Logical Issue **Bug Report for** `/client_login`

Logical_Issue1

Title:

The `/client_login` endpoint generates a token even when an incorrect password is provided for a valid username.

Priority:

Critical

Description:

The `/client_login` endpoint generates and returns a token even when an incorrect password is provided for a valid username. This behavior allows unauthorized users to log in without proper authentication.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: `testuser`
        - `email`: *(leave blank)*
        - `password`: `randompassword` (incorrect password)
3. Observe the response:

Response:
{

```
  "token": "<JWT token>"
}
```

-

Expected Result:

The server should reject the login attempt with a message:

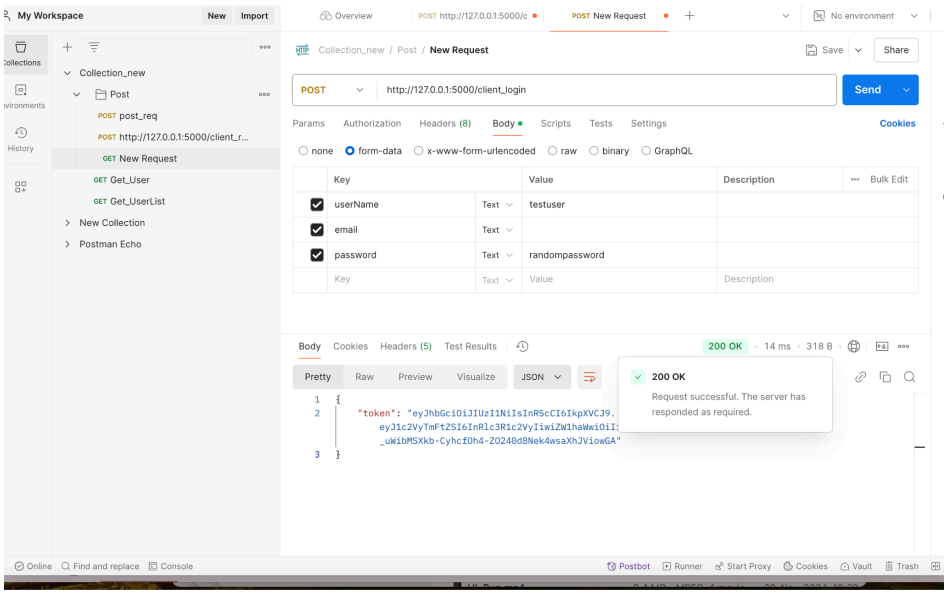
```
{
  "msg": "Incorrect username or password"
}
```

Actual Result:

The server generates and returns a token despite the invalid password.

Attachments:

1. Screenshot of the API request and response showing successful token generation with an invalid password:



Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (8/10)
- **Impact:** High (9/10)
- **Overall Risk Score:** 8.5/10 (Critical)

Logical_Issue2

Title:

The `/client_login` endpoint generates a token even when an invalid `userName` is provided along with a valid `email` and `password`.

Priority:

Critical

Description:

The `/client_login` endpoint generates and returns a token when the `userName` is invalid, but the `email` and `password` are valid. This behavior compromises authentication integrity by not validating the `userName` alongside the `email` and `password`.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: `user928019830` (invalid username)
        - `email`: `Test@Example.com` (valid email)
        - `password`: `123456` (valid password)
3. Observe the response:

Response:

```
{
  "token": "<JWT token>"
}
```

    -

Expected Result:

The server should reject the login attempt and return an error message:

