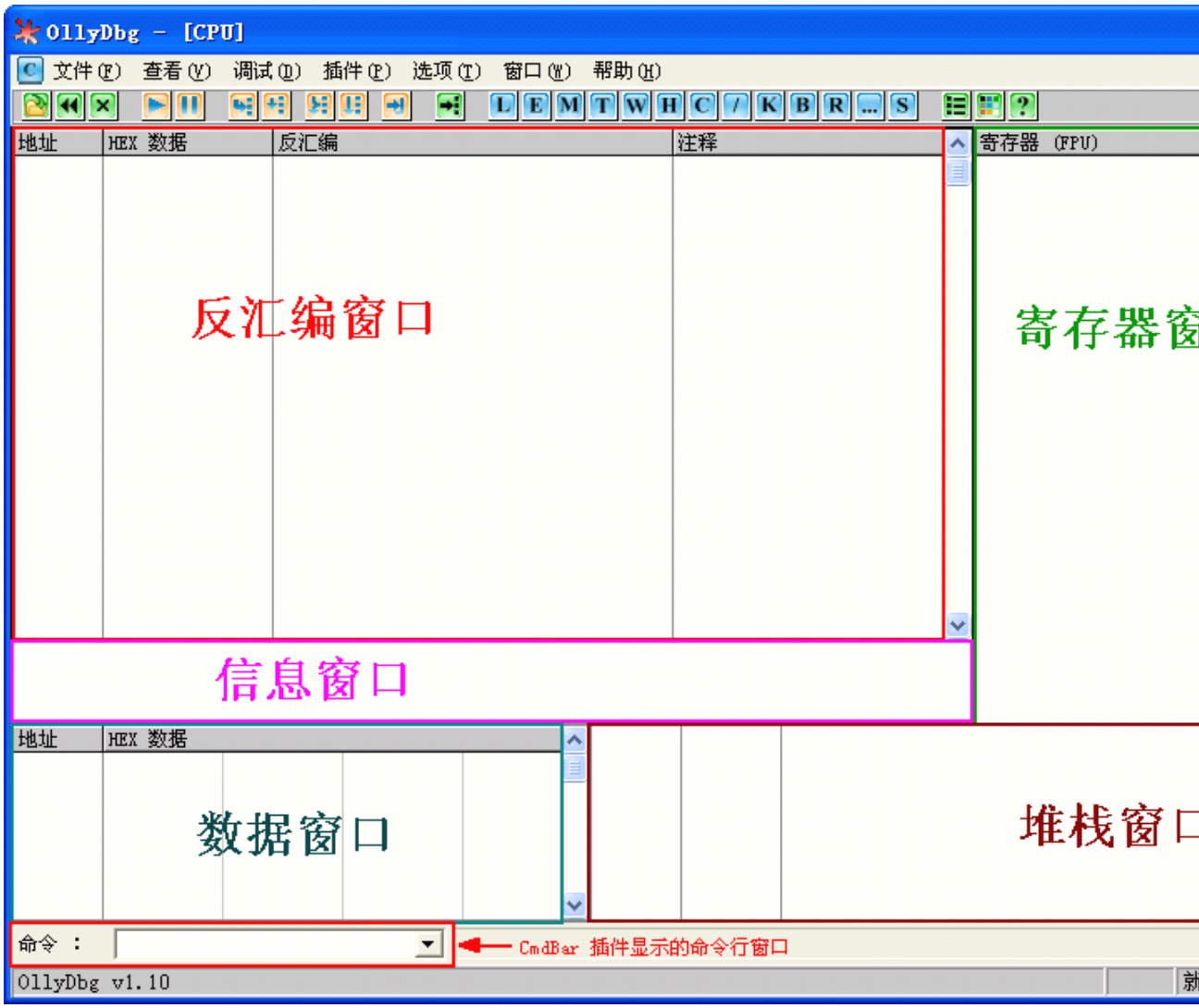
一、OllyDBG 的安装与配置详细

OllyDBG 只要解压到一个目录下，运行 OllyDBG.exe 就可以了。汉化版的发布版本是个 RAR 压缩包，运行 OllyDBG.exe 即可：



OllyDBG 中各个窗口的功能如上图。简单解释一下各个窗口的功能，更详细的内容可以参考 TT 小组翻译的中文帮助：

反汇编窗口：显示被调试程序的反汇编代码，标题栏上的地址、HEX 数据、反汇编、注释可以通过在窗口中右击出现的菜单 界面选项->隐藏标题 或 显示标题 来进行切换是否显示。用鼠标左键点击注释标签可以切换注释显示的方式。

寄存器窗口：显示当前所选线程的 CPU 寄存器内容。同样点击标签 寄存器 (FPU) 可以切换显示寄存器的方式。

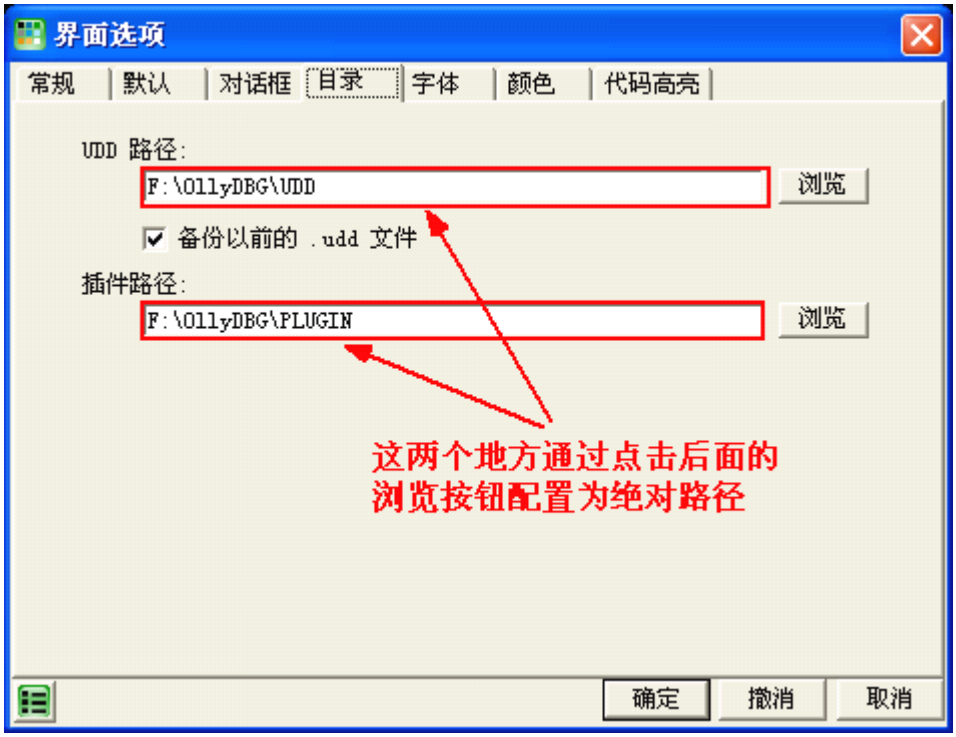
信息窗口：显示反汇编窗口中选中的第一个命令的参数及一些跳转目标地址、字串等。

数据窗口：显示内存或文件的内容。右键菜单可用于切换显示方式。

堆栈窗口：显示当前线程的堆栈。

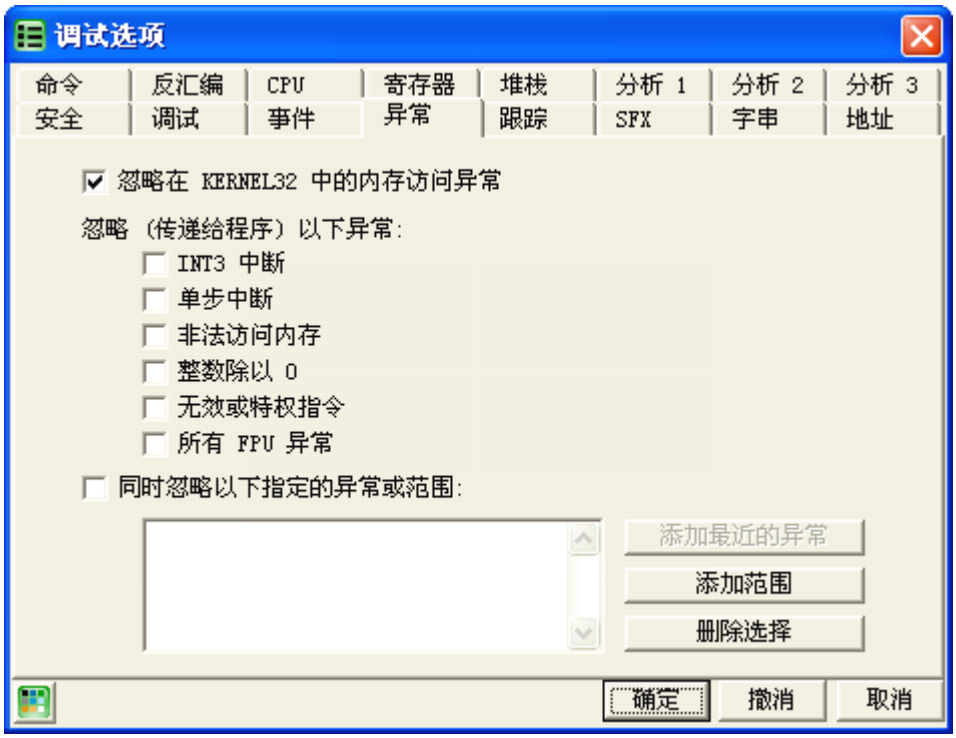
要调整上面各个窗口的大小的话，只需左键按住边框拖动，等调整好了，重新启动一下 OllyDBG 就可以生效了。

启动后我们要把插件及 UDD 的目录配置为绝对路径，点击菜单上的 选项->界面，将会出来一个界面选项的对话框，我们点击其中的目录标签：



因为我这里是把 OllyDBG 解压在 F:\OllyDBG 目录下，所以相应的 UDD 目录及插件目录按图上配置。还有一个常用到的标签就是上图后面那个字体，在这里你可以更改 OllyDBG 中显示的字体。上图中其它的选项可以保留为默认，若有需要也可以自己修改。修改完以后点击确定，弹出一个对话框，说我们更改了插件路径，要重新启动 OllyDBG。在这个对话框上点确定，重新启动一下 OllyDBG，我们再到界面选项中看一下，会发现我们原先设置好的路径都已保存了。有人可能知道插件的作用，但对那个 UDD 目录不清楚。我这简单解释一下：这个 UDD 目录的作用是保存你调试的工作。比如你调试一个软件，设置了断点，添加了注释，一次没做完，这时 OllyDBG 就会把你所做的工作保存到这个 UDD 目录，以便你下次调试时可以继续以前的工作。如果不设置这个 UDD 目录，OllyDBG 默认是在其安装目录下保存这些后缀名为 udd 的文件，时间长了就会显的很乱，所以还是建议专门设置一个目录来保存这些文件。

另外一个重要的选项就是调试选项，可通过菜单 选项->调试设置 来配置：



新手一般不需更改这里的选项，默认已配置好，可以直接使用。建议在对 OllyDBG 已比较熟的情况下再来进行配置。上面那个异常标签中的选项经常会在脱壳中用到，建议在有一定调试基础后学脱壳时再配置这里。

除了直接启动 OllyDBG 来调试外，我们还可以把 OllyDBG 添加到资源管理器右键菜单，这样我们就可以直接在 .exe 及 .dll 文件上点右键选择“用 Ollydbg 打开”菜单来进行调试。要把 OllyDBG 添加到资源管理器右键菜单，只需点菜单 选项->添加到浏览器，将会出现一个对话框，先点击“添加 Ollydbg 到系统资源管理器菜单”，再点击“完成”按钮即可。要从右键菜单中删除也很简单，还是这个对话框，点击“从系统资源管理器菜单删除 Ollydbg”，再点击“完成”就行了。

