

Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre



A Report
on
“Assignment 1: Secure communication between a client and a server”

[Course Code: COMP 492]

[For the partial fulfillment of 4th year/2nd Semester in Computer Engineering]

Submitted by
Sabin Thapa
Roll No.: 54
Registration No.: 024358-18

Submitted to
Suresh Gautam sir
Department of Computer Science and Engineering

Submission Date: February 3, 2023

Background

A company is developing a new messaging service that allows clients to communicate securely with a server. The company wants to ensure that all messages sent between the client and the server are encrypted and secure.

Assignment

Your task is to write an implementation of the AES or RSA encryption algorithm in Python that will be used to encrypt and decrypt messages sent between the client and the server. The implementation should include the ability to generate a key for encryption/decryption and should be able to handle large amounts of data. The code should also include a mechanism for securely exchanging the encryption key between the client and the server.

Introduction

In order to implement secure communication between the server and the client, I've used the RSA(Rivest-Shamir-Adleman) encryption algorithm. The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). The algorithm encrypts and decrypts messages sent between the client and the server. The RSA algorithm works as follows:

Algorithm

1. Choose two large prime numbers p & q
2. Compute $n=pq$ and $z=(p-1)(q-1)$
3. Choose number e , less than n , which has no common factor (other than 1) with z
4. Find number d , such that $ed - 1$ is exactly divisible by z
5. Keys are generated using n, d, e
 - i. Public key is (n,e)
 - ii. Private key is (n, d)
6. Encryption: $c = me \bmod n$
 - i. m is plain text
 - ii. c is cipher text
7. Decryption: $m = cd \bmod n$
8. The public key is shared and the private key is hidden

RSA is an asymmetric encryption algorithm in which both the public and private keys are interchangeable and the variable key size is 512, 1024, or 2048 bits. It is the most common general key algorithm.

Implementation

The program is capable of encrypting and decrypting data exchanged between the client and the server using the RSA algorithm. The encryption key is generated randomly using the *RSA* package (from *Crypto* in *Python*). The detailed implementation is discussed below:

1. Public and private keys are generated for both the client and the server randomly.
2. The keys are stored in their respective *pem* (Privacy Enhanced Mail) files and the exchange of keys between the client and server is accomplished by importing the respective keys.
3. To initiate secure communication, both the client and the server need to be connected to the same IP and PORT.
4. After a successful connection, a client can send messages which are encrypted using the public key of the server.
5. The server then receives the message from the client and it can be decrypted using the private key of the server.
6. Similarly, the server can send an encrypted message using the public key of the client and the client can decrypt the cipher and decode it.

Working

First, the encryption keys are generated randomly using the *RSA* module of the *Crypto* package. The module *Crypto.PublicKey.RSA* provides facilities for generating new *RSA* keys, reconstructing them from known components, exporting them, and importing them. The encryption keys (both public and private) are generated and exported to their respective *pem* files.

The following function, `keys_generator()` is used to generate the keys:

assignments > Chapter_1-A_1 > encryption_keys_generator.py

You, 23 hours ago | 1 author (You)

```
1  from Crypto.PublicKey import RSA
2
3  def keys_generator():
4      key_for_server = RSA.generate(1024)
5      key_for_client = RSA.generate(1024)
6
7      '''Keys Generation'''
8
9      #SERVER
10     server_private_key = key_for_server.exportKey('PEM')
11     server_public_key = key_for_server.publickey().exportKey('PEM')
12
13     #CLIENT
14     client_private_key = key_for_server.exportKey('PEM')
15     client_public_key = key_for_server.publickey().exportKey('PEM')
16
17     '''Keys Storage'''
18
19     # SERVER - PRIVATE KEY
20     with open('server_private_key.pem', 'wb') as file: # 'wb' for writing in the binary file
21         file.write(server_private_key)
22         file.close()
23
24     # SERVER - PUBLIC KEY
25     with open('server_public_key.pem', 'wb') as file:
26         file.write(server_public_key)
27         file.close()
28
29     # CLIENT - PRIVATE KEY
30     with open('client_private_key.pem', 'wb') as file:
31         file.write(client_private_key)
32         file.close()
33     # CLIENT - PUBLIC KEY
34     with open('client_public_key.pem', 'wb') as file:
35         file.write(client_public_key)
36         file.close()
37
```

You, 23 hours ago * feat: add encryption keys gen

Fig: Encryption keys generation

The next task is to implement the client-server architecture. It is accomplished by using Python's socket module. The TCP flow of the socket is shown in the picture below. The left-hand column represents the server and the right-hand column represents the client.

