OWASP Report

Author: Tsvetislav Rangelov

Introduction

This report will show in detail how my software solution deals with the Open Web Application Security Project (OWASP) risks. It will also provide insights into the thought process of handling these risks and the corresponding implementation.

Two scales will be used for measurement. One is the Likelihood Scale, which indicates how likely a security risk is to be abused by an attacker. The other is the Impact and Risk Scale, which shows the severity of the risk being abused by an attacker.

Likelihood scale

Impact and risk scale

OWASP	Likelihood	Impact	Risk	Possible Actions	Planned
A01: Broken Access Control	Unlikely	Severe	Moderate	1.Remove metadata from the web roots.	Yes
A02: Cryptographic Failures	Likely	Severe	High	1.Check the password length and strength. 2.Use the most up-to-date hashing algorithm. 3.JWT token key should be generated	Partially

				cryptographically and randomly. 4. Hash password as a char[] to prevent sniffing on heap memory.	
A03: Injection	Unlikely	High	Moderate	1.Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection. 2.Automatically test all parameters, headers, url, json sent objects and data inputs	No
A04: Insecure Design	Unlikely	Moderate	Low	1. Ensure proper layer segregation depending on the exposure and protection needs. 2.Using thread modeling for critical authentication and access control.	No
A05: Security Misconfiguration	Likely	Low	Low	1.Ensure that my error messages do not contain information that can be used against the security of my application.	Partially

A06: Vulnerable and Outdated Components	Likely	Moderate	Moderate	1.Remove the features, components and dependencies that are not used. 2. Properly update the features, components and/or dependencies that are outdated.	Yes
A07: Identification and Authentication Failures	Unlikely	High	Moderate	1.Implementing authentication via an authentication provider (e.g Microsoft identity utilizing oAuth2.0) 2. Storing logs of all failed log in attempts.	No
A08:2021-Software and Data Integrity failures	Likely	Moderate	High	1.Checking vulnerability of components by using a security tool, such as SonarQube. 2.Ensure proper CI/CD configuration.	Partially
A09: 2021-Security logging and monitoring failures	Likely	Severe	High	1.Having organized logs that can be easily understood. 2.Encode the logged data to prevent injections or attacks against the monitoring system.	No

				3.Ensure that all error messages and thrown exceptions provide adequate data and are easily understandable.	
A10: 2021-Server-Side Request Forgery	Unlikely	High	Moderate	1.Having firewall policies or network access control rules to block any internet traffic that is not essential. 2.Having a list of allowed URLs, or a singular allowed URL.	No

OWASP Top 10 classification reasoning

A01: Broken Access Control

Broken access control indicates that a user has privileges they are not supposed to have. To give an example of my solution, a Standard user should not be able to have the authorization privileges of an Admin user, such as banning other users or censoring posts.

To prevent this, each controller in my API is secured with an aspect, which indicates what privileges the user should have in order to be able to interact with the given function. Additionally, in my SPA, I have implemented a custom react hook that checks if there is currently an authenticated user and if their credentials are valid, if not, they are redirected to the login page of my application and are unable to access any sensitive data until they don't log in.

A02: Cryptographic Failures

Cryptographic failures points out to flaws that are concerning data encryption and how it is stored in the solution. For passwords, a password hashing function, Argon2, is used to hash the user password and securely store it into the database. This is the most secure option as of writing this report, the reasoning for which I have provided In my research document. Something important to point out is that the password is being hashed as an array of

characters instead of a string. This is done to prevent any heap memory leaks from revealing the entire password altogether and is the default mode of operation for Argon2. The database is also configured to only store passwords that are of the length that Argon2 returns, so tampering with the hashing algorithm is mostly out of the question for any attacker.

On the other hand, a potential security risk is the JWT token key used for signing. The secret is currently being stored in the application properties file of my API which should not be the case. An option of taking care of this issue is to use environment variables to inject the key when required.

A03: Injection

Injection most frequently refers to SQL injections, which are a very common method of attack, especially for systems built using PHP. Thankfully, mine isn't, as I'm using Hibernate to manage all calls to and from the database. In essence, SQL injection is about an attacker modifying a query that's being sent to the database, which can for example, retrieve all records from a table instead of only a singular one, thus leaking all of the data.

