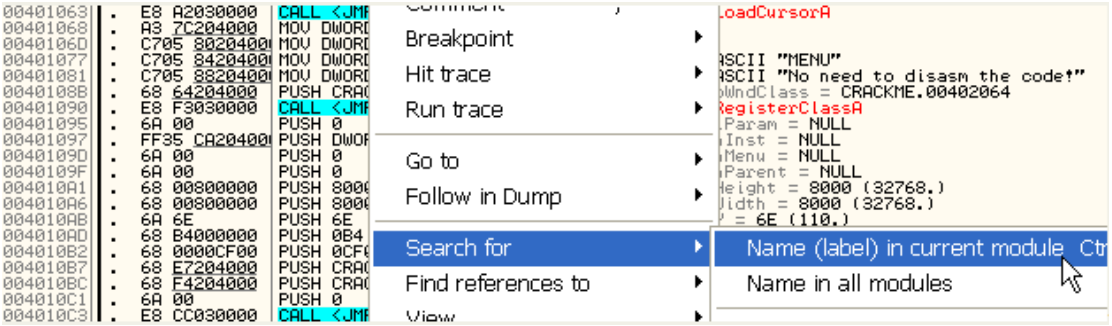
在函数名称上单击鼠标右键弹出的菜单项有:



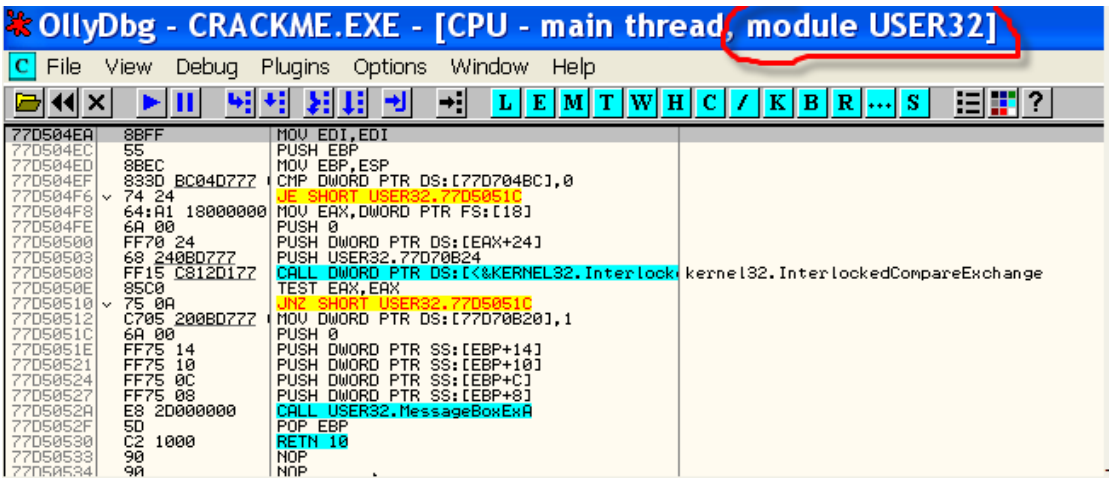
选择 Follow import in Disassembler 选项,我们就可以转到 API 函数的实现处。这也是另一种转到 API 函数实现代码的方法。



在这里新手普遍会犯一个错误,现在单击鼠标右键选择-Search for-Name(label)in current module。这个时候我们得到的函数名称是 USER32.DLL 库中的函数名称,并不是我们的 CrackMe 主程序的导入表中的函数名。菜单项中明确指出是搜索当前模块,我们现在这种情况,当前模块是 USER32,因此 MessageBoxA 是位于该模块当中的。



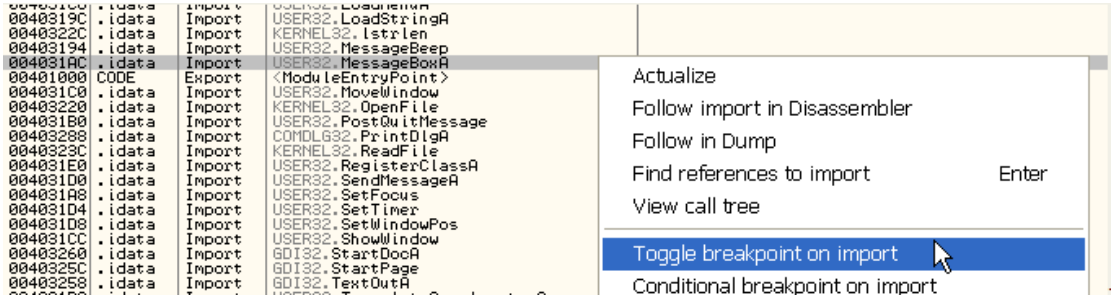
为了避免这样的错误,要记得看一眼调试器窗口的标题,尤其是当前模块的名称。



尽管我们现在并没有执行这个 API 函数,但是我们通过这种方式来看看我们感兴趣的 API 函数。然后按减号键就又可以回到你之前的位置。

Windows NT: 2000, XP 或者 2003

这里我们的教程讨论的是 windows NT/2000 和 XP,2003。对于 NT 系列的操作系统来说,OD 是非常好用的。如果你是 95/98 系统的话,你也可以参考附录部分(windows 9x 的说明).

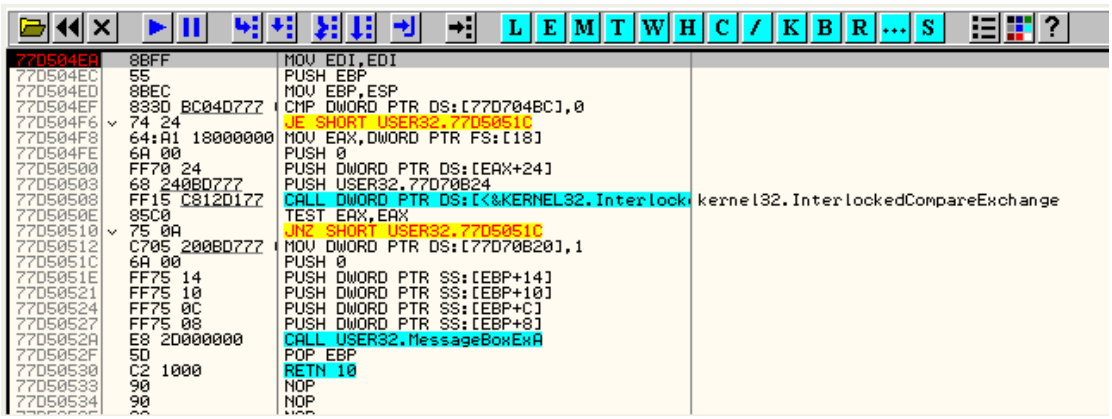


我们在 CrackMe 的 API 列表中单击鼠标右键选择-Toggle breakpoint on import 菜单项来给当前选中的函数设置断点。

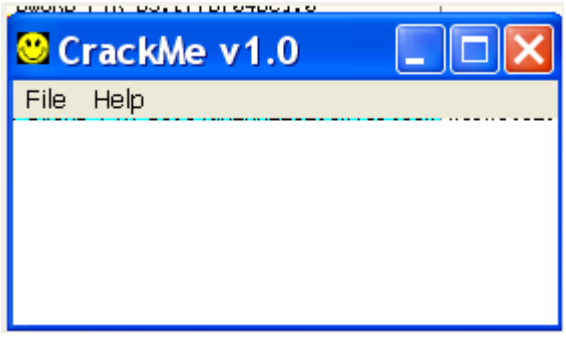
当然你也可以在命令栏中给你关注的函数设置断点。

bp MessageBoxA

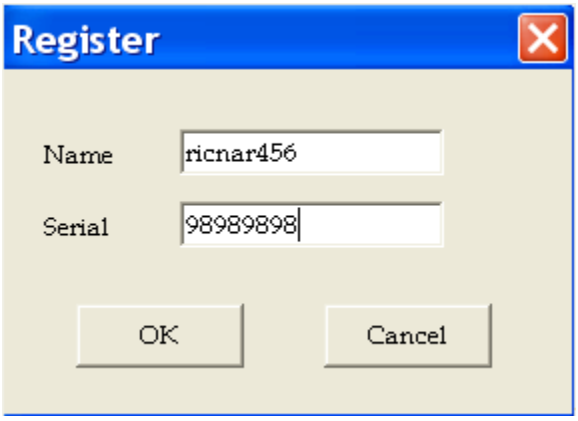
BP 是在指定的函数的第一条指令处设置断点。而不是在函数的调用处。我们可以来到 MessageBoxA 的第一条指令处看看。



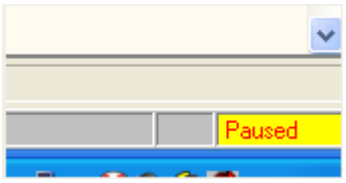
现在我们按下 F9 运行 CrackMe。



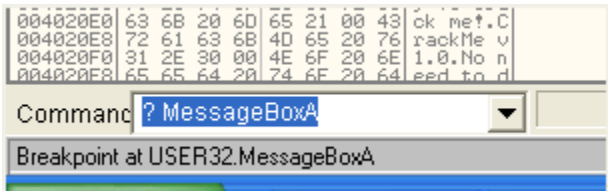
出现了 CrackMe 的主窗口。在菜单中选择 help-Register:



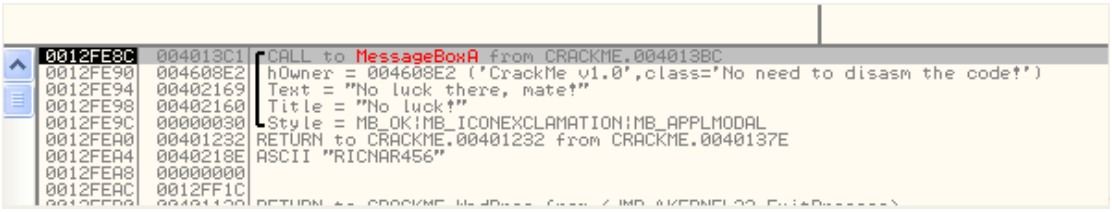
随便输入一个名词和序列号，然后单击确定。这个时候 MessageBoxA 函数就会被调用并且中断下来。右下角的标题显示暂停。



左侧标题会显示调试器暂停的原因。

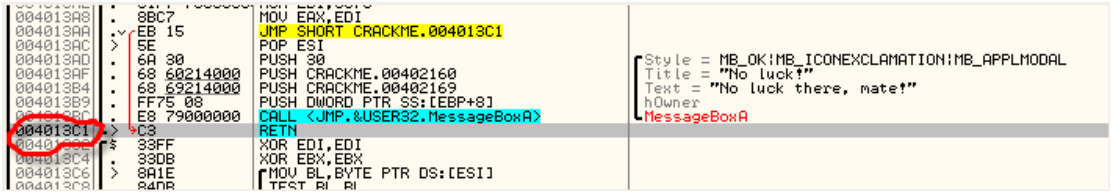


提示说:"中断在 USER32.MessageBoxA 处",这意味着 USER32.MessageBoxA 处设置了断点。此时我们断在了这里。



在该函数调用的时候,你可以看到这个函数的参数值。根据 stdcall 的调用约定,API 的参数通常从右至左的在堆栈中排列。

堆栈的顶部存放返回地址。当前是-4013C1。

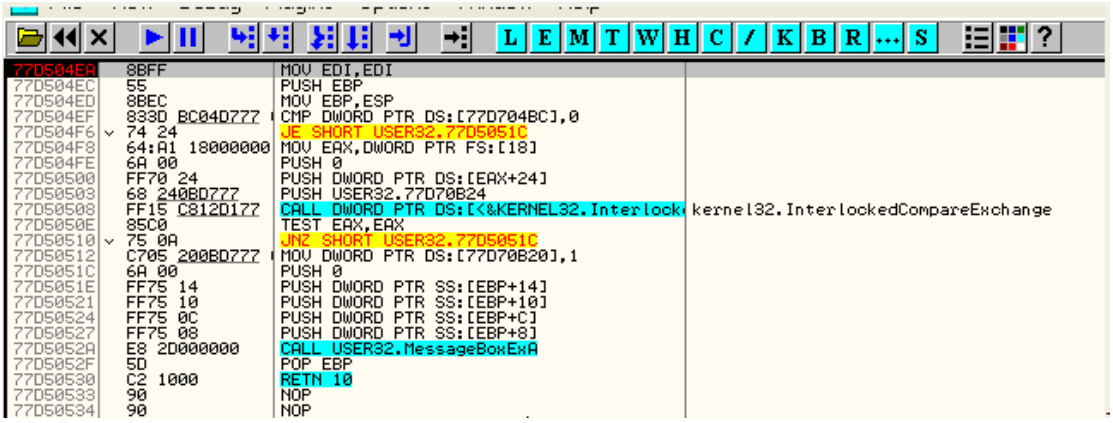


回顾一下我们第 7 章介绍过的 CALL 和 RET,我们就知道调用任何子程序,堆栈顶部存放的都是返回地址。当前调用 MessageBoxA，返回地址就是 4013C1。

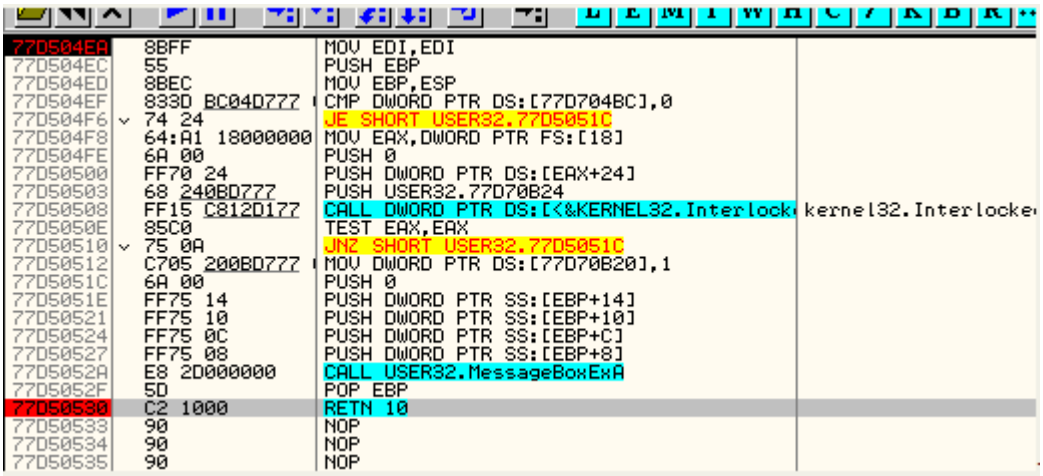
接下来(即下图)是函数的参数。MessageBoxA 有 4 个参数(可以参考 MSDN):父窗口句柄,消息文本,标题文本和风格。

消息文本是:“No luck there,mate!”。由于我们输入的用户名和序列号不正确。

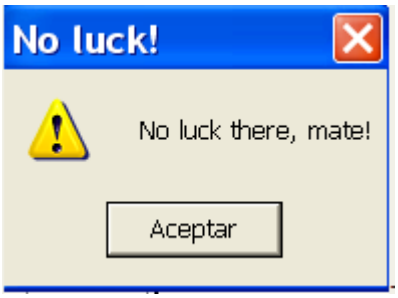
所以即将弹出注册失败的消息框。



为了证明会弹出消息框,我们在下方的 RETN 10 处设置断点。我们对应的地址可能不一样。但是无论如何,第一个 RET 处就是 MessageBoxA 函数的返回处。

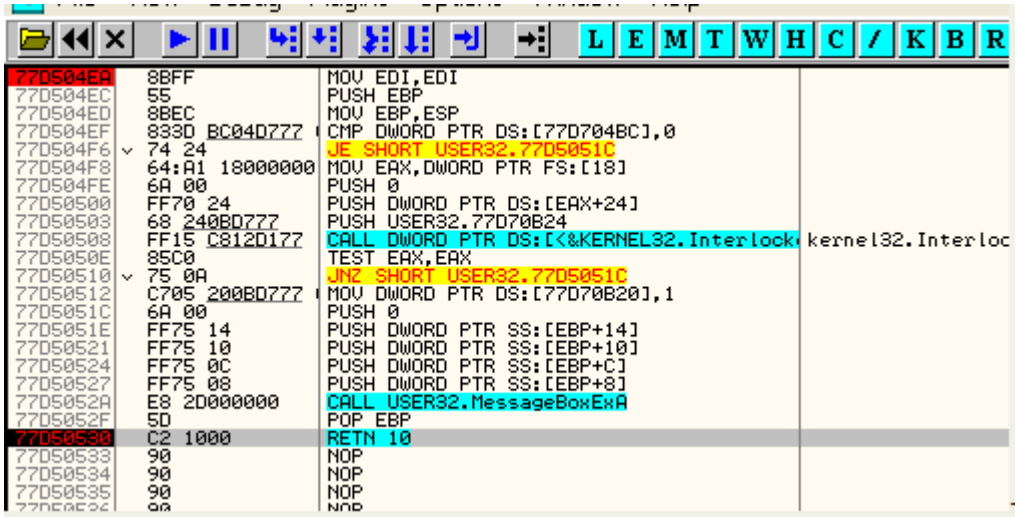


按 F9 键。



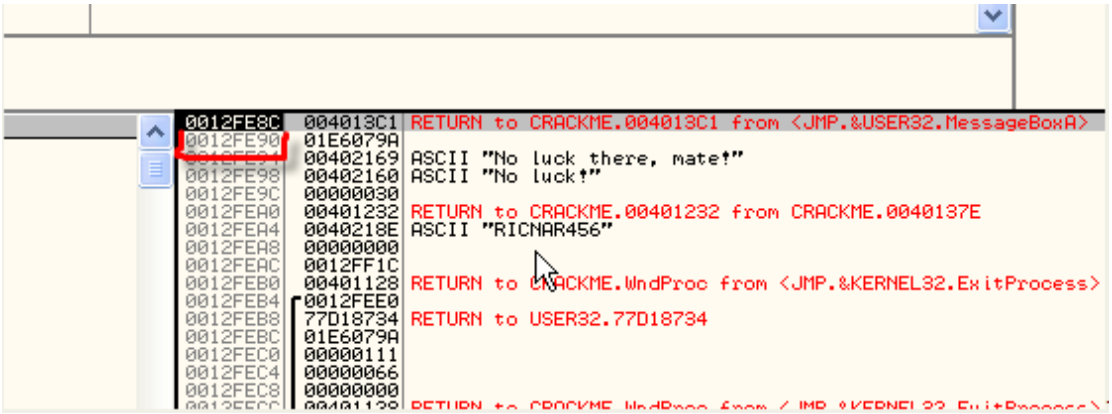
消息框弹了出来。标题文本为 No luck!，文本内容为 No luck there,mate!。表明输入的用户名或者序列号不正确。

