**Security Report**

***Gamehub***

|  |
| --- |
| **Date : 30th of May 2024** |
| **Version : 1.0** |
| **State : In progress** |
| **Author : Catalin Mihai Popoiu** |

Contents

Introduction…………………………………………………………………………………………………….………….3

[Injection (A1): 3](#_Toc154164924)

[Broken Authentication (A2): 4](#_Toc154164928)

[Sensitive Data Exposure (A3): 5](#_Toc154164935)

[XML External Entities (XXE) (A4): 5](#_Toc154164937)

[Broken Access Control (A5): 5](#_Toc154164938)

[Security Misconfigurations (A6): 6](#_Toc154164941)

[Cross-Site Scripting (XSS) (A7): 6](#_Toc154164942)

[Insecure Deserialization (A8): 6](#_Toc154164943)

[Using Components with Known Vulnerabilities (A9): 6](#_Toc154164944)

[Insufficient Logging and Monitoring (A10): 6](#_Toc154164945)

Conclusion............................................................................................................................7

| **Vulnerability** | **Likelihood** | **Impact** | **Risk** |
| --- | --- | --- | --- |
| Injection | Moderate | High | High |
| Broken Authentication | Low | High | Moderate |
| Sensitive Data Exposure | Low | High | Moderate |
| XML External Entities | Low | Moderate | Low |
| Broken Access Control | Moderate | High | High |
| Security Misconfiguration | Moderate | High | High |
| Cross-Site Scripting (XSS) | Moderate | High | High |
| Insecure Deserialization | Low | High | Moderate |
| Using Components with Known Vulnerabilities | Moderate | High | High |
| Insufficient Logging and Monitoring | Low | High | Moderate |

**Introduction**

The security report for GameHub follows the 10 OWASP principles to ensure the application is secure and robust against various security threats. The OWASP principles provide guidelines for best practices in application security, helping to identify and mitigate potential vulnerabilities**.**

**Injection (A1):**SQL injection may result from incorrect database query usage and a lack of input validation. For example, attackers may manipulate input to execute arbitrary SQL instructions if user inputs are concatenated into SQL queries without sufficient sanitization.

In comparison to performing raw SQL queries, the risk of injection is much decreased by utilizing a JPA (Java Persistence API) repository. Because the query structure and user input are kept separate, object-relational mapping (ORM) and specified query methods are used in JPA repositories to prevent SQL injection.

**Predefined Query Methods: (implemented)**

It is possible to specify query methods in JPA repositories by using method names, and the underlying JPA implementation will automatically generate the SQL queries that correspond to those methods. This methodological approach reduces the possibility of injection vulnerabilities because the framework converts method names with secure SQL queries with ease.

**Named Queries: (implemented)**Named queries are used in the program, this functionality is made possible via XML configurations or annotations. By guaranteeing that queries are predefined and go through extensive validation, named queries provide a centralized way to declare queries and hence reduce the danger of injection.

**Query Parameters: (partially implemented)**

The program makes use of named queries, a function made possible by XML configurations or annotations. Named queries provide a centralized way to declare inquiries, hence reducing the risk of injection by guaranteeing that queries are predefined and validated thoroughly.

**Broken Authentication (A2):**

Vulnerabilities may be caused by hard-coded credentials, ineffective session management, or lax password rules. For instance, user accounts may be compromised if passwords are kept in clear text or if inadequate encryption techniques are employed.

Because access tokens (JWT) are used effectively and passwords are securely salted, there is a far lower chance of authentication failure. For a thorough security posture, it is necessary to consider a larger authentication and session management context.

**Password Hashing: (implemented)**

To prevent brute-force and rainbow table attacks, password hashing is done using robust, adaptive hashing algorithms like BCrypt.

**JWT Implementation: (implemented)**

JSON Web Tokens (JWT) are implemented in accordance with industry practices, which include secure signature, specified expiration dates, and careful server-side verification to thwart tampering.

**Session Management: (partly implemented)**Secure session management procedures are included, such as session regeneration after login and session timeouts (without using secure cookie properties, such as HttpOnly, Secure).