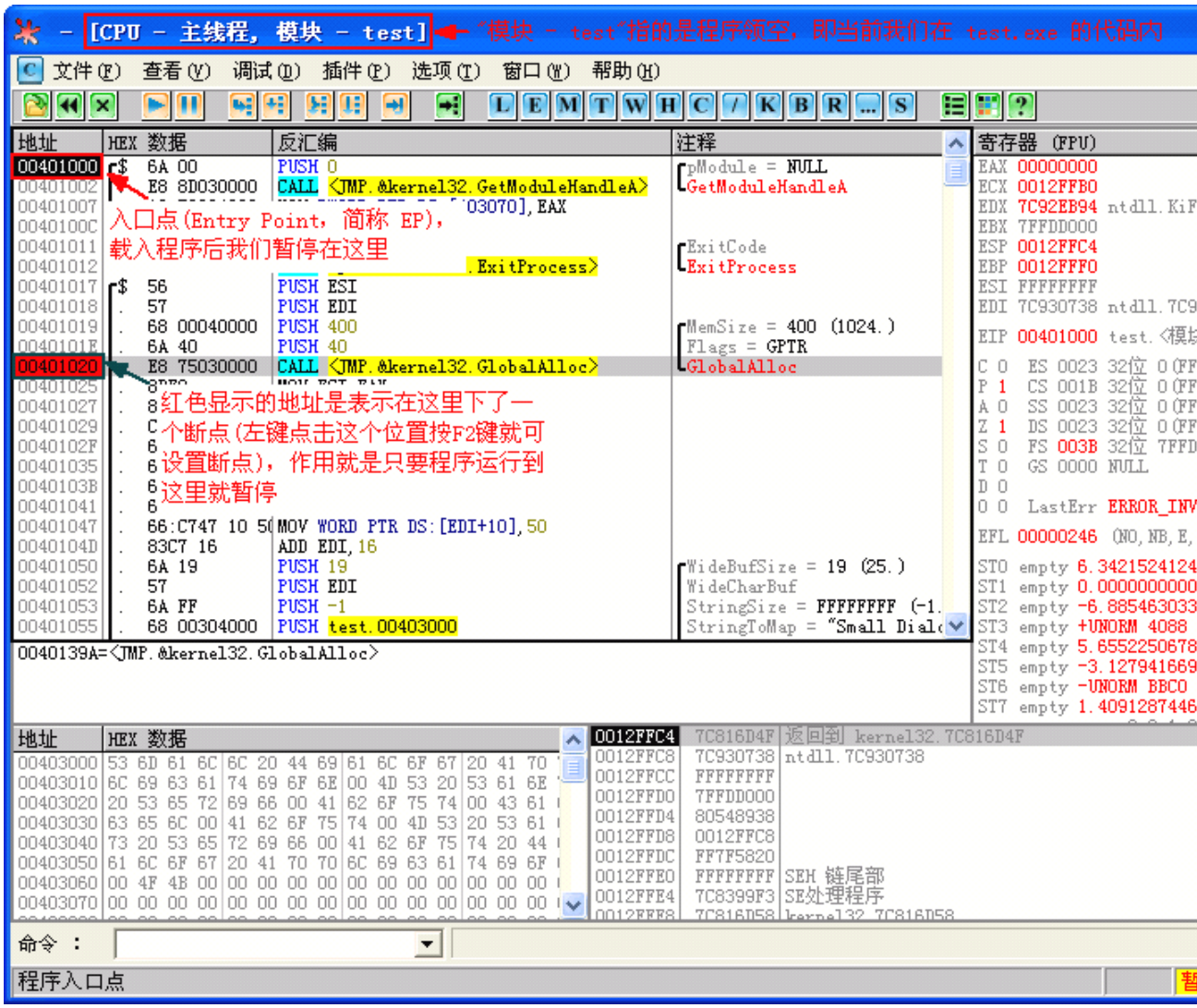
OllyDBG 支持插件功能，插件的安装也很简单，只要把下载的插件（一般是个 DLL 文件）复制到 OllyDBG 安装目录下的 PLUGIN 目录中就可以了，OllyDBG 启动时会自动识别。要注意的是 OllyDBG 1.10 对插件的个数有限制，最多不能超过 32 个，否则会出错。建议插件不要添加的太多。

到这里基本配置就完成了，OllyDBG 把所有配置都放在安装目录下的 ollydbg.ini 文件中。

二、基本调试方法

OllyDBG 有三种方式来载入程序进行调试，一种是点击菜单 文件->打开 （快捷键是 F3）来打开一个可执行文件进行调试，另一种是点击菜单 文件->附加 来附加到一个已运行的进程上进行调试。注意这里要附加的程序必须已运行。第三种就是用右键菜单来载入程序（不知这种算不算）。一般情况下我们选第一种方式。比如我们选择一个 test.exe 来调试，通过菜单 文件->打开 来载入这个程序，OllyDBG 中显示的内容将

会是这样：



调试中我们经常要用到的快捷键有这些：

F2：设置断点，只要在光标定位的位置（上图中灰色条）按 F2 键即可，再按一次 F2 键则会删除断点。（相当于 SoftICE 中的 F9）

F8：单步步过。每按一次这个键执行一条反汇编窗口中的一条指令，遇到 CALL 等子程序不进入其代码。（相当于 SoftICE 中的 F10）

F7：单步步入。功能同单步步过(F8)类似，区别是遇到 CALL 等子程序时会进入其中，进入后首先会停留在子程序的第一条指令上。（相当于 SoftICE 中的 F8）

F4：运行到选定位置。作用就是直接运行到光标所在位置处暂停。（相当于 SoftICE 中的 F7）

F9：运行。按下这个键如果没有设置相应断点的话，被调试的程序将直接开始运行。（相当于 SoftICE 中的 F5）

CTR+F9：执行到返回。此命令在执行到一个 ret (返回指令)指令时暂停，常用于从系统领空返回到我们调试的程序领空。（相当于 SoftICE 中的 F12）

ALT+F9：执行到用户代码。可用于从系统领空快速返回到我们调试的程序领空。（相当于 SoftICE 中的 F11）

上面提到的几个快捷键对于一般的调试基本上已够用了。要开始调试只需设置好断点，找到你感兴趣的代码段再按 F8 或 F7 键来一条条分析指令功能就可以了。就写到这了，改天有空再接着灌。

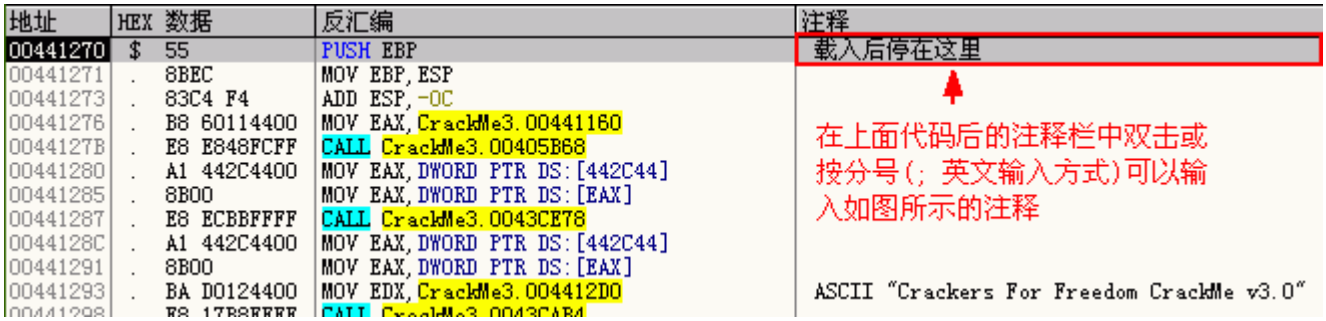
二）一字串参考

上一篇是使用入门，现在我们开始正式进入破解。今天的目标程序是看雪兄《加密与解密》第一版附带光盘中的 crackmes.cjb.net 镜像打包中的 CFF Crackme #3，采用用户名/序列号保护方式。原版加了个 UPX 的壳。刚开始学破解先不涉及壳的问题，我们主要是熟悉用 OllyDBG 来破解的一般方法。我这里把壳脱掉来分析，附件是脱壳后的文件，直接就可以拿来用。先说一下一般软件破解的流程：拿到一个软件先别接着马上用 OllyDBG 调试，先运行一下，有帮助文档的最好先看一下帮助，熟悉一下软件的使用方法，再看看注册的方式。如果是序列号方式可以先输个假的来试一下，看看有什么反应，也给我们破解留下一些有用的线索。如果没有输入注册码的地方，要考虑一下是不是读取注册表或 Key 文件（一般称 keyfile，就是程序读取一个文件中的内容来判断是否注册），这些可以用其它工具来辅助分析。如果这些都不是，原程序只是一个功能不全的试用版，那要注册为正式版本就要自己来写代码完善了。有点跑题了，呵呵。获得程序的一些基本信息后，还要用查壳的工具来查一下程序是否加了壳，若没壳的话看看程序是什么编译器编的，如 VC、Delphi、VB 等。这样的查壳工具有 PEiD 和 FI。有壳的话我们要尽量脱了壳后再来用 OllyDBG 调试，特殊情况下也可带壳调试。下面进入正题：

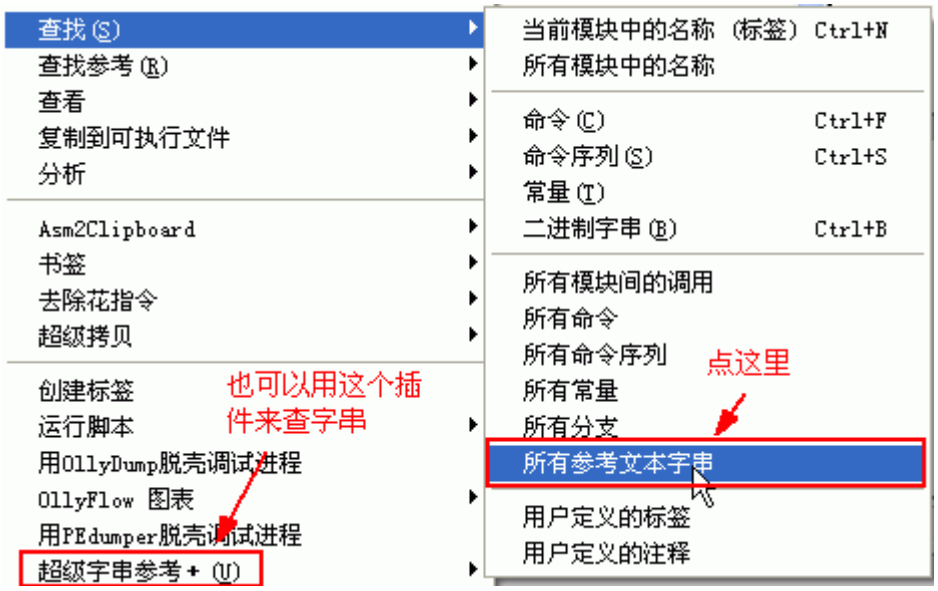
我们先来运行一下这个 crackme（用 PEiD 检测显示是 Delphi 编的），界面如图：

