CSE 350 Network Security
Programming Assignment Number 3

Akshat Saini 2020019 Nakul Thureja 2020528

Project Number = (0019 + 0528) mod 2 = 1
Project 1: RSA-based Public Key Distribution Authority (PKDA)

Command to run:

1. **python3 PKDA_019_528.py**
2. **python3 ClientB_019_528.py**
3. **python3 ClientA_019_528.py**

Make sure to run the three commands on different terminals and in order

Assumptions

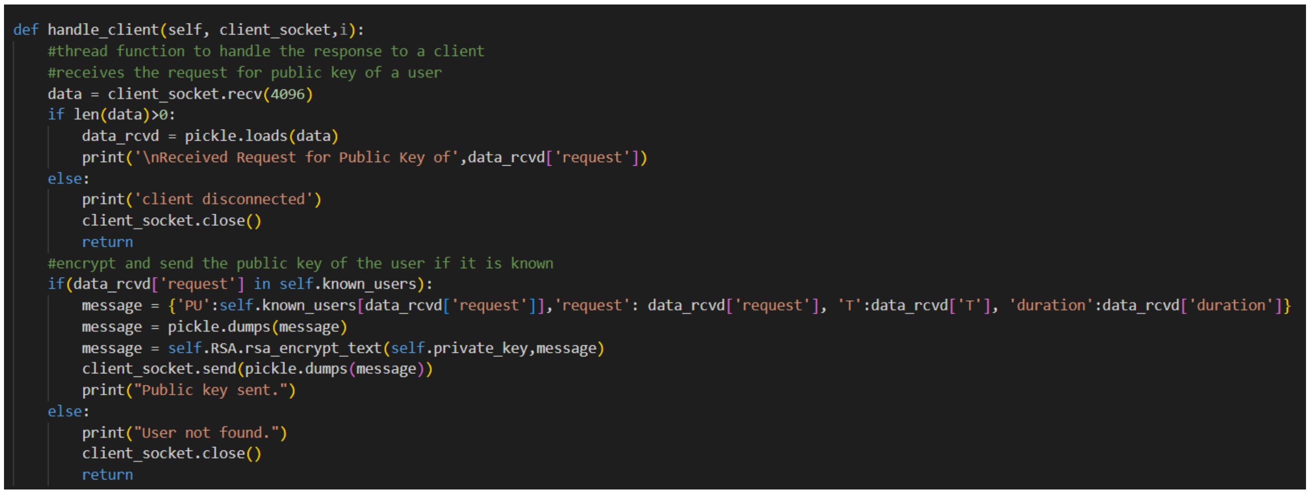
- The Clients/PKDA are self-responsible for generating their pairs of keys, viz. [private-key, public-key].
- The sharing of public keys takes place through sockets before any communication takes place, and the keys are assumed to be correct
- The code works for general cases, but the testing has been coded to simulate the flow provided in the assignment.

System Designed

The system design consists of a PKDA and 2 Clients which can communicate with each other using sockets.

