Project Report: Login System for User Management

Authors

- Diana Lorena Balanta Solano
- Danna Alexandra Espinosa Arenas
- Samuel Adrian Soto Ruiz

Introduction

This project aims to develop a secure authentication system to manage users on a platform. The system distinguishes between two types of users:

1. **Administrator:** Responsible for managing users (viewing user lists, deleting accounts, resetting passwords).
2. **Common Users:** Can change their password and view the date and time of their last login.

To ensure credential security, the **PBKDF2** algorithm with salt was utilized, providing robust protection against brute-force and dictionary attacks. The system was designed with a modular approach, enabling clear separation of responsibilities and functionalities.

Objectives

1. Design and implement a secure login system following best security practices.
2. Implement specific functionalities for administrators and common users.
3. Protect user passwords using modern techniques like hashing with salt.
4. Ensure the system is user-friendly for both administrators and common users.
5. Adapt the development process to time constraints and academic commitments.

Project Development

The system was built using **Python** as the primary language, with the **Flask** framework handling the backend and **Next.js** powering the frontend. This allowed for a well-structured implementation, separating server-side logic from client-side interactions.

Backend

The backend handles the business logic and core functionalities. It is structured into several key modules:

- `auth.py`:
  This module implements functionalities for common users:

  - **Login (`/login`)**:
    - Validates the entered credentials (username and password) by comparing the stored hash with the one generated for the entered password.
    - Updates the date and time of the last login after a successful login.
    - Forces users with blank passwords to update them immediately.
  - **Change Password (`/change_password`)**:
    - Allows users to change their password, using the `hash_password` method to securely store the new hash.
  - **View Last Login (`/last_login`)**:
    - Returns the date and time of the user's last login.
- **`admin.py`**:
  This module provides the administrator with tools for user management:
  - **Register User (`/register`)**:
    - Checks if the username already exists in the database.
    - Uses PBKDF2 to securely hash the password for the new user.
    - Allows assigning administrator permissions.
  - **Delete User (`/delete_user/<user_id>`)**:
    - Safely deletes the data of a specific user.
  - **Reset Password (`/reset_password/<user_id>`)**:
    - Changes the user's password to a blank password hash, forcing the user to update it on the next login.
  - **View Users (`/get_users`)**:
    - Returns a list of existing users with basic details like ID, username, and last login.

Frontend

The frontend, developed with **Next.js**, provides an interactive and user-friendly interface for both administrators and common users. Key features include:

- **Login Page:**
  - A form for users to enter their username and password.
- **Dashboard:**
  - **For administrators:** Displays options to manage users, including viewing the user list, resetting passwords, and deleting accounts.
  - **For common users:** Allows viewing the last login time and changing passwords.

The frontend communicates with the backend through RESTful API endpoints, ensuring secure and seamless data exchange.

Security

- ### Password Hashing:
  The **PBKDF2-HMAC-SHA256** algorithm with 100,000 iterations was implemented. Each password is stored alongside a unique salt, making rainbow table attacks highly ineffective.

PasswordUtils Class

The `PasswordUtils` class provides essential methods for securely handling user passwords in the system. It utilizes the **PBKDF2-HMAC-SHA256** algorithm, which is a secure cryptographic technique for hashing passwords. This class helps in securely storing passwords by hashing them with a salt, and provides a mechanism to verify user credentials by comparing hashed values.

Methods

---

1. **`hash_password(password, salt=None)`**
   - This method hashes the provided password using the **PBKDF2-HMAC-SHA256** algorithm.
   - If no salt is provided, it generates a random 16-byte salt. The salt ensures that even if two users have the same password, their stored password hashes will be different.
   - The method returns the hashed password and the salt used for hashing.

   Parameters:

   - `password` (str): The password that needs to be hashed.
   - `salt` (bytes, optional): A hexadecimal string used as the salt for the hashing process. If not provided, a random salt is generated.

   Returns:

   - A tuple containing:
     - `hashed_password` (str): The hashed password as a hexadecimal string.
     - `salt` (str): The salt used during the hashing process.
2. **`check_password(stored_password_hash, password, salt)`**

- This method verifies whether the provided password matches the stored password hash.
- It hashes the entered password using the provided salt and compares the result with the stored password hash.
- The method returns `True` if the password matches the stored hash, and `False` otherwise.

Parameters:

- `stored_password_hash` (str): The hash of the stored password to compare against.

- `password` (str): The password entered by the user to verify.
- `salt` (str): The salt used during the original hashing process to hash the stored password.

Returns:

- `bool`:
  - `True` if the entered password matches the stored password hash.
  - `False` if the entered password does not match the stored password hash.

- **Validation and Middleware:**
  - `@login_required` ensures that only authenticated users access certain routes.
  - `@admin_required` ensures that administrative routes are restricted to the administrator role.

- **Sensitive Data Handling:**
  Passwords are never stored in plaintext, and no sensitive data is included in API responses.

