UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Análisis y corrección de vulnerabilidades de un producto software con SonarQube

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Esparza Tortosa, Rafael

Tutor/a: Escobar Román, Santiago

CURSO ACADÉMICO: 2022/2023

Abstract

Un producto software debe pasar por diferentes fases durante su ciclo de vida. Este trabajo se centra en la fase de validación, añadiendo una nueva capa con el objetivo de entregar un producto libre de problemas de seguridad y que respete ciertas prácticas de codificación y despliegue seguros. La calidad del código afecta directamente a estas necesidades y su mejora ayuda a prevenir conflictos a medida que un proyecto aumenta de volumen y complejidad. SonarQube es una herramienta que nos permite analizar el código de un proyecto de forma automática, generando informes sobre los problemas de seguridad encontrados, entre los que se encuentran bugs, vulnerabilidades e incumplimiento de algunas buenas prácticas comúnmente utilizadas. En este trabajo se analiza con esta herramienta un producto software actualmente en producción con la intención de inspeccionar la información obtenida por estos informes, analizar la que resulte más relevante para el desarrollo de software seguro y buscar solución para el mayor número de vulnerabilidades que sean detectadas.

Glosario

FRONT-END parte de un sistema o aplicación web que los usuarios visualizan y con la que interactúan.

BACK-END parte de un sistema o aplicación web que se encarga de las comunicaciones con el servidor, del procesamiento y almacenamiento de datos.

REST siglas de Representational State Transfer, es un estilo arquitectónico utilizado en el desarrollo web.

DTO siglas de Data Transfer Object, son un tipo de objetos que sirven únicamente para transportar datos.

JPA es una especificación que indica cómo se debe realizar la persistencia de los objetos en programas Java.

HIBERNATE es una capa de persistencia objeto/relacional que permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos.

NPM siglas de Node Package Manager o manejador de paquetes de node, es la herramienta por defecto de JavaScript para la tarea de compartir e instalar paquetes.

FRAMEWORK es un entorno o marco de trabajo, un conjunto de prácticas, conceptos y criterios a seguir estandarizados.

ANGULAR framework de desarrollo web de código abierto y basado en TypeScript que permite crear aplicaciones web robustas y escalables.

DOCKER plataforma de software que permite a los desarrolladores crear, probar e implementar aplicaciones en contenedores de forma rápida.

EXPOSICIÓN son errores no inherentes al software, firmware, hardware o componente de servicio que lo ponen en riesgo de ser explotado, como configuraciones erróneas, puertos abiertos o credenciales débiles.

TABLA DE CONTENIDOS

1 INTRODUCCIÓN

El desarrollo de software seguro se ocupa de identificar y corregir vulnerabilidades y debilidades en el código, evitando así la explotación de estas vulnerabilidades, ayudando a garantizar la estabilidad y confiabilidad del software a lo largo del tiempo. Esto debería ser una prioridad fundamental en el desarrollo de un producto software, ya que en la actualidad está presente en todos los aspectos de nuestra vida. Desde transacciones financieras, servicios de salud, comunicaciones, gestión empresarial o control de producción entre otros.

Desarrollar software seguro no solo se refiere a la protección contra ataques maliciosos, sino también incluye la prevención de errores y fallas que podrían poner en riesgo la integridad, confidencialidad y disponibilidad de la información. Además de conseguir que los usuarios de estas aplicaciones preserven su privacidad.

Respetando los tres aspectos que se han nombrado anteriormente se consigue que el software garantice que la información sensible de los usuarios como pueden ser los datos médicos o financieros estén más libres de posibles violaciones de datos.

Esto radica en una mayor confianza por parte de los usuarios hacia los productos software que son utilizados en nuestro día a día. Dañar la confianza de los usuarios hacia un producto supone poner en riesgo la reputación de la empresa encargada de él. Además de evitar sanciones que pueden ser impuestas por no respetar el Reglamento General de Protección de Datos (GDPR) [1].

Este trabajo está centrado en el análisis, detección y corrección de los diferentes fallos y errores que han sido encontrados en los proyectos analizados que serán descritos más adelante, consiguiendo así mejorar la calidad del software en producción. Esto acaba siendo beneficioso tanto para los desarrolladores encargados de mantener el software como para la empresa a nivel de costes en horas de mantenimiento y parches.

En resumen, el desarrollo de software seguro es esencial en el proceso de desarrollo de software debido a su capacidad para proteger datos sensibles, salvaguardar la privacidad de los usuarios, generar confianza, evitar daños financieros y legales, y mantener la integridad del software. En un entorno digital cada vez más complejo y amenazante, el enfoque en la seguridad del software se ha convertido en un requisito para cualquier organización que busque ofrecer soluciones tecnológicas robustas y confiables.

1.1 Objetivos

El objetivo principal es conseguir que el producto formado por el software presentado en los Capítulos 4 y 5, sean más fáciles de mantener, respeten unos estándares y salga con menor número de errores y fallos a producción. Para esto se presenta SonarQube como analizador estático de código fuente y así añadir esta fase al ciclo de desarrollo de estos proyectos de software, obteniendo finalmente un ciclo similar al que muestra en la siguiente imagen.

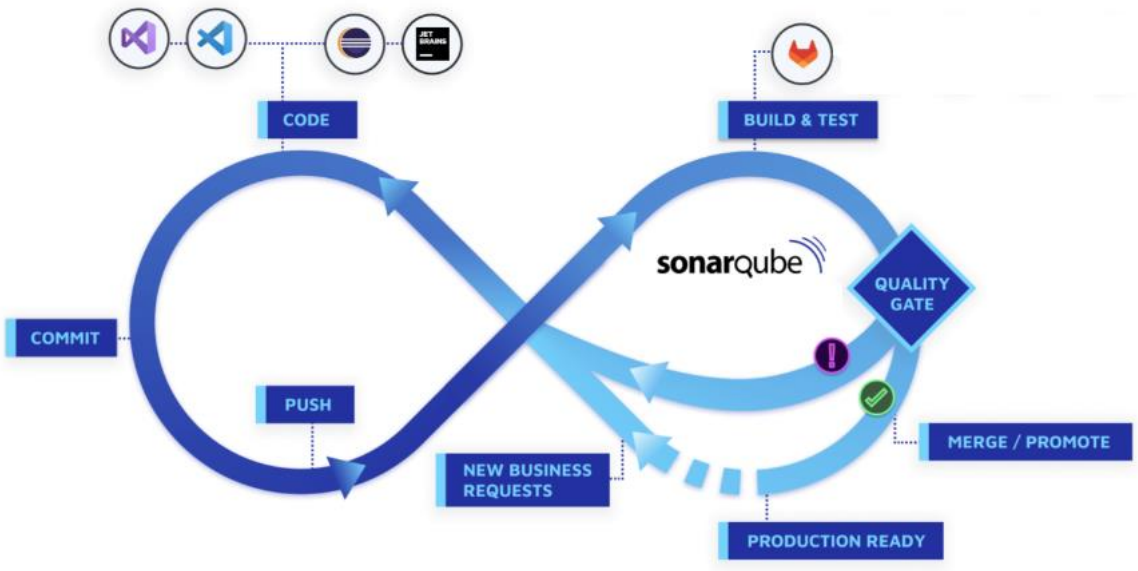

Ilustración 1 - ciclo de desarrollo con SonarQube

Por ello, se ha centrado en encontrar y corregir potenciales bugs y malas prácticas, como errores de referencias de puntero nulo, saltarse las convenciones de nomenclatura estándar del sector, escribir condicionales de una línea sin utilizar llaves, entre otros errores y malas prácticas.

1.2 Metodología

Para el desarrollo del proyecto se ha realizado una investigación previa sobre la herramienta SonarQube la cual se expone en el Capítulo 3. Una vez hecho se ha instalado una instancia de esta herramienta en un contenedor Docker con la versión 10.0 de la herramienta, el proceso está detallado en el Capítulo 3.2 y se han realizado las configuraciones necesarias y creado los proyectos de SonarQube para poder realizar los análisis sobre los proyectos de software. Esto último se trata en los Capítulos 4 y 5.

Respecto el documento, los subtítulos destacados en negrita en los apartados 4 y 5 hacen referencia a la descripción resumida que ofrece SonarQube de cada error. También aparecen partes escritas en cursiva indicando que ese fragmento forma parte de alguna parte de código, ya sean nombres de funciones, paquetes, sentencias o clases.

1.3 Descripción de los proyectos

Los proyectos analizados forman parte de un software de gestión de empresas puesto en producción, para la compañía *Tubsol, Especialistas en trabajos en hierro y acero inoxidable* con los que se gestionan usuarios, artículos, clientes, proveedores y documentos como facturas, presupuestos, pedidos, albaranes y otros documentos concretos que son específicos para el cliente entre otras diversas funcionalidades.

El proyecto 1, es la parte back-end. Este es un proyecto de 41.607 líneas en Java Spring Boot, el cual ofrece un servidor web integrado que despliega un api con sus endpoints y las dependencias son gestionadas con Apache Maven.

Por otra parte, se ha analizado el software encargado de la parte front-end, este contiene 17.158 líneas y es un proyecto que utiliza el framework Angular, que utiliza Typescript como lenguaje de programación y como gestor de dependencias de Node NPM.

2 DESARROLLO DE SOFTWARE SEGURO

Cuando se habla de desarrollo seguro de software se hace referencia a la práctica de diseñar, crear y mantener software que sea resistente a las amenazas y este protegido contra las vulnerabilidades. Consiste en aplicar medidas de seguridad desde las etapas iniciales del proceso de desarrollo hasta la implementación y el mantenimiento continuo del software.

Para lograr un desarrollo de software seguro, es esencial adoptar un enfoque integral que abarque todo el ciclo de vida del desarrollo de software. Esto implica el incorporar medidas de seguridad desde las etapas iniciales de diseño y arquitectura, hasta el desarrollo y las pruebas, y finalmente, durante la implementación y el mantenimiento continuo.

En este trabajo se centra la atención en la fase de desarrollo, integrando una herramienta que utiliza el análisis estático de código fuente para encontrar diferentes tipos de problemas que pueda haber en un proyecto, ya sean errores de programación, malas prácticas o fallos de seguridad.

2.1 Análisis estático de código fuente

El análisis estático de programas es el arte de razonar sobre el comportamiento de los programas informáticos sin llegar a ejecutarlos, esto es útil para la detección automática de errores y otras herramientas que pueden ayudar a los programadores. Actualmente hay herramientas que automatizan esta tarea, denominadas analizadores estáticos y estos son programas que razonan sobre el comportamiento de otros programas [2]. De esta manera es posible contestar automáticamente a preguntas sobre el comportamiento de un programa.

Utilizan una serie de técnicas y algoritmos para identificar problemas comunes, como vulnerabilidades de seguridad, errores de programación, código duplicado, malas prácticas de codificación, entre otros. Además de proporcionar una visión detallada de la calidad del código fuente, permitiendo a los desarrolladores identificar y corregir los problemas antes de que surjan cuando el software está en producción. Estas herramientas generan unos informes detallados con los problemas que encuentran, incluyendo dónde se ha encontrado el problema, recomendaciones y soluciones conocidas. Esto es visto más adelante con la herramienta SonarQube.

