Password

A simple password program has been created. First, compile password.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. After running the program, try to guess the password. FYI, you won't succeed. Well, at least not unless you took a look at the source code! Note that I am using the GNU C++ compiler, something that should be installed on your Linux system if you have installed essential build tools:

```
sudo apt-get install build-essential
```

So how do we break the password? We can reverse engineer! We can use the Radare 2 disassembler/debugger to workaround this. Refer to the notes on installing Radare 2 before you proceed.

Note that it will take some time to familiarize yourself with Radare 2. It's not especially user friendly, but it is capably designed for reversing.

Also note that memory addresses and some symbol names (e.g., function names) are specific to the architecture. I am using x86-64 (64-bit). There are others; for example, x86 (32-bit), i386 (same as x86), x64 (same as x86-64), and amd64 (same as x86-64).

First, make sure that you are in the same directory as the password binary executable. Compile and test it to make sure that it runs properly (even if you don't know the correct password):

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

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

```
r2 -d password
```

This "tutorial" will not go into all of the details of Radare 2. There are simply too many. So, feel free to explore on your own! Check out some online tutorials for assistance.

Also, since we are running this in a VM, the addresses in memory will be different each time we do this. Moreover, they will likely be different for you. So that this is evident, all addresses are highlighted in red.

So, what do we do from here? Radare 2 implements a useful "help" feature that involves typing a question mark (either on its own or after portions of commands):

```
?
```

Let start by getting some information about the binary executable:

```
iI
```

In my case, the architecture is x86, 64-bit, and little endian; and the source code is C++. It can be very useful to list the entry points that the binary executable has (and therefore the original source program):

```
ie
```

The output for this particular binary executable on my system is:
```
[Entrypoints]
vaddr=0x5621f151a0e0 paddr=0x000010e0 haddr=0x00000018
hvaddr=0x5621f1519018 type=program

1 entrypoints
```

The entry point of most interest to us, however, is the **main** entry point (corresponding to the main function):
```
iM
```

Cool! The main function is located at memory address 0x000013fd on my system:
```
[Main]
vaddr=0x5621f151a3fd paddr=0x000013fd
```

We also want to see the various symbols in the binary executable. Symbols are things like function names, variable names, etc:
```
is
```

There are many symbols in this binary executable, some of which are interesting:
```
0x5621f1519000: a local file password.cc
0x5621f151a603: a local function _GLOBAL__sub_I__Z10VALID_HASHB5cxx11
0x5621f151a3fd: a global function main
0x5621f151a1ea: a global function hexify
0x00203260: a global object VALID_HASH
0x5621f151a2d4: a global function hashish
```

OK, so it appears that there's some sort of VALID_HASH (maybe the correct password?), functions that "hashish" strings and hexify characters (whatever that means), and a main function.

It is also useful to list any defined strings in the binary executable:
```
iz
```

Ooooh! The hex stuff looks interesting:
```
0   0x0000189a 0x5621f151a89a 10  11   .rodata ascii Password:
1   0x000018a5 0x5621f151a8a5 9   10   .rodata ascii SUCCESS!\n
2   0x000018af 0x5621f151a8af 9   10   .rodata ascii FAILURE.\n
3   0x000018c0 0x5621f151a8c0 296 297  .rodata ascii
ffffffbcffffffb7ffffffbaffffffbaffffffa5ffffffd2ffffffb6ffffffabffff
fdffffffffb8ffffffadffffffb0ffffffb0ffffffa9ffffffbaffffffacffffffdfff
ffff9cffffff8dffffff96ffffff8cffffff8ffffffff86ffffffdffffffff9cffffff8
dffffff9effffff9cffffff94ffffff9affffff8dffffffdffffffff9cffffff97ffff
ff96ffffff8fffffff8c
```

So let's take a look at the disassembled code. First, let's "seek" to the main function:
```
s main
```

For me, it's at address 0x5621f151a3fd. Next, let's bring up visual mode:
```
V
```

In visual mode, p and P switch through different views. I prefer the third view (i.e., by pressing p twice). Let's look around a bit. Use the Up and Dn arrow keys to go through the code. You can also use PgUp and PgDn. We see obvious things like the “Password:” prompt, the “SUCCESS!” result, and the “FAILURE.” result.

