NETWORKS AND SYSTEM SECURITY

Assignment - 3 (Problem - 2)

Name : Suraj Kiran Mate
Entry No : 2021JCS2387

Problem - 2 : Implementing Customised TLS

Writing the TLS code was really an exciting experience. I was not able to write it perfectly but still it clears all the working details behind this code for me.

I have implemented 3 python files. One for Trusted Third Party, second for Server which will send the OTP message to client and third for the Client itself. Before directly going into the details of all these codes we will first discuss the flow of communication that I have implemented in this codes.

The message exchange in between all the parties are as follows:

1. Client will communicate with server. And will ask the server about the cipher suite it wants to utilise.

2. Out of available options from the cipher suite, server choose one of the supported asymmetric algorithm for key exchange, symmetric encryption and the required hash algorithm. The server will reply these details to the client.

3. Based on server's reply both client and server will generate the keys corresponding to the algorithm that they have decided on.

4. After generation of keys both will communicate to Trusted Third Party (TTP) to generate the corresponding certificates on their public keys. Before that TTP will also create its own certificate.

5. Both client and server will verify each others certificate and also verify the certification authority (ie. TTP here) for authentication.

6. Now, both have verified each other and ready to communicate with each other. Hence client will send the generated secret which is the symmetric key to server.

7. The symmetric key will be encrypted by the key exchange algorithm chosen by server.

8. Server will decrypt the key. So both client and server will end up having common secret.

9. Now, server will encrypt the OTP message with this symmetric key and send it to the client.

10. Client will decrypt the message using the symmetric key and thus message get successfully exchanged between server and client.

These are roughly the 10 steps which are getting followed in my code. These steps will be further elaborated while explaining the code.

The detailed code is explained as below:

In this explanation I will like to go function by function for this code:

1) cipherSuite Function:

The code on client side is as follows:

```
def cipherSuite(self):                                  # exchanging ciphersuite between client and server
    payload = "WHICH KEY EXCHANGE TO USE ECDSA, RSA \n WHICH SYMMETRIC ENCRYPTON TO USE AES, CHACHA20 \n WHICH HASH TO USE SHA256, SHA384"
    clientSoc2.send(payload.encode("utf-8"))
    print("Ciphersuite sent successfully.")

    while True:
        data = clientSoc2.recv(1024)                    # receiving the acknowledgment from the server
        data = data.decode("utf-8")
        print("The received ACK is: ", data)
        data = data.split(" ")
        key_exchange = data[0]
        encryption = data[1]
        hash = data[2]
        private_key_client, public_key_client, symmetric_key = self.generateKey(client_key_path, key_exchange, encryption, hash)       # calling the generate k
        break
    print("Client and Server finalised the ciphersuite.")

    return key_exchange, encryption, hash, private_key_client, public_key_client, symmetric_key
```

The code on server side is as follows:

```
def ciphersuite(self):                                  # This method is used to exchange ciphersuite between clint and server
    while True:
        print("Server is waiting for ciphersuite")
        data = clientSoc2.recv(1024)
        if not data or data.decode("utf-8")=="END":
            break
        data = data.decode("utf-8")
        print("message received is: ", data)
        data = data.split("\n")
        a = data[0]
        b = data[1]
        c = data[2]

        key_exchange = input(a + " :-----> ")           # Taking user input for the ciphersuite to use
        encryption = input(b + " :-----> ")
        hash = input(c + " :-----> ")

        ack = "{} {} {}".format(key_exchange, encryption, hash)             # creating the ack message to send oit back to server.
        while True:
            clientSoc2.send(ack.encode("utf-8"))
            print("Ciphersuite ACK sent successfully.")
            break

        private_key_server, public_key_server = self.generateKey(server_key_path, key_exchange, encryption, hash)       # calling the generate key finction to
        break

    print("client and server finalised the ciphersuite.")
    return key_exchange, encryption, hash, private_key_server, public_key_server
```

Here client is first asking about the key exchange and sending that message to server. The server will receive the message and on server side we are explicitly taking user input for the required algorithms to use.
The user will input the key exchange to use for key exchange, for symmetric encryption and for hashing. From these choices a message is created by the server which is stored in **ack** variable. This message is then sent to the client. Here **clientSoc2** is the socket used for communication between client and server.
We are printing the print statements just to get the acknowledgement about the program flow on the terminal.

We have stored the algorithms of choices in three variables **key_exchange**, **encryption** and **hash.** Then on both client and server side inside this **ciphersuite** function we are calling the **generateKey** function. We are passing all the algorithms of choices in the generateKey function as arguments. Hence form this function we will get the required keys. We will look for more details of this function in the next section. From this cipher suite function we are returning various parameters which we will be using for other functions.

In client side of program also we are passing these parameters to generate key function and getting all the required values from them.

