1.2 Protección de vulnerabilidades en el desarrollo seguro de software

El desarrollo de software seguro implica incorporar consideraciones de **seguridad** en cada etapa del ciclo de vida de un proyecto, con el objetivo de prevenir vulnerabilidades antes de que lleguen a producción. Abordar la seguridad **desde las primeras fases del desarrollo** (en lugar de dejarla solo para el final) reduce drásticamente los riesgos y los costos asociados a fallos de seguridad. De hecho, se estima que **corregir una vulnerabilidad en producción puede costar hasta 100 veces más** que arreglarla durante las etapas iniciales del ciclo de vida. En este capítulo exploraremos el **Ciclo de Vida de Desarrollo de Software Seguro (S-SDLC)**, describiendo sus fases, importancia e implementación práctica en proyectos.

Ciclo de vida de desarrollo de software seguro (S-SDLC)

El **Ciclo de Vida de Desarrollo de Software** (SDLC, por sus siglas en inglés) describe las etapas por las que pasa un proyecto de software, típicamente: **planificación, análisis de requisitos, diseño, implementación (codificación), pruebas, despliegue** y **mantenimiento**. En un modelo tradicional, las actividades de seguridad suelen relegarse a la fase de pruebas o incluso a posteriores, lo cual resulta problemático. Si la seguridad se aborda solo al final, es fácil que se **pasen por alto problemas** importantes o que se descubran demasiado tarde, volviendo su corrección costosa y difícil. Por el contrario, un **SDLC seguro (S-SDLC)** integra actividades de seguridad *en cada fase* del proceso de desarrollo. Este enfoque, a menudo llamado "*Security by Design*" o "*shift-left*" en seguridad, busca **identificar y eliminar vulnerabilidades lo antes posible** en vez de reaccionar a ellas al final.

Fases del S-SDLC y sus actividades de seguridad

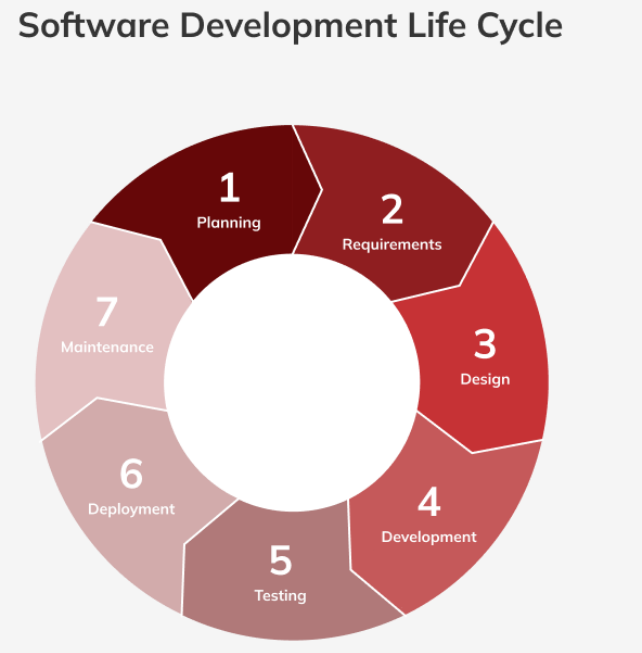

Figura 1 Fases del ciclo de vida de desarrollo de software seguro (S-SDLC)

A continuación se describen las fases típicas de un ciclo de vida de desarrollo de software, junto con las actividades de seguridad que deberían incorporarse en cada una para lograr un desarrollo seguro:

- **Planificación**: En esta etapa inicial se determina el alcance y objetivos del proyecto. Desde el punto de vista de seguridad, se deben **evaluar los riesgos y el panorama de amenazas** potenciales para el sistema. También conviene estimar el posible impacto de incidentes de seguridad (por ejemplo, pérdida de datos o daño reputacional) y planificar recursos para abordarlos. Iniciar la planificación con una mentalidad de riesgo ayuda a tomar decisiones informadas sobre qué controles aplicar y dónde priorizar esfuerzos de seguridad.
- **Análisis de requisitos**: Además de los requisitos funcionales, es fundamental **incluir requisitos de seguridad** específicos. Esto implica definir qué propiedades de seguridad debe cumplir el software (confidencialidad, integridad, disponibilidad, autenticación, autorizaciones, trazabilidad, etc.) y **requisitos de cumplimiento normativo** aplicables (por ejemplo, leyes de protección de datos o estándares de la industria). ISO/IEC 27001 enfatiza que **los requisitos de seguridad deben identificarse desde la fase de requisitos del proyecto** y acordarse con las partes involucradas, quedando documentados como parte integral del sistema. Incluir estos requisitos tempranamente asegura que la seguridad no sea una idea tardía, sino un componente esencial del diseño del sistema.
- **Diseño**: Durante el diseño de la arquitectura y componentes del software, se debe **integrar la seguridad como parte fundamental del plan de arquitectura**. Esto incluye realizar **modelado de amenazas** (identificar posibles ataques y vulnerabilidades en el diseño propuesto) y considerar cómo las decisiones de diseño impactan la seguridad. Por ejemplo, decidir la separación por capas, los mecanismos de comunicación, algoritmos criptográficos a utilizar, gestión de sesiones, etc., todo con un enfoque seguro. Un buen diseño seguro evita, por principio, las debilidades conocidas (ej.: evitar diseños que expongan datos sensibles innecesariamente o que permitan saltearse controles de autenticación). También en esta fase es útil planificar **controles de seguridad específicos**: por ejemplo, diseñar cómo se implementará la validación de datos, la gestión de identidades y accesos, la estrategia de registro de eventos de seguridad, entre otros.
- **Desarrollo (codificación)**: En la implementación del código, los desarrolladores deben seguir **prácticas de codificación segura** rigurosas. Es recomendable capacitar al equipo en tópicos de seguridad (por ejemplo, evitar vulnerabilidades OWASP Top 10). Algunas organizaciones adoptan **guías de codificación segura** y listas de verificación que los programadores deben seguir. Además, se sugiere utilizar herramientas automáticas de análisis de seguridad durante el desarrollo: por ejemplo, análisis estático de código fuente (SAST) y análisis dinámico (DAST) integrados en el proceso de integración continua. Incorporar estas herramientas en la "pipeline" de desarrollo permite detectar vulnerabilidades en el código lo antes posible. También es crucial **gestionar las dependencias de software**: muchos

proyectos utilizan librerías de terceros, las cuales deben mantenerse actualizadas y ser evaluadas para evitar dependencias con vulnerabilidades conocidas. En resumen, en esta fase se trata de *escribir código seguro desde el inicio* y verificar continuamente su seguridad.

- **Pruebas**: Además de las pruebas funcionales, se deben ejecutar **pruebas de seguridad** específicas. Esto incluye **revisiones de código** (manuales o asistidas por herramientas) enfocadas en detectar errores de seguridad, así como pruebas especializadas: análisis estático, análisis dinámico, pruebas de penetración, fuzzing, etc.. Las pruebas de penetración (pentesting) simulan ataques reales para descubrir vulnerabilidades explotables antes de que lo hagan actores maliciosos. Asimismo, se recomienda verificar que los casos de prueba cubran no solo comportamientos esperados, sino también inputs maliciosos o inesperados (por ejemplo, probar ingresar SQL o scripts en campos de entrada para comprobar que la aplicación los neutraliza). Un enfoque sistemático es utilizar estándares como el **OWASP Testing Guide** o criterios del **OWASP ASVS** (Application Security Verification Standard) para asegurar que las pruebas abarcan las principales superficies de ataque. El objetivo de esta etapa es **validar que el software cumple con los requisitos de seguridad definidos** y que no presenta vulnerabilidades evidentes.

- **Implantación (despliegue)**: Antes de liberar el software a producción, es necesario **asegurar la configuración segura del entorno de despliegue**. Esto implica, por ejemplo, verificar configuraciones del servidor web, base de datos, sistema operativo, contenedores o servicios en la nube, asegurándose de que utilizan parámetros seguros (deshabilitar servicios innecesarios, usar certificados válidos, claves y contraseñas seguras, etc.). También se deben **revisar las configuraciones de seguridad de la aplicación en producción**: archivos de configuración, variables de entorno (no exponer secretos en texto claro), políticas de *CORS*, encabezados HTTP de seguridad, *firewalls* de aplicaciones, etc. Idealmente, la infraestructura debería definirse como código (IaC) para mantener consistencia, y aplicarse principios de *hardening*. Muchos incidentes provienen de **errores de configuración o parches faltantes**, por lo que esta fase es crítica para cerrar el ciclo con un entorno lo más resistente posible.

