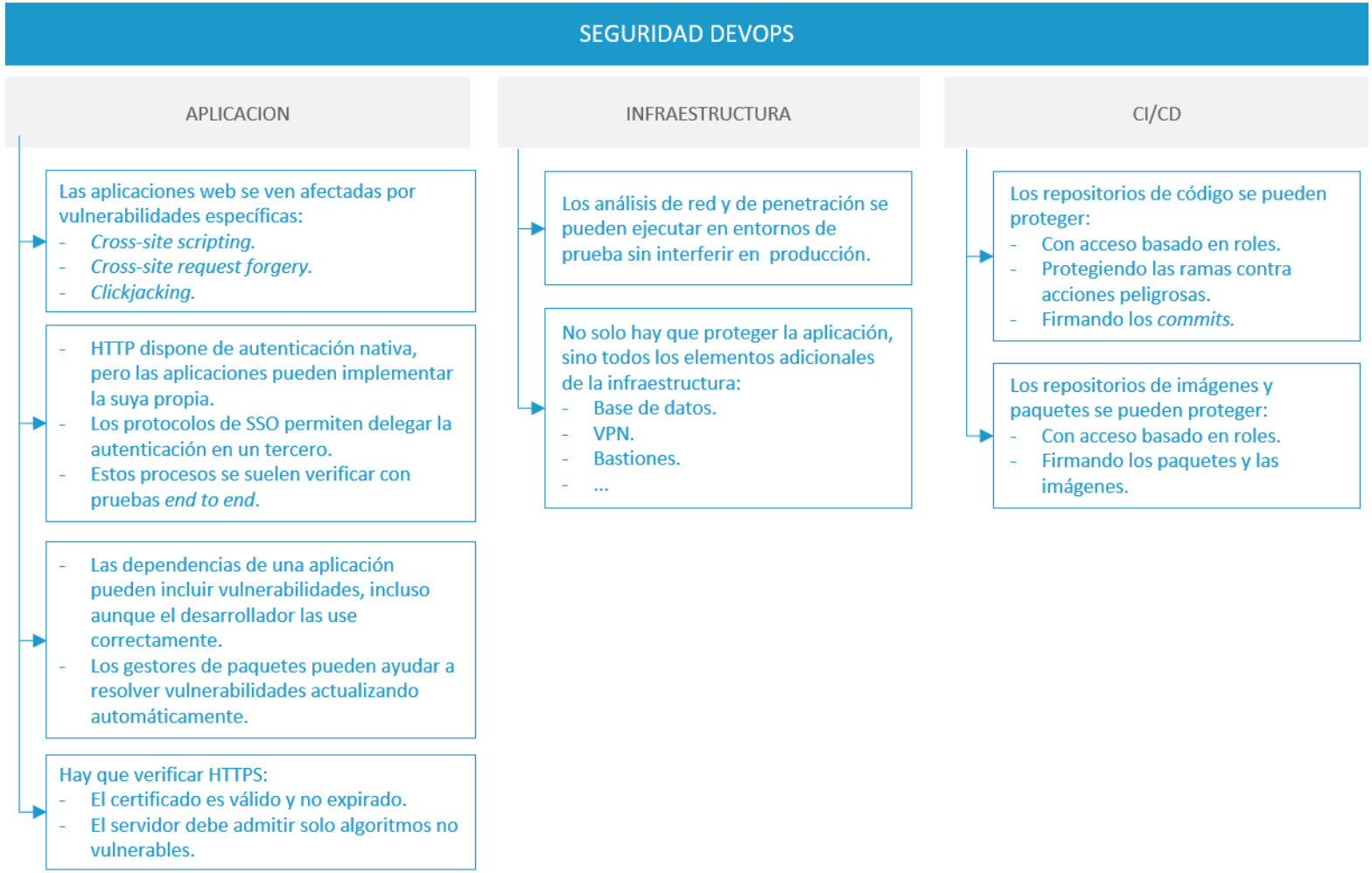
SecDevOps y Administración de Redes para Cloud

Seguridad DevOps

Índice

Esquema



Ideas clave

9.1. Introducción y objetivos

La **protección de una aplicación** abarca muchos elementos, y, si la organización se ha unido al tren de DevOps, es más fácil que nunca **automatizar** esta protección desde que el desarrollador escribe el **código** hasta que este llega a **producción.**

Los **objetivos** de este tema son:

- Identificar los **elementos vulnerables** en una cadena de integración y despliegue continuo.
- Conocer algunas **técnicas** para proteger estos elementos.

