Security Analysis Report

Group 1

Rebah Ozkoc 3363775

Anil Sen 3367606

Ilgin Simay Ozcan 3362779

Enescu Alexandru-Cristian - 3055779

Simeon Nikolov - 3163989

Adham Ehab Magdy Selim - 2948486

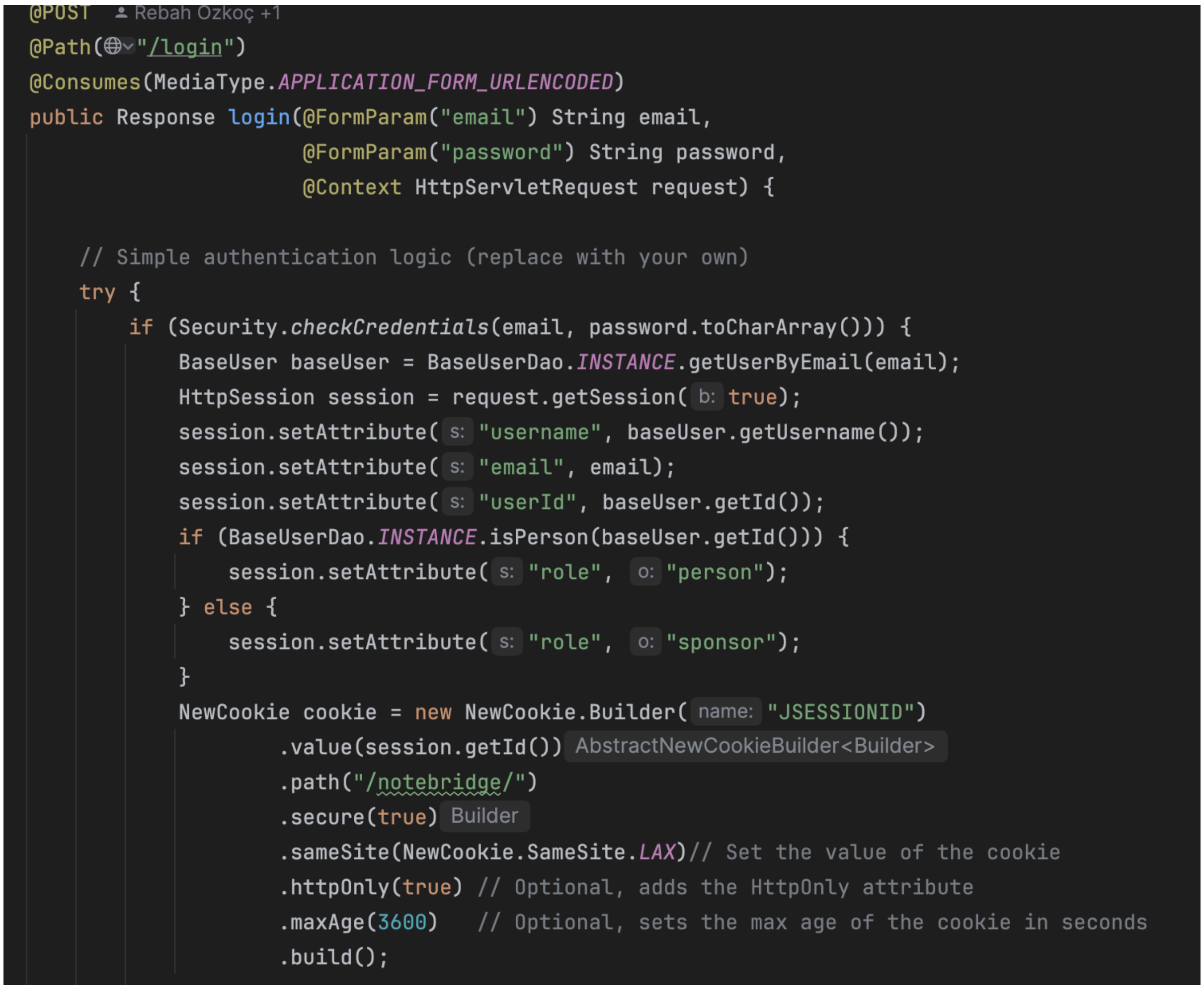
21/06/2024

Content Table

Usage of “HttpOnly” While Setting Cookies

To protect our app against XSS attack, the first thing we did was allow cookie transfer only through HTTP calls by setting the “httpOnly” field to “true” while creating a new session for the users logged in.