- **Mantenimiento**: La seguridad no termina con el despliegue; durante la operación continua del software se debe **monitorear activamente el sistema para detectar amenazas o actividades anómalas**. Esto conlleva implementar sistemas de monitoreo de logs y alertas (por ejemplo, integrar un SIEM – Security Information and Event Management). Además, es esencial establecer un proceso de **gestión de vulnerabilidades** post-despliegue: cuando se descubren nuevos puntos vulnerables o surge alguna brecha, el equipo debe estar preparado para responder rápidamente, aplicar **parches o actualizaciones** y comunicar a los usuarios si es pertinente. El mantenimiento seguro también abarca la gestión de cuentas y accesos en el tiempo (revocar accesos que ya no corresponden), la realización de **auditorías periódicas** de seguridad y la preparación para eventualidades (planes de respuesta a incidentes). Mantener la aplicación segura es un esfuerzo continuo, pues las amenazas

evolucionan y el software puede requerir cambios; por ello, la filosofía del S-SDLC es **cíclica**: tras mantenimiento, el ciclo vuelve a comenzar con nueva planificación y mejoras.

En resumen, un S-SDLC efectivo significa que en **cada fase** del ciclo de vida se llevan a cabo tareas orientadas a la seguridad, logrando así un proceso robusto. Este enfoque proactivo permite **encontrar y solucionar las fallas de seguridad en etapas tempranas**, ahorrando costos de corrección y evitando retrasos o incidentes graves en producción. Integrar la seguridad de manera transversal también ayuda a crear una *cultura* en la que todos los involucrados (analistas, diseñadores, desarrolladores, testers, etc.) comparten la responsabilidad de construir un sistema seguro.

Implementación en proyectos y metodologías (DevSecOps)

Adoptar un S-SDLC en proyectos reales puede requerir ajustes en la metodología de desarrollo y en la organización del equipo. Una práctica cada vez más común es **DevSecOps**, que extiende la filosofía DevOps (integración y entrega continuas con colaboración entre desarrollo y operaciones) incorporando también la **seguridad** como componente de la cadena de valor. DevSecOps implica que **los equipos de desarrollo, operaciones y seguridad trabajen juntos** desde el inicio, automatizando controles de seguridad en la tubería CI/CD (Integración Continua/Despliegue Continuo). La automatización es clave para no ralentizar la entrega: por ejemplo, integrando *scanners* de vulnerabilidades en el pipeline, análisis de composición de dependencias, pruebas unitarias de seguridad, despliegues en entornos seguros reproducibles, etc. De este modo, la seguridad se vuelve un proceso constante y repetible, y no un bloqueo al final del proyecto.

Al implementar un S-SDLC, es útil apoyarse en **marcos y modelos existentes**. La industria y la academia han propuesto múltiples modelos de desarrollo seguro que se pueden tomar como referencia. Por ejemplo, Microsoft desarrolló su **Security Development Lifecycle (SDL)** a partir de 2004, integrando pasos de seguridad formales en cada etapa del desarrollo de software comercial (requisitos, diseño, implementación, verificación, lanzamiento y respuesta). Otros modelos incluyen el **CLASP** de OWASP, el modelo de madurez **OWASP SAMM**, metodologías de alto nivel como el SSE-CMM (Capability Maturity Model en Seguridad de Sistemas) y guías específicas para entornos ágiles. En todos los casos, el principio común es **"construir la seguridad dentro"** del proceso en lugar de tratar de "añadirla después". Un documento relevante es el marco **NIST Secure Software Development Framework (SSDF)** (SP 800-218) publicado por el Instituto Nacional de Estándares y Tecnología de EE.UU., que recomienda un conjunto de **prácticas de desarrollo seguro de alto nivel** para reducir vulnerabilidades en el software y mitigar su explotación. La idea es proporcionar un vocabulario común y un conjunto básico de prácticas (ej. asegurar el código fuente, gestionar adecuadamente las dependencias, responder a vulnerabilidades, etc.) que pueden integrarse en *cualquier* modelo de desarrollo existente. En la práctica, adoptar un S-SDLC requiere adaptar los procesos internos de la organización: establecer **políticas de seguridad de desarrollo**, capacitar al personal, elegir herramientas adecuadas y medir el progreso (por ejemplo, usando indicadores como densidad de vulnerabilidades encontradas en cada fase, tiempo de remediación, cumplimiento de estándares, etc.).

Finalmente, cabe destacar que integrar la seguridad tempranamente también tiene beneficios organizacionales: **une los objetivos de desarrollo y seguridad** bajo una misma dirección, evitando la clásica tensión donde los desarrolladores ven la seguridad como un obstáculo. Con un S-SDLC bien implementado, la seguridad se convierte en parte del flujo de trabajo *normal* del desarrollador (quien escribe código ya pensando en seguridades, con herramientas que le advierten en tiempo real de posibles fallos). La gerencia, por su parte, gana visibilidad y control sobre cómo se construyen productos seguros, facilitando cumplir requisitos regulatorios y minimizando riesgos de negocio. En esencia, la **importancia del S-SDLC** radica en que es la única manera sostenible de garantizar la protección de las aplicaciones **sin demorar ni obstaculizar** el desarrollo, sino más bien mejorándolo con calidad desde el inicio.

Técnicas y mecanismos para la protección de vulnerabilidades

Existen numerosas técnicas, buenas prácticas y mecanismos que los desarrolladores pueden aplicar para proteger sus aplicaciones contra vulnerabilidades. A continuación, se presentan algunas de las **áreas clave de protección** y sus prácticas asociadas. Estas técnicas corresponden a principios universales de desarrollo seguro, muchos de los cuales se encuentran recogidos en proyectos como el **OWASP Top 10 Proactive Controls** (controles proactivos de OWASP). Cada sub-sección resalta medidas concretas que ayudan a **prevenir las vulnerabilidades más comunes** en el desarrollo de aplicaciones web.

- **Uso de frameworks seguros y bibliotecas de confianza**: Un principio general es aprovechar las características de seguridad que brindan los frameworks y librerías en lugar de reinventar la rueda. Los frameworks web modernos suelen proveer **configuraciones seguras por defecto**; por ejemplo, muchas vistas web **escapan el contenido de salida por defecto** (defensa contra XSS) y algunas incluyen protección integrada contra ataques como CSRF. Es importante **mantener esos valores por defecto** y no deshabilitarlos sin razón. Además, utilizar librerías bien mantenidas para funcionalidades críticas (por ejemplo, para el manejo de autenticación, cifrado, validación, etc.) es preferible a escribir implementaciones caseras que podrían contener errores. Se debe verificar la reputación y actualizaciones de seguridad de las dependencias externas, y usar gestores de paquetes (npm, pip, etc.) junto con herramientas de auditoría para detectar versiones vulnerables. En resumen, **apóyese en componentes probados y seguros**: esto acelera el desarrollo y reduce la probabilidad de introducir fallos de seguridad por cuenta propia.
- **Definición de requisitos y políticas de seguridad**: Antes de comenzar a codificar, se deben haber definido claramente los **requisitos de seguridad** del sistema (tal como se mencionó en el S-SDLC). Esto guía a los desarrolladores sobre qué deben proteger y con qué nivel de rigor. Por ejemplo, si el sistema maneja datos financieros, un requisito podría ser cifrar esos datos en toda transmisión y almacenamiento. Complementario a esto es contar con **políticas de seguridad de codificación** en la organización: pautas sobre cómo manejar credenciales, normas de estilo para código seguro, listas de funciones o prácticas prohibidas (por ejemplo, evitar funciones inseguras de C, o el uso de eval en JavaScript), etc. Estas políticas sirven como

referencia cotidiana para que cada desarrollador escriba código con seguridad en mente. OWASP ofrece el proyecto **ASVS (Application Security Verification Standard)** que contiene cientos de requisitos de seguridad clasificados por nivel de rigor, los cuales pueden servir de base para establecer requisitos en un proyecto. Definir estos aspectos desde el inicio asegura un rumbo claro para las medidas técnicas específicas a implementar.

- **Validación rigurosa de entradas**: **"Nunca confíes en la entrada del usuario"** es un mantra en seguridad de software. Toda información proveniente de fuentes externas (usuarios finales, APIs de terceros, formularios, parámetros URL, cabeceras HTTP, etc.) debe considerarse potencialmente maliciosa hasta que se demuestre lo contrario. La **validación de entradas** consiste en comprobar que los datos recibidos cumplen con lo esperado antes de procesarlos o almacenarlos. Esto implica establecer reglas de formato, longitud y contenido válido para cada campo (por ejemplo, que un campo de edad sea numérico y en cierto rango, que un email cumpla la expresión regular correspondiente, etc.) y **rechazar o sanitizar** cualquier dato que no cumpla dichas reglas. Una estrategia robusta es usar un enfoque de **lista blanca** (permitir solo caracteres o patrones conocidos buenos, en lugar de solo intentar bloquear cosas malas específicas). Por ejemplo, si se espera un nombre de usuario alfanumérico de 5-15 caracteres, la aplicación debería rechazar entradas que contengan otros símbolos o longitudes fuera de rango. En muchas plataformas existen **frameworks de validación** (como class-validator en NestJS/Node, o validaciones de modelo en ORMs) que facilitan aplicar estas reglas de manera consistente. La validación debe hacerse tanto en el **cliente** (por usabilidad) como en el **servidor** (por seguridad, nunca confiar solo en la validación del cliente). Una buena validación de entradas previene numerosas categorías de ataques, desde **inyecciones SQL/NoSQL** hasta **Cross-Site Scripting** y otros, ya que detiene los datos malformados en la frontera de la aplicación antes de que causen daño.

