CRIPTOGRAFÍA CON PYTHON

AES-Python

Criptografía Moderna

Autor: Enrique Melchor Iborra Sanjaime (em.iborrasanjaime@edu.gva.es)

Contenido

1. Fundamentos de la Criptografía Simétrica

La criptografía simétrica es un enfoque en el cual tanto el emisor como el receptor utilizan la misma clave secreta para cifrar y descifrar los datos. Es uno de los métodos más utilizados debido a su eficiencia y rapidez. Sin embargo, uno de los principales desafíos es la gestión segura de la clave, ya que si la clave es comprometida, toda la comunicación se ve vulnerada.

Algoritmos de Criptografía Simétrica:

- **AES (Advanced Encryption Standard):** El AES es uno de los algoritmos de criptografía más utilizados y es considerado seguro para la mayoría de las aplicaciones. Opera sobre bloques de 128 bits y admite claves de 128, 192 y 256 bits. AES es el algoritmo que reemplazó al DES y Triple DES debido a su mayor seguridad y eficiencia.

- **DES (Data Encryption Standard):** DES es un algoritmo de cifrado de bloques de 64 bits que utiliza una clave de 56 bits. Aunque fue ampliamente utilizado en su tiempo, actualmente se considera inseguro debido a que la longitud de la clave es corta y los avances en la potencia de computación han hecho posible su descifrado mediante ataques de fuerza bruta.

- **Triple DES (3DES):** Triple DES es una versión más segura del DES, que realiza tres rondas de cifrado con DES utilizando dos o tres claves distintas. A pesar de ser más seguro que DES, 3DES también está en desuso debido a que no ofrece suficiente seguridad frente a ataques modernos y es más lento que otros algoritmos como AES.

Modos de Operación: Los algoritmos de cifrado simétrico pueden utilizar diferentes modos de operación, los cuales definen cómo se aplican las operaciones de cifrado a los bloques de datos. Los modos más comunes son:

- **ECB (Electronic Codebook):** En este modo, cada bloque de texto claro se cifra de forma independiente utilizando la misma clave. Es muy sencillo de implementar, pero no es seguro para la mayoría de las aplicaciones debido a que los bloques idénticos de texto claro producirán bloques idénticos de texto cifrado, lo que puede ser explotado por un atacante.
- **CBC (Cipher Block Chaining):** En el modo CBC, el primer bloque de texto claro se XOR con un vector de inicialización (IV) antes de ser cifrado. Luego, cada bloque subsiguiente de texto claro se XOR con el bloque cifrado anterior. Este modo mejora la seguridad al evitar la repetición de patrones en el texto cifrado.
- **CFB (Cipher Feedback):** CFB es similar a CBC, pero en lugar de operar sobre bloques de texto claro, funciona con unidades más pequeñas de datos, como bytes o bits. Se utiliza cuando es necesario cifrar y descifrar datos de manera más eficiente en tiempo real.
- **GCM (Galois/Counter Mode):** GCM es un modo de operación de cifra y autenticación. Proporciona no solo confidencialidad sino también integridad de los datos, ya que incluye un código de autenticación de mensaje (MAC). Es utilizado en protocolos modernos como TLS y VPNs debido a su eficiencia y seguridad.

2. Implementación en Python

En Python, se puede implementar criptografía simétrica utilizando bibliotecas como `PyCryptodome`, que proporciona una amplia variedad de algoritmos y modos de operación. A continuación, se desarrollan los aspectos clave para implementar AES con la biblioteca `PyCryptodome`:

Encriptación y desencriptación con AES: Para encriptar y desencriptar datos con AES, se debe realizar lo siguiente:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Generación de una clave de 256 bits
key = get_random_bytes(32)

# Cifrado
cipher = AES.new(key, AES.MODE_CBC)
data = b"Mensaje de prueba"
ciphertext = cipher.encrypt(pad(data, AES.block_size))

