CORPORACIÓN UNIVERSITARIA MINUTO DE DIOS – UNIMINUTO

Informe de Pruebas (vulnerabilidades) software StockFlow

Yeison David Negrete Suarez ID - 688627

Pedro David Fernández ID – 694841

Ronald Andrés Arias ID – 221626

Desarrollo de software seguro

Edwin Albeiro Ramos

Bogotá 21 de febrero 2026

Introducción:
El presente documento detallamos los resultados obtenidos tras un escaneo de vulnerabilidades y análisis de calidad realizado al software StockFlow, esta evaluación se llevó a cabo utilizando la plataforma de análisis estático de código SonarQube, con el objetivo primordial de identificar debilidades de seguridad, fallos de fiabilidad y oportunidades de mejora en la mantenibilidad del código fuente.

Durante el proceso de auditoría, se detectó un hallazgo crítico clasificado como Bloqueador relacionado con la exposición de secretos y hashes de contraseñas, lo cual representa un riesgo significativo de compromiso para las cuentas de usuario y la integridad de la aplicación, asimismo, el análisis arrojó métricas relevantes en otras dimensiones de calidad:

Seguridad: 1 cuestión abierta (Impacto alto/Bloqueador).

Fiabilidad: 8 cuestiones abiertas.

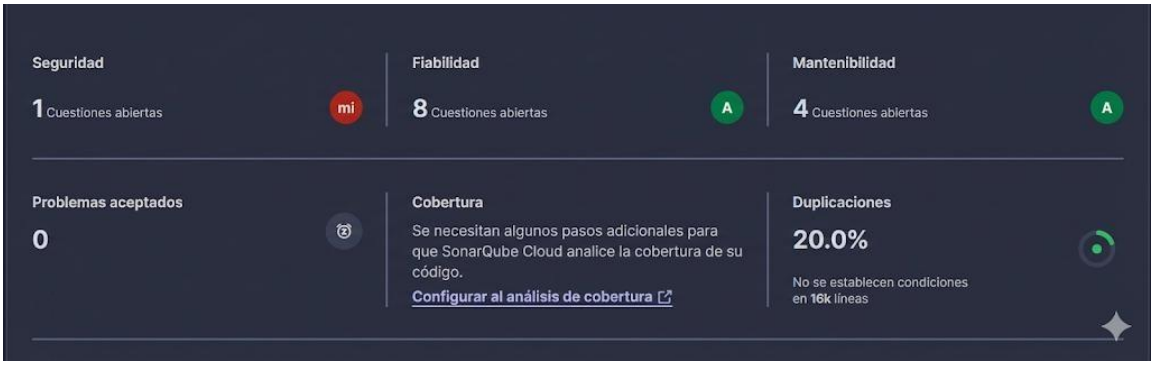
Mantenibilidad: 4 cuestiones abiertas, con un enfoque en la reducción de deuda técnica y mejora de la legibilidad.

Duplicación: Se identificó un 20.0% de código duplicado en las líneas analizadas.

A lo largo de este informe, se describen detalladamente los impactos potenciales de estas vulnerabilidades que pueden ir desde pérdidas financieras hasta la toma de control total del sistema y se proporcionan las recomendaciones técnicas y ejemplos de código conforme necesarios para mitigar estos riesgos y fortalecer la arquitectura del software bajo estándares de desarrollo seguro.

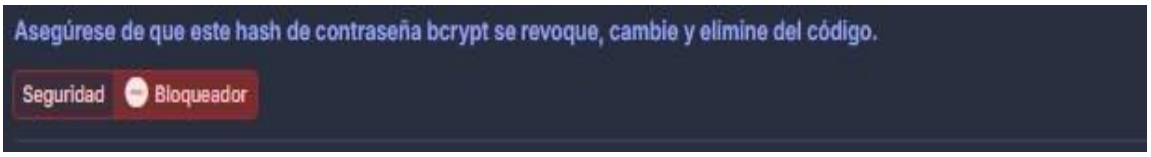
1). Análisis SonarQube StockFlow:

A continuación, realizaremos un escaneo del software para revisar problemas o vulnerabilidades a través del programa en línea sonarQube.



Al realizar la respectiva revisión encontramos un problema de alto impacto en seguridad.

Este problema tiene un impacto negativo en la seguridad de su software, la gravedad ahora está directamente relacionada con la calidad del software afectada, esto significa que cada calidad de software afectada tiene una gravedad, hay cinco niveles de gravedad: bloqueador, alto, medio, bajo e informativo.