- **Escapar y sanitizar las salidas**: Complementario a validar entradas es **sanitizar la salida** que la aplicación genera hacia el usuario o hacia otros sistemas. Esto significa tomar cualquier dato que podría contener contenido especial (proveniente de entradas originalmente) y **codificarlo o escaparlo** según el contexto donde se vaya a insertar. Por ejemplo, si nuestra aplicación inserta el nombre de un usuario dentro de una página HTML, debemos asegurarnos de escapar caracteres como <, > y & para que, si el nombre contiene algo como "<script>alert('xss')</script>", no se interprete como código HTML sino como texto plano visible para el usuario. El escape adecuado depende del contexto: si es HTML, usar entidades HTML; si es para JavaScript dentro de un <script>, escapar de forma diferente; si es para un parámetro de SQL en un script SQL (aunque ahí se recomienda mejor usar consultas parametrizadas, ver más abajo). Muchos frameworks ya realizan auto-*escaping* de salidas en plantillas (Angular, React, Vue, etc. escapan la inyección de variables en la interfaz). No obstante, cuando se formen manualmente strings de salida o se usen técnicas de inyección deliberada de HTML (por ejemplo, en React usar dangerouslySetInnerHTML), **es responsabilidad del desarrollador sanitizar el**

contenido. Herramientas como OWASP Java Encoder, la función htmlspecialchars en PHP, o librerías como DOMPurify en el navegador, facilitan esta tarea. La regla general es: **todo dato dinámico se debe enviar escapado según el contexto**, para que el receptor (navegador u otro sistema) no lo confunda con comandos o código. Esta es la defensa principal contra XSS y otras inyecciones de código en la presentación.

- **Autenticación segura**: La autenticación es el proceso de verificar la identidad de los usuarios, y es un pilar de la seguridad. Una implementación insegura de autenticación puede dar lugar a **suplantación de identidad, filtración de credenciales** u otros problemas graves (de hecho, "Broken Authentication" solía figurar como categoría en OWASP Top 10). Para proteger la autenticación se deben seguir varias prácticas: utilizar **algoritmos de hash robustos para almacenar contraseñas** (por ejemplo, bcrypt, Argon2, nunca guardar contraseñas en texto plano), aplicar políticas de contraseña seguras (mínimo de caracteres, complejidad razonable, impedir contraseñas demasiado comunes), implementar mecanismos de **bloqueo o retardo ante múltiples intentos fallidos** (para mitigar *fuerza bruta*), y ofrecer **autenticación multifactor** en lo posible. Al diseñar el flujo de login/registro/recuperación de contraseña, hay que considerar también aspectos como asegurar la **pregunta de seguridad** o correos de recuperación para que no puedan ser explotados. Otra recomendación es usar **frameworks o servicios de identidad probados** (p. ej. OAuth 2.0 / OpenID Connect mediante proveedores o librerías especializadas) en vez de construir todo el sistema de autenticación manualmente, reduciendo así el margen de error. También se debe proteger la **gestión de sesiones**: establecer cookies de sesión con la bandera *Secure* (solo transmítelas por HTTPS) y *HttpOnly* (no accesibles desde JavaScript), utilizar identificadores de sesión aleatorios y revocar sesiones (logout) correctamente. En aplicaciones de token JWT, asegurar su firma con claves fuertes y limitar su duración. En resumen, la autenticación segura significa garantizar que **solo los usuarios legítimos puedan acceder** y que las credenciales estén bien resguardadas.

- **Autorización y control de accesos**: Una vez autenticado un usuario, la aplicación debe controlar *qué* acciones o recursos tiene permitidos (autorización). Muchas brechas ocurren por fallos en este aspecto (OWASP Top 10 lo refleja: **"Broken Access Control"** está entre los primeros riesgos). Para robustecer la autorización, se deben aplicar principios de **mínimos privilegios** y **control de acceso estricto**. Cada función o recurso de la aplicación debería estar accesible solo para los roles o usuarios autorizados, y **nunca confiar en el lado del cliente para hacer cumplir estas restricciones** (el servidor siempre debe verificar permisos). Es recomendable centralizar la lógica de autorización (por ejemplo, un middleware o guard en el backend que verifique permisos antes de cada petición sensible). Implementar checks como: verificar que el usuario autenticado solo acceda a sus propios datos y no a los de otros (p. ej., impedir manipulación de IDs en la URL para obtener datos ajenos), validar tokens o sesiones en cada petición, y usar sistemas de roles y privilegios bien definidos. Un ejemplo de control de acceso es asegurar que las rutas de API críticas requieren un rol admin, o que ciertas acciones (borrar, modificar datos

sensibles) requieren confirmaciones adicionales. En aplicaciones con **JWT** o tokens, incluir en el token los roles/permisos y validarlos. Además, las cuentas de servicio o acceso a base de datos desde la app también deben usar **credenciales con los privilegios mínimos necesarios** (principio de menor privilegio a nivel de base de datos, de modo que si la aplicación es comprometida, el daño en la BD se limite al rol usado). Para resumir, la autorización efectiva garantiza que cada usuario (humano o componente) **solo pueda realizar las acciones que le están explícitamente permitidas**, cerrando paso a escalamientos de privilegio.

- **Protección de datos sensibles (criptografía)**: Muchas aplicaciones manejan **información sensible** (datos personales, contraseñas, números de tarjeta, expedientes médicos, etc.) que debe protegerse tanto mientras está almacenada (*en reposo*) como cuando se transmite por la red (*en tránsito*). Para proteger datos en tránsito, la regla fundamental es usar siempre **protocolos cifrados** – por ejemplo, HTTPS (TLS) en la comunicación web, asegurando que todo el tráfico entre cliente y servidor va cifrado para que no pueda ser leído ni manipulado por atacantes en la ruta. Para datos en reposo, se debe considerar el **cifrado de bases de datos o campos sensibles**, o al menos cifrado a nivel de disco/archivo de respaldo. Además, no almacenar más datos sensibles de los necesarios (minimización de datos) y, si es posible, aplicar técnicas de **hash/salting** para datos como contraseñas, o tokenización para datos como tarjetas de crédito (reemplazar el dato real por un token referencial). Es vital **no inventar algoritmos criptográficos propios** – siempre usar algoritmos estándares considerados seguros (AES, RSA, SHA-256, etc.) e implementar la criptografía mediante bibliotecas confiables para evitar errores sutiles. La gestión de las **claves criptográficas** es igualmente importante: mantener las claves secretas fuera del código fuente (por ejemplo, en almacenes seguros o variables de entorno protegidas), rotar las claves periódicamente si corresponde y controlar estrictamente quién/qué puede acceder a ellas. Un error común es exponer secretos (claves API, contraseñas de BD, certificados) en repositorios de código público o en clientes – esto debe evitarse a toda costa. En síntesis, proteger datos sensibles significa que, incluso si un atacante obtiene acceso a la base de datos o intercepta comunicaciones, **no podrá leer información confidencial** por estar cifrada o inaccesible.

- **Seguridad en el acceso a bases de datos (prevención de inyecciones)**: Las **inyecciones SQL** (y más recientemente también inyecciones NoSQL en bases de datos no relacionales) son una de las vulnerabilidades más peligrosas y conocidas. Consisten en la inserción de comandos de consulta maliciosos a través de entradas no controladas, logrando desde consultas no autorizadas hasta la toma total de la base de datos. La principal defensa contra inyecciones es **nunca construir dinámicamente las consultas concatenando datos del usuario**. En su lugar, utilizar siempre **consultas parametrizadas o declaraciones preparadas** proporcionadas por la API de la base de datos. Esto implica que en el código, en vez de hacer algo como sql = "SELECT * FROM usuarios WHERE nombre = '" + usuario + "'" (vulnerable si usuario contiene algo como ' OR '1'='1), se utiliza un placeholder y se enlaza el valor

por separado, por ejemplo: sql = "SELECT * FROM usuarios WHERE nombre = $1" y luego pasar el parámetro en una lista/array al ejecutar la consulta. Todas las tecnologías de acceso a BD (drivers de PostgreSQL, MySQL, SQL Server, ORMs como Sequelize/TypeORM, etc.) soportan este mecanismo. Abajo vemos un ejemplo simplificado en Node.js con una consulta insegura vs. una parametrizada:

```
// Ejemplo en Node.js/Express - consulta SQL insegura (vulnerable a inyección)
app.get('/buscar', (req, res) => {
  const user = req.query.user;
  const query = "SELECT * FROM Clientes WHERE nombre = '" + user + "'";
  db.query(query, (err, resultados) => {
    // ... manejar resultados
  });
});

// Versión segura usando parámetros
app.get('/buscar', (req, res) => {
  const user = req.query.user;
  const query = "SELECT * FROM Clientes WHERE nombre = $1";  // placeholder parametrizado
  db.query(query, [user], (err, resultados) => {
    // ... manejar resultados
  });
});
```

En la versión segura, la base de datos recibe la consulta con un **parámetro** en lugar de un valor concatenado, y el motor se encarga de tratar el contenido de user **estrictamente como dato**, sin permitir que altere la sintaxis SQL, bloqueando así la inyección. Esta misma técnica se aplica para cualquier comando (INSERT, UPDATE, etc.) y en cualquier base de datos SQL. En el caso de **ORMs** (p. ej., usando Entity Framework, SQLAlchemy, etc.), por lo general al utilizar sus métodos de alto nivel ya se está protegido porque ellos internamente paramétrizan las consultas; sin embargo, si se ejecutan consultas *raw* con strings, hay que tener la precaución manual de parametrizar.

