**02 New Orleans**

As per the tutorial, the **check_password** function contains the instructions to determine if the entered password is correct or not.

```
44bc <check_password>
44bc:  0e43           clr   r14
44be:  0d4f           mov   r15, r13
44c0:  0d5e           add   r14, r13
44c2:  ee9d 0024      cmp.b @r13, 0x2400(r14)
44c6:  0520           jne   #0x44d2 <check_password+0x16>
44c8:  1e53           inc   r14
44ca:  3e92           cmp   #0x8, r14
44cc:  f823           jne   #0x44be <check_password+0x2>
44ce:  1f43           mov   #0x1, r15
44d0:  3041           ret
44d2:  0f43           clr   r15
44d4:  3041           ret
```

Since r15 contains the address of the password on the stack, the r13 register should contain the address of the byte currently being compared, since r14 is incremented for each byte being compared.

The cmp.b instruction has a .b suffix, which is defined as such in the instruction set manual:
The suffix .B at the instruction mnemonic will result in a byte operation, meaning the cmp.b does a byte-wise comparison as opposed to a word-wise one.

The cmp.b instruction compares the value of the current letter with the value at 0x2400(r14). Since r14 starts from 0 and increments by 1 when it moves on to the next letter, this means that the password should be stored at address 0x2400 on the stack.
Let's take a look at the stack when we are in the check_password function:
2400:  253d 6c39 6137 5900 0000 0000 0000 0000  %=l9a7Y

**03 Sydney**

Once again analyzing the **check_password** function.

The r15 register contains the address of the password we entered, so what these instructions are doing is to compare every dword (2 bytes) stored in r15 with an immediate value. I assume these are values that make up the actual password.

Since the CPU is little-endian, it means that when we see #0x5122 in the instruction, the byte ordering is actually 0x22 0x51 in memory. So we compare 0x22 with 0x0(r15) and 0x51 with 0x1(r15).

From there I took all the immediate values in the cmp instructions and passed it in a hex to ASCII converter online.  2251486f2169647e → "QHo!id~

**04 Hanoi**

I started by looking at the**test_password_valid** function, but it doesn't seem to interact with our password at all which honestly stumped me for a while. The function does some stack operations, calls an INT function, and then returns a value. But nowhere does it reference the actual password stored at 0x2400.

Instead, I decided to fuzz the password field by entering 1024 "A"s. It turns out the input gets truncated to 28 bytes, which is confirmed by a puts call in the login function using 0x1c (28 in decimal). But then, the I/O console displayed a message: "Remember: passwords are between 8 and 16 characters." Which got me to think that there has to be a way to mess with execution flow if we go beyond that limit.

I moved on to the login function, and here is where I noticed that 0x455a, there's a cmp.b instruction that compares the value at 0x2410 with 0x4c (ASCII for 'L'). The weird part? There's no obvious reason for this check. But I remembered that our input starts at 0x2400, and since puts lets us enter 28 characters, we can actually control the 17th byte, so we can directly set 0x2410.

So, if we make sure the 17th character of our password is 'L', we'll pass the check and unlock the door.

**05 - Cusco**

The trivial condition check has been removed from the login function. The rest of the instructions look similar. However, after taking a closer look, I noticed several differences. First, my input is located at 0x43ed on the stack, and it allows a maximum size of 0x30 bytes. The key detail here is that only 16 bytes were actually allocated for the buffer, meaning I have a classic buffer overflow vulnerability.

After entering "AAAA" as my password, the stack looks like this:

```
43e0:   5645 0000 a045 0200 ee43 3000 1e45 4141   VE...E...C0..EAA
43f0:   4141 0000 0000 0000 0000 0000 0000 3c44   AA............<D
```

At address 0x43fd, I see 0x443c (little-endian), which happens to be the return address for the login function. I can confirm this by setting a breakpoint on the **ret** instruction at the end of **login.**

This means I can control the execution flow by overwriting the return address at 0x43fe! Since **unlock_door** is located at 0x4446, I just need to overflow the buffer and place 46 44 after 16 bytes of junk. Password: AAAAAAAAAAAAAAAADF

## 06 - Reykjavik