单击”是”-触发 RETN 10 处的断点。

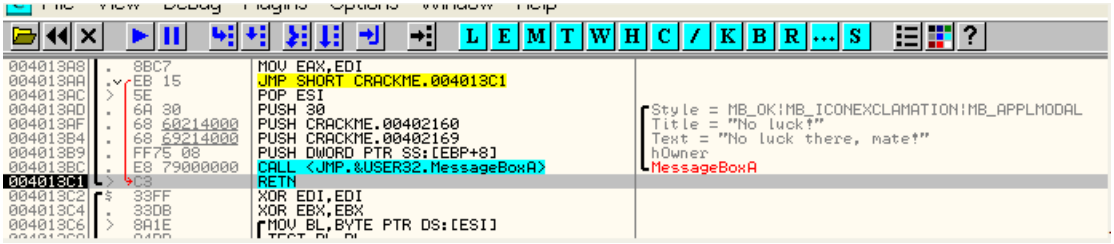


因此,我们得出结论,该消息框是在 MessageBoxA 函数执行过程中弹出的。

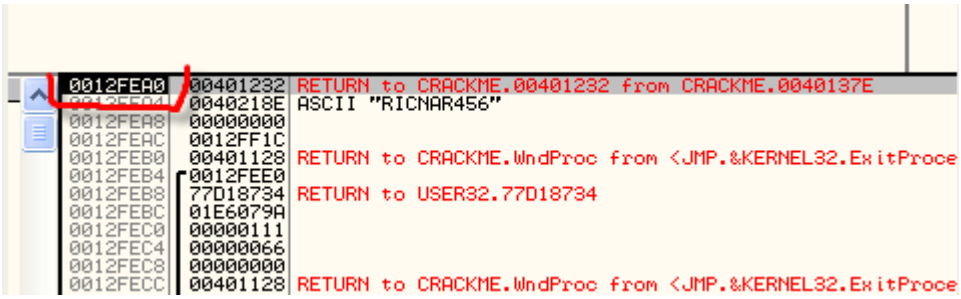
这里 RETN 10 跟通常的 RETN 有点差别。通常的 RET 只是返回到 4013C1 处。



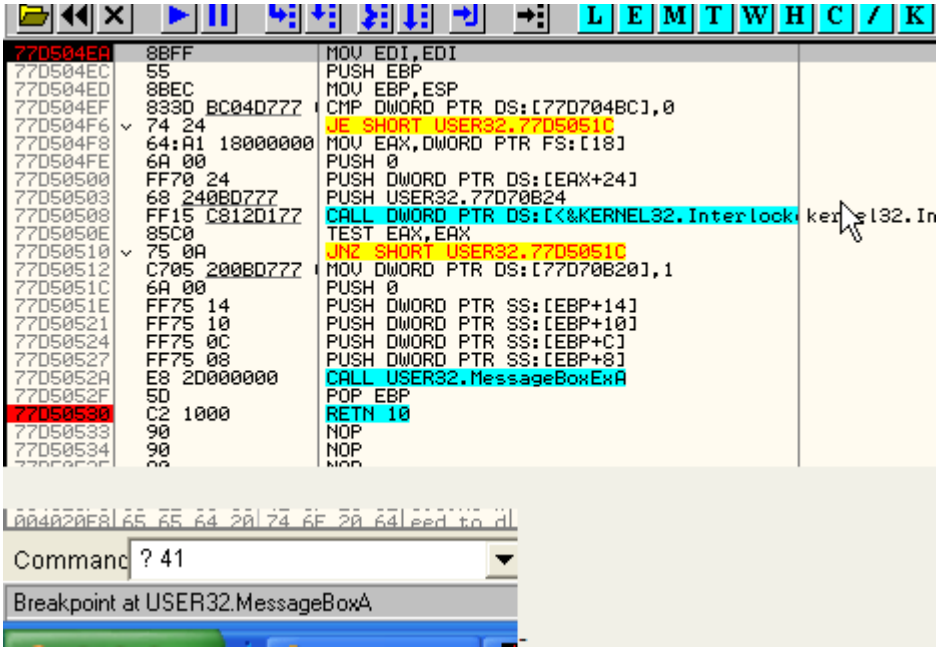
返回以后,返回地址从堆栈中删除。堆栈指针向下移动 4 字节(更确切的说是向高地址一侧)。对于 RET 10H 来说,除了完成 RET 的操作之外,ESP 还要增加 10h。所以,ESP 的增量为 10h + 4h = 14h = 20,让我们按 F7 键看看。



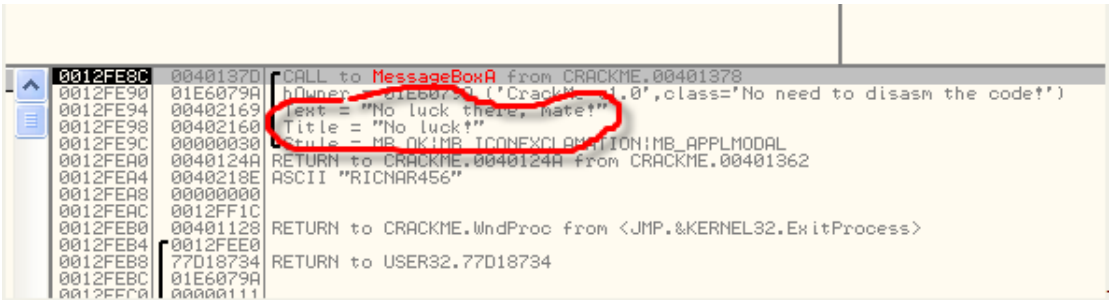
我们可以看到从 API 函数中返回到 CrackMe 主程序代码中。堆栈指针由顶部向下移动了 20 个字节。和之前预期的一样,即 RETN 不是简单的返回,并且还会清理按照 stdcall 调用约定传递给函数参数的堆栈空间。



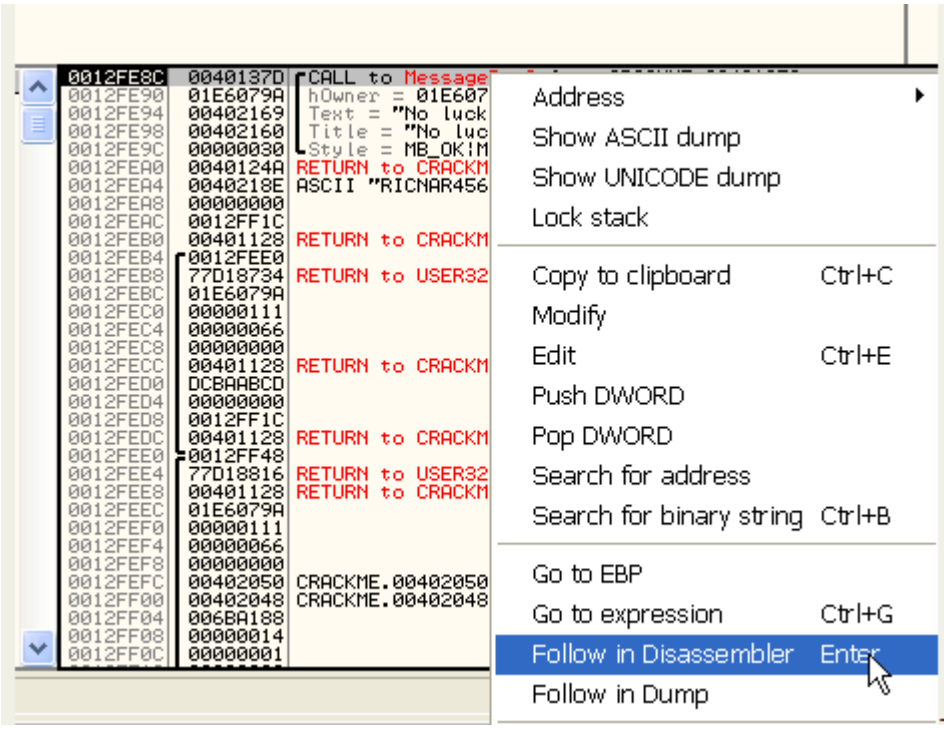
现在 CrackMe 已经知道了序列号不正确,我们按 F9 键。



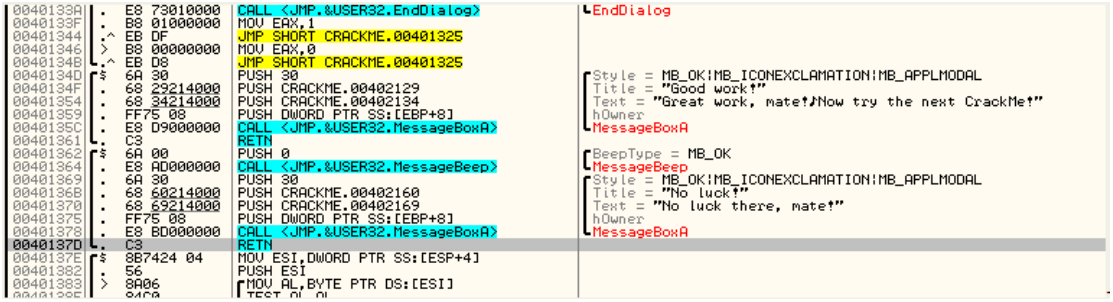
断点再次触发。CrackMe 会弹出错误提示框。



返回地址是 40137D。让我们来看看该地址处的代码。单击鼠标右键选择-Follow in Disassembler 或者 Goto-Expression 输入 40137D。