Using the socket module, the connection is set up using a host IP address and the port number. After both the server and the client connect to the same IP and port, communications can begin. The client can then securely send messages to the server and vice-versa. In this way, the mechanism for secure text messaging in the company is implemented.

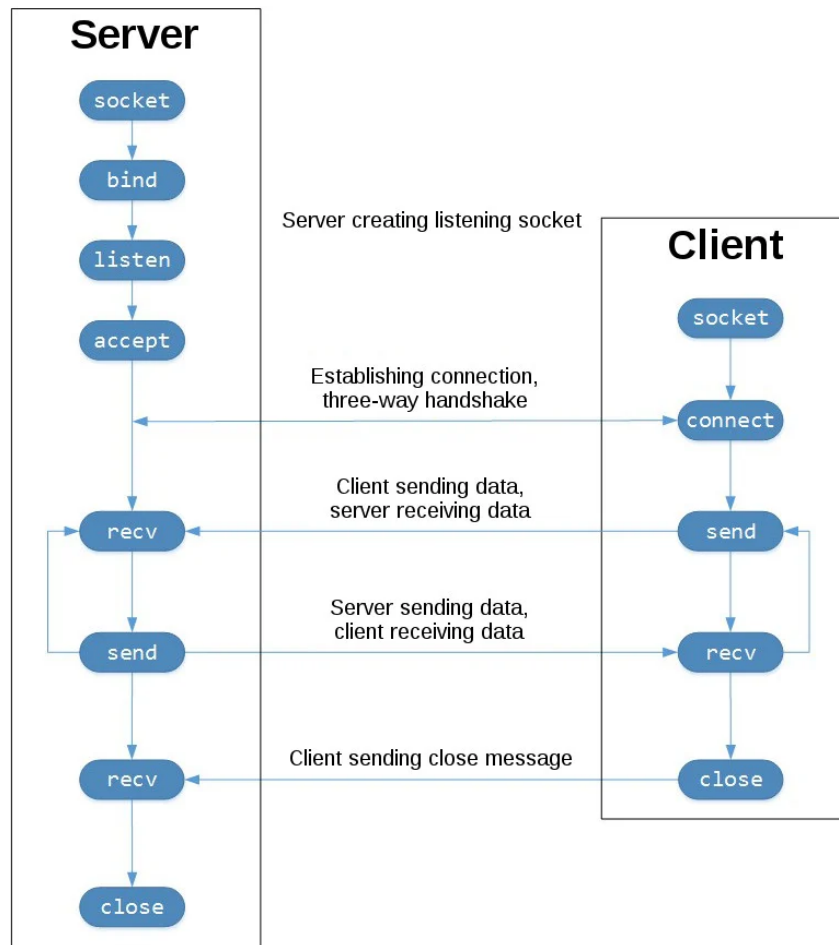


Fig: TCP Socket Flow ([source](#))

The PKCS#1 OAEP module is used for encryption and decryption. It is an asymmetric cipher based on RSA and the OAEP padding. It can only encrypt messages slightly shorter than the RSA modulus (a few hundred bytes). Using this, we can encrypt data by means of the recipient's public key and the recipient uses its own private key to decrypt the message which is stored in the pem file.

```

server_private_key = RSA.import_key(open('server_private_key.pem').read())
server_private_key = PKCS1_OAEP.new(server_private_key)

client_public_key = RSA.import_key(open('client_public_key.pem').read())
client_public_key = PKCS1_OAEP.new(client_public_key)
  
