CSE-451: COMPUTER AND NETWORK SECURITY

Project: Phase4
Group 13

Submission Date:

MAY 14, 2024

Submitted to:

Dr. Ayman Bahaa

Eng. Hesham Fathy

Submitted by:

Mahmoud Mohamed Seddik ElNashar	19P3374
Omar Ashraf Mabrok	19P8102
Zakaria Sobhy Abd-ElSalam Soliman Madkour	19P2676

Table of Contents

Link to the code repo

https://github.com/Zakaria-Madkour/Security.git

Questions to be answered by the end of this phase

1. How are the different modules integrated into the Secure Communication Suite?

Level 0 - Primitive modules (real workers):

1. SHA
   This module wraps and utilizes the SHA-256 hashing function found in cryptography.hazmat.primitives. This hashing function is used in the authentication, AsymmetricCipher, and certify modules. In the authentication module it is used to get the hash of a password to compare it with that saved in the database. It is used in the AsymmetricCipher module as well in verifying the integrity of digital signatures. Integrity verification is one of the most important usages of SHA and it is used in certify module for this purpose as well.
2. AsymmetricCipher
   This module wraps and utilizes the RSA algorithm found in cryptography.hazmat.primitives.asymmetric. This module is responsible for generating public/private key pairs, asymmetric encryption/decryption, generating digital signatures, and verifying them. This module is used by the client, server, and keymanagement modules. The primary usage here is initiating secrecy by allowing the server to share a public key with the client with which the client will encrypt the session symmetric random key.
3. SymmetricCipher
   This module wraps and utilizes the AES algorithm found in cryptography.hazmat.primitives.ciphers and DES algorithm found in Crypto.Cipher. This module allows the use of symmetric ciphers for encryption/decryption, generating a random key and iv for the algorithm.

Level 1 – Intermediate modules:

1. Authenticate
   This module authenticates a user login by the username and password it utilizes the SHA-256 hashing function to hash the password before saving it in the database.

2. KeyManagement
   This module utilizes AsymmetricCipher module by generating public and private key pairs and associating them with an email. They are saved in pem format for later usage.

Level 2 – High-level modules:

1. Client
2. Server
   These two modules utilize all the previously developed modules to mimic the behavior of data sent over the internet and insuring its security.