9.2. Seguridad en un *pipeline* DevOps

A lo largo de este tema se tomará como referencia un ***pipeline*** **DevOps** que desplegará la aplicación como **contenedores en un proveedor de nube.**

El objetivo de esta asignatura no es explicar este concepto en detalle y, además, las **recomendaciones de seguridad** pueden aplicarse a **cualquier escenario** de integración y entrega continua. Por tanto, aunque habrá referencias a tecnologías concretas, el aspecto importante es resaltar los puntos sensibles de seguridad del *pipeline,* de manera que se puedan generalizar a cualquier implementación.

Tampoco se entrará en los detalles de la aplicación, aunque se asumirá que ofrece una a sus usuarios, ya que es una situación muy común en aplicaciones desplegadas en la nube.

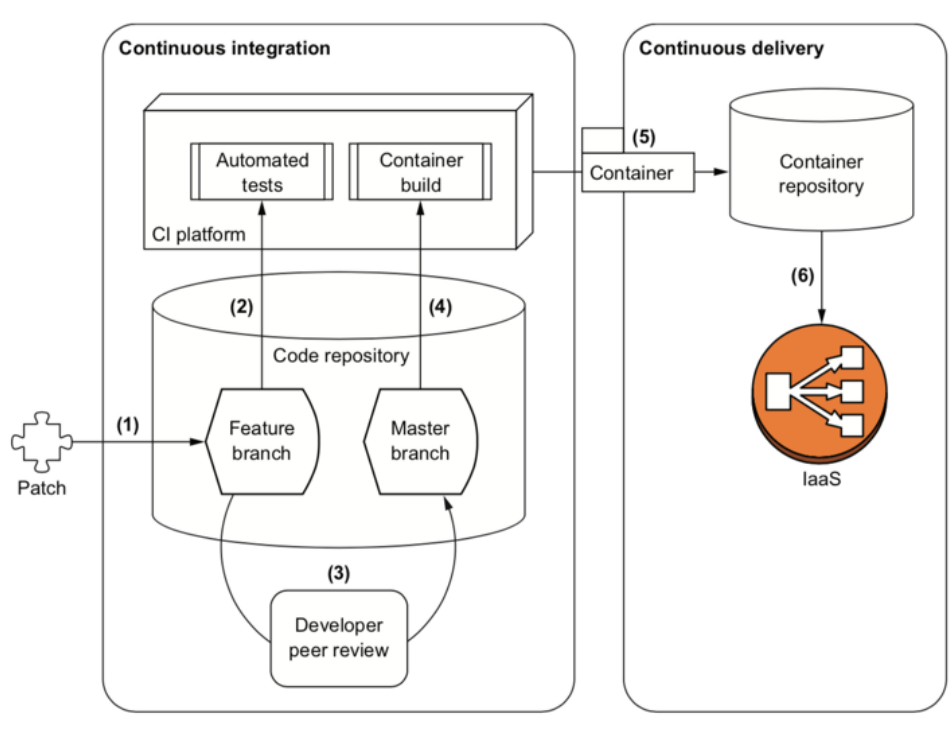


Figura 1. Fases del *pipeline* del CI/CD. Fuente: Vehent, 2018.

