NexusMeals: OWASP TOP 10 Security report

By Plamen Peev

Table of Contents

[Introduction 2](#_Toc169959855)

[OWASP Top 10 Report 2](#_Toc169959856)

[A1: Broken Access Control 2](#_Toc169959857)

[A2: Cryptographic Failures 2](#_Toc169959858)

[A3: Injection 3](#_Toc169959859)

[A4: Insecure Design 3](#_Toc169959860)

[A5: Security Misconfiguration 4](#_Toc169959861)

[A6: Vulnerable and Outdated Components 4](#_Toc169959862)

[A7: Identification and Authentication Failures 5](#_Toc169959863)

[A8: Software and Data Integrity Failures 5](#_Toc169959864)

[A9: Security Logging and Monitoring Failures 6](#_Toc169959865)

[A10: Server-Side Request Forgery (SSRF) 6](#_Toc169959866)

# Introduction

This document provides an analysis of OWASP TOP 10 security vulnerabilities, their potential impact, and the measures implemented to mitigate them within the NexusMeals application. Each identified vulnerability is assessed for its risk level, with detailed elaborations on how these risks are managed or minimized. The goal is to ensure the robustness of the application's security posture by addressing common security flaws, implementing best practices, and continuously monitoring and updating the system to defend against potential threats. The following sections outline the specific vulnerabilities, their impacts, and the current risk levels based on the preventive actions in place.

# OWASP Top 10 Report

## A1: Broken Access Control

**Description:** Broken access control vulnerabilities are a type of security flaw that permits unauthorized users to gain access to restricted resources.

**Impact:** Severe, can cause data breaches, unauthorized access to sensitive data or functionality.

**Preventive Actions:** Implement strong authentication and authorization mechanisms.

**Risk:** Low (Due to comprehensive validation, authentication, and authorization mechanisms in place.)

**Elaboration:**

NexusMeals API Gateway validates every client request before it redirects it to the appropriate microservice. Validation includes: headers, sender origin, request body format.

It also performs authentication verification for users attempting to access the gateway, and performs role-based authorization for actions that are only allowed for certain user groups.

In addition, a me-validation functionality is in place in every function that handles CRUD-ing personal data. The user sessions are kept in Redis, where each microservice can access the current user session based on the session id from the decrypred JWT coming from the client.

## A2: Cryptographic Failures

**Description:** Cryptographic failures occur when sensitive data is not properly protected through encryption or hashing.

**Impact:** High, can lead to data breaches and exposure of sensitive information.

**Preventive Actions:** Use strong encryption algorithms and proper key management practices.

**Risk:** Low (Use of bcrypt for password hashing and strong encryption practices.)

**Elaboration:**

Bcrypt is used to hash the user passwords upon registration, so the password is stored encrypted in the database. If there is a potential database leak, the password will not be immediately visible to the intruders.

## A3: Injection

**Description:** Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query.

**Impact:** Severe, can lead to data loss, corruption, or unauthorized access to data.

**Preventive Actions:** Use parameterized queries and validate input.

**Risk:** Low (Extensive input validation and use of parameterized queries.)

**Elaboration:**

Input format and structure is validated in multiple layers in my application with the help of NejstJS Guards and Interceptors:

1. In the API gateway before the request is forwarded to services.
2. In each microservice when it receives a message via TCP or AMQP.

Parameterized Queries and Prepared Statements are supported by default from the ORM library that I use for database connections (typeorm). This eliminates the possibility of maliscious code injection to the SQL queries. For mongo, I use mongoose ORM, which also has those built-in preventative measures implemented.

## A4: Insecure Design

**Description:** Insecure design refers to flaws in the design phase that make applications susceptible to attacks.

**Impact:** High, can lead to security vulnerabilities being built into the application.

**Preventive Actions:** Incorporate security into the design phase and use secure design principles.

**Risk:** Low (Security best practices considered and validated during the design phase.)

**Elaboration:**

During the design phase, I took into consideration the best practices against security flaws in the architecture. The design was discussed on multiple occasions with teachers and were approved as secure.

## A5: Security Misconfiguration

**Description:** Security misconfiguration occurs when security settings are not defined, implemented, or maintained properly.

