

Escuela Colombiana de Ingeniería Julio Garavito

**Demo Company**  
**IT Security and Privacy**

**Vulnerabilities Management**

Students:

Laura Sofia Gil Chaves

Camilo Castaño Quintanilla

Teacher:

Ing. Daniel Vela

**Business Confidential**

October 30<sup>th</sup>, 2024

Project 01-11

Version 1.0

## Table of Contents

<b>Assessment Overview</b>	3
<b>Assessment Components</b>	3
<b>External Penetration Test</b>	3
<b>Scope</b>	4
<b>Scope Exclusions</b>	4
<b>Client Allowances</b>	4
<b>Executive Summary</b>	4
<b>Attack Summary</b>	4
<b>Security Weaknesses</b>	5
<b>SQL injection</b>	5
<b>Cross-Site Scripting (XSS)</b>	5
<b>Lack of Authentication and Access Control</b>	6
<b>Vulnerabilities in Dependencies</b>	6
<b>External Penetration Test Findings</b>	6
<b>Exploit Proof of Concept</b>	6
<b>Snyk</b>	6
<b>SonarCloud</b>	10

## Assessment Overview

From Thursday, October 24 to Wednesday, October 29, we integrated SonarCloud and Snyk into the project's code repository and performed an initial analysis to identify all vulnerabilities present, classifying them by severity and type. From these results, we began the remediation process, in which we applied code enhancements and fixes to eliminate security vulnerabilities, improve quality, and resolve issues related to insecure dependencies. For each finding, we document the specific steps taken and explain how they contribute to mitigating risks. At the end of the remediation process, we generate a detailed report that includes initial and final metrics, highlighting the improvements achieved in each area. In addition, we explain any vulnerabilities that could not be resolved, providing the reasons, as well as possible solutions or alternative mitigations to address these cases in the future.

1.Planning: We selected SonarCloud and Snyk as key tools for code assessment and configured both in the project repository. We also assigned roles and responsibilities within the team to cover every aspect of analysis, remediation and findings documentation.

2.Discovery: We performed an initial code analysis using SonarCloud and Snyk to identify security vulnerabilities, quality issues and insecure dependencies. Each finding was classified by severity (low, medium, high) to understand the level of risk and prioritize the remediation phase.

3.Attacking: We prioritize vulnerabilities according to their level of severity and impact on system security, addressing the highest risk ones first. We then applied fixes to the affected code and dependencies to mitigate or eliminate the vulnerabilities found. Each fix was followed by testing and further analysis to ensure that the fixes were effective and did not introduce new problems.

4.Reporting: Documentation of vulnerability management and possible mitigation options.

**Plan → Discovery → Attack → Report**

## Assessment Components

### External Penetration Test

In the Task Manager project, we used SonarCloud and Snyk tools to identify and remediate security vulnerabilities. First, we configured both tools in the TaskManager repository, which allowed us to perform thorough analyses of the source code and its external dependencies. SonarCloud helped us detect quality issues and potential vulnerabilities in the code, while

Snyk focused on dependencies, pinpointing outdated components or components with security risks. After identifying and classifying vulnerabilities, we prioritized their remediation based on severity and potential impact.

## Scope

Assessment	Details
External Penetration Test	<a href="https://github.com/LaaSofiaa/SPTI-Testing.git">https://github.com/LaaSofiaa/SPTI-Testing.git</a>

## Scope Exclusions

There will be no scope exclusions regarding vulnerabilities analysis, as all testing will be conducted in a secure and controlled laboratory environment.

## Client Allowances

The required permissions granted by us will allow full access to all necessary files and resources.