2. What types of tests were conducted on the suite?

1. Unit testing
   Each of the modules developed had its unit test to ensure it is working correctly. Here are screenshots of the unit tests and their outputs.
   AsymmetricCipher

```python
plainText = "Hello, world!".encode()

rsa_object = RSA()
public_key, private_key = rsa_object.generate_key_pair()

# Encryption
ciphertext = rsa_object.encrypt(plainText, public_key)
deciphered_text = rsa_object.decrypt(ciphertext, private_key)
print("\nText: ",plainText,"\n\nCiphertext: ",ciphertext,"\n\nDeciphered Text: ",deciphered_text, end="\n\n")

# Digital Signature
digital_signature = rsa_object.generate_digital_signature(plainText, private_key)
print("Digitally signed message: ", digital_signature)
validate = rsa_object.verify_digital_signature(plainText, digital_signature, public_key)
print("\nDigital Signature Validation state: ",validate,end="\n\n")

#Failed verification
public_key2, private_key2 = rsa_object.generate_key_pair()
validate2 = rsa_object.verify_digital_signature(plainText, digital_signature, public_key2)
print("\nDigital Signature Validation state: ",validate2,end="\n\n")
```

```
Text:  b'Hello, world!'

Ciphertext:  b'\x17<6/aj\x90\xdf\x9a\xbb\xf7\xf6\xa8\xa1\xca\x84S\xffo\x12B\\LP\x95"N\x9b\xfbx\xf7<\x86{\xbd\xf8]\r\'\xae\x86%_m\x
8b\xfc\xf3\xb2\xe6\x9e\x19\xf7\xd4\xdb\x18\x8d\x10\xf85\xade\x02Q/^bw\x81\xe3\xee^w\xeb\xc5\xd1\x8c\xa8\x9dV\xfeHm\x16/\x11\xef\x8
4+!\xae\xd0]\xd2\xa3\xce\x84F\xc21%\xab\xb5"S\xfcl\x80\x83\xb45*`p\x12\x9b\xcc\xd1/\xc3D\x8b\x8b\x18z\xf9I0p\xb6)C\xc9\xce\x05\x90
\xc3q\xf2\x00Im\xd9M\xf7\xdf7\xe0\x8c\xc9^\xc9\x0ev\xef\x89,\x05\xc9\x1a!2\x1cT\x10>\xd5j\xfex\x05\x0c\xc1\xa1\xa3\xc2qh"3\xdaRcrt
\x91\x81\x95\x9c,G\xee\xca\x91\xcc\n\xbd`\x89\xff&,w~\xfda\r\xa2e\xf1\xa0\xb0T\x0e|\x0b2\xd9_T`ol\x0er\x85\x12\r^\x97\xa9\x0cQ/^bw
Q/^bw\x81\xe3\xee^w\xeb\xc5\xd1\x8c\xa8\x9dV\xfeHm\x16/\x11\xef\x84+!\xae\xd0]\xd2\xa3\xce\x84F\xc21%\xab\xb5"S\xfcl\x80\x83\xb45*
`p\x12\x9b\xcc\xd1/\xc3D\x8b\x8b\x18z\xf9I0p\xb6)C\xc9\xce\x05\x90\xc3q\xf2\x00Im\xd9M\xf7\xdf7\xe0\x8c\xc9^\xc9\x0ev\xef\x89,\x05
\xc9\x1a!2\x1cT\x10>\xd5j\xfex\x05\x0c\xc1\xa1\xa3\xc2qh"3\xdaRcrt\x91\x81\x95\x9c,G\xee\xca\x91\xcc\n\xbd`\x89\xff&,w~\xfda\r\xa2
e\xf1\xa0\xb0T\x0e|\x0b2\xd9_T`ol\x0er\x85\x12\r^\x97\xa9\x0c\xb5sc\xfbRCX\xc2\xcd\x01\xce\x82\xe4\\Ku4j\xf7\x99H4;\xebz'

Deciphered Text:  b'Hello, world!'

Digitally signed message:  b'\x00H@\xbf\x8d\x89\x99\x13w\x89?d\xfd,\x18l\xf5\xa3H\x9b\x18\xc0\xc8\x07\xc7\x01\xc7\\\x8b\x87\x98\xe
a=%\xd3\xbe\xe4\x02\x05\x04#\x19\x1b\'V \x0f\x1a\x15a\x0bpd\xdb\x95\xa9W\xee\xa8\xd6-\x18m\xd0 ~"\xbe\xf4\xd5\x8cqG\xb8\x8au4\xa7X
|\xe5\xf7E\xd9\xbd\x10\x92\xcb\xe9}\x9e\x1e~g\x04{y\x9c\xb0s\xb3\x02\x95\xf2p\xfa\\|\xbe%\xcc5\x19\xcahD\x1du\xc7\xfa\xe6A\xdd\xf9
V0\xcd\x03Nd,\xe8\t\x8d\'pq;[\'}\xd9\x12\xa0S\xa7\x10\x0e\xe3*\xedx\xd2u\xeb\xe5\xe7\xf8\x9a\xca\n\x86\x94\xff\xd9\x8eV\xddY\x0e\x
ee\xb9\x9a\xa3\x84\x94\x8d6\xdb\xa8\x02u\xaf\xd1\xce\xff\xca\xd2\x8d\xac}[\x18\x80b5\xeeS&\x16{\xb6H\xc9\x8d`mz\x80x\x9a\x97K\x9a\
x7ft\x87\x1c\xe3\x06\x13\x99k\xfa\xa5\x926\xe3\x17\x8bt\xb5\x93=HgK:&\xd5 \xa7*\x06/\x8eX\xe9\x95,B\xc4a{\xd7\xaf'

Digital Signature Validation state:  True


Digital Signature Validation state:  False
```

