Patching an Executable

We've seen how IDA Pro allows us to reverse engineer executables within its environment (e.g., editing the Z bit). However, it would be nice if executables could be modified so that execution changes could be written permanently. For example, entering any password in the password program would work, or the countdown would permanently begin at 10 instead of 2 million. Formally, this is called patching an executable. You may have seen this sort of thing back when many games and applications were copy-protected and required entering information only known to those that actually purchased them.

First, compile countdown.cc, then run the executable. Note that this C++ version is exactly the same as the Java version used in a previous example. As before, compilation can be done on whatever OS you are currently using. However, to reverse engineer it, you will need to compile it on Windows.

Before IDA Pro can patch an executable, we must edit its configuration and enable it to do so:

(1) Using an editor (e.g., notepad), open **idagui.cfg** (probably in C:\Program Files\IDA Free\cfg), and modify it so that **DISPLAY_PATCH_SUBMENU** is set to **Yes**
(2) You will also need to make sure the the file **idaPatcher.exe**[1] is in the same folder as the executable that you wish to patch

Once compiled, launch IDA Pro, and open the executable.

To reverse engineer:

(1) Search for text (via Alt+T) and search for the string "countdown:"
(2) Select the first occurrence of the string
(3) Note the flow of the program (especially the branch that leads to "The password is"), and set a breakpoint at the jump (JG) instruction immediately before this (via F2)
(4) To prevent the window from disappearing so quickly, set a breakpoint at the end of the branch that leads to "The password is" (at the JMP instruction)
(5) Run the debugger (via F9), and continue through several ticks of the countdown
(6) After a few, modify the Z bit so that the other branch is taken, ending the countdown
(7) Since modifying the Z bit is not something that can actually be patched in an executable (it's a runtime flag), we need to find another way (how about changing the initial 2000000 value)
(8) Bring up the hex view (the Hex View-A tab) and search for the following hex bytes (via Alt+B) **in hex**: 1E8480 (which is 2000000)
(9) You'll note that the value is backwards because it is represented in Little Endian format (the least significant byte is placed at the byte with the lowest address)
(10) Let's change it to something smaller (say, 10, or 00000A – but in Little Endian format: 0A0000)
(11) Change the bytes 80841E to 0A0000 via Edit | Patch program | Change byte
(12) Note that this changes the memory space only and not the actual executable; therefore, debugging with the change won't do anything
(13) Save the changes made to a DIF file (countdown.dif) via File | Produce file | Create DIF file...
(14) Close IDA Pro
(15) Open the DIF file in an editor (e.g., notepad) and note that it just lists the byte differences that reflect the changes that were just made, then close the editor
(16) At the command line (Start | Command Prompt or Start | Run | cmd) and in the same folder as the executable, dif file, and idaPatcher.exe, patch the executable via **idaPatcher.exe -i**

1 There are many versions of patchers out there, including one in Python that is platform independent

countdown.exe -p countdown.dif

(17) Execute the patched executable and note that the countdown now begins at 10 instead of 2 million

And now we've patched an executable so that its behavior is changed! The information only available after a countdown of 2 million seconds is now available after only 10 seconds. And this, folks, is reverse engineering!

In a previous example, we modified the Z bit to allow any password to be entered in a program that prompted us for a password. Let's try to patch it now so that the executable will allow any password to be entered. Note that it is not as simple as just changing a value! To begin, open up the executable in IDA Pro and find the prompt for the password. From there, find the point that can branch to either SUCCESS or FAILURE. Note the jump (JZ) instruction that's there. It's in a subroutine that begins at memory address 401ABD.

Before we begin making changes, we must understand the flow of the program. Well, at least this part of the program. The subroutine that begins at memory address 401ABD compares the value entered for the password against the actual password. The JZ instruction jumps if the zero bit is set (i.e., the passwords don't match). If the passwords match, control simply continues past the JZ instruction to the SUCCESS block. At the end of the SUCCESS block, a JMP instruction jumps around the FAILURE BLOCK that begins at memory address 401AE1 and goes directly to memory address 401AFD. If the passwords don't match, control transfers to memory address 401AE1 (i.e., the FAILURE BLOCK). Once the block is finished executing, control continues to memory address 401AFD. The subroutine that begins at memory address 401AFD occurs whether the passwords match or not, and eventually leads to the end of the program.

Here's a summary of the flow of this part of the program:

```
401ABD      compare passwords
401AC1      JZ 401AE1
401AC3      begin the SUCCESS block
   ...
401AE0      end the SUCCESS block (JMP 401AFD)
401AE1      begin the FAILURE block
   ...
401AFC      end the FAILURE block
401AFD      continue to end of program
```

To see what the low-level instructions in this subroutine look like, we need to switch to Hex View-A. Make sure that the current memory address is 401ABD. You should notice that the first instruction is highlighted. The instruction **cmp [ebp+var_61], 0** corresponds to the hex bytes 807D9F00. The hex byte 80 corresponds to the instruction **cmp**, the hex bytes 7D9F correspond to the first operand **[ebp+var_61]**, and the hex byte 00 corresponds to the second operand, **0**.

Clicking on the next hex byte, 74, highlights the relevant bytes for the next instruction. We can infer that the hex bytes 741E correspond to the instruction **jz short loc_401AE1**. In fact, the hex byte 74 corresponds to the instruction **jz**, and the hex byte 1E corresponds to the operand **short loc_401AE1**. Recall the the instruction JZ jumps if the Z bit is set. So if the result of the comparison is false (i.e., the wrong password was entered), then the FAILURE block will be invoked. That is, control will transfer to the operand specified in the JZ instruction (**short loc_401AE1**).

To force the program to execute the SUCCESS block regardless of the password entered, we simply need to clear out the operand of the JZ instruction at memory address 401AC1. Currently, the hex bytes for the instruction **jz short loc_401AE1** are 741E. To clear the operand, we simply need to change the hex byte 1E to 00 via Edit | Patch program | Change byte.

Now, go back to the flowchart view by selecting the IDA View-A tab. Note that it has changed! IDA Pro rearranged some things based on our changes. The JZ instruction, still at memory address 401AC1, has changed! It no longer flows to two separate blocks: SUCCESS or FAILURE. It now flows directly to the SUCCESS block! The JZ instruction now jumps to the next instruction to matter what the status of the Z bit is.

Save the DIF file via File | Produce file | Create DIF file... and exit IDA Pro. As before, patch the executable at the command line via patch the executable via **idaPatcher.exe -i password.exe -p password.dif**. Execute the patched executable and note that the output is always SUCCESS!