

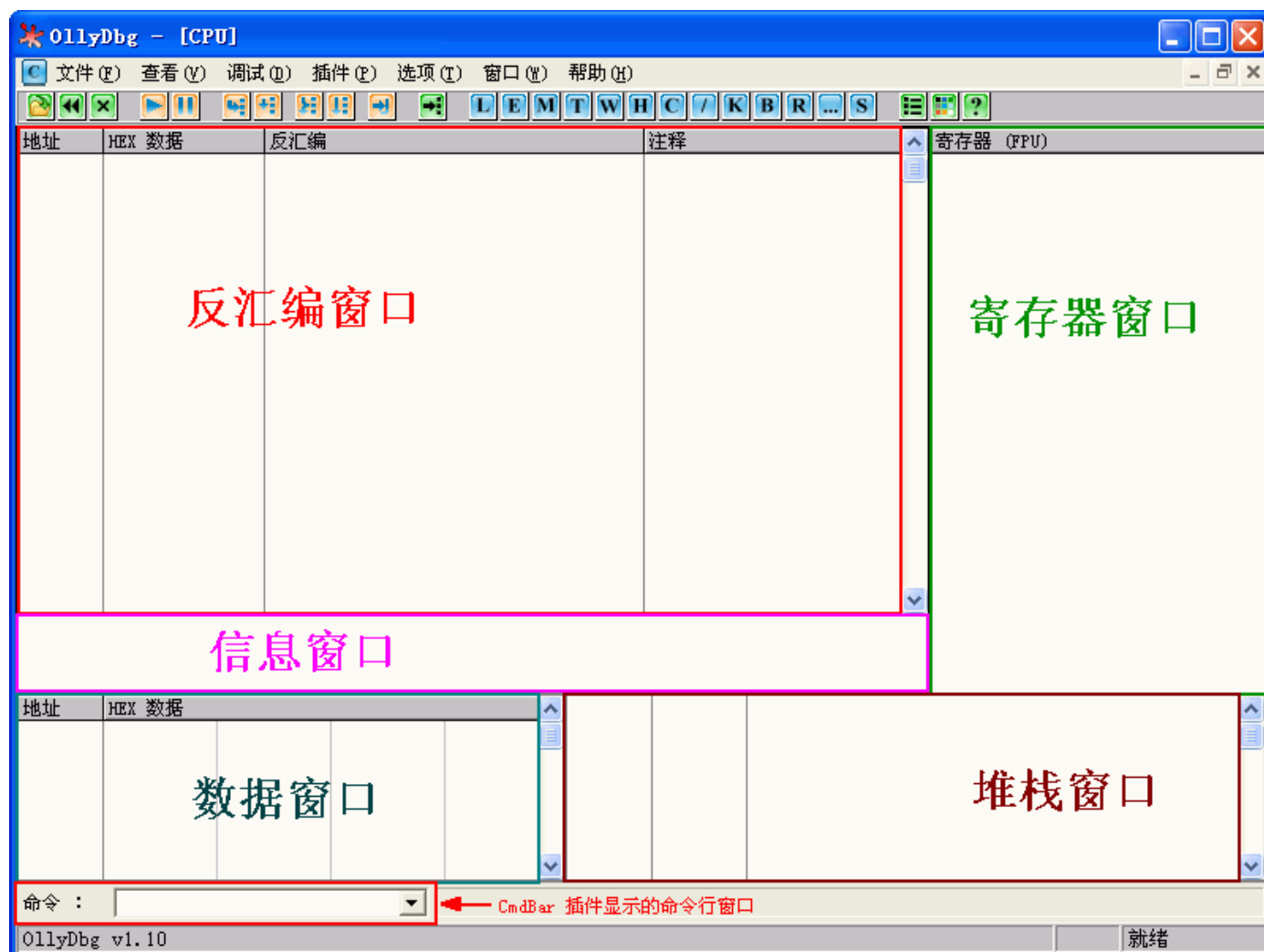
OlllyDBG 完美教程(超强入门级)

关键词：**OD**、**OlllyDBG**、破解入门、调试专用工具、反汇编

一、OlllyDBG 的安装与配置

OlllyDBG 1.10 版的发布版本是个 ZIP 压缩包，只要解压到一个目录下，运行 OlllyDBG.exe 就可以了。

汉化版的发布版本是个 RAR 压缩包，同样只需解压到一个目录下运行 OlllyDBG.exe 即可：



OlllyDBG 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。

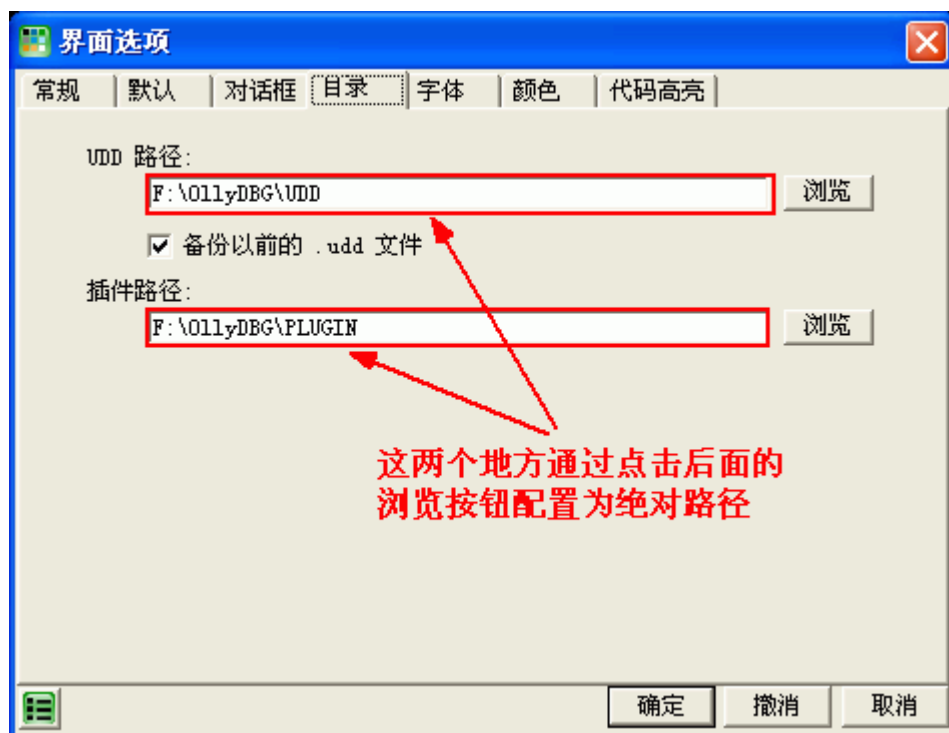
信息窗口：显示反汇编窗口中选中的第一个命令的参数及一些跳转目标地址、字串等。

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。

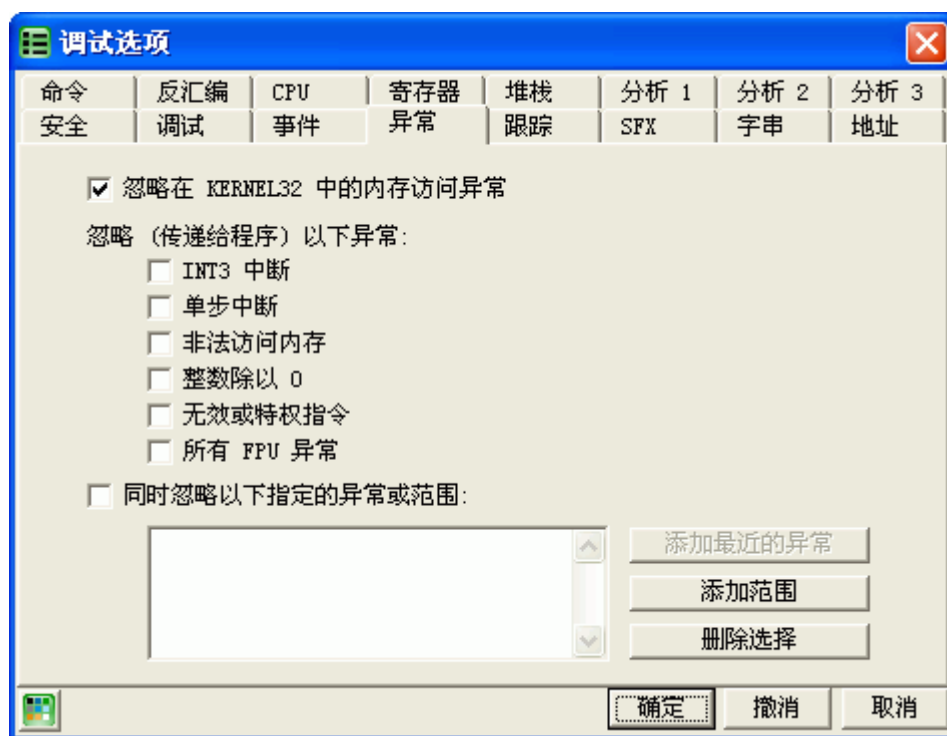
启动后我们要把插件及 UDD 的目录配置为绝对路径，点击菜单上的 选项->界面，将会出来一个界面选项的对话框，我们点击其中的目录标签：



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。

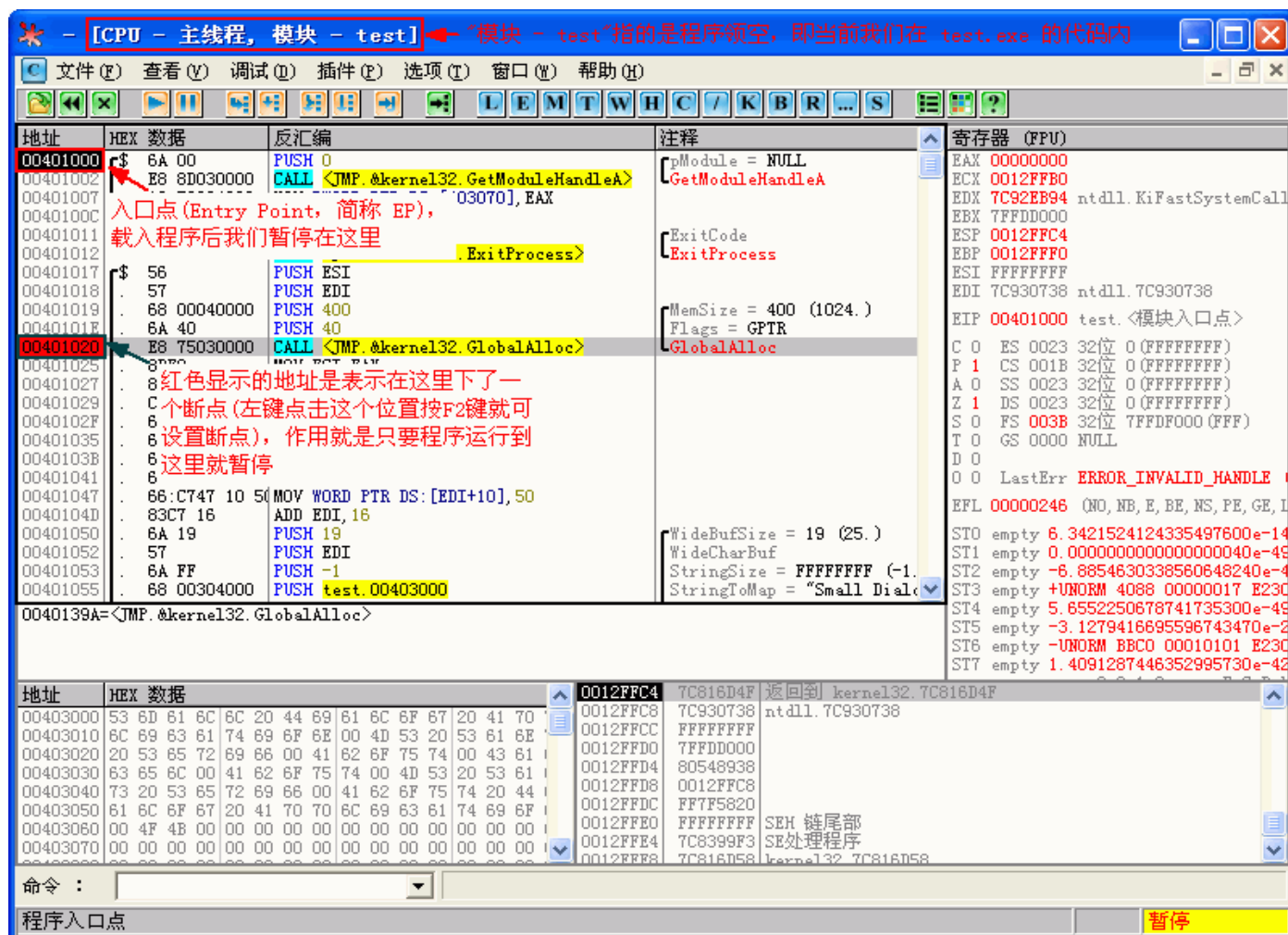
除了直接启动 OllyDBG 来调试外，我们还可以把 OllyDBG 添加到资源管理器右键菜单，这样我们就可以直接在 .exe 及 .dll 文件上点右键选择“用 Ollydbg 打开”菜单来进行调试。要把 OllyDBG 添加到资源管理器右键菜单，只需点菜单 选项->添加到浏览器，将会出现一个对话框，先点击“添加 Ollydbg 到系统资源管理器菜单”，再点击“完成”按钮即可。要从右键菜单中删除也很简单，还是这个对话框，点击“从系统资源管理器菜单删除 Ollydbg”，再点击“完成”就行了。

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。

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。原版加了一个 **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 **OillyDBG** 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 **OillyDBG** 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：



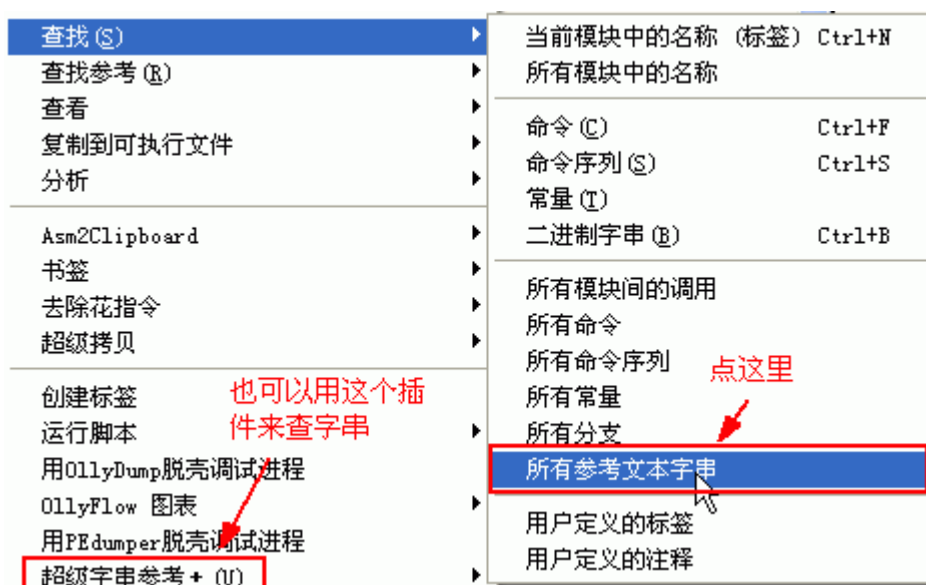
这个 crackme 已经把用户名和注册码都输好了，省得我们动手^_^。我们在那个“Register now !”按钮上点击一下，将会跳出一个对话框：



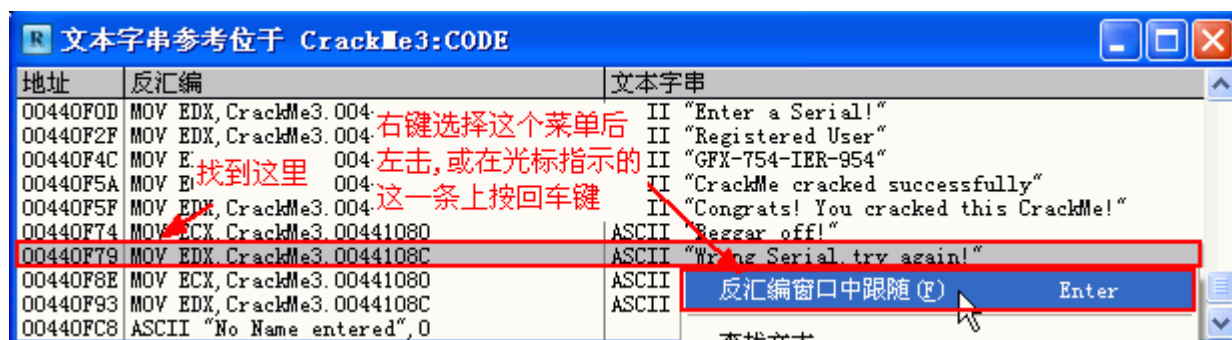
好了，今天我们就从这个错误对话框中显示的“Wrong Serial, try again!”来入手。启动 OllyDBG，选择菜单 文件->打开 载入 CrackMe3.exe 文件，我们会停在这里：

