Módulo 4: Seguridad de software

Lección 2: Principios de Ingeniería de Software Segura

Objetivos de la Lección

Al finalizar, el estudiante podrá:

1. **Aplicar buenas prácticas de codificación segura y validación robusta** en el desarrollo de módulos y microservicios.
2. **Implementar mecanismos de sandboxing y gestión segura de secretos** para fortalecer la protección del entorno de ejecución.
3. **Diseñar flujos de acceso y comunicación entre servicios aplicando sospecha mutua y privilegio mínimo.**
4. **Evaluar y reforzar la mediación completa y la separación de privilegios** en arquitecturas seguras mediante prácticas CI/CD y control de acceso.

Introducción a la Lección

Un algoritmo criptográficamente robusto sirve de poco si el código que lo implementa carece de defensas esenciales. Los *principios de ingeniería de software segura* son reglas atemporales que—cuando se aplican disciplinadamente—reducen la superficie de ataque y aumentan la resiliencia. En esta sesión recorreremos las mejores prácticas de codificación, la sospecha mutua entre componentes, el confinamiento mediante

sandboxing, la minimización y separación de privilegios y la mediación completa; veremos cómo se combinan para construir software confiable desde su concepción.

Desarrollo del Tema

Coding Best Practices

- **Validación robusta de entradas**
  - Es una **técnica esencial de seguridad en el desarrollo de software** que busca **verificar y limpiar (sanitizar)** todos los datos que recibe un sistema antes de procesarlos. Su propósito principal es **evitar que datos maliciosos o inesperados** provoquen vulnerabilidades como **inyección SQL, XSS (Cross-Site Scripting), desbordamientos o ejecución de comandos**.
  - **Principios de Validación Robusta**
    - **Lista blanca –** Solo permite valores que cumplan con reglas definidas. Por ej. Aceptar solo caracteres de letras en el nombre de usuario)
    - **Lista negra** – Se rechazan aquellos valores con patrones peligrosos. Por ej. Comandos Bloquear script
    - **Validar en la Frontera (Edge Validation) –** La validación debe hacerse **justo donde los datos entran** al sistema, por ejemplo, en el endpoint HTTP.
    - **Esquemas de datos (creación y diseño) –** Definir reglas de validación mediante librerías (Python: pydantic, marshmallow). Por ej. Tipos, rangos, formatos de email o longitud máxima.
    - **Ejemplo en Python**

```python
from pydantic import BaseModel, constr, ValidationError

class Comment(BaseModel):
    user: constr(strip_whitespace=True, min_length=3, max_length=20, regex=r"^[A-Za-z0-9_]+$")
    text: constr(strip_whitespace=True, max_length=500)

try:
    data = Comment(user="admin<script>", text="Hola mundo")
except ValidationError as e:
    print("Entrada inválida:", e)
```

Imagen1. Este código **rechaza** entradas con caracteres no permitidos y **limita** la longitud de los textos. Así se evita que se inyecten scripts o comandos.

El *regex=r"^[A-Za-z0-9_]+$"* Siginifica el conjunto de caracteres permitidos:

- `A-Z: letras mayúsculas`
- `a-z: letras minúsculas`
- `0-9: números`
- `_: guion bajo`

- **Principio KISS (“Keep It Simple & Secure”)**
  - Mantener un código simple y seguro (*la complejidad en un código es enemiga de la seguridad*). La complejidad dentro de un código puede traer consigo probabilidad de errores y vulnerabilidades.

- **Gestión segura de secretos**
  - **¿Qué es Secreto de Software?**
    - Un **secreto** es cualquier **dato confidencial** que una aplicación necesita para funcionar, como:
      1. Contraseñas (de base de datos, APIs, correo, etc.)
      2. Claves API (Google, OpenAI, AWS…)
      3. Tokens de autenticación
      4. Certificados o claves privadas
    - Si estos **Secretos** quedan **expuestos** un atacante puede acceder a sistemas críticos o/y robar datos.
  - **Principio *12-Factor App* (Twelve Factor APP)**
    - Este es una serie de mejores prácticas para construir aplicaciones seguras, eficientes y portables. Por ej. Uno de estos principios menciona “*Configurar la aplicación mediante el entorno, no dentro del código*”

```python
# Mala práctica: secreto dentro del código
DB_PASSWORD = "admin123"

# Secreto almacenado en el entorno del sistema
import os
DB_PASSWORD = os.getenv("DB_PASSWORD")

#En el servidor se define como:
```

