Módulo 3: Defensas, controles y contramedidas

Lección 3: Criptografía Aplicada y Programación Defensiva

Objetivos de la Lección

Al terminar la sesión, el estudiante podrá:

1. **Explicar** las diferencias entre cifrado simétrico y asimétrico, así como los conceptos de cifrado en bloque y en transmisión.
2. **Aplicar** firmas digitales usando claves públicas y certificados para garantizar autenticidad, integridad y no-repudio.
3. **Integrar** principios de programación defensiva en el ciclo de desarrollo para prevenir vulnerabilidades comunes.

Introducción a la Lección

La criptografía protege los pilares de la seguridad (CIA) y la programación defensiva minimiza la superficie de ataque en el código. Comprender cómo implementar correctamente algoritmos de cifrado—y escribir código que asuma fallas—es esencial para desarrollar sistemas robustos. Esta lección combina teoría criptográfica con prácticas de desarrollo seguro, creando un puente entre “matemáticas” y “código real”.

Desarrollo del Tema

Encriptación en bloque y en transmisión

- **Algoritmos Estándar: AES, SM4, Camellia.**
    - **AES** – De acuerdo con whitestack (2024), **el cifrado AES (Advanced Encryption Standard)** es una técnica de seguridad que transforma datos legibles en un formato codificado, de manera que solo puedan ser recuperados por quien tenga la clave correcta. Se basa en algoritmos matemáticos fuertes y se usa para proteger información sensible tanto almacenada como en tránsito, evitando accesos no autorizados. **Mas información en: ¿Cómo funciona el cifrado AES?**
    - **SM4 -** Es un algoritmo de cifrado simétrico estándar chino, aprobado por el gobierno de China para proteger comunicaciones oficiales. Usa bloques de 128 bits y una clave de 128 bits. Se emplea en protocolos chinos de seguridad como WAPI (**Wireless LAN Authentication and Privacy Infrastructure**) y en banca digital.
    - **Camellia** – es un cifrado japonés alternativo a AES, reconocido internacionalmente y usado en redes seguras (ej. HTTPS, VPNs).
- **Cifrado en bloque:** procesa datos en bloques fijos (p. ej., AES-128 bits usa 16 bytes; porque **1 byte = 8 bits** → 128 ÷ 8 = 16). **Modos comunes:**
    - **CBC (Cipher Block Chaining)** – Es un modo de operación de cifrado por bloques (usado con AES, por ejemplo). Funciona encadenando cada bloque cifrado con el anterior. Para que el primer bloque no siempre empiece igual, CBC usa un IV (Initialization Vector) en el cual debe de ser de 128 bits esto gracias a **AES**. Esto evita que dos mensajes iguales produzcan siempre el mismo resultado cifrado.
    - **CTR (Counter)** – genera un flujo de claves (keystream) usando un contador (counter). Cada bloque del contador se cifra con una clave, luego produce un bloque de keystream y después se combina con el texto original usando la operación Logica **XOR**.
    - **GCM** (**Galois/Counter Mode**) – Es un modo de cifrado AEAD (Authenticated Encryption with Associated Data). Este tiene dos funciones

que se ejecutan al mismo tiempo los cuales son en que primeramente se cifran los datos (confidencialidad) y luego este genera un tag de autenticación (integridad y autenticidad).

- **Cifrado en transmisión (stream):** El **cifrado en transmisión** cifra los datos **de manera continua, bit a bit o byte a byte**, lo que lo hace ideal para **flujos en tiempo real** como llamadas, videoconferencias o streaming. Ej.: **ChaCha20**.
    - **Algunas Herramientas de Cifrado en Transmisión:**
        - **ChaCha20-Poly1305**
            - **ChaCha20** → un **cifrado de flujo (stream cipher)** muy rápido, diseñado por Daniel J. Bernstein, ideal para **CPU que no tienen aceleración AES**.
            - **Poly1305** → un **algoritmo de autenticación (MAC)** que garantiza **integridad y autenticidad** de los datos.
                - Al combinarse, se obtiene **confidencialidad + autenticidad** en una sola operación.
        - **Salsa20, HC-256 –** alternativas académicas con buen rendimiento en hardware limitado (IoT).
    - **Ataque clásico:** Uno de los ataques al cifrado en transmisión (stream) lo es el reutilizar el *keystream* (nonce-reuse), esto produciría un XOR de dos textos claros, visible al adversario.
- **Errores de implementación más comunes**
    1. **IV predecible** (p. ej. `IV = 0x00...00`) ⇒ CBC susceptible a análisis estadístico.
    2. **Uso de RNG no criptográfico para claves/IV –** Un RNG (Random Number Generator) normal como random.random() en Python o java.util.Random en Java no es criptográficamente seguro. Estos están diseñados para simulaciones o estadísticas, no para seguridad.
        - Si se usan para generar **claves de cifrado** o **vectores de inicialización (IVs)**, un atacante puede **recrear la secuencia** de números y **romper el cifrado**. **Por ejemplo**: si un IV se repite o es predecible, expone patrones en el mensaje.