地址	HEX 数据	反汇编	注释
00441270	\$ 55	PUSH EBP	载入后停在这里
00441271	. 8BEC	MOV EBP, ESP	
00441273	. 83C4 F4	ADD ESP, -0C	
00441276	. B8 60114400	MOV EAX, CrackMe3.00441160	
0044127B	. E8 E848FCFF	CALL CrackMe3.00405B68	
00441280	. A1 442C4400	MOV EAX, DWORD PTR DS:[442C44]	
00441285	. 8B00	MOV EAX, DWORD PTR DS:[EAX]	
00441287	. E8 ECBBFFFF	CALL CrackMe3.0043CE78	
0044128C	. A1 442C4400	MOV EAX, DWORD PTR DS:[442C44]	
00441291	. 8B00	MOV EAX, DWORD PTR DS:[EAX]	
00441293	. BA D0124400	MOV EDX, CrackMe3.004412D0	
00441298	. B8 1782FFFF	CALL CrackMe3.0043CAB4	
			ASCII "Crackers For Freedom CrackMe v3.0"

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



当然如果用上面那个 超级字符串参考+ 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：



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

地址	HEX 数据	反汇编	注释
00440F79	. BA 8C104400	MOV EDX, CrackMe3.0044108C	ASCII "Wrong Serial, try again!"
00440F7E	. A1 442C4400	备份	
00440F83	. 8B00	复制	
00440F85	. E8 DEC0FFFF	二进制	
00440F8A	. EB 18	汇编 (A)	
00440F8C	> 6A 00	Space	ASCII "Beggar off!"
00440F8E	. B9 80104400	标签	ASCII "Wrong Serial, try again!"
00440F93	. BA 8C104400	:	
00440F98	. A1 442C4400	注释	
00440F9D	. 8B00	:	
00440F9F	. E8 C4C0FFFF	断点 (E)	
00440FA4	> 33C0	HIT 跟踪	
00440FA6	. 5A	RUN 跟踪	
00440FA7	. 59		
00440FA8	. 59		
00440FA9	. 64:8910	此处为新 EIP	Ctrl+Gray *
00440FAC	. 68 C10F4400	转到	
00440FB1	> 8D45 FC	数据窗口中跟随	
00440FB4	. E8 E727FCFF		
00440FB9	. C3		
00440FBA	. ^ E9 A122FCFF	查找 (S)	
00440FBF	. ^ EB F0	查找参考 (R)	选定地址 (C) Ctrl+R
00440FC1	. 5B	查看	立即数 (I)
00440FC2	. 59		
00440FC3	. 5D		

发现了两个

这里两个我们都能直接看到。如果想找找看还有没有别的参考的话，我们可以在光标所在行点右键，选择菜单查找参考->立即数

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：

R 参考位于 CrackMe3:CODE 到常量 44108C		
地址	反汇编	注释
00440F79	MOV EDX, CrackMe3.0044108C	(初始 CPU 选择)
00440F93	MOV EDX, CrackMe3.0044108C	ASCII "Wrong Serial, try again!"

这里对应反汇编窗口中光标所在的位置，就是我们刚才看到的第一个参考

这里对应的是其他的参考，在这行上双击可以跳到反汇编窗口中的对应位置

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-IE R-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”这条字符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符串处的：

地址	HEX 数据	反汇编	注释
00440F2C	8B45 EC	MOV EAX, DWORD PTR SS:[EBP-4]	
00440F2F	BA 1E300441014		ASCII "Registered User"
00440F34	E8 F34DF0FF	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 F0	LEA EDX, DWORD PTR SS:[EBP-4]	
00440F3E	8B83 C8020000	MOV EAX, DWORD PTR DS:[EBX+2C8]	
00440F44	E8 D71A		
00440F49	8B45 10		
00440F4C	BA 2C2C442C		ASCII "GFX-754-IER-954"
00440F51	E8 D61A		关键，要用F7跟进去
00440F56	75 1A		这里跳走就完蛋
00440F58	6A 00		
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 F81A		
00440F70	EB 32		
00440F72	6A 00		
00440F74	B9 80		ASCII "Beggar off!"
00440F79	BA 8C		ASCII "Wrong Serial, try again!"
00440F7E	A1 442C4400		
00440F83	8B00		
00440F85	E8 DC0FFFFF	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	
00440FA4	33C0	XOR EAX, EAX	

跳转来自 00440F39

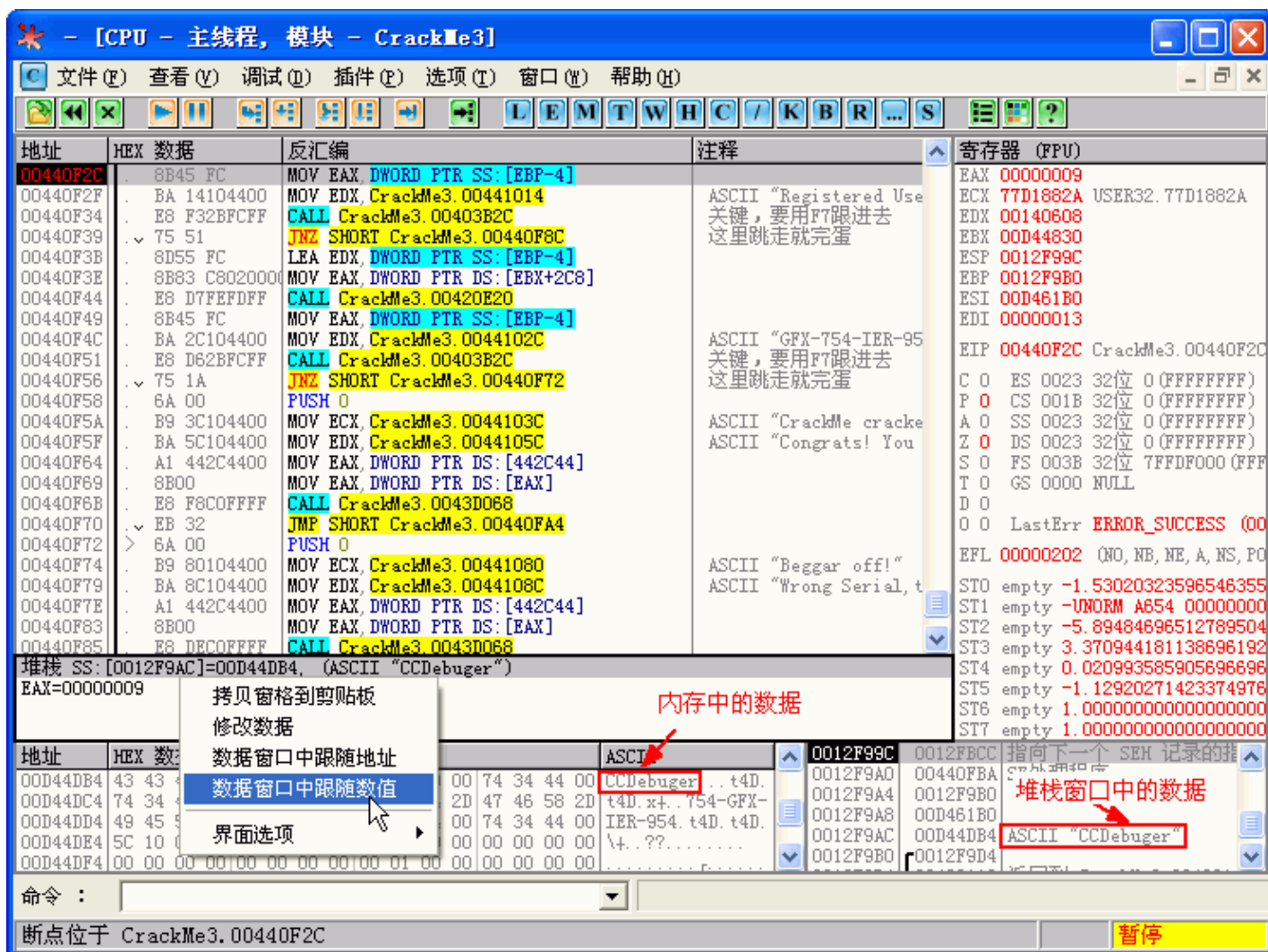
信息窗口中也给我们显示了是从何处跳转到当前光标所在位置的

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 CCDebugger，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 "CCDebugger")

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 CCDebugger。你可以在 堆栈 SS:[0012F9AC]=00D44DB4, (ASCII "CCDebugger") 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 CCDebugger 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“CCDebugger”

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：

地址	HEX 数据	反汇编	注释
00403B2C	53	PUSH EBX	
00403B2D	56	PUSH ESI	
00403B2E	57	PUSH EDI	
00403B2F	89C6	MOV ESI,EAX	

光标停在这里。地址底色显示为黑色时表示我们现在执行到这条指令

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：

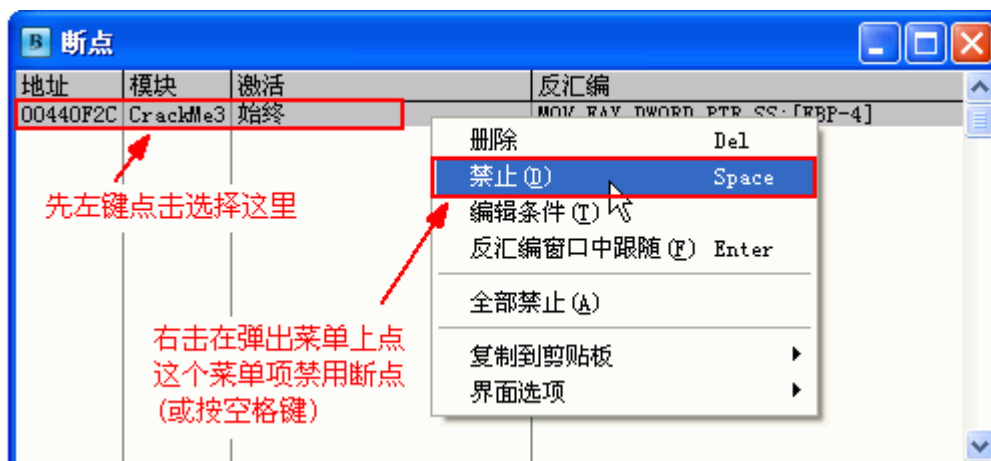
00403B2C /\$ 53	PUSH EBX	
00403B2D . 56	PUSH ESI	
00403B2E . 57	PUSH EDI	
00403B2F . 89C6	MOV ESI,EAX	; 把 EAX 内我们输入的用户名送到 ESI
00403B31 . 89D7	MOV EDI,EDX	; 把 EDX 内的数据“Registered User”送到 EDI
00403B33 . 39D0	CMP EAX,EDX	; 用“Registered User”和我们输入的用户名作比较
00403B35 . 0F84 8F000000	JE CrackMe3.00403BCA	; 相同则跳
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”字符串的长度送 EDX
00403B49 . 29D0	SUB EAX,EDX	; 把用户名长度和“Registered User”字符串长度相减
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

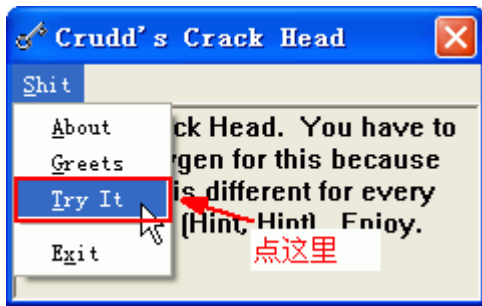
注册码：GFX-754-IER-954

点击“Register now !”按钮，呵呵，终于成功了：



函数参考

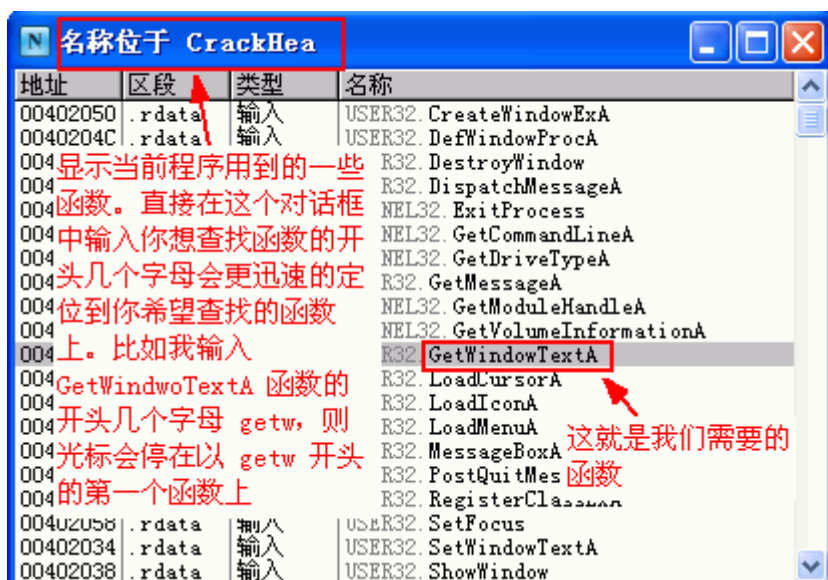
现在进入第三篇，这一篇我们重点讲解怎样使用 OllyDBG 中的函数参考（即名称参考）功能。仍然选择 crackmes.cjb.net 镜像打包中的一个名称为 CrackHead 的 crackme。老规矩，先运行一下这个程序看看：



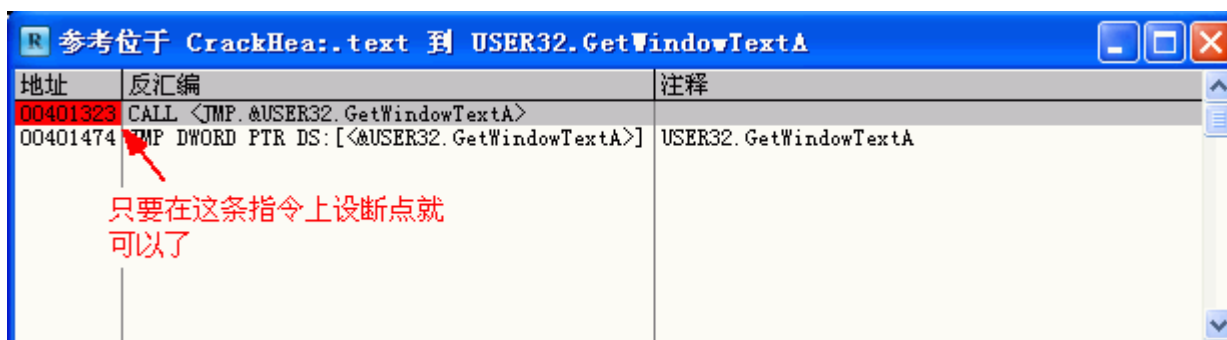
呵，竟然没找到输入注册码的地方！别急，我们点一下程序上的那个菜单“Shit”（真是 Shit 啊，呵呵），在下拉菜单中选“Try It”，会来到如下界面：



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符串进行转换，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>      ; GetWin
dowTextA
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&apos;re done.",0
0040135F |> 6A 00            PUSH 0                              ; Style = MB_OK|MB_APP
LMODAL
00401361 |. 68 0F304000      PUSH CrackHea.0040300F              ; Title = "Crudd
&apos;s Crack Head"
00401366 |. 68 33134000      PUSH CrackHea.00401333              ; Text = "No
w write a keygen and tut and you&apos;re done."
0040136B |. FF75 08          PUSH DWORD PTR SS:[EBP+8]           ; hOwner
0040136E |. E8 19010000      CALL <JMP.&USER32.MessageBoxA>      ; Message
BoxA

```

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>  ; U
SER32.GetWindowTextA
0040147A $- FF25 30204000  JMP DWORD PTR DS:[<&USER32.LoadCursorA>]   ; US
ER32.LoadCursorA
00401480 $- FF25 1C204000  JMP DWORD PTR DS:[<&USER32.LoadIconA>]    ; US

```

ER32.LoadIconA

00401486 \$- FF25 20204000 JMP DWORD PTR DS:[<&USER32.LoadMenuA>] ; US

ER32.LoadMenuA

0040148C \$- FF25 24204000 JMP DWORD PTR DS:[<&USER32.MessageBoxA>] ; U

SER32.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
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	

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：

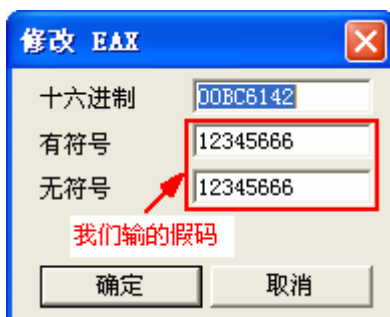
ESI=E6B5F2F9

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符串“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—内存断点 中再讲吧。

内存断点

还记得上一篇的内容吗？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>	; GetWin dowTextA
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：

地址	HEX 数据	反汇编	注释
0040130E	75 63	JNZ SHORT CrackHea.00401373	
00401310	8B35 9C334000	MOV ESI, DWORD PTR DS:[40339C]	关键，记住这个40339C的内存地址
00401316	6A 28	PUSH 28	Count = 28 (40.)
00401318	68 C4334000	PUSH CrackHea.004033C4	Buffer = CrackHea.004033C4
0040131D	FF35 90314000	PUSH DWORD PTR DS:记住这个内存地址，等会要用	hWnd = 003905FC (class='Edit', parent=000B0588)
00401323	E8 4C010000	CALL <JMP.&USER32	GetWindowTextA
00401328	E8 A5000000	CALL CrackHea.0040132D	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	

DS:[0040339C]=9FCF87AA

保持反汇编窗口中光标在地址00401310指令处，在信息窗口中左键点击这一条再右击，在弹出菜单中选择“数据窗口中跟随地址”，看看内存地址中的内容

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：

地址	HEX 数据	ASCII
0040339C	AA 87 CF 9F 00 00 00 00 00 00 00 00 00 00 00 00 00	徽
004033AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
004033BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
004033CC	内存地址040339C中当前值	12345666
004033DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
004033EC	03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
004033FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040340C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040341C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040342C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040343C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040344C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...
0040345C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...

