

第六章：第一次（真正的）破解

一、简介

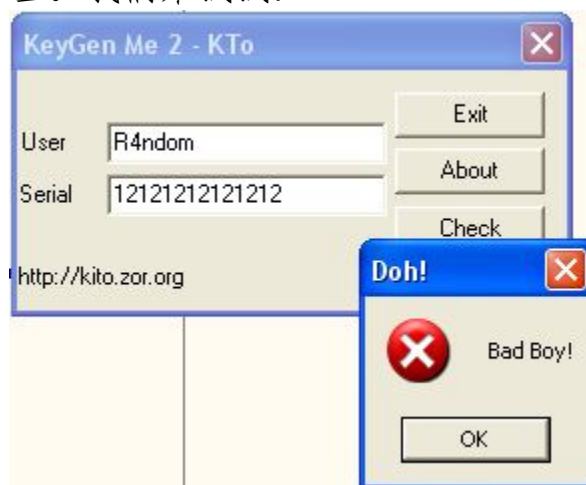
欢迎来到我的教程的第六章。本章我打算离一个真家伙近点：一个真正的 crackme。它也包含在本章的下载中。crackme 是一个渐进式学习逆向工程的好方法，而不应该直接从“真正”的程序入手，crackme 可以从易到难进行，这样你就可以以线性方式学习。最终，我们会一路走到真正的程序，不过也要看到我们才刚刚起步，这些 crackme 也会带给我们巨大的挑战。

你可以在[教程](#)页下载到相关文件和本教程的 PDF 版（译者注：英文版的，此中文翻译我会在教程的最后放出）。

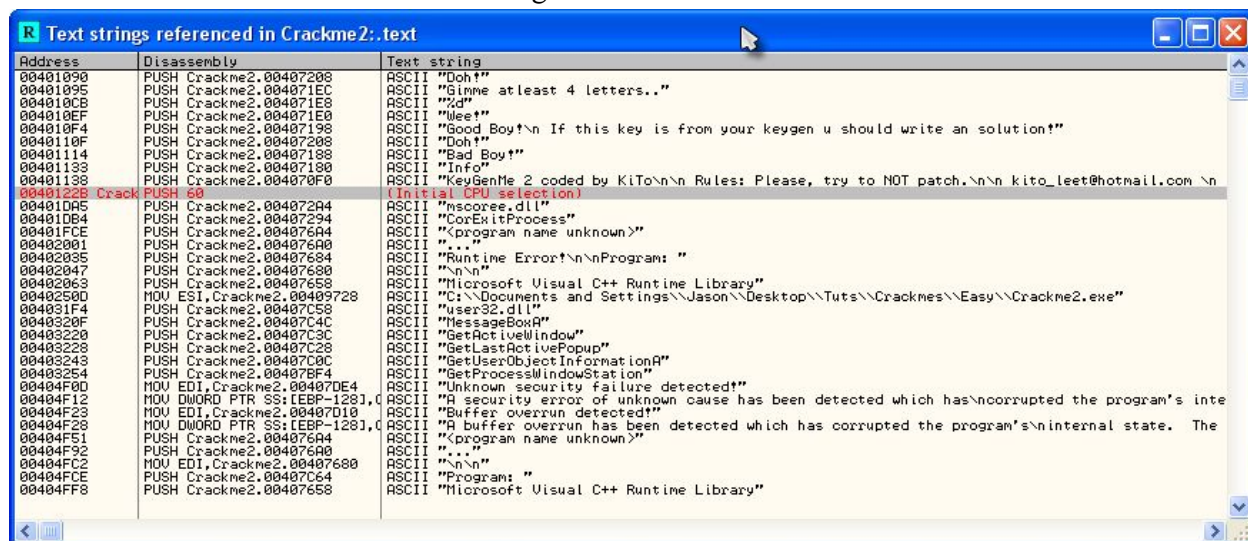
我将会用 OllyDbg1.10（我的版本或原始版本都行，不过如果你用我的版本的话，它看起来和图片一样😄）。我推荐你从工具页面的 Olly Plugins 下载“MnemonicHelp”插件，因为本教程将会用到（教程的下载中也包括的有）。解压后，将其与 x86eas.hlp 文件放到 Olly 文件夹下的 plugins 目录下。如果没有 plugins 文件夹，就在 Olly 的主目录下创建一个。然后打开 Olly 的 Options->Appearance->Directories 标签，然后选择你放置插件的目录。你再在 Olly 的主目录下创建一个叫“UDD”的文件夹，然后让当前设置页的另一个选项也指向这个文件夹。UDD 文件是 Olly 给一个程序做的“便条”，你设置的所有断点、做的注释、一个二进制文件的特有设置都会存储在 UDD 文件中，通常叫做“程序的名字.udd”。如果你在逆向时需要离开一段时间做别的工作，UDD 文件可以让你回来继续对程序进行逆向，因为所有的都被保存起来了。下面是设置两个目录的窗口（带有我的设置）：



