Análisis de Código estático

Índice

1. Introducción

En el desarrollo de software moderno, la calidad del código y la seguridad son factores clave para garantizar aplicaciones mantenibles, robustas y seguras. La creciente complejidad de los sistemas, junto con el uso de tecnologías como contenedores, programación asíncrona y aplicaciones web, hace imprescindible el uso de herramientas que permitan detectar de forma temprana problemas de calidad y vulnerabilidades.

En este contexto, SonarQube se ha utilizado como herramienta de análisis estático para evaluar el código fuente del proyecto. A través de sus reglas y métricas, la herramienta permite identificar *code smells*, problemas de mantenibilidad, errores potenciales y riesgos de seguridad, proporcionando descripciones detalladas del impacto de cada problema y recomendaciones para su mitigación.

El análisis realizado con SonarQube ha permitido detectar distintos tipos de incidencias, como problemas de complejidad cognitiva, uso inseguro de expresiones regulares, malas prácticas en el manejo de promesas en JavaScript, vulnerabilidades web como CSRF, y configuraciones inseguras en entornos Docker. Toda esta información ha servido como base para documentar los riesgos asociados y proponer mejoras en el código y la configuración del sistema.

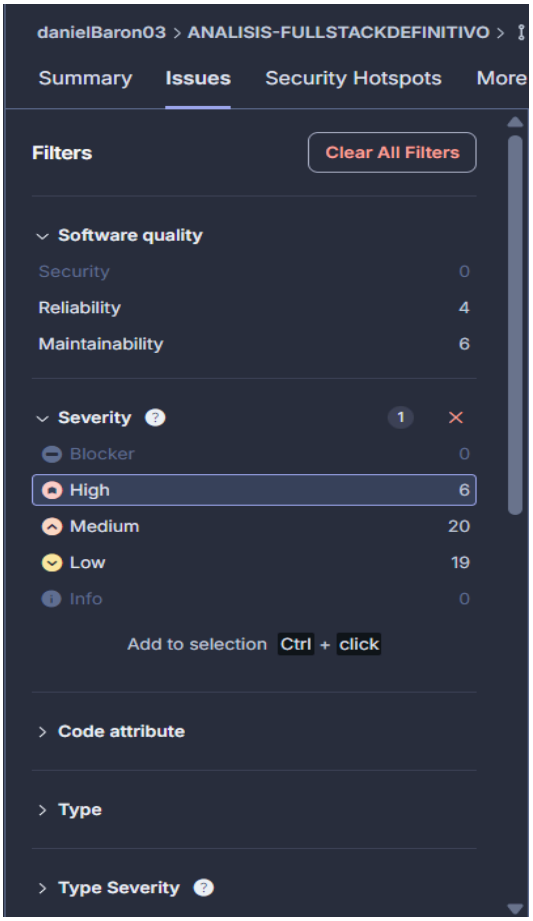
2. Sonar Qube

SonarQube es una plataforma de análisis estático de código que se utiliza para evaluar la calidad y seguridad del software de forma automática. Analiza el código fuente sin ejecutarlo y detecta errores (bugs), vulnerabilidades de seguridad, malas prácticas y code smells (problemas que dificultan el mantenimiento).

Se integra fácilmente en pipelines CI/CD y soporta múltiples lenguajes (Java, Python, JavaScript, C/C++, etc.). SonarQube ayuda a los equipos a mejorar la calidad del código, reducir deuda técnica y prevenir fallos de seguridad antes de que el software llegue a producción.

En vuestro contexto de laboratorio con máquinas virtuales, es ideal para simular auditorías de código y enseñar prácticas de desarrollo seguro sin riesgos reales.

Una vez que realizamos el análisis con la herramienta no encontramos lo siguiente:

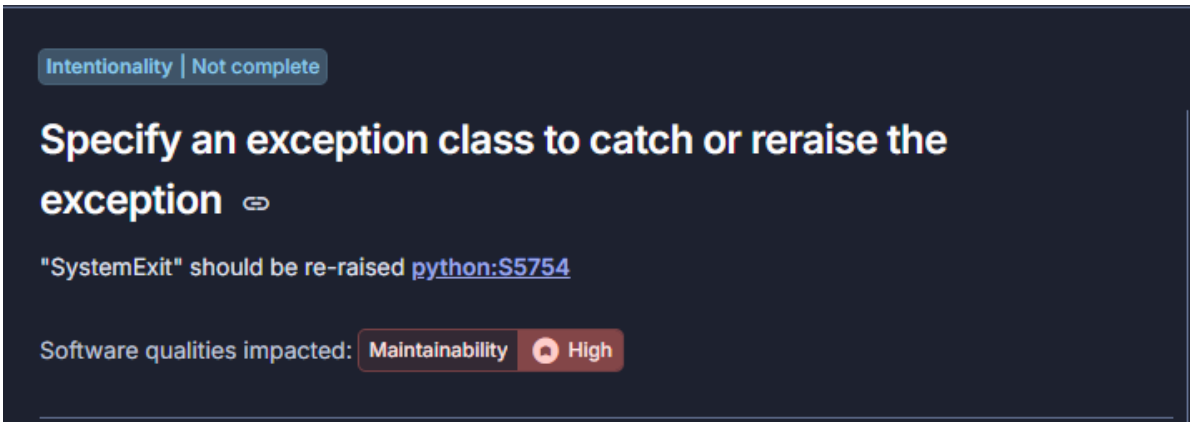


- No encontramos ninguna vulnerabilidad crítica.
- Encontramos un total de 6 vulnerabilidades altas.
- En cuanto a las vulnerabilidades de rango “mediano” se nos muestran un total de 20 vulnerabilidades.
- Por último, la herramienta nos muestra 19 vulnerabilidades bajas.

3. Vulnerabilidades

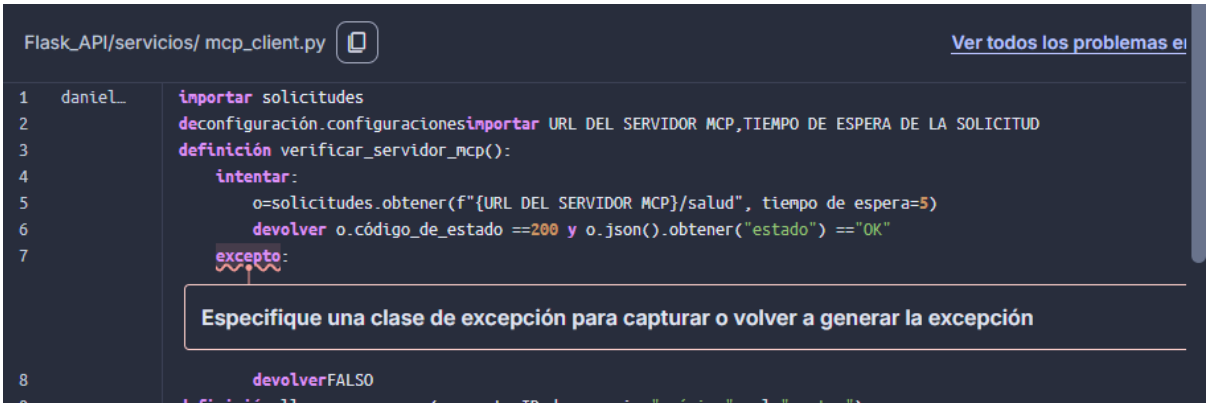
En este apartado “analizaremos” las vulnerabilidades más importantes, empezando por las “altas”.

Primera vulnerabilidad



Especificar una clase de excepción para capturar o volver a generar la excepción.

¿Dónde se encuentra el problema?



¿Por qué es esto un problema?

La excepción SystemExit se genera cuando se llama a sys.exit() desde un programa. Esta excepción se utiliza para indicar al intérprete que debe finalizar la ejecución. Se espera que la excepción se propague hasta que el programa se detenga por completo.

Es posible capturar esta excepción para realizar tareas adicionales, como operaciones de limpieza o liberación de recursos. Sin embargo, debe volver a generarse para permitir que el intérprete finalice la ejecución correctamente. No relanzar esta excepción puede provocar comportamientos no deseados.

Una sentencia except: simple, es decir, un bloque except sin especificar ninguna clase de excepción, es equivalente a except BaseException. Ambas capturan todas las excepciones, incluyendo SystemExit. Por este motivo, se recomienda capturar siempre excepciones más específicas. Si no es posible, la excepción debe relanzarse explícitamente.

Del mismo modo, también es recomendable volver a generar la excepción KeyboardInterrupt. Al igual que SystemExit, KeyboardInterrupt se utiliza para indicar al intérprete que debe finalizar la ejecución del programa (por ejemplo, cuando el usuario interrumpe el proceso manualmente). No relanzar esta excepción también puede dar lugar a comportamientos no deseados.

¿Cómo se puede solucionar?