我们输入的假码

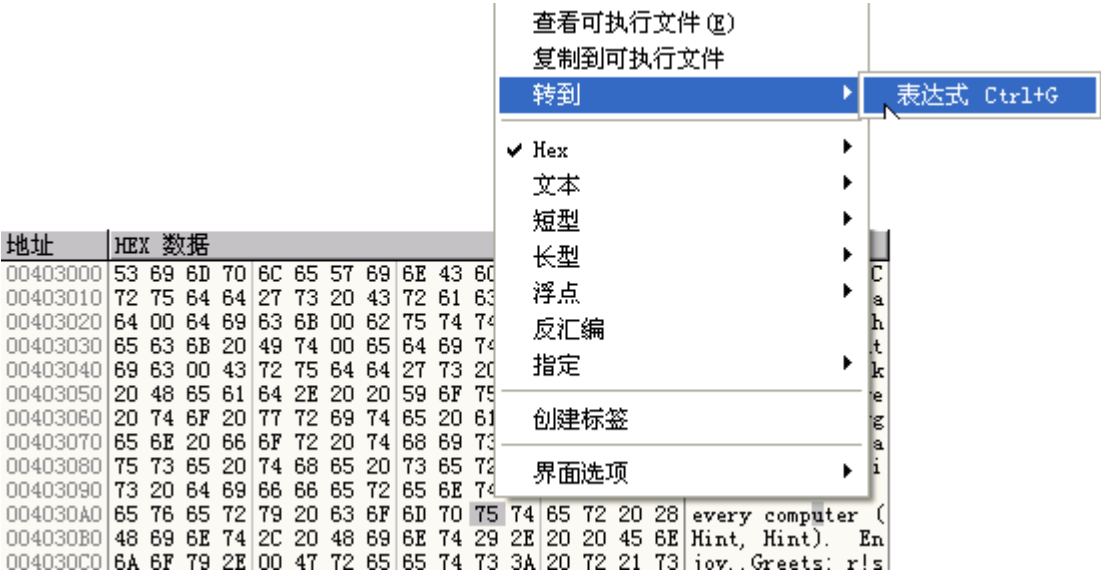
注意一下这个值，等会我们会看到它是怎
么出来的

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：



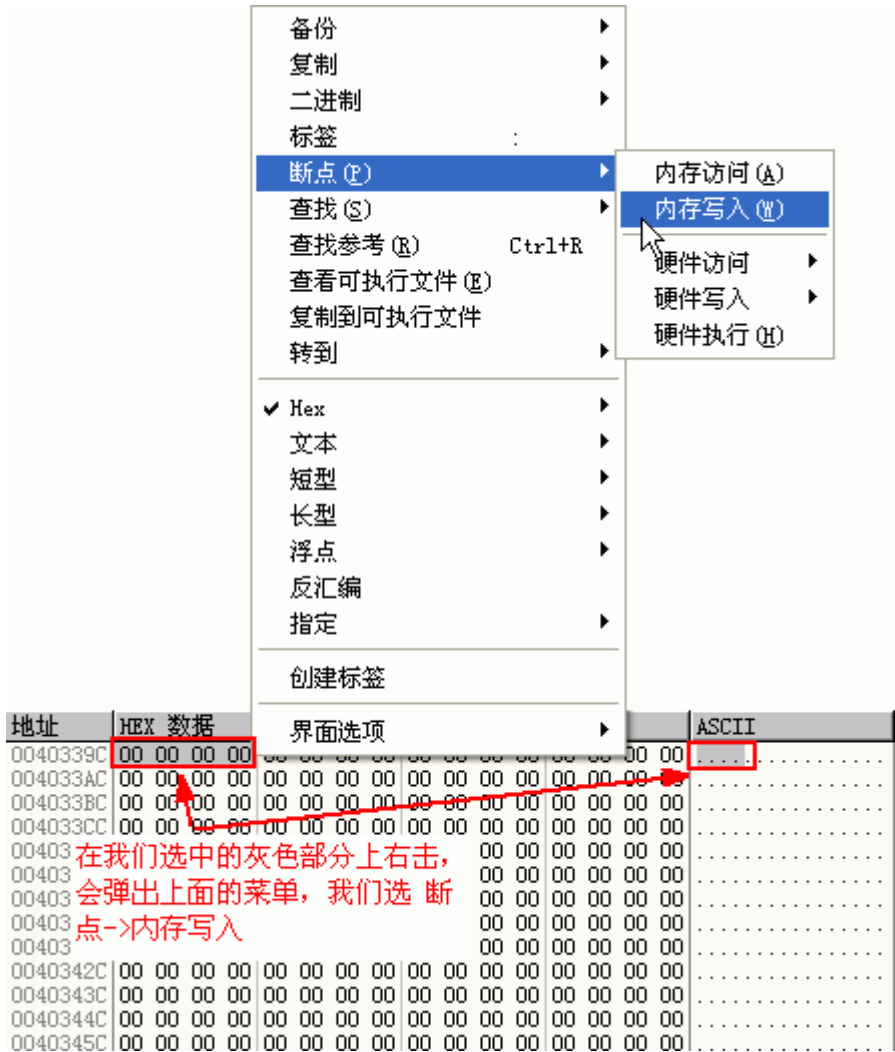
现在将会出现这样一个对话框：

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：

地址	HEX	数据	ASCII
0040339C	00 00 00 00	00 00 00 00 00 00 00 00
004033AC	00 00 00 00	00 00 00 00 00 00 00 00
004033BC	00 00 00 00	00 00 00 00 00 00 00 00
004033CC	00 00 00 00	00 00 00 00 00 00 00 00
004033DC	00 00 00 00	00 00 00 00 00 00 00 00
004033EC	00 00 00 00	00 00 00 00 00 00 00 00
004033FC	00 00 00 00	00 00 00 00 00 00 00 00
0040340C	00 00 00 00	00 00 00 00 00 00 00 00
0040341C	00 00 00 00	00 00 00 00 00 00 00 00
0040342C	00 00 00 00	00 00 00 00 00 00 00 00
0040343C	00 00 00 00	00 00 00 00 00 00 00 00
0040344C	00 00 00 00	00 00 00 00 00 00 00 00
0040345C	00 00 00 00	00 00 00 00 00 00 00 00

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]
```

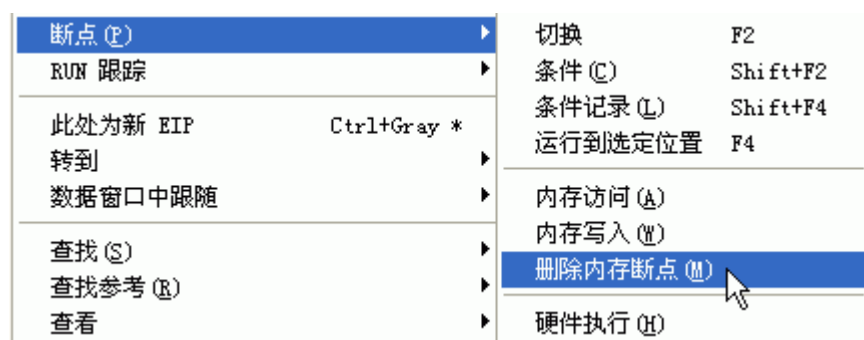
上面就是我们中断后反汇编窗口中的代码。如果你是其它系统，如 Win98 的话，可能会有所不同。没关

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：

地址	HEX 数据	ASCII
0040339C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004033AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004033BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004033CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004033DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004033EC	03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	L.....
004033FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040340C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040341C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040342C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040343C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040344C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040345C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

注意这里被写入内容了，
原来这里是 00 的

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？

地址	HEX 数据	反汇编	注释
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
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
00401402	33C2	XOR EAX, EDX	把EAX与EDX异或。如果我们输入的是负数,则此
00401404	5E	POP ESI	ESI出栈。看到这条和下一条指令,我们要考虑一
00401405	81F6 53757A79	XOR ESI, 797A7553	把ESI中的值与797A7553H异或
0040140B	C3	RETN	
0040140C	\$ 60	PUSHAD	
0040140E	6A 00	PUSH 0	
00401410	EB B4000000	CALL <TMP_00401410>	RootPathName = NULL
00401412	EB 00000000	CALL <TMP_00401412>	GetDriveTypeA
00401414	EB 00000000	CALL <TMP_00401414>	
00401416	EB 00000000	CALL <TMP_00401416>	
00401418	EB 00000000	CALL <TMP_00401418>	
0040141A	EB 00000000	CALL <TMP_0040141A>	
0040141C	EB 00000000	CALL <TMP_0040141C>	
0040141E	EB 00000000	CALL <TMP_0040141E>	
00401420	EB 00000000	CALL <TMP_00401420>	
00401422	EB 00000000	CALL <TMP_00401422>	
00401424	EB 00000000	CALL <TMP_00401424>	
00401426	EB 00000000	CALL <TMP_00401426>	
00401428	EB 00000000	CALL <TMP_00401428>	
0040142A	EB 00000000	CALL <TMP_0040142A>	
0040142C	EB 00000000	CALL <TMP_0040142C>	
0040142E	EB 00000000	CALL <TMP_0040142E>	
00401430	EB 00000000	CALL <TMP_00401430>	
00401432	EB 00000000	CALL <TMP_00401432>	
00401434	EB 00000000	CALL <TMP_00401434>	
00401436	EB 00000000	CALL <TMP_00401436>	
00401438	EB 00000000	CALL <TMP_00401438>	
0040143A	EB 00000000	CALL <TMP_0040143A>	
0040143C	EB 00000000	CALL <TMP_0040143C>	
0040143E	EB 00000000	CALL <TMP_0040143E>	
00401440	EB 00000000	CALL <TMP_00401440>	
00401442	EB 00000000	CALL <TMP_00401442>	
00401444	EB 00000000	CALL <TMP_00401444>	
00401446	EB 00000000	CALL <TMP_00401446>	
00401448	EB 00000000	CALL <TMP_00401448>	
0040144A	EB 00000000	CALL <TMP_0040144A>	
0040144C	EB 00000000	CALL <TMP_0040144C>	
0040144E	EB 00000000	CALL <TMP_0040144E>	
00401450	EB 00000000	CALL <TMP_00401450>	
00401452	EB 00000000	CALL <TMP_00401452>	
00401454	EB 00000000	CALL <TMP_00401454>	
00401456	EB 00000000	CALL <TMP_00401456>	
00401458	EB 00000000	CALL <TMP_00401458>	
0040145A	EB 00000000	CALL <TMP_0040145A>	
0040145C	EB 00000000	CALL <TMP_0040145C>	
0040145E	EB 00000000	CALL <TMP_0040145E>	
00401460	EB 00000000	CALL <TMP_00401460>	
00401462	EB 00000000	CALL <TMP_00401462>	
00401464	EB 00000000	CALL <TMP_00401464>	
00401466	EB 00000000	CALL <TMP_00401466>	
00401468	EB 00000000	CALL <TMP_00401468>	
0040146A	EB 00000000	CALL <TMP_0040146A>	
0040146C	EB 00000000	CALL <TMP_0040146C>	
0040146E	EB 00000000	CALL <TMP_0040146E>	
00401470	EB 00000000	CALL <TMP_00401470>	
00401472	EB 00000000	CALL <TMP_00401472>	
00401474	EB 00000000	CALL <TMP_00401474>	
00401476	EB 00000000	CALL <TMP_00401476>	
00401478	EB 00000000	CALL <TMP_00401478>	
0040147A	EB 00000000	CALL <TMP_0040147A>	
0040147C	EB 00000000	CALL <TMP_0040147C>	
0040147E	EB 00000000	CALL <TMP_0040147E>	
00401480	EB 00000000	CALL <TMP_00401480>	
00401482	EB 00000000	CALL <TMP_00401482>	
00401484	EB 00000000	CALL <TMP_00401484>	
00401486	EB 00000000	CALL <TMP_00401486>	
00401488	EB 00000000	CALL <TMP_00401488>	
0040148A	EB 00000000	CALL <TMP_0040148A>	
0040148C	EB 00000000	CALL <TMP_0040148C>	
0040148E	EB 00000000	CALL <TMP_0040148E>	
00401490	EB 00000000	CALL <TMP_00401490>	
00401492	EB 00000000	CALL <TMP_00401492>	
00401494	EB 00000000	CALL <TMP_00401494>	
00401496	EB 00000000	CALL <TMP_00401496>	
00401498	EB 00000000	CALL <TMP_00401498>	
0040149A	EB 00000000	CALL <TMP_0040149A>	
0040149C	EB 00000000	CALL <TMP_0040149C>	
0040149E	EB 00000000	CALL <TMP_0040149E>	
004014A0	EB 00000000	CALL <TMP_004014A0>	
004014A2	EB 00000000	CALL <TMP_004014A2>	
004014A4	EB 00000000	CALL <TMP_004014A4>	
004014A6	EB 00000000	CALL <TMP_004014A6>	
004014A8	EB 00000000	CALL <TMP_004014A8>	
004014AA	EB 00000000	CALL <TMP_004014AA>	
004014AC	EB 00000000	CALL <TMP_004014AC>	
004014AE	EB 00000000	CALL <TMP_004014AE>	
004014B0	EB 00000000	CALL <TMP_004014B0>	
004014B2	EB 00000000	CALL <TMP_004014B2>	
004014B4	EB 00000000	CALL <TMP_004014B4>	
004014B6	EB 00000000	CALL <TMP_004014B6>	
004014B8	EB 00000000	CALL <TMP_004014B8>	
004014BA	EB 00000000	CALL <TMP_004014BA>	
004014BC	EB 00000000	CALL <TMP_004014BC>	
004014BE	EB 00000000	CALL <TMP_004014BE>	
004014C0	EB 00000000	CALL <TMP_004014C0>	
004014C2	EB 00000000	CALL <TMP_004014C2>	
004014C4	EB 00000000	CALL <TMP_004014C4>	
004014C6	EB 00000000	CALL <TMP_004014C6>	
004014C8	EB 00000000	CALL <TMP_004014C8>	
004014CA	EB 00000000	CALL <TMP_004014CA>	
004014CC	EB 00000000	CALL <TMP_004014CC>	
004014CE	EB 00000000	CALL <TMP_004014CE>	
004014D0	EB 00000000	CALL <TMP_004014D0>	
004014D2	EB 00000000	CALL <TMP_004014D2>	
004014D4	EB 00000000	CALL <TMP_004014D4>	
004014D6	EB 00000000	CALL <TMP_004014D6>	
004014D8	EB 00000000	CALL <TMP_004014D8>	
004014DA	EB 00000000	CALL <TMP_004014DA>	
004014DC	EB 00000000	CALL <TMP_004014DC>	
004014DE	EB 00000000	CALL <TMP_004014DE>	
004014E0	EB 00000000	CALL <TMP_004014E0>	
004014E2	EB 00000000	CALL <TMP_004014E2>	
004014E4	EB 00000000	CALL <TMP_004014E4>	
004014E6	EB 00000000	CALL <TMP_004014E6>	
004014E8	EB 00000000	CALL <TMP_004014E8>	
004014EA	EB 00000000	CALL <TMP_004014EA>	
004014EC	EB 00000000	CALL <TMP_004014EC>	
004014EE	EB 00000000	CALL <TMP_004014EE>	
004014F0	EB 00000000	CALL <TMP_004014F0>	
004014F2	EB 00000000	CALL <TMP_004014F2>	
004014F4	EB 00000000	CALL <TMP_004014F4>	
004014F6	EB 00000000	CALL <TMP_004014F6>	
004014F8	EB 00000000	CALL <TMP_004014F8>	
004014FA	EB 00000000	CALL <TMP_004014FA>	
004014FC	EB 00000000	CALL <TMP_004014FC>	
004014FE	EB 00000000	CALL <TMP_004014FE>	
00401500	EB 00000000	CALL <TMP_00401500>	
00401502	EB 00000000	CALL <TMP_00401502>	
00401504	EB 00000000	CALL <TMP_00401504>	
00401506	EB 00000000	CALL <TMP_00401506>	
00401508	EB 00000000	CALL <TMP_00401508>	
0040150A	EB 00000000	CALL <TMP_0040150A>	
0040150C	EB 00000000	CALL <TMP_0040150C>	
0040150E	EB 00000000	CALL <TMP_0040150E>	
00401510	EB 00000000	CALL <TMP_00401510>	
00401512	EB 00000000	CALL <TMP_00401512>	
00401514	EB 00000000	CALL <TMP_00401514>	
00401516	EB 00000000	CALL <TMP_00401516>	
00401518	EB 00000000	CALL <TMP_00401518>	
0040151A	EB 00000000	CALL <TMP_0040151A>	
0040151C	EB 00000000	CALL <TMP_0040151C>	
0040151E	EB 00000000	CALL <TMP_0040151E>	
00401520	EB 00000000	CALL <TMP_00401520>	
00401522	EB 00000000	CALL <TMP_00401522>	
00401524	EB 00000000	CALL <TMP_00401524>	
00401526	EB 00000000	CALL <TMP_00401526>	
00401528	EB 00000000	CALL <TMP_00401528>	
0040152A	EB 00000000	CALL <TMP_0040152A>	
0040152C	EB 00000000	CALL <TMP_0040152C>	
0040152E	EB 00000000	CALL <TMP_0040152E>	
00401530	EB 00000000	CALL <TMP_00401530>	
00401532	EB 00000000	CALL <TMP_00401532>	
00401534	EB 00000000	CALL <TMP_00401534>	
00401536	EB 00000000	CALL <TMP_00401536>	
00401538	EB 00000000	CALL <TMP_00401538>	
0040153A	EB 00000000	CALL <TMP_0040153A>	
0040153C	EB 00000000	CALL <TMP_0040153C>	
0040153E	EB 00000000	CALL <TMP_0040153E>	
00401540	EB 00000000	CALL <TMP_00401540>	
00401542	EB 00000000	CALL <TMP_00401542>	
00401544	EB 00000000	CALL <TMP_00401544>	
00401546	EB 00000000	CALL <TMP_00401546>	
00401548	EB 00000000	CALL <TMP_00401548>	
0040154A	EB 00000000	CALL <TMP_0040154A>	
0040154C	EB 00000000	CALL <TMP_0040154C>	
0040154E	EB 00000000	CALL <TMP_0040154E>	
00401550	EB 00000000	CALL <TMP_00401550>	
00401552	EB 00000000	CALL <TMP_00401552>	
00401554	EB 00000000	CALL <TMP_00401554>	
00401556	EB 00000000	CALL <TMP_00401556>	
00401558	EB 00000000	CALL <TMP_00401558>	
0040155A	EB 00000000	CALL <TMP_0040155A>	
0040155C	EB 00000000	CALL <TMP_0040155C>	
0040155E	EB 00000000	CALL <TMP_0040155E>	
00401560	EB 00000000	CALL <TMP_00401560>	
00401562	EB 00000000	CALL <TMP_00401562>	
00401564	EB 00000000	CALL <TMP_00401564>	
00401566	EB 00000000	CALL <TMP_00401566>	
00401568	EB 00000000	CALL <TMP_00401568>	
0040156A	EB 00000000	CALL <TMP_0040156A>	
0040156C	EB 00000000	CALL <TMP_0040156C>	
0040156E	EB 00000000	CALL <TMP_0040156E>	
00401570	EB 00000000	CALL <TMP_00401570>	
00401572	EB 00000000	CALL <TMP_00401572>	
00401574	EB 00000000	CALL <TMP_00401574>	
00401576	EB 00000000	CALL <TMP_00401576>	
00401578	EB 00000000	CALL <TMP_00401578>	
0040157A	EB 00000000	CALL <TMP_0040157A>	
0040157C	EB 00000000	CALL <TMP_0040157C>	
0040157E	EB 00000000	CALL <TMP_0040157E>	
00401580	EB 00000000	CALL <TMP_00401580>	
00401582	EB 00000000	CALL <TMP_00401582>	
00401584	EB 00000000	CALL <TMP_00401584>	
00401586	EB 00000000	CALL <TMP_00401586>	
00401588	EB 00000000	CALL <TMP_00401588>	
0040158A	EB 00000000	CALL <TMP_0040158A>	
0040158C	EB 00000000	CALL <TMP_0040158C>	
0040158E	EB 00000000	CALL <TMP_0040158E>	
00401590	EB 00000000	CALL <TMP_00401590>	
00401592	EB 00000000	CALL <TMP_00401592>	
00401594	EB 00000000	CALL <TMP_00401594>	
00401596	EB 00000000	CALL <TMP_00401596>	
00401598	EB 00000000	CALL <TMP_00401598>	
0040159A	EB 00000000	CALL <TMP_0040159A>	
0040159C	EB 00000000	CALL <TMP_0040159C>	
0040159E	EB 00000000	CALL <TMP_0040159E>	
004015A0	EB 00000000	CALL <TMP_004015A0>	
004015A2	EB 00000000	CALL <TMP_004015A2>	
004015A4	EB 00000000	CALL <TMP_004015A4>	
004015A6	EB 00000000	CALL <TMP_004015A6>	
004015A8	EB 00000000	CALL <TMP_004015A8>	
004015AA	EB 00000000	CALL <TMP_004015AA>	
004015AC	EB 00000000	CALL <TMP_004015AC>	
004015AE	EB 00000000	CALL <TMP_004015AE>	
004015B0	EB 00000000	CALL <TMP_004015B0>	
004015B2	EB 00000000	CALL <TMP_004015B2>	
004015B4	EB 00000000	CALL <TMP_004015B4>	
004015B6	EB 00000000	CALL <TMP_004015B6>	
004015B8	EB 00000000	CALL <TMP_004015B8></	

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：