En cuanto a **NoSQL (MongoDB, etc.)**, aunque no usan SQL, también pueden sufrir inyecciones a través de consultas mal formadas. Por ejemplo, en MongoDB una típica consulta insegura ocurre si construimos un objeto de búsqueda directamente con datos del usuario sin validarlos, pudiendo un atacante inyectar operadores $ (como $ne, $gt) para alterar la lógica de la consulta. La prevención aquí pasa por **validar y sanitizar** los inputs antes de usarlos en consultas NoSQL y utilizar APIs seguras. Algunas bases NoSQL permiten consultas JavaScript (p. ej. operador $where en MongoDB), lo cual debe evitarse con datos no confiables, o usarse con extremo cuidado y validación estricta. En resumen: para cualquier interacción con bases de datos, **separar el código de los datos** es fundamental. Usar siempre parámetros o construcciones preparadas impedirá que un atacante "rompa" la cadena de comandos. Adicionalmente, se recomienda que las cuentas de base de datos usadas por la aplicación tengan solo los permisos necesarios (por ejemplo, si solo se requieren lecturas en ciertas tablas, no dar permisos de escritura o de administración). Así, incluso si ocurriera una inyección, el alcance del daño sería limitado.

- **Manejo adecuado de errores y excepciones**: La forma en que la aplicación responde ante errores puede afectar a la seguridad. Los mensajes de error muy verbosos o detallados (especialmente los que incluyen *stack traces*, información de la base de datos, o datos de configuración) pueden filtrar información sensible a un atacante. Por ello, se debe implementar un manejo global de excepciones que, de cara al usuario final, **entregue mensajes de error genéricos** (ej.: "Ocurrió un error, intente más tarde") sin revelar detalles internos. Mientras tanto, internamente, la aplicación debería **loggear** la información técnica del error en un lugar seguro para su análisis por desarrolladores. Este balance permite no dar pistas a atacantes (por ejemplo, un error de SQL expuesto podría indicar qué motor de base de datos se usa, facilitando ataques específicos) y aun así permitir la depuración. Asimismo, nunca se deben propagar excepciones sin control hasta el cliente; es mejor interceptarlas y responder adecuadamente. En entornos Node.js, por ejemplo, usar un middleware de error que capture cualquier excepción no manejada y evite que el servidor caiga o que la excepción se muestre en texto plano. Otro aspecto es **no incluir información sensible en los mensajes de error o logs**: por ejemplo, si falla un login, no especificar "usuario correcto, contraseña incorrecta" (ya que confirma que el usuario existe), simplemente dar un mensaje genérico de credenciales inválidas. En los logs del servidor, tampoco escribir contraseñas, tokens u otra PII (Información Personal Identificable) en claro. Resumiendo, el manejo de errores debe ser **discreto hacia afuera y completo hacia adentro**: al usuario solo lo necesario, al equipo interno suficiente detalle en un log seguro.

- **Registro y monitoreo de eventos de seguridad**: Para poder detectar y responder a incidentes, la aplicación debe contar con **logging** de los eventos relevantes de seguridad y mecanismos de monitoreo/alerta. Esto incluye registrar intentos fallidos de autenticación, cambios críticos en configuración, accesos a funciones sensibles, entradas de datos anómalas, etc. Dichos registros deben almacenarse de forma segura (idealmente, inmutables o en sistemas centralizados) y cumplir con normativas de privacidad (no registrar datos sensibles en claro, como ya se

mencionó). Un buen sistema de **seguridad en registros** también involucra generar **alertas** cuando ocurren ciertos patrones sospechosos (por ejemplo, decenas de intentos fallidos de login pueden indicar un ataque de fuerza bruta; una secuencia de consultas inusuales podría indicar un intento de inyección, etc.). OWASP Top 10 incluye la insuficiencia de **registro y monitoreo** como un riesgo importante, ya que sin ellos es difícil detectar intrusiones. Implementar estos controles permite no solo investigar incidentes después de sucedidos, sino idealmente *identificarlos en tiempo real* y activar respuestas (por ejemplo, bloquear una IP temporalmente tras detectar un comportamiento malicioso repetido). Herramientas como SIEM, monitoreo de aplicaciones (APM) con reglas de seguridad, o incluso soluciones personalizadas, pueden integrarse para supervisar la aplicación en producción. En definitiva, **lo que no se registra no se puede auditar**: llevar bitácoras de seguridad y revisarlas activamente es un mecanismo esencial para mitigar el impacto de vulnerabilidades que pudieran haberse pasado por alto.

- **Actualizaciones y gestión de parches**: Un mecanismo de protección a veces subestimado es la disciplina de **mantener el software actualizado**. Esto abarca tanto el propio código de la aplicación como los componentes de terceros (frameworks, librerías, servidores, SGBD, etc.). Los atacantes suelen explotar vulnerabilidades conocidas para las cuales ya existen parches; por eso, una de las formas más sencillas de proteger un sistema es **aplicar las actualizaciones de seguridad disponibles puntualmente**. En el contexto de desarrollo, esto significa que el equipo debe estar atento a boletines de seguridad (por ejemplo, cuando sale una nueva versión de Node.js o de Angular que parchea una vulnerabilidad, o cuando se anuncia una CVE en una librería utilizada). Integrar herramientas automatizadas de escaneo de dependencias (como npm audit para Node, Retire.js, Snyk, Dependabot, etc.) ayuda a identificar componentes vulnerables en el proyecto. Adicionalmente, durante el mantenimiento, es prudente planificar **ventanas de actualización** regulares donde se revisan y suben de versión los componentes con parches críticos. Por supuesto, se debe probar adecuadamente después de actualizar para garantizar compatibilidad, pero ignorar las actualizaciones de seguridad puede dejar la puerta abierta a fallos ya conocidos. Junto con esto, hay que gestionar la **obsolescencia**: si un framework o plataforma queda sin soporte (end of life), planificar su migración, ya que desde ese punto no recibirá más parches y se volverá un blanco fácil.

Las técnicas anteriores, combinadas, proporcionan una base sólida para desarrollar aplicaciones con un **perfil de riesgo mucho menor**. Cabe mencionar que muchas de ellas están interrelacionadas; por ejemplo, para evitar inyección necesitas validar entradas y parametrizar salidas hacia la BD, para evitar XSS necesitas validar entradas y escapar salidas al navegador, etc. Por eso, los expertos enfatizan adoptar un **enfoque integral de "defensa en profundidad"**: múltiples capas de controles que se refuercen mutuamente. Un desarrollador debe pensar siempre: *¿Qué es lo peor que podría hacer un usuario malicioso con esta funcionalidad?* y a partir de allí asegurarse de que existen barreras para impedirlo o mitigarlo. Como se señaló, organizaciones como **OWASP** proveen guías prácticas (Top 10 de Riesgos, Top 10 de Controles Proactivos, Cheat Sheets de secure coding, etc.) que resumen

estas técnicas y pueden servir como referencia en cualquier proyecto. En la siguiente sección, exploraremos brevemente algunos ejemplos concretos en tecnologías específicas, aplicando los principios discutidos.

Normativas y estándares de seguridad (OWASP, NIST, ISO 27001)

En el campo de la seguridad de la información y la seguridad de aplicaciones, existen numerosas **normativas, estándares y guías** que orientan a las organizaciones y desarrolladores sobre *qué prácticas seguir* para lograr sistemas más seguros. A continuación, nos enfocamos en tres referencias destacadas: **OWASP**, **NIST** e **ISO 27001**, describiendo brevemente su relevancia en el desarrollo seguro de software.

