CRIPTOGRAFÍA CON PYTHON

Aplicaciones avanzadas con Criptografía en Python

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

Contenido

1. Manejo de Certificados Digitales

Introducción a X.509

El estándar **X.509** es uno de los más utilizados para la creación y validación de certificados digitales, que proporcionan autenticación e integridad en las comunicaciones en línea. X.509 define el formato para los certificados que contienen información sobre la identidad de un sujeto (una persona, organización, etc.) y su clave pública, y está basado en un sistema de clave pública.

Partes de un certificado X.509:

- **Versión**: Indica la versión del certificado (por ejemplo, X.509 v3).
- **Número de serie**: Identificador único para el certificado.
- **Algoritmo de firma**: Especifica el algoritmo utilizado para firmar el certificado (por ejemplo, RSA).
- **Emisor**: La autoridad que emite el certificado.
- **Sujeto**: El titular del certificado.
- **Clave pública**: La clave pública asociada con el sujeto.
- **Período de validez**: Fechas de inicio y fin de la validez del certificado.
- **Firma**: Firma digital generada por el emisor usando su clave privada.

Creación y uso de certificados con OpenSSL y Python

OpenSSL es una herramienta ampliamente utilizada para la creación y gestión de certificados digitales. Python puede interactuar con OpenSSL a través de bibliotecas como `cryptography` y `PyOpenSSL`.

Generación de un certificado X.509 con OpenSSL:

1. **Generación de una clave privada RSA**:

```
openssl genpkey -algorithm RSA -out private_key.pem
```

2. **Generación de una solicitud de firma de certificado (CSR)**:

```
openssl req -new -key private_key.pem -out request.csr
```

3. **Generación de un certificado autofirmado (opcional para pruebas)**:

```
openssl req -x509 -key private_key.pem -in request.csr -out certificate.pem
-days 365
```

Generación de un certificado X.509 con Python (con `cryptography`)

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.x509 import CertificateBuilder, Name, NameAttribute
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
import datetime

# Generación de una clave privada
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)

# Creación del certificado
builder = CertificateBuilder()
builder = builder.subject_name(Name([NameAttribute(NameOID.COMMON_NAME,
u"example.com")]))
builder = builder.issuer_name(Name([NameAttribute(NameOID.COMMON_NAME,
u"example.com")]))
builder = builder.not_valid_before(datetime.datetime.today())
builder = builder.not_valid_after(datetime.datetime.today() +
datetime.timedelta(days=365))
builder = builder.public_key(private_key.public_key())
builder = builder.serial_number(1000)

# Firmado con la clave privada
certificate = builder.sign(private_key=private_key, algorithm=hashes.SHA256(),
backend=default_backend())

# Guardar certificado en un archivo
with open("certificate.pem", "wb") as cert_file:

cert_file.write(certificate.public_bytes(encoding=serialization.Encoding.PEM))
```

Generación de un certificado X.509 con mkcert
Instala mkcert desde (https://github.com/FiloSottile/mkcert)
Genera un certificado local válido con: `mkcert localhost`
Usa los archivos generados (por ejemplo, `localhost.pem` y `localhost-key.pem`)

2. Cifrado en Archivos y Datos Sensibles

Encriptación y Desencriptación de Archivos

El cifrado de archivos asegura que los datos almacenados en disco estén protegidos. En Python, se puede utilizar **PyCryptodome** o **cryptography** para cifrar y descifrar archivos.

Encriptación de archivo con AES (PyCryptodome):

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

# Generar clave y IV
key = get_random_bytes(16)  # Clave de 128 bits
iv = get_random_bytes(16)   # IV para el modo CBC

cipher = AES.new(key, AES.MODE_CBC, iv)

# Cifrado de archivo
with open("input.txt", "rb") as f:
    data = f.read()

ciphertext = cipher.encrypt(pad(data, AES.block_size))

with open("encrypted_file.enc", "wb") as f_enc:
    f_enc.write(iv + ciphertext)  # Guardar IV y cifrado

# Desencriptación
with open("encrypted_file.enc", "rb") as f_enc:
    iv = f_enc.read(16)
    ciphertext = f_enc.read()

cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_data = unpad(cipher.decrypt(ciphertext), AES.block_size)

with open("decrypted_file.txt", "wb") as f_dec:
    f_dec.write(decrypted_data)
```

Cifrado en Bases de Datos

El cifrado de datos en bases de datos es fundamental para proteger la información sensible. Puede cifrarse tanto a nivel de campo (datos sensibles como contraseñas, números de tarjetas) como a nivel de base de datos completa.