Hibernate has built-in security features that prevent these kinds of attacks, such as parameterized queries and prepared statements, which escape unsafe characters. The overall threat is low to non-existent for my solution, however the severity is very high.

A04: Insecure Design

Insecure design is about the layering and architectural design of the entire solution.

To ensure the proper layering of my application, I include unit tests which test the flow of all of the data and the interaction between the different layers. My application is also separated into the aforementioned layers to ensure maintainability and scalability, which are – domain, persistence, controller and business.

A05: Security Misconfiguration

Security misconfiguration is about having, as the name states, an improperly defined security configuration, poor error handling etc.

The way I'm dealing with the risks of this issue is by defining a proper configuration class for each feature that needs one, for example a web security config. Additionally, while handling

exceptions, I avoid logging any data from the caught exception that can be used against my application, for example the stack trace.

A06: Vulnerable and Outdated Components

This security risk is about having dependencies and third-party libraries that can cause security issues, such as data leaks.

The way I'm making sure that this does not impact my solution is by utilizing the built in features of my package managers to validate and update my dependenices and libraries. For my SPA, this would be Node Package Manager(NPM) and for my API this would be Gradle. With Gradle, it is very convenient because you can notice an outdated package just by looking at the dependency file, as it is highlighted. With NPM, I routinely check all of my dependencies and validate if NPM finds any vulnerabilities.

A07: Identification and Authentication Failures

This security risk indicates that the user should be who they say they are based on their provided credentials.

A big problem that can arise here is, using weak and short passwords, an attacker may utilize GPU or side channel attacks against the hashed passwords that are stored in the database and crack them eventually, even though the hash function has built in salting that is not provided by the developer, which in this case is me. If that succeeds, they will be able to use the plain text password of the user and log in to their account seamlessly.

For my SPA, I am storing the authentication token generated from my API in-memory, which proves to be a tad-bit more secure than local storage. Storing tokens in browser local storage provides persistence across page refreshes and browser tabs, however if an attacker can achieve running JavaScript in the SPA using a cross-site scripting (XSS) attack, they can retrieve the token stored in local storage. A vulnerability leading to a successful XSS attack can be either in the SPA source code or in any third-party JavaScript code (such as bootstrap, JQuery, or Google Analytics) included in the SPA.

A very strong and security-increasing solution would be to implement oAuth2.0 authentication using an external identity provider, which can include but is not limited to Microsoft, Github or Google.

A08: 2021-Software and Data Integrity Failures

The software and data integrirty failures is about using plugins, libraries and resources that are from untrusted sources.

To handle this issue, using NPM and Gradle to verify that the dependency at hand has no security risks associated with it.

A09:2021- Security Logging and Monitoring Failures

Possible vulnerabilities for this security risk is that application logs are not being monitored for hostile activity and are stored only locally and also that the application is not logging or alerting any possible attacks.

A way to prevent both of these issues from arising is by ensuring that log data can be easily understood and is in a readable format. Additionally, encoding the logged data to prevent data leaks from sniffing tools is also a very good option.

As for any exceptions that can occur at run-time, ensuring that they are caught and handled properly by returning an accurate error message and notifying the user is the most optimal way of dealing with them. For this purpose, I have implemented a custom rest exception handler, that uses a HashMap to map the errors and their messages to keys, which are returned as responses if the request is unable to be processed by the API.

A10: 2021-Server-Side Request Forgery

Server side request forgery occurs if an application is fetching a resource without verifying the target URL. For my use case, I am not calling any external APIs, so this security risk is non-applicable.

If that were the case though, which most of the times is as any production-level application is likely to be communicating with external applications, a way to validate those URLs would need to be implemented to prevent the aforementioned security risk.

Ideally, having proper firewall policies or network access control rules to block any traffic that is not essential is a must. Additionally, providing a whitelist of URLs that can be accepted is also a convenient and simple way to handle this issue. This is what I am currently doing at my controller layer with the @allowedOrigins annotation at the class level. In this case, all of them point to localhost:3000, since this is the address that makes the requests to my API.

Conclusion

Application security is extremely highly regarded, as it should be, in the professional field of Software Engineering. Every corporation that handles sensitive data en masse utilizes various techniques and teams to handle security for them and what I have implemented in my personal project is just the foundation upon which all of these security systems are built. There are still many things I need to learn, but for now, I gave my best to ensure that my solution can be as secure as possible, both for my SPA and my API.