Security Analysis

Team 2

Security was one of our main focuses during the course of this project. Therefore, we tried to alleviate as many security risks as possible and protect our application from malicious access.

I. SQL

Our application uses prepared statements everywhere and in all potentially vulnerable places input is pre-processed with sanitizing regexes (for example, registration form data, password reset data and settings update data is validated both on the client and server sides)

II. Tokens and Authentication filtering of the requests

Our application uses JWT (JWS) tokens for authentication. These are stateless tokens consisting of multiple claims. Our application sets the issuer of the token ("runner"), the subject (username of the user whose token it is), token's expiration and issue dates and a key (a part of user's password hash). Then, this data is signed with HMAC-SHA256 with our Secret key (can be found in *main.java.utwente.team2.settings.ApplicationSettings*).

Token is generated when the user signs into the application. Then, on the client side it is set in a cookie so that the browser automatically attaches it with every new request to the server.

All requests pointing to resources annotated with @Secured annotation will need to go through application authentication filter (*main.java.utwente.team2.filter.AuthenticationFilter*). First, this filter checks if there is a cookie with the token, and if it exists, checks its validity by verifying all the claims that are contained in this token. If there is no cookie or the token is invalid, filter redirects the user to login page with respective error message and 401(unauthorized) status code.

Only if the token is valid the request is forwarded to the requested resource.

If the user requests a resource that may only be viewed by a certain user, then the resource will further check whether the token that is supplied with the request allows access to this resource. For example, if the user CvdB is signed in (i.e. has a token) but wants to access the other user's run (JF, for instance), then the resource will catch it and redirect CvdB to the login page. The same behaviour is implemented for accessing the premium feature resources without a correct token.

When the user signs out, the server sends a request to the client side to remove the token cookie.

As JWT tokens are stateless and our server does not store them, the problem of revoking the tokens arises. To solve this problem, we set the “key” claim inside a token, which is a part of user’s password hash. When the user changes password, the salt changes too which leads to a different hash of password and salt (and there is almost no chance that the hash will be the same again even if the user saves the old password again).

If the attacker steals a password or a valid token, the user needs to change the password and old tokens will be invalid by default.

JWT tokens are also used in our application to create a URL to activate a newly registered account. The user receives an activation email which contains a clickable button-link with JWT token in it. This token is valid for 48 hours.

Password reset functionality also depends on JWT tokens. As in the case with activation emails, when the user request a password reset, a JWT token is generated and sent to the user to reset the password. For security purposes, this token is short-lived (only 15 minutes) and one-time (can be used only once to reset a password, as after the password's change, the token’s “key” claim will be invalid).

III.Password salts and hashes

Finally, newly registered users’ passwords are hashed with SHA256, then concatenated with random salt string and then hashed again. This measure helps our application to defend against rainbow table attacks.

After it is computed, the resulting hash as well as salt used to compute it are stored in the database.