- **OWASP (Open Web Application Security Project)**: OWASP es una **fundación internacional sin ánimo de lucro** dedicada a la seguridad de las aplicaciones web. Uno de sus principios es que todo su material es abierto y gratuito, lo que ha permitido que sus guías se conviertan en estándares de facto ampliamente utilizados en la industria. Quizás el proyecto más conocido de OWASP es el **OWASP Top 10**, un reporte que se actualiza periódicamente enumerando las **10 categorías de riesgo de seguridad más críticas en aplicaciones web**. OWASP denomina este Top 10 como un documento de *concientización* e incluso recomienda que todas las empresas lo incorporen en sus procesos de desarrollo para **mitigar los riesgos listados**. En su edición más reciente (2021), el OWASP Top 10 incluye riesgos como: **control de acceso roto**, **fallos criptográficos**, **inyecciones**, **diseño inseguro**, **configuraciones inseguras**, **componentes vulnerables/desactualizados**, **fallos de identificación y autenticación**, **fallos de integridad de software y datos**, **registro y monitoreo insuficientes**, y **SSRF**. Cada categoría viene con una descripción del problema, ejemplos de ataques y recomendaciones de mitigación. Además del Top 10 de riesgos, OWASP produce otras guías útiles: por ejemplo, el **OWASP ASVS** (estándar de verificación de seguridad de aplicaciones) que lista requisitos de seguridad según niveles; el **OWASP Testing Guide** para pruebas; y el mencionado **OWASP Proactive Controls** para desarrolladores. También mantienen proyectos prácticos (herramientas) como OWASP ZAP (para análisis dinámico) y muchas *cheat sheets* (hojas de referencia rápida) sobre cómo realizar implementaciones seguras de todo tipo de funcionalidades. En resumen, OWASP es una referencia esencial: provee **contenido educativo y herramientas** que los equipos de desarrollo pueden usar para mejorar la seguridad de su ciclo de vida. Incorporar OWASP en la cultura de desarrollo (por ejemplo, entrenar a los desarrolladores con Top 10, usar ZAP en pruebas, seguir ASVS en requisitos) es una excelente manera de alinear prácticas con estándares reconocidos globalmente.

- **NIST (National Institute of Standards and Technology)**: NIST es una agencia de estándares de Estados Unidos que, entre muchas otras áreas, publica lineamientos en ciberseguridad. Varios de sus **estándares especiales (SP)** se relacionan con seguridad en el ciclo de desarrollo. Un ejemplo reciente y notable es el **NIST SP 800-218: Secure Software Development Framework (SSDF)**, publicado en 2022. En su resumen, NIST señala que pocos modelos de ciclo de vida abordan la seguridad con

detalle, por lo que típicamente es necesario **añadir prácticas de desarrollo seguro a cada modelo de SDLC** para asegurar que el software resultante sea seguro. El documento SP 800-218 propone precisamente un conjunto básico de **prácticas de desarrollo seguro de alto nivel** integrables en cualquier SDLC. Entre estas prácticas se incluyen tareas como: preparar la organización (por ej., formar a los desarrolladores y definir un entorno seguro de codificación), proteger el código (control de versiones, análisis de vulnerabilidades en código y dependencias), producir software de forma segura (revisiones, pruebas, gestión de defectos) y responder a vulnerabilidades (mecanismos de reporte, parcheo rápido, aprendizaje post-mortem). La intención es que siguiendo estas prácticas, los **productores de software reduzcan la cantidad de vulnerabilidades** en sus productos y mitiguen el impacto de las que surjan, abordando incluso las causas raíz para prevenir recurrencias. Además, NIST SP 800-218 busca proveer un **lenguaje común** para que tanto desarrolladores como adquirentes de software puedan evaluar si se siguen buenas prácticas de seguridad en el desarrollo. Más allá del SSDF, NIST también tiene otros documentos relevantes: el **SP 800-53** (catálogo de controles de seguridad, que incluye controles aplicables al desarrollo seguro), la guía **SP 800-64** (sobre consideraciones de seguridad en el ciclo de vida de sistemas), y documentos enfocados a DevSecOps, seguridad en contenedores, etc. Si bien los estándares NIST a veces están orientados a entornos gubernamentales o grandes empresas, muchos principios son aplicables universalmente. Adherirse a marcos como el SSDF de NIST demuestra un enfoque sistemático hacia la seguridad en el desarrollo y puede ayudar a cumplir requisitos regulatorios (p. ej., en EE.UU. se está impulsando que los proveedores de software sigan estas prácticas para mejorar la seguridad de la cadena de suministro de software).

- **ISO/IEC 27001**: Es un estándar internacional centrado en **Sistemas de Gestión de Seguridad de la Información** (SGSI). Si bien no se enfoca exclusivamente en desarrollo de software, dentro de sus controles abarca prácticas para asegurar los procesos de desarrollo y mantenimiento de sistemas de información. En la versión 2013 de ISO 27001 (actualizada en 2022), el **Anexo A, dominio 14** ("Adquisición, desarrollo y mantenimiento de sistemas") contiene controles explícitos relacionados con desarrollo seguro. Por ejemplo, la norma indica que se debe **garantizar que la seguridad de la información esté diseñada e implementada dentro del ciclo de vida de desarrollo de los sistemas**. Entre las medidas recomendadas está tener una **política de desarrollo seguro** donde se definan y documenten los procedimientos de seguridad a aplicar durante el ciclo de vida, incluir **controles de validación de datos** en el desarrollo, y proteger la **información sensible** involucrada en los entornos de desarrollo (p. ej., si se usan datos reales en pruebas, anonimizar o generar datos sintéticos). ISO 27001 exige que los requisitos de seguridad se identifiquen desde la fase de requisitos y se integren en todo el proceso, similar a lo ya descrito (este alineamiento no es casual, pues refleja las *mejores prácticas globales*). Las organizaciones certificadas en ISO 27001 deben demostrar que siguen estos controles; por ejemplo, que hacen revisiones de código, que controlan el acceso al repositorio de código fuente, que se aseguran de eliminar cuentas de backdoor antes

de pasar a producción, etc. También se incluyen controles sobre la **seguridad en entornos de desarrollo** (asegurando que los entornos de prueba/desarrollo tengan protecciones si manejan datos sensibles), y la **gestión de vulnerabilidades técnicas** (que haya un proceso formal para manejar parches y avisos de seguridad). En resumen, ISO 27001 proporciona un **marco de gestión** que obliga a incorporar la seguridad en el desarrollo de software de forma **política y procedimental**. A diferencia de OWASP o NIST, que dan guías técnicas específicas, ISO 27001 es más general, pero es muy influyente porque muchas empresas buscan certificarse para demostrar su compromiso con la seguridad. Para un equipo de desarrollo, trabajar en una organización regida por ISO 27001 implica que habrá procesos establecidos que ellos deben seguir para, por ejemplo, gestionar cambios de software con evaluaciones de riesgos, cumplir estándares de codificación segura, etc., todo lo cual refuerza la calidad y seguridad del producto final.

En conclusión, **alinearse con normativas y estándares reconocidos** proporciona múltiples beneficios: por un lado, asegura que no se están ignorando aspectos importantes de seguridad (ya que estos estándares recopilan la experiencia colectiva de la industria); por otro, puede ser necesario por cumplimiento (compliance) contractual o legal en ciertos proyectos. OWASP ofrece guías muy prácticas de aplicación directa para desarrolladores y testers, NIST brinda un respaldo metodológico riguroso e iniciativas como el SSDF que complementan políticas nacionales de seguridad, e ISO 27001 da un paraguas organizativo que integra la seguridad de la información a nivel de gestión empresarial. Un desarrollador de software seguro debe al menos estar familiarizado con **OWASP Top 10** (para conocer las amenazas principales a combatir) y con las políticas internas derivadas de estándares como ISO 27001 o NIST que su organización adopte. Usar estos estándares como referencia es una forma de garantizar **conformidad con las mejores prácticas** y mejorar la confianza en la robustez de los sistemas desarrollados.

Ejemplos prácticos en tecnologías web

A continuación, presentaremos **ejemplos concretos** de cómo se aplican las prácticas de seguridad y protección de vulnerabilidades en algunas de las tecnologías específicas que se usan comúnmente en el desarrollo de aplicaciones web. El propósito es ilustrar consejos puntuales de seguridad en entornos de frontend (Angular, React, Next.js), backend (Node.js con Express, framework NestJS) y bases de datos (SQL como PostgreSQL/MySQL/SQL Server, y NoSQL como MongoDB). No se pretende cubrir exhaustivamente cada tecnología, sino **destacar situaciones típicas de riesgo y sus mitigaciones** para que sirvan de referencia en proyectos reales.

Seguridad en el frontend (Angular, React, Next.js)

En las aplicaciones del lado del cliente, una de las mayores preocupaciones de seguridad es el **Cross-Site Scripting (XSS)**, es decir, la inyección de scripts o HTML malicioso que pudiera ejecutarse en el navegador del usuario. Los frameworks modernos **Angular** y **React** (y por extensión frameworks meta como **Next.js**, que se basa en React) incluyen protecciones por defecto contra XSS, pero es importante entenderlas y no anularlas inadvertidamente.

