# Name :- Sarthak Pagar

# Roll No. :- 40

# Class :- TE (IT)

# Practical B1 :- Implement a client and a server on different computers using python. Perform the communication between these two entities by using RSA cryptosystem.

**server.py**

import time

import socket

import sys

import pickle

# from util.HashAlgo import HashAlgo

from util.Operations import Operations

from util.RSA import RSA

# from util.SAES import SAES

class Server:

def \_\_init\_\_(self):

print("[STARTING] Server is starting...")

self.PORT = 5050

# Local IP Addess of the host

self.SERVER\_IP = socket.gethostbyname(socket.gethostname())

self.ADDR = (self.SERVER\_IP, self.PORT)

self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.server\_socket.bind(self.ADDR)

def listen(self):

print(f'[LISTENING] Server listening on {self.SERVER\_IP}...')

self.server\_socket.listen()

self.conn, self.addr = self.server\_socket.accept()

return self.conn, self.addr

def inputKeyParameters(self):

print("Enter the space seperated key parameters p, q and e:")

self.p, self.q, self.e = map(int, input().split())

def generateServerKeys(self):

self.private\_key, self.public\_key = RSA.generateKeys(

self.p, self.q, self.e)

def recieveMsg(self):

msg = self.conn.recv(1024)

msg = pickle.loads(msg)

return msg

def sendMsg(self, data):

data = pickle.dumps(data)

self.conn.send(data)

def decrypt(self, priv\_key, c\_text):

d, n = priv\_key

x = ''

m = 0

print("cipher text", c\_text)

for i in c\_text:

if (i == 400):

x += ' '

else:

m = (int(i) \*\* d) % n

m += 65

c = chr(m)

x += c

return x

server\_obj = Server()

conn, add = server\_obj.listen()

print("[Connected] Connection created with IP: {} on PORT: {}".format(add[0], add[1]))

server\_obj.inputKeyParameters()

# Now verify key parameters

is\_verified = RSA.verifyParameters(server\_obj.p, server\_obj.q, server\_obj.e)

if not is\_verified:

server\_obj.inputKeyParameters()

server\_obj.generateServerKeys()

print("Server Private key, Server Public Key", server\_obj.private\_key, server\_obj.public\_key)

# Recieving client's request

msg = server\_obj.recieveMsg()

if msg == 'Y':

server\_obj.sendMsg(server\_obj.public\_key)

print(f'[Sending] Public key to {server\_obj.ADDR} (Client)...')

else:

print("[Sever] Closing the connection...")

server\_obj.server\_socket.close()

# Recieving client's message

print("Receiving Ciphertext from Client")

ciphertext = server\_obj.recieveMsg()

plaintext = server\_obj.decrypt(server\_obj.private\_key, ciphertext)

print("Plaintext is ", plaintext)

**client.py**

import socket

import sys

import pickle

# from util.HashAlgo import HashAlgo

from util.Operations import Operations

from util.RSA import RSA

# from util.SAES import SAES

class Client:

def \_\_init\_\_(self):

print("[STARTING] Client is starting...")

self.PORT = 5050

# Local IP Addess of the host

self.SERVER\_IP = socket.gethostbyname(socket.gethostname())

self.ADDR = (self.SERVER\_IP, self.PORT)

self.client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

def connect(self):

print(f'[Connecting] Client trying to connect {self.SERVER\_IP}...')

self.client\_socket.connect(self.ADDR)

print(f'[Connected] Secure connection established with server.')

def sendMsg(self, data):

data = pickle.dumps(data)

self.client\_socket.send(data)

def recieveMsg(self):

msg = self.client\_socket.recv(1024)

msg = pickle.loads(msg)

return msg

def inputMessage(self):

print("Enter your message to be send to the server:")

self.message = input()

def inputKeyParameters(self):

print("Enter the space seperated key parameters p, q and e:")

self.p, self.q, self.e = map(int, input().split())

def generateClientKeys(self):

self.private\_key, self.public\_key = RSA.generateKeys(self.p, self.q, self.e)

def encrypt(self, pub\_key, n\_text):

e, n = pub\_key

x = []

m = 0

for i in n\_text:

if (i.isupper()):

m = ord(i) - 65

c = (m \*\* e) % n

x.append(c)

elif (i.islower()):

m = ord(i) - 97

c = (m \*\* e) % n

x.append(c)

elif (i.isspace()):

spc = 400

x.append(spc)

return x

client\_obj = Client()

client\_obj.connect()

print("Do you want to request server for it's public key? Y or N")

res = input()

if res.lower() == 'y':

# Requesting server for it's public key

print("[Requesting] Server's public key...")

client\_obj.sendMsg("Y")

# Recieving server's public key

client\_obj.server\_public\_key = client\_obj.recieveMsg()

print("Server's public key recieved!", client\_obj.server\_public\_key)

# Workflow continues

client\_obj.inputMessage()

# Creating ciphertext

print("\n[Client] Encrypting...")

ciphertext = client\_obj.encrypt(client\_obj.server\_public\_key, client\_obj.message)

print("Ciphertext is ", ciphertext)

print("Sending Ciphertext to server....")

client\_obj.sendMsg(ciphertext)

else:

print("[Client] Closing the connection...")

client\_obj.client\_socket.close()

**util/Operations.py**

# Operations.py

class Operations:

@staticmethod

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

@staticmethod

def modInverse(e, phi):

# Extended Euclidean Algorithm

def egcd(a, b):

if a == 0:

return b, 0, 1

g, y, x = egcd(b % a, a)

return g, x - (b // a) \* y, y

g, x, \_ = egcd(e, phi)

if g != 1:

raise Exception('modular inverse does not exist')

else:

return x % phi

@staticmethod

def isPrime(n):

if n <= 1:

return False

if n <= 3:

return True

if n % 2 == 0 or n % 3 == 0:

return False

i = 5

while i \* i <= n:

if n % i == 0 or n % (i + 2) == 0:

return False

i += 6

return True

**util/RSA.py**

# RSA.py

from util.Operations import Operations

class RSA:

@staticmethod

def generateKeys(p, q, e):

n = p \* q

phi = (p - 1) \* (q - 1)

if Operations.gcd(e, phi) != 1:

raise ValueError("e and φ(n) must be coprime.")

d = Operations.modInverse(e, phi)

public\_key = (e, n)

private\_key = (d, n)

return private\_key, public\_key

@staticmethod

def verifyParameters(p, q, e):

if not (Operations.isPrime(p) and Operations.isPrime(q)):

print("[RSA] p and q must be prime.")

return False

phi = (p - 1) \* (q - 1)

if e <= 1 or e >= phi:

print("[RSA] e must be in range (1, φ(n)).")

return False

if Operations.gcd(e, phi) != 1:

print("[RSA] e must be coprime with φ(n).")

return False

return True

**Output :-**