**Impact:** High, can expose the application to a variety of attacks.

**Preventive Actions:** Regularly update and patch systems, and review security settings.

**Risk:** Low (Proper error handling, environment variable management, and configuration practices.)

**Elaboration:**

The error interceptor in the API Gateway is responsible for filtering error messages before returning them to the client. Error handing does not provide sensible information about the application or any users. Only specific manually-thrown errors of a specific abstract type are exposed to the client. Any error that does not belong to that abstract type is being logged only server-side, and an Internal Server Error with no specific message is sent to the client.

In addition, environment variables and other configurations are not exposed in the GitHub repository – any sensitive data is loaded from a GitHub secret. In the case of environment variables (which might contain usernames and passwords for database and message broker connections) for qualification and production, the variables are securely copied form a github secret to a Kubernetes configmap before applying the other manifests.

## A6: Vulnerable and Outdated Components

**Description:** Using components with known vulnerabilities can compromise the security of the entire application.

**Impact:** High, can lead to exploitation of known vulnerabilities.

**Preventive Actions:** Regularly update and patch all components and dependencies.

**Risk:** Low (Regular updates and use of tools to monitor vulnerabilities.)

**Elaboration:**

NPM is the package manager that I use. I always update my dependencies whenever advised. In addition, I have GitGuardian and Quodana which also produce warnings on packages with vulnerabilities/outdated versions.

## A7: Identification and Authentication Failures

**Description:** Failures in authentication and session management can allow attackers to compromise passwords, keys, or session tokens.

**Impact:** Severe, can lead to unauthorized access to user accounts.

**Preventive Actions:** Implement multi-factor authentication and secure session management.

**Risk:** Low (Strong password policies, token expiration, and session management.)

**Elaboration:**

* Brute force attacks are minimized by enforcing users to have strong passwords matching the following regex: /^(?!.\*[\s])(?=.\*[A-Z])(?=.\*[.!@#$\*])(?=.\*[0-9])(?=.\*[a-z].\*[a-z].\*[a-z]).{8,}$/m
* JWT that contains the session id is kept in sync with the server-side session management.
* Access token expire in 1 hour.
* Refresh token + Redis session expire in 1 week.
* Both tokens + the redis session are updated upon refreshing request.
* Upon authentication/authorization, API gateway always checks whether the user related to the session id still exists.

## A8: Software and Data Integrity Failures

**Description:** Software and data integrity failures occur when code and data are not protected from integrity violations.

**Impact:** High, can result in data tampering or malicious code execution.

**Preventive Actions:** Use digital signatures and integrity checks.

**Risk:** Low (Use of verified dependencies and secure CI/CD pipeline.)

**Elaboration:**

To prevent failures in my Node.js projects, I ensured that NPM used only verified dependencies with no associated risks. Additionally, I properly configured my GitHub Actions CI/CD pipeline to further enhance code quality and security.

## A9: Security Logging and Monitoring Failures

**Description:** Inadequate logging and monitoring can allow attackers to go undetected.

**Impact:** High, can delay the detection of and response to security breaches.

**Preventive Actions:** Implement comprehensive logging and monitoring.

**Risk:** Low (Comprehensive logging and monitoring with Datadog.)

**Elaboration:**

All system logs are securely stored in my Datadog project. In addition, I’ve enabled multiple Datadog features for tracking potential breaches and problems. Custom error messages are thrown for inadequate/ authentication or authorization issues which I can easily track and explore in Datadog.

## A10: Server-Side Request Forgery (SSRF)

**Description:** SSRF vulnerabilities occur when an attacker can make requests to unintended locations via a vulnerable server.

**Impact:** High, can lead to unauthorized access to internal systems.

**Preventive Actions:** Validate and sanitize user inputs and restrict outbound network access.

**Risk:** Moderate-High (While moderate protection measures are in place, SSRF can be complex to fully mitigate without stringent outbound network controls.)

**Elaboration:**

Typically, SSRF involves implementing firewall policies or network access control rules to block non-essential internet traffic. In my application, I have implemented moderate protection measures, such as sanitizing and validating all client-supplied input data and preventing clients from making arbitrary HTTP redirects.