

# Assignment 1 - Security Patterns

## (Group 2)

### 1. Introduction

This report documents the implementation and security aspects of a secure software design project. The application under review is designed to run exclusively over HTTPS, ensuring encrypted communication. It includes authentication, authorization, and logging mechanisms to maintain high security standards.

---

### 2. Pre-requisites

Before running the application, the following steps must be completed:

#### 2.1 SSL Certificate Requirement:

Since this application is designed to run exclusively over HTTPS, an SSL certificate must be associated with it. For this demo, a self-signed certificate is created using the following command:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem  
-days 365 -nodes
```

#### 2.2 Dependency Installation:

Install the required dependencies by executing the command `pip install -r requirements.txt` in the root directory. This will install the following dependencies:

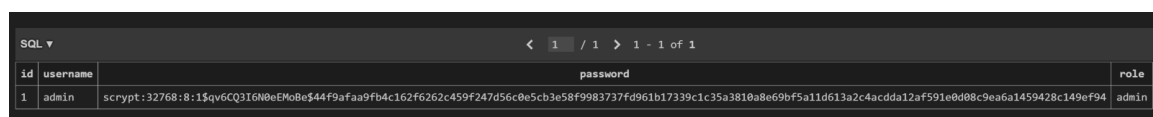
- python-dotenv
- werkzeug
- flask
- cryptography

#### 2.3 Running the Application:

Start the application running `python app.py` in the root directory.

#### 2.4 Admin User Initialization:

Upon the first run, an admin user is automatically created with credentials securely stored in an SQLite3 database.



The screenshot shows a SQLite3 database viewer interface. At the top, it says 'SQL' with a dropdown arrow. Below that, there's a navigation bar with '< 1 / 1 > 1 - 1 of 1'. The main area displays a table with the following data:

id	username	password	role
1	admin	scrypt:32768:8:1\$qv6CQ3IGN0eEK08e\$44f9afaa9fb4c162f6262c459f247d56c0e5cb3e58f9983737fd961b17339c1c35a3810a8e69bf5a11d613a2c4acdda12af591e0d08c9ea6a1459428c149ef94	admin

