**OWASP Document**

**Vlad Cojocariu**

Contents

[1) Injection 2](#_Toc61812541)

[2) Broken Authentication 3](#_Toc61812542)

[3) Sensitive data exposure 3](#_Toc61812543)

[5) Broken Access Control 4](#_Toc61812544)

# Injection

Injection flaws, such as SQL injection should not occur when data is sent as part of a command or query. The attackers can trick the interpreter into executing unwanted commands or accessing data without authorization. The core of a code injection vulnerability is the lack of validation of the data used by the web application, which means that this vulnerability can be present on almost any type of technology.

Two things are important to keep in mind from this point of view. To separate the data from the web application logic and to implement settings to limit data exposure in case of successful attacks.

As for my application, the API that I created is safe so to speak because I am using ORM, having my repositories as JpaRepositories. The data is separated from the API and also there are no SQL queries that could be injected. Of course, the application is not fully secured, there are other ways to introduce SQL injection.

# Broken Authentication

When authentication and session management are implemented incorrectly, attackers are authorized to compromise passwords, keys, or to assume other users’ identities temporarily or permanently, in a worst case gaining complete control over the application . Usually this happens due to the usage of weak passwords or by allowing users to brute force username/password combination. Attackers could automate the tries of credentials and eventually get the right combination.

The authentication logic that I created is initializing a JWT Token every time a login is being made. So, without passing the right token to the backend no authentication is possible. All the passwords are protected, and no credentials are shipped or deployed, especially the admin users credentials. The biggest risk that my application has would be the brute force attempts because there is no limit for the login attempts and also the failure attempts are not logged, and the admins are not alerted in case of an attack would happen.

# Sensitive data exposure

A lot of web applications and APIs do not properly protect sensitive data. Nowadays this is one of the most requested features, because attackers may steal or modify weakly protected data to conduct credit card fraud, identity theft or other crimes. Credit card fraud is not exactly relevant for my application because there are not any functionalities involving payment. The sensitive data should have specific protection when stored in the database, or when is exchanged with the browser. There are two types of data: Stored data and Transmitted data (data that is transmitted internally between servers or to web browsers). Both should be protected.

One way to protect the data in transit is having an SSL (Secure Socket Layer) certificate which is the standard technology for having an encrypted link between server and browser. In my case the database that I am using is not allowing SSL due to free version. All sensitive data should be encrypted at rest.

Either way, the only sensitive data that my users would store is the account password which is protected by the backend and is only being passed during login. So, in my case there is almost no sensitive data and the biggest risk would still be the broken authentication.

1. XML External Entities (XXE)

XML External Entity attack is a type of attack against an application that parses XML input. For avoiding this kind of attacks, less complex data formats, such as JSON, should be used and avoiding serialization of sensitive data.

My application is not a case in which these attacks would be relevant because there is no use of XML files and especially no XML input.

1. Broken Access Control

Restrictions on what authenticated users are authorized to do are often not properly enforced. Attackers can use these flaws to access unauthorized functionalities and data like accessing other users accounts, view sensitive files, modify user data, change access rights and so on. For reducing the risks, the no longer used accounts should be deleted, the authorized rights should be carefully grated, unnecessary services should be deleted, and tests should be made for this risk. For preventing broken access control everybody should implement with a security-first philosophy, deny by default.

In my case only the authorized admin can make changes. I minimized the CORS usage and after logout the JWT Token is being invalidated on the server. There are more things that we could do to prevent broken access control, but I think, for my application it might be enough.

1. Security Misconfiguration

Security misconfiguration is a big issue in modern applications. This is a result of insecure default configurations, incomplete or ad hoc configurations and verbose error messages containing sensitive information. All operating systems, frameworks, libraries, and applications should be securely configured, but they must be patched/upgraded in a timely fashion, as the ways of fraud are evolving in time. Misconfiguration can happen at any level of an application, including network services, web server, database, storage.

My platform is minimal, without any unnecessary features, components. There are no features or frameworks installed and unused. I kept everything updated and there are no samples with a default configuration left from development. So, it should be enough secure from this point of view.

1. Cross Site Scripting (XSS)

Cross Site Scripting is a widespread vulnerability that affects many web applications. XSS attacks consist of injecting client-side scripts into a website and using the website as a propagation method.  It allows an attacker to inject content into a website and modify how it is displayed, forcing a victim’s browser to execute the code provided by the attacker while loading the page.

In my case a Stored XSS attack would be relevant. That means that in the case of an attack the API will store unwanted user input given by the attacker and that is considered high or critical risk. For that matter I am using the latest ReactJS which is automatically escaping XSS by design.

1. Insecure Deserialization

The process of serialization is converting objects to byte strings. Deserialization is converting byte strings to objects. So, attackers, by deserializing, changing, and serializing back the exchanged data could very well change his role attribute into admin and take control over the site. The best way to protect a web application from this type of risk is not to accept serialized objects from untrusted sources.

Unfortunately my application is not secure from this perspective, but it should log deserialization exceptions or failures, monitor deserialization processes and alerting if a user is doing it constantly.

1. Using Components with known Vulnerabilities

Components, such as libraries, frameworks run with the same privileges as the application. This could lead to data loss or even server takeover. Every product should receive updates regarding security, but some just get new versions and the old ones remain unprotected.

For this, I removed all unnecessary dependencies, I used only well know frameworks or components, from official sources and I got rid of the components that are not active.

1. Insufficient Logging and Monitoring

This is a very important topic regarding the maintenance of an application, but unfortunately in my case is not relevant because my app is not launched or available to the open world so monitoring and keeping logs is not really possible.

|  |  |  |  |
| --- | --- | --- | --- |
| Owasp top 10 |  |  | State |
|  |  |  |  |
| Injection |  |  | Secure(partially) |
| Broken Authentication |  |  | Secure(partially) |
| Sensitive Data Exposure |  |  | Secure |
| XML External Entities(XXE) |  |  | Not Relevant |
| Broken Access Control |  |  | Secure |
|  |  |  |  |
| Security Misconfigurations |  |  | Secure |
| Cross Site Scripting(XSS) |  |  | Secure(partially) |
| Insecure Deserialization |  |  | Unsecure |
| Using Components with  Known Vulnerabilities |  |  | Secure |
| Insufficient Logging and Monitoring |  |  | Not Relevant |
|  |  |  |  |