

DevSecOps

Ingeniería de Software 2 (CC3091) - 2026

DevSecOps

Semestre 02, 2026

Introducción

Con integración y despliegue continuos, el tiempo entre un commit y producción se reduce de meses a horas. Una auditoría manual de dos semanas por versión no puede ejecutarse en cada despliegue.

La revisión manual de seguridad al final del ciclo no escala a un ritmo de varios despliegues por día.

DevSecOps implementa los controles de seguridad como etapas automatizadas del pipeline, ejecutadas en cada commit.

DevSecOps

El NIST describe DevSecOps como la práctica que integra la seguridad como un componente fundamental del modelo DevOps, desde las primeras etapas del desarrollo.

Los requisitos de seguridad se definen junto con los funcionales, y los controles los mantiene el mismo equipo que desarrolla y opera el sistema.

Se implementa con herramientas de análisis integradas al repositorio, al pipeline de CI/CD y al entorno de ejecución.

Importancia de la seguridad

El informe Cost of a Data Breach 2025 de IBM y el Ponemon Institute ofrece una medida concreta del problema.

- **Costo promedio global de una brecha:** 4.44 millones de dólares.
- **Costo promedio en Estados Unidos:** 10.22 millones de dólares, el más alto registrado para cualquier región.
- **Tiempo para identificar y contener una brecha:** 241 días en promedio, el valor más bajo en nueve años y aun así casi ocho meses.

Además del costo directo, una brecha tiene consecuencias técnicas, legales y operativas.

- **Reputación:** la pérdida de clientes es uno de los componentes principales del costo de una brecha.
- **Regulación:** leyes como el GDPR europeo imponen multas y obligaciones de notificación.
- **Continuidad:** un ataque de ransomware puede detener la operación completa de una organización.
- **Privacidad:** una brecha expone datos personales de usuarios que no controlan cómo se almacenan.
- **Daño en cadena:** una biblioteca o un proveedor comprometido afecta a todos los sistemas que dependen de él.

El modelo tradicional

En el modelo tradicional, la revisión de seguridad es una etapa posterior al desarrollo, ejecutada por un equipo separado antes de la liberación.

- Los hallazgos se reportan semanas después del commit que los introdujo.
- Corregir un defecto de diseño al final obliga a rehacer trabajo que ya se había dado por terminado.
- La capacidad del equipo de seguridad limita el número de liberaciones posibles.
- Los hallazgos no críticos se difieren y se acumulan como deuda de seguridad.

Shift left

Shift left significa mover las actividades de seguridad hacia la izquierda de la línea de tiempo del ciclo de vida, es decir, hacia el diseño y la codificación.

El objetivo, según la guía DevSecOps de OWASP, es detectar los problemas de seguridad, de diseño o de implementación, lo más rápido posible.

Un hallazgo reportado en el editor o en el pull request lo corrige el autor del cambio antes del merge. En producción requiere respuesta a incidentes, un parche de emergencia y análisis forense.

Shift left no elimina los controles en producción. Después del despliegue se publican CVE nuevos que afectan a código y dependencias que ya pasaron el pipeline, así que el monitoreo, el WAF y el escaneo periódico de dependencias siguen siendo necesarios.

Principios de DevSecOps

- **Responsabilidad compartida:** cada persona del equipo es responsable de la seguridad de lo que construye.
- **Automatización:** los controles se ejecutan en cada commit, sin depender de la memoria de nadie.
- **Retroalimentación temprana:** el hallazgo llega a quien escribió el código mientras todavía tiene el contexto.
- **Seguridad como código:** las políticas, las reglas de análisis y la infraestructura se versionan en el repositorio.
- **Mínimo privilegio:** cada persona, servicio y pipeline tiene solo los permisos que necesita.
- **Monitoreo continuo:** el sistema se observa en producción para detectar ataques y vulnerabilidades nuevas.

El marco SSDF del NIST

El Secure Software Development Framework (NIST SP 800-218) organiza las prácticas de desarrollo seguro en cuatro grupos.

- **Preparar la organización (PO):** definir requisitos de seguridad, roles, capacitación y herramientas.
- **Proteger el software (PS):** proteger el código, los artefactos y el proceso de build contra manipulación.
- **Producir software bien asegurado (PW):** diseñar con modelado de amenazas, revisar el código y probarlo.
- **Responder a vulnerabilidades (RV):** identificar, priorizar y corregir las vulnerabilidades que aparezcan después de la entrega.

Seguridad en el ciclo de vida

Cada etapa del ciclo de vida tiene su propio control de seguridad.

