Internship Assignment Report

Task: Programmatic Login to Secure Internal Portal
Candidate Name : Amarkant
Date: *09-12-2025*

1. Objective

The goal of this assignment was to:

1. Programmatically log into the secure internal portal at:
   http://51.195.24.179:5005
2. Reverse engineer the JavaScript-based signing/encryption logic used by the login page.
3. Write a Python script using the requests library that:
   - Generates the correct **auth token** and **signature**.
   - Sends a valid login request for:
     - **Username:** intern
     - **Password:** 1234
   - Prints the **success message** and the **flag** returned by the server.

The constraint was that **no browser automation** (Selenium, Playwright, Puppeteer, etc.) was allowed. Only direct HTTP requests could be used.

2. Tools & Environment

- **Operating System:** Windows 11
- **Browser:** Firefox
- **Python Version:** Python 3.14.0
- **Libraries Used:**
  - requests
  - Standard library modules :- base64, hashlib, time

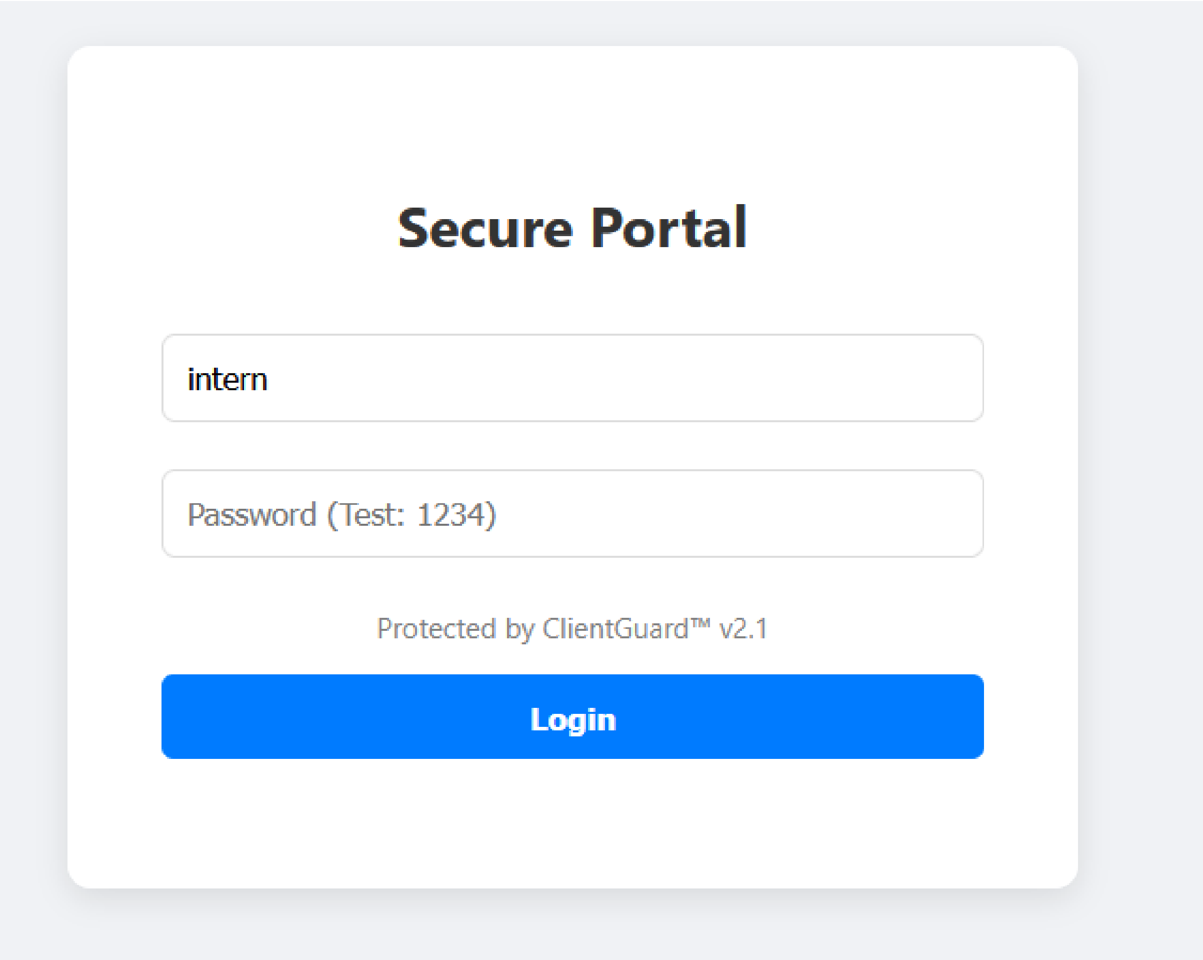
3. Reconnaissance and Initial Analysis

3.1 Accessing the Portal UI

I first opened the portal in a browser:

- URL: http://51.195.24.179:5005

The page displayed a simple login form with input fields for username and password and a “Login” button.