Cifrado de datos sensibles en base de datos con `cryptography`: Para cifrar datos antes de almacenarlos en la base de datos, se puede usar AES:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os

# Generar clave y IV
```

```
key = os.urandom(32)  # Clave de 256 bits
iv = os.urandom(16)   # IV para el modo CBC

# Función para cifrar
def encrypt_data(data):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(data.encode()) + encryptor.finalize()
    return ciphertext

# Almacenamiento cifrado en base de datos (simulado aquí con un archivo)
encrypted_data = encrypt_data("Datos sensibles")
with open("encrypted_data.db", "wb") as f:
    f.write(encrypted_data)
```

3. Comunicación Segura

Introducción a TLS/SSL

TLS (Transport Layer Security) y su predecesor **SSL (Secure Sockets Layer)** son protocolos criptográficos diseñados para proporcionar comunicaciones seguras a través de una red, como la web. TLS se utiliza ampliamente para asegurar las conexiones HTTPS en aplicaciones web.

TLS proporciona:

- **Confidencialidad**: Los datos se cifran durante la transmisión.
- **Integridad**: Se asegura que los datos no hayan sido alterados.
- **Autenticación**: Verifica la identidad de los participantes mediante certificados.

Uso de Sockets Seguros en Python

Python proporciona el módulo `ssl` que permite envolver sockets para realizar comunicaciones seguras a través de TLS.

Ejemplo de cliente y servidor con TLS en Python:

- **Servidor SSL/TLS:**

```
import socket
import ssl

# Crear el socket del servidor
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 12345))
server_socket.listen(1)

# Envolver el socket en SSL
ssl_socket = ssl.wrap_socket(server_socket, keyfile="server_key.pem",
certfile="server_cert.pem", server_side=True)

print("Esperando conexiones...")
client_socket, client_address = ssl_socket.accept()
print(f"Conexión desde {client_address}")

client_socket.send(b"Hola desde el servidor seguro")
client_socket.close()
```

- **Cliente SSL/TLS:**

```
import socket
import ssl

# Crear el socket del cliente
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Envolver el socket en SSL
ssl_socket = ssl.wrap_socket(client_socket, keyfile=None, certfile=None,
server_side=False, server_hostname="localhost")

# Conectar al servidor
ssl_socket.connect(('localhost', 12345))

# Recibir mensaje seguro
response = ssl_socket.recv(1024)
print("Mensaje del servidor:", response.decode())

ssl_socket.close()
```

En este ejemplo, el servidor y el cliente se comunican de forma segura mediante TLS. El servidor utiliza su certificado y clave privada para cifrar la comunicación y el cliente se conecta de forma segura al servidor.

---

4. Implementación de un servidor HTTPS en Python

Para crea un servidor http simple

```
$ python3 -m http.server 8000 --directory /ruta/a/tu/directorio
$ python3 -m http.server 8000
```

Uso de `ssl` y `http.server` para crear un servidor seguro

Python ofrece la biblioteca **ssl** que permite envolver un servidor HTTP en una capa de cifrado, convirtiéndolo en un servidor HTTPS. Junto con la librería **http.server**, que implementa un servidor web básico, podemos configurar un servidor HTTPS en pocos pasos.

Pasos para implementar un servidor HTTPS básico:

1. **Crear o conseguir un certificado SSL/TLS** (autofirmado o proporcionado por una Autoridad Certificadora).
2. **Utilizar la librería** `ssl` para envolver un servidor HTTP básico con cifrado TLS.
3. **Configurar el servidor para escuchar en el puerto 443** (el puerto predeterminado para HTTPS).

Ejemplo de implementación de servidor HTTPS en Python:

```
import http.server
import ssl

# Configuración del servidor
def run_https_server():
    # Crear el servidor HTTP básico
    server_address = ('', 4443)  # El puerto 4443 es usado para fines de
demostración
```

```
    httpd = http.server.HTTPServer(server_address,
http.server.SimpleHTTPRequestHandler)