```

00401425 |. 68 9C334000    PUSH CrackHea.0040339C          ; |VolumeName
Buffer = CrackHea.0040339C

0040142A |. 6A 00          PUSH 0              ; |RootPathName = NULL

0040142C |. E8 A3000000    CALL <JMP.&KERNEL32.GetVolumeInformationA> ; \GetV
olumeInformationA

00401431 |. 8D35 9C334000    LEA ESI,DWORD PTR DS:[40339C]          ; 把 crackm
e 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：

存储器	地址
...	
47H	40339CH
41H	40339DH
4DH	40339EH
45H	40339FH
...	

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 $b = a * [n + (n - 1) + (n - 2) + \dots + 1] = a * [n * (n + 1) / 2]$

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
e = NULL
0040141B |. 6A 00          PUSH 0          ; |pFileSystemNameBuffer
r = NULL
0040141D |. 6A 00          PUSH 0          ; |pFileSystemFlags = NULL
0040141F |. 6A 00          PUSH 0          ; |pMaxFilenameLength = NULL
ULL
00401421 |. 6A 00          PUSH 0          ; |pVolumeSerialNumber
r = NULL
00401423 |. 6A 0B          PUSH 0B         ; |MaxVolumeNameSize
e = 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 OillyDBG 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：

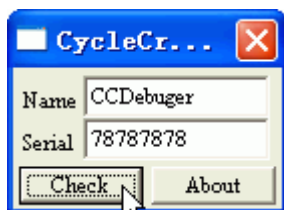
Value	Meaning	返回在 EAX 中的值
DRIVE_UNKNOWN	The drive type cannot be determined.	0
DRIVE_NO_ROOT_DIR	The root directory does not exist.	1
DRIVE_REMOVABLE	The disk can be removed from the drive.	2
DRIVE_FIXED	The disk cannot be removed from the drive.	3
DRIVE_REMOTE	The drive is a remote (network) drive.	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。

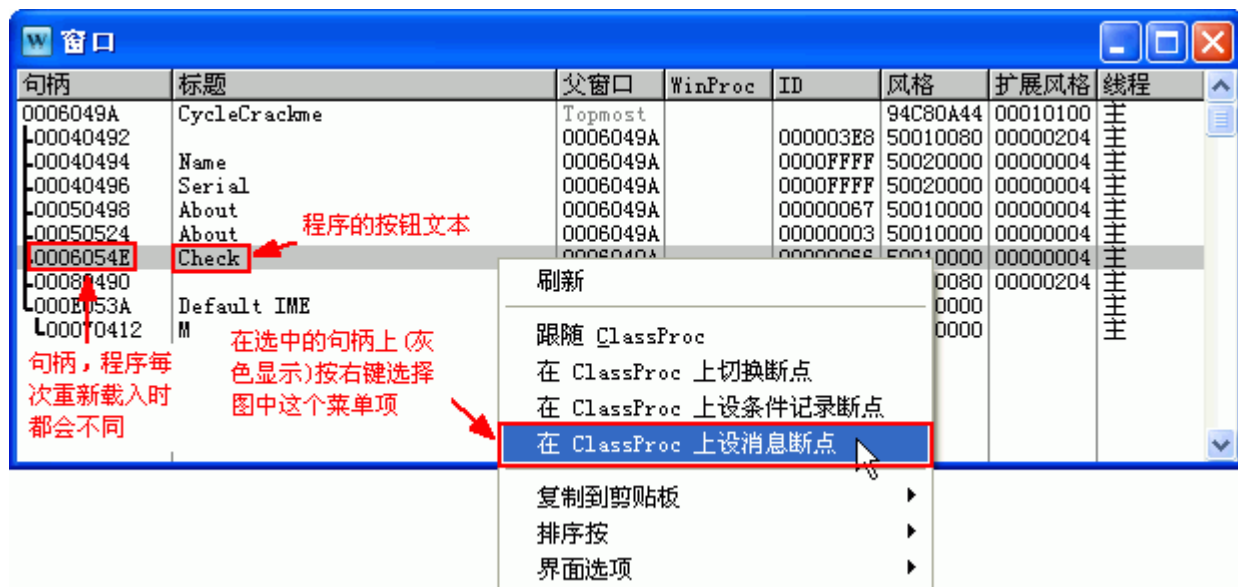
消息断点及 RUN 跟踪

找了几十个不同语言编写的 crackme，发现只用消息断点的话有很多并不能真正到达我们要找的关键位置，想想还是把消息断点和 RUN 跟踪结合在一起讲，更有效一点。关于消息断点的更多内容大家可以参考 jingulong 兄的那篇《几种典型程序 Button 处理代码的定位》的文章，堪称经典之作。今天仍然选择 crackmes.cjb.net 镜像打包中的一个名称为 cycle 的 crackme。按照惯例，我们先运行一下这个程序看看：

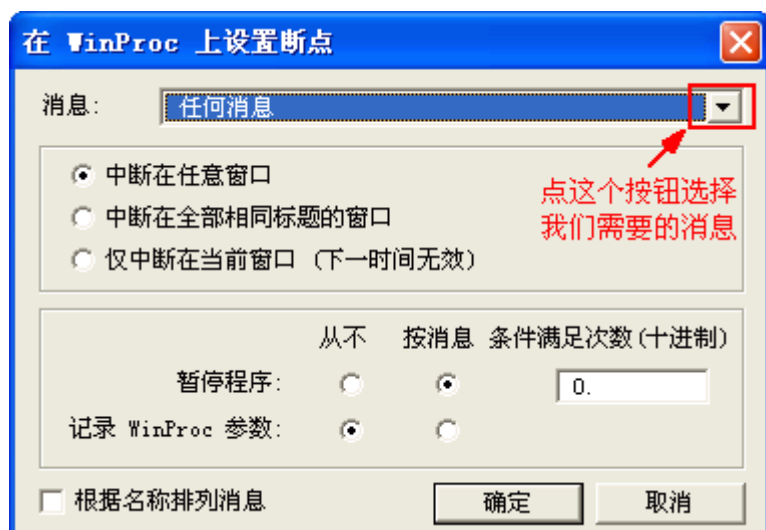


我们输入用户名 CCDebugger，序列号 78787878，点上面那个“Check”按钮，呵，没反应！看来是要注册码正确才有动静。现在关掉这个 crackme，用 PEiD 查一下壳，原来是 MASM32 / TASM32 [Overlay]。启动 OllyDBG 载入这个程序，F9 让它运行。这个程序按我们前面讲的采用字串参考或函数参考的方法都很容易断下来。但我们今天主要学习的是消息断点及 RUN 跟踪，就先用消息断点来断这个程序吧。在设消息断点前，有两个内容我们要简单了解一下：首先我们要了解的是消息。Windows 的中文翻译就是“窗口”，而 Windows 上面的应用程序也都是通过窗口来与用户交互的。现在就有一个问题，应用程序是如何知道用户作了什么样的操作的？这里就要用到消息了。Windows 是个基于消息的系统，它在应用程序开始执行后，为该程序创建一个“消息队列”，用来存放该程序可能创建的各种不同窗口的信息。比如你创建窗口、点击按钮、移动鼠标等等，都是通过消息来完成的。通俗的说，Windows 就像一个中间人，你要干什么事是先通知它，然后它才通过传递消息的方式通知应用程序作出相应的操作。说到这，又有个问题了，在 Windows 下有多个程序都在运行，那我点了某个按钮，或把某个窗口最大化，Windows 知道我是点的哪个吗？这里就要说到另一个内容：句柄（handle）了。句柄一般是个 32 位的数，表示一个对象。Windows 通过使用句柄来标识它代表的对象。比如你点击某个按钮，Windows 就是通

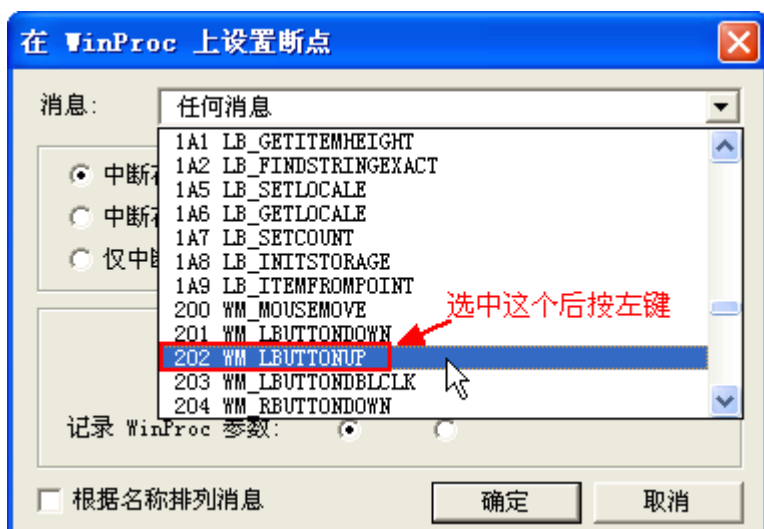
过句柄来判断你是点击了那一个按钮，然后发送相应的消息通知程序。说完这些我们再回到我们调试的程序上来，你应该已经用 OllyDBG 把这个 crackme 载入并按 F9 键运行了吧？现在我们输入用户名“CC Debugger”，序列号“78787878”，先不要点那个“Check”按钮，我们来到 OllyDBG 中，点击菜单 查看 -> 窗口（或者点击工具栏上那个“W”的图标），我们会看到以下内容：



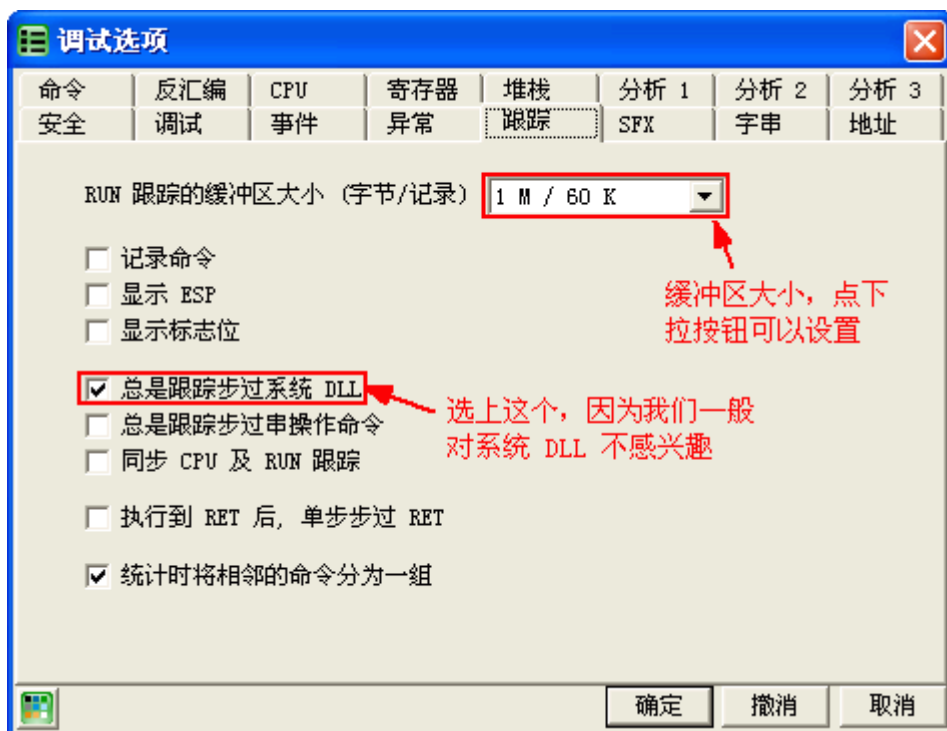
我们在选中的条目上点右键，再选择上图所示的菜单项，会来到下面这个窗口：



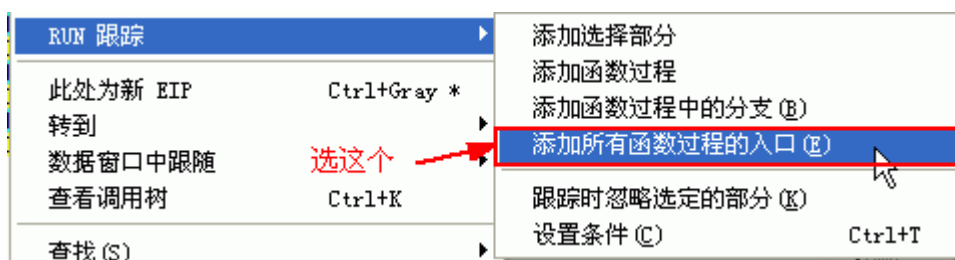
现在我们点击图上的那个下拉菜单，呵，原来里面的消息真不少。这么多消息我们选哪个呢？注册是个按钮，我们就在按下按钮再松开时让程序中断。查一下 MSDN，我们知道这个消息应该是 WM_LBUTTONDOWN，看字面意思也可以知道是左键松开时的消息：



