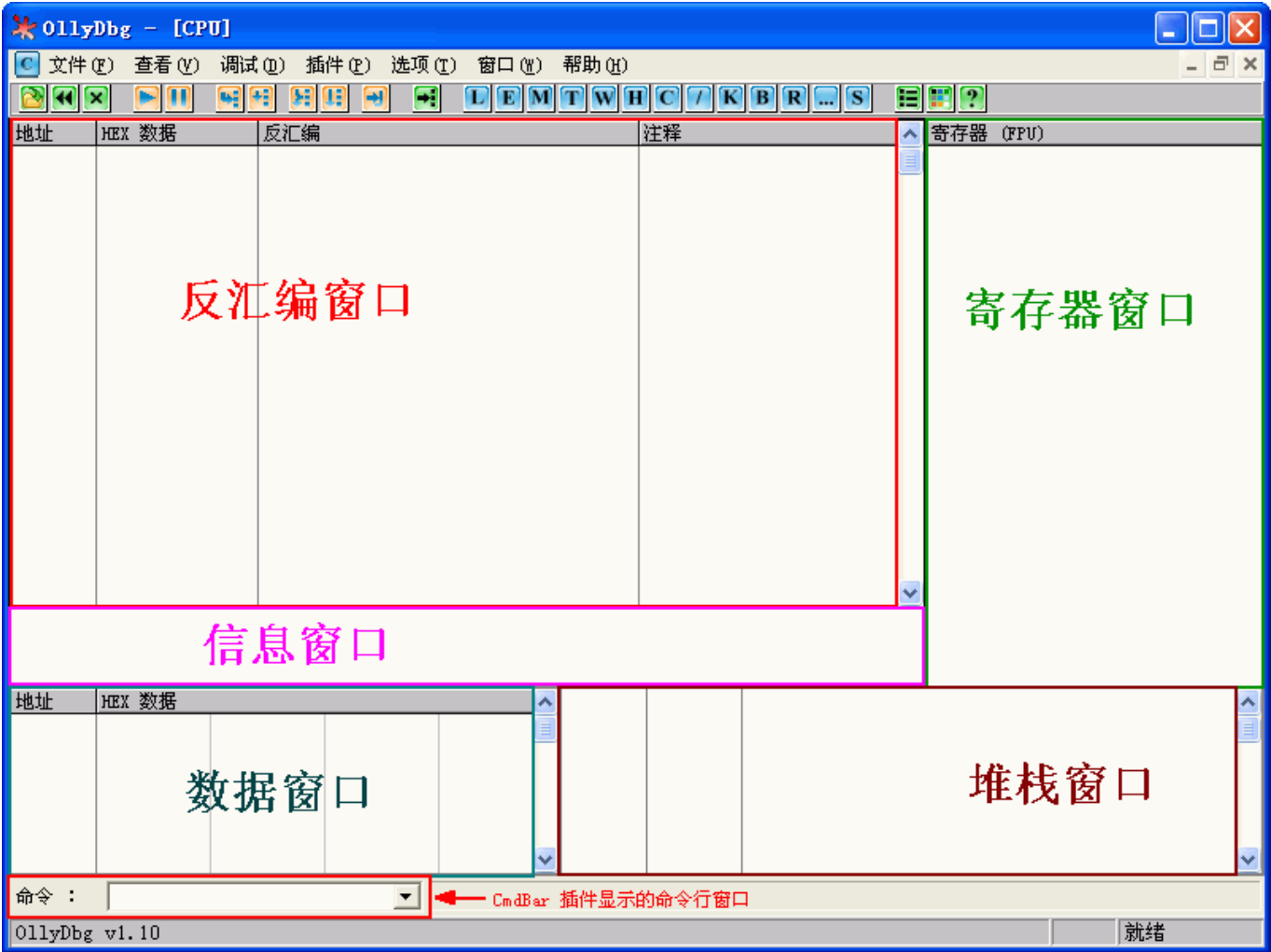


OlllyDBG 入门系列（一）—认识 OlllyDBG

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。

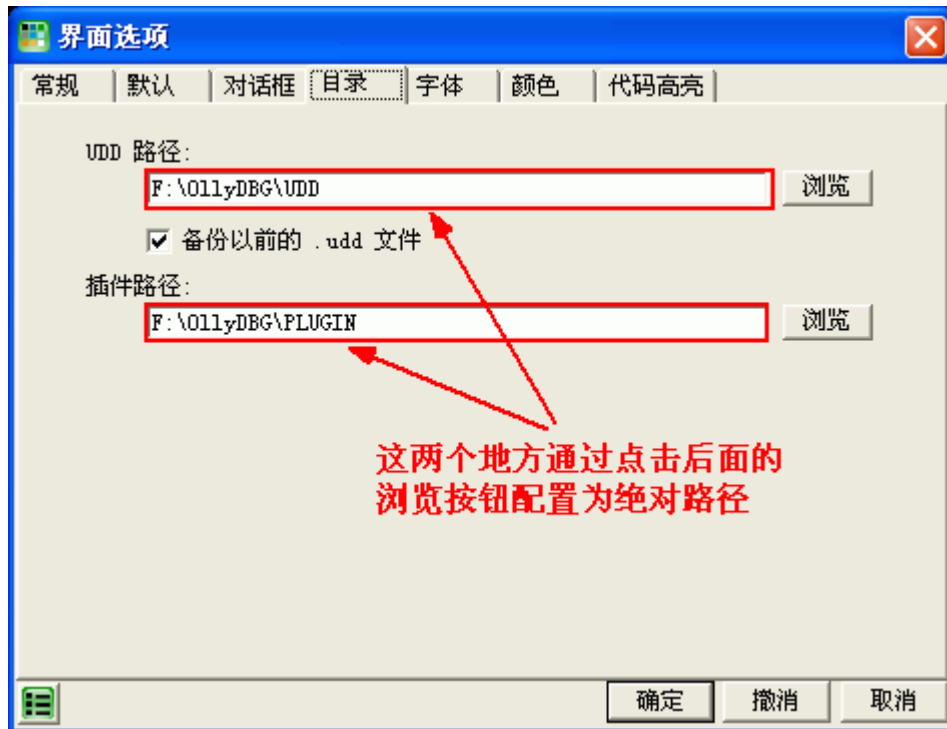
信息窗口：显示反汇编窗口中选中的第一个命令的参数及一些跳转目标地址、字串等。

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。

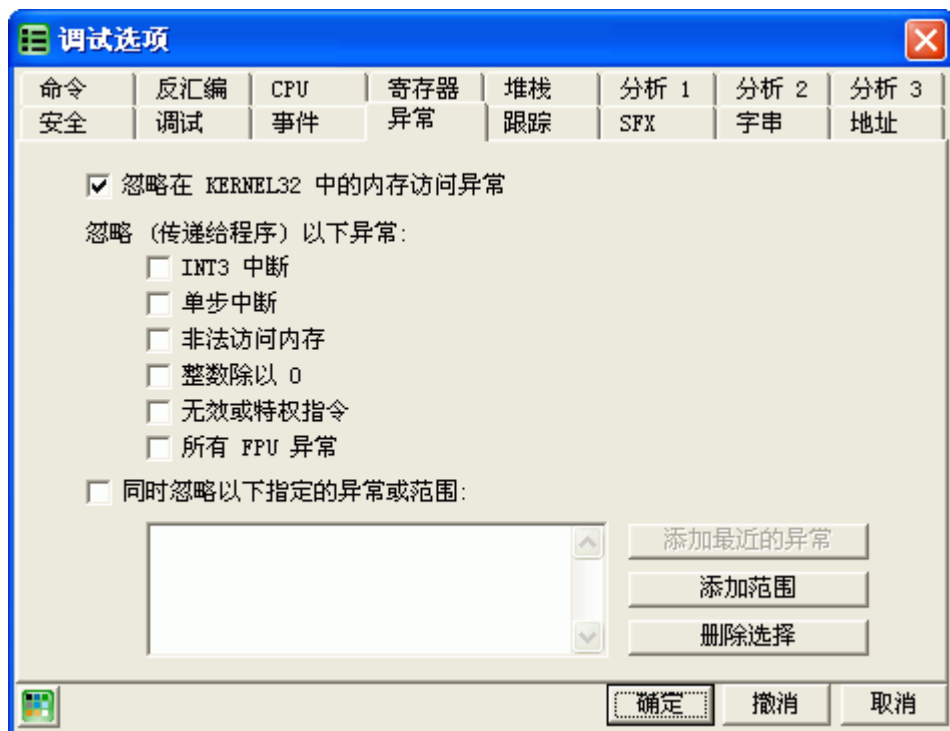
要调整上面各个窗口的大小的话，只需左键按住边框拖动，等调整好了，重新启动一下 OllyDBG 就可以生效了。

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。

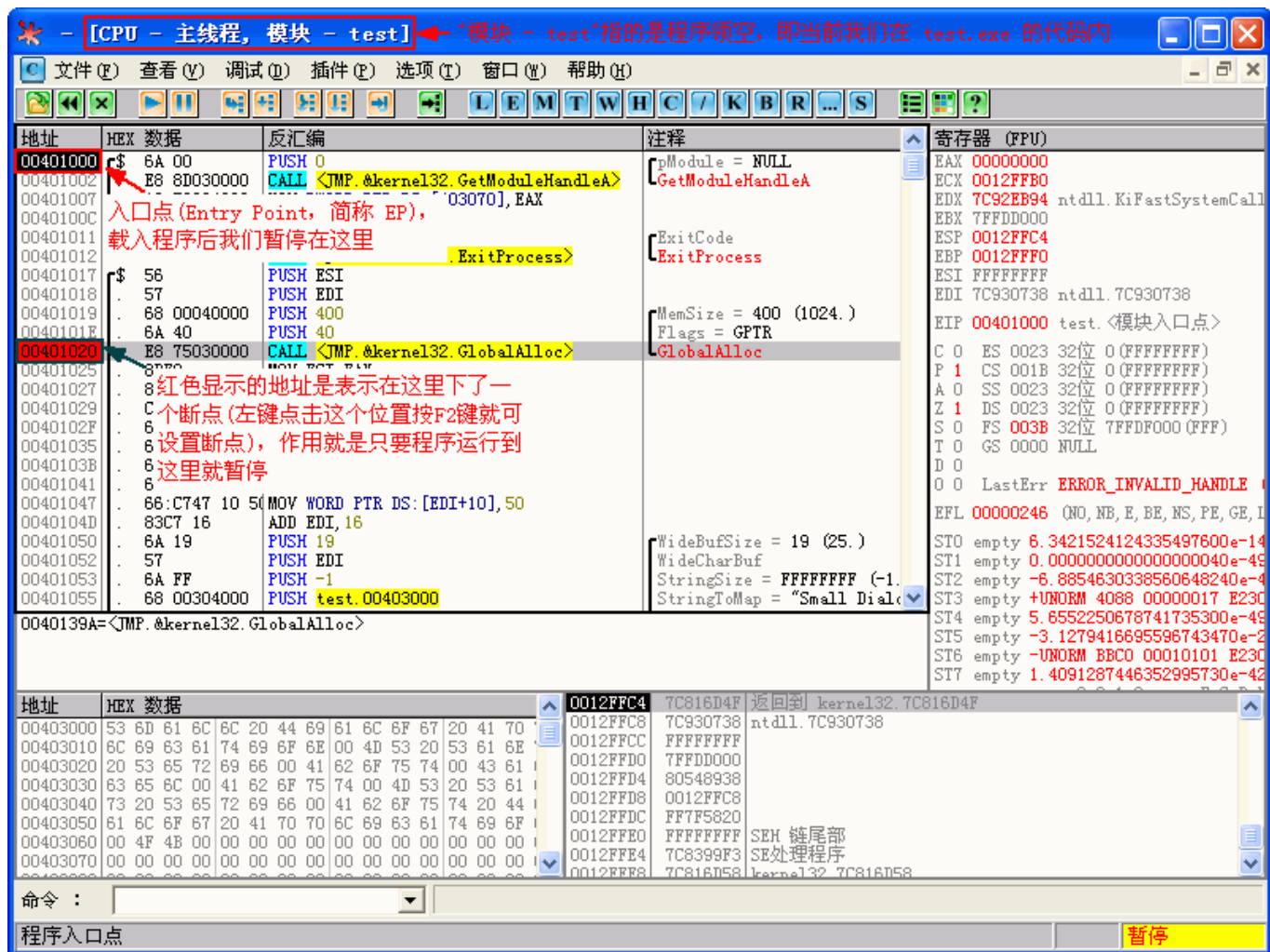
除了直接启动 OllyDBG 来调试外，我们还可以把 OllyDBG 添加到资源管理器右键菜单，这样我们就可以直接在 .exe 及 .dll 文件上点右键选择“用 Ollydbg 打开”菜单来进行调试。要把 OllyDBG 添加到资源管理器右键菜单，只需点菜单 选项->添加到浏览器，将会出现一个对话框，先点击“添加 Ollydbg 到系统资源管理器菜单”，再点击“完成”按钮即可。要从右键菜单中删除也很简单，还是这个对话框，点击“从系统资源管理器菜单删除 Ollydbg”，再点击“完成”就行了。