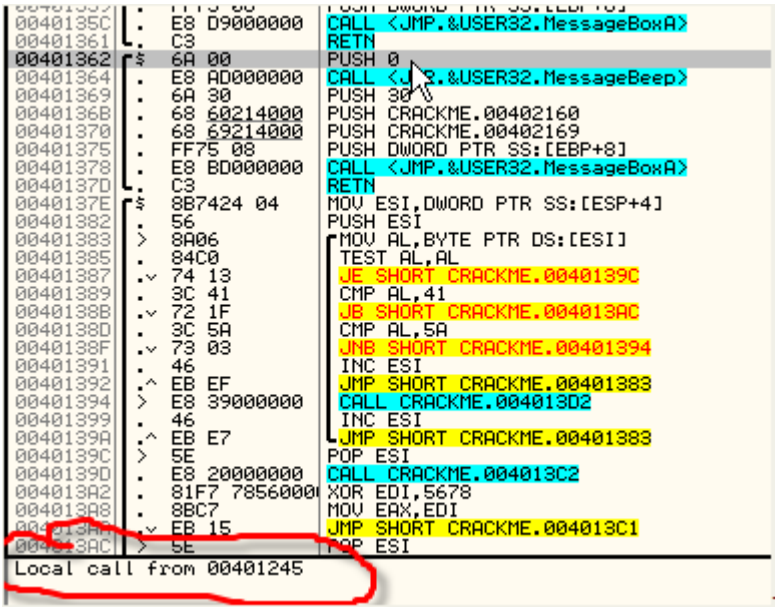
来到了 40137D 处,上面是一个 CALL 指令,调用 MessageBox(401378)。



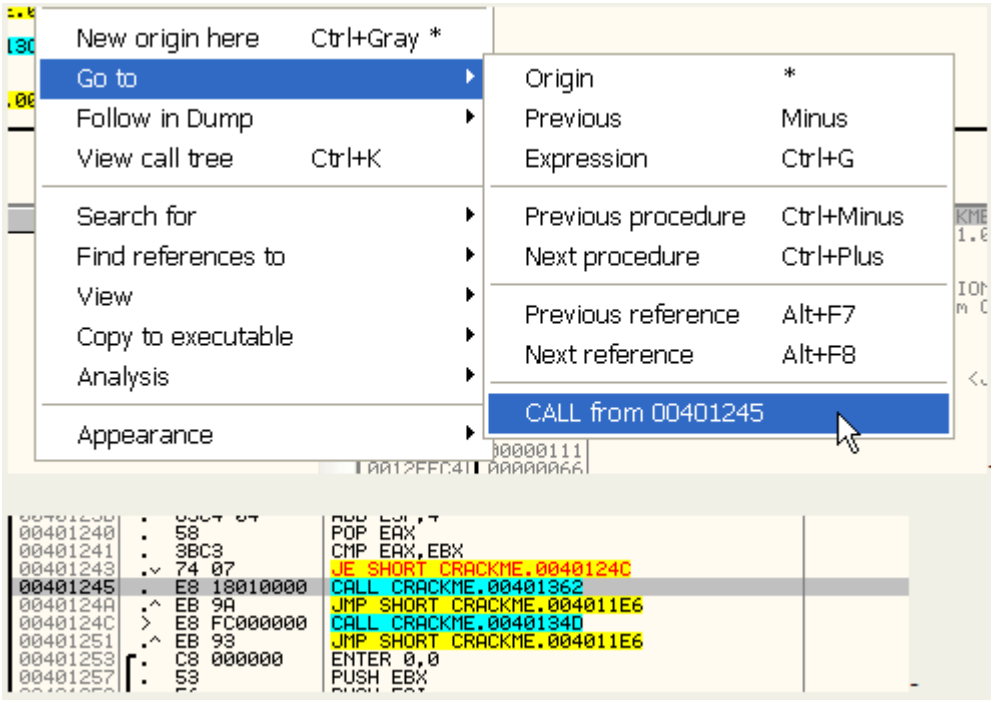
可以看到上面还有一个 MessageBox,但是提示的是”Great work,mate!Now try the next CrackMe!”

初步分析调试器显示的代码是函数的一部分,一个功能开始于 401362 弹出 No luck 的消息框。另一个功能开始于 40134D 弹出 Good work 的消息框。

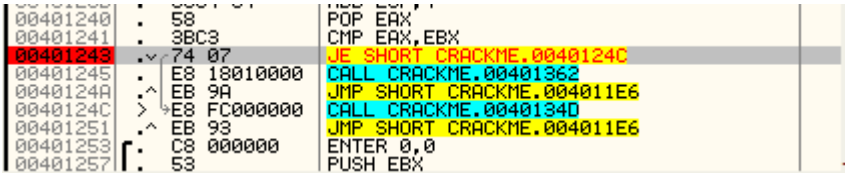
如果突出显示该函数的第一行(地址 401362),提示框中会显示以下信息:



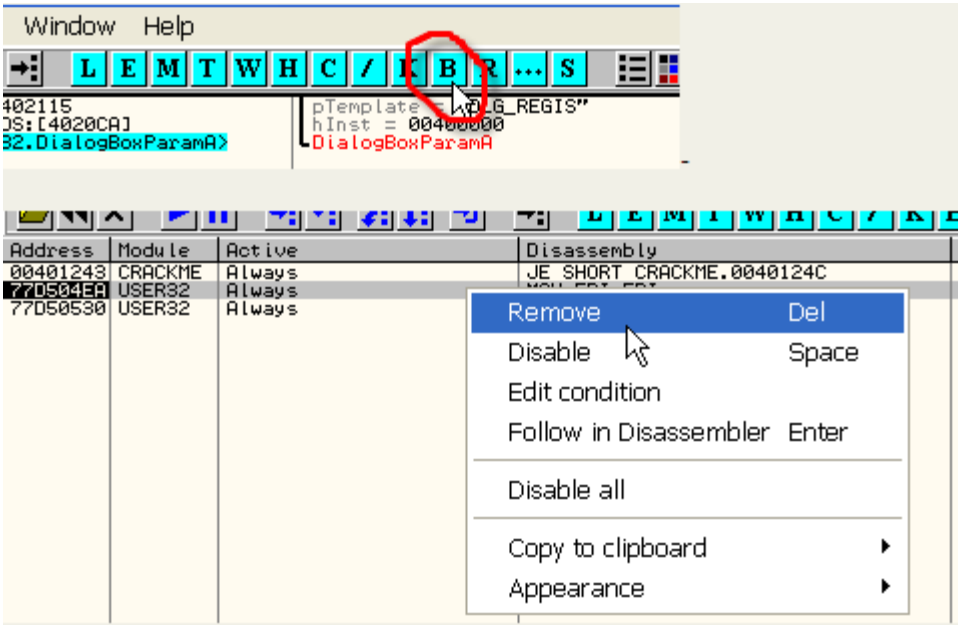
调试器提示调用来至于 401245,单击鼠标右键选择-Goto-CALL from 401245。



这里是典型的比较代码:一个分支是 No luck!,另一个分支是 Great work。这里我们可不能错过。我们在条件跳转处设置断点。

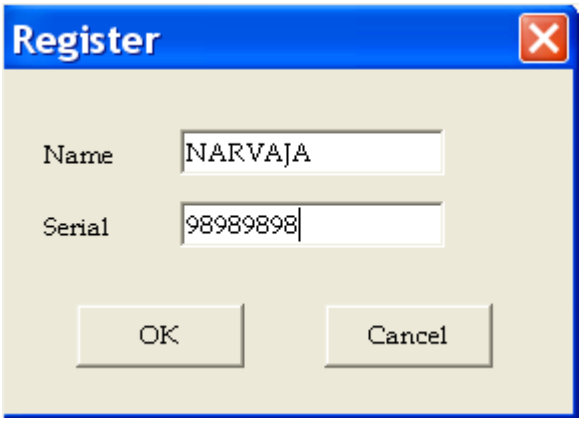


同时取消 MessageBox 入口处的断点。这里我们可以单击调试器窗口工具栏中的 B 按钮,即显示断点窗口:

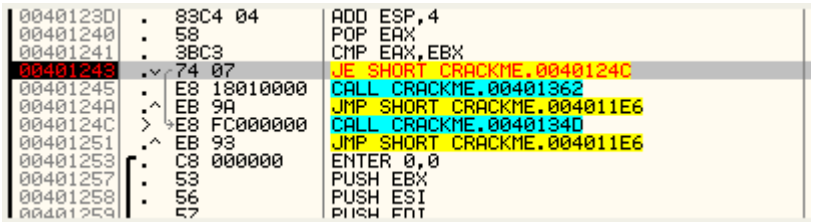


单击鼠标右键选择-Remove。删除两个调用 MessageBox 的断点,只留下 401243 处条件判断语句的断点。

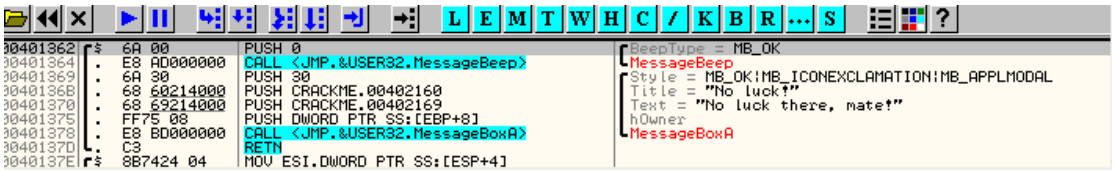
我们 F9 运行起来,弹出 No luck 消息框,刚才我们已经跟踪过一次了。单击确定关闭消息框,然后继续打开注册窗口,输入名称和序列号。



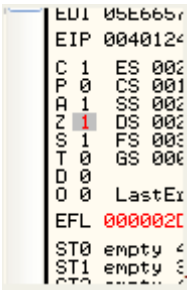
单击 OK(确定)。



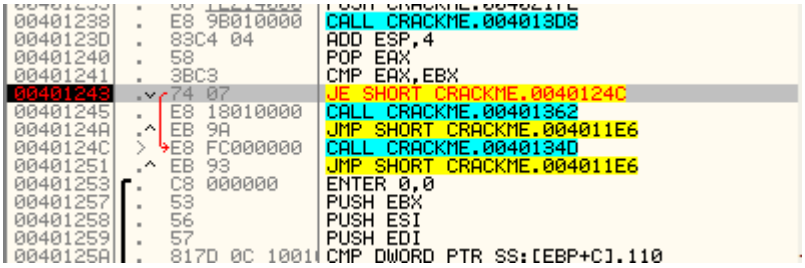
跳转不会发生,因为 EAX 和 EBX 的值不相等。因此 CALL 401362 弹出 No luck 消息框,如果你忘记了。可以单击鼠标右键-Goto-Expression 来到 401362 处看一看。



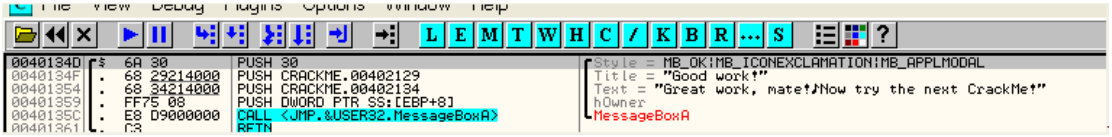
但是如果我们修改零标志位 Z 的值呢?将 Z 标志位的值修改为相反的状态。



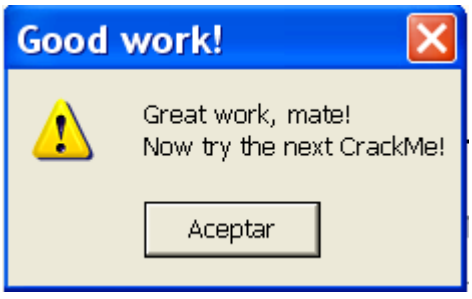
Z 标志位为 1 表示 EAX=EBX。即 EAX 与 EBX 的差值为零。跳转将会发生。



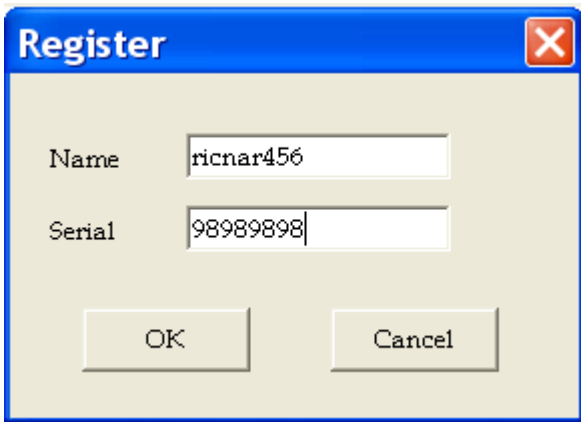
现在会调用另一个函数。如下:



按 F9 运行起来。

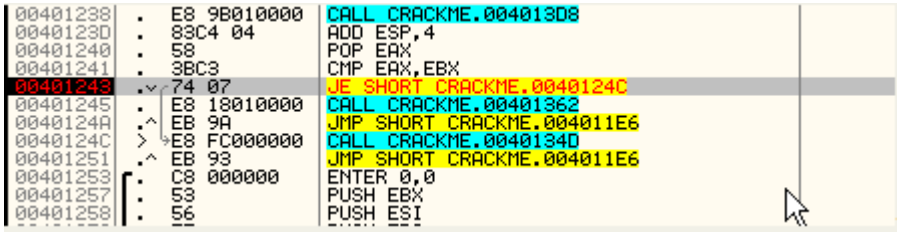


因此,比较是验证序列号的一个关键点。如果 EAX 和 EBX 的值相等则弹出 Great work 消息框,否则弹出 No luck 消息框。我们注意到代码中有两处 No luck 的消息框提示,只有一个是最后一次的提示。CrackMe 当名称中包含数字的时候会提示 No luck 消息框,当序列号错误的时候还会提示 No luck 消息框。