El análisis estático ha sido históricamente empleado para la optimización de compiladores de software. Sin embargo, más recientemente, se ha comprobado su utilidad en la

detección de errores, herramientas de verificación y en el apoyo al desarrollo de programas en los entornos de desarrollo integrados (IDE).

2.2 OWASP

Open Web Application Security Project, https://owasp.org, es una fundación sin ánimo de lucro dedicada a mejorar la seguridad del software mediante la producción de artículos, documentación, metodologías, herramientas y tecnologías.

Esta fundación mantiene un listado de los tops 10 vulnerabilidades de seguridad en aplicaciones web desde el 2003 y van actualizando cada dos o tres años según los avances y los cambios en el mercado de AppSec Market [3].

OWASP también pretende educar tanto a desarrolladores como a diseñadores, arquitectos o empresarios del sector sobre los riesgos asociados con las vulnerabilidades más comunes que pueden aparecen en aplicaciones web.

En el Capítulo 3 se comentan algunas de las vulnerabilidades detectadas por SonarQube que se basan en este estándar.

2.3 CWE

Common Wekness Enumeration o CWE, *https://cwe.mitre.org*, es una comunidad que ha creado un listado gratuito que trata diversas vulnerabilidades de seguridad tanto software como hardware que son comunes. Este listado se presenta como un diccionario online dónde se identifican cada uno de los puntos débiles y está disponible de forma gratuita y accesible en todo el mundo.

MITRE Corporation es la responsable de mantener este diccionario de vulnerabilidades informáticas y su propósito con este estándar es facilitar la utilización de herramientas que permitan identificar, localizar y corregir errores, vulnerabilidades y fallas en el software antes de que este pase a producción o venta al público.

En la parte de software, tienen el listado *CWE Top 25 Most Dangerous Software Weaknesses* [4]. Este listado identifica errores de programación que van desde los más generales a los más críticos los cuales pueden llevar tener expuesto un software a vulnerabilidades graves. Fue lanzado por primera vez en 2009 y fue construido con respuestas de encuestas de una amplia selección de desarrolladores, analistas de seguridad, investigadores y

proveedores que nominaron las vulnerabilidades que consideraban más frecuentes o importantes para poder crear un ranking que fuera útil para la comunidad. A partir del año 2019, se adoptaron un nuevo enfoque basado en datos, repetible y programable, permitiendo así generar de manera actualizada y menos costosa este listado. Actualmente es un recurso comúnmente utilizado en la comunidad tecnológica comprometida con la seguridad [5].

2.4 CVE

Common Vulnerabilities and Exposures o CVE, *https://cve.mitre.org*, es un diccionario público de vulnerabilidades y exposiciones de seguridad conocidas mantenido por la Corporación MITRE y proporciona un identificador estandarizado para cada vulnerabilidad o exposición. Este identificador permite a las organizaciones compartir datos sobre vulnerabilidades y exposiciones de seguridad entre diferentes herramientas y bases de datos facilitando la comunicación entre investigadores de seguridad, proveedores y usuarios [6].

Los listados de CWE están relacionados con el diccionario de CVE ya que esta primera utiliza los identificadores del diccionario de CVE para referenciar vulnerabilidades o exposiciones específicas en su listado. A modo de ejemplo, podemos asignar un identificador CVE a una vulnerabilidad específica encontrada en una aplicación de software. Esta vulnerabilidad puede ser clasificada bajo una debilidad CWE específica. De esta manera, las organizaciones pueden realizar un seguimiento y gestionar las vulnerabilidades y exposiciones utilizando un lenguaje común y un conjunto de identificadores estandarizados [7].

3 ANÁLISIS ESTÁTICO CON SONARQUBE

Existen diferentes herramientas para realizar este tipo de análisis, como Fortify Static Code Analyzer, Jetbrains qodana o SonarQube. En este trabajo se ha decidido utilizar SonarQube dado que es una herramienta open source y tiene una versión community totalmente gratuita que ofrece análisis estático para los lenguajes que hacían falta para los dos proyectos, concretamente Java, Javascript, Typescript, HTML y CSS [8]. Además, esta versión nos permite detectar tanto bugs como vulnerabilidades básicas, code smells y fallos de seguridad, además de obtener métricas sobre la calidad del código, cobertura de test, duplicaciones de código y documentación entre otras características, analizando y creando un historial sobre los análisis que se van realizando en un proyecto [9].

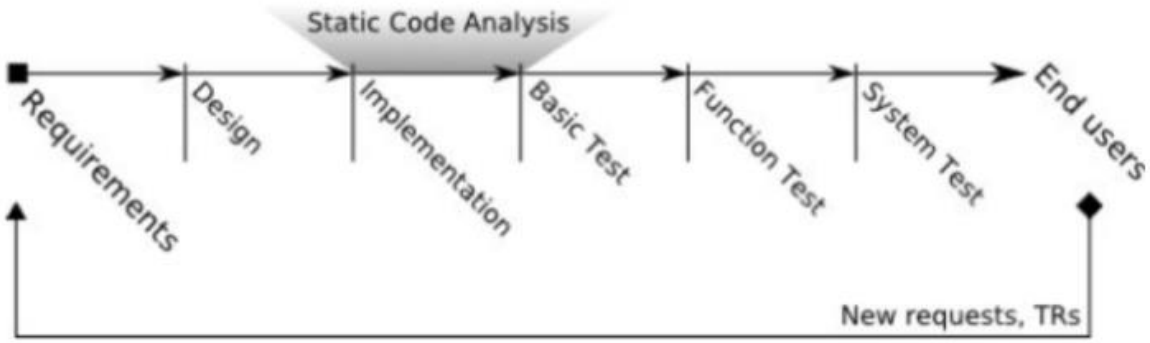

Ilustración 2 - Fase análisis código estático

SonarQube tiene en cuenta las reglas de seguridad en categorías que se encuentran en los primeros puestos de los estándares OWASP Top 10 y CWE Top 25. Las reglas en categorías que se encuentran en los primeros puestos de los estándares OWASP Top 10 y CWE Top 25 se consideran de alta prioridad de revisión, si no hay reglas correspondientes a una categoría OWASP determinada activada en el perfil de calidad, no se obtiene problemas vinculados a esa categoría específica [10]

Algunas de las problemáticas o issues que detecta SonarQube relacionadas con el estándar OWASP son [11]:

- Code Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control

- Cross-Site Request Forgery (CSRF)
- Using Components with Known Vulnerabilities
- Insufficient Logging and Monitoring

Algunas de las issues detectadas relacionadas con el listado del top de vulnerabilidades de la CWE son [CWE list]:

- Improper Input Validation (CWE-20): Esta vulnerabilidad ocurre cuando una aplicación no valida correctamente el input de un usuario, lo que potencialmente permite que un atacante inyecte código malicioso o datos en una aplicación.
- Hard-coded Credentials (CWE-798): Esta vulnerabilidad se produce cuando una aplicación utiliza credenciales escritas directamente en el código fuente, nombres de usuario y contraseñas, que pueden ser fácilmente descubiertas por atacantes.
- Unrestricted Upload of File with Dangerous Type (CWE-434): Esta vulnerabilidad ocurre cuando una aplicación permite a los usuarios cargar archivos sin validar adecuadamente el tipo de archivo o su contenido, lo que potencialmente permite que un atacante cargue archivos maliciosos que pueden ser ejecutados en el servidor o utilizados para robar datos sensibles.

3.1 Los siete ejes de SonarQube

SonarQube se centra en siete puntos para realizar las mediciones de sus análisis, estos son [12]:

1. Potential bugs: Este eje identifica posibles fallos, como excepciones de punteros nulos y fugas de recursos.
2. Coding rules: Este eje examina el cumplimiento de buenas prácticas que pueden ayudar a la legibilidad, el mantenimiento y la fiabilidad del código.
3. Tests: Este eje se centra en la calidad y la cobertura de las pruebas unitarias, que pueden contribuir a la fiabilidad y el mantenimiento.
4. Duplications: Este eje identifica el código que se ha copiado y pegado, lo que puede provocar problemas de mantenimiento y errores.
5. Comments: Este eje examina la calidad y cantidad de comentarios en el código, lo que puede ayudar a su legibilidad y mantenimiento.
6. Architecture and design: Este eje se centra en el diseño general y la arquitectura del código, incluida la modularidad, el acoplamiento y la cohesión.

7. Complexity: Este eje examina la complejidad del código, incluido el número de líneas de código, el número de ramas y el número de bucles.

3.2 Instalación de una instancia SonarQube

Hay diferentes maneras de instalar una instancia de la herramienta. Una opción es instalando manualmente la herramienta desde un archivo zip que se puede descargar de la misma página web de sonarsource:

Ilustración 3 - instalación de una instancia SonarQube desde un archivo zip [13]

La otra opción y la cual se ha utilizado en este proyecto es creando un contenedor Docker utilizando una de sus imágenes, las cuales están subidas en Docker hub:

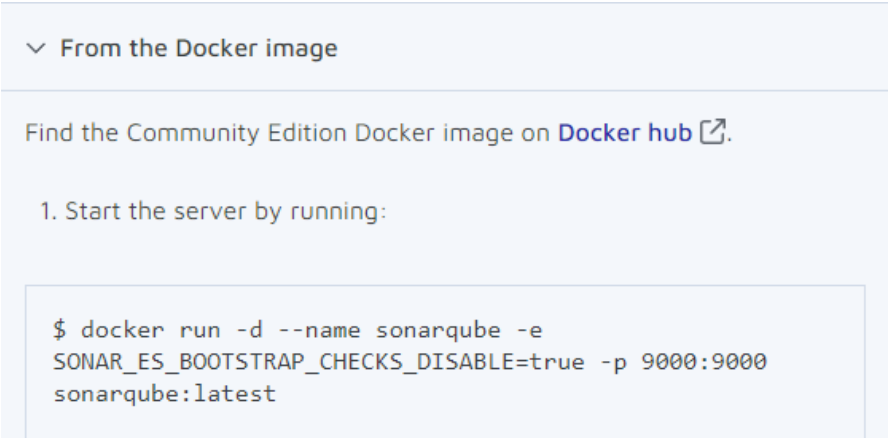

Ilustración 4 - Instalación de una instancia SonarQube desde un contenedor Docker [13]

En ambos casos, una vez se ha instalado la herramienta o desplegado el contenedor, se accede al panel de administración mediante un navegador mediante la url: *http://localhost:9000* con las credenciales por defecto:

- Login: admin
- Password: admin

3.3 Análisis de un proyecto

Una vez iniciada una sesión desde el panel de administración, desde la pestaña *analizar un proyecto* nos permite crear un proyecto seleccionando un repositorio de código local o un repositorio remoto que esté en alguna plataforma DevOps como puede ser GitHub, GitLab, Bitbucket o Azure [13].

Los pasos para seguir para realizar un análisis serían los siguientes:

1. Seleccionar la opción *Crear un nuevo proyecto.*
2. Proporcionar una clave de proyecto o generar una nueva e indicar el nombre con el que se quiere configurar el proyecto SonarQube.
3. Seleccionar Generar un token, asignarle un nombre y seleccionar Generar.
4. Selecciona el idioma principal de tu proyecto en *Ejecutar análisis en tu proyecto* y seguir las instrucciones para realizar el primer análisis. En este punto, si no lo has hecho previamente, se deberá de descargar y ejecutar un escáner. Hay varias

opciones en este caso, si se está utilizando Maven o Gradle, el escáner se descargará automáticamente. Si no, se muestra un enlace a la página dónde descargar el analizador.