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。

OlllyDBG 入门系列（二）一字串参考

引用：

感谢 chuxuezhe 朋友的反馈：

<http://bbs.pediy.com/showthread.php?s=&threadid=24703>

经检查才发现原来是写文章前曾用修改过的 Ultra String Reference 插件查找过字串，这个修改后的插件会把找到的字串自动添加到代码后面作为注释，且所有字母都一律小写，导致原来文章写的时候注释中的大小写分不清楚，比较混乱。这次把文章一些地方修改了一下，全部用 OD 自带功能进行操作，重新制作了几个图片。因为我自己的失误，在此对大家造成了阅读中的困惑表示抱歉！

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 OlllyDBG 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 OlllyDBG 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 OlllyDBG 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：

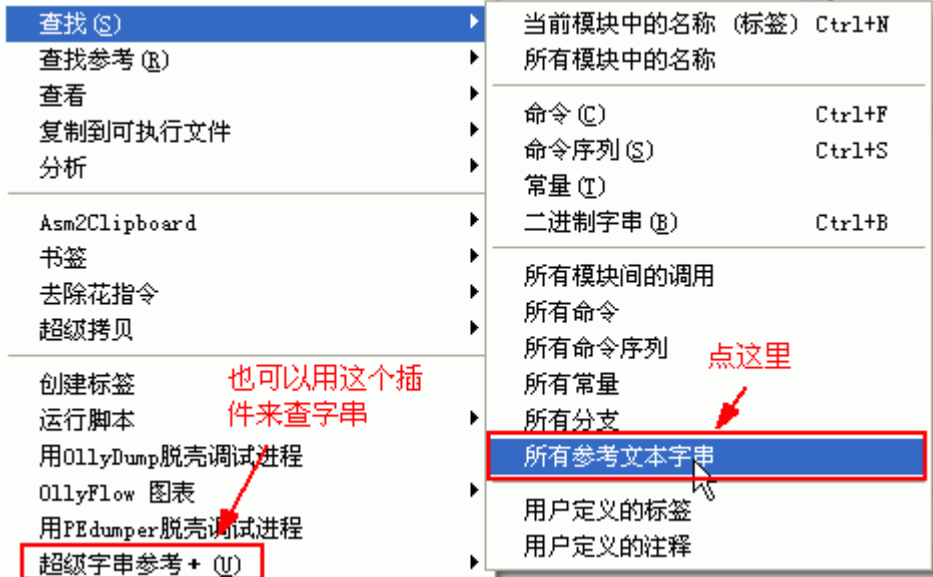


好了，今天我们就从这个错误对话框中显示的“Wrong Serial, try again!”来入手。启动 OIlyDBG，选择菜单 文件 ->打开 载入 CrackMe3.exe 文件，我们会停在这里：

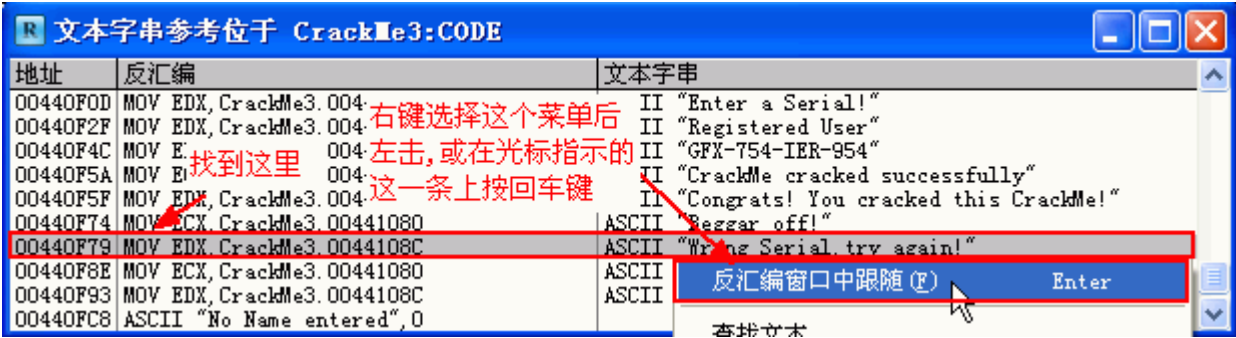
地址	HEX 数据	反汇编	注释
00441270	\$ 55	PUSH EBP	载入后停在这里
00441271	. 8BEC	MOV EBP, ESP	
00441273	. 83C4 F4	ADD ESP, -0C	
00441276	. B8 60114400	MOV EAX, CrackMe3.00441160	
0044127B	. E8 E848FCFF	CALL CrackMe3.00405B68	
00441280	. A1 442C4400	MOV EAX, DWORD PTR DS:[442C44]	
00441285	. 8B00	MOV EAX, DWORD PTR DS:[EAX]	
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 EDI, CrackMe3.004412D0	
00441298	. E8 17B8FFFF	CALL CrackMe3.0043CAB4	

在上面代码后的注释栏中双击或按分号 (; 英文输入方式) 可以输入如图所示的注释

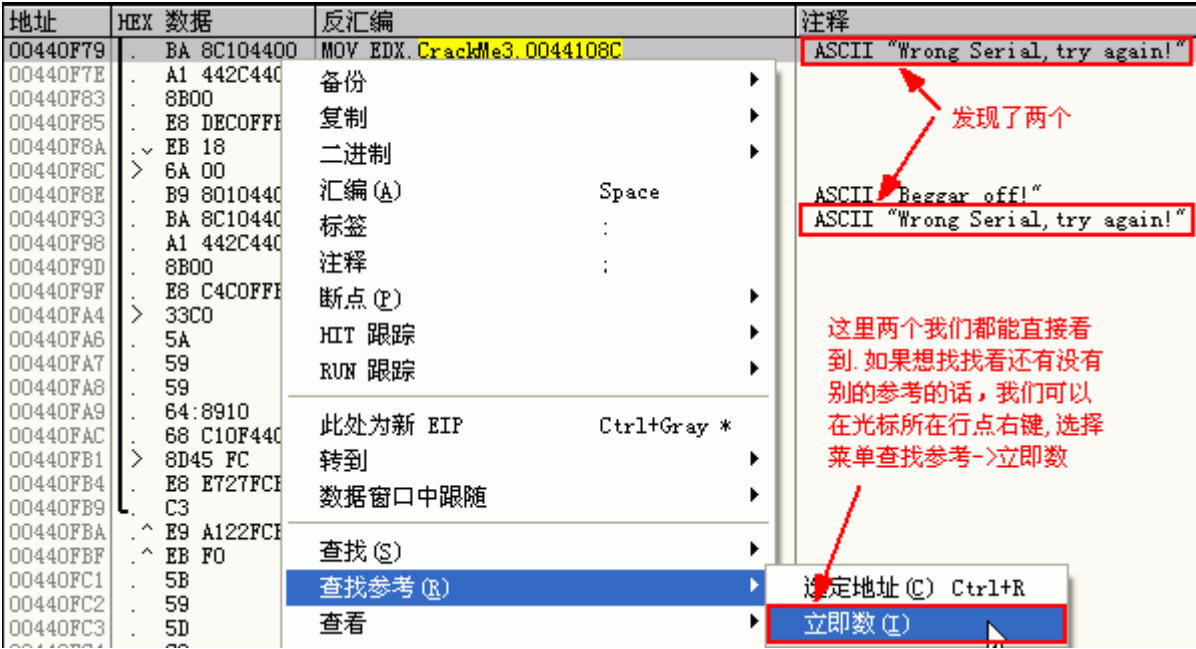
我们在反汇编窗口中右击，出来一个菜单，我们在 查找->所有参考文本字符串 上左键点击：



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 OIlyDBG 的一些操作，我就尽量使用 OIlyDBG 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：



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 D7FEFDFD	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 7C	MOV EAX,DWORD PTR SS:[EBP-4]	
00440F2F	BA 1E3 00441014	MOV ECX,00441014	ASCII "Registered User"
00440F34	E8 F3C0FFFF	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 D7	CALL CrackMe3.00441014	
00440F49	8B45 1	MOV ECX,DWORD PTR SS:[EBP-4]	
00440F4C	BA 2C	MOV ECX,2C	ASCII "GFX-754-IER-954"
00440F51	E8 D6	CALL CrackMe3.00441014	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,0044103C	ASCII "CrackMe cracked successfully"
00440F5F	BA 5C104400	MOV EDX,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 F8	CALL CrackMe3.00441014	
00440F70	EB 32	MOV EBX,EBP	
00440F72	6A 00	PUSH 0	
00440F74	B9 80	MOV ECX,80	ASCII "Beggan off!"
00440F79	BA 8C	MOV ECX,8C	ASCII "Wrong Serial,try again!"
00440F7E	A1 44	MOV EAX,DWORD PTR DS:[442C44]	
00440F83	8B00	MOV EAX,DWORD PTR DS:[EAX]	
00440F85	E8 DC0FFFF	CALL CrackMe3.0043D068	
00440F8A	EB 18	MOV EBX,EBP	
00440F8C	6A 00	PUSH 0	
00440F8E	B9 80104400	MOV ECX,00441080	ASCII "Beggan off!"
00440F93	BA 8C104400	MOV EDX,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：

[- CPU - 主线程, 模块 - CrackMe3]

文件(F) 查看(V) 调试(D) 插件(P) 选项(O) 窗口(W) 帮助(H)

地址 HEX 数据 反汇编 注释 寄存器 (FPU)

00440F2C	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]		EAX 00000009
00440F2F	BA 14104400	MOV EDX, CrackMe3.00441014	ASCII "Registered Use	ECX 77D1882A USER32.77D1882A
00440F34	E8 F32BFCFF	CALL CrackMe3.00403B2C	关键, 要用F7跟进去	EDX 00140608
00440F39	75 51	JNZ SHORT CrackMe3.00440F8C	这里跳走就完蛋	EBX 00D44830
00440F3B	8D55 FC	LEA EDX, DWORD PTR SS:[EBP-4]		ESP 0012F99C
00440F3E	8B83 C8020000	MOV EAX, DWORD PTR DS:[EBX+2C8]		EBP 0012F9B0
00440F44	E8 D7FEFDFE	CALL CrackMe3.00420E20		ESI 00D461B0
00440F49	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]		EDI 00000013
00440F4C	BA 2C104400	MOV EDX, CrackMe3.0044102C	ASCII "GFX-754-IER-95	EIP 00440F2C CrackMe3.00440F2C
00440F51	E8 D62BFCFF	CALL CrackMe3.00403B2C	关键, 要用F7跟进去	C 0 ES 0023 32位 0 (FFFFFFFF)
00440F56	75 1A	JNZ SHORT CrackMe3.00440F72	这里跳走就完蛋	P 0 CS 001B 32位 0 (FFFFFFFF)
00440F58	6A 00	PUSH 0		A 0 SS 0023 32位 0 (FFFFFFFF)
00440F5A	B9 3C104400	MOV ECX, CrackMe3.0044103C	ASCII "CrackMe cracke	Z 0 DS 0023 32位 0 (FFFFFFFF)
00440F5F	BA 5C104400	MOV EDX, CrackMe3.0044105C	ASCII "Congrats! You	S 0 FS 003B 32位 7FFDF000 (FFF
00440F64	A1 442C4400	MOV EAX, DWORD PTR DS:[442C44]		T 0 GS 0000 NULL
00440F69	8B00	MOV EAX, DWORD PTR DS:[EAX]		D 0
00440F6B	E8 F8C0FFFF	CALL CrackMe3.0043D068		O 0 LastErr ERROR_SUCCESS (00
00440F70	EB 32	JMP SHORT CrackMe3.00440FA4		EFL 00000202 (NO, NB, NE, A, NS, PO
00440F72	6A 00	PUSH 0		ST0 empty -1.53020323596546355
00440F74	B9 80104400	MOV ECX, CrackMe3.00441080	ASCII "Beggan off!"	ST1 empty -UNORM A654 00000000
00440F79	BA 8C104400	MOV EDX, CrackMe3.0044108C	ASCII "Wrong Serial, t	ST2 empty -5.89484696512789504
00440F7E	A1 442C4400	MOV EAX, DWORD PTR DS:[442C44]		ST3 empty 3.370944181138696192
00440F83	8B00	MOV EAX, DWORD PTR DS:[EAX]		ST4 empty 0.020993585905696696
00440F85	E8 DEC0FFFF	CALL CrackMe3.0043D068		ST5 empty -1.12920271423374976
				ST6 empty 1.000000000000000000
				ST7 empty 1.000000000000000000

堆栈 SS:[0012F9AC]=00D44DB4, (ASCII "CCDebugger")

EAX=00000009

拷贝内容到剪贴板
修改数据
数据窗口中跟随地址
数据窗口中跟随数值
界面选项

内存中的数据

ASCII "CCDebugger" .. t4D.
t4D.xf..754-GFX-
IER-954. t4D. t4D.
\\t..??.....