- **Angular**: Este framework fue diseñado con una fuerte consideración por la seguridad contra XSS. Angular **escapa automáticamente** cualquier contenido que se interpole en las vistas. Por ejemplo, si en un componente se vincula una variable a la plantilla HTML mediante {{variable}}, Angular convertirá caracteres especiales en entidades HTML, evitando que fragmentos <script> u otros se interpreten como código ejecutable. Incluso al usar binding de propiedades [innerHTML], Angular aplica un **sanitizador interno** que eliminará contenido potencialmente peligroso (como etiquetas <script>). Angular también trae integrada una protección contra **CSRF**: su módulo HttpClient por defecto añadirá un header X-XSRF-TOKEN en las peticiones HTTP si detecta una cookie especial, facilitando la implementación de tokens anti-CSRF en conjunto con el backend. **Buenas prácticas en Angular** incluyen: nunca usar métodos como bypassSecurityTrustHtml (del DomSanitizer) a menos que esté totalmente justificado y se sepa lo que se hace – este método deshabilita las medidas de seguridad de Angular para insertar contenido HTML arbitrario, lo cual puede abrir la puerta a XSS si el contenido proviene de usuarios. En resumen, **aprovechar las protecciones nativas**: no construir manualmente HTML con innerHTML sin sanitizar, no desactivar el escape de Angular y mantener actualizado el framework para recibir parches de seguridad.
- **React**: En React (incluyendo React + Next.js), la protección contra XSS también viene automáticamente en el proceso de renderizado de componentes. Por defecto, cuando uno inserta una variable en JSX, por ejemplo {userInput} , React escapará los caracteres especiales en userInput de modo que aunque contenga HTML/script, el resultado se mostrará como texto plano. React *solo* ejecutará código HTML si el desarrollador utiliza la propiedad especial dangerouslySetInnerHTML. Esta es la vía para insertar HTML crudo en el DOM, y justamente se llama "peligrosa" porque **es responsabilidad del desarrollador** asegurar que el contenido es seguro antes de usarla. Si una aplicación React **no utiliza dangerouslySetInnerHTML**, el riesgo de XSS se reduce enormemente (aunque aún es importante validar la entrada para otras cosas, como prevenir HTML roto que pueda afectar el layout, etc.). En caso de necesitar insertar contenido enriquecido proporcionado por usuarios (por ejemplo, mostrar un fragmento de HTML que un usuario introdujo), se debe **sanitizar ese contenido**. Una estrategia es usar una biblioteca como **DOMPurify** que limpia un string HTML removiendo cualquier cosa maliciosa (scripts, eventos, etc.). Por ejemplo, en React podríamos hacer:

```
// Inseguro: inserta HTML sin sanitizar (podría ejecutarse script si content es malicioso)
<div dangerouslySetInnerHTML={{ __html: userContent }} />

// Seguro: sanitizar el HTML antes de insertarlo usando DOMPurify u otra librería
import DOMPurify from 'dompurify';
const cleanContent = DOMPurify.sanitize(userContent);
```

<div dangerouslySetInnerHTML={{ __html: cleanContent }} />

En este ejemplo, userContent representa datos HTML que potencialmente provienen del usuario. La primera forma (insegura) podría conducir a XSS; la segunda utiliza DOMPurify para filtrar etiquetas o atributos peligrosos antes de pasarle el contenido a React. **Next.js**, al basarse en React, comparte estos mismos principios de front-end. Sin embargo, Next.js agrega consideraciones de seguridad en el **renderizado del lado del servidor** (SSR) y en sus rutas API. Cuando Next genera páginas en el servidor con datos dinámicos, se debe tener el mismo cuidado de escapado de contenido al renderizar. Next 11+ permite configurar fácilmente **cabeceras de seguridad** globales (Content Security Policy, etc.) mediante next.config.js, lo que es recomendable para mitigar ataques XSS complementariamente (CSP puede restringir la ejecución de scripts no autorizados). Asimismo, Next.js al ofrecer **funciones serverless/API** debe considerarse también desde perspectiva backend (ver apartado de Node/Express más abajo): validar los inputs que llegan a esas API routes, autenticar correctamente las llamadas, etc.

En resumen, en el frontend con frameworks SPA/SSR: **no introducir manualmente vulnerabilidades que el framework ya previene**. Usar siempre los mecanismos de templating seguros proporcionados, sanitizar cualquier HTML dinámico necesario y aprovechar opciones como establecer políticas de contenido (CSP) para añadir capas defensivas. También hay que mantener dependencias front-end actualizadas, pues a veces surgen vulnerabilidades en bibliotecas de terceros (ej.: una librería de componentes podría tener una falla XSS si no escapa algo). Pero si se siguen las guías oficiales (Angular y React tienen secciones de seguridad en su documentación), la superficie de ataque en el frontend queda bastante controlada.

Seguridad en el backend (Node.js/Express, NestJS)

Del lado del servidor, donde residen la lógica de negocio y la interacción directa con bases de datos y sistemas, aplican muchos de los principios ya discutidos (validación, autenticación, etc.). Nos enfocaremos en el ecosistema **Node.js** con el popular framework **Express.js** y su contraparte orientada a arquitectura limpia **NestJS**, que son entornos muy comunes en aplicaciones web modernas.

- **Node.js & Express**: Express es un framework minimalista, lo que significa que no impone por sí mismo muchas capas de seguridad; depende en gran medida de cómo el desarrollador lo configure. Una de las primeras medidas recomendadas en cualquier app Express es habilitar los **middlewares de seguridad estándar**, por ejemplo usar **Helmet** (un middleware que ajusta cabeceras HTTP de seguridad). Helmet puede establecer cabeceras como Content-Security-Policy (CSP), X-Frame-Options (para prevenir clickjacking), X-Content-Type-Options (evitar interpretaciones MIME erróneas), etc., con configuraciones por defecto sensatas. Agregar app.use(require('helmet')()); al iniciar la aplicación aporta múltiples protecciones con un par de líneas. Otro aspecto es habilitar **CORS** de forma segura si la API será consumida desde otro dominio: usar la librería cors configurando orígenes permitidos específicos, en lugar de simplemente app.use(cors()) abierto a todos (lo cual no es inseguro por sí mismo pero puede llevar a exposición no intencional). En cuanto a la

gestión de sesiones/autenticación, si se usan sesiones basadas en cookie (por ej. express-session), asegurarse de usar cookies seguras (Secure, HttpOnly, SameSite). Si se usan JWT para stateless auth, validar siempre la firma y verificar su vigencia (no aceptar JWT expirados) y audiencias/roles correctos. Express no provee esto internamente, pero hay middleware o librerías (como passport para integrar estrategias de auth).

La **validación de datos** en Express típicamente se hace manualmente o con ayuda de paquetes como express-validator o joi/celebrate. Es importante validar parámetros de ruta, query strings, cuerpo de peticiones JSON, etc., antes de usarlos. Por ejemplo, si se espera que un ID sea numérico, devolver un error 400 si en la ruta viene un valor que no es número en lugar de proceder con una consulta posiblemente maliciosa. Asimismo, al interactuar con la base de datos (ya sea directamente con drivers como pg para PostgreSQL, mysql2 para MySQL, orms como mongoose para Mongo, etc.), siempre utilizar **consultas parametrizadas** como se mostró anteriormente, o métodos del ORM que internamente logren el mismo objetivo. Una vulnerabilidad a evitar en Node es la **inyección de comandos del sistema**: a veces los desarrolladores usan módulos como child_process.exec pasando entradas de usuario a comandos de shell; esto puede ser explotado similar a una inyección SQL, pero en el shell del servidor. La solución es igualmente no construir comandos shell con datos no confiables (o usar spawn con argumentos en lista para separar comando y args de forma segura).

Otra práctica en Node es manejar adecuadamente las **excepciones asíncronas**: por ejemplo, usando try/catch en funciones async/await o capturando los errores de promesas, para evitar que un error inesperado cierre la aplicación (y potencialmente deje al sistema en un estado inseguro). También configurar límites, como **rate limiting** en rutas críticas (hay paquetes como express-rate-limit) para mitigar fuerza bruta o abuso de APIs, y **validar payloads** de tamaño excesivo (configurar límites de body parser) para prevenir ataques de denegación de servicio por payloads grandes.

