Lab #5: Dynamic Binary Analysis:

CYBR 570 : Reverse Engineering
Spring 2025

Submitted By: Lilly Carlascio
Submission Date: 5/7/25

Part 1 - Get past the errors:

crackme: Here are the steps I walked through to reach the success message:

1. To open up the binary in r2, analyze, auto-analyze all the functions, move into the main function and then print the functions I ran these commands:

```
r2 -d ./crackme 123456
aaa
s main
pdf
```

2. Based on looking at the diasassembly of the main function, I saw that the program expects one argument (the password to be tested) and it validates the string using a few different function calls: check_id_xor() and check_id_sum().

3. After both of the function calls complete there is an instruction (test eax, eax) followed by a conditional jump that will either go to the line of code that prints "The password was correct" (which is printed in the main function) or "Incorrect password" (which was printed in the sym.error function).

4. After running into issues dynamically changing the return values of the check_id_xor() and check_id_sum() functions, I instead dynamically change the control flow of the program dynamically by patching the instruction pointer (RIP) to the memory address of the success message. Here are the commands I ran:

```
db 0x652dc38713f9
dc
dr rip=0x652dc3871413
dc
```

5. These commands set a breakpoint at the location that calls the check_id_xor function, I run the program until the breakpoint is hit, I then changed the RIP value (the register holding the address of the next instruction to execute) to the address that will print the success message, and then I ran the rest of the program and received "The password was correct".

6. I was able to find the correct address to assign to the RIP by examining this line:

```
0x652dc3871413      488d3d140c00.  lea rdi, str.The_password_was_correct
; 0x652dc387202e ; "The password was correct"
```

7. By jumping straight to the successful print statement, I avoided the comparison check entirely which achieved the goal without providing the correct password and without modifying the binary on the disk, solely with dynamic changes using r2.

.

good_job: Here are the steps I walked through to reach the success message:

1. To open up the binary in r2, analyze, auto-analyze all the functions, move into the main function and then print the functions I ran these commands:

```
r2 -d ./good_job 123456
aaa
s main
pdf
```

2. Looking at the disassembly of the main function, this function has both success and failure statements printed in main, and there are a few compare statements comparing the user input string with the correct string that will give the success message.

3. I looked for the line that printed the success message and then ran similar commands to the previous problem:

```
db 0x651b9149811c
dc
dr rip=0x651b91498153
dc
```

4. These commands set a break point at the conditional jump point, I then ran the program until the break point was hit, I then set the RIP (instruction pointer) to the memory address of where the success message would print, and then ran the program to completion, where I got the message, "Good job!".

5. I was able to find the correct address to assign to the RIP by examining this line:

```
0x651b91498153      488d3dc50e00.  lea rdi, str.Good_job_
; 0x651b9149901f ; "Good job!"
```

6. By jumping straight to the successful print statement, I avoided the comparison check entirely which achieved the goal without providing the correct password and without modifying the binary on the disk, solely with dynamic changes using r2.

.

Part 2 - What's wrong?

crackmetoo: For this binary, I analyzed the binary to figure out what is wrong with it.

1. From inspecting the encrypt_password function that is executed twice in this binary, I was able to find out that it encrypts and input string and stores the result in a buffer. The hardcoded string that is used in OpenSSL's EVP API to be put through AES-256-CBC is "7a4baaac9cc258c33b7459113da21360ef68864859cdb4dc389b20e24513c110". This string looks to be used twice in the cncrypt_password() function, once as a baseline and again to encrypt the user's input for comparison. The result of each encryption is stored at:

```
[rbp - 0x50] ; for the first call
[rbp - 0xd0] ; for the second call
```

2. I set a breakpoint after each encrypt_password() call and used the px command in r2 to capture the outputs.

```
px 0x40 @ 0x7fffffffe550
```

   I ran the program twice and received different outputs both times, which I wasn't expecting, but leads me to the first error. Even though the input string was the same in both runs, the cipher-text(see output1.txt and output2.txt) produced was different. The error comes from the use of AES-256-CBC cipher in OpenSSL and the function EVP_EncryptInit_ex which internally consumes or alters the initialization vector which means the encryption results will differ if the same context isn't reused or reinitialized identically, even though the same static key and IV are used. This means that the comparison between the two cipher-texts will always fail, even if the plain-texts are identical. (https://stackoverflow.com/questions/38125254/what-is-the-default-iv-when-encrypting-with-aes-256-cbc-cipher)

3. The other error I discovered, which is unrelated to the encrypted_password() exists in the input handlind where the input is read using fgets() and then sanitized with strcspn() to remove the newline character. The result of strcspn() is used incorrectly in this function:

```
mov byte [rbp + rax - 0xa0], 0
```

   This line is supposed to null-terminate the input, but the calculated offset (rax - 0xa0) is applied relative to rbp, not relative to the actual buffer. This may be overwriting the wrong memory location. This increases the chance of a failed string (even if correct) since the incorrect null-termination can cause string functions to maybe read out of bounds.

4. As for getting it to work for the command line, due to the encryption bug explained in Step 1/2, since the ciphertext of the known correct tsring is generated and stored on-the-fly durin program execution, any comparison with a string from the user will fail due to non-deterministic IV state. Even using the hardcorded string "7a4baaac9cc258c33b7459113da213 60ef68864859cdb4dc389b20e24513c110" directly as input will fail since the encryption context reinitialize and generates different cipher-text, even for the same input.