Evaluación Técnica - Consultor Senior de Seguridad de Aplicaciones

Consultor:

Erazo Mendoza Jeremy Sebastián

Offensive Security and Exploit Development Engineer

Quito, Junio del 2025

Contenido

Fase #1 – Preguntas teóricas

De acuerdo con tu experiencia profesional, enumera y explica las etapas del SDLC donde incorporaste controles de seguridad.

La incorporación de controles de seguridad en el SDLC es un elemento esencial para garantizar la resiliencia de cualquier sistema. Desde una perspectiva técnica y estratégica, la seguridad debe estar presente en cada fase del ciclo, desde la planificación hasta la operación en producción. Este enfoque no solo previene vulnerabilidades técnicas, sino que también permite que el producto final cumpla con estándares regulatorios, minimice el riesgo reputacional y mantenga la confianza del usuario.

En la etapa de planificación, los controles de seguridad se orientan a la identificación temprana de riesgos. Es el momento en el que se definen los objetivos de protección, el alcance de seguridad y las amenazas relevantes para el sistema. Incluir análisis de riesgo y modelado de amenazas en esta etapa permite alinear los esfuerzos de seguridad con los activos críticos del negocio.

Durante la fase de requisitos, la seguridad debe formalizarse a través de especificaciones claras que contemplen aspectos como control de acceso, protección de datos, integridad de las comunicaciones y cumplimiento normativo. Los requisitos de seguridad no funcionales deben documentarse con la misma rigurosidad que los funcionales.

En el diseño, se deben aplicar principios arquitectónicos de seguridad como defensa en profundidad, separación de responsabilidades, y el principio de mínimo privilegio. Aquí también se definen los mecanismos de control de autenticación, autorización, cifrado y segmentación. Un diseño seguro reduce considerablemente la probabilidad de introducir errores críticos durante el desarrollo.

Durante el desarrollo, se aplican controles técnicos orientados a prevenir la introducción de vulnerabilidades. Esto incluye la adopción de prácticas de codificación segura, revisiones de código, y la integración de herramientas de análisis estático (SAST), detección de secretos y validación de dependencias. La automatización de estos controles permite mantener un flujo de trabajo eficiente sin sacrificar visibilidad ni trazabilidad.

En la etapa de pruebas, se incorporan mecanismos de validación técnica como análisis dinámico (DAST), pruebas de composición de software (SCA), pruebas manuales específicas y fuzzing. Esta fase busca detectar comportamientos anómalos, errores lógicos y brechas de seguridad antes del despliegue. Un enfoque efectivo combina herramientas automatizadas con validación manual selectiva.

Durante el despliegue, la seguridad se enfoca en el aseguramiento de la infraestructura y la cadena de suministro. La verificación de integridad de artefactos, generación de SBOMs y escaneo de contenedores son prácticas clave. Además, deben definirse controles para la gestión de secretos, configuración segura y despliegues reproducibles.

Finalmente, en la fase de mantenimiento, la seguridad debe sostenerse de forma continua mediante monitoreo, gestión de parches, revisión de vulnerabilidades emergentes y respuesta

ante incidentes. Este ciclo cerrado permite adaptar el sistema a nuevas amenazas sin comprometer la estabilidad ni la seguridad operacional.

Este enfoque integral permite no solo reducir el costo de corrección de fallas de seguridad, sino también establecer una cultura de desarrollo seguro que trasciende herramientas específicas y se convierte en parte del ciclo de vida del producto.

¿En qué parte del pipeline CI/CD aplicaste las pruebas SAST, DAST, SCA?

Las pruebas SAST, SCA y DAST se aplican en puntos distintos del pipeline CI/CD, de acuerdo a su naturaleza técnica y al impacto que generan en los tiempos de integración y entrega.

Las pruebas SAST se integran en etapas tempranas del pipeline, inmediatamente después del commit y durante el proceso de build. Esto permite que el análisis del código fuente o bytecode se realice de forma temprana, evitando la propagación de vulnerabilidades hacia etapas posteriores. Se ejecutan como parte del workflow de integración continua, normalmente en los *pull requests*, para garantizar que solo se integren ramas libres de fallas críticas o high.

Las pruebas SCA también se colocan en la etapa temprana del pipeline, junto al análisis estático. Analizan el árbol de dependencias declarado y transitorio para detectar vulnerabilidades conocidas (con base en CVEs o bases como OSS Index, Sonatype, etc.). Su ejecución temprana permite detener builds que incorporen paquetes comprometidos, desactualizados o con licencias incompatibles. Generalmente se aplican en paralelo al SAST para optimizar tiempos de ejecución.

