**Final Year B.Tech. (CSE) – VII [ 2024-25]**

**6CS451: Cryptography and Network Security Lab (C&NS Lab)**

**Date: 22/10/2024**

**Assignment 9**

**PRN:** 21510042  **Name:** Omkar Rajesh Auti

**Digital Signature System Documentation**

**Overview**

This Python program implements a digital signature system using RSA encryption and SHA-256 hashing. It provides the ability to generate RSA key pairs (private and public), sign a message by hashing it and creating a signature, verify the signature against the hash of the message, and save the RSA keys to files. The program is menu-driven and operates through a command-line interface.

**Key Concepts:**

* **RSA Encryption**: An asymmetric encryption method that uses a public-private key pair for secure operations.
* **SHA-256 Hashing**: A cryptographic hash function producing a 256-bit hash value to ensure message integrity.
* **Digital Signature**: A method of verifying the authenticity of a message using cryptographic signatures.

**Dependencies**

The program uses the following libraries:

* cryptography.hazmat.backends.default\_backend
* cryptography.hazmat.primitives.asymmetric.rsa
* cryptography.hazmat.primitives.asymmetric.padding
* cryptography.hazmat.primitives.hashes
* cryptography.hazmat.primitives.serialization
* cryptography.exceptions.InvalidSignature
* hashlib

To install the required libraries, use:

bash

Copy code

pip install cryptography

**Functions**

**1. generate\_keys()**

Generates a new RSA private and public key pair with a key size of 2048 bits.

* **Returns**:
  + private\_key: RSA private key object.
  + public\_key: RSA public key object.

**2. sign\_message(private\_key, message)**

Hashes the message using SHA-256 and signs the hash using the private key.

* **Parameters**:
  + private\_key: RSA private key used to sign the message.
  + message: The message to be signed.
* **Returns**:
  + signature: The digital signature.
  + message\_hash: The SHA-256 hash of the message in hexadecimal format.

**Note**: The message hash is printed to the console.

**3. verify\_signature(public\_key, message\_hash, signature)**

Verifies the signature by checking it against the provided message hash using the public key.

* **Parameters**:
  + public\_key: RSA public key used to verify the signature.
  + message\_hash: The SHA-256 hash of the message (hexadecimal format).
  + signature: The signature to verify.
* **Returns**:
  + True: If the signature is valid.
  + False: If the signature is invalid.

**4. save\_keys\_to\_file(private\_key, public\_key)**

Saves the RSA private and public keys to files in PEM format (private\_key.pem and public\_key.pem).

* **Parameters**:
  + private\_key: The RSA private key.
  + public\_key: The RSA public key.

The private key is saved in an unencrypted format for simplicity.

**Menu Interface**

The program provides a menu with the following options:

**1. Generate RSA Keys**

Generates a new RSA key pair (private and public).

**2. Sign a Message**

Prompts the user to input a message, hashes it, and signs the hash using the private key. The message hash is displayed for user reference.

**3. Verify Signature**

Prompts the user to input the hash of the message and verifies the signature using the public key.

**4. Save Keys to Files**

Saves the generated private and public keys to files (private\_key.pem and public\_key.pem).

**5. Exit**

Exits the program.

**Usage**

1. **Generating Keys**: After starting the program, select the option to generate RSA keys.
2. **Signing a Message**: Sign a message by selecting the appropriate menu option and entering the message.
3. **Verifying a Signature**: Verify the signature by inputting the hash of the message.
4. **Saving Keys**: Save the generated RSA keys to files for later use.

**Security Considerations**

* Ensure that private keys are kept secure and not shared.
* Private keys can be stored encrypted for additional security, though this implementation stores them unencrypted for simplicity.