Thanks to that, even if malicious scripts are executed on our app, they will not be able to steal the cookie information by executing “document.cookie”.



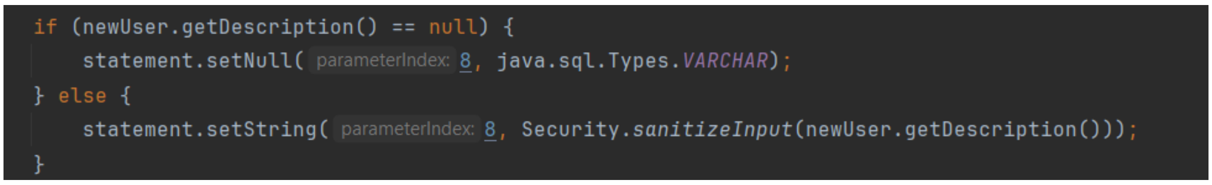
```java
@POST   Rebah Ozkoç +1
@Path("/login")
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public Response login(@FormParam("email") String email,
                      @FormParam("password") String password,
                      @Context HttpServletRequest request) {

    // Simple authentication logic (replace with your own)
    try {
        if (Security.checkCredentials(email, password.toCharArray())) {
            BaseUser baseUser = BaseUserDao.INSTANCE.getUserByEmail(email);
            HttpSession session = request.getSession( b: true);
            session.setAttribute( s: "username", baseUser.getUsername());
            session.setAttribute( s: "email", email);
            session.setAttribute( s: "userId", baseUser.getId());
            if (BaseUserDao.INSTANCE.isPerson(baseUser.getId())) {
                session.setAttribute( s: "role",  o: "person");
            } else {
                session.setAttribute( s: "role",  o: "sponsor");
            }
            NewCookie cookie = new NewCookie.Builder( name: "JSESSIONID")
                    .value(session.getId()) AbstractNewCookieBuilder<Builder>
                    .path("/notebridge/")
                    .secure(true) Builder
                    .sameSite(NewCookie.SameSite.LAX)// Set the value of the cookie
                    .httpOnly(true) // Optional, adds the HttpOnly attribute
                    .maxAge(3600)   // Optional, sets the max age of the cookie in seconds
                    .build();
```

Reflected XSS & Stored XSS and Precautions

Our group is aware of the possible danger possessed by reflected and stored XSS; therefore, we took necessary actions to prevent both.

To prevent reflected XSS attacks, inherently our app is secure against this type of attack since we do not reflect user input at all in any part of the app. We also have not yet implemented search functionality and when we do so, we know that we will be careful against reflecting back to user input while putting “Searched for : {userInput}” after the results are displayed.

Moreover, for stored XSS, we of course take user input and store it in our database. This situation endangers our app in a way that some malicious code could be inserted too. Due to this reason, we implemented sanitization functionality on our “Security” class.



```
if (newUser.getDescription() == null) {
    statement.setNull( parameterIndex: 8, java.sql.Types.VARCHAR);
} else {
    statement.setString( parameterIndex: 8, Security.sanitizeInput(newUser.getDescription()));
}
```