# Desencriptación
decipher = AES.new(key, AES.MODE_CBC, cipher.iv)
plaintext = unpad(decipher.decrypt(ciphertext), AES.block_size)

print("Texto original:", data)
print("Texto cifrado:", ciphertext)
print("Texto descifrado:", plaintext)
```

Manejo de Inicialización de Vectores (IV): El vector de inicialización (IV) es un elemento crucial en los modos de operación como CBC y CFB. El IV debe ser único para cada operación de cifrado, y no debe ser secreto. En el código anterior, se observa que el IV es generado automáticamente por

`AES.new()` y se incluye junto con el texto cifrado. Es importante guardar el IV junto con el texto cifrado para que el proceso de descifrado sea posible.

Generación segura de claves: La generación de claves debe ser aleatoria y segura. Utilizamos `get_random_bytes()` de `PyCryptodome` para generar claves de una longitud segura (como 256 bits para AES). No se deben usar claves predecibles o fácilmente adivinables, por lo que es fundamental emplear funciones de generación de números aleatorios criptográficamente seguras.

Generación de un IV aleatorio: El IV también se genera de manera aleatoria, y su tamaño debe coincidir con el tamaño de bloque del algoritmo. En el caso de AES, el tamaño de bloque es 128 bits (16 bytes), por lo que el IV también será de 16 bytes.