Output of the code on client side is:

```
surajmate@Surajs-MacBook-Air untitled folder 2 % python3 client.py
Enter username for which certificate to be genarated: SURAJ
connecting to the new socket created by server
Ciphersuite sent successfully.
The received ACK is:  RSA AES SHA256
```

Output of the code on server side is:

```
surajmate@Surajs-MacBook-Air untitled folder 2 % python3 server.py
Enter username for which certificate to be genarated: BANK
Creating new socket to connect to client
Serve is waiting for connection
Server is waiting for ciphersuite
message received is:  WHICH KEY EXCHANGE TO USE ECDSA, RSA
 WHICH SYMMETRIC ENCRYPTON TO USE AES, CHACHA20
 WHICH HASH TO USE SHA256, SHA384
WHICH KEY EXCHANGE TO USE ECDSA, RSA  :-----> RSA
 WHICH SYMMETRIC ENCRYPTON TO USE AES, CHACHA20  :-----> AES
 WHICH HASH TO USE SHA256, SHA384 :-----> SHA256
Ciphersuite ACK sent successfully.
```

It is clearly visible from the screenshots that the message exchange becomes successful in between client and server.

2) generateKey Function:

The client ans server side code is shown in the following screenshots. Both are using same key generation schemes. We are using **cryptography** library for generation of the keys. The documentation for this library is mentioned in the documentation.

We can see in the screenshots that we are generating more keys for client the the server. Since the server will choose the ciphers it wants to use. Hence it becomes essential to generate the same type of keys.

In symmetric key cryptography this differentiation is not there. All algorithms can use the same randomly generated secret. But in asymmetric the keys for any algorithms should be generated by the methods dedicate for this algorithm.

Since ECC uses the keys generated from the curve and RSA uses the keys having some mathematical relation. Hence we are using cryptography library for tis purpose.

Client side code:

```
def generateKey(self, key_path, key_exchange, encryption, hash):
    if(key_exchange == "ECDSA"):
        private_key_client = ec.generate_private_key(ec.SECP384R1())
        public_key_client = private_key_client.public_key()

    elif(key_exchange == "RSA"):
        private_key_client = rsa.generate_private_key(public_exponent=65537, key_size=2048)
        public_key_client = private_key_client.public_key()


    if(encryption == "AES"):
        symmetric_key = os.urandom(16)

    elif(encryption == "CHACHA20"):
        symmetric_key = os.urandom(32)

    if not os.path.exists('Client'):
        os.makedirs("Client")

    with open("Client/private_key.pem", "wb") as f:
            f.write(private_key_client.private_bytes(encoding=serialization.Encoding.PEM, format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),))

    with open("Client/public_key.pem", "wb") as f:
        f.write(public_key_client.public_bytes(encoding=serialization.Encoding.PEM, format=serialization.PublicFormat.SubjectPublicKeyInfo))

    return private_key_client, public_key_client, symmetric_key
```

Server Side Code:

```
def generateKey(self, key_path, key_exchange, encryption, hash):

    if(key_exchange == "ECDSA"):                                    # if the algorithm mentioned is ECDSA then generating the keys for ECDSA
        private_key_server = ec.generate_private_key(ec.SECP384R1())
        public_key_server = private_key_server.public_key()            # All are then methods from cryptography library.
        print("ECDSA key pair generated successfully")


    elif(key_exchange == "RSA"):                                    # if the algorithm is RSA then generating keys for RSA
        private_key_server = rsa.generate_private_key(public_exponent=65537, key_size=2048)
        public_key_server = private_key_server.public_key()
        print("RSA key pair generated successfully")

    if not os.path.exists('Server'):
        os.makedirs("Server")

    # storing both public and private keys of server

    with open("Server/private_key.pem", "wb") as f:
        f.write(private_key_server.private_bytes(encoding=serialization.Encoding.PEM, format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),))

    with open("Server/public_key.pem", "wb") as f:
        f.write(public_key_server.public_bytes(encoding=serialization.Encoding.PEM, format=serialization.PublicFormat.SubjectPublicKeyInfo))


    print("Keys generated successfully.")
    return private_key_server, public_key_server
```

In client side code we are also generating the random secret key ie. symmetric key. The symmetric key used for **RSA** can start from **16 bytes** but for **CHACHA20** it can support only one key size ie **32 bytes** (in cryptography library source code only one key size is mentioned for CHACHA20).
The client will generate this symmetric key for future sharing.
From this function we are returning all the keys we have generated and also storing the keys in the directory.

Now we will create separate sockets for both client and server to communicate with the Trusted Third Party to get the certificates on the public keys generated for both of them. Thus both client and server will go to TTP to get the certificates on their public keys.

