# Name :- Sarthak Pagar

# Roll No. :- 40

# Class :- TE (IT)

# Practical B2 :- Implement a client and a server on different computers using python. Perform the authentication of sender between these two entities by using RSA digital signature cryptosystem.

**server.py**

# server.py

import socket

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

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 separated 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 workFlow(self):

data = self.recieveMsg()

client\_public\_key = data['client\_public\_key']

ciphertext = data['ciphertext']

client\_signature = data['client\_signature']

encrypted\_secret\_key = data['secret\_key']

SAES.is\_padded = data['padded']

# Decrypt secret key

decrypted\_secret\_key = int(RSA.decrypt(self.private\_key, encrypted\_secret\_key))

print("\nDecrypted secret key:", decrypted\_secret\_key)

# Generate subkeys and reverse them for decryption

print("\n[Server] Decrypting client's message...")

subkeys = SAES.generate\_subkeys(decrypted\_secret\_key)

subkeys[0], subkeys[4] = subkeys[4], subkeys[0]

subkeys[1], subkeys[5] = subkeys[5], subkeys[1]

# Convert hex back to binary

ciphertext\_bin = ['{:016b}'.format(int(h, 16)) for h in ciphertext]

plaintext = SAES.decrypt(ciphertext\_bin, subkeys)

print("Decrypted plaintext:", plaintext)

if SAES.is\_padded:

plaintext = plaintext[:-1]

# Generate digest

hash\_code = HashAlgo.generateHashCode(message=plaintext)

print(f"\nMessage digest: {hash\_code}")

# Verify signature

is\_verified = RSA.verify(client\_public\_key, hash\_code=hash\_code, client\_sign=client\_signature)

print("Signature verified:", is\_verified)

self.server\_socket.close()

# =============================

server\_obj = Server()

conn, add = server\_obj.listen()

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

server\_obj.inputKeyParameters()

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

if not is\_verified:

server\_obj.inputKeyParameters()

server\_obj.generateServerKeys()

msg = server\_obj.recieveMsg()

if msg == 'Y':

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

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

server\_obj.workFlow()

else:

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

server\_obj.server\_socket.close()

**client.py**

# 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

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 inputMessage(self):

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

self.message = input()

def inputKey(self):

print("Enter the secret key. Value should be in range 0 to {} as the key size is {}:".format(2 \*\* SAES.key\_size - 1, SAES.key\_size))

self.key = int(input())

if (self.key < 0) or (self.key > (2 \*\* SAES.key\_size - 1)):

print("Follow the rules for the key")

exit(1)

def inputKeyParameters(self):

print("Enter the space separated 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 recieveMsg(self):

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

msg = pickle.loads(msg)

return msg

def sendMsg(self, data):

data = pickle.dumps(data)

self.client\_socket.send(data)

def \_ciphertextHex(self, ciphertext):

ciphertext\_hex = []

for i in ciphertext:

ciphertext\_hex.append("{:04x}".format(int(i, 2)))

return ciphertext\_hex

def workFlow(self):

# Encrypting secret key

encrypted\_secret\_key = RSA.encrypt(self.server\_public\_key, str(self.key))

print("\nEncrypted secret key:", RSA.printHexList(encrypted\_secret\_key))

# Creating ciphertext

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

subkeys = SAES.generate\_subkeys(self.key)

ciphertext = SAES.encrypt(self.message, subkeys)

ciphertext\_hex = self.\_ciphertextHex(ciphertext)

print('Ciphertext:', ''.join(ciphertext\_hex))

# Generating digest

hash\_code = HashAlgo.generateHashCode(message=self.message)

print(f"\nMessage digest: {hash\_code}")

# Creating digital signature using hash code

client\_sign = RSA.sign(self.private\_key, hash\_code)

print("Client signature:", RSA.printHexList(client\_sign))

# arranging datas to be send to server

data = {

'secret\_key': encrypted\_secret\_key,

'ciphertext': ciphertext\_hex,

'client\_signature': client\_sign,

'client\_public\_key': self.public\_key,

'padded': SAES.is\_padded

}

# Sending data to server

self.sendMsg(data)

self.client\_socket.close()

# =============================

client\_obj = Client()

client\_obj.connect()

client\_obj.inputMessage()

client\_obj.inputKey()

client\_obj.inputKeyParameters()

# Now verify key parameters

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

if not is\_verified:

client\_obj.inputKeyParameters()

# Generate public and private keys

client\_obj.generateClientKeys()

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

res = input()

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

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

client\_obj.sendMsg("Y")

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

print("Server's public key received!")

# Proceed to workflow

client\_obj.workFlow()

else:

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

client\_obj.client\_socket.close()

**util/HashAlgo.py**

# util/HashAlgo.py

import hashlib

class HashAlgo:

@staticmethod

def generateHashCode(message):

# SHA-256 hash of the message

return hashlib.sha256(message.encode()).hexdigest()

**util/Operations.py**

# util/Operations.py

class Operations:

@staticmethod

def sampleOperation():

return "Just a placeholder"

**util/RSA.py**

# util/RSA.py

class RSA:

@staticmethod

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

@staticmethod

def modInverse(e, phi):

for d in range(1, phi):

if (e \* d) % phi == 1:

return d

return None

@staticmethod

def encrypt(key, plaintext):

e, n = key

return [pow(ord(char), e, n) for char in plaintext]

@staticmethod

def decrypt(key, ciphertext):

d, n = key

return ''.join([chr(pow(char, d, n)) for char in ciphertext])

@staticmethod

def generateKeys(p, q, e):

n = p \* q

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

d = RSA.modInverse(e, phi)

if d is None:

raise Exception("Modular inverse not found")

return (d, n), (e, n) # private, public

@staticmethod

def sign(private\_key, hash\_code):

d, n = private\_key

return [pow(ord(char), d, n) for char in hash\_code]

@staticmethod

def verify(public\_key, hash\_code, client\_sign):

e, n = public\_key

decrypted\_hash = ''.join([chr(pow(char, e, n)) for char in client\_sign])

return decrypted\_hash == hash\_code

@staticmethod

def printHexList(lst):

return [hex(i) for i in lst]

@staticmethod

def verifyParameters(p, q, e):

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

return RSA.gcd(e, phi) == 1

**util/SAES.py**

# util/SAES.py

class SAES:

key\_size = 4 # 4-bit key size for example/demo

is\_padded = False

@staticmethod

def pad\_message(msg):

SAES.is\_padded = False

if len(msg) % 2 != 0:

msg += ' '

SAES.is\_padded = True

return msg

@staticmethod

def generate\_subkeys(key):

# Dummy subkeys for example

return [key] \* 6

@staticmethod

def encrypt(message, subkeys):

message = SAES.pad\_message(message)

binary\_cipher = []

for char in message:

bin\_char = format(ord(char) ^ subkeys[0], '04b')

binary\_cipher.append(bin\_char)

return binary\_cipher

@staticmethod

def decrypt(ciphertext, subkeys):

decrypted = ''

for bin\_char in ciphertext:

xor\_val = int(bin\_char, 2) ^ subkeys[0]

decrypted += chr(xor\_val)

return decrypted

**Output :-**



