SECURITY REPORT

ULTRAS Application

Mitov,Lachezar L.G.

Contents

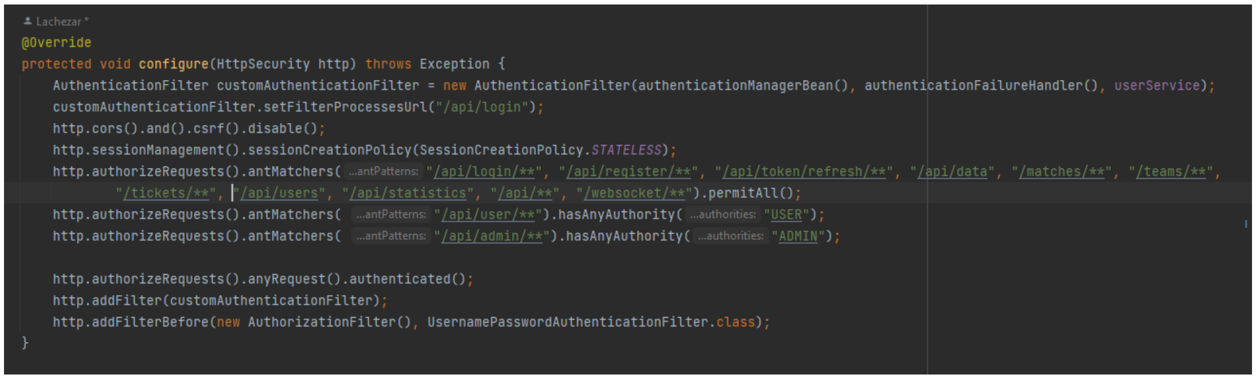
Introduction

The security report is a standard awareness document for developers and web application security. It represents a broad consensus about the OWASP Top 10 most critical security risks to web applications. In this report each risk will be explained, and an example will be given about what has been implemented in the application to prevent this kind of risk.

OWASP Top 10

A01: 2021- Broken Access Control

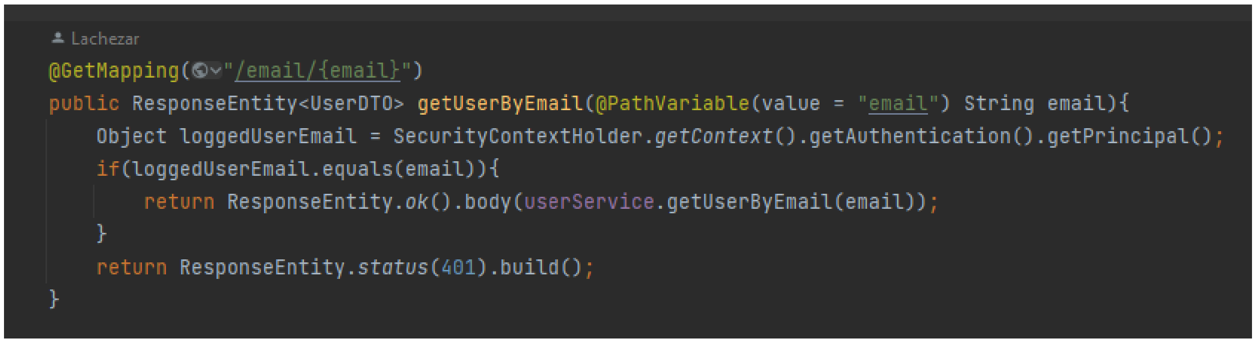
Broken Access Control failures lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. To combat that the application has restricted the access to API endpoints based on the role, which is contained in the JSON Web Token.



```
Lachezar *
@Override
protected void configure(HttpSecurity http) throws Exception {
    AuthenticationFilter customAuthenticationFilter = new AuthenticationFilter(authenticationManagerBean(), authenticationFailureHandler(), userService);
    customAuthenticationFilter.setFilterProcessesUrl("/api/login");
    http.cors().and().csrf().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests().antMatchers( ...antPatterns: "/api/login/**", "/api/register/**", "/api/token/refresh/**", "/api/data", "/matches/**", "/teams/**",
            "/tickets/**", "/api/users", "/api/statistics", "/api/**", "/websocket/**").permitAll();
    http.authorizeRequests().antMatchers( ...antPatterns: "/api/user/**").hasAnyAuthority( ...authorities: "USER");
    http.authorizeRequests().antMatchers( ...antPatterns: "/api/admin/**").hasAnyAuthority( ...authorities: "ADMIN");

    http.authorizeRequests().anyRequest().authenticated();
    http.addFilter(customAuthenticationFilter);
    http.addFilterBefore(new AuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

Moreover, the application has a check so that each user cannot access or modify personal information that is not theirs.



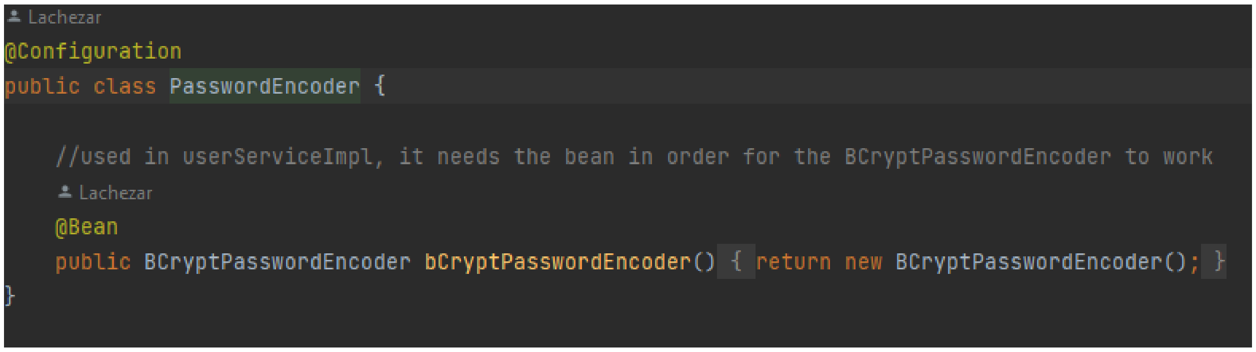
```
Lachezar
@GetMapping("/email/{email}")
public ResponseEntity<UserDTO> getUserByEmail(@PathVariable(value = "email") String email){
    Object loggedUserEmail = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    if(loggedUserEmail.equals(email)){
        return ResponseEntity.ok().body(userService.getUserByEmail(email));
    }
    return ResponseEntity.status(401).build();
}
```

By default, if the token doesn't contain the requested role for the API call, an error with status 403 is being send from the sever.

A02: 2021- Cryptographic Failures

This risk relates to the protection of sensitive data. The software solution has 2 types of sensitive data, which are passwords and user's personal information.

For passwords the application uses 'bcrypt' to hash the passwords before storing them in the database.