SymmetricCipher

```python
plainText = "Hello, world!".encode()
print("\n\n",plainText,end="\n\n\n")

aes_strategy = AESEncryption()
iv = aes_strategy.generate_iv()

# Example usage of AES128
key128 = aes_strategy.generate_key(16)
encryptor = Encryptor(aes_strategy)
cipher_text_AES128 = encryptor.encrypt_text(plainText, key128, iv)
deciphered_text_AES128 = encryptor.decrypt_text(cipher_text_AES128, key128, iv)
print("\tAES128\nCipherText: ",cipher_text_AES128,"\nDecipheredText: ",deciphered_text_AES128, end="\n\n")

# Example usage of AES192
key192 = aes_strategy.generate_key(24)
encryptor = Encryptor(aes_strategy)
cipher_text_AES192 = encryptor.encrypt_text(plainText, key192, iv)
deciphered_text_AES192 = encryptor.decrypt_text(cipher_text_AES192, key192, iv)
print("\tAES192\nCipherText: ",cipher_text_AES192,"\nDecipheredText: ",deciphered_text_AES192, end="\n\n")

# Example usage of AES256
key256 = aes_strategy.generate_key(32)
encryptor = Encryptor(aes_strategy)
cipher_text_AES256 = encryptor.encrypt_text(plainText, key256, iv)
deciphered_text_AES256 = encryptor.decrypt_text(cipher_text_AES256, key256, iv)
print("\tAES256\nCipherText: ",cipher_text_AES256,"\nDecipheredText: ",deciphered_text_AES256, end="\n\n")

# Example usage of DES
des_strategy = DESEncryption()
key64 = des_strategy.generate_key()
iv_8 = des_strategy.generate_iv()
encryptor = Encryptor(des_strategy)
cipher_text_DES = encryptor.encrypt_text(plainText, key64, iv_8)
deciphered_text_DES = encryptor.decrypt_text(cipher_text_DES, key64, iv_8)
print("\tDES\nCipherText: ",cipher_text_DES,"\nDecipheredText: ",deciphered_text_DES, end="\n\n")
```

```
 b'Hello, world!'


        AES128
CipherText:  b'\x7f7\xd8\xb6\xd8Q\x04A:\x03\x9c\x8e\x1d\x96\x02K'
DecipheredText:  b'Hello, world!'

        AES192
CipherText:  b'L\xf3\xb60\xbe\xbcb_j_\x11 \xa10\xcaM'
DecipheredText:  b'Hello, world!'

        AES256
CipherText:  b"3h\xa3C'N\x18\xb7\xaa\x99\xa0\xe6\xad\xce\xb5%"
DecipheredText:  b'Hello, world!'

        DES
CipherText:  b'sZ\xe4\xc7\xdbc`\x15K.\x03\xc1\x82N\xdd\xcc'
DecipheredText:  b'Hello, world!'
```

Authenticate

```python
if __name__ == "__main__":
    auth = Authenticator()
    print(auth.authenticate_user("admin", "admin@1234"))
    print(auth.add_new_user("admin", "admin@1234"))
    print(auth.authenticate_user("admin", "admin@1234"))
    print(auth.authenticate_user("admin", "admin@_1234"))
    print(auth.add_new_user("admin2", "admin"))
```

```
True
Username Already Exsists! Try using another username.
True
False
Username Already Exsists! Try using another username.
```