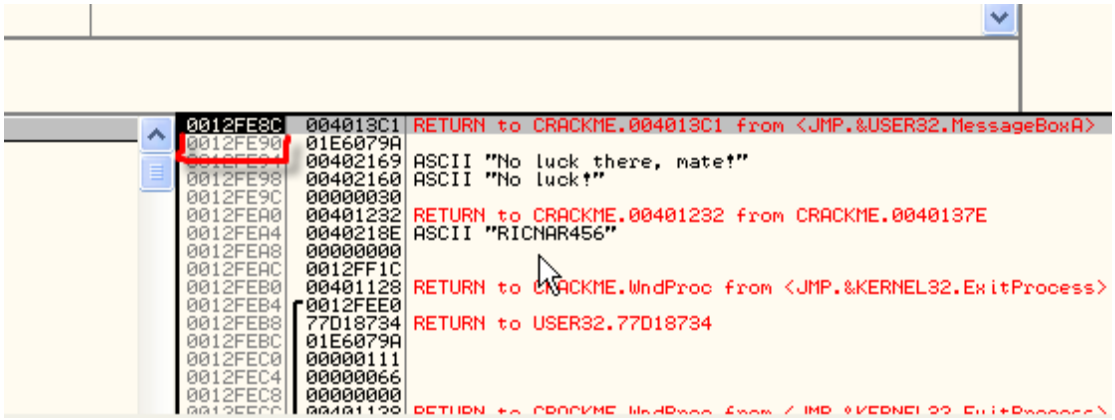


输入包含数字的名称。单击 OK。

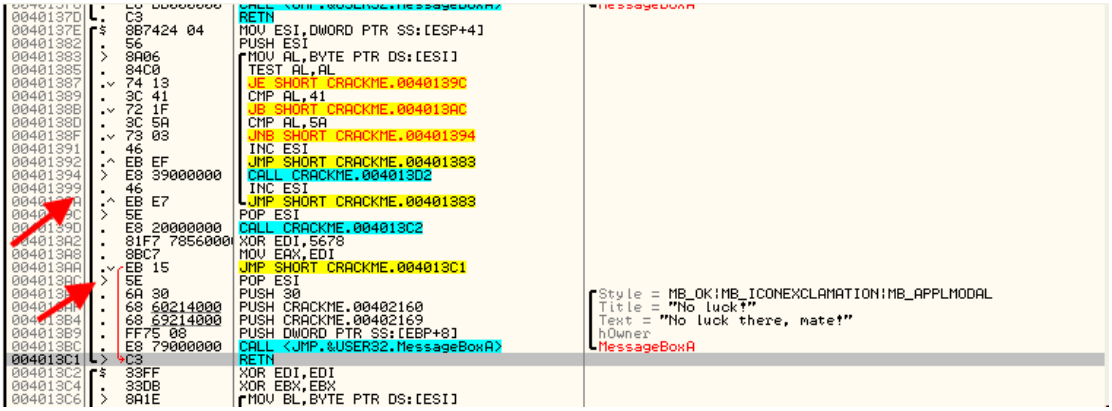
弹出消息框,单击确定,就会在条件判断处中断下来。



F9 运行以后,又弹出消息框。

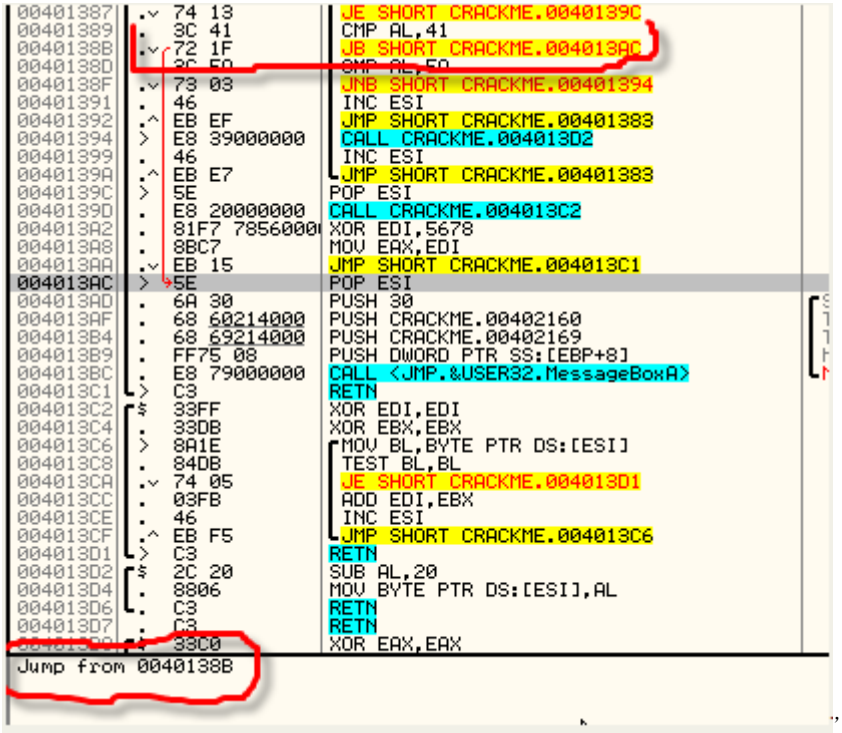


返回地址为 4013C1。

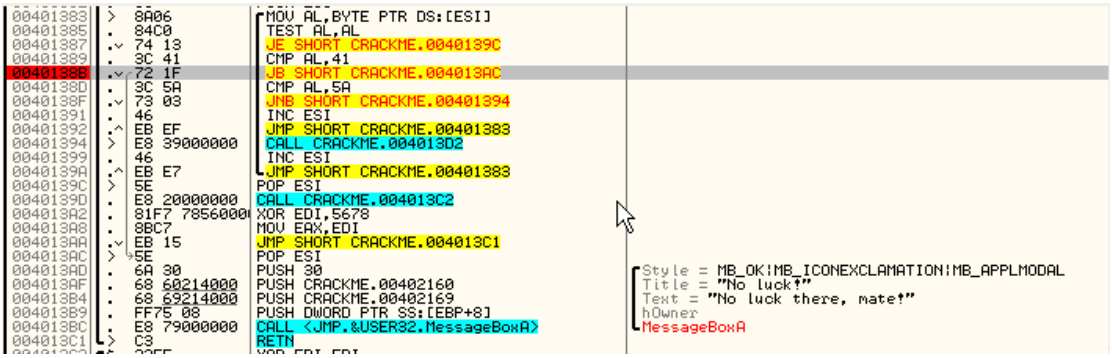


OD 的分析显示该子过程开始于 40137E,结束于 4013C1。调用 MessageBox 后就返回了。

请注意,4013AC 的箭头（>),这表示,该处是一个跳转地址。我们顺着红线可以找到跳转来至于哪里。



现在我们面对的是更多的比较和条件跳转指令,可以导致弹出 No luck 消息框。我们在这里设置断点。

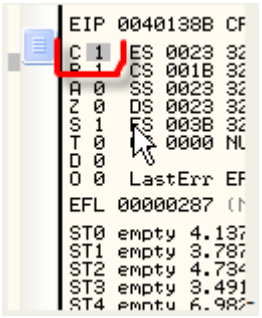


再次让程序运行起来,还是输入刚才的用户名和序列号,单击确定。

```
00401375  .  FF75 08       PUSH DWORD PTR SS:[EBP+8]
00401378  .  E8 BD000000   CALL <JMP.&USER32.MessageBoxA>
0040137D  .  C3            RETN
0040137E  $  8B7424 04     MOV ESI,DWORD PTR SS:[ESP+4]
00401382  .  56            PUSH ESI
00401383  >  8A06          MOV AL,BYTE PTR DS:[ESI]
00401385  .  84C0          TEST AL,AL
00401387  .v 74 13         JE SHORT CRACKME.0040139C
00401389  .  3C 41         CMP AL,41
0040138B  .v 72 1F         JB SHORT CRACKME.004013AC
0040138D  .  3C 5A         CMP AL,5A
0040138F  .v 73 03         JNB SHORT CRACKME.00401394
00401391  .  46            INC ESI
00401392  .^ EB EF         JMP SHORT CRACKME.00401383
00401394  >  E8 39000000   CALL CRACKME.004013D2
00401399  .  46            INC ESI
0040139A  .^ EB E7         JMP SHORT CRACKME.00401383
0040139C  >  5E            POP ESI
0040139D  .  E8 20000000   CALL CRACKME.004013C2
004013A2  .  81F7 7856000( XOR EDI,5678
004013A8  .  8BC7          MOV EAX,EDI
004013AA  .v EB 15         JMP SHORT CRACKME.004013C1
004013AC  >  5E            POP ESI
004013AD  .  6A 30         PUSH 30
004013AF  .  68 60214000   PUSH CRACKME.00402160
004013B4  .  68 69214000   PUSH CRACKME.00402169
```

这里的代码会循环检测用户名的所有字母。一旦用户名中包含数字的话,就会跳转到 No luck 消息框处。每检测一个字母,就会中断在 40138B 处。按 F9 键就会接着检测下一个字母。

当循环到第 7 次的时候(也就是用户名 ricnar456 中的第一个数字-'4'的时候),跳转就会发生,但是我们可以修改进位标志位 C 的值。



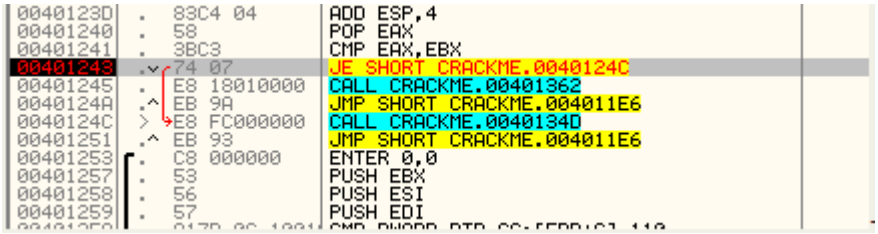
JB 条件跳转是根据进位标志位 C 来决定是否跳转的,我们通过双击 C 标志位修改其值。

```
00401383  >  8A06          MOV AL,BYTE PTR DS:[ESI]
00401385  .  84C0          TEST AL,AL
00401387  .v 74 13         JE SHORT CRACKME.0040139C
00401389  .  3C 41         CMP AL,41
0040138B  .v 72 1F         JB SHORT CRACKME.004013AC
0040138D  .  3C 5A         CMP AL,5A
0040138F  .v 73 03         JNB SHORT CRACKME.00401394
00401391  .  46            INC ESI
00401392  .^ EB EF         JMP SHORT CRACKME.00401383
00401394  >  E8 39000000   CALL CRACKME.004013D2
00401399  .  46            INC ESI
0040139A  .^ EB E7         JMP SHORT CRACKME.00401383
0040139C  >  5E            POP ESI
0040139D  .  E8 20000000   CALL CRACKME.004013C2
004013A2  .  81F7 7856000( XOR EDI,5678
004013A8  .  8BC7          MOV EAX,EDI
004013AA  .v EB 15         JMP SHORT CRACKME.004013C1
004013AC  >  5E            POP ESI
004013AD  .  6A 30         PUSH 30
004013AF  .  68 60214000   PUSH CRACKME.00402160
004013B4  .  68 69214000   PUSH CRACKME.00402169
004013B9  .  FF75 08       PUSH DWORD PTR SS:[EBP+8]
004013BC  .  E8 79000000   CALL <JMP.&USER32.MessageBoxA>
```

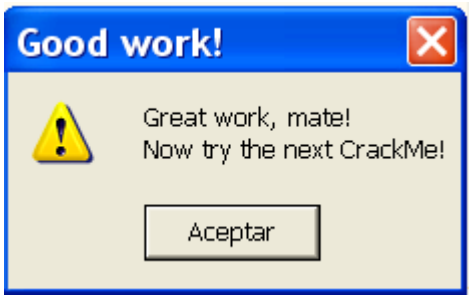
这样,跳转就不会发生了。针对于其余的数字我们同样可以这么做让跳转不发生。

```
00401238  .  E8 9B010000   CALL CRACKME.004013D8
0040123D  .  83C4 04       ADD ESP,4
00401240  .  58            POP EAX
00401241  .  3BC3          CMP EAX,EBX
00401243  .v 74 07         JE SHORT CRACKME.0040124C
00401245  .  E8 18010000   CALL CRACKME.00401362
0040124A  .^ EB 9A         JMP SHORT CRACKME.004011E6
0040124C  >  E8 FC000000   CALL CRACKME.0040134D
00401251  .^ EB 93         JMP SHORT CRACKME.004011E6
00401253  .  C8 000000     ENTER 0,0
00401257  .  53            PUSH EBX
00401258  .  56            PUSH ESI
00401259  .  57            PUSH EDI
0040125A  .  817D 0C 1001( CMP DWORD PTR SS:[EBP+C],110
00401261  .v 74 34         JE SHORT CRACKME.00401297
00401263  .  817D 0C 1101( CMP DWORD PTR SS:[EBP+C],111
0040126A  .v 74 35         JE SHORT CRACKME.004012A1
```