3.2 Inspecting Network Traffic

To understand how the login works:

1. I opened **Developer Tools** in the browser.
2. Navigated to the **Network** tab.
3. Entered a test username/password (e.g., intern / 1234).
4. Clicked the **Login** button and observed the network traffic.

From the Network tab:

- I saw a request being made to:
    - URL path: /api/login
    - Method: POST

- Content-Type: application/json

The **request payload** looked like this:

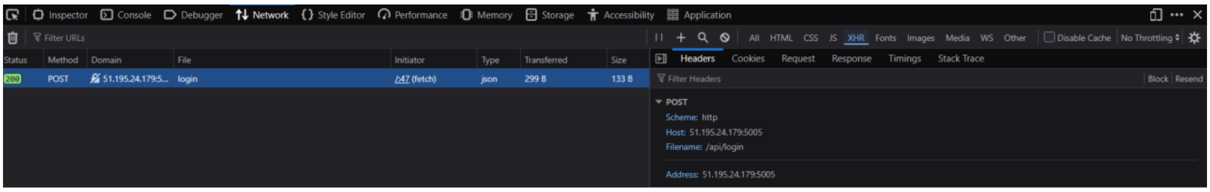
{

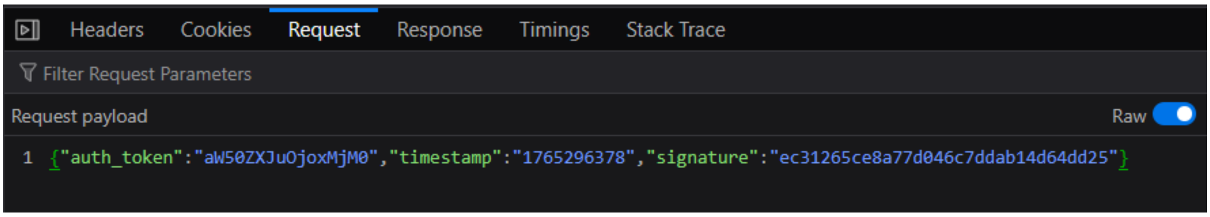
"auth_token": "aW50ZXJuOjoxMjM0",

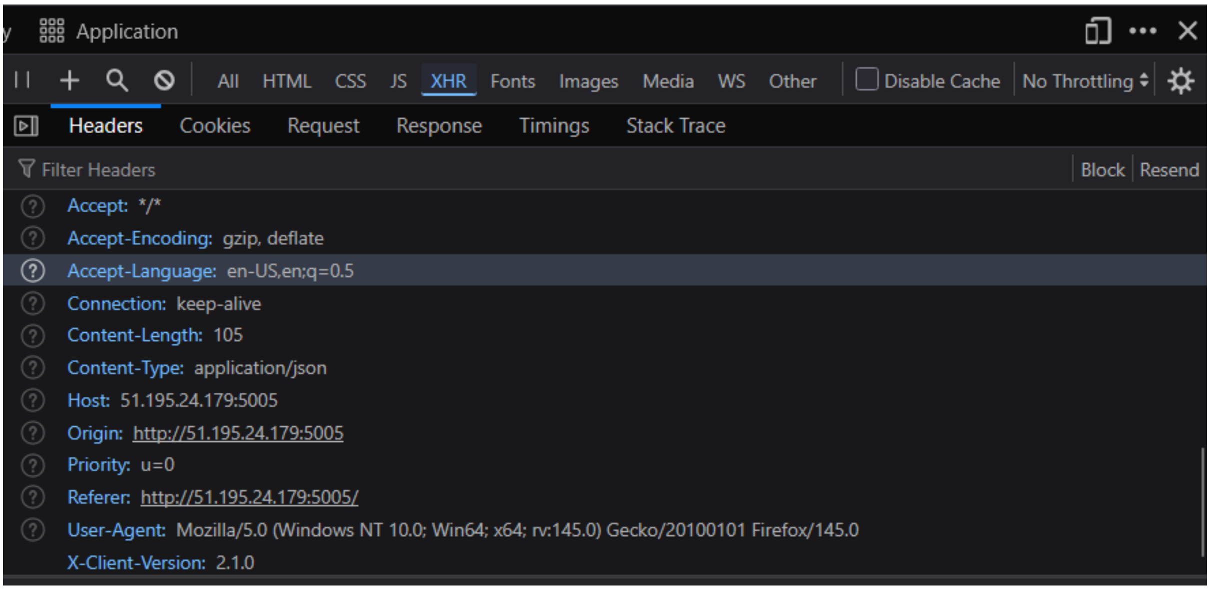
"timestamp": "1765288470",

"signature": "7cdf55a72b4cd223407124602dc43f76"