Imagen2. Ej. (Python) Mostrando parte de los 12 principios *Twelve Factor App*

- **Gestor de Secretos:**
  - Servicios que guardan y protegen secretos cifrados, tales como:
    1. **AWS Secrets Manager**
    2. **HashiCorp Vault**
    3. **Azure Key Vault**
    4. **Google Secret Manager**

- **Errores controlados y observabilidad**
  - Todo error debe de ser **detectado**, **registrado** y **reportado,** pero sin exponer información sensible a los end points (Usuario final)**.**
  - **Registrar *solo* lo necesario:** Según el **GDPR (Reglamento General de Protección de Datos)**, solo se debe registrar la **mínima información necesaria** para fines de seguridad o auditoría. No se deben guardar datos personales directamente (como emails o nombres), sino versiones **anonimizadas o con hash.**

- **CI/CD y herramientas de soporte**
  - **CI/CD (Continuous Integration / Continuous Deployment)** es un conjunto de prácticas y herramientas que **automatizan el proceso de construcción, prueba y despliegue de software**.
    - **Su objetivo es:**
      1. Detectar errores a nivel preventivo
      2. Asegura calidad y seguridad en el código.

3. Entrega actualizaciones periódicas sin interrumpir un servicio en particular.

- **Ciberseguridad:** el pipeline CI/CD también actúa como **una capa de defensa**, verificando que el código **cumpla estándares de seguridad antes de ser publicado.**
- **Componentes Principales:**
  1. *Git hooks* pre-commit: Son **scripts automáticos** que se ejecutan **antes de confirmar (commit) los cambios en Git**. Sirven para **detener código inseguro o mal formateado** antes de que llegue al repositorio.
  2. Herramienta recomendada:
     - *Bandit:* Analiza el código fuente buscando vulnerabilidades (e.g. eval(), contraseñas hardcodeadas). La siguiente imagen muestra cómo se podría utilizar *bandit* con un repositorio en lenguaje Python en GitHub. Esto se realiza antes de ejecutar *commit* en Git (local de tu ordenador).

```
- repo: https://github.com/PyCQA/bandit
  rev: 'latest'
  hooks:
    - id: bandit
```

Imagen3. Cada vez que el usuario haga o ejecute git commit, se ejecutará **Bandit** automáticamente para revisar el código.

- ***Supply-chain (Seguridad en la cadena de Suministro)*:** La **cadena de suministro del software** incluye todas las **dependencias externas** (bibliotecas, frameworks, módulos de terceros). **El objetivo** es **asegurar que las dependencias no tengan vulnerabilidades** ni versiones comprometidas.
  - **Herramientas Comunes:**

1. **Dependabot –** Analiza dependencias (GitHub) y sugiere actualizaciones seguras.
2. **Renovate –** Actualiza librerías automáticamente.
3. **pip-audit –** Escanea paquetes Python (pip) y verifica vulnerabilidades conocidas (CVE).

- **Ejemplo de pip-audit:**

```
pip install pip-audit
pip-audit
```

Imagen4. Para el pasado ejemplo, con el comando (pip-audit) se muestra si alguna librería (por ejemplo, flask==1.1.0) tiene vulnerabilidades y qué versión segura se debería usar.

- **Pipeline con políticas de bloqueo (Quality Gates)**

  - **Pipeline CI/CD** ejecuta pruebas automáticas antes de aprobar un despliegue. Para seguridad, se configuran **condiciones mínimas obligatorias** (gates):

    1. **Cobertura de pruebas ≥ 80 %** → asegura que la mayoría del código esté probado.
    2. **Bandit: sin vulnerabilidades MEDIUM+** → si detecta una vulnerabilidad media o alta, **el despliegue se detiene automáticamente.**

Ejemplo Python — Sanitización, límites y registro estructurado

Ver documento adjuntado en su blackboard junto con la lección 2 dentro de este módulo.