3.4 Security hotspot

Los security hotspots son un tipo de problemas que SonarQube identifica como una posible vulnerabilidad pero que dependiendo del contexto no tiene por qué representar una amenaza real. Serian fragmentos de código que podrían ser vulnerables a un ataque y requieren una investigación adicional. La herramienta los encuentra mediante la búsqueda de patrones asociados comúnmente con vulnerabilidades de seguridad del estilo de la utilización de algoritmos criptográficos inseguros o la presencia de vulnerabilidades de SQL injection.

La manera con la que se tratan este tipo de problemas desde SonarQube es algo distinta al resto de problemáticas que detecta. Una vez identificadas, se muestra información relevante acerca del problema y te permite indicar qué tipo de acción se va a realizar o se ha realizado para corregirlo como se muestra en la siguiente imagen.

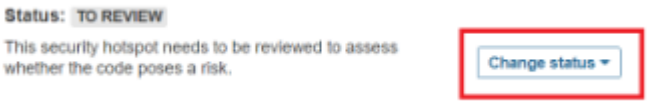

Ilustración 5 - Security hotspot change status

Cabe remarcar que no todos los problemas que son encontrados por SonarQube tienen porqué ser problemas que representen un defecto o una amenaza real para la aplicación en el contexto en el que se utiliza o es desplegada. Por lo tanto, el personal encargado de revisar el código tiene que ser responsable de realizar una evaluación de los errores encontrados para determinar si realmente representan una amenaza o no y qué medidas tomar [14].

Para aclarar la diferencia entre un security hotspot y el resto de las vulnerabilidades detectadas por la herramienta, es que un security hotspot es una vulnerabilidad la cual necesita ser revisada antes de tomar una decisión de que acciones se va a realizar sobre ella, ya que hay un contexto que puede afectar, mientras que una vulnerabilidad representa un problema real para la seguridad de la aplicación y debería de ser corregida lo antes posible

[15]. Algunos ejemplos de security hotspots que pueden ser detectados por SonarQube se pueden ver en la ilustración 6.

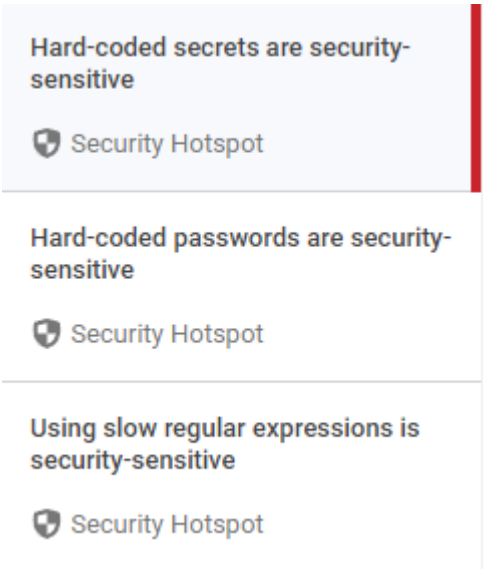

Ilustración 6 - Ejemplos de security hotspots

3.4.1 Ciclo de vida

Como he nombrado anteriormente, los security hotspots tienen un ciclo de vida diferente al resto de vulnerabilidades. Para realizar cambios en su estado, el usuario necesita tener el nivel de permiso de Administrador de security hotspots y los estados existentes que están disponibles son [15]:

- To review: Seria el estado por defecto de las detecciones nuevas. La herramienta informa de que ha encontrado un problema y notifica que debe comprobarse.
- Acknowledged: Un desarrollador ha revisado el punto crítico de seguridad y está pendiente la resolución del riesgo destacado. Esto incluye los casos en los que se está realizando una corrección o en los que se necesita tiempo para determinar el siguiente paso.
- Fixed: Un desarrollador ha revisado el problema de seguridad y ha aplicado una solución.
- Safe: Un desarrollador ha revisado el punto crítico de seguridad y ha determinado que no es necesario ningún cambio, ya sea porque ya existen otras protecciones más relevantes o porque dado el contexto no suponen un riesgo.

3.4.2 Flujo de trabajo

SonarQube recomienda un flujo de trabajo a seguir para revisar estos puntos conflictivos y aplicar las correcciones que se consideren necesarias [15]:

- Review priority: Esta prioridad viene determinada por la categoría de seguridad de cada regla de seguridad. Se considera que las reglas de las categorías mejor clasificadas en los estándares OWASP Top 10 y CWE Top 25 tienen una prioridad de revisión alta. Las reglas en categorías que no están clasificadas como altas o que no se mencionan en los estándares OWASP Top 10 o CWE Top 25 se clasifican como Media o Baja.
- Review hotspots: Cuando se revisa un hostpot hay que evaluar cuál es el nivel de amenaza que este supone. Una vez revisado sería el momento de realizar las acciones que se consideraran necesarias.

3.5 Issues

Como parte del análisis, SonarQube compara el código con un conjunto de reglas. Cuando se incumple una regla, se marca un problema en la línea en la que se ha producido y se cataloga con uno de los tipos Bug, Vulnerabilidad o Code smell. Además, cada problema dentro de cada tipo se clasifica en uno de los cinco niveles de gravedad siguientes: Blocker, Critical, Major, Minor e Info [16]. En el siguiente punto, se explica cada uno con mayor detalle.

3.5.1 Niveles de gravedad

Los niveles de gravedad clasifican en cinco niveles cada tipo de issues detectadas por SonarQube. Los bugs y bugs potenciales son los errores que requieren mayor atención y suelen ser issues catalogadas como Blocker o Critical. Por otra parte, problemas detectados que puedan provocar ineficiencias o errores futuros de programación suelen catalogarse como Major o Minor. En la siguiente imagen, puede verse con mayor claridad el tipo de errores según su clasificación [17].

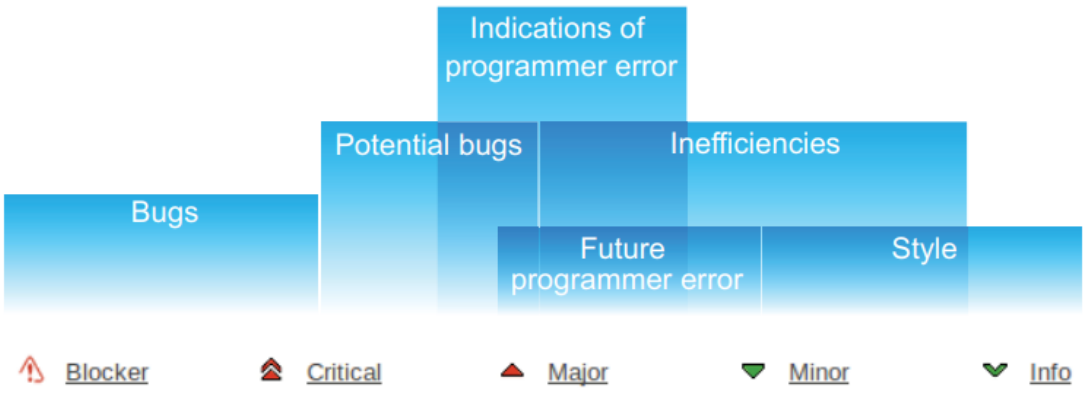

Ilustración 7 - SonarQube gravedad de las issues [18]

Blocker: Problemas que impiden que el código funcione correctamente o introducen vulnerabilidades de seguridad. Estos problemas requieren una atención inmediata y deben solucionarse antes de poner la aplicación en producción. Un ejemplo puede ser, una fuga de memoria o alguna conexión a base de datos (JDBC) que no se haya cerrado tras su uso.

Critical: Problemas que tienen un gran impacto en la funcionalidad o seguridad del código. Estos problemas deben solucionarse lo antes posible. Algún ejemplo sería, un bloque catch vacío o un fragmento dónde se pueda realizar una inyección SQL.

Major: Problemas que tienen un impacto más moderado en la funcionalidad o seguridad del código y que pueden afectar en gran medida a la productividad del desarrollador. Estos problemas deben solucionarse, aunque con menor urgencia. Algunos ejemplos serían un fragmento de código sin cubrir, bloques duplicados o parámetros definidos, pero no utilizados.

Minor: Problemas con escaso impacto en la funcionalidad o seguridad del código. Estos problemas pueden solucionarse más adelante. Por ejemplo, las líneas no deben ser demasiado largas y las sentencias *switch* deben tener al menos tres casos.

Info: Problemas que proporcionan información sobre el código, pero no afectan a su funcionalidad o seguridad. Estos problemas no requieren una atención inmediata y pueden resolverse a discreción del desarrollador.

3.5.2 Tipos

Son las categorías con las que se clasifican los errores encontrados. Los tipos están relacionados con los niveles de gravedad explicados en el punto anterior. Cada error es clasificado con un tipo, a éste se le asocia un nivel de gravedad, el cual indica cuanto de importante es para así dar a conocer cuanta atención hay que darle a cada error.

Bug

La primera categoría y la que requiere mayor atención es la de Bugs. Los problemas categorizados como Bugs representan amenazas seguras en el código y deben de tratarse lo antes posible. Este tipo de problemas se refiere a errores de programación que pueden causar fallos o comportamientos inesperados en el software [17].

En el mejor de los casos, este tipo de problema provoca una degradación del rendimiento que acaba requiriendo reinicios innecesarios de la aplicación. En el peor de los casos, es probable que los usuarios tengan que lidiar con comportamientos extraños del programa y mensajes de error, o directamente con bloqueos y crashes en la aplicación [19]. Algunos ejemplos de bugs que pueden ser detectados por SonarQube se pueden ver en la ilustración 8.

Los tipos de problemas categorizados como bugs son:

- Errores lógicos que llevan a excepciones de puntero nulo.
- Fallos al cerrar manejadores de fichero o conexiones a bases de datos.
- Mal comportamiento en un entorno multihilo.
- Métodos diseñados para comprobar la igualdad que siempre devuelven falso o verdadero.
- Casts de clases que no son imposibles de realizar.

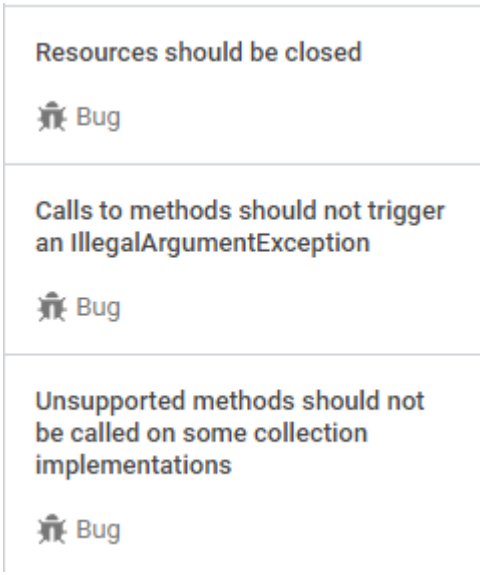

Ilustración 8 - Ejemplos de bugs [20]

Vulnerabilidad

Las vulnerabilidades son puntos en el código de la aplicación que están abiertos a un posible ataque malicioso. Estas vulnerabilidades representan fallas que pueden ser explotadas para comprometer la integridad, confidencialidad o disponibilidad del sistema y por tanto requieren ser tratadas inmediatamente [21].