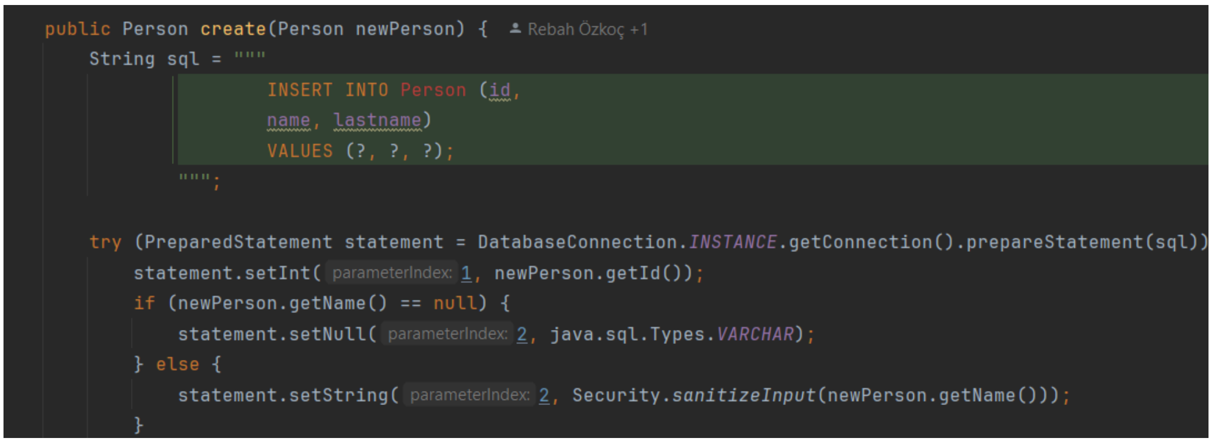
For each UPDATE & INSERT statement in our data access object classes, we sanitize our string inputs to make sure they do not contain malicious scripts, etc. and we do not serve them in our app to our users.

SQL Injection Thread and Precautions

SQL injection is a common threat where attackers can manipulate SQL queries by injecting malicious code. This method of attack can severely harm the confidentiality, integrity, and availability of the system. There are a couple of suggested protections against this type of attack. Some of them are:

- Using prepared statements with parameterized queries. This ensures that SQL code is separated from data input.
- Validating and sanitizing all user inputs, ensuring they conform to expected formats, and rejecting suspicious data.

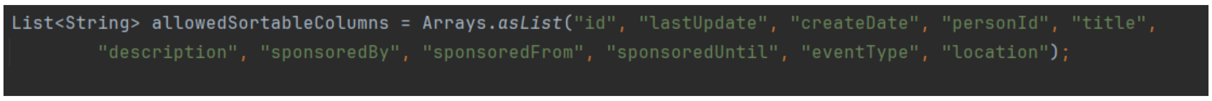
We tried to conform to these principles and protect our application from SQL injection attacks. We used prepared statements all the time to prevent the dangerous string variables from getting into the database.



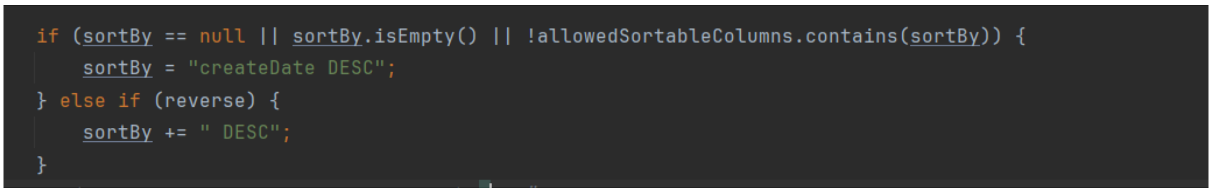
```
public Person create(Person newPerson) {  Rebah Özkoç +1
    String sql = """
                INSERT INTO Person (id,
                name, lastname)
                VALUES (?, ?, ?);
            """;

    try (PreparedStatement statement = DatabaseConnection.INSTANCE.getConnection().prepareStatement(sql))
        statement.setInt( parameterIndex: 1, newPerson.getId());
        if (newPerson.getName() == null) {
            statement.setNull( parameterIndex: 2, java.sql.Types.VARCHAR);
        } else {
            statement.setString( parameterIndex: 2, Security.sanitizeInput(newPerson.getName()));
        }
```

Here is an example insert query where the user input is first sanitized and then added to the SQL statement with the parameterized inputs.