3) getCertificate Function:

Client Side Code:

```
def getCertificate(self, client_key):
    csr=x509.CertificateSigningRequestBuilder().subject_name(x509.Name([               # creating certificate signing request
        x509.NameAttribute(NameOID.COUNTRY_NAME, input("Enter Country Code: ")),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, input("Enter state or provience: ")),
        x509.NameAttribute(NameOID.LOCALITY_NAME, input("Enter Locality Name: ")),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, input("Enter organisation name: ")),
        x509.NameAttribute(NameOID.COMMON_NAME, input("Enter domain name: ")),])).add_extension(x509.SubjectAlternativeName([x509.DNSName(u"mysite.com"),
        x509.DNSName(u"www.{}.com".format(input(client_cn))),
        x509.DNSName(u"subdomain.mysite.com"),]),critical=False,).sign(client_key, hashes.SHA256())        # Sign the CSR with our private key.

    with open("Client/csr.pem", "wb") as f:
        f.write(csr.public_bytes(serialization.Encoding.PEM))                          # soring the csr file

    signing_req_path = "Client/csr.pem"
    payload = "CERTIFICATE SIGNING REQUEST AT {} AND PUBLIC KEY AT Client/public_key.pem FOR USERNAME {}".format(signing_req_path, client_cn)
    CA_cert_path = ""

    while True:
        try:
            clientSoc1.send(payload.encode("utf-8"))
            print("Certificate signing request sent successfully")

            data = clientSoc1.recv(1024)
            data = data.decode("utf-8")
            data = data.split(" ")
            if(data[0] == "SUCCESSFUL"):
                print("Certificate created successfully.")
            CA_cert_path = data[1]
            print(CA_cert_path)

        except:
            print("TTP unavailable. Try again Later")
        break

    clientSoc1.close()
    return CA_cert_path
```

Server Side Code:

```
def getCertificate(self, server_key):                        # Using This method server will communicate with TTP to get the certificate
    csr=x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, input("Enter Country Code: ")),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, input("Enter state or provience: ")),              # providing all details for cetificate signing request
        x509.NameAttribute(NameOID.LOCALITY_NAME, input("Enter Locality Name: ")),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, input("Enter organisation name: ")),
        x509.NameAttribute(NameOID.COMMON_NAME, input("Enter domain name: ")),])).add_extension(x509.SubjectAlternativeName([x509.DNSName(u"mysite.com"),
        x509.DNSName(u"www.{}.com".format(input(server_cn))),
        x509.DNSName(u"subdomain.mysite.com"),]),critical=False,).sign(server_key, hashes.SHA256())         # Sign the CSR with servers private key.

    with open("Server/csr.pem", "wb") as f:
        f.write(csr.public_bytes(serialization.Encoding.PEM))          # storig the CSR in Server directory.

    # creating certificate signing request and sending it to TTP

    signing_req_path = "Server/csr.pem"
    payload = "CERTIFICATE SIGNING REQUEST AT {} AND PUBLIC KEY AT Server/public_key.pem FOR USERNAME {} \n".format(signing_req_path, server_cn)
    CA_cert_path = ""

    while True:
        try:
            # sending the signing request to TTP

            serverSoc1.send(payload.encode("utf-8"))
            print("Certificate signing request sent successfully")

            data = serverSoc1.recv(1024)
            data = data.decode("utf-8")
            data = data.split(" ")
            if(data[0] == "SUCCESSFUL"):
                print("Certificate created successfully.")

            print(data)
            CA_cert_path = data[1]
            print(CA_cert_path)

        except:
            print("TTP unavailable. Try again Later")

        break

    serverSoc1.close()
    return CA_cert_path
```

Both the codes are exactly same. They both are first generating certificate signing request and then. In this request they have to submit some of this details. We can also hardcode these values.

After creating CSR it is stored in the directory.
They both are sending the car path to the TTP. The TTP will then access the car from their corresponding directories check for the signatures of the corresponding requesting entity and based on this verification create the certificate for the these entities.

TTP Side Code:

```
def create_CA_Certificate(self):
    ''' Create CA and Key'''

    CA_key = rsa.generate_private_key(public_exponent=65537, key_size=2048,)
    if not os.path.exists('CA'):
        os.makedirs('CA')

    with open(CA_key_path, "wb") as f:
        f.write(CA_key.private_bytes(encoding=serialization.Encoding.PEM, format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),))

    subject = issuer = x509.Name([x509.NameAttribute(NameOID.COUNTRY_NAME, input("Enter country code: ")),
                x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, input("Enter state or province name: ")),
                x509.NameAttribute(NameOID.LOCALITY_NAME, input("Enter locality name: ")),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Enter Organisation Name: "),
                x509.NameAttribute(NameOID.COMMON_NAME, input("Enter website domain name: ")),])

    cert = x509.CertificateBuilder().subject_name(subject).issuer_name(issuer).public_key(CA_key.public_key()).serial_number(x509.random_serial_number()).not_val

    with open(CA_cert_path, "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    return CA_key
```

```
def create_Client_Certificate(self, CA_key, client_cn, sock, prefix):

    subject = issuer = x509.Name([x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
                x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"California"),
                x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
                x509.NameAttribute(NameOID.COMMON_NAME, u"mysite.com"),])

    cert = x509.CertificateBuilder().subject_name(subject).issuer_name(issuer).public_key(CA_key.public_key()).serial_number(x509.random_serial_number()).not_val

    with open("{}/{}.pem".format(prefix, client_cn), "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))

    while True:
        try:
            payload = "SUCCESSFUL {}".format(CA_cert_path)
            sock.send(payload.encode("utf-8"))

        except:
            print("Server Unavailable...")

        break
    return
```

Before creating the certificates for these entities, TTP will first create its own certificate. After TTP certificate the requested certificates from client and server are created and. Here while creating the certificated for client and server the values are hardcoded which are passed to certificate builder function.