3. **Olvidar autenticar los datos** – Si usas **AES-CBC sin autenticación**, los datos pueden ser alterados por un atacante mediante **bit-flipping**. La solución es usar **cifrado autenticado (AEAD)** o añadir un **MAC (Message Authentication Code)**.
    - **Bit-flipping – Los ataques Bit-Flipping explotan la relación predecible entre el texto cifrado y el texto plano.** En modos como **CBC**, un atacante que no conoce la clave puede **modificar bits en el ciphertext**.

- **Buenas prácticas:**
    - Utilizar **AES-GCM (AES** protege datos, **GCM (Galois/Counter Mode)** garantiza **confidencialidad e integridad)** o **ChaCha20-Poly1305**.
    - Generar IV/nonce único por mensaje; no reutilizar clave+nonce.
        - Un **nonce** (*number used once*) es un **valor aleatorio o secuencial que se usa una sola vez** en criptografía para garantizar que cada operación de cifrado sea **única** y evitar ataques por repetición (**replay attacks**).

Ejemplo python

A) Cifrar Texto con AES-GCM

```
# Ejemplo: cifrado AES-GCM con la librería cryptography

# Importamos la clase AESGCM (AES en modo Galois/Counter)
# que permite cifrar y autenticar datos en una sola operación
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

# Librerías estándar:
# - os: para generar números aleatorios seguros (nonce)
# - base64: para mostrar el resultado cifrado en un formato legible
import os, base64
```

```
# Generamos una clave secreta AES de 128 bits (16 bytes). Esta clave solo se le ofrece al usuario receptor de manera confidencial, esto al igual con “nonce”
# Esta clave es la "llave" que servirá para cifrar y descifrar
key = AESGCM.generate_key(bit_length=128)

# Generamos un nonce (Number Used Once) de 12 bytes aleatorios
# GCM recomienda usar 12 bytes de longitud
# Sirve como "número único" para evitar repeticiones en el cifrado
nonce = os.urandom(12)

# Definimos el mensaje original (texto en claro) que queremos proteger
# La 'b' indica que es un string en formato de bytes
texto = b'Numero de tarjeta 4111-1111-1111-1111'

# Creamos un objeto AESGCM usando la clave secreta
# Este objeto sabe cómo cifrar y descifrar con AES-GCM
aesgcm = AESGCM(key)

# Ciframos el mensaje usando la clave y el nonce
# encrypt(nonce, texto, None) → devuelve el mensaje cifrado + tag de autenticación
# El "None" indica que no usamos datos adicionales (AAD) en este ejemplo
cifrado = aesgcm.encrypt(nonce, texto, None)

# Imprimimos el texto cifrado convertido a Base64 para hacerlo legible
print("Cifrado:", base64.b64encode(cifrado).decode())

# Desciframos el mensaje usando la misma clave y el mismo nonce
```

```
# Recuperamos el texto original (si el cifrado se hubiera manipulado, daría error)
claro = aesgcm.decrypt(nonce, cifrado, None)

# Mostramos el texto en claro recuperado
print("Claro:", claro.decode())
```

Bien importante mencionar que si queremos descifrar dicho mensaje se debe de tener, el usuario receptor, la clave "Key" y "Nonce".

