TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

SEGURIDAD EN SOFTWARE

USO DE BUENAS PRÁCTICAS

USO DE BUENAS PRÁCTICAS

El uso de buenas prácticas en el desarrollo y gestión de software es esencial para garantizar la calidad, seguridad y eficiencia de los sistemas (Omaña, 2012). Estas prácticas, fundamentadas en estándares y experiencias comprobadas, permiten minimizar errores, reducir vulnerabilidades y facilitar el mantenimiento del código. Al aplicarlas de manera constante, los profesionales aseguran procesos más organizados y productos finales confiables, fortaleciendo así la confianza de los usuarios y la integridad del software.

Principios de codificación segura

Los principios de codificación segura son un conjunto de prácticas y normas diseñadas para que los desarrolladores escriban código que minimice las vulnerabilidades y proteja las aplicaciones contra ataques maliciosos. Estos principios deben integrarse desde las etapas iniciales del desarrollo para asegurar que el software sea confiable, resistente y cumpla con los estándares de seguridad requeridos (Omaña, 2012).

Uno de los principios más importantes es la validación y saneamiento de entradas. Se refiere a que cualquier dato recibido desde fuentes externas (usuarios, APIs, bases de datos) debe ser cuidadosamente filtrado y validado antes de ser procesado (Giménez Albacete, 2023). Por ejemplo, una aplicación web que recibe datos de formularios debe eliminar caracteres peligrosos o patrones sospechosos para evitar ataques como inyección SQL o Cross-Site Scripting (XSS). Si un desarrollador no valida adecuadamente las entradas, un atacante podría insertar código malicioso que comprometa la base de datos o el navegador del usuario.

Otro principio fundamental es el principio de menor privilegio, que implica que cada componente del sistema, ya sea un usuario, proceso o módulo, debe operar con los mínimos permisos necesarios para realizar su función. Esto reduce el impacto potencial de un ataque, ya que el atacante no podrá escalar privilegios fácilmente (Giménez Albacete, 2023). Por ejemplo, una cuenta de base de datos que sólo necesita leer información no debe tener permisos para modificar o eliminar datos.

La gestión segura de errores y excepciones también es crucial. El software debe manejar los errores sin revelar información sensible como rutas internas, configuraciones o detalles técnicos que puedan ser útiles para un atacante. En lugar de mostrar mensajes de error detallados al usuario final, el sistema debe registrar la información en logs accesibles sólo para los administradores.

El uso de cifrado fuerte para datos sensibles es otro pilar en la codificación segura (Omaña, 2012). Por ejemplo, las contraseñas deben almacenarse utilizando algoritmos de hashing seguros, como bcrypt o Argon2, en lugar de texto plano, lo que previene que sean expuestas fácilmente en caso de una filtración. Además, la transmisión de datos debe realizarse mediante protocolos seguros como HTTPS para proteger la confidencialidad e integridad en tránsito.

Adicionalmente, el código debe evitar la exposición innecesaria de funcionalidades o datos. Esto implica restringir APIs, proteger rutas sensibles y asegurar que sólo los usuarios autorizados tengan acceso a información crítica, siguiendo políticas estrictas de autenticación y autorización.

Como ejemplo, en un sistema de gestión de usuarios, la codificación segura garantiza que solo el administrador pueda modificar roles, que las contraseñas nunca se muestren o almacenen en texto plano, y que todos los datos ingresados por el usuario sean validados para evitar ataques.

En resumen, seguir estos principios no solo previene vulnerabilidades comunes, sino que también mejora la estabilidad y mantenibilidad del software. La codificación segura es un compromiso constante que requiere actualización frente a nuevas amenazas y debe estar integrada en todas las fases del ciclo de desarrollo para crear aplicaciones robustas y confiables.

Gestión de contraseñas y autenticación segura

La seguridad de los sistemas depende en gran medida de cómo se protegen los mecanismos de acceso. En este contexto, la gestión de contraseñas y la autenticación segura son elementos esenciales para evitar accesos no autorizados y proteger la integridad de la información (Omaña, 2012).

1. Almacenamiento seguro de contraseñas

Una de las primeras buenas prácticas en el desarrollo de software consiste en evitar almacenar contraseñas en texto plano. En su lugar, se utilizan algoritmos de hash criptográficos como bcrypt, scrypt o Argon2, que transforman la contraseña en una cadena irreconocible y no reversible.