Code:

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.exceptions import InvalidSignature

import hashlib

# Function to generate RSA private and public keys

def generate\_keys():

    private\_key = rsa.generate\_private\_key(

        public\_exponent=65537,

        key\_size=2048,

        backend=default\_backend()

    )

    public\_key = private\_key.public\_key()

    return private\_key, public\_key

# Function to sign a message and print its hash

def sign\_message(private\_key, message):

    # Hash the message using SHA-256

    message\_hash = hashlib.sha256(message.encode()).hexdigest()

    # Sign the hashed message using RSA private key

    signature = private\_key.sign(

        bytes.fromhex(message\_hash),  # Convert the hex hash to bytes

        padding.PSS(

            mgf=padding.MGF1(hashes.SHA256()),

            salt\_length=padding.PSS.MAX\_LENGTH

        ),

        hashes.SHA256()

    )

    print(f"Hash of the message: {message\_hash}")  # Print the message hash

    return signature, message\_hash

# Function to verify the signature using the provided hash

def verify\_signature(public\_key, message\_hash, signature):

    try:

        # Verify the signature using RSA public key

        public\_key.verify(

            signature,

            bytes.fromhex(message\_hash),  # Convert the hex hash back to bytes

            padding.PSS(

                mgf=padding.MGF1(hashes.SHA256()),

                salt\_length=padding.PSS.MAX\_LENGTH

            ),

            hashes.SHA256()

        )

        return True

    except InvalidSignature:

        return False

# Function to save keys to files

def save\_keys\_to\_file(private\_key, public\_key):

    # Save private key

    with open("private\_key.pem", "wb") as private\_file:

        private\_file.write(

            private\_key.private\_bytes(

                encoding=serialization.Encoding.PEM,

                format=serialization.PrivateFormat.PKCS8,

                encryption\_algorithm=serialization.NoEncryption()

            )

        )

    # Save public key

    with open("public\_key.pem", "wb") as public\_file:

        public\_file.write(

            public\_key.public\_bytes(

                encoding=serialization.Encoding.PEM,

                format=serialization.PublicFormat.SubjectPublicKeyInfo

            )

        )

    print("Keys saved to files: private\_key.pem and public\_key.pem")

# Menu for the digital signature system

def menu():

    private\_key, public\_key = None, None

    signature = None

    message\_hash = None

    while True:

        print("\n===== Digital Signature System =====")

        print("1. Generate RSA Keys")

        print("2. Sign a Message")

        print("3. Verify Signature")

        print("4. Save Keys to Files")

        print("5. Exit")

        choice = input("Enter your choice (1/2/3/4/5): ")

        if choice == '1':

            # Generate RSA private and public keys

            private\_key, public\_key = generate\_keys()

            print("\nRSA Keys Generated!")

        elif choice == '2':

            # Sign a message

            if private\_key is None:

                print("You need to generate RSA keys first.")

            else:

                message = input("Enter the message to sign: ")

                signature, message\_hash = sign\_message(private\_key, message)

                print("\nMessage signed successfully!")

        elif choice == '3':

            # Verify the signature

            if public\_key is None or message\_hash is None or signature is None:

                print("You need to sign a message first.")

            else:

                input\_hash = input("Enter the hash of the message to verify: ")

                verification\_result = verify\_signature(public\_key, input\_hash, signature)

                if verification\_result:

                    print("\nSignature verified successfully! The message is authentic.")

                else:

                    print("\nSignature verification failed! The message is not authentic.")

        elif choice == '4':

            # Save RSA keys to files

            if private\_key is None or public\_key is None:

                print("You need to generate RSA keys first.")

            else:

                save\_keys\_to\_file(private\_key, public\_key)

        elif choice == '5':

            print("Exiting the program...")