In visual mode, we can also get help:
```
?
```

Press q to exit help in visual mode. Then, exit visual mode entirely:
```
q
q
```

Let's do a search for the string “SUCCESS”:
```
/ SUCCESS
```

A-ha! It seems that this string is found at memory address 0x5621f151a8a5 on my system. It is useful to see what references this particular string. To do this, we must first have Radare 2 analyze the binary executable:
```
aaa
```

Next, we can see what instruction (actually, its memory address) references the string “SUCCESS”:
```
axt 0x5621f151a8a5
```

It appears that it is referenced from the main function at address 0x5621f151a49e on my system:
```
main 0x5621f151a49e [DATA] lea rsi, str.SUCCESS
```

Let's seek there and check it out in visual mode:
```
s 0x5621f151a49e
V
```

OK, that's what we saw earlier. This is when the “SUCCESS!” result is displayed. In visual mode, we can observe a “flow” of the program. It renders arrows in the left margin that identify selection and repetition structures. Take a look at the ones around the “SUCCESS!” and “FAILURE.” results.

Exit visual mod:
```
q
```

The binary executable can then be executed:
```
dc
```

When prompted, type any password and press enter. The program will then end. To fully terminate the process, continue debugging:
```
dc
```

Let's reload the binary executable:
```
ood
```

When we reload the binary executable, the addresses are now likely to be different. So let's seek to main and search for the string SUCCESS again:

```
s main
/ SUCCESS
```

For me, the new address where SUCCESS is found is 0x5576b14148a5. Let's check out the reference address again:

```
axt 0x5576b14148a5
```

Let's seek there (0x5576b141449e for me) and check it out in visual mode agin:

```
s 0x5576b141449e
V
```

Now, move up one line (using Up) to show the JE (Jump if Equal) instruction. Let's set a breakpoint there at 0x5576b141449c on my system. We can do this several ways. First, we can manually do so in visual mode as follows:

```
B
```

Notice the lowercase letter b that appears to the right of the current instruction's memory address. To illustrate the second method of setting a breakpoint, remove the breakpoint and exit visual mode:

```
B
q
```

And set the breakpoint using another method:

```
db 0x5576b141449c
```

Then, check that a breakpoint was set by going back to visual mode:

```
V
```

A quick word about the TEST instruction above the JE instruction. The identifier bl represents the lower 8-bits of the register EBX. The TEST instruction ANDs the bits in the register (with themselves) but doesn't change it's value. Instead, it only updates various flags (like the z-bit). In this case, it checks to see if bl is 0. If so, then the z-bit is set (i.e., 1). The JE instruction then jumps if the z bit is 1. How does TEST actually work? Suppose that bl is 1; therefore, TEST bl, bl results in 1 & 1 – which is 1. Therefore, bl is not zero (clearly). Suppose that bl is 0; therefore, TEST bl, bl results in 0 & 0 – which is 0. Therefore, bl is 0.

The instruction TEST bl, bl is essentially the same as CMP bl, 0; however, it is smaller (i.e., it takes up fewer bytes in the program). You may be wondering what instruction(s) set bl. This actually occurs in the calls to the function sym.std::__cxx11::basic_string_char_std::[...]. This function does a character-by-character comparison of the strings.

Make sure that the JE instruction is at the top of the code, and exit visual mode:

```
q
```

A neat feature of Radare 2 is the ability to add comments to any instruction. This is useful if you figure something out that's not particularly readable. Let's add a comment to the JE instruction:

```
CC True = FAILURE.
```

Go into visual mode again to see the newly added comment:

```
V
q
```

It's sometimes helpful to see the various debugging options:

```
d?
```

Let's run the binary executable, providing any password, and see if it breaks at the breakpoint:

```
dc
```

It worked!

```
hit breakpoint at: 5576b141449c
```

Go back to visual mode:

```
V
```

The memory address is now highlighted, indicating that the code is suspended at this point. We can step through the instructions in visual mode, one instruction at a time, by pressing s. Since the correct password was not entered, flow transferred from the JE instruction to memory address 0x5576b14144b3 on my system (i.e., to the "FAILURE." result).

Exit visual mode and continue executing until the process is terminated:

```
dc
dc
```

Breaking password

Since we are in an interactive debugger, we can modify just about anything! In fact, let's try to modify the z-bit to see if we can force the program flow to go to the "SUCCESS!" result even if an incorrect password is provided. First, reload the binary executable:

```
ood
```

Note that the breakpoint at the JE instruction is still there (good). Begin execution of the binary executable:

```
dc
```

At the breakpoint (for me, it's now at address 55758c47049c), the program is at the JE instruction (check in visual mode if you wish). We can check the status of the z-bit:

```
dr zf
```

If an incorrect password was provided, the z-bit is 1 (i.e., it is set). This will cause the JE instruction to jump to its operand (now at memory address 0x55758c4704b3 on my system – the "FAILURE." result). We can manually change the z-bit to 0 (indicating a correct result) and continue executing the binary executable:

```
dr zf=0
dc
```

Notice that the "SUCCESS!" result was reached even if an incorrect password was provided! Finally, exit Radare 2:

```
q
Y
```

```
Y
```

Bypassing the password check is great; however, exiting Radare 2 and running the binary executable at the command line doesn't allow us to do the same thing. In other words, this is just a temporary workaround. Often, we'd prefer to permanently patch the binary executable so that any password could be entered and reach the “SUCCESS!” result.