### 3. Implemented APIs and Security Tests

#### 3.1 POST /register

Requirements:

- Accepts a JSON request body containing username and password.
- Enforces unique usernames.
- Responses:
  - 201 Created if the user is registered successfully.
  - 400 Bad Request with an appropriate error message for invalid requests.

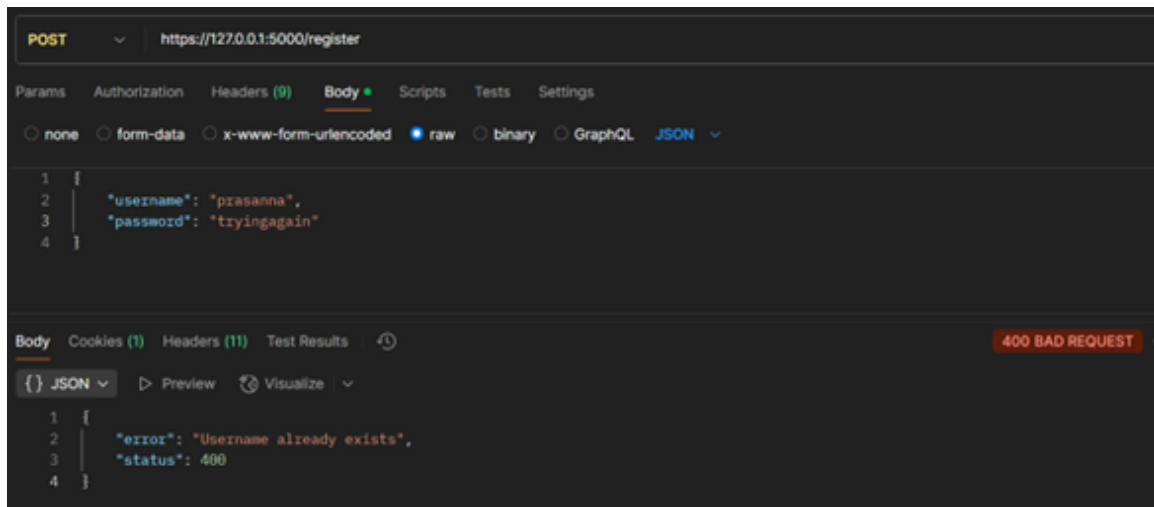
Test Results:

- Successful Registration: A new user can register successfully; the credentials are securely stored in SQLite3, and a 201 response is sent.

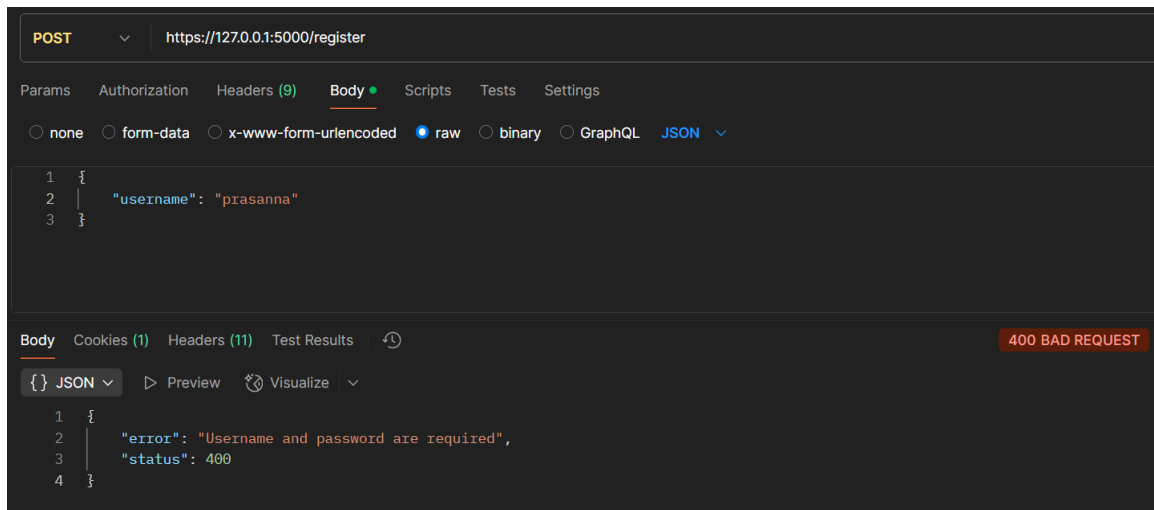
The screenshot displays a REST client interface for a POST request to `https://127.0.0.1:5000/register`. The request body is a JSON object with `"username": "prasanna"` and `"password": "thesecurityguy"`. The response is a 201 status with a JSON body: `{ "message": "User registered successfully", "status": 201 }`. Below the response, an SQL query `SELECT * FROM users;` is shown, resulting in a table with two rows: an admin user and the newly registered user 'prasanna'.