- Por ejemplo, un sistema que usa bcrypt en un entorno Node.js cifra la contraseña antes de guardarla, de modo que incluso si la base de datos es comprometida, los atacantes no puedan recuperar las contraseñas originales.

2. Políticas de contraseñas robustas

La fuerza de una contraseña depende de su complejidad. Las buenas prácticas exigen contraseñas largas (al menos 12 caracteres) que combinen letras mayúsculas, minúsculas, números y caracteres especiales (Giménez Albacete, 2023).

Por ejemplo, una contraseña como S3guridad!2025# es mucho más segura que admin123. Asimismo, se recomienda forzar cambios periódicos de contraseñas y evitar el uso de credenciales previamente comprometidas.

3. Autenticación multifactor (MFA)

La autenticación multifactor añade una capa de seguridad adicional, solicitando al usuario dos o más factores para verificar su identidad (Giménez Albacete, 2023). Estos factores pueden incluir:

- Algo que sabe (contraseña)
- Algo que tiene (código enviado al móvil o token de autenticación)
- Algo que es (huella dactilar o reconocimiento facial)

Por ejemplo, al acceder a un sistema bancario, el usuario ingresa su contraseña y luego un código temporal enviado a su celular. Aunque la contraseña se filtre, el atacante no podrá ingresar sin el segundo factor.

4. Control de sesiones y bloqueo por intentos fallidos

Una buena práctica es limitar la cantidad de intentos de inicio de sesión permitidos. Después de varios intentos fallidos, el sistema puede:

- Bloquear temporalmente la cuenta
- Solicitar un segundo factor de autenticación
- Notificar al usuario sobre la actividad sospechosa

Por ejemplo, tras cinco intentos fallidos, un sistema puede bloquear la cuenta durante 30 minutos y enviar un correo alertando al usuario.

5. Autenticación federada y Single Sign-On (SSO)

Muchas aplicaciones modernas permiten autenticarse usando cuentas de proveedores externos como Google, Microsoft o Facebook mediante protocolos como OAuth2 y OpenID Connect. Esta autenticación federada reduce el número de contraseñas que los usuarios deben recordar y aprovecha infraestructuras de autenticación ya consolidadas (Giménez Albacete, 2023).

Por ejemplo, una plataforma educativa puede permitir el inicio de sesión con la cuenta institucional de Google del estudiante, evitando la necesidad de crear nuevas credenciales.

6. Notificaciones y monitoreo de accesos

El sistema debe ser capaz de registrar los intentos de acceso y notificar al usuario en caso de actividad sospechosa. Asimismo, se recomienda implementar funciones como:

- Verificación de dispositivos nuevos
- Registro de ubicación y dirección IP
- Cierre de sesiones inactivas

Por ejemplo, si un usuario inicia sesión desde un país diferente, se le puede pedir confirmar la actividad mediante correo electrónico.

Actualización y mantenimiento continuo del software

En el ámbito de la seguridad del software, la actualización y el mantenimiento continuo constituyen una práctica esencial para preservar la integridad, disponibilidad

y confiabilidad de los sistemas a lo largo del tiempo. Este proceso no solo se enfoca en agregar nuevas funcionalidades, sino, sobre todo, en corregir errores, cerrar vulnerabilidades y adaptarse a entornos cambiantes (Giménez Albacete, 2023).

1. Importancia de mantener el software actualizado

Cada día surgen nuevas amenazas que ponen en riesgo la seguridad de las aplicaciones. Los atacantes buscan vulnerabilidades conocidas en versiones desactualizadas de bibliotecas, sistemas operativos o frameworks. Mantener el software actualizado garantiza que las brechas ya identificadas hayan sido corregidas por los fabricantes o la comunidad (Omaña, 2012).

Ejemplo: Una empresa que utiliza una versión antigua de Apache Struts sin parches expone sus aplicaciones a fallos conocidos que pueden permitir la ejecución remota de código. En cambio, mantener el sistema actualizado reduce significativamente este riesgo.