## Executive Summary

In the Task Manager project, we implemented SonarCloud and Snyk as essential tools to identify and manage vulnerabilities in the code and its external dependencies, allowing us to strengthen security and improve project quality. We configured SonarCloud to detect code quality issues and potential vulnerabilities, while Snyk was used to analyze dependencies, identifying outdated components or components with known security risks. During the analysis process, SonarCloud and Snyk detected several vulnerabilities that were classified into three severity levels: high, medium and low, prioritizing the correction of those with the highest impact. This included updating critical dependencies and optimizing insecure configurations that could be exploited. After corrections, we performed reanalysis to validate the effectiveness of the implemented measures, ensuring that no new problems were introduced into the process. In addition, we kept both tools integrated into the CI/CD pipeline, ensuring continuous monitoring and enabling early detection of vulnerabilities with every code change. The implementation of SonarCloud and Snyk not only significantly reduced the security risk in the Task Manager project, but also ensured a solid foundation for code quality and reliability, promoting proactive security management in the development of the project.

## Attack Summary

The following table describes how we gained internal network access, step by step:

Step	Action	Recommendation
1	Configure SonarCloud and Snyk in the repository.	Configure automatic scans on every commit/pull request to detect vulnerabilities in time and ensure continuous security monitoring.
2	Execute initial code and dependency analysis.	Perform a thorough analysis when integrating tools to get a complete picture of the project's security and quality issues.
3	Review vulnerability reports from both tools.	Classify the problems identified according to their severity (high, medium and low) and prioritize those with the highest risk for application security.
4	Classify vulnerabilities according to their severity.	Focus first on high severity vulnerabilities to minimize immediate risk, then address medium and low severity vulnerabilities.
5	Implement solutions for critical vulnerabilities.	Perform fixes on insecure configurations, update critical dependencies, and validate that remediations are effective through reanalysis.
6	Rework code and dependencies after fixes.	Ensure that fixes have resolved vulnerabilities without introducing new problems.
7	Maintain SonarCloud and Snyk in the CI/CD pipeline.	Continue to monitor on every code change to proactively detect and resolve new vulnerabilities.

## Security Weaknesses

The following is a description of the weaknesses that a system may have if it has not performed vulnerability management:

### SQL injection

SQL injection allows an attacker to execute malicious SQL queries through user input. To prevent this vulnerability, it is recommended to use prepared queries or ORMs (Object-Relational Mapping) that separate the data from the query logic. In addition, always validate the input data to ensure that only the expected types and formats are accepted.

### Cross-Site Scripting (XSS)

Cross-Site Scripting allows attackers to inject scripts into web pages viewed by other users. To mitigate this risk, it is crucial to validate and encrypt all user input. Implementing content

security policies (CSP) can also help limit the execution of unauthorized scripts in the user's browser.

## Lack of Authentication and Access Control

The absence of adequate authentication and access control mechanisms may allow unauthorized users to access sensitive resources. It is recommended to implement strong authentication, such as multi-factor authentication (MFA), and to apply least privilege principles, ensuring that users only have access to the resources required for their roles.

## Vulnerabilities in Dependencies

The use of outdated libraries or frameworks can introduce known vulnerabilities into the project. To address this problem, it is recommended to keep track of all dependencies and perform regular updates. Use tools such as Snyk to automatically scan dependencies for vulnerabilities and apply security patches.