id	username	password	role
1	admin	scrypt:32768:8:1\$ngTKj1RIqvygUssr\$6b170ec6c8f9c58ed1e69dcac4aff108e9ef76384a627439c2c01953dc0ea73ffa396b0e2a134bfebf3433de88131876412e5dcbf520d3b7717384c6a03246bf9	admin
2	prasanna	scrypt:32768:8:1\$QbUawLHhIGrZTaP0\$e4dd634704933c8c5e1ff24f56b8ca64af4ebc0e2c1d5ee2bcf34edafa7c9810473bcfdd1f26e8264b8eda5f26714b9403e148d2899e811665644fbf49182443	user

- Duplicate Registration: Registering with an existing username returns a 400 Bad Request error.



- Invalid Request: Missing a required field (e.g., no password) results in a 400 Bad Request error.



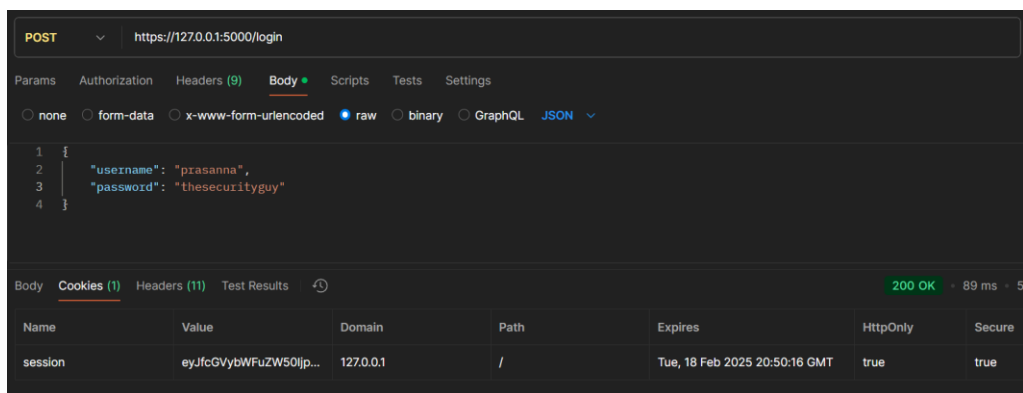
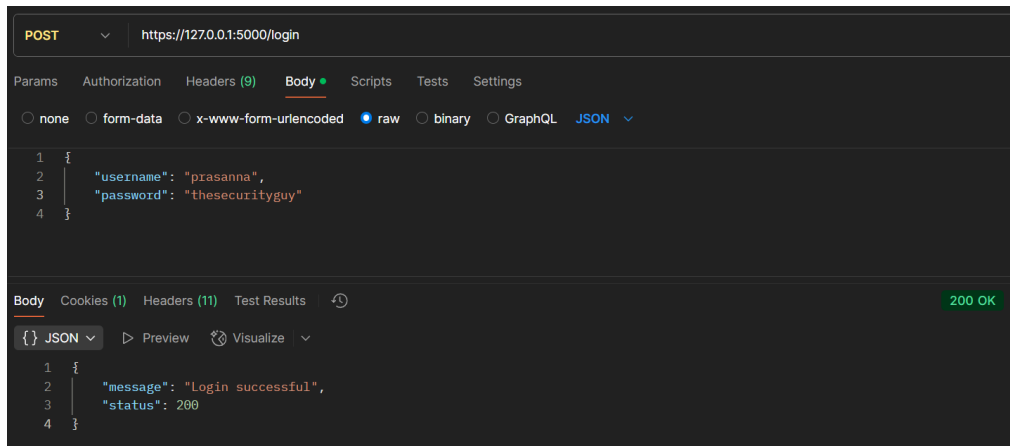
### 3.2 POST /login

Requirements:

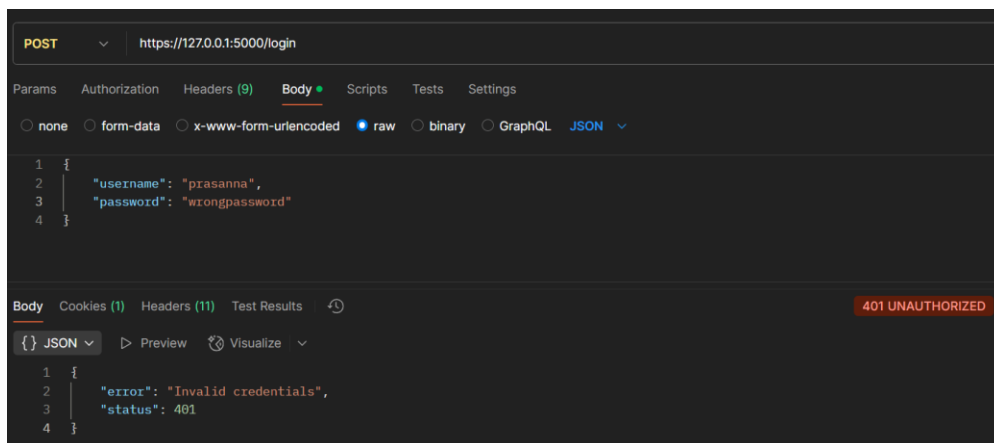
- Accepts a JSON request body containing username and password.
