

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | SpendSence | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | OWASP Report |  | | | |
|  | | | | June, 2025—Viktoria Todorova—First Draft |  | | | |
|  | | |  | | |  | | |

# Introduction

This report is a security review of a web application developed for a university project - SpendSense. The app is made up of three separate microservices that are built using Spring Boot. Two of these services use MySQL databases, and they interact with the database using something called JPA, which helps manage data more easily.

The app uses RabbitMQ to send messages between the different microservices, which helps them work together without getting in each other’s way. There’s also an API gateway called Traefik that acts like a traffic manager—it makes sure each user request goes to the right service safely.

For login and security, the app uses OAuth2, with a tool called Keycloak to manage users and their permissions. All communication happens over HTTPS, and the app has a real TLS certificate to make sure data is secure while it travels over the internet.

To keep an eye on how the app is running, I added Prometheus to collect system data and Grafana to show that data in easy-to-read dashboards.

This report follows the OWASP Top 10 (2021) list, which covers the most common and serious web security issues. The goal is to check if any of these issues exist in the app and give suggestions on how to fix or prevent them.

# Methodologies

The security assessment of the application mainly relied on static code analysis through manual inspection. This means carefully reviewing the source code to look for any security weaknesses, such as improper input handling or unsafe configurations.

In addition to manual checks, the project utilized Snyk, an automated tool that scans the code and its dependencies for known vulnerabilities. Snyk helps identify outdated or vulnerable libraries and suggests fixes.

For future improvements, SonarQube is planned to be used. SonarQube will provide continuous code quality and security checks, helping to catch issues early during development.

Together, these methods aim to cover both the code’s internal logic and external risks from third-party components.

# Vulnerability Assessment

#### Broken Access Control

Broken Access Control occurs when users can access features or data they shouldn’t have permission for. This can lead to data leaks or unauthorized actions.

The application currently supports only one user role for regular users. The only administrative access is within Keycloak, where admins manage realms, clients, and users. Since Keycloak handles this admin interface separately, the security depends on how well Keycloak itself is protected.

As part of this assessment, the internal protection of Keycloak’s admin interface was not fully tested, so the strength of admin access controls remains uncertain.

Risk Level: **Medium** (due to uncertainty about admin protection)

Recommendations:

* Verify that Keycloak’s admin console is secured with strong authentication and limited access.
* Consider enabling multi-factor authentication (MFA) for admin accounts.
* Monitor admin login activity and audit Keycloak access logs regularly.
* As the application evolves to include more roles, implement proper role-based access control checks within the app itself.

##### Cryptographic Failures

Cryptographic Failures occur when sensitive data is not properly protected during storage or transmission. This includes weak encryption, lack of encryption, or exposure of sensitive information.

The application uses HTTPS secured with a valid TLS certificate, ensuring that data transmitted over the network is encrypted and protected. User credentials, including passwords and emails, are managed and stored securely within Keycloak’s database, not within the application’s own databases.

The two MySQL databases used by the application do not currently encrypt data at rest. However, they do not store sensitive user information like passwords.

Risk Level: **Low to Medium**

Recommendations:

* Continue to ensure that all sensitive user data remains within Keycloak, which handles encryption appropriately.
* Consider implementing encryption at rest for the MySQL databases to protect any sensitive or critical application data they hold.

##### Injection

Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query, which can allow attackers to execute unintended commands or access data without authorization.

The application uses Spring Data JPA for database interactions, which by default uses parameterized queries and helps prevent SQL injection. No custom or raw SQL queries were found that might introduce injection risks.

However, input validation is not explicitly implemented at the API gateway or within the services, which could increase the risk of injection if new features are added in the future.

Risk Level: **Low**

Recommendations:

* Continue using parameterized queries for all database access to protect against SQL injection.
* Implement input validation and sanitization in services to reduce risks from malicious user inputs.
* Consider adding input filtering or validation middleware at the API gateway layer.

##### Insecure Design

Insecure Design refers to security weaknesses caused by missing or flawed design patterns, which can lead to vulnerabilities even if the code is implemented correctly.

The application’s design uses RabbitMQ as a message bus to communicate between microservices. Currently, there is no message-level authentication or encryption, which means that messages could potentially be intercepted or tampered with if an attacker gains access to the message broker.

Additionally, while Keycloak handles authentication, some aspects such as token revocation or session management could be improved to strengthen overall security.

Risk Level: **Medium**

Recommendations:

* + Use encryption for sensitive messages or data in transit within the message bus.
  + Improve session and token management in Keycloak, such as implementing proper token revocation and expiration.
  + Regularly review the system design for new threats as the application evolves.

##### Security Misconfiguration

Security Misconfiguration happens when security settings are not defined properly or left at insecure defaults, which attackers can exploit to gain unauthorized access or information.

During the assessment, it was found that the Prometheus and Grafana monitoring dashboards were accessible without authentication. This could allow unauthorized users to view sensitive system metrics and potentially gain information useful for attacks.

Other components like Traefik and Keycloak appear to have proper configurations, but monitoring tools lack proper access restrictions.