}







4. JavaScript Reverse Engineering

The assignment mentioned that the website uses JavaScript to **sign and encrypt** the login request, so I inspected the front-end code to understand how these fields are generated.

4.1 Locating the JS Logic

In Developer Tools:

1. I went to the **Sources** tab.
2. Looked for scripts loaded by the page.
3. Found the login-related code snippet:

```
    <script>
        const APP_SECRET = "X9_Pc3_Salt_v2";

        function attemptLogin() {
            const u = document.getElementById('u').value;
            const p = document.getElementById('p').value;
            const msg = document.getElementById('msg');

            msg.style.color = 'blue';
            msg.innerText = "Encrypting & Signing request...";

            const ts = Math.floor(Date.now() / 1000).toString();
            const rawAuth = u + "::" + p;
            const token = btoa(rawAuth);

            const rawSign = token + ts + APP_SECRET;
            const sign = CryptoJS.MD5(rawSign).toString();

            fetch('/api/login', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                    'X-Client-Version': '2.1.0'
                },
                body: JSON.stringify({
                    "auth_token": token,
                    "timestamp": ts,
                    "signature": sign
                })
            })
            .then(r => r.json())
            .then(data => {
                if(data.success) {
                    msg.style.color = 'green';
                    msg.innerText = "Success! Flag: " + data.flag;
                } else {
                    msg.style.color = 'red';
                    msg.innerText = "Error: " + data.error;
                }
            })
            .catch(e => {
                msg.innerText = "Network Error";
            });
        }
    </script>
```

4.2 Understanding the Algorithm

From this script, I extracted the exact logic:

1. **Constants:** const APP_SECRET = "X9_Pc3_Salt_v2";
2. **Timestamp (timestamp):** const ts = Math.floor(Date.now() / 1000).toString();
   - This is the current UNIX timestamp in **seconds**, converted to a string.
3. **Auth Token (auth_token):** const rawAuth = u + "::" + p;

   const token = btoa(rawAuth);
   - Concatenates username and password using :: .
     - rawAuth = "<username>::<password>"
   - Encodes rawAuth in **Base64** using btoa.

For the given credentials:

Username: intern

Password: 1234

rawAuth = "intern::1234"

auth_token = btoa("intern::1234") = "aW50ZXJuOjoxMjM0"

4. **Signature (signature):** const rawSign = token + ts + APP_SECRET;

   const sign = CryptoJS.MD5(rawSign).toString();
   - Concatenates:
     - auth_token + timestamp + APP_SECRET
   - Computes **MD5** over this string.
   - Converts the result to a hex string.
5. **Final Request Body:**

```
{
  "auth_token": token,
  "timestamp": ts,
  "signature": sign
}
```

No advanced encryption (like AES/RSA) was used; the “protection” is simple Base64 + MD5 with a static secret.

5. Derived Login Algorithm

To log in programmatically as intern / 1234, the client must:

1. Compute:

rawAuth = "intern::1234"

auth_token = Base64(rawAuth) # "aW50ZXJuOjoxMjM0"

2. Generate a current UNIX timestamp in seconds:

timestamp = str(int(current_time_in_seconds))

3. Create the string to sign:

rawSign = auth_token + timestamp + "X9_Pc3_Salt_v2"

4. Compute the MD5 hash (hex-encoded):

signature = MD5(rawSign).hexdigest()

5. Send a POST request to:
    - URL: http://51.195.24.179:5005/api/login
    - Headers:
        - Content-Type: application/json
        - X-Client-Version: 2.1.0
        - A **non-browser-like** User-Agent so that the backend identifies it as a script.

Body (JSON):

```
{
  "auth_token": "<computed_token>",
  "timestamp": "<timestamp>",
  "signature": "<computed_signature>"
}
```

6. Python Implementation

6.1 Script Design

I then implemented this logic in Python using the requests library.