指向下一个 SEH 记录的指
堆栈窗口中的数据
ASCII "CCDebugger"

命令 :
断点位于 CrackMe3.00440F2C 暂停

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	MOV ESI, ESI	
00403B2E	57	MOV ESI, ESI	
00403B2F	89C6	MOV EAX, EDI	

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：

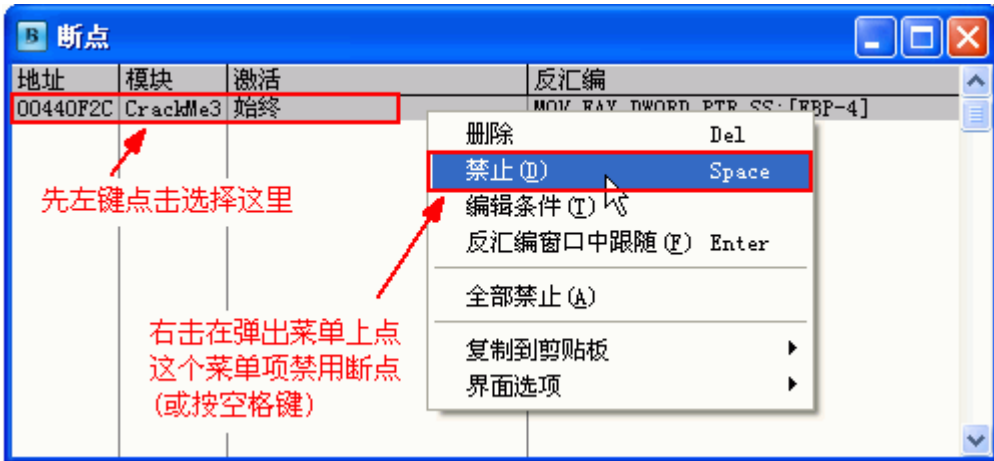
00403B2C /\$ 53	PUSH EBX	
00403B2D . 56	PUSH ESI	
00403B2E . 57	PUSH EDI	
00403B2F . 89C6	MOV ESI, EAX	; 把 EAX 内我们输入的用户名送到 ESI
00403B31 . 89D7	MOV EDI, EDX	; 把 EDX 内的数据“Registered User”送到 EDI
00403B33 . 39D0	CMP EAX, EDX	; 用“Registered User”和我们输入的用户名作比较
00403B35 . 0F84 8F000000	JE CrackMe3. 00403BCA	; 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

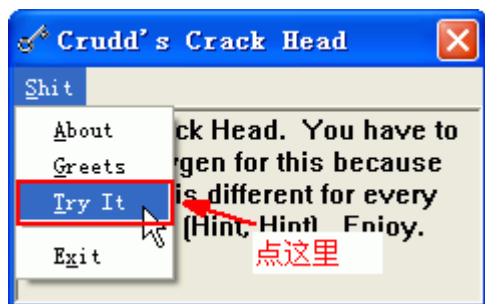
注册码：GFX-754-IER-954

点击“Register now !”按钮，呵呵，终于成功了：



OllyDBG 入门系列（三）—函数参考

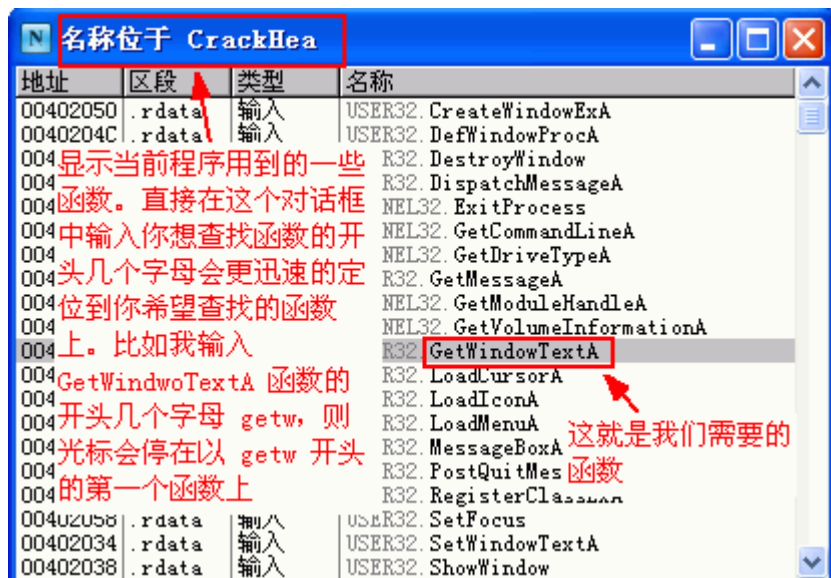
现在进入第三篇，这一篇我们重点讲解怎样使用 OllyDBG 中的函数参考（即名称参考）功能。仍然选择 crackmes.cjb.net 镜像打包中的一个名称为 CrackHead 的 crackme。老规矩，先运行一下这个程序看看：



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：

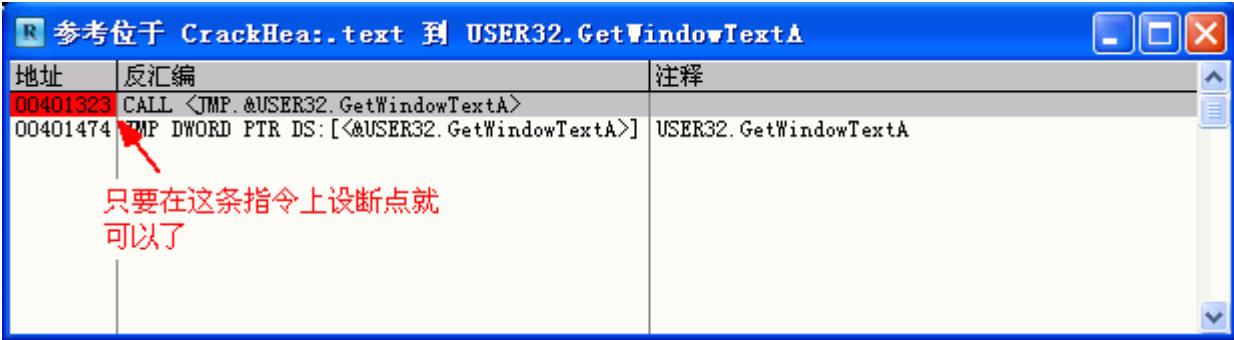


我们点一下那个“Check It”按钮试一下，哦，竟然没反应！我再输个“78787878”试试，还是没反应。再试试输入字母或其它字符，输不进去。由此判断注册码应该都是数字，只有输入正确的注册码才有动静。用 PEiD 检测一下，结果为 MASM32 / TASM32，怪不得程序比较小。信息收集的差不多了，现在关掉这个程序，我们用 OlllyDBG 载入，按 F9 键直接让它运行起来，依次点击上面图中所说的菜单，使被调试程序显示如上面的第二个图。先不要点那个“Check It”按钮，保留上图的状态。现在我们没有什么字符串好参考了，我们就在 API 函数上下断点，来让被调试程序中断在我们希望的地方。我们在 OlllyDBG 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 OlllyDBG 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 OlllyDBG 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>      ; 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 keygen"
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

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同，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 [OlllyDBG 入门系列（四）—内存断点](#) 中再讲吧。

OlllyDBG 入门系列（四）—内存断点

还记得上一篇《[OlllyDBG 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 OlllyDBG 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 OlllyDBG 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
00401318	. 6A 28	PUSH 28	
00401318	. 68 C4334000	PUSH CrackHea.004033C4	Count = 28 (40.)
0040131D	. FF35 90314000	PUSH DWORD PTR DS:[40339C]	Buffer = CrackHea.004033C4
00401323	. E8 4C010000	CALL <JMP. &USER32. GetWindowTextA>	hWnd = 003905FC (class='Edit', parent=000B0588)
00401328	. E8 A5000000	CALL CrackHea.004013D2	GetWindowTextA
0040132D	. 3BC6	CMP EAX, ESI	关键，要按F7键跟进去
0040132F	. 75 42	JNZ SHORT CrackHea.00401373	比较
00401331	. EB 2C	JMP SHORT CrackHea.0040135F	不等则完蛋
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	微软
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 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	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	0 注意一下这个值，等 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0040342C	0 我们会看到它是怎 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0040343C	0 么出来的 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0040344C	0 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	

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：

查看可执行文件 (E)

复制到可执行文件

转到

Hex

文本

短型

长型

浮点

反汇编

指定

创建标签

界面选项

表达式 Ctrl+G

地址	HEX 数据
00403000	53 69 6D 70 6C 65 57 69 6E 43 60
00403010	72 75 64 64 27 73 20 43 72 61 63
00403020	64 00 64 69 63 68 00 62 75 74 74
00403030	65 63 6B 20 49 74 00 65 64 69 74
00403040	69 63 00 43 72 75 64 64 27 73 20
00403050	20 48 65 61 64 2E 20 20 59 6F 75
00403060	20 74 6F 20 77 72 69 74 65 20 61
00403070	65 6E 20 66 6F 72 20 74 68 69 73
00403080	75 73 65 20 74 68 65 20 73 65 72
00403090	73 20 64 69 68 68 65 72 65 6E 74
004030A0	65 76 65 72 79 20 63 6F 6D 70 75
004030B0	48 69 6E 74 2C 20 48 69 6E 74 29
004030C0	6A 6F 79 2E 00 47 72 65 65 74 73

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 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	00 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：

备份

复制

二进制

标签

断点 (B)

查找 (S)

查找参考 (R) Ctrl+R

查看可执行文件 (E)

复制到可执行文件

转到

Hex

文本

短型

长型

浮点

反汇编

指定

创建标签

界面选项

内存访问 (A)

内存写入 (W)

硬件访问

硬件写入

硬件执行 (H)

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	00 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

在我们选中的灰色部分上右击，会弹出上面的菜单，我们选断点->内存写入

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
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。

断点 (P)

RUN 跟踪

此处为新 EIP

转到

数据窗口中跟随

查找 (S)

查找参考 (R)

查看

Ctrl+Gray *

切换 F2

条件 (C) Shift+F2

条件记录 (L) Shift+F4

运行到选定位置 F4

内存访问 (A)

内存写入 (W)

删除内存断点 (M)

硬件执行 (H)

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	上面一条和这一条指令主要是用来判断是否已把把EDX中的值与经过上面运算后的ECX中值相加送把EAX与EDX异或。如果我们输入的是负数，则此ESI出栈。看到这条和下一条指令，我们要考虑一把ESI中的值与797A7553H异或
004013FD	^ 75 F1	JNZ SHORT CrackHea.004013F0	
004013FF	. 8D040A	LEA EAX, DWORD PTR DS:[EDX+ECX]	
00401402	. 33C2	XOR EAX, EDX	
00401404	. 5E	POP ESI	
00401405	. 81F6 53757A79	XOR ESI, 797A7553	
0040140B	L C3	RETN	RootPathName = NULL GetDriveTypeA pFileSystemNameSize = NULL pFileSystemNameBuffer = NULL pFileSystemFlags = NULL pMaxFilenameLength = NULL pVolumeSerialNumber = NULL MaxVolumeNameSize = B (11.) VolumeNameBuffer = CrackHea.0040339C RootPathName = NULL GetVolumeInformationA ALT+F9返回后来到位置
0040140C	r\$ 60	PUSHAD	
0040140D	. 6A 00	PUSH 0	
0040140F	. E8 B4000000	CALL <TMP.&KERNEL32.GetDriveTypeA>	
00401414	. A2 EC334000	MOV BY 我们上一篇分析的代码，	
00401419	. 6A 00	PUSH 0	
0040141B	. 6A 00	PUSH 0	
0040141D	. 6A 00	PUSH 0	
0040141F	. 6A 00	PUSH 0	
00401421	. 6A 00	PUSH 0	
00401423	. 6A 0B	PUSH 0	
00401425	. 68 9C334000	PUSH 0	
0040142A	. 6A 00	PUSH 0	
0040142C	. E8 A3000000	CALL <TMP.&KERNEL32.GetVolumeInformationA>	
00401431	. 8D35 9C334000	LEA ESI, DWORD PTR DS:[40339C]	
00401437	. 0FB60D EC334000	MOVZX ECX, BYTE PTR DS:[4033EC]	
0040143E	. 33FF	XOR EDI, EDI	