Due to the design of the JDBC, we cannot pass the column names with parametrized inputs to the SQL statement. Thus when we need to dynamically change the column name in an SQL statement we use a whitelist to prevent unwanted inputs for this parameter.



```java
List<String> allowedSortableColumns = Arrays.asList("id", "lastUpdate", "createDate", "personId", "title",
        "description", "sponsoredBy", "sponsoredFrom", "sponsoredUntil", "eventType", "location");
```



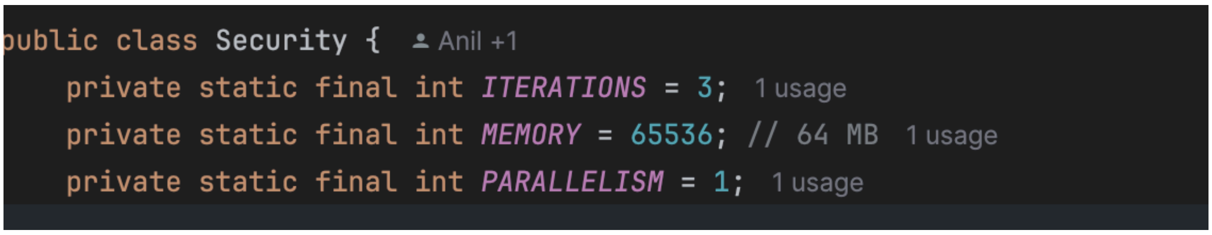
```java
  if (sortBy == null || sortBy.isEmpty() || !allowedSortableColumns.contains(sortBy)) {
      sortBy = "createDate DESC";
  } else if (reverse) {
      sortBy += " DESC";
  }
```

Current Progress Regarding the CSRF Tokens

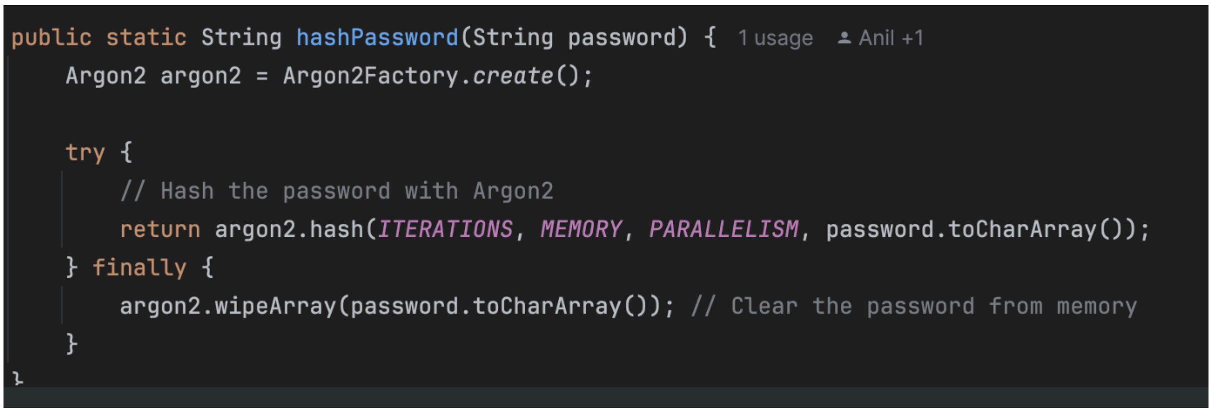
Our app currently does not send CSRF tokens to the client in the backend section and does not send CSRF tokens back to the server while making POST & PUT requests in the frontend section. We are planning to work on the functionality of CSRF tokens to protect our app against “Cross-Site Request Forgery” attacks.

