Secure Locker App

Samruddhi Deshmukh
G01520038
George Mason University
Fairfax, VA
sdeshmu@gmu.edu

Manasa Mummadi
G01515437
George Mason University
Fairfax, VA
mmummadi@gmu.edu

***Abstract*— In the current digital era, where data breaches and credential theft are a serious problem where relying solely on traditional passwords is no longer sufficient to secure user accounts. In this paper we present the Secure Locker App, a lightweight web application designed to store passwords securely using the PBKDF2 algorithm, a slow hashing algorithm, along with SHA-256 and unique salts for each password, which mitigates the risk of brute force and rainbow table attacks. It also features QR code generation and the use of Google Authenticator, which generates the 6-digit OTP. It also has role-based access control – user and admin.**

I. Introduction

As cyber threats are becoming increasingly sophisticated, the need for a secure and reliable application for user authentication and secure password storage has become even more critical. The traditional method of storing and authentication seems not to be sufficient It takes only a single weak password for an attacker to take over an entire system, a risk that remains present in many contemporary web applications. However, many systems continue to rely on traditional password-based authentication only, despite the fact that its flaws are well known. Users tend to choose weak, recycled, or easily guessable passwords, making them perfect targets for brute-force attacks, credential stuffing, and database intrusions. Once one password is cracked, easy unauthorized access is gained if no other measures are implemented. To thwart these challenges, we've developed a web application that incorporates a slow, secure hash function, a unique salt for each password, and a live checklist that demonstrates password security.

Secure Locker App strengthens authentication through the application of the PBKDF2 (Password-Based Key Derivation Function 2) algorithm combined with SHA-256 hashing and cryptographically secure salts. PBKDF2, with its high iteration count (200,000), makes it much more time-consuming and computationally costly for an attacker to attempt a password guess.

In addition to secure password storage, the application integrates Time-based One-Time Password (TOTP) functionality via the Google Authenticator app. During registration, users scan a QR code to link their account with an authenticator and are required to enter a 6-digit OTP during every login attempt, providing a robust second layer of security

The system defines two distinct roles:

User: Can register, log in with 2FA, and request QR regeneration.

Admin: Has access to a secure panel where they can search, view, and delete user accounts for management purposes.

II. Methodology

We developed the Secure Locker App using a modular, security-focused approach with an emphasis on usability, cryptographic best practices, and role-based access control. This section describes in detail the components of the system.

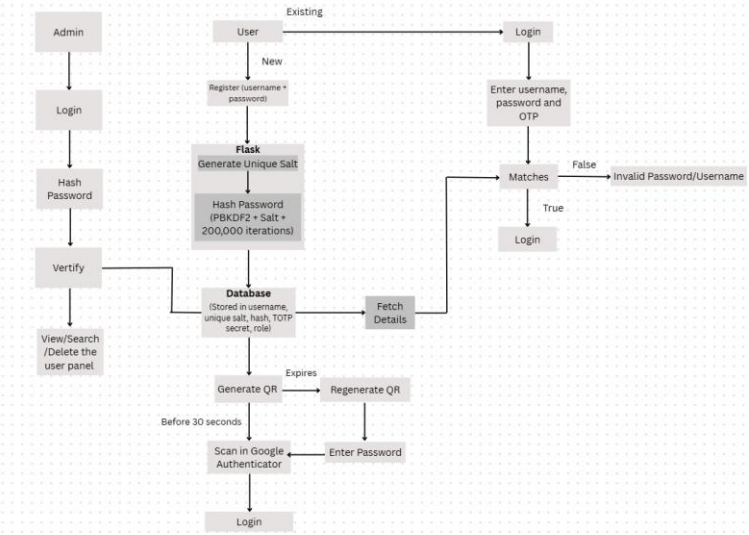


Figure 1: Architecture of Secure Locker App

1. Technologies Used

The following tools and libraries were used:

- **Flask (Python)** – A minimal web framework used for handling routes, sessions, and authentication logic.
- **SQLite** – A lightweight, serverless SQL database for secure, structured storage of user credentials and roles.
- **Hashlib + PBKDF2-HMAC-SHA256** – For securely hashing passwords with a high iteration count and per-user salt.
- **Secrets module** – Used for generating random salts and performing secure comparisons to defend against timing attacks.
- **PyOTP** – Generates time-based one-time passwords (TOTP) for Google Authenticator integration.
- **qrcode** – Generates QR codes for the TOTP URI to be scanned using the Google Authenticator mobile app.
- **HTML, CSS, Bootstrap** – Used to build responsive and interactive web interfaces.