Ejemplo de código no conforme:

```
intentar :
    ...
excepto SystemExit:   # No compatible: la excepción SystemExit no se vuelve a generar
    .

intentar :
    ...
excepto BaseException:   # No compatible: Las BaseExceptions abarcan excepciones SystemExit y deben volver a generarse
    .

intentar :
    ...
excepto :   # No conforme: las excepciones capturadas por esta declaración deben volver a generarse o debe capturarse una excepción más específica
    .
```

Ejemplo de posible solución:

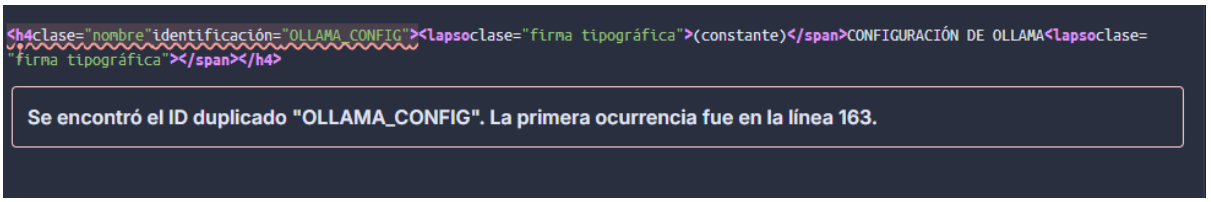
```
intentar :
    ...
excepto SystemExit como e:
    ...
    elevar e

intentar :
    ...
excepto BaseException como e:
    ...
    elevar e

intentar :
    ...
excepto FileNotFoundError:
    ... # Manejar una excepción más específica
```

Segunda vulnerabilidad

Se encontró el ID duplicado “OLLAMA_CONFIG”. La primera ocurrencia fue ocurrencia fue en la línea 163.



¿Por qué es esto un problema?

La especificación HTML exige que los valores del atributo id sean únicos dentro de un documento. Cuando varios elementos comparten el mismo atributo id, pueden aparecer diversos problemas:

- Los métodos de JavaScript, como document.getElementById(), solo devuelven el primer elemento coincidente, lo que provoca que los demás elementos con el mismo id queden inaccesibles.
- Los selectores CSS que apuntan a un id pueden no aplicar los estilos de forma consistente a todos los elementos previstos.
- Los lectores de pantalla y otras tecnologías de asistencia pueden confundirse al navegar entre elementos con identificadores duplicados.
- Las etiquetas de formulario que utilizan el atributo for pueden no asociarse correctamente con los controles de formulario correspondientes.

Estos problemas pueden dar lugar a fallos de funcionalidad, una mala experiencia de usuario y barreras de accesibilidad, especialmente para usuarios con discapacidades.

Impacto potencial:

El uso de atributos id duplicados puede interrumpir la funcionalidad del código JavaScript, generar inconsistencias en el estilo visual y barreras de accesibilidad. Como consecuencia, los usuarios pueden no ser capaces de interactuar correctamente con ciertos elementos de la página y las tecnologías de asistencia pueden no funcionar como se espera.

Ejemplo de código erróneo:

```
< div  id = "content" > Contenido principal </ div >
< div  id = "content" > Contenido secundario </ div >  <!-- No conforme: id duplicado -->
```

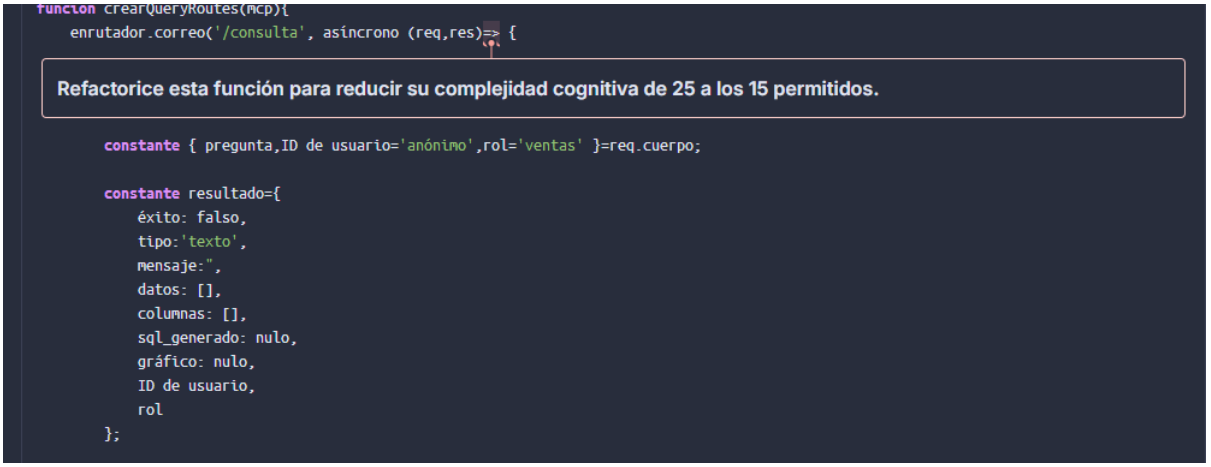
Ejemplo de solución:

```
< div  id = "main-content" > Contenido principal </ div >
< div  id = "secondary-content" > Contenido secundario </ div >
```

Tercera vulnerabilidad

Refactorizar esta función para reducir su complejidad cognitiva de 25 a los 15 permitidos.

¿Dónde está el problema?




```
    constante quiereGráfico=detectarPeticionGrafico(pregunta);
    constante tipoGrafico=quiereGráfico ? detectarTipoGrafico(pregunta) : nulo;

    si(quiereGráfico){
        consola.log(`>> Gráfico solicitado: ${tipoGrafico}`);
    }

    // Generar SQL
    consola.log('>> Generando SQL...');
    dejar SQL= esperargenerarSQL(pregunta);
    SQL=limpiarSQL(SQL);
    resultado.sql_generado =SQL;
    consola.log(`>> SQL: ${SQL}`);

    // Validar
    constante validación=validarSQL(SQL);
    si(!validación.válido){
        resultado.mensaje =`SQL rechazado: ${validación.mensaje}`;
        devolver res.json(resultado);
    }

    // Ejecutar
    consola.log('>> Ejecutando consulta...');
    constante respuesta= esperarmcp.consulta(SQL);

    si(!respuesta.éxito){
        resultado.mensaje =`Error: ${respuesta.error}`;
        devolver res.json(resultado);
    }
```

¿Por qué esto es un problema?

La complejidad cognitiva mide la dificultad para comprender el flujo de control de una unidad de código. Un código con una complejidad cognitiva elevada resulta más difícil de leer, entender, probar y modificar.

Como regla general, una alta complejidad cognitiva es un indicador de que el código debería refactorizarse en partes más pequeñas, claras y fáciles de mantener.

¿Qué sintaxis del código afecta a la puntuación de complejidad cognitiva?

La complejidad cognitiva aumenta cada vez que el código interrumpe el flujo de lectura lineal normal. Esto incluye, entre otros, los siguientes elementos:

- Estructuras de control como bucles y condicionales.
- Bloques de captura de excepciones.
- Estructuras switch.
- Saltos a etiquetas.
- Condiciones que combinan múltiples operadores lógicos.

Cada nivel adicional de anidamiento incrementa la complejidad. Durante la lectura del código, cuanto más se profundiza en las capas anidadas, mayor es el esfuerzo necesario para mantener el contexto y comprender el comportamiento del programa.

Las llamadas a métodos no incrementan la complejidad cognitiva. Un nombre de método bien seleccionado actúa como un resumen de varias líneas de código, permitiendo al lector obtener primero una visión general

del comportamiento del sistema y profundizar posteriormente en los detalles de las funciones llamadas.

¿Cuál es el impacto potencial?

Una complejidad cognitiva elevada ralentiza la implementación de cambios, dificulta la evolución del software y aumenta significativamente el coste de mantenimiento.

¿Cómo se puede solucionar?

Reducir la complejidad cognitiva puede resultar un desafío. A continuación, se presentan algunas buenas prácticas para facilitar la comprensión y el mantenimiento del código:

- **Extraer condiciones complejas en funciones independientes**
  La combinación de múltiples operadores en una misma condición incrementa la complejidad cognitiva. Extraer estas condiciones en una función separada con un nombre descriptivo ayuda a reducir la carga mental y mejora la legibilidad del código.
- **Dividir funciones grandes**
  Las funciones extensas suelen ser difíciles de entender y mantener. Si una función realiza demasiadas tareas, es recomendable dividirla en funciones más pequeñas y manejables. Cada función debería tener una única responsabilidad.
- **Evitar una anidación profunda mediante retornos tempranos**
  Para reducir la anidación de estructuras condicionales, es aconsejable gestionar primero los casos excepcionales y finalizar la ejecución de la función lo antes posible cuando sea necesario.
- **Utilizar operaciones seguras frente a valores nulos, cuando el lenguaje lo permita**
  El uso de operadores como ?. o ?? puede reemplazar múltiples

comprobaciones explícitas de valores nulos, simplificando el flujo de control y reduciendo la complejidad del código.