La Figura 1 muestra los **pasos** que atravesará el código desde que un desarrollador escribe una **nueva funcionalidad** hasta que aparece en producción (Vehent, 2018):

- **Paso 1:** el desarrollador escribe una nueva funcionalidad y envía el código al repositorio principal, bien como una *pull request,* como un parche o en una rama específica.

- **Paso 2:** se ejecuta la batería de pruebas automáticas. En ciertos casos, este paso puede implicar la creación de un entorno de aplicación completo.

- **Paso 3:** el parche es revisado por otros desarrolladores y, en caso de ser aprobado, se incorpora a la rama principal.

- **Paso 4:** a partir del código de la rama principal, con el nuevo código se construye una nueva imagen de la aplicación. Este paso varía entre tipos de despliegues: puede significar solamente la compilación de un binario, la generación de un paquete de instalación de `yum` o el archivado de ficheros estáticos en un zip.

- **Paso 5:** la imagen se envía al repositorio. De nuevo, este paso puede implicar el envío de un paquete al repositorio `yum` o la transferencia del fichero zip a un *bucket* de S3.

- **Paso 6:** se usa la última versión de la imagen para desplegar de nuevo los contenedores de la aplicación en el entorno de producción.

Una vez vistos los **pasos generales,** los siguientes apartados analizan elementos específicos que pueden ser **vulnerables** y es necesario **proteger.**

9.3. Pruebas de aplicación

El paso dos está dedicado a las **pruebas de la aplicación.** Pueden contener, únicamente, pruebas unitarias o de estilo, en las que el código se analiza de manera **estática,** aunque es habitual que se ejecuten también pruebas *end-to-end* o de integración.

En estos casos, especialmente si la aplicación es un servidor, es necesario arrancar este con el código de la **nueva funcionalidad,** preferiblemente en un entorno nuevo que se pueda **desmantelar** al terminar las pruebas.

Vulnerabilidades web

En este tipo de *pipelines* las pruebas no tienen por qué restringirse a las **funcionalidades de la aplicación.** Hay utilidades como OWASP Zed Attack Proxy, o ZAP, que permiten automatizar comprobaciones contra vulnerabilidades típicas de aplicaciones web. Algunos ejemplos se pueden ver en la Tabla 1.

Ataques de *cross-site scripting* o de scripting cruzado	Un atacante consigue introducir código JavaScript en el contenido de una página. Eso se puede conseguir incluso sin modificar el código fuente simplemente introduciendo el código en, por ejemplo, el campo de descripción del perfil de un usuario en una red social. Si ese campo no se valida, el navegador de un usuario que visualice la información del perfil del atacante ejecutará el código contenido en ese campo.
Ataques de *cross-site request forgery* o de peticiones maliciosas cruzadas	Una página maliciosa incluye enlaces que redirigen al usuario a algún punto de la aplicación en la que realizará una acción que el usuario no intentaba hacer, como el borrado de un recurso. Esto se consigue sin necesidad de modificar el código de la página original ni introducir datos manipulados en ningún formulario, simplemente confiando en las *cookies* del dominio de la aplicación, si el usuario ya ha iniciado sesión.
***Clickjacking* o captura de clics**	Un atacante inserta parte de la aplicación en un marco, o *iframe,* de una web maliciosa, engañando al usuario para que haga clic en un botón de la aplicación que realmente no es visible.

Tabla 1. Vulnerabilidades típicas de aplicaciones web. Fuente: elaboración propia.

ZAP es capaz de **automatizar comprobaciones** contra este tipo de vulnerabilidades antes de que el código llegue a producción. Esto no significa que la aplicación no deba escanearse en producción, pero con este tipo de pruebas se puede **reducir** el número de **parches** que hay que añadir a raíz de un análisis o una auditoría.

Autenticación

HTTP **soporta,** nativamente, **autenticación** básica con la cabecera `Authentication`, pero las aplicaciones pueden implementar sus propios **mecanismos** mediante *cookies* o cabeceras personalizadas. En estos casos, las pruebas deben cubrir elementos como el **cifrado no reversible** de contraseñas de usuario.