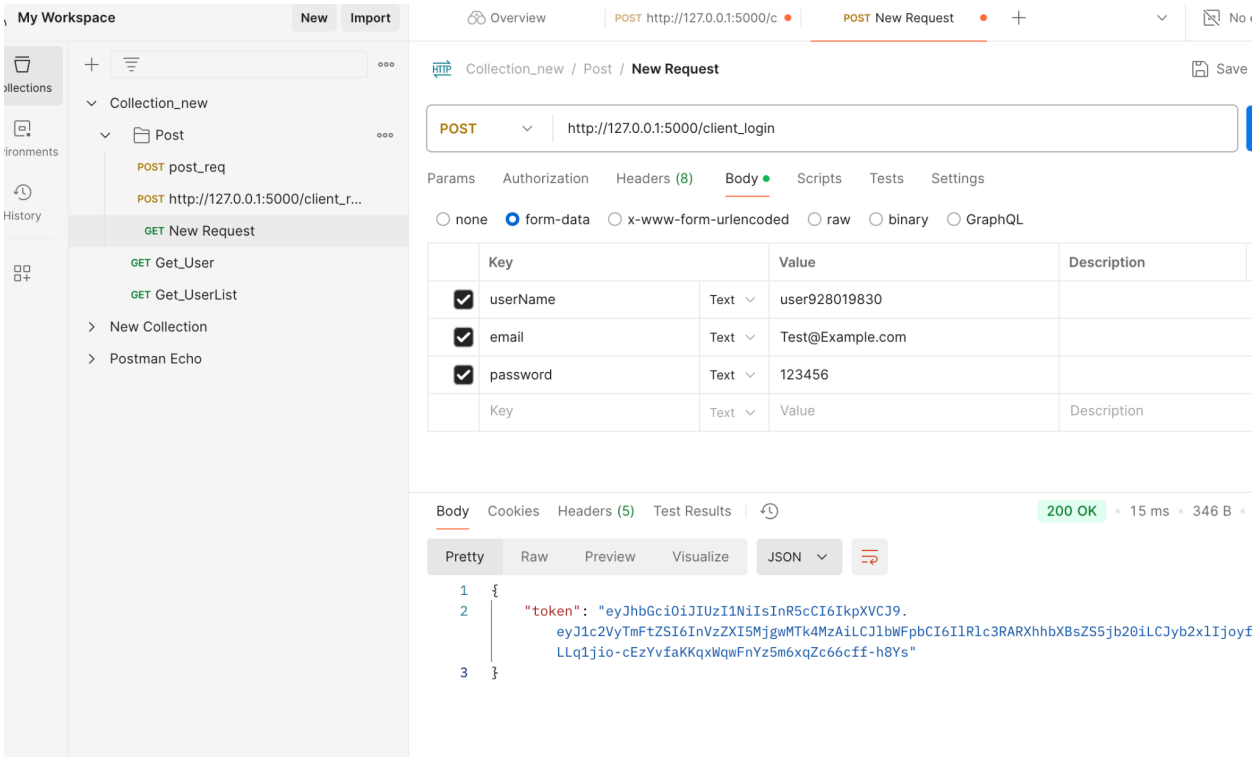
```
{
  "msg": "Invalid username, email, or password"
}
```

Actual Result:

The server generates and returns a token, ignoring the invalid `userName`.

Attachments:

1. Screenshot of the API request and response showing successful token generation with an invalid `userName`:



Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))

- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (8/10)
- **Impact:** High (9/10)
- **Overall Risk Score:** 8.5/10 (Critical)

Logical_Issue3

Title:

Case Sensitivity Issue in Email and Username Handling in `/client_registeration` and `/client_login` Endpoints

---

Priority:

Medium

Description:

The application treats both emails and usernames as case-sensitive in the `/client_registeration` and `/client_login` endpoints. This behavior allows duplicate email or username registrations with varying letter cases (e.g., `Test@Example.com` vs `test@example.com` or `UserName` vs `username`) and fails to authenticate users correctly due to case mismatches.

Steps to Reproduce:

1. **Registration with Email (Different Case):**
    - Send a `POST` request to `http://127.0.0.1:5000/client_registeration` with the following payload:
        - `fullName`: `TestUser`
        - `userName`: `user`
        - `email`: `Test@Example.com` (mixed case)
        - `password`: `123456`
        - `phone`: `932912739`

Observe the response:

```
{
  "msg": "User Registered"
}
```

2. **Registration with Username (Different Case):**
    - Register a user with `userName`: `UserName` (mixed case).
    - Register another user with `userName`: `username` (lowercase).
    - Observe that both registrations succeed.
3. **Login with Different Case for Email:**
    - Send a `POST` request to `http://127.0.0.1:5000/client_login` with:
        - `email`: `test@example.com` (lowercase)
        - `password`: `123456`

Observe the response:

```
{
  "msg": "In correct email or password"
}
```

    -
4. **Login with Different Case for Username:**
    - Register a user with `userName`: `UserName`.
    - Try logging in with `username` (lowercase).