1. PKDA

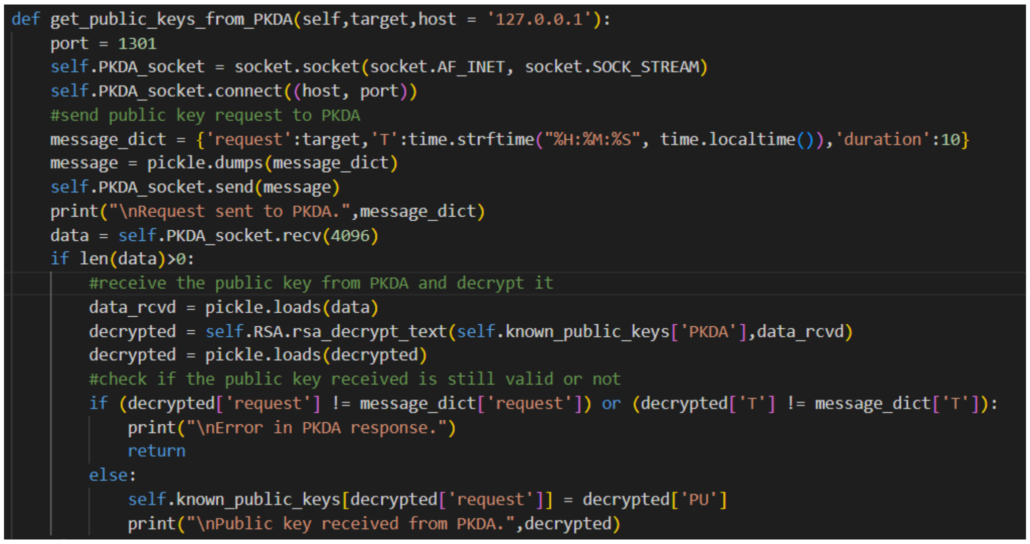
The PKDA is capable of responding to clients that seek their own public-key certificates or that of other clients.



```python
def handle_client(self, client_socket,i):
    #thread function to handle the response to a client
    #receives the request for public key of a user
    data = client_socket.recv(4096)
    if len(data)>0:
        data_rcvd = pickle.loads(data)
        print('\nReceived Request for Public Key of',data_rcvd['request'])
    else:
        print('client disconnected')
        client_socket.close()
        return
    #encrypt and send the public key of the user if it is known
    if(data_rcvd['request'] in self.known_users):
        message = {'PU':self.known_users[data_rcvd['request']],'request': data_rcvd['request'], 'T':data_rcvd['T'], 'duration':data_rcvd['duration']}
        message = pickle.dumps(message)
        message = self.RSA.rsa_encrypt_text(self.private_key,message)
        client_socket.send(pickle.dumps(message))
        print("Public key sent.")
    else:
        print("User not found.")
        client_socket.close()
        return
```

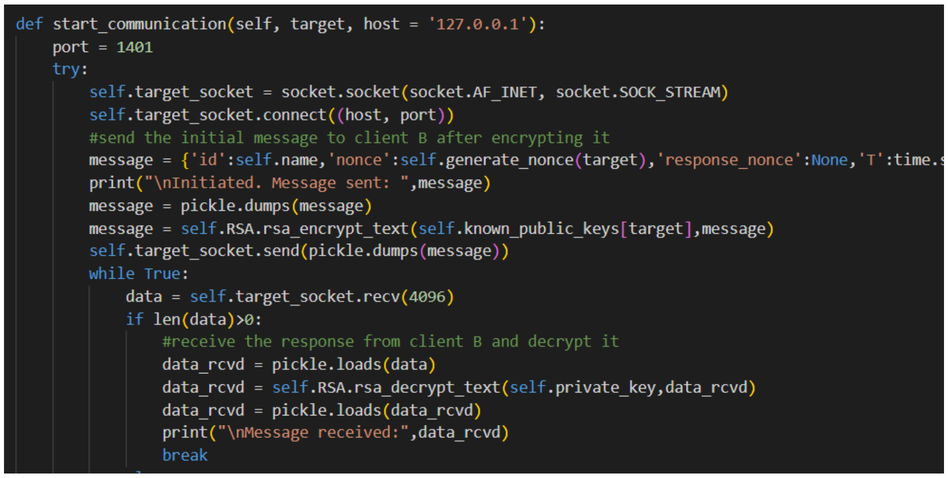
The PKDA consists of a socket that waits for any client request for a public key. If a request arrives, a new thread is created. Based on the request sent, the PKDA goes through its saved public keys. The response of PKDA consists of the requested public key, along with the time of the client request and also the duration. This message as a whole is encrypted using RSA and sent back.