Otra opción es usar **protocolos de identidad** para delegar la autenticación en una tercera parte, como SAML o OpenID Connect. Estos protocolos, conocidos genéricamente como SSO, de *single sign-on*, permiten que una aplicación **confíe** en esta tercera parte para **identificar al usuario** sin alojar sus credenciales.

Las pruebas de autenticación no son sencillas, especialmente si se usa un **mecanismo propio** o si se implementa un mecanismo de SSO. En el primer caso, las pruebas unitarias deberán incluir **comprobaciones específicas.** En el segundo, las pruebas **end-to-end** con simulaciones de interacción en un navegador, como Selenium o Taiko.

En *frameworks,* para pruebas de navegador como los citados, el desarrollador **especifica los pasos,** por los que tiene que atravesar la aplicación y el contenido que tiene que introducir el usuario. Por ejemplo, en Taiko, una prueba puede empezar así:

```
await openBrowser();
await goto("myapp.com");
await write("something", into(textBox({placeholder: "Username"})))
await click(button(near("Login")))
await text('You are logged in').exists()
```

Estas pruebas **modelan** el comportamiento de la aplicación, sin necesidad de conocer cómo funciona esta internamente. Podría servir para **comprobar** si una sesión caduca tras el tiempo indicado o si un recurso privado es accesible sin autenticación.

Dependencias

Es habitual que las **aplicaciones modernas** deleguen funcionalidades en **librerías externas** (por ejemplo, del repositorio de npmjs.com para NodeJS o pypi.org para python). Esto acelera el **tiempo de desarrollo** ya que permite que un desarrollador sin conocimientos avanzados en un campo concreto pueda hacer **uso de código** que expone una funcionalidad completa con una interfaz sencilla. Por ejemplo, nadie en

la industria intentará escribir de cero un algoritmo de *hash* MD5, ya que la **posibilidad de error** es muy alta y las librerías disponibles están de sobra probadas.

Sin embargo, estas librerías pueden introducir **vulnerabilidades propias** sin que el desarrollador sea consciente de ello. Los **instaladores de dependencias** pueden automatizar la actualización de dependencias en cada ejecución del *pipeline,* pero esto puede provocar otros **errores** si la librería incorpora cambios no compatibles hacia atrás.

Pueden aparecer problemas incluso cuando las librerías siguen las directrices de *semantic versioning*, así que los desarrolladores tienden a **fijar las versiones** para actualizarlas solo proactivamente.

Esto lleva a que las aplicaciones no reciban los **parches de seguridad** que las librerías incorporan en nuevas versiones. Una de las pruebas que se puede incorporar en el *pipeline* es verificar si las versiones en uso sufren de **vulnerabilidades conocidas.** Por ejemplo, NodeJS incluye la *Node Security Platform* en su ecosistema. Mediante `npm audit` es posible obtener la lista de vulnerabilidades de las versiones en uso. También hay **analizadores** similares para python, como pyup.io o requires.io.

HTTPS

Las aplicaciones que sirven **sitios web** o **interfaces API REST** deben proteger la comunicación HTTP. El protocolo SSL y su sucesor, TLS, añaden **confidencialidad, autenticación** e **integridad** a HTTP. Habilitar HTTPS no asegura que el tráfico sea seguro por sí solo. Las pruebas deberán evaluar aspectos como:

- La aplicación solo es accesible por HTTPS, no por HTTP. Como mucho, cualquier petición por HTTP deberá redirigir a la misma ruta con prefijo `https://`.
- Los certificados se han generado con la longitud de clave suficiente, no han caducado y no han sido revocados.

- El servidor solo soportará las *suites* de cifrado más modernas o, al menos, no soportará *suites* con vulnerabilidades conocidas. La **adopción de algoritmos** modernos no tiene por qué ser rápida en todos los clientes, así que las organizaciones deben encontrar un equilibrio entre los clientes que soportan y los algoritmos que quieren dejar de soportar.