KeyManagemet

```python
key_manager = KeyManager()
public, private = key_manager.get_key_pair("admin")
print("Public key: ",public,"\nPrivate Key: ",private)
```


```
Public key:  b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4NshHVBOV005Q1Q0PcS9\nIIrLqPEsXMD3N/CGakMLP4zlrl+XdVbE0hdyzH4
7v1yxWPpex0pHqv3JijU1znJs\n1LXuZDZ+VvlpXpS6jdarfV9zLKShxe2oZ95tOcHMoiavMas6QDinZqxDlBoINLtP\n59kXVkjwWyDT9ktBNtzx6NBv8uV6j3MQqvn7xZB3Vtfn0JMWNxHzttA
HjKerZDX0\nH8ltHRnJ+wDH5AMd7X28YLEo5RQPe9LZPd2N8FdfoQa90Wu58EHScxu30fNtrNZQ\nqhmipEvBkUarUcK/KzYMG28K0CL6NiUMIfnw5OJNDR0C5PP2HE7Bn3z92/BCfipK\nCQIDA
QAB\n-----END PUBLIC KEY-----\n'
Private Key:  b'-----BEGIN PRIVATE KEY-----\nMIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDg2yEdUE5XTT1C\nVDQ9xL0gisuo8SxcwPc38IZqQws/jOWuX5d1V
sTSF3LMfju/XLFY+17HSkeq/cmK\nNTXOcmzUte5kNn5W+WlelLqN1qt9X3MspKHF7ahn3m05wcyiJq8xqzpAOKdmrEOU\nGgg0u0/n2RdWSPBbINP2S0E23PHo0G/y5XqPcxCq+fvFkHdW1+fQk
xY3EfO20AeM\np6tkNfQfyW0dGcn7AMfkAx3tfbxgsSjlFA970tk93Y3wV1+hBr3Ra7nwQdJzG7fR\n822s1lCqGaKkS8GRRqtRwr8rNgwbbwrQIvo2JQwh+fDk4k0NHQLk8/YcTsGffP3b\n8EJ
+KkoJAgMBAAECggEABQt5jtkvRsdnZFj0Cpo8RYuDpfzSOlsEeOtzMpViPljk\nitDfeq0ZkZ6fz116LPERFA5a+8Trfnc+8zQ+Kq13yl5yFKrB0yT5o5r55AaH1rJi\naMMhU4nhXEI56GC0HQO
57Mb7EbeLrvKouf0LB9bXhDvDSabtiJfgFY+q7QaQQWFI\nC6QYLpeNkzMHkPlw9h9pQbjS+kFnlvDpERiaM2BPMniiMaeVTvUwGUTIw8JScDQ8\nE1uOx9mGv7914J5X4Sp/iGSe7Pbl8Jy3Nht
j7YfbsaWBQqMsK6ZBcddRUiE3y2ux\n1F8U9ce5ozTaXHIQWMJjFGLNGRvPzNlIN6pVS2P5sQKBgQDwNzyR0n4INfZKRwjW\nPqqydIBhU3jMPntBq7UVgokguV9TfhfJWvPD+GtPSIwhnjIoB9n
a0nAwLlnfuwrO\nqiZh92kh3DkoQkPAr1/7WZvPx9h7XGZR3U8QmK5aXrH1kQMkkMvS2pn9Gauh6Vut\nZCyVj/OJrvhTUhKlN0CXsrUtsQKBgQDvoYUH3thR82sb7+oeZUMZ9VvRXPhZ9zXY\nn
ZrOeC/1MWKmbA43zzZVbk0WpqtzBARBBxOS4/Y9rVCwfY7QSySSzP2kUOoPij3I\nScDygIh8OcXnNQR7dt5+vY+LhlKy6BeLnTho9DkXW53tmxAYGzLVLouO5NVCufSp\nToRdj9g/2QKBgDxF3
9DG/BCbWZzxjeoT90inqr7A7wJwewTTTLCE/FtVp/GPJfDD\n/X4iiuptzZtx0ivxLFyXiMbOGobkq6WHU/Io3hw84aqUVeO3HWI37byI7GynlpTb\n7sa/SKAsOoLfS+aU3k+dwh2bn1EU+IfdJ
sKwR+84E970LPF/1514obVBAoGBAJ3T\nZ+bTIu52wCjaTvxN/W/vZdXywa+fXluZEgjCp/N531t097hSRZrm1ucARxOtpY+e\ntYmdkuT0bZyQVatikmgNk1DR6hbqGd781JQji8El2+53wMgAd
nJLL2vZwEYNDtTp\na+nUUHXN5apE4si9wTEfQtOOa9dS2o1YJV3qC6ehAoGAHsa4AxWuFDUHl+gAX49Q\nR2Do39dicJaG9UJNEX4bKyUZUAI37QFUU3NttkJwEX78KNZOk71UCFdzbvSBVp64\
nRGySRppIIthVHaZ5oXMNyS2FZs+WqQXNQrfj+5CLBN+r0U9jLAESXktIwugC5Fxq\nCMUIZ4YEhrYZpooZGSSX41O=\n-----END PRIVATE KEY-----\n'
```


```
keys
  admin
    private.pem
    public.pem
```


```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4NshHVBOV005Q1Q0PcS9
IIrLqPEsXMD3N/CGakMLP4zlrl+XdVbE0hdyzH47v1yxWPpex0pHqv3JijU1znJs
1LXuZDZ+VvlpXpS6jdarfV9zLKShxe2oZ95tOcHMoiavMas6QDinZqxDlBoINLtP
59kXVkjwWyDT9ktBNtzx6NBv8uV6j3MQqvn7xZB3Vtfn0JMWNxHzttAHjKerZDX0
H8ltHRnJ+wDH5AMd7X28YLEo5RQPe9LZPd2N8FdfoQa90Wu58EHScxu30fNtrNZQ
qhmipEvBkUarUcK/KzYMG28K0CL6NiUMIfnw5OJNDR0C5PP2HE7Bn3z92/BCfipK
CQIDAQAB
-----END PUBLIC KEY-----
```