接下来我依然要来比较 EAX 和 EBX 的值,向之前一样通过修改零标志位 Z 来让跳转发生。

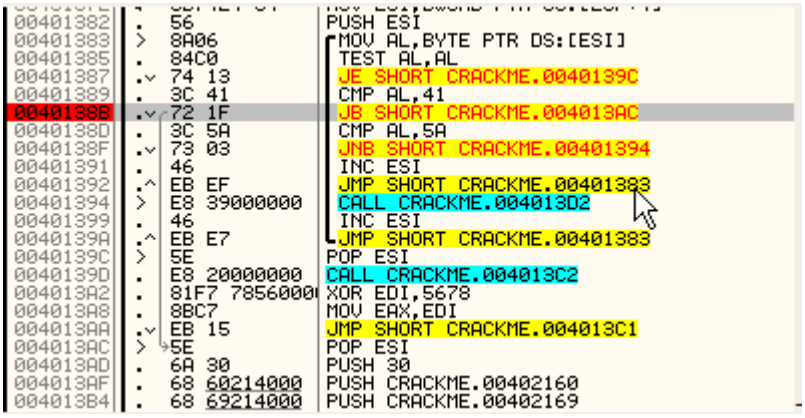


按 F9 键。

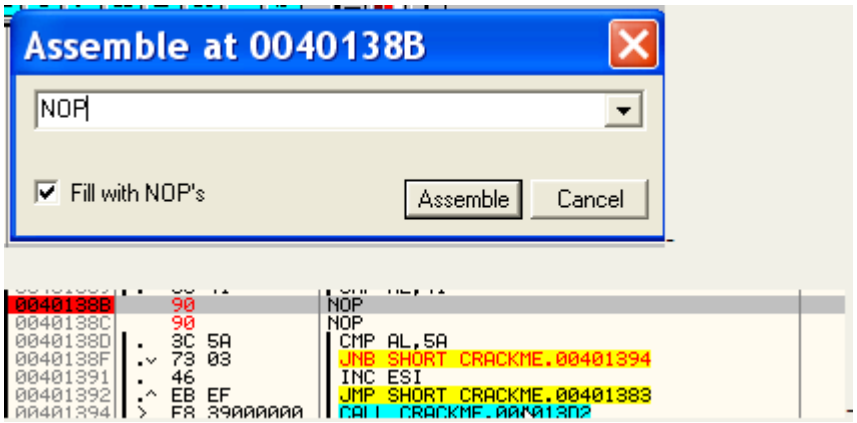


我们不可能通过修改 C 和 Z 标志位来实现功能注册吧!我们需要通过别的方式来让 CrackMe 通过注册。

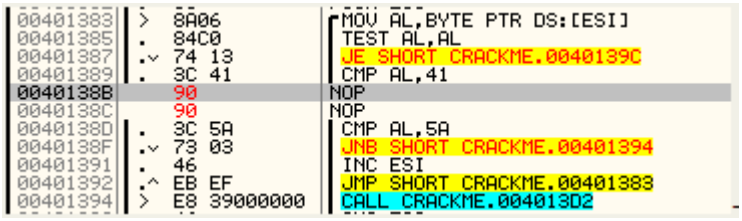
选择第一个条件分支(设置一个断点)。



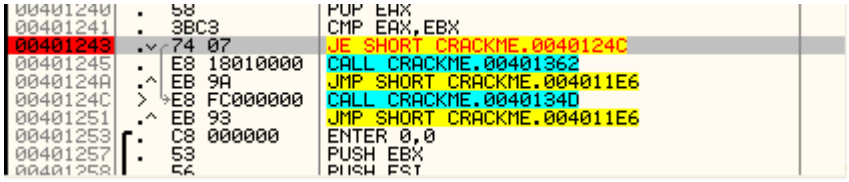
这次我们不改变进位标志位 C,而是直接简单粗暴的把该指令填 NOP。在该指令上单击鼠标右键选择-Assemble 输入 NOP。