从下拉菜单中选中那个 202 WM_LBUTTONDOWN，再按确定按钮，我们的消息断点就设好了。现在我们要做一件事，就是把 RUN 跟踪打开。有人可能要问，这个 RUN 跟踪是干什么的？简单的说，RUN 跟踪就是把被调试程序执行过的指令保存下来，让你可以查看被调试程序运行期间干了哪些事。RUN 跟踪会把地址、寄存器的内容、消息以及已知的操作数记录到 RUN 跟踪缓冲区中，你可以通过查看 RUN 跟踪的记录来了解程序执行了那些指令。在这还要注意一个缓冲区大小的问题，如果执行的指令太多，缓冲区满了的话，就会自动丢弃前面老的记录。我们可以在调试选项->跟踪中设置：



现在我们回到 OllyDBG 中，点击菜单调试->打开或清除 RUN 跟踪（第一次点这个菜单是打开 RUN 跟踪，在打开的情况下点击就是清除 RUN 跟踪的记录，对 RUN 跟踪熟悉时还可以设置条件），保证当前在我们调试的程序领空，在反汇编窗口中点击右键，在弹出菜单中选择 RUN 跟踪->添加所有函数过程的入口：



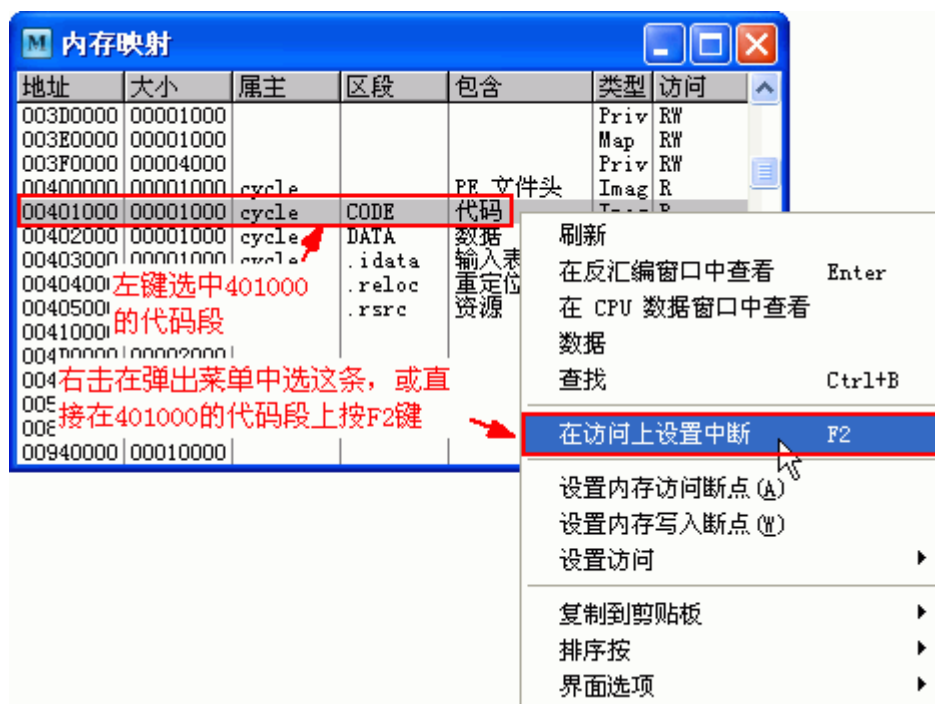
我们可以看到 OllyDBG 把识别出的函数过程都在前面加了灰色条：

地址	HEX 数据	反汇编	注释
00401000	6A 00	PUSH 0	pModule = NULL
00401002	E8 A4020000	CALL <JMP.@KERNEL32.GetModuleHandleA>	GetModuleHandleA
00401007	A3 94214000	MOV DWORD PTR DS:[402194], EAX	
0040100C	6A 00	PUSH 0	lParam = NULL
0040100E	68 29104000	PUSH cycle.00401029	DlgProc = cycle.00401029
00401013	6A 00	PUSH 0	hOwner = NULL
00401015	6A 00	PUSH 0	pTemplate = 68
00401017	FF35 00000000	JMP DWORD PTR DS:[402194]	hInst = 00400000
0040101D	E8 87020000	CALL <JMP.@USER32.ShowDialogParamA>	DialogBoxParamA
00401022	6A 00	PUSH 0	ExitCode = 0
00401024	E8 7C020000	CALL <JMP.@KERNEL32.ExitProcess>	ExitProcess

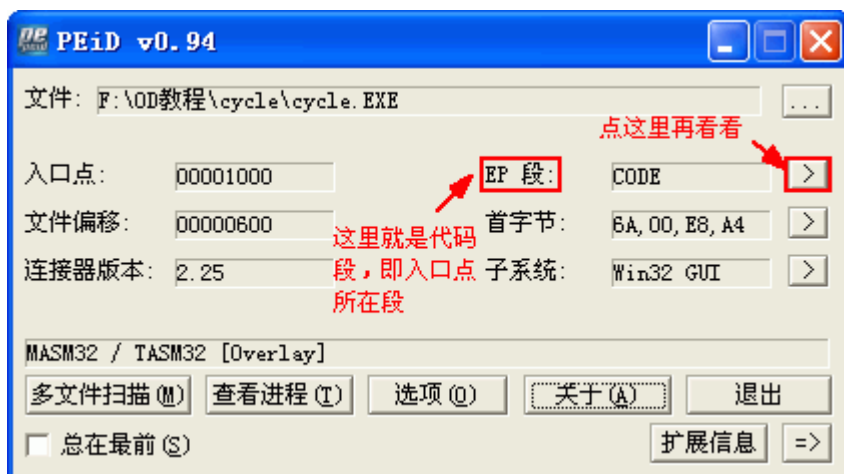
现在我们回到那个 crackme 中按那个“Check”按钮，被 OllyDBG 断下了：

地址	HEX 数据	反汇编	注释
77D3B00E	8BFF	MOV EDI, EDI	
77D3B010	55	PUSH EBP	
77D3B011	8BE	MOV EBP, ESP	
77D3B013	8B4C	MOV ECX, DWORD PTR SS:[EBP+8]	

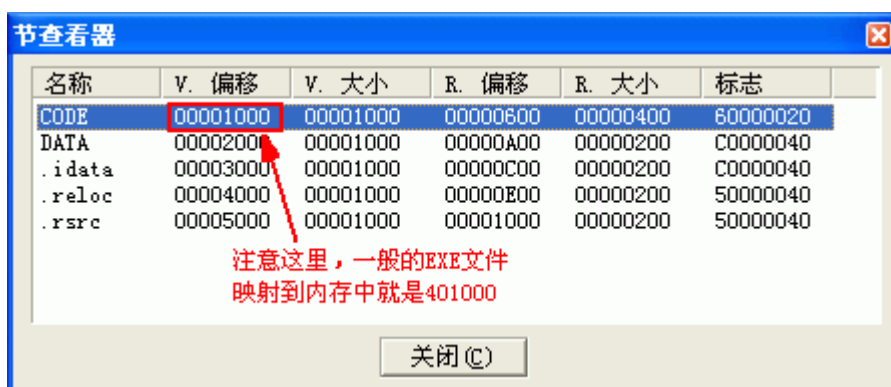
这时我们点击菜单查看->内存，或者点击工具栏上那个“M”按钮（也可以按组合键 ALT+M），来到内存映射窗口：



为什么在这里设访问断点，我也说一下。我们可以看一下常见的 PE 文件，没加过壳的用 PEiD 检测是这样：



点一下 EP 段后面那个">"符号，我们可以看到以下内容：



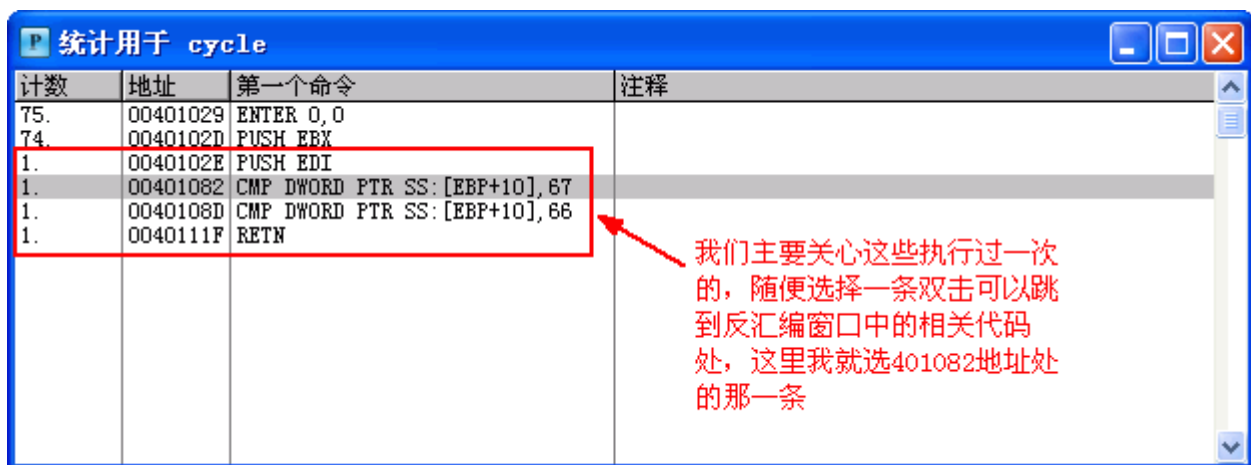
看完上面的图我们应该了解为什么在 401000 处的代码段下访问断点了，我们这里的意思就是在消息断点断下后，只要按 F9 键运行时执行到程序代码段的指令我们就中断，这样就可以回到程序领空了（当然在 401000 处所在的段不是绝对的，我们主要是要看程序的代码段在什么位置，其实在上面图中 OlllyDBG 内存窗口的“包含”栏中我们就可以看得很清楚了）。设好访问断点后我们按 F9 键，被 OlllyDBG 断下：

地址	HEX 数据	反汇编	注释
00401022	6A 00	PUSH 0	
00401024	E8 7C020000	CALL <TMP.&KERNEL32.ExitProcess>	ExitCode = 0
00401029	C8 000000	ENTER 0,0	ExitProcess
0040102D	53	PUSH EBX	
0040102E	57	PUSH EDI	
0040102F	56	PUSH ESI	
00401030	817D 0C 10010	CMP DWORD PTR SS:[EBP+C],110	
00401037	74 25	JE SHORT cycle.0040105E	
00401039	817D 0C 11010	CMP DWORD PTR SS:[EBP+C],111	
00401040	74 40	JE SHORT cycle.00401082	
00401042	837D 0C 10	CMP DWORD PTR SS:[EBP+C],10	
00401046	74 0F	JE SHORT cycle.00401057	
00401048	837D 0C 02	CMP DWORD PTR SS:[EBP+C],2	
0040104C	74 09	JE SHORT cycle.00401057	
0040104E	33C0	XOR EAX,EAX	
00401050	5E	POP ESI	
00401051	5F	POP EDI	
00401052	5B	POP EBX	
00401053	C9	LEAVE	
00401054	C2 1000	RETN 10	
00401057	6A 00	PUSH 0	

现在我们先不管，按 F9 键（或者按 CTR+F12 组合键跟踪步过）让程序运行，再点击菜单查看->RUN 跟踪，或者点击工具栏上的那个"... "符号，打开 RUN 跟踪的记录窗口看看：



我们现在再来看看统计的情况：



在地址 401082 处的那条指令上双击一下，来到以下位置：

地址	HEX 数据	反汇编	注释
00401080	EB CE	JMP SHORT cycle.00401050	
00401082	837D 10 67	CMP DWORD PTR SS:[EBP+10], 67	
00401086	75 05	JNZ SHORT cycle.0040108D	
00401088	E8 C4000000	CALL cycle.00401151	双击后来到的位置
0040108D	837D 10 66	CMP DWORD PTR SS:[EBP+10], 66	
00401091	75 05	JNZ SHORT cycle.00401098	
00401093	E8 04000000	CALL cycle.0040109C	
00401098	33C0	XOR EAX, EAX	
0040109A	EB B4	JMP SHORT cycle.00401050	
0040109C	C705 82214000	MOV DWORD PTR DS:[402182], FEDCBA98	
004010A6	6A 11	PUSH 11	
004010A8	68 71214000	PUSH cycle.00402171	
004010AD	68 E9030000	PUSH 3E9	
004010B2	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004010B5	E8 07000000	CALL &USER32.GetDlgItemTextA	GetDlgItemTextA
004010BA	0BC0	OR EAX, EAX	
004010BC	74 6	JE SHORT cycle.0040111F	这个函数熟悉吧
004010BE	6A 1	PUSH 1	
004010C0	68 60214000	PUSH cycle.00402160	
004010C5	68 E8030000	PUSH 3E8	
004010CA	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004010CD	E8 F7010000	CALL <JMP.&USER32.GetDlgItemTextA>	
004010D2	0BC0	OR EAX, EAX	
004010D4	74 49	JE SHORT cycle.0040111F	看到我们输入用户名的变换了
004010D6	B9 10000000	MOV ECX, 10	
004010DB	2BC8	SUB ECX, EAX	
004010DD	BE 60214000	MOV ESI, cycle.00402160	
004010E2	8BFE	MOV EDI, ESI	
004010E4	03F8	ADD EDI, EAX	

现在我们在地址 4010A6 处的那条指令上按 F2，删除所有其它的断点，点菜单调试->关闭 RUN 跟踪，现在我们可以开始分析了：

```

004010E2 |. 8BFE      MOV EDI,ESI                ; 用户名送 EDI
004010E4 |. 03F8      ADD EDI,EAX
004010E6 |. FC       CLD
004010E7 |. F3:A4     REP MOVSB BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
004010E9 |. 33C9      XOR ECX,ECX                ; 清零，设循环计数器
004010EB |. BE 71214000  MOV ESI,cycle.00402171      ; 注册码送 ESI
004010F0 |> 41       INC ECX
004010F1 |. AC       LODS BYTE PTR DS:[ESI]      ; 取注册码的每个字符
004010F2 |. 0AC0      OR AL,AL                ; 判断是否为空
004010F4 |. 74 0A     JE SHORT cycle.00401100        ; 没有则跳走
004010F6 |. 3C 7E     CMP AL,7E                ; 判断字符是否为非 ASCII 字符
004010F8 |. 7F 06     JG SHORT cycle.00401100        ; 非 ASCII 字符跳走
004010FA |. 3C 30     CMP AL,30                ; 看是否小于 30H，主要是

```

判断是不是数字或字母等

```
004010FC |. 72 02      JB SHORT cycle.00401100      ; 小于跳走
004010FE |.^ EB F0      JMP SHORT cycle.004010F0
00401100 |> 83F9 11      CMP ECX,11      ; 比较注册码位数，必须为十进制 17 位
00401103 |. 75 1A      JNZ SHORT cycle.0040111F
00401105 |. E8 E7000000  CALL cycle.004011F1      ; 关键，F7 跟进去
0040110A |. B9 01FF0000  MOV ECX,0FF01
0040110F |. 51          PUSH ECX
00401110 |. E8 7B000000  CALL cycle.00401190      ; 关键，跟进去
00401115 |. 83F9 01      CMP ECX,1
00401118 |. 74 06      JE SHORT cycle.00401120
0040111A |> E8 47000000  CALL cycle.00401166      ; 注册失败对话框
0040111F |> C3          RETN
00401120 |> A1 68214000  MOV EAX,DWORD PTR DS:[402168]
00401125 |. 8B1D 6C214000  MOV EBX,DWORD PTR DS:[40216C]
0040112B |. 33C3        XOR EAX,EBX
0040112D |. 3305 82214000  XOR EAX,DWORD PTR DS:[402182]
00401133 |. 0D 40404040  OR EAX,40404040
00401138 |. 25 77777777  AND EAX,77777777
0040113D |. 3305 79214000  XOR EAX,DWORD PTR DS:[402179]
00401143 |. 3305 7D214000  XOR EAX,DWORD PTR DS:[40217D]
00401149 |.^ 75 CF      JNZ SHORT cycle.0040111A      ; 这里跳走就完蛋
0040114B |. E8 2B000000  CALL cycle.0040117B      ; 注册成功对话框
```