Por ejemplo, `testssl.sh` es una herramienta de **línea de comandos** que genera informes sobre los certificados y las *suites* de cifrado ofrecidas por el servidor.

9.4. Protección de la infraestructura

Ya se trate de una **aplicación con una interfaz HTTPS** o un **servidor interno** que analice un *data lake* periódicamente, las instancias en las que se ejecutan los procesos deben estar **protegidas** a nivel de red. En un entorno de nube, además, hay que tener en cuenta no solo las instancias de cómputo, sino otros **objetos nativos** del proveedor, como balanceadores de carga y redes virtuales.

Pruebas de red

Este tipo de **despliegues** son **ideales** para las **pruebas de esta fase.** Se puede desplegar un entorno que simule lo más posible el de producción, aunque a menor escala, y ejecutar pruebas de acceso sobre él. En este caso, no se trata de comprobar **vulnerabilidades** a nivel de aplicación, sino de verificar que las **reglas de *firewall*** (o el objeto equivalente, como un grupo de seguridad) está configurado correctamente.

Como estos entornos son privados, se pueden llevar a cabo **escaneos de puertos** sin afectar el funcionamiento del entorno de producción.

Bases de datos y almacenamiento

Las **aplicaciones** harán uso de algún tipo de **persistencia,** ya sea una base de datos desplegada *ad hoc,* una base de datos como servicio (por ejemplo, RDS en Amazon o MongoDB Atlas) o un almacenamiento de bloque o de objeto como Amazon S3 o Backblaze B2. Estos **elementos** deben ser también **protegidos,** por lo que las pruebas comprobarán que no se permite acceso **sin credenciales**, o que los puertos de la base de datos solo están abiertos a los **servidores de la aplicación.**

Otros elementos de infraestructura

Además de la **red,** hay otros elementos de la infraestructura que no están relacionados directamente con el desarrollo de la aplicación. Por ejemplo, puede haber **servidores bastión** para el acceso de operadores o puntos de entrada por **VPN.**

Es posible que cada **entorno de pruebas** no se despliegue con toda esta infraestructura; en estos casos, será en el **entorno de producción** donde se comprueben que las configuraciones son correctas: los **equipos bastión** solo soportan acceso con **clave pública** (es decir, no con contraseña), el servidor VPN acepta solo clientes modernos con algoritmos sin vulnerabilidades conocidas, etc.

9.5. Protección de las herramientas de CI/CD

Si un **atacante** consigue acceder a una de las **herramientas del *pipeline* de CI/CD,** cualquier otra medida que se haya implementado no servirá de nada. Da igual que todos los elementos de la aplicación estén protegidos y las pruebas finalicen satisfactoriamente: quien tenga **acceso completo** al *pipeline* puede modificar a su gusto cualquier elemento de la aplicación o de la infraestructura.

Por ejemplo, podría **desactivar** aquellas que detecten una **vulnerabilidad** que el atacante quiere activar en la aplicación, o podría cambiar una imagen del repositorio, que ya debería haber sido validada, para ejecutar **código arbitrario.**

La Figura 2 muestra algunos de los puntos en lo que habrá que prestar atención, por ejemplo:

- En todas las herramientas habrá que configurar el acceso basado en roles, de manera que cada usuario reciba el mínimo conjunto de permisos necesarios.
- Los *commits* deben estar firmados para evitar que se incorporen a la rama principal si son de un individuo ajeno a la organización o, en caso de que se lleguen a aceptar, tener la posibilidad de auditar su origen.