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDg2yEdUE5XTT1C
VDQ9xL0gisuo8SxcwPc38IZqQws/jOWuX5d1VsTSF3LMfju/XLFY+17HSkeq/cmK
NTXOcmzUte5kNn5W+WlelLqN1qt9X3MspKHF7ahn3m05wcyiJq8xqzpAOKdmrEOU
Ggg0u0/n2RdWSPBbINP2S0E23PHo0G/y5XqPcxCq+fvFkHdW1+fQkxY3EfO20AeM
p6tkNfQfyW0dGcn7AMfkAx3tfbxgsSjlFA970tk93Y3wV1+hBr3Ra7nwQdJzG7fR
822s1lCqGaKkS8GRRqtRwr8rNgwbbwrQIvo2JQwh+fDk4k0NHQLk8/YcTsGffP3b
8EJ+KkoJAgMBAAECggEABQt5jtkvRsdnZFj0Cpo8RYuDpfzSOlsEeOtzMpViPljk
itDfeq0ZkZ6fz116LPERFA5a+8Trfnc+8zQ+Kq13yl5yFKrB0yT5o5r55AaH1rJi
aMMhU4nhXEI56GC0HQO57Mb7EbeLrvKouf0LB9bXhDvDSabtiJfgFY+q7QaQQWFI
C6QYLpeNkzMHkPlw9h9pQbjS+kFnlvDpERiaM2BPMniiMaeVTvUwGUTIw8JScDQ8
E1uOx9mGv7914J5X4Sp/iGSe7Pbl8Jy3Nhtj7YfbsaWBQqMsK6ZBcddRUiE3y2ux
1F8U9ce5ozTaXHIQWMJjFGLNGRvPzNlIN6pVS2P5sQKBgQDwNzyR0n4INfZKRwjW
PqqydIBhU3jMPntBq7UVgokguV9TfhfJWvPD+GtPSIwhnjIoB9na0nAwLlnfuwrO
qiZh92kh3DkoQkPAr1/7WZvPx9h7XGZR3U8QmK5aXrH1kQMkkMvS2pn9Gauh6Vut
ZCyVj/OJrvhTUhKlN0CXsrUtsQKBgQDvoYUH3thR82sb7+oeZUMZ9VvRXPhZ9zXY
nZrOeC/1MWKmbA43zzZVbk0WpqtzBARBBxOS4/Y9rVCwfY7QSySSzP2kUOoPij3I
ScDygIh8OcXnNQR7dt5+vY+LhlKy6BeLnTho9DkXW53tmxAYGzLVLouO5NVCufSp
ToRdj9g/2QKBgDxF39DG/BCbWZzxjeoT90inqr7A7wJwewTTTLCE/FtVp/GPJfDD
/X4iiuptzZtx0ivxLFyXiMbOGobkq6WHU/Io3hw84aqUVeO3HWI37byI7GynlpTb
7sa/SKAsOoLfS+aU3k+dwh2bn1EU+IfdJsKwR+84E970LPF/1514obVBAoGBAJ3T
Z+bTIu52wCjaTvxN/W/vZdXywa+fXluZEgjCp/N531t097hSRZrm1ucARxOtpY+e
tYmdkuT0bZyQVatikmgNk1DR6hbqGd781JQji8El2+53wMgAdnJLL2vZwEYNDtTp
a+nUUHXN5apE4si9wTEfQtOOa9dS2o1YJV3qC6ehAoGAHsa4AxWuFDUHl+gAX49Q
R2Do39dicJaG9UJNEX4bKyUZUAI37QFUU3NttkJwEX78KNZOk71UCFdzbvSBVp64
RGySRppIIthVHaZ5oXMNyS2FZs+WqQXNQrfj+5CLBN+r0U9jLAESXktIwugC5Fxq
CMUIZ4YEhrYZpooZGSSX41O=
-----END PRIVATE KEY-----
```

2. Integration testing
   Integration testing is performed by the client and server which implement https protocol for secure data exchange it will be explained in details in this next section.

3. How does the suite secure internet services?

Our suite works exactly the same as https protocol except for the certificate part which needs a CA. A dummy certify module was implemented to allow the server to generate a self-signed certificate.