看起来比较直接了当。我们来试试：

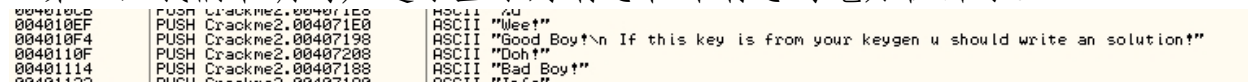


这不是我们想要的。我们看看能不能让 Olly 做些有用的事。回到 Olly，我们来试试我们当前知道的第一个（也是唯一一个）工具。搜索下字符串。右键->Search For->All Referenced Text Strings：



看起来前途光明啊！这里有几件事要注意下。第一，序列号至少得 4 个字符。

第二，我们准确的知道了显示好消息和坏消息的地方在哪了。



点一下 4010F4 那行，看看有什么东西：


```
004010C0 > 69FF 39050000 IMUL EDI,EDI,539
004010C6 . 57 PUSH EDI
004010C7 . 8D4424 4C LEA EAX,DWORD PTR SS:[ESP+4C]
004010CB . 68 E8714000 PUSH Crackme2.004071E8
004010D0 . 58 PUSH EAX
004010D1 . E8 08000000 CALL Crackme2.004011AE
004010D6 . 83C4 8C ADD ESP,8C
004010D9 . 8D4C24 28 LEA ECX,DWORD PTR SS:[ESP+28]
004010DD . 51 LEAH ECX
004010DE . 8D5424 4C LEA EDX,DWORD PTR SS:[ESP+4C]
004010E2 . 52 PUSH EDX
004010E3 . FF15 00704000 CALL DWORD PTR DS:[&KERNEL32.lstrcmpA]
004010E9 . 85C0 TEST EAX,EAX
004010EB . 75 20 JNZ SHORT Crackme2.0040110D
004010ED . 6A 40 PUSH 40
004010EF . 68 E0714000 PUSH Crackme2.004071E0
004010F4 . 68 98714000 PUSH Crackme2.00407198
004010F9 . 53 PUSH EBX
004010FA . FF15 DC704000 CALL DWORD PTR DS:[&USER32.MessageBoxA]
00401100 . 5F POP EDI
00401101 . B8 01000000 MOV EAX,1
00401106 . 5B POP EBX
00401107 . 83C4 60 ADD ESP,60
0040110A . C2 1000 RETN 10
0040110D > 6A 10 PUSH 10
0040110F . 68 08724000 PUSH Crackme2.00407208
00401114 . 68 08714000 PUSH Crackme2.00407188
00401119 . 53 PUSH EBX
0040111A . FF15 DC704000 CALL DWORD PTR DS:[&USER32.MessageBoxA]
00401120 . 5F POP EDI
00401121 . B8 01000000 MOV EAX,1
00401126 . 5B POP EBX
00401127 . 83C4 60 ADD ESP,60
0040112A . C2 1000 RETN 10
0040112D > 8D4424 6C MOV EAX,DWORD PTR SS:[ESP+6C]
00401131 . 6A 40 PUSH 40
00401133 . 68 80714000 PUSH Crackme2.00407180
00401138 . 68 F0704000 PUSH Crackme2.004070F0
0040113D . 5B PUSH EBX
0040113E . FF15 DC704000 CALL DWORD PTR DS:[&USER32.MessageBoxA]
00401144 . 5F POP EDI
00401145 . B8 01000000 MOV EAX,1
0040114A . 5B POP EBX
0040114B . 83C4 60 ADD ESP,60
```

ASCII "%d"

String2 = "%%\&"
String1 = F09C0000 ???
lstrcmpA

Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
Title = "Wee!"
Text = "Good Boy!\n If this key is from your keyge
hOwner = NULL
MessageBoxA
0012F8AC
0012F8AC

Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
Title = "Doh!"
Text = "Bad Boy!"
hOwner = NULL
MessageBoxA
0012F8AC
0012F8AC

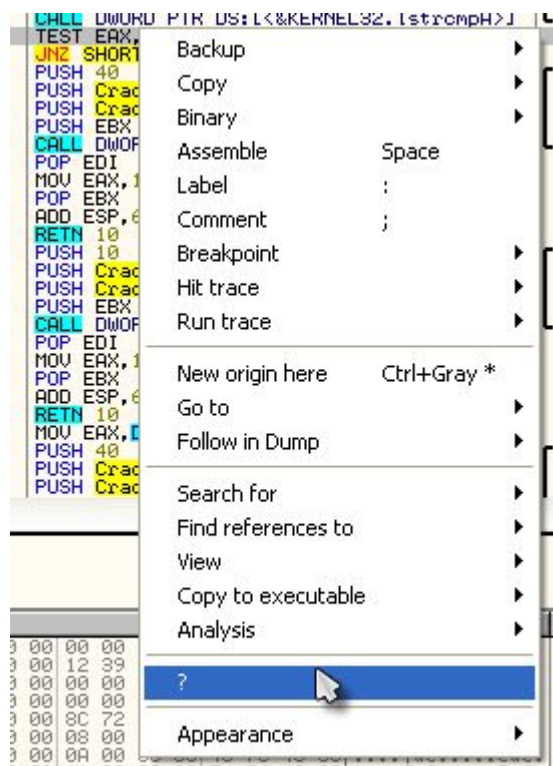
Case SEF of switch 0040102F
Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
Title = "Info"
Text = "KeyGenMe 2 coded by KiTo\n\n Rules: Please
hOwner = 00000003
MessageBoxA
0012F8AC
0012F8AC