- Responses:
  - 200 OK with a valid session cookie if authentication is successful.
  - 401 Unauthorized for incorrect credentials.

Test Results:

- Valid Credentials: Logging in with correct credentials returns 200 OK with a session cookie.



- Invalid Credentials: Logging in with incorrect credentials results in a 401 Unauthorized error.



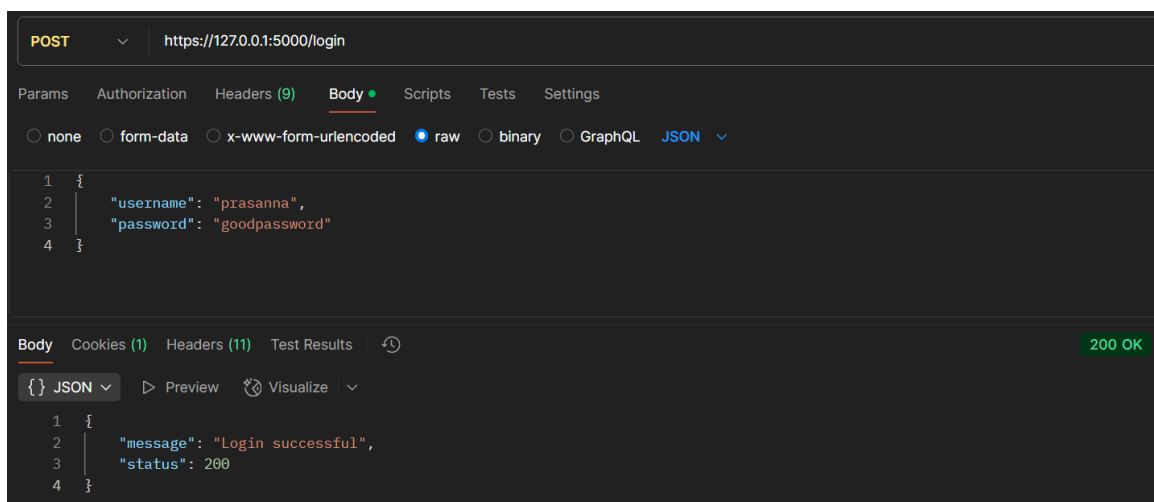
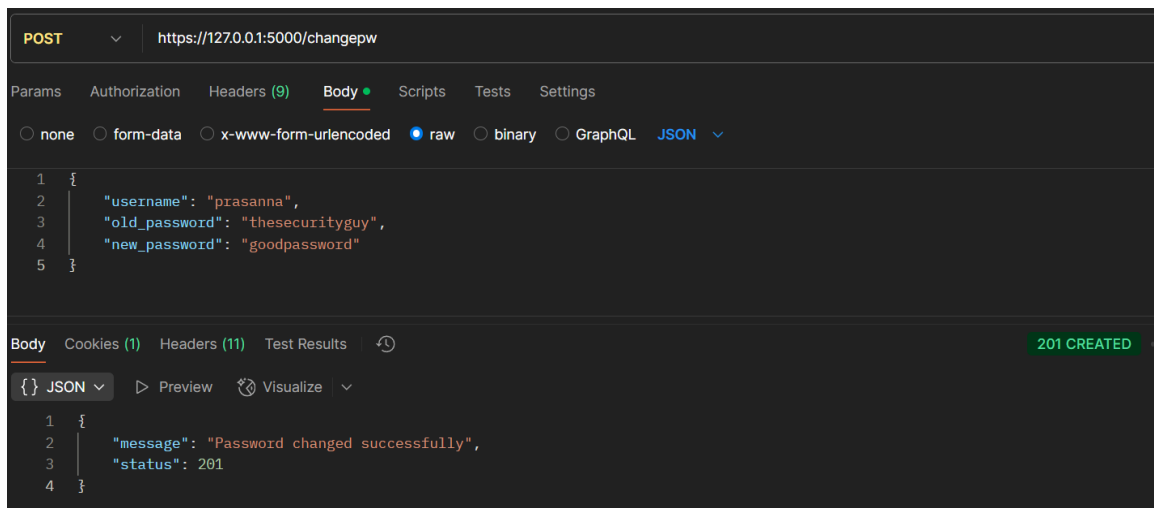
### 3.3 POST /changepw

#### Requirements:

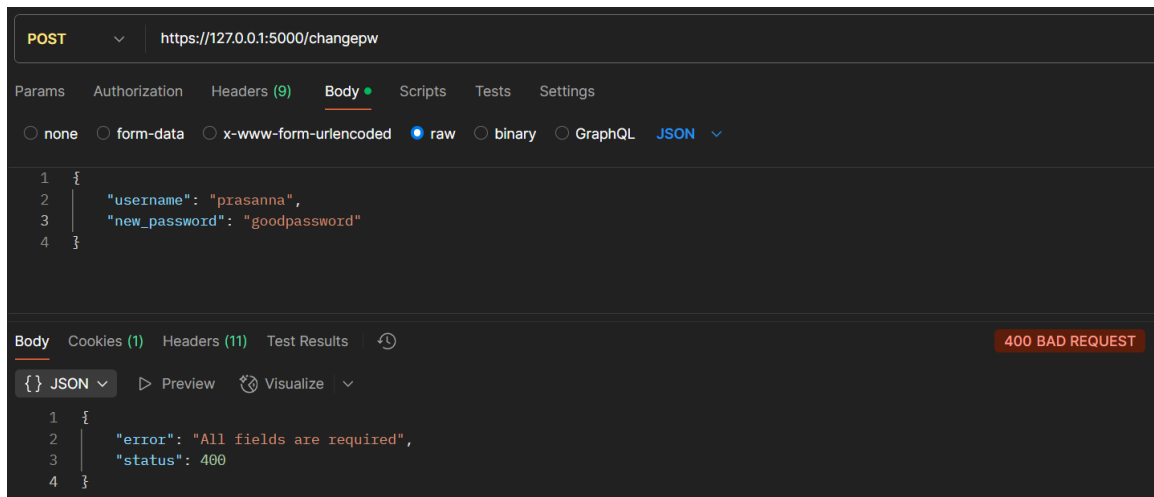
- Accepts a JSON request body containing username, old\_password, and new\_password.
- Responses:
  - 201 Created if the password is changed successfully.
  - 400 Bad Request for invalid requests.

#### Test Results:

- Successful Password Change: Authorized users can change their passwords and log in with the new credentials.