En contraste, las pruebas DAST se ejecutan en entornos desplegados, ya sea en staging o ambientes temporales levantados por el pipeline, una vez que la aplicación está disponible vía HTTP/S. Se ejecutan después del deploy automatizado, como parte de los test funcionales, en pipelines de entrega continua. En este punto, ya no se analiza código, sino comportamiento. Se evalúan endpoints expuestos, flujos lógicos y errores de configuración desde una perspectiva externa, sin acceso al código fuente.

La secuencia ideal en el pipeline es:

- Pre-build o post-commit: SAST + SCA
- Post-deploy: DAST

Describe cómo implementaste o implementarías un modelo de “shift-left security” en una organización que apenas comienza su madurez en DevSecOps.

El modelo lo implementaría empezando por integrar pruebas SAST y SCA en el pipeline de CI, ejecutadas en cada *pull request* y en el build principal. Semgrep, SonarQube o CodeQL se configuran con reglas personalizadas para el stack del equipo, y se establece una política de calidad que bloquea merges si hay findings críticos sin justificar.

A nivel de dependencias, uso SCA (como Snyk o OWASP Dependency-Check) al momento del build para evitar inclusión de librerías vulnerables o desactualizadas. Esto se refuerza con escaneo continuo en el repositorio y alertas integradas en los issues del proyecto.

Paralelamente, defino checklists de revisión de código seguras, específicas por lenguaje y tipo de componente. Estas se integran en el flujo de revisión con enfoque en uso seguro de funciones, manejo de errores, validación de input y exposición de datos sensibles. No se requiere aprobación de pull request sin revisión manual bajo estas guías.

En etapas post-deploy, se automatiza DAST en staging usando OWASP ZAP en modo *headless*, como parte del pipeline. Las pruebas dinámicas se lanzan tras cada despliegue de entorno efímero. También se incorporan validaciones de IaC con herramientas como tfsec o Checkov sobre Terraform y Docker, previo a aplicar cambios de infraestructura.

Para evitar fricción inicial, los hallazgos se clasifican por severidad. Al principio, sólo los críticos bloquean el pipeline; los demás generan alertas para análisis posterior. Esto permite alinear al equipo sin detener entregas.

Finalmente, se implementan métricas como porcentaje de builds fallidos por seguridad, tiempo de corrección por severidad y cobertura de análisis por tipo. Con eso se ajusta el modelo en función de los cuellos técnicos reales del equipo y se mantiene trazabilidad de mejoras en madurez.

Explica cómo has trabajado el análisis de vulnerabilidades comunes (OWASP Top 10) en un flujo CI/CD (procesos automáticos, tareas manuales)?

En la práctica, cuando integro seguridad en un pipeline CI/CD, el OWASP Top 10 es uno de los marcos de referencia principales para definir la cobertura mínima esperada. Lo abordo con una combinación bien organizada de análisis automático y validaciones manuales más enfocadas.

Empiezo por los análisis SAST. Los integro desde las primeras etapas del pipeline, generalmente en los *pull requests* o directamente sobre los *push* a ramas protegidas. Herramientas como Semgrep, CodeQL o SonarQube me han dado buenos resultados dependiendo del lenguaje. Con estas herramientas cubro principalmente vulnerabilidades como *Injection (A03)*, *Broken Access Control (A01)* y *Security Misconfiguration (A05)*. Las reglas están ajustadas por contexto, y si es posible, alineadas con las guías internas de desarrollo seguro de la organización. Además, configuro el pipeline para que findings críticos bloqueen el merge, mientras que los de menor severidad generan alertas visibles pero no detienen la entrega.

En paralelo, incorporo SCA justo después del análisis estático, en la etapa de *build*. Ahí utilizo herramientas como Snyk, OWASP Dependency-Check o npm audit, dependiendo del stack. Este análisis lo oriento a cubrir específicamente *Vulnerable and Outdated Components (A06)*. Es importante revisar tanto las dependencias directas como las transitivas, por lo que siempre habilito escaneo profundo. Si una dependencia tiene un CVE con CVSS crítico, el pipeline falla de forma automática y se genera una tarea en el backlog de remediación.

Luego, paso a la etapa dinámica. Una vez que el entorno de *staging* o *pre-producción* está desplegado, integro DAST, generalmente usando OWASP ZAP en modo *headless*, o también

BurpSuite Pro cuando el análisis requiere una lógica más específica. Aquí busco vulnerabilidades como *XSS (A03)*, *Insecure Design (A04)*, *Cryptographic Failures (A02)*, y *Authentication Failures (A07)*. Lo interesante del DAST es que me permite validar lo que no detecta el análisis estático: errores en el comportamiento real, configuraciones erróneas, respuestas inseguras del servidor, o flows que exponen más de lo que deberían.