Authentication and Authorization Security

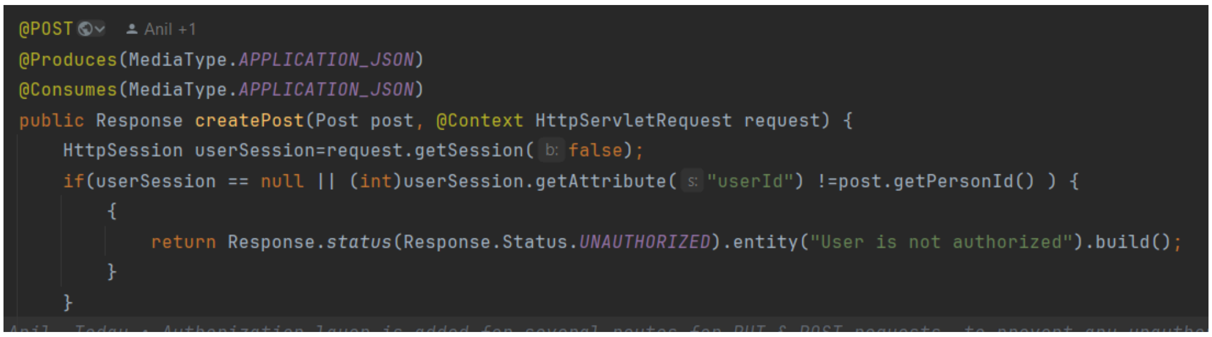
For authentication and authorization, we have a user sign-up and sign-in mechanism. When the user signs up it uses a unique email and a strong password. This password is salted, peppered, and hashed with the Argon2 password-hashing function which is a suggested memory hard hash function.

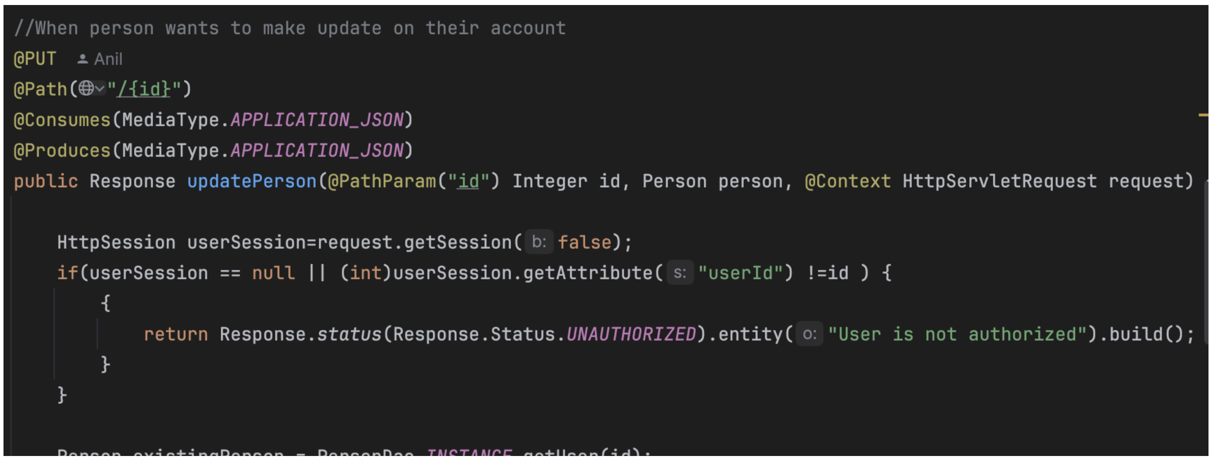


```java
public class Security {   Anil +1
    private static final int ITERATIONS = 3;   1 usage
    private static final int MEMORY = 65536; // 64 MB   1 usage
    private static final int PARALLELISM = 1;   1 usage
```