Patching
There are several ways to permanently patch the binary executable to ensure the “SUCCESS!” result regardless of the provided password:

(1) Remove the JE instruction entirely, allowing the flow to continue directly to the “SUCCESS!” result;
(2) Reverse the condition of the JE instruction (i.e., change it to a JNE (Jump if Not Equal) instruction); and
(3) Change the target address in the JE instruction to jump to the “SUCCESS!” result instead of the “FAILURE.” result.

Removing the JE instruction
Let's take these in order. First, removing the JE instruction entirely. In IA-32 assembly, the NOP instruction means “no operation.” This effectively means that nothing happens (i.e., just go to the next instruction). The NOP instruction takes up one byte of program space. You may have noticed that the JE instruction takes up two bytes of program space. Therefore, we will have to replace the JE instruction with two consecutive NOP instructions. To do this, let's reload the binary executable in Radare 2. This time, however, let's load it in write mode instead of debug mode since we'll be patching the binary executable:

```
r2 -w password
```

Let's seek to the JE instruction:

```
s main
aaa
/ SUCCESS
axt 0x000018a5
s 0x149e
V
p
p
```

(go up one instruction to the JE instruction – which is at 0x0000149c for me)

```
q
```

Now, replace the JE instruction with two NOPs:

```
"wa nop; nop"
```

Note the quotes. This is necessary when specifying more than one instruction. Make sure that the change was successful:

```
Written 2 byte(s) (nop; nop) = wx 9090
```

To view the change, go to visual mode:

```
V
```

Note the code surrounding the NOPs:
```
0x0000149a      84db            test bl, bl
0x0000149c      90              nop
0x0000149d      90              nop
0x0000149e      488d35000400    lea rsi, str.SUCCESS
```

The TEST instruction will result in the z-bit being set to 1. The code then flows through the two NOPs, directly to the "SUCCESS!" result! Exit visual mode and Radare 2:
```
q
q
```

Since the change is permanent (i.e., it physically patched the binary executable), we can exit Radare 2 and run the program manually, getting the same result. Try it out!

Reversing the condition of the JE instruction
Before trying the second patching method, the source code must be recompiled. After all, it has been patched to bypass the "FAILURE." result! Compile and test the binary executable to ensure that it is in its original form:
```
g++ -o password password.cc
```

Again, load the binary executable in write mode in Radare 2:
```
r2 -w password
```

Seek to the JE instruction:
```
s 0x149c
```

Next, reverse the condition from JE to JNE:
```
wao recj
```

To view the change, go to visual mode:
```
V
p
p
```

Note the change:
```
0x0000149c      7515            jne 0x14b3
```

Exit visual mode and Radare 2:
```
q
q
```

Again, since the change is permanent, we can execute the program manually and bypass the "FAILURE." result.

Changing the target memory address in the JE instruction
The final patch involves changing the target address in the JE instruction to the "SUCCESS!" result instead of the "FAILURE." result. Again, compile and test the binary executable to ensure that it is in its original form. Then, load the binary executable in write mode in Radare 2:

```
r2 -w password
```

Next, seek to the JE instruction:
```
s 0x149c
```

This time, let's go to the appropriate view in visual mode right away:
```
V
p
p
```

The JE instruction is at 0x0000149c. The beginning of the "FAILURE." result is at 0x000014b3. The beginning of the "SUCCESS!" result is at 0x0000149e. Currently, the JE instruction jumps to the "FAILURE." result. We can change the target address from 0x000014b3 to 0x0000149e. Exit visual mode:
```
q
```

And change the target address:
```
wa je 0x0000149e
```

Make sure that the change was written:
```
Written 2 byte(s) (je 0x0000149e) = wx 7400
```

And check it in visual mode:
```
V
```

Note the changed flow arrows in the left margin! Finally, exit visual mode and Radare 2:
```
q
q
```

And now you can test the program to ensure that it bypasses the "FAILURE." result, regardless of the password entered.

Getting the correct password

You may be wondering if it's possible to determine the correct password. Since it's not specified directly in the source code (e.g., stored as a string), it's unfortunately not evident (nor is it particularly easy to find). The only way to potentially obtain the password is to reverse engineer the code further.

