CloudSek CTF – Proof of Concept (PoC)

1.CTF Write-Up: Nitro (Scripting Challenge)

Category: Scripting

Points: 100

Status: Solved ✔

Challenge Description

The challenge provides a `/task` endpoint which returns a **random string** inside an HTML snippet.

The goal is to **process this string under strict time limitations** and submit the correct result to `/submit`.

Manual solving is impossible because the server invalidates the string after a few milliseconds —

Automation is mandatory.

Objective

For every request:

1. Fetch the random string from `/task`
2. Reverse the string
3. Base64-encode the reversed string
4. Wrap using the exact format:

```
CSK__{base64_string}__2025
```

5. POST it to `/submit` before the timer expires

6. Retrieve the flag upon correct submission

Approach Summary

To reliably solve the challenge, I wrote an automation script in Python that:

- Uses a persistent HTTP session for faster repeated communication
- Extracts the dynamic string using a regex pattern (handles different HTML formats)
- Performs all required transformations
- Submits the payload instantly
- Repeats until the server accepts the answer and returns the flag

This ensures success **even under extreme timing constraints**.

Important Logic Explained

1. Extracting the Dynamic String

The server's response structure is not fixed (`<p>` tag sometimes present, sometimes not),

so I used a flexible regex:

```
input string: <optional <p>> + ([A-Za-z0-9+/=]+)
```

This made extraction robust against format variations.

2. Applying the Required Transformations

The challenge expected the following pipeline:

- Reverse the string:
  `s[::-1]`
- Base64 encode:
  `base64.b64encode().decode()`

- Produce final formatted payload:

  `CSK__{encoded_value}__2025`

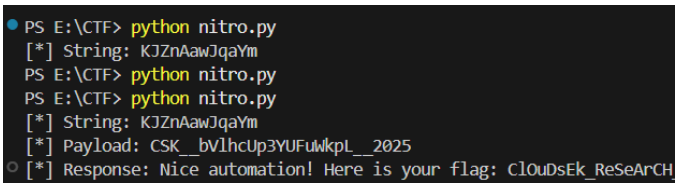
Any deviation (missing underscores, wrong casing, extra space) results in rejection.

3. Automating Submission with `requests.Session()`

Using `Session()` instead of making new HTTP connections each time reduces latency and avoids dropped submissions due to handshake delays.

My Final Script (screenshot below)

```python
import requests
import re
import base64

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

def get_task_string():
    r = session.get(BASE_URL + "/task")
    html = r.text


    m = re.search(r"input string:\s*(?:<p>)?([A-Za-z0-9+/=]+)(?:</p>)?", html)
    if not m:
        raise ValueError(f"Random string nahi mili! HTML: {html!r}")

    return m.group(1).strip()

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

def submit_payload(payload: str) -> str:

    r = session.post(BASE_URL + "/submit", data=payload)
    return r.text

def main():
    while True:
        s = get_task_string()
        payload = make_payload(s)

        print("[*] String:", s)
        print("[*] Payload:", payload)

        resp = submit_payload(payload)
        print("[*] Response:", resp)

        if "flag" in resp.lower():
            print(" ")
            break

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