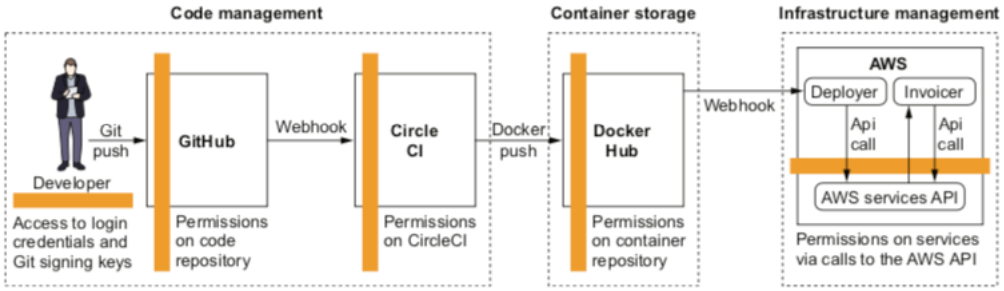


Figura 2. Niveles de protección a lo largo de un *pipeline* de CI/CD. Fuente: Vehent, 2018.

Repositorios de código

Algunas de las **buenas prácticas de control de acceso,** no solo en aplicaciones de control de versiones, sino en cualquier sistema, son las siguientes:

- Mantener la **lista de usuarios** con acceso ilimitado al repositorio lo más pequeña posible. Siempre es necesario que haya uno o varios administradores principales, pero no se debe caer en la tentación de dar acceso indiscriminado a los usuarios solo para facilitar la asignación de permisos.

- Requerir **autenticación multifactor** (MFA). Cada vez más servicios de control de versiones soportan autenticación de doble factor, lo que ofrece una capa de seguridad adicional. Además de las credenciales, el usuario deberá introducir un código que puede obtener, por ejemplo, a través de un SMS, de un correo electrónico, de una aplicación de móvil o de un *token* físico, que es como se suelen denominar unos dispositivos específicos que muestran un código por pantalla.

- **Auditar** regularmente los miembros de los grupos. Por ejemplo, se pueden comprobar los usuarios que pertenecer a un grupo de desarrolladores a través de la API de GitHub y compararlos con los usuarios de un Directorio Activo local. Así se evitan discrepancias entre un entorno y otro.

Firma de *commits*

Como se ha mencionado, los *commits* que llegan a la **rama principal** de una aplicación pueden contener código fraudulento si no se establecen los **controles necesarios.** Si el acceso a un repositorio se ve comprometido, un atacante podría **inyectar código fuente fraudulento** en la aplicación sin que los desarrolladores lo noten.

Sistemas como GitHub ofrecen características de seguridad, como la protección de ramas para evitar operaciones sensibles, como añadir *commits* directamente a la rama máster. Si el acceso basado en roles está bien configurado, este tipo de restricciones proporcionan una buena capa de seguridad.

Pero un atacante sería capaz de **deshabilitar** estas protecciones si consigue **acceso al repositorio.** La firma de *commits* con *git* proporciona una capa adicional. *Git* soporta la firma de *commits* y etiquetas con PGP. Esta funcionalidad consiste en **aplicar firmas criptográficas** a cada parche, *commit* y etiqueta, utilizando claves que los desarrolladores mantienen en secreto.

Los algoritmos que se usan no son diferentes a los que se usan en HTTPS, aunque las herramientas no son las mismas. Si los *commits* se firman con una clave válida dentro de la organización, se pueden considerar confiables.

En el siguiente vídeo, titulado **«Firma de cambios con git»,** podrás ver una explicación paso a paso de cómo firmar *commits* y cómo se presentan estos en GitHub.

Infraestructura y contenedores

Las herramientas de **integración continua,** como Jenkins o CircleCI, así como el acceso a los proveedores de nube, deben estar protegidas con un correcto **acceso basado en roles.** Por ejemplo, el acceso a AWS debe estar controlado con políticas de IAM.

IAM permite, además, **distribuir credenciales de AWS** a aplicaciones que se ejecuten en su infraestructura con los roles de IAM.

La distribución de secretos es un tema sensible y nada fácil de solucionar, por lo que hay que aprovechar estas funcionalidades allá donde estén. Kubernetes también permite **guardar secretos** y **entregarlos** a los *pods* de manera segura, y Jenkins permite **alojar secretos internamente** y **exponerlos** a los trabajos, solo donde sea expresamente necesario.