From both the requests certificates are builded. For client and server. The generated certificates are showed in the corresponding directories of client and server.

```
surajmate@Surajs-MacBook-Air untitled folder 2 % python3 ttp.py
Enter country code: IN
Enter state or province name: D
Enter locality name: D
Enter website domain name: D
TTP waiting for client to connect
Certification request successfully received from client.
Client certificate created successfully.
TTP waiting for server to connect
Certification request successfully received for server.
Server certificate created successfully.
surajmate@Surajs-MacBook-Air untitled folder 2 %
```

```
surajmate@Surajs-MacBook-Air untitled folder 2 % python3 server.py
Enter username for which certificate to be genarated: BANK
Creating new socket to connect to client
Serve is waiting for connection
Server is waiting for ciphersuite
message received is:  WHICH KEY EXCHANGE TO USE ECDSA, RSA
 WHICH SYMMETRIC ENCRYPTON TO USE AES, CHACHA20
 WHICH HASH TO USE SHA256, SHA384
WHICH KEY EXCHANGE TO USE ECDSA, RSA  :-----> RSA
 WHICH SYMMETRIC ENCRYPTON TO USE AES, CHACHA20  :-----> AES
 WHICH HASH TO USE SHA256, SHA384 :-----> SHA256
Ciphersuite ACK sent successfully.
RSA key pair generated successfully
Keys generated successfully.
client and server finalised the ciphersuite.
Enter Country Code: IN
Enter state or provience: S
Enter Locality Name: S
Enter organisation name: S
Enter domain name: S
BANK
Certificate signing request sent successfully
Certificate created successfully.
```

```
RSA Key pair successfully generated.
AES symmetric Key successfully generated.
Client and Server finalised the ciphersuite.
Enter Country Code: IN
Enter state or provience: Y
Enter Locality Name: Y
Enter organisation name: Y
Enter domain name: Y
SURAJ
Certificate signing request sent successfully
Certificate created successfully.
CA/cert.pem
Certificate verification message sent successfully
The req is:  VERIFY SERVER CERTIFICATE AND KEY AT PATH Server/BANK.pem AND Server/public_key.pem
Certification verification request successfully received.
CA Certificate valid for 9 days
Server Certificate valid for 9 days
Server certificate successfully Verified.
Ready sent successfully.
Encrypted key sent
The ciphertext is:  b'\x07\xf5\x07\xd5^.\xad\xe1\tw\x1f\xee\x90\xf6\xa2b8n\xcc\xfd;:%I\xaa\xcc\x96Q\xc8\xe1PM\xeb\x90\x1e\x02$\x884\x93\xe3\x8b=\xc6\xd0w\xe3gh\xe8~\x92I\x15/
```

The above screenshot shows the certificate signing procedure. Most of the intermediate steps are not printed in the output.

4) verifyCertificate:

After creating the certificate both client and server will again communicate with each other and they will verify each others certificate before having trust on each other. Here both will have the path to certificate directory of each other and they will access the directory to retrieve the certificate

Client Side Code:

```
vrfReq = "VERIFY CLIENT CERTIFICATE AND KEY AT PATH {} AND {}".format(client_cert_path, client_key_path)
while True:
    try:
        clientSoc2.send(vrfReq.encode("utf-8"))
        print("Certificate verification message sent successfully")

        while True:
            try:
                data = clientSoc2.recv(1024)
                if not data or data.decode("utf-8")=="END":
                    break
                req = data.decode("utf-8")
                print("The req is: ", req)
                req = req.split(" ")
                if(req[0]=="VERIFY" and req[1] == "SERVER" and req[2]=="CERTIFICATE" and req[3]=="AND" and req[4]=="KEY"):
                    print("Certification verification request successfully received.")
                    server_cert_path = req[7]
                    server_key_path = req[9]

                    cs.verifyCertificates()

                else:
                    nack = "VALUE ERROR, TRY AGAIN"
                    clientSoc2.send(bytes(nack, "utf-8"))

            except:
                print("Server unreachable.")
            break

    except:
        print("Server unreachable...")
    break
```

Before directly going for verifying the certificates they will first exchange the verification requests among each other. This is the client side code and the corresp[onding server side code will also be exactly same.

