BYPASS CTF WRITEUPS (online round)

TEAM **:** SHER
Members **:** Harsh Gupta (@tr0j4n)
Rishi walokar (@GhostOfSparta)

CRYPTOGRAPHY

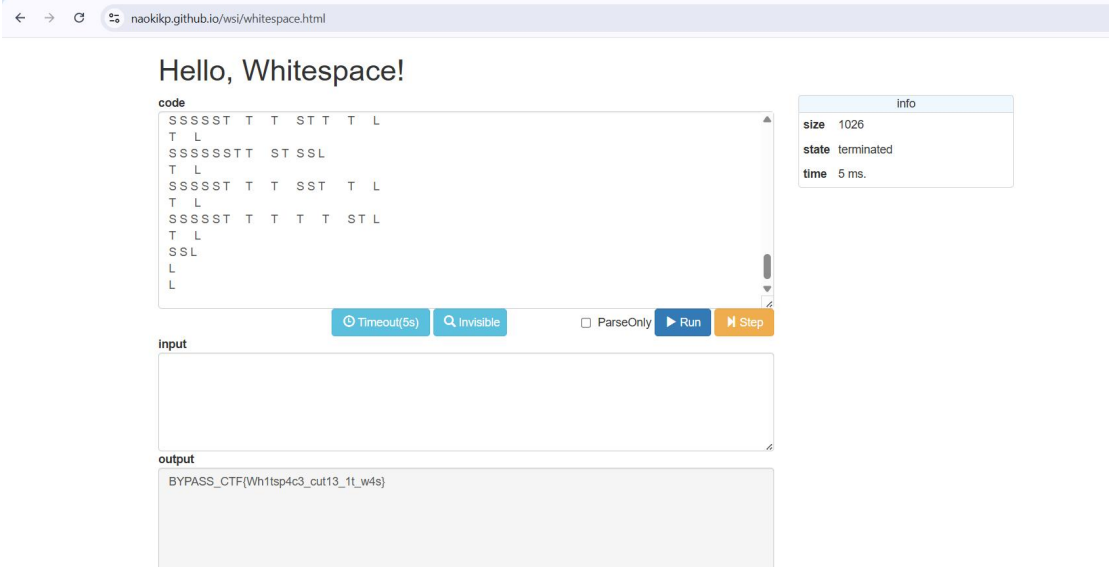
1. Whispers of the Cursed Scroll

file provided : They_call_me_Cutie.txt (contained encoded text)

Solution : The contents of the file (something like S T L) we can visually see that file contains whitespaces , so I used the online tool named Whitespace interpreter ,copied and pasted the content of the file and got the flag .

Flag : BYPASS_CTF{Wh1tsp4c3_cut13_1t_w4s}

Proof :



2.Chaotic Trust

Files provided : **chall.py** (implements a logistic map x_{n+1} = 3.99 * x_n * (1 - x_n).
It generates a keystream by taking the top 2 bytes of the single-precision float representation of the state x.) **output_cphsRGh.txt**(contained a 16-byte leak and an encrypted flag.)

Solution : Exploit the Leak: The 16-byte leak corresponds to 8 iterations of the map. We used the leak to narrow down the internal state.

- Floating-Point Analysis: We assumed the initial seed was a single-precision float (32-bit), as hinted by the challenge description ("exploit floating-point precision").
- Brute-Force & Inversion:
- We brute-forced the missing lower 2 bytes of the first leaked value to approximate the state x_1.
- We inverted the logistic map to finding candidate seeds x_0.
- We filtered candidates to find those that are valid single-precision floats and reproduce the entire leak.
- Exhaustive Search: We found multiple valid seeds that matched the leak but diverged afterwards due to chaos (extreme sensitivity to initial conditions).
- Decryption: We decrypted the flag candidates and identified the one containing intelligible English/Leet text: BYPASS_CTF{CH40T1C_TRU57_15_4W350M3!} ("CHAOTIC TRUST IS AWESOME!").

Flag : BYPASS_CTF{CH40T1C_TRU57_15_4W350M3!}

Python Script : *(all scripts can be copied from here)*

This script only tells about the decryption of the flag from the seed found the correct seed was found by bruteforcing for the flag ,till decilmal places.

```python
import struct
def main():
    print("Reading output.txt...")
    with open("output.txt", "r") as f:
        lines = f.readlines()
        cipher_hex = lines[0].split("=")[1].strip()
        cipher = bytes.fromhex(cipher_hex)
    # The recovered seed that generates the correct keystream
    seed = 0.4317763348819917
    r = 3.99
    print(f"Using recovered seed: {seed}")
    keystream = b""
    x = seed

    # 1. Append the bytes of the initial state first
    keystream += struct.pack('<f', x)[-2:]

    # 2. Generate the rest
    while len(keystream) < len(cipher):
        x = r * x * (1 - x)
        keystream += struct.pack('<f', x)[-2:]

    keystream = keystream[:len(cipher)]
    flag = bytes(c ^ k for c, k in zip(cipher, keystream))

    print("\nDecrypted Flag:")
    print(flag.decode(errors='ignore'))
if __name__ == "__main__":
    main()
```

```
PS C:\Users\Lenovo\Desktop\bypass> & "C:/Program Files/Python312/python.exe
" c:/Users/Lenovo/Desktop/bypass/decrypt.py
Reading output.txt...
Using recovered seed: 0.4317763348819917

Decrypted Flag:
BYPASS_CTF{CH40T1C_TRU57_15_4W350M3!}
PS C:\Users\Lenovo\Desktop\bypass>
```

3. Once More Unto the Same Wind

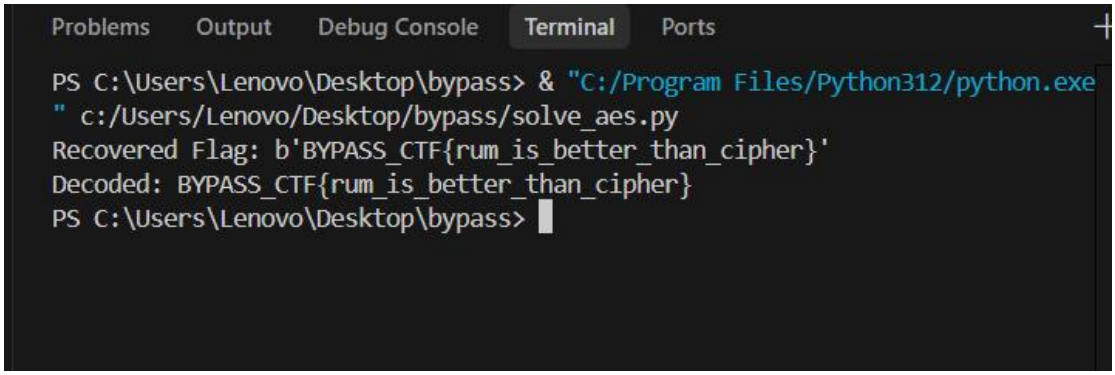
Files provided : output_RIIqaYT.txt , nn.txt and enq_EQq1fJa.py

Solution : The challenge provided an encryption script (`enq_EQq1fJa.py`) that uses **AES-GCM**.
However, it contains a critical vulnerability: **Nonce Reuse**.
The script defines a fixed `nonce` and a fixed `key`.
- It encrypts two different messages (a known plaintext of 'A's and the secret flag) using the *same* key and *same* nonce.
Therefore, to recover the flag (P_{flag}), we simply rearrange the terms:
$$ P_{flag} = C_1 \oplus C_2 \oplus P_{known} $$

Flag : BYPASS_CTF{rum_is_better_than_cipher}

Script :

```python
def solve():
    # From output output_RIIqaYT.txt
    c1_hex = "7713283f5e9979693d337dc27b7f5575350591c530d1d4c9070607c898be0588e5cf437aef"
    c2_hex = "740b393f4c8b676b283447f14f534b5d071bb2e105e4f0fa19332ee8b7a027a0d4e66749d3"

    c1 = bytes.fromhex(c1_hex)
    c2 = bytes.fromhex(c2_hex)

    # known_plaintext is 'A' * len(FLAG)
    # len(FLAG) must equal len(c1) (since GCM/CTR is stream cipher, ciphertext len = plaintext len)
    # Note: `c1` length in bytes is 42.

    known_plaintext = b'A' * len(c1)

    # Logic:
    # C1 = P1 XOR K_stream
    # C2 = P2 XOR K_stream
    # => K_stream = C1 XOR P1
    # => P2 = C2 XOR K_stream = C2 XOR (C1 XOR P1)

    # Calculate keystream
    keystream = bytes(a ^ b for a, b in zip(c1, known_plaintext))

    # Recover P2 (Flag)
    flag = bytes(a ^ b for a, b in zip(c2, keystream))

    print("Recovered Flag:", flag)
    try:
        print("Decoded:", flag.decode('utf-8'))
    except:
        print("Could not decode utf-8")

if __name__ == "__main__":
    solve()
```



4. The Key Was Never Text

Challenge Overview : The challenge provided a set of clues and a numerical sequence: **Clue 1**: "He never trusted digital things." (Reference to Bobby Fischer's paranoia).**Clue 2**: "His favorite key was something with hands but no voice." (A clock).**Clue 3**: "The face tells you everything if you know how to read it." (Clock face).