这是处理简单 crackme 相当标准的流程（简单点的商业程序也是一样）。搜索文本字符串、找到显示你的注册码/密码/许可证正确与否的相关信息，然后转到那部分代码，你就会看到好的和坏的消息彼此间靠的相当的近呢。那么，根据 R. E. T. A. R. D. 的 2 号规则，查找 比较/跳转语句，以及你想要的那个 CALL。咱们来找找那个跳转语句。

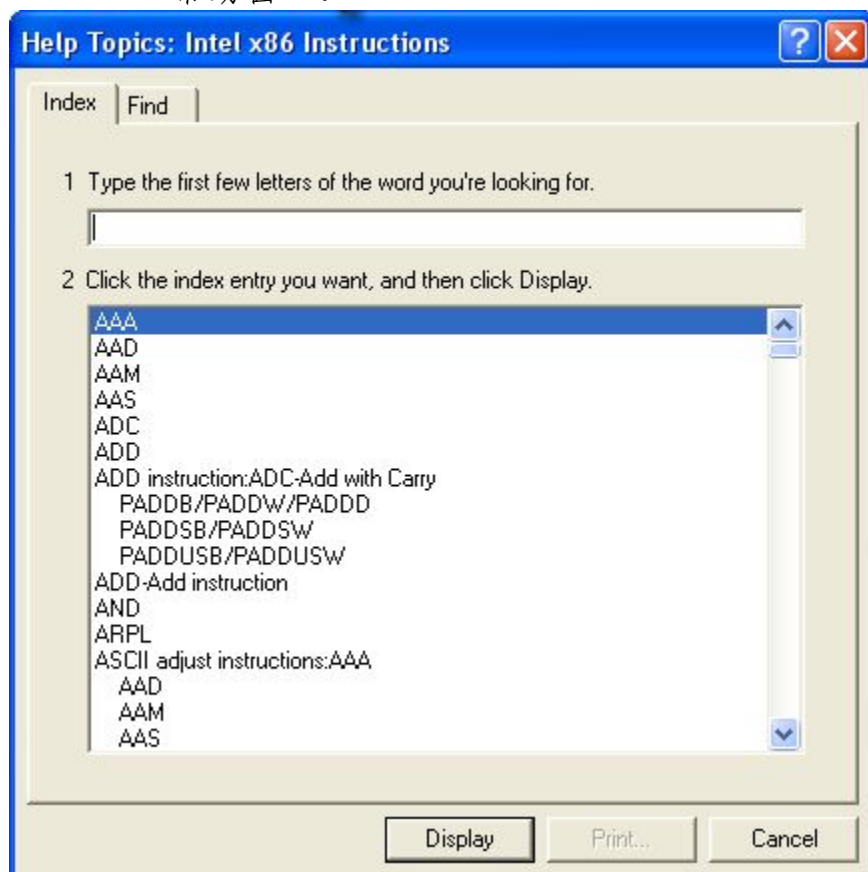
我们找到的第一个跳转是在 4010EB，一个 JNZ 语句。如果我们点击这一行，011y 就会向我显示它会跳向哪里。

Address	Disassembly	Comment
004010E3	FF15 00704000	JMP WORD PTR DS:[<&KERNEL32.1strcmph>]
004010E9	85C0	TEST EAX,EAX
004010EB	75 20	JNZ SHORT Crackme2.0040110D
004010ED	6A 40	PUSH 40
004010EF	68 E0714000	PUSH Crackme2.004071E0
004010F4	68 98714000	PUSH Crackme2.00407198
004010F9	53	PUSH EBX
004010FA	FF15 DC704000	CALL DWORD PTR DS:[<&USER32.MessageBoxA>]
00401100	5F	POP EDI
00401101	B8 01000000	MOV EAX,1
00401106	5B	POP EBX
00401107	83C4 60	ADD ESP,60
0040110A	C2 1000	RET 10
0040110D	6A 10	PUSH 10
0040110F	68 08724000	PUSH Crackme2.00407208
00401114	68 88714000	PUSH Crackme2.00407188
00401119	53	PUSH EBX
0040111A	FF15 DC704000	CALL DWORD PTR DS:[<&USER32.MessageBoxA>]
00401120	5F	POP EDI
00401121	B8 01000000	MOV EAX,1
00401126	5B	POP EBX