Sospecha Mutua (Mutual Suspicion)

- **Concepto**: significa que **cada componente del sistema (microservicio, módulo o API)** debe **actuar como si los demás pudieran estar comprometidos o fallar.** En otras palabras, ningún servicio debe **confiar ciegamente** en la información que recibe de otro, aunque pertenezca al mismo sistema.
- **Medios de aplicación**

Medida	Definición	Ejemplo práctico
1. Autenticación y firma interna	Toda comunicación entre microservicios debe incluir un **token JWT (JSON Web Token) firmado (HS256 o RS256)**. Cada servicio valida los campos aud (audiencia), exp (expiración) e iss (emisor).	El servicio A envía una petición al servicio B con un JWT firmado. Si el token está expirado o manipulado, B **rechaza la solicitud**.
2. Rate limiting entre servicios	Evita que un microservicio envíe demasiadas peticiones (ataque DDoS interno).	Configurar **NGINX o Envoy** con global_throttle para limitar a, por ejemplo, 100 peticiones por segundo entre servicios.
3. Contratos explícitos (OpenAPI/IDL)	Define un esquema de datos estricto (por ejemplo, usando OpenAPI o gRPC IDL). Si un servicio recibe un campo no esperado o tipo incorrecto → se rechaza la llamada.	Microservicio “Users” solo acepta {id:int, email:string}. Si recibe {id:"abc"} o un campo adicional, **rompe la llamada**.
4. Circuit breakers (resiliencia)	Mecanismo para **cortar el flujo** hacia un servicio que falla continuamente, evitando que el error se propague al resto del sistema.	Librerías como **pybreaker** en Python o **Hystrix** en Java interrumpen temporalmente llamadas al servicio defectuoso.

- **Caso de uso**
    - Microservicio *Invoice* confía en cabecera `X-User-Role`; un atacante la inyecta como `admin`.
    - Solución: el *Gateway* firma la cabecera (`HMAC`), *Invoice* verifica la firma, cualquier cabecera modificada se descarta.

Confinamiento (Sandboxing) [What is a Sandboxing?]

Ejecutar código o procesos en un **entorno aislado y con recursos/permisos limitados**, de modo que, si ese código falla o está comprometido, el daño quede contenido y no afecte al sistema global.

