Secure Notes

Table of Contents

General description

This is an end-to-end encrypted note taking app. The user can make an account and access their notes from different devices in a secure manner.

The program requires Docker and Docker compose to be installed. To run it, first `*.env.template*` file should be copied to `*.env*` file in the `*backend*` directory. Then `*docker compose up –build -d*` should be run first in the the `*backend*` directory and then in `*frontend*` directory. The frontend can then be accessed on `*localhost:3000*`.

Structure

The program consists of frontend, backend and database. The frontend uses Next.js with TypeScript to introduce type safety. The backend uses Flask with Python 3. The database is PostgreSQL database.

The frontend handles generating encryption keys and encrypting and decrypting the notes. Because only one password is used, it needs to be hashed with two different salts. One of them is used for logging in and the other for encrypting the encryption key. This way the backend is never able to derive the hash used for encryption to access the data.

The hashing is done using bcrypt algorithm in the frontend. The login hash is also hashed again in the backend with Argon2 before storing it in a database. This is done because otherwise it would technically be the same as storing plain text passwords as any attacker that would gain access to the database could authenticate as any user. However, it still wouldn’t make it possible to decrypt any note data. An encryption key is used because otherwise all the note data would need to be re-encrypted when the user changes their password. This way only the encryption key needs to be re-encrypted.

The sign up and login flows between frontend and backend are shown below in figure 1.