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

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。

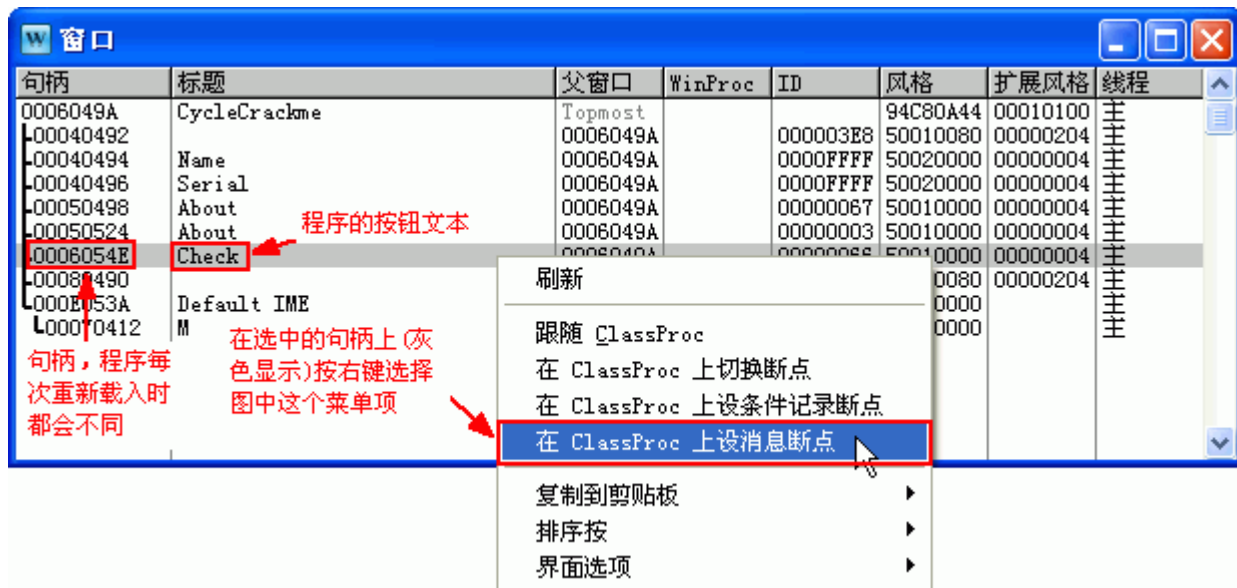
感谢 mirrormask 兄指出本文中的错误！

01lyDBG 入门系列（五）—消息断点及 RUN 跟踪

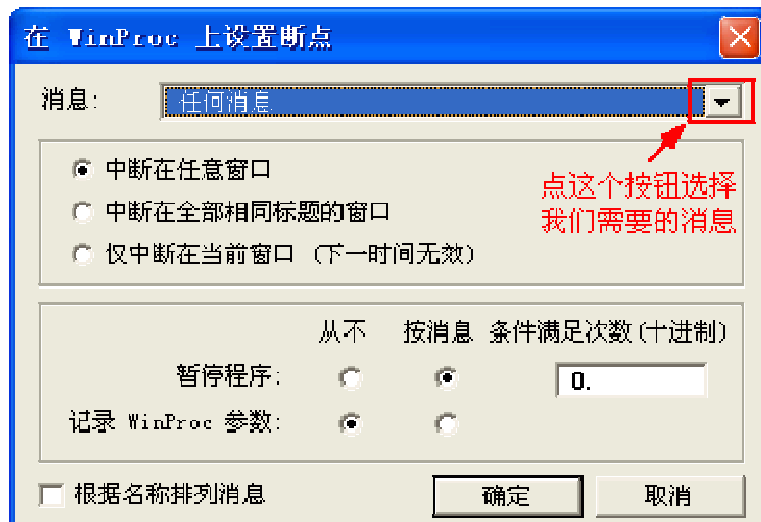
找了几十个不同语言编写的 crackme，发现只用消息断点的话有很多并不能真正到达我们要找的关键位置，想想还是把消息断点和 RUN 跟踪结合在一起讲，更有效一点。关于消息断点的更多内容大家可以参考 jingulong 兄的那篇《几种典型程序 Button 处理代码的定位》的文章，堪称经典之作。今天仍然选择 crackmes.cjb.net 镜像打包中的一个名称为 cycle 的 crackme。按照惯例，我们先运行一下这个程序看看：



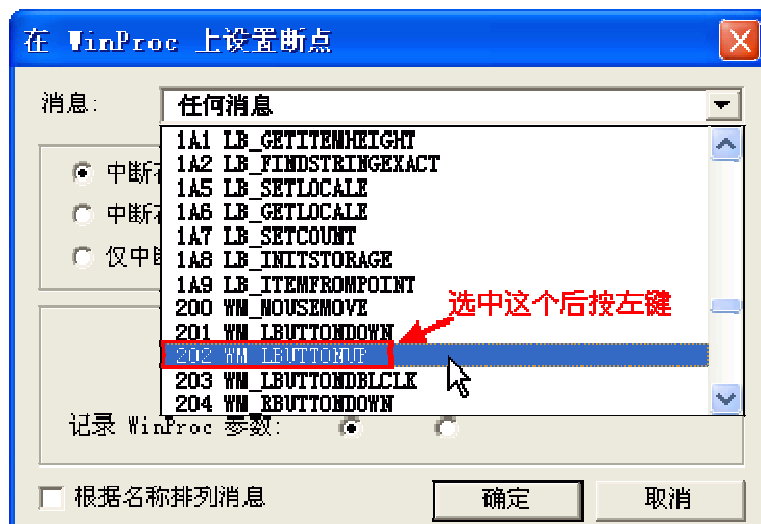
我们输入用户名 CCDebugger，序列号 78787878，点上面那个“Check”按钮，呵，没反应！看来是要注册码正确才有动静。现在关掉这个 crackme，用 PEiD 查一下壳，原来是 MASM32 / TASM32 [Overlay]。启动 01lyDBG 载入这个程序，F9 让它运行。这个程序按我们前面讲的采用字符串参考或函数参考的方法都很容易断下来。但我们今天主要学习的是消息断点及 RUN 跟踪，就先用消息断点来断这个程序吧。在设消息断点前，有两个内容我们要简单了解一下：首先我们要了解的是消息。Windows 的中文翻译就是“窗口”，而 Windows 上面的应用程序也都是通过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 01lyDBG 把这个 crackme 载入并按 F9 键运行了吧？现在我们输入用户名“CCDebugger”，序列号“78787878”，先不要点那个“Check”按钮，我们来到 01lyDBG 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：



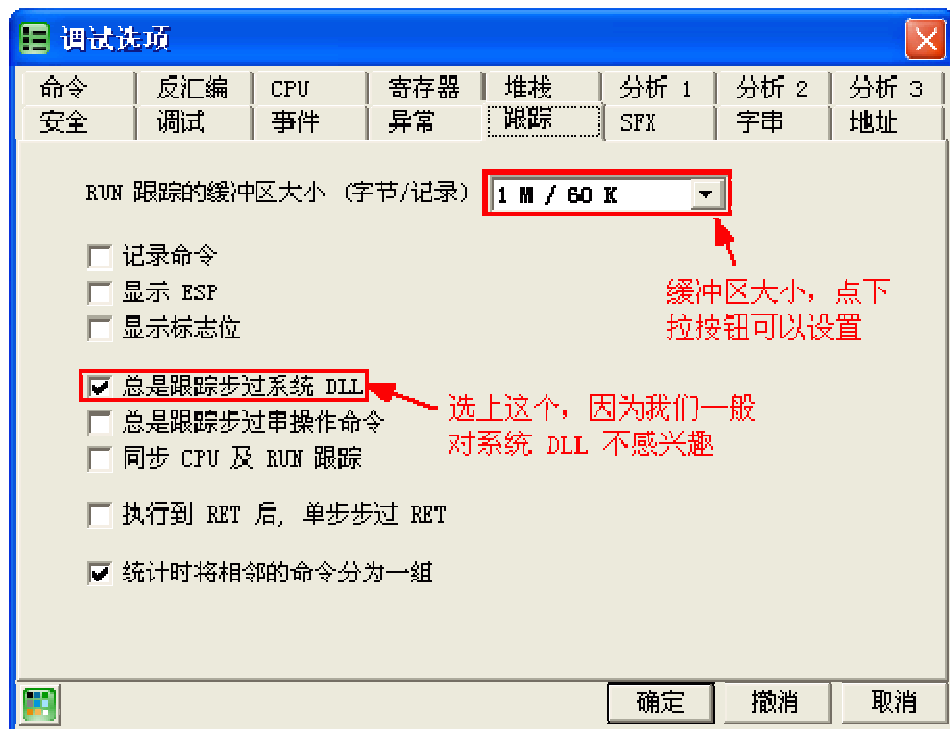
我们在选中的条目上点右键，再选择上图所示的菜单项，会来到下面这个窗口：



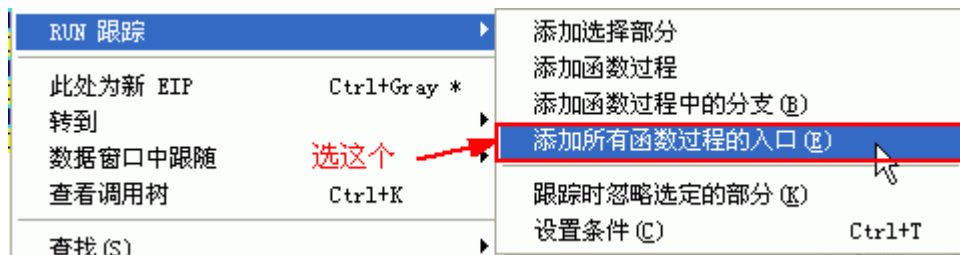
现在我们点击图上的那个下拉菜单，呵，原来里面的消息真不少。这么多消息我们选哪个呢？注册是个按钮，我们就在按下按钮再松开时让程序中断。查一下 MSDN，我们知道这个消息应该是 WM_LBUTTONDOWN 看字面意思也可以知道是左键松开时的消息：



从下拉菜单中选中那个 202 WM_LBUTTON_UP，再按确定按钮，我们的消息断点就设好了。现在我们还要做一件事，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：



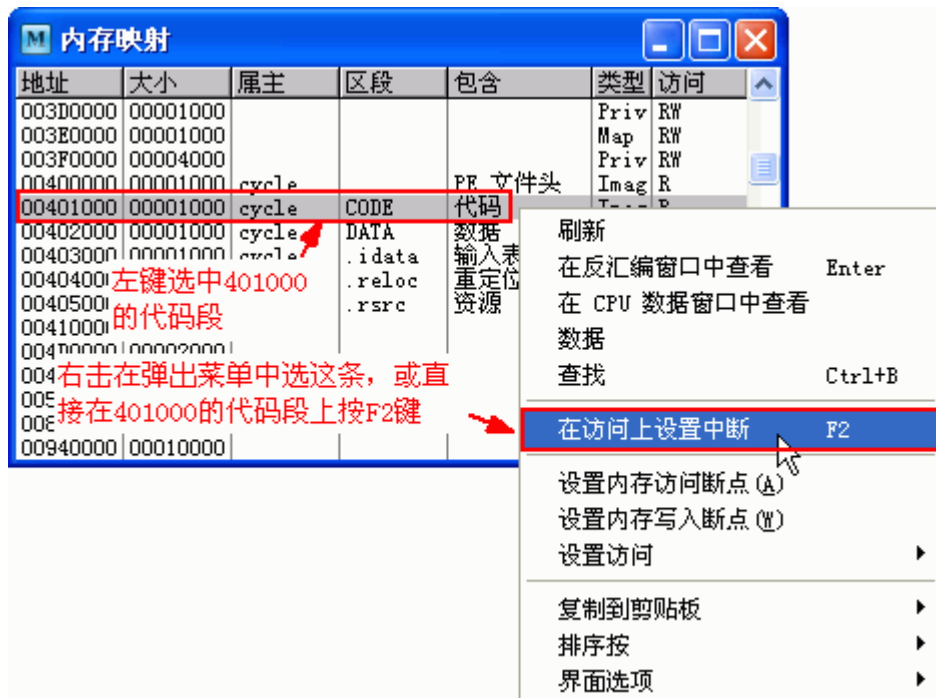
我们可以看到 OllyDBG 把识别出的函数过程都在前面加了灰色条：

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 68	PUSH 68	pTemplate = 68
00401017	FF35	JMP DWORD PTR DS:[402194]	hInst = 00400000
0040101D	E8 86020000	CALL <JMP.&USER32.DialogBoxParamA>	DialogBoxParamA
00401022	6A 00	PUSH 0	ExitCode = 0
00401024	E8 7C020000	CALL <JMP.&KERNEL32.ExitProcess>	ExitProcess

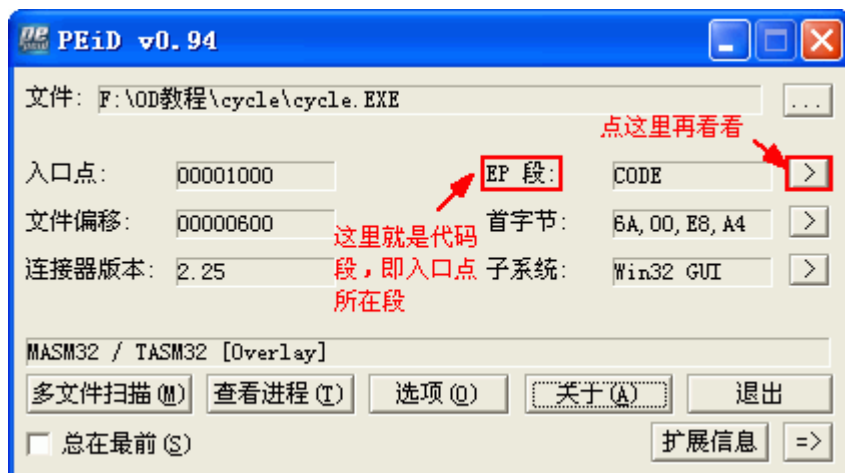
现在我们回到那个 crackme 中按那个“Check”按钮，被 OllyDBG 断下了：