B) Decifrar Texto

```
# se le añade el siguiente código, al principio del código en donde se importa "InvalidTag"

from criptography.exceptions import InvalidTag

# Luego se escribe los parámetros o el código para descifrar el texto

try:
   claro_bytes = aesgcm.decrypt(nonce, cifrado, None) # devuelve bytes
   claro_texto = claro_bytes.decode() # convertir bytes -> str (utf-8)
    print("Mensaje descifrado (directo):", claro_texto)
except InvalidTag:
    print("Error: autenticación fallida al descifrar (tag inválido).")
```

Imagenes de Cifrado y Decifrado con salida de consola en Visual Studio Code

Cifrado AES-GCM.py

```python
# Ejemplo: cifrado AES-GCM con la librería cryptography
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.exceptions import InvalidTag
import os, base64

key = AESGCM.generate_key(bit_length=128)
nonce = os.urandom(12)
texto = b'Numero de tarjeta 4111-1111-1111-1111'

aesgcm = AESGCM(key)
cifrado = aesgcm.encrypt(nonce, texto, None)
print("Cifrado:", base64.b64encode(cifrado).decode())
claro   = aesgcm.decrypt(nonce, cifrado, None)

#decifrado
try:
    claro_bytes = aesgcm.decrypt(nonce, cifrado, None)
    claro_texto = claro_bytes.decode()
    print("Mensaje descifrado (directo):", claro_texto)
except InvalidTag:
    print("Error: autenticación fallida al descifrar (tag inválido).")

```

```
Cifrado: q/cAV6+fwA+nqUqcB9ZFUY2U9zT+i/kwv+TK88RFfbdMFRL8AstIs/sLQe8XugREvaU11hI=
Mensaje descifrado (directo): Numero de tarjeta 4111-1111-1111-1111
```

Criptografía privada / simétrica

- **Definición:** una *misma clave* para cifrar y descifrar; clave debe permanecer secreta.
- **Algoritmos modernos:** AES-128/192/256, ChaCha20.
- **Tamaños de clave recomendados (NIST SP 800-57):**
  - 128 bits (seguro hasta ≈ 2030), 256 bits (largo plazo, resiliente a avances cuánticos pre-fault-tolerant).
- **Rotación de claves:**

- ❖ Basada en *crypto-period*: El *crypto-period* es **el tiempo máximo durante el cual una clave criptográfica puede usarse de manera segura** para un propósito específico. Se establece para **limitar el riesgo** en caso de que la clave sea comprometida. Una vez terminado ese período, la clave debe **rotarse** (reemplazarse por una nueva). Por ej: datos de tarjeta (PCI-DSS) rotan cada 12 meses; backups "fríos" pueden conservar la clave hasta expiración del contrato.

- **Aceleración hardware:**
  - ❖ AES-NI (Intel), NEON (ARM) → reduce riesgo de *timing-side-channel* al mover operaciones a micro-instrucciones constantes en tiempo.
- **Ventajas:** rápido y eficiente para grandes volúmenes de datos.
- **Reto principal:** distribución segura de la clave.
- **Escenario de mal uso:** cifrar base de datos con AES-CBC, almacenar la clave en repositorio Git; atacante accede a la clave => cifrado inútil. *Solución:* usar KMS (AWS KMS, HashiCorp Vault) + envelope encryption.

4.3 Criptografía pública / asimétrica

- **Definición:** par de claves: *pública* (distribuible) y *privada* (secreta) para cifrar y descifrar mensajes.
- **Intercambio de claves:**
  1. **Diffie-Hellman / ECDHE:** es un protocolo criptográfico para intercambiar claves criptográficas de forma segura a través de un canal público. Permite a dos partes que nunca se han comunicado antes establecer una clave secreta compartida para una comunicación cifrada.
- **Algoritmos típicos:**
  1. **Rivest–Shamir–Adleman (RSA) -** Se basa en el hecho de que la factorización de grandes números primos requiere una potencia de cálculo significativa, y fue el primer algoritmo en aprovechar el paradigma de clave pública/clave privada. Existen distintas longitudes de clave

asociadas a RSA, siendo la de 2048 bits la estándar en la mayoría de los sitios web actuales.