- Invalid Requests: Malformed or incorrect password change requests return 400 Bad Request.



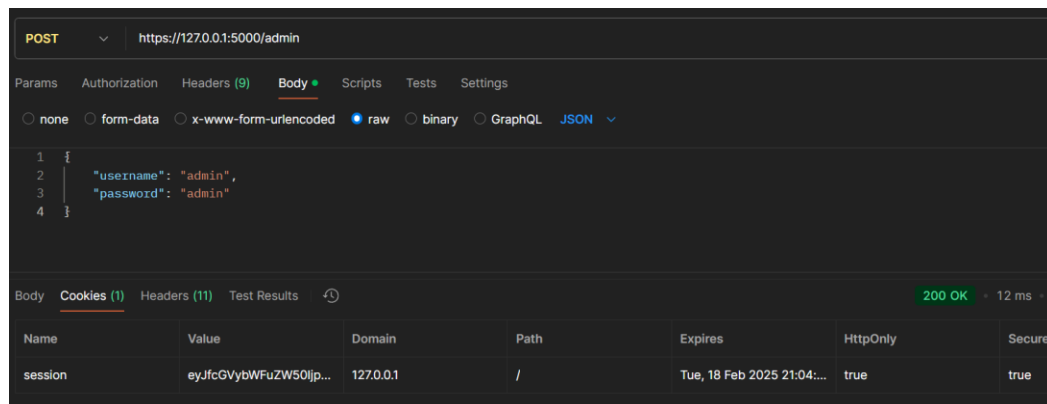
### 3.4 GET /admin

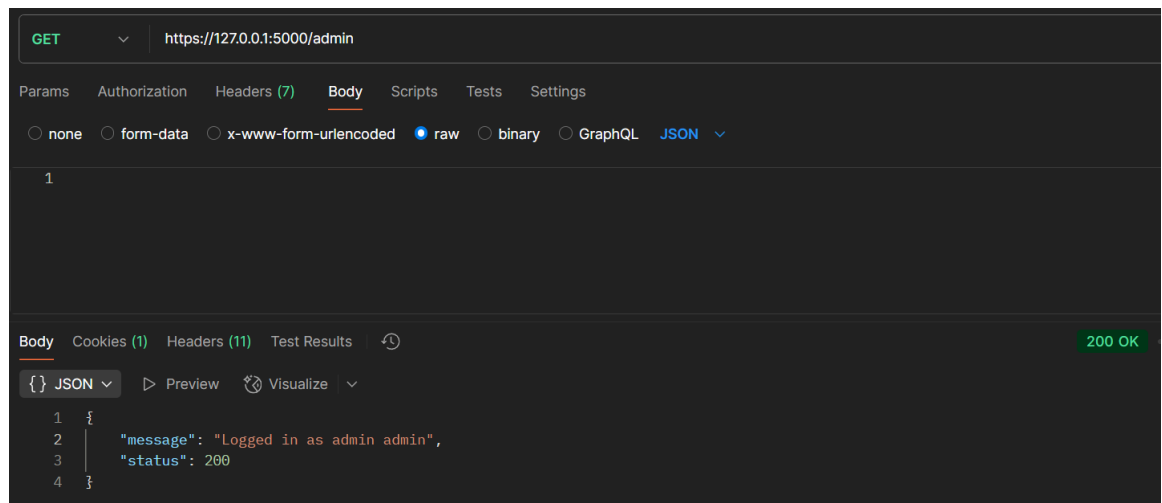
#### Requirements:

- Requires a valid session cookie set during login, received from API endpoint: `/login`
- Admin users can access the endpoint and see: `Logged in as admin <username>`
- Non-admin users are denied access.

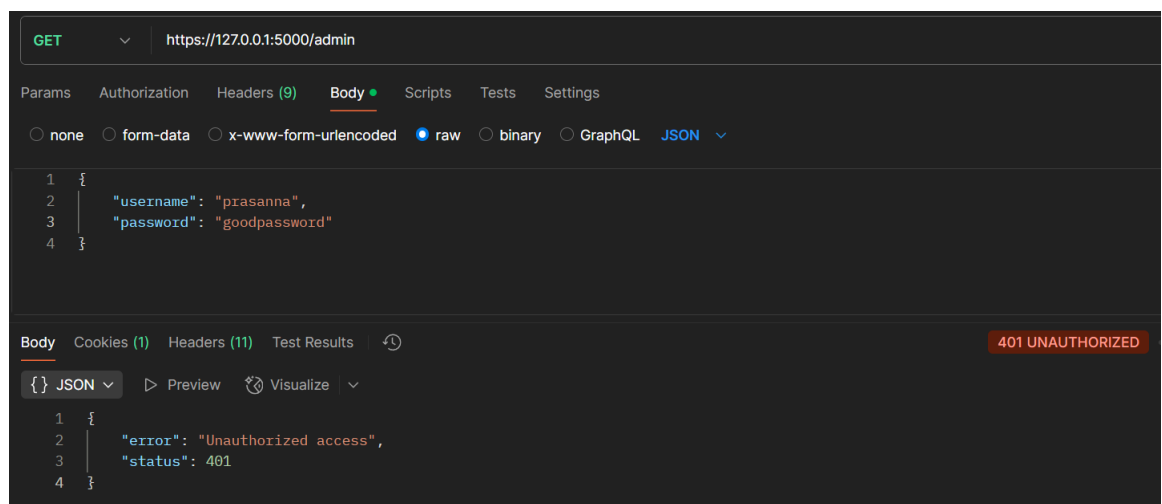
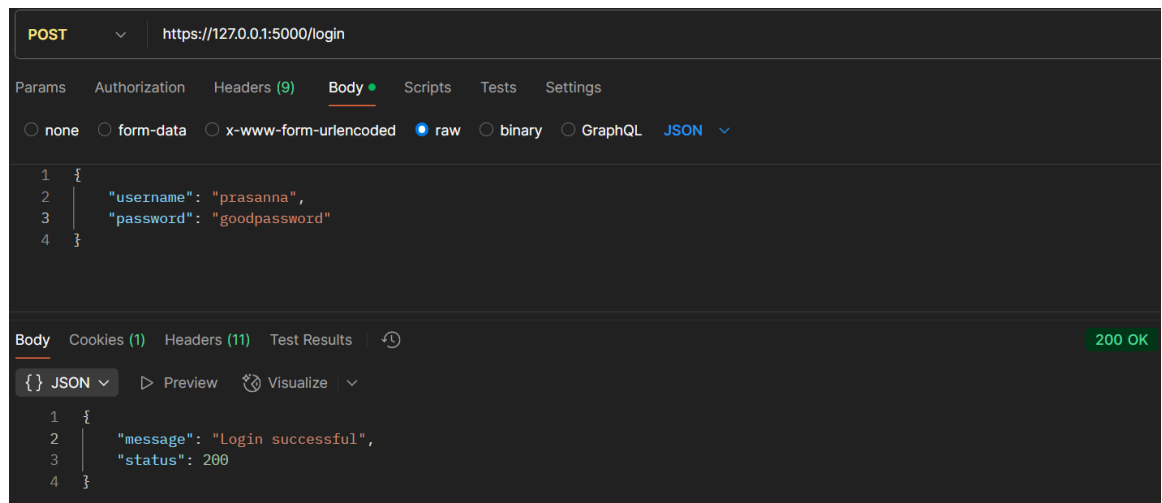
#### Test Results:

- Admin Access: Admin users successfully access `/admin` with a valid session cookie.





- Unauthorized Access: Non-admin users attempting to access /admin receive an *"Unauthorized access"* error.



## 4. Security Implementation Details

#### 4.1 Role-Based Authorization

- The system enforces role-based access control (RBAC), ensuring only users with an admin role can access privileged endpoints.