```
PS E:\CTF> python nitro.py
[*] String: KJZnAawJqaYm
PS E:\CTF> python nitro.py
PS E:\CTF> python nitro.py
[*] String: KJZnAawJqaYm
[*] Payload: CSK__bVlhcUp3YUFuWkpL__2025
[*] Response: Nice automation! Here is your flag: ClOuDsEk_ReSeArCH
```

Result

2.CTF Write-Up: Bad Feedback (Web Challenge)

Category: Web Exploitation

Points: 100

Status: Solved ✅

Challenge Description

The challenge exposes a “Feedback Portal” where users can submit a name and a free form message. The description hints that all feedback is blindly trusted and that “the flag is in the root” of the server.

The objective is to identify how user-supplied input is processed, abuse any unsafe parsing logic, and extract the flag file from the server without any authentication. The final exploit turns the feedback feature into an XML External Entity (XXE) vector to read `/flag.txt`

2. Reconnaissance & Surface Mapping

1. Accessed the challenge URL and observed a simple feedback form with two fields: `Name` and `Message`, and a “Thank you for your feedback!” page reflecting back submitted data.
2. Initial tests with template-style payloads (for example, using `{{7*7}}` in the message) showed that input was reflected as plain text, suggesting no template-engine injection but confirming full server-side processing and reflection.
3. Used Burp Suite as an intercepting proxy to capture the POST request; noticed:
    - Endpoint: `POST /feedback`

- Response stack: `Server: Werkzeug/x.x Python/x.x`, indicating a Python/Flask-style backend.

4. Experimented by changing the `Content-Type` header to `application/xml` and sending a simple XML body instead of traditional form-encoded data; the server accepted XML and still rendered back `Name` and `Message` fields, confirming XML parsing on the backend.

Security observation: Accepting untrusted XML from the client without hardening the XML parser (especially on Python stacks) is a strong indicator that XXE may be possible.

3. Vulnerability Identification – XML External Entity (XXE)

3.1 Hypothesis

Since the server:

- accepts XML with `Content-Type: application/xml`, and
- reflects the `<name>` and `<message>` values in the HTML response,

there is a high probability that the backend XML parser allows external entities and expands them before rendering. This would allow reading local files via a malicious `<!DOCTYPE>` definition.

