Modulo 4: Seguridad de Software

Lección 1: Gestión de Vulnerabilidades Críticas y Pruebas de Seguridad en el Ciclo de Vida del Software

Objetivos de la Lección

Al finalizar esta lección, el estudiante podrá:

1. **Identificar** vulnerabilidades críticas de software que afectan la seguridad del sistema.
2. **Analizar** los impactos sobre la Confidencialidad, Integridad y Disponibilidad (CIA).
3. **Aplicar** pruebas unitarias, de integración y dinámicas para detectar fallos tempranos.
4. **Implementar** buenas prácticas de corrección, parcheo y programación defensiva.

Introducción a la Lección

Las vulnerabilidades de software representan una de las mayores amenazas para la ciberseguridad moderna. Errores de diseño o implementación —como los **buffer overflows**, **inyecciones de código** o **condiciones de carrera**— pueden ser explotados para comprometer datos, servicios o sistemas completos.

Sin embargo, no solo los fallos técnicos crean riesgos: prácticas inseguras como los **backdoors ocultos** o los **parches apresurados (patch-and-fix)** también introducen debilidades difíciles de detectar.

Por ello, una **gestión integral de vulnerabilidades** requiere no solo conocer las fallas comunes, sino también aplicar **pruebas sistemáticas de seguridad** (SAST, DAST, fuzzing) y **procesos de corrección disciplinados**, alineados con los principios del *Secure Software Development Framework (NIST SP 800-218)* y las guías OWASP.

Desarrollo del Tema

Principales Vulnerabilidades Críticas

1. Buffer Overflow (Desbordamiento de Búfer)

Descripción:

Ocurre cuando un programa escribe más datos en una zona de memoria (búfer) de la que tiene asignada. Al sobrepasar los límites, se sobrescriben partes de memoria que podrían contener direcciones de retorno o variables críticas, lo que permite al atacante ejecutar código malicioso.

Es un fallo clásico en lenguajes como C y C++, aunque puede simularse en otros entornos.

Ejemplo:

Un programa que reserva 8 bytes de memoria para un texto, pero intenta copiarle 20 bytes. Los bytes adicionales corrompen memoria cercana, lo que podría alterar la dirección de retorno y ejecutar código arbitrario.

Impacto CIA:

- **Confidencialidad:** el atacante puede leer información sensible desde la memoria.
- **Integridad:** puede modificar valores internos o instrucciones.
- **Disponibilidad:** el sistema puede colapsar (crash).

Mitigaciones:

- **ASLR (Address Space Layout Randomization):** aleatoriza las direcciones de memoria, dificultando predecir dónde inyectar código.
- **Stack Canaries:** valores especiales en la pila que, si se alteran, detienen la ejecución antes de que el ataque tenga éxito.
- **AddressSanitizer:** herramienta que detecta accesos indebidos a memoria durante las pruebas.

2. Inyección de Código (Code Injection)

Descripción:

Consiste en insertar comandos o código malicioso dentro de una entrada del usuario que luego es ejecutada por el sistema. Esto ocurre cuando la aplicación no valida o limpia los datos antes de procesarlos.

El atacante aprovecha este error para modificar consultas, ejecutar comandos o acceder a datos no autorizados.

Ejemplo:

Una consulta SQL que concatena directamente lo que escribe el usuario:

SELECT * FROM users WHERE name = 'usuario';

Si no se valida, el atacante puede escribir ' OR 1=1-- y obtener todos los registros.

Impacto CIA:

- **Confidencialidad:** se filtran datos privados.
- **Integridad:** se alteran registros o configuraciones.
- **Disponibilidad:** en ataques de gran escala, puede inutilizar la base de datos.

Mitigaciones:

- **Sentencias Parametrizadas:** separan los datos del código, impidiendo que el texto del usuario se ejecute como comando.
- **Escapado de Entradas:** limpiar caracteres peligrosos antes de procesar (shlex.quote, html.escape).
- **WAF (Web Application Firewall):** detecta patrones maliciosos y bloquea solicitudes inusuales.
- **Principio de Privilegio Mínimo:** la cuenta de base de datos usada por la aplicación debe tener permisos estrictamente necesarios (solo lectura, por ejemplo).

3. Condiciones de Carrera (Race Condition)

Descripción:

Sucede cuando dos o más procesos o hilos acceden simultáneamente a un recurso compartido (variable, archivo, base de datos) sin la debida sincronización.
El resultado depende del orden en que se ejecuten las operaciones, lo que puede causar datos inconsistentes o pérdidas de integridad.