2. ClientA and ClientB



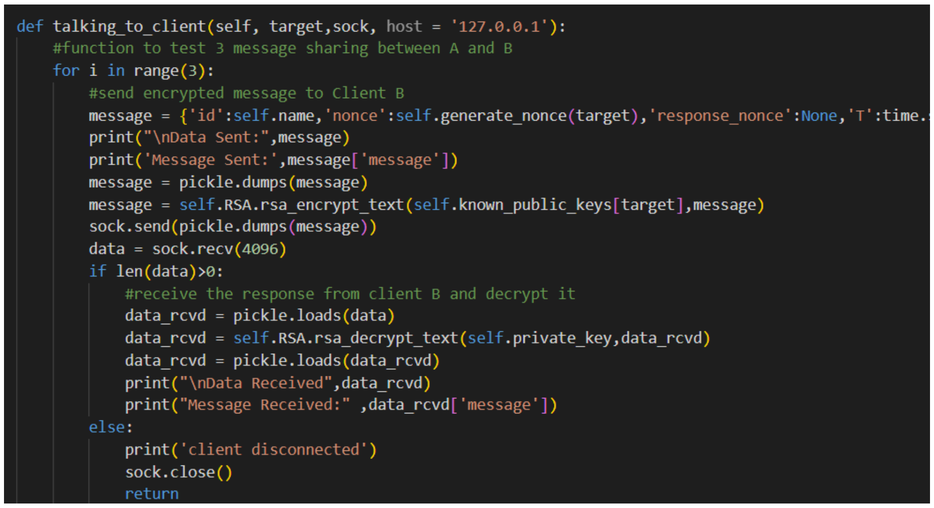
```python
def get_public_keys_from_PKDA(self,target,host = '127.0.0.1'):
    port = 1301
    self.PKDA_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.PKDA_socket.connect((host, port))
    #send public key request to PKDA
    message_dict = {'request':target,'T':time.strftime("%H:%M:%S", time.localtime()),'duration':10}
    message = pickle.dumps(message_dict)
    self.PKDA_socket.send(message)
    print("\nRequest sent to PKDA.",message_dict)
    data = self.PKDA_socket.recv(4096)
    if len(data)>0:
        #receive the public key from PKDA and decrypt it
        data_rcvd = pickle.loads(data)
        decrypted = self.RSA.rsa_decrypt_text(self.known_public_keys['PKDA'],data_rcvd)
        decrypted = pickle.loads(decrypted)
        #check if the public key received is still valid or not
        if (decrypted['request'] != message_dict['request']) or (decrypted['T'] != message_dict['T']):
            print("\nError in PKDA response.")
            return
        else:
            self.known_public_keys[decrypted['request']] = decrypted['PU']
            print("\nPublic key received from PKDA.",decrypted)
```

This function is responsible for getting the public key from PKDA.
The function first generates a request containing the ID of the client for whom the public key is required and also the current time and duration of the message. After that, the function waits for the PKDA to respond. Once the response is received, the data is decrypted, and the received public key is stored for future use.



```python
def start_communication(self, target, host = '127.0.0.1'):
    port = 1401
    try:
        self.target_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.target_socket.connect((host, port))
        #send the initial message to client B after encrypting it
        message = {'id':self.name,'nonce':self.generate_nonce(target),'response_nonce':None,'T':time.s
        print("\nInitiated. Message sent: ",message)
        message = pickle.dumps(message)
        message = self.RSA.rsa_encrypt_text(self.known_public_keys[target],message)
        self.target_socket.send(pickle.dumps(message))
        while True:
            data = self.target_socket.recv(4096)
            if len(data)>0:
                #receive the response from client B and decrypt it
                data_rcvd = pickle.loads(data)
                data_rcvd = self.RSA.rsa_decrypt_text(self.private_key,data_rcvd)
                data_rcvd = pickle.loads(data_rcvd)
                print("\nMessage received:",data_rcvd)
                break
```

This function is responsible for the communication between A and B such that steps (3), (6) and (7) are completed. The code differs for ClientA and ClientB. The ClientA function first sends a message to Client B containing its ID, nonce, time and duration, encrypted with the public key of B. After that, B first decrypts the message from A using its private key and then gets the public key of A from PKDA and then sends a response back to A, which contains the response to A's original nonce and also its own nonce, with the time and the duration encrypted with the public key of A. A, on receiving the message from B, decrypts it using its own private key, checks the validity of the nonce and sends a response back to B with a response for B's nonce. B again receives the message, decrypts it and checks for the validity of the nonce. Once this process is concluded, A and B are free to communicate.
Note that the nonces are randomly generated, and their responses are produced through hashing using SHA256.