这个 crackme 已经把用户名和注册码都输好了，省得我们动手^_^。我们在那个“Register now !”按钮上点击一下，将会跳出一个对话框：

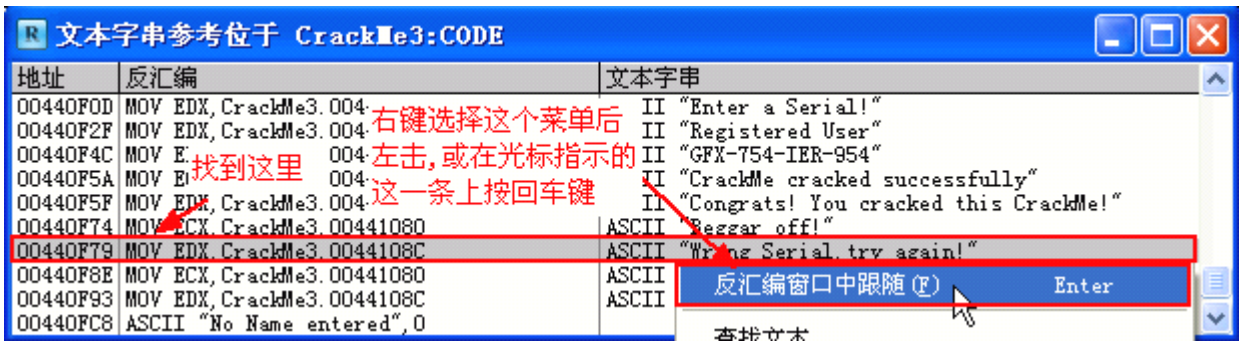
好了，今天我们就从这个错误对话框中显示的“Wrong Serial, try again!”来入手。启动 OllyDBG，选择菜单 文件->打开 载入 CrackMe3.exe 文件，我们会停在这里：



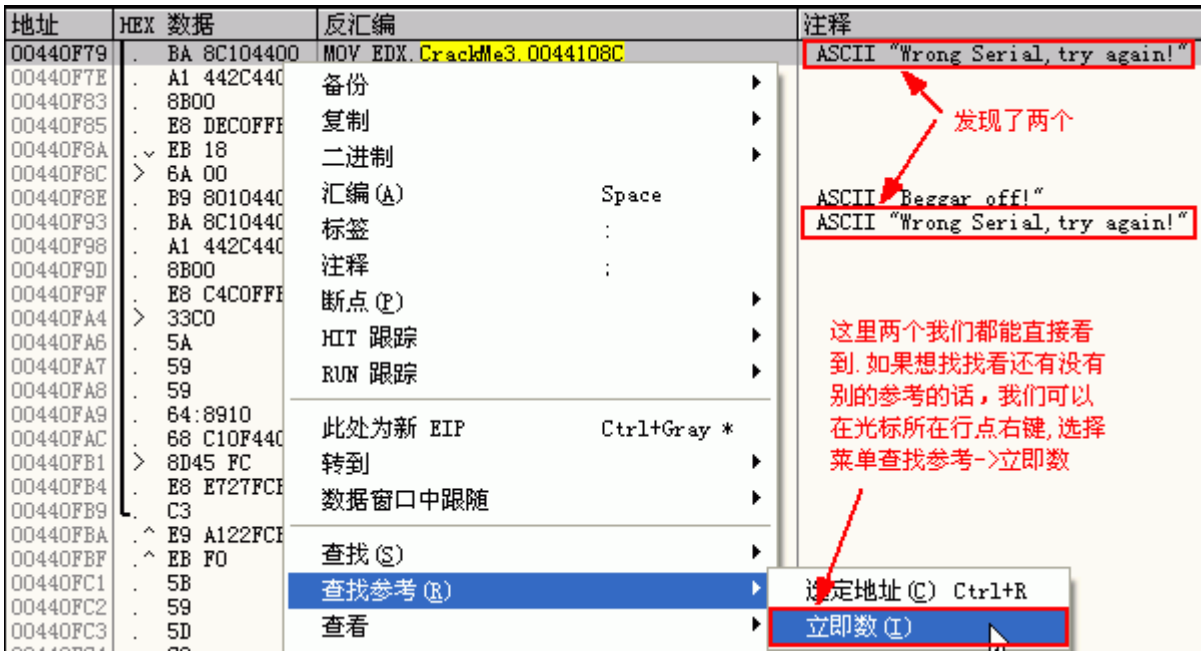
我们在反汇编窗口中右击，出来一个菜单，我们在 查找->所有参考文本字串 上左键点击：



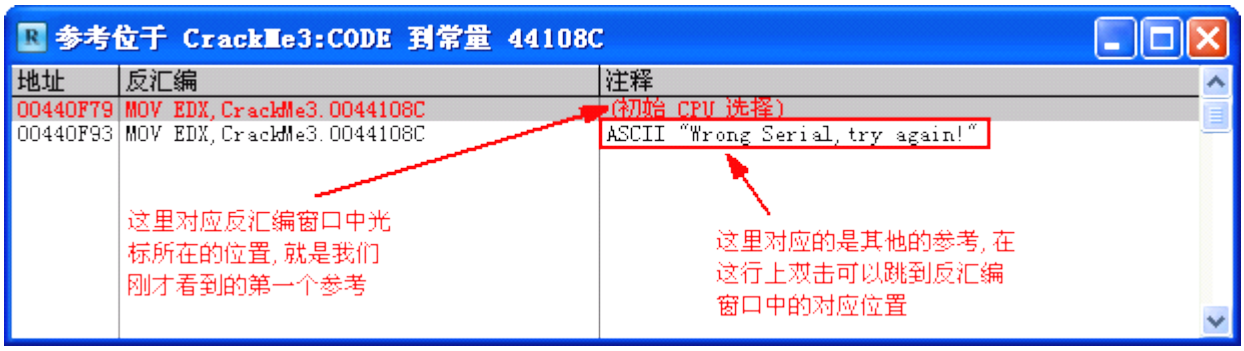
当然如果用上面那个 超级字串参考＋ 插件会更方便。但我们的目标是熟悉 OllyDBG 的一些操作，我就尽量使用 OllyDBG 自带的功能，少用插件。好了，现在出来另一个对话框，我们在这个对话框里右击，选择“查找文本”菜单项，输入“Wrong Serial, try again!”的开头单词“Wrong”（注意这里查找内容要区分大小写）来查找，找到一处：



在我们找到的字串上右击，再在出来的菜单上点击“反汇编窗口中跟随”，我们来到这里：



见上图，为了看看是否还有其他的参考，可以通过选择右键菜单查找参考->立即数，会出来一个对话框：



分别双击上面标出的两个地址，我们会来到对应的位置：

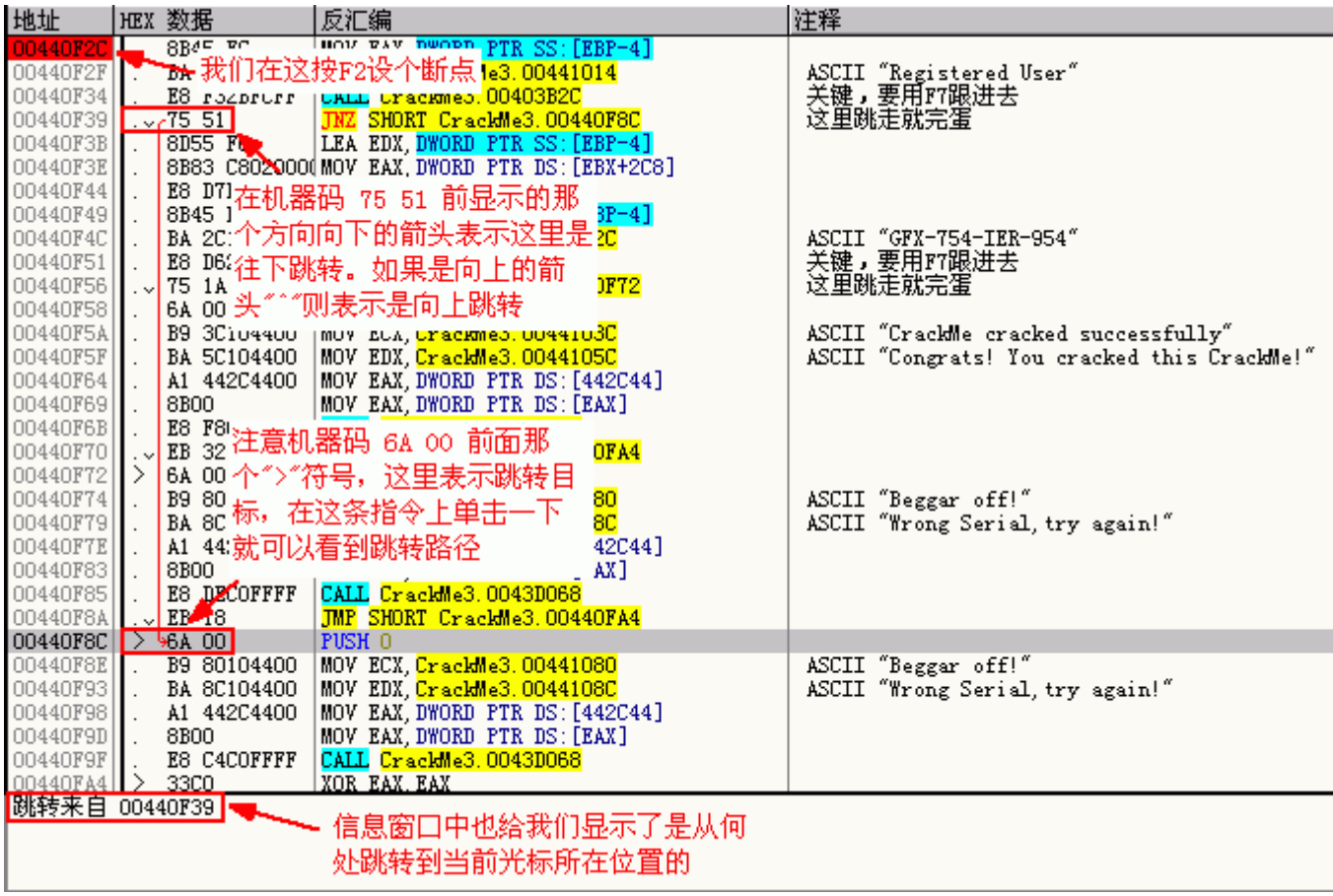
```
00440F79 |. BA 8C104400      MOV EDX,CrackMe3.0044108C                ; ASCII "Wrong
Serial,try again!"
00440F7E |. A1 442C4400      MOV EAX,DWORD PTR DS:[442C44]
00440F83 |. 8B00              MOV EAX,DWORD PTR DS:[EAX]
00440F85 |. E8 DEC0FFFF      CALL CrackMe3.0043D068
00440F8A |. EB 18             JMP SHORT CrackMe3.00440FA4
00440F8C |> 6A 00             PUSH 0
00440F8E |. B9 80104400      MOV ECX,CrackMe3.00441080                ; ASCII "Beggar
off!"
00440F93 |. BA 8C104400      MOV EDX,CrackMe3.0044108C                ; ASCII "Wrong
Serial,try again!"
00440F98 |. A1 442C4400      MOV EAX,DWORD PTR DS:[442C44]
00440F9D |. 8B00              MOV EAX,DWORD PTR DS:[EAX]
00440F9F |. E8 C4C0FFFF      CALL CrackMe3.0043D068
```