可以看到，这条指令跳过了“Good Boy”，直接跳到了“Bad Boy”。这看起来是一个关键点。我们都知道，一个跳转的前面一般都会进行比较，以此来决定是不是要进行跳转。往 JNZ 指令的上面看，我们可以看到一条 TEST EAX, EAX。你可能还没有学到汇编语言书籍关于 TEST 指令的部分，我们来看看能不能找到这个 TEST 指令是干什么的。在本章的前面你已经安装了 MnemonicHelp 插件，那就是我们要用到的。在 TEST 指令上右键，你会在右键菜单中看到一个问号。点它：



就会打开 Mnemonic 帮助窗口：



在上面的文本框中输入“Test”，然后选择（双击）“TEST”。然后就会显示相关指令助记符的帮助：

Intel x86 Instructions

File Edit Bookmark Options Help

Contents Index Back Print

TEST—Logical Compare

See also

Opcode	Instruction	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	AND <i>imm8</i> with AL; set SF, ZF, PF according to result
A9 <i>iw</i>	TEST AX, <i>imm16</i>	AND <i>imm16</i> with AX; set SF, ZF, PF according to result
A9 <i>id</i>	TEST EAX, <i>imm32</i>	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result
F6 <i>ib</i>	TEST <i>imm8</i> with <i>rm8</i> ; set SF, ZF, PF according to result	AND <i>imm8</i> with <i>rm8</i> ; set SF, ZF, PF according to result
F7 <i>iw</i>	TEST <i>imm16</i> with <i>rm16</i> ; set SF, ZF, PF according to result	AND <i>imm16</i> with <i>rm16</i> ; set SF, ZF, PF according to result
F7 <i>id</i>	TEST <i>imm32</i> with <i>rm32</i> ; set SF, ZF, PF according to result	AND <i>imm32</i> with <i>rm32</i> ; set SF, ZF, PF according to result
84 <i>ir</i>	TEST <i>rm8</i> , <i>r8</i>	AND <i>r8</i> with <i>rm8</i> ; set SF, ZF, PF according to result
85 <i>ir</i>	TEST <i>rm16</i> , <i>r16</i>	AND <i>r16</i> with <i>rm16</i> ; set SF, ZF, PF according to result
85 <i>ir</i>	TEST <i>rm32</i> , <i>r32</i>	AND <i>r32</i> with <i>rm32</i> ; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

```

TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP = 0
    THEN ZF ← 0;
    ELSE ZF ← 1;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(*AF is Undefined*)

```

Flags Affected

The OF and CF flags are cleared to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

我们就可以看到 TEST 指令意思是 “*Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.*” (译者注：这段就不翻译了，一是这是帮助中的原文，主要是向大家演示；二是，TEST 指令的意思咱们也可以 GOOGLE 的，中文比看这个容易多了。)”。大部分的时间里，如果 TEST 指令正在测试的两个寄存器的指令相同，就意味着它正在检查它们是不是 0。所以这个满足我们跳转之前要进行比较的需求：

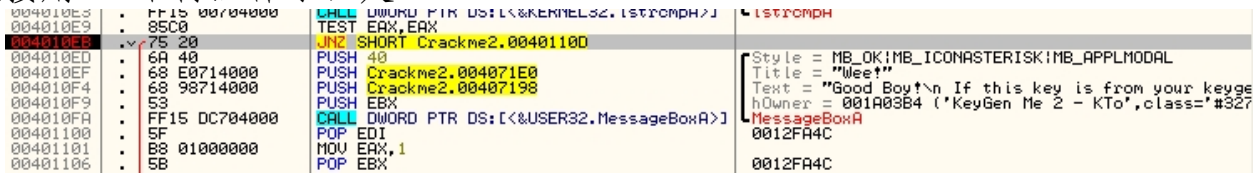
```

CALL DWORD PTR DS:[&KERNEL32.lstrcmpA]
TEST EAX,EAX
JNZ SHORT Crackme2.0040110D
PUSH 40

```

这两条语句的意思是 “如果 EAX 不等于 0，就跳到 40110D”，也就是 “Bad Boy” 那里。好吧，这当然不是我们想要的，咱们来试试我们的推测。在 JNZ 指令处设置一个断点，重启应用。输入用户名和序列号（记住，至少四个字符😄），点击 crackme

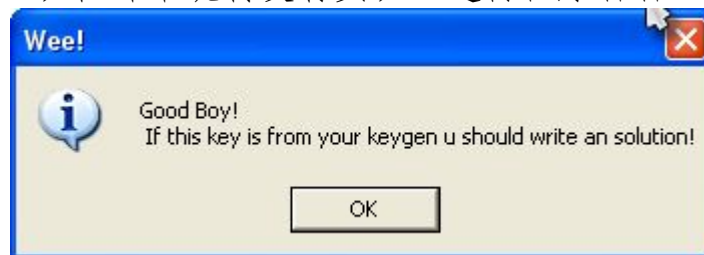
上的 check 按钮。01ly 就会断在我们的 BP（译者注：BP 即是 breakpoint，以后就直接用 BP 不再注释了）处：