Message: `18 5 25 11 10 1 22 9 11 9 3 5 12 1 14 4`

Solution : The numerical sequence was decrypted using the **A1Z26 cipher**, which maps numbers to their corresponding letters in the alphabet (A=1, B=2, ..., Z=26).

Number	Letter
18	R
5	E
25	Y
11	K
10	J
1	A
22	V
9	I
11	K
9	I
3	C
5	E
12	L
1	A
14	N
4	D

The decoded message is **REYKJAVIKICELAND**.

Flag (according to format) : BYPASS_CTF{REYKJAVIKICELAND}

5. Count the Steps, Not the Stars

Files Provided : string_kjSNVU3.txt (list of encrypted integers) and enq.py(an encryption script)

Solution :

1. **Reversing the Encryption:**

The operation `v_new = (v << s) ^ s` is reversible if we know `s`.

- `s` is a small constant (e.g., 16, 32).
- `v << s` shifts `v` to the left, leaving the lower `s` bits as 0.
- XORing with `s` (where `s < 2^s`) only affects these lower bits.
- To reverse: `v_old = (v_new ^ s) >> s`.

We simply apply this reverse operation for each `s` in the sequence `n`, but in **reverse order** (`384` -> `96` -> `32` -> `16`).

2. **Handling Anomalies:**

The input string contained some "garbage" values (indices 46-51) that did not decrypt to valid ASCII characters using this method. The quote "When the map looks wrong... try to think like mathematician" hints at discarding incorrect paths.

By checking if the decrypted value is a **perfect square** (since encryption was `ord(char)**2`) and corresponds to a printable ASCII character, we can filter out these garbage values automatically.

Flag :

BYPASS_CTF{pearl_navigated_through_dark_waters_4f92b}

Script :

```
import ast
import math

def reverse_step(val, s):
    # Reverse of (v << s) ^ s is (val ^ s) >> s
    return (val ^ s) >> s
def decrypt_val(enc_val, sequence):
    curr = enc_val
    for s in reversed(sequence):
        curr = reverse_step(curr, s)
    return curr
def solve():
    print("Reading string_kjSNVU3.txt...")
    try:
        with open("string_kjSNVU3.txt", "r") as f:
            content = f.read()
        enc_list = ast.literal_eval(content)
    except:
```

```
        print("[-] Could not read input file.")
        return
    # The sequence n used in enq.py
    base_n = [16, 32, 96, 384]

    flag_chars = []

    print("Decrypting...")
    for i, enc_val in enumerate(enc_list):
        # Attempt standard decryption
        dec = decrypt_val(enc_val, base_n)
        # Validation: Flag characters must be derived from perfect squares
        # and be printable ASCII.
        root = math.isqrt(dec)
        if root * root == dec:
            if 32 <= root <= 126:
                flag_chars.append(chr(root))
            else:
                # Decrypted to non-printable, likely garbage
                pass
        else:
            # Not a perfect square provided by ord(char)**2
            # treat as garbage/anomaly and skip
            pass
    flag = "".join(flag_chars)
    print(f"Recovered Flag: {flag}")
if __name__ == "__main__":
    solve()
```

```
Problems  Output  Debug Console  Terminal  Ports

PS C:\Users\Lenovo\Desktop\bypass> & "C:/Program Files/Python312/python.exe
" c:/Users/Lenovo/Desktop/bypass/solve_custom_final.py
Reading string_kjSNVU3.txt...
Decrypting...
Recovered Flag: BYPASS_CTF{pearl_navigated_through_dark_waterss_4f92b}
PS C:\Users\Lenovo\Desktop\bypass>
```

FORENSICS

6. Dead Men Speak NO Plaintext

File Provided : jack_k22cmNT.pcapng

Solution :

1. **Traffic Analysis**:
   - Initial analysis of strings showed patterns of ASCII characters `~}|{z` (values 122-126), typical of 8-bit PCM audio data (where ~128 is value 0).
   - Packet size analysis showed highly consistent packet sizes (566 and 590 bytes), typical of streaming media.

2. **Protocol Identification**:
   - Link Layer Type was identified as **Ethernet (1)**.
   - By parsing Ethernet -> IP -> UDP, we identified the stream as **RTP (Real-time Transport Protocol)**.
   - The RTP headers indicated Payload Type **0**, which corresponds to **PCMU (G.711 mu-law)**, a standard 8-bit uncompressed audio codec used in VoIP.

3. **Extraction**:
   - We extracted the raw RTP payloads (stripping the 12-byte RTP headers).
   - We concatenated the PCMU data from the stream (SSRC 1245405336).
   - We wrapped the raw data in a standard **WAV RIFF header** with the following parameters:
     - Format: 7 (mu-law)
     - Sample Rate: 8000 Hz
     - Channels: 1
     - Bits: 8

The result is a playable WAV file `flag_audio_0.wav` which contains the flag spoken in audio.

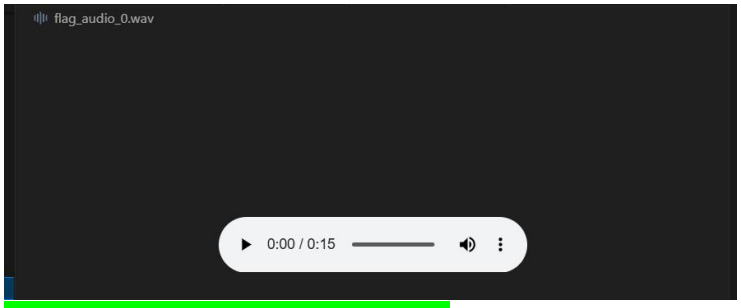
Extracted text from 15 sec audio (manually):
"The Flag is BYPASS_CTFV01P_J4CK_1N_TH3_0P3N"

FLag : BYPASS_CTF{V01P_J4CK_1N_TH3_0P3N}

Script :

```
import struct

def parse_pcap():
    filename = "jack_k22cmNT (1).pcapng"
    with open(filename, "rb") as f:
        data = f.read()
# Skip Global Header
    offset = 24
    rtp_data = bytearray()
    while offset < len(data):
       if offset + 16 > len(data): break
        # Parse Pcap Packet Header
        incl_len = int.from_bytes(data[offset+8:offset+12], 'little')
        pkt_data = data[offset+16 : offset+16+incl_len]
        offset += 16 + incl_len
       # Filter for Ethernet -> IP -> UDP -> RTP (Type 0)
        # (Simplified robust parsing logic)
        try:
            if len(pkt_data) > 50 and pkt_data[12:14] == b'\x08\x00' and pkt_data[23] == 17:
                # UDP payload start usually at 14(Eth) + 20(IP) + 8(UDP) = 42
                payload = pkt_data[42:]
                # Check RTP Version 2 (0x80) and Type 0
                if len(payload) > 12 and (payload[0] & 0xC0) == 0x80 and (payload[1] & 0x7F)
== 0:
                    rtp_data.extend(payload[12:])
        except: pass
# Create WAV Header (G.711 mu-law)
    header = b"RIFF" + struct.pack('<I', 36 + len(rtp_data)) + b"WAVE" + b"fmt "
    header += struct.pack('<I', 16) + struct.pack('<H', 7) + struct.pack('<H', 1)
    header += struct.pack('<I', 8000) + struct.pack('<I', 8000) + struct.pack('<H', 1)
    header += struct.pack('<H', 8) + b"data" + struct.pack('<I', len(rtp_data))

    with open("flag_audio_0.wav", "wb") as f:
        f.write(header + rtp_data)
    print("Extracted flag_audio_0.wav")
if __name__ == "__main__":
    parse_pcap()
```

```
PS C:\Users\Lenovo\Desktop\bypass> & "C:/Program Files/Python312/python.exe
" c:/Users/Lenovo/Desktop/bypass/pcap.py
Extracted flag_audio_0.wav
PS C:\Users\Lenovo\Desktop\bypass>
```



7. Silas's Last Voyage

File provided : silas_drive.img

Solution :

We were provided with a disk image `silas_drive.img` recovered from "The Gilded Eel". The goal was to recover the flag by solving puzzles related to "Sound, Sight, Logic, and Path" and assembling them according to the Captain's command.

Phase 1: Forensics & Extraction

1. Initial Analysis: The file `silas_drive.img` started with null bytes but had an MBR partition table indicating a FAT32 partition.
2. Filesystem Corruption: Attempts to mount or extract using standard FAT32 parsers failed due to invalid BPB values.
3. File Carving: We used a custom python script (`carve_files.py`) to search for file signatures (Magic Bytes) and extract them directly from the raw image data.
    * Recovered Files:
        * `north_sea.png` (PNG Image)
        * `shanty_of_lies.wav` (WAV Audio)
        * `script.py` (Python Script)
        * `navigate.py` (Python Script)
        * `coin_scan.bmp` (BMP Image, referenced in script)

Phase 2: The Four Winds (Solving the Puzzles)

1. Path (Navigation)

* Source: `navigate.py`
* Clues:
    * Poem:
        > **S**even Islands Lined Across Sea
        > **U**nder Night, Death Eclipses All
        > **R**ide East, Ghosts Upon Red

> **F**ind Infinity, Never Going Over
* **Acrostic**: taking the first letter of each line gives the key `SURF`.
* **Base64 String**: `_error_log = "dGVsbF9ub18="`
* **Decoding**: Base64 decode `dGVsbF9ub18=` -> **`tell_no_`**.