写到这准备跟踪算法时，才发现这个 crackme 还是挺复杂的，具体算法我就不写了，实在没那么多时间详细跟踪。有兴趣的可以跟一下，注册码是 17 位，用户名采用复制的方式扩展到 16 位，如我输入“CC Debugger”，扩展后就是“CCDebuggerCCDebug”。大致是先取扩展后用户名的前 8 位和注册码的

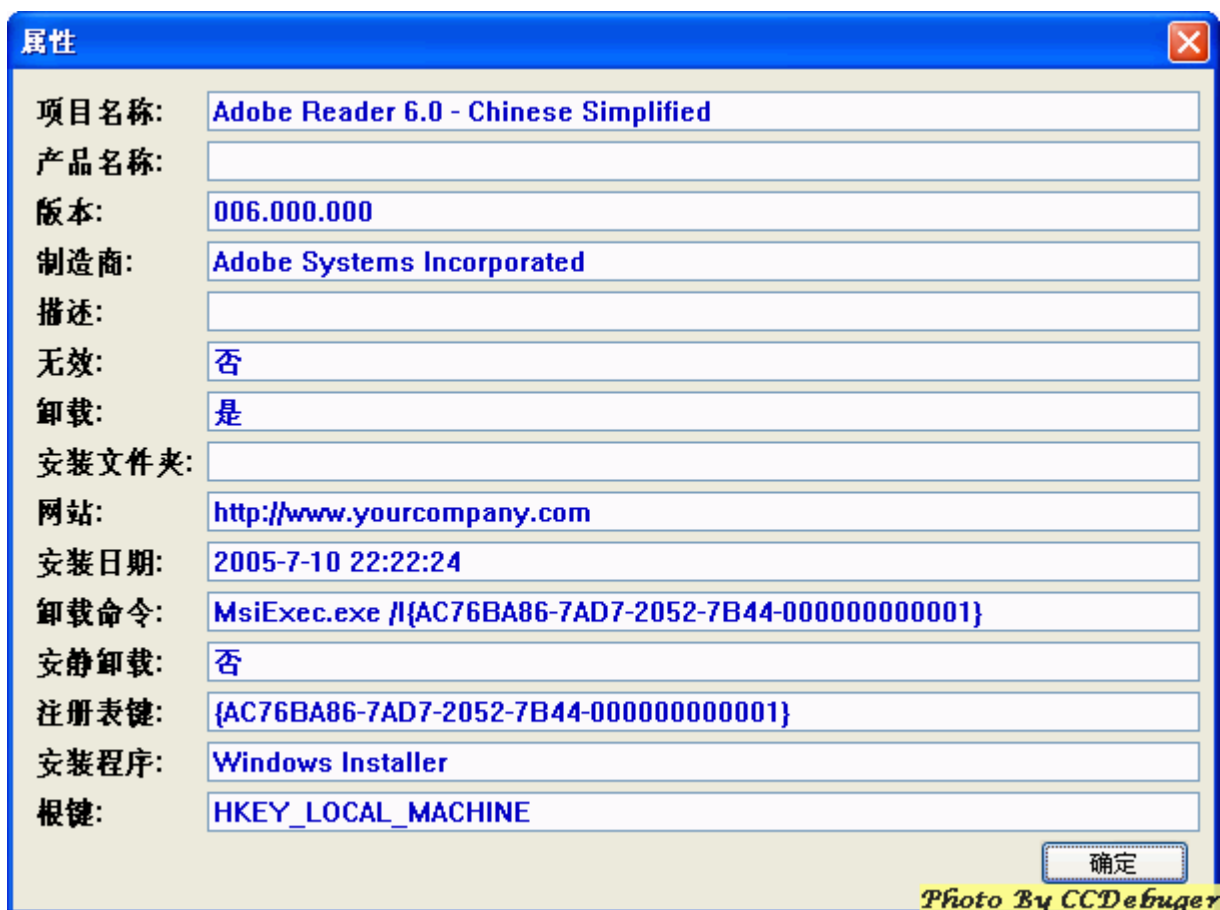
前 8 位，把用户名的前四位和后四位分别与注册码的前四位和后四位进行运算，算完后再把扩展后用户名的后 8 位和注册码的后 8 位分两部分，再与前面用户名和注册码的前 8 位计算后的值进行异或计算，最后结果等于 0 就成功。注册码的第 17 位我尚未发现有何用处。对于新手来说，可能这个 crackme 的难度大了一点。没关系，我们主要是学习 OllyDBG 的使用，方法掌握就可以了。

最后说明一下：

- 1、这个程序在设置了消息断点后可以省略在代码段上设访问断点那一步，直接打开 RUN 跟踪，消息断点断下后按 CTR+F12 组合键让程序执行，RUN 跟踪记录中就可以找到关键地方。
- 2、对于这个程序，你可以不设消息断点，在输入用户名和注册码后先不按那个“Check”按钮，直接打开 RUN 跟踪，添加“所有函数过程的入口”后再回到程序中断“Check”按钮，这时在 OllyDBG 中打开 RUN 跟踪记录同样可以找到关键位置。

汇编功能

今天我们的目标程序是 MyUninstaller 1.34 版。这是一个非常小的程序卸载工具，VC6 编写，大小只有 61K。我拿到的这个上次闪电狼兄弟给我的，附带在里面的简体中文语言文件是由六芒星制作的。这个程序有个毛病：就是在列出的可卸载程序上双击查看属性时，弹出的属性窗口的字体非常难看，应该就是系统字体（SYSTEM_FONT）：



我们今天的目标就是利用 OlllyDBG 的汇编功能把上面显示的字体改成我们常见的 9 号（小五）宋体。首先我们用 OlllyDBG 载入程序，按 CTR+N 组合键查找一下有哪些 API 函数，只发现一个和设置字体相关的 CreateFontIndirectA。现在我们按鼠标右键，选择“在每个参考上设置断点”，关掉名称对话框，F9 运行，程序已经运行起来了。我们在程序的列表框中随便找一项双击一下，很不幸，那个字体难看的界面又出现了，OlllyDBG 没有任何动作。可见创建这个窗口的时候根本没调用 CreateFontIndirectA，问题现在就变得有点复杂了。先点确定把这个字体难看的对话框关闭，现在我们从另一个方面考虑：既然没有调用设置字体的函数，那我们来看看这个窗口是如何创建的，跟踪窗口创建过程可能会找到一些对我们有用的信息。现在我们再回到我们调试程序的领空，按 CTR+N 看一下，发现 CreateWindowExA 这个 API 函数比较可疑。我们在 CreateWindowExA 函数的每个参考上设上断点，在 MyUninstaller 的列表框中再随便找一项双击一下，被 OlllyDBG 断下：

```
00408F5E |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowExA>] ; \断在这里
```

上下翻看一下代码：

```

00408F3B |. 50          |PUSH EAX                      ; |hInst
00408F3C |. 8B45 C0     |MOV EAX,DWORD PTR SS:[EBP-40] ; |
00408F3F |. 6A 00       |PUSH 0                        ; |hMenu = NULL
00408F41 |. 03C6        |ADD EAX,ESI                   ; |
00408F43 |. FF75 08     |PUSH DWORD PTR SS:[EBP+8]     ; |hParent
00408F46 |. FF75 D0     |PUSH DWORD PTR SS:[EBP-30]    ; |Height
00408F49 |. 57          |PUSH EDI                      ; |Width
00408F4A |. 50          |PUSH EAX                      ; |Y
00408F4B |. FF75 BC     |PUSH DWORD PTR SS:[EBP-44]    ; |X
00408F4E |. FF75 EC     |PUSH DWORD PTR SS:[EBP-14]    ; |Style
00408F51 |. 68 80DE4000 |PUSH myuninst.0040DE80       ; |WindowName = ""
00408F56 |. 68 DCD94000 |PUSH myuninst.0040D9DC       ; |Class = "STATIC"
00408F5B |. FF75 D4     |PUSH DWORD PTR SS:[EBP-2C]    ; |ExtStyle
00408F5E |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \断在这里
00408F64 |. 6A 00       |PUSH 0                        ; 第一处要修改的地方
00408F66 |. 8945 F4     |MOV DWORD PTR SS:[EBP-C],EAX
00408F69 |. E8 A098FFFF |CALL <myuninst.sub_40280E>
00408F6E |. 50          |PUSH EAX                      ; |hInst
00408F6F |. 8B45 DC     |MOV EAX,DWORD PTR SS:[EBP-24] ; |
00408F72 |. 6A 00       |PUSH 0                        ; |hMenu = NULL
00408F74 |. 03F0        |ADD ESI,EAX                   ; |
00408F76 |. FF75 08     |PUSH DWORD PTR SS:[EBP+8]     ; |hParent
00408F79 |. FF75 CC     |PUSH DWORD PTR SS:[EBP-34]    ; |Height
00408F7C |. 53          |PUSH EBX                      ; |Width
00408F7D |. 56          |PUSH ESI                      ; |Y
00408F7E |. FF75 D8     |PUSH DWORD PTR SS:[EBP-28]    ; |X
00408F81 |. FF75 E8     |PUSH DWORD PTR SS:[EBP-18]    ; |Style

```

```

00408F84 |. 68 80DE4000 |PUSH myuninst.0040DE80 ; |WindowN
ame = ""
00408F89 |. 68 D4D94000 |PUSH myuninst.0040D9D4 ; |Clas
s = "EDIT"
00408F8E |. FF75 B8 |PUSH DWORD PTR SS:[EBP-48] ; |ExtStyle
00408F91 |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
00408F97 | 8945 F0 |MOV DWORD PTR SS:[EBP-10],EAX ; 第二处要
修改的地方
00408F9A | 8B45 F8 |MOV EAX,DWORD PTR SS:[EBP-8]
00408F9D |. FF30 |PUSH DWORD PTR DS:[EAX] ; /<%s>
00408F9F |. 8D85 B0FEFFFF |LEA EAX,DWORD PTR SS:[EBP-150] ; |
00408FA5 |. 68 D0D94000 |PUSH myuninst.0040D9D0 ; |forma
t = "%s:"
00408FAA |. 50 |PUSH EAX ; |s
00408FAB |. FF15 90B14000 |CALL DWORD PTR DS:[<&MSVCRT.sprintf>] ; \
sprintf
00408FB1 |. 8B35 84B24000 |MOV ESI,DWORD PTR DS:[<&USER32.SetWindowText
A>] ; USER32.SetWindowTextA
00408FB7 |. 83C4 0C |ADD ESP,0C
00408FBA |. 8D85 B0FEFFFF |LEA EAX,DWORD PTR SS:[EBP-150]
00408FC0 |. 50 |PUSH EAX ; /Text
00408FC1 |. FF75 F4 |PUSH DWORD PTR SS:[EBP-C] ; |hWnd
00408FC4 |. FFD6 |CALL ESI ; \SetWindowTextA
00408FC6 |. 8D85 ACFAFFFF |LEA EAX,DWORD PTR SS:[EBP-554]
00408FCC |. 50 |PUSH EAX ; /Arg3
00408FCD |. FF75 FC |PUSH DWORD PTR SS:[EBP-4] ; |Arg2
00408FD0 |. FF35 00EF4000 |PUSH DWORD PTR DS:[40EF00] ; |Arg
1 = 00BEADCC
00408FD6 |. E8 1884FFFF |CALL <myuninst.sub_4013F3> ; \sub_40
13F3

```

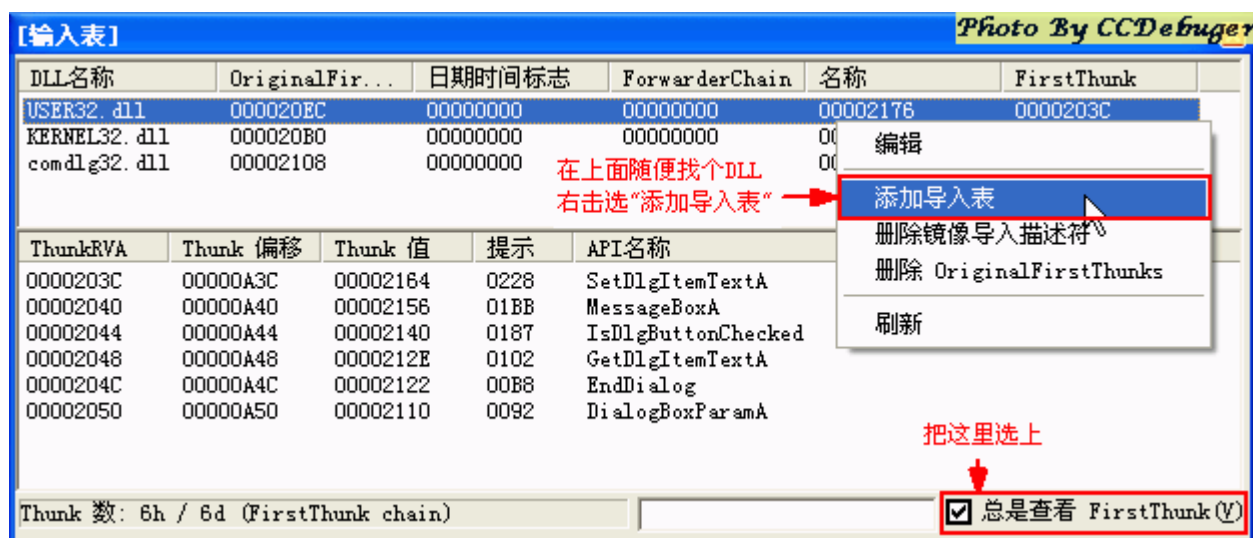
```

00408FDB |. 83C4 0C      |ADD ESP,0C
00408FDE |. 50            |PUSH EAX
00408FDF |. FF75 F0      |PUSH DWORD PTR SS:[EBP-10]
00408FE2 |. FFD6         |CALL ESI
00408FE4 |. FF45 FC      |INC DWORD PTR SS:[EBP-4]
00408FE7 |. 8345 F8 14    |ADD DWORD PTR SS:[EBP-8],14
00408FEB |. 837D FC 0F    |CMP DWORD PTR SS:[EBP-4],0F
00408FEF |.^ 0F8C 32FFFFFF \JL <myuninst.loc_408F27>
00408FF5 |. 5F           POP EDI
00408FF6 |. 5E           POP ESI
00408FF7 |. 5B           POP EBX
00408FF8 |. C9           LEAVE
00408FF9 \. C3         RETN

```

我想上面的代码我不需多做解释，OlllyDBG 自动给出的注释已经够清楚的了。我们双击 MyUninstall.r 列表框中的的某项查看属性时，弹出的属性窗口上的 **STATIC** 控件和 **EDIT** 控件都是由 **CreateWindowExA** 函数创建的，然后再调用 **SetWindowTextA** 来设置文本，根本没考虑控件上字体显示的问题，所以我们看到的都是系统默认的字体。我们要设置控件上的字体，可以考虑在 **CreateWindowExA** 创建完控件后，在使用 **SetWindowTextA** 函数设置文本之前调用相关字体创建函数来选择字体，再调用 **SendMessageA** 函数发送 **WM_SETFONT** 消息来设置控件字体。思路定下来后，我们就开始来实施。首先我们看一下这个程序中的导入函数，**CreateFontIndirectA** 这个字体创建函数已经有了，再看看 **SendMessageA**，呵呵，不错，原程序也有这个函数。这样我们就省事了。有人可能要问，如果原来并没有这两个导入函数，那怎么办呢？其实这也很简单，我们可以直接用 **LordPE** 来在程序中添加我们需要的导入函数。我这里用个很小的 **PE** 工具 **zeroadd** 来示范一下，这个程序里面没有 **CreateFontIndirectA** 和 **SendMessageA** 函数（这里还有个问题说一下，其实我们编程时调用这两个函数时都是直接写 **CreateFontIndirect** 及 **SendMessage**，一般不需指定。但在程序中写补丁代码时我们要指定这是什么类型的函数。这里在函数后面加个“A”表示这是 **ASCII** 版本，同样 **UNICODE** 版本在后面加个“W”，如 **SendMessageW**。在 **Win9X** 下我们一般都用 **ASCII** 版本的函数，**UNICODE** 版本的函数很多在 **Win9X** 下是不能运行的。而 **NT** 系统如 **WinXP** 一般都是 **UNICODE** 版本的，但如果我们用了 **ASCII** 版本的函数，系统会自动转换调用 **UNICODE** 版本。这样我们写补丁代码的时候就可以直接指定为 **ASCII** 版本的函数，可