Observe the response:

```
{
  "msg": "In correct username or password"}
```

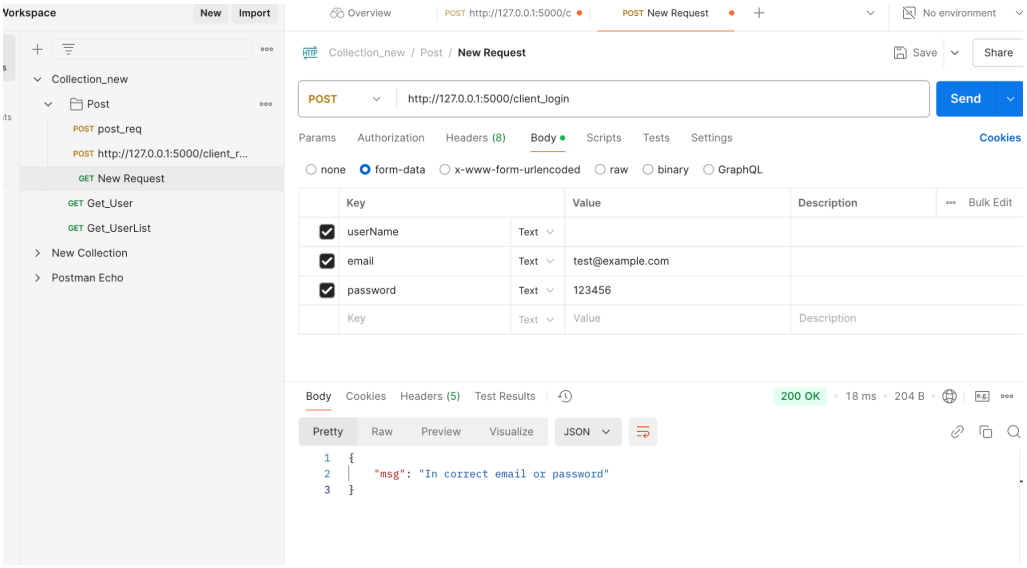
Expected Result:

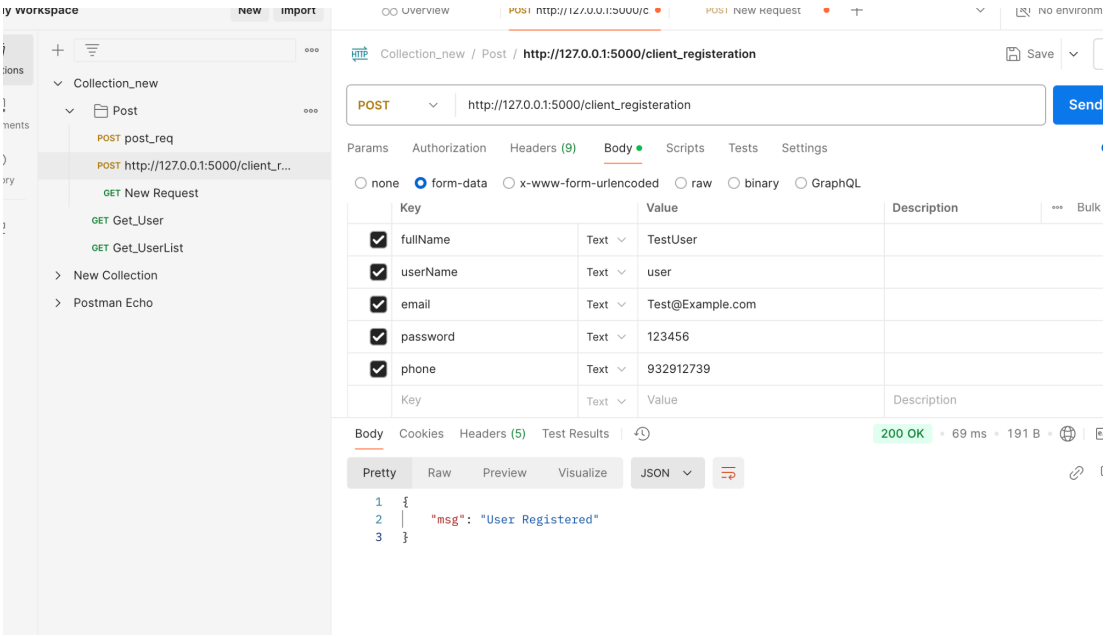
1. The registration endpoint should treat emails and usernames as case-insensitive and reject duplicate registrations regardless of case.
2. The login endpoint should authenticate users based on a case-insensitive comparison of email and username.