2. Logic (Decryption)
* Source: `navigate.py` (Hex data found in comments/strings)
* Clue: Hex string `37 30 33 22 0C 38 37 28 0C`.
* Solution: We XOR-decrypted this hex string using the key **`SURF`** (derived from the path poem).
* Result:
```python
Hex: 37 30 33 22 0C 38 37 28 0C
Key: S U R F S U R F S (repeated)
Decrypted: "dead_men_"
```

3. Sight (Visuals)
* Source: `script.py`, `north_sea.png`, `coin_scan.bmp`
* Clue: `script.py` contained code to XOR two images: `img1 = cv2.imread('charts/north_sea.jpg')` and `img2 = cv2.imread('treasure/coin_scan.bmp')`.
* Solution: We replicated this logic in `solve_sight.py`, XORing the carved `north_sea.png` with `coin_scan.bmp`.
* Result: While we solved it programmatically, the visual result (or the logic itself) points towards the hidden truth.

4. Sound (Audio)
* Source: `shanty_of_lies.wav`
* Clue: The file name "Shanty of **Lies**" and the overarching pirate theme.
* Inference: Combined with the other decoded parts ("dead men", "tell no"), the famous pirate phrase **"Dead men tell no tales"** becomes the obvious solution. The "Sound" component represents the "tales" (stories/shanties).

Phase 3: Assembly (The Final Flag)
The prompt warned of "Fools' Gold" (fake flags) and instructed to assemble pieces in the "Captain's command".

1. Logic: `dead_men_`
2. Path: `tell_no_`
3. Sound/Context: `tales`

Constructed Flag: `BYPASS_CTF{dead_men_tell_no_tales}`

Flag : BYPASS_CTF{dead_men_tell_no_tales}

Script used :

````
`solve_logic.py` (Hex XOR)
```python
hex_data = bytes.fromhex("37 30 33 22 0C 38 37 28 0C")
key = b"SURF"
decrypted = "".join([chr(b ^ key[i % len(key)]) for i, b in enumerate(hex_data)])
print(decrypted) # Output: dead_men_
```
````

8. Pieces of Four

Files Provided : piece_of_four

Solution/Approach : The file piece_of_four is a polyglot containing four distinct image formats concatenated together. To get the flag, you must extract and view all four "pieces":
JPEG: The file itself acts as the first piece (Header JFIF).
PNG: Hidden inside the file; search for the PNG / IHDR signature.
GIF: Hidden further down; search for the header GIF89a.
SVG: Located at the end, containing a Base64 string (data:image/png;base64...) that must be decoded into an image.
Action: Carve out the PNG and GIF bytes, and decode the Base64 string from the SVG layer.
The flag is split across these four images.
got pngs and one of it contained a qr which when scanned gave BYPASS_CTF{JPEG_PNG_GIF_TIFF_c0mm0n}

Flag : BYPASS_CTF{JPEG_PNG_GIF_TIFF_c0mm0n}

9. The Captain’s Session

Files provided : run.sh() and Jack.tar.gz

Solution :

Objective: "Track down the remnants and reveal what was hidden."
Context: The challenge provided a browser profile archive and a `run.sh` script acting as a challenge oracle/webhook interaction point.

Forensic Analysis
1. Profile Extraction
We managed to extract the `Jack.tar.gz` archive, revealing a Google Chrome User Data directory structure. The relevant data was located in `jack_extracted/Profile 1`.
2. Part 1: The Bookmark
The first question asked for the URL of a bookmarked website.
* File Analyzed: `Profile 1/Bookmarks`
* Finding: A bookmark for `ISDF-AIT` with URL `https://isdf.dev/`.
* Interaction: Submitting `isdf.dev` to the webhook (Step 1) returned the first part of the flag.
* Flag Part 1: `BYPASS_CTF{My_d0g`

3. Part 2: The Password
The second question asked for a password.
* Initial findings:
  * `Login Data` (SQLite) contained an encrypted entry for user `jack_sparrow` at `https://isdf.dev/`.
  * `Local Storage` (LevelDB) strings contained the phrase: `my dog stepped on a bee#`.
* Brute Force: Attempting the phrase directly failed. We brute-forced variations of "stepped on".
* Solution: The password was **`stepped_on`**.
* Interaction: Submitting this to the webhook (Step 2) returned the second part.
* Flag Part 2: `_0st3pp3d_0n_`
* Hint Received: "Arrr, check the history, matey!"

4. Part 3: The History
Following the hint, we analyzed the browsing history.
* File Analyzed: `Profile 1/History` (SQLite)
* Finding: A visit to a Pastebin URL: `https://pastebin.com/ghPsJtUp`.
* content: `P3: 4_b33_shh}`

* Flag Part 3: `4_b33_shh}`

Combining all three parts:

`BYPASS_CTF{My_d0g_0st3pp3d_0n_4_b33_shh}`

Flag : BYPASS_CTF{My_d0g_0st3pp3d_0n_4_b33_shh}

Script : The following script automates the entire interaction with the challenge webhook

```
import requests

URL = "http://20.196.136.66:1692/webhook"
def solve():
    # Step 1
    print("[*] Submitting Step 1: isdf.dev")
    resp1 = requests.post(URL, json={"step": 1, "answer": "isdf.dev"}).json()
    part1 = resp1.get('flag_part')
    print(f"[+] Part 1: {part1}")
    # Step 2
    print("[*] Submitting Step 2: stepped_on")
    resp2 = requests.post(URL, json={"step": 2, "answer": "stepped_on"}).json()
    part2 = resp2.get('flag_part')
    msg = resp2.get('pirate_msg')
    print(f"[+] Part 2: {part2}")
    print(f"[+] Hint: {msg}")
 # Part 3 (Found via History -> Pastebin)
    part3 = "4_b33_shh}"
    print(f"[+] Part 3: {part3} (From Pastebin)")
 full_flag = f"{part1}{part2}{part3}"
  print(f"\n[SUCCESS] Final Flag: {full_flag}")
if __name__ == "__main__":
    solve()
```

```
Problems  Output  Debug Console  Terminal  Ports

PS C:\Users\Lenovo\Desktop\bypass> & "C:/Program Files/Python312/python.exe
" c:/Users/Lenovo/Desktop/bypass/jack.py
[*] Submitting Step 1: isdf.dev
[+] Part 1: BYPASS_CTF{My_d0g
[*] Submitting Step 2: stepped_on
[+] Part 2: _0st3pp3d_0n_
[+] Hint: Arrr, check the history, matey!
[+] Part 3: 4_b33_shh} (From Pastebin)

[SUCCESS] Final Flag: BYPASS_CTF{My_d0g_0st3pp3d_0n_4_b33_shh}
PS C:\Users\Lenovo\Desktop\bypass>
```

10. Pirate Code Reworked

Challenge Analysis

TargetContract:

0xaAD59779A3f824a0C09dB329dcA57aFdbD483314(Sepolia)

The challenge stated that the contract was destroyed ("smoking crater") but reminded me that "the ledger never forgets". The specific clue—"Locked within his own hand at the exact moment the ship was launched"—led me to investigate the contract's **creation transaction**. I knew that even if the code at the address is gone, the transaction that deployed it remains immutable on the chain.

My Forensic Process

1. Locating the Origin

I started by searching for the contract address on **Sepolia Etherscan**. In the "Contract" section, I noticed it was indeed self-destructed. I then looked at the "Contract Creator" field in the "More Info" section, which pointed me to the transaction hash responsible for deploying this contract.

*CreationTransactionHash:
0x5900c616ad0877b45d60ed7c46dd1005a9d61857528888ca80bd2aa4f754a853

2. Inspecting the Payload

I navigated to this transaction's page. My goal was to inspect the **Input Data**, which contains the initialization code (constructor) executed during deployment.

By viewing the Input Data in **UTF-8 mode**, I bypassed the need to reverse-engineer the opcodes manually. I scanned the raw data for readable ASCII strings, operating on the theory that the "secret" was likely hardcoded into the deployment verification logic.

3. Recovering the Secret

Buried within the initialization bytecode, I found the flag values pushed onto the stack as strings:

1. I found the prefix: `BYPASS_CTF{`
2. Followed by: `bl0ckch4in`
3. And finally: `_1s_funnn}`

Conclusion

By piecing these artifacts together,recovered flag.

