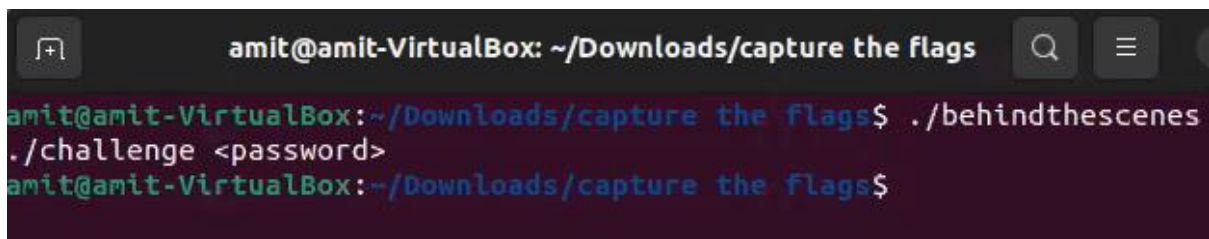# Behing The Scenes

Capture the Flag Challenge.

Link: Challenge can be found here.

Overview: The task of this CTF is to retrieve the flag from ELF binary program.

The method to achieve this task is to reverse engineering the program in order to extract the correct string that will provide us with the flag.

Method: The first thing that I tried is to run the program normally:

And I immediately found the the program requires some password.

So I tried to run it again, with some string:

And with custom string, nothing happens.

So clearly the objective to crack the program is to find the correct string.

But how?

The first thing that was tried is to run 'strings' on the program, however that did not reveal anything meaningful, so I had to look further on the assembly commands.

So I did 'objdump -d behindthescenes':

The first part of the assembly code is initialization of the data :

```
0000000000001261 <main>:
    1261:    f3 0f 1e fa                endbr64
    1265:    55                         push    %rbp
    1266:    48 89 e5                   mov     %rsp,%rbp
    1269:    48 81 ec b0 00 00 00       sub     $0xb0,%rsp
    1270:    89 bd 5c ff ff ff          mov     %edi,-0xa4(%rbp)
    1276:    48 89 b5 50 ff ff ff       mov     %rsi,-0xb0(%rbp)
    127d:    64 48 8b 04 25 28 00       mov     %fs:0x28,%rax
    1284:    00 00
    1286:    48 89 45 f8                mov     %rax,-0x8(%rbp)
    128a:    31 c0                      xor     %eax,%eax
    128c:    48 8d 85 60 ff ff ff       lea     -0xa0(%rbp),%rax
    1293:    ba 98 00 00 00             mov     $0x98,%edx
    1298:    be 00 00 00 00             mov     $0x0,%esi
    129d:    48 89 c7                   mov     %rax,%rdi
    12a0:    e8 7b fe ff ff             call    1120 <memset@plt>
    12a5:    48 8d 85 60 ff ff ff       lea     -0xa0(%rbp),%rax
    12ac:    48 83 c0 08                add     $0x8,%rax
    12b0:    48 89 c7                   mov     %rax,%rdi
    12b3:    e8 78 fe ff ff             call    1130 <sigemptyset@plt>
    12b8:    48 8d 05 6a ff ff ff       lea     -0x96(%rip),%rax        # 1229 <segill_sigaction>
    12bf:    48 89 85 60 ff ff ff       mov     %rax,-0xa0(%rbp)
    12c6:    c7 45 e8 04 00 00 00       movl    $0x4,-0x18(%rbp)
    12cd:    48 8d 85 60 ff ff ff       lea     -0xa0(%rbp),%rax
    12d4:    ba 00 00 00 00             mov     $0x0,%edx
    12d9:    48 89 c6                   mov     %rax,%rsi
    12dc:    bf 04 00 00 00             mov     $0x4,%edi
    12e1:    e8 fa fd ff ff             call    10e0 <sigaction@plt>
    12e6:    0f 0b                      ud2
    12e8:    83 bd 5c ff ff ff 02       cmpl    $0x2,-0xa4(%rbp)
    12ef:    74 1a                      je      130b <main+0xaa>
    12f1:    0f 0b                      ud2
    12f3:    48 8d 3d 0a 0d 00 00       lea     0xd0a(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    12fa:    e8 d1 fd ff ff             call    10d0 <puts@plt>
    12ff:    0f 0b                      ud2
    1301:    b8 01 00 00 00             mov     $0x1,%eax
    1306:    e9 2e 01 00 00             jmp     1439 <main+0x1d8>
    130b:    0f 0b                      ud2
    130d:    48 8b 85 50 ff ff ff       mov     -0xb0(%rbp),%rax
    1314:    48 83 c0 08                add     $0x8,%rax
    1318:    48 8b 00                   mov     (%rax),%rax
    131b:    48 89 c7                   mov     %rax,%rdi
    131e:    e8 cd fd ff ff             call    10f0 <strlen@plt>
    1323:    48 83 f8 0c                cmp     $0xc,%rax
    1327:    0f 85 05 01 00 00          jne     1432 <main+0x1d1>
```

In line 12e8 we can observe there is a comparison of stack variable to 2, the purpose of this comparison is to ensure there are indeed 2 input arguments – the program itself and the user's input string.