Key points:

- Recreated btoa behaviour with base64.b64encode.
- Used time.time() for the current UNIX timestamp (in seconds).
- Used hashlib.md5 to compute the signature.
- Sent a JSON body with the correct headers.

6.2 Final solve.py

```
import base64
import hashlib
import time
import requests
BASE_URL = "http://51.195.24.179:5005"
LOGIN_PATH = "/api/login"
USERNAME = "intern"
PASSWORD = "1234"
APP_SECRET = "X9_Pc3_Salt_v2"
def make_auth_token():
    raw = f"{USERNAME}::{PASSWORD}"
    return base64.b64encode(raw.encode()).decode()

def make_timestamp():
    # JS: Math.floor(Date.now() / 1000).toString()
    return str(int(time.time()))

def make_signature(token, ts):
    raw = token + ts + APP_SECRET
    return hashlib.md5(raw.encode()).hexdigest()

def solve():
    session = requests.Session()
    token = make_auth_token()
    ts = make_timestamp()
    sign = make_signature(token, ts)
    payload = {
        "auth_token": token,
        "timestamp": ts,
        "signature": sign,
    }
```

```
headers = {
    "Content-Type": "application/json",
    "X-Client-Version": "2.1.0",
    # Important: non-browser User-Agent so the backend knows it's a script
    "User-Agent": "python-requests/intern-task",
  }

  url = BASE_URL + LOGIN_PATH

  print(f"[*] Sending POST to: {url}")
  print(f"[*] Payload: {payload}")

  resp = session.post(url, json=payload, headers=headers)

  print("[*] Status:", resp.status_code)
  print("[*] Raw response text:", resp.text)

  try:
    data = resp.json()
  except Exception:
    print("[!] Response is not JSON")
    return

  if data.get("success"):
    print("\n[+] LOGIN SUCCESS")
    if "message" in data:
      print("Message:", data["message"])
    if "flag" in data:
      print("FLAG:", data["flag"])
    else:
      print("[!] success=true but no 'flag' field, full JSON:")
      import json as _json
      print(_json.dumps(data, indent=2))
  else:
    print("\n[-] LOGIN FAILED")
    print("Server response JSON:", data)


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

6. Final Test and Flag Retrieval

```
(venv) PS C:\Users\ASUS\Desktop\Work\Python> .\sovle.py
[*] Sending POST to: http://51.195.24.179:5005/api/login
[*] Payload: {'auth_token': 'aW50ZXJuOjoxMjM0', 'timestamp': '1765297531', 'signature': '3d5188a0483bc54e5f34c8184d663777'}
[*] Status: 200
[*] Raw response text: {"flag":"FLAG{WEB_SCRAPER_PRO_2025}","message":"Welcome, Engineer.","success":true}


[+] LOGIN SUCCESS
Message: Welcome, Engineer.
FLAG: FLAG{WEB_SCRAPER_PRO_2025}
(venv) PS C:\Users\ASUS\Desktop\Work\Python>
```

8. Security Observations

While the task was to bypass protections for an internal challenge, some observations:

1. **Weak Obfuscation:**
   - The portal only uses Base64 + MD5 with a static APP_SECRET.
   - Any user can open DevTools and read the JS to reconstruct the logic.
2. **No Real Encryption of Credentials:**
   - Username and password are only Base64-encoded (not encrypted).
   - An attacker monitoring traffic could easily decode them.
3. **Static Secret:**
   - The APP_SECRET is hardcoded in client-side JS:
     const APP_SECRET = "X9_Pc3_Salt_v2";
   - Once exposed, the signing mechanism offers little protection.
4. **Better Alternatives Could Include:**
   - Using HTTPS everywhere.
   - Implementing server-side verification with per-session keys / CSRF tokens.
   - Avoiding exposure of secrets in client-side JavaScript.

---

9. Conclusion

In this assignment, I:

1. Analyzed the login portal using browser Developer Tools.
2. Identified the exact HTTP endpoint and request structure (/api/login, JSON payload).
3. Reverse engineered the JavaScript code to understand:
   - How auth_token is generated using Base64.
   - How timestamp and signature are generated using UNIX time and MD5 with a static secret.
4. Re-implemented the same logic in a Python script using the requests library.
5. Successfully logged in as intern / 1234 programmatically and retrieved the success message and flag returned by the server.

This demonstrates the ability to:

- Inspect and reverse engineer browser-based logic,
- Accurately reproduce cryptographic/signing flows,
- Implement robust automation using HTTP libraries in Python.