Ahora, más allá de las herramientas automáticas, siempre incluyo una capa manual de validación. Me enfoco en las partes críticas del sistema: autenticación, gestión de sesiones, endpoints con lógica de negocio sensible, etc. Aquí hago pruebas como *IDOR*, abuso de roles, bypass de lógica de permisos, validación débil de tokens y todo lo que cae dentro de *Broken Access Control (A01)* o *Business Logic Flaws*. En estos casos utilizo BurpSuite con extensiones como Autorize, Turbo Intruder o ParamMiner para automatizar ciertos casos, pero sin perder la lógica de ataque controlado.

También incluyo revisiones de infraestructura como código, sobre todo cuando trabajo con Terraform, Docker o Kubernetes. Para eso uso Checkov o tfsec, lo que me permite cubrir aspectos del OWASP como *Security Misconfiguration (A05)* antes de que se apliquen los cambios en la nube.

Finalmente, todos los resultados —automáticos y manuales— los categorizo bajo el identificador OWASP correspondiente (por ejemplo: A01-BrokenAccessControl, A03-XSS) y los vinculo al sistema de gestión de tickets (GitHub Issues, Jira, etc.). Esto facilita el seguimiento y permite generar métricas como porcentaje de findings por categoría OWASP, tiempo de resolución y densidad de vulnerabilidades por módulo.

Diseña una arquitectura segura de una aplicación web con autenticación federada, sesiones seguras y protección contra ataques comunes.

Diseño la arquitectura partiendo de los vectores que suelen explotarse en entornos reales. No se trata solo de agregar controles, sino de eliminar superficies innecesarias, reducir privilegios por diseño y anticipar abuso del flujo de autenticación.

La autenticación federada la implemento con OpenID Connect sobre OAuth2, usando un IdP externo (Auth0, Azure AD, Okta) aislado del backend. Solo acepto tokens firmados con claves válidas (JWKs rotativas) y valido iss, aud, exp, nonce y at_hash. Si el frontend es público, el flujo incluye PKCE y detección de reusos de código para evitar *authorization code interception*. Revoco tokens si detecto *client impersonation* por falta de validación de estado o redirect_uri.

Las sesiones se gestionan *stateless* con JWT firmados con HS256 o RS256, expiración corta (≤15 min) y rotación mediante refresh_token sólo en backchannel. Nada de sesiones en frontend, ni tokens en localStorage — solo cookies Secure, HttpOnly, SameSite=Strict, renovadas con cada interacción válida. Esto mitiga XSS + session fixation. Si necesito almacenar estado (por ejemplo, para invalidación manual), uso Redis con TTL y revocación activa.

El backend está detrás de un API Gateway que aplica *rate limiting*, validación estructural de requests y bloqueo de User-Agent/referers anómalos. Se rechazan peticiones sin Content-Type válido o con Origin/Referer inconsistentes. Se aplican controles antifraude con fingerprints, y hay lógica específica para detectar uso de proxies/TOR en flujos sensibles.

A nivel de capa lógica, aplico validación estricta de claims y control de acceso por contexto, no por endpoint. Cualquier recurso accesible mediante resource_id o user_id está validado server-side, evitando IDOR incluso con JWT válidos. Además, toda funcionalidad crítica (cambio de contraseña, gestión de API keys, eliminación de datos) requiere reautenticación o confirmación secundaria.

Desde el lado ofensivo, sé que donde fallan estas arquitecturas es en los extremos: callbacks sin validación fuerte, tokens sin expiración real, validación de sesión en el frontend, o errores de configuración en CORS o CSP. Por eso el diseño parte de asumir que el atacante ya está en el navegador o interceptó el tráfico. Si eso ocurre, no debe escalar más allá de una sesión limitada, no persistente, y revocable.

¿Qué consideraciones de seguridad tomarías en una arquitectura serverless?

Desde mi experiencia en ofensiva, cuando veo una arquitectura serverless, pienso más en cómo se pueden abusar las conexiones y permisos que en buscar vulnerabilidades tradicionales de servidores. No hay un sistema operativo directamente explotable, pero hay muchas puertas abiertas si no controlas bien las funciones, los triggers y los privilegios.

Lo primero que hago es revisar que cada función tenga el menor privilegio posible. Esto significa que si una función solo debe escribir en un bucket de S3, no debe tener permisos para listar otros buckets ni acceder a secretos que no necesita. En el pasado, he visto cómo un error de este tipo permite escalar desde una función comprometida a toda la infraestructura.

Luego, me aseguro que los triggers externos estén protegidos. Por ejemplo, si una función se activa con un endpoint HTTP, no puede estar abierta a todo el mundo. Siempre debe haber un mecanismo de autenticación o validación del origen. Me he encontrado funciones que cualquiera podía invocar porque no tenían ni siquiera un token de autorización, lo cual es un grave error.