Algunas de las vulnerabilidades detectadas por SonarQube son:

- Vulnerabilidades de SQL injection: Se trata de un tipo de vulnerabilidad de seguridad que permite a los atacantes inyectar sentencias SQL maliciosas en la base de datos de una aplicación.
- Vulnerabilidades en la gestión de contraseñas: Se trata de un tipo de vulnerabilidad de seguridad que se produce cuando las contraseñas no se almacenan de forma segura o son fácilmente adivinables.
- Fallos en la gestión de errores y registro de logs: Se trata de problemas relacionados con la forma en que una aplicación gestiona los errores y registra la información.

SonarQube tiene dos tipos diferentes de reglas de seguridad: Hotspots y Vulnerabilidades. Un hotspot, como fue explicado en el Capítulo 3, es una vulnerabilidad de seguridad potencial que requiere de ser revisada antes de realizar una acción sobre ella, mientras que una vulnerabilidad es un problema de seguridad confirmado que necesita ser solucionado inmediatamente [22]. Algunos de los ejemplos de vulnerabilidades que puede detectar SonarQube se muestran en la ilustración 9.

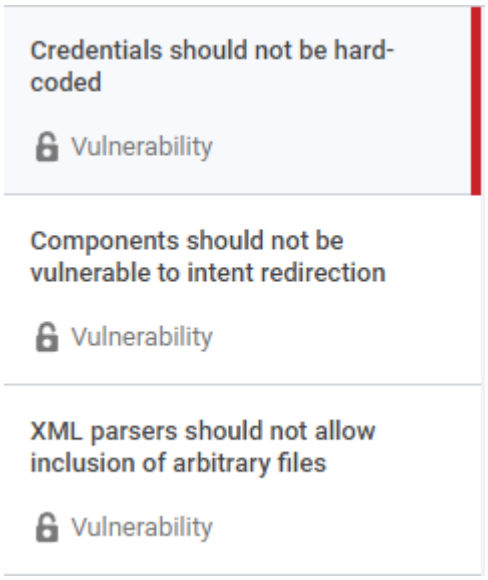

Ilustración 9 - Ejemplos de vulnerabilidades [23]

Code smell

Los code smells son problemas relacionados con el mantenimiento del código. No realizar ninguna acción sobre los code smells detectados significa que, en el mejor de los casos, los desarrolladores que mantengan el código lo tendrán más difícil de lo que deberían a la hora de hacer cambios. En el peor de los casos, puede crear confusiones sobre el funcionamiento del código que estén modificando, lo que puede provocar la aparición de nuevos errores adicionales al realizar los cambios [24].

Este tipo de errores no suelen ser técnicamente incorrectos y no impiden que el programa funcione correctamente. Simplemente muestran violaciones de los fundamentos del desarrollo de software que disminuyen la calidad del código y que no tratarlos puede conllevar costes innecesarios de mantenimiento a largo plazo [25].

Algunos tipos de errores categorizados por SonarQube como code smells son [26]:

- Código duplicado: Este error ocurre cuando el mismo código es copiado y pegado en más de un lugar en el código.
- Método largo: Este se produce cuando un método es demasiado largo y complejo, lo que dificulta su comprensión y mantenimiento.
- Complejidad: Es producido cuando el código es demasiado complejo, lo que hace que sea difícil de entender y mantener.

- Código muerto: Este error se produce cuando el código ya no se utiliza, pero todavía está presente en la base de código.

Algunos ejemplos de los posibles code smells que SonarQube puede detectar se pueden ver en la siguiente ilustración.

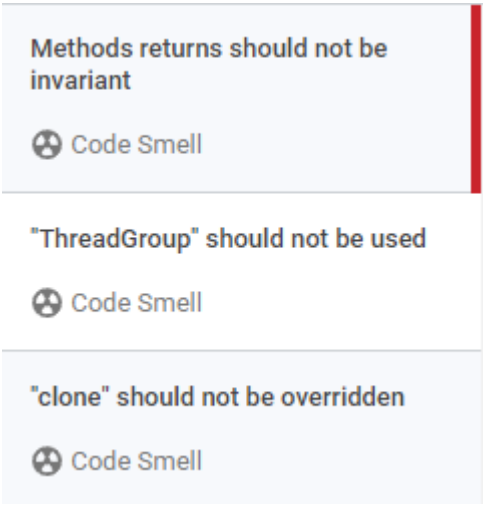

Ilustración 10 - Ejemplos code smells [27]

3.5.3 Reglas

Existen cuatro tipos de reglas que SonarQube utiliza para generar los problemas cuando se realizan los análisis, estos son los code smells, los bugs, vulnerabilidades y security hotspots. SonarQube tiene un catálogo de reglas iniciales que pueden ser utilizadas como punto de entrada, aunque también pueden crearse nuevas en base a otras plantillas que hay disponibles [28].

Por un lado, en los code smells y bugs, se espera que no aparezcan falsos positivos. Mientras que, en el caso de las vulnerabilidades, aproximadamente el 80% de los problemas detectados se espera que sean verdaderos positivos. Por otra parte, los security hotspots, requieren de ser revisados por un desarrollador y se espera que más del 80% de los problemas se resuelvan rápidamente como revisados.

Por defecto, al seleccionar el elemento de menú superior Reglas, se pueden ver todas las reglas disponibles instaladas en la instancia de SonarQube que se esté utilizando [28]:

- Lenguaje: el lenguaje sobre el que se aplica una regla como Java, Js o C#.
- Tipo: Bugs, vulnerabilities, code smells o security hotspots.

- Tag: Es posible añadir etiquetas a las reglas para clasificarlas y ayudar a descubrirlas más fácilmente.
- Repositorio: El motor o analizador que aporta reglas a SonarQube.
- Severidad por defecto: La severidad original de la regla definida por SonarQube.
- Estado: Las reglas pueden tener 3 estados diferentes:
    - Beta: La regla se ha implementado recientemente y aún no hemos recibido suficientes comentarios de los usuarios, por lo que puede haber falsos positivos o falsos negativos.
    - Obsoleta: La regla ya no debe utilizarse porque existe una regla similar, pero más potente y precisa.
    - Preparada: La regla está lista para ser utilizada en producción.
- Disponible desde: La fecha en que una regla se añadió por primera vez en SonarQube. Esto es útil para listar todas las nuevas reglas desde la última actualización de un plugin, por ejemplo.
- Plantilla: Muestra plantillas de reglas que permiten crear reglas personalizadas.
- Perfil de calidad: inclusión o exclusión de un perfil específico.

Si se selecciona un perfil de calidad, también es posible comprobar su gravedad activa y si se hereda o no.

3.5.4 Tags

Las etiquetas sirven para clasificar las reglas y las issues. Las issues heredan las etiquetas de las reglas que las generaron. Algunas de las etiquetas son específicas de cada lenguaje, pero muchas otras aparecen en todos los lenguajes [29].

Además de las tags por defecto que tienen asignadas las reglas y las issues, desde el panel de administración de SonarQube se pueden añadir otras que el desarrollador crea conveniente.

Algunos de los ejemplos de tags pueden ser:

- Bad-practice: El código probablemente funciona como fue diseñado, pero la forma en que fue diseñado sigue una conocida mala práctica.

- Clumsy: Esta etiqueta se utiliza para reglas que indican que el código es difícil de entender.
- Confusing: Indica que las personas encargadas de mantener tardarán más en entenderlo de lo que realmente está justificado por lo que el código hace en realidad.
- Unused: Código no utilizado; por ejemplo, una variable privada que nunca se utiliza.
- Security: Esta etiqueta se utiliza para reglas que indican que el código tiene problemas de seguridad.
- Performance: Esta etiqueta se utiliza para las reglas que indican que el código tiene problemas de rendimiento.

En la ilustración 11, se muestran algunos ejemplos de tags por defecto que tiene SonarQube en su base de datos:

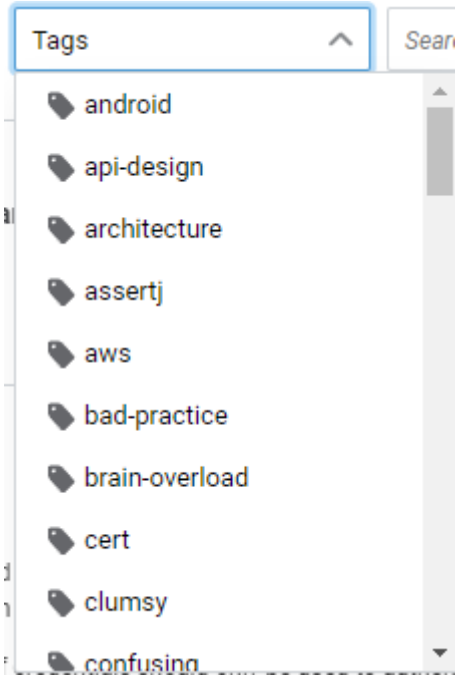

Ilustración 11 - Ejemplos de Tags [30]

3.6 Mediciones

SonarQube utiliza multitud de mediciones para calcular las puntuaciones que asigna a cada apartado analizado al realizar un análisis de un proyecto. A continuación, se detallan las mediciones que se han considerado más relevantes [31].

Complejidad

Esta métrica hace referencia a la complejidad ciclomática. Esta es una medida cuantitativa que se utiliza para calcular el número de posibles rutas que puede ir siguiendo el código. Cada vez que el flujo de control de una función se ramifica, el contador de complejidad ciclomática aumenta en uno. Cada función tiene una complejidad mínima de 1. Es importante tener en cuenta que el cálculo de la complejidad ciclomática puede variar ligeramente según el lenguaje de programación, debido a las diferencias en las palabras clave y funcionalidades. Al analizar la complejidad ciclomática, los desarrolladores pueden comprender mejor la complejidad del código e identificar zonas del código dónde sean potencialmente optimizables y tengan capacidad de refactorización.

En la imagen siguiente, se muestran qué elementos del código se tienen en cuenta por SonarQube para calcular esta métrica. Concretamente muestra para los lenguajes Java, JS/TS que son los lenguajes utilizados en los proyectos analizados posteriormente.

Java	Keywords incrementing the complexity: `if`, `for`, `while`, `case`, `&&`, `||`, `?`, `->`.
JS/TS, PHP	Complexity is incremented by one for each: function (i.e non-abstract and non-anonymous constructors, functions, procedures or methods), `if`, short-circuit (AKA lazy) logical conjunction (`&&`), short-circuit (AKA lazy) logical disjunction (`||`), ternary conditional expressions, loop, `case` clause of a `switch` statement, `throw` and `catch` statement, `go to` statement (only for PHP).

Ilustración 12 - Detalles segun java, js/ts y php [31]

Mantenibilidad

Code smells (code_smells): Es el recuento total de problemas de tipo code smell.

Nuevos code smells (new_code_smells): Recuento total de code smells detectados por primera vez en el análisis de un fragmento de código nuevo.