2. **Algoritmo de firma digital (DSA) -** utiliza un algoritmo diferente al RSA para crear claves públicas/privadas, basado en la exponenciación modular y el problema del logaritmo discreto. Ofrece los mismos niveles de seguridad que RSA para claves de tamaño equivalente. DSA fue propuesto por el Instituto Nacional de Estándares y Tecnología (NIST) en 1991 y fue adoptado por el Estándar Federal de Procesamiento de la Información (FIPS) en 1993.
3. **Criptografía de curva elíptica (ECC) -** se basa en algoritmos matemáticos que rigen la estructura algebraica de las curvas elípticas sobre campos finitos. Proporciona niveles de fuerza criptográfica equivalentes a RSA y DSA, con longitudes de clave más cortas. Al igual que DSA, ECC cuenta con la certificación FIPS y también está avalado por la Agencia de Seguridad Nacional (NSA).
4. **Criptografía post-quantum -** La criptografía post-cuántica (también conocida como criptografía a prueba de cuántica, segura para la cuántica o resistente a la cuántica) se refiere a los algoritmos criptográficos (a menudo algoritmos de clave pública) que se espera que sean seguros contra un asalto criptoanalítico por parte de un **ordenador cuántico**. El objetivo de la criptografía **post-cuántica** es crear sistemas criptográficos que sean seguros tanto frente a los ordenadores cuánticos como frente a los convencionales y que, al mismo tiempo, sean compatibles con los protocolos y redes de comunicación existentes.

- **Usos principales:**
    1. **Intercambio de claves** (TLS handshake).
    2. **Firmas digitales** (garantizar origen).
    3. **Cifrado de pequeños fragmentos** (sellar clave simétrica).
- **Limitaciones:** más lento que cifrado simétrico, requiere gestión de certificados.

Firmas digitales y certificados

1. **Definición:** Una **firma digital** es un *sello criptográfico* que se añade a un archivo, mensaje o transacción para probar tres cosas simultáneamente:
    1. **Autenticidad** – demuestra quién generó el contenido (el firmante posee la clave privada).
    2. **Integridad** – garantiza que el documento no ha sido alterado desde que se firmó; cualquier cambio rompe la verificación.
    3. **No-repudio** – impide que el firmante niegue haber realizado la operación, porque sólo él controla su clave privada.
2. **Flujo de firma digital (RSA-PSS) paso a paso:**
    - Se calcula un *hash* (huella) del mensaje con un algoritmo como SHA-256.
    - Ese hash se “cifra” (en realidad se aplica una operación matemática) usando la **clave privada** del firmante → esto produce la firma.
    - El destinatario usa la **clave pública** correspondiente para “des-cifrar” la firma y compara el resultado con un nuevo hash del mensaje:
        - Si coinciden → firma válida.
        - Si no → el contenido cambió o la firma es falsa.
3. **Certificados (X.509)**
    - **Definición:** Un **certificado digital** (normalmente en formato **X.509**) es un documento electrónico que **vincula** una clave pública con la identidad de su propietario (una persona, empresa o sitio web). Funciona como un “carné” expedido y firmado por una **Autoridad de Certificación (CA)** de confianza.
    - Contienen clave pública, identidad (CN), autoridad emisora (CA) y validez.
4. **Elementos clave de un certificado X.509**

Campo	Propósito principal
Sujeto (Subject)	Nombre o dominio del titular (**`CN=www.ejemplo.com`**)
Clave pública	La mitad “pública” del par de claves RSA/ECC
Emisor (Issuer)	CA que firma el certificado
Validez	Fechas *Not Before* / *Not After*
Uso de clave	Qué puede hacer la clave (cifrar, firmar código, TLS, etc.)
Firma de la CA	Hash del contenido cifrado con la clave privada “privé” de la CA

Para verificar un certificado, los navegadores y sistemas operativos mantienen un *almacén* de CAs raíz en las que confían. Así se construye una **cadena de confianza**: Certificado del servidor → CA intermedia → CA raíz.

Relación entre firmas y certificados

- **Certificado:** prueba que una clave pública pertenece a “Empresa X” o “Ana Pérez”.
- **Firma digital:** usa esa clave privada (correspondiente) para sellar un documento.

En la web (HTTPS), el navegador primero **valida el certificado** del sitio; luego, durante el handshake TLS, el servidor **firma** datos efímeros con su clave privada para que el

cliente compruebe la autenticidad de la sesión. **Para mas información visitar el siguiente enlace: What Is an X.509 Certificate?**

5. **Ejemplo práctico (Python + cryptography):**

python

```
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding, rsa

# Generar clave
priv = rsa.generate_private_key(65537, 2048)
pub  = priv.public_key()

mensaje = b'Orden 123: Pagar $10,000 a Proveedor'
# Firmar
firma = priv.sign(
    mensaje,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=32),
    hashes.SHA256())
# Verificar
pub.verify(
    firma, mensaje,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=32),
    hashes.SHA256())
```

VERIFICAR ARCHIVO SUBIDO A BLACKBOARD PARA PROBAR EN VS Code.

