**Flask Application Security Overview**

This document provides an overview of the security measures implemented in the Flask application to align with industry standards like the **OWASP Top 10 Secure Coding Practices** and **role-based access control (RBAC)**.

**1. OWASP Top 10 Secure Coding Practices**

* **Input Validation & Output Encoding:**
  + All user inputs (like username and password during signup and login) are validated for required fields and length.
  + Passwords are hashed before being stored using the werkzeug.security library.
  + **Error handling**:
    - User-friendly and non-disclosing error messages are used to prevent attackers from gaining insights into potential vulnerabilities (e.g., "Invalid username or password" rather than specifying whether the username or password was incorrect).
* **SQL Injection Prevention:**
  + SQL queries use **SQLAlchemy ORM** to avoid direct SQL execution, which reduces the risk of SQL injection. SQLAlchemy automatically escapes values in queries, thus preventing injection attacks.

User.query.filter\_by(username=username).first() # Safe query using SQLAlchemy ORM

* **Secure Error Handling:**
  + All exceptions are logged, but stack traces are not exposed to the client. Errors are generalized with appropriate HTTP status codes (e.g., 400 for bad requests, 401 for unauthorized).

**2. Role-based Access Control (RBAC)**

* **Role-based Access Control** is implemented using **JWT claims**. The user’s role (admin or user) is included as a claim in the JWT token.
* The @role\_required decorator ensures that routes are only accessible by users with the appropriate roles. The decorator checks the role claim in the JWT to enforce access control.

**Code Example**:

@role\_required('admin')

def admin\_only():

return jsonify({'message': 'Welcome, admin!'})

* Users with the **admin** role have access to routes restricted to admins, while both **admin** and **user** roles have access to the /user-or-admin route.

**3. Audit Trail for Data Changes**

* An **Audit Log** is implemented using the AuditLog model to track changes made to sensitive data.
  + **Audit logs** are created for significant actions (e.g., user signup, changes to user data).
  + Every action is logged with details about the action, entity, user ID, and timestamp.

**Audit Log Code**:

def log\_audit(action, entity, entity\_id):

user\_id = get\_jwt\_identity()

audit\_log = AuditLog(action=action, entity=entity, entity\_id=entity\_id, user\_id=user\_id)

db.session.add(audit\_log)

db.session.commit()

Example log entries could look like this:

* + **Action:** "User Signup"
  + **Entity:** "User"
  + **Entity ID:** 1 (The ID of the user)
  + **Timestamp:** 2025-05-09 12:30:00

**4. HTTPS & Secure Storage for Credentials**

* **HTTPS (SSL/TLS)**:
  + The application is configured to run over **HTTPS** using ssl\_context='adhoc' in Flask, which enables SSL encryption for secure communication.

**Code Example**:

app.run(debug=True, port=5001, ssl\_context='adhoc') # Enabling HTTPS

* **Secure Storage of Secrets**:
  + Sensitive information like the **JWT secret key** and **database credentials** are stored in a .env file, which is not committed to version control (.gitignore includes .env).
  + The JWT secret key is generated dynamically and stored in the .env file, ensuring that the key remains confidential and unique for each environment.

**Code Example**:

if not os.getenv('JWT\_SECRET\_KEY'):

new\_secret = secrets.token\_hex(32)

with open(".env", "a") as env\_file:

env\_file.write(f"\nJWT\_SECRET\_KEY={new\_secret}\n")

os.environ['JWT\_SECRET\_KEY'] = new\_secret

This ensures the secret key is rotated and securely stored.

* **Using Secrets Management**:
  + For production systems, it’s recommended to store sensitive secrets like database passwords, API keys, and JWT secret keys in a secure service like **AWS Secrets Manager**, **Azure Key Vault**, or **HashiCorp Vault**.

**5. Password Security & JWT**

* **Password Security**:
  + Passwords are hashed using **werkzeug.security**'s generate\_password\_hash method. This ensures passwords are never stored in plain text and are protected even if the database is compromised.

**Code Example**:

hashed\_password = generate\_password\_hash(password)

* **JWT Authentication**:
  + JWT tokens are used for user authentication, and they include the user's role and other claims (e.g., username).
  + The **access token** has a limited lifespan of **1 hour** and automatically expires after this period.
  + Tokens are signed using a secret key (JWT\_SECRET\_KEY), ensuring their integrity and authenticity.

**Code Example**:

access\_token = create\_access\_token(

identity=str(user.id),

additional\_claims={"username": user.username, "role": user.role}

)

* **Token Expiration**:
  + The JWT\_ACCESS\_TOKEN\_EXPIRES configuration option ensures that JWT tokens expire after 1 hour by default, requiring the user to log in again after that period.

**Configuration**:

app.config['JWT\_ACCESS\_TOKEN\_EXPIRES'] = timedelta(hours=1) # JWT expiration set to 1 hour

**6. Preventing Common Security Vulnerabilities**

* **Cross-Site Scripting (XSS):**
  + The application ensures that no user-generated content is rendered without proper sanitization. However, as the application mainly handles APIs, this concern is handled through output encoding in the frontend (e.g., React).
* **Cross-Site Request Forgery (CSRF):**
  + While not implemented in the backend, CSRF protection can be added using Flask extensions such as Flask-WTF or using double-submit cookies.
  + CSRF protection should be implemented at the frontend to prevent malicious attacks.

**Security Summary**

The following security measures have been implemented:

1. **Input Validation & SQL Injection Prevention**: All inputs are validated, and SQLAlchemy ORM is used to prevent SQL injection.
2. **Role-based Access Control (RBAC)**: Only authorized roles (user, admin) can access specific routes using JWT.
3. **Audit Trail for Sensitive Data**: All critical actions are logged with an audit trail to track data changes.
4. **HTTPS & Secure Storage of Secrets**: SSL is enabled for secure communication, and secrets are securely stored in environment files.
5. **Password Security & JWT**: Passwords are hashed, and JWT tokens are used for secure authentication with expiration times.
6. **Preventing Common Vulnerabilities**: Measures are taken to mitigate XSS and CSRF risks.