Server Side Code:
In server side code we are first receiving the certificate verification request from the client and then acting on that request.
Since certificates of both client and server are stored in the directory hence they will share the paths for certificates to each other. Based on these paths they will access each others certificate and check for verification.
The standard format for certificate verification request is also checked before directly working on the request.
“**VERIFY CLIENT CERTIFICATE AND KEY AT PATH ____ AND ____**” this format is first checked.

```
while True:
    try:
        print("server waiting for connection")

        data = clientSoc2.recv(1024)
        if not data or data.decode("utf-8")=="END":
            break
        req = data.decode("utf-8")
        req = req.split(" ")
        if(req[0]=="VERIFY" and req[1] == "CLIENT" and req[2]=="CERTIFICATE" and req[3]=="AND" and req[4]=="KEY"):
            print("Certification verification request successfully received.")
            client_cert_path = req[7]
            client_key_path = req[9]

            sc.verifyCertificates()

        else:
            nack = "VALUE ERROR, TRY AGAIN"
            clientSoc2.send(bytes(nack, "utf-8"))

        vrfReq = "VERIFY SERVER CERTIFICATE AND KEY AT PATH {} AND {}".format(server_cert_path, server_key_path)

        while True:
            try:
                clientSoc2.send(vrfReq.encode("utf-8"))
                print("Certificate verification message sent succesfully")

            except:
                print("Client unreachable 1")

            break

    except:
        print("Client unreachable 2")

    break
```

After checking the correctness of the request both client and server will start working on these requests.

Client Side Function :

```
def verifyCertificates(self):
    print(server_cert_path)
    print(server_key_path)
    print(CA_cert_path)

    with open(server_cert_path, "r") as f:
        server_cert = crypto.load_certificate(crypto.FILETYPE_PEM, f.read())

    with open(server_key_path, "r") as f:
        server_key = crypto.load_publickey(crypto.FILETYPE_PEM, f.read())

    with open(CA_cert_path, "r") as f:
        CA_cert = crypto.load_certificate(crypto.FILETYPE_PEM, f.read())

    ca_expiry = datetime.strptime(str(CA_cert.get_notAfter(), 'utf-8'),"%Y%m%d%H%M%SZ")
    now = datetime.now()
    validity = (ca_expiry - now).days
    print ("CA Certificate valid for {} days".format(validity))

    CA_pubkey = CA_cert.get_pubkey()
    if(CA_pubkey == CA_key):
        print("CA Certificate is successfully validated.")

    server_expiry = datetime.strptime(str(server_cert.get_notAfter(), 'utf-8'),"%Y%m%d%H%M%SZ")
    now = datetime.now()
    validity = (server_expiry - now).days
    print ("Server Certificate valid for {} days".format(validity))

    server_pubkey = server_cert.get_pubkey()
    print("Server certificate is successfully Validated.")
```

Both client side and server side functions are exactly same hence no need to put them again. We can see that they both will first check the validity of TTP certificate since TTP has issued the certificates to both of them.
Honestly speaking I could not find the certificate verification methods using cryptography library. Also openSSL library is also abstract in this explanation. Hence I have only verified these certificates based on their validity.

Here, **ca_expiry** variable stores the expiry days for CA certificate and similarly **server_expiry** and **client_expiry** will store the days for expiry of server and client certificate respectively.
When the certificates are successfully validated then both client and server are ready to trust each other. They can now exchange messages between each other using asymmetric key cryptography.

5) communicateKey:

Client Side Code:

```
def communicateKey(self, symmetric_key, key_exchange):      # Symmetric key is communicated between client and server.
    public_key_server = None

    with open(server_key_path, "rb") as f:
        public_key_server = serialization.load_pem_public_key(f.read())


    if(key_exchange == "RSA"):                  # if key exchange algo is RSA then using it for key encryption
        ciphertext = public_key_server.encrypt(symmetric_key, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
        while True:
            try:
                clientSoc2.send(ciphertext)
                print("Encrypted key sent")
                data = clientSoc2.recv(1024)

                print("The ciphertext is: ", ciphertext)

                print("The data is : ", data)

                print("Key reached succcessfully.")

            except Exception as e:
                print("server unavailable !")
                print("The exception is : ", e)

            break

    elif(key_exchange == "ECDSA"):              # if key exchange algo is ECDSA then using it for key encryption
        private_key = ec.generate_private_key(ec.SECP384R1())
        data = bytes(str(symmetric_key), "utf-8")
        signature = private_key.sign(data, ec.ECDSA(hashes.SHA256()))
        while True:
            try:
                clientSoc2.send("{} {}".format(symmetric_key, signature))
                print("Encrypted key sent successfully.")
                data = clientSoc2.recv(1024)
                data = data.decode("utf-8")
                if(data == "SUCCESSFUL"):
                    print("Key reached succcessfully.")
            except:
                print("server unavailable !")
            break
```

Since client was the one to generate the symmetric key hence client will also send the encrypted version of this key to server. Both server and client has decided the key exchange algorithm to be used in between them. Hence both will know using which algorithm to encrypt and decrepit the symmetric key.
Thus client will extract the public key of server from the server directory and send the encrypted version of the key to the server (encrypted using the public key of the user). Now, the key exchanged from client to server will be in encrypted format hence server will decrypt that key using his private key for the corresponding algorithm.