Los hashes de contraseñas filtrados permiten a los atacantes realizar ataques de fuerza bruta o de diccionario fuera de línea, recuperando potencialmente contraseñas de usuarios y comprometiendo cuentas, en la mayoría de los casos, se violan los límites de confianza cuando se expone un secreto en un repositorio de código fuente o en un entorno de implementación no controlado, personas no autorizadas que no necesitan conocer el secreto podrían acceder a él y usarlo para obtener acceso no deseado a servicios o recursos asociados, el problema de la confianza puede ser más o menos grave dependiendo del rol y los derechos de las personas.

2). impacto potencial:

Las consecuencias varían considerablemente según la situación y el público expuesto al secreto. Aun así, cabe considerar dos escenarios principales.

- **Pérdida financiera**

Pueden producirse pérdidas financieras cuando se utiliza un secreto para acceder a un servicio de pago de terceros y se divulga como parte del código fuente de las aplicaciones cliente, al tener el secreto, cada usuario de la aplicación podrá usarlo sin restricciones para usar el servicio de terceros según sus necesidades, incluso de forma inesperada.

- **Reducción de la seguridad de la aplicación**

Una degradación puede ocurrir cuando el secreto divulgado se utiliza para proteger activos o funciones sensibles de la aplicación, dependiendo del activo o la función afectada, el impacto práctico puede ir desde una filtración de información sensible hasta la toma de control total de la aplicación, su servidor de alojamiento u otro componente vinculado, por ejemplo, una aplicación que divulgara un secreto utilizado para firmar tokens de autenticación de usuarios correría el riesgo de suplantación de identidad, un atacante que accediera al secreto filtrado podría firmar tokens de sesión para usuarios arbitrarios y apropiarse de sus privilegios y autorizaciones.

3). ¿Cómo solucionar el problema?

- **Revocar el secreto:** Revocar cualquier secreto filtrado y eliminarlo del código fuente de la aplicación, antes de revocar el secreto, nos aseguramos de que ninguna otra aplicación o proceso lo esté utilizando, otros usos del secreto también se verán afectados.
- **Analizar el uso reciente de secretos:** En tanto no es posible, analizamos los registros de autenticación para identificar cualquier uso no intencionado o malicioso del secreto desde su fecha de divulgación, esto permitirá determinar si un atacante se aprovechó del secreto filtrado y en qué medida, esta operación debe ser parte de un proceso global de respuesta a incidentes.
- **Utilice una bóveda secreta:** Se debe usar una bóveda secreta para generar y almacenar el nuevo secreto, esto garantizará nuestra seguridad y evitará cualquier divulgación inesperada, dependiendo de la plataforma de desarrollo y del tipo de secreto filtrado, actualmente hay múltiples soluciones disponibles.

Nunca debemos codificar secretos, ni siquiera los valores predeterminados.

- En primer lugar, los secretos predeterminados codificados suelen ser breves y pueden verse fácilmente comprometidos incluso por atacantes que no tienen acceso a la base del código.
- En segundo lugar, los secretos predeterminados codificados pueden causar problemas si es necesario cambiarlos o reemplazarlos.
- Y lo más importante es que siempre existe la posibilidad de establecer accidentalmente secretos predeterminados para los servicios de producción, lo que puede generar vulnerabilidades de seguridad y hacer que la producción sea insegura de forma predeterminada.

- **Ejemplo de código NO conforme:**

```
#MD5
usuario:$1$0aI0baup$QSOwAQexU5cWBdkvRcJZM1:19000:0:99999:7:::

#bcrypt
usuario:$2y$10$DmFiNUeGvhCh9//LA0hCXujxAnwkKLsjt7OHob5IxhkSGKfgONkWK:19000:0:99999:7:::

# SHA-256
usuario:$5$CPoDVd6t59j/w6rR$Umw3LuzCDnYB5MosjcEmLo1Pvf6XPdgtbmUdXP0QBC.:19000:0:99999:7:::
```

