# SECURITY REPORT

OWASP top 10 security risks

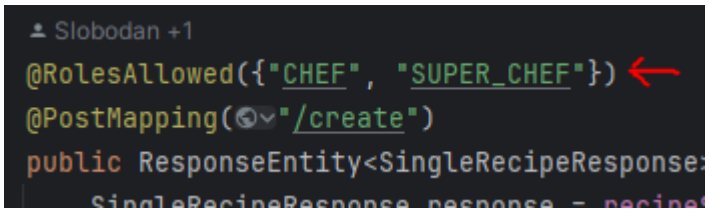
SLOBODAN STARCEVIC
Semester 3

# Contents

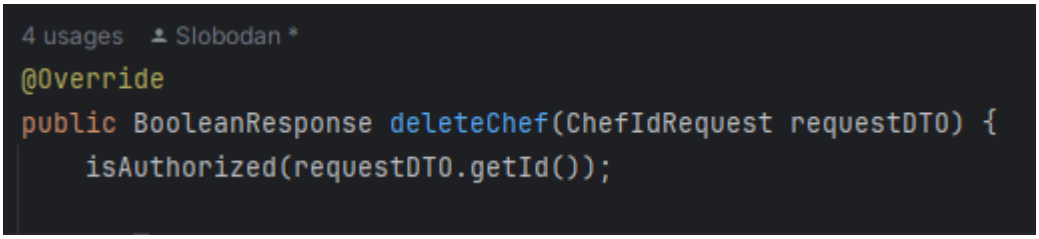
## Broken Access Control

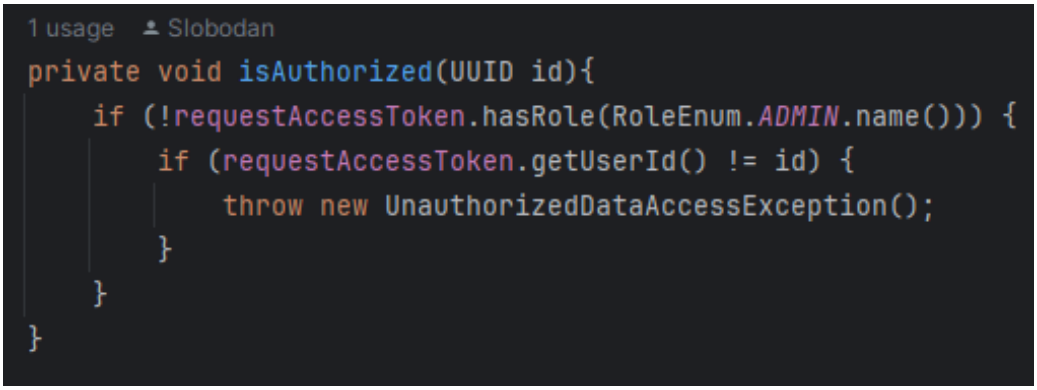
In the dishdive app I have 2 ways of making sure users don't act outside of their permissions. To make it possible the front end sends the users jwt token, if available, to the backend with each request. Inside of this token all data needed for authorization can be found like the user id and role.

The first one is a role restriction on API endpoints. As you can see the endpoint is annoted with '@RolesAllowed' which is a built in feature of spring security. The specifics of how the authorization is handled can be found in the WebSecurityConfig file.




The second method of securing access is done manually by checking the id of the owner of the object. First it is checked to see if the user is an admin, who have complete clearance. Then it is checked if the requester has the same id as the object owner. So in case the owner of a recipe is not the same as the requester that wants to delete it, the UnauthorizedDataAccessException is thrown.