            break

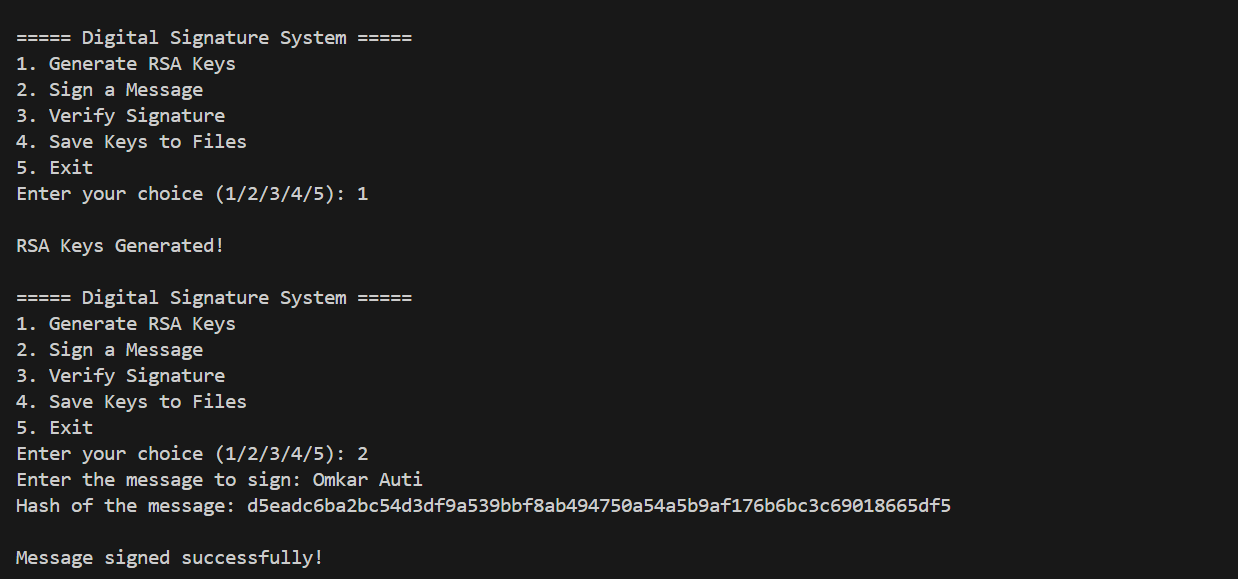
        else:

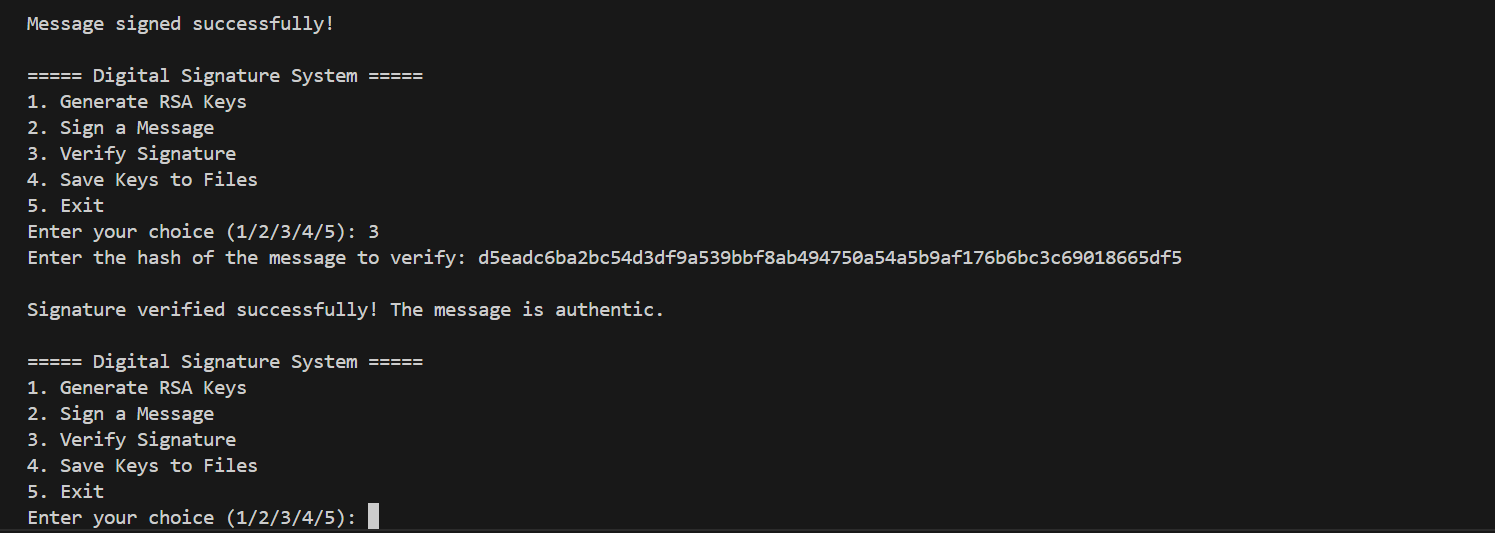
            print("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    menu()

Output:





VLAB