```
# Generación de un IV aleatorio
iv = get_random_bytes(AES.block_size)
```

Resumen

- **Criptografía simétrica** usa una clave compartida para cifrar y descifrar los datos.
- **AES** es el algoritmo moderno de criptografía simétrica más utilizado y seguro.
- **Modos de operación** como **ECB**, **CBC**, **CFB**, y **GCM** permiten un control adicional sobre cómo se cifra el texto claro.
- En **Python**, la biblioteca `PyCryptodome` facilita la implementación de cifrado simétrico, proporcionando herramientas para la encriptación, desencriptación, manejo de IVs y generación segura de claves.

Esta base te permitirá implementar sistemas de cifrado simétrico seguros y eficientes en aplicaciones de software.

---

3. Actividades: Criptografía Simétrica

Actividad 1: Conceptos clave de AES, DES y Triple DES

1. Realiza una investigación sobre los algoritmos AES, DES y Triple DES. Responde las siguientes preguntas:
    - ¿Qué características diferencian a AES de DES?
    - ¿Por qué se considera que DES ya no es seguro?
    - Explica cómo Triple DES mejora la seguridad de DES.
2. Completa el siguiente cuadro comparativo:

Algoritmo	Longitud de clave	Bloque de datos	Seguridad actual
DES			
Triple DES			

Algoritmo	Longitud de clave	Bloque de datos	Seguridad actual
AES			

Actividad 2: Encriptación y desencriptación con AES

1. Escribe un programa en Python que realice lo siguiente:
    - Generar una clave de 256 bits para AES.
    - Encriptar un mensaje de texto utilizando el modo CBC.
    - Desencriptar el mensaje y verificar que el resultado coincide con el texto original.

Pista: Usa la librería `Cryptography` *o* `PyCryptodome`.

2. Prueba tu implementación con los siguientes textos y claves generadas:
    - Texto: "Este es un mensaje secreto".
    - Texto: "La criptografía es fascinante".

Actividad 3: Manejo de inicialización de vectores (IV)

1. Investiga el propósito del vector de inicialización (IV) en modos como CBC y CFB. Responde:
    - ¿Qué sucede si no se utiliza un IV?
    - ¿Es seguro reutilizar el mismo IV con la misma clave? Explica.
2. Modifica el programa de la **Actividad 3** para:
    - Generar un IV aleatorio para cada encriptación.
    - Mostrar el IV generado junto con el texto cifrado en formato hexadecimal.

Actividad 4: Generación segura de claves

1. Implementa un script en Python que genere claves de forma segura utilizando el módulo `os.urandom` o funciones de la librería `Cryptography`.

2. Incluye en tu script un mecanismo para guardar y cargar claves desde un archivo en formato seguro.

Reto adicional: Implementa una función para verificar la integridad de la clave al cargarla.

Actividad 5: Crea una herramienta de cifrado y descifrado simple

1. Desarrolla un programa en Python que permita al usuario:
    - Elegir un algoritmo de cifrado (AES, DES o Triple DES).
    - Seleccionar un modo de operación (CBC, ECB, etc.).
    - Introducir un mensaje para cifrar.
    - Ver el mensaje cifrado y descifrado.
2. Asegúrate de que:
    - Las claves y los IV se generen de forma segura.
    - Los datos cifrados se presenten en formato hexadecimal o base64.
3. Agrega una interfaz básica en la terminal para interactuar con el programa.

Actividad 6: Análisis de desempeño de algoritmos y modos de operación

Objetivo: Evaluar la eficiencia de los algoritmos y modos de operación en diferentes escenarios.

1. **Implementación y medición:**
    - Crea un programa en Python que mida el tiempo de encriptación y desencriptación utilizando AES en los modos ECB, CBC, CFB y GCM.
    - Realiza las pruebas con mensajes de diferentes tamaños: 16 bytes, 1 KB, 1 MB.
2. **Análisis:**
    - Completa un cuadro con los resultados obtenidos:

Modo de operación	Tamaño del mensaje	Tiempo de encriptación	Tiempo de desencriptación

3. **Reflexión:**
    - Responde las siguientes preguntas:
        - ¿Qué modo es más rápido en promedio?
        - ¿Cómo influye el tamaño del mensaje en el rendimiento?
        - ¿Qué ventajas podría tener un modo más lento en términos de seguridad?

Actividad 7: Simulación de un ataque en modo ECB

Objetivo: Demostrar las vulnerabilidades del modo ECB mediante un ejemplo práctico.

1. **Preparación:**
    - Utiliza una imagen en formato BMP donde los píxeles repetidos sean evidentes (por ejemplo, una imagen con patrones).
    - Escribe un programa en Python que:
        - Encripte la imagen usando AES en modo ECB.
        - Guarde la imagen cifrada en un nuevo archivo.
2. **Análisis:**
    - Observa la imagen cifrada y responde:
        - ¿Qué partes del patrón original aún son visibles?
        - ¿Por qué ocurre esto en el modo ECB?
3. **Reto adicional:**
    - Modifica el programa para cifrar la misma imagen usando el modo CBC.
    - Compara los resultados visuales entre ECB y CBC.

4. Fundamentos del Hashing

El **hashing** es un proceso matemático que toma una entrada (o "mensaje") de longitud arbitraria y la transforma en una salida de longitud fija, generalmente representada como una cadena alfanumérica. El resultado es denominado "hash" o "resumen". Las funciones hash son fundamentales en la criptografía y la seguridad de la información, ya que permiten almacenar y verificar datos de manera eficiente.

Propiedades de las funciones hash

Las funciones hash criptográficas deben cumplir con ciertas propiedades para ser útiles en seguridad:

1. **Determinismo**: La misma entrada siempre debe generar el mismo hash.
2. **Velocidad**: La función hash debe ser rápida de calcular, incluso con entradas grandes.
3. **Resistencia a colisiones**: Es difícil encontrar dos entradas diferentes que produzcan el mismo hash.
4. **Resistencia a preimagen**: Dada una salida de hash, debe ser computacionalmente inviable obtener la entrada original.
5. **Avalancha**: Un pequeño cambio en la entrada (incluso un solo bit) debe generar un cambio drástico en el hash, lo que garantiza que las funciones hash son sensibles a pequeñas variaciones.

Algoritmos comunes de Hashing

1. **MD5 (Message Digest Algorithm 5)**: Aunque MD5 es rápido y ampliamente usado, ha sido considerado inseguro debido a la posibilidad de encontrar colisiones. En la práctica, no se recomienda para aplicaciones sensibles.
2. **SHA-1 (Secure Hash Algorithm 1)**: Aunque más robusto que MD5, SHA-1 también ha sido vulnerable a ataques de colisión. A pesar de esto, todavía se usa en aplicaciones no críticas, aunque se recomienda su reemplazo por versiones más seguras.
3. **SHA-256 (Secure Hash Algorithm 256 bits)**: Parte de la familia SHA-2, es una de las funciones hash más seguras disponibles. Produce un hash de 256 bits y se utiliza ampliamente en criptografía y almacenamiento seguro de contraseñas.
4. **SHA-3**: Es la última versión de la familia de funciones hash SHA, construida sobre un algoritmo diferente llamado Keccak. Ofrece un nivel de seguridad superior y se considera más resistente a ataques en comparación con SHA-2.

5. Implementación en Python

Python ofrece bibliotecas como `hashlib` para generar y trabajar con funciones hash de manera sencilla. Vamos a ver cómo implementar el hashing y la autenticación con HMAC (Hash-based Message Authentication Code) en Python.

Generación de hashes con `hashlib`

`hashlib` es una biblioteca estándar de Python que proporciona acceso a varios algoritmos de hashing. A continuación se muestra cómo generar hashes usando algunos de los algoritmos más comunes.