Ejemplo de código no conforme:

```
función  calcularPrecioFinal ( usuario,carrito ) {
   let total = calcularTotal (carrito);
  if (user. hasMembership                        // +1 (si)
    && user. orders > 10                         // +1 (más de una condición)
    && user. accountActive
    && !user. hasDiscount
    || user. orders === 1 ) {                     // +1 (cambio de operador en la condición)
      total = applyDiscount (user, total);
  }
   devolver total;
}
```

Ejemplo de solución::

```
función  calcularPrecioFinal ( usuario,carrito ) {
   let total = calcularTotal (carrito);
  si ( isEligibleForDiscount (usuario)) {        // +1 (si)
    total = aplicarDescuento (usuario, total);
  }
   devolver total;
}

función  isEligibleForDiscount ( usuario ) {
   return usuario.hasMembership
    && usuario.orders > 10 // +1 (más de una condición)
    && usuario.accountActive
    && !usuario.hasDiscount ||
    usuario.orders === 1 // +1 (cambio de operador en la condición )
}
```

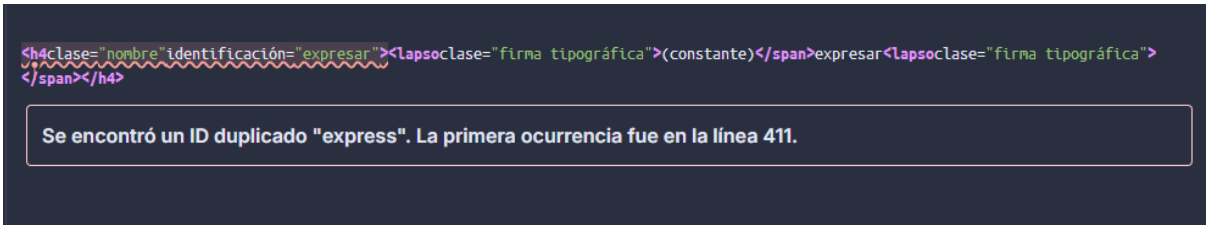
Por último, analizaremos las trés últimas vulnerabilidades.

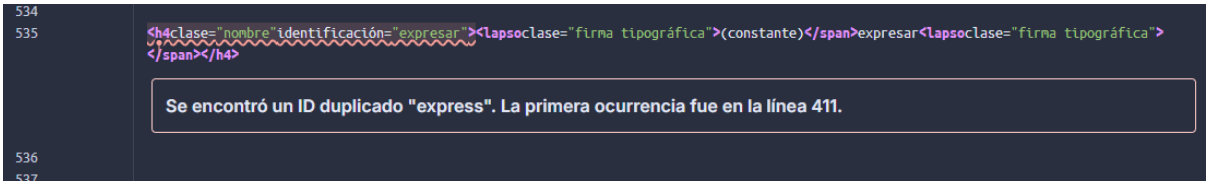
En este caso las tres últimas vulnerabilidades son la misma:

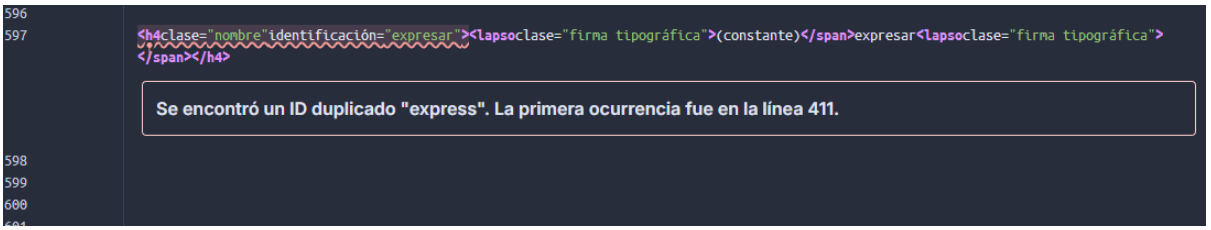
Se ha encontrado un ID duplicado “express".

¿Dónde se encuentra el problema?

En este caso esto ha sido encontrado en 3 líneas distintas.







¿Por qué se considera esto un problema?

La especificación HTML exige que los valores del atributo id sean únicos dentro de un documento. Cuando varios elementos comparten el mismo atributo id, pueden surgir diversos problemas:

- Los métodos de JavaScript, como document.getElementById(), solo devuelven el primer elemento coincidente, lo que provoca que los demás elementos con el mismo id queden inaccesibles.
- Los selectores CSS que apuntan a un id pueden no aplicar los estilos de forma consistente a todos los elementos previstos.
- Los lectores de pantalla y otras tecnologías de asistencia pueden experimentar dificultades al navegar entre elementos con identificadores duplicados.

- Las etiquetas de formulario que utilizan el atributo for pueden no asociarse correctamente con los controles de formulario correspondientes.

Estos problemas pueden provocar fallos en la funcionalidad, una mala experiencia de usuario y barreras de accesibilidad, especialmente para usuarios con discapacidades.

Impacto potencial

El uso de atributos id duplicados puede interrumpir la funcionalidad del código JavaScript, generar incoherencias en el estilo visual y crear barreras de accesibilidad. Como consecuencia, los usuarios pueden no ser capaces de interactuar correctamente con determinados elementos de la página y las tecnologías de asistencia pueden no funcionar como se espera.

¿Cómo se puede solucionar?

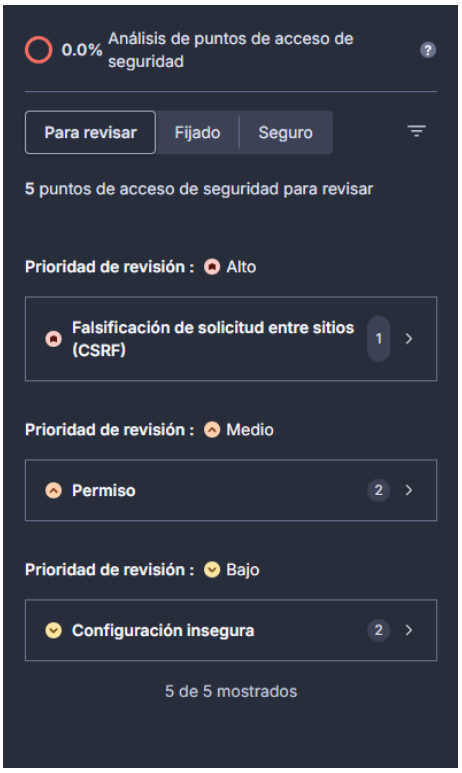
Ejemplo de código no conforme:

```
< div  id = "content" > Contenido principal </ div >
< div  id = "content" > Contenido secundario </ div >  <!-- No conforme: id duplicado -->
```

Ejemplo de solución:

```
< div  id = "main-content" > Contenido principal </ div >
< div  id = "secondary-content" > Contenido secundario </ div >
```

Lo siguiente que haremos será analizar los **SECURITY HOTSPOTS**.

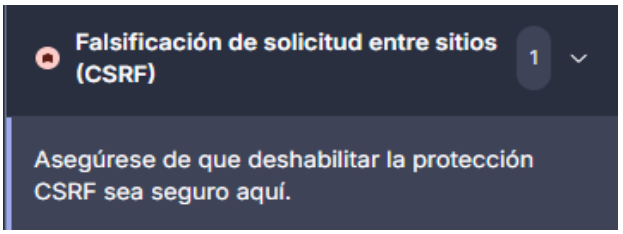


En este punto vemos un total de 5 a revisar:

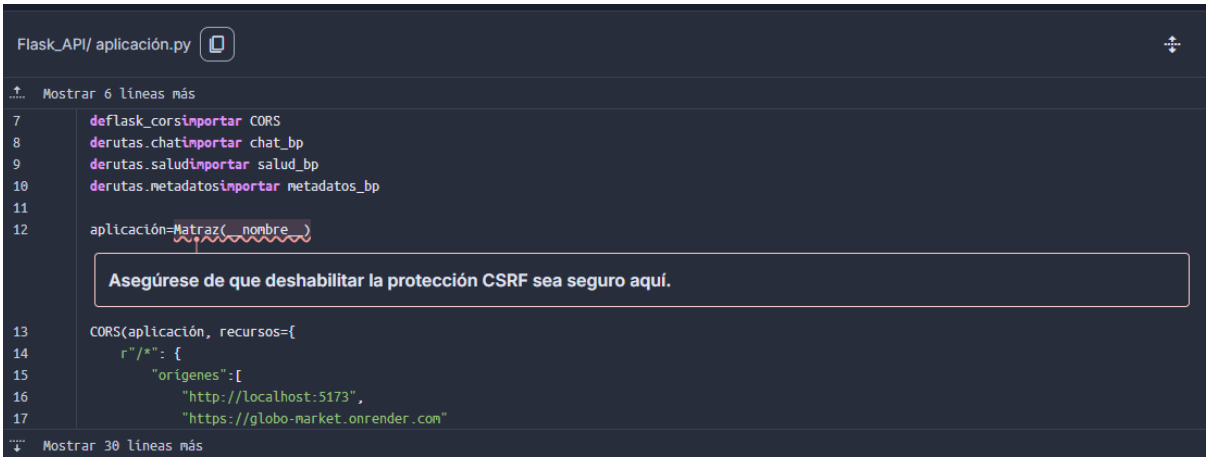
- Una de alta prioridad.
- Dos de prioridad media.
- Dos de prioridad media.

