Tampere University

COMP.SEC.300 Secure Programming

Spring 2025

Programming project report

Tuukka Salonen

151125973

14.5.2025

Contents

1. Description

The program is a web file storage application. The user can view the files available and upload new files to the server where they are encrypted and saved in the database. The user can also delete and download files from the server. Users can optionally download all files as a zip. The user must be logged in to access the file features and only the files the user has uploaded can be accessed by the user. Users can register with a username or email directly or sign in via Google or GitHub account. If the user does not have an account yet when signed in with external OAuth providers, a new account is created. Logged in users can enable two-factor authentication, which is set up with a QR-code that is read in a mobile authenticator application. When this is enabled, users are required to provide the corresponding code that is generated by the authenticator during login. This feature can be disabled; however, the code is required to be submitted once more.

2. Program structure

The program consists of React frontend, Python backend and PostgreSQL as the database solution. User authentication is done with JWT tokens, having both access and refresh tokens. The backend uses Flask framework which provides functions for some of the JWT token handling, such as token creation, setting tokens to the response and route protection. Also, it creates identities for the tokens that include custom claims. These are used to apply customized information in the tokens. The Flask app has a custom config that includes environment-variables and configurations, such as cookie settings, OAuth

client secrets and redirect URIs, database configurations and secret keys for file encryption and multi-factor authentication. Talisman library is used to set security headers, Flask-CORS to set CORS policy and Flask-limiter to apply rate limiting. The OAuth provider settings are also set up and registered via Authlib library.

The database includes user information, the uploaded files and user sessions. Models for these are used in the backend with SQLAlchemy library. During user login, a session is saved in the database where the refresh token expiration time is included. When the user logs out, the session is deleted. Also, the backend has an automated interval function where expired sessions are deleted every 5 minutes. The files are also saved in the database using the BYTEA field. The file size is restricted to 100MB to keep processing smooth in the application, even though the column could hold up to 1 GB. The database is backed up automatically on a 12-hour interval.

3. Secure programming solutions

- JSON Web Tokens are used to authenticate users. They are sent as cookies with the response on a successful login. The token is secured with a secret. Each of the backend routes that require authentication, have Flask-JWT library function that checks the validity and existence of the token and the corresponding user from the database. Access and refresh tokens are generated during login. Access tokens have a 30-minute lifespan, and refresh token a week.
- CSRF tokens are sent with the JWT access tokens and are sent with each request to the backend where they are verified together with the access tokens. The Flask JWT library handles the check automatically and blocks the request if they don't match or exist. This setup attempts to prevent cross-site request forgery attacks.
- Passwords are hashed in the database using Werkzeug security library.
- Files are encrypted using AES-256-CBC. When user registers, a key and iv are generated and the key is encrypted in the database with a master key. This user key can only be decrypted using the master key and the iv. The master key is set as an environment variable. When a file is encrypted, a file specific key and iv are generated. The file key is encrypted using the user key after the file itself has been encrypted. The same works during decryption. First the user key is decrypted using the master key, then the file key is decrypted using the user key and lastly the file is decrypted using the file key.
    - The goal is to have user-specific encryption for the files and also encrypt the files separately themselves. The master key exists so that if the database is compromised, the files cannot be decrypted without the master key.
- Rate limiting is set for each route having a reasonable amount of calls/minute per requester. This should prevent attempted DoS attacks.
- Each route has been given the allowed methods that can be used. Other requests are blocked.

- Both frontend and backend have Content-Security Policy settings, such as script-source and Cross-Origin policies to prevent XSS attacks for example.
- Backend has CORS setup where only the frontend address is accepted, credentials are included for the JWT tokens and Content-Disposition header is exposed for the file downloads to read the filename. The goal is to accept requests only from the frontend and expose only the required headers.
- Both frontend and backend have user input validation and file size checks for the file uploads.
- Two-Factor authentication with pyotp-library used to generate and verify the one-time passwords. User specific multi-factor authentication secret key is generated and encrypted using Fernet library during the setup phase. On login, when two-factor authentication has been enabled, a short-lived (5-minute) temporary access token is generated on successful credentials containing details about the temporary token in the claims. The user cannot log in with the temporary token. Full access token is generated when the two-factor authentication is successful. The MFA-code verification route checks that the temporary token exists before allowing code verification. This means that the verification process is available for 5 minutes.
- Stored user sessions are scheduled to be checked every 5 minutes, and the expired ones are deleted. The purpose is to make sure to delete sessions if the user has not logged out, but the tokens lifespan has ended.
- Scheduled database backups to ensure the data is not lost during an unexpected failure or corruption. 10 backup files are kept, and the oldest ones are overwritten during a new backup.
- Database queries are done with the library using parameterized queries to prevent injections.
- Restricted file types with checks in backend and frontend even though quite many types are allowed.
- Frontend routes are protected from users that are not logged in. For example, when trying to access a protected route using address bar, authentication is checked before loading the content and the user is redirected back to home page if not authenticated.
    - Similar to these are routes that require that the user has not logged in (login and register). If the user has logged in and tried to access the routes, for example via the address bar, the user is redirected to home.
    - The goal is to prevent unauthorized access to view the contents and also prevent unnecessary access to the routes the user doesn't need.
- Logging is included in each backend route. Events are logged in a file on successful and failed requests. The log includes the route, user id and other related details. Also, the automated database backups and session cleanups are logged. 5 log files are kept, and the oldest one is cleared and overwritten when the files are full.

