Project Title: SecureX Encryptor

Message Encryption with Secure Key Exchange and Tamper Detection

Developed By- Vaibhav Kumar Sahu

Abstract

The **SecureX Encryptor** project is designed to provide a secure message encryption system, combining multiple cryptographic techniques to address modern cybersecurity challenges. The project implements **RSA** (Rivest-Shamir-Adleman) for secure key exchange, **AES** (Advanced Encryption Standard) for message encryption, and **HMAC** (Hash-based Message Authentication Code) for integrity verification. The goal of this project is to ensure that sensitive messages can be securely transmitted, with tamper detection and key protection, offering a comprehensive encryption solution suitable for real-world applications.

Introduction

In today's digital age, the need for secure communication is paramount. Cyber threats, including man-in-the-middle attacks, data breaches, and tampering, have made traditional communication methods inadequate for safeguarding sensitive information. The **SecureX Encryptor** project aims to address these vulnerabilities by combining powerful encryption and integrity verification mechanisms to ensure data confidentiality and authenticity.

This project uses the following cryptographic techniques:

- **RSA encryption** for public-key cryptography to securely exchange keys.
- **AES encryption** for efficiently encrypting and decrypting messages.
- **HMAC** to ensure that the messages have not been tampered with during transmission.

Objectives

The key objectives of the **SecureX Encryptor** project are:

1. **Secure Message Encryption**: Encrypt messages securely using AES to protect sensitive information.
2. **Secure Key Exchange**: Use RSA encryption to securely transmit the AES encryption key.
3. **Tamper Detection**: Use HMAC to ensure the integrity of the message and detect any tampering attempts.
4. **Cryptographic Best Practices**: Implement secure cryptographic standards to protect data during transmission.

Technical Stack

- **Programming Language**: Python
- **Libraries Used**:
  - cryptography for RSA and AES encryption.
  - hashlib and hmac for message integrity using HMAC.
- **Security Standards**:
  - RSA key generation: 2048-bit keys.
  - AES encryption using symmetric keys.
  - SHA-256 hashing for HMAC.

System Design

The project is designed to work in a client-server model where:

- **Sender** encrypts the message using AES and sends the encrypted message, HMAC, and the RSA-encrypted AES key.
- **Receiver** decrypts the AES key using RSA, verifies the integrity of the message with HMAC, and then decrypts the actual message.

The system operates in three main phases:

1. **Key Generation and Exchange**:
   - RSA keys are generated and saved as public and private key files (public_key.pem and private_key.pem).
   - The AES key is encrypted with the RSA public key before transmission.
2. **Message Encryption and Decryption**:
   - The sender encrypts the message using AES and the generated AES key.
   - HMAC is generated for the encrypted message to ensure it hasn't been altered.
   - The encrypted AES key is sent with the message to the receiver.
   - The receiver uses the private RSA key to decrypt the AES key, verifies the HMAC, and decrypts the original message.
3. **Tamper Detection**:
   - HMAC is used to verify the message integrity. If the HMAC doesn't match during decryption, an alert is triggered indicating possible tampering.

Detailed Explanation of the Code

1. RSA Key Generation

The function generate_rsa_key() generates a pair of RSA keys:

- A **public key** for encryption.
- A **private key** for decryption.
  These keys are saved as PEM files for future use.

Code :-

```
def generate_rsa_key():
    private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    public_key = private_key.public_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()))

    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("RSA keys generated and saved as 'private_key.pem' and 'public_key.pem'.")
```

2. AES Message Encryption

The function aes_encrypt_message() encrypts a message using AES. It generates a random AES key for each message encryption.

- The message is then encrypted with the generated AES key using the **Fernet** module.

Code :-

```
def aes_encrypt_message(message):
    aes_key = Fernet.generate_key()
    cipher = Fernet(aes_key)
    encrypted_message = cipher.encrypt(message.encode())
    return aes_key, encrypted_message
```

3. HMAC for Message Integrity

The function create_hmac() generates a HMAC for a given message and key. The HMAC ensures the integrity of the encrypted message during transmission.

Code :-

```
def create_hmac(message, key):
    return hmac.new(key, message, hashlib.sha256).hexdigest()
```

4. RSA Key Encryption and Decryption

The AES key is encrypted using the RSA public key before transmission. The receiver uses the RSA private key to decrypt the AES key.

Code :-

```
def rsa_encrypt_key(aes_key, public_key_path):
    with open(public_key_path, "rb") as key_file:
        public_key = serialization.load_pem_public_key(key_file.read())

    encrypted_aes_key = public_key.encrypt(
        aes_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return encrypted_aes_key
```


5. Decrypting the Message

The function decrypt_message() decrypts the message in the following steps:

1. Decrypt the AES key using RSA.
2. Verify the integrity of the message using HMAC.
3. Finally, decrypt the message using AES.

Code :-

```
def decrypt_message(encrypted_message, encrypted_aes_key, private_key_path, hmac_key,
received_hmac):
    aes_key = rsa_decrypt_key(encrypted_aes_key, private_key_path)

    # Verify HMAC
    if not verify_hmac(encrypted_message, hmac_key, received_hmac):
        raise ValueError("Message integrity check failed. Possible tampering detected!")

    # Decrypt the message
    return aes_decrypt_message(encrypted_message, aes_key)
```

User Interface

The project allows the user to choose between three options:

1. **Generate RSA keys**.
2. **Encrypt a message**.
3. **Decrypt a message**.

The user is prompted to input the necessary details such as the message, HMAC passphrase, and the encrypted message for decryption.

Output:-

```
1. Generate RSA Keys
2. Encrypt a Message
3. Decrypt a Message
Enter your choice:
```

Testing and Output

- **Generating RSA Keys**:
  Running the program with option 1 generates the public and private keys and stores them as files.
- **Encrypting a Message**:
  When option 2 is selected, the user is prompted to enter a message. The program returns

the encrypted message, the HMAC of the encrypted message, and the RSA-encrypted AES key.

- **Decrypting a Message**:
  In option 3, the user inputs the encrypted message and HMAC. The system validates the HMAC and, if correct, decrypts the message, showing the original text.

Future Enhancements

1. **User Authentication**: Add a secure method for authenticating the sender and receiver.
2. **Multithreading**: Implement support for multiple simultaneous encryption/decryption sessions.
3. **GUI Interface**: Develop a user-friendly graphical interface for easier usage.
4. **Cloud Integration**: Extend the system to support cloud-based storage of encrypted messages.

Conclusion

The **SecureX Encryptor** project successfully integrates modern cryptographic techniques to secure communication and ensure data integrity. By combining RSA, AES, and HMAC, the system provides a robust solution for encrypting messages, securely exchanging keys, and detecting tampering. This project offers a high level of security for sensitive data transmission, making it a strong candidate for real-world applications in secure messaging systems.