Ahora procederemos a analizar los riesgos más importantes

El primer riesgo que analizaremos será el de más alta prioridad, en este caso:



¿Dónde se encuentra el riesgo?



¿Cuál es el riesgo?

Falsificación de solicitudes entre sitios (CSRF)

Un ataque de falsificación de solicitud entre sitios (Cross-Site Request Forgery, CSRF) ocurre cuando un atacante logra que un usuario legítimo y autenticado de una aplicación web realice acciones sensibles sin su consentimiento, como actualizar su perfil, enviar mensajes o, en general, ejecutar cualquier operación que modifique el estado de la aplicación.

El atacante puede engañar a la víctima para que haga clic en un enlace que ejecuta una acción privilegiada o para que visite un sitio web malicioso que incluye una solicitud web oculta. Dado que los navegadores web envían automáticamente las cookies de sesión, la solicitud se realiza utilizando la sesión autenticada del usuario, permitiendo que la acción se ejecute con sus privilegios.

¿Cómo se puede solucionar?

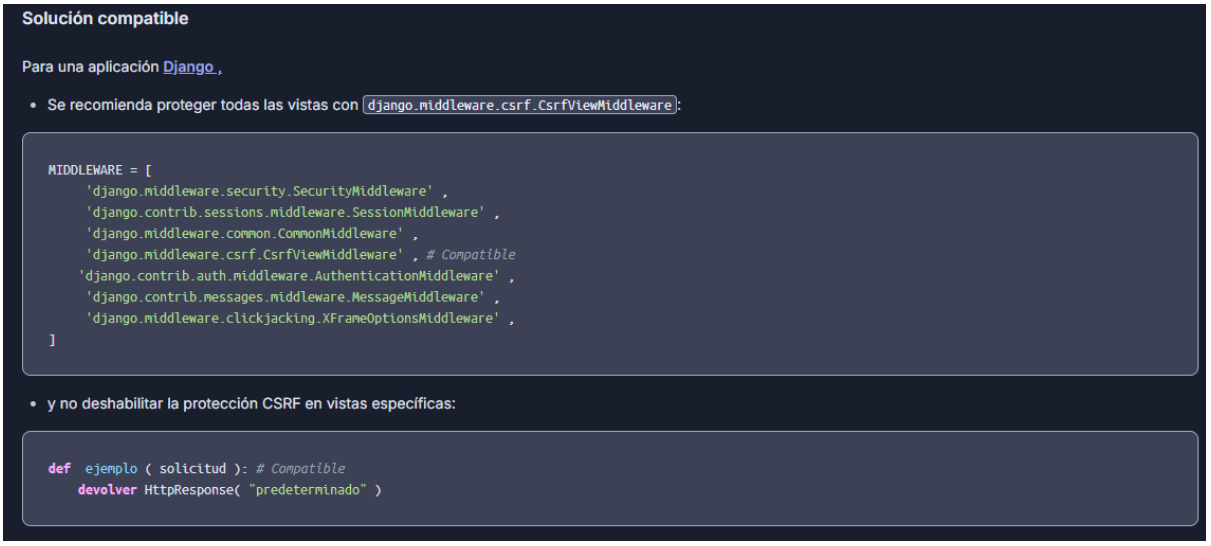
Se recomienda encarecidamente implementar mecanismos de protección frente a ataques de falsificación de solicitudes entre sitios (CSRF).

Estas medidas de seguridad deberían activarse de forma predeterminada para todos los métodos HTTP no seguros, es decir, aquellos que modifican el estado de la aplicación.

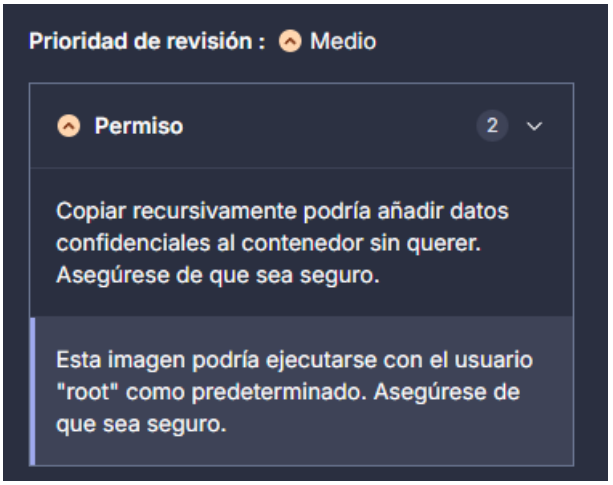
Una protección habitual consiste en el uso de tokens CSRF imposibles de adivinar, que se validan en cada solicitud sensible para garantizar que la petición ha sido generada legítimamente por la aplicación.

Asimismo, es importante tener en cuenta que las operaciones sensibles no deben realizarse mediante métodos HTTP seguros como GET, los cuales están diseñados exclusivamente para la recuperación de información y no para realizar cambios en el estado del sistema.

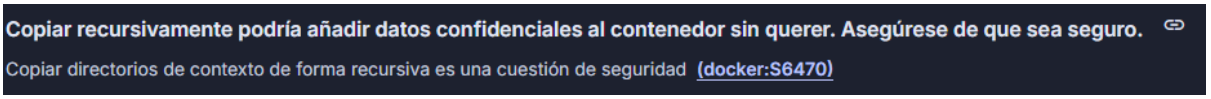
Ejemplo de una solución compatible:



Lo siguiente que haremos será analizar los dos Security Hotspots de rango medio:



En primer lugar, analizaremos el primer riesgo:



¿Dónde se encuentra?

¿Cuál es el riesgo?

Al crear una imagen de Docker a partir de un Dockerfile, se utiliza un directorio de contexto que se envía al demonio de Docker antes de que comience el proceso de compilación. Este directorio de contexto suele contener el propio Dockerfile, junto con todos los archivos necesarios para que la construcción de la imagen se realice correctamente.

Normalmente, el contexto de compilación incluye:

- El código fuente de las aplicaciones que se van a configurar dentro del contenedor.
- Archivos de configuración de otros componentes de software.
- Paquetes u otros recursos necesarios para el funcionamiento de la imagen.

Las directivas COPY y ADD del Dockerfile se utilizan para copiar contenido desde el directorio de contexto al sistema de archivos de la imagen resultante.

Sin embargo, cuando se emplean COPY o ADD para copiar de forma recursiva directorios completos de nivel superior, o múltiples elementos cuyos nombres se determinan durante la compilación, existe el riesgo de que se copien archivos inesperados al sistema de archivos de la imagen. Esto puede suponer un problema de seguridad, ya que dichos archivos podrían contener información sensible y afectar a la confidencialidad del sistema.

¿Cómo se puede solucionar?

Ejemplos de código sensible:

Copiar el directorio de contexto completo:

```
DESDE ubuntu: 22.04
#
COPIA sensible . .
CMD /run.sh
```

Copiar varios archivos y directorios cuyos nombres se expanden en el momento de la compilación:

```
DESDE ubuntu: 22.04 #
COPIA sensible
./ejemplo* / COPIA ./run.sh / CMD /run.sh
```

Posible solución

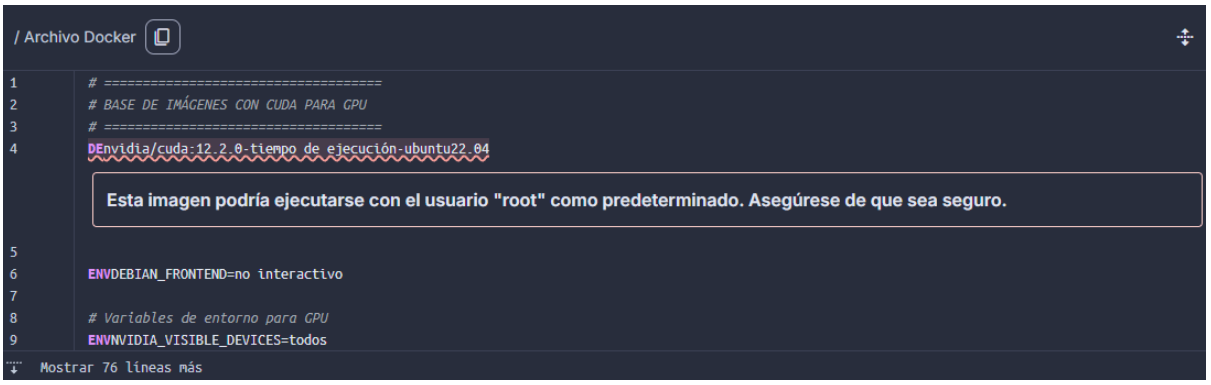
```
DESDE ubuntu: 22.04
COPIA ./ejemplo1 /ejemplo1
COPIA ./ejemplo2 /ejemplo2
COPIA ./run.sh /
CMD /run.sh
```

Ahora pasaremos a analizar el segundo riesgo:

Esta imagen podría ejecutarse con el usuario "root" como predeterminado. Asegúrese de que sea seguro.

