SECURE OTP GENERATION AND DISTRIBUTION SYSTEM

MOHAMMAD SHAMIM HOSSAIN

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Project Overview

The Secure OTP Generation and Distribution System is a robust solution designed to enhance the security of OTP (One-Time Password) generation and distribution. OTPs play a important role in secure authentication, and this project aims to elevate their security levels through advanced cryptographic techniques and decentralized architecture.

1.2 Project Goals

- Enhance security in OTP generation and distribution.
- Implement RSA encryption for secure communication.
- Develop a decentralized system for key generation and distribution.
- Integrate with existing authentication systems like KERBEROS.

1.3 System Components

The system comprises several key components:

- **Key Generation Clients**: These entities are responsible for generating secret phrases and communicating securely with the server.
- **Key Generation Server**: A central server that distributes public keys and receives encrypted secret phrases.
- **OTP Requesting Clients**: End-users who request and use OTPs for authentication.
- **RSA Encryption**: The RSA cryptographic algorithm is used for secure communication between clients and the server.
- **Secret Phrase Generator**: A module that generates secret phrases based on user input and selected hash algorithms.
- **Key Distribution Server**: A server that securely distributes OTPs to authorized clients.
- **Secure Multiparty Computation**: Techniques are employed to ensure the security and decentralization of the key generation process.

1.4 Technologies Used

The following technologies are used to implement the system:

- **Python**: The primary programming language for implementing the system.
- **RSA cryptography library**: Used for generating RSA keys and encryption/decryption.
- **hashlib**: Utilized for hashing operations.
- **Socket programming**: Facilitates communication between clients and the server.

2. SYSTEM ARCHITECTURE

2.1 Key Generation Clients

Key generation clients are responsible for generating secret phrases and securely transmitting them to the server. These clients can use different algorithms to generate secret phrases, enhancing the security of OTPs.

2.2 Key Generation Server

The key generation server manages the distribution of public keys and receives encrypted secret phrases. It plays a central role in the secure distribution of OTPs to clients.

2.3 OTP Requesting Clients

OTP requesting clients are end-users who connect to the OTP distribution server to receive OTPs. These clients can use the OTPs for secure authentication.

2.4 RSA Encryption

The RSA cryptographic algorithm is employed for secure communication between key generation clients and the server. RSA ensures that data transmitted between clients and the server remains confidential.

2.5 Secret Phrase Generator

The secret phrase generator module generates secret phrases based on user input and selected hash algorithms. Clients use these secret phrases as OTPs for authentication.

2.6 Key Distribution Server

The key distribution server securely distributes OTPs to authorized clients. It plays a crucial role in ensuring that OTPs are delivered securely.

2.7 Secure Multiparty Computation

The system uses secure multiparty computation techniques to decentralize the key generation process. This decentralization reduces the risk of compromise and enhances security.

3. CODE SNIPPETS

Let's explore code snippets from various components of the Secure OTP Generation and Distribution System.

3.1. Key_Generation_Client

Let's take **client.py** as an example. This client script generates secret phrases and communicates with the server.

3.1.1 Code Snippet: Client.py

```
import socket
import rsa
from Secret_Phrase_Generator import generate_int_from_random
from RSA_encrypt_decrypt import encrypt
```

The code begins by importing necessary Python modules:
socket*:* Used for creating network sockets and communication.
rsa: A library for working with RSA encryption.
generate_int_from_random: A custom function from the 'Secret_Phrase_Generator' module, which generates an integer from random data.
encrypt: A custom function from the 'RSA_encrypt_decrypt' module for encrypting data using RSA encryption.

```
server_ip = '127.0.0.1'
server_port = 55540
client_id = '1'

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_ip, server_port))
```

This section defines the IP address and port of the server to connect to, as well as a client identifier. A socket object is created using the socket module and then connected to the specified server address and port.

```
public_key_bytes_received = sock.recv(4096)

received_public_key = rsa.PublicKey.load_pkcs1(public_key_bytes_received)

print("Received Public Key: " + str(received_public_key))

with open("public_Key_Server.pem", 'wb') as p:
    p.write(received_public_key.save_pkcs1('PEM'))
```

The script receives a public key from the server as bytes, then loads and converts it to a rsa.PublicKey object. It also saves this public key to a file named "public_Key_Server.pem."

```
with open("public_Key_Server.pem", 'wb') as p:
    p.write(received_public_key.save_pkcs1('PEM'))

while True:
    try:
        user_input = int(input("Please enter an integer Value: "))
        break

    except ValueError:
        print("Invalid input. Please enter a valid integer.")

print("You entered:", user_input)
```

The code asks the user to enter an integer and repeatedly asks until get a valid input.

```python
# This is the generated X
secret_phrase = str(generate_int_from_random(send, int(client_id)))
print(f'The secret phrase is: {secret_phrase}')

# This is the encrypted X
encrypted_data = encrypt(secret_phrase, received_public_key)

print("This is the encrypted format of the secret phrase")
print(encrypted_data)

sock.send(encrypted_data)
sock.send(client_id.encode('utf-8'))

sock.close()
```

The code creates a secret phrase then encrypts it with the received public key. After that secret phrase and Clients ID are sent to the server. At the end, the connection is closed.