我们在反汇编窗口中向上滚动一下再看看：

```
00440F2C |. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
00440F2F |. BA 14104400       MOV EDX,CrackMe3.00441014                ; ASCII
"Registered User"
00440F34 |. E8 F32BFCFF      CALL CrackMe3.00403B2C                   ; 关键，要用 F7
跟进去
00440F39 |. 75 51             JNZ SHORT CrackMe3.00440F8C              ; 这里跳走就完蛋
00440F3B |. 8D55 FC          LEA EDX,DWORD PTR SS:[EBP-4]
00440F3E |. 8B83 C8020000   MOV EAX,DWORD PTR DS:[EBX+2C8]
00440F44 |. E8 D7FEFDFF      CALL CrackMe3.00420E20
00440F49 |. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
00440F4C |. BA 2C104400       MOV EDX,CrackMe3.0044102C                ; ASCII
"GFX-754-IER-954"
00440F51 |. E8 D62BFCFF      CALL CrackMe3.00403B2C                   ; 关键，要用 F7
跟进去
00440F56 |. 75 1A             JNZ SHORT CrackMe3.00440F72              ; 这里跳走就完蛋
00440F58 |. 6A 00 PUSH 0
00440F5A |. B9 3C104400       MOV ECX,CrackMe3.0044103C                ; ASCII
"CrackMe cracked successfully"
00440F5F |. BA 5C104400       MOV EDX,CrackMe3.0044105C                ; ASCII
"Congrats! You cracked this CrackMe!"
00440F64 |. A1 442C4400      MOV EAX,DWORD PTR DS:[442C44]
00440F69 |. 8B00              MOV EAX,DWORD PTR DS:[EAX]
00440F6B |. E8 F8C0FFFF      CALL CrackMe3.0043D068
00440F70 |. EB 32             JMP SHORT CrackMe3.00440FA4
```

```
00440F72 |> 6A 00              PUSH 0
00440F74 |. B9 80104400      MOV ECX,CrackMe3.00441080                ; ASCII "Beggar
off!"
00440F79 |. BA 8C104400      MOV EDX,CrackMe3.0044108C                ; ASCII "Wrong
Serial,try again!"
00440F7E |. A1 442C4400      MOV EAX,DWORD PTR DS:[442C44]
00440F83 |. 8B00              MOV EAX,DWORD PTR DS:[EAX]
00440F85 |. E8 DEC0FFFF      CALL CrackMe3.0043D068
00440F8A |. EB 18             JMP SHORT CrackMe3.00440FA4
00440F8C |> 6A 00             PUSH 0
00440F8E |. B9 80104400      MOV ECX,CrackMe3.00441080                ; ASCII "Beggar
off!"
00440F93 |. BA 8C104400      MOV EDX,CrackMe3.0044108C                ; ASCII "Wrong
Serial,try again!"
00440F98 |. A1 442C4400      MOV EAX,DWORD PTR DS:[442C44]
00440F9D |. 8B00              MOV EAX,DWORD PTR DS:[EAX]
00440F9F |. E8 C4C0FFFF      CALL CrackMe3.0043D068
```

大家注意看一下上面的注释，我在上面标了两个关键点。有人可能要问，你怎么知道那两个地方是关键点？其实很简单，我是根据查看是哪条指令跳到“wrong serial,try again”这条字串对应的指令来决定的。如果你在 调试选项->CPU 标签中把“显示跳转路径”及其下面的两个“如跳转未实现则显示灰色路径”、“显示跳转到选定命令的路径”都选上的话，就会看到是从什么地方跳到出错字串处的：

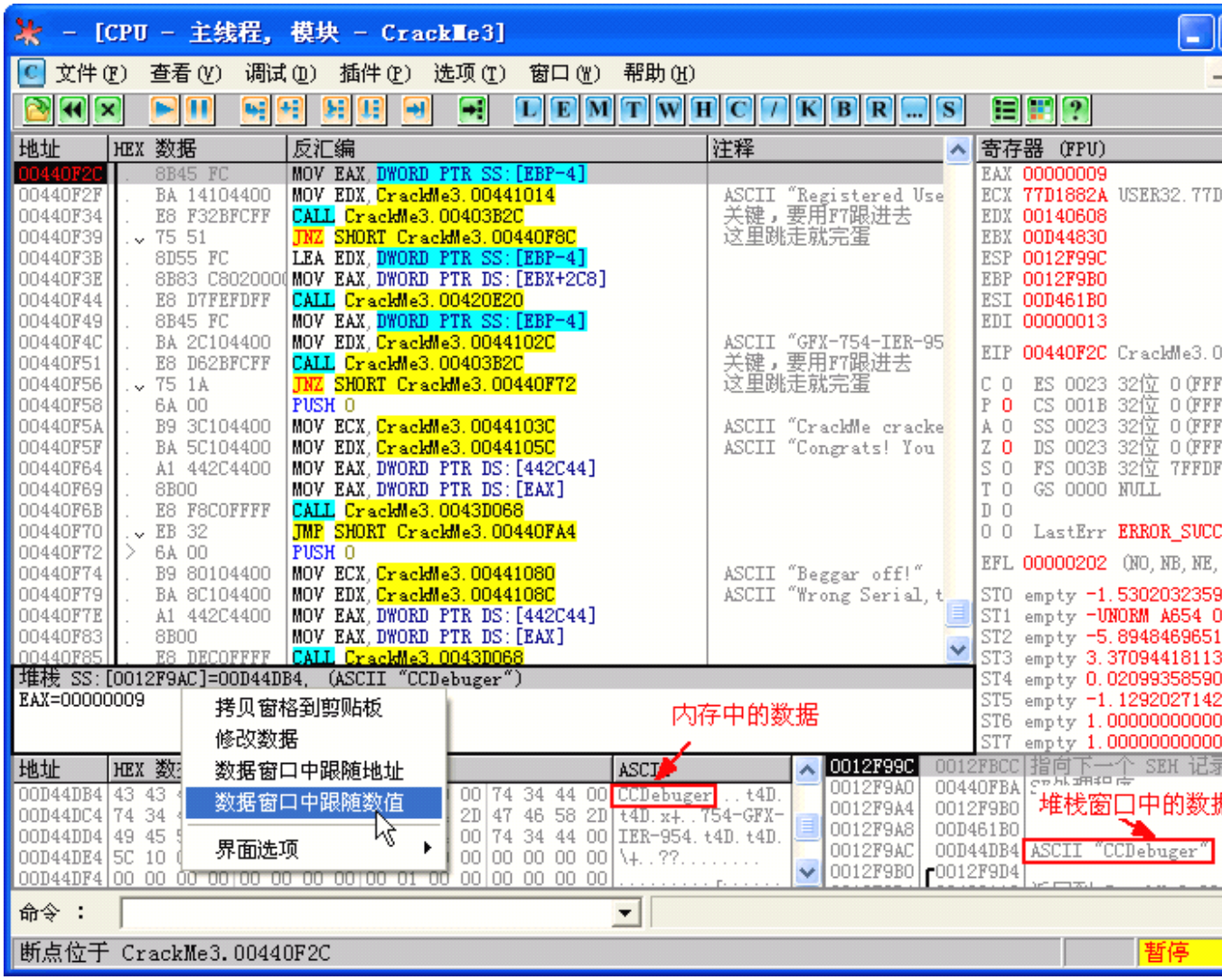


(未完待续/入门系列（二）一字串参考（2）)

我们在上图中地址 00440F2C 处按 F2 键设个断点，现在我们按 F9 键，程序已运行起来了。我在上面那个编辑框中随便输入一下，如 CCDebuger，下面那个编辑框我还保留为原来的“754-GFX-IER-954”，我们点一下那个“Register now !”按钮，呵，OllyDBG 跳了出来，暂停在我们下的断点处。我们看一下信息窗口，你应该发现了你刚才输入的内容了吧？我这里显示是这样：

堆栈 SS:[0012F9AC]=00D44DB4, (ASCII "CCDebuger")
EAX=00000009

上面的内存地址 00D44DB4 中就是我们刚才输入的内容，我这里是 CCDebuger。你可以在 堆栈 SS:[0012F9AC]=00D44DB4, (ASCII "CCDebuger") 这条内容上左击选择一下，再点右键，在弹出菜单中选择“数据窗口中跟随数值”，你就会在下面的数据窗口中看到你刚才输入的内容。而 EAX=00000009 指的是你输入内容的长度。如我输入的 CCDebuger 是 9 个字符。如下图所示：



现在我们来按 F8 键一步步分析一下：

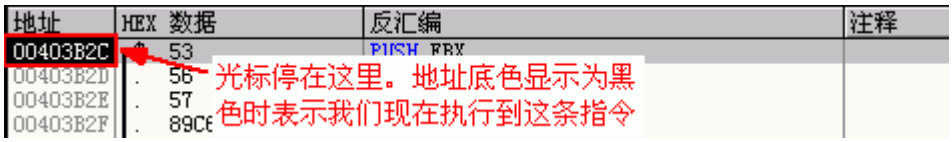
00440F2C |. 8B45 FC MOV EAX,DWORD PTR SS:[EBP-4] ；把我们输入的内容送到 EAX，我这里是“CCDebuger”

00440F2F |. BA 14104400 MOV EDX,CrackMe3.00441014 ；ASCII "Registered User"

00440F34 |. E8 F32BFCFF CALL CrackMe3.00403B2C ；关键，要用 F7 跟进去

00440F39 |. 75 51 JNZ SHORT CrackMe3.00440F8C ；这里跳走就完蛋

当我们按 F8 键走到 00440F34 |. E8 F32BFCFF CALL CrackMe3.00403B2C 这一句时，我们按一下 F7 键，进入这个 CALL，进去后光标停在这一句：



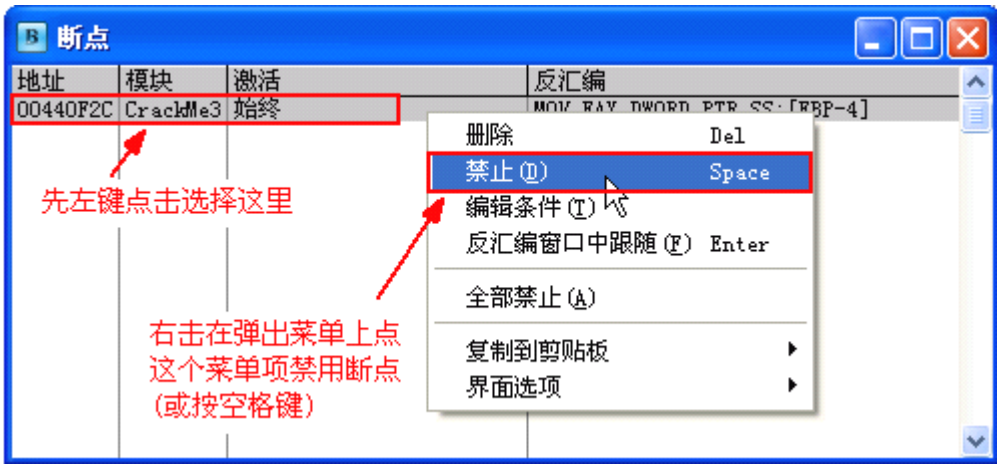
我们所看到的那些 PUSH EBX、 PUSH ESI 等都是调用子程序保存堆栈时用的指令，不用管它，按 F8 键一步步过来，我们只关心关键部分：