地址	HEX 数据	反汇编	注释
77D3B00E	8BFF	MOV EDI, EDI	
77D3B010	55	PUSH EBP	
77D3B011	8BF8	MOV EBP, ESP	
77D3B013	8B4C	MOV ECX, DWORD PTR SS:[EBP+8]	

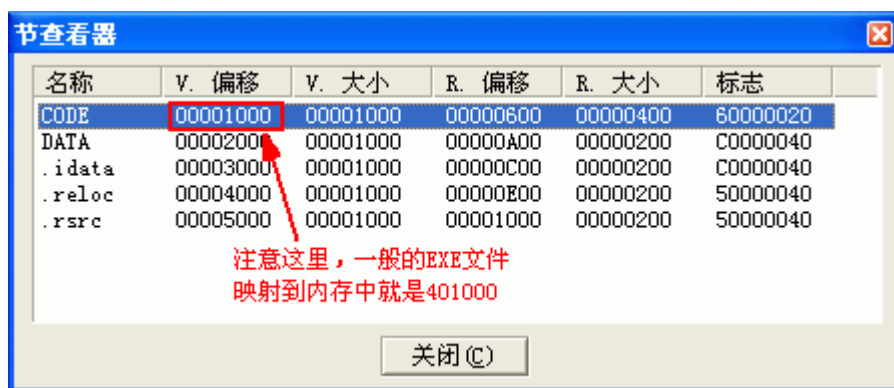
这时我们点击菜单查看->内存，或者点击工具栏上那个“M”按钮（也可以按组合键 ALT+M），来到内存映射窗口：



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 OllyDBG 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 OllyDBG 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：

RUN 跟踪

返回	线程	模块	地址	修改后的寄存器
217.	主	ntdll	7C92EB94	RETN
216.	主	cycle	00401029	ENTER 0,0
215.	主	cycle	0040102D	PUSH EBX
214.	主	cycle	0040102E	PUSH EDI
213.	主	cycle	0040102F	PUSH ESI
212.	主	cycle	00401030	CMP DWORD PTR SS:[EBP+10],67
211.	主	cycle	00401031	CMP DWORD PTR SS:[EBP+10],66
210.	主	cycle	00401032	JE SHORT cycle.00401034
209.	主	cycle	00401034	CALL DWORD PTR SS:[EBP+10]
208.	主	cycle	00401035	CALL DWORD PTR SS:[EBP+10]
207.	主	cycle	00401036	CALL DWORD PTR SS:[EBP+10]
206.	主	cycle	00401037	CALL DWORD PTR SS:[EBP+10]
205.	主	cycle	00401038	CALL DWORD PTR SS:[EBP+10]
204.	主	cycle	00401039	CALL DWORD PTR SS:[EBP+10]
203.	主	cycle	0040103A	CALL DWORD PTR SS:[EBP+10]

左击随便定位到一条cycle(当前调试的)记录,右击选择菜单中的统计模块,看看有哪些指令执行过

反汇编窗口中跟随 (F) Enter
显示注释 (S)
高亮寄存器 (Q)
标记选择的地址 (A)
标记地址
转到上一行 (P) Minus
转到下一行 (N) Plus
统计模块 (M)
全局统计 (G)
清除 RUN 跟踪
记录到文件
复制到剪贴板
界面选项

我们现在再来看看统计的情况:

统计用于 cycle

计数	地址	第一个命令	注释
75.	00401029	ENTER 0,0	
74.	0040102D	PUSH EBX	
1.	0040102E	PUSH EDI	
1.	00401082	CMP DWORD PTR SS:[EBP+10],67	
1.	0040108D	CMP DWORD PTR SS:[EBP+10],66	
1.	0040111F	RETN	

我们主要关心这些执行过一次,随便选择一条双击可以跳到反汇编窗口中的相关代码处,这里我就选401082地址处的那一条

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	
00401088	75 05	JNZ SHORT cycle.0040108D	
00401088	E8 C4000000	CALL cycle.00401151	
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	Count = 11 (17.)
004010AD	68 E9030000	PUSH 3E9	Buffer = cycle.00402171
004010B2	FF75 08	PUSH DWORD PTR SS:[EBP+8]	ControlID = 3E9 (1001.)
004010B5	E8 07000000	CALL <JMP.&USER32.GetDlgItemTextA>	hWnd
004010BA	0BC0	OR EAX, EAX	GetDlgItemTextA
004010BC	74 6	JE SHORT cycle.0040111F	
004010BE	6A 1	PUSH 1	
004010C0	68 60214000	PUSH cycle.00402160	Count = 11 (17.)
004010C5	68 E8030000	PUSH 3E8	Buffer = cycle.00402160
004010CA	FF75 08	PUSH DWORD PTR SS:[EBP+8]	ControlID = 3E8 (1000.)
004010CD	E8 F7010000	CALL <JMP.&USER32.GetDlgItemTextA>	hWnd
004010D2	0BC0	OR EAX, EAX	GetDlgItemTextA
004010D4	74 49	JE SHORT cycle.0040111F	
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	
004010E4	03F8	ADD EDI, EAX	

双击后来到的位置

我们就在这里设
个断点吧

这个函数熟悉吧

看到我们输入
用户名的
变换了

ASCII "CCDebuggerCCDebug"

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，现在我们就可以开始分析了：

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“CCDebugger”，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”后再回到程序中点“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。

OllyDBG 分析报告系列(6)---OD 异常处理机制

这是分析报告的第六部分，请多多指教。😊

Ollydbg（以下均简称为 OD）中对各类断点、各类热键以及菜单消息的处理都是通过一个不停循环的消息线程处理的，当然对于调试程序异常的处理也包括在里面。

在 OD 中该循环位于 WinMain 函数的中间部分，先说一下其流程，然后具体分析代码：

1. 得到开机到现在的时间，根据时间间隔更新日志文件，流程如下：
 - a) 检查是否创建了记录 OD 调试信息日志文件，若有创建继续检查距离上次更新是否超过 30 秒，若超过则将调试信息新增部分写入到文件中；
 - b) 检查是否创建了运行跟踪结果的日志文件，若有创建继续检查距离上次更新是否超过 30 秒，若超过则将运行跟踪结果新增部分写入到文件中；
 - c) 保存最后一次更新时间；
2. 检查是否需要刷新反汇编窗口、主窗口，若需要则检查刷新间隔是否超过 0.5 秒，超过则刷新，并记录该次刷新的时间，否则就不刷新；

3. 接受线程的消息队列，并将消息信息放入 MSG 结构体中，并检查消息是否接受成功，若失败跳过第 4 步对消息的处理；
4. 转换子窗体中的快捷键消息，若转换成功则跳过对其他消息的处理；
5. ……一般消息的判断以及消息收发，这里跳过，没有分析；
6. 对 EXE、DLL 做相应的检查以及设置（添加临时断点、添加插件栏）；
7. 使用 WaitForDebugEvent 函数接收异常，若没有接收到则发送 NULL 消息，跳转到第 20 步；
8. 先让插件处理异常，然后检查接收到的异常信息中进程 Id 是否为调试进程的 Id，异常码是否为 EXCEPTION_DEBUG_EVENT，若不是则调用 ContinueDebugEvent 继续调试程序，其 ContinueStatus 参数设置为 DBG_CONTINUE，跳转到第 1 步继续循环；这一步比较关键，当异常为调试异常时并没有调用 ContinueDebugEvent 函数让函数继续执行，这样调试程序就被断下了；
9. 修改调试程序的状态标识为被调试状态，调用函数检查 int3 断点，若是运行状态则设置断点；
10. 检查异常、线程的信息结构体，设置标识，调用一个函数处理所有的异常；
11. 检查更新线程信息，检查是否开了运行跟踪文件，若开了则更新文件，将调试程序状态设置回去；
12. 检查异常信息以及平台情况，做相应的错误处理；
13. 下面开始为对调试功能的处理，一共有 9 种情况（0-No animation、1-Animate into、Animate over、Execute till RET、Skip RET instruction、Execute till user code、Run trace in、Run trace over、Gracefully stop animation），有一个全局变量标识其状态；
14. 对 Execute till RET（执行到返回）进行处理，检查调试寄存器信息，检查第一个机器码是否为 C2（retn…）C3（retn）CA（retf…）CB（retf）或者 CF（iretd），若不是则跳过对其处理的部分；若是则开始处理：
 - a) 调用 _G0 函数检查线程信息，设置断点，检查其合法性，根据检查结果设置提示信息；
 - b) 调整优先级，对调试程序结构体信息、OD 界面显示信息以及插件信息的更新
 - c) 发送广播到子窗体，要求更新窗口内容，设置窗口获得焦点，最后返回到第 1 步
15. 对 Execute till user code（执行到用户代码）进行处理，检查 eip 所在的模块是否为系统模块，若不是则调整其优先级，若是则跳过设置优先级；根据 ebx 判断是否调用函数设置所有断点；
16. 对 Gracefully stop animation（停止自动跟踪）处理，检查线程信息，检查完毕后设置 OD 的界面，发送子窗体更新广播，更新插件；
17. 对于其他操作调用 _G0 函数进行处理，除了线程 ID，其他参数都设 0 值；
18. 对 No animation（无自动跟踪）进行处理，调用 Set_all_Bpoint 函数处理，然后跳转到第 20 步，函数中调用 ContinueDebugEvent 函数继续调试程序，其 ContinueStatus 参数设置为 DBG_EXCEPTION_NOT_HANDLED；
19. ……部分省略，是一些对 OD 操作的处理，跟上面雷同，一般都是调用 _G0 函数执行代码；检查更新标识；用 _Broadcast 函数发送更新广播；检查是否有自动跟踪文件，对其进行更新等等；
20. 两个对系统时间全局变量的判断以及设置，更新窗口，跳转到循环体开始处

得到开机到现在的时间，检查是否创建了记录 OD 调试信息的日志文件以及运行跟踪结果的日志文件，根据结果进行相应的跳转：

```

00439077 > E8 566>call <jmp.&KERNEL32.GetTickCount>
0043907C . |8945 C>mov dword ptr [ebp-34], eax
0043907F . |833D E>cmp dword ptr [4D55E8], 0 ; 检查调试信息日志文件指针是否存在
00439086 . |75 09 jnz short 00439091 ; 存在则跳转到下面检查暂停时间
00439088 . |833D 4>cmp dword ptr [4D9E40], 0 ; 检查运行跟踪的文件指针是否存在
0043908F . |74 43 je short 004390D4 ; 存在则跳转到下面检查暂停时间
这里为检查时间处，根据时间更新文件：
00439091 > |A1 283>mov eax, dword ptr [4E3B28]

```


00439096 . |05 307>add eax, 7530 ; 上次更新时间增加 30 秒
0043909B . |3B45 C>cmp eax, dword ptr [ebp-34] ; 与当前时间比较, 检查是否间隔超过 30 秒
0043909E . |73 34 jnb short 004390D4 ; 超过则跳过文件的更新
004390A0 . |833D E>cmp dword ptr [4D55E8], 0 ; 检查调试信息日志文件指针是否存在
004390A7 . |74 0D je short 004390B6 ; 不存在跳过更新被调试的文件

更新调试信息日志文件:

004390A9 . |8B15 E>mov edx, dword ptr [4D55E8]
004390AF . |52 push edx
004390B0 . |E8 B3B>call 004A4F68 ; _fflush 更新调试信息日志文件
004390B5 . |59 pop ecx

检查运行跟踪文件指针是否存在, 不存在则跳过更新运行跟踪文件:

004390B6 > |833D 4>cmp dword ptr [4D9E40], 0
004390BD . |74 0D je short 004390CC ; 不存在则跳过更新运行跟踪文件
004390BF . |8B0D 4>mov ecx, dword ptr [4D9E40]
004390C5 . |51 push ecx
004390C6 . |E8 9DB>call 004A4F68 ; _fflush 更新运行跟踪文件
004390CB . |59 pop ecx

将开机到当前的时间赋值给一个全局变量, 保存文件最后一次更新时间:

004390CC > |8B45 C>mov eax, dword ptr [ebp-34]
004390CF . |A3 283>mov dword ptr [4E3B28], eax

检查是否需要刷新反汇编窗口, 若不需要则跳过刷新:

004390D4 > |833D 3>cmp dword ptr [4E3B34], 0 ; [4E3B34] 反汇编窗口刷新标识
004390DB . |74 26 je short 00439103

检查刷新窗口的间隔时间是否超过 0.5 秒, 若超过则刷新所有窗口, 并设置刷新时间以及刷新标识:

004390DD . |8B55 C>mov edx, dword ptr [ebp-34]
004390E0 . |2B15 3>sub edx, dword ptr [4E3B38] ; 得到两次刷新窗口的时间间隔
004390E6 . |81FA F>cmp edx, 1F4 ; 比较时间间隔是否低于等于 0.5 秒
004390EC . |76 15 jbe short 00439103 ; 若低于 0.5 秒则跳过刷新
004390EE . |E8 D95>call _Redrawdisassembler ; 刷新反汇编窗口
004390F3 . |8B4D C>mov ecx, dword ptr [ebp-34]
004390F6 . |33C0 xor eax, eax
004390F8 . |890D 3>mov dword ptr [4E3B38], ecx ; 记录刷新最后时间
004390FE . |A3 343>mov dword ptr [4E3B34], eax ; 将刷新标识置零

检查主窗口更新标识, 若不需要更新则将当前时间设置到记录主窗口更新时间的全局变量, 然后跳过对主窗口的刷新:

00439103 > |833D C>cmp dword ptr [4D57C0], 0 ; 检查主窗口更新标识
0043910A . |74 09 je short 00439115
0043910C . |833D 3>cmp dword ptr [4E3B30], 0 ; 检查 nNumber 是否为 0
00439113 . |75 0B jnz short 00439120
00439115 > |8B55 C>mov edx, dword ptr [ebp-34] ; 记录更新时间
00439118 . |8915 2>mov dword ptr [4E3B2C], edx
0043911E . |EB 55 jmp short 00439175 ; 跳过对主窗口的更新
检查距离上次刷新的时间间隔是否超过 0.5 秒, 小于则跳过刷新:
00439120 > |8B0D 2>mov ecx, dword ptr [4E3B2C]
00439126 . |81C1 E>add ecx, 3E8

0043912C . |3B4D C>cmp ecx, dword ptr [ebp-34]

0043912F . |73 44 jnb short 00439175

检查 nNumber 是否超过 50, 小于则跳过更新:

00439131 . |833D 3>cmp dword ptr [4E3B30], 32

00439138 . |7C 2B jl short 00439165

间隔时间*1000/edx (edx > 50), 最后用计算的值更新:

0043913A . |8B45 C>mov eax, dword ptr [ebp-34]

0043913D . |8B15 3>mov edx, dword ptr [4E3B30]

00439143 . |2B05 2>sub eax, dword ptr [4E3B2C]

00439149 . |50 push eax ; /Divisor

0043914A . |68 E80>push 3E8 ; |Multiplier = 3E8 (1000.)

0043914F . |52 push edx ; |Multiplicand => 0

00439150 . |E8 FB5>call <jmp.&KERNEL32.MulDiv> ; \MulDiv

00439155 . |50 push eax ; /Arg2

00439156 . |8D8E 6>lea ecx, dword ptr [esi+3162] ; |

0043915C . |51 push ecx ; |Arg1

0043915D . |E8 CA8>call _Flash ; _Flash

00439162 . |83C4 0>add esp, 8

00439165 > |33C0 xor eax, eax

00439167 . |8B55 C>mov edx, dword ptr [ebp-34]

0043916A . |A3 303>mov dword ptr [4E3B30], eax ; 将除数置零

0043916F . |8915 2>mov dword ptr [4E3B2C], edx ; 将系统时间记录

接受主线程的消息队列, 并将消息信息放入 MSG 结构体中, 并检查消息是否接受成功, 若失败跳过下面对消息的处理:

00439175 > |6A 01 push 1 ; /RemoveMsg = PM_REMOVE

00439177 . |6A 00 push 0 ; |MsgFilterMax = WM_NULL

00439179 . |6A 00 push 0 ; |MsgFilterMin = WM_NULL

0043917B . |6A 00 push 0 ; |hWnd = NULL

0043917D . |8D8D 3>lea ecx, dword ptr [ebp-9C8] ; |

00439183 . |51 push ecx ; |pMsg

00439184 . |E8 A56>call <jmp.&USER32.PeekMessageA> ; \PeekMessageA

00439189 . |85C0 test eax, eax

0043918B . |0F84 C>jc 00439454

转换子窗口消息中的快捷键消息:

00439191 . |8D85 3>lea eax, dword ptr [ebp-9C8]

00439197 . |50 push eax ; /pMsg

00439198 . |8B15 8>mov edx, dword ptr [4D3B80] ; |

0043919E . |52 push edx ; |hClient => 00020762 (class='MDIClient', parent=000607B6)

0043919F . |E8 3E6>call <jmp.&USER32.TranslateMDISysA>; \TranslateMDISysAccel

004391A4 . |85C0 test eax, eax

004391A6 . |0F85 9>jnz 00439442

若子窗口有操作消息, 则跳过对主窗体的消息的判断, 对子窗口消息进行处理; 下面是对一般消息的判断, 处理还是在下面, 这里跳过:

.....

.....

这里开始为对所有消息的处理:

```
00439454 > |833D 8>cmp    dword ptr [4D5780], 0
0043945B . |0F84 9>je      004395F9
00439461 . |833D 1>cmp    dword ptr [4E3610], 0      ; (initial cpu selection)
00439468 . |0F84 8>je      004395F9
0043946E . |E8 F95>call   0045F36C      ; 得到加载的 dll 模块信息
```

```
00439473 . |E8 9C7>call   _Listmemory      ; 检查内存链表
```

将模块信息给 edi 寄存器, 下面对其进行检查, DLL 以及 EXE 程序分开检查:

```
00439478 . |8B3D 1>mov    edi, dword ptr [4D7218]      ; edi : pModule
```

检查是否为 DLL, 若是则初始化参数, 并跳转到条件判断处, 用一个循环体进行判断:

```
0043947E . |833D A>cmp    dword ptr [4D6EA0], 0
00439485 . |74 7E je      short 00439505
00439487 . |85FF test    edi, edi
00439489 . |74 7A je      short 00439505
0043948B . |33DB xor     ebx, ebx
0043948D . |EB 6E jmp     short 004394FD
```

若是 DLL 则用一个循环体检查调试的模块是否是加载的模块, 若是则在入口设置 int3 临时断点:

```
0043948F > |8D049B lea    eax, dword ptr [ebx+ebx*4]
00439492 . |8D0480 lea    eax, dword ptr [eax+eax*4]
00439495 . |8D0480 lea    eax, dword ptr [eax+eax*4]
00439498 . |8D04C0 lea    eax, dword ptr [eax+eax*8]
0043949B . |F64407>test   byte ptr [edi+eax+8], 10      ; 检查模块是否需要记录
004394A0 . |75 5A jnz     short 004394FC      ; 不需要则遍历下个模块
004394A2 . |8D149B lea    edx, dword ptr [ebx+ebx*4]
004394A5 . |8D1492 lea    edx, dword ptr [edx+edx*4]
004394A8 . |8D1492 lea    edx, dword ptr [edx+edx*4]
004394AB . |8D14D2 lea    edx, dword ptr [edx+edx*8]
004394AE . |03D7 add     edx, edi
004394B0 . |83C2 5>add    edx, 50      ; 检查模块路径名与加载的 DLL 是否相同
004394B3 . |52 push     edx            ; /Arg2
004394B4 . |68 805>push   004D5B80      ; |
004394B9 . |E8 FEA>call   004A38BC      ; \_stricmp
004394BE . |83C4 0>add    esp, 8
004394C1 . |85C0 test    eax, eax
004394C3 . |75 37 jnz     short 004394FC      ; 不同则遍历下个模块
004394C5 . |8D0C9B lea    ecx, dword ptr [ebx+ebx*4]
004394C8 . |8D0C89 lea    ecx, dword ptr [ecx+ecx*4]
004394CB . |8D0C89 lea    ecx, dword ptr [ecx+ecx*4]
004394CE . |8D0CC9 lea    ecx, dword ptr [ecx+ecx*8]
004394D1 . |837C0F>cmp    dword ptr [edi+ecx+28], 0      ; 检查模块入口点是否存在
004394D6 . |74 2D je      short 00439505      ; 不存在则跳出循环
004394D8 . |8D049B lea    eax, dword ptr [ebx+ebx*4]
004394DB . |6A 00 push   0            ; /Arg4 = 00000000
004394DD . |6A 00 push   0            ; |Arg3 = 00000000
```

```

004394DF . |68 000>push 800 ; |Arg2 = 00000800 临时断点
004394E4 . |8D0480 lea eax, dword ptr [eax+eax*4] ; |
004394E7 . |8D0480 lea eax, dword ptr [eax+eax*4] ; |
004394EA . |8D04C0 lea eax, dword ptr [eax+eax*8] ; |
004394ED . |8B5407>mov edx, dword ptr [edi+eax+28] ; |
004394F1 . |52 push edx ; |Arg1
004394F2 . |E8 690>call _Setbreakpointtext ; \_Setbreakpointtext
004394F7 . |83C4 1>add esp, 10
004394FA . |EB 09 jmp short 00439505
004394FC > |43 inc ebx

```

这里为循环体判断处:

```

004394FD > |3B1D 0>cmp ebx, dword ptr [4D7200]
00439503 . ^|7C 8A jl short 0043948F

```

下面为对 EXE 程序的检查, 跳转到条件判断处, 用一个循环体进行判断, 设置工具栏:

```

00439505 > |833D 5>cmp dword ptr [4D7350], 0
0043950C . |0F84 B>je 004395CB
00439512 . |85FF test edi, edi ; 检查模块信息是否存在
00439514 . |0F84 B>je 004395CB
0043951A . |33DB xor ebx, ebx
0043951C . |EB 14 jmp short 00439532
0043951E > |8D049B lea eax, dword ptr [ebx+ebx*4]
00439521 . |8D0480 lea eax, dword ptr [eax+eax*4]
00439524 . |8D0480 lea eax, dword ptr [eax+eax*4]
00439527 . |8D04C0 lea eax, dword ptr [eax+eax*8]
0043952A . |F64407>test byte ptr [edi+eax+8], 10 ; 检查模块是否需要记录
0043952F . |74 09 je short 0043953A
00439531 . |43 inc ebx
00439532 > |3B1D 0>cmp ebx, dword ptr [4D7200]
00439538 . ^|7C E4 jl short 0043951E
0043953A > |3B1D 0>cmp ebx, dword ptr [4D7200]
00439540 . |7D 6B jge short 004395AD
00439542 . |6A 00 push 0 ; /Arg1 = 00000000
00439544 . |E8 47A>call _Suspendprocess ; \_Suspendprocess 暂停线程
00439549 . |59 pop ecx
0043954A . |833D 2>cmp dword ptr [4DE924], 0
00439551 . |74 05 je short 00439558
00439553 . |E8 58E>call 00497BB0
00439558 > |833D 1>cmp dword ptr [4E2F10], 0
0043955F . |74 05 je short 00439566
00439561 . |E8 223>call 0049CD88
00439566 > |8D96 8>lea edx, dword ptr [esi+318E]
0043956C . |52 push edx ; /Arg2
0043956D . |6A 00 push 0 ; |Arg1 = 00000000
0043956F . |E8 BC8>call _Message ; \_Message

```

```

00439574 . |83C4 0>add esp, 8
00439577 . |33DB xor ebx, ebx
00439579 . |EB 25 jmp short 004395A0
0043957B > |8D049B lea eax, dword ptr [ebx+ebx*4]
0043957E . |8D0480 lea eax, dword ptr [eax+eax*4]
00439581 . |8D0480 lea eax, dword ptr [eax+eax*4]
00439584 . |8D04C0 lea eax, dword ptr [eax+eax*8]
00439587 . |F64407>test byte ptr [edi+eax+8], 10
0043958C . |74 11 je short 0043959F
0043958E . |8D149B lea edx, dword ptr [ebx+ebx*4]
00439591 . |8D1492 lea edx, dword ptr [edx+edx*4]
00439594 . |8D1492 lea edx, dword ptr [edx+edx*4]
00439597 . |8D14D2 lea edx, dword ptr [edx+edx*8]
0043959A . |836417>and dword ptr [edi+edx+8], FFFFFFFF
0043959F > |43 inc ebx
004395A0 > |3B1D 0>cmp ebx, dword ptr [4D7200]
004395A6 . ^|7C D3 jl short 0043957B
004395A8 . |E8 9F6>call 0046044C ; 设置工具栏
004395AD > |33DB xor ebx, ebx
004395AF . |EB 12 jmp short 004395C3
004395B1 > |8D049B lea eax, dword ptr [ebx+ebx*4]
004395B4 . |8D0480 lea eax, dword ptr [eax+eax*4]
004395B7 . |8D0480 lea eax, dword ptr [eax+eax*4]
004395BA . |8D04C0 lea eax, dword ptr [eax+eax*8]
004395BD . |834C07>or dword ptr [edi+eax+8], 10 ; 检查模块是否需要记录
004395C2 . |43 inc ebx

```

这里为循环体判断处:

```

004395C3 > |3B1D 0>cmp ebx, dword ptr [4D7200]
004395C9 . ^|7C E6 jl short 004395B1
设置 edi 为线程信息, 检查线程信息是否存在, 循环遍历活动线程, 将其激活:
004395CB > |8B3D B>mov edi, dword ptr [4D7DB0] ; pThreadInfo
004395D1 . |85FF test edi, edi
004395D3 . |74 1C je short 004395F1
004395D5 . |33DB xor ebx, ebx
004395D7 . |EB 10 jmp short 004395E9
004395D9 > |8B47 0>mov eax, dword ptr [edi+C]
004395DC . |50 push eax ; /hThread
004395DD . |E8 925>call <jmp.&KERNEL32.ResumeThread> ; \ResumeThread
004395E2 . |43 inc ebx
004395E3 . |81C7 6>add edi, 66C