2. Tipos de mantenimiento de software

El mantenimiento puede clasificarse en varias categorías, cada una con un propósito específico:

- **Correctivo:** Soluciona errores detectados en producción (bugs, fallos de compatibilidad).
- **Adaptativo:** Ajusta el software a cambios del entorno (nuevas versiones de sistemas operativos, navegadores, APIs).
- **Perfectivo:** Mejora el rendimiento, la usabilidad o refactoriza el código sin cambiar su funcionalidad.
- **Preventivo:** Introduce mejoras que eviten futuros errores o problemas de seguridad.

Ejemplo: Un equipo detecta que una función de inicio de sesión causa errores intermitentes. El mantenimiento correctivo es reparar ese fallo, mientras que el preventivo revisa otros módulos similares para evitar que se repita.

3. Automatización de actualizaciones

Automatizar las actualizaciones mediante herramientas de integración continua (CI/CD) permite aplicar parches y nuevas versiones de manera eficiente, minimizando el error humano. Herramientas como Dependabot (para GitHub) o Renovate monitorean dependencias y generan alertas o pull requests automáticos cuando se liberan versiones más seguras (Omaña, 2012).

Ejemplo: Un repositorio que integra Dependabot puede recibir una notificación cuando una librería como lodash tiene una nueva versión que corrige una vulnerabilidad crítica, permitiendo actualizarla rápidamente.

4. Evaluación de riesgos antes de actualizar

No todas las actualizaciones deben aplicarse de forma automática. Algunas pueden provocar errores o incompatibilidades (Giménez Albacete, 2023). Por eso, es recomendable validar cada cambio en un entorno de prueba, hacer pruebas de regresión y revisar la documentación de cambios (changelog).

Ejemplo: Antes de actualizar la versión de un framework como Angular, el equipo técnico revisa si existen breaking changes y prueba la nueva versión en un entorno de staging.

5. Documentación y trazabilidad de cambios

Toda actualización o tarea de mantenimiento debe registrarse adecuadamente para mantener un historial claro de lo que se ha modificado, cuándo y por qué. Esto facilita auditorías, resolución de incidentes y cumplimiento de normativas de seguridad (Giménez Albacete, 2023).

Ejemplo: Una organización que desarrolla un software financiero mantiene un changelog estructurado y aplica etiquetas como security-patch, minor-fix o dependency-update para identificar claramente cada tipo de cambio realizado.

Control de acceso y principio de privilegios mínimos

En el contexto de la seguridad del software, el control de acceso y el principio de privilegios mínimos constituyen pilares fundamentales para restringir el uso indebido de los recursos del sistema y reducir la superficie de ataque. Ambos conceptos están estrechamente relacionados y buscan asegurar que cada usuario, proceso o componente solo pueda realizar acciones estrictamente necesarias para cumplir su función (Giménez Albacete, 2023).

1. Control de acceso: definición y propósito

El control de acceso es el conjunto de mecanismos que regulan quién puede acceder a determinados recursos dentro de un sistema y en qué condiciones (Omaña, 2012). Este control puede aplicarse a usuarios, servicios, dispositivos o incluso procesos internos, y se apoya en políticas bien definidas para autorizar o denegar acciones.

Ejemplo: En una aplicación bancaria, el cliente tiene acceso solo a sus cuentas personales, mientras que un empleado del área de soporte puede visualizar, pero no modificar, información financiera de los usuarios.

2. Modelos de control de acceso

Existen diferentes modelos para implementar el control de acceso según el nivel de granularidad requerido:

- **DAC (Control de Acceso Discrecional):** El propietario del recurso define quién puede acceder.
- **MAC (Control de Acceso Obligatorio):** El acceso se regula con etiquetas de seguridad rígidas, común en entornos gubernamentales.

- **RBAC (Control Basado en Roles):** Se asignan permisos a roles y los usuarios se asocian a esos roles.
- **ABAC (Control Basado en Atributos):** Se toman decisiones en función de atributos del usuario, recurso y entorno (ej. ubicación o tiempo).

Ejemplo: Un sistema hospitalario basado en RBAC puede tener roles como "Médico", "Enfermero" y "Administrador", cada uno con permisos diferentes sobre los expedientes clínicos.

3. Principio de privilegios mínimos

Este principio indica que cada entidad debe poseer únicamente los permisos necesarios para realizar sus tareas, y nada más. Aplicarlo limita el daño potencial si un actor es comprometido, ya que sus capacidades dentro del sistema están restringidas (Omaña, 2012).