```
import hashlib

# Hashing con MD5
data = "Mi mensaje".encode('utf-8')
md5_hash = hashlib.md5(data).hexdigest()
print(f"MD5: {md5_hash}")

# Hashing con SHA-1
sha1_hash = hashlib.sha1(data).hexdigest()
print(f"SHA-1: {sha1_hash}")

# Hashing con SHA-256
sha256_hash = hashlib.sha256(data).hexdigest()
print(f"SHA-256: {sha256_hash}")

# Hashing con SHA-3-256
sha3_256_hash = hashlib.sha3_256(data).hexdigest()
print(f"SHA-3-256: {sha3_256_hash}")
```

En este código, primero se convierte el mensaje a bytes con `.encode('utf-8')`, ya que las funciones hash en Python esperan datos binarios como entrada. Luego, se genera el hash correspondiente utilizando los métodos `md5()`, `sha1()`, `sha256()`, y `sha3_256()`.

HMAC para autenticación

HMAC (Hash-based Message Authentication Code) es un mecanismo de autenticación basado en el uso de una clave secreta y una función hash. HMAC garantiza que los datos no hayan sido alterados y que provienen de una fuente confiable.

```
import hmac   ## al igual que haslib, hmac viene incluido en python
import hashlib

# Definir la clave secreta y el mensaje
clave_secreta = b"clave_secreta"
mensaje = b"Este es un mensaje importante."

# Crear un objeto HMAC utilizando SHA-256
hmac_obj = hmac.new(clave_secreta, mensaje, hashlib.sha256)

# Obtener el código HMAC
hmac_hash = hmac_obj.hexdigest()
print(f"HMAC-SHA-256: {hmac_hash}")
```

En este código, `hmac.new()` toma tres parámetros: la clave secreta, el mensaje, y el algoritmo de hashing (en este caso, SHA-256). El resultado es un código HMAC que puede ser utilizado para verificar la integridad y autenticidad del mensaje.

Verificación de la integridad de los datos

Para verificar que los datos no han sido alterados, se genera un hash de los datos recibidos y se compara con el hash previamente calculado o almacenado. Si coinciden, significa que los datos son íntegros.

Ejemplo de verificación de la integridad usando HMAC:

```
# Función para verificar la integridad
def verificar_integridad(mensaje, hash_esperado, clave_secreta):
    # Generar el HMAC del mensaje recibido
    hmac_obj = hmac.new(clave_secreta, mensaje.encode('utf-8'), hashlib.sha256)
    hmac_calculado = hmac_obj.hexdigest()

    # Comparar los hashes
    if hmac_calculado == hash_esperado:
        print("La integridad es verificada.")
    else:
        print("Los datos han sido alterados.")

