Security Analysis

Firstly, there was work done in order to prevent Cross Site Scripting (XSS) in the back-end. As with the authentication system, a Jersey filter was designed. This filter takes in the query parameters, headers, cookies and JSON body of the request and passes it through an XSS sanitiser. This sanitiser escapes all the HTML tags and commands and then replaces the received parameters that were dirty with the clean ones. This does not allow the input commands to be run, regardless of whether it is a stored or reflected Cross Site Scripting attack. Our filter uses the standard OWASP library. Since our implementation deals mostly with JSON objects, the JSON body is recursively cleaned by iterating through all the elements and escaping the content where necessary.

Secondly, to prevent SQL injection, prepared statements were used throughout our project to send queries to the database, in order to prevent the mixing of code and data. Logging in as an admin without the correct credentials is usually done using SQL injection but it practically can not work in our implementation since authentication is done by getting the stored hash of the password for a particular user and using PBKDF2 to check if the password is valid by comparing it to the hash. This authentication is not done by simply checking if a record exists in the database. The user's password is hashed using a library that implements the PBKDF2 algorithm. Initially, we implemented ARGON2 instead of PBKDF2 but ARGON2 was not compatible with Previder or at least that is what we concluded from our inspection, thus the change to PBKDF2.

Thirdly, in our implementation of authentication, credentials are needed to log in and the JWT token is used. With a secret signing key stored on the server, the SHA256 hashing algorithm and the timestamp of the issue is used to protect against a replay attack at a later time. The JWT token is produced by the server and attached to the cookie. This can not be reproduced without the secret signing key stored at the server. Without a token, a user is not allowed to log in and the user will get a 'bad request' error with code 400. With JWT, it is possible to specify the validity of the token. So, we specified the validity of the token and it expires in 2 hours. Thus, the user is required to log in every two hours. Logging in elsewhere revokes the old token.

Lastly, for authorization, there is a role assigned to each user. This means that not all users can use all the functionality of the website. This was implemented using the custom @Secured annotation.