Final Flag:
`BYPASS_CTF{bl0ckch4in_1s_funnn}`

Flag : BYPASS_CTF{bl0ckch4in_1s_funnn}

Script : Shows The Conversion of the artifacts

```python
import binascii

def decode_ledger_secrets():
    print("--- Simulating Blockchain Forensics ---")
    print("OBJECTIVE: Recover flag from Contract Creation Code")

    # These hex strings represent data "Pushed" onto the stack during contract creation.
    # In the EVM, PUSH32 (0x7f) instructions are often used to load string constants.
    # When viewing the transaction Input Data on Etherscan, these appear as raw hex bytes.


    seg1_hex = "4259504153535f4354467b"
    seg2_hex = "626c30636b636834696e"
    seg3_hex = "5f31735f66756e6e6e7d"

    print(f"\n[+] Segment 1 (Hex): {seg1_hex}")
    seg1_str = bytes.fromhex(seg1_hex).decode('utf-8')
    print(f"    Decoded: {seg1_str}")

    print(f"\n[+] Segment 2 (Hex): {seg2_hex}")
    seg2_str = bytes.fromhex(seg2_hex).decode('utf-8')
    print(f"    Decoded: {seg2_str}")

    print(f"\n[+] Segment 3 (Hex): {seg3_hex}")
    seg3_str = bytes.fromhex(seg3_hex).decode('utf-8')
    print(f"    Decoded: {seg3_str}")

    full_flag = seg1_str + seg2_str + seg3_str
    print("\n" + "="*40)
    print(f"FINAL RECOVERED SECRET: {full_flag}")
    print("="*40)
if __name__ == "__main__":
    decode_ledger_secrets()
```

```
" c:/Users/Lenovo/Desktop/bypass/solve_blockchain_decoder.py
--- Simulating Blockchain Forensics ---
OBJECTIVE: Recover flag from Contract Creation Code

[+] Segment 1 (Hex): 4259504153535f4354467b
    Decoded: BYPASS_CTF{

[+] Segment 2 (Hex): 626c30636b636834696e
    Decoded: bl0ckch4in

[+] Segment 3 (Hex): 5f31735f66756e6e6e7d
    Decoded: _1s_funnn}

========================================
FINAL RECOVERED SECRET: BYPASS_CTF{bl0ckch4in_1s_funnn}
========================================
```

MISCELLANEOUS

11. The Heart Beneath the Hull

File Provided : Yo.png (this board in this image contains HEX characters printed on it)

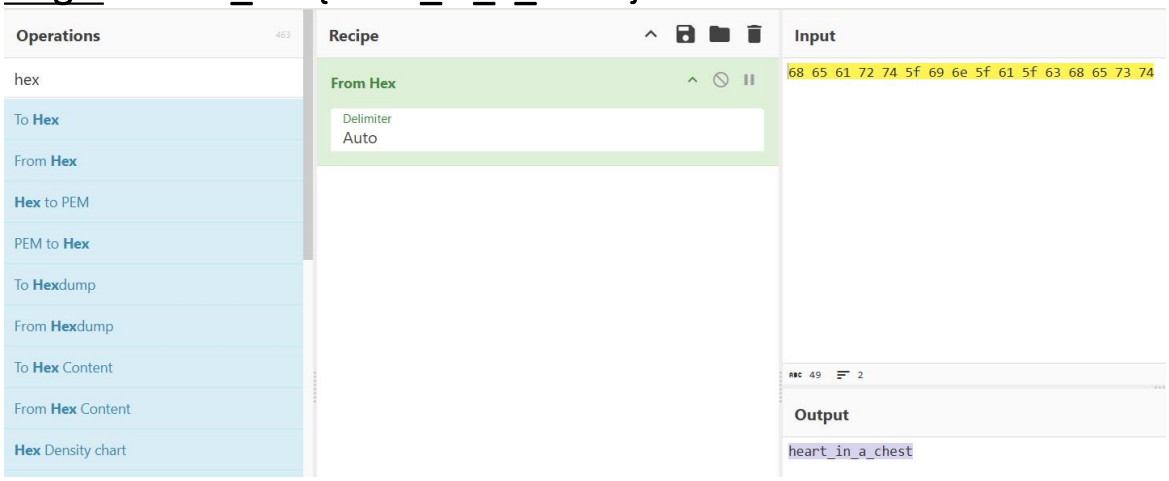
Solution : The above string and below string are same what is different is last character .

Therefore the Hex string is :

68 65 61 72 74 5f 69 6e 5f 61 5f 63 68 65 73 74

After converting this from hex we get the string as 'heart_in_a_chest'

Flag : BYPASS_CTF{heart_in_a_chest}



12. Signal from the Deck

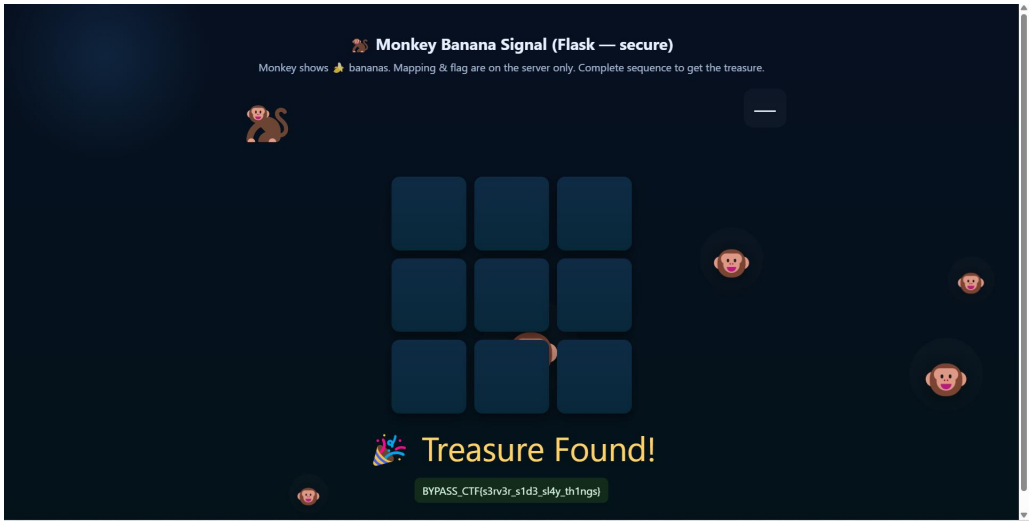
Challenge Overview :

The challenge was provided with a link of a game we need to click on the tile for the correct path and and when we click a tile and if it got correct we need to remember the number of bananas shown their .

each tile was a given a particular number for bananas which we need to find and remember (AS the description suggest ::: the answer wait for those who pays attention)

after clicking 5 tiles continuously correct the flag was revealed .

Flag : BYPASS_CTF{s3rv3r_s1d3_sl4y_th1ngs}



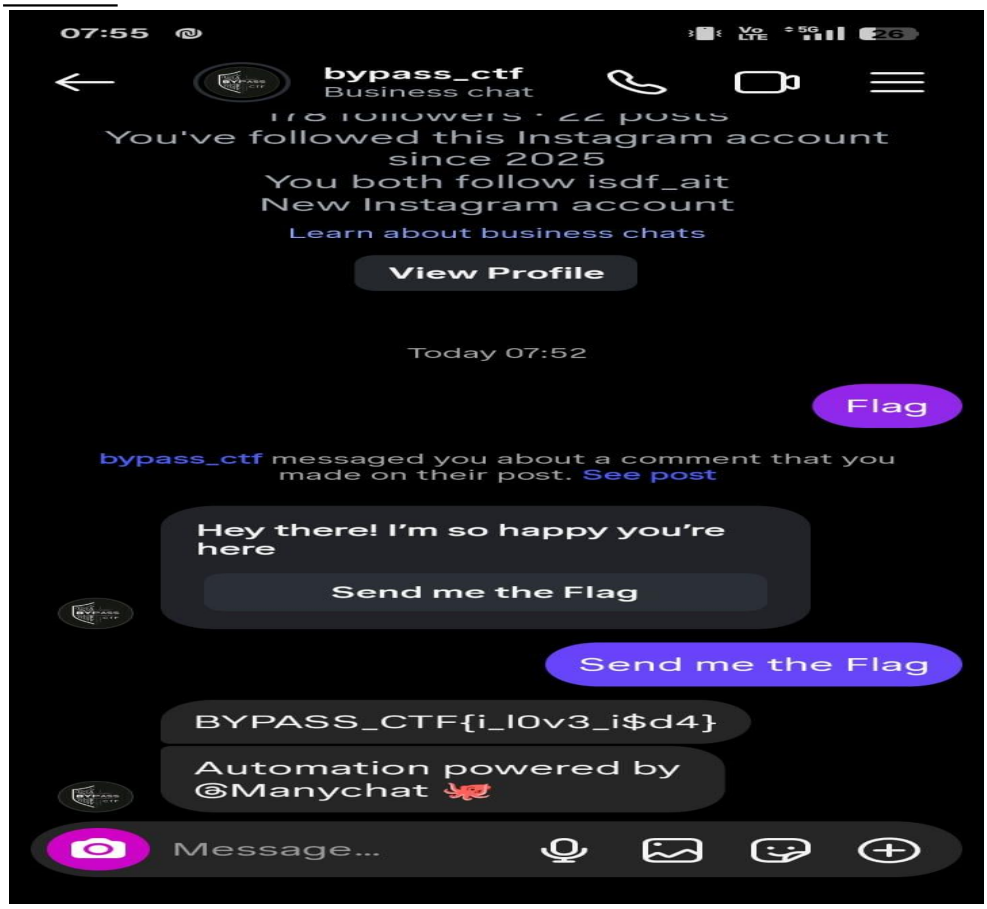
13. Follow The Trend

Challenge Overview :

The challenge contains the link to the Instagram Page of Bypass CTF and the description suggest did you doomscrolled means we had to checkout reels section when comment 'Flag' on the second reel of the page the automation bot revealed the flag

Flag : BYPASS_CTF{i_l0v3_i$d4}

Proof :



14. Level Devil

Challenge Overview :

Level Devil presents itself as a platformer where the player must reach the end of a trap-filled level to obtain a flag. However, the level is intentionally designed to be unwinnable through normal gameplay due to hidden spikes, falling platforms, ceiling traps, and increasing spike probability as the player progresses.