00403B2C /$ 53 PUSH EBX
00403B2D |. 56 PUSH ESI
00403B2E |. 57 PUSH EDI
00403B2F |. 89C6 MOV ESI,EAX ；把 EAX 内我们输入的用户名送到 ESI
00403B31 |. 89D7 MOV EDI,EDX ；把 EDX 内的数据“Registered User”送到 EDI
00403B33 |. 39D0 CMP EAX,EDX ；用“Registered User”和我们输入的用户名作比较
00403B35 |. 0F84 8F000000 JE CrackMe3.00403BCA ；相同则跳
00403B3B |. 85F6 TEST ESI,ESI ；看看 ESI 中是否有数据，主要是看看我们有没有输入用户名
00403B3D |. 74 68 JE SHORT CrackMe3.00403BA7 ；用户名为空则跳
00403B3F |. 85FF TEST EDI,EDI
00403B41 |. 74 6B JE SHORT CrackMe3.00403BAE
00403B43 |. 8B46 FC MOV EAX,DWORD PTR DS:[ESI-4] ；用户名长度送 EAX
00403B46 |. 8B57 FC MOV EDX,DWORD PTR DS:[EDI-4] ；“Registered User”字串的长度送 EDX
00403B49 |. 29D0 SUB EAX,EDX ；把用户名长度和“Registered User”字串长度相减
00403B4B |. 77 02 JA SHORT CrackMe3.00403B4F ；用户名长度大于“Registered User”长度则跳
00403B4D |. 01C2 ADD EDX,EAX ；把减后值与“Registered User”长度相加，即用户名长度
00403B4F |> 52 PUSH EDX
00403B50 |. C1EA 02 SHR EDX,2 ；用户名长度值右移 2 位，这里相当于长度除以 4
00403B53 |. 74 26 JE SHORT CrackMe3.00403B7B ；上面的指令及这条指令就是判断用户名长度最少不能低于 4
00403B55 |> 8B0E MOV ECX,DWORD PTR DS:[ESI] ；把我们输入的用户名送到 ECX
00403B57 |. 8B1F MOV EBX,DWORD PTR DS:[EDI] ；把“Registered User”送到 EBX
00403B59 |. 39D9 CMP ECX,EBX ；比较
00403B5B |. 75 58 JNZ SHORT CrackMe3.00403BB5 ；不等则完蛋

根据上面的分析，我们知道用户名必须是“Registered User”。我们按 F9 键让程序运行，出现错误对话框，点确定，重新在第一个编辑框中输入“Registered User”，再次点击那个“Register now !”按钮，被 OllyDBG 拦下。因为地址 00440F34 处的那个 CALL 我们已经分析清楚了，这次就不用再按 F7 键跟进去了，直接按 F8 键通过。我们一路按 F8 键，来到第二个关键代码处：

```
00440F49 |. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]          ; 取输入的注册码
00440F4C |. BA 2C104400       MOV EDX,CrackMe3.0044102C             ; ASCII "GFX-754-IER-954"
00440F51 |. E8 D62BFCFF       CALL CrackMe3.00403B2C                ; 关键，要用 F7 跟进去
00440F56 |. 75 1A            JNZ SHORT CrackMe3.00440F72           ; 这里跳走就完蛋
```

大家注意看一下，地址 00440F51 处的 CALL CrackMe3.00403B2C 和上面我们分析的地址 00440F34 处的 CALL CrackMe3.00403B2C 是不是汇编指令都一样啊？这说明检测用户名和注册码是用的同一个子程序。而这个子程序 CALL 我们在上面已经分析过了。我们执行到现在可以很容易得出结论，这个 CALL 也就是把我们输入的注册码与 00440F4C 地址处指令后的“GFX-754-IER-954”作比较，相等则 OK。好了，我们已经得到足够的信息了。现在我们在菜单 查看->断点 上点击一下，打开断点窗口（也可以通过组合键 ALT+B 或点击工具栏上那个“B”图标打开断点窗口）：



为什么要做这一步，而不是把这个断点删除呢？这里主要是为了保险一点，万一分析错误，我们还要接着分析，要是把断点删除了就要做一些重复工作了。还是先禁用一下，如果经过实际验证证明我们的分析是正确的，再删不迟。现在我们把断点禁用，在 OllyDBG 中按 F9 键让程序运行。输入我们经分析得出的内容：

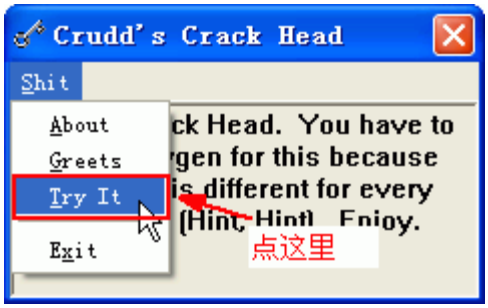
用户名：Registered User

注册码：GFX-754-IER-954

点击“Register now !”按钮，呵呵，终于成功了：

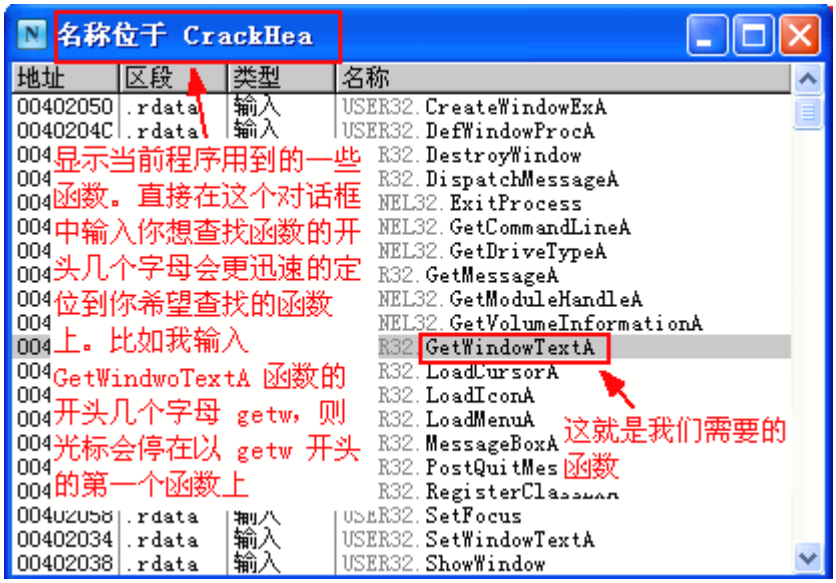
（三）—函数参考

现在进入第三篇，这一篇我们重点讲解怎样使用 OllyDBG 中的函数参考（即名称参考）功能。仍然选择 crackmes.cjb.net 镜像打包中的一个名称为 CrackHead 的 crackme。老规矩，先运行一下这个程序看看：



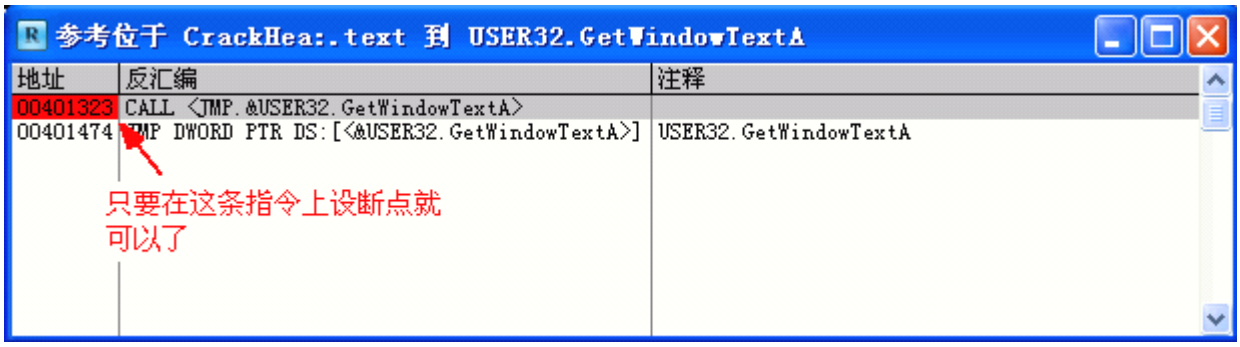
呵，竟然没找到输入注册码的地方！别急，我们点一下程序上的那个菜单“Shit”（真是 Shit 啊，呵呵），在下拉菜单中选“Try It”，会来到如下界面：

我们点一下那个“Check It”按钮试一下，哦，竟然没反应！我再输个“78787878”试试，还是没反应。再试试输入字母或其它字符，输不进去。由此判断注册码应该都是数字，只有输入正确的注册码才有动静。用 PEiD 检测一下，结果为 MASM32 / TASM32，怪不得程序比较小。信息收集的差不多了，现在关掉这个程序，我们用 OllyDBG 载入，按 F9 键直接让它运行起来，依次点击上面图中所说的菜单，使被调试程序显示如上面的第二个图。先不要点那个“Check It”按钮，保留上图的状态。现在我们没有什么字串好参考了，我们就在 API 函数上下断点，来让被调试程序中断在我们希望的地方。我们在 OllyDBG 的反汇编窗口中右击鼠标，在弹出菜单中选择 查找->当前模块中的名称 (标签)，或者我们通过按 CTR+N 组合键也可以达到同样的效果（注意在进行此操作时要在 OllyDBG 中保证是在当前被调试程序的领空，我在第一篇中已经介绍了领空的概念，如我这里调试这个程序时 OllyDBG 的标题栏显示的就是“[CPU - 主线程, 模块 - CrackHea]”，这表明我们当前在被调试程序的领空）。通过上面的操作后会弹出一个对话框，如图：



对于这样的编辑框中输注册码的程序我们要设断点首选的 API 函数就是 GetDlgItemText 及 GetWindowText。每个函数都有两个版本，一个是 ASCII 版，在函数后添加一个 A 表示，如 GetDlgItemTextA，另一个是 UNICODE 版，在函数后添加一个 W 表示。如 GetDlgItemTextW。对于编译为 UNCODE 版的程序可能在 Win98 下不能运行，因为 Win98 并非是完全支持 UNICODE 的系统。而 NT 系统则从底层支持 UNICODE，它可以在操作系统内对字串进行转换，同时支持 ASCII 和 UNICODE 版本函数的调用。一般我们打开的程序看到的调用都是 ASCII 类型的函数，以“A”结尾。又跑题了，呵呵。现在回到我们调试的程序上来，我们现在就是要找一下我们调试的程序有没有调用 GetDlgItemTextA 或 GetWindowTextA 函数。还好，找到一个 GetWindowTextA。在这个函数上右击，在弹出菜单上选择“在每个参考上设置断点”，我们会在 OllyDBG 窗口最下面

的那个状态栏里看到“已设置 2 个断点”。另一种方法就是那个 GetWindowTextA 函数上右击，在弹出菜单上选择“查找输入函数参考”（或者按回车键），将会出现下面的对话框：



看上图，我们可以把两条都设上断点。这个程序只需在第一条指令设断点就可以了。好，我们现在按前面提到的第一条方法，就是“在每个参考上设置断点”，这样上图中的两条指令都会设上断点。断点设好后我们转到我们调试的程序上来，现在我们在被我们调试的程序上点击那个“Check It”按钮，被 OllyDBG 断下：

```
00401323 |. E8 4C010000          CALL <JMP.&USER32.GetWindowTextA>                ;
GetWindowTextA
00401328 |. E8 A5000000          CALL CrackHea.004013D2                           ; 关键，
要按 F7 键跟进去
0040132D |. 3BC6                 CMP EAX,ESI                                      ; 比较
0040132F |. 75 42                JNZ SHORT CrackHea.00401373                      ; 不等则完
蛋
00401331 |. EB 2C                JMP SHORT CrackHea.0040135F
00401333 |. 4E 6F 77 20 7>       ASCII "Now write a keyg"
00401343 |. 65 6E 20 61 6>       ASCII "en and tut and y"
00401353 |. 6F 75 27 72 6>       ASCII "ou're done.",0
0040135F |> 6A 00                PUSH 0                                           ; Style =
MB_OK|MB_APPLMODAL
00401361 |. 68 0F304000          PUSH CrackHea.0040300F                           ; Title =
"Crudd's Crack Head"
00401366 |. 68 33134000          PUSH CrackHea.00401333                           ; Text =
"Now write a keygen and tut and you're done."
0040136B |. FF75 08              PUSH DWORD PTR SS:[EBP+8]                        ;
hOwner
0040136E |. E8 19010000          CALL <JMP.&USER32.MessageBoxA>                   ;
MessageBoxA
```