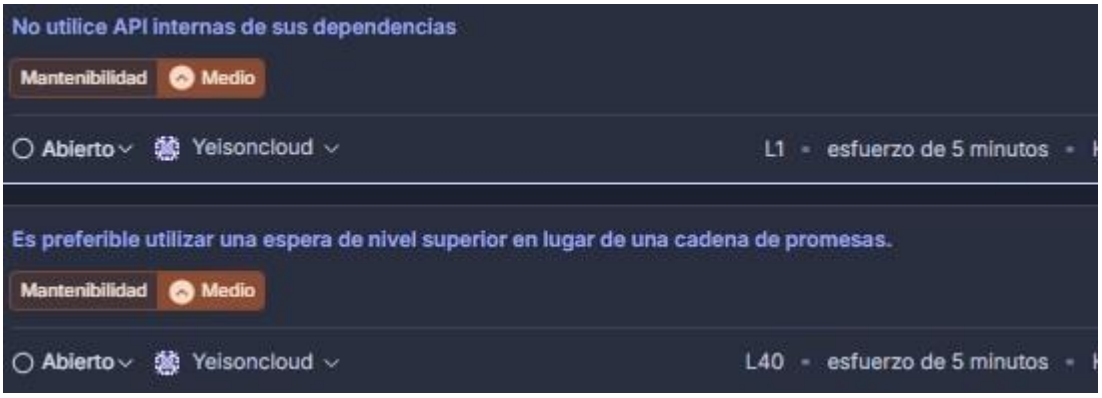
- **Ejemplo de código conforme o recomendado:**

```
#MD5
usuario:$1$salt$redactedhash:19000:0:99999:7:::

#bcrypt
usuario:$2y$salt$redactedhash:19000:0:99999:7:::

# SHA-256
usuario:$5$salt$redactedhash:19000:0:99999:7:::
```

4). Fiabilidad:

Gravedad del impacto:

Este problema tiene un impacto medio en la capacidad de mantenimiento de nuestro software, la gravedad ahora está directamente relacionada con la calidad del software afectada.



La API pública de un framework, plugin o biblioteca se ajusta a la forma en que su proveedor la diseñó, la estabilidad y compatibilidad de la API, dentro del mismo número de versión principal de una biblioteca se garantizan únicamente para su API pública.

Las API internas son meros detalles de implementación y son propensas a cambios importantes a medida que cambia la implementación de la biblioteca, por lo tanto, los usuarios no deben usarlas, ni siquiera cuando estén visibles.

- **Impacto potencial y estabilidad del código:**

Si no se soluciona, nuestro código podría fallar cuando la biblioteca se actualice a una nueva versión, incluso si solo cambia el número de versión secundaria o el número de parche**.**

5). ¿Cómo solucionar el problema?

Reemplazamos el uso de la API interna con la API pública diseñada para el caso de uso, esto puede implicar una refactorización del código afectado si no hay un reemplazo completo disponible en la API pública, si se requiere una funcionalidad específica, copiar las partes necesarias de la implementación en nuestro código puede ser incluso mejor que usar la API interna.

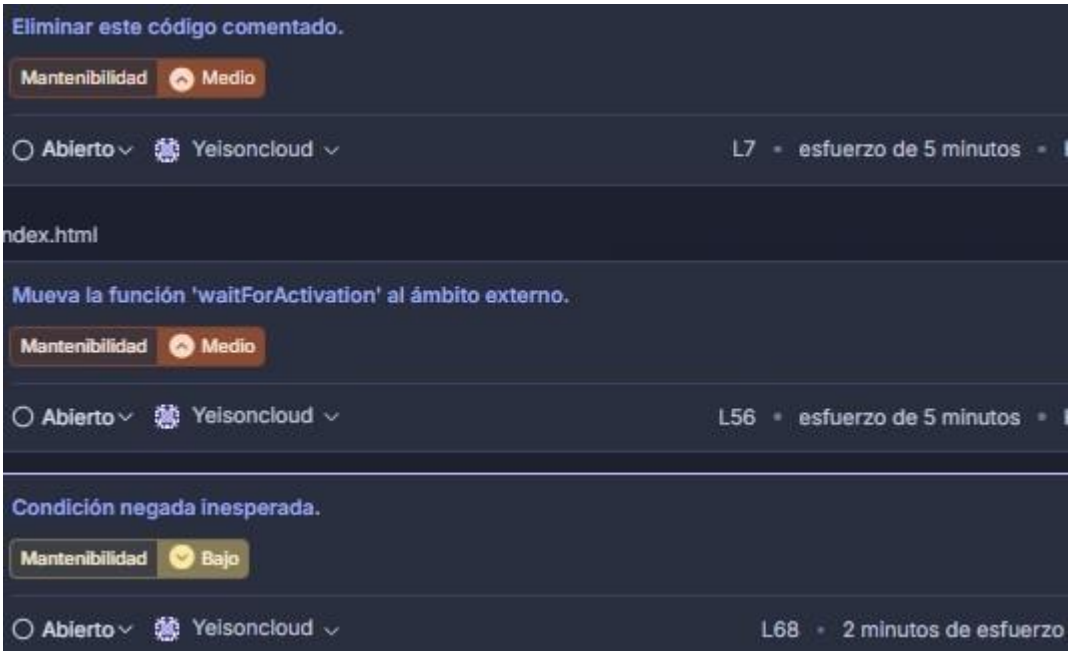
- **Ejemplo de código NO conforme**

```
importar { _parseWith } desde  './node_modules/foo/helpers'
```

- **Ejemplo de código conforme**

```
importar { parse } desde  'foo'
```

6). Mantenibilidad:



- El código comentado distrae la atención del código que se está ejecutando, crea ruido que incrementa el código de mantenimiento y como nunca se ejecuta, rápidamente queda obsoleto e inválido, el código comentado debe eliminarse y puede recuperarse del historial de control de origen si es necesario.

