Evaluación Técnica - Consultor Senior de Seguridad de Aplicaciones

Consultor:

Erazo Mendoza Jeremy Sebastián

Offensive Security and Exploit Development Engineer

Quito, Junio del 2025

Contenido

Respuestas y documentación del Pipeline

Selecciona mínimo 2 pruebas a incluir en el pipeline y justifica por qué las seleccionaste sobre las demás.

En el diseño del pipeline CI/CD, se ha optado por incluir dos pruebas de seguridad automatizadas que consideramos estratégicas en la detección temprana de vulnerabilidades: **SAST (Static Application Security Testing)** y **SCA (Software Composition Analysis)**.

A continuación, se justifica su elección desde una perspectiva técnica y de gestión de riesgos.

1. Static Application Security Testing (SAST)

SAST es una prueba de seguridad en etapa temprana (shift-left) que permite analizar el código fuente de la aplicación antes de que se ejecute. Su principal ventaja es que puede integrarse de forma directa en el ciclo de desarrollo (por ejemplo, al momento del commit o build), permitiendo detectar vulnerabilidades como *inyecciones de código*, *fugas de información*, *referencias inseguras a objetos*, y errores de lógica antes de que el código llegue a entornos de staging o producción.

Desde una perspectiva técnica, SAST ofrece trazabilidad a nivel de función y flujo de datos, facilitando a los desarrolladores identificar el origen exacto de la vulnerabilidad, su propagación y posible impacto. Esta granularidad lo convierte en una herramienta clave para fortalecer la calidad del código y reducir la deuda técnica de seguridad.

A diferencia de pruebas dinámicas (como DAST), SAST no requiere una instancia ejecutable de la aplicación, lo que lo hace idóneo para ser utilizado en fases tempranas del pipeline. Además, es compatible con los principios de “Secure by Design” al promover la identificación y remediación proactiva de errores estructurales.

2. Software Composition Analysis (SCA)

SCA responde a una problemática moderna crítica: la proliferación de dependencias externas en los proyectos de software. En la mayoría de aplicaciones actuales, más del 70% del código proviene de librerías, frameworks y paquetes de terceros. Este ecosistema expone a las organizaciones a vulnerabilidades conocidas (CVE) que pueden ser explotadas incluso si el código propio está libre de errores.

La inclusión de SCA en el pipeline permite escanear automáticamente todas las dependencias declaradas (por ejemplo, en archivos como requirements.txt, package.json, pom.xml, etc.) y contrastarlas con bases de datos de amenazas como la *National Vulnerability Database (NVD)* o *GitHub Security Advisories*. Además, permite verificar aspectos legales como el cumplimiento de licencias open-source (GPL, LGPL, MIT), que podrían derivar en riesgos de cumplimiento normativo o sanciones regulatorias.

Desde una perspectiva operativa, SCA tiene una baja latencia de ejecución y puede ejecutarse incluso antes de que la aplicación sea empaquetada. Esto reduce

considerablemente los tiempos de respuesta ante alertas y facilita la rotación preventiva de versiones vulnerables.

Proporciona el archivo de configuración del pipeline con comentarios que expliquen cada etapa.

Este pipeline de CI/CD está diseñado con enfoque DevSecOps e implementado en GitLab CI/CD. Se ha estructurado para ejecutar una serie de pruebas automáticas que refuercen la seguridad del código desde la fase de desarrollo hasta su despliegue, siguiendo prácticas actuales de hardening de SDLC y cumplimiento de estándares como OWASP SAMM o NIST SSDF.

A continuación, se detallan los principales componentes de seguridad incorporados en el pipeline:

1. SAST – Static Application Security Testing

Se ejecuta al inicio del pipeline y tiene como propósito detectar vulnerabilidades en el código fuente antes de que este se compile.

Motivo técnico: Integrar SAST desde etapas tempranas permite identificar problemas como inyecciones, fallos de validación o mal uso de librerías inseguras sin requerir la ejecución de la app. Se ejecuta con herramientas como Semgrep o SonarQube, que analizan patrones de código. Esto reduce significativamente el costo de corregir errores, ya que se encuentran *shift-left*.

2. SCA – Software Composition Analysis

Evalúa las dependencias del proyecto, buscando vulnerabilidades conocidas (CVEs) en paquetes de terceros.

Motivo técnico: Gran parte del código en proyectos modernos proviene de librerías externas. Usar Trivy o Syft para generar un mapa de dependencias permite identificar rápidamente componentes con vulnerabilidades críticas. El control de estos componentes es fundamental para prevenir ataques en cadena de suministro.

3. DAST – Dynamic Application Security Testing

Ejecuta pruebas de caja negra sobre la aplicación desplegada para identificar vulnerabilidades en tiempo de ejecución.