Git repo link - https://github.com/Samruddhi0931/Secure-Locker-App.git

- **JavaScript (Client-side)** – Used for the live password strength validation checklist during registration.

2. User Registration Workflow

The user registration flow includes real-time client-side validation and secure server-side credential handling:

Password Hashing with PBKDF2

- When the user submits the registration form, a 16-byte random **salt** is generated using secrets.token_bytes().
- The password is hashed with **PBKDF2-HMAC-SHA256**, using 200,000 iterations, the chosen salt, and the entered password (converted to bytes).
- The resulting hash is converted to a hexadecimal string and stored in the database alongside the salt and user role.

Two-Factor Authentication Setup

- A unique TOTP secret is generated.
- This secret is used to create a **provisioning URI**, which is encoded into a **QR code** using the qrcode library.
- The QR code is displayed to the user, who scans it using the **Google Authenticator** app. This links their account to the TOTP system.
- All user data, including username, salt, hashed_password, 2FA secret, and role ('user') are stored securely in SQLite.

Live Password Strength Checklist (Frontend)

We implemented a **real-time password strength validator** in JavaScript and rendered on the registration page using Bootstrap styling. As the user types, the script checks if the password:

- Is at least 8 characters long
- Contains both uppercase and lowercase letters
- Includes at least one digit
- Has at least one special character (e.g., !, @, #, $)
- Avoids common words or easy patterns (optional)

Each condition is displayed as a checklist item that dynamically updates as the user types, helping users meet security requirements.

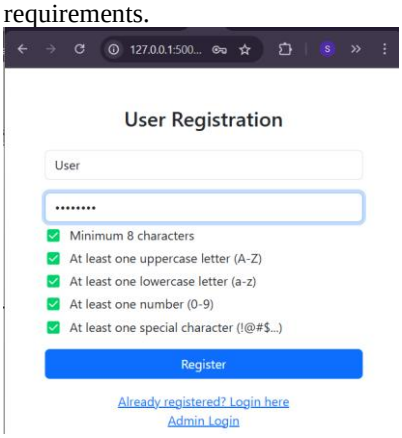


Figure 2: User Registration Page

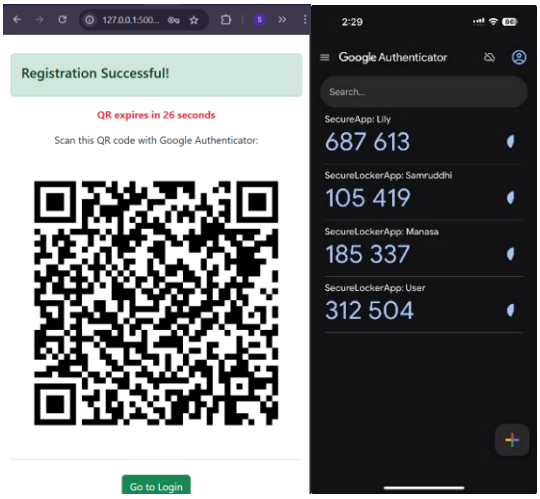


Figure 3: Successful generation of QR Page and snippet of Google Authenticator code

3. User Login Workflow with 2FA

The login process uses two stages to validate the user:

1. Password Verification

- The system retrieves the user's stored salt and hashed password from the database.
- The entered password is hashed again using the stored salt and PBKDF2.
- The newly generated hash is compared with the stored hash using secrets.compare_digest() to defend against timing-based attacks.

