Countdown

A simple countdown program has been created. First, compile countdown.cc; then, run the executable. This can be done on whatever OS you are currently using. However, to reverse engineer it, you will need to compile it on Linux. It seems that a countdown from 2 million, counting down every second, begins. Who knows what happens when it reaches 0. Perhaps we should wait? No. Let's reverse engineer it instead.

Again, note that memory addresses and some symbol names (e.g., function names) are specific to the architecture. I am using x86-64 (64-bit). Also, make sure that you are in the same directory as the countdown binary executable. Compile and test it to make sure that it runs properly (break after a few seconds).

```
g++ -o countdown countdown.cc
./countdown
```

Next, launch Radare 2 in debug mode on the password binary executable:

```
r2 -d countdown
```

Feel free to try out many of the analysis types of operations that were covered in the previous "tutorial." For now, let's just jump straight into it. First, let's see the various symbols in the binary executable:

```
is
```

There are many symbols in this binary executable, some of which are interesting:

```
0x00400ea6: a global function obfuscate(string)
0x00400fa3: a global function main
```

Let's also list any defined strings:

```
iz
```

The hash looks interesting:

```
000 0x000011e8 0x004011e8  11  12 (.rodata) ascii COUNTDOWN:
001 0x00001200 0x00401200  32  33 (.rodata) ascii
47781b01794fc6e5962b1e70455c1c1c
002 0x00001221 0x00401221  16  17 (.rodata) ascii The password is
```

Why not have Radare 2 do an analysis of the code and start checking out the disassembled code in the main function:

```
aaa
s main
V
p
```

Here's what's going on. The code begins by setting up an initial countdown value at memory address 0x00400fbb:

```
mov dword [var_64h], 0x1e8480
```

Evidently, the value 0x1e8480 (2,000,000) is stored in a memory address pointed to by a variable known as var_64h. Hey, there's our 2 million! Then, the string "COUNTDOWN:" is output. A bit

further, at memory address 0x00400fd1, this value is compared to 0:
```
cmp dword [var_64h], 0
```

Evidently, it's checking if the countdown has reached 0 (i.e., if it has expired). If the result of the comparison is less than or equal to 0, the code jumps to memory address 0x401010. This occurs in the JLE (Jump if Less than or Equal) instruction at memory address 0x00400fd5:
```
jle 0x401010
```

Specifically, if the countdown value in var_64h is less than or equal to 0, the code jumps. Otherwise, it continues on. A countdown value that has not yet reached 0 results in it being displayed and subsequently decremented (at memory address 0x0040100a). The code then unconditionally jumps back up to memory address 0x400fd1 (where the countdown is again compared to 0):
```
sub dword [var_64h], 1
jmp 0x400fd1
```

The part of the code that begins at memory address 0x00401010 is reached when the countdown expires. This part of the code outputs “The password is ...” There is an interesting string at memory address 0x00401033: str.47781b01794fc6e5962b1e70455c1c1c. Could it be this easy? You wouldn't be so lucky...especially during Cyber Storm! In fact, we can inspect the obfuscate function (which is at address 0x00400ea6):
```
q
s 0x00400ea6
V
```

But this appears quite confusing! Taking a look at the source code, however, clears this up. But remember that having access to the source code is not typical. Why reverse engineer if you have the source code?

Patching
So, how do we patch the binary executable so that the end of the countdown is reached much, much more quickly? There are several things that we can do:
(1) Change the original countdown value; and
(2) Swap the condition in the JLE instruction (i.e., make it jump if the result is greater than).

Changing the original countdown value
First, exit Radare 2 and reopen the binary executable in write mode:
```
r2 -w countdown
```

Next, let's analyze and seek to the instruction that initializes the countdown and check it out in visual mode:
```
aaa
s 0x00400fbb
V
p
p
```

At this point, we can use the Rt and Lt arrow keys to move in memory, 1 byte at a time. Notice that the MOV instruction at the current memory address is 6 bytes long (from 0x00400fbb through 0x00400fc1). The instruction is:

```
mov dword [var_64h], 0x1e8480
```

However, the corresponding bytes (that make up this instruction) are:

```
c7459c80841e
```

Here's the entire line:

```
0x00400fbb      c7459c80841e.  mov dword [var_64h], 0x1e8480
```

The relevant portion of this instruction is the value 0x1e8480 (2,000,000):

```
c7459c80841e
```

Notice how it's specified: backwards! This is because the architecture is Little Endian. That is, the most significant byte is to the right. Conversely, the least significant byte occurs first (Little Endian). It would be great to directly change these bytes. We can do so by pressing the Rt key 3 times to reach the specified bytes. Notice how the instructions completely change. This is because the disassembler is attempting to dissassemble the code from the current byte. We can ignore that for now. To directly edit the bytes, we can press P twice to change the view mode to a hex format:

```
P
P
```

Note the current bytes (at memory address 0x00400fbe):

```
0x001e8480
```

Let's insert new bytes at this location. Why not change the initial countdown from 2 million to, say, 5:

```
i
00000005
```

Press ESC until you get back to the main radare2 screen. Then, seek back to the original instruction, and view it in visual mode:

```
s 0x00400fbb
V
p
p
```

Notice how the value was changed:

```
mov dword [var_64h], 0x5000000
```

What? It seemingly inserted the new value backwards! Why don't we try this out by exiting Radare 2 and running the binary executable. Gah! We are now counting down from 83,886,080. This is worse! Let's go back and try again. This time, let's insert the hex bytes backwards (in Little Endian format):

```
i
05000000
```

Press ESC until you get back to the main radare2 screen. Then, seek back to the original instruction, and view it in visual mode:

```
s 0x00400fbb
V
p
p
```

Note the instruction now:

```
mov dword [var_64h], 5
```

A-ha! This looks much better. Exit Radare 2 and run the binary executable. Sure enough, the countdown now expires in five seconds. And you may have noticed that the password is actually not the hash in the disassembled code. It is, actually, quite different!

Swapping the condition in the JLE instruction

To attempt the second patch, recompile the source code so that the binary executable is in its original form. Then, open the binary executable in write mode in Radare 2. Analyze the binary executable and seek to the JLE instruction at memory address 0x00400fd5:

```
aaa
s 0x00400fd5
V
```

Next, swap the condition:

```
q
wao recj
V
```

Notice how the instruction changed to JG (Jump if Greater):

```
jg 0x401010
```

It will now jump if the result is greater than 0. Well, 2 million is definitely greater than 0! The expected behavior is to immediately expire the countdown. Try it out.