Ejemplo: Un script automatizado encargado de respaldar una base de datos no debería tener permiso para eliminar registros ni modificar tablas, sólo para lectura y exportación.

4. Beneficios clave de aplicar estos principios

- **Reducción del riesgo interno y externo:** Minimiza los efectos de errores humanos o intrusiones.
- **Mejora en la trazabilidad:** Permite auditar quién hizo qué y cuándo.
- **Cumplimiento de normativas:** Ayuda a cumplir estándares como ISO 27001 o el Reglamento General de Protección de Datos (GDPR).

5. Prácticas recomendadas para su implementación

- Establecer roles bien definidos y documentados.
- Aplicar separación de funciones en procesos críticos (por ejemplo, quien desarrolla no debería desplegar en producción).
- Utilizar autenticación multifactor combinada con políticas de control de acceso.
- Auditar y revisar periódicamente los permisos otorgados.
- Implementar herramientas de gestión de identidades (IAM) para automatizar la asignación y revocación de accesos.

Uso de entornos seguros de desarrollo y despliegue

El uso de entornos seguros de desarrollo y despliegue es una práctica crítica en la ingeniería de software, orientada a proteger el ciclo de vida completo de una aplicación frente a vulnerabilidades, accesos no autorizados y manipulaciones maliciosas. A través de la implementación de controles, separación de ambientes y uso de herramientas especializadas, se garantiza que tanto el código fuente como

los datos y procesos asociados permanezcan íntegros, disponibles y confidenciales (Giménez Albacete, 2023).

1. Separación de entornos: desarrollo, pruebas y producción

Una práctica esencial es la separación estricta entre los entornos de desarrollo, pruebas y producción. Cada uno debe tener su configuración aislada para evitar que errores o cambios realizados en desarrollo afectan la operación del sistema en vivo.

Ejemplo: En una empresa de comercio electrónico, los desarrolladores trabajan sobre un entorno que simula la plataforma real, pero sin acceso a datos de usuarios reales. Los cambios se prueban exhaustivamente antes de ser promovidos al entorno de producción.

2. Control de acceso por entorno

El acceso a cada entorno debe ser controlado según roles y responsabilidades. Los desarrolladores no deberían tener permisos de escritura directa en producción, mientras que los operadores de infraestructura no deberían modificar el código fuente sin aprobación (Giménez Albacete, 2023).

Ejemplo: En una compañía financiera, solo los ingenieros de DevOps tienen acceso a los scripts de despliegue en producción, mientras que los desarrolladores usan ramas protegidas y pull requests para introducir cambios.

3. Seguridad en el entorno de desarrollo

Los equipos deben trabajar en estaciones seguras, con sistemas operativos actualizados, protección antivirus, control de versiones (como Git), y mecanismos de autenticación robusta (Omaña, 2012).

Ejemplo: Un equipo de desarrollo remoto utiliza entornos virtualizados con autenticación de dos factores y cifrado de disco completo, evitando exfiltraciones incluso en caso de pérdida del dispositivo.

4. Integración continua segura

La implementación de pipelines de integración y entrega continua (CI/CD) debe incluir controles de seguridad, como el escaneo automático de vulnerabilidades, análisis estático de código (SAST), revisión de dependencias y validaciones automatizadas antes del despliegue.

Ejemplo: Un sistema de CI/CD en una startup tecnológica ejecutan pruebas automatizadas y un escáner de seguridad (como SonarQube) antes de permitir que una nueva versión se publique.

5. Monitoreo y registro en entornos de despliegue

Todo el entorno de producción debe estar bajo vigilancia constante, con registros centralizados, alertas de comportamiento anómalo y auditoría de acciones administrativas. Esto permite detectar incidentes tempranos y cumplir con requisitos de cumplimiento normativo (Omaña, 2012).

Ejemplo: En una aplicación de salud, se monitorean todas las peticiones al servidor y se generan alertas si un usuario intenta acceder a más de cinco registros clínicos por minuto.

6. Uso de entornos replicables y automatizados

La infraestructura como código (IaC) permite definir y replicar entornos de forma segura y consistente. Además, evita la configuración manual, que es propensa a errores y omisiones.

Ejemplo: Un equipo de desarrollo utiliza Terraform para construir entornos en la nube que replican la configuración exacta de producción, incluyendo reglas de firewall, instancias y bases de datos, todo bajo control de versiones.