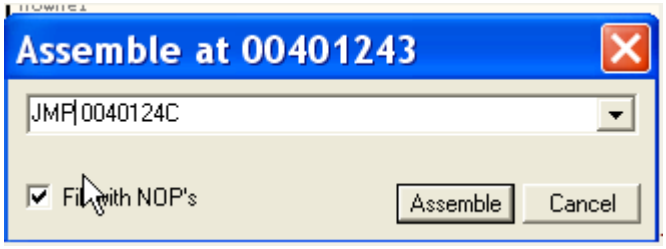
可以将断点去掉了,只需要按下 F2 键。

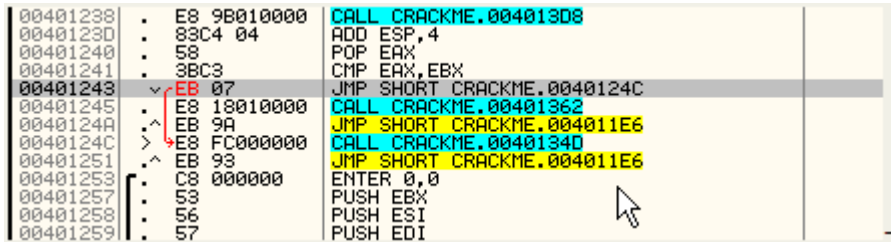


转到第二个关键跳转处。

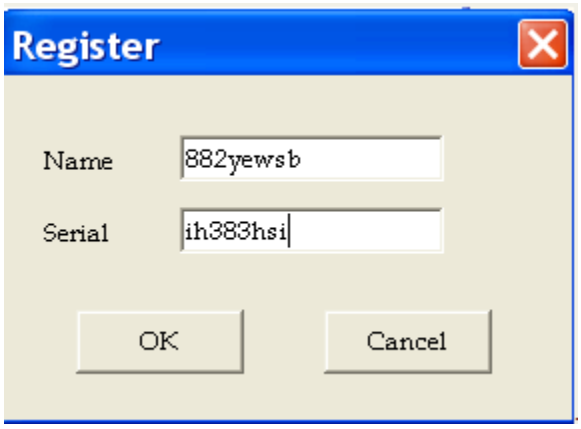


这里我们需要跳转始终成立,所以我们将 JE 改成 JMP。

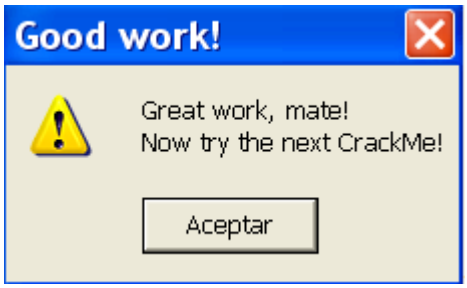




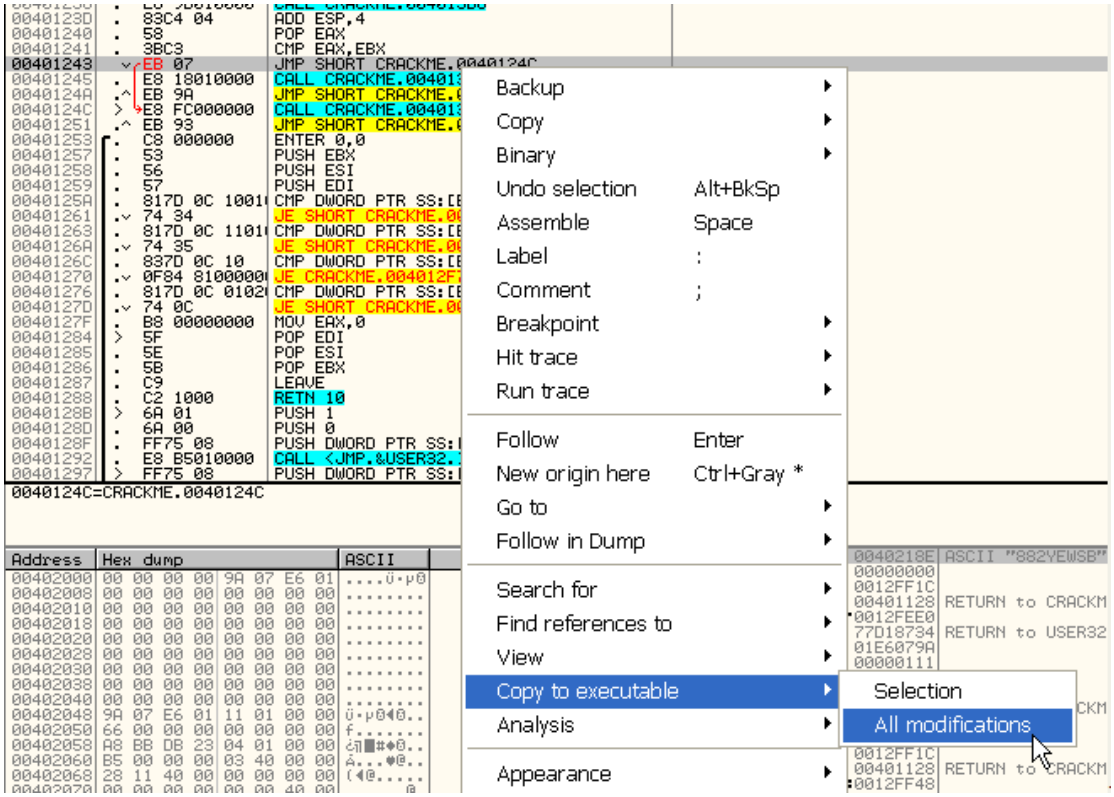
然后还是按 F2 删除断点。然后按 F9 键。输入用户名和序列号。



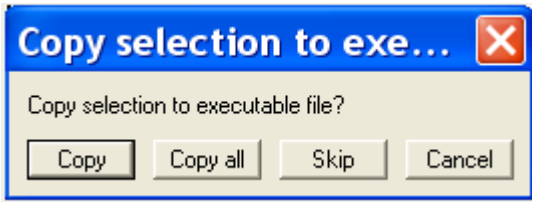
单击确定。



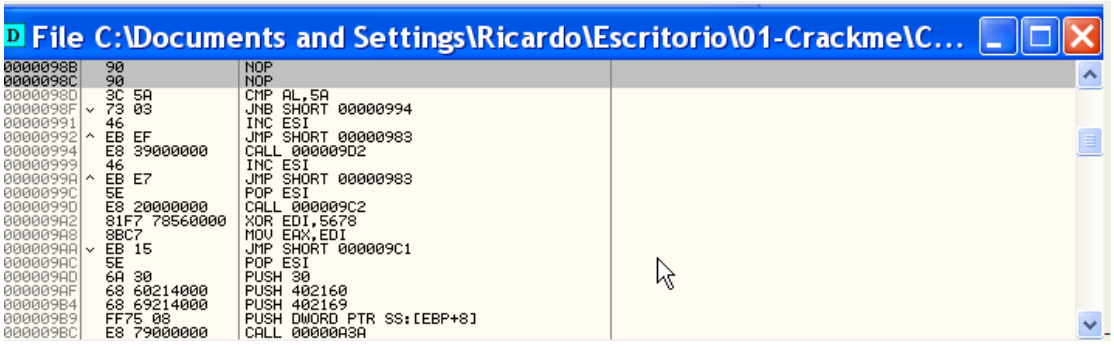
当前这些修改是在针对于内存的,并不会影响可执行文件。如果需要保存修改到可执行文件的话,请执行以下操作:



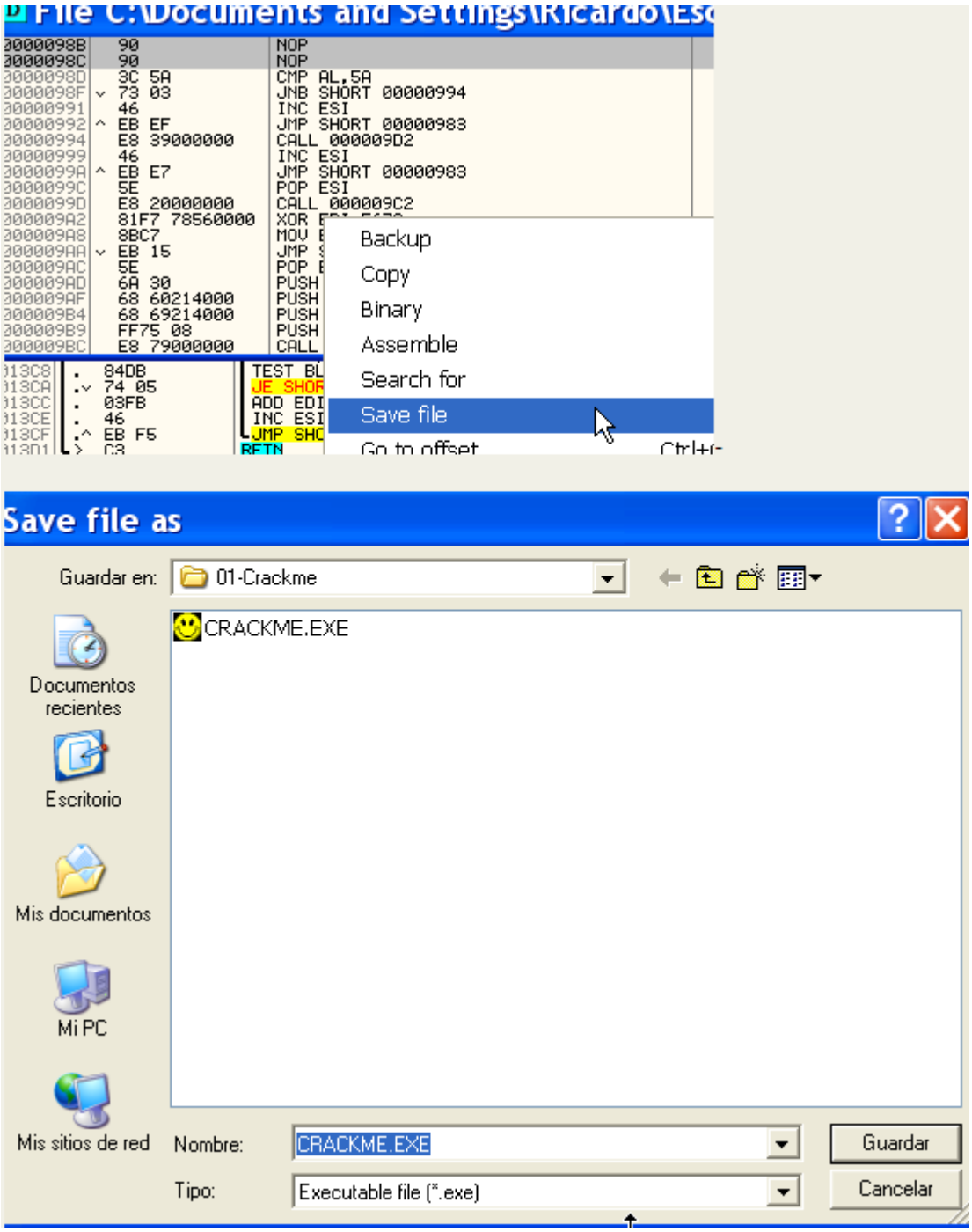
在反汇编窗口的任意位置单击鼠标右键选择-Copy to executable-All modifications。然后会显示:



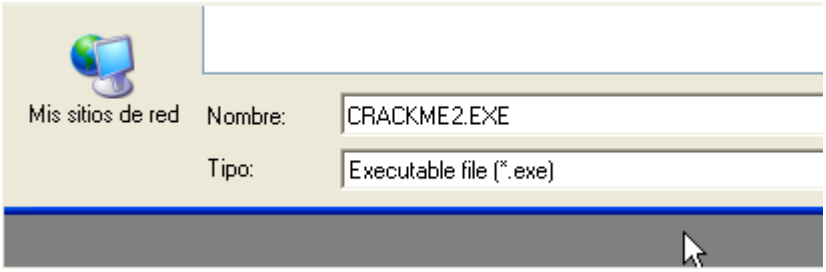
选择 Copy all 确保拷贝我们所做的所有修改。



弹出一个新窗口,再次单击鼠标右键选择-Save File。

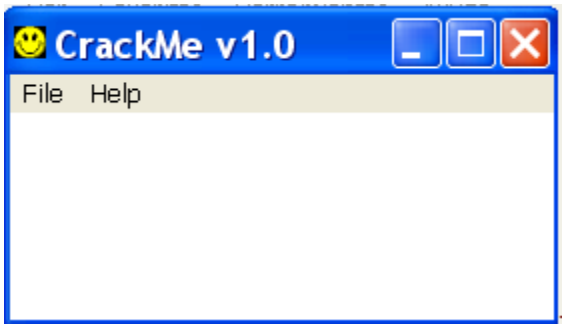


将文件保存一个新名字(原来的可执行文件我们还有用处,其实这个时候覆盖也是行不通的,这个时候可执行文件正在被占用)。新的可执行文件我们取名叫 CRACKME2.EXE。

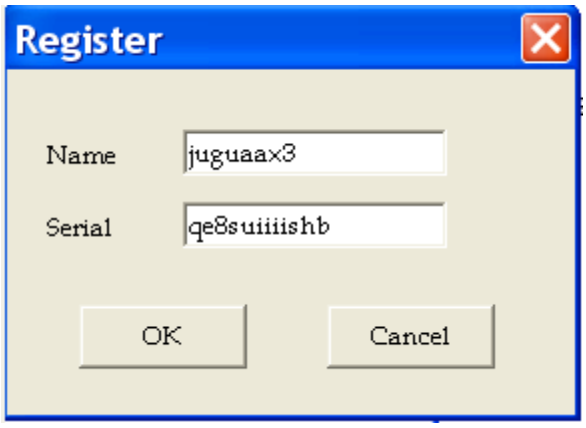


这时候,我们可以关闭 OD 了。

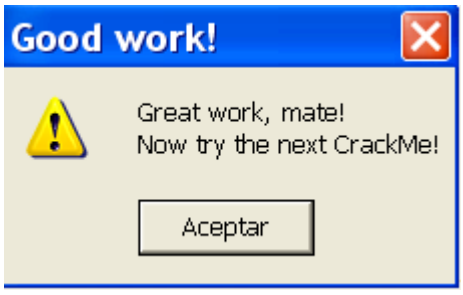
我们双击打开我们 CRACKME2 来看一看修改的效果。



选择-help-Register。



单击 OK。



我们成功破解这个 CrackMe!但是这是不够的-以后我们会尝试找到正确的注册码而非该程序打补丁的方式来通过注册。我们还有很多的东西需要学习。

(下面的一个点点内容是针对于 windows 9x 系列系统的,考虑到 windows 9x 已经淘汰很久了。连 XP 系统前不久都已经退役了。这里就不做翻译了。)