Ejecutar contenedores como un usuario privilegiado es un asunto de seguridad sensible
(docker:S6471)

¿Dónde se encuentra?



¿Cuál es el riesgo?

Ejecutar contenedores utilizando un usuario con privilegios elevados debilita significativamente su seguridad en tiempo de ejecución, ya que permite que cualquier código que se ejecute dentro del contenedor pueda realizar acciones administrativas.

En contenedores Linux, el usuario con privilegios suele ser root, mientras que en contenedores Windows el rol equivalente es ContainerAdministrator.

Un usuario malintencionado puede ejecutar código dentro del sistema aprovechando acciones que podrían considerarse legítimas según la lógica de negocio de la aplicación o los mecanismos de administración operativa, o bien mediante acciones claramente maliciosas, como la ejecución de código arbitrario tras explotar un servicio alojado en el contenedor.

Si el contenedor no está adecuadamente protegido para restringir el uso de shells, intérpretes o capacidades de Linux, un atacante podría:

- Leer y exfiltrar archivos del sistema, incluidos volúmenes de Docker.
- Abrir nuevas conexiones de red.
- Instalar software malicioso.
- Comprometer el aislamiento del contenedor mediante la explotación de otros componentes del sistema.

Este escenario facilita el robo de archivos críticos de infraestructura, propiedad intelectual o datos personales.

Dependiendo del nivel de resiliencia de la infraestructura, los atacantes podrían extender el alcance del ataque a otros servicios, como clústeres de Kubernetes o proveedores de servicios en la nube, aumentando así el impacto del incidente de seguridad.

¿Cómo se puede solucionar?

- Cree un nuevo usuario sin privilegios y establézcase como usuario predeterminado mediante la instrucción USER.
- Algunos contenedores ya incluyen usuarios específicos no privilegiados, como postgresql o zookeeper. En estos casos, se recomienda utilizar dichos usuarios en lugar de root.
- En contenedores Windows, puede utilizar el usuario ContainerUser con el mismo propósito.

En el momento de la ejecución del contenedor:

- Utilice el argumento --user al ejecutar Docker o defina el usuario correspondiente en el archivo docker-compose.
- Añada únicamente las capacidades de Linux necesarias para realizar acciones específicas que requieran privilegios elevados, evitando otorgar permisos de superusuario completos.

Si una imagen ya está configurada explícitamente para iniciarse con un usuario sin privilegios, puede añadirse a la propiedad de la regla de lista de imágenes seguras de la instancia de SonarQube, sin incluir la etiqueta de la imagen.

¿Posibles soluciones?

Ejemplo de código sensible:

```
# Sensible
DESDE alpino

PUNTO DE ENTRADA [ "id" ]
```

```
DESDE alpine AS builder
 COPIAR Makefile ./src /
EJECUTAR make build
USUARIO nonroot

# Las configuraciones de usuario anteriores y confidenciales se eliminan
FROM alpine AS runtime
 COPY --from=builder bin/production /app
ENTRYPOINT [ "/app/production" ]
```

Posibles soluciones

Para imágenes basadas en Linux e imágenes basadas en scratch que descomprimen una distribución de Linux:

```
DESDE alpino

EJECUTAR addgroup -S nonroot \
    && adduser -S no raíz -G no raíz

USUARIO no root

PUNTO DE ENTRADA [ "id" ]
```

Para las imágenes basadas en Windows, puede usar `ContainerUser` o crear un nuevo usuario:

```
DESDE mcr.microsoft.com/windows/servercore:ltsc2019

EJECUTAR net user /add nonroot

USUARIO no root
```

Para compilaciones de varias etapas, el usuario no root debe estar en la última etapa:

```
DESDE alpine como constructor
 COPIAR Makefile ./src /
EJECUTAR make build

DESDE alpine como tiempo de ejecución
 EJECUTAR addgroup -S nonroot \
    && adduser -S nonroot -G nonroot
COPIA --from=builder bin/production /app
USUARIO nonroot
 PUNTO DE ENTRADA [ "/app/production" ]
```

O también

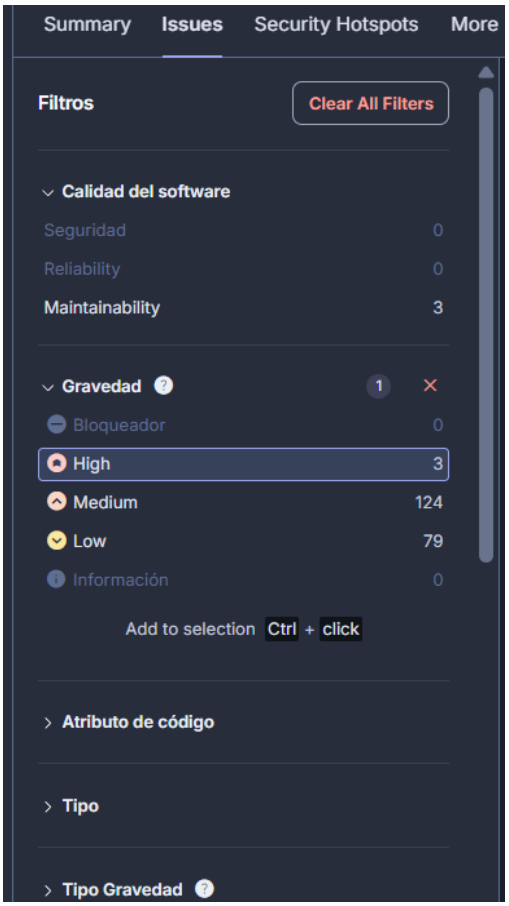
```
DESDE alpine:latest como proveedor_de_seguridad
 EJECUTAR addgroup -S nonroot \
    && adduser -S no raíz -G no raíz

DESDE cero como producción
 COPIA --from=security_provider /etc/passwd /etc/passwd
USUARIO nonroot
 COPIA production_binary /app
PUNTO DE ENTRADA [ "/app/production_binary" ]
```

Visto esto, ya tenemos analizadas las principales vulnerabilidades del código de nuestro equipo de Data Science.

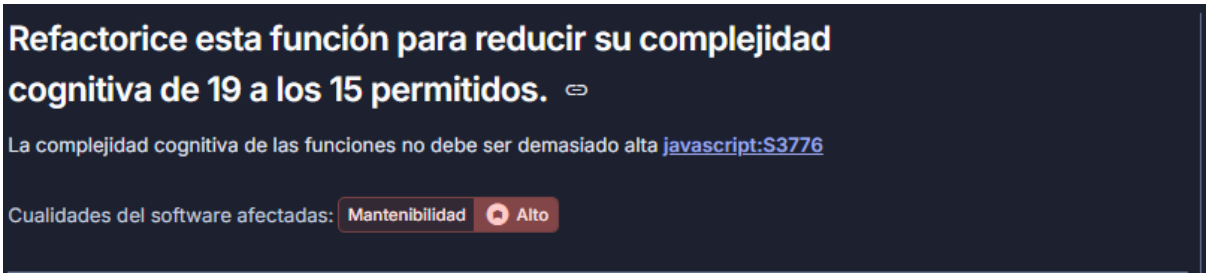
Ahora en el siguiente paso analizaremos el código estático del código del equipo de FULL STACK:

Lo primero que vemos cuando desplegamos la herramienta y analizamos el repositorio en el que han subido el código, son las vulnerabilidades.



Nos fijamos que en concreto hay tres vulnerabilidades de rango alto, las cuales las analizaremos.

En primer lugar, analizaremos esta:



¿Dónde se encuentra el problema?

¿Por qué esto es un problema?

La complejidad cognitiva mide la dificultad para comprender el flujo de control de una unidad de código. Un código con una complejidad cognitiva elevada resulta más difícil de leer, entender, probar y modificar.

Como regla general, una alta complejidad cognitiva es un indicador claro de que el código debería refactorizarse en componentes más pequeños, simples y fáciles de mantener.

¿Qué sintaxis del código afecta a la puntuación de complejidad cognitiva?

La complejidad cognitiva aumenta cada vez que el código interrumpe el flujo de lectura lineal normal. Esto incluye, por ejemplo, el uso de estructuras de bucle, condicionales, bloques de captura de excepciones, estructuras switch, saltos a etiquetas y condiciones que combinan múltiples operadores lógicos.

Cada nivel adicional de anidamiento incrementa la complejidad. Durante la lectura del código, cuanto más se profundiza en las capas anidadas,

mayor es la dificultad para mantener el contexto y comprender el comportamiento del programa.