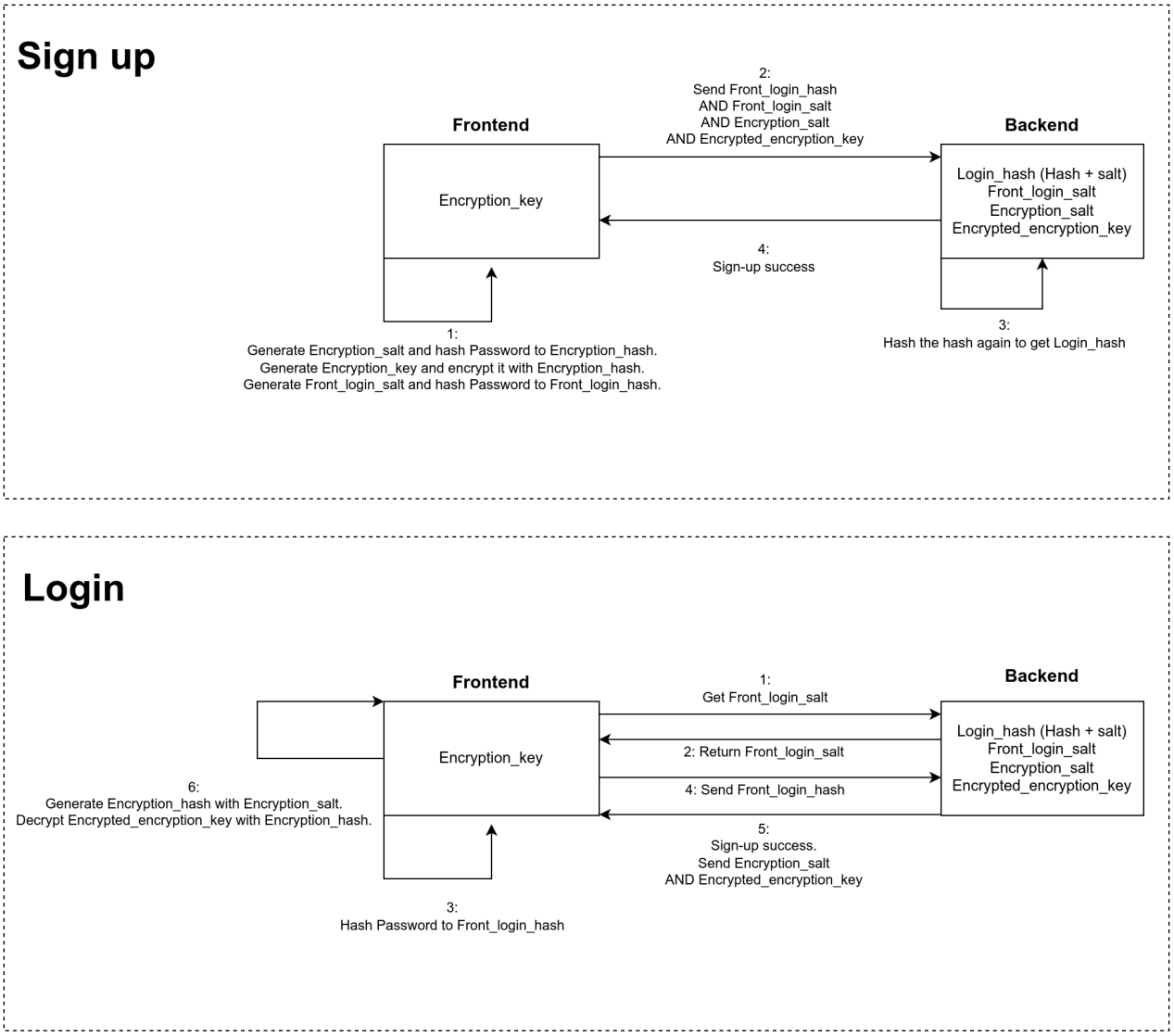

Figure 1: Sign up and login flows

Secure programming solutions

OWASP Top 10 list was used:

1. Broken Access Control

Access control is provided by session tokens that are generated and stored in the backend when logging in. These tokens are used to authorize users when accessing endpoints that provide user data. The session token is invalidated when logging out and user can also invalidate all their access tokens by logging out from everywhere. The tokens also have a creation time stored, meaning they

could be expired after a given time period if not refreshed. Session token is stored as an HTTP-only cookie in the frontend to prevent accessing it from JavaScript.

Correct implementation of access control is verified with automated tests.

2. Cryptographic Failures

The hash algorithms chosen introduce security for passwords. They are key derivation functions, meaning they are purposefully built for this task with tweakable parameters to slow down the process of hashing. Argon2 is newer and more secure and it provides three parameters to tweak; execution time, memory required and degree of parallelism. These parameters tweak how much resources the hashing takes to slow down brute forcing. The passwords are salted on both front- and backend and in the backend they are also peppered with a secret from .env file.

All user notes are encrypted with AES-256. The encryption and hashing are implemented with common libraries to prevent misconfigurations.

3. Injection

SQL injections are prevented by using prepared statements provided by the psycopg2-library.

There is also data validation in the backend and the database to ensure that users can’t store unlimited amount of data in the provided fields. There’s a max size for individual note title + body and a maximum for the amount of notes stored. The username field has maximum length and other frontend supplied fields have a set size to prevent arbitrary data in them.

CSRF tokens are implemented when posting a note or logout request. They use Flask’s implementation where server secret is used to sign them and they are signed with a session so that no other session can use the same token.

4. Insecure Design

Docker is used to limit possible exploits and zero-days. This provides some safe guards as even if the server software was compromised, the whole machine still couldn’t be accessed.

5. Security Misconfiguration

Docker is also relevant here. No default accounts are created, except in the tests which use a separate database and backend instance.

6. Vulnerable and Outdated Components

All the libraries use currently latest versions but the versions are not automatically updated to prevent sudden breakages. This means that they should be regularly updated in the future manually.

7. Identification and Authentication Failures

Login supports MFA with time-based one-time passwords (TOTP) to improve user account security.

The shortcomings of current password policy and brute force prevention are discussed in “Not implemented” chapter.

8. Software and Data Integrity Failures

This should be a future consideration.

9. Security Logging and Monitoring Failures

Logging is implemented for the development environment, but should also be implemented for the production environment. Automatic monitoring of logs should be a future consideration.

10. Server-Side Request Forgery

This threat is not really applicable for the application.

Testing

There has been a lot of manual testing done for both backend and frontend.

There are also automatic tests for the backend. There are a couple of unit tests for the backend functions but most of the tests test the API directly. There are a lot of test cases for testing the authorization and authentication of different endpoints to ensure no resources can be accessed without correct authorization. Some tests also test invalid inputs and maximum amount of data.

Not implemented

There are still a lot of secure programming related things that could be implemented in the future.

Rate limits should be added to different endpoints. For example, currently the OTP can be brute forced by testing all 1 000 000 possibilities through the login endpoint.

HTTPS should be implemented between frontend and backend to prevent eavesdroppers from acquiring credentials when they are run on separate machines.

There should also be some requirements for the user generated password. For example, minimum length and different character sets. The user should also be able to change their password.

When adding TOTP, password should be prompted from the user to prevent stolen session cookies from being able to add TOTP. Also, TOTP code should be prompted when removing TOTP from the account for the same reason.

User agents and approximate location could be parsed and saved so the user can view individual logged in sessions to notice any malicious sessions. It should be taken into account to not save user supplied input here and not allow unlimited sessions to prevent malicious user from saving arbitrary data in these fields.

Possible security issues

Lack of rate limits is the biggest security issue in the final implementation.

Lack of encryption in-transit is also a security issue when running in different machines. Even with end-to-end encryption, an attacker could serve fake frontend to the user that could send their encryption key to the attacker.