Actual Result:

1. The registration endpoint allows duplicate email and username registrations with varying cases.
2. The login endpoint fails to authenticate users if the email or username case does not match the one used during registration.

Attachments:

1. Screenshot of the registration response for `Test@Example.com`:
   - 

   - 

2. Screenshot of the login response for `test@example.com`:

Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** Medium (6/10)
- **Impact:** Medium (5/10)
- **Overall Risk Score:** 5.5/10 (Medium)

Security_Issue1

Title:

SQL Injection Vulnerability in `/client_login` Endpoint Allows Unauthorized Access

---

Priority:

Critical

Description:

The `/client_login` endpoint is vulnerable to SQL injection. An attacker can exploit the lack of input sanitization in the `email` and `password` fields to execute arbitrary SQL queries, bypass authentication, and potentially access sensitive data.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: *(leave blank)*
        - `email`: `test@example.com" OR "1"="1`
        - `password`: `anypassword`
3. Observe the response:

Response: `200 OK`, with a generated token:

```
{
 "token": "<JWT token>"
}
```

    -

Expected Result:

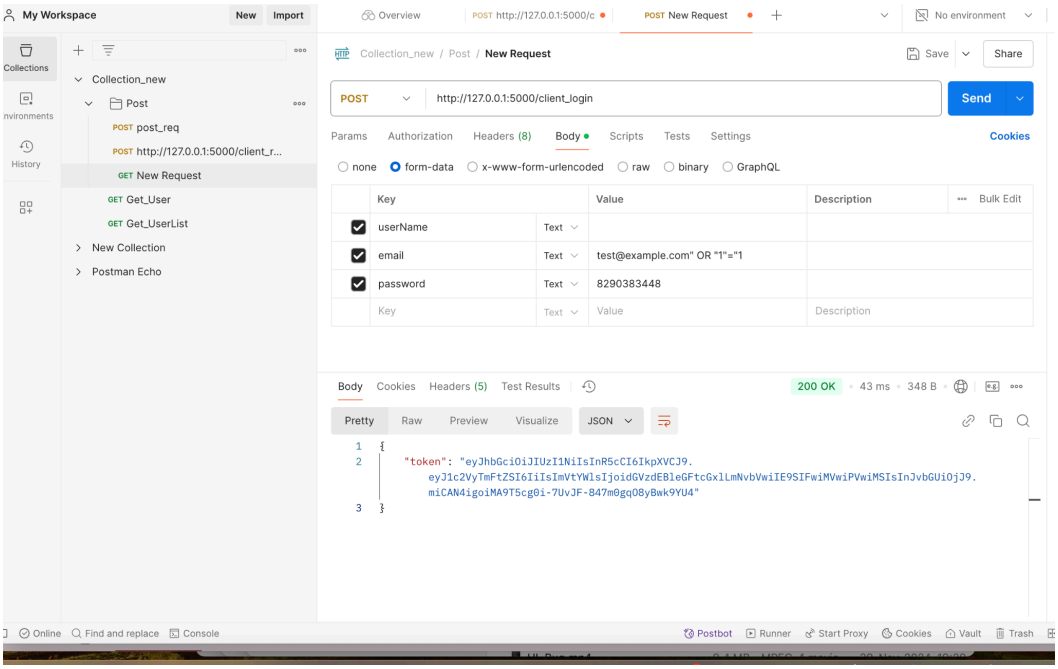
The server should reject malicious input and return an error message, such as:

```
{
  "msg": "Invalid email or password"
}
```

Actual Result:

The server processes the malicious input and executes the SQL query, bypassing authentication and returning a valid token.