    # Envolver el servidor HTTP con SSL/TLS
    # Nota: Asegúrate de tener un certificado y clave para usar
    httpd.socket = ssl.wrap_socket(httpd.socket,
                                   keyfile="path/to/private.key",
                                   certfile="path/to/certificate.crt",
                                   server_side=True)

    print("Servidor HTTPS corriendo en https://localhost:4443")
    httpd.serve_forever()

# Ejecutar el servidor HTTPS
run_https_server()
```

Explicación del código:

- **http.server.HTTPServer**: Este es un servidor HTTP básico de Python.
- **ssl.wrap_socket**: Este método envuelve el socket de red con cifrado SSL/TLS, utilizando el certificado y la clave privada proporcionados.
- **port 4443**: Por convención, los servidores HTTPS suelen operar en el puerto 443, pero para pruebas, puedes usar otros puertos (en este caso, 4443).
- **path/to/private.key** y **path/to/certificate.crt**: Estos son los archivos que contienen la clave privada y el certificado público. Puedes generar estos archivos de forma manual o utilizando herramientas como OpenSSL (ver más adelante).

Comunicaciones cifradas entre servidor y cliente:

Cuando el servidor esté funcionando, podrás conectarte a él desde un navegador usando la URL `https://localhost:4443`. El navegador mostrará una advertencia si el certificado es autofirmado, pero podrás continuar la conexión de todos modos.

---

5. Gestión de Claves

La correcta gestión de claves criptográficas es esencial para garantizar la seguridad de los sistemas criptográficos. Esto incluye su almacenamiento, rotación y expiración.

Almacenamiento Seguro de Claves

Las claves criptográficas deben almacenarse de forma segura para evitar su exposición a ataques. Las claves no deben ser guardadas en texto plano, ya que podrían ser fácilmente comprometidas si se obtiene acceso no autorizado al sistema.

Buenas prácticas para el almacenamiento de claves:

- **Almacenamiento en un hardware seguro (HSM)**: Un **Módulo de Seguridad de Hardware** (HSM) es un dispositivo físico que almacena las claves de manera segura y realiza operaciones criptográficas sin exponer la clave a la memoria del sistema.
- **Almacenamiento en un archivo cifrado**: Si no se dispone de un HSM, las claves pueden almacenarse de forma cifrada en archivos, utilizando un algoritmo de cifrado robusto.

- **Gestión de claves con bibliotecas especializadas**: Librerías como `cryptography` en Python pueden utilizarse para cifrar y gestionar claves.

Ejemplo de almacenamiento seguro de clave con AES:

```
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import scrypt
from Crypto.Random import get_random_bytes
import base64

# Generación de clave secreta
password = b"supersecreta"
salt = get_random_bytes(16)
key = scrypt(password, salt, dklen=32, N=2**14, r=8, p=1)

# Cifrado de una clave secreta
data_to_encrypt = b"claveSecretaParaServidor"
cipher = AES.new(key, AES.MODE_GCM)
ciphertext, tag = cipher.encrypt_and_digest(data_to_encrypt)

# Almacenamiento seguro
with open("secure_key_storage.txt", "wb") as f:
    f.write(base64.b64encode(cipher.nonce + tag + ciphertext))
```

Rotación y Expiración de Claves

Las claves criptográficas no deben usarse de forma indefinida. La **rotación de claves** implica reemplazar las claves antiguas por nuevas claves de manera periódica, mientras que la **expiración de claves** garantiza que las claves dejen de ser válidas después de un tiempo determinado.

Buenas prácticas para la rotación y expiración de claves:

- **Rotación automática**: Configura la rotación de claves para que las claves se cambien automáticamente después de un período determinado.
- **Expiración y revocación**: Establece fechas de expiración para las claves y utiliza mecanismos para revocar las claves comprometidas.

Ejemplo de código para rotar claves (simulando expiración):

```
import time

# Tiempo de expiración de la clave en segundos
key_expiration_time = 3600  # 1 hora

# Fecha de expiración
expiration_timestamp = time.time() + key_expiration_time

def is_key_expired(expiration_timestamp):
    if time.time() > expiration_timestamp:
        print("La clave ha expirado.")
        return True
    return False