# Ejemplo de uso
mensaje_recibido = "Este es un mensaje importante."
hash_esperado = hmac_hash  # Este es el HMAC previamente calculado
verificar_integridad(mensaje_recibido, hash_esperado, clave_secreta)
```

En este código, la función `verificar_integridad` toma el mensaje recibido, calcula su HMAC con la misma clave secreta y lo compara con el hash esperado. Si ambos coinciden, la integridad del mensaje está garantizada.

6. Actividades: Hashing y Autenticación con Python

Actividad 1: Propiedades de las funciones hash

1. Investiga y explica las siguientes propiedades de las funciones hash:
    - Determinismo.
    - Resistencia a colisiones.
    - Resistencia a preimágenes.
    - Resistencia a la segunda preimagen.
    - Avalancha (efecto avalanche).
2. Responde:
    - ¿Por qué el hashing no se utiliza para encriptar datos, sino para verificarlos?
    - ¿Qué sucede si una función hash no es resistente a colisiones?

Actividad 2: Comparación de algoritmos hash

1. Investiga los algoritmos MD5, SHA-1, SHA-256 y SHA-3, y completa el siguiente cuadro:

Algoritmo	Longitud de salida	Seguridad actual	Uso recomendado
MD5			
SHA-1			
SHA-256			
SHA-3			

2. Responde:
    - ¿Por qué MD5 y SHA-1 ya no son seguros para aplicaciones críticas?
    - ¿Qué ventajas tiene SHA-3 sobre SHA-256?

Actividad 3: Generación de hashes con hashlib

1. Escribe un programa en Python que genere hashes para un texto dado usando MD5, SHA-1, SHA-256 y SHA-3.
    - Entrada: "La criptografía es fundamental para la seguridad".
    - Salida: Imprime el hash generado por cada algoritmo en formato hexadecimal.
2. **Reto adicional:**
    - Permite que el usuario ingrese un archivo y genera el hash de su contenido.

Actividad 4: Implementación de HMAC para autenticación

1. Investiga qué es HMAC (Hashed Message Authentication Code) y responde:
    - ¿Qué problema de seguridad resuelve HMAC?

- ¿Por qué HMAC es más seguro que simplemente calcular un hash?

2. Escribe un programa en Python que:
    - Genere un código HMAC para un mensaje usando una clave secreta y SHA-256.
    - Permita verificar el HMAC de un mensaje recibido.

Pista: Usa la librería `hmac` *de Python.*

Actividad 5: Verificación de integridad de datos

1. Escribe un programa que permita verificar la integridad de un archivo utilizando SHA-256:
    - Genera el hash de un archivo original y guárdalo.
    - Permite comparar el hash original con el de una copia para verificar si el archivo fue modificado.
2. **Reto adicional:**
    - Implementa un programa que automatice la verificación de integridad para múltiples archivos en una carpeta.

,

Actividad 6: Desarrolla una aplicación completa que combine los conceptos aprendidos

1. Funcionalidades:
    - Generar y guardar hashes de texto o archivos usando SHA-256 o SHA-3.
    - Implementar HMAC para autenticar mensajes entre dos usuarios.
    - Verificar la integridad de datos transmitidos o almacenados.
2. **Requisitos adicionales:**
    - Diseña una interfaz en línea de comandos o una GUI básica.
    - Agrega validaciones para manejar errores comunes, como archivos inexistentes o hashes mal formateados.

---

7. Fundamentos de la Criptografía Asimétrica

La criptografía asimétrica, también conocida como criptografía de clave pública, es un método criptográfico que utiliza un par de claves relacionadas: una **clave pública** para cifrar los datos y una **clave privada** para descifrarlos. A diferencia de la criptografía simétrica, donde se usa la misma clave tanto para cifrar como para descifrar, la criptografía asimétrica mejora la seguridad al permitir la distribución pública de una de las claves.

Algoritmos principales de criptografía asimétrica

- **RSA (Rivest-Shamir-Adleman):** RSA es uno de los algoritmos más conocidos y utilizados en criptografía asimétrica. Se basa en la dificultad de factorizar números grandes. La seguridad de RSA depende del tamaño de las claves y de la imposibilidad de factorizar números primos grandes en un tiempo razonable con los métodos actuales. RSA es comúnmente usado en aplicaciones de firmas digitales, cifrado de datos y autenticación.

En RSA, el par de claves (pública y privada) se genera mediante operaciones matemáticas complejas sobre grandes números primos, y el proceso de cifrado/desencriptado se realiza usando una relación matemática entre estas claves.

- **ECC (Elliptic Curve Cryptography):** La criptografía de curvas elípticas (ECC) es un enfoque más moderno y eficiente que RSA. En lugar de basarse en la factorización de números grandes, ECC se basa en la dificultad de resolver el problema del logaritmo discreto sobre curvas elípticas. ECC ofrece el mismo nivel de seguridad que RSA pero con claves mucho más pequeñas, lo que hace que sea más eficiente en términos de tiempo de procesamiento y uso de recursos.

ECC se utiliza en aplicaciones como cifrado, autenticación, y firmas digitales, y es especialmente adecuado para entornos de recursos limitados, como dispositivos móviles y sistemas embebidos.

Gestión de claves públicas y privadas: En la criptografía asimétrica, las claves públicas pueden ser compartidas libremente, mientras que las claves privadas deben ser mantenidas en secreto por el propietario. La clave pública se utiliza para cifrar los mensajes, y solo la clave privada correspondiente puede descifrarlos. En el caso de las firmas digitales, la clave privada se utiliza para firmar los mensajes, y la clave pública correspondiente se utiliza para verificar la firma.

El manejo adecuado de las claves es crucial para la seguridad del sistema. Las claves deben ser generadas de manera segura, almacenadas de forma protegida y distribuidas con cuidado para evitar que caigan en manos equivocadas.

8. Implementación en Python

En Python, una de las bibliotecas más populares para trabajar con criptografía asimétrica es `PyCryptodome` para RSA y `ecdsa` para ECC. A continuación se muestra cómo implementar criptografía asimétrica usando estas bibliotecas.

Generación de pares de claves RSA: Utilizando `PyCryptodome`, podemos generar un par de claves públicas y privadas para RSA. Aquí se muestra cómo generar claves RSA de 2048 bits:

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes

# Generar un par de claves RSA de 2048 bits
key = RSA.generate(2048)

# Obtener la clave privada
private_key = key.export_key()

# Obtener la clave pública
public_key = key.publickey().export_key()
```

```
# Guardar las claves en archivos
with open("private.pem", "wb") as f:
    f.write(private_key)

with open("public.pem", "wb") as f:
    f.write(public_key)
```