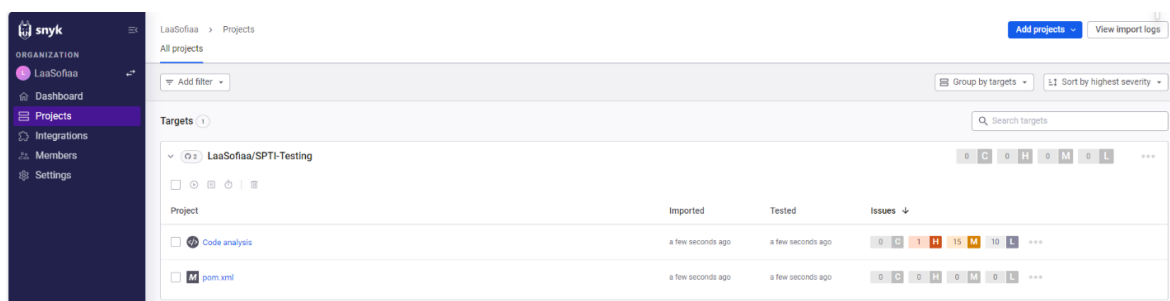
## External Penetration Test Findings

### External Penetration Test Findings

<b>Description:</b>	In the Task Manager project, we configured SonarCloud and Snyk to identify and fix security vulnerabilities. SonarCloud analyzed the source code, while Snyk focused on dependencies, detecting quality issues and obsolete components. After classifying the vulnerabilities, we prioritized their remediation according to their severity and potential impact.
<b>Impact:</b>	Critical
<b>System:</b>	Windows
<b>References:</b>	<a href="https://github.com/LaaSofiaa/SPTI-Testing.git">https://github.com/LaaSofiaa/SPTI-Testing.git</a>

## Exploit Proof of Concept

### Snyk



Currently, our security analysis with Snyk has revealed a total of 28 vulnerabilities in our project, distributed as follows:

**2 high severity issues:** These critical vulnerabilities require priority attention. They include security issues such as Cross-Site Request Forgery (CSRF) and permissive CORS configurations that expose the application to possible external attacks. Our next action will be to enable CSRF protection and restrict domains in CORS policies.

**15 medium severity issues:** These intermediate level security issues include configurations and validations that need adjustments to avoid potential risks. We will focus on fixing these vulnerabilities after addressing the high severity ones.

**11 low severity issues:** These vulnerabilities are less urgent and are mostly related to practices in test code, such as the use of hardcoded credentials in backend testing. We plan to evaluate the relevance of these issues and define whether they will be mitigated or ignored, as they do not represent a risk in the production environment.


## HIGH PRIORITY



**Metrics detected before remediation:** Initially, Snyk detected a vulnerability in our code due to the use of a sensitive constant (such as a secret key) that was hardcoded directly in the source code. This vulnerability was classified as a high severity security issue, with a priority score of 754, as exposing sensitive data in this way significantly increases security risks.

**Remediation process:** To resolve this issue, we decided to replace the hardcoded value with an environment variable. We created a separate file to store this sensitive key and configured it so that the code takes the value from the environment instead of having it embedded in the code. This improves the security and flexibility of the code, as we can now change the key without modifying the source code.

**Metrics after remediation:** After implementing this fix, Snyk reanalyzed our code and confirmed that the vulnerability was resolved. The security metrics were updated, and the score disappeared, reflecting the improved security of our project. Thanks to this remediation, we have mitigated the risks of sensitive data exposure and strengthened the overall security of our code.


**Cross-Site Request Forgery (CSRF)**

SCORE  
**754**

SNYK CODE | [CWE-352](#)

```

33 |      * @throws Exception Si ocurre algún error al configurar la seguridad.
34 |      */
35 |      @Bean
36 |      public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
37 |          http
      
```

CSRF protection is disabled by `csrf`. This allows the attackers to execute requests on a user's behalf.

[src/main/java/edu/eci/cvds/task\\_back/Config/SecurityConfig.java](#) 1 step in 1 file

[Learn about this type of vulnerability and how to fix it](#)

**Metrics detected prior to remediation:** Cross-Site Request Forgery (CSRF) Snyk identified a vulnerability in the security configuration in which CSRF protection was disabled, which would allow attackers to execute requests on behalf of users without their consent. This vulnerability has a priority score of 754, classified as high severity.

**Remediation process:** For the CSRF issue, we enabled protection by enabling CSRF in the security settings, which reduces the risk of request forgery attacks in the context of our application.

**Metrics after remediation:** After implementing this fix, Snyk performed a new analysis and confirmed that both vulnerabilities were resolved, indicating an improvement in the security of our application. Thanks to these remediations, we now have a more secure and controlled environment, mitigating the risks of CSRF attacks.