从上面的代码，我们很容易看出 00401328 地址处的 CALL CrackHea.004013D2 是关键，必须仔细跟踪。而注册成功则会显示一个对话框，标题是“Crudd's Crack Head”，对话框显示的内容是“Now write a keygen and tut and you're done.”现在我按一下 F8，准备步进到 00401328 地址处的那条 CALL CrackHea.004013D2 指令后再按 F7 键跟进去。等等，怎么回事？怎么按一下 F8 键跑到这来了：

```
00401474 $- FF25 2C204000 JMP DWORD PTR DS:[<&USER32.GetWindowText> ; USER32.GetWindowTextA
0040147A $- FF25 30204000 JMP DWORD PTR DS:[<&USER32.LoadCursorA>] ; USER32.LoadCursorA
00401480 $- FF25 1C204000 JMP DWORD PTR DS:[<&USER32.LoadIconA>] ; USER32.LoadIconA
00401486 $- FF25 20204000 JMP DWORD PTR DS:[<&USER32.LoadMenuA>] ; USER32.LoadMenuA
0040148C $- FF25 24204000 JMP DWORD PTR DS:[<&USER32.MessageBoxA>] ; USER32.MessageBoxA
```

原来是跳到另一个断点了。这个断点我们不需要，按一下 F2 键删掉它吧。删掉 00401474 地址处的断点后，我再按 F8 键，呵，完了，跑到 User32.dll 的领空了。看一下 OllyDBG 的标题栏："[CPU - 主线程, 模块 - USER32]，跑到系统领空了，OllyDBG 反汇编窗口中显示代码是这样：

```
77D3213C 6A 0C PUSH 0C
77D3213E 68 A021D377 PUSH USER32.77D321A0
77D32143 E8 7864FEFF CALL USER32.77D185C0
```

怎么办？别急，我们按一下 ALT+F9 组合键，呵，回来了：

```
00401328 |. E8 A5000000 CALL CrackHea.004013D2 ; 关键，要按 F7 键跟进去
0040132D |. 3BC6 CMP EAX,ESI ; 比较
0040132F |. 75 42 JNZ SHORT CrackHea.00401373 ; 不等则完蛋
```

光标停在 00401328 地址处的那条指令上。现在我们按 F7 键跟进：

```
004013D2 /$ 56 PUSH ESI ; ESI 入栈
004013D3 |. 33C0 XOR EAX,EAX ; EAX 清零
004013D5 |. 8D35 C4334000 LEA ESI,DWORD PTR DS:[4033C4] ; 把注册码框中的数值送到 ESI
004013DB |. 33C9 XOR ECX,ECX ; ECX 清零
004013DD |. 33D2 XOR EDX,EDX ; EDX 清零
004013DF |. 8A06 MOV AL,BYTE PTR DS:[ESI] ; 把注册码中的每个字符送到 AL
004013E1 |. 46 INC ESI ; 指针加 1，指向下一个字符
004013E2 |. 3C 2D CMP AL,2D ; 把取得的字符与 16 进制值为 2D 的字符(即"-")比较，这里主要用于判断输入的是不是负数
004013E4 |. 75 08 JNZ SHORT CrackHea.004013EE ; 不等则跳
```

004013E6 |. BA FFFFFFFF MOV EDX,-1 ；如果输入的是负数，则把-1 送到 EDX，即 16 进制 FFFFFFFF
004013EB |. 8A06 MOV AL,BYTE PTR DS:[ESI] ；取“-”号后的第一个字符
004013ED |. 46 INC ESI ；指针加 1，指向再下一个字符
004013EE |> EB 0B JMP SHORT CrackHea.004013FB
004013F0 |> 2C 30 SUB AL,30 ；每位字符减 16 进制的 30，因为这里都是数字，如 1 的 ASCII 码是“31H”，减 30H 后为 1，即我们平时看到的数值
004013F2 |. 8D0C89 LEA ECX,DWORD PTR DS:[ECX+ECX*4] ；把前面运算后保存在 ECX 中的结果乘 5 再送到 ECX
004013F5 |. 8D0C48 LEA ECX,DWORD PTR DS:[EAX+ECX*2] ；每位字符运算后的值与 2 倍上一位字符运算后值相加后送 ECX
004013F8 |. 8A06 MOV AL,BYTE PTR DS:[ESI] ；取下一个字符
004013FA |. 46 INC ESI ；指针加 1，指向再下一个字符
004013FB |> 0AC0 OR AL,AL
004013FD |.^ 75 F1 JNZ SHORT CrackHea.004013F0 ；上面一条和这一条指令主要是用来判断是否已把用户输入的注册码计算完
004013FF |. 8D040A LEA EAX,DWORD PTR DS:[EDX+ECX] ；把 EDX 中的值与经过上面运算后的 ECX 中值相加送到 EAX
00401402 |. 33C2 XOR EAX,EDX ；把 EAX 与 EDX 异或。如果我们输入的是负数，则此处功能就是把 EAX 中的值取反
00401404 |. 5E POP ESI ; ESI 出栈。看到这条和下一条指令，我们要考虑一下这个 ESI 的值是哪里运算得出的呢？
00401405 |. 81F6 53757A79 XOR ESI,797A7553 ；把 ESI 中的值与 797A7553H 异或
0040140B \. C3 RETN

一函数参(2)

这里留下了一个问题：那个 ESI 寄存器中的值是从哪运算出来的？先不管这里，我们接着按 F8 键往下走，来到 0040140B 地址处的那条 RETN 指令（这里可以通过在调试选项的“命令”标签中勾选“使用 RET 代替 RETN”来更改返回指令的显示方式），再按一下 F8，我们就走出 00401328 地址处的那个 CALL 了。现在我们回到了这里：

0040132D |. 3BC6 CMP EAX,ESI ；比较
0040132F |. 75 42 JNZ SHORT CrackHea.00401373 ；不等则完蛋

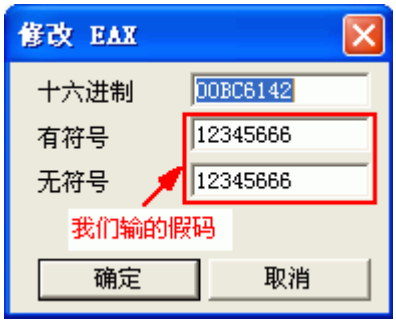
光标停在了 0040132D 地址处的那条指令上。根据前面的分析，我们知道 EAX 中存放的

是我们输入的注册码经过计算后的值。我们来看一下信息窗口：

ESI=E6B5F2F9
EAX=FF439EBE

左键选择信息窗口中的 ESI=E6B5F2F9，再按右键，在弹出菜单上选“修改寄存器”，我们会看到这样一个窗口：

可能你的显示跟我不一样，因为这个 crackme 中已经说了每个机器的序列号不一样。关掉上面的窗口，再对信息窗口中的 EAX=FF439EBE 做同样操作：



由上图我们知道了原来前面分析的对我们输入的注册码进行处理后的结果就是把字符格式转为数字格式。我们原来输入的是字串“12345666”，现在转换为了数字 12345666。这下就很清楚了，随便在上面那个修改 ESI 图中显示的有符号或无符号编辑框中复制一个，粘贴到我们调试的程序中的编辑框中试一下：

呵呵，成功了。且慢高兴，这个 crackme 是要求写出注册机的。我们先不要求写注册机，但注册的算法我们要搞清楚。还记得我在前面说到的那个 ESI 寄存器值的问题吗？现在看

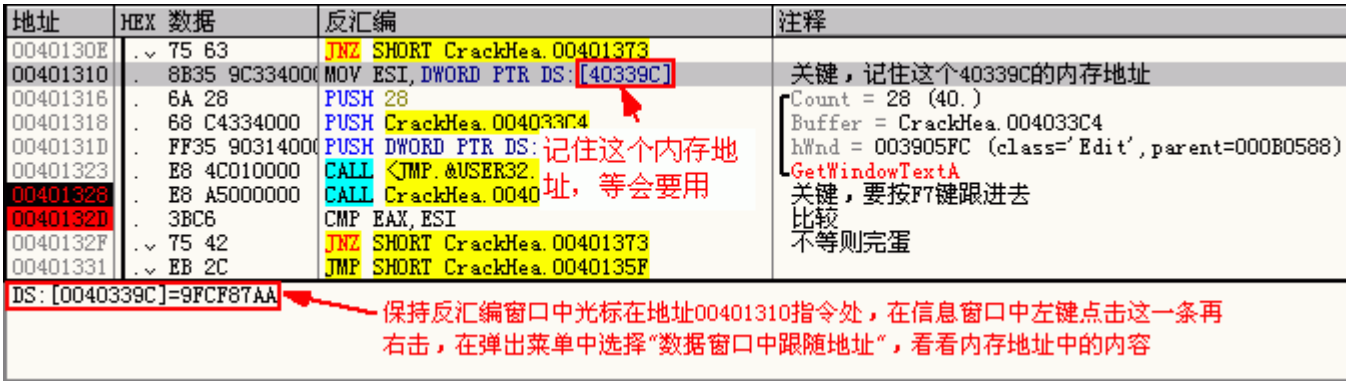
看我们上面的分析，其实对做注册机来说是没有多少帮助的。要搞清注册算法，必须知道上面那个 ESI 寄存器值是如何产生的，这弄清楚后才能真正清楚这个 crackme 算法。今天就先说到这里，关于如何追出 ESI 寄存器的值我就留到下一篇 OllyDBG 入门系列（四）－内存断点 中再讲吧。

（四）－内存断点

还记得上一篇《OllyDBG 入门系列（三）－函数参考》中的内容吗？在那篇文章中我们分析后发现一个 ESI 寄存器值不知是从什么地方产生的，要弄清这个问题必须要找到生成这个 ESI 值的计算部分。今天我们的任务就是使用 OllyDBG 的内存断点功能找到这个地方，搞清楚这个值是如何算出来的。这次分析的目标程序还是上一篇的那个 crackme，附件我就不再上传了，用上篇中的附件就可以了。下面我们开始：
还记得我们上篇中所说的关键代码的地方吗？温习一下：

```
00401323 |. E8 4C010000          CALL <JMP.&USER32.GetWindowTextA>          ; GetWindowTextA
00401328 |. E8 A5000000          CALL CrackHea.004013D2                     ; 关键，要按 F7 键跟进去
0040132D |. 3BC6                 CMP EAX,ESI                                ; 比较
0040132F |. 75 42                JNZ SHORT CrackHea.00401373                ; 不等则完蛋
```

我们重新用 OllyDBG 载入目标程序，F9 运行来到上面代码所在的地方（你上次设的断点应该没删吧？），我们向上看看能不能找到那个 ESI 寄存器中最近是在哪里赋的值。哈哈，原来就在附近啊：



