Note-Bridge 1 – Assignment 3 Report

Security Analysis:

To ensure the security of our application, we have implemented mechanisms against SQL injections and XSS attacks. These mechanisms have been thoroughly tested and proven to be efficient.

To prevent SQL injections, we utilize Prepared Statements whenever we send queries to the database. This approach protects against malicious SQL code injection by treating query parameters as separate entities, eliminating the risk of unauthorized database access.

Additionally, we perform input validation on both the client and server sides to enhance security. Currently, we validate the fields related to registration and login. The server-side input validation methods are located within the Security class and include `validatePassword()`, `validateEmail()`, and `validateName()`. These methods verify that the input corresponds to the expected format using regular expressions.

The format for password validation is: `/^((?=\S*?[A-Z])(?=\S*?[a-z])(?=\S*?[0-9])[^\\s]{6,})$/` This requires a minimum of 6 characters, including at least 1 number, 1 lowercase letter, 1 uppercase letter, and no whitespace.

The format for email validation is: `/^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.[a- z]{2})?)$/` This follows the pattern `X@X.YYY` or `X@X.YY.YY`, where X can be a string made of any number of digits or letters, and Y can only be replaced with letters.

The format for name validation is: `/^(?![-. ])[-a-zA-Z. ]+$/` It should start with a letter and can include spaces, hyphens "-", and dots "."

To protect against XSS attacks, we utilize the jsoup library, which allows us to remove HTML tags from user input. We apply the `removeTags()` method from the Security class specifically when storing messages in the database.

To ensure the security of our users' passwords, we employ a robust approach. Rather than storing passwords in plain text, we store only their hashed versions. Initially, we considered using scrypt for password hashing, but due to its heavy computational cost, we opted for argon2, which has proven to be a solid choice. For each user, we store their hashed password and a salt value, both represented as byte arrays (utilizing PostgreSQL's bytea data type). The salt adds an additional layer of security by randomizing the hashes. To further strengthen the security of the hashes and protect against rainbow attacks, we utilize a constant pepper. Currently, the pepper is stored in the UserDAO class, but it should ideally be moved to a secure configuration file dedicated to storing sensitive data.

For effective session management, we rely on Jakarta's HttpSession mechanism. When a user logs into their account, a unique session with a corresponding session ID is associated with their browser. The session ID is stored in a jsessionid cookie, which is sent to the user upon their initial login. Subsequent requests from the user include this cookie, allowing us to validate the session automatically (handled by Jakarta). To enhance security, the jsessionid cookie is

marked as "httponly," making it inaccessible to JavaScript commands. This attribute prevents potential cross-site scripting (XSS) attacks that could attempt to steal the session ID.

To handle inactivity and maintain session security, we have implemented a session timeout feature. If a user remains inactive for more than 30 minutes (subject to change), their session will be automatically invalidated, requiring them to log in again. The responsibility for session management lies within the LoginServlet class.

To mitigate cross-site request forgery (CSRF) attacks, we employ a CSRF token. This token is generated uniquely for each user upon logging in, which corresponds to the creation of their session. The CSRF token acts as a protective measure by validating the authenticity of requests. Currently, we store the CSRF token as a session attribute using `session.setAttribute("csrfToken", token)`.

Overall, these security measures, including hashed passwords with salt and pepper, secure session management using Jakarta HttpSession, and CSRF token implementation, collectively strengthen the security of our application and safeguard against common security threats.

Testing:

We have conducted comprehensive JUnit tests for the UserDAO and Security classes, aligning with the user stories. We prioritize testing these classes as users and security form the foundation of our project. Since user-related data is involved in nearly every interaction, thorough testing is essential.

During the testing phase, we encountered some issues where certain methods did not function as expected. However, thanks to the modularity of our project, we easily identified and addressed these mistakes in the code, leading to improvements and bug fixes. By conducting testing during the development phase, before deployment, we ensure not only a secure application but also a thoroughly tested one.

UserDAOTest:

- `testGetUsers()`: Verifies if the entire table of users is retrieved accurately from the database and converted into a list of User objects.
- `testGetUserById()`: Ensures the correct User is returned when specifying an ID.
- `testInsertUpdateDeleteUser()`: Validates the proper insertion of a user by checking if there is no existing user with the specified email before insertion, updating the fields, and finally deleting the row.
- `testCheckUserExistsByEmail()`: Tests the functionality of the `checkUserExistsByEmail()` method.
- `testGetIdByEmail()`: Verifies that the correct ID is returned when specifying an email.
- `testGetUserByEmail()`: Ensures that the correct User object is returned when specifying an email, and that all fields of the object match the corresponding row in the database.

- `testCountUsers()`: Checks if the number of rows returned by the `countUsers()` method corresponds to the number of rows in the users table.

SecurityTest:

- `testValidatePassword()`: Verifies the proper validation of the password regex.
  - Unvalidatable examples: Password with 3 whitespaces, 123456, abcdef, ABCDEF, ABCdef, ABC123, abc123, Aa0
  - Validatable examples: Pass12, 12Pass, Adbsaj12u!@8123ihjds
- `testValidateEmail()`: Ensures the proper validation of the email regex.
  - Unvalidatable examples: Random string, my@email, ...
  - Validatable examples: My@email.com, ...
- `testValidateName()`: Tests the accurate validation of the name regex.
  - Unvalidatable examples: [blankspace]Name, .Name, ...
  - Validatable examples: First Last, ...
- `testRemoveTags()`: Verifies the removal of HTML tags by the `removeTags()` method.
  - Before/After removeTags: <> Heyo => <> Heyo, Heyo => Heyo, Heyo => Heyo, <script>console.log()</script> => [blankspace]