Ejemplo:

Dos hilos intentan retirar dinero del mismo balance. Ambos leen “100”, restan 100 y escriben el resultado “0” dos veces, generando un saldo final erróneo o negativo.

Impacto CIA:

- **Integridad:** los datos pierden consistencia.
- **Disponibilidad:** el servicio puede colapsar si se bloquea por error.
- **Confidencialidad:** en algunos casos, puede usarse para acceder a información antes de que se valide.

Mitigaciones:

- **Locks o Mutex:** bloquean el recurso mientras un hilo lo usa, evitando accesos simultáneos.
- **Operaciones Atómicas:** aseguran que una operación completa no pueda interrumpirse a mitad de ejecución.
- **Pruebas de Concurrencia:** herramientas como *ThreadSanitizer* ayudan a detectar estas condiciones durante el desarrollo.

4. TOCTOU (Time-Of-Check to Time-Of-Use)

Descripción:

Es una vulnerabilidad de sincronización que ocurre cuando un programa revisa una

condición (como permisos de un archivo) y luego, en otro momento, utiliza ese mismo recurso asumiendo que no ha cambiado.
Un atacante puede aprovechar ese intervalo de tiempo para reemplazar o modificar el recurso verificado.

Ejemplo:
Un programa comprueba si puede escribir en /tmp/data y luego abre el archivo. Si el atacante cambia /tmp/data por un enlace simbólico hacia /etc/passwd, el programa podría sobrescribir archivos críticos del sistema.

Impacto CIA:

- **Integridad:** alteración de archivos del sistema.
- **Confidencialidad:** acceso a información restringida.
- **Disponibilidad:** daño o bloqueo de archivos necesarios.

Mitigaciones:

- **Operar con Descriptores Seguros:** usar funciones como openat2() o O_NOFOLLOW para acceder al recurso sin depender de su nombre de ruta.
- **Directorios Exclusivos:** trabajar en rutas donde solo el proceso tenga permisos.
- **Revalidación:** verificar de nuevo el recurso justo antes de usarlo.

5. Backdoors (Puertas Traseras)

Descripción:
Un backdoor es una vía oculta, intencional o accidental, que permite el acceso no autorizado a un sistema sin pasar por los controles normales de autenticación.
Puede estar en el código fuente, en un firmware o incluso en hardware comprometido.

Ejemplo:

Una función secreta en un programa que responde a la URL /debug?token=admin y devuelve información sensible del sistema.

Impacto CIA:

- **Confidencialidad:** expone datos sin registro en logs.
- **Integridad:** permite modificar configuraciones internas.
- **Disponibilidad:** el atacante puede tomar control del sistema.

Mitigaciones:

- **Revisión de Código Externa:** todo cambio debe revisarse por al menos dos desarrolladores no autores del código.
- **Escaneo de Secretos:** herramientas como *TruffleHog* o *Git-Secrets* buscan credenciales embebidas.
- **Firma Digital del Código y Firmware:** garantiza que los binarios no fueron alterados tras la compilación.
- **Secure Boot / TPM:** evita ejecutar software no verificado al iniciar el sistema.

6. Patch-and-Fix (Parches Reactivos o Improvisados)

Descripción:

Es un error de gestión que ocurre cuando los desarrolladores aplican correcciones rápidas sin analizar la causa raíz del problema. Esto suele provocar nuevos fallos o incluso reabrir la vulnerabilidad original.

Este patrón refleja una cultura de reacción en lugar de prevención.

Ejemplo:

Tras descubrir una vulnerabilidad, un parche apresurado soluciona solo un vector, dejando otros caminos abiertos (como ocurrió con *PrintNightmare* en Windows 2021).

Impacto CIA:

- **Confidencialidad, Integridad y Disponibilidad:** pueden verse comprometidas si el parche introduce un nuevo fallo.

Mitigaciones:

- **Gestión de Versiones Segura:** aplicar los parches en ramas separadas (hotfix/) y fusionarlos solo tras validación.
- **Pruebas de Regresión:** ejecutan automáticamente escenarios previos para confirmar que no se rompió ninguna función existente.
- **Revisión Postmortem (5 Whys):** analizar la raíz del problema y mejorar el proceso, no solo el código.
- **Despliegue Controlado (Canary Release):** liberar el parche a un grupo pequeño antes de implementarlo globalmente.