**Secure Token Storage: (partly implemented)**Session storage is a secure place to keep access tokens. To reduce the possibility of cross-site scripting (XSS) attacks compromising token integrity, caution is used.

**Secure Communication: (not implemented)**

Make sure that secure channels (HTTPS) are used for all client-server communication, particularly during authentication. In doing so, man-in-the-middle attacks are less likely.

**Password Reset Mechanism: (implemented)**

If possible, employ multi-factor authentication when implementing a secure password reset procedure to assist users in safely recovering access to their accounts.

**Sensitive Data Exposure (A3):**

Data exposure may result from storing sensitive data (credit card numbers, for example) without using the appropriate encryption or from disclosing sensitive data via unreliable APIs. Sensitive data could be accessed by unauthorized parties due to inadequate protection measures.

The main cause of the Sensitive Data Exposure risk is improper handling or inadequate security of private or personally identifiable information (PII). The risks associated with sensitive data exposure are naturally reduced if the application does not handle sensitive data.

**Security by Design: (partly implemented by default)**It’s a good idea to include security measures in the design and development process of your application, even if there isn’t any sensitive data. This creates a solid security base for any features or updates that may come later.

**XML External Entities (XXE) (A4):**

An attacker may use improperly validated XML input that a program parses to include foreign entities, exposing confidential information or causing a denial-of-service attack. This might occur if the application makes use of unreliable XML processing setups or libraries.

Because the application does not use XML and does not process XML input, there is a far lower chance of XXE attacks.

The specific danger related to XXE is not relevant is the program does not process XML.

**Broken Access Control (A5):**

Unauthorized users may be able to access restricted capabilities due to inconsistent access control checks. For instance, users may be able to access privileged actions without authorization if access controls are not correctly implemented on the server side.

**Role-Based Access Control (RBAC): (implemented)**

Clear roles and corresponding permissions have been defined through the implementation of role-based access control. To provide safe and reliable access management, the program strictly enforces these restrictions at every layer.

**Least Privilege Principle: (partly implemented)**

The least privilege concept has been implemented, ensuring that roles and users have access to only the minimal amount of information required to perform their assigned duties. Roles that are unduly liberal and could provide needless access have been avoided.

**Security Misconfigurations (A6):**

Security flaws could be revealed by improperly configured server settings, permissions, or superfluous services. This could happen if default configurations for security settings are left in place due to improper configurations.

**Cross-Site Scripting (XSS) (A7):**

XSS vulnerabilities may result from a lack of output encoding and input validation. If user inputs aren’t cleaned up before being shown, hackers could insert dangerous scripts that run in the browsers of other people.

**Insecure Deserialization (A8):**

An attacker may alter serialized data to run arbitrary code if an application accepts serialized objects from untrusted sources without conducting the necessary validation. If the program doesn’t secure and validate the deserialization operations, this might occur.

**Using Components with Known Vulnerabilities (A9):**

The application may be vulnerable to known vulnerabilities in out-of-date libraries, frameworks, or dependencies if it uses them without receiving regular updates.

**Insufficient Logging and Monitoring (A10):**

Delays or inefficiencies in incident response might be caused by inadequate logging and monitoring procedures. Security incidents could go unnoticed if the program doesn’t use enough logging techniques or doesn’t keep an eye on logs for questionable activity.

**Conclusion**

Strong password hashing, JWT-based access control, and role-based access management are just a few of the security measures that have been put in place to protect the application. These steps, which are in line with the OWASP Top Ten best practices, successfully reduce possible hazards.  
Although the security posture as it stands now is praiseworthy, constant watchfulness is essential. The robustness of the program is enhanced by frequent audits, secure coding techniques, and ongoing monitoring. Keeping up this proactive strategy guarantees that the application is secure even in the face of changing threats.

Finally, the security measures that have been put in place offer a strong base. Nonetheless, maintaining long-term integrity and resistance to threats requires a dedication to constant security awareness and advancements.