```

Fig: Use of PKCS#1 Module

```

with conn:
    print(f'Connection established with {addr}')
    init_msg = b'Server waves hi!'
    init_msg = client_public_key.encrypt(init_msg)

    conn.sendall(init_msg)

    while True:
        data = conn.recv(2048)
        data = server_private_key.decrypt(data)
        if not data:
            print('Connection Terminated! No data from client!')
            break
        print('Client: ' + str(data))
        data = input('Message: ')
        data = client_public_key.encrypt(data.encode())
        conn.sendall(data)
    conn.close()

```

Fig: Use of PKCS#1 Module

Here, as we can see in the figure above, the server encrypts the message using the client's public and then it decrypts the message using its own private key.

Testing

In order to check whether the implementation is correct or not, unit testing is performed. The *unittest* library of Python is used to do so. In this implementation, testing can be performed to check whether the encrypted message from one node is decrypted properly in another node or not. So, test cases are written both for the server and the client.

```

msg_from_client = 'I am a client'
msg_from_server = 'I am a server'

#Test Cases
class TestSecureCommunication(unittest.TestCase):
    def test_server_encryption(self):
        self.ciphertext = server_public_key.encrypt(msg_from_client.encode())
        self.plaintext = server_private_key.decrypt(self.ciphertext)
        self.plaintext = self.plaintext.decode('utf-8')
        self.assertEqual(self.plaintext, msg_from_client)

    def test_client_encryption(self):
        self.ciphertext = client_public_key.encrypt(msg_from_server.encode())
        self.plaintext = client_private_key.decrypt(self.ciphertext)
        self.plaintext = self.plaintext.decode('utf-8')
        self.assertEqual(self.plaintext, msg_from_server)