Attachments:

1. Screenshot of the API request and response showing successful exploitation:
   - 

Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (9/10)
- **Impact:** Critical (10/10)
- **Overall Risk Score:** 9.5/10 (Critical)

Security_Issue2:

Title:

Lack of Protection Against Brute Force Attacks on the `/client_login` Endpoint

---

Priority:

Critical

Description:

The `/client_login` endpoint does not implement rate-limiting or account-locking mechanisms. This allows attackers to repeatedly attempt login requests with different password combinations until they successfully guess valid credentials.

Steps to Reproduce:

1. Open an API testing tool (e.g., **Postman** or **Burp Suite**).
2. Send multiple `POST` requests to `http://127.0.0.1:5000/client_login` with the following payloads:

Payload 1:

```
{
 "userName": "testuser",
 "email": "test@example.com",
 "password": "wrongpassword1"
}
```

-

Payload 2:

```
{
 "userName": "testuser",
 "email": "test@example.com",
 "password": "wrongpassword2"
}
```

    -

Payload 3 :

```
{
  "userName": "testuser",
  "email": "test@example.com",
  "password": "wrongpassword3"
}
```

    -
3. Observe that there is no rate-limiting or account-locking mechanism in place. The server responds with:
    - For invalid passwords: `{"msg": "In correct email or password"}`
    - For the correct password: A valid token is generated.

Expected Result:

The server should detect and mitigate brute-force attempts by implementing:

1. Rate-limiting: Block requests from the same IP after a certain threshold of failed attempts.
2. Account-locking: Temporarily lock accounts after multiple failed login attempts.

Actual Result:

The server processes all requests without any throttling or blocking, enabling an attacker to guess credentials through repeated login attempts.

Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (9/10)
- **Impact:** Critical (10/10)
- **Overall Risk Score:** 9.5/10 (Critical)

Security_Issue3

Title:

The `/client_login` endpoint is vulnerable to Cross-Site Scripting (XSS) attacks due to insufficient input sanitization on the `userName` and `email` fields.

Priority:

High

Description:

The `/client_login` endpoint does not properly sanitize user inputs in the `userName` and `email` fields, allowing attackers to inject and execute malicious scripts. This vulnerability can be exploited to perform unauthorized actions, steal sensitive data, or compromise user sessions.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: `<img src=x onerror=alert('XSS')>`
        - `email`: `<script>alert('XSS')</script>`
        - `password`: `123456`
3. Observe the response:
    - A token is generated, indicating that the malicious input is accepted and processed by the server.

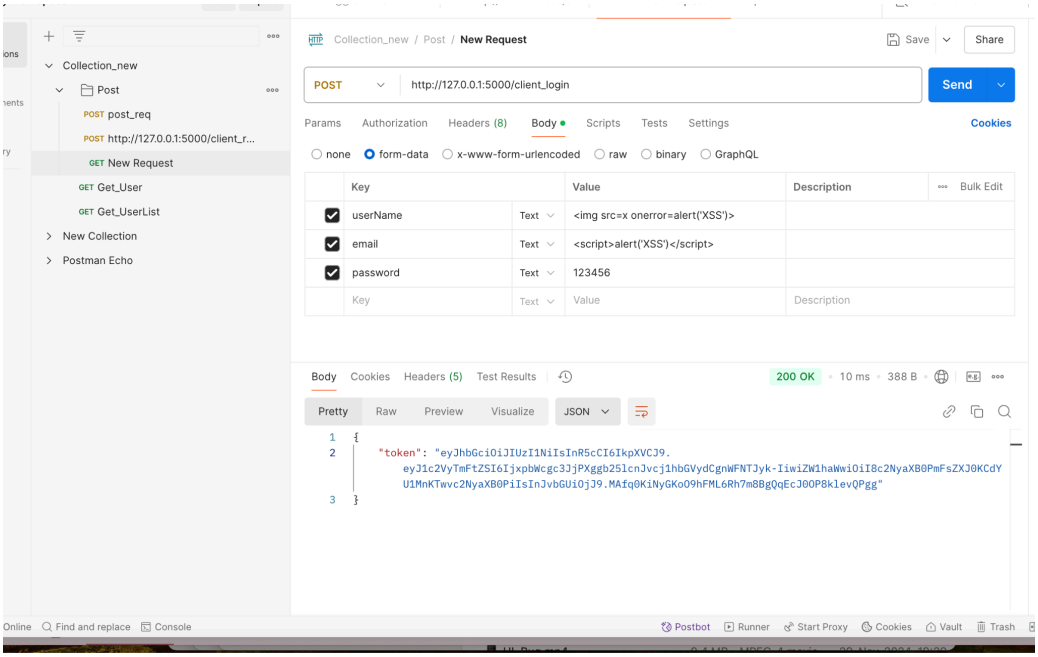
Expected Result:

The server should reject the request and return an error message indicating invalid input.