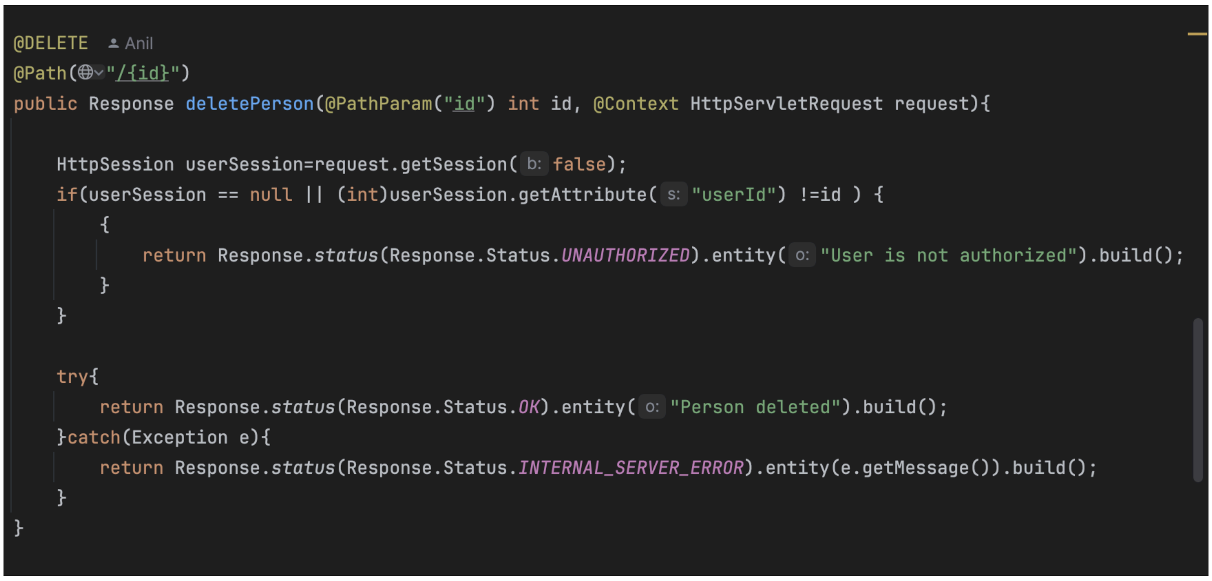

```java
public static String hashPassword(String password) {  1 usage  Anil +1
    Argon2 argon2 = Argon2Factory.create();

    try {
        // Hash the password with Argon2
        return argon2.hash(ITERATIONS, MEMORY, PARALLELISM, password.toCharArray());
    } finally {
        argon2.wipeArray(password.toCharArray()); // Clear the password from memory
    }
}
```

Also, we implemented authorization checks for some crucial endpoints. Our app does not allow any UPDATE or PUT requests if the request is not coming from the owner of that entity. Through reaching out to user sessions that we give when users log in, we check if they are authorized to do the intended action.


```java
@POST  Anil +1
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response createPost(Post post, @Context HttpServletRequest request) {
    HttpSession userSession=request.getSession( b: false);
    if(userSession == null || (int)userSession.getAttribute( s: "userId") !=post.getPersonId() ) {
        {
            return Response.status(Response.Status.UNAUTHORIZED).entity("User is not authorized").build();
        }
    }
Anil, Today • Authorization layer is added for several routes for PUT & POST requests, to prevent any unauthe
```



```
//When person wants to make update on their account
@PUT  Anil
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response updatePerson(@PathParam("id") Integer id, Person person, @Context HttpServletRequest request)

    HttpSession userSession=request.getSession( b: false);
    if(userSession == null || (int)userSession.getAttribute( s: "userId") !=id ) {
        {
            return Response.status(Response.Status.UNAUTHORIZED).entity( o: "User is not authorized").build();
        }
    }

    Person existingPerson = PersonDao.INSTANCE.getUser(id);
```



```
@DELETE  Anil
@Path("/{id}")
public Response deletePerson(@PathParam("id") int id, @Context HttpServletRequest request){

    HttpSession userSession=request.getSession( b: false);
    if(userSession == null || (int)userSession.getAttribute( s: "userId") !=id ) {
        {
            return Response.status(Response.Status.UNAUTHORIZED).entity( o: "User is not authorized").build();
        }
    }

    try{
        return Response.status(Response.Status.OK).entity( o: "Person deleted").build();
    }catch(Exception e){
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity(e.getMessage()).build();
    }
}
```