3.2 Proof-of-Concept Payload

In Burp Repeater, the request body was replaced with the following XML payload (pretty-printed in the report; in Burp, this is sent as the raw body):

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

Key points:

- `<!DOCTYPE foo [...]>` defines an external entity `xxe` pointing to `file:///flag.txt` on the server filesystem.
- The `<name>` field uses `&xxe;`, so when the XML parser expands entities, the content of `/flag.txt` is substituted where the name should be.

3.3 Results

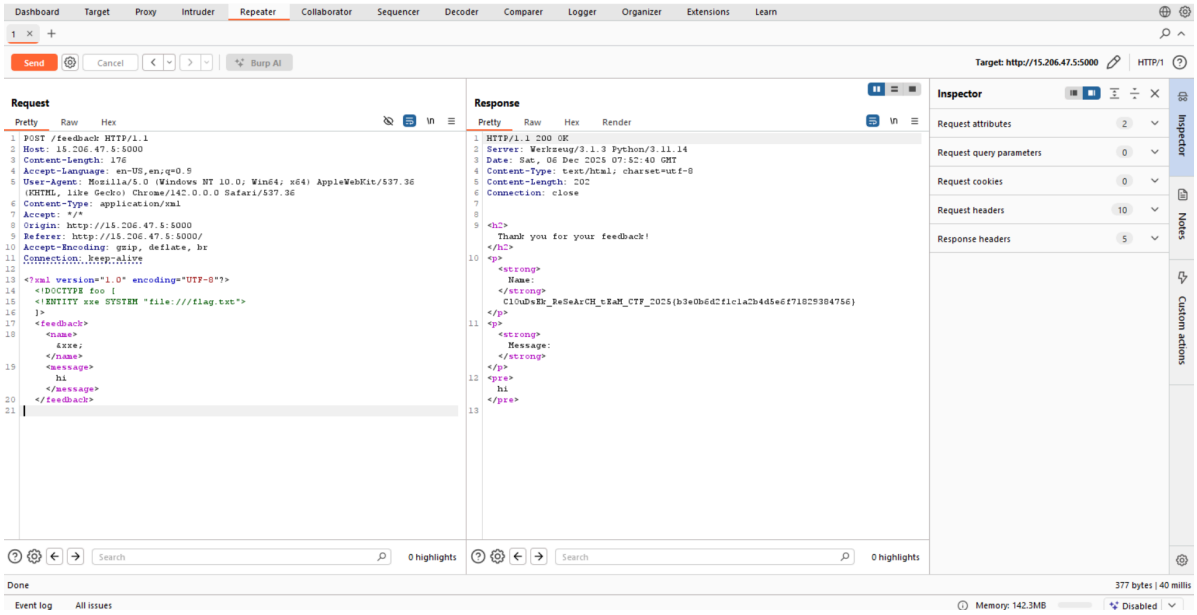
- Request was sent to `POST /feedback` with header `Content-Type: application/xml`.
- The HTTP response returned a normal “Thank you for your feedback!” page.
- In the `Name:` section of the response, instead of the literal string `&xxe;`, the content of the `flag.txt` file appeared in clear text (for example, `CLOUDSEK{...}`), confirming a working XXE file-read.

This conclusively demonstrates:

- The XML parser accepts external entities.
- Local files under the server’s root are readable via user-controlled XML.

Screenshots to include in PDF:

- Feedback form UI.
- Burp request showing XML payload and headers.
- Burp response panel clearly displaying the flag in the `Name:` field.



4. Impact Assessment

- **Confidentiality:** Critical – an attacker can read arbitrary files accessible to the application user, including configuration files, source code, credentials, and flags/secrets such as `/flag.txt`, `/etc/passwd`, or environment variables.
- **Integrity:** Indirect impact – exposure of application secrets can enable further compromise (e.g., database credentials, API keys) leading to data tampering.
- **Availability:** Typically unchanged by this exploit, but XXE can be extended to SSRF or DoS (e.g., huge external entities) if network and parser configuration allow.

From a hiring / real-world perspective, this demonstrates a high-risk misconfiguration of XML parsers in web services.

5. Step-by-Step Exploitation Guide

1. Configure Burp Suite as proxy and browse to the Feedback Portal.
2. Submit any dummy values in `Name` and `Message` to capture the original `POST /feedback` request.

3. Send the request to **Repeater**.