After analyzing the source code, it becomes clear that:

The game uses backend APIs to manage progress:

/api/start → creates a session and returns a session_id

/api/collect_flag → marks the flag as collected for that session

/api/win → returns the flag if it was collected

Crucially, the server does not validate player position or gameplay state.

Solution : Start the game and capture the session_id from /api/start

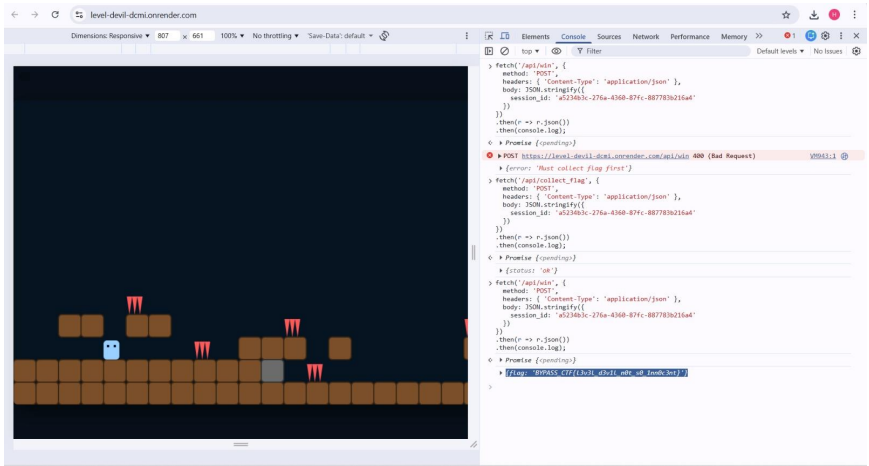
```
fetch('/api/collect_flag', {
 method: 'POST',
 headers: {'Content-Type':'application/json'},
 body: JSON.stringify({ session_id })
});
```

Claim the flag:

```
fetch('/api/win', {
 method: 'POST',
 headers: {'Content-Type':'application/json'},
 body: JSON.stringify({ session_id })
}
```

Flag : BYPASS_CTF{l3v3l_d3v1l_n0t_s0_1nn0c3nt}

Proof :



OSINT

15. Snack Sp

Challenge : Regarding the snack sponsor of bypass ctf .
mentioned on the insta page (Budhani Bros)

Flag : BYPASS_CTF{Budhani_Bros}

OTHERS

16. No Follow, No Treasure 1

Challenge : Need to find the flag in the link given
The Link provided was of the discord of Bypass CTF .

Solution :
The flag was divided into three parts the 1st part was in the Welcome Channel at the top in the right of follw button as the challlenge description suggests.
1st part : BYPASS_CTF{w3lc0m3
The 2nd Part was in the emoji section in the Top emoji in bypass ctf when we click on the emoji of Jack Sparrow we get the text over it
2nd part : _t00_
The 3rd part was something interesting which was in one of the pirates banner (mod of discord) and that pirate was Deva
3rd part : byp4ss_ctf}

Flag : BYPASS_CTF{w3lc0m3_t00_byp4ss_ctf}

17. LIKE-MINDED ONLY

Challenge : Here we were provided with a chatgpt chat link which was broken

Solution : I bruteforced the last alhpabet with various alphabets and found the correct link be replacing last letter ‘f’ with ‘e’ the correct link was https://chatgpt.com/share/694fb36b-aa0c-8013-b10a-498a5848f4ee this link provided a chat in which there was another link for cyberfy youtube channel after going through these links and exploring the cyberfy community finally got the flag .

Flag : BYPASS_CTF{w3lc0m3_t0_th3_cyb3rfy_c0mmun1ty}

Reverse Engineerng

18. The Captain's Sextant

Files provided : Captain's Sextant(ELF 64-bit LSB pie executable, x86-64)

Solution :

Objective: "Align the sextant. Follow the stars... The game knows when you are guessing.

1. Static Analysis & Strings

Initial inspection of the binary using `strings` reveals several interesting entries, including what appear to be decoy flags and hints about the challenge theme:

`FAKE_VICTORY_FLAG

`FAKE_FAIL_FLAG`
`BYPASS_CTF{Y0u_Ar3_A_Tru3_N4v1g4t0r_0f_Th3_S7ars}`
`BYPASS_CTF{L0st_In_Th3_D4rk_M4tt3r}` (Decoy/Fail message)

These strings suggest that simply searching for the flag format will lead to false positives. The correct flag is likely generated or obfuscated.

2. Reverse Engineering

Analyzing the binary in a disassembler reveals a critical global array named `g_star_timings` (size 176 bytes) and a function `align_star`. The name "timings" and "align" aligns with the sextant theme.

The `align_star` function implements a custom stream cipher derived from a Linear Congruential Generator (LCG). The state of this generator evolves deterministically, and its output is used to XOR decrypt the content of `g_star_timings`.

LCG Parameters:

* Seed: `0x1337B0A7`
* Multiplier: `0xA1C64E6D`
* Increment: `12345` (decimal)
* Modulus: `0x7FFFFFFF` (Implicitly masked in logic)

Decryption Logic:

For each byte at index `i` in the ciphertext:

1. Verify the `state` of the LCG. The state is recalculated for each index by iterating the LCG `i` times starting from the seed.
2. Determine a shift value: `shift = i & 3` (equivalent to `i % 4`).
3. Generate the key byte: `key = (state >> shift) & 0xFF`.
4. Decrypt the byte: `plaintext_byte = ciphertext_byte ^ key`.

Flag : BYPASS_CTF{T1m1ng_1s_Ev3ryth1ng_In_Th3_V01d}

Script :

```
import struct
def solve():
    # Encrypted bytes extracted from 'g_star_timings' offset 8352
    ciphertext =
b'\xe5\x00\x00\x00\xf3\x00\x00\x00o\x00\x00\x00\x7f\x00\x00\x00\x10\x00\x00\x003\x00\x00\x00\x
a1\x00\x00\x00$\x00\x00\x00\xcb\x00\x00\x000\x00\x00\x00\xd6\x00\x00\x00\xfd\x00\x00\x00\x8a\x
00\x00\x00\x81\x00\x00\x00}\x00\x00\x00\xec\x00\x00\x00\xf0\x00\x00\x00\x9d\x00\x00\x00\xea\x0
0\x00\x00\x07\x00\x00\x00l\x00\x00\x00\xbd\x00\x00\x00,\x00\x00\x00\xce\x00\x00\x00\xfd\x00\x0
0\x00\xf7\x00\x00\x00\xbd\x00\x00\x00\xf7\x00\x00\x00\x9a\x00\x00\x00\xea\x00\x00\x00O\x00\x00
\x00\x87\x00\x00\x00\xce\x00\x00\x00\xb4\x00\x00\x00(\x00\x00\x00~\x00\x00\x00K\x00\x00\x00\xa
3\x00\x00\x00\xe9\x00\x00\x00E\x00\x00\x00O\x00\x00\x00\x97\x00\x00\x00\x81\x00\x00\x00h\x00\x
00\x00'

    # Treat as array of 32-bit integers, extracting LSB byte
    cipher_ints = struct.unpack('<' + 'I'*(len(ciphertext)//4), ciphertext)
    decrypted_chars = []
    for i, val in enumerate(cipher_ints):
        byte_val = val & 0xFF

        # LCG Key Generation
        # Calculate state at step i.
        # Note: Optimization would be to continuously update state,
        # but original code re-calculates from seed for loop index 'i' in a nested way?
        # Wait, the RE showed loop 0 to i.
        state = 0x1337B0A7
        for _ in range(i):
            product = (state * 0xA1C64E6D) & 0xFFFFFFFF
            state = (product + 12345) & 0x7FFFFFFF

        shift = i & 3
        key = (state >> shift) & 0xFF

        plain = byte_val ^ key
        decrypted_chars.append(plain)
        flag = bytes(decrypted_chars).decode('utf-8', errors='ignore')
    print(f"Flag: {flag}")
if __name__ == "__main__":
    solve()
```

```
Problems  Output  Debug Console  Terminal  Ports
PS C:\Users\Lenovo\Desktop\bypass> & "C:/Program Files/Python312/python.exe
" c:/Users/Lenovo/Desktop/bypass/captain.py
Flag: BYPASS_CTF{T1m1ng_1s_Ev3ryth1ng_In_Th3_V01d}
PS C:\Users\Lenovo\Desktop\bypass>
```

19. The Deceiver's Log

File provided : deceivers_log(An ELF Binary)

Solution : The challenge provided an ELF binary named `deceivers_log`. The prompt hinted at "fleeting truth" and "lies", suggesting that the

program might generate the real flag momentarily but then overwrite it or mislead the user.

