OWASP top 10

## Introduction

The OWASP top 10 lists the most critical risks when it comes to web application development. The list is updated on regular intervals. The most recent updated happened in 2021. This updated added 3 new threats to the list as well as reorder the importance of the remaining seven.

## OWASP top 10 – 2021

1. Broken Access Control
2. Cryptographic Failure
3. Injection Flaws
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery

## Code security verification tools

There are multiple code security verifications that can be added to the CI/CD pipeline for example:

* Static Application Security Analysis (SAST)
  + Static code analysis tool can be added to IDE for realtime notification of security issues. Limited scope hard to detect configuration issues, authentication problems, access control, insecure cryptography
* Dynamic Appication Security Analysis (DAST)
  + Requires a running application. Tests for common application attacks like SQL injection, cross-site forgery XSS attacks etc. Also tests for unexpected loads and malformed requests.
* Software Composition Analysis (SCA)
  + Checks for outdated or vulnerabilities in used libraries. The libraries code is not checked by the SAST in the vast majority of situations.
* Runtime Application Self-Protection (RASP)
  + Run on application start and protects against zero-day exploits, XSS attacks and email and messaging app attacks. Analyses the attack’s behaviour as to differentiate between legitimate requests and attacks to minimize false positives.

## Broken Access Control

Jumped to first place since the last OWASP update due to the increase in cloud computing and API usage. This issue is hard to detect with automated scanners.

Ways to prevent the issue:

* For any non-public resource deny the request by default.
  + **Change controller to always use my authentication scheme and specify when anonymous requests are accepted**
* Re-use access control mechanisms throughout the application. Minimal CORS usage.
  + Not working in an environment where we publish our application working with minimal CORS is not a viable option.
* Enforce record ownership
* Rate limiting of API to minimize results of automated attacks
* Usage of short lived JWT tokens.
  + Expiration time on tokens has been set to 2 hours for my application.

## Cryptographic Failure

* Never store sensitive data (creditcard details, health info for example) unless absolutely required. If required make sure the data is encrypted.
* Passwords should be stored with a salt/hashing function
  + Password are stored by salting and hashing using the Microsoft identity library which uses the PBKDF2 algorithm.
* No deprecated cryptographic functions are used within my application

## Injection

* Using a safe API that migrates to ORM tools.
* By not using lazy loading and only loading absolutely required data when a function is called preventing the exposure of large amount of data in case of SQL injection

## Insecure Design

* Write unit and integration tests for both front and backend
* Limit resource consumption by user or service so that a single user can’t make unlimited requests. Reminds of DDOS attacks

## Security Misconfiguration

* Platform should be as minimal as possible thus remove any unused framework/library/feature. The more libraries that are used the more potential openings for hackers
  + Thinking of the recent Log4J vulnerability

## Vulnerable and Outdated Components

* Remove anything within the application that isn’t used or is not/no longer necessary.
* Have a list of all dependencies and their version and update it continuously. Monitor whether any of the used dependencies show up on common vulnerability lists. Process can be automated with SCA tools.
* **Only ever download packages from official sources**

## Identification and Authentication failures

* **Never deploy application with default credentials included!**
* Usage of generic error messages for example in my application I use a error message that merely says the login/register request failed because one of the provided parameters caused an error. Returning an error message that specifies the password is wrong could be used by hackers to confirm a email/username is in use and they can try multiple passwords
* Password requirements to prevent users from using simply passwords like admin or password123

## Software and Data Integrity Failures

* Setting up a review process to prevent a single person from implementing malicious code or configuration. In the groups project we did this by forcing pull requests so that new code always had to be checked by 2 other developers.
* Using a tool such as OWASP Dependency check to verify the code doesn’t contain known vulnerabilities

## Security Logging and Monitoring Failures

* All login, access control and server-side input validation failures can be logged. This info should be stored in such a way that it can be analysed later.
* A development team should have a response plan in place in case a security breach is found. Again the recent Log4J vulnerability comes to mind where many companies where affected

## Serve Side Request Forgery (SSRF)

* Access control should block all but the essential traffic by default. In my application I setup my CORS to only allow requests from the localhost port of my frontend. **This should never allow ANY.**