```

Fig: Unit Testing

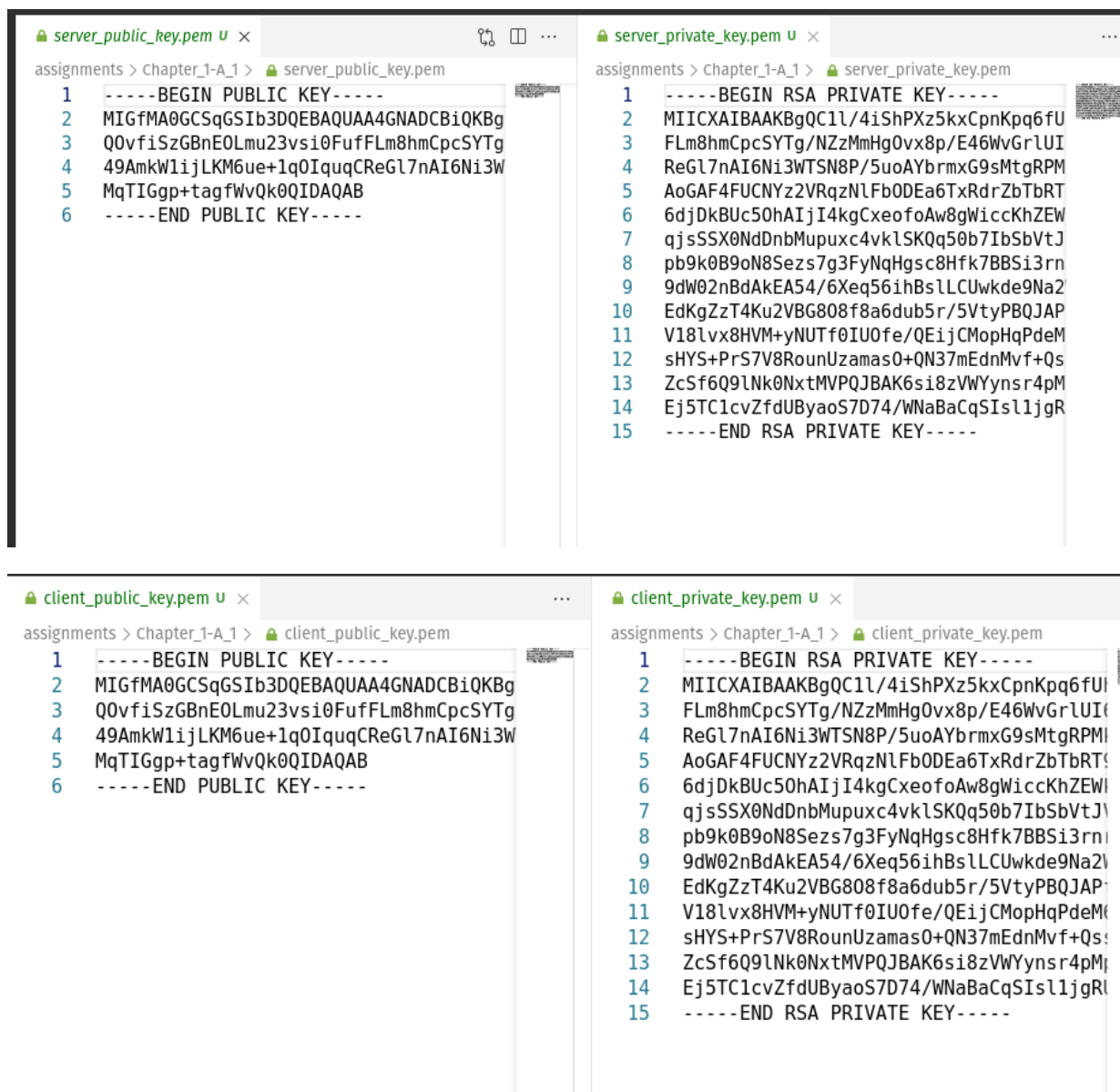
Output

The output of the program is shown below:

First, the server.py file is run which generates all the required keys and starts the server at a dedicated host and port.

```
o sabinthapa pop-os ../Chapter_1-A_1 main venv python3 server.py
Enter the HOST IP: (default: 127.0.0.1):
Enter the PORT number: 8080
Server has started listening at: 127.0.0.1:8080
Waiting for client ...
```

Fig: Starting the server



```
server_public_key.pem
1 -----BEGIN PUBLIC KEY-----
2 MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBg
3 Q0vfiSzGBnEOLmu23vsi0FufFLm8hmCpcSYTg
4 49AmkW1ijLKM6ue+1q0IquqCReG17nAI6Ni3W
5 MqTIGgp+tagfWvQk0QIDAQAB
6 -----END PUBLIC KEY-----

server_private_key.pem
1 -----BEGIN RSA PRIVATE KEY-----
2 MIICXAIBAAKBgQC1L/4iShPXz5kxCpnKpq6fU
3 FLm8hmCpcSYTg/NZzMMHg0vx8p/E46WvGrLUI
4 ReG17nAI6Ni3WTSN8P/5uoAYbrmxG9sMtgRPM
5 AoGAF4FUCNYz2VRqzNlFb0DEa6TxRdrZbTbRT
6 6djDkBUc50hAIjI4kgCxeofAw8gWiccKhZEW
7 qjsSSX0NdDnbMupuxc4vkLSKQq50b7IbSbVtJ
8 pb9k0B9oN8Sezs7g3FyNqHgsc8Hfk7BBSi3rn
9 9dW02nBdAkeA54/6Xeq56ihBsLLCUwkde9Na2
10 EdKgZzT4Ku2VBG808f8a6dub5r/5VtyPBQJAP
11 V18lvx8HVM+yNUTf0IU0fe/QEijCMopHqPdeM
12 sHYS+PrS7V8RounUzamas0+QN37mEdnMvf+Qs
13 ZcSf6Q9lNk0NxtMVPQJBAK6si8zVWYynsr4pM
14 Ej5TC1cvZfdUByaoS7D74/WNaBaCqSIs1ljgR
15 -----END RSA PRIVATE KEY-----

client_public_key.pem
1 -----BEGIN PUBLIC KEY-----
2 MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBg
3 Q0vfiSzGBnEOLmu23vsi0FufFLm8hmCpcSYTg
4 49AmkW1ijLKM6ue+1q0IquqCReG17nAI6Ni3W
5 MqTIGgp+tagfWvQk0QIDAQAB
6 -----END PUBLIC KEY-----

client_private_key.pem
1 -----BEGIN RSA PRIVATE KEY-----
2 MIICXAIBAAKBgQC1L/4iShPXz5kxCpnKpq6fU
3 FLm8hmCpcSYTg/NZzMMHg0vx8p/E46WvGrLUI
4 ReG17nAI6Ni3WTSN8P/5uoAYbrmxG9sMtgRPM
5 AoGAF4FUCNYz2VRqzNlFb0DEa6TxRdrZbTbRT
6 6djDkBUc50hAIjI4kgCxeofAw8gWiccKhZEW
7 qjsSSX0NdDnbMupuxc4vkLSKQq50b7IbSbVtJ
8 pb9k0B9oN8Sezs7g3FyNqHgsc8Hfk7BBSi3rn
9 9dW02nBdAkeA54/6Xeq56ihBsLLCUwkde9Na2
10 EdKgZzT4Ku2VBG808f8a6dub5r/5VtyPBQJAP
11 V18lvx8HVM+yNUTf0IU0fe/QEijCMopHqPdeM
12 sHYS+PrS7V8RounUzamas0+QN37mEdnMvf+Qs
13 ZcSf6Q9lNk0NxtMVPQJBAK6si8zVWYynsr4pM
14 Ej5TC1cvZfdUByaoS7D74/WNaBaCqSIs1ljgR
15 -----END RSA PRIVATE KEY-----
```