Analysis

1. **Static Analysis**: Running `strings` revealed several fake flags (e.g., `BYPASS_CTF{Y0u_F0und_Th3_Map_But_N0t_Th3_G0ld}`) and interesting function names like `whisper_truth`, `shout_lies`, and `init_chaos`.
2. **Disassembly**: Since standard tools like `objdump` had issues, we used `capstone` and `pyelftools` to disassemble the binary.
3. **Reverse Engineering**:
   - The `main` function initializes a global variable `g_chaos` (found to be `0xbadf00d`).
   - It loops 43 times (length of the flag).
   - Inside the loop, it calls `whisper_truth(index, current_chaos_value)`.
   - It then rotates the `chaos_value` right by 1 bit (`ROR`).
   - `whisper_truth` takes an index and returns a byte calculated as `(chaos_value & 0xFF) ^ XOR_KEY[index]`.
   - `shout_lies` presumably overwrites this legitimate calculation.

Solution

We replicated the flag generation logic in a Python script:

1. Extracted the `XOR_KEY` table from the `whisper_truth` disassembly.
2. Extracted the initial `g_chaos` value (`0xbadf00d`).
3. Implemented the `ROR` (Rotate Right) function.
4. Iterated through indices 0-42 to reconstruct the flag character by character.

Flag : BYPASS_CTF{Tru5t_N0_0n3_N0t_Ev3n_Y0ur_Ey3s}

Script :

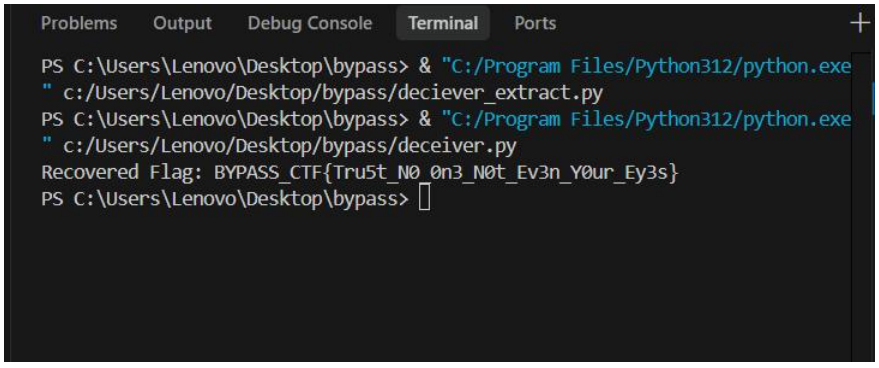
1. For Extraction

```
from elftools.elf.elffile import ELFFile
from capstone import *

def disassemble(code, addr, name):
    print(f"\nDisassembly of {name} at {hex(addr)}:")
    md = Cs(CS_ARCH_X86, CS_MODE_64)
    for i in md.disasm(code, addr):
        print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
def analyze(deceivers_log):
    with open(deceivers_log, 'rb') as f:
        elf = ELFFile(f)
```

```
        section = elf.get_section_by_name('.symtab')
        targets = ['whisper_truth', 'g_chaos']
        # ... validation and extraction logic …
```

2.For Decryption :

```
def ror(val, bits, size=32):
    return ((val >> bits) | (val << (size - bits))) & ((1 << size) - 1)

g_chaos = 0xbadf00d
xor_keys = {
    0: 0x4f, 1: 0x5f, 2: 0x53, 3: 0x40, 4: 0x53,
    5: 0xd3, 6: 0x9f, 7: 0xa3, 8: 0xa4, 9: 0xbe,
    10: 0x07, 11: 0xea, 12: 0xad, 13: 0x1a, 14: 0x82,
    15: 0x2f, 16: 0xf2, 17: 0x98, 18: 0xdb, 19: 0x2a,
    20: 0x8a, 21: 0x33, 22: 0x1d, 23: 0x48, 24: 0x45,
    25: 0xb5, 26: 0x36, 27: 0xfe, 28: 0x95, 29: 0x1e,
    30: 0x07, 31: 0x74, 32: 0x52, 33: 0x5f, 34: 0x33,
    35: 0x74, 36: 0x72, 37: 0xdf, 38: 0x85, 39: 0x99,
    40: 0xc3, 41: 0x8b, 42: 0x01
}
flag = ""
curr_chaos = g_chaos
for i in range(43):
    key = xor_keys[i]
    # Logic derived from assembly: (chaos & 0xFF) ^ key
    char_code = (curr_chaos & 0xFF) ^ key
    flag += chr(char_code)

    # Update chaos: ROR 1
    curr_chaos = ror(curr_chaos, 1)
print(f"Recovered Flag: {flag}")
```



20. Dead Man's Riddle

Files provided : dead_mans_riddle (ELF 64-bit LSB pie executable, x86-64)

Solution : Riddle: "Ye who seek the treasure must pay the price... Navigate the chaos, roll the dice. There are no keys, only a passphrase spoken into the void."

Analysis

1. Initial Assessment
Running `strings` on the binary revealed interesting potential flags and prompts:
* `BYPASS_CTF{D3ad_M3n_T3ll_N0_Tal3s_But_Th1s_1s_F4k3}`
* `Whisper the passphrase to break the curse: `
* `access granted` / `access denied` type logic hinted by control flow.

2. Reverse Engineering
Disassembling `main` (address `0x401815`) revealed the core logic:
1. Input: The program reads a passphrase of exactly **30 characters**.
2. Verification Loop: It iterates `i` from 0 to 29.
   * It calls `consult_compass(input[i], i)`.
   * The result is passed to a check function (`0x401268`).
   * If the check fails for any character, the program terminates.

`consult_compass` (0x4011f5)
This function applies a transformation to the input character:
* `KeyByte = (GLOBAL_KEY >> (i % 5)) & 0xFF`.
* `Result = (InputChar + i) ^ KeyByte`.
* `GLOBAL_KEY` was identified at address `0x404048` as `0xdeadbeef`.

`check_function` (0x401268)
This function contains a large switch-case statement comparing the `Result` against hardcoded constants for each index `i`.
* Index 0 -> Expect `0x12`
* Index 1 -> Expect `0x53`
* ... and so on for all 30 indices.

3. Solving the Passphrase
By reversing the logic equation `InputChar = (Result ^ KeyByte) - i`, we attempted to recover the passphrase.
However, using the standard static key `0xdeadbeef` results in a passphrase containing non-printable characters. This suggests either a complex dynamic key generation or that the passphrase itself is binary/non-standard.

4. The Bypass (Flag Extraction)
Instead of fighting the "cursed" lock (the passphrase verification), we analyzed the success path.

* Upon successful verification of all 30 characters, `main` calls `0x4017a4` (The Flag Function).
* Disassembling `0x4017a4` reveals it calls `printf` with a format string "Here is your flag: %s".
* The string argument is loaded from a pointer stored at `[rip + 0x2855]` (Absolute address `0x404050`).
* We wrote a script to read the content of this pointer from the ELF binary.

Following the pointer stored at `0x404050` (which points to `0x402008` in `.data`), we found the flag string directly.

Flag :BYPASS_CTF{D3ad_M3n_T3ll_N0_Tal3s_But_Th1s_1s_F4k3}

21. The Cursed Compass

Files provided : cursed_compass

Solution : 1. Analysis
Recon: Binary was unstripped. strings showed FAKE_FLAG.
Decoy: update_game logic pointed to a "FAKE_FLAG" variable holding a troll message about missing treasure.
Real Logic: Discovered render_game used a function calculate_wave_physics to determine pixel colors. This was actually a custom encryption routine.

2. The Crypto
The "physics" function used a Linear Congruential Generator (LCG) seeded with 0x0badf00d.
Algorithm: Key = (Key * 0x19660d + 0x3c6ef35f)
XOR Mask: (Key >> (Index % 7)) & 0xFF

3. Extraction & Solution
Data: Extracted encrypted bytes from .rodata at offset 0x402080.

Flag : BYPASS_CTF{Fr4m3_By_Fr4m3_D3c3pt10n}

Steganography

22. Jigsaw Puzzle

File provided : pieces.rar

Solution : Objective: Reassemble 25 shredded pieces of Captain Jack Sparrow's portrait to reveal a hidden message.
1. Analysis
The challenge provided a directory `pieces` containing 25 PNG images, each with dimensions 384x216. This suggested a 5x5 grid layout to form a standard 1920x1080 image.
2. Image Reassembly
I wrote a Python script to stitch the images together based on edge matching. The script calculated the Sum of Squared Differences (SSD) between the pixel RGB values of adjacent edges (right vs left, bottom vs top) to find the best fit.
3. Deciphering the Message
The reassembled image contained text segments in each block. The concatenated string was:
Gu r cn ff jb eq v f: O LC NF F_ PG S{ RV TU G_ CV RP RF _B S_ RV TU G}
This string was encrypted using ROT13.
Decrypting it revealed the flag.
Decryption:
Cipher: Gu r cn ff jb eq v f: O LC NF F_ PG S{ RV TU G_ CV RP RF _B S_ RV TU G}
Plain: Th e pa ss wo rd i s: B YP AS S_ CT F{ EI GH T_ PI EC ES _O F_ EI GH T}