- Cuando las funciones se definen en ámbitos anidados innecesariamente, se crean varios problemas:

- **Impacto en el rendimiento:** Las funciones definidas dentro de otras funciones se recrean cada vez que se ejecuta la función externa, esto desperdicia memoria y tiempo de procesamiento, especialmente en código de llamadas frecuentes.

- **Optimización del motor:** Los motores como V8 tienen límites de optimización. Las funciones en ámbitos anidados son más difíciles de optimizar, lo que puede ralentizar la aplicación.
- **Legibilidad del código:** Las funciones en niveles superiores son más fáciles de encontrar y comprender, cuando una función no depende de su contexto, ubicarla en el nivel superior hace que la estructura del código sea más clara.
- **Uso de memoria:** Cada instancia de función ocupa memoria, crear la misma función repetidamente en ámbitos anidados aumenta el consumo de memoria innecesariamente.
- La regla identifica funciones que no capturan variables de su ámbito circundante, lo que significa que pueden moverse de forma segura a un nivel superior sin cambiar su comportamiento.

7). ¿Como solucionarlo?

Movemos la declaración de función al nivel de módulo cuando no captura ninguna variable del ámbito circundante.

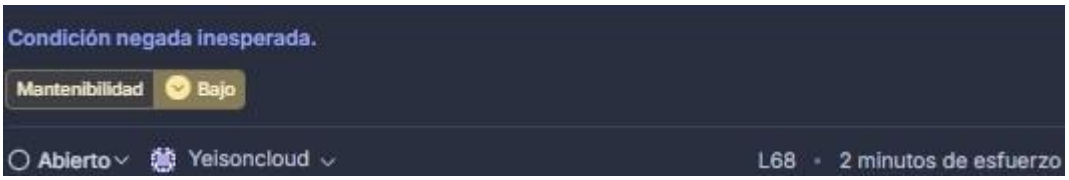
- **Ejemplo de código NO conforme**

```
export function doFoo ( foo ) {
        // No captura nada del alcance
        function doBar ( bar ) { // No conforme
                return bar === 'bar' ;
        }

        devolver doBar;
}
```

- **Ejemplo de código conforme**

```
función doBar ( bar ) {
        return bar === 'bar' ;
}

función de exportación doFoo ( foo ) {
        devolver doBar;
}
```

Las condiciones negadas en las sentencias if-else pueden dificultar la lectura y comprensión del código, al ver [falta contexto] if (!condition), el cerebro tiene que procesar la negación, lo que añade carga cognitiva.



Las condiciones positivas suelen ser más fáciles de entender porque describen lo que es verdadero en lugar de lo que no lo es. Cuando se tienen ramas if y else, normalmente se puede invertir la condición e intercambiar las ramas para que el código sea más legible.

Por ejemplo, requiere que pienses "si el usuario NO está activo", mientras que es más directo: "si el usuario está activo".if (!user.isActive)if (user.isActive)

Este patrón es especialmente problemático con:

Negación booleana usando el !operador

Comparaciones de desigualdades como !==y!=

Expresiones complejas donde la negación hace que la lógica sea más difícil de seguir. La regla solo marca los casos en los que hay una cláusula else porque las declaraciones if simples con condiciones negadas a veces son la forma más clara de expresar "hacer algo cuando esta condición es falsa".

8). Impacto potencial:

Si bien este problema no afecta la funcionalidad, sí afecta la mantenibilidad y la legibilidad del código, el código con condiciones negadas tarda más en comprenderse, especialmente para los desarrolladores:

- Nuevo en el código base.
- Trabajando bajo presión.
- Hablantes no nativos de inglés.
- Depuración de lógica compleja.

Una mejor legibilidad da como resultado menos errores, revisiones de código más rápidas y un mantenimiento más sencillo.

9). ¿Como solucionarlo?

Inviertimos la condición booleana negada e intercambie los bloques if y else.

- **Ejemplo de código NO conforme**

```
if (!isValid) { //
  Error de manejo no conforme ();
} demás {
  procesoData ();
 }
```

- **Ejemplo de código conforme**

```
si (es Válido) {
  procesoData ();
} de lo contrario {
   manejarError ();
}
```

10). Referencias.

https://sonarcloud.io/project/overview?id=Yeisoncloud_StockFlow