Las llamadas a métodos no incrementan la complejidad cognitiva. Un nombre de método bien seleccionado actúa como un resumen de varias líneas de código, permitiendo al lector obtener primero una visión general de la funcionalidad y profundizar posteriormente en los detalles de las funciones llamadas.

¿Cuál es el impacto potencial?

La alta complejidad cognitiva ralentiza los cambios y aumenta el coste de mantenimiento.

¿Cómo se puede solucionar?

- Extraer condiciones complejas en funciones independientes.
  La combinación de múltiples operadores dentro de una condición incrementa la complejidad cognitiva. Extraer esta lógica a una función separada con un nombre descriptivo reduce la carga mental y mejora la legibilidad del código.
- Dividir funciones grandes.
  Las funciones extensas suelen ser difíciles de comprender y mantener. Si una función realiza demasiadas tareas, es recomendable dividirla en funciones más pequeñas y manejables, asegurando que cada una tenga una única responsabilidad.

- Evitar la anidación profunda mediante retornos tempranos.

  Para reducir la profundidad de las estructuras condicionales, se deben gestionar primero los casos excepcionales y finalizar la ejecución de la función lo antes posible cuando sea necesario.

- Utilizar operaciones seguras frente a valores nulos, cuando el lenguaje lo permita.

  El uso de operadores como ?. o ?? puede sustituir múltiples comprobaciones de valores nulos, simplificando el flujo de control y reduciendo la complejidad del código.

Ejemplo de código no conforme:

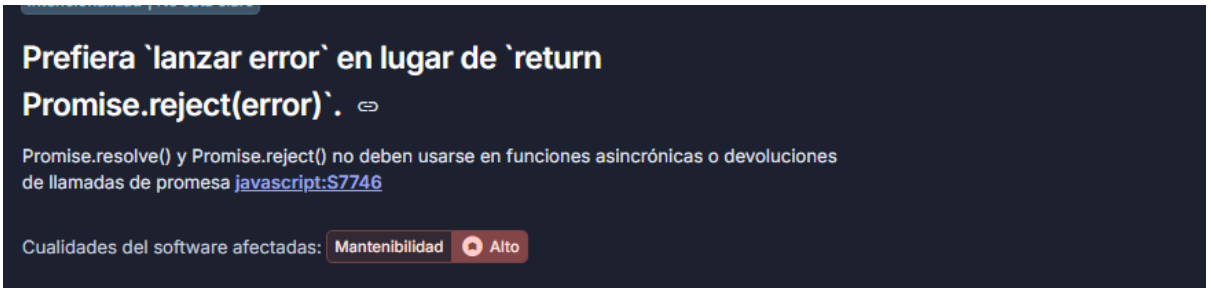
```
función  calcularPrecioFinal ( usuario,carrito ) {
   let total = calcularTotal (carrito);
  if (user. hasMembership                    // +1 (si)
    && user. orders > 10                     // +1 (más de una condición)
    && user. accountActive
    && !user. hasDiscount
    || user. orders === 1 ) {                 // +1 (cambio de operador en la condición)
      total = applyDiscount (user, total);
  }
   devolver total;
}
```

Solución conforme:

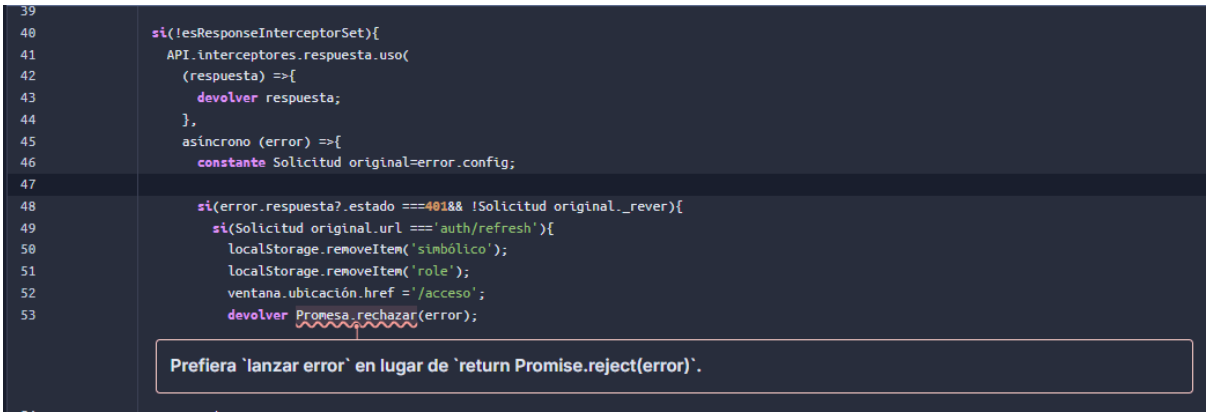
```
función  calcularPrecioFinal ( usuario,carrito ) {
   let total = calcularTotal (carrito);
  si ( isEligibleForDiscount (usuario)) {       // +1 (si)
    total = aplicarDescuento (usuario, total);
  }
   devolver total;
}

función  isEligibleForDiscount ( usuario ) {
   return usuario.hasMembership
    && usuario.orders > 10 // +1 (más de una condición)
    && usuario.accountActive
    && !usuario.hasDiscount ||
    usuario.orders === 1 // +1 (cambio de operador en la condición )
}
```

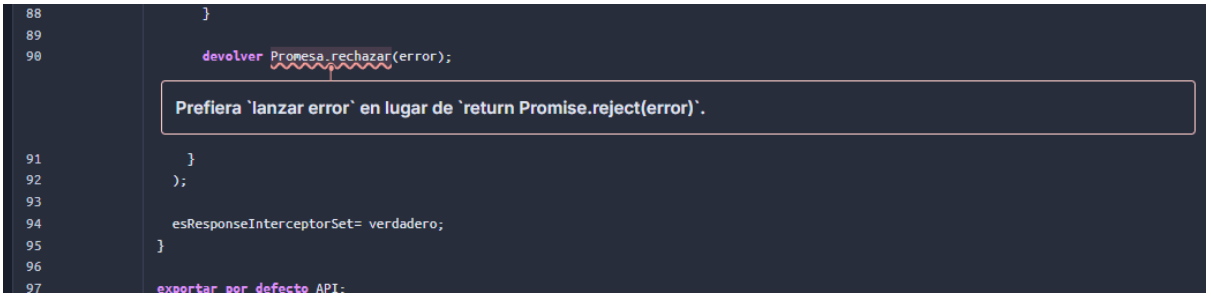
Lo siguiente que haremos será analizar la segunda y la tercera vulnerabilidad:



En este caso esta vulnerabilidad se encuentra dos veces:



Y luego en esta otra línea:



¿Por qué esto es un problema?

En JavaScript, las funciones asíncronas y los métodos de manejo de promesas encapsulan automáticamente los valores de retorno dentro de

una promesa. Por este motivo, el uso explícito de Promise.resolve() o Promise.reject() en estos contextos introduce complejidad y redundancia innecesarias.

Cuando se devuelve un valor desde una función async, este se envuelve automáticamente en una promesa resuelta. De forma similar, al devolver un valor desde los callbacks de los métodos .then(), .catch() o .finally(), dicho valor también se encapsula automáticamente en una promesa.

En cuanto a la gestión de errores, lanzar una excepción mediante throw es una práctica más clara e idiomática que devolver Promise.reject(). La instrucción throw expresa de manera explícita que se ha producido una condición de error, mientras que el uso de return Promise.reject() puede ocultar esta intención y dificultar la comprensión del flujo de ejecución.

Esta redundancia afecta negativamente a la legibilidad del código, especialmente para desarrolladores que están aprendiendo los patrones de programación asíncrona. Además, suele reflejar una comprensión incorrecta del funcionamiento de las promesas y de las funciones asíncronas.

¿Cuál es el posible impacto potencial?

Este problema afecta principalmente a la mantenibilidad y legibilidad del código. Aunque no suele provocar errores en tiempo de ejecución, puede:

- Hacer que el código sea innecesariamente verboso y más difícil de entender.
- Sugerir confusión sobre los conceptos fundamentales de la programación asíncrona.
- Inducir a otros desarrolladores a malinterpretar el comportamiento real del código.

- Reducir la coherencia con las buenas prácticas del uso de async/await en JavaScript moderno.

¿Cuál es una posible solución?

Ejemplo de código no conforme:

```
constante  fetchData = async () => {
   const resultado = await  obtenerDatos ();
  devolver  Promesa . resolver (resultado); // No conforme
};
```

Ejemplo de código compatible:

```
constante  fetchData = async () => {
   const resultado = await  obtenerDatos ();
  devolver resultado;
};
```

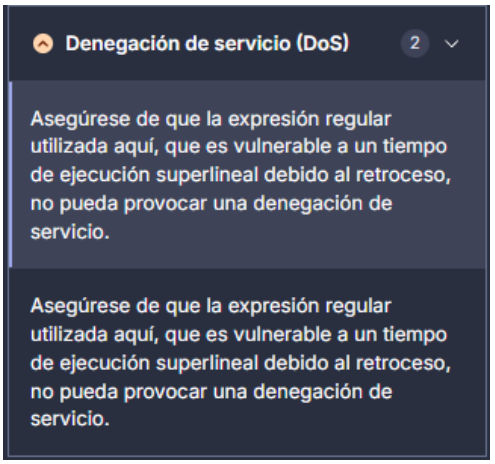
Ahora que ya hemos analizado las vulnerabilidades del código, pasaremos a analizar los Security Hotspots del código.