Conceptos Claves en Base a las firmas digitales y verificación

- **No-repudio:** el firmante no puede negar su participación porque la clave privada es única.

- **Revocación:** CRL u OCSP si la clave se compromete.

Programación defensiva

1. **Validar siempre entradas**
    - *Principio:* “never trust, always verify”.
    - Ejemplo (Flask): usar `werkzeug.security.safe_str_cmp` y validaciones regex estrictas.
2. **Asumir fallas y codificar para el peor caso**
    - Verificar resultados de operaciones críticas (`if not db.commit(): rollback()`), usar timeouts y *circuit breakers*.
3. **Principio de mínimo privilegio**
    - Contenedor que ejecuta app con UID no root y permisos solo de lectura en `/app`.
4. **Fail-safe defaults (*Configuración por defecto debe ser segura*.)**
    - Firewall predeterminado a `DROP`, API deniega si token JWT (**Jason Web Tocken**) expiró.
5. **Defensa en profundidad**
    - Input validation → ORM (Object-Relational Mapping) parametrizado → WAF → IDS (Intrusion Detection System)
6. **Revisión constante (Secure SDLC)**
    - Análisis estático (Bandit, SonarQube) en pipelines CI/CD.
        i. Pipelines CI/CD - onjunto automatizado de pasos que llevan el software desde el **código fuente** hasta la **producción**.
        ii. CI (Continuous Integration / Integración Continua) - Automatiza la construcción, pruebas y validación del código cada vez que un desarrollador hace un commit.
        iii. CD (Continuous Delivery / Despliegue Continuo) - Automatiza el despliegue del software en entornos (staging, producción).
    - *Code Review* cruzado con checklist OWASP ASVS.
7. **catálogo de patrones y anti-patrones**

Principio	Patrón seguro	Anti-patrón (qué NO hacer)
Validación	Lista blanca + length check	Regex permisivo
Codificación de salida	`html.escape` en HTML	Concatenar strings en `<script>`
Gestión de errores	Mensaje genérico + log detallado	`print(e)` al usuario
Prueba de fallas	`assert db.commit()`	Suponer que commit siempre funciona
Privilegio mínimo	Contenedor ejecuta UID 65534	Ejecutar como root
Logging seguro	Hash de PII (`sha256(user.email)`)	Registro en claro
Timeouts & circuit breaker	`requests.get(url, timeout=3)`	Llamada bloqueante indefinida

Herramientas de apoyo

- **Bandit** (Python) – detecta uso inseguro de funciones aleatorias y construcción de comandos.
- **Semgrep** – reglas personalizadas para buscar patrones “crypto-misuse”.
- **Chaos engineering** – inyectar fallos en API de cifrado (clave incorrecta, CA offline) y verificar resiliencia.

8. **Ejemplo de patch defensivo (Python):**

python

```
import re, html
def safe_comment(user_input: str) -> str:
    assert len(user_input) < 500, "Comentario muy largo"
    sane = html.escape(user_input)        # evita XSS
    if re.search(r"(https?://|<script)", sane, re.I):
        raise ValueError("Contenido prohibido")
    return sane
```

Explicación de Codigo de Defensa en Python

`. import re, html`

Se importan dos módulos:

re → permite usar expresiones regulares para buscar patrones sospechosos.

html → tiene funciones para escapar caracteres HTML peligrosos.

`. def safe_comment(user_input: str) -> str:`

Define la función safe_comment, que recibe un texto escrito por un usuario (ejemplo: un comentario en un blog).

El parámetro user_input es un str (string).

Devuelve otro string ya sanitizado.

```
. assert len(user_input) < 500, "Comentario muy largo"
```

Primera defensa: control de tamaño.

Si el comentario tiene más de 500 caracteres, la función lanza un error (AssertionError).

Esto evita ataques de buffer overflow o spam masivo con comentarios enormes.

```
. sane = html.escape(user_input)        # evita XSS
```

Segunda defensa: escapar caracteres HTML especiales (<, >, &, ").

Si el usuario escribe <script>alert("hack")</script>, al escapar se convierte en:

<script>alert("hack")</script>

Resultado: el navegador lo muestra como texto y no lo ejecuta como JavaScript.

