UiA Faculty of Engineering and Science

Assignment 3 - User Authentication

Course code:	IKT-222
Semester:	Fall 2025
Pages:	10
Name:	Aleksander Einem Wågen & Teodor Log Bjørvik

Contents

1 Architectural Choices

The system was designed with a security-first mindset, balancing simplicity, maintainability, and robustness. Choices were made to align with assignment requirements while incorporating best practices for authentication systems.

1.1 Framework and Language

Python was chosen for its readability, extensive security libraries, and rapid development capabilities. Flask serves as the web framework due to its lightweight nature, fine-grained control over request handling, and seamless integration with databases like SQLite and libraries for authentication.

1.2 Database Layer

SQLite was used as the persistence layer for user accounts and authentication metadata. It requires no external server, lowering operational complexity. It guarantees persistence across restarts, distinguishing it from in-memory approaches that cannot survive failures. And it is sufficient in scale and performance for a single-host authentication prototype. The schema includes:

- `users` table: Columns for `id`, `username` (unique), `email` (unique, for OAuth2), `password_hash` (nullable for OAuth users), `created_at`, `failed_login_attempts`, `totp_secret`, `totp_enabled`.
- `login_throttle` table: Tracks attempts per username-IP pair for brute-force protection.

This design optimizes for fast queries (e.g., indexed uniques) and secures sensitive data by avoiding plaintext storage.

1.3 Password Storage Model

During database implementation, the user credentials were stored in plaintext, but was later changed. Instead, passwords were hashed using the bcrypt function before being saved to the database. Bcrypt was chosen over other hashing functions because of its adaptive cost factor (slow hashing) and salting, making it resistant to brute-force and rainbow-table attacks. For OAuth2 users, no password is stored, reducing risk.

1.4 Session Management

The application uses Flask's default session mechanism to maintain authenticated state. Flask stores session data client-side in a signed cookie: the cookie payload is serialized and signed using the application's secret key ('"FLASK_SECRET_KEY"'). In the current implementation the code sets the secret key using:

```
app.secret_key = os.environ.get("FLASK_SECRET_KEY") or os.urandom(24)
```

1.5 Protection Against Brute Force Attacks

The login endpoint is protected by a layered defence strategy. First, a stateful lockout policy is implemented using a dedicated throttle table in SQLite to persist failed attempts per username and IP pair across restarts. Second, endpoint-level rate limiting is applied using `Flask-Limiter`, which enforces global limits per IP and per username to discourage rapid automated guessing attempts.

1.6 Two-Factor Authentication

The implementation of two-factor authentication uses the time-based one time password standard generated on the client side and verified on the server side using the `pyotp` library. This approach was chosen because:

- **No external dependency on SMS or email**: TOTP is generated offline on the user's device.
- **Industry standard**: TOTP is compatible with widely adopted authenticator apps without proprietary protocols.
- **Minimal server state**: Only a static per-user secret is stored. This reduces attack surface.
- **Low operational cost**: TOTP imposes no recurring service or API cost.

1.7 OAuth2 Integration

The system incorporates OAuth2 as a client implementing the Authorization Code Flow, specifically with Google as the third-party provider. This choice complements conventional username-password login by reducing the application's direct handling of sensitive credentials, thereby minimizing risks associated with password storage and management. Instead, authentication is offloaded to a trusted provider, leveraging their robust security infrastructure, such as Google's multi-factor authentication and account recovery mechanisms.

In the implementation, provider-specific configurations: Including `client_id`, `client_secret`, `authorize_url`, `token_url`, `userinfo endpoint`, and `scopes` are stored in the Flask app's config dictionary, sourced securely from environment variables to avoid hardcoding secrets. The flow begins at the `/authorize/google` route, where a random state parameter is generated using Python's secrets module and stored in the session for CSRF protection. A query string is constructed containing `client_id`, `redirect_uri` (pointing to `/callback/google`), `response_type='code'`, `scopes` (limited to email access), and the state. The user is then redirected to Google's authorization endpoint.

Upon user consent, Google redirects back to the `/callback/google` route with an authorization code and state. The callback first validates the state against the session value to prevent CSRF attacks. If valid, it exchanges the code for an access token via a POST request to the `token_url` using the requests library, including `client_secret` and `grant_type='authorization_code'`. With the token, a GET request fetches user details from the `userinfo endpoint`, extracting the email. The system then queries the database for an existing user by email; if none exists, a new user is created with a username derived from the email prefix. Finally, the session is set with the username, redirecting to the dashboard.