En esta gráfica que nos enseña la herramienta, podemos ver un total de 7 tipos de riesgos, de los cuales encontramos:

- Cuatro riesgos de media prioridad.
- Tres riesgos de prioridad leve.

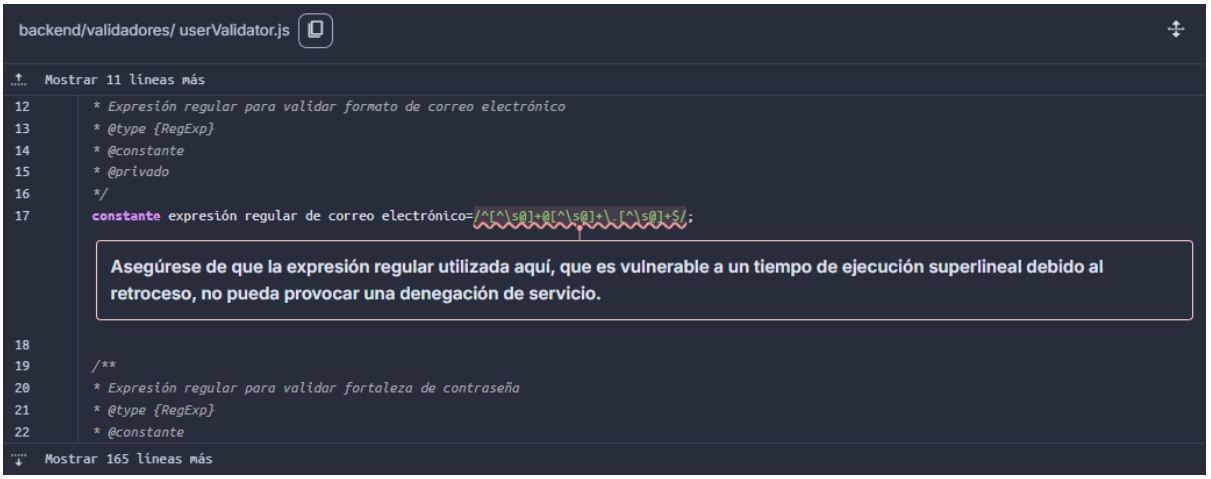
Ahora procederemos a realizar un análisis sobre los riesgos más importantes del análisis.

En primer lugar, analizaremos el primer y segundo Hotspots:



¿Dónde se encuentran estos riesgos?

La primera que se nos muestra se encuentra en:



La segunda se encuentra en:

¿Cuál es el principal riesgo?

La mayoría de los motores de expresiones regulares utilizan el mecanismo de retroceso (backtracking) para evaluar una entrada, probando todas las rutas de ejecución posibles de la expresión regular. En determinados casos, este comportamiento puede provocar graves problemas de rendimiento, conocidos como retroceso catastrófico.

En el peor de los escenarios, la complejidad de ejecución de una expresión regular puede crecer de forma exponencial respecto al tamaño de la entrada. Esto implica que una cadena pequeña y cuidadosamente diseñada, por ejemplo, de tan solo 20 caracteres, puede desencadenar un retroceso catastrófico y provocar una denegación de servicio (DoS) en la aplicación. De forma similar, las expresiones regulares con complejidad superlineal pueden generar el mismo impacto cuando se procesan entradas grandes, del orden de miles de caracteres.

Esta regla analiza la complejidad temporal de las expresiones regulares e informa cuando su comportamiento no es lineal, ayudando a identificar patrones potencialmente peligrosos desde el punto de vista del rendimiento y la seguridad.

¿Cuáles son las posibles soluciones?

En todos los casos descritos a continuación, el retroceso catastrófico solo puede producirse si la parte problemática de la expresión regular va seguida de un patrón que puede fallar, provocando que el motor de expresiones regulares tenga que retroceder para probar rutas alternativas.

- Repeticiones ambiguas (r* o r*?)

  Si una repetición permite múltiples formas de coincidir con la misma entrada (posiblemente con longitudes diferentes), el tiempo de ejecución en el peor de los casos puede ser exponencial.

  Esto ocurre cuando r contiene partes opcionales, alternancias o repeticiones adicionales. No sucede cuando la repetición está definida de forma que solo exista una única manera de coincidir.
- Repeticiones consecutivas que coinciden con el mismo contenido

  Cuando varias repeticiones consecutivas pueden coincidir con la misma parte de la entrada, o están separadas únicamente por un separador opcional o ambiguo, el tiempo de ejecución puede ser polinómico (O(n^c), donde c es el número de repeticiones problemáticas).

  Por ejemplo:
  - a*b* no supone un problema, ya que a* y b* coinciden con conjuntos de caracteres distintos.
  - a*_a* tampoco es problemático, ya que las repeticiones están separadas por _ y no pueden coincidir con él.
  - En cambio, expresiones como a*a* o .*_.* presentan un tiempo de ejecución cuadrático.
- Expresiones no ancladas al inicio de la cadena

  Si la expresión regular no está anclada al inicio (^), evitar ejecuciones cuadráticas resulta especialmente complicado. Cuando una coincidencia falla, el motor intenta de nuevo desde el siguiente

índice de la cadena.

Esto implica que cualquier repetición ilimitada seguida de un patrón que puede fallar puede provocar ejecuciones cuadráticas. Por ejemplo, str.split(/\s*,/) puede ejecutarse en tiempo cuadrático sobre cadenas formadas únicamente por espacios o por grandes secuencias de espacios no seguidas de una coma.

ESTRATEGIAS PARA EVITAR RETROCESO CATASTRÓFICO

Para reescribir expresiones regulares problemáticas, se pueden aplicar las siguientes técnicas:

- Limitar el número de repeticiones esperadas mediante cuantificadores acotados, por ejemplo {1,5} en lugar de +.
- Refactorizar cuantificadores anidados para reducir el número de formas en que el cuantificador externo puede coincidir. Por ejemplo, la expresión (ba+)+ no presenta problemas de rendimiento, ya que el grupo interno solo puede coincidir si existe exactamente un carácter b por repetición.
- Optimizar las expresiones emulando cuantificadores posesivos y agrupaciones atómicas.
- Utilizar clases de caracteres negados en lugar de . para excluir separadores cuando sea posible. Por ejemplo, la expresión cuadrática .*_.* puede convertirse en lineal sustituyéndola por [^_]*_.*.

ALTERNATIVAS CUANDO NO ES POSIBLE UNA EXPRESIÓN LINEAL

En algunos casos no es posible reescribir la expresión regular para que sea lineal y mantenga el comportamiento deseado, especialmente cuando no está anclada al inicio de la cadena. En estas situaciones se recomienda:

- Resolver el problema sin utilizar expresiones regulares.

- Emplear motores de expresiones regulares sin retroceso, como RE2 de Google o node-re2.
- Usar múltiples pasadas sobre la cadena, por ejemplo, preprocesando o posprocesando la entrada manualmente, o combinando varias expresiones regulares. Un caso típico sería usar str.split(/\s*,\s*/) o dividir primero con str.split(",") y luego recortar los espacios.

En algunos escenarios, también es posible hacer que la expresión regular sea infalible haciendo opcionales todas las partes que podrían fallar, evitando así el retroceso. Aunque esto implica aceptar más cadenas de las previstas, el resultado puede validarse posteriormente mediante grupos de captura. Por ejemplo, la expresión x*y puede reemplazarse por x*(y)? y verificar posteriormente si el grupo opcional ha coincidido.

Ejemplo de código sensible:

```
/(a+)+$/. prueba (
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaa!
" // Sensible
```

Ejemplo de posible solución:

```
/((?=(a+))\ 2 )+$/. prueba (
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" +
 "aaaaaaaaaaaaaaa!
" // Cumple
```

Por último, analizaremos las dos últimas Hotspots "altas" en este caso relacionados con los permisos.

¿Dónde se encuentra el riesgo?

¿Cuál es el riesgo?

Al crear una imagen de Docker a partir de un Dockerfile, se utiliza un directorio de contexto que se envía al demonio de Docker antes de que comience el proceso de compilación. Este directorio de contexto suele contener el propio Dockerfile, junto con todos los archivos necesarios para que la construcción de la imagen se realice correctamente.

Normalmente, el contexto de compilación incluye:

- El código fuente de las aplicaciones que se van a configurar dentro del contenedor.
- Archivos de configuración de otros componentes de software.
- Paquetes u otros recursos necesarios para el funcionamiento de la imagen.

Las directivas COPY y ADD del Dockerfile se emplean para copiar contenido desde el directorio de contexto al sistema de archivos de la imagen resultante.