Encriptación y desencriptación con RSA: En RSA, el proceso de cifrado y descifrado implica el uso de la clave pública y privada. A continuación, mostramos cómo cifrar y descifrar mensajes con RSA:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes

# Cargar la clave pública y privada desde los archivos
with open("public.pem", "rb") as f:
    public_key = RSA.import_key(f.read())

with open("private.pem", "rb") as f:
    private_key = RSA.import_key(f.read())

# Crear el objeto de cifrado con la clave pública
cipher_rsa = PKCS1_OAEP.new(public_key)

# Mensaje a cifrar
message = b"Mensaje secreto"
ciphertext = cipher_rsa.encrypt(message)

# Crear el objeto de descifrado con la clave privada
cipher_rsa_private = PKCS1_OAEP.new(private_key)

# Descifrado del mensaje
decrypted_message = cipher_rsa_private.decrypt(ciphertext)

print(f"Mensaje original: {message}")
print(f"Mensaje cifrado: {ciphertext}")
print(f"Mensaje descifrado: {decrypted_message}")
```

Firmas digitales y verificación con RSA: Las firmas digitales permiten verificar la integridad y autenticidad de un mensaje. La firma se genera con la clave privada, y la verificación se realiza con la clave pública.

```
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256

# Cargar la clave pública y privada desde los archivos
with open("private.pem", "rb") as f:
    private_key = RSA.import_key(f.read())

with open("public.pem", "rb") as f:
    public_key = RSA.import_key(f.read())

# Mensaje a firmar
message = b"Este es un mensaje para firmar"

# Crear el hash del mensaje
hash_message = SHA256.new(message)

# Crear la firma con la clave privada
```

```
signer = pkcs1_15.new(private_key)
signature = signer.sign(hash_message)

# Verificar la firma con la clave pública
verifier = pkcs1_15.new(public_key)
try:
    verifier.verify(hash_message, signature)
    print("La firma es válida.")
except (ValueError, TypeError):
    print("La firma no es válida.")
```

Generación de pares de claves ECC: Usando la biblioteca `ecdsa`, podemos generar un par de claves ECC y realizar operaciones de cifrado y firma digital.