Risk Level: **High**

Recommendations:

* Protect Prometheus and Grafana dashboards by enabling authentication and restricting access to authorized personnel only.
* Regularly review configurations for all components to ensure no default or unsafe settings remain.
* Keep all software up to date to reduce risks from known configuration vulnerabilities.

##### Vulnerable and Outdated Components

Using outdated or vulnerable software components can expose applications to known security flaws that attackers can exploit.

The microservices were originally built using an outdated base Docker image (openjdk 17), which was flagged by Snyk as containing a large number of vulnerabilities. The base image has since been updated to a newer version (openjdk 25), which addresses these security issues.

The MySQL connectors and other libraries were found to be up to date with no major vulnerabilities.

Risk Level: **Low**

Recommendations:

* Maintain regular dependency and base image scanning using tools like Snyk to catch vulnerabilities early.
* Keep base images and libraries updated to their latest secure versions.
* Automate vulnerability scanning in the build pipeline to ensure continuous security checks.

##### Identification and Authentication Failures

These failures occur when authentication mechanisms are weak, allowing attackers to compromise user accounts or bypass login controls.

The application relies on OAuth2 with Keycloak for authentication, which is a strong and widely used system. User credentials and sessions are managed by Keycloak, which supports secure token-based authentication.

However, no multi-factor authentication (MFA) is currently enabled for user accounts, including admin users in Keycloak. This increases the risk of account compromise if passwords are weak or leaked.

Risk Level: **Medium**

Recommendations:

* Enable Multi-Factor Authentication (MFA) for all user accounts, especially admin accounts in Keycloak.
* Enforce strong password policies to prevent weak passwords.
* Monitor authentication logs for suspicious login attempts.
* Implement session expiration and token revocation policies to reduce the risk of stolen tokens.

##### Software and Data Integrity Failures

This issue occurs when software updates, critical data, or configurations can be tampered with or aren’t properly verified, leading to potential compromise or unauthorized changes.

Currently, the application is deployed manually, and there are no automated checks like signed container images or checksum verification during deployment. This increases the risk of accidentally deploying compromised or altered software.

Data transmitted via RabbitMQ is not encrypted or signed, which might allow tampering or replay attacks if the message broker is compromised.

Risk Level: **Medium**

Recommendations:

* When moving to automated deployment with GitHub CI/CD pipelines, implement signed container images and verify signatures to ensure software integrity.
* Use checksums or hashes to validate critical files and configurations during deployment.
* Enable encryption and message signing in RabbitMQ to protect message integrity.
* Perform manual audits of deployment steps until automation is in place to catch any issues early.

##### Security Logging and Monitoring Failures

Failure to log and monitor security events can delay the detection of attacks, making it harder to respond to incidents quickly.

The application uses Prometheus and Grafana for monitoring, with alerts set up for technical issues like microservice downtime. However, security-related events are not clearly logged or monitored, and there is no alerting system for suspicious activities or potential security breaches.

Additionally, Prometheus and Grafana dashboards were found to be accessible without authentication, increasing exposure.

Risk Level: **High**

Recommendations:

* Secure Prometheus and Grafana dashboards with authentication and restrict access to authorized users only.
* Implement centralized logging that includes security events from all microservices.
* Set up alerts specifically for security incidents, such as repeated failed login attempts or suspicious API calls.
* Regularly review security logs and alerts to detect and respond to threats promptly.

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

SSRF vulnerabilities happen when an attacker tricks the server into making unintended requests to internal or external systems, which can lead to data exposure or system compromise.

The application currently does not have explicit protections against SSRF attacks. Since it uses microservices and communicates over internal networks, SSRF could potentially allow attackers to access internal services or sensitive endpoints if input validation is not enforced.

No evidence of SSRF exploitation was found during testing, but this remains an area to watch.

Risk Level: **Medium**

Recommendations:

* Implement strict input validation and allowlisting for any URLs or endpoints the application needs to access.
* Restrict outbound requests from the application to only trusted services and networks.
* Monitor and log outgoing requests for suspicious activity.
* Review and harden firewall and network configurations to limit internal service exposure.

# Conclusion

This report evaluated the security posture of the microservice-based application using the OWASP Top 10 methodology. The application follows many modern architectural practices, including the use of OAuth2 authentication via Keycloak, HTTPS with a valid TLS certificate, and monitoring with Prometheus and Grafana.

While the application avoids several high-risk issues such as SQL injection or broken authentication, there are still areas for improvement. Key concerns include unauthenticated access to monitoring tools, lack of centralized security logging, outdated base images (now resolved), and missing protections around message queues and internal request validation.

As development continues and the application is integrated into a CI/CD pipeline, applying secure defaults and proactively monitoring components will be essential.

Summary Recommendations:

* Secure internal tools like Grafana and Prometheus.
* Add centralized logging and security-specific alerts.
* Enable multi-factor authentication in Keycloak.
* Protect RabbitMQ and review internal network exposure.
* Keep dependencies and base images up to date with automated scanning.

By addressing these issues early, the application will be more resilient to common web threats and better aligned with industry best practices.