Fig: Encrypted keys saved as files

The client is then connected to the server as shown below:

```
sabinthapa pop-os > ../Chapter_1-A_1 main > venv > python3 client.py
Enter the HOST IP: (default: 127.0.0.1):
Enter the PORT number: 8080
Server : Server waves hi!
Message: █
```

Fig: Client connecting to the server

After a successful connection, the client can then send a message to the server securely and communication can take place.

```
sabinthapa pop-os > ../Chapter_1-A_1 main > venv > python3 server.py
Enter the HOST IP: (default: 127.0.0.1):
Enter the PORT number: 8080
Server has started listening at: 127.0.0.1:8080
Waiting for client ...
Connection established with ('127.0.0.1', 36684)
Client: Hi. could you share me the password to log into the master pc?
Message: Sure. The password is shasdkjl3872940
Client: Thanks
Message: No problem
Client: I hope the messages are encrypted. We cannot be sharing passwords like this otherwise
Message: Don't worry. RSA encryption algorithm is used for encryption.
Client: end
Message: █
```

Fig: Server

```
sabinthapa pop-os > ../Chapter_1-A_1 main > venv > python3 client.py
Enter the HOST IP: (default: 127.0.0.1):
Enter the PORT number: 8080
Server : Server waves hi!
Message: Hi. could you share me the password to log into the master pc?
Server : Sure. The password is shasdkjl3872940
Message: Thanks
Server : No problem
Message: I hope the messages are encrypted. We cannot be sharing passwords like this otherwise
Server : Don't worry. RSA encryption algorithm is used for encryption.
Message: end
sabinthapa pop-os > ../Chapter_1-A_1 main > venv > ;6~█
```

Fig: Client

Output of Testing

```
sabinthapa pop-os > ../Chapter_1-A_1 main > venv > python3 test.py
..
-----
Ran 2 tests in 0.005s

OK
```


Conclusion

In this way, the RSA encryption algorithm can be used in Python to encrypt and decrypt messages sent between the client and the server using the keys generated randomly and securely to encrypt and decrypt the messages. The program is tested using unit testing library in Python.