4. Modify the request:
    - Change header: `Content-Type: application/xml`.
    - Replace body with the crafted XXE XML payload shown in section 3.2.
5. Send the modified request.
6. Inspect the HTTP response body; the flag is visible inside the `Name:` field, confirming successful XXE exploitation.

These steps are easily reproducible and require no brute force, only understanding of XML parsers and XXE.

---

6. Remediation Recommendations

To fix this vulnerability in production systems:

- **Disable external entities** in the XML parser (e.g., for Python libraries, explicitly disable `resolve_entities` or use a hardened parser such as `defusedxml`).
- **Avoid XML for simple forms** where JSON or form-encoded input is sufficient; this reduces attack surface.
- **Implement strict content-type handling:** reject unexpected `Content-Type: application/xml` for endpoints designed only for standard HTML forms.
- **Least-privilege filesystem access:** run the web application under a dedicated low-privilege user with limited filesystem visibility

3.CTF Write-Up: Triangle (PHP Type Juggling Bypass)

Category: Web / Authentication Bypass

Points: 100

Status: Solved ✅

Target URL: `http://15.206.47.5:8080/login.php`

Challenge Description

The “Triangle” application presents a polished login page with a username, password, and three separate OTP fields, marketed as “layered authentication.” In reality, all three OTP checks rely on a subtle PHP type-juggling mistake. The objective is to reverse-engineer the login logic and bypass every OTP step without ever computing the real codes, by abusing how PHP compares user input against the generated OTP values

Recon & Source Analysis

1. Viewing the page source showed a developer comment mentioning backup files (`.bak`) and a PHP-based 2FA implementation (`google2fa.php`).
2. Direct access to `login.php.bak` revealed the server-side login logic:
   - Username is hard-coded as `admin`.
   - Password is `admin`, stored as `password_hash("admin", ...)` and verified with `password_verify`.
   - Three OTPs (`otp1`, `otp2`, `otp3`) are validated using `Google2FA::verify_key(...)`.
3. In the `google2fa.php.bak` file, the verification loop was implemented as:

   ```
   phpif (self::oath_hotp($binarySeed, $ts) == $key) {