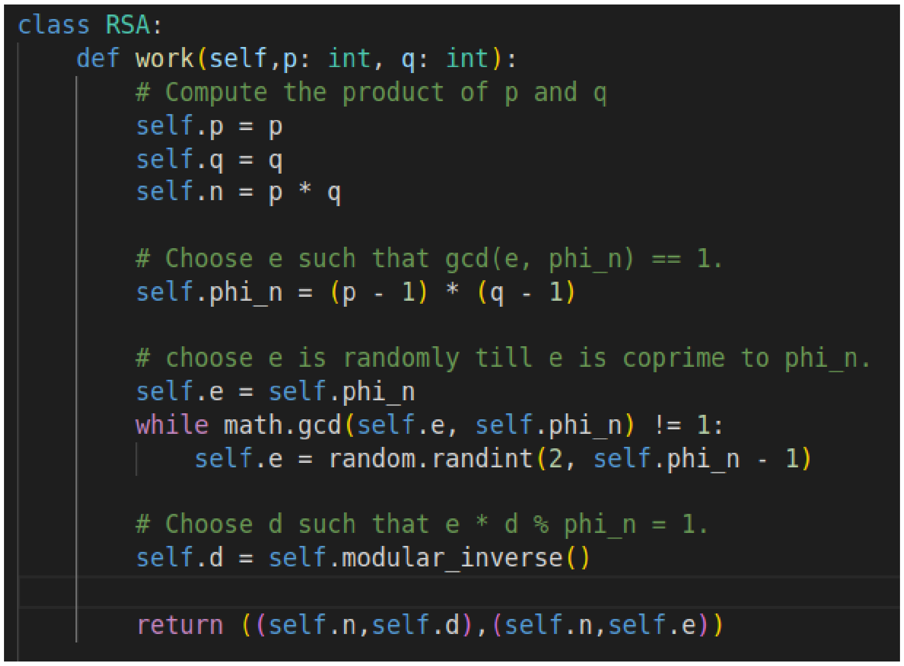


```
def talking_to_client(self, target,sock, host = '127.0.0.1'):
    #function to test 3 message sharing between A and B
    for i in range(3):
        #send encrypted message to Client B
        message = {'id':self.name,'nonce':self.generate_nonce(target),'response_nonce':None,'T':time.
        print("\nData Sent:",message)
        print('Message Sent:',message['message'])
        message = pickle.dumps(message)
        message = self.RSA.rsa_encrypt_text(self.known_public_keys[target],message)
        sock.send(pickle.dumps(message))
        data = sock.recv(4096)
        if len(data)>0:
            #receive the response from client B and decrypt it
            data_rcvd = pickle.loads(data)
            data_rcvd = self.RSA.rsa_decrypt_text(self.private_key,data_rcvd)
            data_rcvd = pickle.loads(data_rcvd)
            print("\nData Received",data_rcvd)
            print("Message Received:" ,data_rcvd['message'])
        else:
            print('client disconnected')
            sock.close()
            return
```

This is the test function that sends three messages from ClientA and three responses from ClientB. All the messages are encrypted using the public key of the other client and then decrypted using their own private key. Each message contains the nonce, time and duration along with the message - Hi/Got-it. Nonce validity checks are also maintained for each response.

3. RSA

We have created an RSA class which takes p and q as inputs, assuming p and q are different prime numbers. Now we find n = p*q and phi_n = p-1*q-1. We find e such that it is coprime to n. Then we find d such that e*d mod n = 1. Now to encrypt a string, we iterate over the characters and convert it into int (m) and then find encryption by m^e mod n and convert it into characters again. Similarly, to decrypt a string, we iterate over the characters and convert it into int (c) and then find decryption by m^d mod n and convert it into characters again.



```python
class RSA:
    def work(self,p: int, q: int):
        # Compute the product of p and q
        self.p = p
        self.q = q
        self.n = p * q

        # Choose e such that gcd(e, phi_n) == 1.
        self.phi_n = (p - 1) * (q - 1)

        # choose e is randomly till e is coprime to phi_n.
        self.e = self.phi_n
        while math.gcd(self.e, self.phi_n) != 1:
            self.e = random.randint(2, self.phi_n - 1)

        # Choose d such that e * d % phi_n = 1.
        self.d = self.modular_inverse()

        return ((self.n,self.d),(self.n,self.e))
```