```
@app.before_request
def before_request():
    g.user = None
    if 'username' in session:
        g.user = session['username']
        g.role = session['role']
```

```
@auth_bp.route('/admin', methods=['GET'])
def admin():
    if g.user and g.role == 'admin':
        return jsonify({'message': f'Logged in as admin {g.user}', 'status': 200}), 200
    return jsonify({'error': 'Unauthorized access', 'status': 401}), 401
```

## 4.2 Secure Communication

- All communications occur over **HTTPS** using a self-signed SSL certificate (adhoc).
- Enforces **HTTPS** using **TLS** to prevent credential sniffing in MITM attacks.

```
if __name__ == '__main__':
    app.run(ssl_context='adhoc', port=5000)
```

[illegible]



### 4.3 Audit and Logging

- Key activities such as user registration, login attempts, password changes, and failed logins are logged in `app.log` in the application repository.

```
logging.info(f'User {username} registered')
logging.info(f'User {username} logged in')
logging.info(f'User {username} changed their password')
```

```
Secure Software Design > app.log
1 2025-02-18 12:00:28,651:INFO:[31mesc[1mWARNING: This is a development server. Do not use it in a production deploy
2 | * Running on https://127.0.0.1:5000
3 2025-02-18 12:00:28,651:INFO:[33mPress CTRL+C to quit[esc[0m
4 2025-02-18 12:10:42,653:INFO:User prasanna registered
5 2025-02-18 12:10:42,653:INFO:127.0.0.1 - - [18/Feb/2025 12:10:42] "[esc[35mesc[1mPOST /register HTTP/1.1[esc[0m" 201 -
6 2025-02-18 12:13:38,028:INFO:127.0.0.1 - - [18/Feb/2025 12:13:38] "[esc[31mesc[1mPOST /register HTTP/1.1[esc[0m" 400 -
7 2025-02-18 12:14:52,161:INFO:127.0.0.1 - - [18/Feb/2025 12:14:52] "[esc[31mesc[1mPOST /register HTTP/1.1[esc[0m" 400 -
8 2025-02-18 12:16:34,830:INFO:User prasanna logged in
9 2025-02-18 12:16:34,831:INFO:127.0.0.1 - - [18/Feb/2025 12:16:34] "POST /login HTTP/1.1" 200 -
10 2025-02-18 12:19:12,494:WARNING:Failed login attempt for username: prasanna
11 2025-02-18 12:19:12,495:INFO:127.0.0.1 - - [18/Feb/2025 12:19:12] "[esc[31mesc[1mPOST /login HTTP/1.1[esc[0m" 401 -
12 2025-02-18 12:20:16,348:INFO:User prasanna logged in
13 2025-02-18 12:20:16,348:INFO:127.0.0.1 - - [18/Feb/2025 12:20:16] "POST /login HTTP/1.1" 200 -
14 2025-02-18 12:26:06,214:WARNING:Failed login attempt for username: prasanna
15 2025-02-18 12:26:06,214:INFO:127.0.0.1 - - [18/Feb/2025 12:26:06] "[esc[31mesc[1mPOST /login HTTP/1.1[esc[0m" 401 -
16 2025-02-18 12:26:25,430:DEBUG:Received change password request: prasanna
17 2025-02-18 12:26:25,605:INFO:User prasanna changed their password
```

### 4.4 Secure Session Management

- User sessions timeout after 30 minutes of inactivity.
- Session cookies:
  - Are stored securely.
  - Use the `HttpOnly` flag to prevent JavaScript access (mitigates XSS attacks).
  - Have the `SameSite` attribute set to prevent CSRF attacks.