     return true;
   }
   ```

   The critical detail is the **loose comparison** operator `==` between the generated OTP (string) and the user-supplied value.

Vulnerability – PHP Type Juggling

In PHP, loose comparison triggers type juggling: when a non-numeric string is compared to a boolean, both sides are coerced, and expressions like `"123456" == true` evaluate to `true`.

Because the application accepts JSON, it is possible to send the OTP parameters as **native booleans** instead of strings. When the backend compares the generated OTP string with the JSON boolean `true` using `==`, the expression becomes true, and the OTP check passes without knowing the actual 6-digit code.

This behavior affects all three OTP checks, so bypassing the entire “triangle” of verification is as simple as sending boolean `true` for each OTP field.

Exploitation Steps

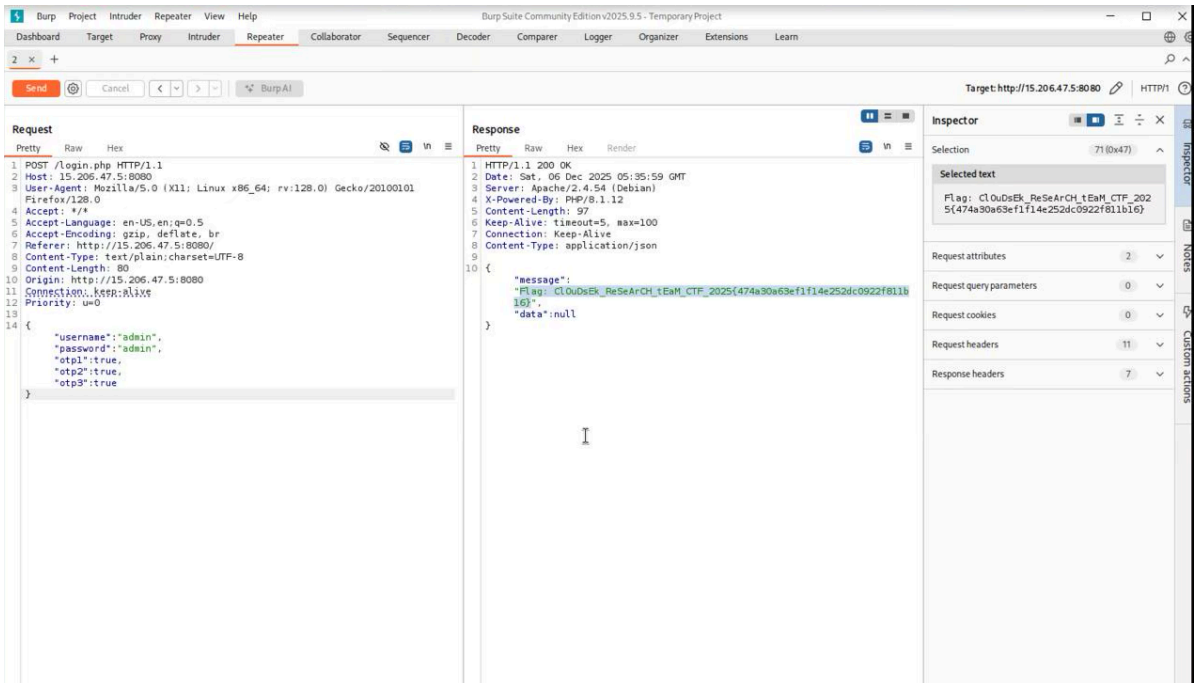
1. **Intercept the login request** with Burp Suite after submitting any dummy credentials on the web form.
2. Send the `POST /login.php` request to **Repeater**. Confirm that the body is JSON with fields: `username`, `password`, `otp1`, `otp2`, `otp3`.
3. Modify the JSON payload to:

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

   Make sure `otp1`, `otp2`, `otp3` are unquoted booleans, not strings.
4. Send the modified request. The server now evaluates each check as:

   ```
   phpself::oath_hotp($binarySeed, $ts) == true
   ```

   Due to type juggling, the comparison returns `true` for a wide range of values, and all three OTP validations succeed.
5. The JSON response returns a success message containing the flag. A Burp screenshot capturing this request–response pair serves as the proof of exploitation.



Impact & Security Lessons

- **Impact:** Any attacker who understands the login logic can authenticate as the admin without knowing any OTP secrets, completely defeating the multi-factor design.
- **Root cause:** Use of loose comparison (`==`) for security-critical checks, combined with JSON inputs that can carry native booleans.
- **Remediation:**
  - Use strict comparison (`===`) for all authentication and authorization checks in PHP.
  - Validate input types on the server and reject unexpected types (e.g., booleans where a numeric OTP string is expected).