- **Planificación:** requisitos de seguridad y modelado de amenazas antes de escribir código.
- **Codificación:** hooks de pre-commit, detección de secretos y linters de seguridad en el editor.
- **Build:** análisis estático del código (SAST) y análisis de dependencias (SCA).
- **Pruebas:** análisis dinámico (DAST) contra una instancia en ejecución de la aplicación.
- **Despliegue:** análisis de la infraestructura como código y de las imágenes de contenedor, firma de artefactos.
- **Operación:** monitoreo, registro de eventos de seguridad y respuesta a incidentes.

Modelado de amenazas

El modelado de amenazas es la actividad de diseño que pregunta qué puede salir mal antes de construir.

El modelo STRIDE de Microsoft clasifica las amenazas en seis categorías.

- **Spoofing:** suplantar la identidad de otro usuario o servicio.
- **Tampering:** modificar datos o código sin autorización.
- **Repudiation:** negar haber realizado una acción porque no quedó registro.
- **Information disclosure:** exponer información a quien no debe verla.
- **Denial of service:** dejar el sistema sin capacidad de atender.
- **Elevation of privilege:** obtener permisos que no corresponden.

Es la única práctica de la lista que encuentra defectos de diseño, porque ninguna herramienta automática sabe lo que el sistema debería hacer.

Tipos de análisis automatizado

Técnica	Qué analiza	Herramientas
Detección de secretos	Credenciales en el código y en el historial	Gitleaks, TruffleHog, GitHub secret scanning
SAST	El código fuente, sin ejecutarlo	Semgrep, SonarQube, CodeQL
SCA	Las dependencias de terceros	npm audit, Dependabot, OWASP Dependency-Check
DAST	La aplicación en ejecución, desde afuera	OWASP ZAP, Burp Suite
IaC	Archivos de Terraform, Kubernetes y Docker	Checkov, tfsec
Contenedores	Las imágenes y sus paquetes del sistema	Trivy, Gype

SAST y DAST

Las dos técnicas detectan clases distintas de defectos y se ejecutan en etapas distintas del pipeline.

- **SAST** analiza el código fuente sin ejecutarlo. Corre en el build, reporta archivo y línea y no necesita la aplicación desplegada. Tiene una tasa mayor de falsos positivos y no detecta errores de configuración del entorno.
- **DAST** envía peticiones maliciosas a la aplicación desplegada en un entorno de pruebas. Detecta errores de configuración y de comportamiento en ejecución, pero requiere un entorno desplegado, es más lento y no reporta la línea de código responsable.

Caso Equifax, 2017

El 7 de marzo de 2017 Apache publicó un parche para una vulnerabilidad crítica en el framework Apache Struts (CVE-2017-5638). Equifax tenía un portal de disputas que usaba la versión vulnerable.

El parche nunca se aplicó en ese portal. Desde mediados de mayo los atacantes tuvieron acceso a la información, y la brecha se descubrió hasta finales de julio. Se expusieron datos personales de cerca de 147 millones de personas.

La organización no tenía un inventario que relacionara cada aplicación con las versiones de sus dependencias.

Prácticas que lo habrían evitado. Un inventario de dependencias por aplicación, análisis de composición de software (SCA) en cada build y una política de actualización con plazos medidos.

Caso SolarWinds, 2020

Los atacantes comprometieron el sistema de build de SolarWinds e inyectaron un malware, llamado Sunburst, en las actualizaciones de su producto Orion.

Las actualizaciones salieron firmadas digitalmente por SolarWinds, así que los clientes no tenían forma de distinguirlas de una actualización legítima. Cerca de 18,000 organizaciones instalaron la versión comprometida, entre ellas agencias del gobierno de Estados Unidos.

El malware no estaba en el repositorio. Se inyectaba durante la compilación, así que la revisión de código no podía detectarlo.

Prácticas que lo habrían mitigado. Proteger el pipeline como un sistema de producción, builds reproducibles que permitan comparar el artefacto con el código, y registros de procedencia como los que propone el marco SLSA.

Caso Codecov, 2021

Codecov es un servicio de cobertura de pruebas que miles de equipos invocaban desde sus pipelines de CI con un script de Bash descargado en cada corrida.

Un atacante modificó ese script en el servidor de Codecov. Durante dos meses, el script alterado envió a un servidor externo las variables de entorno de los pipelines que lo ejecutaban, incluidas credenciales y tokens.

Las variables de entorno del pipeline contenían credenciales de despliegue, tokens de repositorios y claves de servicios en la nube.

Prácticas que lo habrían mitigado. Verificar la integridad de todo script descargado, fijar versiones de las acciones y herramientas del pipeline, y dar a cada pipeline solo los secretos mínimos que necesita.

Caso Log4Shell, 2021

En diciembre de 2021 se publicó una vulnerabilidad en Log4j, la biblioteca de registro más usada en Java (CVE-2021-44228). Recibió la severidad máxima de 10 en CVSS, porque una sola cadena de texto registrada en un log permitía ejecutar código remoto en el servidor.