## Cryptographic Failures

In the Dishdive app I have 2 ways of securing requests and data by using encoding.

```
Slobodan
public AccessTokenEncoderDecoderImpl(@Value("${jwt.secret}") String secretKey) {
    byte[] keyBytes = Decoders.BASE64.decode(secretKey);
    this.key = Keys.hmacShaKeyFor(keyBytes);
}
```

The first one is the encoding of the jwt token signature. As we know a jwt token can simply be decoded to get the payload and header, but the signature is something secured by having a secret key in the backend, which I have then made a environment variable out of in the application.property file where it cant be reached. The decoded signature proves that the holder of the token is the actual owner and thus gains the rightful access.

```
String encodedPassword = passwordEncoder.encode(requestDTO.getPassword());

ChefEntity newChef = ChefEntity.builder()
        .id(UUID.randomUUID())
        .username(requestDTO.getUsername())
        .email(requestDTO.getEmail())
        .password(encodedPassword)
        .userRole(RoleEnum.CHEF)
        .build();
```

The second point of security is on the creation of a new user. The encoding of a password is important to make sure it isn't exposed in the database which would be catastrophic if the database was breached. As you can see I make use of a encoding library. The decision to let a library do it instead of making it custom-made, is simply that is it more secure. Making something myself would not only be amateur like but also be low security.

## Injection

SQL injection are very unlikely to happen in the Dishdive app as it makes use of Java Persistence API (JPA). This ORM makes use of parameterized queries and prepared statements which means user input is not directly embedded into SQL queries and thus ensures no injection can occur.

## Insecure Design

To ensure a secure design of the Dishdive application I have made the following considerations.

### Secure network architecture

To ensure the network is secure the backend only allows the connection of the front end domain, this means that other domains aren't allowed to make use of my backend endpoints, ensuring access security.

### Session management

To ensure user can be logged in and stay logged in securely token were made with an expiration date which ensures each token has a time to their session and cant just be used as an easy entry point forever. To make sure of this the front end has an implementation which checks the expiration time routinely.

### Insecure Data Storage

For this point we have talked about it already in the previous chapter 'Cryptographic Failures'

### Inadequate Error Handling and Logging

I have ensured all fronts are covered in the backend protecting the app from unknown errors which it then does not know how to handle resulting in a crash. Of course it is irresponsible to say that you can have all possible errors covered, and for that I have made sure to also add code to catch any additional exception that I have not covered already.

### Insecure Direct Object References (IDOR)

We have talked about this in the previous chapter 'Broken Access Control'

## Security Misconfiguration

For the topic Security Misconfiguration, I don't have much to say as it is a topic leaning more on handling the network around the application and ensuring it is secure, which is a topic beyond this semester. The one point I can talk about is not showing the full stack trace when error are handled, this is ensured by having custom exceptions with pre-determined messages telling the user what went wrong.

## Vulnerable and Outdated Components

Currently the dependencies of the Dishdive app are all the latest versions, ensuring an up to date and secure dependency library. The versions of all dependencies can be found in the backend in the build.gradle file and in the frontend in the package.json file. Knowing the version is an important part of also being aware of your application.

## Identification and Authentication Failures

This is currently not implemented but is a high priority.

To ensure secure authentication Dishdive will require strong password creation, needing a capital letter, lowercase letter, a number, and a minimum of 8 characters. With this I ensure that bad actors can't use techniques like credential stuffing to randomly guess users credentials.

Besides that, as I mentioned before, on each request made to the backend the jwt token is also sent as an authorization header which lets the backend check if the token holder is also the token owner.

## Software and Data Integrity Failures

As mentioned before, because of the use of the JPA ORM sql injection is not possible, ensuring data integrity in the database. Also mentioned before, using a signature on the jwt token ensures data cant be tampered with if you aren't the rightful owner of the data.

Additionally, Dishdive will ensure that all user inputs are 'sanitized' by having input validation on the frond end and backend, thus never letting unwanted data reach the database.

## Security Logging and Monitoring Failures

Not implemented. OWASP has made me realise I should take care of it which will now become a new item on my to do list.

## Server-Side Request Forgery (SSRF)

The Dishdive app makes use of secure API endpoints which ensure only specific data is allowed to be processed. This is done by having requestDTOs which have annotated fields with various requirements ensuring the receiving of only data that is allowed.