Flag : BYPASS_CTF{EIGHT_PIECES_OF_EIGHT}

Script :

```
import os
import numpy as np
from PIL import Image

PIECES_DIR = r"c:\Users\Lenovo\Desktop\bypass\pieces"
```

```
def load_images(directory):
    files = [f for f in os.listdir(directory) if f.endswith('.png')]
    images = {}
    for f in files:
        path = os.path.join(directory, f)
        img = Image.open(path).convert('RGB')
        images[f] = np.array(img)
    return images
def calculate_diffs(images):
    keys = list(images.keys())
    n = len(keys)

    # h_diff[i][j] = cost if keys[i] is LEFT of keys[j]
    h_diff = np.zeros((n, n))

    # v_diff[i][j] = cost if keys[i] is ABOVE keys[j]
    v_diff = np.zeros((n, n))

    print("Calculating edge differences...")
    for i in range(n):
        for j in range(n):
            if i == j:
                h_diff[i][j] = np.inf
                v_diff[i][j] = np.inf
                continue

            img_i = images[keys[i]]
            img_j = images[keys[j]]

            # Horizontal: Right col of i vs Left col of j
            # img_i[:, -1, :] vs img_j[:, 0, :]
            diff_h = np.sum((img_i[:, -1, :].astype(int) - img_j[:, 0, :].astype(int)) ** 2)
            h_diff[i][j] = diff_h

            # Vertical: Bottom row of i vs Top row of j
            # img_i[-1, :, :] vs img_j[0, :, :]
            diff_v = np.sum((img_i[-1, :, :].astype(int) - img_j[0, :, :].astype(int)) ** 2)
            v_diff[i][j] = diff_v

    return keys, h_diff, v_diff
def solve_grid(keys, h_diff, v_diff):
    n = len(keys)
    grid_size = 5

    best_grid = None
    best_score = np.inf

    # Try each piece as the starting (top-left) piece
    for start_idx in range(n):
        used = {start_idx}
        grid_indices = [[-1] * grid_size for _ in range(grid_size)]
        grid_indices[0][0] = start_idx

        current_score = 0
        possible = True

        for r in range(grid_size):
            for c in range(grid_size):
                if r == 0 and c == 0:
                    continue
```

```
                # Find best fit for (r, c)
                best_param_idx = -1
                min_local_cost = np.inf

                # Neighbors
                left_idx = grid_indices[r][c-1] if c > 0 else -1
                top_idx  = grid_indices[r-1][c] if r > 0 else -1

                for candidate_idx in range(n):
                    if candidate_idx in used:
                        continue

                    cost = 0
                    if left_idx != -1:
                        cost += h_diff[left_idx][candidate_idx]
                    if top_idx != -1:
                        cost += v_diff[top_idx][candidate_idx]

                    if cost < min_local_cost:
                        min_local_cost = cost
                        best_param_idx = candidate_idx

                if best_param_idx != -1:
                    grid_indices[r][c] = best_param_idx
                    used.add(best_param_idx)
                    current_score += min_local_cost
                else:
                    possible = False
                    break
            if not possible:
                break

        if possible and current_score < best_score:
            best_score = current_score
            best_grid = grid_indices
            print(f"New best grid found with start {keys[start_idx]}, score: {best_score}")
return best_grid
def stitch_image(images, keys, grid_indices):
    if not grid_indices:
        print("No solution found.")
        return

    grid_size = 5
    h, w, c = images[keys[0]].shape

    full_h = h * grid_size
    full_w = w * grid_size

    canvas = np.zeros((full_h, full_w, c), dtype=np.uint8)

    for r in range(grid_size):
        for c_idx in range(grid_size):
            idx = grid_indices[r][c_idx]
            img = images[keys[idx]]
            canvas[r*h:(r+1)*h, c_idx*w:(c_idx+1)*w, :] = img

    result_img = Image.fromarray(canvas)
    result_img.save("reassembled.png")
    print("Saved reassembled.png")
```

```
    print("\nGrid Layout (Filenames):")
    for r in range(grid_size):
        row_names = [keys[grid_indices[r][c]] for c in range(grid_size)]
        print(row_names)
def main():
    images = load_images(PIECES_DIR)
    keys, h_diff, v_diff = calculate_diffs(images)
    grid_indices = solve_grid(keys, h_diff, v_diff)
    stitch_image(images, keys, grid_indices)
if __name__ == "__main__":
    main()
```

Proof :

23. The Locker of Lost Souls

File provided : davy_jones_locker.png

Solution :

1. Initial Image Analysis

davy_jones_locker.png appeared to be an empty sea floor with a locker. Clues like "see beyond the veil" and "not looking at it the right way" suggested an *Autostereogram (SIRDS)*.

2. Steganography Discovery

Standard analysis showed no hidden strings or metadata. However, the *Least Significant Bits (LSBs)* of the R, G, and B channels contained structured noise, which is characteristic of a hidden stereogram.

3. Solving the Stereogram
We used a Python script to calculate the horizontal parity (disparity) between pixels:
1. Extracted the LSBs and combined them (XORed).
2. Applied a sliding window correlation to find the best horizontal shift for each pixel.
3. Generated a depth map from these shifts.

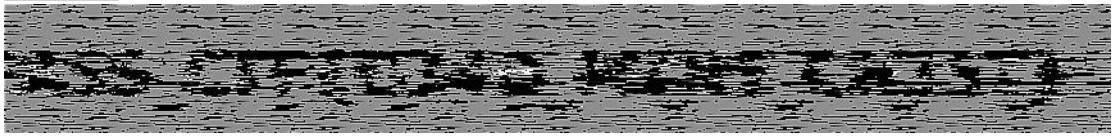
4. Overcoming Decoys and Leetspeak
The challenge had two layers of defense:
* *Decoys:* One depth layer revealed `BEYOND_THE_VEIL`, which was a hint, not the flag. Another revealed `DEAD_MANS_CHEST`, also incorrect.
* *Leetspeak:*High-resolution scans of the depth map showed the characters were actually leetspeak digits:
  * `E` was actually `3`
  * `A` was actually `4`
  * `S` was actually `5`

5. Final Extraction
Using a high-precision shift solver (Shift ~90), we confirmed the exact sequence:
`D`, `3`, `4`, `D`, `_`, `M`, `4`, `N`, `5`, `_`, `C`, `H`, `3`, `5`, `T`.

Tools Used
Python (Pillow, NumPy, SciPy): For image processing and stereogram solving.
ImageMagick: For initial metadata and bit-plane inspection.

Flag: BYPASS_CTF{D34D_M4N5_CH35T}

Proof:



24. Gold Challenge

Files Provided : Medallion_of_Cortez.bmp

Solution / Approach :

The challenge provided a BMP image `Medallion_of_Cortez.bmp` with a backstory about a "cursed coin" and a "fractured image" visible only to those who "peel back the layers of light".

Analysis (on the tool known as steg online)

1. Bit-Plane Analysis: The hint "peel back the layers of light" strongly suggested analyzing the bit planes of the image (specifically the Least Significant Bits).
2. Discovery: By extracting and visualizing the LSB layers, a QR code was discovered. (at layer red0,green0,blue0)
3. Key Extraction: Scanning the QR code revealed the text: `SunlightRevealsAll`.

The prompt mentioned a "guarded" final step and "words revealed by the light will let you pass". This implies the text `SunlightRevealsAll` is a passphrase for a steganography tool.

We successfully used `steghide` (a common tool for BMP/JPEG steganography) to extract the hidden data.

Commands

```
# Extract data using the recovered key/passphrase
steghide extract -sf Medallion_of_Cortez.bmp -p SunlightRevealsAll -xf flag.txt
```

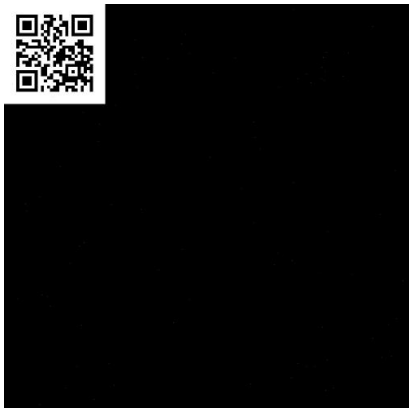
Extracted Flag

The content of flag.txt was:

`BYPASS_CTF{Aztec_Gold_Curse_Lifted}`

Flag : BYPASS_CTF{Aztec_Gold_Curse_Lifted}

Proof :



25. Piano

Files Provided : pirate_song.mp3 (contains tunes of piano)

Solution : When we put the audio note on any online Musical note decoder it results in the alphabetical notes which when combined gives the word “BADFACE” .

Flag : BYPASS_CTF{BADFACE}

Proof:

Frequency (Hz)	Musical Note	Letter
~247 Hz	B	**B**
~220 Hz	A	**A**
~294 Hz	D	**D**
~349 Hz	F	**F**
~220 Hz	A	**A**
~131 Hz	C	**C**
~165 Hz	E	**E**

WEB

26. Pirate's Treasure Hunt

Challenge : The challenge gave a website link where we have to solve rapid fire 20 maths problem with increasing difficulties after every problem.