The next part in the string authentication is the 'strlen' function call – taking place in line 131e – it checks for the length of the string.

Its result is loaded to rax registers, and on the next line – 1323, it compares the result to 0xc, meaning 12.

So it tells us that the desired string length we are looking for, is 12 characters long.

So when the input string passes the length test – the next stage in the reverse engineering process begins which is to extract the string itself:

The next section in the assembly code indeed taking care of this action:

```
132d:       0f 0b                       ud2
132f:       48 8b 85 50 ff ff ff        mov     -0xb0(%rbp),%rax
1336:       48 83 c0 08                 add     $0x8,%rax
133a:       48 8b 00                    mov     (%rax),%rax
133d:       ba 03 00 00 00              mov     $0x3,%edx
1342:       48 8d 35 d2 0c 00 00        lea     0xcd2(%rip),%rsi       # 201b <_IO_stdin_used+0x1b>
1349:       48 89 c7                    mov     %rax,%rdi
134c:       e8 6f fd ff ff              call    10c0 <strncmp@plt>
1351:       85 c0                       test    %eax,%eax
1353:       0f 85 d0 00 00 00           jne     1429 <main+0x1c8>
1359:       0f 0b                       ud2
135b:       48 8b 85 50 ff ff ff        mov     -0xb0(%rbp),%rax
1362:       48 83 c0 08                 add     $0x8,%rax
1366:       48 8b 00                    mov     (%rax),%rax
1369:       48 83 c0 03                 add     $0x3,%rax
136d:       ba 03 00 00 00              mov     $0x3,%edx
1372:       48 8d 35 a6 0c 00 00        lea     0xca6(%rip),%rsi       # 201f <_IO_stdin_used+0x1f>
1379:       48 89 c7                    mov     %rax,%rdi
137c:       e8 3f fd ff ff              call    10c0 <strncmp@plt>
1381:       85 c0                       test    %eax,%eax
1383:       0f 85 97 00 00 00           jne     1420 <main+0x1bf>
1389:       0f 0b                       ud2
138b:       48 8b 85 50 ff ff ff        mov     -0xb0(%rbp),%rax
1392:       48 83 c0 08                 add     $0x8,%rax
1396:       48 8b 00                    mov     (%rax),%rax
1399:       48 83 c0 06                 add     $0x6,%rax
139d:       ba 03 00 00 00              mov     $0x3,%edx
13a2:       48 8d 35 7a 0c 00 00        lea     0xc7a(%rip),%rsi       # 2023 <_IO_stdin_used+0x23>
13a9:       48 89 c7                    mov     %rax,%rdi
13ac:       e8 0f fd ff ff              call    10c0 <strncmp@plt>
13b1:       85 c0                       test    %eax,%eax
13b3:       75 62                       jne     1417 <main+0x1b6>
13b5:       0f 0b                       ud2
13b7:       48 8b 85 50 ff ff ff        mov     -0xb0(%rbp),%rax
13be:       48 83 c0 08                 add     $0x8,%rax
13c2:       48 8b 00                    mov     (%rax),%rax
13c5:       48 83 c0 09                 add     $0x9,%rax
13c9:       ba 03 00 00 00              mov     $0x3,%edx
13ce:       48 8d 35 52 0c 00 00        lea     0xc52(%rip),%rsi       # 2027 <_IO_stdin_used+0x27>
13d5:       48 89 c7                    mov     %rax,%rdi
13d8:       e8 e3 fc ff ff              call    10c0 <strncmp@plt>
13dd:       85 c0                       test    %eax,%eax
13df:       75 2d                       jne     140e <main+0x1ad>
```

We can immediately notice that the 'strncmp' is being called 4 times – indicating our string got divided for 4 parts and each part got compared with its relative counterpart in the password string.

Before we analyze the assembly commands of 'strncmp' – lets cover the actual c function 'strncmp' parameters:

const char *str1, const char *str2, size_t n.

where str1 and str2 points to the strings we check, and n is the value of how much bytes we compare.