2 Resources

The implementation uses the following libraries and tools:

- **Flask**: Lightweight Python web framework chosen for simplicity and control over request flow.
- **SQLite3**: Standard Python library for database operations; lightweight and embedded.
- **bcrypt**: Password hashing library providing salted, slow hashing resistant to common attacks.
- **Flask-Limiter**: HTTP endpoint rate-limiting library used for per-IP and per-user rate limits.
- **pyotp**: Generates and verifies TOTP codes compatible with Google Authenticator/TOTP apps.
- **qrcode (with Pillow)**: QR code generation for 2FA setup.
- **requests**: HTTP client for OAuth2 token exchange and user info.
- **urllib.parse** and **secrets**: Standard libraries for URL handling and secure randoms in OAuth2
- **base64** and **io**: For embedding QR images in templates.

These were selected for their maturity, security focus, and minimal dependencies. Environment variables secure sensitive configs like client IDs.

3 Challenges & Solutions

This section details security challenges, vulnerabilities, and mitigations for each task, along with implementation difficulties and resolutions.

3.1 Storing Passwords

In the initial implementation, user passwords were stored directly in the SQLite database as plaintext:

```
INSERT INTO users (username, password) VALUES ('{username}', '{password}')
```

This approach is a critical security vulnerability. If the database file is leaked or accidentally committed to a public domain, all user passwords are immediately exposed in plain view. Users also often reuse passwords across multiple services, which could also result in their accounts being compromised elsewhere.

3.1.1 Solution: Hashing with Salt

To solve the issue, plaintext storage was replaced with salted password hashing using the `bcrypt` library. Hashing ensures that original passwords cannot be recovered from stored values. When a user registers, the password is hashed before being written to the database:

```
rounds = int(os.environ.get("BCRYPT_ROUNDS", "12"))
salt = bcrypt.gensalt(rounds)  # bytes
password_hash = bcrypt.hashpw(password.encode("utf-8"), salt).decode("utf-8")
```

During login, the password is hashed and compared against the stored hash using:

```
bcrypt.checkpw(password.encode("utf-8"), user["password_hash"].encode("utf-8"))
```

3.2 Guessing at scale

Even though the password is now secure with hashing, the login endpoint can be abused for large numbers of credential guesses. Without rate controls, an attacker can rotate usernames and IP addresses to bypass simple counters, eventually succeeding against weak passwords while also consuming server resources.

3.2.1 Solution: Rate-Limiting and Lockout

Limiting the amount of login attempts in a certain amount of time, and blocking out the user after too many failed attempts, are good ways to prevent Brute Force attacks. This defence was implemented as such:

1. **Rate limiting** using per-IP and per-username ceilings at the HTTP route level (`10/min` per IP, `5/15min` per username). This disrupts the attacker's scripts.

2. **Lockout** for a specific `(username, IP)` pair after three consecutive failures, with a short cooldown. The state is persisted in the database and resets on success or window expiry.

3.3 Two-Factor Authentication

Even after securing passwords with hashing and adding brute force protections, a single authentication factor is still a weak point. If a password is leaked, guessed, or phished, the attacker gains full access. To address this, a second factor based on TOTP was added.

3.3.1 Stolen or Reused Passwords

An attacker who obtains a valid password can bypass all server-side security and log in normally.

3.3.2 Solution: Enforce 2FA

A TOTP secret is generated during registration and linked to an authenticator app. During login a 6-digit, 30-second rotating TOTP is required in addition to the password.

3.3.3 Secret exposure or Replaying of Codes

If the TOTP secret or transmitted codes were logged or intercepted, an attacker could replay them to bypass the second factor.

3.3.4 Solution: Time-Bound Codes and Controlled Exposure

The TOTP codes are never stored, only generated and verified in memory. Codes rotate every 30 seconds and are validated at the same time window.

```
totp.verify(token, valid_window=1)
```

The shared secret is only shown once during registration and not retrievable afterward, eliminating replayability.

3.3.5 User Enumeration

If the system responded differently depending on whether a user has 2FA enabled, an attacker could identify which accounts are more or less protected and direct attacks accordingly.

3.3.6 Solution: Uniform Responses

The login handler does not reveal whether 2FA is configured for a given username. The same error message is returned on failure, and 2FA checks are performed only after verifying the password to avoid observable timing differences.