- Added an option to delete account, which deletes everything from the database regarding the user (files, user data and sessions) to provide an option for user to remove all of the data related to the user.

4. Testing

4.1. Testing practices

- Manual testing
- Safety scan Python package
    - Checks the possible vulnerabilities in the backend. It didn't find anything new.
- Pipeline that includes outputs:
    - Dependency check
    - SBOM
    - Semgrep-report
    - Trivy File System Scan
    - OWASP Zap DAST-report
- GitHub dependabot
    - Analyzes the code and checks for vulnerabilities and recommends updates.
- Unit tests
    - Frontend unit tests include:
        - XSS tests, where scripts are attempted to be executed. It is checked whether the script was executed or not. These tests include scripts embedded in text, html tags inside text, scripts in image load and scripts on image load on error.
        - Route protection tests, where routes that are for logged in users only are tried to be reached while unauthenticated. It is tested that if the user is not logged in, user is navigated to the home page.
        - Content tests where it is tested that the correct page content is rendered based on user authentication status. For example, unauthenticated users should see login and register buttons on the home page. Authenticated users on the other hand should see their own username displayed and buttons for multi-factor authentication setup and files routes.
        - Tests for unauthenticated routes, that only unauthenticated should have access to, such as register and login. It's tested and redirected to home if the user is logged in.
    - Backend unit tests include:
        - Login tests where correct login responses are checked. Cookies (access token, refresh token, csrf tokens) are asserted that they are included in correct login. Incorrect password and login attempt for users that multi-factor authentication enabled are checked to have correct responses. Also, token refresh and token check routes are tested to have correct responses for both logged in and unauthenticated users. Also, a logout

test has been made, where it is checked that all the token cookies are removed.

- Register tests where successful and unsuccessful registrations are checked to have correct responses. These include tests for duplicate usernames, weak passwords and bad usernames.
- Multi-factor authentication tests where it is tested that responses for related routes are correct. It includes tests, for login with MFA enabled, generating QR-code for MFA setup and MFA removal route. The MFA-code checks are mocked, and the responses are asserted. The tests include incorrect code tests and MFA-setup attempt tests when it is already enabled.
- OAuth tests to test Google and GitHub account login. Redirects and OAuth authorizations are mocked. The routes are checked to have correct responses and redirect to the frontend URL. Access tokens are also mocked and simulated to test responses.
- File route tests where the routes are tested to expect the correct response. These tests include tests for different file actions, such as file list requests, uploading a file, downloading a specific file and deleting a file. In addition, there are tests for unauthenticated users and user input for incorrect file ids.
- CSRF-token checks are included in the tests where login is required.

4.2. Changes made due to unit tests

- Added file id validation to check the id is correct format (UUID).
- Registration and login routes were accessible when logged in. Changed to redirect user to home page if tried to access them.

4.3. Changes made due to pipeline

1. Added security headers based on DAST-report.
    - Permissions-Policy
    - Content-Security-Policy
    - X-Frame-Options
    - Cross-Origin policies
    - Cache headers

    This fixed all the alerts leaving only informal risks.

2. Updated packages to reduce the number of issues found in Dependency-Check. However, I couldn't get rid of all of the issues.

4.4. Changes made due to GitHub dependabot

- Vite version update from 6.2.5 to 6.3.4.

5. Known security issues and vulnerabilities

According to the dependency check, the project has some vulnerabilities regarding the dependencies of the used libraries. I reduced the amount of them but couldn't get rid of all of them without having version issues. There could be a way to get rid of the issues, but I ran out of time.

6. Possible further development

- File scans to identify malicious files.
- More supported OAuth providers.
- Option to insert code instead of reading a QR-code to set up MFA since that code is provided by the authenticator applications as an option.
- Add options for user data modification, such as username and password change.
- Https to increase communication security.

7. AI usage

AI was used in problem situations to debug and to recommend certain encryption solutions and Python packages. AI was also used to make some of the CSS styles in the frontend.