```
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(minutes=30)
app.config['SESSION_COOKIE_SECURE'] = True
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'
```

### 4.5 Password Management

- Passwords are hashed using `werkzeug.security.generate_password_hash` before being stored in the database.

```
hashed_password = generate_password_hash(password)
```

SQL ▾			
id	username	password	role
1	admin	scrypt:32768:8:1\$qv6Q3IGN0eEM0e\$44f9a9fb4c162f6262c459f247d56c0e5cb3e58f9983737fd961b17339c1c35a3810a8e69bf5a11d613a2c4acdda12af591e0d08c9ea6a1459428c149ef94	admin
2	prasanna	scrypt:32768:8:1\$w12e5qevK1lrw19U\$78293cf13cf0468a9c714cb884dd081e4f592f5fe45b8fd1761ec166ff82d5adc5d5ea52016f2a0be4f42f1faec2469a8e594bccdd2d134c59421818a1ca1a2f5	user

#### 4.6 Input Validation and Sanitization

- A dedicated function, `validate_input()`, checks all user inputs using regex.
- Helps mitigate injection attacks (SQL Injection, XSS, etc.).

```
def validate_input(data):
    if not data or not isinstance(data, dict):
        logging.debug('Invalid input: data is not a dictionary or is empty')
        return False, 'Invalid input: data is not a dictionary or is empty'
    for key, value in data.items():
        if not isinstance(value, str) or not re.match(r'^[a-zA-Z0-9_]+$', value):
            logging.debug(f'Invalid input: {key}={value} does not match required pattern')
            return False, f'Invalid input: {key} does not match required pattern'
    return True, ''
```

#### 4.7 Security Headers

- Additional security headers are enforced through the `add_security_headers` function to protect against:
  - XSS attacks
  - Clickjacking
  - MIME-sniffing vulnerabilities

```
def add_security_headers(response):
    response.headers['Content-Security-Policy'] = "default-src 'self'"
    response.headers['X-Content-Type-Options'] = 'nosniff'
    response.headers['X-Frame-Options'] = 'DENY'
    response.headers['X-XSS-Protection'] = '1; mode=block'
    return response
```

---

### 5. Environment and Deployment Security

- Environment Variables:
    - `SECRET_KEY` is stored in a `.env` file to sign session cookies.
    - In a CI/CD pipeline, these values should be stored securely in a `secrets management provider` in production deployments.
  - Development Assistance:
    - GitHub Copilot was utilized to assist with code development.
-

## 6. Conclusion

This report outlines the implementation and security mechanisms of the secure application described in Assignment 1.

The system incorporates best practices in the industry, including SSL encryption, role-based access control, session management, and input validation. Besides this, we added rate limiting (10req/min) for more security. Further enhancements could include:

- Integration with an OAuth provider for more robust authentication.
- Implementation of multi-factor authentication (MFA) to enhance security.
- More distinct input validation for each type of data.

By following these security practices, the application ensures confidentiality, integrity, and availability for its users.