**Ejercicio Práctico – Revisión de Código y Pipeline**

1. Identificación de vulnerabilidades en el código (SAST)

| **Vulnerabilidad** | **Archivo y Línea** | **Severidad** | **Descripción Detallada** | **Remediación** |
| --- | --- | --- | --- | --- |
| **Server-Side Request Forgery** | src/main/java/org/owasp/webgoat/lessons/jwt/claimmisuse/JWTHeaderJKUEndpoint.java, línea 57 | Crítica | El código construye una URL con datos de entrada del usuario, permitiendo a un atacante enviar peticiones a servidores internos o externos. El impacto es la potencial exposición de datos sensibles y el acceso no autorizado a la red. | Usar una lista de URLs permitidas (allowlist) y validar la entrada para asegurar que solo se usan URLs seguras. |
| **Path Traversal** | src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java, líneas 100-101 | Crítica | Se detectó que la entrada del usuario podría controlar la ubicación de un archivo, permitiendo a un atacante navegar hacia atrás en el directorio con ../. Esto es un tipo de **Broken Access Control (A01:2021)**. | Sanitizar la entrada para que solo contenga el nombre del archivo, sin la ruta. Se puede usar un método de utilidad como org.apache.commons.io.FilenameUtils.getName(...). |
| **Inyección SQL** | src/main/java/org/owasp/webgoat/lessons/sqlinjection/introduction/SqlInjectionLesson8.java, línea 142 | Alta | El código genera consultas SQL de forma dinámica, permitiendo que la entrada del usuario manipule la lógica de la consulta. Un atacante puede acceder a información no autorizada o ejecutar comandos del sistema operativo. | Reemplazar las consultas dinámicas por consultas parametrizadas o PreparedStatement para separar los datos del usuario de la lógica de la consulta SQL. |
| **Cross-Site Request Forgery (CSRF)** | src/main/java/org/owasp/webgoat/container/service/SessionService.java, línea 22 | Media | Un método con la anotación RequestMapping no especifica el método HTTP. Esto podría hacer que una acción que cambia el estado de la aplicación sea vulnerable a un ataque CSRF, engañando a un usuario para que realice acciones no deseadas. | Añadir el campo method para especificar explícitamente el método HTTP (ej. RequestMethod.POST) y utilizar tokens anti-CSRF para los cambios de estado. |
| **Cryptographic Failures (Uso de algoritmos débiles)** | src/main/java/org/owasp/webgoat/lessons/cryptography/HashingAssignment.java, línea 39 | Media | La aplicación utiliza un algoritmo de hash inseguro o arriesgado (MD5 o SHA1). Estos algoritmos son vulnerables a colisiones, lo que puede comprometer la integridad y confidencialidad de la información. | Se recomienda cambiar a algoritmos de hash más seguros como Argon2id o PBKDF2, especialmente para el almacenamiento de contraseñas. |

1. Crea un pipeline CI/CD con etapas de seguridad

Desde mi punto de vista como profesional, la elección de las pruebas de seguridad fue estratégica. Me fui por el análisis de composición de software y en el escaneo de secretos porque son dos de los riesgos más críticos y de alto impacto en aplicaciones modernas. El SCA es vital porque la mayoría de nuestro código depende de librerías de terceros, y una vulnerabilidad en una de ellas puede comprometer todo el sistema. El escaneo de secretos, por otro lado, es un "quick win" fundamental para evitar que credenciales, claves API o tokens de acceso se filtren accidentalmente al repositorio.

Para llevar a cabo estas tareas, configuré un archivo .gitlab-ci.yml con jobs dedicados en la etapa de security. Para el SCA, utilicé Trivy, y para el escaneo de secretos, usé gitleaks, porque son herramientas open source robustas y fáciles de integrar. El resultado es que el pipeline no solo corre las pruebas, sino que también guarda los reportes de Trivy y gitleaks como artefactos. Estos reportes, que ahora están disponibles en el repositorio junto con los archivos del pipeline.

stages:

- build

- security

build\_job:

stage: build

image: maven:latest # Usa una imagen de Docker con Maven para proyectos Java

script:

- echo "Compilando la aplicación WebGoat..."

- mvn clean install # Comando estándar para limpiar y compilar el proyecto

# -----------------------------------------------------------------------------

# Etapa de seguridad

#

# En esta etapa se ejecutan los análisis de seguridad. Los jobs aquí

# corren en paralelo por defecto si están en el mismo stage.

# -----------------------------------------------------------------------------

sca\_job:

stage: security

image:

name: "aquasec/trivy:latest" # Trivy es una herramienta open-source para SCA

entrypoint: [""] # Sobrescribe la entrada para ejecutar comandos de shell

script:

- echo "Realizando análisis de composición de software (SCA)..."

- trivy fs --format json -o sca-report.json . # Escanea el filesystem en busca de dependencias vulnerables

artifacts:

when: always # Guarda el reporte incluso si el job falla

paths:

- sca-report.json # El reporte se guarda como un artefacto

allow\_failure: true # El pipeline continúa aunque se encuentren vulnerabilidades

secrets\_scan\_job:

stage: security

image: "zricethezav/gitleaks:latest" # Gitleaks es una herramienta para escanear secretos

script:

- echo "Escaneando el repositorio en busca de secretos..."

- gitleaks detect --report-path=secrets-report.json --redact --exit-code 0 # Detecta secretos y crea un reporte

artifacts:

when: always

paths:

- secrets-report.json # Guarda el reporte de secretos como artefacto

allow\_failure: true

1. Plantea controles de calidad de código seguro

En cuanto a controles para este código plantearía un flujo de seguridad dentro del CI/CD que sea el eje central en donde se automatizan todas las pruebas de seguridad que no pueden ser pasadas por alto, que es el primer filtro y el más importante ya que en estos jobs incluiría los sast, sca y analisis de secrets para tener un buen primer filtro con respecto a la postura de seguridad actual.

Después de que se ejecutan los escaneos, implementaría una prueba de calidad que evalúa los resultados y decide si el código cumple con los estándares de seguridad definidos.

Para medir y mejorar la seguridad, me enfocaría en las siguientes métricas:

* Vulnerabilidades críticas y altas por pull request
* Tiempo Promedio de remediación.
* Densidad de vulnerabilidades

Definimos umbrales claros para que no haya dudas sobre qué es aceptable.

* Vulnerabilidades Críticas: El umbral es cero. Un pull request con un hallazgo crítico no puede ser aprobado ni fusionado.
* Vulnerabilidades Altas: Se permite un número muy bajo), pero con un plazo estricto para su corrección. La meta sigue siendo cero.
* Vulnerabilidades Medias y Bajas: No bloquean el pipeline, pero se registran en un sistema de seguimiento (ej. Jira) para ser corregidas en iteraciones futuras, evitando que se acumulen.

Configuraría el primer filtro de fallo del pipe en analisis de calidad, en donde se encuentre un parametro que permita fallar el pipe segun los umbrales es decir si tengo mas de 5 criticas que falle.

En el SAST también incluiria un exit code 1 para fallar si encuentra findings peligrosos.