Log4j estaba presente en millones de sistemas, con frecuencia como dependencia transitiva, sin declararse en el archivo de dependencias del proyecto.

Prácticas que lo habrían mitigado. Una lista de materiales de software (SBOM) por cada artefacto, que permite responder en minutos qué sistemas contienen un componente, y análisis de dependencias transitivas en el pipeline.

Caso xz-utils, 2024

En marzo de 2024, un ingeniero de Microsoft notó que las conexiones SSH de su máquina tardaban medio segundo más de lo normal. Al investigar encontró una puerta trasera en xz-utils, una biblioteca de compresión presente en casi todas las distribuciones de Linux (CVE-2024-3094).

La puerta trasera la introdujo un colaborador que pasó cerca de dos años ganándose la confianza del proyecto hasta convertirse en mantenedor. El código malicioso no estaba en el repositorio visible, sino en los archivos de distribución empaquetados.

El ataque se detectó días antes de llegar a las versiones estables de las distribuciones principales.

Prácticas que lo habrían mitigado. Builds reproducibles que verifiquen que el paquete distribuido corresponde al código del repositorio, y monitoreo del comportamiento en ejecución.

Caso Toyota, 2022

Un contratista de Toyota publicó por error parte del código fuente de la aplicación T-Connect en un repositorio público de GitHub. El código contenía una clave de acceso a un servidor de datos.

La clave estuvo expuesta casi cinco años, de diciembre de 2017 a septiembre de 2022, con acceso a cerca de 296,000 direcciones de correo de clientes.

Prácticas que lo habrían evitado. Un hook de pre-commit que detecte secretos antes de que salgan de la máquina, secret scanning en el repositorio y credenciales guardadas en un gestor de secretos, nunca en el código.

Lecciones de los casos

Caso	Qué falló	Práctica DevSecOps
Equifax	Dependencia vulnerable sin parchar	SCA e inventario de dependencias
SolarWinds	Sistema de build comprometido	Integridad del build y procedencia
Codecov	Script del pipeline alterado	Fijar versiones y mínimo privilegio en CI
Log4Shell	No se sabía dónde estaba la biblioteca	SBOM y análisis de dependencias transitivas

Caso	Qué falló	Práctica DevSecOps
xz-utils	Mantenedor malicioso	Verificación de artefactos y monitoreo
Toyota	Clave de acceso en el código	Detección de secretos y gestor de secretos

Secretos en el código

Una credencial escrita en el código queda en el historial de Git para siempre, aunque un commit posterior la borre.

```
// Inseguro, la clave queda en el repositorio
const stripe = new Stripe('sk_live_51Hx9QfK2...');

// Seguro, la clave se lee del entorno
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

- La clave se guarda en un gestor de secretos o en los secretos del servidor de CI, y la aplicación la recibe como variable de entorno.
- Si una clave llega a publicarse, borrarla del código no basta. Se revoca y se emite una nueva, porque cualquier copia del repositorio ya la contiene.

Un hook de pre-commit con Gitleaks revisa cada commit en la máquina del desarrollador y lo rechaza si encuentra un secreto.

```
repos:
- repo: https://github.com/gitleaks/gitleaks
  rev: v8.24.2
hooks:
```

- id: gitleaks

Un pipeline DevSecOps

Se requiere un workflow de GitHub Actions que, en cada pull request dirigido al main branch, busque secretos en el historial, revise las dependencias en busca de vulnerabilidades conocidas y ejecute un análisis estático con las reglas de OWASP Top 10. Si cualquiera de los controles encuentra un problema, el pull request no se puede fusionar.