```
from ecdsa import ellipticcurve, SECP256k1, SigningKey

# Generar una clave privada ECC
sk = SigningKey.generate(curve=SECP256k1)

# Generar la clave pública ECC
vk = sk.get_verifying_key()

# Guardar las claves en archivos
with open("ecc_private.pem", "wb") as f:
    f.write(sk.to_pem())

with open("ecc_public.pem", "wb") as f:
    f.write(vk.to_pem())
```

Firmas digitales con ECC: Al igual que con RSA, se puede firmar un mensaje usando la clave privada y verificarlo con la clave pública en ECC.

```
from ecdsa import VerifyingKey
from ecdsa.util import sigdecode_string

# Cargar la clave pública y privada
with open("ecc_private.pem", "rb") as f:
    sk = SigningKey.from_pem(f.read())

with open("ecc_public.pem", "rb") as f:
    vk = VerifyingKey.from_pem(f.read())

# Mensaje a firmar
message = b"Este es un mensaje para firmar"

# Firmar el mensaje con la clave privada
signature = sk.sign(message)

# Verificar la firma con la clave pública
try:
    vk.verify(signature, message)
    print("La firma es válida.")
except:
    print("La firma no es válida.")
```

---

9. Actividades: Criptografía Asimétrica

Actividad 1: Comparación entre RSA y ECC

1. Realiza una investigación sobre RSA y ECC y responde las siguientes preguntas:
    - ¿Cuáles son las principales diferencias entre RSA y ECC en términos de seguridad y eficiencia?
    - ¿Por qué ECC es preferido en dispositivos con recursos limitados?
    - Completa el siguiente cuadro comparativo:

      Algoritmo **Longitud típica de clave** **Seguridad equivalente** **Uso recomendado**
      RSA
      ECC

Actividad 2: Gestión de claves públicas y privadas

1. Responde:
    - ¿Qué diferencia hay entre una clave pública y una clave privada?
    - ¿Por qué la seguridad de la criptografía asimétrica depende de mantener la clave privada en secreto?
    - Explica qué es un certificado digital y cuál es su relación con las claves públicas.
2. **Reto práctico:**
    - Investiga cómo se almacenan las claves públicas y privadas en un archivo (por ejemplo, en formatos PEM o DER).
    - Identifica qué información contiene un archivo PEM.

Actividad 3: Generación de pares de claves

1. Crea un programa en Python que realice lo siguiente:
    - Generar un par de claves (pública y privada) utilizando RSA.
    - Guardar las claves en archivos separados (`clave_privada.pem` y `clave_publica.pem`).

Pista: Usa librerías como `Cryptography` *o* `PyCryptodome`.

2. **Reto adicional:**
    - Implementa la generación de claves ECC.

Actividad 4: Encriptación y desencriptación con RSA

1. Escribe un programa en Python que permita:
    - Cargar las claves generadas en la **Actividad 3**.
    - Encriptar un mensaje con la clave pública.
    - Desencriptar el mensaje con la clave privada.
2. **Reto adicional:**
    - Modifica el programa para aceptar un archivo como entrada y cifrar su contenido.

Actividad 5: Firmas digitales y verificación

1. Implementa un programa que realice:
    - Generar una firma digital para un mensaje usando una clave privada.
    - Verificar la autenticidad del mensaje usando la clave pública correspondiente.
2. **Reto adicional:**
    - Incluye una función para detectar si el mensaje fue alterado después de la firma.
    - Muestra cómo se genera un resumen (hash) del mensaje antes de firmarlo.

Actividad 6: Desarrolla una aplicación completa de criptografía asimétrica

1. Diseña una herramienta que permita:
    - Generar pares de claves (RSA o ECC).
    - Cargar y guardar claves en archivos.
    - Cifrar y descifrar mensajes o archivos.
    - Firmar mensajes digitalmente y verificar las firmas.
2. **Requisitos adicionales:**
    - Crea una interfaz gráfica simple o una interfaz en línea de comandos para interactuar con la herramienta.
    - Asegúrate de manejar errores comunes, como el uso de claves incorrectas o archivos dañados.