How HTTPS works:

1. Handshake:

- The client initiates a connection to the server and requests a secure connection.
- The server responds by sending its certificate (in our case only the public key), which includes its public key and other information.

2. Certificate Verification:

- The client verifies the server's SSL/TLS certificate. This involves checking the certificate's authenticity, expiration date, and ensuring it was issued by a trusted Certificate Authority (CA).
- If the certificate is valid and trusted, the client proceeds. Otherwise, it may terminate the connection or display a warning to the user.

3. Key Exchange:

- The client generates a symmetric encryption key, which will be used for encrypting data during the session. This key is encrypted using the server's public key obtained from the SSL/TLS certificate.
- The server decrypts the symmetric key using its private key.

4. Secure Connection Establishment:

- With the symmetric key established, both client and server can encrypt and decrypt data using symmetric encryption algorithms (such as AES or DES).
- From this point onwards, all data transmitted between the client and server is encrypted using the symmetric key, providing confidentiality.

5. Data Transfer:

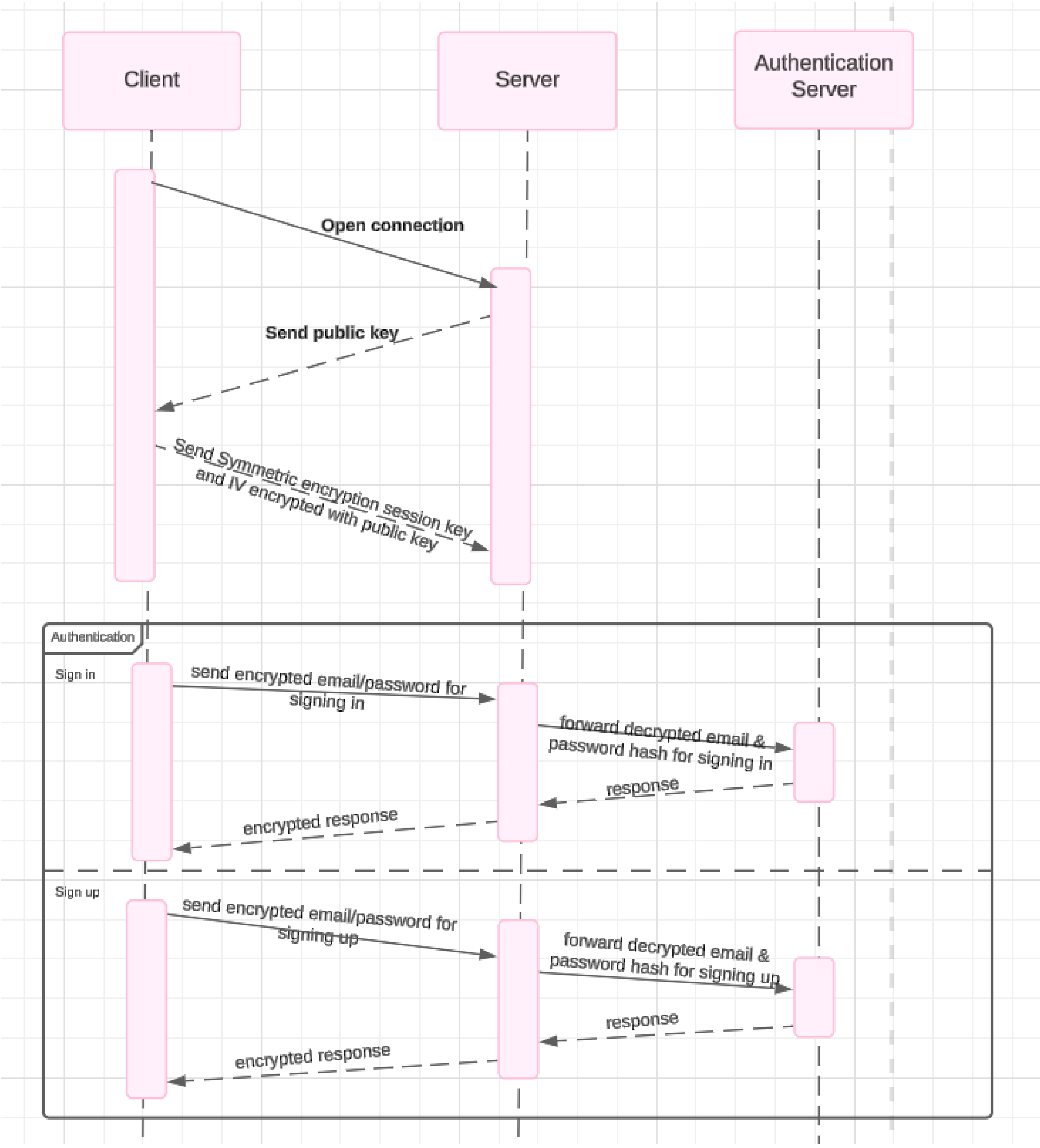
- Once the secure connection is established, the client and server can exchange data as in a regular HTTP connection. However, all data is encrypted before transmission and decrypted upon receipt. [In our case we will use connection to test the authenticate module which authenticates a user by the username and password.]

6. Session Termination:

- When the session is complete or terminated, the connection is closed, and the symmetric key is discarded.
- If the connection needs to be re-established, a new key exchange process will occur.

The following section has a sequence diagram that shows how our implementation of https protocol works and how we integrated all the modules developed in the client and server scripts.

Sequence Diagram



Screenshots of execution

Server end