Sin embargo, cuando se utilizan COPY o ADD para copiar de forma recursiva directorios completos de nivel superior, o múltiples elementos

cuyos nombres se determinan dinámicamente durante la compilación, existe el riesgo de que se incluyan archivos inesperados en el sistema de archivos de la imagen. Esto puede suponer un problema de seguridad, ya que dichos archivos podrían contener información sensible y afectar a la confidencialidad del sistema.

¿Cómo se puede solucionar?

Se recomienda limitar el uso de globbing en la definición de las fuentes de las instrucciones COPY y ADD. El uso de patrones genéricos puede provocar la copia involuntaria de archivos innecesarios o sensibles al sistema de archivos de la imagen.

Asimismo, debe evitarse copiar todo el directorio de contexto dentro de la imagen. En su lugar, es preferible especificar de forma explícita únicamente los archivos y directorios necesarios para que la imagen funcione correctamente.

Este enfoque mejora la seguridad, reduce el tamaño de la imagen y minimiza el riesgo de exponer información confidencial.

Ejemplo de código sensible:

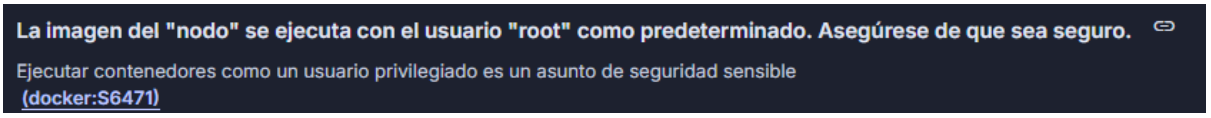
```
DESDE ubuntu: 22.04
#
COPIA sensible . .
CMD /run.sh
```

```
DESDE ubuntu: 22.04 #
COPIA sensible
./ejemplo* / COPIA ./run.sh / CMD /run.sh
```

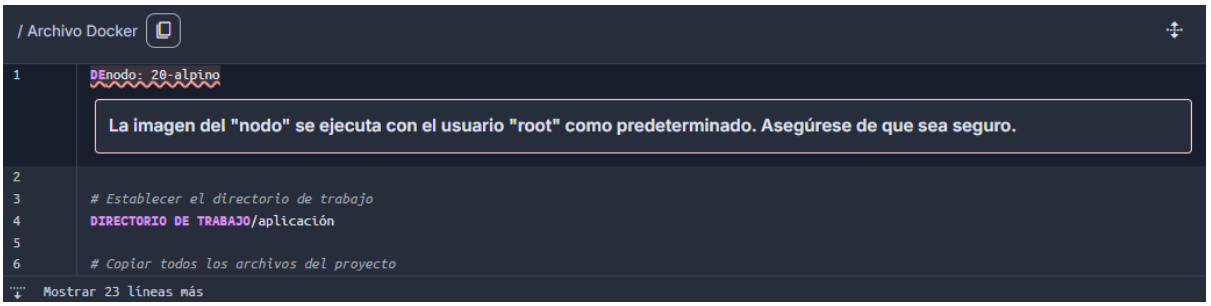
Ejemplo de solución compatible:

```
DESDE ubuntu: 22.04
COPIA ./ejemplo1 /ejemplo1
COPIA ./ejemplo2 /ejemplo2
COPIA ./run.sh /
CMD /run.sh
```

Por último, analizaremos el último riesgo “caracterizado” como alto.



¿Dónde está el riesgo?



¿Cuál es el riesgo?

Ejecutar contenedores utilizando un usuario con privilegios elevados debilita significativamente la seguridad en tiempo de ejecución, ya que permite que cualquier código que se ejecute dentro del contenedor pueda realizar acciones administrativas.

En los contenedores Linux, el usuario con privilegios suele ser root, mientras que en los contenedores Windows el rol equivalente es ContainerAdministrator.

Un usuario malintencionado puede ejecutar código en el sistema aprovechando acciones que podrían considerarse legítimas según la lógica de negocio de la aplicación o los mecanismos de administración operativa, o bien mediante acciones claramente maliciosas, como la ejecución de código arbitrario tras explotar un servicio alojado en el contenedor.

Si el contenedor no está correctamente protegido para restringir el uso de shells, intérpretes o capacidades de Linux, un atacante podría:

- Leer y exfiltrar archivos del sistema, incluidos volúmenes de Docker.
- Abrir nuevas conexiones de red.
- Instalar software malicioso.
- Comprometer el aislamiento del contenedor mediante la explotación de otros componentes del sistema.

Este escenario facilita el robo de archivos críticos de infraestructura, propiedad intelectual o datos personales.

Dependiendo del nivel de resiliencia de la infraestructura, los atacantes podrían extender el alcance del ataque a otros servicios, como clústeres de Kubernetes o proveedores de servicios en la nube, aumentando así el impacto del incidente de seguridad.

¿Cómo se podría solucionar?

En el Dockerfile:

- Cree un nuevo usuario sin privilegios y establézcalo como usuario predeterminado mediante la instrucción USER.
- Algunos contenedores ya incluyen usuarios específicos no privilegiados, como postgresql o zookeeper, aunque no estén configurados explícitamente como usuarios predeterminados. Se recomienda utilizar estos usuarios en lugar de root.
- En contenedores Windows, puede emplearse el usuario ContainerUser con el mismo propósito.

En el momento de la ejecución del contenedor:

- Utilice el argumento --user al ejecutar Docker o defina el usuario correspondiente en el archivo docker-compose.

- Añada únicamente las capacidades de Linux necesarias para realizar acciones específicas que requieran privilegios elevados, evitando conceder permisos de superusuario completos.

Si la imagen ya está configurada explícitamente para iniciarse con un usuario sin privilegios, puede añadirse a la propiedad de la regla de *lista de imágenes seguras* de la instancia de SonarQube, sin incluir la etiqueta de la imagen.

Ejemplo de código sensible:

```
# Sensible
DESDE alpino

PUNTO DE ENTRADA [ "id" ]
```

```
DESDE alpine AS builder
 COPIAR Makefile ./src /
EJECUTAR make build
USUARIO nonroot

# Las configuraciones de usuario anteriores y confidenciales se eliminan
FROM alpine AS runtime
 COPY --from=builder bin/production /app
ENTRYPOINT [ "/app/production" ]
```

Ejemplo de una posible solución:

```
DESDE alpino

EJECUTAR addgroup -S nonroot \
    && adduser -S no raíz -G no raíz

USUARIO no root

PUNTO DE ENTRADA [ "id" ]
```

```
DESDE mcr.microsoft.com/windows/servercore:ltsc2019

EJECUTAR net user /add nonroot

USUARIO no root
```

```
DESDE alpine como constructor
 COPIAR Makefile ./src /
EJECUTAR make build

DESDE alpine como tiempo de ejecución
 EJECUTAR addgroup -S nonroot \
    && adduser -S nonroot -G nonroot
COPIA --from=builder bin/production /app
USUARIO nonroot
 PUNTO DE ENTRADA [ "/app/production" ]
```

4. Recomendaciones

A partir de los resultados obtenidos con SonarQube, se recomienda que los equipos de desarrollo adopten las guías y buenas prácticas proporcionadas por la herramienta como apoyo durante el proceso de desarrollo.

SonarQube no solo detecta problemas, sino que también ofrece explicaciones claras sobre su impacto potencial y propone estrategias concretas para corregirlos. Estas guías pueden ser utilizadas por los equipos para:

- Reducir la complejidad del código y mejorar su legibilidad y mantenibilidad.
- Prevenir vulnerabilidades de seguridad comunes en aplicaciones web.

- Evitar errores frecuentes en programación asíncrona.
- Mejorar la seguridad en la creación y ejecución de contenedores Docker.
- Estandarizar buenas prácticas de desarrollo dentro del equipo.

Integrar SonarQube de forma temprana en el ciclo de desarrollo permite detectar problemas antes de que lleguen a producción, reduciendo el coste de corrección y facilitando la adopción de una cultura de calidad y seguridad desde el diseño.

5. Conclusión

El uso de SonarQube en este proyecto ha demostrado ser una herramienta eficaz para identificar problemas de calidad y seguridad en el código, así como configuraciones potencialmente peligrosas en el entorno de ejecución. Gracias a su análisis estático y a las explicaciones detalladas que acompaña a cada regla, ha sido posible comprender el impacto real de cada incidencia y documentar adecuadamente los riesgos asociados.

Además, SonarQube no solo actúa como un detector de errores, sino también como una guía formativa para los desarrolladores, ayudando a mejorar sus conocimientos sobre buenas prácticas de programación y seguridad. Esto resulta especialmente útil en entornos académicos y de laboratorio, donde el objetivo no es únicamente corregir errores, sino también aprender a evitarlos en futuros desarrollos.

En conclusión, el uso de SonarQube aporta un valor significativo al proceso de desarrollo, facilitando la creación de software más seguro, mantenible y de mayor calidad, y convirtiéndose en una herramienta recomendable tanto para entornos educativos como profesionales.