This challenge nicely demonstrates how subtle PHP type juggling bugs can completely break otherwise well-designed authentication flows.

4.CTF Write-Up: Ticket The Strike Bank Heist

Category: Web + Mobile Exploitation

Points: 100

Status: Solved ✅

Challenge Description

Strike Bank recently detected unusual activity in their customer portal.

During a routine review of their **Android application**, several suspicious clues were uncovered.

Your mission was to:

- Investigate the APK
- Analyze the BeVigil report
- Explore the associated web portal
- Chain findings from mobile to web
- And finally uncover the hidden flag

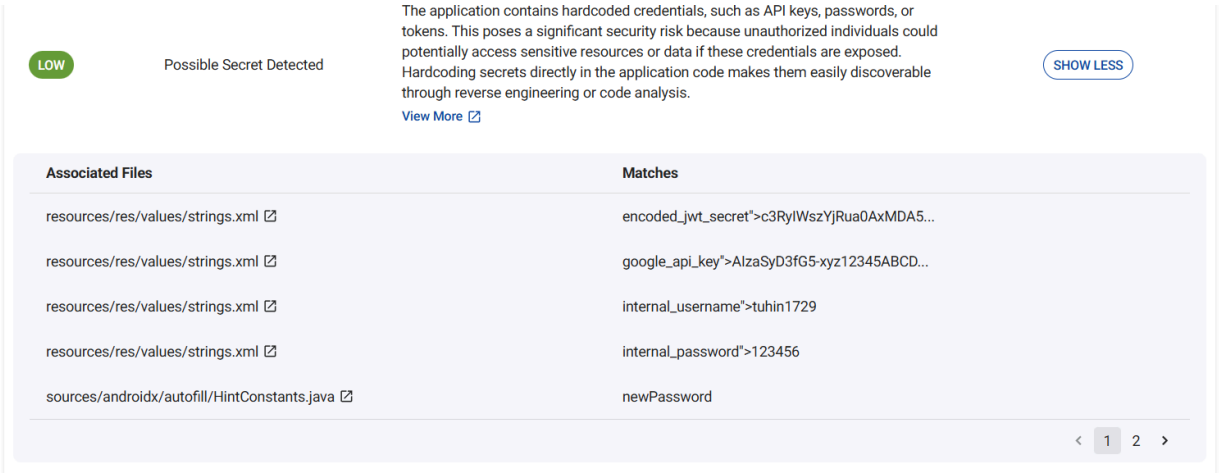
This challenge was a perfect test of **mobile OSINT**, **hardcoded secret extraction**, and a **JWT forgery attack** to gain **admin access** on the Strike Bank portal.

Recon via BeVigil

I began the analysis by scanning the APK:

`com.strikebank.netbanking` on **BeVigil**.

The scan immediately revealed **hardcoded sensitive information** in `resources/strings.xml`.



Why this is important?

Mobile apps often contain embedded secrets.

Anyone can download the APK → extract secrets → misuse them.

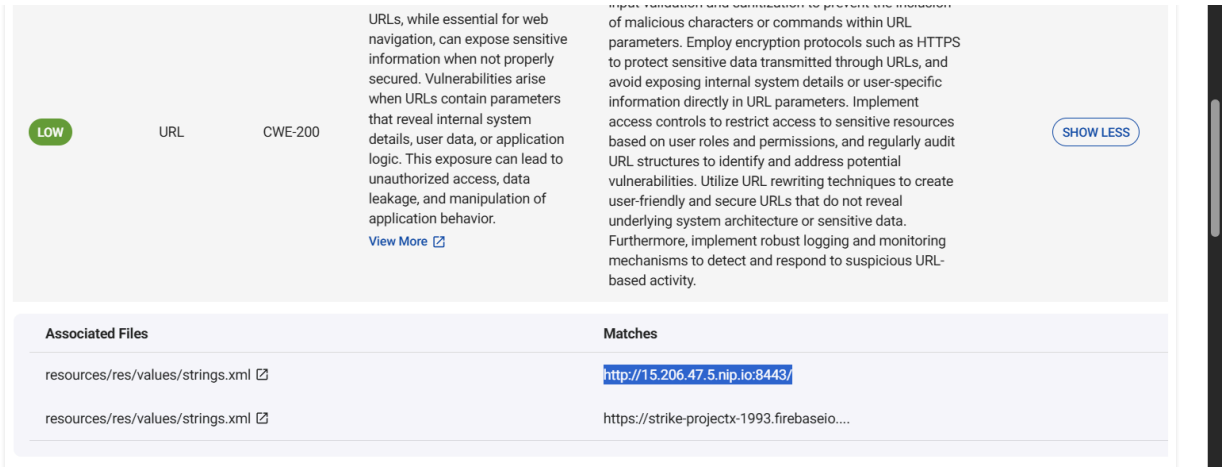
Here, the app exposed:

- Base64-encoded JWT signing key
- Internal employee username
- Internal employee password

This already compromises the mobile security and gives us an entrypoint.

Extracting the Internal Web Portal URL

The APK also contained an **internal web portal link**, which is not supposed to be public:



http://15.206.47.5.nip.io:8443

Why is this dangerous?

A hidden internal portal becomes accessible to anyone who parses the APK.

This expands the attack surface and directly enables further exploitation.

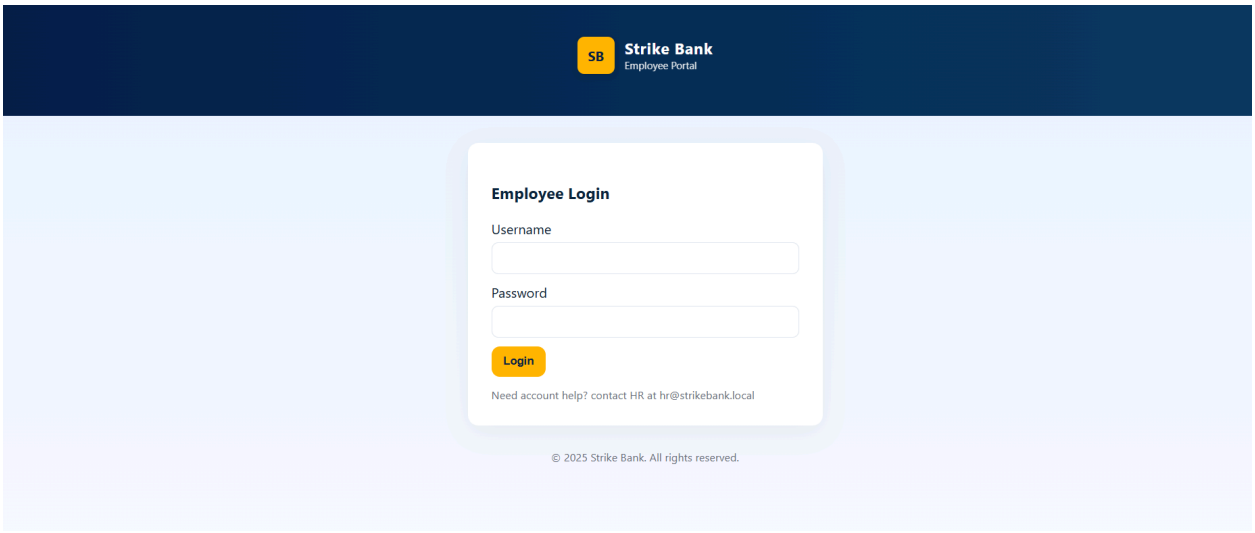
Logging In Using Leaked Credentials

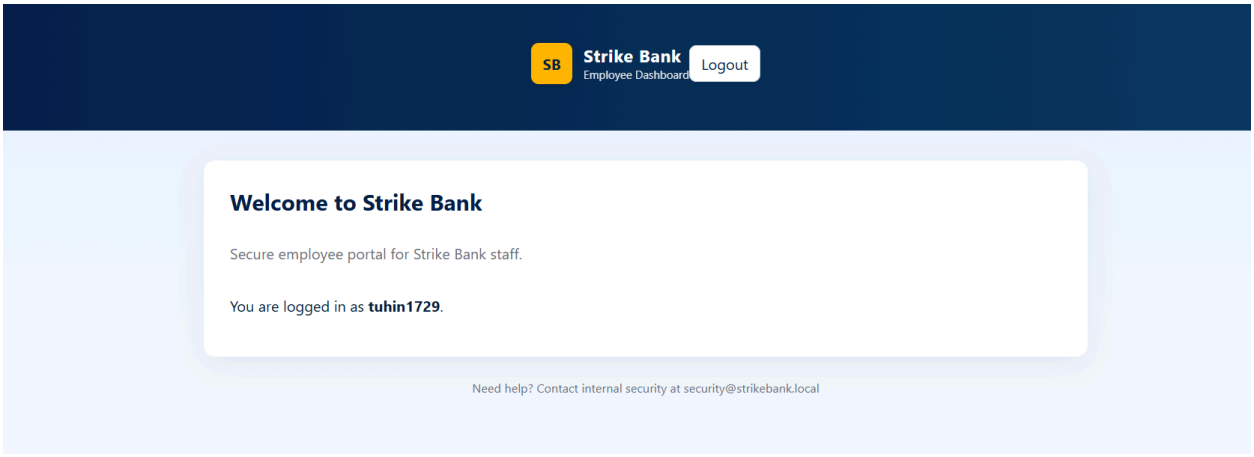
When I visited the exposed URL, it displayed an employee login page.

Using the credentials from the APK:

- Username → **tuhin1729**
- Password → **123456**

I successfully logged in.



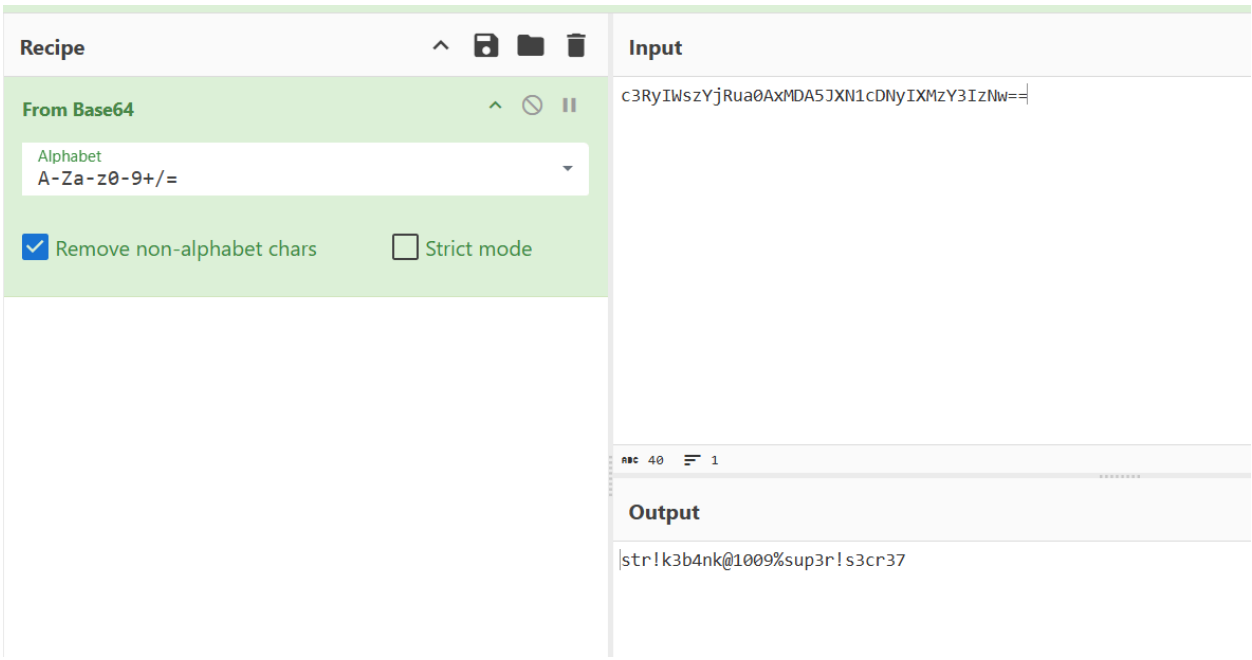


What this shows?

Hardcoded credentials inside mobile apps = **instant authentication bypass**.

Extracting & Decoding the JWT Signing Secret

Inside the APK, an encoded string was found:




```
c3RyIWszYjRua0AxMDA5JXN1cDNyIXMzY3IzNw==
```

After decoding via CyberChef, the actual secret key was revealed:

```
str!k3b4nk@1009%sup3r!s3cr37
```

Why this is critical?

This is the **JWT signing key** used by the web server.

If you have the signing key → you can generate **valid tokens for ANY USER**, even admin.

This completely breaks authentication.

Harvesting the Original JWT (Browser DevTools)

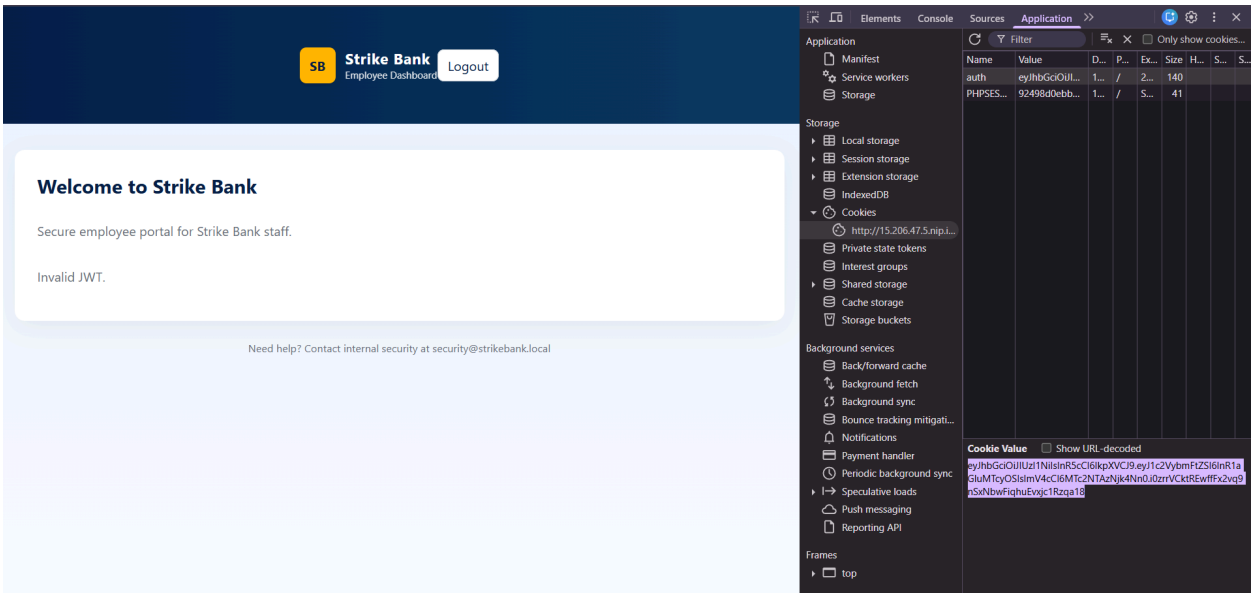
After login, I inspected the browser cookies.

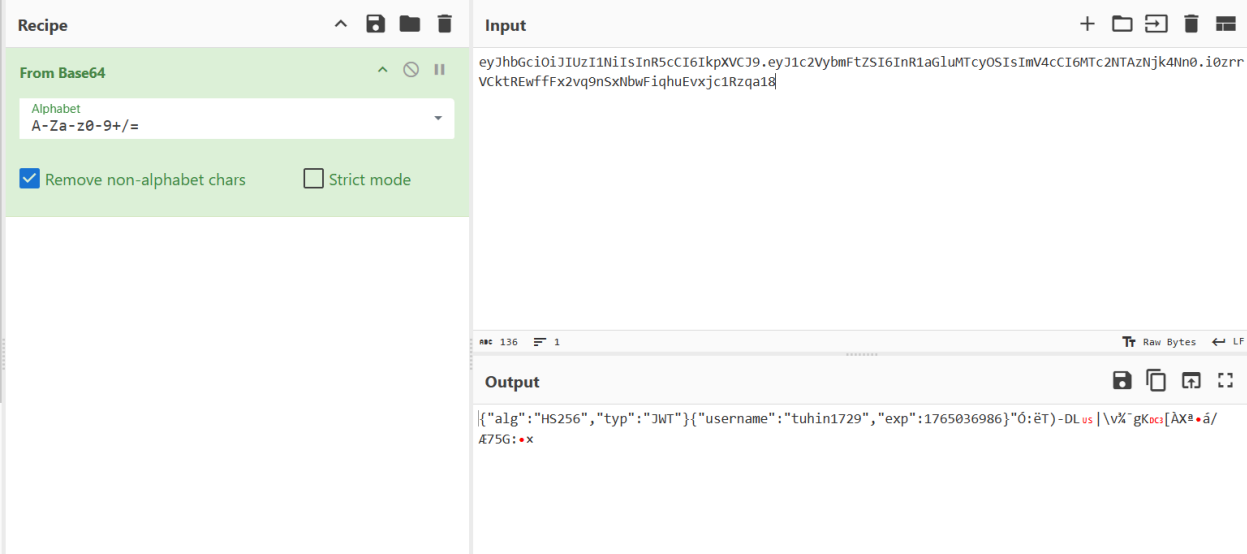
The app stored:

- `auth` → JWT Token
- `PHPSESSID`

I decoded the JWT payload to understand the structure.

It contained:





"username": "tuhin1729"
"exp": <expired timestamp>

Purpose?

This gives us the exact format of the token so we can replicate it while forging an admin token.

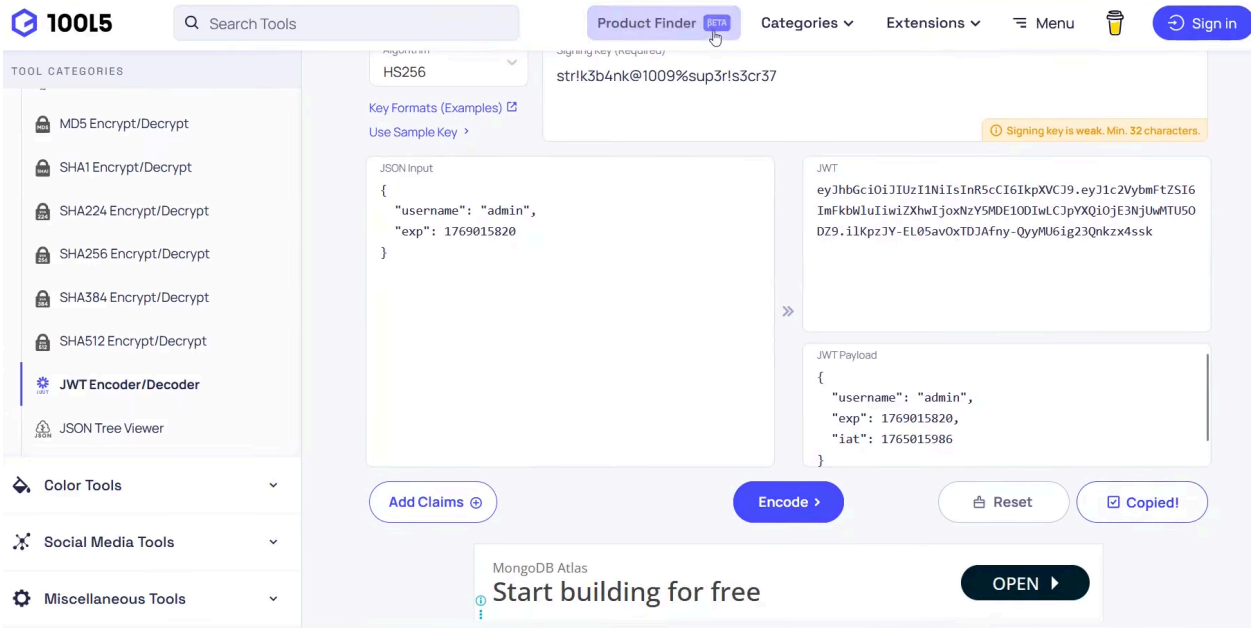
Forging a Valid Admin JWT

Using the leaked secret + JWT tool:

Algorithm → **HS256**

Key → **str!k3b4nk@1009%sup3r!s3cr37**

I crafted a new payload:




```
{
  "username": "admin",
  "exp": 1769015820
}
```

Then signed it to generate a valid **admin-level JWT**.

Why does this work?

JWT verification succeeds because:

✔️ Correct algorithm

✔️ Correct signing key

✔️ Correct structure

Server now believes I am **admin**.

Cookie Replacement & Privilege Escalation

I replaced the existing `auth` cookie with the forged admin JWT.

Refresh the page →

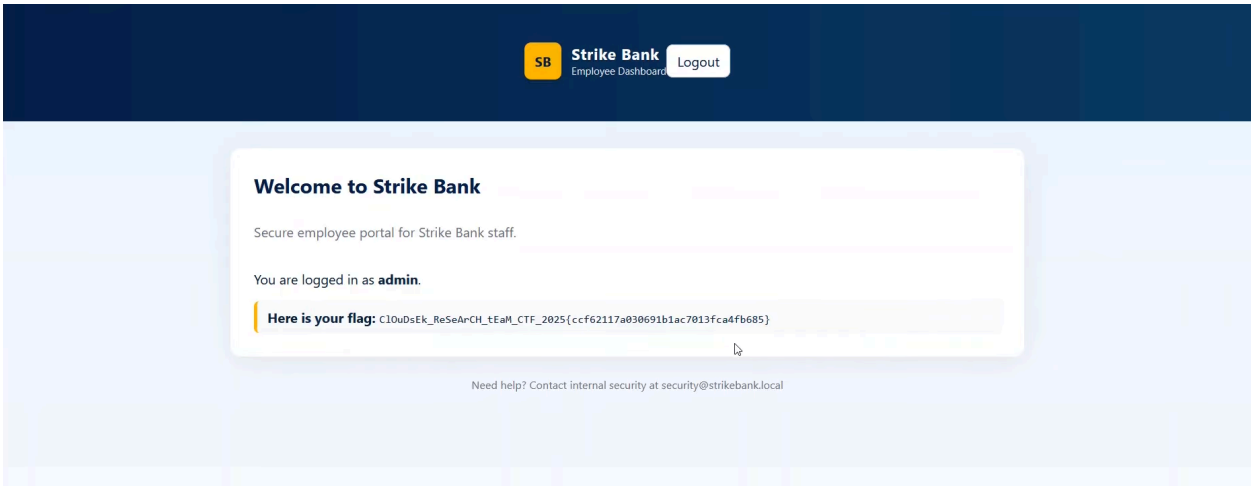
The application accepted the forged token without any verification or rotation.

Meaning?

The entire authentication system was compromised.

Final Flag Extraction

Upon successful privilege escalation, the admin dashboard displayed the final flag:



ClOuDsEk_ReSeArCH_tEaM_CTF_2025{ccf62117a030691b1ac7013fca4fb685}