Los secretos son otro punto crítico. Nada de variables de entorno planas con claves o contraseñas. Uso gestores de secretos como AWS Secrets Manager o Vault, con acceso controlado y rotación automática. El objetivo es que si una función se compromete, el atacante no tenga acceso directo a credenciales que le permitan pivotar.

Además, siempre insisto en la monitorización exhaustiva: logs detallados de cada invocación, con origen, parámetros y respuesta. Esto me permite detectar patrones raros o picos inusuales de ejecución que pueden indicar ataques o abuso.

No menos importante, defino límites estrictos para las funciones: tiempo de ejecución, memoria y tamaño máximo de payload. Esto evita ataques como ZIP bombs o que una función se quede corriendo infinitamente causando DoS interno.

También reviso muy bien la validación de entradas. Serverless suele procesar mucho JSON o eventos; por eso la validación debe ser muy estricta para evitar inyección, path traversal o SSRF hacia APIs internas como la metadata de AWS.

Finalmente, nunca dejo de auditar la infraestructura como código con herramientas como Checkov o tfsec, para evitar despliegues con roles demasiado amplios o funciones expuestas sin autenticación.

Explica cómo has utilizado el CWE en tu experiencia profesional para priorizar la corrección de vulnerabilidades.

En mi experiencia, el CWE (Common Weakness Enumeration) es una referencia útil para clasificar y entender las categorías técnicas de vulnerabilidades, ayudando a estandarizar el lenguaje entre equipos de desarrollo, seguridad y gestión. Sin embargo, su antigüedad y naturaleza generalista hacen que no pueda ser el único criterio para priorizar correcciones.

El CWE ofrece una base sólida para identificar tipos de fallas como *buffer overflows*, *injection*, o *cross-site scripting* pero no refleja por sí solo la criticidad real que una vulnerabilidad representa para una organización específica. En la práctica, he observado que la criticidad debe contextualizarse con factores mucho más amplios.

Para priorizar correctamente, integro la información del CWE con:

- Contexto de la Inteligencia de Amenazas (CTI): Evaluar si existen explotaciones activas o campañas específicas dirigidas a la vulnerabilidad.
- Naturaleza y criticidad del activo afectado: Por ejemplo, una vulnerabilidad crítica en un sistema interno poco expuesto no tiene la misma prioridad que una media en un sistema de producción expuesto al internet.
- Explotabilidad técnica real: Considerar si la vulnerabilidad requiere condiciones muy específicas para ser explotada o si existen exploits públicos funcionales y fáciles de aplicar.
- Impacto en la confidencialidad, integridad y disponibilidad: Cómo afectaría al negocio o a los datos críticos en caso de explotación exitosa.

De esta manera, la clasificación CWE actúa como punto de partida para la identificación técnica, pero la priorización se enriquece con análisis de riesgo real y contexto operacional.

Con este enfoque, he logrado optimizar recursos en los procesos de remediación, enfocando la atención en aquellas vulnerabilidades que realmente representan un riesgo tangible y no solo un problema teórico.

Describe cómo has mantenido un SBOM actualizado en proyectos anteriores y por qué consideras que es importante basado en tu experiencia.

En proyectos anteriores, he utilizado principalmente Snyk para mantener el SBOM (Software Bill of Materials) actualizado de forma automatizada. Lo integré directamente en el pipeline CI/CD, tanto en GitHub Actions como en GitLab CI, para que en cada *build* o *commit* relevante, Snyk generara el SBOM y lo almacenara junto al artefacto generado.

La configuración fue sencilla: con un solo comando (snyk sbom --format=cyclonedx1.4 --json > sbom.json), exportábamos el SBOM en formato estándar CycloneDX. Esto se ejecutaba

automáticamente en cada release o actualización de dependencias. En algunos casos, lo versionábamos junto con el Dockerfile o el package-lock.json, dependiendo del lenguaje.

Lo más útil fue vincular el SBOM con el motor de análisis de Snyk, que no solo listaba componentes, sino que alertaba sobre vulnerabilidades conocidas asociadas a cada librería. Esto permitió detectar, por ejemplo, cuándo una nueva CVE afectaba una versión que ya teníamos en producción desde semanas atrás.

En cuanto a su importancia, lo veo como una pieza crítica para la trazabilidad y para tener un control real de la superficie de ataque asociada a dependencias. Me ha tocado responder ante incidentes donde no había visibilidad de qué versiones exactas estaban desplegadas. Tener un SBOM actualizado cambia completamente ese escenario: te permite actuar con datos, no con supuestos.