## OWASP Top 10

## 2021

|  |  |
| --- | --- |
| Author | Ivan Koitchev |
| Project name | Video Game Store |
| Last update | 12.19.2021 |
| OWASP version | 2021 |

Table of Contents

**Introduction2**

**OWASP 2017 vs 2021**

**OWASP top 103**

Broken Access Control

Cryptographic failures

Injection**4**

Insecure Design

Security Misconfiguration**5**

Vulnerable and Outdated Components**6**

Identification and Authentication Failures**7**

Software and Data Integrity Failures**8**

Security Logging and Monitoring Failures**9**

Server-Side Request Forgery (SSRF)**10**

**Introduction**

OWASP (Open Web Application Security Project) is an international organization that focuses on developing applications that can be trusted. The OWASP foundation has created a list with the most common security mistakes that occur, and which could have been prevented. In this document I will address how each of the top 10 most important risks are handled (or not) by my application.

**OWASP 2017 vs 2021**

There are three new categories, four categories with naming and scoping changes, and some consolidation in the Top 10 for 2021. In my report I will be following the list from 2021. By following the newest convention my application will be less exposed to security threats.

**OWASP Top 10**

# **1. Broken Access Control**

**Description**

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits.

**Relevance and risks**

Broken access control is addressed in my application by restricting access to all endpoints by default and granting access to the user to the ones they need. Access to any information or editing any resource is granted only after the application has made the necessary verification that the user should receive those privileges.

# **2. Cryptographic Failures**

**Description**

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS).

**Relevance and risks**

User credentials that are sent to and from the frontend are stored in a JWT, which can be read only in the backend. This ensures the protection of the user’s credentials. Other than that, the application does not store any personal information like credit card or personal information, because it is not required for the app functionality.

# **3. Injection**

**Description**

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection.

An application is vulnerable to attack when:

* User-supplied data is not validated, filtered, or sanitized by the application.
* Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
* Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
* Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

**Relevance and risks**

The database CRUD operations are handled by the Object Relational Mapping of JPA repository. It ensures that no unvalidated parameters get passed into the SQL queries. Therefore, the application is not at risk from SQL injections.

# **4. Insecure Design**

**Description**

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks

**Relevance and risks**

Secure design was not taken into consideration at the start of the project. Some of the criteria such as *unit tests* and *Limit resource consumption by user* are present, but overall the application does not fully deal with the risk of an insecure design.

# **5. Security Misconfiguration**

**Description**

The application might be vulnerable if the application is:

* Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
* Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
* Default accounts and their passwords are still enabled and unchanged.
* Error handling reveals stack traces or other overly informative error messages to users.
* For upgraded systems, the latest security features are disabled or not configured securely.
* The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
* The server does not send security headers or directives, or they are not set to secure values.
* The software is out of date or vulnerable

**Relevance and risks**

The application deals with this risk only partially. All the dependencies and component used in the implementation are up-to-date and their most recent or stable version is used. There are not default accounts available. Spring security is used for authentication.

Possible risk is caused by error handling, as it might reveal stack traces/overly informative messages.

# **6. Vulnerable and Outdated Components**

**Description**

You are likely vulnerable:

* If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
* If the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
* If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
* If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.
* If software developers do not test the compatibility of updated, upgraded, or patched libraries.
* If you do not secure the components’ configurations.

**Relevance and risks**

Components and dependencies used in the application are mostly the latest stable version. However, in the frontend app there are currently some deprecated packages that are to be updated.

Patching is for the most part not applicable, as the application is still in development.

There are no integration tests implemented yet.

No components currently have specific configurations added to them.

# **7. Identification and Authentication Failures**

**Description**

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

* Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
* Permits brute force or other automated attacks.
* Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
* Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
* Uses plain text, encrypted, or weakly hashed passwords data.
* Has missing or ineffective multi-factor authentication.
* Exposes session identifier in the URL.
* Reuse session identifier after successful login.
* Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

**Relevance and risks**

The application is currently exposed to credential stuffing, brute force, and weak/default credential attacks. It is in the scope of the project to fix those issues, but it’s not yet implemented. There is not multi-factor authentication.

The app, however, is not exposed to “knowledge-based” credential recovery methods, as the forgotten password functionality is intended to be using email verification, rather than a *secret question.* The password is stored as hash in the database. The session token is deleted upon logout and also expires after a short amount of time by default.

# **8. Software and Data Integrity Failures**

**Description**

Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.

**Relevance and risks**

The application does not use any dependencies from untrusted sources. Moreover, it does not implement any automatic update functionality. Therefore, this type of exploit does not pose threat to my system.

# **9. Security Logging and Monitoring Failures**

**Description**

This category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:

* Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
* Warnings and errors generate no, inadequate, or unclear log messages.
* Logs of applications and APIs are not monitored for suspicious activity.
* Logs are only stored locally.
* Appropriate alerting thresholds and response escalation processes are not in place or effective.
* Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
* The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

**Relevance and risks**

Potential risk. Logging is done only at a few parts of the application. Most endpoints return a response entity indicating whether the request was successful or not.

# 

# **10. Server-Side Request Forgery**

**Description**

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

**Relevance and risks**

# This threat is not relevant to my application, as it does not process any external URL resources.