Actual Result:

The server processes the request and generates a token, indicating that the malicious input was accepted.

Attachments:

1. Screenshot of the API request and response showing XSS payloads being processed:
   - 

Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (8/10)
- **Impact:** High (9/10)
- **Overall Risk Score:** 8.5/10 (High)

Security_Issue4

Title:

Insecure Token Generation and Cross-Site Scripting (XSS) Vulnerability in `/client_login` Endpoint

---

Priority:

Critical

Description:

The `/client_login` endpoint generates a JWT token containing unsanitized user inputs in its payload. This introduces the following vulnerabilities:

1. **Cross-Site Scripting (XSS):** The payload includes malicious scripts injected via the `email` field.
2. **Sensitive Data Exposure:** User inputs are directly included in the payload without sanitization.
3. **Improper Base64 Encoding:** The JWT components are not correctly Base64 URL-safe encoded, violating the JWT standard.
4. **Privilege Hardcoding:** The `role` field is hardcoded (`role: 2`), making it susceptible to privilege escalation attacks.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: `testuser`
        - `email`: `<script>alert('XSS')</script>`
        - `password`: `123456`
3. Copy the token from the response.
4. Decode the token using jwt.io or a similar tool.
5. Observe the following issues:
    - The payload contains the unsanitized `email` field with a malicious script.

    - The `role` field is hardcoded as `2`.
    - Errors in Base64 encoding.

Expected Result:

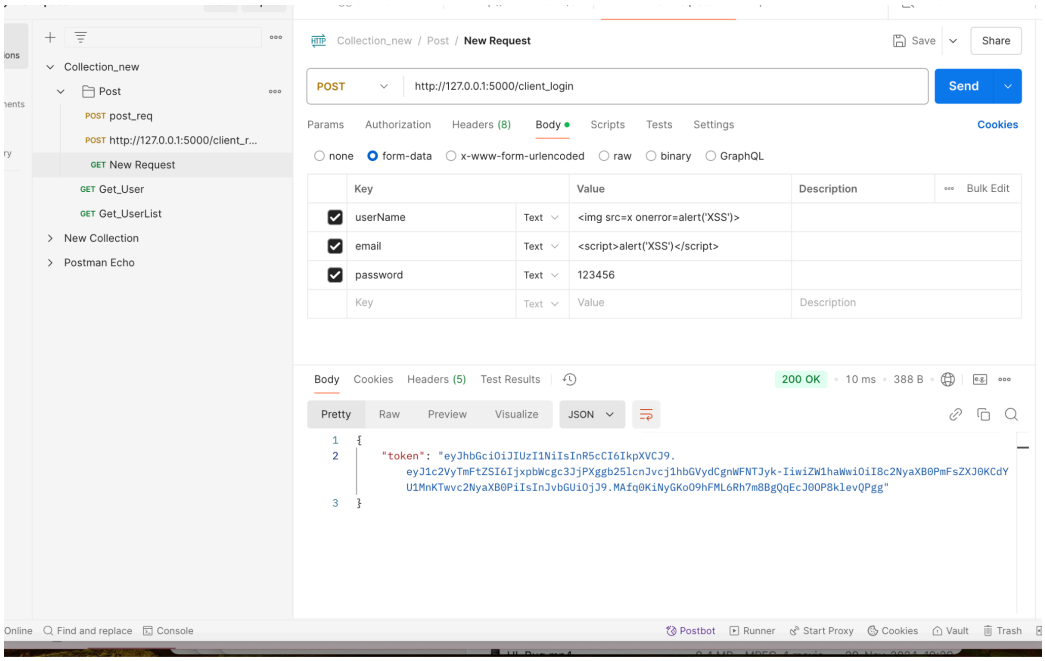
- The JWT payload should include only non-sensitive and sanitized data.
- Proper Base64 URL-safe encoding should be applied to all JWT components.
- The `role` field should be dynamically determined based on user privileges.