Server Side Code:

```python
def communicateKey(self, key_exchange):                # This method is used to communicate the keys between client and server
    symmetric_key = None
    while True:
        ciphertext = clientSoc2.recv(1024)
        print("ciphertext is: ", ciphertext)

        if(key_exchange == "RSA"):                     # if key exchange is RSA then using RSA encryption on the key.
            print("Entered for decryption.")
            symmetric_key = private_key_server.decrypt(ciphertext, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))

            print("Key decrypted successfully.")
            suc = "SUCCESSFUL"
            while True:
                try:
                    clientSoc2.send(suc.encode("utf-8"))        # sending the ack that the ciphertext is received.
                    print("ack sent successfully.")
                except:
                    print("Cleint unavailable ")
                break

        elif(key_exchange == 'ECDSA'):                 # using ECDSA for key exchange
            data = ciphertext.split(" ")
            symmetric_key = data[0]
            signature = data[1]
            try:
                public_key_server.verify(signature, bytes(str(symmetric_key), "utf-8"), ec.ECDSA(hashes.SHA256()))
                print("Signature verified successfully")
                suc = "SUCCESSFUL"
                clientSoc2.send(suc.encode("utf-8"))
            except:
                print("Error Occured in signature verification")

            print("Client unreachable...")

        break
    return symmetric_key
```

Thus server will decrypt that encrypted key using its private key. Thus both client and server will end up having same symmetric key. Now they can use this key to further exchange the bigger messages.

Client Side Output:

```
Certification verification request successfully received
.
Server/IN.pem
Server/public_key.pem
CA/cert.pem
CA Certificate valid for 9 days
Server Certificate valid for 9 days
Server certificate is successfully Validated.
```

Server Side Output:

```
Certification verification request successfully received.
Client/DURAJ.crt
Client/public.key
CA/cert.pem
CA Certificate valid for 9 days
Client Certificate valid for 9 days
Client certificate is successfully Validated.
Certificate verification message sent succesfully
```

Thus we can see from above outputs the certification validation is getting successful.

6) sendOTP and **recevOTP:**

Server Side Code:

```
def sendOTP(self, symmetric_key, encryption):                    # the method is used to exchange OTP between server and client
    OTP = b'''The OTP for transferring Rs 1,00,000 to your friend's account is 256345.'''
    nonce = os.urandom(16)                  # generating the nonce value to encrypt the message
    while True:
        try:
            clientSoc2.send(nonce)
            print("Nonce sent successfully")

        except:
            print("Client unreachable.")
        break

    if(encryption == "AES"):                # if encryption os AES then using this to send the encrypted message
        print("Encrypting using AES")
        plaintext = pad(OTP, AES.block_size)
        print("The padded plaintext is : ", plaintext)
        cipher = Cipher(algorithms.AES(symmetric_key), modes.CBC(nonce))
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(plaintext) + encryptor.finalize()
        print("CIPHERTEXT SUCCESSFULLY GENERATED: ", ciphertext)

    elif(encryption == "CHACHA20"):         # if encryption os CHACHA20 then using this to send the encrypted message
        print("Encrypting using CHACHA20")
        algorithm = algorithms.ChaCha20(symmetric_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(OTP)
        print("CIPHERTEXT SUCCESSFULLY GENERATED: ", ciphertext)


    while True:
        try:
            clientSoc2.send(ciphertext)                # Sending the encrypted message to client
            print("ciphertext sent successfully.")

            while True:
                try:
                    data = clientSoc2.recv(1024)               # Receiving the acknowledgment from client
                    data = data.decode("utf-8")
                    print(data)
                    if(data == "SUCCESSFUL"):                  # checking whether the transfer was successful or not
                        print("OTP sent and decrypted successfully.")
                except Exception as e:
                    print("Exception is: ", e)
                break

        except:
            print("Client unavailbale in OTP")

        break
```

In this above code we can see that we are first creating the message that we want to share between server and client. After creating the message then we are creating the one time secret between client and server ie. **nonce.** Then we are first sharing this nonce from server to client. Since nonce is needed in both AES and CHACHA20 encryption schemes hence it is important to share it before exchanging the encrypted message between them.
Once client receives the nonce then both client and server becomes eligible to share the encrypted messages. Here we are not sending the nonce in encrypted format. Thus after this we will check the encryption algorithm to be used in between client and server.
We will be using conditionals that's why we passed the parameter **encryption** which stores the algorithm that they have finalised on.

Client Side Code:

```
def recevOTP(self, symmetric_key, encryption):      # This funnction is called to receive the OTP message from server
    nonce = None
    ciphertext = None

    while True:
        try:
            nonce = clientSoc2.recv(1024)           # getting the nonce value from server to decrypt the messages
            print(nonce)
            print("Nonce received successfully: ", nonce)
        except:
            print("Server unavalable in nonce 1.")
        break

    while True:
        print("Waiting for ciphertext...")
        try:
            ciphertext = clientSoc2.recv(1024)      # getting the ciphertext from server
            print("Ciphertext received is : ",ciphertext)
            ack = "SUCCESSFUL"
            clientSoc2.send(ack.encode("utf-8"))
        except:
            print("Server unavailable in nonce 2")

        break

    print("Decrypting the ciphertext...")

    if(encryption == "AES"):                        # if encryption algo is AES then decrypting using AES
        print("decrypting using AES.")

        print("Length of nonce is: ", len(nonce))

        cipher = Cipher(algorithms.AES(symmetric_key), modes.CBC(nonce))
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    elif(encryption == "CHACHA20"):                 # encryption algo is CHACHA20 then decryption using CHACHA20
        print("decrypting using CHACHA20.")
        algorithm = algorithms.ChaCha20(symmetric_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(ciphertext)

    print("ciphertext decrypted successfully.")
    print("Plaintext is: ", plaintext)
    plaintext = plaintext.decode("utf-8")
    return plaintext                                # returning the plaintext
```