现在，我们可以看到我们将会跳过 good boy，直接到 bad boy。咱们来让它不发生。帮 01ly 翻转 0 标志位（参见前面的教程）：

```
C 0 ES 002  
P 0 CS 001  
A 0 SS 002  
Z 1 DS 002  
S 0 FS 003  
T 0 GS 000  
O 0
```

我们可以看到，现在那个跳转没有实现。运行程序看看：

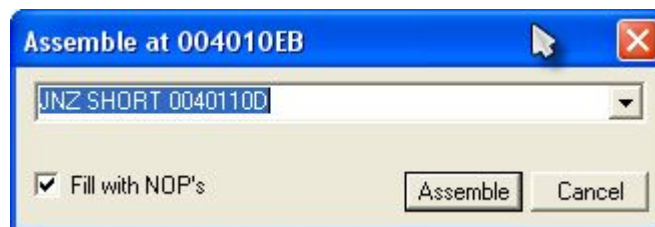


耶，这就是我们想要的。***忽略那个关于 keygen 的消息，有些 crackme 还有其他的目的是要求，不过我还是用它们，我们也需要来学习它的其他两点。一旦我们从这个系列教程中学到了更多的知识，我们还会回来使用它们中的许多。

三、打补丁

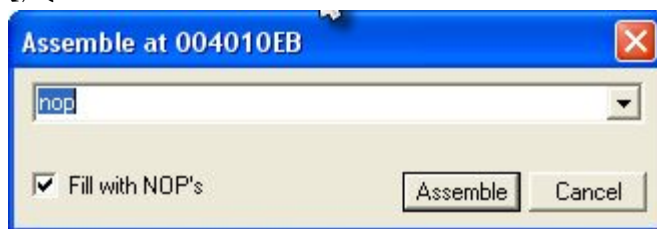
重启 crackme，运行之，输入用户名和序列号，01ly 就会断在我们的断点处。你会注意到，我们会再次跳到 bad boy，因为改变 01ly 的标志位只是临时的方法。这回我们不去临时修改标志位，我打算修改二进制文件中的代码来完全我们想要的。这个叫做打补丁。

点击我们暂停的那行（4010EB），点一下该行的指令列（有 JNZ SHORT...的那部分），然后按一下空格键。会有一个显示该行指令的窗口弹出，也是修改指令的对话框：



现在，我们要做的是将这个跳转到 bad boy 消息处的跳转改成永远不会跳，意思是我们确实不想让这个跳转实现。我们准备做的是，将其替换成一个什么都不做

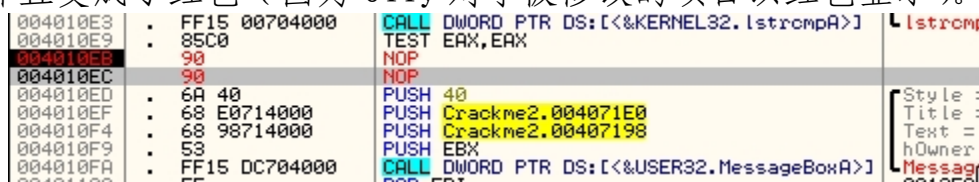
的指令，那就是 NOP 指令。NOP 意思是 No OPeration（不操作）。将对话框中的 JNZ SHORT 0040110D 修改成 NOP:



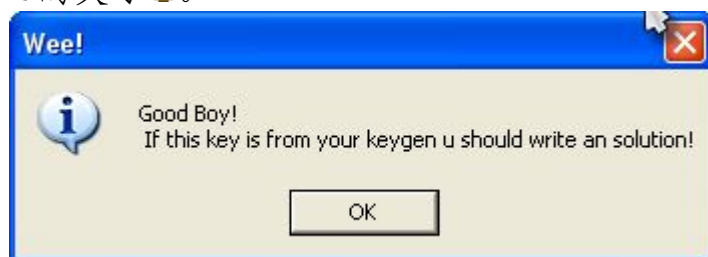
那个“Fill with NOP's”复选框就留那不用管。现在点一下 Assemble 按钮，提交所做的修改，再点一下 Cancel 按钮关闭窗口。

***顺便说一下，如果你没有点那个 Cancel 按钮，而是一直点 Assemble 的话，你会一行一行的修改每一行。这是 Olly 的一个“特性”，用来让你一次修改好几行代码用的。可以让你不用每行都敲空格键。我保证你第一次打补丁的时候会让你疯掉的: X。

注意我们暂停的那行已经改变了，那条指令现在变成了两个 NOP，而不是 JNZ 指令了，并且变成了红色（因为 Olly 对于被修改的项目以红色显示）。



有两个 NOP 的原因是，NOP 操作码只有一个字节长，而被替换的 JNZ 指令有两个字节长，所以 Olly 用两个 NOP 来替换。你也会注意到跳转箭头消失了，因为这行已经不再有任何跳转了！现在单步运行，你会走到 good boy 处。然后 good boy 显示出来了，你开心的笑了😊。