```

循环体判断处, 检查线程是否遍历完:

```

004395E9 > |3B1D 9>cmp ebx, dword ptr [4D7D98]
004395EF . ^|7C E8 jl short 004395D9

```

根据线程信息设置标识:

004395F1 > |33D2 xor edx, edx

004395F3 . |8915 1>mov dword ptr [4E3610], edx

检查被调试程序是否运行状态，若不是则遍历所有插件，用其处理异常，然后跳过运行时的处理：

004395F9 > |833D 5>cmp dword ptr [4D5A5C], 3

00439600 . |74 14 je short 00439616

00439602 . |6A 00 push 0 ; /Arg1 = 00000000

00439604 . |E8 43D>call 00496B4C ; \Plugin_SetDebugEvent

00439609 . |59 pop ecx

0043960A . |6A 01 push 1 ; /Timeout = 1. ms

0043960C . |E8 AB5>call <jmp.&KERNEL32.Sleep> ; \Sleep

00439611 . |E9 D80>jmp 0043A2EE

使用 WaitForDebugEvent 接收异常，并检查是否接收到异常，若接收到异常则跳过下面的处理：

00439616 > |6A 00 push 0 ; /Timeout = 0. ms

00439618 . |68 145>push 004D5714 ; |pDebugEvent = 0 |lydbg. 004D5714

0043961D . |E8 E85>call <jmp.&KERNEL32.WaitForDebugEv>; \WaitForDebugEvent

00439622 . |85C0 test eax, eax

00439624 . |75 44 jnz short 0043966A

检查标识以及系统时间，根据判断结构处理，发送 WM_NULL 消息，设置标识为 0；或者跳过这个处理让插件处理异常：

00439626 . |833D 5>cmp dword ptr [4E3B54], 0

0043962D . |74 27 je short 00439656

0043962F . |8B0D 5>mov ecx, dword ptr [4E3B58]

00439635 . |83C1 6>add ecx, 64

00439638 . |3B4D C>cmp ecx, dword ptr [ebp-34]

0043963B . |73 19 jnb short 00439656

0043963D . |6A 00 push 0 ; /lParam = 0

0043963F . |6A 00 push 0 ; |wParam = 0

00439641 . |6A 00 push 0 ; |Message = WM_NULL

00439643 . |A1 5C3>mov eax, dword ptr [4E3B5C] ; |

00439648 . |50 push eax ; |ThreadId => 340

00439649 . |E8 F25>call <jmp.&USER32.PostThreadMessag>; \PostThreadMessageA

0043964E . |33D2 xor edx, edx

00439650 . |8915 5>mov dword ptr [4E3B54], edx

让插件处理 0 号异常然后跳过处理下面的处理：

00439656 > |6A 00 push 0 ; /Arg1 = 00000000

00439658 . |E8 EFD>call 00496B4C ; \Plugin_SetDebugEvent

0043965D . |59 pop ecx

0043965E . |6A 00 push 0 ; /Timeout = 0. ms

00439660 . |E8 575>call <jmp.&KERNEL32.Sleep> ; \Sleep

00439665 . |E9 840>jmp 0043A2EE

将异常事件压栈，让插件处理异常：

0043966A > |68 145>push 004D5714 ; /Arg1

0043966F . |E8 D8D>call 00496B4C ; \Plugin_SetDebugEvent

00439674 . |59 pop ecx

检查是否是被调试程序所报的异常，若不是则设置相关的异常信息，检查异常事件是否为调试异常：

00439675 . |8B0D 1>mov ecx, dword ptr [4D5718] ; DebugEvent.dwProcessId

0043967B . |3B0D 7>cmp ecx, dword ptr [4D5A70]

00439681 . |74 56 je short 004396D9

00439683 . |A1 185>mov eax, dword ptr [4D5718]

00439688 . |50 push eax ; /Arg5 => 00000BA8

00439689 . |8B15 1>mov edx, dword ptr [4D5714] ; |

0043968F . |52 push edx ; |Arg4 => 00000001

00439690 . |8D8E 8>lea ecx, dword ptr [esi+D86] ; |

00439696 . |51 push ecx ; |Arg3

00439697 . |6A 00 push 0 ; |Arg2 = 00000000

00439699 . |6A 00 push 0 ; |Arg1 = 00000000

0043969B . |E8 6C0>call _Addtolist ; _Addtolist

004396A0 . |83C4 1>add esp, 14

004396A3 . |833D 1>cmp dword ptr [4D5714], 1 ; EXCEPTION_DEBUG_EVENT

004396AA . |75 10 jnz short 004396BC ; 不是则跳转

004396AC . |833D 7>cmp dword ptr [4D5770], 0 ; STATUS_WAIT_0

004396B3 . |74 07 je short 004396BC ; 若是则跳转

设置 ContinueStatus 为 DBG_EXCEPTION_NOT_HANDLED (不忽略异常):

004396B5 . |BB 010>mov ebx, 80010001

004396BA . |EB 05 jmp short 004396C1 ;

设置 ContinueStatus 为 DBG_CONTINUE (忽略异常继续执行):

004396BC > |BB 020>mov ebx, 10002

调用 ContinueDebugEvent 让调试程序继续执行, 跳转到最上面继续接收消息:

004396C1 > |53 push ebx ; /ContinueStatus

004396C2 . |A1 1C5>mov eax, dword ptr [4D571C] ; |

004396C7 . |50 push eax ; |ThreadId => 340

004396C8 . |8B15 1>mov edx, dword ptr [4D5718] ; |

004396CE . |52 push edx ; |ProcessId => BA8

004396CF . |E8 EA5>call <jmp.&KERNEL32.ContinueDebugE>; \ContinueDebugEvent

004396D4 . ^E9 9EF>jmp 00439077

将调试程序的状态赋值给一个局部变量, 然后将其设置为 STAT_EVENT, 即进程暂停、被调试; 并根据 int3 断点表设置断点:

004396D9 > |8B0D 5>mov ecx, dword ptr [4D5A5C]

004396DF . |C705 5>mov dword ptr [4D5A5C], 2 ; STAT_EVENT

004396E9 . |33C0 xor eax, eax

004396EB . |C705 F>mov dword ptr [4D56FC], 1

004396F5 . |A3 745>mov dword ptr [4D5774], eax

004396FA . |894D B>mov dword ptr [ebp-50], ecx

004396FD . |8B15 3>mov edx, dword ptr [4D8134]

00439703 . |8915 0>mov dword ptr [4D5700], edx

00439709 . |E8 961>call 0041B5A4

检查模块数量, 若为 0 则跳过下面的取得模块信息部分:

0043970E . |833D 4>cmp dword ptr [4D7348], 0

00439715 . |74 11 je short 00439728

```

00439717 . |E8 505>call 0045F36C ; 得到模块信息
0043971C . |E8 F37>call _Listmemory ; 检查内存链表
00439721 . |E8 463>call 0042D56C ; 清空临时信息表
00439726 . |EB 3C jmp short 00439764
检查线程、异常相关信息，设置标识：
00439728 > |833D 2>cmp dword ptr [4D812C], 0
0043972F . |74 29 je short 0043975A
00439731 . |833D 9>cmp dword ptr [4D7D98], 1 ; 线程数量是否超过 1
00439738 . |7F 20 jg short 0043975A
0043973A . |833D 3>cmp dword ptr [4D8130], 0 ; 异常码 ExceptionCode 是否存在
00439741 . |74 17 je short 0043975A
00439743 . |833D 1>cmp dword ptr [4D5714], 1 ; 异常事件是否为 EXCEPTION_DEBUG_EVENT
0043974A . |75 0E jnz short 0043975A
0043974C . |8B0D 2>mov ecx, dword ptr [4D572C]
00439752 . |3B0D 3>cmp ecx, dword ptr [4D8130] ; ecx: ExceptionCode
00439758 . |74 0A je short 00439764
0043975A > |C705 7>mov dword ptr [4D7C7C], 1

```

初始化几个变量，调用函数处理异常，即是在内存断点的处理中提到的那个函数：

```

00439764 > |33C0 xor eax, eax
00439766 . |33D2 xor edx, edx
00439768 . |A3 308>mov dword ptr [4D8130], eax ; ExceptionAddress
0043976D . |8D4D A>lea ecx, dword ptr [ebp-54] ; pReg
00439770 . |C605 2>mov byte ptr [4E3A20], 0 ; rtrace_Buf
00439777 . |8915 5>mov dword ptr [4E3B54], edx
0043977D . |51 push ecx ; /Arg1
0043977E . |E8 4D5>call 0042EBD0 ; \CheckDebugEvent
00439783 . |59 pop ecx

```

检查更新线程信息，若开了运行跟踪，则还要将运行跟踪的信息写入文件：

```

00439784 . |6A 00 push 0 ; /Arg2 = 00000000
00439786 . |8BD8 mov ebx, eax ; |
00439788 . |8B45 A>mov eax, dword ptr [ebp-54] ; |
0043978B . |50 push eax ; |Arg1
0043978C . |E8 871>call 0048AF18 ; \check_ThreadInfo
00439791 . |83C4 0>add esp, 8
00439794 . |803D 2>cmp byte ptr [4E3A20], 0
0043979B . |74 0B je short 004397A8
0043979D . |68 203>push 004E3A20
004397A2 . |E8 A51>call 0048AE4C set_file_rtrace
004397A7 . |59 pop ecx

```

将调试程序状态重新设置回去：

```

004397A8 > |33D2 xor edx, edx
004397AA . |8B4D B>mov ecx, dword ptr [ebp-50]
004397AD . |8915 5>mov dword ptr [4D8D5C], edx
004397B3 . |890D 5>mov dword ptr [4D5A5C], ecx

```

检查异常信息，并做相应的跳转：

```
004397B9 . |833D 1>cmp dword ptr [4D5714], 1 ; 异常事件是否为 EXCEPTION_DEBUG_EVENT
004397C0 . |75 42 jnz short 00439804
004397C2 . |833D 7>cmp dword ptr [4D5770], 0 ; dwFirstChance == 0 ??
004397C9 . |75 39 jnz short 00439804
004397CB . |833D D>cmp dword ptr [4D36D8], 2 ; 程序的执行平台是否为 windows
004397D2 . |74 0C je short 004397E0
004397D4 . |813D 2>cmp dword ptr [4D572C], 80000000 ; 异常地址是否大于 80000000h
004397DE . |73 24 jnb short 00439804
```

这里为程序平台为 windows 的处理，弹出窗口，调整优先级：

```
004397E0 > |833D 8>cmp dword ptr [4D578C], 0
004397E7 . |75 11 jnz short 004397FA
004397E9 . |8D86 A>lea eax, dword ptr [esi+31A5]
004397EF . |50 push eax ; /Arg2
004397F0 . |6A 00 push 0 ; |Arg1 = 00000000
004397F2 . |E8 397>call _Message ; \_Message
004397F7 . |83C4 0>add esp, 8
004397FA > |6A 00 push 0 ; /Arg1 = 00000000
004397FC . |E8 D78>call _Animate ; \_Animate
00439801 . |59 pop ecx
00439802 . |33DB xor ebx, ebx
```

检查调试操作是否为 Execute till RET（执行到返回），若不是则跳过该处理部分：

```
00439804 > |833D C>cmp dword ptr [4D57CC], 3 ; [4D57CC] == 3 Execute till RET
0043980B . |0F85 1>jnz 00439928
```

检查寄存器信息是否为空，若为空也跳过该处理部分：

```
00439811 . |837D A>cmp dword ptr [ebp-54], 0 ; [ebp-54] : pReg
00439815 . |0F84 0>je 00439928
```

将寄存器的 EIP 所在的机器码读取出来：

```
0043981B . |8D85 9>lea eax, dword ptr [ebp-468]
00439821 . |50 push eax ; /Arg2
00439822 . |8B55 A>mov edx, dword ptr [ebp-54] ; |
00439825 . |8B4A 2>mov ecx, dword ptr [edx+2C] ; |
00439828 . |51 push ecx ; |Arg1
00439829 . |E8 567>call _Readcommand ; \_Readcommand
0043982E . |83C4 0>add esp, 8
```

检查第一个机器码是否为 C2（retn...）C3（retn）CA（retf...）CB（retf）或者 CF（iretd），若不是则跳过对其处理的部分：

```
00439831 . |85C0 test eax, eax
00439833 . |0F86 E>jbe 00439928
00439839 . |33C0 xor eax, eax
0043983B . |8A85 9>mov al, byte ptr [ebp-468]
00439841 . |25 F60>and eax, 0F6
00439846 . |3D C20>cmp eax, 0C2
0043984B . |74 14 je short 00439861
```

```
0043984D . |3D2 xor    edx, edx
0043984F . |8A95 9>mov    dl, byte ptr [ebp-468]
00439855 . |81FA C>cmp    edx, 0CF
0043985B . |0F85 C>jnz    00439928
```