Lets analyze the first comparsion, taking place from line 135b to line 1381 – it takes 3 parameters: the pointer of our string – loaded from stack

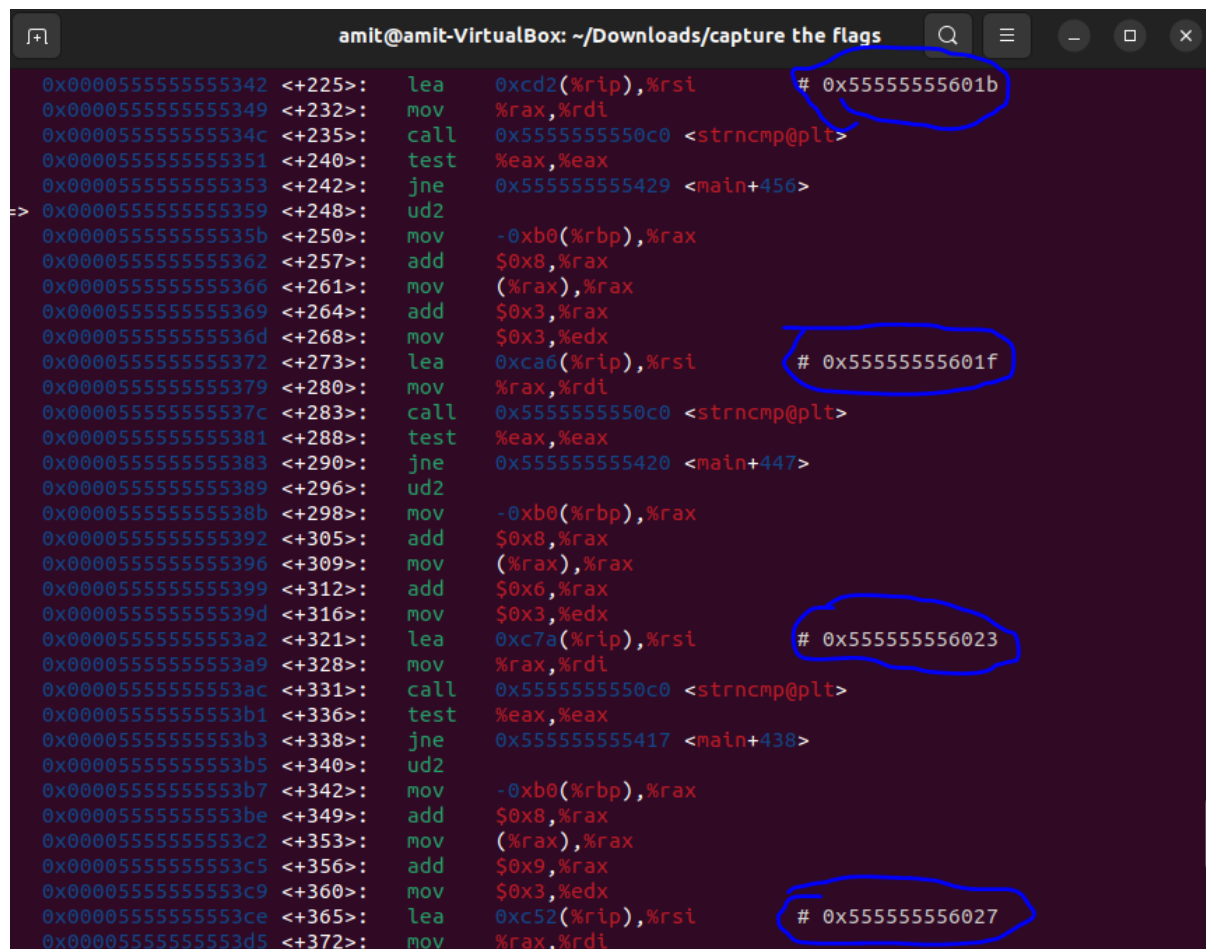(-0xb0(%rbp), and its content raise by 8, the result is the pointer to out string.

In later iterations we add to the point value of 3*i to reach the required part of the string), that would be our str1.

The str2 would be the pointer address to the string of the actual quarter of a password loaded from memory to rsi register.

The value 3 would be the n, loaded to edx memory.

So what I needed to do is to inspect the pointer to str2, It was done with gdb debugger:

I ran the gdb and put the command: 'disassemble main':

We established that the actual pointers of each quarter of the actual password are loaded from the memory to rsi register, and the actual value that holds the string is circled on the right.

So what we need to do, is to inspect each address for the first 3 bytes, in order:

The password is the ASCII values of the bytes, in the order presented. after quick decryption, the password is:

Itz_onLy_UD2

Lets run the program with the password:

```
amit@amit-VirtualBox:~/Downloads/capture the flags$ ./behindthescenes Itz_0nLy_UD2
> HTB{Itz_0nLy_UD2}
amit@amit-VirtualBox:~/Downloads/capture the flags$ 
```

WE GOT THE FLAG!

Conclusion: the CTF was fairly easy, well structed for begginers.

I mostly learned from this how to extract data that is stored in memory, and the use and important of the assembly command 'lea'- load effective address.

I also learn about work method - where to focus in the program inspection, and what can be a distraction.

This challenge also has a Demonstration Video (In Hebrew) that can be downloaded [here](#).