四、保存补丁

有一个重要的事情要注意，如果你重载或重启应用的话你所打的补丁就没有了，除非你将补丁保存到二进制文件中。你可以看到补丁在起作用，回到 Olly 打开 Patch 窗口（点击 Pa 图标或 Ctrl+p):

Patches					
Address	Size	State	Old	New	Comment
004010EB	2	Active	JNZ SHORT Crackme2.0040110D	NOP	

Patch 窗口显示的是我们给程序打的所有补丁。注意地址是红色的，以及 State 列的“Active”。我们的程序仍然在运行，就意味着我们的补丁已经实现，如果 CPU 运行了这个代码，它运行的将是打过补丁的版本。现在，重启应用 (Ctrl+F2)。首先，01ly 可能会显示一个错误，一个很长很复杂的错误，基本上是告诉我们补丁（以及断点）没有“坚守”在原来位置，因为 01ly 无法追踪它们（其实比这个要复杂一点，我们后面会看到）。关掉那个窗口，打开断点窗口：

Breakpoints				
Address	Module	Active	Disassembly	Comment
004010EB	Crackme2	Disabled	JNZ SHORT Crackme2.0040110D	

看看我们的断点已经失效了😞。重新激活断点（空格键），01ly 会再次断在该断点。运行程序，输入用户名和密码，我们会停止我们前面打补丁的那一行（它上面的断点又可用了）：

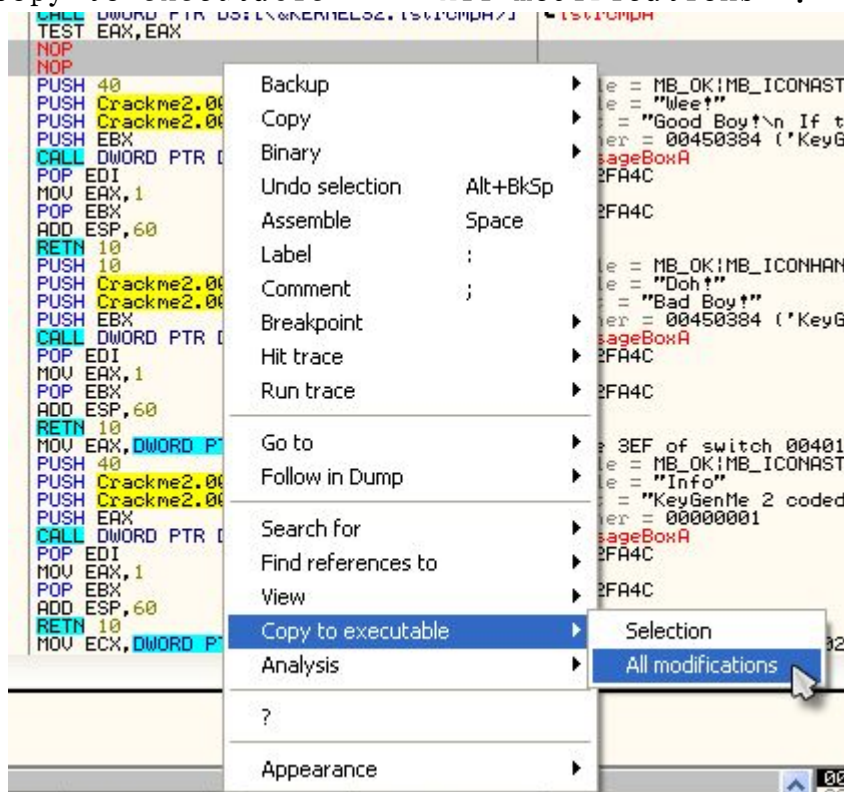
004010E2	:	52	PUSH EAX
004010E3	:	FF15 00704000	CALL DWORD PTR DS:[&KERNEL32.lstrcmpA]
004010E9	:	85C0	TEST EAX, EAX
004010EB	:	75 20	JNZ SHORT Crackme2.0040110D
004010ED	:	6A 40	PUSH 40
004010EF	:	68 E0714000	PUSH Crackme2.004071E0
004010F4	:	68 98714000	PUSH Crackme2.00407198
004010F9	:	53	PUSH EBX

看看，我们的两个 NOP 消失了，原始的代码又回来了（不过变成了灰色）。我们的补丁被回收了！现在回到 Patch 窗口：

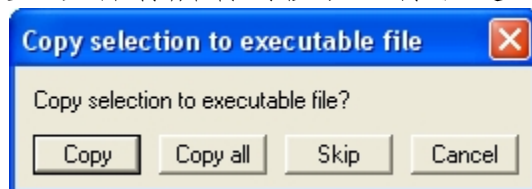
Patches					
Address	Size	State	Old	New	Comment
004010EB	2	Removed	JNZ SHORT Crackme2.0040110D	NOP	

注意那个地址不是红色的了，State 也变成了“Removed”。01ly 已经禁用了我们的补丁，并且在我们每一次重启程序时都会这么做。我们想要做的就是让这个补丁永远有效，而不用每一次都激活它。