- **NestJS**: NestJS es un framework construido sobre Node (generalmente usando Express internamente, aunque permite usar Fastify) que aporta una arquitectura modular y herramientas out-of-the-box, incluidas varias para seguridad. Con Nest, es sencillo habilitar **validación global** de DTOs usando class-validator y class-transformer. Por ejemplo, definimos un DTO (objeto de transferencia de datos) para una ruta, con decoradores como @IsString(), @IsEmail(), etc., y al usar app.useGlobalPipes(new ValidationPipe()), automáticamente Nest validará y rechazará peticiones que no cumplan esos esquemas antes de llegar al controlador. Esto proporciona una **capa uniforme de validación** muy recomendable. Nest también integra facilmente **Helmet** (existe el paquete @nest/middleware-helmet o se puede usar directamente) y soporta configuración de CORS de manera sencilla al iniciar la app (app.enableCors() con opciones). Para autenticación, Nest propone el uso de **Guards** – por ejemplo, Passport JWT guard – que se pueden aplicar a controladores o rutas para protegerlas. Un ejemplo: usar @UseGuards(AuthGuard('jwt')) encima de un controlador asegurará que solo se accede con un JWT válido. Nest también fomenta la **inyección de dependencias**, lo

cual puede ayudar a gestionar de forma centralizada aspectos como un servicio de hashing de contraseñas, un servicio de logging central, etc., y eso facilita implementar las prácticas de seguridad de forma transversal.

En Nest, como en cualquier backend, también es importante **sanear los datos de salida** si por alguna razón se reenvían al frontend tal cual entraron (por ejemplo, si una API toma un campo y lo envía de vuelta formateado en HTML o en un PDF, etc.). Y un punto a destacar: NestJS (y Express) permiten escribir **pruebas unitarias** y de integración; utilizar tests para simular inputs maliciosos y confirmar que el sistema los maneja correctamente es muy útil en un contexto de desarrollo seguro.

En ambos casos (Express puro o Nest), siempre se debe considerar la **configuración segura en producción**: deshabilitar *debug* o *stack traces* en las respuestas de error, habilitar registro seguro, asegurar que la aplicación se ejecuta con una identidad de sistema con privilegios limitados, etc. Node.js en particular debe estar actualizado (para recibir parches de seguridad del motor) y se deben monitorear sus dependencias (por ejemplo, usar npm audit regularmente). También vigilar las vulnerabilidades propias del ecosistema, como deserialización insegura en cookies o en JSON Web Tokens (usando librerías actualizadas para JWT se previene esto, ya que en el pasado hubo CVEs en librerías de JWT por algoritmos débiles).

En suma, **programar en Node/Express/NestJS de forma segura** requiere apoyarse en los *middlewares* y utilidades que proveen (Helmet, validación, guards de auth), y seguir principios de saneamiento de datos en cada frontera (entre cliente-servidor, servidor-BD, servidor-sistemas externos). El desarrollador debe estar atento tanto a las amenazas clásicas (XSS, inyección SQL) como a las particulares del entorno (ej. prototipo poisoning en objetos JavaScript si no se maneja correctamente la entrada JSON, o ataques a las dependencias como la typosquatting de paquetes). Afortunadamente, la comunidad de Node es activa en proveer contramedidas y la adopción de frameworks como Nest indica una madurez en facilitar patrones seguros.

Consideraciones de seguridad en bases de datos

Las bases de datos son donde residen los datos críticos de las aplicaciones, por lo que protegerlas es vital. Ya discutimos la medida técnica más importante a nivel aplicación: prevenir **inyecciones** mediante consultas parametrizadas y validación. Sin embargo, hay más consideraciones específicas según el motor de base de datos que se utilice:

- **PostgreSQL / MySQL / SQL Server (motores SQL)**: Estos sistemas de gestión de bases de datos relacionales comparten principios de seguridad similares. Además de usar siempre consultas parametrizadas, es importante configurar **cuentas de usuario con privilegios limitados** para la aplicación. Por ejemplo, si nuestra aplicación web necesita leer y escribir en ciertas tablas, pero nunca debería borrar tablas completas, podemos otorgarle una cuenta SQL que tenga SELECT, INSERT, UPDATE, DELETE en esas tablas específicas, pero no permisos de DROP o ALTER (que solo un DBA o administrador necesitaría). De esta forma, incluso si se logra comprometer la aplicación, el daño potencial en la BD está acotado. Otro punto es

habilitar el cifrado de la conexión: para bases de datos remotas, usar SSL/TLS en la conexión de la aplicación a la BD para que datos sensibles (como contraseñas SQL o datos consultados) no viajen en claro en la red. La **gestión de parches** es crucial: mantener PostgreSQL/MySQL/SQL Server actualizados, ya que a veces se descubren vulnerabilidades a nivel de motor que podrían ser explotadas (especialmente en MySQL en el pasado han existido fallos de auth, etc.). Activar los **logs de consultas** puede ayudar a detectar comportamientos anómalos (por ejemplo, un patrón de consultas diferente si alguien intenta inyecciones). En cuanto a **configuración**, asegurar parámetros seguros: en producción, restringir accesos (por ejemplo, permitir conexiones solo desde el servidor de aplicaciones, no desde cualquier IP), en MySQL no usar cuentas con contraseña vacía, en PostgreSQL editar pg_hba.conf adecuadamente, etc. Para SQL Server, integrarlo con autenticación Windows o usar cuentas dedicadas con contraseñas fuertes. Finalmente, respaldos: aunque no es directamente "seguridad" contra ataques, tener respaldos protegidos (cifrados y almacenados seguros) ayuda a recuperarse de incidentes (por ejemplo, ante un ataque ransomware o sabotaje).

- **MongoDB (y otros NoSQL)**: Con MongoDB, la inyección se previene con similar estrategia — usando los métodos de la API (p. ej. find({ campo: valor }) automáticamente manejará valor como dato; el problema viene si uno construye consultas usando operadores provenientes del cliente). Mongo, al ser schemaless, también requiere *validar el esquema a nivel de aplicación*: si esperamos que un campo sea string, no introducirlo como número, etc., para evitar casos donde tipos inesperados puedan causar un comportamiento no previsto. En versiones modernas de MongoDB, la configuración por defecto ya exige autenticación (en versiones viejas era opcional y eso llevó a muchas bases expuestas). Asegúrese de **habilitar autenticación** en la base, usar usuarios con roles (Mongo tiene roles granulares, como readWrite en una DB específica), y **comunicación cifrada TLS** si es un despliegue remoto. MongoDB ofrece opciones de cifrado de datos en reposo en sus versiones Enterprise; si se manejan datos muy sensibles, podría considerarse. Un consejo de seguridad es evitar operaciones peligrosas como db.eval() o $where con lógica JavaScript del lado de la base, ya que pueden abrir puertas a ejecución arbitraria; casi siempre hay formas alternativas de lograr lo mismo con consultas normales. Para otras bases NoSQL similares (Redis, Cassandra, etc.), las recomendaciones generales son: aplicar autenticación, limitar el acceso de red, usar conexiones seguras y actualizar regularmente. Redis, por ejemplo, no debería estar expuesto directamente a Internet y conviene establecer una contraseña y reglas de firewall.

- **Interacción aplicación-BD**: Independientemente del motor, la aplicación debe manejar bien los **errores de la base de datos**. Si una consulta falla (por ejemplo, por sintaxis o por violación de restricciones), no devolver el error crudo al usuario ya que podría revelar detalles (p. ej., "error de sintaxis cerca de…", que puede dar pistas para SQLi). En lugar de eso, loguearlo y mostrar un mensaje genérico. También, implementar **reintentos controlados** ante fallos transitorios de conexión, pero con

límites para no causar bucles infinitos que saturen la BD. Cuidar la **concurrencia y transacciones** para no terminar con datos inconsistentes que un atacante pudiera explotar (aunque esto es más de integridad que de seguridad, pero un sistema inconsistente a veces deriva en brechas lógicas).

Como se puede ver, muchas medidas de seguridad en bases de datos recaen más en **una buena administración y configuración** que en cambios de código. En un contexto académico, es importante que el estudiante no solo confíe en la base de datos para "mantenerse segura", sino que entienda que la aplicación es la primera barrera (evitando inyecciones) y la configuración es la segunda (limitando daños si algo pasa). Un caso famoso fue el de instancias de MongoDB expuestas sin autenticación que fueron atacadas en masa (**ataque "Meow"**), simplemente porque los desarrolladores no habilitaron la seguridad predeterminada. La lección es que al desplegar cualquier base de datos, debemos **asegurarla igual que aseguramos un servidor**: con contraseñas, parches, y principios de mínimo privilegio.

Actividad práctica propuesta

Para consolidar los conceptos de desarrollo seguro estudiados, se propone la siguiente **actividad práctica integradora**. El objetivo es que los alumnos apliquen las técnicas de protección de vulnerabilidades en un proyecto de software real o simulado, recorriendo varias etapas del S-SDLC. Se recomienda realizar esta actividad en equipos pequeños, aprovechando un proyecto web (preferiblemente el que estén desarrollando en el curso con las tecnologías mencionadas: Angular/React/Next en frontend, Node/Nest/Express en backend, base de datos correspondiente).

