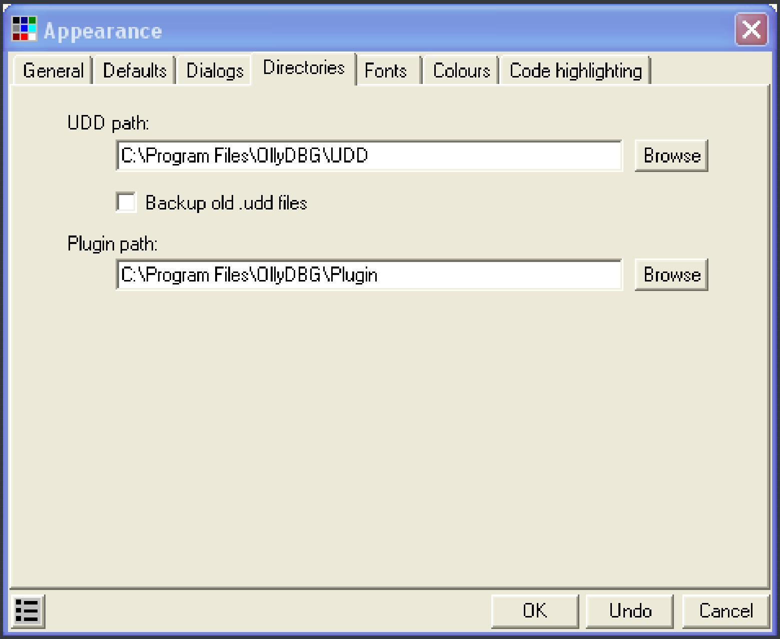
Tutorial #6: Our First (True) Crack

by R4ndom on Jun.11, 2012, under Reverse Engineering, Tutorials

Introduction

Welcome to Part 6 of my tutorial. In this tutorial we are going to get a little closer to the real thing: a real crackme. It is included in the download of this tutorial. Crackme's are a great way to take the incremental steps to learning reverse engineering as, instead of jumping into a 'real' program (having no idea the difficulty of reversing it) crackmes can be ordered from easy to hard, so you can learn in a linear fashion. Eventually, we will work out way up to real programs, but seeing as we're still just getting started, these crackmes should give us plenty of challenge.

We will be using OllyDBG 1.10 (either my version or the original, though if you use mine it will look like the pictures 😃). I recommend that you download the plugin "MnemonicHelp" from the tools page under Olly Plugins as I will be referencing it in this tutorial (it is also included in the download of this tut). Unzip it and put it and the x86eas.hlp file into your plugins directory in the Olly folder. If there isn't a plugin folder, create one in the main Olly folder. You will then need to go to Options->Appearance-> Directories tab in Olly and select the directory where you placed your plugin. While you're there, you may as well create a directory in the main Olly folder called "UDD" and point the other option on this setting page to point to that folder as well. UDD files are Olly's 'notes' on an app, so every breakpoint you set, comment you make, and specific setting for that binary will be stored in the UDD file, usually called "AppName.UDD". These UDD files are a lifesaver if you want to take a break and come back to reversing an app, as everything will be saved. Here's the window where you set the two directories (along with my settings):



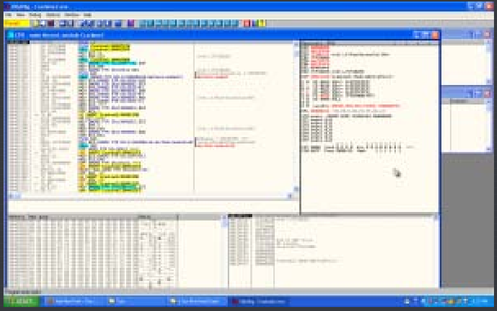
UDD path:
C:\Program Files\OllyDBG\UDD

Backup old .udd files

Plugin path:
C:\Program Files\OllyDBG\Plugin

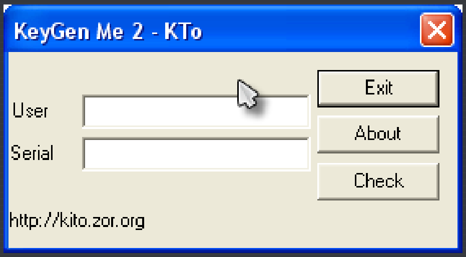
Investigating the binary

Go ahead and load up Crackme2.exe:



As I have said before, one of the most important things you can do before getting started is running the app and studying it. It gives you a plethora of information; is there a time trial? Are certain features disabled? Are there a certain amount of times it can be run? Is there a registry screen that you can enter a registration code?

These are all really important things to know, and as you get better in reverse engineering, you will gain more and more experience as to what you should be looking for (how long did it take to validate the code? Is it forcing you to a web site?…)



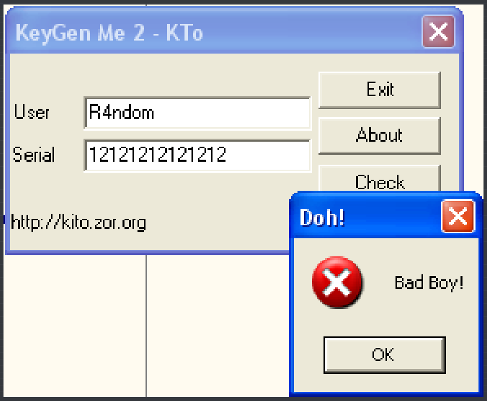
KeyGen Me 2 - KTo

User

Serial

http://kito.zor.org

Seems pretty straight forward. Let try it:



KeyGen Me 2 - KTo

User R4ndom

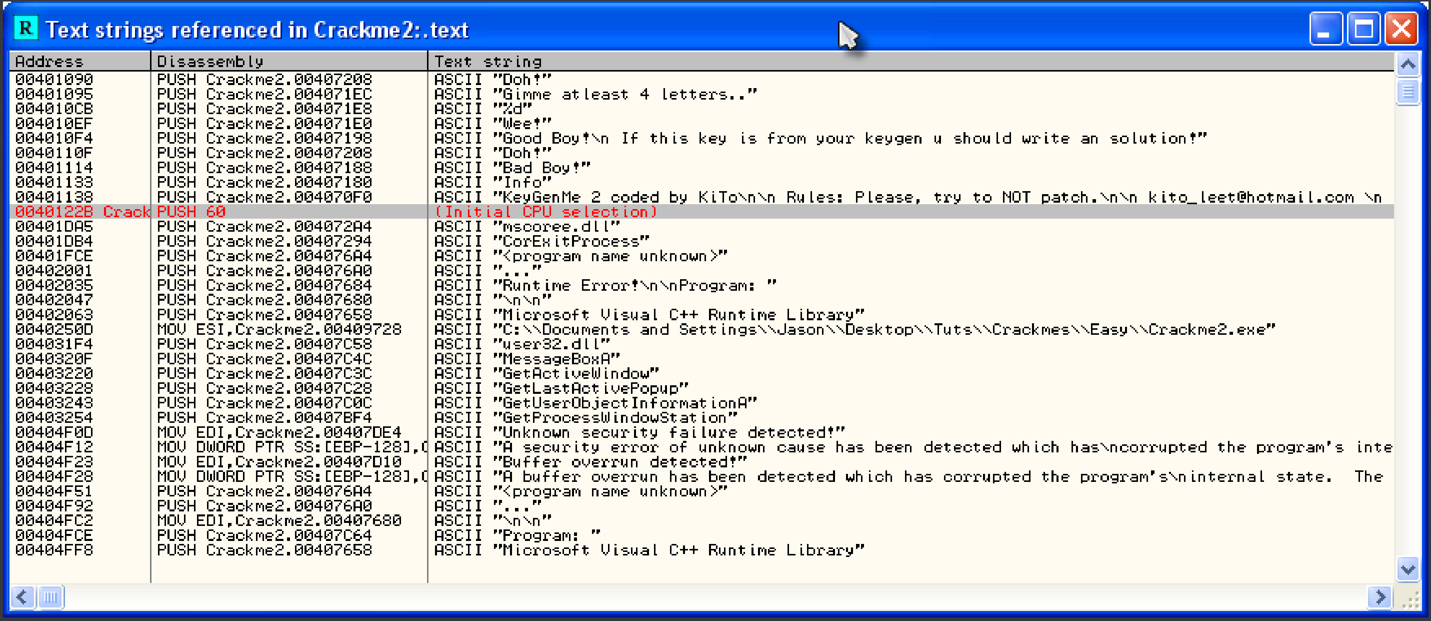
Serial 12121212121212

http://kito.zor.org

Doh!

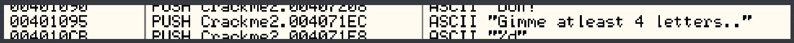
Bad Boy!

That's not what we want. Let's see if we can help Olly do the right thing. Go back to Olly and let's try our first (and only) tool we currently know. Let's search for text strings. Right-click->"Search For"->"All Referenced Text Strings":



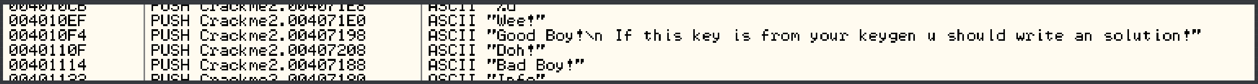
```
Text strings referenced in Crackme2:.text

Address   Disassembly                     Text string
00401090  PUSH Crackme2.00407208          ASCII "Doh!"
00401095  PUSH Crackme2.004071EC          ASCII "Gimme atleast 4 letters.."
004010CB  PUSH Crackme2.004071E8          ASCII "%d"
004010EF  PUSH Crackme2.004071E0          ASCII "Wee!"
004010F4  PUSH Crackme2.00407198          ASCII "Good Boy!\n If this key is from your keygen u should write an solution!"
0040110F  PUSH Crackme2.00407208          ASCII "Doh!"
00401114  PUSH Crackme2.00407188          ASCII "Bad Boy!"
00401133  PUSH Crackme2.00407180          ASCII "Info"
00401138  PUSH Crackme2.004070F0          ASCII "KeyGenMe 2 coded by KiTo\n\n Rules: Please, try to NOT patch.\n\n kito_leet@hotmail.com \n
0040122B Crack PUSH 60                    (Initial CPU selection)
00401DA5  PUSH Crackme2.004072A4          ASCII "mscoree.dll"
00401DB4  PUSH Crackme2.00407294          ASCII "CorExitProcess"
00401FCE  PUSH Crackme2.004076A4          ASCII "<program name unknown>"
00402001  PUSH Crackme2.004076A0          ASCII "..."
00402035  PUSH Crackme2.00407684          ASCII "Runtime Error!\n\nProgram: "
00402047  PUSH Crackme2.00407680          ASCII "\n\n"
00402063  PUSH Crackme2.00407658          ASCII "Microsoft Visual C++ Runtime Library"
0040250D  MOV ESI,Crackme2.00409728       ASCII "C:\\Documents and Settings\\Jason\\Desktop\\Tuts\\Crackmes\\Easy\\Crackme2.exe"
004031F4  PUSH Crackme2.00407C58          ASCII "user32.dll"
0040320F  PUSH Crackme2.00407C4C          ASCII "MessageBoxA"
00403220  PUSH Crackme2.00407C3C          ASCII "GetActiveWindow"
00403228  PUSH Crackme2.00407C28          ASCII "GetLastActivePopup"
00403243  PUSH Crackme2.00407C0C          ASCII "GetUserObjectInformationA"
00403254  PUSH Crackme2.00407BF4          ASCII "GetProcessWindowStation"
00404F0D  MOV EDI,Crackme2.00407DE4       ASCII "Unknown security failure detected!"
00404F12  MOV DWORD PTR SS:[EBP-128],(    ASCII "A security error of unknown cause has been detected which has\ncorrupted the program's inte
00404F23  MOV EDI,Crackme2.00407D10       ASCII "Buffer overrun detected!"
00404F28  MOV DWORD PTR SS:[EBP-128],(    ASCII "A buffer overrun has been detected which has corrupted the program's\ninternal state.  The
00404F51  PUSH Crackme2.004076A4          ASCII "<program name unknown>"
00404F92  PUSH Crackme2.004076A0          ASCII "..."
00404FC2  MOV EDI,Crackme2.00407680       ASCII "\n\n"
00404FCE  PUSH Crackme2.00407C64          ASCII "Program: "
00404FF8  PUSH Crackme2.00407658          ASCII "Microsoft Visual C++ Runtime Library"
```

This looks promising. There are several things of note here. The first is that we now know that the serial requires at least 4 characters:



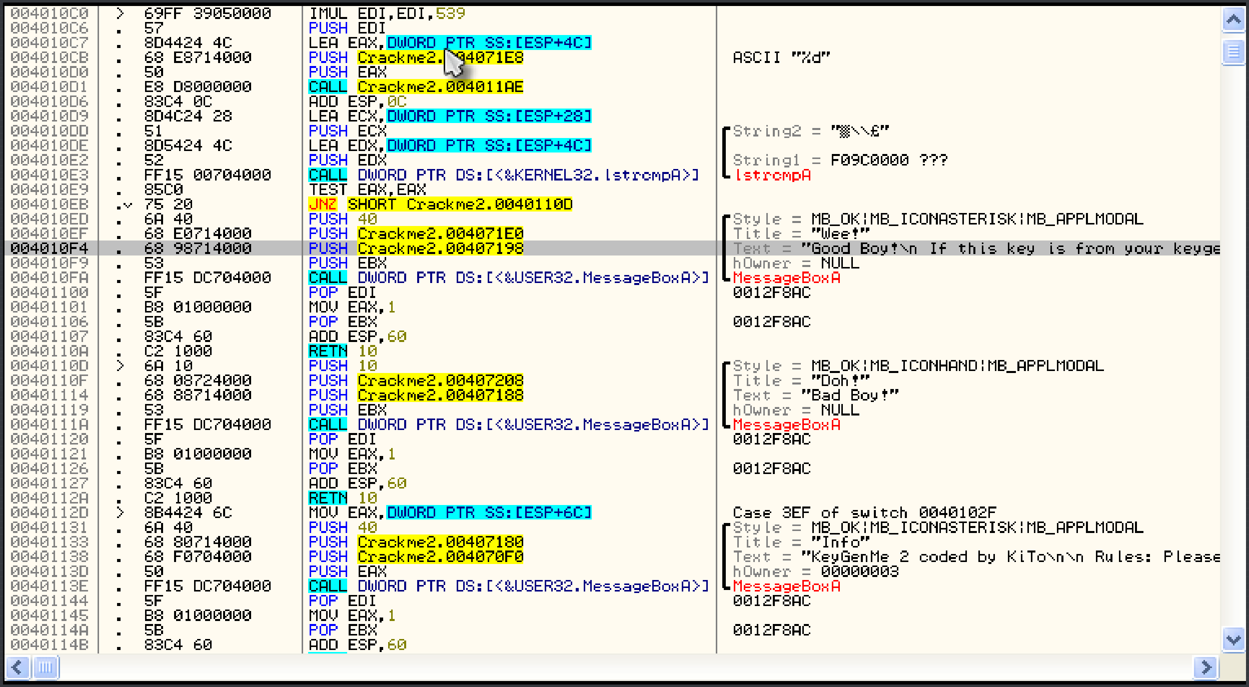
```
00401095  PUSH Crackme2.004071EC          ASCII "Gimme atleast 4 letters.."
```

and the second is we now know exactly where the good and bad messages are displayed:



```
004010EF  PUSH Crackme2.004071E0          ASCII "Wee!"
004010F4  PUSH Crackme2.00407198          ASCII "Good Boy!\n If this key is from your keygen u should write an solution!"
0040110F  PUSH Crackme2.00407208          ASCII "Doh!"
00401114  PUSH Crackme2.00407188          ASCII "Bad Boy!"
```

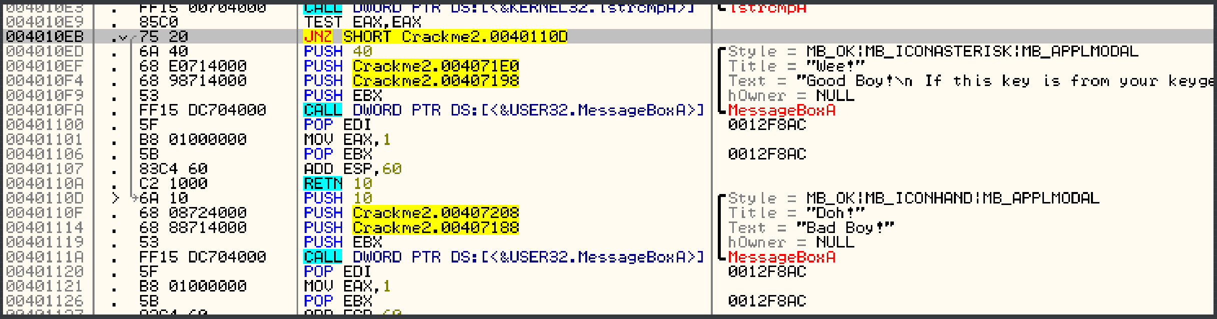
So, let's click on the good boy at address 4010F4 and see what we got:



```
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                     ASCII "%d"
004010D0  .  50              PUSH EAX
004010D1  .  E8 D8000000     CALL Crackme2.004011AE
004010D6  .  83C4 0C         ADD ESP,0C
004010D9  .  8D4C24 28       LEA ECX,DWORD PTR SS:[ESP+28]
004010DD  .  51              PUSH ECX                                  [String2 = "▒\\£"
004010DE  .  8D5424 4C       LEA EDX,DWORD PTR SS:[ESP+4C]             |
004010E2  .  52              PUSH EDX                                  |String1 = F09C0000 ???
004010E3  .  FF15 00704000   CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]  [lstrcmpA
004010E9  .  85C0            TEST EAX,EAX
004010EB  .v 75 20           JNZ SHORT Crackme2.0040110D
004010ED  .  6A 40           PUSH 40                                   [Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004010EF  .  68 E0714000     PUSH Crackme2.004071E0                    |Title = "Wee!"
004010F4  .  68 98714000     PUSH Crackme2.00407198                    |Text = "Good Boy!\n If this key is from your keyge
004010F9  .  53              PUSH EBX                                  |hOwner = NULL
004010FA  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] [MessageBoxA
00401100  .  5F              POP EDI                                   0012F8AC
00401101  .  B8 01000000     MOV EAX,1
00401106  .  5B              POP EBX                                   0012F8AC
00401107  .  83C4 60         ADD ESP,60
0040110A  .  C2 1000         RETN 10
0040110D  >  6A 10           PUSH 10                                   [Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
0040110F  .  68 08724000     PUSH Crackme2.00407208                    |Title = "Doh!"
00401114  .  68 88714000     PUSH Crackme2.00407188                    |Text = "Bad Boy!"
00401119  .  53              PUSH EBX                                  |hOwner = NULL
0040111A  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] [MessageBoxA
00401120  .  5F              POP EDI                                   0012F8AC
00401121  .  B8 01000000     MOV EAX,1
00401126  .  5B              POP EBX                                   0012F8AC
00401127  .  83C4 60         ADD ESP,60
0040112A  .  C2 1000         RETN 10
0040112D  >  8B4424 6C       MOV EAX,DWORD PTR SS:[ESP+6C]             Case 3EF of switch 0040102F
00401131  .  6A 40           PUSH 40                                   [Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401133  .  68 80714000     PUSH Crackme2.00407180                    |Title = "Info"
00401138  .  68 F0704000     PUSH Crackme2.004070F0                    |Text = "KeyGenMe 2 coded by KiTo\n\n Rules: Please
0040113D  .  50              PUSH EAX                                  |hOwner = 00000003
0040113E  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] [MessageBoxA
00401144  .  5F              POP EDI                                   0012F8AC
00401145  .  B8 01000000     MOV EAX,1
0040114A  .  5B              POP EBX                                   0012F8AC
0040114B  .  83C4 60         ADD ESP,60
```