```
name: Seguridad
on:
  pull_request:
    branches: [main]
permissions:
  contents: read
jobs:
  analizar:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - uses: gitleaks/gitleaks-action@v2
        env:
          GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
      - run: npm ci
      - run: npm audit --audit-level=high
      - run: >
        pipx run semgrep scan
        --config p/owasp-top-ten --error
```

- El workflow corre en cada pull request hacia el main branch, antes del merge, cuando el cambio todavía no está en la rama principal.

- La clave `permissions` limita el token del pipeline a solo lectura. Si un paso o una acción de terceros se compromete, el token no permite escribir en el repositorio.
- `fetch-depth: 0` descarga el historial completo, porque un secreto borrado en un commit posterior sigue existiendo en los anteriores.
- Gitleaks revisa los commits del pull request en busca de claves, tokens y contraseñas. El hook de pre-commit depende de la configuración de cada máquina, y este paso aplica el mismo control en el servidor.
- `npm audit --audit-level=high` compara las dependencias instaladas con la base de vulnerabilidades conocidas y falla solo si encuentra severidad alta o crítica.
- Semgrep aplica las reglas de OWASP Top 10 al código fuente, y `--error` hace que el paso falle si hay hallazgos.
- Con una regla de protección del main branch que exige este check, el pull request no se puede fusionar mientras falle.

Gestión de hallazgos

Los hallazgos se clasifican y priorizan para definir cuáles bloquean el merge y cuáles se registran.

- **Severidad:** CVSS asigna a cada vulnerabilidad una puntuación de 0 a 10, y el umbral de bloqueo se define sobre esa escala.
- **Umbral de bloqueo:** las vulnerabilidades críticas y altas detienen el pipeline; las medias y bajas se registran como deuda.
- **Triage:** cada hallazgo se clasifica como real, falso positivo o riesgo aceptado, y la decisión queda documentada.

- **Plazos de corrección:** cada severidad tiene un plazo máximo, por ejemplo siete días para críticas y treinta para altas.
- **Excepciones con fecha:** un riesgo aceptado lleva fecha de vencimiento y responsable, no se acepta para siempre.

Cadena de suministro de software

Una aplicación moderna tiene más código de terceros que código propio, y las dependencias se ejecutan con los mismos permisos que el código propio.

- **SBOM:** la lista de materiales de software enumera todos los componentes de un artefacto con su versión, en formatos estándar como CycloneDX o SPDX.
- **Actualización automática:** Dependabot y Renovate abren pull requests con las versiones corregidas, y el pipeline las verifica antes del merge.
- **Versiones fijadas:** los archivos de bloqueo y las acciones fijadas por versión impiden que una dependencia cambie sin que nadie lo note.
- **Firma de artefactos:** herramientas como Sigstore firman el artefacto para que el destino verifique que no fue alterado.
- **SLSA:** un marco de niveles que describe qué tan protegido está el proceso que produce un artefacto.

Relación con OWASP Top 10

Cada categoría del Top 10 tiene un control automatizado que la detecta en el pipeline.

Categoría OWASP	Control en el pipeline
-----------------	------------------------

A02 Security Misconfiguration	Análisis de IaC y DAST
A03 Software Supply Chain Failures	SCA, SBOM y firma de artefactos
A04 Cryptographic Failures	Detección de secretos y reglas SAST
A05 Injection	SAST y DAST
A06 Insecure Design	Modelado de amenazas, no lo detecta ninguna herramienta

Antipatrones frecuentes

- **Seguridad como departamento:** los controles los configura y atiende un equipo externo al equipo de desarrollo.
- **Herramienta antes que proceso:** comprar un escáner sin definir quién atiende los hallazgos ni en qué plazo.
- **Análisis concentrados al final:** SAST, SCA y DAST se ejecutan solo antes del despliegue, lejos del commit que introdujo el defecto.
- **Falsos positivos sin triage:** un volumen alto de hallazgos sin clasificar lleva a desactivar o ignorar los controles.
- **Umbral mal definido:** bloquear por cualquier severidad detiene el desarrollo, y no bloquear nunca deja pasar vulnerabilidades críticas.
- **Confiar solo en herramientas:** los escáneres no encuentran defectos de diseño ni errores de lógica de negocio.

Cultura de seguridad

- **Security champions:** una persona de cada equipo de desarrollo con formación adicional en seguridad, que sirve de enlace con el equipo de seguridad.
- **Postmortems sin culpables:** el análisis de un incidente se centra en las causas técnicas y de proceso.
- **Capacitación continua:** el equipo conoce las vulnerabilidades más comunes de su stack y cómo se ven en el código.
- **Seguridad en la definición de terminado:** una historia no está terminada si el pipeline de seguridad no está en verde.

Métricas de DevSecOps

- **Tiempo medio de corrección:** cuánto pasa desde que se detecta una vulnerabilidad hasta que se corrige, por severidad.
- **Vulnerabilidades abiertas:** cuántas hay en cada severidad y cómo evolucionan en el tiempo.
- **Vulnerabilidades escapadas:** cuántas se encontraron en producción en lugar de en el pipeline.
- **Cobertura de análisis:** qué porcentaje de los repositorios tiene los controles activos.
- **Tasa de falsos positivos:** qué porcentaje de los hallazgos resulta no ser un problema real.

Ejercicios

Ejercicio 1. Agregar a un repositorio propio el workflow de seguridad de la lección, activar la regla de protección del main branch que exige el pipeline en verde, y abrir un pull request que introduzca una clave falsa para comprobar que

el pipeline lo rechaza.

Ejercicio 2. Elaborar un modelo de amenazas STRIDE para el proyecto del curso. Identificar al menos una amenaza por categoría, el componente afectado y el control que la mitiga, e indicar cuáles de esos controles se pueden automatizar en el pipeline.