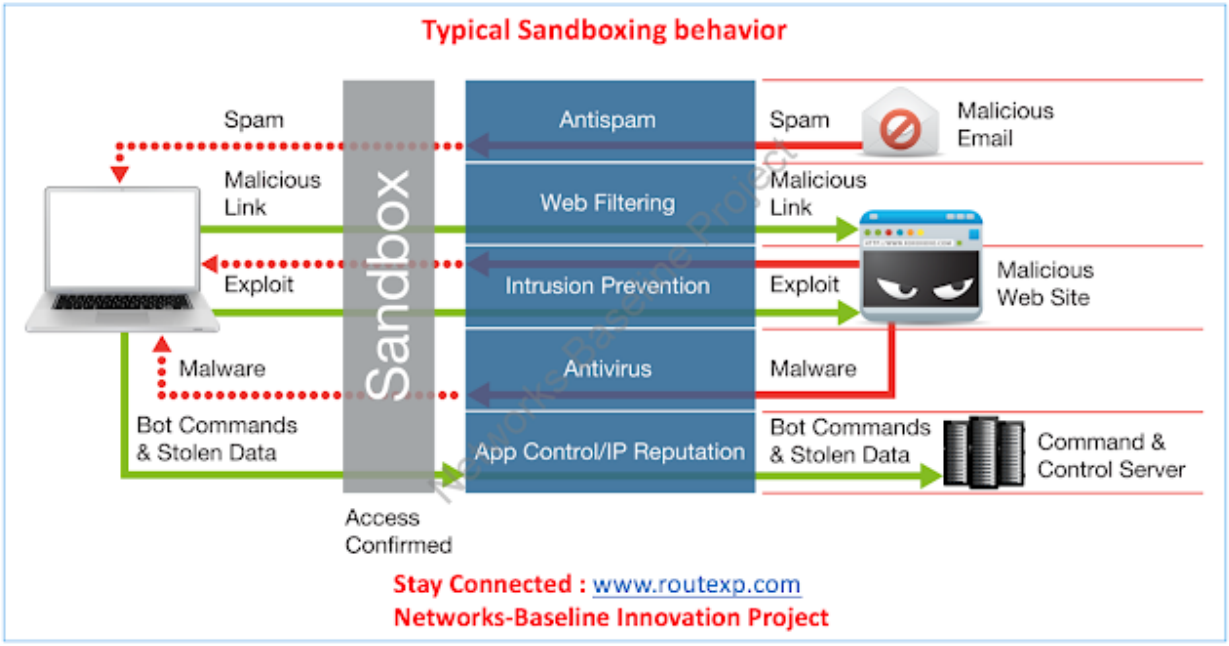

Imagen5. El sandbox decide si el archivo o enlace puede pasar al usuario o si se bloquea antes de llegar a él.

- **Técnicas Comunes**

    1. **Contenedores (Docker) — aislamiento por namespaces + cgroups**:

        - **Docker usa**:

- **namespaces** (PID, mount, net, user, ipc) → aísla visibilidad de procesos, red y sistema de archivos.
- **cgroups** → limita CPU, memoria, I/O.
- **capabilities** → granulariza privilegios en lugar de ser root absoluto.
- **seccomp, AppArmor/SELinux** → control de syscalls y políticas LSM.
- Ejemplo con **Docker:**

```
docker run \
  --read-only \
  --cap-drop ALL \
  --user 1001 \
  Myimage:latest
```

Imagen6. Ejecuta el contenedor **con sistema de archivos solo lectura**, **sin privilegios especiales** y **como usuario no root**. Esto evita que el proceso **modifique el sistema** o **eleve permisos**, manteniendo el contenedor **aislado y seguro**.

2. **Filtro de sistema de archivos en navegadores — Site Isolation**
   - En navegadores modernos como (Firefox y Chrome):
     - Cada pestaña o *renderer process* corre en un proceso separado, con políticas que limitan el acceso al sistema de archivos y recursos.
     - **Site Isolation** aísla páginas de distintos orígenes en procesos distintos para mitigar ataques tipo Spectre/Rowhammer y XSS que intenten robar datos de otras pestañas.

- ❖ Usan políticas como **Content-Security-Policy (CSP)** para bloquear ejecución de scripts no autorizados.

Privilegio Mínimo

Asignar a un proceso solo los derechos estrictamente necesarios y durante el tiempo indispensable.

- **Reglas concretas**
  - Definir funciones IAM “por caso de uso”, no “por departamento”.
  - *Just-in-time* privilege: Significa que **las cuentas administrativas solo obtienen privilegios elevados durante el tiempo exacto necesario para ejecutar una tarea** y luego **pierden esos permisos automáticamente**.
    - ***Ejemplo:***

Un administrador necesita reiniciar un servidor o aplicar un parche → Solicita elevación temporal (por 15 min, por ejemplo) → Termina la tarea → el sistema **revoca automáticamente** los privilegios.

- **Herramientas**
  - **gVisor/auditd** para monitorizar syscall y descubrir exceso de privilegios.
  - **OPA (Open Policy Agent) Gatekeeper** en Kubernetes: regla `containers[image].runAsNonRoot == true`.

Separación de Privilegios

Dividir tareas sensibles entre componentes (partes del sistema o del software) o roles para evitar que un único fallo conceda control total.

- **Doble aprobación**: módulo que genera pagos y módulo separado que los autoriza.
- **Split tokens**: firma digital en dos HSM (**Hardware Security Module**); se necesita ambos para firmar firmware.

- **Modelos comunes**
    - **Two-man rule**: firmar firmware. Ej. aprobar transferencias bancarias > $1 M.
    - **Micro-kernel**: los servicios importantes del sistema operativo (como red o archivos) **no se ejecutan dentro del núcleo**, sino **en procesos separados (espacio de usuario)** con **privilegios reducidos**.
        - Ejemplo: En **QNX** o **seL4**, si el servicio de red (network) sufre una vulnerabilidad, el sistema sigue estable porque ese servicio corre **fuera del núcleo**, aislado. (**QNX** y **seL4** son sistemas operativos con microkernel: QNX se usa en autos por su estabilidad, y seL4 en defensa por su seguridad verificada.)
    - **Front-end / back-end**: UI no toca base de datos directamente; backend expone sólo procedimientos seguros. (**Entiéndase** que el **Front-End** es lo que ves y usas; el **Back-End** es lo que hace que todo funcione detrás.)
- **Ejemplo**
    - Aplicación de pagos: microservicio *Payment-Writer* con token de **firma**; microservicio *Payment-Reader* con token solo **lectura**. Ninguno posee ambos privilegios.