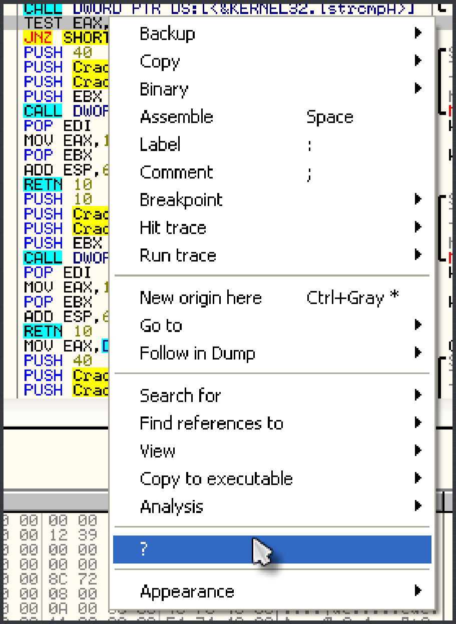
This is a pretty standard process when working with easier crackmes (and easier commercial programs as well). You do a search for referenced text strings, you find a message that is displayed whether you got the registration code/password/license number right or wrong, you go to that part of the code, and you see both the good and bad messages pretty close to each other. And then, according to R.E.T.A.R.D. rule #2, you search for the compare/jump that calls the one you want. Let's find that jump.

The first jump we find is at address 4010EB, a JNZ statement. If we click on this line, Olly will be so kind as to show us where it will jump:

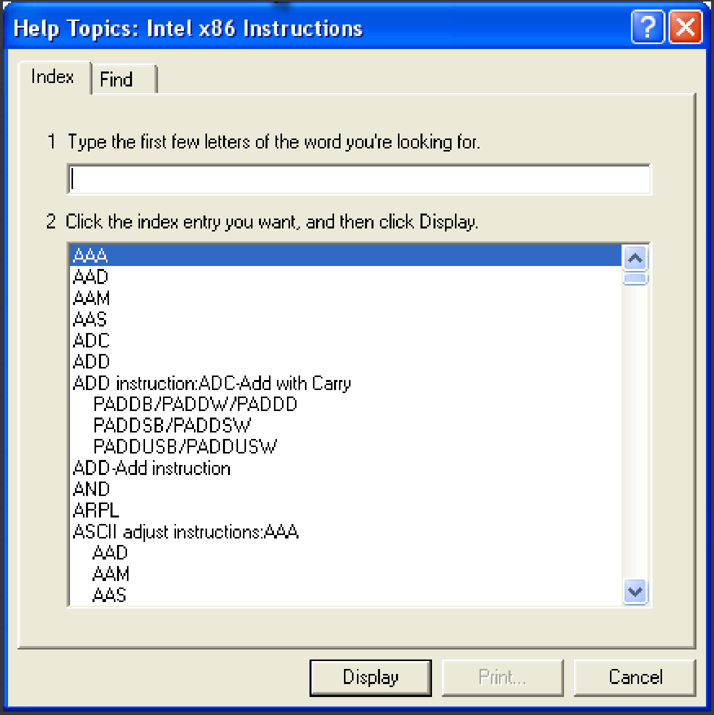


```
004010E3  .  FF15 00704000   CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]  └lstrcmpA
004010E9  .  85C0            TEST EAX,EAX
004010EB  .v 75 20           JNZ SHORT Crackme2.0040110D
004010ED  .  6A 40           PUSH 40                                   ┌Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004010EF  .  68 E0714000     PUSH Crackme2.004071E0                    │Title = "Wee!"
004010F4  .  68 98714000     PUSH Crackme2.00407198                    │Text = "Good Boy!\n If this key is from your keyge
004010F9  .  53              PUSH EBX                                  │hOwner = NULL
004010FA  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] └MessageBoxA
00401100  .  5F              POP EDI                                    0012F8AC
00401101  .  B8 01000000     MOV EAX,1
00401106  .  5B              POP EBX                                    0012F8AC
00401107  .  83C4 60         ADD ESP,60
0040110A  .  C2 1000         RETN 10
0040110D  >  6A 10           PUSH 10                                   ┌Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
0040110F  .  68 08724000     PUSH Crackme2.00407208                    │Title = "Doh!"
00401114  .  68 88714000     PUSH Crackme2.00407188                    │Text = "Bad Boy!"
00401119  .  53              PUSH EBX                                  │hOwner = NULL
0040111A  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] └MessageBoxA
00401120  .  5F              POP EDI                                    0012F8AC
00401121  .  B8 01000000     MOV EAX,1
00401126  .  5B              POP EBX                                    0012F8AC
```

As we can see, this instruction jumps the good boy and goes directly to the bad boy. This seem like a prime place to start. We also know that prior to a jump there is usually a compare to determine if the jump is taken or not. Looking above the JNZ instruction we see a TEST EAX, EAX. Seeing as you may not have gotten to the TEST instruction in your assembly book yet, let's see if we can find out what this TEST does. Since, at the beginning of this tutorial you installed the MnemonicHelp plugin, that's what we'll use. Right click on the TEST instruction and you should see a question mark as one of the entries in the context menu. Choose this:



That will open the Mnemonic Help window:

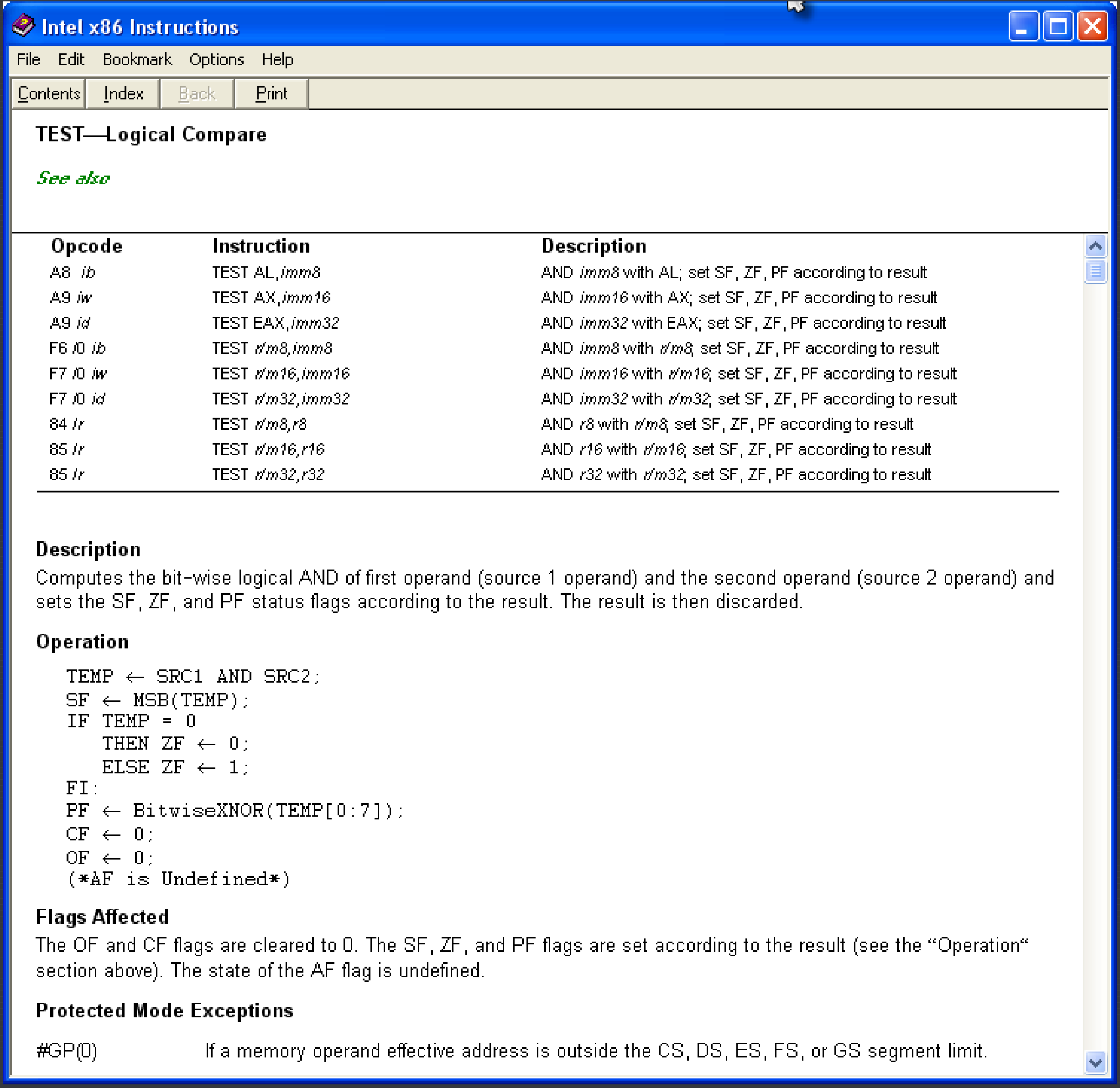


1 Type the first few letters of the word you're looking for.

2 Click the index entry you want, and then click Display.

```
AAA
AAD
AAM
AAS
ADC
ADD
ADD instruction:ADC-Add with Carry
   PADDB/PADDW/PADDD
   PADDSB/PADDSW
   PADDUSB/PADDUSW
ADD-Add instruction
AND
ARPL
ASCII adjust instructions:AAA
   AAD
   AAM
   AAS
```

Type “test” into the top bar and choose (double-click) “TEST”. This will bring up help on that mnemonic:



TEST—Logical Compare

See also

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8*,*imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16*,*imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32*,*imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8*,*r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16*,*r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32*,*r32*	AND *r32* with *r/m32*; 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*)
```

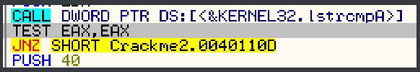
Flags Affected

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

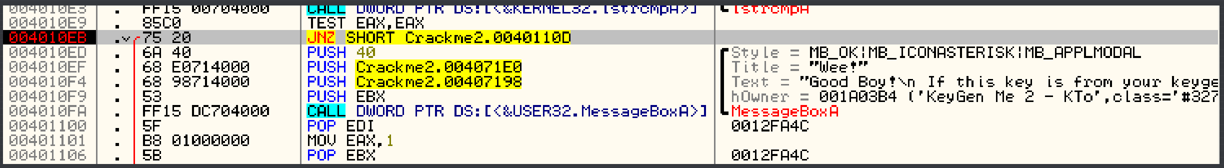
#GP(0)   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

As we can see, the TEST instruction *"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."* Most of the time, if the test instruction is testing two registers that are the same, it means it's checking whether it's a zero or not. So this definitely fills our requirement of a compare before the jump:



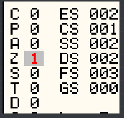
```
CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]
TEST EAX,EAX
JNZ SHORT Crackme2.0040110D
PUSH 40
```

What these two statements mean is "If EAX does not equal zero, jump to 40110D", which is our bad boy. Well, we definitely don't want that, so let's test our hypothesis. Place a breakpoint on the JNZ instruction and restart the app. Enter a username and serial number (remember, at least four characters 😃) and click check in the crackme. Olly will then break at our BP:



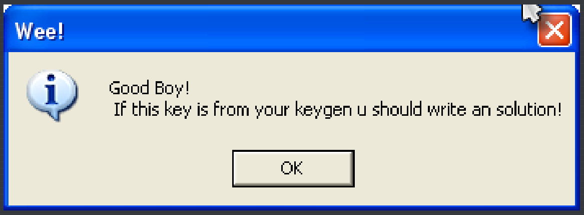
```
004010E3  .  FF15 00704000  CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]   └lstrcmpA
004010E9  .  85C0           TEST EAX,EAX
004010EB  .v 75 20          JNZ SHORT Crackme2.0040110D
004010ED  .  6A 40          PUSH 40                                   ┌Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004010EF  .  68 E0714000    PUSH Crackme2.004071E0                    │Title = "Wee!"
004010F4  .  68 98714000    PUSH Crackme2.00407198                    │Text = "Good Boy!\n If this key is from your keyge
004010F9  .  53             PUSH EBX                                  │hOwner = 001A03B4 ('KeyGen Me 2 - KTo',class='#327
004010FA  .  FF15 DC704000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>] └MessageBoxA
00401100  .  5F             POP EDI                                    0012FA4C
00401101  .  B8 01000000    MOV EAX,1
00401106  .  5B             POP EBX                                    0012FA4C
```

Now, we can see that we are going to jump past the good boy, straight into the arms of the bad boy. Let's not let that happen. Help Olly out by flipping the zero flag (see previous tuts):



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

And we see that we are now not taking the jump. Run the app and:



Wee!

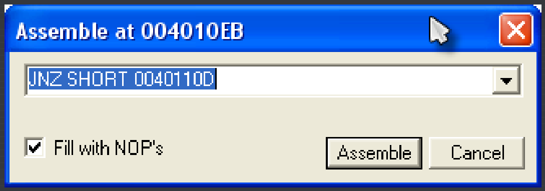
Good Boy!
If this key is from your keygen u should write an solution!

Yep, exactly what we wanted. ****ignore the message about the keygen- some of these crackmes are for other purposes, but I am using them because they highlight things we need to learn as well. Many of them we will come back to and use as intended, once we get some more knowledge under our belt* 😃

Patching

Restart the crackme, run it, enter the name and serial, and Olly will break at our trusted BPs. You will notice that, again, we are taking the jump to the bad boy, as changing a flag in Olly is only temporary. Now this time, instead of temporarily changing a flag, we are going to change the actual code in the binary to do what we want. This is called a patch.

Click on the line we are paused at (address 4010EB) click on the instruction column of the line (the part that has JNZ SHORT…) and press the space bar. You will see a window pop up that shows us the instructions at that line, as well as a dialog to change them:

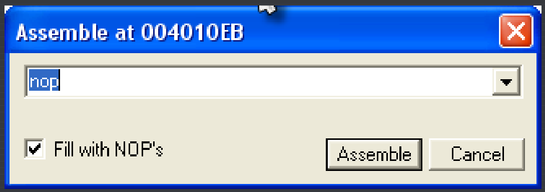


Assemble at 004010EB

JNZ SHORT 0040110D

☑ Fill with NOP's

Now, what we want to do is change this from jumping to the bad boy message, to NEVER jumping- meaning we really don't even want this instruction performed. So what we are going to do is replace it with an instruction that does nothing, the NOP instruction. NOP stands for No OPeration. Go into the dialog window with the instruction in it and change the JNZ SHORT 0040110D to NOP:



Assemble at 004010EB

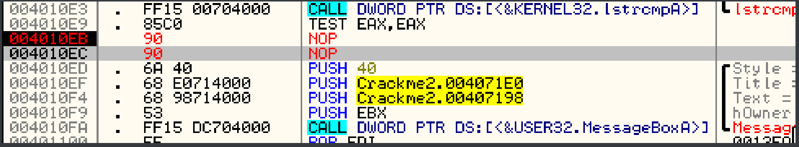
nop

☑ Fill with NOP's

You can leave the "Fill with NOP's" checked. Now click Assemble, to commit that line, and then cancel to close the window.

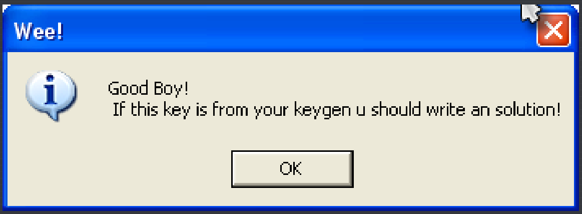
****btw, if you did not click cancel and kept clicking Assemble, you would assemble each line, one after the other. This is a 'feature' of Olly and it is for when you want to replace several lines of code. It keeps you from having to hit the space bar on every line. I guarantee that when you first start patching this will drive you nuts :X)*

You will notice that the line we are paused on has been changed- the instruction now shows two NOPs instead of the JNZ instruction and they are in red (because Olly shows items that have been changed in red):



```
004010E3  .  FF15 00704000   CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]  └lstrcmp
004010E9  .  85C0            TEST EAX,EAX
004010EB     90              NOP
004010EC     90              NOP
004010ED  .  6A 40           PUSH 40                                   ┌Style =
004010EF  .  68 E0714000     PUSH Crackme2.004071E0                    │Title =
004010F4  .  68 98714000     PUSH Crackme2.00407198                    │Text =
004010F9  .  53              PUSH EBX                                  │hOwner
004010FA  .  FF15 DC704000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>] └Messag
```

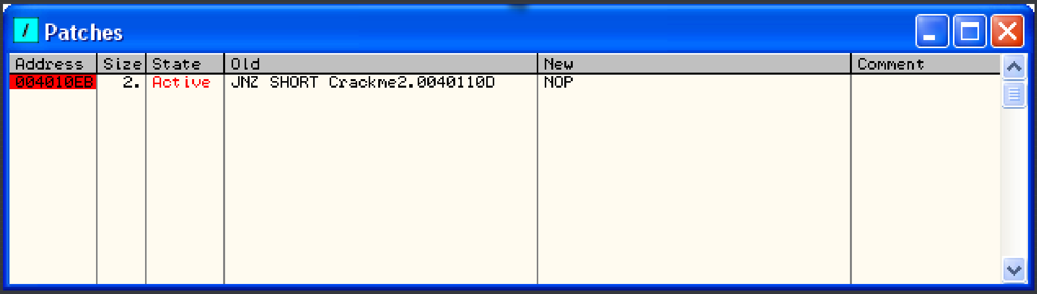
The reason for the two NOPs is because the NOP opcode is only one byte long and the the statement we replaced, the JNZ, is two bytes, so Olly replaced both bytes with NOPs. You will also notice that the jump arrow has disappeared; this is because there is no longer any jump in this line! Now single step and you will make your way into the good boy. And the good boy is displayed. And your smile grows 😀



```
Wee!

Good Boy!
 If this key is from your keygen u should write an solution!