Índice de mantenibilidad (sqale_rating): calificación con la que se puntúa un proyecto en relación con su valor de ratio de deuda técnica. La tabla de asignaciones de puntuación respecto al ratio seria la siguiente:

Puntuación	Ratio
A	0,05
B	0,06 – 0,1
C	0,11 – 0,20
D	0,21 - 0,5
E	0,51 - 1

Tabla 1 – Tabla relación de la puntuación respecto el ratio de deuda técnica

La Tabla 1 se puede explicar de la siguiente manera. Si se ha dedicado menos del 5% del tiempo a la aplicación, la calificación es A. Si se ha dedicado entre el 6% y el 10% del tiempo, la calificación es B. Para un tiempo dedicado entre el 11% y el 20%, la calificación es C. Si se ha utilizado entre el 21% y el 50% del tiempo, la calificación es D. Si se ha utilizado más del 50% del tiempo, la calificación es E.

Ratio de la deuda técnica (sqale_debt_ratio): La relación entre el coste de desarrollar el software y el coste de repararlo. La fórmula con la que se calcula este ratio es:

$$\frac{Coste\ de\ reparación}{Coste\ de\ desarrollo}$$

Que puede ser reformulada de la siguiente manera:

$$\frac{Coste\ de\ correccion\ de\ errores}{(Coste\ de\ desarrollo\ de\ 1\ linea)\ x\ Numero\ de\ lineas\ de\ codigo}$$

Quality profiles

Los perfiles de calidad de SonarQube se basan en un conjunto de reglas que definen los criterios de calidad del código. Estas reglas se basan en las buenas prácticas que son comúnmente conocidas y en los estándares de programación y se basan en la complejidad del código, la duplicación de código, los estándares de programación y los errores potenciales [32].

Los perfiles de calidad son configurables, lo que permite que los desarrolladores adapten qué reglas cumplen con sus requisitos específicos y son relevantes a la hora de analizar sus repositorios. Además, las reglas se van actualizando y mejorando continuamente a partir de los comentarios de la comunidad y expertos del sector, lo que garantiza que los perfiles de calidad sigan siendo relevantes y eficaces.

Las métricas que se asocian estrechamente con los perfiles de calidad son las de las puertas de calidad y son las siguientes:

- Quality gate details (quality_gate_details): Para cada una de las condiciones de la puerta de calidad se indica qué puntuación tiene, de manera que puede saberse qué condiciones provocan que falle la puerta de calidad.
- Quality gate status (alert_status): El estado de la puerta de calidad asociada a cada proyecto analizado. Los valores posibles son ERROR y OK. Este estado se determina comparando los resultados del análisis con las condiciones predefinidas. Si el código cumple todas las condiciones, el estado de la puerta de calidad se aprueba, lo que indica que el código cumple las normas de calidad exigidas.

Si el código no cumple alguna de las condiciones, el estado de la puerta de calidad falla, lo que indica que el código necesita más mejoras antes.

- Quality of the code: La calidad del código se representa mediante una letra de calificación, que va de la A a la E, siendo A la puntuación mayor y E la menor.

El sistema de puntuación por letras proporciona una forma rápida y sencilla de evaluar la calidad general del código e identificar las áreas que necesitan mejoras. Los desarrolladores pueden priorizar sus tareas entorno a esta puntuación y centrarse en las áreas que tendrán un mayor impacto. Además, es una manera sencilla de comunicar resultados con otras partes involucradas en el proyecto como directores de proyecto y clientes.

La letra de calificación se basa en una combinación de los resultados del análisis del código y los umbrales de calidad predefinidos por el usuario. La letra de la puntuación es calculada siguiendo los siguientes criterios:

- A: El código cumple todos los umbrales de calidad.
- B: El código cumple la mayoría de los umbrales de calidad.
- C: El código cumple algunos de los umbrales de calidad.
- D: El código cumple pocos umbrales de calidad.
- E: El código no cumple la mayoría o todos los umbrales de calidad.

C es la última puntuación en la que se considera que el código es aceptable para lanzar a producción, mientras que D y E ya se considera como código de poca calidad y que se deberían de corregir errores antes de lanzarlo.

Fiabilidad

Bugs (bugs): El número total de bugs encontrados.

Nuevos bugs (new_bugs): El número de nuevos bugs.

Índice de fiabilidad (reliability_rating): Este sigue el mismo método comentado en el punto anterior de puntuación por letras.

- A = 0 fallos
- B = al menos 1 fallo menor
- C = al menos 1 fallo grave
- D = al menos 1 fallo crítico
- E = al menos 1 error de bloqueo

Esfuerzos para mejorar la fiabilidad (reliability_remediation_effort): El esfuerzo para solucionar todos las issues correspondientes a los bugs. La métrica se almacena en minutos en la base de datos. Se supone una jornada de 8 horas cuando los valores se muestran en días.

Esfuerzos para mejorar la fiabilidad en código nuevo (new_reliability_remediation_effort): Lo mismo que en la métrica anterior, pero sobre el código cambiado en el código nuevo analizado.

Security

Vulnerabilidades (vulnerabilities): Número total de issues clasificadas como vulnerabilidades.

Vulnerabilidades en nuevo código (new_vulnerabilities): Numero de nuevas vulnerabilidades.

Rating de seguridad (security rating): La puntuación va de la letra A a la E.

- A = 0 vulnerabilidades detectadas.
- B = al menos 1 vulnerabilidad Minor detectada.
- C = al menos 1 vulnerabilidad Major detectada.
- D = al menos 1 vulnerabilidad Critical detectada.
- E = al menos 1 vulnerabilidad Blocker detectada.

Esfuerzo invertido en corregir vulnerabilidades de seguridad (security_remediation_effort): El esfuerzo para solucionar todos los problemas de vulnerabilidad. La medida se almacena en minutos en la base de datos. Se supone un día de 8 horas cuando los valores se muestran en días.

Security hotspots (security_hotspots): El número de hotspots de seguridad detectados.

Calificación de la revisión de seguridad (security_review_rating): La calificación de revisión de seguridad se califica mediante letras basada en el porcentaje de hotspots de seguridad revisados. Tenga en cuenta que los puntos críticos de seguridad se consideran revisados si están marcados como Acknowledged, Fixed or Safe.

- A = >= 80%
- B = >= 70% y <80
- C = >= 50% y <70
- D = >= 30% y <50
- E = < 30%

Security hotspots reviewed (security_hotspots_reviewed): El porcentaje de hotspots de seguridad revisados. La fórmula de proporción utilizada para el cálculo es la siguiente:

$$\frac{Número\ de\ \ hotspots\ revisados\ x\ 100}{Hotspots\ por\ revisar\ +\ hotspots\ revisados}$$

Tamaño

Las métricas relacionadas con el tamaño son utilizadas para obtener una medida cuantitativa del tamaño del código del proyecto y es útil para evaluar la escala y la complejidad del software analizado en términos de la cantidad de líneas de código. El seguimiento y análisis de la métrica de tamaño puede ser útil para comprender la complejidad y el mantenimiento del software a medida que va evolucionando con el tiempo.

La métrica de tamaño además de las líneas incluye el número de clases, métodos o funciones presentes en el código.

Clase (classes): El número de clases incluidas las anidadas, interfaces, enums y anotaciones.

Líneas comentadas (comment_lines): Número de líneas que contienen comentarios o código comentado.

En el caso de Java, las cabeceras de los ficheros no se cuentan como líneas comentadas ya que comúnmente son utilizadas para definir el tipo de licencia.

```
/**                                          +0 => empty
comment line
 *                                           +0 => empty
comment line
 * This is my documentation                  +1 =>
significant comment
 * although I don't                          +1 =>
significant comment
 * have much                                 +1 =>
significant comment
 * to say                                    +1 =>
significant comment
 *                                           +0 => empty
comment line
 *************************                  +0 => non-
significant comment
 *                                           +0 => empty
comment line
 * blabla...                                 +1 =>
significant comment
 */                                          +0 => empty
comment line

/**                                          +0 => empty
comment line
 * public String foo() {                     +1 =>
commented-out code
 *   System.out.println(message);            +1 =>
commented-out code
 *   return message;                         +1 =>
commented-out code
 * }                                         +1 =>
commented-out code
 */                                          +0 => empty
comment line
```

Ilustración 13 - Ejemplo de recuento de comentarios

Comentarios (%) (comment_lines_density): Esta métrica se calcula con la siguiente formula:

$$\frac{Lineas\ de\ comentario}{(Lineas\ de\ código + líneas\ de\ comentario)\ x\ 100}$$

Si se obtiene un 50% significa que el número de líneas de código es igual al número de líneas de comentario, mientras que si se tiene un 100% significa que el archivo sólo contiene líneas de comentario.

Directorios (directories): Numero de directorios que contiene el repositorio analizado.

Fichero (files): El número de ficheros.

Líneas (lines): El número de líneas físicas (número de retornos de carro).

Líneas de código (ncloc): El número de líneas físicas que contienen al menos un carácter que no es ni un espacio en blanco ni una tabulación ni parte de un comentario.

Funciones (functions): El número de funciones. Dependiendo del lenguaje, una función se define como una función, un método o un párrafo. En caso de Java, los métodos en clases anónimas son ignorados.

Sentencias (statements): El número de sentencias totales.

Test

Esta métrica se refiere a la evaluación de las pruebas unitarias y de integración que se ejecutan en el proyecto analizado. Proporciona información sobre la cobertura de los tests, es decir, la cantidad de código que ha sido probado testeado mediante casos de test.

Los tests ayudan a verificar que el código se comporte como se espera y que las funcionalidades se mantengan correctamente a medida que el software va evolucionando. Una buena cobertura de tests asegura que se han considerado múltiples casos y escenarios, lo que aumenta la confianza en la calidad y la estabilidad del software.

Cobertura de condición (Branch_coverage): Esta métrica se refiere a la evaluación de las expresiones booleanas en cada línea de código respondiendo a la pregunta: *¿Se ha evaluado cada expresión booleana tanto como verdadera como falsa?*

La cobertura de condición representa la cantidad de condiciones posibles en las estructuras de control de flujo que se han seguido durante la ejecución de las pruebas unitarias. Con la siguiente formula que utiliza las siguiente tres variables, se calcula la cobertura:

CT = condiciones que han sido evaluadas como 'verdaderas' al menos una vez.

CF = condiciones que han sido evaluadas como 'falsas' al menos una vez.

B = número total de condiciones.

$$Condition\ coverage = \frac{CT + CF}{2\ x\ B}$$

Cobertura (coverage): Una mezcla entre de cobertura de línea y la cobertura de condición. Pretende dar un resultado más preciso sobre que parte del código ha sido cubierta mediante las pruebas unitarias. A la formula anterior se le añaden dos variables más:

LC = líneas cubiertas = líneas cubiertas - líneas descubiertas

EL = número total de líneas ejecutables (líneas_a_cubrir)

$$Coverage = \frac{CT + CF + LC}{2\ x\ B + EL}$$

Cobertura de línea (line_coverage): Pretende contabilizar que lineas de código han sido ejecutadas mediante las pruebas unitarias. Por lo tanto, es la densidad de lineas cubiertas por los tests unitarios.

$$Linea\ coverage = \frac{LC}{EL}$$

Condiciones sin cubrir (uncovered_conditions): El número de condiciones que no han sido cubiertas por los tests unitarios.