2. TOTP Verification

- If the password matches, the 6-digit TOTP (from the user's Google Authenticator app) is verified using pyotp.TOTP(secret).verify(code).
- If the OTP is valid, they are redirected to a success page.
- If either the password or OTP is incorrect, the user is shown an appropriate error using a Flask flash message.

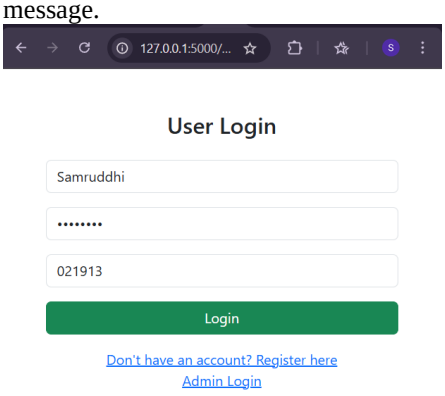


Figure 4: User Login Page

Git repo link - https://github.com/Samruddhi0931/Secure-Locker-App.git

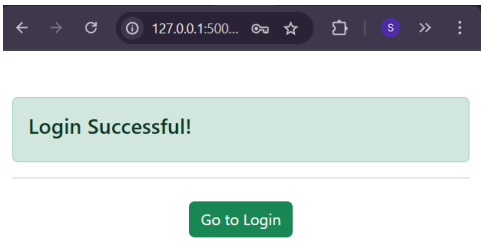


Figure 5: After Successful Login Screen

4. Security features

- **PBKDF2 with high iteration count**: Increases the time needed to guess passwords through brute-force.
- **Per-user salt**: Prevents attackers from using precomputed hash databases (rainbow tables).
- **Secure password comparisons**: Done with secrets.compare_digest() to eliminate timing attacks.
- **Session-based access control**: All routes verify session roles (e.g., 'user', 'admin') to protect endpoints.
- **QR regeneration**: Requires password re-authentication before generating a new 2FA key, to prevent unauthorized resets.

5. Admin Panel & Role-Based Access

The Secure Locker App supports two roles: **User** and **Admin**.

Admin Capabilities:

- Admins log in this endpoint (/admin_login), where credentials are verified similarly to users.
- Once authenticated, admins are redirected to the /admin_panel route, which displays all registered users.
- Admins can:
  - Search users using usernames
  - View usernames and masked hashes
  - Delete users via a secure POST request to /delete_user/<username>

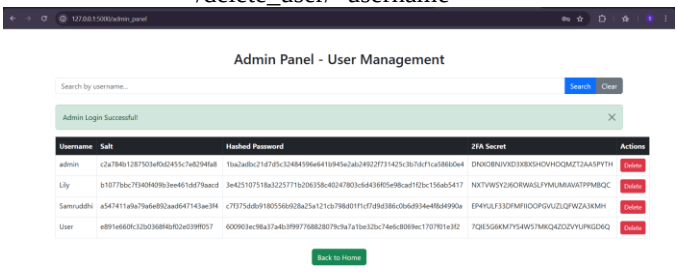


Figure 6: Admin Panel

6. QR Code Regeneration Flow

- A route (/request_regenerate_qr) allows users to initiate regeneration.
- Before a new QR code is generated, the user must re-enter their password on a **reauthentication form**.
- Upon successful verification, a new TOTP secret is generated and stored, and a new QR code is displayed for scanning.

