Project Assignment 3

Security Analysis

Security Risks

Just like any other application which enters and displays information into and from a database, we have some pretty serious security risks to consider. Our approach to preventing the major ones is discussed below.

Cross-Site Scripting (XSS)

There are two steps to our XSS protection. First, we almost exclusively use allow-list constrained inputs for information that will be reflected from our database. Second, said reflected data is additionally parsed and sanitised. These two methods combined should provide adequate protection against stored XSS attacks. Reflected XSS attacks are hard to protect against from an application context and will be mostly left to the user's responsibility to not be stupid.

SQL Injection (SQLI)

We follow some industry-standard procedures for protection against SQLI attacks. Like with our XSS protection, our inputs are allow-list constrained, so the possibility to inject SQL fragments is low from the get-go. Furthermore, we exclusively use prepared statements when handling user-supplied data in our server application. This reduces our vulnerability to injection attacks from low to (theoretically) not.

Miscellaneous

We do not use system command shells, so we are not vulnerable to command injection attacks.

User Authentication

It is critical we store user account passwords in a secure way. Therefore, we will use the recommended safety precautions. Sadly, due to our dependency restrictions, we are not allowed to use Argon2. As such, we will instead use Java's SecretKeyFactory, using PBKDF2, HMAC and SHA512. Additionally, we use a randomly generated salt (generated using SecureRandom) of course. This should provide very decent password protection. Session management is sadly quite simple, simply a session token (a JWT that is randomly generated each time) with expiration. Using OAuth2 would've been much better, but is simply beyond the scope of this project.

Testing Report

For software testing purposes, we wrote unit tests for the DAO functions we defined and used JetBrains's OpenAPI coverage test. What this does is that it takes our RESTful API documentation and ensures each endpoint returns a response that is valid according to the data schemas defined in the OpenAPI format. Further testing will simply be done by hand, which we understand might be unwise. However, due to the limited amount of time for this project, we figured it'd be better to show more functionalities as opposed to a smaller amount of thoroughly tested functionalities. Our reasoning for this decision was as follows; this project is, in our view, meant as more of a demo. Therefore, our client really isn't all that interested in the fact that our API handles every edge case perfectly. Conversely, we must deliver a good amount of functionalities so that the client decides to invest in our product. At such a time, it would be time to implement a more thorough testing environment, including end-to-end (e2e) tests which would directly test our front-end for expected user interactions, and an API coverage fuzzing test, which is similar to JetBrains's coverage test except that it fuzzes input data (id est calls the functions with randomly generated but valid data) and then validates multiple conditions and constraints on the effects caused by using that input data.