Análisis de Riesgos y Daños

Cada vulnerabilidad debe analizarse según:

- **Probabilidad de explotación** (alta, media, baja).
- **Impacto CIA** (qué compromete).
- **Vector de ataque** (local, remoto, cadena de suministro).

Ejemplo: Un overflow en un servicio web crítico con acceso remoto puede tener un **riesgo CVSS ≥ 9.8**, requiriendo parche inmediato (< 24 h).

Pruebas de Seguridad y sus Limitaciones

Tipo de Prueba	Objetivo	Herramientas Python	Limitaciones
Unidad	Validar funciones individuales.	pytest, unittest	No detecta errores de integración.
Integración	Comprobación entre módulos o APIs.	pytest + docker-compose	Ambientes simulados pueden ocultar fallas.
Sistema (E2E)	Validar el sistema completo.	Selenium, Playwright	Alto costo; requiere entorno real.
Estática (SAST)	Revisar el código sin ejecutarlo.	bandit, semgrep	No detecta errores lógicos.
Dinámica (DAST)	Pruebas durante ejecución.	OWASP ZAP, sqlmap	Cobertura limitada.
Fuzzing	Entradas aleatorias para detectar fallos ocultos.	Atheris, fuzzingbook	Alto consumo de CPU.

Buenas Prácticas de Corrección y Pruebas

1. **Programación defensiva:** uso de `assert`, validaciones de entrada, manejo de excepciones.
2. **Control de versiones seguro:** crear ramas `hotfix/` para aplicar parches sin romper el código base.
3. **Automatización de pruebas:** integración de `pytest` o `unittest` en pipelines CI/CD.
4. **Revisión de código:** aplicar el principio de “mínimo dos revisores externos”.

5. **Validación de correcciones:** ejecutar pruebas de regresión y escaneo DAST antes del despliegue.

Este ejemplo muestra cómo una **validación simple y pruebas unitarias automatizadas** ayudan a prevenir errores de lógica o parches defectuosos antes de su despliegue.

Relación con Otros Conceptos

- **Programación Defensiva y Pruebas:** vincula los conceptos de diseño seguro (principios de *least privilege*, *complete mediation*, *mutual suspicion*).
- **Análisis de Riesgos:** conecta con los métodos del módulo anterior, aplicando la CIA para priorizar vulnerabilidades.
- **Ciclo de Vida del Software Seguro (SSDLC):** integra la seguridad desde la codificación hasta el mantenimiento, siguiendo NIST SP 800-218 y OWASP.

Resumen de la Lección

En esta lección unificada se integran los aspectos **técnicos y procesales** de la seguridad de software. Se estudiaron las principales vulnerabilidades críticas, sus vectores de ataque y contramedidas. Asimismo, se destacaron las **pruebas automatizadas**, la **programación defensiva** y la **gestión disciplinada de parches**, como pilares para mantener un desarrollo confiable y seguro.

Conclusión: La seguridad del software no depende solo del código, sino de una cultura de pruebas, revisión y mejora continua en cada fase del ciclo de vida.

Actividad de la Lección — "Auditoría Segura y Corrección Controlada"

Escenario:

Se entrega un microservicio Flask con las siguientes vulnerabilidades:

- Inyección SQL simulada.
- Backdoor en /debug?token=admin.

- Error de tipo “patch-and-fix” que rompe la función principal.

Tareas:

1. **Detección:** ejecutar bandit y OWASP ZAP para identificar las fallas.
2. **Corrección:** eliminar el backdoor y reparar el código con validaciones.
3. **Pruebas:**
    - Crear pruebas unitarias y de integración (pytest).
    - Confirmar que /debug devuelva 404.
4. **Reporte:** redactar un documento REPORT.md con:
    - Descripción del hallazgo y riesgo CIA.
    - Capturas de pruebas.
    - Justificación de controles aplicados.

Referencias

- Pfleeger, C. P., Pfleeger, S. L., & Coles-Kemp, L. (2023). *Security in Computing* (6ª ed.). Addison-Wesley.
- Du, W. (2022). *Computer Security: A Hands-on Approach* (3ª ed.). Independently published.
- OWASP (2025). *Code Review Guide v2* y *Testing Guide v5*.
- NIST SP 800-218 (2023). *Secure Software Development Framework (SSDF)*.
- Google SRE (2022). *Testing for Reliability*.