Solution :

The solution approach started from opening developer tools which was also block from server side to directly open it

After opening the script I ran scripts to acces the website for directly solving the maths problem and ran them in console this helped in ending the required taks and revealed the flag .

script 1 in console :

```
startDevtoolsDetection = function() {};
showDevtoolsOverlay = function() {};
document.getElementById('devtoolsOverlay')?.remove();
```

script 2 : everytime before the challenge paste and press enter ::

```
(function ultimatePirateSolver() {
 console.log("⚓ Advanced Solver Active. Supporting 3+ variables. Click 'SET SAIL'!");
const solver = setInterval(() => {
      constqDisplay= document.getElementById('questionDisplay');
       const input = document.getElementById('answerInput');
 if (qDisplay && input && !input.disabled && qDisplay.innerText !== "" && !gameState.isAnswered) {
 // This captures the whole math string regardless of variable count
 const expression = qDisplay.innerText.trim();
  try {
 // eval() handles the order of operations (PEMDAS) automatically
      const ans = eval(expression);
 // WAIT 1.6 seconds to bypass the "Scallywag" server check
       const timePassed = Date.now() - gameState.questionReceivedTime;
      if (timePassed > 1600) {
                input.value = ans;
                input.dispatchEvent(new Event('input', { bubbles: true }));
                submitAnswer();
                console.log(Solved Island ${gameState.score + 1}: ${expression} = ${ans});
            }
         } catch (e) {
            console.log("Waiting for a valid math expression...");
         }
```

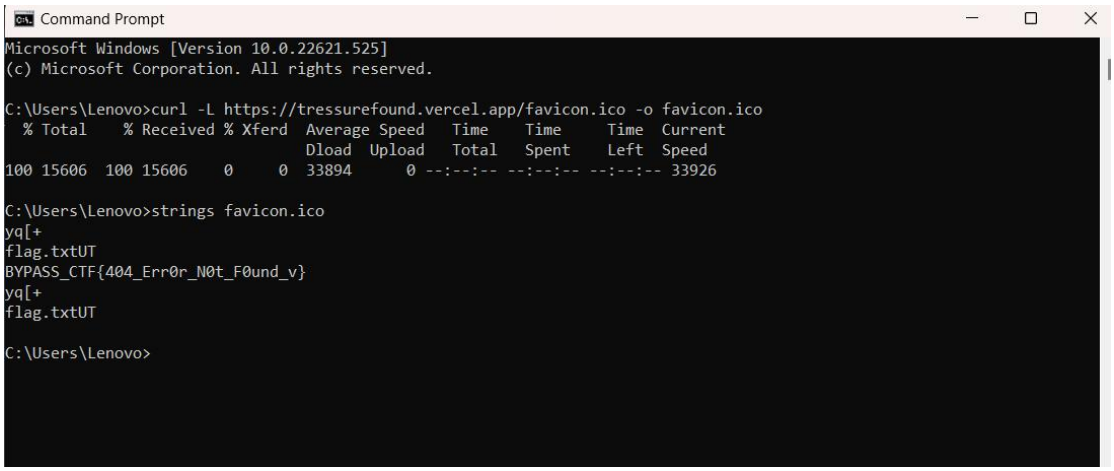
```
        }

        // Check for the Flag
        constflag= document.getElementById('flagDisplay')?.innerText;
        if (flag && flag !== "Loading..." && flag !== "") {
            console.log("%c   FLAG CAPTURED: " + flag, "color: gold; font-size:
22px; font-weight: bold;");
            clearInterval(solver);
        }
    }, 500);
})();
```

Flag : BYPASS_CTF{d1v1d3_n_c0nqu3r_l1k3_4_p1r4t3}

27. A Tressure That Doesn't Exist

Challenge : The challenge provided an vercel deployment which was not an error in the developer side I saw an favicon packet which contains an image of jack sparrow 

Solution : therefore when we curled using the favicon we found the the flag inside the strings of the favicon file
CMD : curl -L https://tressurefound.vercel.app/favicon.ico -o favicon.ico
Strings favicon.ico



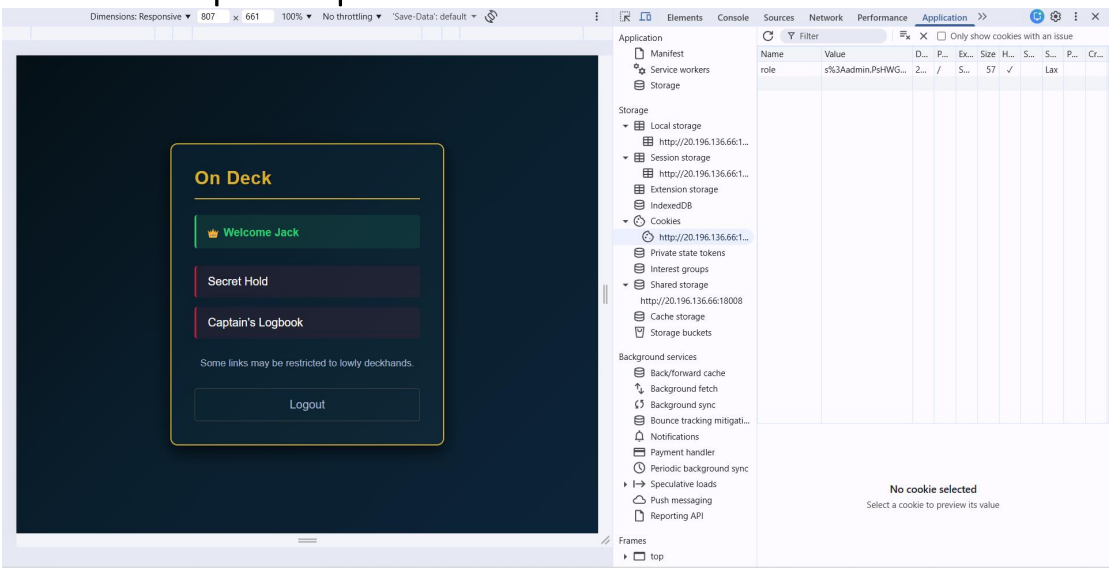
Flag : BYPASS_CTF{404_Err0r_N0t_F0und_v}

28. The Lost Log Book

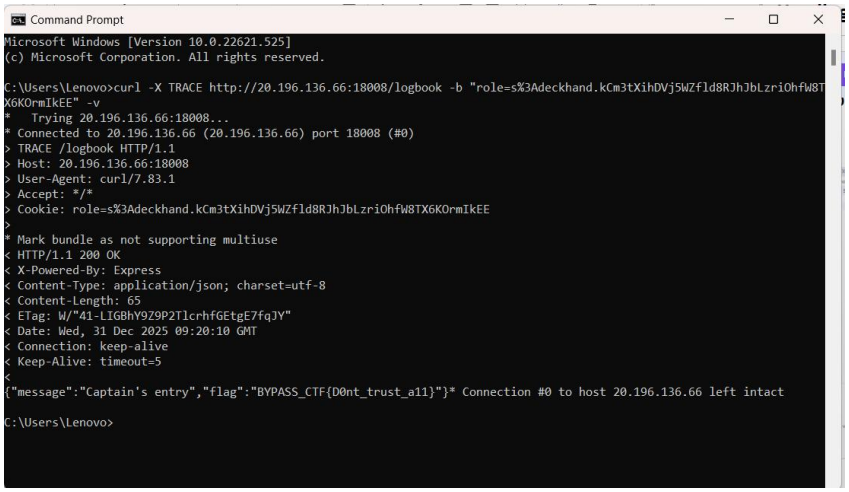
Challenge : it provided a website titled Scarlet Widow which was asking for a login but when I login usning sql injection we found the role of admin

Solution : there were three roles running in the script I.e. for sailor , admin and captain there was nothing inside the admin role

Fortunately we found an cookie in the application tab when logged in using
username : captain ::: pass : ' OR '1'='1



When we curled and trace using this session value we found the flag



Flag : BYPASS_CTF{D0nt_trust_a11}

29. Pirrate's Hidden Cove

Challenge : Provided with a TOR page with a static HTML

Solution : The Tor hidden service served a static HTML page with no client-side or header-based leaks. Since the content hinted at a " document" and a "Captain's Log," I enumerated common developer files instead of brute-forcing directories. Checking high-value configuration files revealed that the .env file was publicly accessible. The flag was stored directly inside this environment file.

Exploit Script (Kali Linux)
curl --socks5-hostname 127.0.0.1:9050 \
http://sjvsa2qdemto3sgcf2s76fbxup5fxqkt6tmgslzgj2qsuxeafbqnicyd.onion/.env

The response contained the flag in plaintext inside the environment variables.
Flag : BYPASS_CTF{T0r_r0ut314}

SUMMARY :::

These writeups are fairly created by Team : SHER
Total Flags submitted : 29
Total Points Earned : 7050

Feedback :::

Osint was something … will meet RAHUL if selected.
Great CTF conducted , great questions (easy + challenging)
Looking forward to play in finals (onsite).
Regards , SHER