Appendix

The implementation can be found on GitHub at the following [link](https://github.com/sabin-thapa/COMP492_BlockChain):

https://github.com/sabin-thapa/COMP492_BlockChain

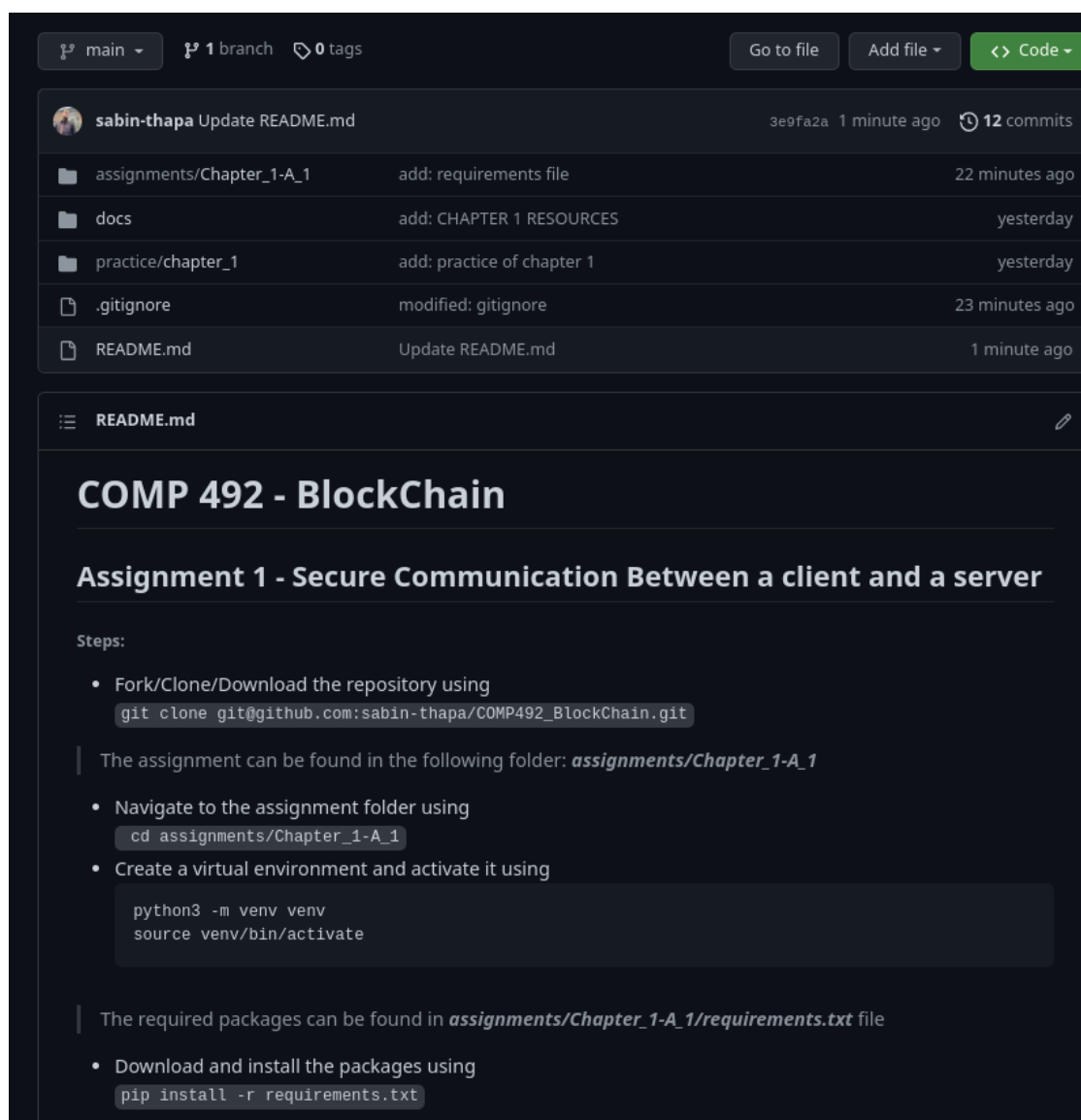


Fig: GitHub Repository