3.1.2 Secret_Phrse_Generator.py

```python
import hashlib


def generate_int_from_random(random_value, client_id):
    value = client_id
    if value == 1:
        print("Use SHA-256")
        random_bytes = str(random_value).encode('utf-8')
        hash_object = hashlib.sha256()
        hash_object.update(random_bytes)
        hash_hex = hash_object.hexdigest()
        generated_int = int(hash_hex, 16)
        return generated_int
    elif value == 2:
        print("Use SHA-512")
        random_bytes = str(random_value).encode('utf-8')
        hash_object = hashlib.sha512()
        hash_object.update(random_bytes)
        hash_hex = hash_object.hexdigest()
        generated_int = int(hash_hex, 16)
        return generated_int
    elif value == 3:
        print("Use SHA-384")
        random_bytes = str(random_value).encode('utf-8')
        hash_object = hashlib.sha384()
        hash_object.update(random_bytes)
        hash_hex = hash_object.hexdigest()
        generated_int = int(hash_hex, 16)
        return generated_int
    elif value == 4:
        print("Use SHA-512")
        random_bytes = str(random_value).encode('utf-8')
        hash_object = hashlib.sha512()
        hash_object.update(random_bytes)
        hash_hex = hash_object.hexdigest()
        generated_int = int(hash_hex, 16)
        return generated_int
```

The generate_int_from_random function generates a secret integer based on a random value and client ID. Different hash algorithms are used based on the clients.

3.1.2. Code Snippet: Client2.py

Similar to Client.py with a different client_id.

3.1.3. Code Snippet: Client3.py

Similar to Client.py with a different client_id.

3.1.4. Code Snippet: Client4.py

Similar to Client.py with a different client_id.

3.2. Key_Generation_Server

3.2.1 Generate_Server_key.py

```
import rsa


def generate_server_keys():
    public_key, private_key = rsa.newkeys(2048)
    print(public_key)
    print(private_key)

    with open('public_Key_Server.pem', 'wb') as p:
        p.write(public_key.save_pkcs1('PEM'))

    with open('private_Key_Server.pem', 'wb') as p:
        p.write(private_key.save_pkcs1('PEM'))


generate_server_keys()
```

- This script generates RSA public and private keys for the server.
- The keys are saved in PEM format.

3.2.2. Response_Phrse_Generator.py

```
import hashlib


def generate_int_from_random(random_value):
    random_bytes = str(random_value).encode('utf-8')
    hash_object = hashlib.sha256()
    hash_object.update(random_bytes)
    hash_hex = hash_object.hexdigest()
    generated_int = int(hash_hex, 16)
    return generated_int
```

The generate_int_from_random function generates integer based on a random value.

3.2.3. RSA_encrypt_decrypt.py

```
# Use pip install rsa if you do to have rsa in your local pc.
import rsa


def generateKeys():
    (publicKey, privateKey) = rsa.newkeys(1024)
    with open('publicKey.pem', 'wb') as p:
        p.write(publicKey.save_pkcs1('PEM'))
    with open('privateKey.pem', 'wb') as p:
        p.write(privateKey.save_pkcs1('PEM'))


def loadKeys():
    with open('public_Key_Server.pem', 'rb') as p:
        publicKey = rsa.PublicKey.load_pkcs1(p.read())
    with open('private_Key_Server.pem', 'rb') as p:
        privateKey = rsa.PrivateKey.load_pkcs1(p.read())
    return privateKey, publicKey


def encrypt(message, key):
    return rsa.encrypt(message.encode('ascii'), key)


def decrypt(ciphertext, key):
    try:
        return rsa.decrypt(ciphertext, key).decode('ascii')
    except:
        return False
```

RSA is a widely used encryption algorithm for securing data. Here is a description of the functions in this code:

generateKeys():
This function generates a pair of RSA keys, namely a public key and a private key, each containing 1024 bits.
The keys are saved in format into two separate files: 'publicKey.pem' and 'privateKey.pem'.

loadKeys():
This function loads RSA keys from previously generated PEM files. It reads the public key from 'public_Key_Server.pem' and the private key from 'private_Key_Server.pem'.The loaded keys are returned as a tuple containing the private key and the public key.

encrypt(message, key):
This function is used to encrypt a given message using an RSA key. It takes the message (in ASCII format) and the key (either public or private) as input. The message is converted into bytes and then encrypted using the RSA key.

decrypt(ciphertext, key):

This function is used to decrypt a ciphertext using an RSA key. It takes the ciphertext and the key (either private or public) as input. The ciphertext is decrypted using the RSA key, and the result is returned as a decoded ASCII string. If decryption fails for any reason, it returns False.