## MEDIUM PRIORITY


**Origin Validation Error**

SCORE  
**600**

SNYK CODE | [CWE-942 + 1 MORE](#)

```

28 |      * Maneja la solicitud de registro de un nuevo usuario.
29 |      * @param request Contiene la información del usuario a registrar.
30 |      * @return Respuesta HTTP con el estado de la operación.
31 |      */
32 |      @CrossOrigin(origins = "**")
      
```

CORS policy `**` might be too permissive. This allows malicious code on other domains to communicate with the application, which is a security risk

[src/main/java/edu/eci/cvds/task\\_back/Controller/LoginController.java](#) 2 steps in 1 file

Ignore
Full details



**Origin Validation Error**

SNYK CODE | CWE-942 + 1 MORE

SCORE  
**600**

```

176 * Recupera el nombre de usuario basado en su ID.
177 * @param idUser El ID del usuario cuyo nombre se desea recuperar.
178 * @return El nombre del usuario o un mensaje de error si no se encuentra.
179 */
180 @CrossOrigin(origins = "**")

```

CORS policy "\*" might be too permissive. This allows malicious code on other domains to communicate with the application, which is a security risk

src/main/java/edu/eci/cvds/task\_back/Controller/UserController.java

2 steps in 1 file

Ignore

Full details

**Metrics detected prior to remediation:** During the security analysis, Snyk identified CORS vulnerabilities in the UserController and LoginController. In both controllers, the policy was configured permissively (`@CrossOrigin(origins = "**")`), allowing requests from any domain. This configuration represents a security risk, as it allows any domain to access the application, potentially facilitating the execution of malicious code from other sites.

**Remediation process:** To resolve this issue, we modified the CORS configuration in both controllers (UserController and LoginController), making the route more specific and allowing only authorized domains to access our application. This configuration adjustment reduced the exposure of the application and limited access to trusted sources only.

**Post-remediation results:** Snyk confirmed that the remediation was effective, eliminating 8 vulnerabilities related to the CORS configuration. This remediation increases application security by preventing unauthorized access and controlling the origins of incoming requests.

## LOW PRIORITY

**Use of Hardcoded Credentials**

SNYK CODE | CWE-798

SCORE  
**270**

```

119 @Test
120 public void testRegister_Success1() throws Exception {
121     // Preparar
122     RegisterRequest request = new RegisterRequest();
123     request.setUsername("user1");

```

Do not hardcode credentials in code.

src/test/java/edu/eci/cvds/task\_back/AuthServiceTest.java

1 step in 1 file

Ignored 6 days ago by

Laura Gil

Ignored path

\*

Type

Ignored permanently

Expires

Never

Reason

no es vulnerable

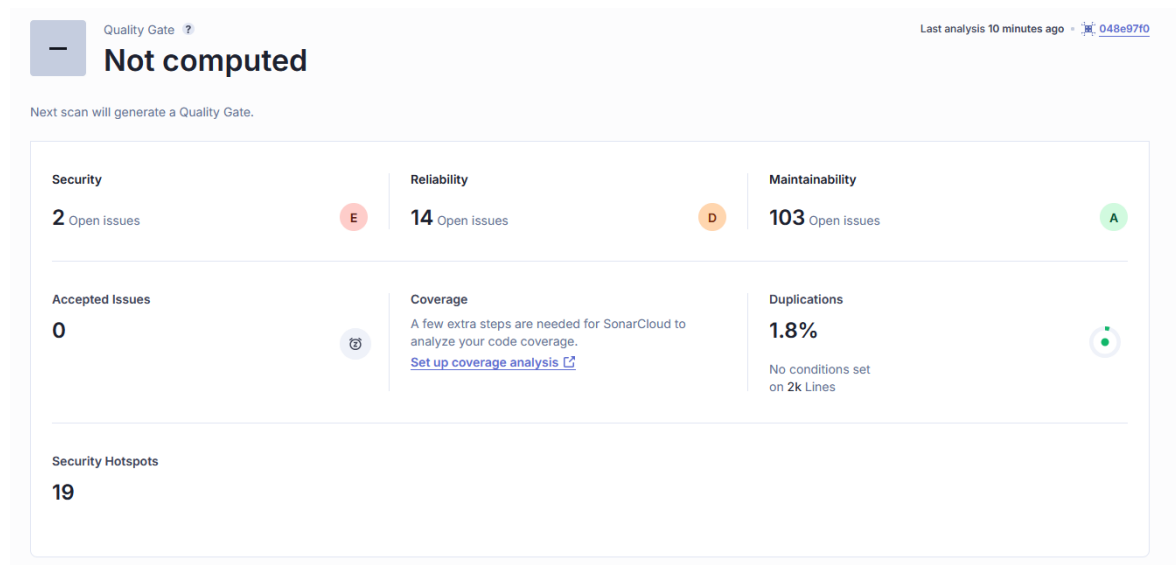
**Metrics detected prior to remediation:** Snyk identified a vulnerability of type CWE-798 in the AuthServiceTest.java file, where embedded credentials (user1) are used within unit tests.