```
Lachezar
@Configuration
public class PasswordEncoder {

    //used in userServiceImpl, it needs the bean in order for the BCryptPasswordEncoder to work
    Lachezar
    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() { return new BCryptPasswordEncoder(); }
}
```

There is, however, a potential risk as the protocols are not encrypted. Therefore, the interaction between the server and the user interface is clearly visible and could be potentially used for malicious actions by an attacker who has remote access to the machine of the user.

A03: 2021- Injection

I suppose I could reply that since this project just uses JPA named queries, the danger is not entirely applicable to the application.

By not using string concatenation, we are reducing the risk of SQL injection. However, we are not using a WAF (Web application firewall), so there is still a potential risk. It's important to use all available resources to prevent SQL injection.

To separate the query from the supplied data and avoid malicious SQL searches in the parameters, I would use a prepared statement (parametrized query) if I needed a more precise query.

A04: 2021- Insecure Design

To prevent this risk the application has been thoroughly tested by unit and integration tests to validate that all critical flows are delivering the expected results.

A05: 2021- Security Misconfiguration

For this project, we are using SonarQube to ensure that the security configuration is effective and to identify any potential issues. This risk does not apply to the application because it does not contain any unnecessary features or components.

A06:2021-Vulnerable and Outdated Components

At the start of the project, I carefully selected libraries that are widely used and regularly updated. I thoroughly researched the libraries I am using for both the frontend and backend of the application as more information can be seen in the Design Document. To ensure security, only official sources from secure links were obtained.

To improve monitoring, we could set up automated patch management, which would simplify the process and ensure that everything is kept up to date.

A07:2021-Identification and Authentication Failures

In terms of Authentication & Authorization the application has

- JWT Authentication Login (access and refresh token)
- JWT Authorization on API calls
- Role Authorization
- Hashing password and storing them hashed in the database

The application isn't fully secured by the OWASP rule, and this could be achieved by further implementing:

- Weak password check
- Multi-factor authentication