3.4 Concepts and implementation of OAuth2

3.4.1 Security Challenges

Integrating OAuth2 involves delegating authentication to external providers, which introduces challenges such as ensuring secure communication, handling redirects safely, and managing dependencies on third-party services that could be unavailable or compromised. Additionally, fetching and storing user data from providers requires careful handling to avoid privacy breaches or data inconsistencies.

3.4.2 Vulnerability and Mitigations

- **Vulnerability**: Cross-Site Request Forgery (CSRF) attacks during the authorization redirect process, where an attacker could trick a user into authorizing unintended actions.
  - **Mitigation**: Implement a random `state` parameter generated using `secrets.token_urlsafe()`, stored in the session, and validated upon callback to ensure the request originated from the application.
- **Vulnerability**: Interception or leakage of authorization codes or access tokens, especially if transmitted over insecure channels or logged inadvertently.
  - **Mitigation**: Use short-lived authorization codes and avoid storing access tokens persistently; enforce HTTPS for all endpoints in production to encrypt traffic; handle tokens only in memory during the exchange process.
- **Vulnerability**: Over-privileged access grants, where the application requests more scopes than necessary, potentially exposing excess user data if the provider is breached.
  - **Mitigation**: Request minimal scopes, such as 'email' only, as configured in `app.config['OAUTH2_PROVIDERS']`.
- **Vulnerability**: Provider-specific implementation differences, such as varying endpoint URLs or response formats, leading to integration errors or unhandled exceptions that could leak information.
  - **Mitigation**: Use configurable dictionaries for provider details (e.g., Google's `token_url` as '"https://oauth2.googleapis.com/token"'); include robust error handling in routes, returning generic 401/404 messages without revealing details.

3.4.3 Benefits of OAuth2

OAuth2 offers advantages by allowing secure delegation of authentication to established providers like Google, reducing the application's responsibility for password storage and management. It leverages the provider's advanced security features, such as built-in multi-factor authentication and anomaly detection, providing users with a more robust protection layer than a standalone system might offer.
From a user experience perspective, it simplifies sign-ins with familiar ”Sign in with Google” buttons, lowering barriers to entry and supporting identities across services. Additionally, it enables seamless integration of user data (e.g., email verification) without requiring custom verification flows.

4 Recommendations

To further improve the system's security, scalability, usability, and compliance, several enhancements are proposed based on identified limitations and industry standards. These recommendations would help transition the prototype toward a more robust, production-ready application.

- **Enhance OAuth2 Integration**: Expand to support multiple providers, such as GitHub or Azure AD, by extending the configuration dictionary in `app.py`. This reduces single-provider dependency and offers users more choices. Incorporate Proof Key for Code Exchange (PKCE) to bolster security against code interception, especially client-side flows, aligning with RFC 7636 for modern OAuth implementations.
- **Advanced Security Features**: Integrate CAPTCHA (e.g., Google reCAPTCHA) after repeated failed logins to mitigate bot-driven attacks, complementing rate-limiting. Add real-time email notifications for anomalous activities, like logins from new IPs, potentially using Flask-Mail with secure SMTP.
- **Address Weak Passwords**: To counter users selecting vulnerable passwords, implement strength validation during registration. Use server-side checks requiring minimum length, character diversity, and entropy assessment via libraries like zxcvbn. Additionally, query breach databases such as Have I Been Pwned API in a hashed, privacy-safe manner to reject compromised credentials.
- **Scalability Improvements**: Migrate from SQLite to PostgreSQL for better concurrency and query performance under load, paired with Redis for session and throttle caching. Containerize the app with Docker to facilitate deployment on cloud platforms like AWS or Heroku, ensuring secrets are managed via environment variables or vaults for secure scaling.
- **User Experience (UX) Improvements**: Allow optional 2FA for OAuth2 users post-login. Develop password reset mechanisms using time-limited, signed tokens sent via email, providing self-service recovery without weakening security.
- **Testing**: Employ automated testing with pytest for robust testing of edge cases.
- **Compliance and Monitoring**: Align with GDPR by adding consent prompts for data storage and deletion options. Implement comprehensive logging of auth events (using Python's logging module or ELK stack) for forensic analysis, ensuring traceability in compliance audits.

Implementing these would elevate the system, showcasing competence in secure, scalable design while addressing real-world threats effectively.