Motivo técnico: DAST es esencial para validar configuraciones reales, cabeceras, sesiones, rutas abiertas y fallos en la lógica de negocio que no se detectan por SAST. Se emplea OWASP ZAP o Nikto como motor de escaneo. Este test suele integrarse en un entorno staging temporal tras el build.

4. SBOM – Software Bill of Materials

Genera una lista completa de los componentes del software, versión a versión.

Motivo técnico: Un SBOM se convierte en una pieza clave para responder ante incidentes de seguridad, ya que permite rastrear rápidamente si una versión de nuestro software contiene una librería vulnerable. Además, es requerido por estándares como la Executive Order 14028 en EE.UU. Se usa Syft o CycloneDX CLI.

5. Gestión de Secretos – Secrets Scanning

Busca claves, tokens, contraseñas y secretos inadvertidamente comprometidos en el repositorio.

Motivo técnico: Las filtraciones de secretos siguen siendo una de las principales causas de compromisos en la nube. Herramientas como Gitleaks o TruffleHog son esenciales para evitar la exposición de credenciales hardcodeadas antes del build.

6. Validaciones y Condiciones de Despliegue

Antes de cualquier despliegue a producción, se agregan condiciones para bloquear la entrega si se detectan vulnerabilidades críticas.

Motivo técnico: Esta etapa aplica lógica de "gates" en el pipeline: si se detecta una vulnerabilidad de severidad crítica en el SAST/SCA/DAST, el despliegue se cancela automáticamente. Así se asegura que sólo builds seguros lleguen al entorno productivo, alineándose con políticas tipo “Fail Fast”.

Mecanismos para garantizar prácticas de codificación segura y mantener la calidad del código en WebGoat

Para asegurar que el código en WebGoat cumpla con altos estándares de seguridad y calidad, es fundamental integrar controles automáticos en el pipeline desde las primeras etapas del desarrollo. Esto implica incorporar análisis estático (SAST), dinámico (DAST) y escaneo de dependencias (SCA) para detectar vulnerabilidades o malas prácticas lo antes posible. Más allá de la automatización, es vital que el equipo de desarrollo implemente revisiones de código enfocadas en seguridad, con personas que tengan experiencia en identificación de riesgos y vectores de ataque reales. La capacitación constante en prácticas seguras y el seguimiento de estándares reconocidos, como OWASP y normas ISO, deben ser parte integral del proceso. Así, la calidad y seguridad del código se mantienen como un compromiso constante y no solo un requisito.

i. Métricas específicas por monitorear

Para medir la efectividad de las prácticas de seguridad y calidad, es necesario monitorear métricas claras y accionables. Se debe llevar control del número y severidad de vulnerabilidades detectadas en cada build, clasificadas según CVSS para priorizar su atención. También es importante medir la cobertura de pruebas automatizadas, buscando que supere el 85% para asegurar una buena validación del código. Otro indicador relevante es el tiempo desde la detección hasta la corrección de vulnerabilidades críticas, con un objetivo de menos de 48 horas, siguiendo buenas prácticas en gestión de vulnerabilidades. Además, la complejidad del código, especialmente la ciclomática, ayuda a identificar áreas que podrían presentar riesgos por su dificultad de mantenimiento y prueba. Estas métricas deben estar visibles para el equipo y actualizadas con frecuencia para guiar las acciones necesarias.

ii. Umbrales aceptables para vulnerabilidades

Los umbrales deben ser estrictos pero realistas. Vulnerabilidades con puntaje CVSS igual o superior a 9 se consideran críticas y no deben existir en el código activo; su detección debe detener cualquier avance hasta su corrección. Para vulnerabilidades altas (CVSS entre 7 y 8.9), se permite un máximo de una vulnerabilidad abierta por sprint, siempre con un plan claro para corregirla en menos de 48 horas. Vulnerabilidades de nivel medio o bajo se deben manejar mediante ciclos de revisión continuos, evitando que se acumulen y generen deuda técnica o riesgos futuros. Este enfoque permite balancear la seguridad con la continuidad del desarrollo sin perder foco en los riesgos reales.

iii. Mecanismos para bloquear el pipeline si se detectan riesgos críticos

El pipeline debe detenerse automáticamente ante la detección de vulnerabilidades críticas (CVSS ≥ 9), utilizando análisis estático, dinámico y escaneo de dependencias. Esto evita que código inseguro avance a etapas posteriores o producción. Además, se deben generar alertas inmediatas al equipo para asegurar una respuesta rápida. Para vulnerabilidades críticas, además del bloqueo automático, se debe contar con una revisión manual por expertos en seguridad que certifiquen la resolución antes de liberar el pipeline. Este doble filtro garantiza un control riguroso, minimizando riesgos y asegurando que la seguridad sea un proceso continuo y efectivo.