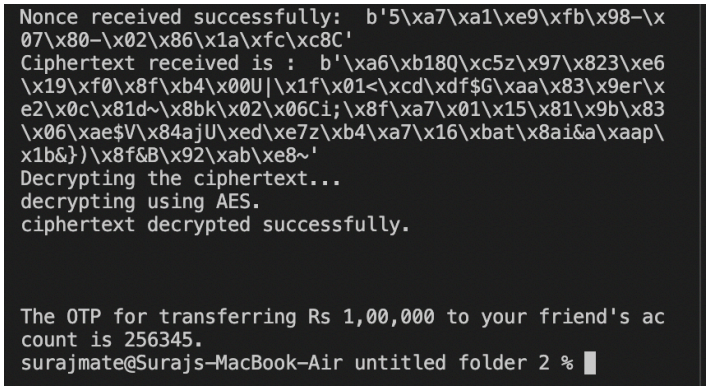
After encrypting the message server will send that message to the client. Then client will decrypt the message using the same algorithm by which it was actually encrypted. Hence after decryption the client will get he required OTP from the server.

The output for this function on server side:

```
Nonce sent successfully
Encrypting using AES
The padded plaintext is :  b"The OTP for transferring Rs
1,00,000 to your friend's account is 256345.\x08\x08\x08\
x08\x08\x08\x08\x08"
CIPHERTEXT SUCCESSFULLY GENERATED:  b'\xa6\xb18Q\xc5z\x97
\x823\xe6\x19\xf0\x8f\xb4\x00U|\x1f\x01<\xcd\xdf$G\xaa\x8
3\x9er\xe2\x0c\x81d~\x8bk\x02\x06Ci;\x8f\xa7\x01\x15\x81\
x9b\x83\x06\xae$V\x84ajU\xed\xe7z\xb4\xa7\x16\xbat\x8ai&a
\xaap\x1b&})\x8f&B\x92\xab\xe8~'
ciphertext sent successfully.
SUCCESSFUL
OTP sent and decrypted successfully.
surajmate@Surajs-MacBook-Air untitled folder 2 %
```

From server side output we can see that server sent the nonce successfully to the client then AES was the chosen encryption algorithm hence server is going to encrypt using AES. Then the padded message is printed. Then the ciphertext is printed. And the ciphertext is sent to the client.

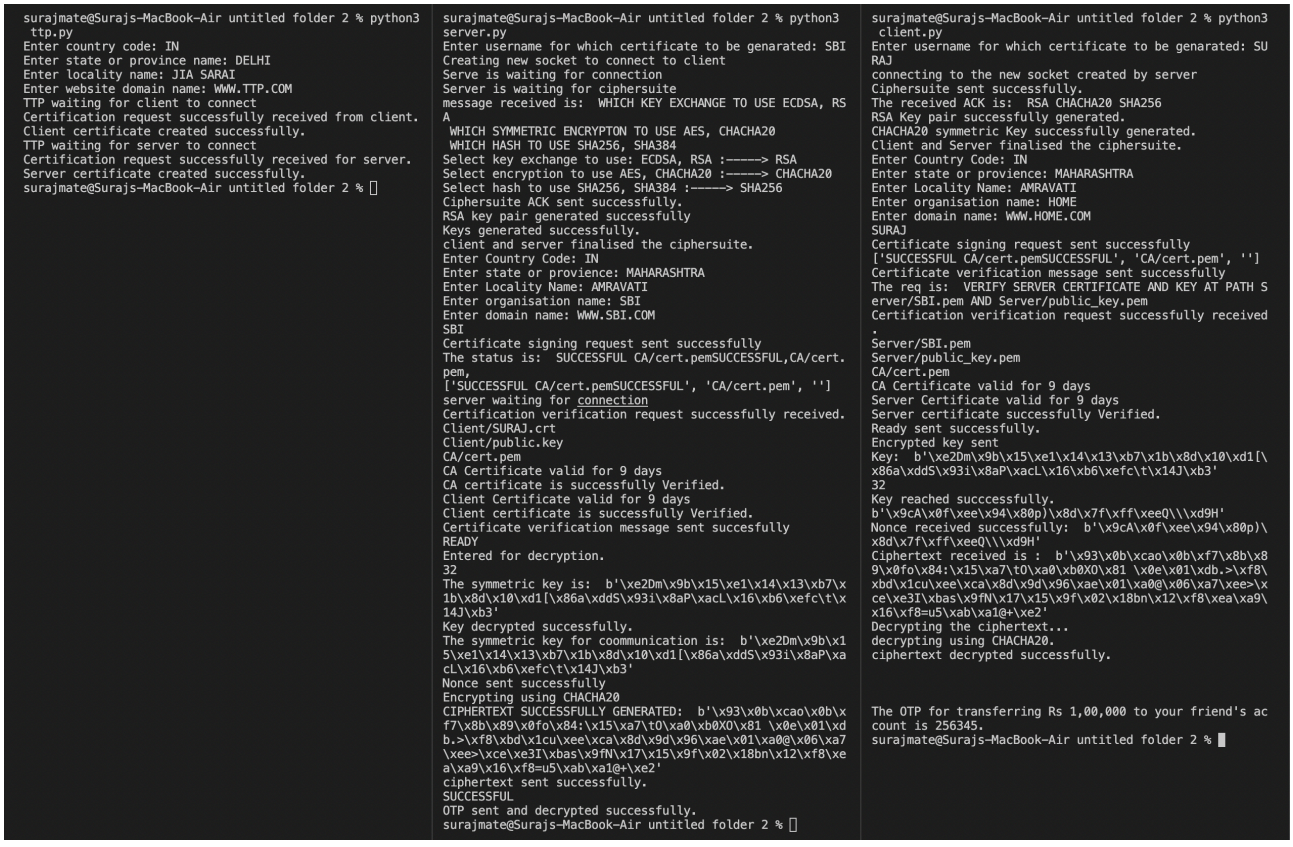
The output for this function on client side:



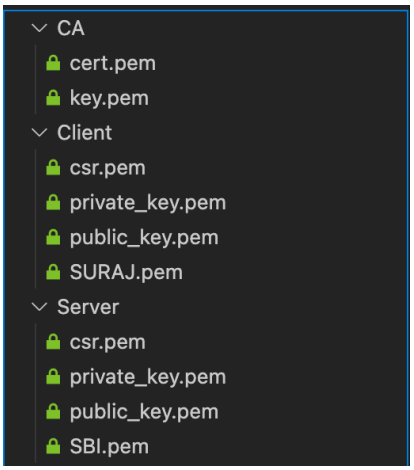
We can see that on client output window we have first received the nonce from server. The we have received the ciphertext. Then using the same symmetric key the ciphertext is decrypted.
At the last line the plaintext sent by the server is printed. The plaintext is representing the message corresponding to the OTP to be communicated between client and server.

In this way all of our 10 methods were carried out in python code. There are many improvements needed in this code like proper certificate verification. Proper CSR verification and signing. I also could not implement the negotiation part.

So with everything I tried to print the overall output as :



In the above final output the leftmost section is for TTP the middle section is for server and the rightmost section is for client.



The above screenshot represents the directory of CA, Client and Server. In these directories the certificates of them and key are stored.

7) How to Run this code:

1. First open three terminals and run **my_ttp.py** file using **python3 my_ttp.py** command.
2. Run **my_server.py**. Give the name of the server you want. Then similarly run **my_client.py** and give the name of the client you want.
3. Then you have to input the cipher suite ie. RSA, AES, CHACHA20 and SHA256, SHA384.
4. Then input the country code and all certification details as input in the server terminal.
5. Similarly input all certification details as input in the client terminal.
6. Then automatically verification and message exchange will be done.

8) Security and Efficiency of Protocol:

1. Since we could not verify the certificates perfectly, hence it can act like security hole for our protocol.
2. We could encrypt the nonce before sharing from server to client, but we are not doing it. It is just the additional layer of security that we can get out of it.

8) References:

a) The documentation of libraries are the best references to build the code from scratch. Also special thanks to stack overflow and stack exchange for very good question and answering platform which really resolved many of my doubts and errors.

b) Below is the link of python openSSL documentation it helped me lot in knowing some implementation details of the certificates.

https://www.pyopenssl.org/en/stable/

c) The below link is the best reference for my work. The cryptography library has implemented almost all the algorithms elegantly from key generation to certificate creation.

https://cryptography.io/en/latest/x509/tutorial/#creating-a-certificate-signing-request-csr

d) I haven't used any particular functions from the library mentioned below (Pycryptodome) but padding and unpadding of data during symmetric encryption and decryption gives error in cryptography. Hence, this library served as very good source to overcome this issue.

https://www.pycryptodome.org/en/latest/

e) Thanks to **Prof. Vireshwar Kumar Sir** for clearing all our basics required to implement this code.

9) Problems I faced:

Many times the implementation of some functionality from one library is good but the related functionality from other library gives an error. As I mentioned earlier padding and unpadding from cryptography was giving erroneous output then specifically using these functionalities from some other library becomes hectic mostly due to compatibility issues. The same kind of issue I faced during loading and storing the keys and certificates in the directories. Because the formats of storage of one library is good but the keys generated by some other library don't support that store and load format. Hence it became complete permutation and combination problem.

Thank You !