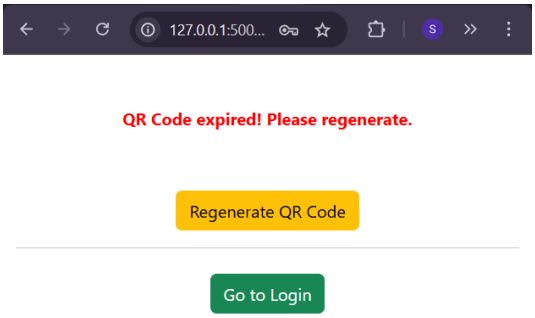


Figure 7: QR regeneration screen

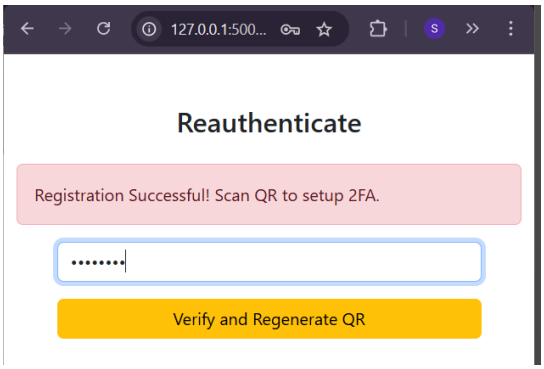
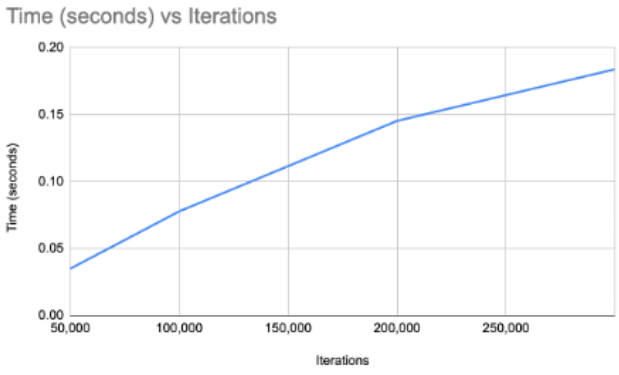


Figure 8: Password entry field for QR regeneration

III. Latency Experiment

To achieve a reasonable trade-off between security and performance, we experimented with PBKDF2 hashing using SHA-256. As the graph shows, the run time increases approximately linearly with the number of iterations, from 0.03 seconds at 50,000 to around 0.18 seconds at 300,000. Based on this analysis, we selected 200,000 iterations as a good compromise: it provides strong protection against brute-force attacks while keeping login times under 0.15 seconds for a better user experience.



Iterations	Time (seconds)
50,000	0.034951
100,000	0.077751
200,000	0.145285
300,000	0.183809

Figure 9: Performance Metrics represented using graph and table

Git repo link - https://github.com/Samruddhi0931/Secure-Locker-App.git

IV. Limitations

- **Manual 2FA Setup:** Users must manually scan a QR code and install the Google Authenticator app.
- **Login Time:** High iteration counts (200k) improve security but slightly delay the login time.
- **Scalability:** The use of SQLite and basic Flask architecture is ideal for small systems but may not be optimal for large-scale deployment.
- **No Alternative for login:** If the user doesn't have the authenticator app, the user can't log in to the system as the OTP is mandatory to log in.

V. Challeneges faced

1. **Password Enforcement:**
   Users often chose weak passwords. Implementing real-time password validation helped ensure strong security while maintaining user-friendliness.
2. **Time-sensitive QR Codes:**
   QR codes generated for 2FA expire quickly (30 seconds). Ensuring users scan them in time without confusion required user guidance and interface clarity.
3. **User Education:**
   Users unfamiliar with 2FA or authenticator apps required assistance in setup. Proper in-app instructions were added to overcome this hurdle.
4. **Shoulder Surfing Risks:**
   Since OTPs are visible for short durations, ensuring privacy in public use cases remains a concern not entirely solved.

VI. Results and conclusion

The Secure Locker App effectively demonstrates a more secure login system than traditional methods. Through the combination of:

- Strong password enforcement by adding live checklist
- Password hashing with salt and high iteration count
- Time-based 2FA

Even if a user's password is compromised, unauthorized access is still blocked. The Admin Panel adds a layer of control for system managers to monitor and manage accounts. Latency experiments confirmed that while security is slightly trade-offed with speed, the app remains responsive with acceptable delay times.

Git repo link - https://github.com/Samruddhi0931/Secure-Locker-App.git