The practice of statically embedding credentials in code can pose a security risk if not properly managed.

**Ignore decision:** In this case, we decided to ignore this vulnerability because the credentials are contained only in test files, which are used to test the backend and are not part of the production environment. Unit tests require these credentials to simulate specific scenarios without involving real risks in the development environment.

**Post-testing results:** Snyk allows to mark this type of vulnerabilities as “ignored” when they do not represent an active risk. In total, 11 vulnerabilities related to embedded credentials in tests were ignored, ensuring that these warnings do not affect the overall security score of our project.

## SonarCloud



In sonar cloud we can see two Security Hotspot issues and 117 of Maintainability

## Dependency Injection

It is recommended to fix dependency injection within a constructor instead of an external form.

```

import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.servlet.config.annotation.CorsRegistry;

import java.util.List;

@EnableWebSecurity
@Configuration
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final AuthenticationProvider authProvider;

    @Autowired
    public SecurityConfig(JwtAuthenticationFilter jwtAuthenticationFilter, AuthenticationProvider authProvider) {
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
        this.authProvider = authProvider;
    }

    /**
     * Configura la cadena de filtros de seguridad.
     * @param http Configuración de seguridad HTTP.
     * @return La cadena de filtros de seguridad configurada.
     * @throws Exception Si ocurre algún error al configurar la seguridad.
     */
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        ...
    }
}

```

**Consistency | Not conventional**

**Remove this field injection and use constructor injection instead.**

Field dependency injection should be avoided [java:S6813](#)

Software qualities impacted: **Maintainability** **Reliability**

☐ Open ☒ Not assigned ☐ Code Smell ☐ Major

**Where** **Why** **How** **Activity** **More info**

**Tags**  
No tags

**Line affected**  
L22

**Effort**  
5 min

**Introduced**  
8 days ago

[Open in IDE](#)

```

import org.springframework.web.cors.CorsConfigurationSource;

import java.util.List;

@EnableWebSecurity
@Configuration
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;
    @Autowired
    private AuthenticationProvider authProvider;
}

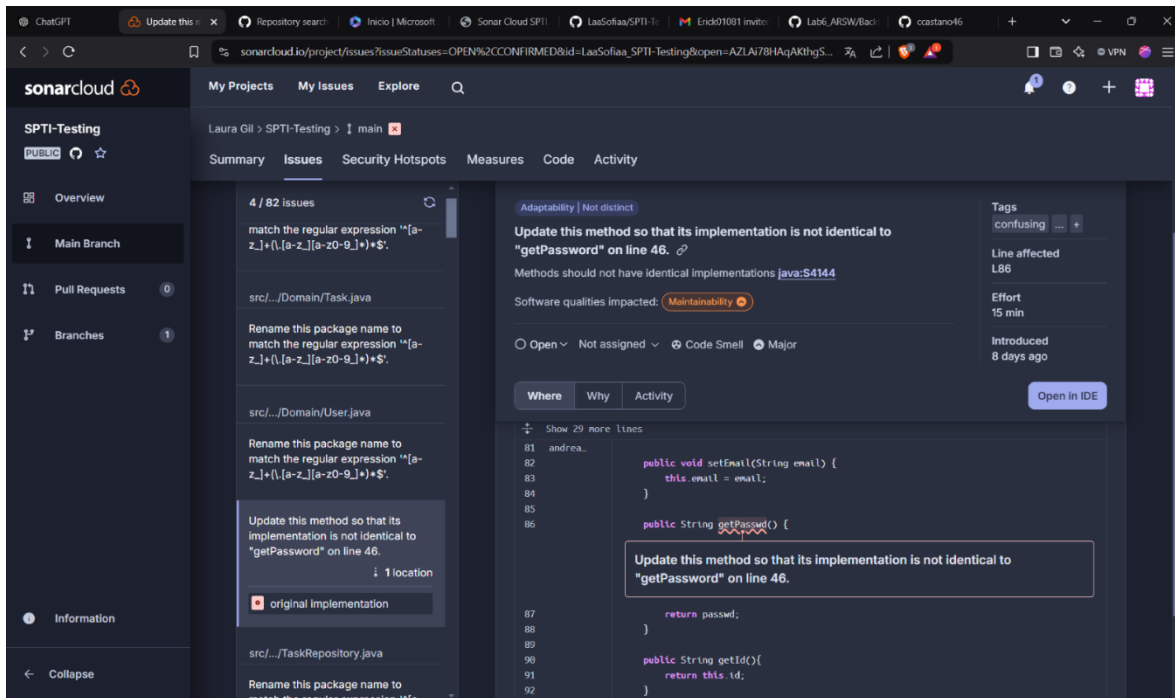
```