3.2.4. Server.py

```
import socket
import RSA_encrypt_decrypt
import Response_Phrase_Generator

private_key, public_key = RSA_encrypt_decrypt.loadKeys()

public_key_bytes = public_key.save_pkcs1(format='PEM')
private_key_bytes = private_key.save_pkcs1(format='PEM')

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind(('127.0.0.1', 55540))
server_socket.listen(5)

while True:
    print("Waiting for Connections from Trusted Authorities...........")
    connection, client_address = server_socket.accept()
    print(f"Connected to {client_address}")

    print("Connected to the Trusted Authority!")

    connection.sendall(public_key_bytes)
    received_encrypted_data = connection.recv(4096)
    client_id = int(connection.recv(1024).decode('utf-8'))
    print("This is the received encrypted secret phrase:")
    print(received_encrypted_data)
    print(f"Client ID: {client_id}")

    # This is X
    decrypted_secret_phrase = RSA_encrypt_decrypt.decrypt(received_encrypted_data,
private_key)

    print("The decrypted secret data:")
    print(decrypted_secret_phrase)
    # This is the y
    response_secret_phrase =
Response_Phrase_Generator.generate_int_from_random(decrypted_secret_phrase)

    with open("key_store.txt",'a') as file:
        file.write(str(response_secret_phrase) + "\n")

    connection.close()
```

- This script sets up a socket server to listen for client connections.
- It sends the server's public key to clients.
- Upon receiving encrypted data from clients, it decrypts the data using the server's private key.
- The decrypted data is used to generate a response phrase, which is stored in key_store.txt.

3.3. OTP Resuesting Clients

3.3.1. Client.py

```
import socket
salt = 5000908999
client_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

client_socket.connect(('127.0.0.1', 55465))
data = client_socket.recv(1024).decode('ascii')

print("This is the received OTP")
print(int(data)-salt)
```

- This script connects to the OTP distribution server.
- It receives an OTP, subtracts a salt value, and prints the resulting OTP.

4. THE PRACTICAL IMPLEMENTATION OF THAT SYSTEM

In this practical implementation, we will integrate the Decentralised OTP Generator system as a token generation service within the KDC, enabling secure and dynamic token-based authentication for users.

Components:

KDC (Key Distribution Server): This is the central authentication server responsible for granting access tokens to users.

Decentralised OTP Generator System: This system includes Key Generation Clients, a Key Generation Server, and OTP Requesting Clients. It is responsible for generating, encrypting, and distributing OTPs securely.

Implementation Steps:

1. Public Key Exchange:
   - Key Generation Clients initiate communication with the Key Generation Server.
   - The clients request the public key from the server.
   - The server sends its public key to the clients.

2. User Input and Secret Phrase Generation:
   - Each Key Generation Client takes an integer input from the user.
   - Using the provided integer, the client generates a secret phrase by combining it with a random number and the client's unique identifier (client_id).
   - The client determines which hash algorithm to use based on its client_id.
   - The secret phrase is hashed using the selected algorithm.

3. RSA Encryption:
   - The client encrypts the secret phrase using the server's public key.
   - This ensures that the secret phrase is securely transmitted to the server.

4. Data Transmission to Server:
   - The client sends the encrypted data (secret phrase) to the Key Generation Server.
   - The client also sends its client_id for identification.

5. Decryption and Hashing on Server:
   - The Key Generation Server receives the encrypted data and decrypts it using its private key.
   - The server hashes the decrypted data multiple times.
   - The resulting value is stored in the Key Distribution Server.

6. OTP Request by Client:
   - When a Key Requesting Client needs an OTP, it sends a request to the Key Distribution Server.

7. OTP Distribution:
   - The Key Distribution Server responds with the OTP data, including a salt value.
   - The Key Requesting Client receives the OTP data.

8. Salt Removal and OTP Usage:
   - The Key Requesting Client extracts the salt value from the received data.
   - The OTP is calculated by removing the salt from the received data.

5. ENHANCING SECURITY AND COMPLEXITY

Enhanced Cryptographic Techniques:
- Utilize stronger cryptographic algorithms for OTP generation and encryption. Consider using algorithms like SHA-256 or AES for added security.
- Implement hashing with multiple iterations (salting and hashing multiple times) to increase the complexity of secret phrase storage on the server.

Dynamic OTP Generation:
- Implement dynamic OTP generation based on real-time variables such as time, location, or device identifiers. This makes OTPs time-sensitive and location-bound, adding complexity for attackers.

Security Audits and Penetration Testing:
- Conduct regular security audits and penetration testing to identify vulnerabilities and weaknesses in the system.
- Hire third-party security experts to perform thorough assessments.

Secure Communication Protocols:
- Ensure that all communication between clients and servers is encrypted using strong encryption protocols like TLS/SSL.

6. CONCLUSION

The Cryptography Project provides a secure and decentralized system for key generation and OTP distribution. By implementing various algorithms and encryption techniques, this project ensures the security and reliability of cryptographic operations. The system ensures the confidentiality and integrity of user access to diverse applications and services.

Future Improvements:
In the future, my plan to improve the system by:
I'm working on making practical apps that are super easy to use. Exploring advanced encryption techniques to further secure data. Supporting additional cryptographic algorithms to provide more options for users.

Overall Summary:
This project offers a versatile solution for secure key generation and OTP distribution. Its practical applications extend to authentication, Single Sign-In, and secure communication. By maintaining decentralization and continuously improving security measures, the project aims to meet the evolving demands of cryptography in various domains.