Así bloquea ataques de XSS (inyección de scripts en páginas web).

```
. if re.search(r"(https?://|<script)", sane, re.I):
      raise ValueError("Contenido prohibido")
```

Tercera defensa: filtro adicional con expresiones regulares.

Busca si el texto contiene:

- http:// o https:// (intentos de meter enlaces externos).
- <script> (posibles scripts maliciosos).

La bandera re.I = búsqueda insensible a mayúsculas/minúsculas (ej. <ScRiPt> también sería detectado).

Si encuentra algo → lanza un error ValueError("Contenido prohibido").

```
. return sane
```

Si pasa todas las validaciones, devuelve el comentario seguro y limpio, listo para ser guardado o mostrado.

Imágenes de ejemplo en VS Code

1. Codigo de Defensa

Codigo Defensa.py

```python
import re, html
def safe_comment(user_input: str) -> str:
    assert len(user_input) < 500, "Comentario muy largo"
    sane = html.escape(user_input)        # evita XSS
    if re.search(r"(https?://|<script)", sane, re.I):
        raise ValueError("Contenido prohibido")
    return sane

try:
    entrada = '<script>alert("hackeado!")</script>'
    print("Entrada maliciosa:", entrada)
    salida = safe_comment(entrada)
    print("Salida:", salida)
except Exception as e:
    print("Bloqueado:", e)

print("-" * 50)

try:
    entrada = '¡Hola, excelente artículo! Gracias por compartir.'
    print("Entrada segura:", entrada)
    salida = safe_comment(entrada)
    print("Salida segura:", salida)
except Exception as e:
    print("Bloqueado:", e)

```

2. Respuesta de ejemplo

```
Entrada maliciosa: <script>alert("hackeado!")</script>
Salida: &lt;script&gt;alert(&quot;hackeado!&quot;)&lt;/script&gt;
--------------------------------------------------
Entrada segura: ¡Hola, excelente artículo! Gracias por compartir.
Salida segura: ¡Hola, excelente artículo! Gracias por compartir.
```

En Resumen:

El código es un **filtro defensivo de comentarios de usuario**. Hace 3 cosas:

1. **Controla longitud** → no permite comentarios muy largos.
2. **Escapa HTML** → evita que el navegador ejecute código (previene XSS).
3. **Bloquea patrones prohibidos** → enlaces y etiquetas <script>.

Relación con Otros Conceptos

- El cifrado **simétrico** protege la confidencialidad de grandes datos, mientras que la **clave pública** facilita intercambio seguro y *firmas*; ambos sustentan la autenticación fuerte (paso previo al control de acceso).
- La **programación defensiva** complementa la criptografía: un algoritmo seguro mal implementado (p. ej., IV reutilizado) sigue siendo vulnerable.

Resumen de la Lección

Comprendimos cómo el cifrado en bloque y en transmisión enfrenta amenazas diferentes, por qué la criptografía simétrica es veloz pero depende de la protección de la clave, y cómo la criptografía asimétrica aporta autenticidad y no-repudio mediante firmas y certificados X.509. Finalmente, enlazamos estos conceptos a la práctica diaria de programación defensiva—validar entradas, principio de mínimo privilegio y fail-safe defaults—para que el código no se convierta en el eslabón más débil.

Actividad de la Lección

Mini-Proyecto “Mensaje Seguro” (120 min, en parejas)

1. Implementen un script CLI que:
    - Cifre un archivo de texto con AES-GCM (clave secreta aleatoria).
    - Cifre la clave AES con la clave pública del destinatario.
    - Firme la clave AES cifrada con la clave privada del remitente.
2. Incluyan controles defensivos:
    - Validar longitud del archivo (< 1 MB).

    - Manejar excepciones (“clave incorrecta”, “firma no válida”).
3. Demuestren intercambio exitoso (enviar archivo + clave + firma).
4. Entreguen repositorio Git y README con instrucciones.

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.).
- NIST FIPS-197 (2001). *Advanced Encryption Standard (AES)*.
- NIST SP 800-57 (2020). *Key Management Guidelines*.
- OWASP. *Cryptographic Storage Cheat Sheet* (2025).
- SSL.com. (s. f.). *What is an X.509 certificate?* SSL.com. https://www.ssl.com/faqs/what-is-an-x-509-certificate/
- Whitestack. (s. f.). *Cifrado AES*. Whitestack. https://whitestack.com/es/blog/cifrado-aes/