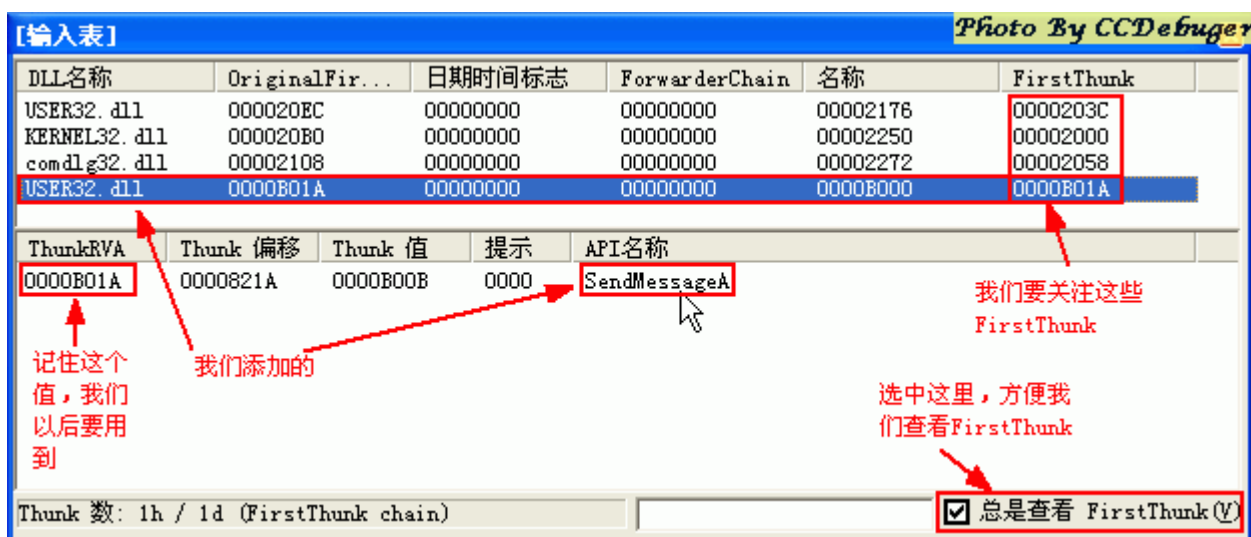
以兼容各个系统)：我们用 LordPE 的 PE 编辑器载入 zeroadd 程序，选择“目录”，再在弹出的目录表对话框中选择输入表后面的那个“...”按钮，会弹出一个对话框：



因为 SendMessageA 在 USER32.dll 中，我们在右键菜单中点击按钮“添加导入表”，来到下面：



按上面的提示完成后点“确定”，我们回到原先的那个“输入表”对话框：



从上图我们可以看出多出了一个 USER32.dll，这就是我们添加 SendMessageA 的结果。这也是用工具添加的一个缺点。我们一般希望把添加的函数直接放到已存在的 DLL 中，而不是多出来一个，这样显得不好看。但用工具就没办法，LordPE 默认是建一个 1K 的新区段来保存添加后的结果，由此出现了上图中的情况。如果你对 PE 结构比较熟悉的话，也可以直接用 16 进制编辑工具来添加你需要的函数，这样改出来的东西好看。如果想偷懒，就像我一样用工具吧，呵呵。在上图中我还标出了要注意 FirstThunk 及那个 ThunkRVA 的值，并且要把“总是查看 FirstThunk”那个选项选上。有人可能不理解其作用，我这里也解释一下：一般讲述 PE 格式的文章中对 FirstThunk 的解释是这样的：FirstThunk 包含指向一个 IMAGE_THUNK_DATA 结构数组的 RVA 偏移地址，当把 PE 文件装载到内存中时，PE 装载器将查找 IMAGE_THUNK_DATA 和 IMAGE_IMPORT_BY_NAME 这些结构数组来决定导入函数的地址，随后用导入函数真实地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值。这样说起来还是让人不明白，我举个例子：比如你有个很要好的朋友，他是个大忙人，虽然你知道他的家庭住址，可他很少回家。如果你哪天想找他，直接去他家，很可能吃个闭门羹，找不到他人。怎么办？幸好你有他的手机号码，你就给他拨了一个电话：“小子，你在哪呢？”，他告诉你：“我正在 XXX 饭店喝酒呢！”这时你怎么办？（当然是杀到他说的那家饭店去蹭饭了！^_^）这里的 ThunkRVA 就相当于你朋友的手机号码，SendMessageA 就相当于你那个朋友。而 FirstThunk 就是你手机里的号码分组。你把你的多个朋友都放在 FirstThunk 这样的号码分组里，每个 ThunkRVA 就是你一个朋友的手机号码。你要找他们，就是通过 ThunkRVA 这样的手机号码来和他们联系，直接去他家找他你很可能要碰壁。而移动或联通就相当于操作系统，他们负责把你的手机号码和你的朋友对应上。而 FirstThunk 这样的号码分组还有一个好处就是你可以不记你某个朋友的具体号码，只要记得 FirstThunk 号码分组的值，你的朋友会按顺序在里面排列。比如上图中 USER32.dll 中的第一个函数是 SendMessageA，它的 ThunkRVA 值就是 FirstThunk 值。如果还有第二个函数，比如是 MessageBoxA，它的值就是 FirstThunk 值加上 4，其余

类推。你只要记住各个函数的位置，也可以通过 FirstThunk 加上位置对应值来找到它。当然这比不上直接看 ThunkRVA 来得方便。说了上面这些，我们就要考虑怎么在程序中调用了。你可能会说，我在 OllyDBG 中直接在我们要修改的程序中这样调用：CALL SendMessageA。哦，别这样。这等于我上面说的都是废话，会让我感到伤心的。你这里的 CALL SendMessageA 就相当于也不跟你朋友打个招呼就直接去他家找他，很有可能你会乘兴而去，败兴而归。别忘了他的手机号码，我们只有通过号码才知道他到底在什么地方。我们应该这样：CALL DWORD PTR [40B01A]，这里的 40B01A 就是上面的 SendMessageA 在程序载入后的所在的地方，由基址 00400000 加上 ThunkRVA 0000B01A 得到的。这就是你要找的人所在的地方，不管他跑到哪，你有他的手机号码就能找到他。同样道理，你只要记住了 ThunkRVA 值，就按这个来调用你需要的函数，在别的 Windows 系统下也是没有问题的。系统会自动把你找到函数和 ThunkRVA 值对应上。而你在 OllyDBG 中写 CALL SendMessageA，可能你在你的系统上成功了，可放到别的系统下就要出错了。为什么？因为你找的人已经不在原来的位置了，他跑到别的地方去了。你还到老地方找他，当然看不见人了。说了这么多废话，也不知大家听明白了没有，别越听越糊涂就行了。总之一句话，别像 CALL SendMessageA 这样直接调用某个函数，而应该通过 ThunkRVA 值来调用它。下面我们回到我们要修改的 MyUninstaller 上来，先用 LordPE 打开看一下，呵呵，原来 CreateFontIndirectA 和 SendMessageA 原程序里面都有了，省了我们不少事情。看一下这两个函数的 ThunkRVA 值，CreateFontIndirectA 在 GDI32.dll 里面，ThunkRVA 值是 0000B044，这样我们就知道在程序中调用它的时候就是 CALL DWORD PTR [0040B044]。同样，SendMessageA 的 ThunkRVA 值是 0000B23C，调用时应该是这样：CALL DWORD PTR [0040B23C]。了解了这些东西我们就来考虑怎么写代码了。首先我们来看一下 CreateFontIndirectA 和 SendMessageA 这两个函数的定义：

CreateFontIndirectA:

```
HFONT CreateFontIndirect(
```

```
CONST LOGFONT *lpLf // pointer to logical font structure
```

```
);
```

CreateFontIndirect 的返回值就是字体的句柄。

对于这个函数我们需要的参数就是给它一个 LOGFONT 的字体结构指针，我们只要要在要修改程序的空白处建一个标准的 9 号（小五）宋体的 LOGFONT 字体结构，再把指针给 CreateFontIndirectA 就可以了。

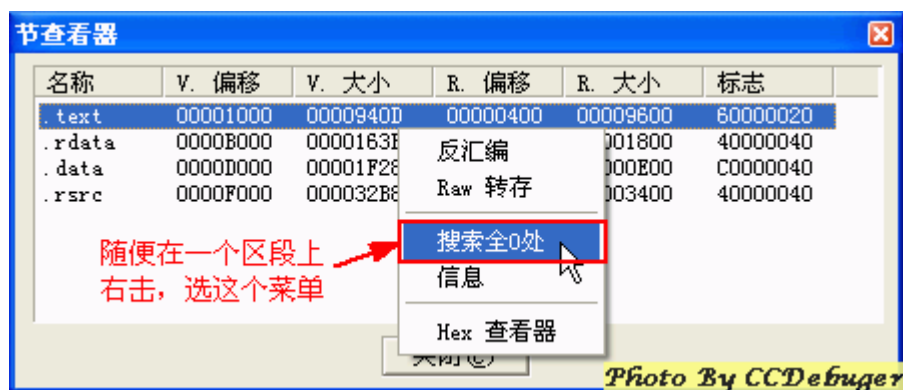
SendMessageA:

```
LRESULT SendMessage(  
    HWND hWnd, // handle of destination window  
    UINT Msg, // message to send  
    WPARAM wParam, // first message parameter  
    LPARAM lParam // second message parameter  
);
```

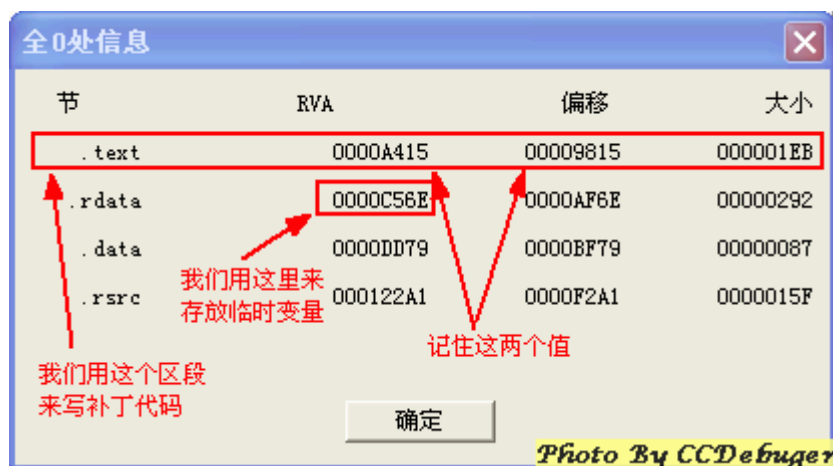
上面的第一个参数是窗口句柄，我们知道 **CreateWindowExA** 返回的就是窗口句柄，我们可以直接拿来用。第二个消息参数我们这里是设置字体，选 **WM_SETFONT**，这个值是 **30H**。第三个参数是字体句柄，可以由上面的 **CreateFontIndirectA** 获得。第四个参数我们不需要，留空。现在我们准备开始写代码，首先我们要在程序中建一个标准 9 号宋体的 **LOGFONT**，以便于我们调用。对于 **LOGFONT**，我们再来看一下定义：

```
typedef struct tagLOGFONT { // If  
    LONG lfHeight;  
    LONG lfWidth;  
    LONG lfEscapement;  
    LONG lfOrientation;  
    LONG lfWeight;  
    BYTE lfItalic;  
    BYTE lfUnderline;  
    BYTE lfStrikeOut;  
    BYTE lfCharSet;  
    BYTE lfOutPrecision;  
    BYTE lfClipPrecision;  
    BYTE lfQuality;  
    BYTE lfPitchAndFamily;  
    TCHAR lfFaceName[LF_FACESIZE];  
} LOGFONT;
```

这样我们的标准 9 号宋体的 LOGFONT 值应该是 32 字节，16 进制就像这样：F4FFFFFF000000000000000000000000900100000000008600000000CBCECCE5。现在在程序中找个空地。我们用 PE iD 来帮助我们寻找，用 PEiD 打开程序，点 EP 段后面的那个 > 号，随便选择一个区段右击，选“搜索全 0 处”（原版好像是 cave 什么的）：



我们看到 PEiD 把搜索到的空间都给我们列出来了：



现在我们用 WinHEX 打开我们要修改的程序，转到偏移 9815 处，从 9815 处选择 32 字节（16 进制是 0X20）的一个选块，把光标定位到 9815 处，右键选择菜单 剪贴板数据->写入(从当前位置覆写)，随后的格式选择 ASCII Hex，把我们 LOGFONT 的 16 进制值

F4FFFFFF000000000000000000000000900100000000008600000000CBCECCE5

写入保存。现在我们用 OllyDBG 载入已添加了 LOGFONT 数据的程序，先转到 VA 40A415 处（从上图看到的）往下看一下：

0040A431	CB	DB CB	
0040A432	CE	DB CE	
0040A433	CC	INT3	
0040A434	E5	DB E5	
0040A435	00	DB 00	
0040A436	00	DB 00	
0040A437	00	DB 00	
0040A438	00	DB 00	
0040A439	00	DB 00	
0040A43A	00	DB 00	前面留点空间，就从这里开始写补丁代码吧
0040A43B	00	DB 00	
0040A43C	00	DB 00	
0040A43D	00	DB 00	
0040A43E	0000	ADD BYTE PTR DS:[EAX], AL	就从这里开始写补丁代码吧
0040A440	0000	ADD BYTE PTR DS:[EAX], AL	
0040A442	0000	ADD BYTE PTR DS:[EAX], AL	

Photo By CCDebugger

因为 SendMessageA 还要用到一个窗口句柄，我们可以通过前面的 CreateWindowExA 来获得。现在我们就把前一张图中的 .rdata 区段中的地址 0040C56E 作为我们保存窗口句柄 HWND 值的临时空间。一切就绪，开始写代码。先回顾一下我们最先说的那两个要修改的地方：

第一个要改的地方：

```

00408F5E |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
00408F64    6A 00    PUSH 0                                ; 修改前
00408F66    8945 F4    MOV DWORD PTR SS:[EBP-C],EAX
00408F69 |. E8 A098FFFF |CALL <myuninst.sub_40280E>

```

修改后：

```

00408F5E |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
00408F64    E9 D5140000 JMP myuninst.0040A43E                ; 跳转到我们的补丁代码处
00408F69 |. E8 A098FFFF |CALL <myuninst.sub_40280E>

```

第二个要改的地方:

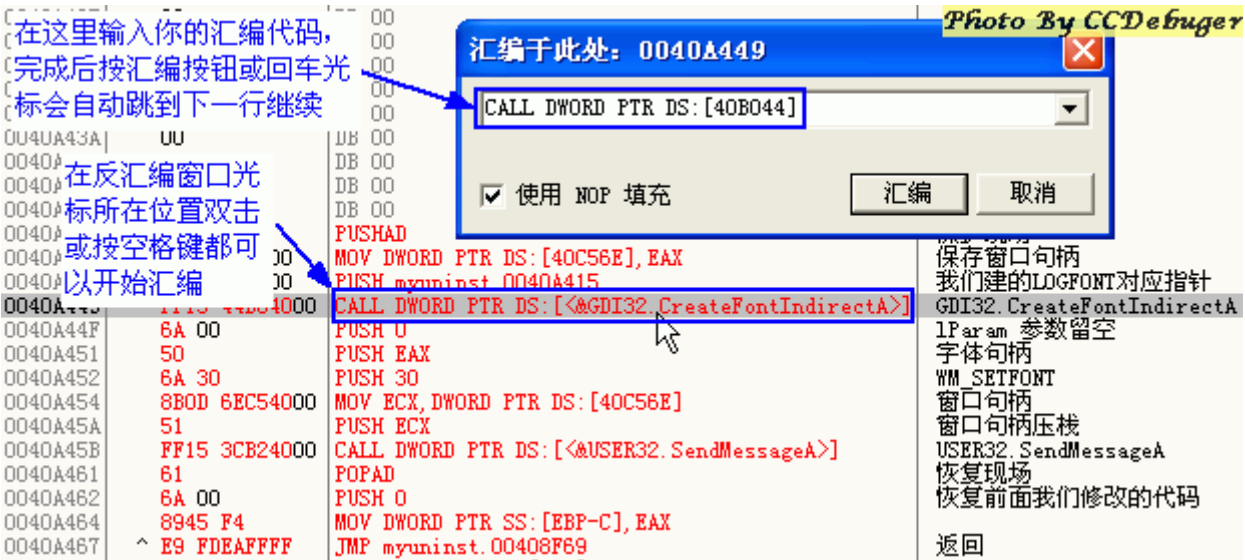
```
00408F91 |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
00408F97 8945 F0      MOV DWORD PTR SS:[EBP-10],EAX ; 改这里
00408F9A 8B45 F8      MOV EAX,DWORD PTR SS:[EBP-8]
00408F9D |. FF30      |PUSH DWORD PTR DS:[EAX] ; /<%s>
00408F9F |. 8D85 B0FEFFFF |LEA EAX,DWORD PTR SS:[EBP-150] ; |
00408FA5 |. 68 D0D94000 |PUSH myuninst.0040D9D0 ; |forma
t = "%s:"
00408FAA |. 50      |PUSH EAX ; |s
00408FAB |. FF15 90B14000 |CALL DWORD PTR DS:[<&MSVCRT.sprintf>] ; \
sprintf
00408FB1 |. 8B35 84B24000 |MOV ESI,DWORD PTR DS:[<&USER32.SetWindowText
A>] ; USER32.SetWindowTextA
```