**Remove this field injection and use constructor injection instead.**

Because field injection is discouraged. It allows the creation of objects in an invalid state and makes testing more difficult. The dependencies are not explicit when instantiating a class that uses field injections. In addition, field injections are not compatible with final fields. Keeping dependencies immutable where possible makes the code easier to understand, easing development and maintenance.

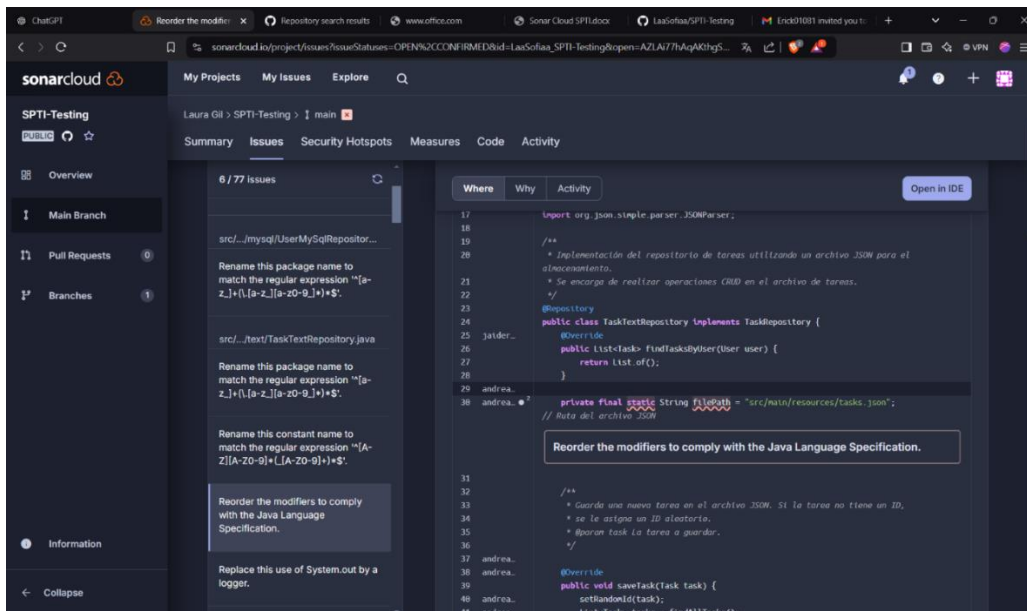
## GetPassword

In this case we have two methods of doing the same action



At this point we just erase one of the two methods

Recorder



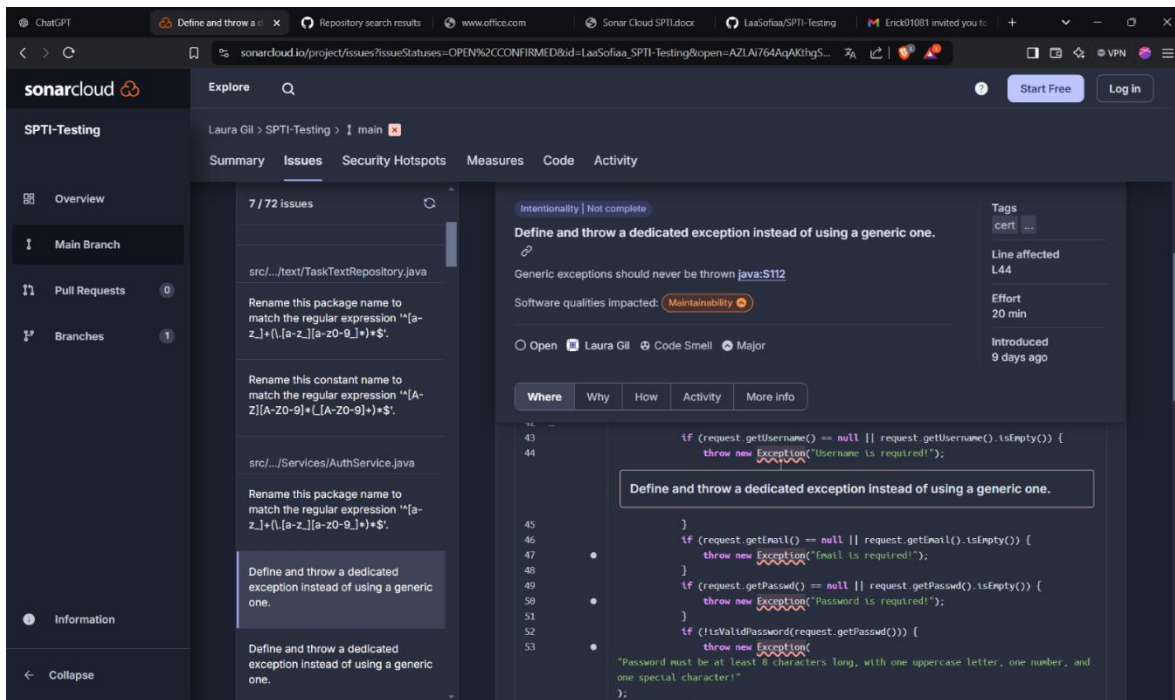
The Java Language Specification recommends listing modifiers in the following order:

- Annotations
- public

- protected
- private
- abstract
- static
- end
- transient
- volatile
- synchronized
- native
- default
- strictfp

Not following the convention this has no technical impact but will reduce the code's readability because most developers are used to the standard order.

## Exception



Throwing generic exceptions such as Error, RuntimeException, Throwable, and Exception will have a negative impact on any code trying to catch these exceptions.

From a consumer perspective, it is best practice to only catch the exceptions you intend to handle. Other exceptions should ideally be let to propagate up the stack trace so that they can be dealt with appropriately. When a generic exception is thrown, it forces consumers to catch exceptions they do not intend to handle, which they then have to re-throw.

Furthermore, when working with a generic type of exception, the only way to distinguish between multiple exceptions is to check their message, which is error-prone and difficult to maintain. Legitimate exceptions may be unintentionally silenced, and errors may be hidden.

For instance, when a Throwable is caught and not re-thrown, it may mask errors such as OutOfMemoryError and prevent the program from terminating gracefully.

When throwing an exception, it is therefore recommended to throw the most specific exception possible so that consumers can intentionally manage it.

**Recommendation:**

<b>Who:</b>	The system security team and developers of the programs.
<b>Vector:</b>	Remote.
<b>Action:</b>	<i>Item 1:</i> Schedules regular security audits and penetration tests to identify and remediate vulnerabilities before they can be exploited.  <i>Item 2:</i> Use multiple layers of security, such as firewalls, intrusion detection systems and multi-factor authentication (MFA).