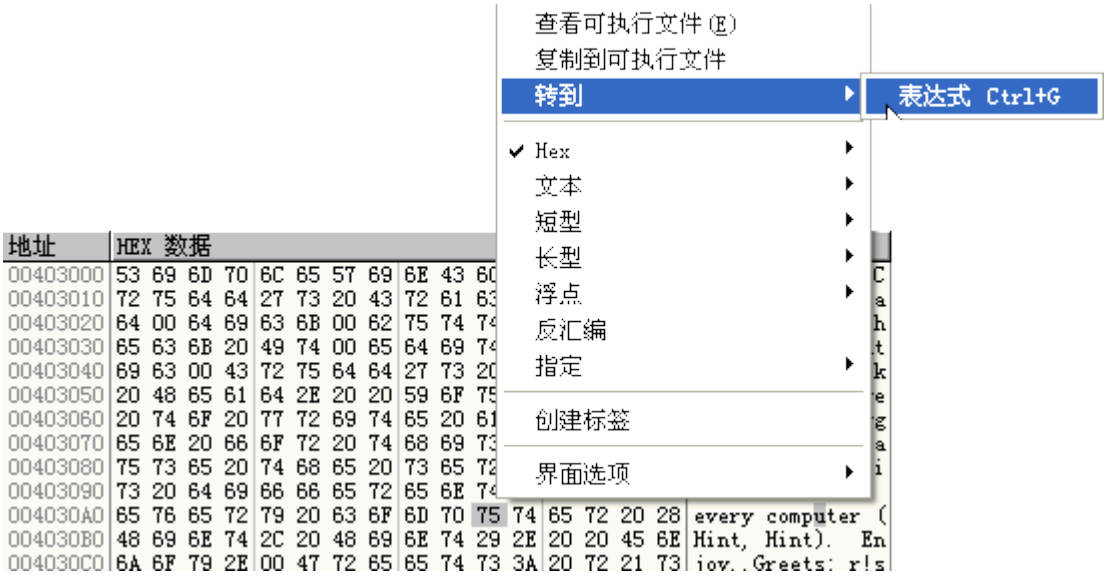
我们现在知道 ESI 寄存器的值是从内存地址 40339C 中送过来的，那内存地址 40339C 中的数据是什么时候产生的呢？大家注意，我这里信息窗口中显示的是 DS:[0040339C]=9FCF87AA，你那可能是 DS:[0040339C]=XXXXXXXX，这里的 XXXXXXXX 表示的是其它的值，就是说与我这里显示的 9FCF87AA 不一样。我们按上图的操作在数据窗口中看一下：

从上图我们可以看出内存地址 40339C 处的值已经有了，说明早就算过了。现在怎么办呢？我们考虑一下，看情况程序是把这个值算出来以后写在这个内存地址，那我们要是能让 OllyDBG 在程序开始往这个内存地址写东西的时候中断下来，不就有可能知道目标程序是怎么算出这个值的吗？说干就干，我们在 OllyDBG 的菜单上点 调试->重新开始，或者按 CTR+F2 组合键（还可以点击工具栏上的那个有两个实心左箭头的图标）来重新载入程序。这时会跳出一个"进程仍处于激活状态"的对话框（我们可以在在调试选项的安全标签下把"终止活动进程时警告"这条前面的勾去掉，这样下次就不会出现这个对话框了），问我们是否要终止进程。这里我们选"是"，程序被重新载入，我们停在下面这一句上：

00401000 >/$ 6A 00 PUSH 0 ; pModule = NULL

现在我们就要来设内存断点了。在 OllyDBG 中一般我们用到的内存断点有内存访问和内存写入断点。内存访问断点就是指程序访问内存中我们指定的内存地址时中断，内存写入断点就是指程序往我们指定的内存地址中写东西时中断。更多关于断点的知识大家可以参考 论坛精华 7->基础知识->断点技巧->断点原理 这篇 Lenus 兄弟写的《如何对抗硬件断点之一--- 调试寄存器》文章，也可以看这个帖：http://bbs.pediy.com/showthread.php?threadid=10829。根据当前我们调试的具体程序的情况，我们选用内存写入断点。还记得前面我叫大家记住的那个 40339C 内存地址吗？现在

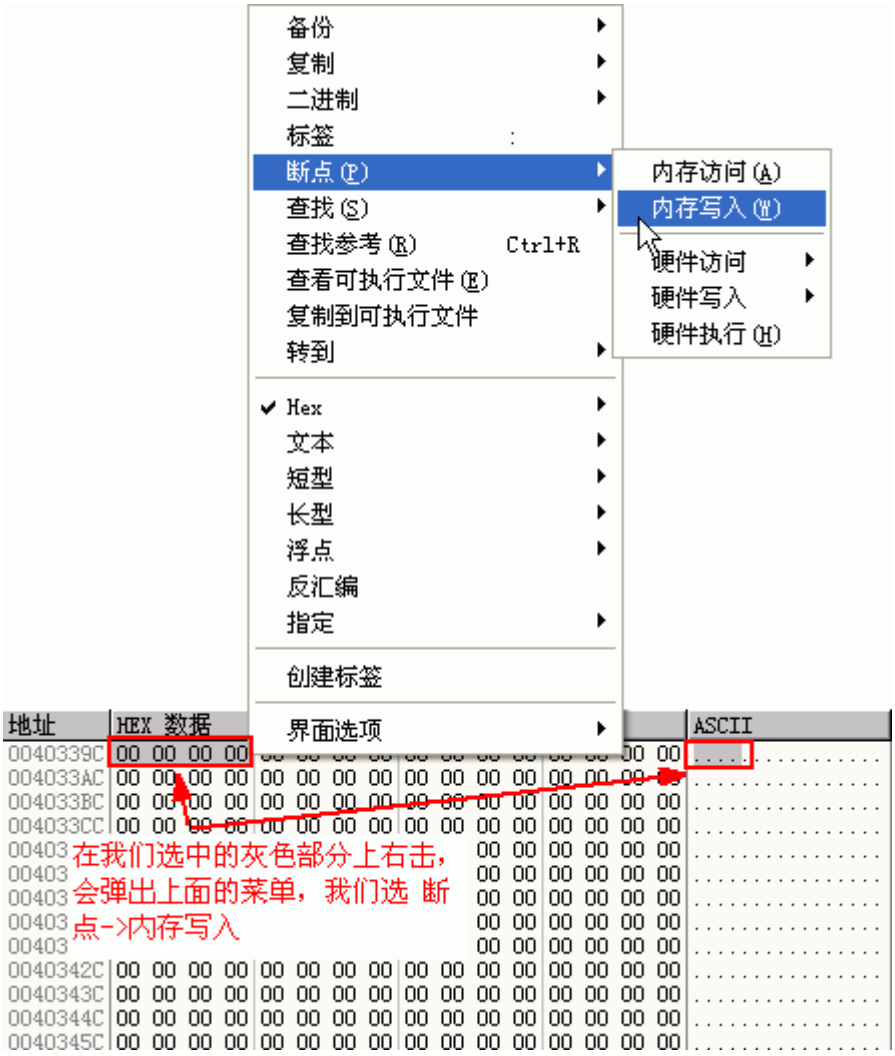
我们要用上了。我们先在 OllyDBG 的数据窗口中左键点击一下，再右击，会弹出一个如下图所示的菜单。我们选择其中的转到->表达式（也可以左键点击数据窗口后按 CTR+G 组合键）。如下图：



现在将会出现这样一个对话框：

我们在上面那个编辑框中输入我们想查看内容的内存地址 40339C，然后点确定按钮，数据窗口中显示如下：

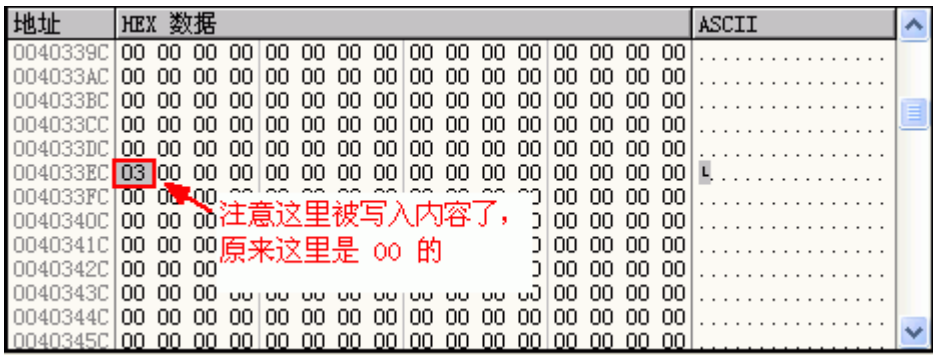
我们可以看到，40339C 地址开始处的这段内存里面还没有内容。我们现在在 40339C 地址处后面的 HEX 数据或 ASCII 栏中按住左键往后拖放，选择一段。内存断点的特性就是不管你选几个字节，OllyDBG 都会分配 4096 字节的内存区。这里我就选从 40339C 地址处开始的四个字节，主要是为了让大家提前了解一下硬件断点的设法，因为硬件断点最多只能选 4 个字节。选中部分会显示为灰色。选好以后松开鼠标左键，在我们选中的灰色部分上右击：



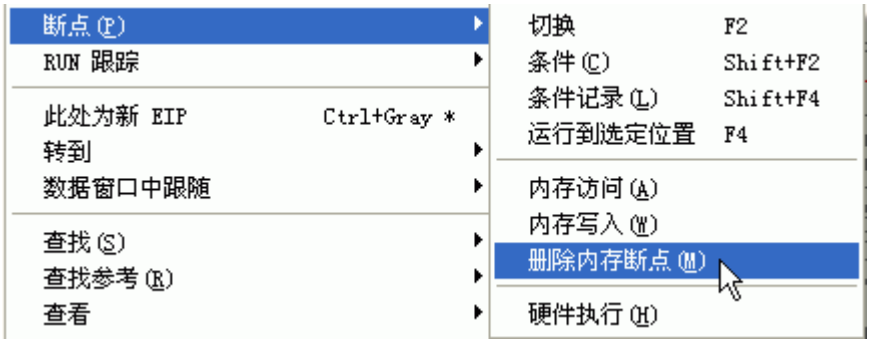
经过上面的操作，我们的内存断点就设好了（这里还有个要注意的地方：内存断点只在当前调试的进程中有效，就是说你如果重新载入程序的话内存断点就自动删除了。且内存断点每一时刻只能有一个。就是说你不能像按 F2 键那样同时设置多个断点）。现在按 F9 键让程序运行，呵，OllyDBG 中断了！

```
7C932F39 8808                MOV BYTE PTR DS:[EAX],CL                ; 这就是
我们第一次断下来的地方
7C932F3B 40                  INC EAX
7C932F3C 4F                  DEC EDI
7C932F3D 4E                  DEC ESI
7C932F3E ^ 75 CB             JNZ SHORT ntdll.7C932F0B
7C932F40 8B4D 10             MOV ECX,DWORD PTR SS:[EBP+10]
```

上面就是我们中断后反汇编窗口中的代码。如果你是其它系统，如 Win98 的话，可能会有所不同。没关系，这里不是关键。我们看一下领空，原来是在 ntdll.dll 内。系统领空，我们现在要考虑返回到程序领空。返回前我们看一下数据窗口：



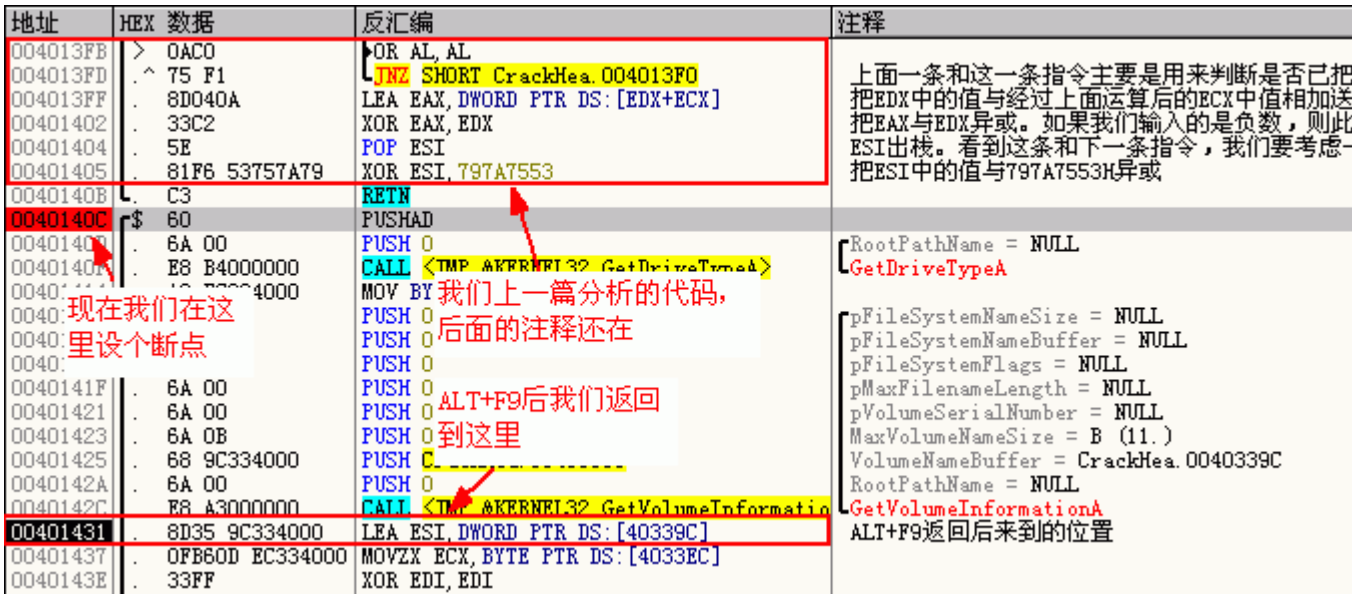
现在我们转到反汇编窗口，右击鼠标，在弹出菜单上选择断点->删除内存断点，这样内存断点就被删除了。