# Verificar si la clave ha expirado
if is_key_expired(expiration_timestamp):
    # Rotar clave: generar nueva clave
    new_key = get_random_bytes(32)
    print("Clave rota y reemplazada.")
```

6. Ataques Comunes y Cómo Mitigarlos

Los sistemas criptográficos pueden ser vulnerables a varios tipos de ataques. Es crucial conocer estos ataques y cómo mitigarlos.

Fuerza Bruta

El **ataque de fuerza bruta** consiste en probar todas las combinaciones posibles hasta encontrar la correcta. Aunque los algoritmos de cifrado modernos son resistentes a ataques de fuerza bruta, la seguridad se ve comprometida si se utilizan claves débiles.

Mitigación contra ataques de fuerza bruta:

- Utilizar **claves largas y complejas**.
- Usar **algoritmos criptográficos robustos** como AES-256, que son más resistentes a ataques de fuerza bruta.
- Aplicar **limitaciones de intentos** de autenticación para prevenir ataques de fuerza bruta en sistemas de autenticación.

Ataques de Canal Lateral

Los **ataques de canal lateral** explotan información que se obtiene del comportamiento físico de los dispositivos al realizar operaciones criptográficas, como el tiempo de ejecución o las variaciones en el consumo de energía.

Mitigación de ataques de canal lateral:

- **Algoritmos resistentes**: Utilizar algoritmos diseñados para minimizar los riesgos de ataques de canal lateral, como la implementación constante en el tiempo de las operaciones criptográficas.
- **Uso de bibliotecas seguras**: Usar bibliotecas de criptografía que implementen contramedidas contra estos ataques.

Mitigación: Salting, Padding y Algoritmos Robustos

- **Salting**: Se recomienda añadir un **sal** único a los datos antes de realizar operaciones de hash (como contraseñas) para evitar ataques de diccionario y de tabla arco iris. El sal se debe almacenar junto al hash, pero debe ser único y aleatorio.

Ejemplo de salting en Python con `hashlib`:

```
import hashlib
import os

password = b"contraseñaSegura"
salt = os.urandom(16)  # Generación de un sal aleatorio
salted_password = salt + password
hashed_password = hashlib.sha256(salted_password).hexdigest()

print(f"Contraseña con sal: {hashed_password}")
```

- **Padding**: El **padding** garantiza que los bloques de datos tengan el tamaño adecuado para los algoritmos de cifrado. Es crucial para los modos de operación como **CBC** en AES.

Ejemplo de padding con `PyCryptodome`:

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

data = b"Datos sensibles"
key = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CBC)
ciphertext = cipher.encrypt(pad(data, AES.block_size))

# Desencriptación con unpad
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
```

- **Algoritmos robustos**: Usar algoritmos criptográficos robustos como **AES-256**, **RSA-2048**, y **ECDSA** para garantizar la resistencia frente a ataques.

7. Auditoría y Pruebas

Las auditorías y las pruebas son fundamentales para verificar la seguridad de los sistemas criptográficos. Las herramientas de pruebas pueden detectar vulnerabilidades antes de que sean explotadas por atacantes.

Pruebas de Robustez Criptográfica

Es importante realizar pruebas para verificar la robustez de los algoritmos criptográficos, especialmente cuando se integran en sistemas de producción.

Pruebas de robustez incluyen:

- **Análisis de la fuerza de las claves**: Verificar que las claves generadas sean suficientemente largas y seguras.
- **Análisis de vulnerabilidades en los algoritmos criptográficos**: Usar herramientas como **Hashcat** para realizar pruebas de fuerza bruta sobre algoritmos débiles.
- **Pruebas de resistencia a ataques de canal lateral**: Utilizar técnicas de análisis de tiempo y consumo de energía para verificar la resistencia a ataques de canal lateral.

Herramientas para Pentesting de Criptografía

El **pentesting** o prueba de penetración de la criptografía es crucial para identificar posibles fallos en la implementación de algoritmos criptográficos.

- **Hashcat**: Herramienta utilizada para realizar ataques de fuerza bruta sobre hashes.
- **John the Ripper**: Una de las herramientas más comunes para realizar ataques de fuerza bruta y de diccionario sobre contraseñas.
- **OpenSSL**: Se puede usar para generar y verificar la robustez de certificados y claves.

Ejemplo de uso de OpenSSL para verificar la robustez de una clave:

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:2048
openssl rsa -in private_key.pem -check
```