修改后:

```
00408F91 |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
00408F97 E9 D4140000   JMP myuninst.0040A470 ; 跳到我们的
第二部分补丁代码处
00408F9C 90      NOP
00408F9D |. FF30      |PUSH DWORD PTR DS:[EAX] ; /<%s>
00408F9F |. 8D85 B0FEFFFF |LEA EAX,DWORD PTR SS:[EBP-150] ; |
00408FA5 |. 68 D0D94000 |PUSH myuninst.0040D9D0 ; |forma
t = "%s:"
00408FAA |. 50      |PUSH EAX ; |s
00408FAB |. FF15 90B14000 |CALL DWORD PTR DS:[<&MSVCRT.sprintf>] ; \
sprintf
00408FB1 |. 8B35 84B24000 |MOV ESI,DWORD PTR DS:[<&USER32.SetWindowText
```

A>] ; USER32.SetWindowTextA

这两个地方的修改都是把原代码改成跳转，跳到我们的补丁代码那继续执行。在修改之前先把原代码复制下来，以便恢复。我们在 OllyDBG 中按 CTR+G 组合键，来到 0040A43E 地址处，开始输补丁代码：



同样，我们也在 0040A470 地址处输入我们另一部分的补丁代码。两部分的补丁代码分别如下：

补丁代码 1：

0040A43E	60	PUSHAD	； 保护现场
0040A43F	A3 6EC54000	MOV DWORD PTR DS:[40C56E],EAX	； 保存窗口句柄
0040A444	68 15A44000	PUSH myuninst.0040A415	； 传递字体句柄 LOGFONT
0040A449	FF15 44B04000	CALL DWORD PTR DS:[<&GDI32.CreateFontIndirectA>]	； GDI32.CreateFontIndirectA
0040A44F	6A 00	PUSH 0	； lParam 参数留空
0040A451	50	PUSH EAX	； 字体句柄 LOGFONT
0040A452	6A 30	PUSH 30	； WM_SETFONT
0040A454	8B0D 6EC54000	MOV ECX,DWORD PTR DS:[40C56E]	； 窗口句柄送 ECX


```

0040A45A  51          PUSH ECX                      ; 压入窗口句柄参数
0040A45B  FF15 3CB24000  CALL DWORD PTR DS:[<&USER32.SendMessage
A>]      ; USER32.SendMessageA
0040A461  61          POPAD                          ; 恢复现场
0040A462  6A 00       PUSH 0                          ; 恢复原代码
0040A464  8945 F4     MOV DWORD PTR SS:[EBP-C],EAX
0040A467  ^ E9 FDEAFFFF  JMP myuninst.00408F69            ; 返回

```

补丁代码 2:

```

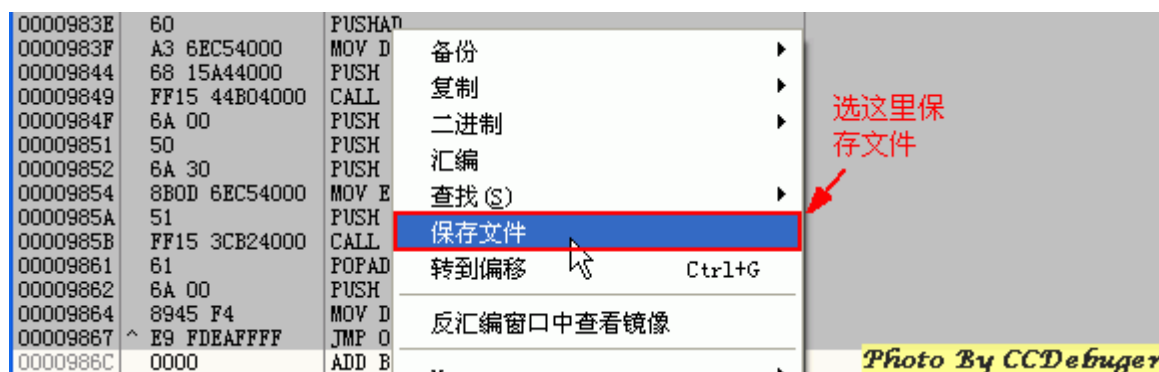
0040A470  > \60      PUSHAD
0040A471  . A3 6EC54000  MOV DWORD PTR DS:[40C56E],EAX
0040A476  . 68 15A44000  PUSH myuninst.0040A415          ; /pLogfon
t = myuninst.0040A415
0040A47B  . FF15 44B04000  CALL DWORD PTR DS:[<&GDI32.CreateFontIndirect
A>]      ; \CreateFontIndirectA
0040A481  . 6A 00       PUSH 0                          ; /iParam = 0
0040A483  . 50          PUSH EAX                      ; |iParam
0040A484  . 6A 30       PUSH 30                     ; |Message = WM_SET
FONT
0040A486  . 8B0D 6EC54000  MOV ECX,DWORD PTR DS:[40C56E]    ; |
0040A48C  . 51          PUSH ECX                      ; |hWnd => NULL
0040A48D  . FF15 3CB24000  CALL DWORD PTR DS:[<&USER32.SendMessage
A>]      ; \SendMessageA
0040A493  . 61          POPAD
0040A494  . 8945 F0     MOV DWORD PTR SS:[EBP-10],EAX
0040A497  . 8B45 F8     MOV EAX,DWORD PTR SS:[EBP-8]
0040A49A  . ^ E9 FEEAFFFF  JMP myuninst.00408F9D

```

补丁代码 2 因为与补丁代码 1 类似，我就不做详细解释了。现在我们的代码都写完了，现在我们开始保存我们的工作，选中我们修改的代码，点击鼠标右键，会出来一个菜单：



我们左键选所有修改（当然选它了，要不然只会保存我们选定的这一部分。关于这个地方还要说一下，有的时候我们修改完程序选“复制到可执行文件”时只有“选择”菜单，没有“所有修改”菜单项。按 OllyDBG 帮助里关于备份功能的说法，好像是受内存块限制的，补丁功能也同样是这样的。对于备份及补丁功能我用的比较少，并不是很了解，这方面的内容还是大家自己去研究吧，有什么好的心得也希望能共享一下。我遇到不能保存所有修改的情况就是先把补丁代码全部复制下来，同时利用二进制功能复制代码，先选一段补丁代码保存为文件，再用 OllyDBG 打开保存后的文件，转到相应位置分别把我们复制下来的补丁二进制代码粘贴上去后保存。纯属笨办法，当然你也可以用 HexView 这样的工具来修改代码），随后会出来一个“把选中的内容复制到可执行文件”的对话框，我们选“全部复制”，又出来一个对话框，我们在上面点右键，在弹出的菜单上选“保存文件”：

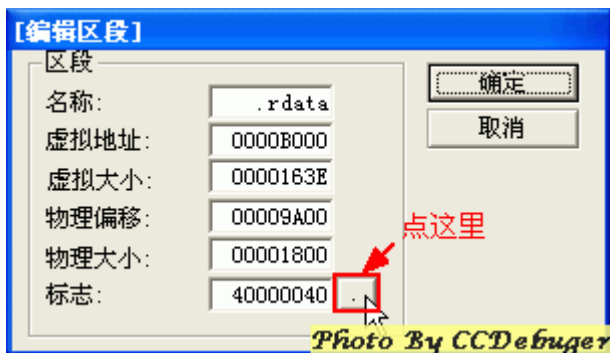


这时会出来一个另存文件的对话框，我们另选一个名字如 myuninst1.exe 来保存，不要直接覆盖原文

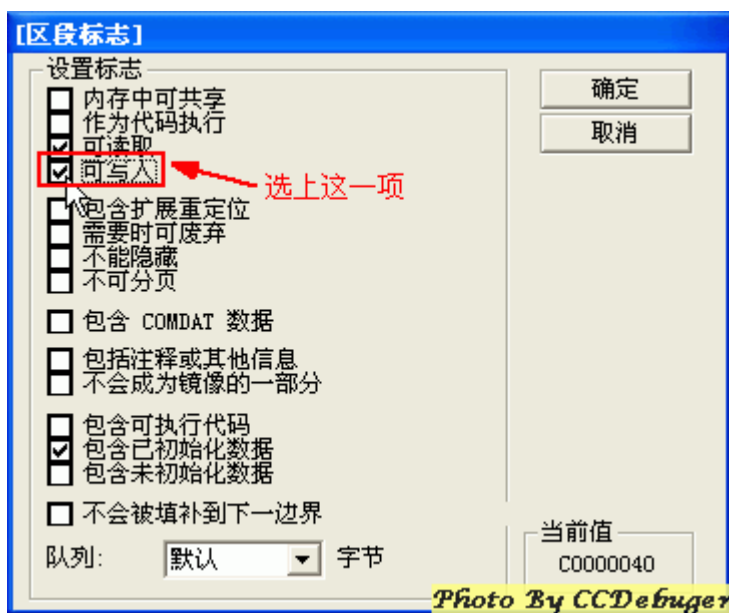
件 myuninst.exe,以便于出错后好修改。现在关闭 OllyDBG,先不要急着运行刚刚修改过的文件,因为我们还有个地方要改一下。大家还记得我们在 .rdata 中用了个地方作为我们保存临时变量的地方吧?原先的 .rdata 段属性设置是不可写的,现在我们要修改一下 .rdata 段的属性。用 LordPE 的 PE 编辑器打开我们修改后的程序,点“区段”按钮,在弹出的对话框中点击 .rdata 段,右键选择弹出菜单中的“编辑区段”:



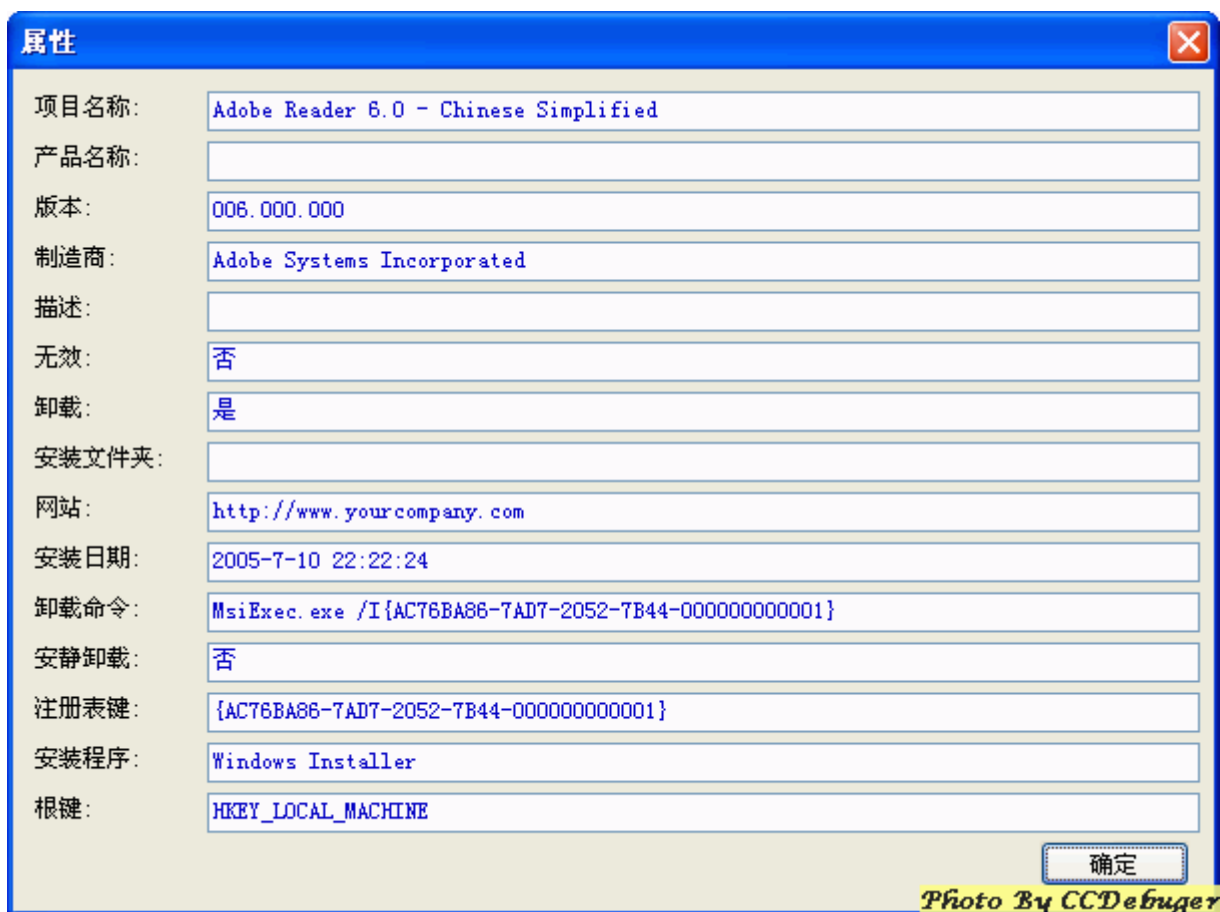
在弹出的对话框中选标志后面那个“...”按钮:



现在我们把区段标志添加一个可写入的属性:



完成后按确定保存我们所做的工作，运行一下修改后的程序，呵呵，终于把字体改过来了：



如果你运行出错也没关系，用 OlllyDBG 调试一下你修改后的程序，看看错在什么地方。这一般都是输入补丁代码时造成的，你只要看一下你补丁代码运行的情况就可以了。到这里我们的任务似乎也完成了，但

细心的朋友可能会发现补丁代码 1 和补丁代码 2 前面的代码基本上是相同的。一个两个这样的补丁还好，如果要是多的话，这样重复就要浪费不少空间了，况且工作量也相应加大了。既然前面有很多代码都是重复的，为什么我们不把这些重复的代码做成一个子程序呢？这样调用起来要方便的多。下面我们把前面的补丁代码修改一下，我们先把补丁代码 1 的代码改成这样：

```
0040A43E    60          PUSHAD                      ; 保护现场
0040A43F    A3 6EC54000  MOV DWORD PTR DS:[40C56E],EAX      ; 保存
窗口句柄
0040A444    68 15A44000  PUSH myuninst.0040A415             ; 我们建的 L
OGFONT 对应指针
0040A449    FF15 44B04000 CALL DWORD PTR DS:[<&GDI32.CreateFontIndirect
A>] ; GDI32.CreateFontIndirectA
0040A44F    6A 00        PUSH 0                          ; IParam 参数留空
0040A451    50          PUSH EAX                       ; 字体句柄
0040A452    6A 30        PUSH 30                        ; WM_SETFONT
0040A454    8B0D 6EC54000 MOV ECX,DWORD PTR DS:[40C56E]      ; 窗
口句柄
0040A45A    51          PUSH ECX                       ; 窗口句柄压栈
0040A45B    FF15 3CB24000 CALL DWORD PTR DS:[<&USER32.SendMessage
A>] ; USER32.SendMessageA
0040A461    61          POPAD                          ; 恢复现场
0040A462    C3          RETN                          ; 返回
```

这样我们的子程序代码就写好了。现在我们在子程序代码后面写上两个补丁代码，当然不要忘了改前面原程序中的跳转：

修改后的补丁代码 1：

```
0040A467    E8 D2FFFFFF  CALL myuninst.0040A43E             ; 调用子程序
0040A46C    6A 00        PUSH 0                          ; 恢复前面修改过的代码
0040A46E    8945 F4      MOV DWORD PTR SS:[EBP-C],EAX
```

```
0040A471 ^ E9 F3EAFFFF JMP myuninst.00408F69 ; 返回继续执行
```

修改后的补丁代码 2:

```
0040A47A E8 BFFFFFFF CALL myuninst.0040A43E
0040A47F 8945 F0      MOV DWORD PTR SS:[EBP-10],EAX
0040A482 8B45 F8      MOV EAX,DWORD PTR SS:[EBP-8]
0040A485 ^ E9 13EBFFFF JMP myuninst.00408F9D
```

我在每个补丁代码片段间留了 4 个字节来分隔。同样，我们还要修改一下我们前面的跳转：

第一个要修改跳转的地方：

```
00408F5E |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \断在这里
```

```
00408F64 E9 FE140000 JMP myuninst.0040A467 ; 跳到我们的
第一部分补丁代码处
```

```
00408F69 |. E8 A098FFFF |CALL <myuninst.sub_40280E>
```

第二个要修改跳转的地方：

```
00408F91 |. FF15 98B24000 |CALL DWORD PTR DS:[<&USER32.CreateWindowEx
A>] ; \CreateWindowExA
```


```
00408F97 E9 DE140000 JMP myuninst.0040A47A ; 跳到我们的
第二部分补丁代码处
```

```
00408F9C 90          NOP
```

```
00408F9D |. FF30      |PUSH DWORD PTR DS:[EAX] ; /<%s>
```

修改好后保存，同样不要忘了再修改一下 .rdata 区段的属性。运行一下，一切 OK！

原作者：[CCDebugger](#)(再次感谢 CCDebugger)

上傳者后記：再次感谢作者 CCDebugger !!!希望對您能起到幫助，如果你需要
文中所涉及的軟件（包括 01lyDBG 以及相關的 crackme 等，可與
我聯系，我將盡快發給你，聯系 [QQ: 20003138](#)，謝謝！）