Security report

|  |  |
| --- | --- |
| Security breach | Covered? |
| Protection against SQL injection | Yes |
| Protection against Cross-site scripting | Yes |
| Protection against CSRF | Yes |
| Protection against table lookup TMTO | Yes |
| Protection against Man-in-the-middle attack | No |

# Protection against SQL injection

Our web application contains multiple input boxes that are linked with issuing different CRUD requests through Jersey Servlet Container to our database e.g., when a user forgets their password, previous password will be replaced with a new password using HTTP PUT method. In order to prevent an attacker from violating the integrity of different DB requests, we have decided to implement two solutions: input sanitation for words such as: “WHERE”, “FROM”, “SELECT”, and prepared statements. The combination of the two aforementioned features prevents a perpetrator from violating the Integrity of DB requests, and should be more than enough to defy most of SQLi commands.

Protection against Cross-site scripting

In order for the web application to be safe, it should only allow to run scripts that follow the Same-Origin-Policy. If not handled properly, JavaScript code can: Alter page content, Track events, Issue HTTP requests & read responses, as well as, read and set cookies. This is why we have decided to implement a proper input sanitation for every box that involves user input. Symbols like: “<”, “>”, “%” are prohibited, and cannot be included anywhere, where user provides his input e.g., providing email for password recovery, providing email for log-in form. We believe that nothing more can be really implemented (except external libraries, or use of 3rd party software), since all of the necessary symbols for successful reflected or stored XSS, are prohibited anywhere in the webapp.

# Protection against Cross-site Request Forgery

When it comes to CSRF, there are parts that are secure, and parts that are vulnerable. Since XSS is mostly prevented, an attacker will not be able to create DOM structure, creating his own hyperlink or button pointing to rogue URL. Unfortunately, we haven’t implemented cookies that would store ambiguous and unpredictable session ids that would protect the session, thus if a user is logged in and opens a rogue URL, a breach of integrity and confidentiality may occur. Fortunately, our web app does not contain a lot of user input forms, thus CSRF will be barely ever useful for the perpetrator.

# Protection against lookup tables (Time-Memory Trade Off)

Our web application also contains very crucial feature that protects it against lookup tables used for TMTO. Each user/admin password, either new or changed, is hashed with a one-way hash function, contains encrypted salt that itself contains yet another encrypted salt. The Passwords besides being hashed and salted they are base64 encode. The encrypted salt and salt that was used to encrypt the first salt is also base64 encoded. The Algorithm that was used to hash the password is PBKDF2 algorithm and for encrypting and decrypting the salt java AES algorithms were used. These encryptions, hashing protect the web app against different lookup tables attacks. Furthermore, the salts are named as extra1 and extra2, adding yet another layer of ambiguity if an attacker has an access to our DB. Only issue is the lack of Key Derivation Functions that would ensure memory-hardness and deter majority of potential perpetrators.

# Protection against Man-in-the-middle attack

Unfortunately, while some of security issues are prevented in our web application … MITM is not one of them. Since cookies or proper certificates are not used, the attacker will easily be able to violate the confidentiality and possibly integrity of user’s request. Implementing: unpredictable and meaningless cookies and/or implementing Secure Socket Layer certificates for HTTPs, would resolve this issue and originally that was our plan, if we had more time to work on the development of our app.