Mediación Completa

Cada acceso a un recurso debe pasar por un control; no confiar en cachés o resultados previos.

- **Buena práctica**: cada solicitud verifica firma JWT y caducidad (`exp`).
- **Ejemplo**: `open()` se llama cada vez que se necesita el archivo, no se mantiene descriptor global si los permisos pudieran cambiar.
- **Aplicación a archivos**
    - Abrir archivos con `O_PATH + openat2()` para revalidar ruta y permisos.
- **Aplicación a API**
    - Comprobar permiso **en cada request** incluso si la sesión es ya autenticada: `hasScope("invoice:read")`.

- **Caso extendido**
    - En sistemas UNIX, `sudo` re-verifica la política para CADA comando (mediación completa) y mantiene un timestamp corto (5 min) para UX (mantiene una sección abierta por 5min).

Otros Principios Relevantes

- **Defensa en profundidad** — capas redundantes de seguridad. Aplicar *journaling* + RAID para fallas de disco (integridad), TLS + WAF + IPS para tráfico (confidencialidad).
- **Fail-safe defaults** — la opción por omisión es denegar (`DROP` como política de firewall). Políticas de firewall, cortafuegos de base de datos (`REVOKE ALL` y luego `GRANT` selectivo).
- **Economía de mecanismos** — mantener el diseño lo más simple posible para facilitar su auditoría. Microservicio con una sola responsabilidad: reduces superficie, haces auditoría más fácil.
- **Registro y auditoría** — logs inmutables y firma de registros para detección y atribución de incidentes.
- **Usabilidad segura** – feedback claro (“contraseña inválida” genérico) sin filtrar si el usuario existe o no.

Relación con Otros Conceptos

- Estos principios se apoyan en las **pruebas de unidad/integración** vistas en la lección anterior: no se puede aplicar *least privilege* si las dependencias no están desacopladas y testeadas.
- La **mediación completa** refuerza los modelos de **control de acceso** (DAC/MAC/RBAC/ABAC) y evita brechas de sesión.
- **Sandboxing** y **privilegio mínimo** limitan el radio de explosión de vulnerabilidades tipo *buffer overflow*.

Resumen de la Lección

Adherirse a buenas prácticas de codificación, asumir desconfianza entre módulos, encerrar código riesgoso en sandboxes, aplicar mínimo y separación de privilegios y exigir mediación completa en cada solicitud son pilares que mantienen la seguridad aun cuando surjan vulnerabilidades inevitables. Estos principios, combinados, constituyen la cultura de una ingeniería de software verdaderamente segura.

Actividad de la Lección — “Refactor Seguro en 90 min”

1. Se distribuye un pequeño proyecto Flask con:
    - Ruta `/admin` protegida solo por cookie de sesión (sin verificación de rol).
    - Script Python que ejecuta `os.system("ping " + host)` con entrada del usuario.
2. En parejas:
    - **Aplicar** principio de *least privilege*: crear rol `admin` y verificarlo en cada request (*mediación completa*).
    - **Encerrar** el comando `ping` en un contenedor Docker con capacidades mínimas y whitelisting de host (`sandboxing`).
    - **Agregar** validación de lista blanca al parámetro `host` (good coding practice).
3. Escribir `README_secure.md` explicando qué principio se aplicó en cada cambio y cómo se prueban.
4. Entregar PR en GitHub; CI ejecutará `pytest` y un contenedor OWASP ZAP para validar que ya no hay inyección de comandos.

Referencias Adicionales

- Pfleeger, C. P., Pfleeger, S. L., & Coles-Kemp, L. (2023). *Security in Computing* (6.ª ed.).
- Bishop, M. (2018). *Computer Security: Art and Science* (2.ª ed.). Addison-Wesley.
- OWASP. *Secure Coding Practices Checklist* (2025).
- NIST SP 800-218 (2023). *Secure Software Development Framework (SSDF)*.

- Google. *BeyondProd: Zero Trust for Web Services* (whitepaper, 2022).