Descripción de la actividad: Implementar mejoras de seguridad en una aplicación web existente, siguiendo estos pasos:

1. **Análisis de vulnerabilidades en la aplicación**: Cada equipo debe tomar su aplicación web (o un módulo de la misma) y realizar un análisis para identificar posibles vulnerabilidades. Para guiarse, usar el **OWASP Top 10** como lista de comprobación. Por ejemplo, verificar si la aplicación podría ser susceptible a inyección SQL (¿concatena entradas en consultas?), XSS (¿refleja entradas del usuario en la interfaz sin escapar?), problemas de autenticación (¿almacena contraseñas en texto plano o reutiliza sesiones inseguramente?), etc. Pueden emplear herramientas automatizadas sencillas, como **OWASP ZAP** o **browser dev tools**, para detectar ciertas issues, pero principalmente se busca un análisis manual guiado por lo aprendido. **Entregar**: una breve lista de potenciales vulnerabilidades encontradas o aspectos que podrían mejorarse (al menos 3 ítems).

2. **Definición de requerimientos de seguridad y plan de mejora**: Con base en los hallazgos, definir **objetivos de seguridad** para el proyecto. Por ejemplo: "Evitar inyección SQL en el módulo de búsqueda", "Proteger las credenciales de usuario con hash seguro y política de complejidad", "Implementar validación de campos en el formulario X para evitar datos inválidos". Cada equipo listará qué acciones concretas de mejora va a realizar en la aplicación para subsanar o mitigar los riesgos

identificados. Priorizar las acciones de mayor impacto (p. ej., si no se hacía validación alguna de inputs, eso sería prioridad alta). **Entregar**: una lista de medidas de seguridad que implementarán, alineadas con los problemas del punto 1 (por ejemplo, junto a cada potencial vulnerabilidad, anotar la solución propuesta).

3. **Implementación de las mejoras en el código**: Proceder a **codificar** las soluciones planificadas. Aquí entran en juego las tecnologías específicas:
    - Implementar **validación de datos** en el backend (por ejemplo, usando express-validator en rutas Express, o DTOs con class-validator en NestJS) para los inputs identificados como críticos.
    - Sanitizar/escapar cualquier salida que muestre datos del usuario en el frontend Angular/React. Si se encuentra algún lugar donde se usa innerHTML o dangerouslySetInnerHTML, corregirlo para que use datos sanitizados o cambiar el enfoque para no necesitar HTML crudo.
    - Añadir protecciones genéricas: por ejemplo integrar Helmet en el servidor Node, configurar políticas de CORS apropiadas, asegurarse de usar HTTPS (si es local, al menos conceptualizarlo o usar localhost).
    - Revisar las consultas a la base de datos: sustituir concatenaciones por consultas parametrizadas/preparadas. Si usan un ORM, verificar que no estén usando métodos que ejecuten SQL crudo sin parámetros. Hacer pruebas con inputs maliciosos (por ejemplo, comillas ' en campos de texto) para confirmar que la aplicación ya no es vulnerable.
    - Mejorar la **gestión de autenticación**: si la aplicación no forzaba políticas de contraseña, agregarlas (por ejemplo, en el registro rechazar contraseñas muy débiles); asegurar el hash de las contraseñas (usar bcrypt u otro algoritmo fuerte con salt). Implementar un mecanismo de logout y expiración de sesión/token si no existía.
    - Activar **logging de eventos de seguridad**: por ejemplo, registrar en el servidor cuándo ocurren 5 intentos fallidos de login, o cuándo un usuario cambia su contraseña, etc., guardando estos eventos en un archivo o consola para su posterior revisión.
    - Cualquier otra mejora pertinente según el plan (por ejemplo, agregar un captcha si hubiera problemas de bots, limitar tamaño de subidas de archivos para evitar DoS, etc.), dependiendo de la naturaleza de la aplicación.

Entregar: el código actualizado (o fragmentos relevantes de código) que reflejen las mejoras. Comentar en el propio código o en un documento qué se cambió y por qué, haciendo referencia a la práctica de seguridad implementada. Por ejemplo: “Se añadió validación al campo X en el endpoint Y para prevenir inyección (OWASP A03: Injection)”.

4. **Pruebas y verificación**: Una vez aplicadas las medidas, cada equipo debe **probar la aplicación** para verificar que los riesgos se mitigaron sin romper la funcionalidad. Por

ejemplo, intentar de nuevo ingresar un input malicioso y comprobar que ahora la aplicación lo rechaza o escapa correctamente. Usar herramientas como OWASP ZAP en modo **scan** contra la aplicación para ver si aún detecta vulnerabilidades comunes. También probar casos de uso normales para asegurarse de que las validaciones añadidas no impiden funcionalidades válidas (ej.: que un usuario legítimo pueda registrarse con una contraseña válida, etc.). **Entregar**: un breve informe de resultados de pruebas, indicando para cada mejora si se confirmó su eficacia. Incluir capturas de pantalla o outputs de herramientas si es relevante (por ejemplo, "antes ZAP reportaba XSS en tal página, después de la corrección ya no lo reporta").

5. **Documentación y reflexión**: Finalmente, redactar una **conclusión** sobre los cambios realizados y su importancia. ¿Qué aprendieron sobre el impacto de cada vulnerabilidad y la forma de prevenirla? ¿Encontraron algún desafío integrando las medidas de seguridad (por ejemplo, hubo que refactorizar código, o equilibrar validaciones entre frontend y backend)? ¿Qué harían diferente en el futuro desde el inicio de un proyecto para incorporar la seguridad desde la fase de diseño? Esta reflexión busca afianzar la mentalidad de *Secure by Design*. **Entregar**: una o dos páginas de discusión grupal sobre estas preguntas, mencionando cómo el ejercicio mejoró la seguridad de su proyecto y qué podrían implementar a futuro (p. ej., pruebas de seguridad automatizadas en el pipeline CI, o adopción de otro estándar OWASP, etc.).

Criterios de evaluación: Se valorará la **integralidad** de las mejoras (que aborden distintos tipos de vulnerabilidades: al menos una de validación/inyección, una de autenticación/autorización, y una de otra categoría como XSS, configuración, etc.), la **correcta aplicación técnica** en el código (que las soluciones sean efectivas y sigan las recomendaciones vistas en clase), y la **claridad de la documentación** entregada. Asimismo, se tendrá en cuenta la creatividad para identificar problemas no evidentes y resolverlos, y la capacidad de trabajar en equipo gestionando la seguridad en un proyecto existente. No es requisito que la aplicación quede 100% "blindada" o perfecta, pero sí que se evidencie un progreso significativo en seguridad gracias a las acciones realizadas por el equipo.

Con esta actividad, los estudiantes pondrán en práctica el ciclo de vida de desarrollo seguro en pequeña escala: identificando amenazas, planificando controles, implementando y verificando su efectividad. Al finalizar, habrán experimentado de primera mano cómo **mejorar la seguridad de una aplicación web** aplicando tanto principios generales (S-SDLC, OWASP Top 10) como soluciones técnicas específicas en las tecnologías que utilizan día a día, fortaleciendo sus competencias para desarrollar software de forma profesional y segura.

Bibliografía y recursos recomendados:

[1]. Allen, J., et al. *Software Security Engineering: A Guide for Project Managers*. Addison-Wesley, 2008. (Guía práctica desde el CERT sobre cómo integrar seguridad en proyectos de software).

[2]. OWASP Foundation. *OWASP Top 10:2021 – The Ten Most Critical Web Application Security Risks*. (Documento en línea).

[3]. OWASP Foundation. *OWASP Proactive Controls* (Controles proactivos de seguridad para desarrolladores).

[4]. Souppaya, M., et al. NIST SP 800-218, *Secure Software Development Framework (SSDF) Version 1.1*. NIST, 2022. (Marco de prácticas de desarrollo seguro recomendado por NIST).

[5]. Red Hat. **"Seguridad en el ciclo de vida de desarrollo del software"** (Artículo en línea).

[6]. Imperva. **"Secure Software Development Life Cycle (SSDLC)"** – Imperva Learning Center (Artículo en línea).

[7]. Howard, M. & Lipner, S. *The Security Development Lifecycle*. Microsoft Press, 2006. (Descripción del SDL de Microsoft, uno de los primeros procesos formales de desarrollo seguro).

[8]. ISO/IEC. *ISO/IEC 27001:2022* – **Anexo A.8 (anterior A.14)**: Controles de seguridad para adquisición, desarrollo y mantenimiento de sistemas.

[9]. Sitio oficial de Angular – *Guía de Seguridad* (Angular Security Guide).

[10]. Sitio oficial de React – *React DOM Purify Example* (ejemplos de uso seguro de dangerouslySetInnerHTML).

[11]. Documentación de NestJS – *Security* (capítulo sobre seguridad, que incluye uso de Helmet, CORS, rate limiting, serialización segura, etc.).

[12]. OWASP Zed Attack Proxy (ZAP) – Herramienta gratuita para escanear vulnerabilidades web.

[13]. PortSwigger Web Security Academy – Laboratorios en línea gratuitos para practicar explotación de vulnerabilidades (incluyendo XSS, SQLi, etc., con explicaciones de soluciones).