现在我们来按一下 ALT+F9 组合键，我们来到下面的代码：

00401431 |. 8D35 9C334000 LEA ESI,DWORD PTR DS:[40339C] ;
ALT+F9 返回后来到的位置
00401437 |. 0FB60D EC334000 MOVZX ECX,BYTE PTR DS:[4033EC]
0040143E |. 33FF XOR EDI,EDI

我们把反汇编窗口往上翻翻，呵，原来就在我们上一篇分析的代码下面啊？



现在我们在 0040140C 地址处那条指令上按 F2 设置一个断点，现在我们按 CTR+F2 组合键重新载入程序，载入后按 F9 键运行，我们将会中断在我们刚才在 0040140C 地址下的那个断点处：

```
0040140C /$ 60                PUSHAD
0040140D |. 6A 00              PUSH 0                                  ;
/RootPathName = NULL
0040140F |. E8 B4000000        CALL <JMP.&KERNEL32.GetDriveTypeA>      ;
\GetDriveTypeA
00401414 |. A2 EC334000        MOV BYTE PTR DS:[4033EC],AL             ; 磁盘
类型参数送内存地址 4033EC
00401419 |. 6A 00              PUSH 0                                  ;
/pFileSystemNameSize = NULL
0040141B |. 6A 00              PUSH 0                                  ;
|pFileSystemNameBuffer = NULL
0040141D |. 6A 00              PUSH 0                                  ;
|pFileSystemFlags = NULL
0040141F |. 6A 00              PUSH 0                                  ;
|pMaxFilenameLength = NULL
00401421 |. 6A 00              PUSH 0                                  ;
|pVolumeSerialNumber = NULL
00401423 |. 6A 0B              PUSH 0B                                 ;
|MaxVolumeNameSize = B (11.)
00401425 |. 68 9C334000        PUSH CrackHea.0040339C                  ;
|VolumeNameBuffer = CrackHea.0040339C
0040142A |. 6A 00              PUSH 0                                  ; |RootPathName
= NULL
0040142C |. E8 A3000000        CALL <JMP.&KERNEL32.GetVolumeInformationA>  ;
\GetVolumeInformationA
00401431 |. 8D35 9C334000      LEA ESI,DWORD PTR DS:[40339C]           ; 把
crackme 程序所在分区的卷标名称送到 ESI
00401437 |. 0FB60D EC334000    MOVZX ECX,BYTE PTR DS:[4033EC]          ;
磁盘类型参数送 ECX
0040143E |. 33FF              XOR EDI,EDI                             ; 把 EDI 清零
00401440 |> 8BC1              MOV EAX,ECX                             ; 磁盘类型参
数送 EAX
00401442 |. 8B1E              MOV EBX,DWORD PTR DS:[ESI]              ; 把卷标
名作为数值送到 EBX
00401444 |. F7E3              MUL EBX                                 ; 循环递减取磁盘
类型参数值与卷标名值相乘
00401446 |. 03F8              ADD EDI,EAX                             ; 每次计算结果
再加上上次计算结果保存在 EDI 中
00401448 |. 49                DEC ECX                                 ; 把磁盘类型参数
作为循环次数，依次递减
00401449 |. 83F9 00           CMP ECX,0                               ; 判断是否计算
```

完
0040144C |.^ 75 F2 JNZ SHORT CrackHea.00401440 ；没完继续
0040144E |. 893D 9C334000 MOV DWORD PTR DS:[40339C],EDI ；把计算后值送到内存地址 40339C，这就是我们后来在 ESI 中看到的值
00401454 |. 61 POPAD
00401455 \. C3 RETN

通过上面的分析，我们知道基本算法是这样的：先用 GetDriveTypeA 函数获取磁盘类型参数，再用 GetVolumeInformationA 函数获取这个 crackme 程序所在分区的卷标。如我把这个 Crackme 程序放在 F:\OD 教程\crackhead\ 目录下，而我 F 盘设置的卷标是 GAME，则这里获取的就是 GAME，ASCII 码为“47414D45”。但我们发现一个问题：假如原来我们在数据窗口中看到的地址 40339C 处的 16 进制代码是“47414D45”，即“GAME”，但经过地址 00401442 处的那条 MOV EBX,DWORD PTR DS:[ESI] 指令后，我们却发现 EBX 中的值是“454D4147”，正好把我们上面那个“47414D45”反过来了。为什么会这样呢？如果大家对 x86 系列 CPU 的存储方式了解的话，这里就容易理解了。我们知道“GAME”有四个字节，即 ASCII 码为“47414D45”。我们看一下数据窗口中的情况：

0040339C 47 41 4D 45 00 00 00 00 00 00 00 00 00 00 00 00 GAME.....

大家可以看出来内存地址 40339CH 到 40339FH 分别按顺序存放的是 47 41 4D 45。如下图：

系统存储的原则为“高高低低”，即低字节存放在地址较低的字节单元中，高字节存放在地址较高的字节单元中。比如一个字由两个字节组成，像这样：12 34 ，这里的高字节就是 12 ，低字节就是 34。上面的那条指令 MOV EBX,DWORD PTR DS:[ESI] 等同于 MOV EBX,DWORD PTR DS:[40339C]。注意这里是 DWORD，即“双字”，由 4 个连续的字节构成。而取地址为 40339C 的双字单元中的内容时，我们应该得到的是“454D4147”，即由高字节到低字节顺序的值。因此经过 MOV EBX,DWORD PTR DS:[ESI] 这条指令，就是把从地址 40339C 开始处的值送到 EBX，所以我们得到了“454D4147”。好了，这里弄清楚了，我们再接着谈这个程序的算法。前面我们已经说了取磁盘类型参数做循环次数，再取卷标值 ASCII 码的逆序作为数值，有了这两个值就开始计算了。现在我们把磁盘类型值作为 n，卷标值 ASCII 码的逆序数值作为 a，最后得出的结果作为 b，有这样的计算过程：

第一次：b = a * n
第二次：b = a * (n - 1) + b
第三次：b = a * (n - 2) + b

…
第 n 次：b = a * 1 + b
可得出公式为 b = a * [n + (n - 1) + (n - 2) + … + 1] = a * [n * (n + 1) / 2]
还记得上一篇我们的分析吗？看这一句：

00401405 |. 81F6 53757A79 XOR ESI,797A7553 ；把 ESI 中的值与 797A7553H 异或

这里算出来的 b 最后还要和 797A7553H 异或一下才是真正的注册码。只要你对编程有所了解，这个注册机就很好写了。如果用汇编来写这个注册机的话就更简单了，很多内容可以直接照抄。
到此已经差不多了，最后还有几个东西也说一下吧：
1、上面用到了两个 API 函数，一个是 GetDriveTypeA，还有一个是 GetVolumeInformationA，关于这两个函数的具体用法我就不多说了，大家可以查一下 MSDN。这里只要大家注意函数参数传递的次序，即调用约定。先看一下这里：

```
00401419 |. 6A 00                  PUSH 0                                  ;
/pFileSystemNameSize = NULL
0040141B |. 6A 00                  PUSH 0                                  ;
|pFileSystemNameBuffer = NULL
0040141D |. 6A 00                  PUSH 0                                  ;
|pFileSystemFlags = NULL
0040141F |. 6A 00                  PUSH 0                                  ;
|pMaxFilenameLength = NULL
00401421 |. 6A 00                  PUSH 0                                  ;
|pVolumeSerialNumber = NULL
00401423 |. 6A 0B                  PUSH 0B                                 ;
|MaxVolumeNameSize = B (11.)
00401425 |. 68 9C334000            PUSH CrackHea.0040339C                  ;
|VolumeNameBuffer = CrackHea.0040339C
0040142A |. 6A 00                  PUSH 0                                  ; |RootPathName
= NULL
0040142C |. E8 A3000000            CALL <JMP.&KERNEL32.GetVolumeInformationA>   ;
\GetVolumeInformationA
```

把上面代码后的 OllyDBG 自动添加的注释与 MSDN 中的函数原型比较一下：

```
BOOL GetVolumeInformation(
LPCTSTR lpRootPathName,             // address of root directory of the file system
LPTSTR lpVolumeNameBuffer,          // address of name of the volume
DWORD nVolumeNameSize,              // length of lpVolumeNameBuffer
LPDWORD lpVolumeSerialNumber,       // address of volume serial number
LPDWORD lpMaximumComponentLength,   // address of system's maximum filename
length
LPDWORD lpFileSystemFlags,          // address of file system flags
```

LPTSTR lpFileSystemNameBuffer, // address of name of file system
DWORD nFileSystemNameSize // length of lpFileSystemNameBuffer
);

大家应该看出来点什么了吧？函数调用是先把最后一个参数压栈，参数压栈顺序是从后往前。这就是一般比较常见的 stdcall 调用约定。

2、我在前面的 00401414 地址处的那条 MOV BYTE PTR DS:[4033EC],AL 指令后加的注释是“磁盘类型参数送内存地址 4033EC”。为什么这样写？大家把前一句和这一句合起来看一下：

0040140F |. E8 B4000000 CALL <JMP.&KERNEL32.GetDriveTypeA> ; \GetDriveTypeA
00401414 |. A2 EC334000 MOV BYTE PTR DS:[4033EC],AL ; 磁盘类型参数送内存地址 4033EC

地址 0040140F 处的那条指令是调用 GetDriveTypeA 函数，一般函数调用后的返回值都保存在 EAX 中，所以地址 00401414 处的那一句 MOV BYTE PTR DS:[4033EC],AL 就是传递返回值。查一下 MSDN 可以知道 GetDriveTypeA 函数的返回值有这几个：

Value	Meaning	返回在 EAX 中的值
DRIVE_UNKNOWN	The drive type cannot be determined.	0
DRIVE_NO_ROOT_DIR	The root directory does not exist.	1
DRIVE_REMOVABLE	The disk can be removed from the drive.	2
DRIVE_FIXED	The disk cannot be removed from the drive.	3
DRIVE_REMOTE	The drive is a remote (network) drive.	4
DRIVE_CDROM	The drive is a CD-ROM drive.	5
DRIVE_RAMDISK	The drive is a RAM disk.	6

上面那个“返回在 EAX 中的值”是我加的，我这里返回的是 3，即磁盘不可从驱动器上删除。

3、通过分析这个程序的算法，我们发现这个注册算法是有漏洞的。如果我的分区没有卷标的话，则卷标值为 0，最后的注册码就是 797A7553H，即十进制 2038068563。而如果你的卷标和我一样，且磁盘类型一样的话，注册码也会一样，并不能真正做到一机一码。