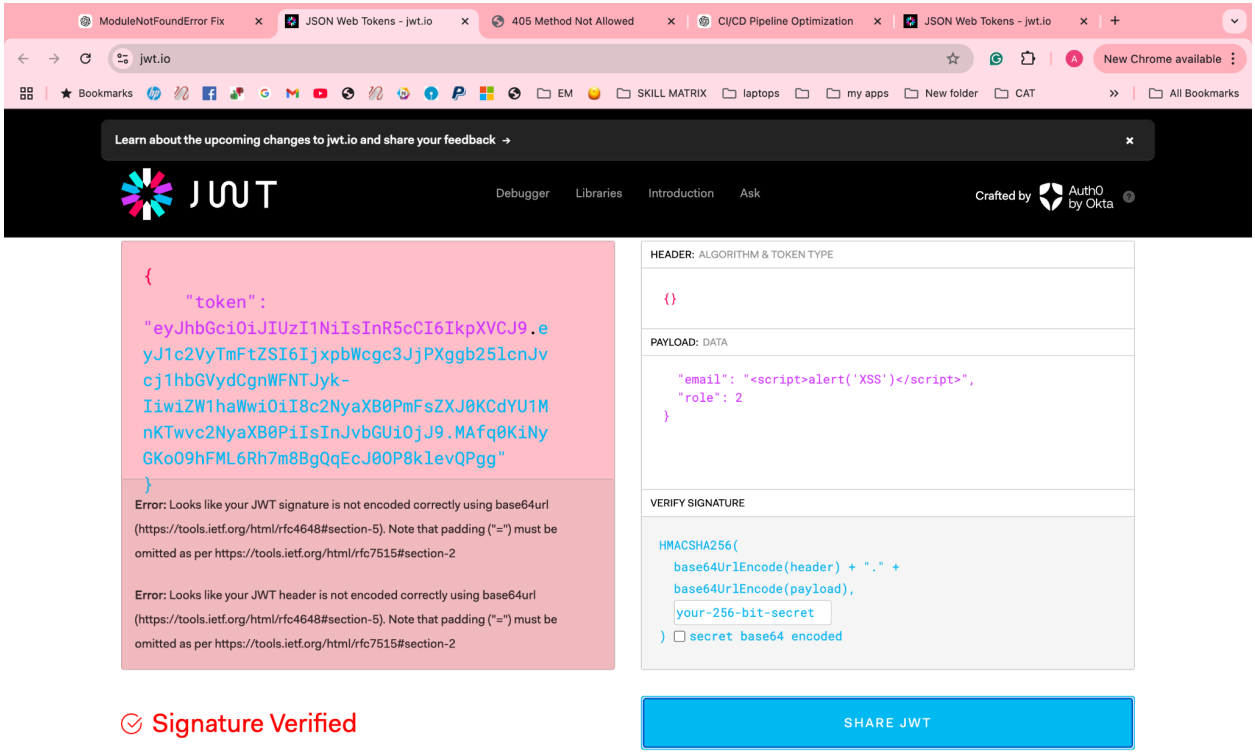
Actual Result:

- Unsanitized user inputs are included in the JWT payload, leading to potential XSS attacks.
- The `role` field is hardcoded as `2`, making the system vulnerable to privilege escalation.
- Base64 encoding errors in the JWT.

Attachments:

1. Screenshot of the API request and response:





2. Screenshot of the decoded token in jwt.io.

Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

.

Risk Scoring:

- **Likelihood:** High (9/10)
- **Impact:** Critical (10/10)
- **Overall Risk Score:** 9.5/10 (Critical)

Security_Issue5

Title:

Improper JWT Header and Signature Encoding, Sensitive Data Exposure, and Hardcoded Privileges in `/client_login`.

Priority:

Critical

Description:

The `/client_login` endpoint generates a JWT token with the following vulnerabilities:

1. **Improper Encoding:**
    - JWT header and signature are not encoded correctly using Base64 URL-safe encoding, as per the JWT standard.
    - Errors in encoding create compatibility and security issues.
2. **Sensitive Data Exposure:**
    - User information, such as `userName` and `email`, is directly included in the payload, exposing sensitive data.
3. **Privilege Hardcoding:**
    - The `role` field is hardcoded to `2`, which can lead to privilege escalation attacks if manipulated.