- Credential recovery
- Log failed logging attempts/alert administrators

A08:2021-Software and Data Integrity Failures

This is prevented by ensuring that libraries and dependencies, npm and Gradle, are consuming trusted repositories. The components used are checked as well if they have any vulnerabilities.

Moreover, the CI/CD pipeline has proper segregation and configuration to ensure the integrity of the code flowing through the build and deploy processes.

Finally, there is review process for code and configuration to prevent malicious code or configuration be introduced into the software pipeline.

To lower the risk even more points can be added like:

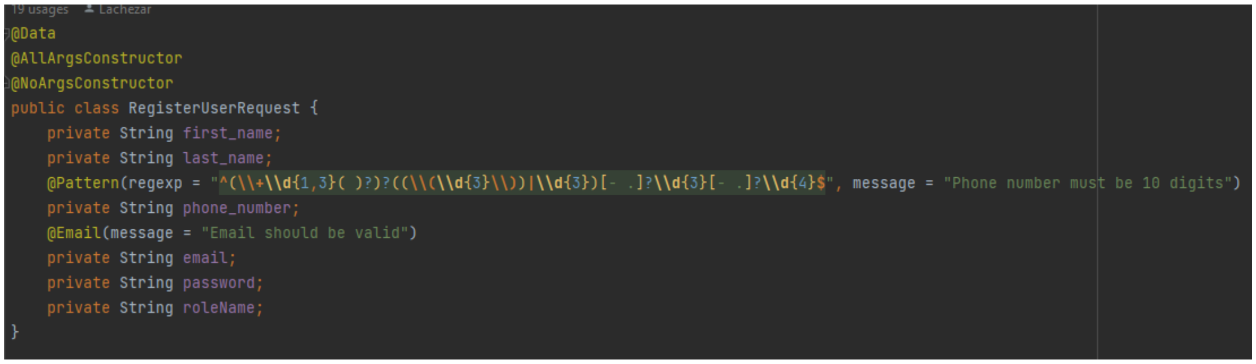
- Using digital signatures to verify the software or data
- Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients

A09:2021-Security Logging and Monitoring Failures

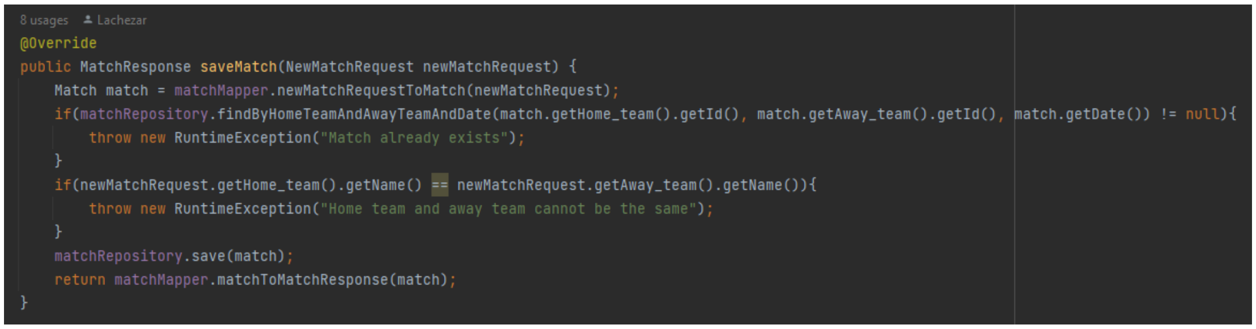
The messages and actions that are logged are carefully modified to avoid revealing any sensitive information. However, they are not stored in any form and are only meant to provide information to the developer or user. A potential improvement in this area would be to store and monitor the logged values so that developers can be alerted to potential attempts to breach the system.

A10:2021-Server-Side Request Forgery

To combat this risk the application validates all client-supplied input data. These are example how the software solution is combating this risk:



```
19 usages  Lachezar
@Data
@AllArgsConstructor
@NoArgsConstructor
public class RegisterUserRequest {
    private String first_name;
    private String last_name;
    @Pattern(regexp = "^(\\+\\d{1,3}( )?)?((\\(\\d{3}\\))|\\d{3})[- .]?\\d{3}[- .]?\\d{4}$", message = "Phone number must be 10 digits")
    private String phone_number;
    @Email(message = "Email should be valid")
    private String email;
    private String password;
    private String roleName;
}
```



```
8 usages  Lachezar
@Override
public MatchResponse saveMatch(NewMatchRequest newMatchRequest) {
    Match match = matchMapper.newMatchRequestToMatch(newMatchRequest);
    if(matchRepository.findByHomeTeamAndAwayTeamAndDate(match.getHome_team().getId(), match.getAway_team().getId(), match.getDate()) != null){
        throw new RuntimeException("Match already exists");
    }
    if(newMatchRequest.getHome_team().getName() == newMatchRequest.getAway_team().getName()){
        throw new RuntimeException("Home team and away team cannot be the same");
    }
    matchRepository.save(match);
    return matchMapper.matchToMatchResponse(match);
}
```