Challenges

1. **Secure Password Management:**
   - Implementing PBKDF2 with salt required understanding how to handle salts and hashes for storage and verification effectively.
2. **Time Constraints:**
   - This project was developed alongside other academic responsibilities and projects, presenting a significant challenge in terms of time management. The team had to prioritize key functionalities and coordinate tasks efficiently to meet the deadlines.
3. **Backend and Frontend Integration:**
   - Initially, we faced difficulties connecting the backend (Flask) running on `localhost:5000` with the frontend (Next.js) running on `localhost:3000`. Since they were running on different ports, we needed to configure CORS (Cross-Origin Resource Sharing) properly.
   - After configuring CORS, the issue persisted, and it was traced back to the handling of credentials. While CORS was working as expected, we were using Flask's session cookies for authentication. However, the browser was not storing the cookie, which led to authentication failures. To resolve this, we switched to using **JWT (JSON Web Tokens)** for managing user authentication, which fixed the issue and allowed seamless communication between the backend and frontend.

Best Practices Implemented in the Project

This section outlines the key best practices followed during the development of the project, focusing on security, code quality, and maintainability.

1. **Use of Secure Password Hashing (PBKDF2-HMAC-SHA256)**

- **Practice**: The password hashing mechanism uses the PBKDF2-HMAC-SHA256 algorithm along with a random salt for each user.
- **Reason**: This approach ensures that even if two users have the same password, their hashes will be different due to the unique salt, significantly improving security.
- **Good Practice**: The password is hashed and stored securely using Python's `hashlib` library, which is a cryptographically secure approach for password storage.

2. **JWT Authentication for Secure Access**

- **Practice**: JSON Web Tokens (JWT) are used for authenticating and authorizing users.
- **Reason**: JWT is a stateless authentication mechanism that ensures secure transmission of authentication data without requiring server-side sessions. It is widely adopted for APIs and modern web applications.
- **Good Practice**: The JWT token includes an expiration time to ensure tokens are periodically refreshed, reducing the risk of unauthorized access with stolen tokens.

3. **Separation of Concerns (Single Responsibility Principle)**

- **Practice**: The application follows the **Single Responsibility Principle (SRP)**, which ensures that each module, class, or function handles one specific task.
- **Reason**: This improves maintainability and scalability, as each component can be independently modified without affecting others.
- **Good Practice**:
    - The `User` model handles user-related data and authentication logic.
    - Password hashing, checking, and validation logic is encapsulated in a separate `PasswordUtils` class, making it reusable and focused on one responsibility.
    - JWT-related logic is handled in a dedicated `jwt_utils` module.

4. **Modularization and Code Reusability**

- **Practice**: Code is divided into reusable modules such as `password_utils.py`, `jwt_utils.py`, and `models.py`.
- **Reason**: This modular approach makes the codebase easier to maintain, test, and extend.
- **Good Practice**: For instance, the `PasswordUtils` class can be reused wherever password hashing is needed, and `jwt_utils.py` provides functions that can be reused across different routes for JWT handling.

5. Middleware for Access Control (Role-Based Authentication)

- **Practice**: Custom middlewares, such as `login_required` and `admin_required`, are used to enforce role-based access control (RBAC).
- **Reason**: These middlewares ensure that users can only access resources that they are authorized to view, enhancing security.
- **Good Practice**:
    - `login_required`: Ensures that the user is authenticated via JWT before accessing any route.
    - `admin_required`: Restricts access to certain routes to users with administrative privileges, preventing unauthorized access to sensitive resources.

6. Input Validation and Sanitization

- **Practice**: The application performs input validation and sanitization on user-provided data, such as passwords, to ensure they meet security criteria.
- **Reason**: This prevents malicious input that could lead to vulnerabilities such as SQL injection, cross-site scripting (XSS), or other attacks.
- **Good Practice**: Passwords must meet complexity requirements (e.g., length, uppercase, special characters), ensuring they are strong and difficult to guess.

7. Code Comments and Documentation

- **Practice**: All code is thoroughly commented in **English**, explaining the purpose of classes, functions, and key logic.
- **Reason**: This helps future developers (or even your future self) understand the reasoning behind the code, making maintenance and collaboration easier.
- **Good Practice**: The `User` class, `PasswordUtils`, and the middlewares are all well-documented with docstrings that describe their behavior, inputs, and outputs.

8. Unit Testing

- **Practice**: Unit tests are implemented to validate the correctness of critical components, such as password hashing and user authentication.
- **Reason**: This ensures that the application behaves as expected, particularly when changes are made to the codebase. It helps catch potential bugs early and provides a safety net for refactoring.
- **Good Practice**: Tests for functions like `check_password` and `hash_password` ensure that the logic works as intended and maintains security.

9. Environment Configuration and Secret Management

- **Practice**: The application uses environment variables for configuration, including sensitive information such as JWT secret keys.

- **Reason**: This improves security by keeping sensitive information out of the source code and making it easier to manage different configurations for various environments (e.g., development, production).
- **Good Practice**: The JWT signing key is stored securely in the environment, and is not hardcoded, reducing the risk of exposure.

10. **Scalability Considerations**

- **Practice**: The code is written in a way that supports scalability.
- **Reason**: As the application grows, new features or modules can be added with minimal disruption to the existing codebase.
- **Good Practice**:
  - The database model (e.g., `User`) is designed to scale, supporting the addition of more users without major changes to the core logic.
  - The modular design and role-based access control allow for easy feature expansion, such as adding more user roles or permissions.

Conclusions

- **Security:**
  The implementation of PBKDF2 with salt, along with validation and authorization practices, resulted in a robust system resistant to common threats like brute force and dictionary attacks.
- **Modular Design:**
  Separating functionalities by user roles and specific routes made the system scalable and easy to maintain.
- **Adaptation to Constraints:**
  Despite time limitations, the team successfully developed a functional system that meets the primary objectives.