To begin, open the binary executable in debug mode in Radare 2:
```
r2 -d password
```

Earlier, we noticed various interesting symbols (through the command `is`). When inspecting the main function, we noticed a call to a function named hashish. Here's the relevant symbol again (which is at address 0x55a1950112d4):
```
hashish(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >)
```

Let's seek to this function and check it out:
```
s 0x55a1950112d4
V
```

```
p
p
```

What does this function do? Without knowing much assembly, it's a bit hard to figure this out. But essentially, the top part instantiates an empty string and starts going through the password passed in as an argument. Evidently the hashish function takes a string and hashes it somehow. If the password has not yet been processed, the code continues at 0x55a19501134a. This is where a hexify function is eventually called on each character of the password passed in as an argument. The JMP instruction at 0x55a195011396 loops back to check if the password has been fully processed. We need to examine the hexify function. So, let's seek to this function and check it out (which is at address 0x55a1950111ea):

```
q
s 0x55a1950111ea
V
```

What does this function do? Notice the stringstream. This means that a string is being built in this function. OK, so a single character passed in as an argument is converted to some sort of string. There's an interesting set of instructions at 0x55a195011253:

```
     movzx eax, byte [rbp - 0x1ac]
     not eax
```

It appears that one byte is being moved from some address relative to RBP into the EAX register. The MOVEZX instruction left pads the value with 0s in order to fill the register (i.e., it extends the value with 0's: ZX means zero extend). The value in the register is then inverted (i.e., NOT'd). A-ha! This may give us a clue about what the functions hexify and hash do. Perhaps the VALID_HASH is a version of the correct password that has all of its bits inverted (i.e., 1's are actually 0's, and vice versa)!

A few notes. Yes, this is where knowledge of assembly is useful. Yes, this will take some time. Yes, you need to tinker, make assumptions, and keep trying...

At the moment, there's no easy way to automate the process of determining the correct password. Well, that is unless we write code that takes the hex that we found earlier and inverts the bits! Maybe we need to write some Python code for this...

...and, here's the code:

```
# Takes the long hex string from the disassembly of the binary
# executable and "decodes" it.
# Inspecting the executable seemed to indicate a NOT operation on
# each bit.
# Let's try that and see if we get good ASCII.

from sys import stdout

DEBUG = False

# the "password"
password =
"ffffffbcffffffb7ffffffbaffffffbaffffffa5ffffffd2ffffffb6ffffffabffff
ffdffffffb8ffffffadffffffb0ffffffb0ffffffa9ffffffbaffffffacffffffdff
```

```
fffff9cffffff8dffffff96ffffff8cffffff8ffffff86ffffffdffffff9cffffff
8dffffff9effffff9cffffff94ffffff9affffff8dffffffdffffff9cffffff97fff
fff96ffffff8ffffff8c"

# step through each hex word
i = 0
while (i < len(password)):
     # grab 8 nibbles (4 bytes)
     word = password[i:i+8]
     if (DEBUG):
          print word
     # convert to an int (source base is 16)
     word = int(word, 16)
     if (DEBUG):
          print word
     # invert the bits by XORing with 1's
     word ^= 0xffffffff
     if (DEBUG):
          print word
     # convert to ASCII
     word = chr(word)
     stdout.write(word)

     # go to the next word
     i += 8
print
```

Most of the Python code should be readable. There are several debug statements that made it useful to debug the program as it was being written. The output statement `stdout.write` is used instead of `print` so that the decoded password appears together on a single line.

Essentially, the code takes the hex string, 8 characters at a time (i.e., in 4-byte, integer-sized chunks), converts each chunk to its numeric equivalent, inverts the bits using XOR with a mask of all 1's, and converts the resulting value to its corresponding ASCII character.

You may be wondering why XOR is used to invert the bits of the value instead of, say, NOT. This is because of the way that integers are represented in computers: two's complement. Discussing two's complement in detail is beyond the scope of this “tutorial.” But we can see the potential results by running an example of the first chunk through the Python interpreter:

```
>>> a = 0xffffffbc
>>> a
4294967228
>>> ~a # NOT a
-4294967229
```

Well, this doesn't help us at all! Since Python interprets the number as an integer, using the unary NOT operator (~) results in -(0xffffffbc + 1). This is just the way that Python deals with integers using two's complement notation:

```
>>> -(0xffffffbc+1)
```

```
-4294967229
```

The strategy is to instead use a mask of all 1's and XOR it with the original number. Here's an example of how this works:

```
original number:    00111010
mask of 1's:        11111111
XOR'd:              11000101
```

Notice how the result of the XOR operation effectively inverts the bits of the original number. Now, it works as intended:

```
>>> a ^= 0xffffffff # XOR a with all 1's
>>> a
67
>>> chr(a) # get the ASCII representation
'C'
```

67 is C! Now this makes more sense. Running the program on all of the hex chunks results in the correct password: CHEEZ-IT GROOVES crispy cracker chips.