检查线程信息:

```
00439861 > |833D C>cmp    dword ptr [4D57C8], 0
00439868 . |0F84 A>je     0043991A
0043986E . |6A 00 push    0                ; /Arg2 = 00000000
00439870 . |8B4D A>mov    ecx, dword ptr [ebp-54] ; |
00439873 . |51  push    ecx                ; |Arg1
00439874 . |E8 9F1>call   0048AF18          ; \check_ThreadInfo
00439879 . |83C4 0>add    esp, 8
0043987C . |FF05 3>inc    dword ptr [4E3B30]
```

调用 OD 导出函数_G0 使调试程序继续执行, 该函数里面一样对断点进行了设置以及处理, 最后检查返回值是否为 0, 即标识执行成功, 若执行失败则跳过:

```
00439882 . |6A 01 push    1                ; /Arg5 = 00000001
00439884 . |6A 00 push    0                ; |Arg4 = 00000000
00439886 . |6A 02 push    2                ; |Arg3 = 00000002
00439888 . |6A 00 push    0                ; |Arg2 = 00000000
0043988A . |6A 00 push    0                ; |Arg1 = 00000000
0043988C . |E8 83B>call   _Go              ; \_Go
00439891 . |83C4 1>add    esp, 14
00439894 . |85C0 test    eax, eax
00439896 . |74 75 je     short 0043990D
```

调整优先级, 对调试程序结构体信息、OD 界面显示信息以及插件信息的更新:

```
00439898 . |6A 00 push    0                ; /Arg1 = 00000000
0043989A . |E8 398>call   _Animate          ; \_Animate
0043989F . |59  pop    ecx
004398A0 . |E8 0B4>call   0042E2B0          ; update_ThreadRegInfo
004398A5 . |6A 01 push    1                ; /Arg1 = 00000001
004398A7 . |E8 CC8>call   00431978          ; \ update_subwin
004398AC . |59  pop    ecx
004398AD . |68 145>push    004D5714          ; /Arg4 = 004D5714
004398B2 . |8B45 A>mov    eax, dword ptr [ebp-54] ; |
004398B5 . |8B15 7>mov    edx, dword ptr [4D5774] ; |
004398BB . |50  push    eax                ; |Arg3
004398BC . |83CA 0>or     edx, 0            ; |
004398BF . |6A 00 push    0                ; |Arg2 = 00000000
004398C1 . |52  push    edx                ; |Arg1
004398C2 . |E8 BDD>call   00496B84          ; \update_Plugin
004398C7 . |83C4 1>add    esp, 10
004398CA . |85C0 test    eax, eax
004398CC . ^ 0F85 A>jnz    00439077
004398D2 . |68 050>push    105              ; /Arg5 = 00000105
```

```

004398D7 . |6A 00 push 0 ; |Arg4 = 00000000
004398D9 . |6A 00 push 0 ; |Arg3 = 00000000
004398DB . |6A 00 push 0 ; |Arg2 = 00000000
004398DD . |8B0D 1>mov ecx, dword ptr [4D571C] ; |
004398E3 . |51 push ecx ; |Arg1 => 00000340
004398E4 . |E8 2F3>call _Setcpu ; \_Setcpu
004398E9 . |83C4 1>add esp, 14

```

发送广播到子窗体，要求更新窗口内容，设置窗口获得焦点，最后返回到消息循环开始处继续：

```

004398EC . |6A 00 push 0 ; /Arg3 = 00000000
004398EE . |6A 00 push 0 ; |Arg2 = 00000000
004398F0 . |68 740>push 474 ; |Arg1 = 00000474
004398F5 . |E8 7A0>call _Broadcast ; \_Broadcast
004398FA . |83C4 0>add esp, 0C
004398FD . |A1 7C3>mov eax, dword ptr [4D3B7C]
00439902 . |50 push eax ; /hWnd
00439903 . |E8 925>call <jmp.&USER32.SetForegroundWin>; \SetForegroundWindow
00439908 . ^E9 6AF>jmp 00439077

```

调整优先级，返回到消息循环开始处继续：

```

0043990D > |6A 04 push 4 ; /Arg1 = 00000004
0043990F . |E8 C48>call _Animate ; \_Animate
00439914 . |59 pop ecx
00439915 . ^E9 5DF>jmp 00439077

```

检查标识，调整优先级：

```

0043991A > |85DB test ebx, ebx
0043991C . |7E 02 jle short 00439920
0043991E . |33DB xor ebx, ebx
00439920 > |6A 00 push 0
00439922 . |E8 B18>call _Animate
00439927 . |59 pop ecx

```

检查调试操作是否为 Execute till user code（执行到用户代码），若不是则跳过该处理部分：

```

00439928 > |83D C>cmp dword ptr [4D57CC], 5
0043992F . |75 30 jnz short 00439961

```

一样是检查 EIP 寄存器信息是否存在，若不存在跳过该处理部分：

```

00439931 . |837D A>cmp dword ptr [ebp-54], 0
00439935 . |74 2A je short 00439961

```

找到 EIP 所在的模块，检查是否是系统模块，若是则跳到该处理结束处：

```

00439937 . |8B45 A>mov eax, dword ptr [ebp-54]
0043993A . |8B50 2>mov edx, dword ptr [eax+2C]
0043993D . |52 push edx ; /Arg1
0043993E . |E8 D54>call _Findmodule ; \_Findmodule
00439943 . |59 pop ecx
00439944 . |8BF8 mov edi, eax
00439946 . |85C0 test eax, eax
00439948 . |74 09 je short 00439953

```

```
0043994A . |83BF F>cmp dword ptr [edi+3F5], 0 ; issystemdll == 0 ??
```

```
00439951 . |75 0E jnz short 00439961
```

根据 ebx 标识设置优先级，以及进行相应的跳转：

```
00439953 > |85DB test ebx, ebx
```

```
00439955 . |7E 02 jle short 00439959
```

```
00439957 . |33DB xor ebx, ebx
```

```
00439959 > |6A 00 push 0
```

```
0043995B . |E8 788>call _Animate
```

```
00439960 . |59 pop ecx
```

```
00439961 > |83FB 0>cmp ebx, 2
```

```
00439964 . |75 24 jnz short 0043998A
```

检查调试操作是否为 Gracefully stop animation（停止自动跟踪），若不是则跳过该处理部分，该部分的处理就是调用 0D 的导出函数 _Go，跳转到对调试信息处理的结束判断处，不再做其他检查：

```
00439966 . |833D C>cmp dword ptr [4D57CC], 8
```

```
0043996D . |74 1B je short 0043998A
```

```
0043996F . |6A 00 push 0 ; /Arg5 = 00000000
```

```
00439971 . |6A 00 push 0 ; |Arg4 = 00000000
```

```
00439973 . |6A 00 push 0 ; |Arg3 = 00000000
```

```
00439975 . |6A 00 push 0 ; |Arg2 = 00000000
```

```
00439977 . |A1 1C5>mov eax, dword ptr [4D571C] ; |
```

```
0043997C . |50 push eax ; |Arg1 => 00000340
```

```
0043997D . |E8 92B>call _Go ; \_Go
```

```
00439982 . |83C4 1>add esp, 14
```

```
00439985 . |E9 640>jmp 0043A2EE
```

根据 ebx 标识跳过下面的处理函数：

```
0043998A > |83FB 0>cmp ebx, 1
```

```
0043998D . |75 41 jnz short 004399D0
```

检查调试操作是否为 No animation（无自动跟踪），若不是则跳过该处理部分：

```
0043998F . |833D C>cmp dword ptr [4D57CC], 0
```

```
00439996 . |74 09 je short 004399A1
```

下面对一些参数进行检查，根据检查结果设置 ebx，以及是否设置所有的断点：

```
00439998 . |833D 1>cmp dword ptr [4D5714], 1 ; 检查异常码是否为 EXCEPTION_DEBUG_EVENT
```

```
0043999F . |74 2F je short 004399D0
```

```
004399A1 > |6A 00 push 0
```

```
004399A3 . |6A 00 push 0
```

```
004399A5 . |833D F>cmp dword ptr [4D56FC], 2
```

```
004399AC . |74 0D je short 004399BB
```

```
004399AE . |833D F>cmp dword ptr [4D56FC], 3
```

```
004399B5 . |74 04 je short 004399BB
```

```
004399B7 . |33D2 xor edx, edx
```

```
004399B9 . |EB 05 jmp short 004399C0
```

```
004399BB > |BA 010>mov edx, 1
```

```
004399C0 > |52 push edx ; |Arg2
```

```
004399C1 . |6A FF push -1 ; |Arg1 = FFFFFFFF
```


004399C3 . |E8 687>call 00431430 ; \Set_all_Bpoint

004399C8 . |83C4 1>add esp, 10

004399CB . |E9 1E0>jmp 0043A2EE ; 跳转到消息循环体最后

下面一块是对线程信息的检查，检查完毕后设置 OD 的界面，发送子窗体更新广播，更新插件：

004399D0 > |43 inc ebx

004399D1 . |0F85 9>jnz 00439A71

004399D7 . |E8 386>call _Listmemory

004399DC . |833D 2>cmp dword ptr [4DE924], 0

004399E3 . |74 05 je short 004399EA

004399E5 . |E8 C6E>call 00497BB0 ; update_topwin

004399EA > |833D 1>cmp dword ptr [4E2F10], 0

004399F1 . |74 05 je short 004399F8

004399F3 . |E8 903>call 0049CD88

004399F8 > |837D A>cmp dword ptr [ebp-54], 0

004399FC . |74 0E je short 00439A0C

004399FE . |6A 00 push 0 ; /Arg2 = 00000000

00439A00 . |8B4D A>mov ecx, dword ptr [ebp-54] ; |

00439A03 . |51 push ecx ; |Arg1

00439A04 . |E8 0F1>call 0048AF18 ; \ update_ThreadInfo

00439A09 . |83C4 0>add esp, 8

00439A0C > |68 010>push 101 ; /Arg5 = 00000101

00439A11 . |6A 00 push 0 ; |Arg4 = 00000000

00439A13 . |6A 00 push 0 ; |Arg3 = 00000000

00439A15 . |6A 00 push 0 ; |Arg2 = 00000000

00439A17 . |A1 1C5>mov eax, dword ptr [4D571C] ; |

00439A1C . |50 push eax ; |Arg1 => 00000340

00439A1D . |E8 F63>call _Setcpu ; _Setcpu

00439A22 . |83C4 1>add esp, 14

00439A25 . |6A 00 push 0 ; /Arg3 = 00000000

00439A27 . |6A 00 push 0 ; |Arg2 = 00000000

00439A29 . |68 740>push 474 ; |Arg1 = 00000474

00439A2E . |E8 410>call _Broadcast ; _Broadcast

00439A33 . |83C4 0>add esp, 0C

00439A36 . |8B15 7>mov edx, dword ptr [4D3B7C]

00439A3C . |52 push edx ; /hWnd

00439A3D . |E8 585>call <jmp.&USER32.SetForegroundWin>; \SetForegroundWindow

00439A42 . |33C9 xor ecx, ecx

00439A44 . |33C0 xor eax, eax

00439A46 . |890D 3>mov dword ptr [4E3B30], ecx

00439A4C . |A3 343>mov dword ptr [4E3B34], eax

00439A51 . |6A 04 push 4 ; /Arg1 = 00000004

00439A53 . |E8 207>call 00431978 ; \update_subwin

00439A58 . |59 pop ecx

00439A59 . |68 145>push 004D5714 ; /Arg4 = 004D5714

```

00439A5E . |6A 00 push 0 ; |Arg3 = 00000000
00439A60 . |6A 00 push 0 ; |Arg2 = 00000000
00439A62 . |6A 02 push 2 ; |Arg1 = 00000002
00439A64 . |E8 1BD>call 00496B84 ; \update_Plugin
00439A69 . |83C4 1>add esp, 10
00439A6C . |E9 7D0>jmp 0043A2EE

```

下面省略部分是一些对 OD 操作的处理, 跟上面雷同, 一般都是调用 _G0 函数执行代码; 检查更新一些标识; 用 _Broadcast 函数发送更新广播; 检查是否有自动跟踪文件, 对其进行更新等等:

.....

两个对系统时间全局变量的判断以及设置, 更新窗口, 跳转到循环体开始处:

```

0043A2EE > |833D 1>cmp dword ptr [4E3A1C], 0
0043A2F5 . |74 0A je short 0043A301
0043A2F7 . |A1 1C3>mov eax, dword ptr [4E3A1C]
0043A2FC . |3B45 C>cmp eax, dword ptr [ebp-34]
0043A2FF . |72 1C jb short 0043A31D
0043A301 > |833D 1>cmp dword ptr [4E3814], 0
0043A308 . ^ 0F84 6>je 00439077
0043A30E . |8B15 1>mov edx, dword ptr [4E3814]
0043A314 . |3B55 C>cmp edx, dword ptr [ebp-34]
0043A317 . ^ 0F83 5>jnb 00439077
0043A31D > |E8 3E7>call 00431960
0043A322 . ^\E9 50E>jmp 00439077

```