为了让我们的补丁能够长久有效，我们必须将修改的版本保存到磁盘。首先，选中补丁再按下空格键以重新启用补丁。那个 JNZ 指令就会变回到我们的 NOP，那两个 NOP 也会以红色字体重新出现在反汇编窗口。现在，在反汇编窗口的任何地方右键，选择” Copy to executable “->” All modifications “:



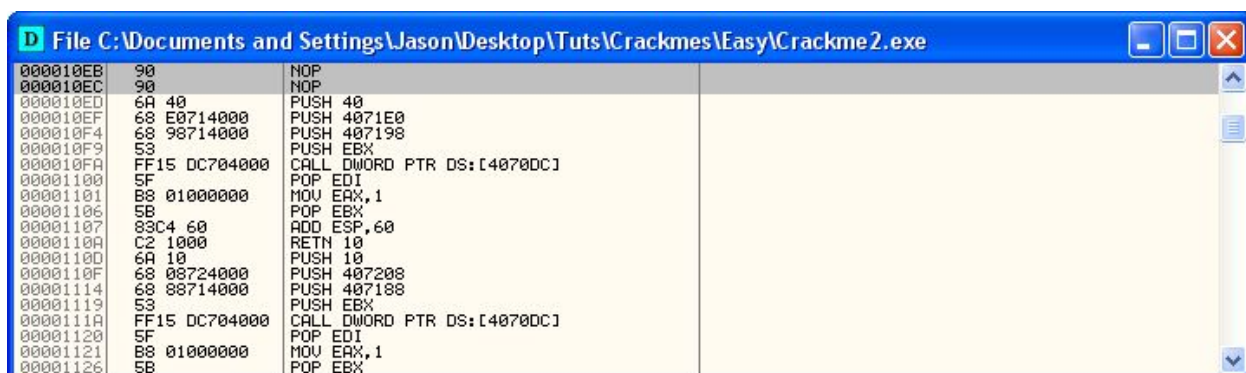
如果弹出窗口问你是否要保存所有的修改，你就选” Copy All “:



当你打了多个补丁，并且想一次性全部保存的话，这么做很重要。因为有时候，你很容易就忘记你打过多个补丁。本例中，即使我们只打了一个补丁，选择所有的补丁也只会保存这一个。当然，只有在 Patch 窗口中被激活的补丁才会被保存。

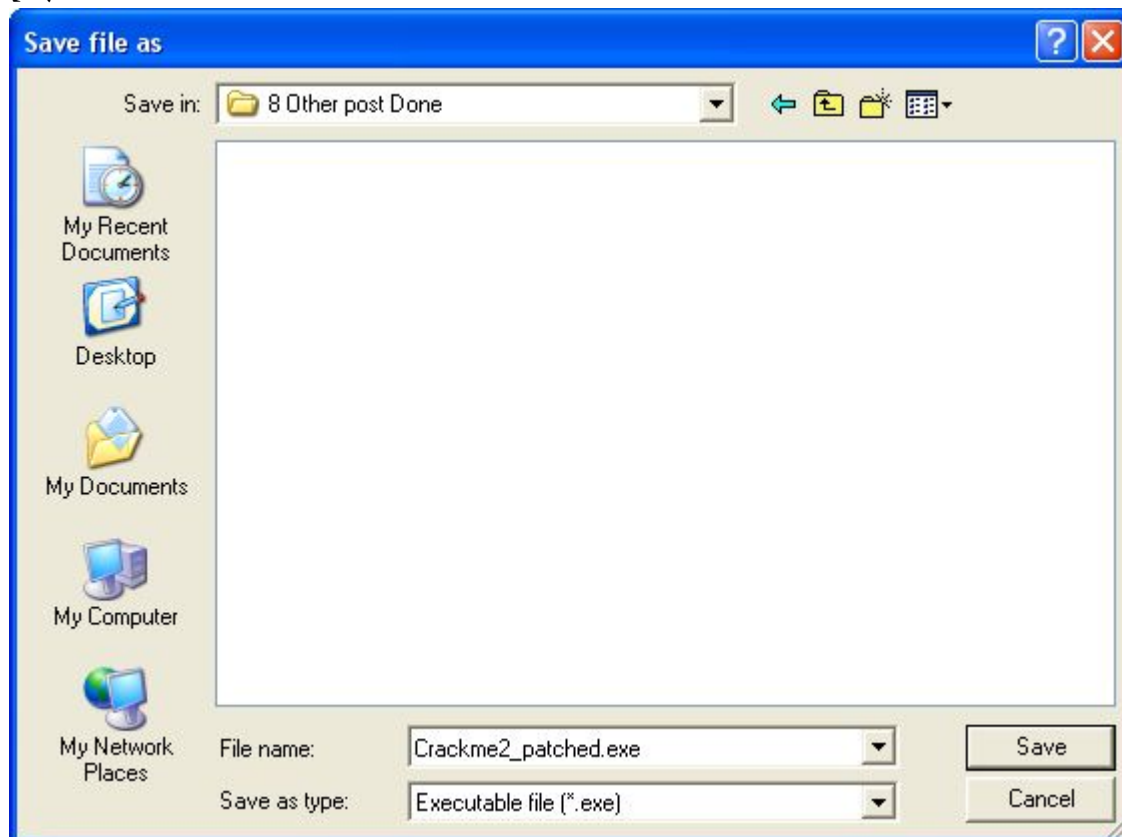
后面，你可能想要选择” Selection “而不是” All Modifications “，但是你必须保证你在反汇编窗口所做的修改是高亮显示(通过点击或拖拽以选中所有修改的行)。如果你选中的行比修改过的行要多也行，因为 Olly 只会更改已经修改的行。