Tests unitarios (tests): El número de tests unitarios que hay en el proyecto analizado.

Duración de los tests unitarias (test_execution_time): El tiempo necesario para ejecutar todos los tests unitarios.

Errores de los tests unitarios (test_errors): El número de tests unitarios que han fallado.

Fallos de los tests unitarios (test_failures): El número de tests unitarios que han fallado con una excepción inesperada.

Densidad de éxito de las pruebas unitarias (%) (test_success_density): Esta métrica evalúa qué tan bien están funcionando los tests unitarios en el proyecto analizado. Cuenta el número total de pruebas unitarias, la cantidad de errores y fallos encontrados durante la ejecución de las pruebas.

$$Test\ success\ density = \frac{Unit\ tests - (Unit\ test\ errors + Unit\ test\ failures)}{Unit\ tests\ x\ 100}$$

4 ANÁLISIS DEL PROYECTO 1: JAVA SPRING

El proyecto 1 utiliza para la gestión de dependencias Apache Maven, así que para realizar el análisis se ha creado el proyecto SonarQube como un proyecto Maven. Una vez creado, se obtiene el comando que se muestra en la Ilustración 14. Además de los parámetros por defecto, se añadió el parámetro *sonar.projectVersion, lo* cual permite que se guarde cada análisis con la versión del proyecto que le asignes para asi seguir un registro en el histórico que hay en el panel de administración y el parámetro *sonar.sources* para indicar desde que directorio comenzará a analizar. Ejecutando este comando por consola, comienza el análisis del proyecto, el cual, al terminar, reflejará los resultados en el panel de administración de SonarQube.

Ilustración 14 - Comando sonar para un repositorio con Maven

En el caso de un proyecto Maven, se recomienda realizar el análisis con el analizador de SonarScanner for Maven, ya que a través de este hace que se puede ejecutar el análisis sin necesidad de descargar manualmente, configurar y mantener una instalación del analizador SonarQube. La compilación de Maven ya contiene gran parte de la información necesaria para que SonarQube analice correctamente un proyecto. Al preconfigurar el análisis basándose en esa información, la necesidad de configuración manual se reduce significativamente [33].

4.1 Resultados

El análisis inicial, con la configuración por defecto, pasa la puerta de calidad mostrando, 302 bugs, 8 vulnerabilidades y 77 security hostpots. Una vez realizado el primer análisis, SonarQube nos ofrece un tiempo estimado que puede tomar corregir cada uno de los

problemas detectados. En este caso, sin tener en cuenta los code smells hacía una estimación total de 6 días y 1 hora. Finalmente fue menos tiempo ya que la mayoría de los errores eran el mismo error en diferentes ficheros.

Inicialmente se han revisado los security hotspots ya que algunos se han podido corregir y otros se ha decidido no realizar ninguna acción sobre ellos ya que como la aplicación no está expuesta a internet, no supone un riesgo de seguridad a considerar.

En la siguiente imagen, se puede observar el número de errores según su tipo y su nivel de gravedad. Además, dado la cantidad de errores críticos se decidió no contemplar los code smells, dado que la gran mayoría se debían a que no se estaba indicando el tipo de dato a contener a la hora de crear listas, pilas o hash maps. Por lo que se ha configurado una regla específica para analizar este proyecto de manera que este tipo de errores fueran excluidos del análisis.

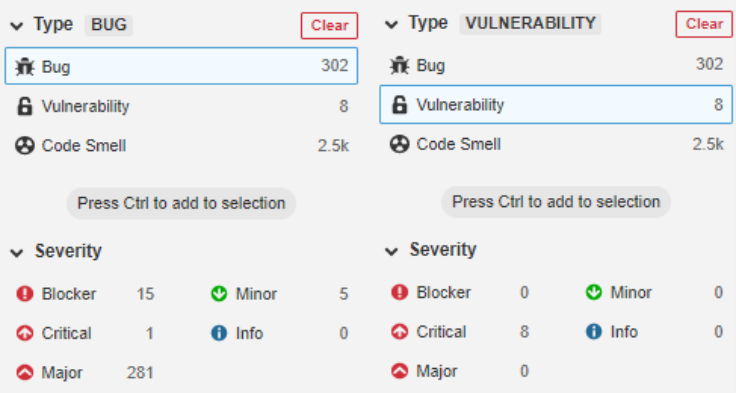

Ilustración 15 – Errores detectados en el primer análisis del proyecto 1

Algunos de los hostpots destacables que se encontraron son referentes al cifrado y al desactivar alguna de las políticas de seguridad.

1. Cross-Site Request Forgery (CSRF): Make sure disabling Spring Security's CSRF protection is safe here. Disabling CSRF protections is security-sensitive [34].
2. Weak Cryptography: Make sure that using this pseudorandom number generator is safe here. Using pseudorandom number generators (PRNGs) is security-sensitive [35].

Los ataques de CSRF, también llamados ataques de falsificación de petición en sitios cruzados, ocurre en entornos web cuando un atacante engaña a un usuario para que realice una acción no deseada en un sitio web en el que el usuario está autenticado.

El segundo hotspot que se menciona, resulta interesante porque hace referencia a la propia librería de java, la cual, indica que el método *random* de la librería *java.util.Random* se basa en una generador de números pseudoaleatorios puede generar valores predecibles. De manera que un atacante podría terminar adivinando el siguiente valor que se generaría.

Una vez añadida la regla para evitar que aparezcan los code smells en el análisis del resultado fue el siguiente.

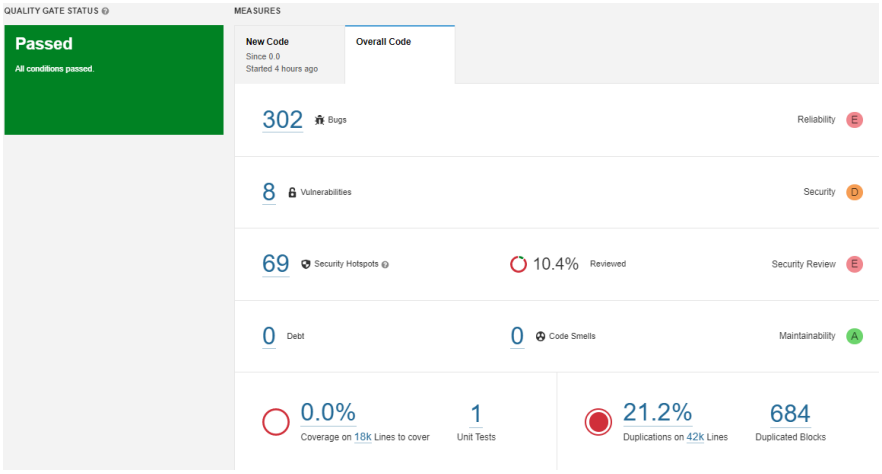

Ilustración 15 - Resultado del análisis al proyecto 1

Una vez realizado esto, se ha centrado la atención en subir un parche para arreglar inicialmente las vulnerabilidades y después los bugs. De todo el conjunto de los errores, se van a explicar los que más reportes e incidencias estaban generando en la plataforma de soporte técnico del cliente.

Replace this persistent entity with a simple POJO or DTO object

[Vulnerability][Critical] Replace this persistent entity with a simple POJO or DTO object Persistent entities should not be used as arguments of "@RequestMapping" methods.

Esta vulnerabilidad aparece en los métodos asociados a las llamadas REST GET y POST que recibían un objeto como parámetro de entrada en el body de la petición pero estaba tipado como un objeto entidad, o sea, en la implementación de la clase estaban etiquetados con la anotación *Entity* del paquete *javax.persistence*, indicando que dicha clase es una entidad JPA, por lo que java asume que se mapeara a una tabla en base de datos llamada con el nombre de la clase [36].

```
@PostMapping("/add-linea/{idOrden}")
public ResponseEntity<?> addLinea(@PathVariable("idOrden") BigInteger idOrden,  @RequestBody LineaOrdenTrabajo linOrden)
```

Ilustración 16 - Ejemplo del error replace this persistent entity POJO

Spring vincula automáticamente los parámetros de una solicitud a los objetos declarados como argumentos de los métodos anotados con *RequestMapping*. Esta característica posibilita que se introduzcan campos que no se están controlando. Por otra parte, los objetos persistentes, en este caso de tipo *Entity*, se vinculan automáticamente a su correspondiente tabla de la base de datos y se actualizan automáticamente mediante Hibernate.

Por lo tanto, queda expuesto por un lado la problemática de recibir un objeto en la llamada con atributos distintos a la entidad lo cual genera un error en tiempo de ejecución y por otro lado el que un atacante pueda modificar cualquiera campo de la tabla de la base de datos relacionada a esa entidad simplemente añadiendo el campo en el objeto enviado en la petición.

La solución recomendada y que ha sido aplicada fue crear objetos DTO para cada entidad que estaba siendo utilizada en las peticiones REST, de esta manera se restringe el control de que campos son enviados o recibidos y una vez esto ya se pueden tratar como se desee antes de realizar su acción de persistencia en la base de datos.

Resources should be closed

[Bug][Blocker] Use try-with-resources or close this "BufferedReader" in a "finally" clause: Resources should be closed.

Este bug aparece en múltiples partes del proyecto dónde hay objetos que gestionan archivos, en este caso: *BufferedOutputStream*, *DataInputStream* y *XSSFWorkbook*. Presenta un problema común en el manejo de recurso en Java. Los objetos que gestionan recursos como pueden ser conexiones, flujos de datos y archivos entre otros implementan una interfaz denominada *Closeable*. Estos objetos deben ser cerrados una vez se han terminado de utilizar para que los recursos que están utilizando sean liberados.

```
public static String contentToString(File file) {
    try {
        FileReader fileReader = new FileReader(file);
        BufferedReader lector = new BufferedReader(fileReader);
        return cadena.toString();
    } catch (IOException e) {
        logger.error("Error extrayendo el contenido del fichero: " + file.getAbsolutePath(), e);
```

Ilustración 17 - Ejemplo error resources should be closed

No cerrar adecuadamente los recursos conlleva que se produzcan fugas de recursos, lo cual significa que puedan ir acumulándose con el tiempo y causar diferentes tipos de problemas [37].

Algunos de los problemas más comunes ocasionados por esta mala gestión de los recursos son:

- Fugas de memoria, lo cual provoca que se vayan acumulando gradualmente la cantidad de recursos utilizados por la aplicación consumiendo así la memoria del disponible.
- Recursos bloqueados, al permanecer el recurso abierto es posible que quede bloqueado y no esté disponible para su posterior utilización. Un ejemplo típico son las conexiones a las bases de datos, cuando no son cerradas adecuadamente, el servidor de base de datos acumula el número de conexiones existentes de manera que cuando llega al límite rechaza el resto de las conexiones que se hagan hasta que alguna conexión quede liberada.
- Pérdida de datos, en el caso de los archivos, al no cerrar el flujo de datos puede corromperse el fichero ocasionando así una pérdida parcial o completa de los datos del fichero.