Steps to Reproduce:

1. Open **Postman** or any API testing tool.
2. Send a `POST` request to `http://127.0.0.1:5000/client_login` with the following payload:
    - **form-data**:
        - `userName`: `testuser`
        - `email`: `anshu@gmail.com`
        - `password`: `123456`

3. Copy the generated token from the response.
4. Decode the token using jwt.io or a similar tool.
5. Observe the following issues:
    - Encoding errors in the header and signature.
    - Sensitive data like `userName` and `email` in the payload.
    - The `role` field is hardcoded.

Expected Result:

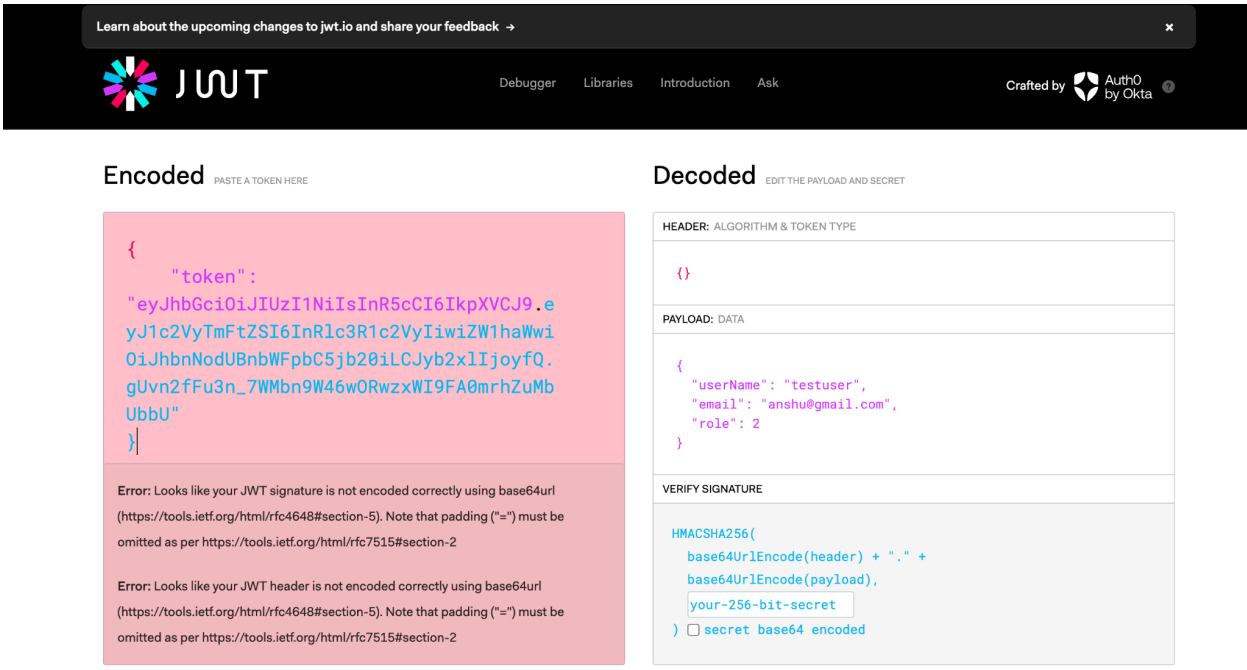
- The JWT should be properly Base64 URL-safe encoded.
- The payload should only include non-sensitive, essential information (e.g., `userId` or `role`).
- Privileges should be dynamically assigned based on user information.

Actual Result:

- The JWT contains improperly encoded header and signature.
- The payload exposes sensitive data (`userName` and `email`).
- Privileges are hardcoded to `role: 2`.

Attachments:

1. Screenshot of the JWT decoding with errors and payload data:



Environment:

- **Postman Version:** 11.22.1
- **OS:** macOS (Sonoma 14.0 (23A344))
- **Backend:** Flask Application (Localhost: 127.0.0.1:5000)

Risk Scoring:

- **Likelihood:** High (9/10)
- **Impact:** High (9/10)
- **Overall Risk Score:** 9/10 (Critical)