在点击” Copy All “以后会打开一个新窗口，里面基本上是整个进程的数据，不过我们的补丁也在里面:

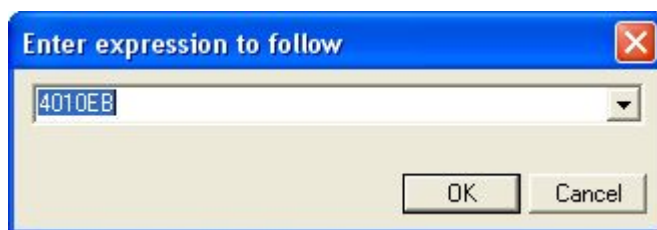


Address	Hex	Instruction
000010EB	90	NOP
000010EC	90	NOP
000010ED	6A 40	PUSH 40
000010EF	68 E0714000	PUSH 4071E0
000010F4	68 98714000	PUSH 407198
000010F9	53	PUSH EBX
000010FA	FF15 DC704000	CALL DWORD PTR DS:[4070DC]
00001100	5F	POP EDI
00001101	B8 01000000	MOV EAX,1
00001106	5B	POP EBX
00001107	83C4 60	ADD ESP,60
0000110A	C2 1000	RETN 10
0000110D	6A 10	PUSH 10
0000110F	68 08724000	PUSH 407208
00001114	68 88714000	PUSH 407188
00001119	53	PUSH EBX
0000111A	FF15 DC704000	CALL DWORD PTR DS:[4070DC]
00001120	5F	POP EDI
00001121	B8 01000000	MOV EAX,1
00001126	5B	POP EBX

在顶部你可以看到我们的补丁。但要意识到这个只是在内存中的修订版本，还没有保存到磁盘呢。不过，如果你关了这个窗口或重启了程序，它是不会被保存的！咱们来保存好它：右键新窗口的任意位置，选择” save file “。这会将该进程的内存空间数据保存到一个文件中。一个另存为对话框会显示出来。将文件另存为 Crackme2-patched（我通常在后门加一个” -patched “用来区分，你也可以加任何你喜欢的）：



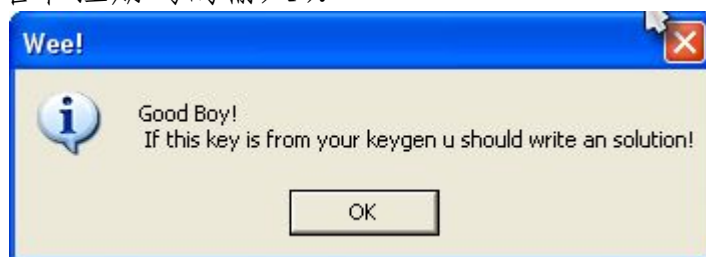
我们现在有了一个 crackme 的打补丁版本。咱们来试试看。在 011y 中打开这个新文件（打过补丁的）。按下 Ctrl+G 或点击 GOTO 图标，输入我们打过补丁的地址：



看看咱们的补丁:

004010E3	: FF15 00704000	CALL DWORD PTR DS:[&KERNEL32.l
004010E9	: 85C0	TEST EAX,EAX
004010EB	: 90	NOP
004010EC	: 90	NOP
004010ED	: 6A 40	PUSH 40
004010EF	: 68 E0714000	PUSH Crackme2.004071E0
004010F4	: 68 98714000	PUSH Crackme2.00407198
004010F9	: 53	PUSH EBX
004010FA	: FF15 DC704000	CALL DWORD PTR DS:[&USER32.Mess.

yes, 补丁还在那。现在运行程序, 输入 info 和 viola (译者注: info 和 viola 是作者用了当用户名和注册码的输入):



现在, 我们有了我们第一个破解过并打过补丁的二进制文件: 0。

五、作业

本章的作业很简单 (只要你一直在学习汇编语言 😊)。

思考题: 你可以将 4010E9 处的 "TEST EAX, EAX" 修改成什么, 来防止跳转到显示 bad boy 处?

要注意的是无论你将 TEST 指令修改成什么, 都不能超过 2 字节, 那是 TEST EAX, EAX 指令的长度。如果你打了一个长点的补丁, 就会覆盖掉 JNZ 指令后面的指令.....

ps. 如果你需要提示的话, 请点[这里](#)。不过你应该真正意义上自己试着做。那是学习的最好方法!