Una de las soluciones para reparar este error es añadir una cláusula *finally* en la que se cierre el objeto que está gestionando el recurso, en este caso, *BufferedReader,* con la sentencia *stream.close()*. De esta manera garantizamos que el recurso se cerrará automáticamente al finalizar el bloque *try.*

Null pointers should not be dereferenced

[Bug][Major] A "NullPointerException" could be thrown; "albaranVenta" is nullable here.

Este es uno de los errores más comunes y trata sobre el tratamiento incorrecto de objetos que pueden ser nulos. Presentan un grave error ya que realizar una operación en un objeto cuya referencia es nula provoca un error en tiempo de ejecución, *NullPointerException.* Este error si no es controlado provoca un cierre inesperado de la aplicación.

```
List<AlbaranVenta> albaran = albaranVentaService.getByNumero( 2 albaranVenta.getNumero());
```

Ilustración 18 - Ejemplo null poointers should not be dereferenced

En este caso, se está obteniendo el atributo *Numero* del objeto *albaranVenta,* el cual podía ser nulo en las sentencias anteriores a la mostrada en la captura anterior. La solución a este error común es sencilla, simplemente requiere que se haga una comprobación previa a la utilización del objeto para saber si su valor es nulo. En caso de que sea nulo, no se debería realizar ninguna operación sobre el objeto o inicializarlo antes de realizar alguna.

Call “Optional#isPresent()” or “!Optional#isEmpty()” before accessing the value

[Bug][Major] Call "Optional#isPresent()" or "!Optional#isEmpty()" before accessing the value: Optional value should only be accessed after calling isPresent().

Este bug está muy relacionado con el explicado anteriormente, lo que lo hace diferente es la utilización del objeto *Optional<T>.* Este tipo de objetos representan un contenedor de otra clase la cual es su valor, en este caso, el método *getOne* del servicio *albaranVenta-Service* devuelve un objeto de tipo *Optional<AlbaranVenta>.* Lo característico de este tipo de objetos es que presentan diferentes métodos los cuales ayudan al desarrollador a trabajar de una manera más segura los objetos que pueden ser nulos. Presentan dos tipos de estado, *Present* y *Empty*. Para acceder al valor que contiene el objeto *Optional* simplemente habría que utilizar su método *get* como se ve en la imagen siguiente [38].

```
if (!albaranVentaService.existsById(id)) {
    return new ResponseEntity<>(new Mensaje("No existe la albaran venta"), HttpStatus.NOT_FOUND);
}

AlbaranVenta albaranVenta = albaranVentaService.getOne(id).get();
```

Ilustración 19 - Ejemplo null error con java Optional

La problemática del uso incorrecto de este tipo de dato, presentada en la imagen anterior es la misma que se detalla en el punto anterior. Antes de acceder al valor que contiene el objeto *Optional* mediante su método *get*, es posible hacer la comprobación de que contiene algún valor utilizando el método *isPresent*. Por lo tanto, el error detectado por el analizador es dado porque se está llamando al método *get* sin haber realizado la comprobación de si hay un valor presente, lo que puede provocar que, si realmente el objeto *Optional* no contiene un objeto *AlbaranVenta*, al acceder a él, ocurrirá un error *NullPointerException*.

A diferencia del punto anterior, para hacer la comprobación de que el objeto contenido no es nulo se puede utilizar el método *isPresent* o *isEmpty* de la clase *Optional*. De esta manera, una vez garantizado que existe un valor, poder realizar la operación *get* para obtener el objeto contenido y utilizarlo como se desee.

Finalmente, una vez realizadas las acciones convenientes para cada uno de los problemas. Desde la pestaña *activity* del panel de administración del proyecto SonarQube tenemos acceso a información resumida dónde se muestran los resultados para cada uno de los análisis realizados durante el proceso.

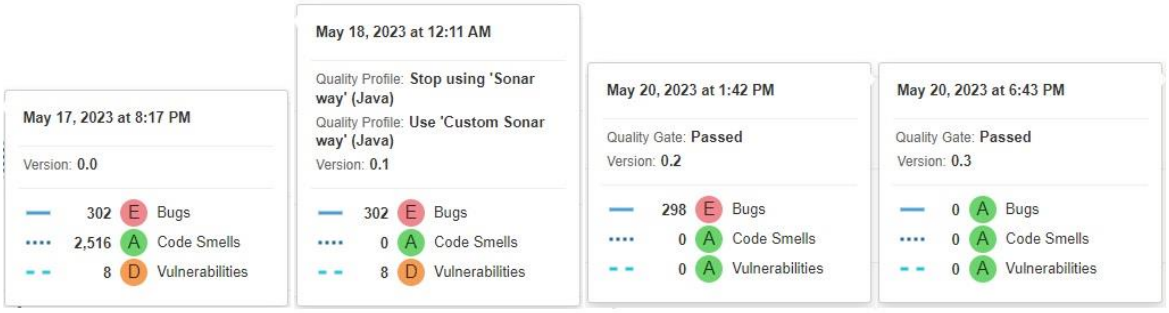

Ilustración 20 - Resumen resultados de los análisis del proyecto 1

Durante el proceso, se realizaron 4 análisis, resumidos en la Ilustración 20, uno inicial, un segundo en el que se añadió la regla para ignorar los code smells, con este paso se consiguió que no aparecieran en el siguiente análisis para poder centrarse en los que realmente estaban provocando un problema para el cliente, los bugs y las vulnerabilities.

En el tercer análisis se dio solución a las 8 vulnerabilidades y algunos bugs que estaban relacionados y finalmente un cuarto en el que se corrigieron el resto de los bugs.

5 ANÁLISIS DEL PROYECTO 2: ANGULAR

El proyecto 2, es un proyecto angular que utiliza como lenguaje Typescript y como gestor de dependencias está utilizando NPM. En este caso el proyecto de SonarQube ha tenido que ser creado como un proyecto genérico, el cual incluye lenguajes de programación como: Javascript, Typescript, Go, Python o PHP. Al igual que en la creación del proyecto 1, Sonaqube nos facilita un prompt con el comando sonar correspondiente para esa configuración.

Ilustración 21 - Comando sonar para un repositorio JS/TS

Para los proyectos que no tienen un analizador específico para su sistema de build, SonarQube utiliza SonarScanner como analizador por defecto para analizar el repositorio de código.

En este caso, se ha creado un fichero llamado *sonar-project.properties* en la raíz del proyecto, en el cual se configuran los parámetros con los que se quiere que ejecute el comando sonar-scanner. El fichero de configuración tenía la siguiente forma:

```
sonar.projectKey=proyecto-1-front
sonar.projectName=proyecto-1-front
sonar.token=sqp_d276a66998010c0828a6211d1ae6193bfa057faf
sonar.host.url=http://localhost:9000
sonar.projectVersion=0.2
sonar.sources=./src/
```

Ilustración 22 - Ejemplo de fichero sonar-project.properties

A diferencia del proyecto 1, como el lenguaje sobre el que se programa ofrece mayor flexibilidad a la hora programar, permitiendo tipado dinámico, o que las funciones no tengan un valor de retorno consistente puede provocar errores difíciles de detectar durante la etapa de desarrollo y aumenta la complejidad al trabajar en equipo.

Por lo tanto, se han tenido en gran consideración los errores detectados como code smells en este caso, ya que esta falta de consistencia en la escritura del código estaba provocando muchas perdidas de recursos para la empresa.

5.1 Resultados

Con el análisis de este proyecto se ha conseguido mayoritariamente eliminar gran cantidad de código duplicado en los diferentes componentes de la aplicación, gran cantidad de código comentado innecesario que pertenecía a versiones anteriores, importaciones de módulos que no estaban siendo utilizados y diferentes errores que aparecían al haber conflictos con reglas de estilos CSS.

Por otra parte, también había componentes que ya no se utilizaban en ninguna parte del proyecto y algunos componentes que referenciaban a otros componentes que ya no existían.

El análisis inicial presentado en la siguiente imagen muestra 24 bugs y 169 code smells.

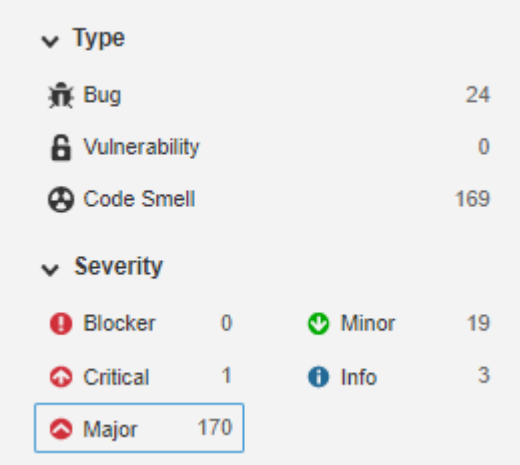

Ilustración 23 - Errores detectados en el primer análisis del proyecto 2

De todo el conjunto de errores, se van a explicar los que se ha considerado que afectaba en mayor manera al software, por supuesto, teniendo en cuenta el nivel de gravedad y ocurrencia.

Shorthand properties that override related longhand properties should be avoided

[Bug][critical] Unexpected shorthand "padding" after "padding-left": Shorthand properties that override related longhand properties should be avoided.

Este error aparece con diferentes reglas de estilos, indica que al insertar una regla *padding* para ajustar un espacio entre los elementos HTML, si se inserta otra regla *padding* explícita como pueda ser *padding-left* provocará que una sobrescriba la otra [40].

Las recomendaciones ante este tipo de errores son claras, se debe mantener una estructura de código clara y coherente de manera que las reglas de estilo no interfieran entre sí. Este tipo de errores causan una importante pérdida de tiempo cuando el desarrollador no sabe qué reglas de estilo tienen mayor prioridad o cuales están siendo aplicadas en cada componente.

Properties should not be duplicated

[Bug][Major] Unexpected duplicate "padding-right"

Las reglas duplicadas son un claro fallo del desarrollador, que ensucian el código y pueden crear confusión a futuros desarrolladores que se integren en el proyecto. Simplemente la última instancia de un nombre duplicado determinará el valor que obtendrá la regla, en este caso *padding-right* [41].

Functions should use "return" consistently

[Code Smell][Major] Unlike strongly typed languages, JavaScript does not enforce a return type on a function.

Como se comentó previamente, dada la flexibilidad que ofrece TypeScript, la funciones no tienen por qué imponer un valor de retorno. Esto provoca que dentro de una misma función se puedan tomar diferentes caminos que devuelvan variables de diferente tipo o incluso que un camino devuelva un valor y por el otro no se realice ninguna sentencia de *return* [42].

Este tipo de error especialmente puede resultar confuso sobre todo para desarrolladores que vengan de otros tipos de lenguajes dónde la sintaxis sea más formal y estructurada como en el caso de Java en el proyecto 1. Esto supone que el comportamiento impredecible de la sentencia *return* puede dificultar la comprensión y el uso correcto de la función.

Esto puede conllevar a errores en tiempo de ejecución, ya que el desarrollador que utilice la función espera que devuelva un tipo de valor concreto y dada esta flexibilidad no sabrá con exactitud el tipo de valor va a recibir o si no recibirá ninguno.

La solución recomendada, es dar un tipo de valor de retorno claro y consistente. En caso de que se requiera cierta flexibilidad se debería indicar en la función que el valor de retorno puede ser *undefined* o *null* con el operador | .