Para finalizar, los **paquetes de la aplicación** pueden protegerse frente a alteraciones, al igual que los *commits.* Por ejemplo, si la aplicación se despliega a base de contenedores, se puede firmar el tráfico enviado a un repositorio de Docker mediante Docker Content Trust.

9.6. Referencias bibliográficas

Vehent, J. (2018). *Securing DevOps.* Manning Publications.

A fondo

Transformación DevOps

Hering, M. (2018). *DevOps for the Modern Enterprise.* IT Revolution.

Este libro habla de la transformación a DevOps. El capítulo 7 trata concretamente de pruebas y menciona algunos de los aspectos de seguridad tratados. Se recomienda para dar contexto sobre cómo afrontan estas transformaciones.

Este pequeño artículo repasa algunas ideas que sirven para reflexionar sobre cómo afrontar un despliegue o una migración de redes a la nube. Es específico de AWS, pero las ideas que plantea aplican a cualquier proveedor.

Herramientas de seguridad de CI/DC

DevOpsTV. (2019, septiembre 18/). *Security in CI CD Pipelines: Tips for DevOps Engineers* [Vídeo]. YouTube. https://www.youtube.com/watch?v=S7TfXEyhLck

Este vídeo explica algunas de las herramientas que facilitan la integración de pruebas de seguridad en un *pipeline* de CI/CD. Es más teórico que práctico, pero se pueden conseguir buenas ideas para aplicar en entornos reales.

Test

1. ¿Cuáles de los siguientes elementos hay que proteger en un *pipeline* DevOps?

A. La herramienta de CI/CD.

B. La infraestructura.

C. La aplicación.

D. Todos los anteriores.

2. ¿Cuándo se puede analizar una aplicación en búsqueda de vulnerabilidades?

A. En cada nuevo parche.

B. En producción, periódicamente.

C. Todos los anteriores.

D. Manualmente, siguiendo una agenda prefijada.

3. ¿Qué ventajas tiene incorporar pruebas de seguridad en el *pipeline* DevOps?

A. Se pueden detectar pruebas y vulnerabilidades antes incluso de publicar el parche en producción.

B. Se ejecutan de manera automática, por lo que el funcionamiento es consistente.

C. Se ejecutan de manera automática, por lo que no hay peligro de que, por un error humano, no se ejecuten.

D. Todas las anteriores.

4. ¿Cómo se puede proteger un sistema de control de versiones?

A. Aplicando un correcto control de acceso basado en roles.

B. Firmando los *commits.*

C. Protegiendo las ramas para evitar operaciones peligrosas.

D. Todas las anteriores.

5. ¿Por qué son necesarias las pruebas de seguridad *end-to-end*?

A. Porque las pruebas estáticas no son capaces de descubrir ninguna vulnerabilidad.

B. Porque hay vulnerabilidades que solo se pueden detectar si se prueba el flujo completo, por ejemplo, la autenticación.

C. Normalmente no hacen falta y es suficiente con pruebas unitarias.

D. Porque así se puede probar la aplicación en un navegador.

6. ¿Qué puede detectar una prueba de certificados HTTPS?

A. Si el certificado ha expirado.

B. Si el certificado ha sido revocado.

C. Si el certificado ha sido expedido por una CA de confianza.

D. Todas las anteriores.

7. ¿En qué casos pueden aparecer vulnerabilidades en las dependencias de la aplicación, incluso si no se ha cambiado el código propio?

A. Si el gestor de paquetes decide cambiar la versión por una más antigua.

B. Si el gestor de paquetes actualiza la versión a una más nueva que introduce una vulnerabilidad inesperada.

C. Si se descubre una vulnerabilidad en la versión que usa la aplicación y no se actualiza a una versión que la soluciona.

D. B y C son correctas.

8. ¿Cuál es el gestor de dependencias típico de NodeJS?

A. npm.

B. pip.

C. maven.

D. yum.