OK
```

Saving The Patch

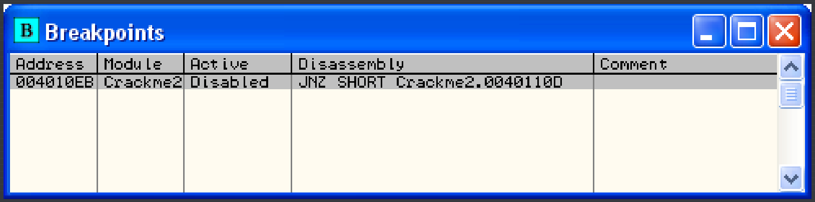
One important thing to know is that your patch will not stay in place if you reload or re-start the app until you save it back to the binary. You can see this in action- click back in Olly and open the Patch Window (the “Pa” icon or ctrl-P):



Patches

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

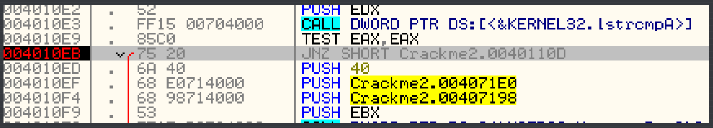
The Patches Window shows all of the patches we have made in our app. Notice that the address is red and there is the word “Active” in the state column. As our app is still running, this means that this patch is currently implemented and if the CPU runs this code, it will run the patched version. Now, re-start the app (ctrl-F2). First of all, Olly may bring up an error, a very long, complicated error that basically tells us our patch (and our breakpoints) may not “stick” because Olly can’t keep track of them (it’s a little more complicated than this, but we’ll see later). Close the window and then go to the breakpoints window:



Breakpoints

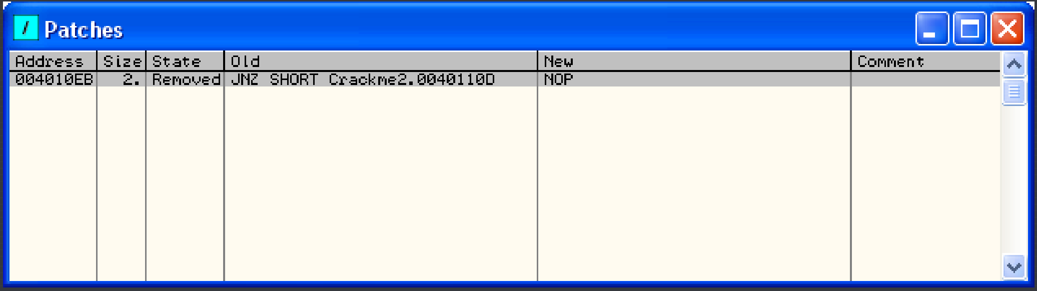
Address	Module	Active	Disassembly	Comment
004010EB	Crackme2	Disabled	JNZ SHORT Crackme2.0040110D	

You see that our BP has been disabled 😞 Re-activate the BP (space bar) and now Olly will break on it again. Run the app, enter the username and password and we will stop on the line we previously patched (with our breakpoint re-enabled on it):



```
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
```

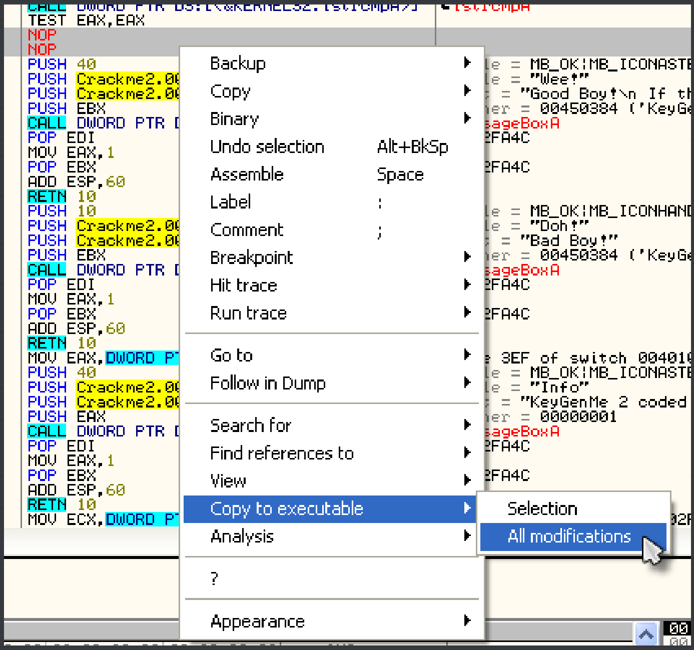
As you can see, our two NOPs are gone and the original code is back in (but grey this time). Our patch has been reverted back! Now, go back to the Patches Window:



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

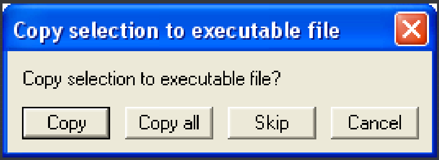
You will notice that the address is no longer red and the State column lists it as “Removed”. Olly has disabled our patch, and will do this every time we re-start the app. What we want to do is permanently apply this patch so that we don’t need to activate it every time.

In order to keep our patch permanent, we must save the changed version to the binary on disk. First of all, re-enable the patch by clicking on it and hitting the space bar. The JNZ instruction should change back to our NOPs, and the NOPs will re-appear in the disassembly window in red. Now, right-click anywhere in the disassembly window and select “Copy to executable” and choose “All modifications”:



```
TEST EAX,EAX
NOP
NOP
PUSH 40
PUSH Crackme2.00
PUSH Crackme2.00
PUSH EBX
CALL DWORD PTR D
POP EDI
MOV EAX,1
POP EBX
ADD ESP,60
RETN 10
PUSH 10
PUSH Crackme2.00
PUSH Crackme2.00
PUSH EBX
CALL DWORD PTR D
POP EDI
MOV EAX,1
POP EBX
ADD ESP,60
RETN 10
MOV EAX,DWORD PT
PUSH 40
PUSH Crackme2.00
PUSH Crackme2.00
PUSH EAX
CALL DWORD PTR D
POP EDI
MOV EAX,1
POP EBX
ADD ESP,60
RETN 10
MOV ECX,DWORD PT
```

Select “copy all” if it asks you if you want to save all of the modifications:

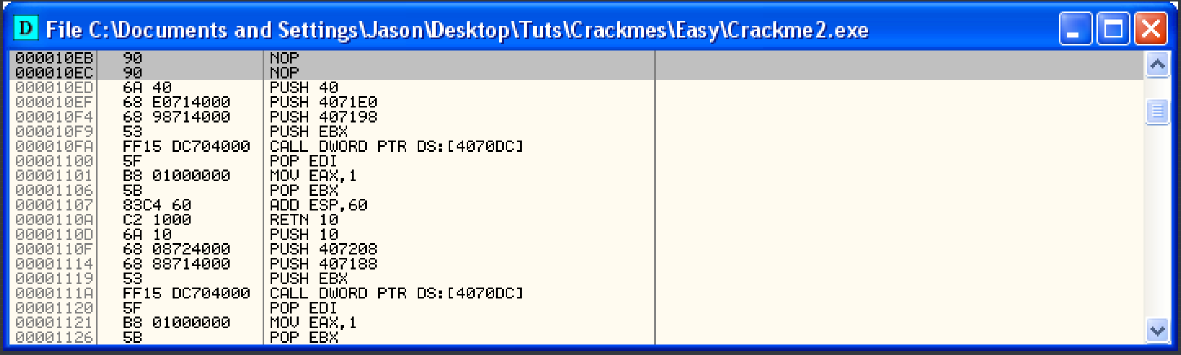


Copy selection to executable file?

This will be important when you implement multiple patches and want to save them all at once, as sometimes it’s easy to forget you’ve created multiple patches. In this case, even though we only have one patch, selecting all patched will still only save our one. Of course, only patches that are set to active in the Patches Window will be saved.

Later, you may want to choose “Selection” instead of “All modifications”, but you must make sure your modification is highlighted in the disassembly window (by clicking and dragging all of the lines with modifications in it). It is OK if you select more than just the lines that have been modified- Olly will only change the modified lines.

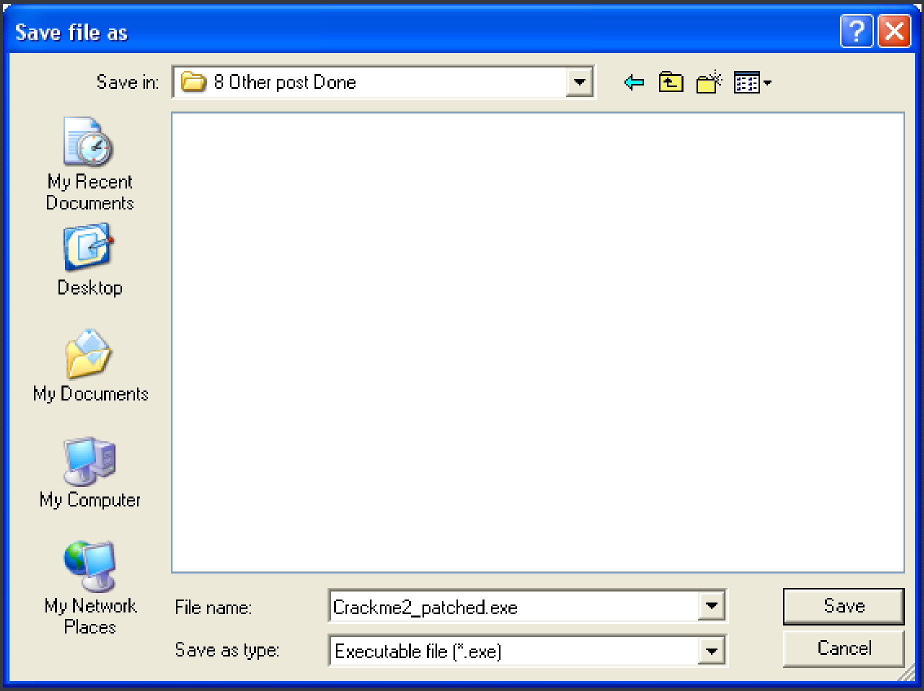
After clicking “Copy all” a new window will open that is basically a dump of the entire process but with our patch in it:



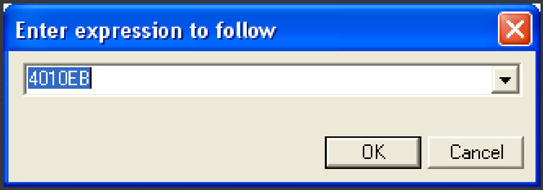
File C:\Documents and Settings\Jason\Desktop\Tuts\Crackmes\Easy\Crackme2.exe

```
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
```

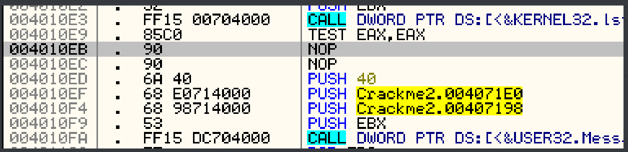
You can see our patch at the top. But realize that this is just a revised version of our executable *in memory*- it has not been saved to disk yet, therefore, if you close this window or re-start the app, it will not be saved! Let's save it for good: Right-click anywhere in this new window and select "save file". This will save this memory process's space to an actual file. A save dialog will show up. Save the file as Crackme2_patched (I usually add a "_patched" at the end to keep track but you can add whatever you want):


Save file as
Save in: 8 Other post Done
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
File name: Crackme2_patched.exe
Save as type: Executable file (*.exe)
Save
Cancel

We now have a patched version of our crackme. Let's try it out. Open this new file in Olly (the patched one) . Click ctrl-G or hit the GOTO icon and enter our patched address:

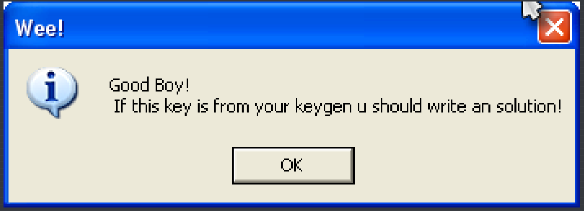


and now look at our patch:



```
004010E3  .  FF15 00704000   CALL DWORD PTR DS:[<&KERNEL32.ls
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
```

Yup, there's the patch. Now run the app, enter the info and viola:


Wee!

Good Boy!
If this key is from your keygen u should write an solution!

We now have a our first cracked and patched binary :0

Homework

The homework in this tutorial is very straightforward (as long as you have been studying your assembly language 😉).

Here's the question: "What could you change the instruction "TEST EAX, EAX" at address 4010E9 to in order to keep the jump from showing our bad boy message?"

Keep in mind that whatever you change the TEST instruction to, it cannot be more than 2 bytes, as that's how long the TEST EAX, EAX instruction is, and if you put in a longer patch, it will overwrite the JNZ instruction right after it…

-till next time

R4ndom

ps. If you need a hint, you can click here, though you should really try yourself first. That is the best way to learn!