Attributes deprecated in HTML5 should not be used

[Code Smell][Major] Remove this deprecated "align" attribute.

En este caso se han dejado obsoletos muchos atributos antiguos. Para garantizar la mejor experiencia de usuario, no se deben utilizar atributos obsoletos. Esta regla verifica la presencia de los siguientes atributos HTML obsoletos, los cuales deberían ser reemplazados por las reglas de estilo CSS correspondientes [43].

Two branches in a conditional structure should not have exactly the same implementation

[Code Smell][Major] This branch's code block is the same as the block for the branch on line 41.

Como se muestra en la siguiente imagen, este error aparece cuando en un bloque *switch* o en una cadena de bloques *if* implementan el mismo código. Este error de duplicidad de código es una práctica propensa a errores por parte del programador que afecta a la mantenibilidad del código. En una cadena de *if* deberían combinarse, o en un *switch* uno debería pasar al otro sin interrupción [44].

```
if (control.hasError(typeA)) 1 {
    let message = this.defaultMessages.get(typeA);
    return this.replaceMessageTemplate(message, fieldName);
}
else if(control.hasError(typeB)) {
    let message = this.defaultMessages.get(typeB);
    return this.replaceMessageTemplate(message, fieldName);
}
```

Ilustración 24 - Ejemplo de if-else con error

Ternary operators should not be nested

[Code Smell][Major] Extract this nested ternary operation into an independent statement.

La anidación de operadores ternarios puede parecer comprensible en el momento de escribirlo para el desarrollador que lo escribe, pero con el paso del tiempo, puede generar confusión y dificultades para los desarrolladores encargados del mantenimiento del código. Esta práctica, la cual ni es común ni recomendable, puede provocar situaciones de frustración para los programadores que intenten comprender su funcionamiento, además de la pérdida de tiempo y propensión a errores que supone el no comprender correctamente el objetivo de la sentencia [45].

```
        let msg = entity.nombre ? entity.nombre : entity.nombreFiscal ? entity.nombreFiscal :
entity.codigo ? entity.codigo : entity.id;
```

Ilustración 25 - Ejemplo de error de operador ternario anidado

En lugar de utilizar la anidación de operadores ternarios, se recomienda priorizar la claridad del código. Una de las alternativas que se puede realizar es añadir una línea adicional para expresar la operación anidada como una declaración separada. Evitar este tipo de anidamiento reduce esta complejidad excesiva lo cual facilita la labor de mantenimiento en un futuro y promueve un código más comprensible.

En la siguiente imagen, se muestra el resumen de los tres analisis que se han realizado durante el proceso.

Ilustración 26 - Resumen resultados de los análisis del proyecto 2

La primera tarjeta representa el análisis inicial, la segunda tarjeta el segundo análisis después de arreglar algunos de los errores y finalmente, la tercera dónde se acaban de arreglar los últimos bugs. No se dio solución a todos los code smells, ya que como se comentó inicialmente, dada la flexibilidad del lenguaje, el equipo de desarrollo había adoptado

algunas prácticas que no son recomendadas por SonarQube. Realizar modificaciones sobre estas prácticas podría generar un impacto más negativo que positivo para el proyecto. Por lo tanto, se decidió no solucionar todos los code smells identificados, considerando el contexto y las implicaciones potenciales en el proyecto.

6 CONCLUSIONES

Como se ha podido ver durante el trabajo, desarrollar software seguro es esencial para que nuestras aplicaciones estén libres de posibles vulnerabilidades y ataques maliciosos. Seguir buenas prácticas y aplicar reglas ya sean internas a la empresa desarrolladora o por estándares conlleva una gran cantidad de beneficios los cuales llevan a obtener un producto final con menor número de errores, garantizando así una mejor calidad del producto que sale al mercado. De esta manera, se puede preservar la integridad, confidencialidad y disponibilidad de la información gestionada por las aplicaciones que desarrollamos invirtiendo menor tiempo en mantenimientos innecesarios y resolución de errores los cuales llevan a que los usuarios finales pierdan la confianza en nuestros productos y en peor caso, que se dañe la imagen de la empresa encargada de la aplicación.

Por otra parte, se ha visto que hay herramientas que facilitan la tarea de mantener el software más seguro realizando análisis que generan reportes acerca del estado del código, indicando código duplicado, diferentes errores encontrados con posibles soluciones a ellos y otras características diversas que ayudan a comprender mejor los repositorios sobre los cuales se está trabajando, concretamente, en el proyecto se ha visto la herramienta SonarQube que utiliza el análisis estático.

Para concluir, desarrollar un software de calidad de principio a fin es costoso y hay muchos factores a los que hay que prestar atención desde las fases iniciales para poder garantizar que pequeños problemas no acaben suponiendo un riesgo en el futuro. Hay que saber dividir las fuerzas invertidas en un proyecto y utilizar todas las herramientas necesarias para no dejar ninguna fase del desarrollo sin cubrir. No solo basta con que un software realice la función para la cual ha sido desarrollada, sino que hay más factores a los que hay que prestar atención para terminar lanzando un producto final de calidad, estable y seguro.

REFERENCIAS

[1] Reglamento general de protección de datos, Reglamento, *Accedido el 10/07/2023: https://europa.eu/youreurope/business/dealing-with-customers/data-protection/data-protection-gdpr/index_es.htm*

[2] Static Program Analysis, Anders Møller and Michael I. Schwartzbach, Libro, *https://cs.au.dk/~amoeller/spa/*

[3] OWASP Top 10 2021, *Accedido el 07/07/2023:* Synopsys application security, *https://www.synopsys.com/glossary/what-is-owasp-top-10.html*

[4] CWE Top 25 Most Dangerous Software Weaknesses, *Accedido el 07/07/2023: https://cwe.mitre.org/top25/index.html*

[5] CWE About us, *Accedido el 07/07/2023: https://cwe.mitre.org/about/index.html*

[6] About the CVE Program, Website, *Accedido el 07/07/2023: https://www.cve.org/About/Overview*

[7] Making Sense of Vulnerabilities and Software Weaknesses with CVE, CWE, CVSS and CWSS, Blog, *Accedido el 07/07/2023: https://www.automox.com/blog/vulnerabilities-software-weaknesses-acronym-breakdown*

[8] About Sonarsource, Website, *Accedido el 25/06/2023: https://www.sonarsource.com/company/about/*

[9] SonarQube in Action, Chapter 1, page 4-7 Libro

[10] Security reports User Guide, Guía, *Accedido el 25/06/2023: https://docs.SonarQube.org/10.0/user-guide/security-reports*

[11] OWASP Top 10 list 2021, *Accedido el 14/07/2023: https://owasp.org/Top10/*

[12] SonarQube in Action, Chapter 1, page 13-14, Libro

[13] Try out SonarQube, documentación, *Accedido el 25/06/2023: https://docs.sonarsource.com/sonarqube/10.0/try-out-sonarqube/*

[14] White-Box Testing Automation With SonarQube, M. J. Andrade, Paper, *Accedido el 14/07/2023: https://www.semanticscholar.org/paper/White-Box-Testing-Automation-With-SonarQube-Andrade/673983865870426029b576256a2e1ddf036a41ca*

[15] SonarQube Security Hotspots, Guía, *Accedido el 25/06/2023: https://docs.sonarqube.org/10.0/user-guide/security-hotspots/*

[16] SonarQube in Action, Chapter 2, Page 30, Libro

[17] SonarQube issues, Guía, *Accedido el 26/06/2023: https://docs.sonarqube.org/10.0/user-guide/issues/*

[18] SonarQube in Action, Chapter 2, page 59, Libro

[19] SonarQube in Action, Chapter 2, page 31, Libro

[20] Sonarsource Bugs, Documentación, *Accedido el 26/06/2023: https://rules.sonarsource.com/java/type/Bug/*

[21] Sonar to idenfity security vulnerabilities, Sonarsource, Blog, *Accedido el 26/06/2023: https://www.sonarsource.com/blog/sonar-to-identify-security-vulnerabilities/*

[22] How to detect vulnerabilities in your code, Bitegarden, Blog, *Accedido el 26/06/2023: https://www.bitegarden.com/how-to-detect-vulnerabilities-in-your-code*

[23] Sonarsource vulnerabilities, Documentación, *Accedido el 26/06/2023: https://rules.sonarsource.com/java/type/Vulnerability/*

[24] SonarQube user guide concepts, Documentación, *Accedido el 26/06/2023: https://docs.sonarqube.org/10.0/user-guide/concepts/*

[25] Everything you need to know about code smells, Codegrip, Blog, *Accedido el 27/06/2023: https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/*

[26] Avoiding common code smells in c with sonarqube, Medium, Blog, *Accedido el 27/06/2023: https://medium.com/c-sharp-progarmming/avoiding-common-code-smells-in-c-with-sonarqube-4c19f2bb4b39*

[27] Sonarsource code smells, Documentación, *Accedido el 27/06/2023: https://rules.sonarsource.com/java/type/Code%20Smell/*

[28] SonarQube rules overview, Guía, *Accedido el 27/06/2023: https://docs.sonarqube.org/10.0/user-guide/rules/overview/*

[29] SonarQube built in rule tags, Documentación, *Accedido el 27/06/2023: https://docs.sonarqube.org/10.0/user-guide/rules/built-in-rule-tags/*

[30] Sonarsource tags, Documentación, *Accedido el 27/06/2023: https://rules.sonarsource.com/java/tag*

[31] Sonarsource metric definitions, Guía, *Accedido el 11/07/2023: http://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions*

[32] Sonarsource quality profiles, Documentación, *Accedido el 11/07/2023: https://docs.sonarsource.com/sonarqube/latest/instance-administration/quality-profiles/*

[33] SonarScanner for Maven, Documentación, *Accedido el 10/06/2023: https://docs.SonarQube.org/latest/analyzing-source-code/scanners/sonarscanner-for-maven/*

[34] Security Hotspot RSPEC-4502, Documentación, *Accedido el 10/06/2023: https://rules.sonarsource.com/java/RSPEC-4502*

[35] Security Hotspot RSPEC-2245, Documentación, *Accedido el 10/06/2023: https://rules.sonarsource.com/java/RSPEC-2245*

[36] Vulnerability RSPEC-4684, Documentación, *Accedido el 10/06/2023: https://rules.sonarsource.com/java/type/Vulnerability/RSPEC-4684*

[37] Bug RSPEC-2095, Documentación, *Accedido el 10/06/2023: https://rules.sonarsource.com/java/type/Bug/RSPEC-2095*

[39] Bug RSPEC-3655, Documentación, *Accedido el 10/06/2023: https://rules.sonarsource.com/java/type/Bug/RSPEC-3655/?search=Call%20Optional*

[40] Bug RSPEC-4657, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/css/RSPEC-4657*

[41] Bug RSPEC-4656, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/css/RSPEC-4656*

[42] Code smell RSPEC-3801, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/javascript/RSPEC-3801*

[43] Code smell RSPEC-1827, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/html/RSPEC-1827*

[44] Code smell RSPEC-1871, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/typescript/RSPEC-1871*

[45] Code smell RSPEC-3358, Documentación, *Accedido el 11/06/2023: https://rules.sonarsource.com/typescript/RSPEC-3358*