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.
- **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.

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.

This system provides a solid foundation for future expansions, such as implementing token-based authentication (JWT) or improving user experience.