```
PS D:\Studying\Semester10\Security\Project\Development> & d:/Studying/Semester10/Security/Project
/Development/.virt/Scripts/python.exe d:/Studying/Semester10/Security/Project/Development/server.
py
Got connection from client at:  ('192.168.220.1', 61954)
Sent public key to the client.
Received dictionary: {'mode': 'AES256', 'key': b'\xc3\x90\x10=\xf7\xea\xfe\x18\xdf<?t\xa7@\xf0\x8
e\xba\x10\x90\x95,\x8b!\xc8\xec\xd5e\x19U\xb1[Y', 'iv': b'+p)\x89\xc9\xd3\xd8\xf9\x0c\xb0\xd1\x04
\x00t\x17G'}
Finished Handshake. Waiting for client request.
{'mode': 'SignIn', 'username': 'ziko', 'password': '1234'}
Login status True
```

```
{'mode': 'SignUp', 'username': 'ziko', 'password': '5678'}
Username Already Exsists! Try using another username.
```

```
{'mode': 'SignUp', 'username': 'admin4', 'password': '0987'}
User added successfully.
```

```
User added successfully.
{'mode': 'SignIn', 'username': 'admin4', 'password': '0987'}
```

```
{'mode': 'SignIn', 'username': 'admin4', 'password': '1234'}
Login status False
```

Client end

```
PS D:\Studying\Semester10\Security\Project\Development> d:/Studying/Semester10/Security/Project/D
evelopment/.virt/Scripts/python.exe d:/Studying/Semester10/Security/Project/Development/client.py

Recieved public key from the server.
Key :  b'\xc3\x90\x10=\xf7\xea\xfe\x18\xdf<?t\xa7@\xf0\x8e\xba\x10\x90\x95,\x8b!\xc8\xec\xd5e\x19
U\xb1[Y'
iv :  b'+p)\x89\xc9\xd3\xd8\xf9\x0c\xb0\xd1\x04\x00t\x17G'
Sent session key and iv to server.
Sign in --> 0
Sign up --> 1
0
Username: ziko
Password: 1234
Login status True
Sign in --> 0
Sign up --> 1
Exit --> Enter
```

```
Username: ziko
Password: 5678
Username Already Exsists! Try using another username.
Sign in --> 0
Sign up --> 1
Exit --> Enter
```

```
Username: admin4
Password: 0987
User added successfully.
Sign in --> 0
Sign up --> 1
Exit --> Enter
```

```
Username: admin4
Password: 0987
Login status True
```

```
Username: admin4
Password: 1234
Login status False
```

Code Appendix

Phase 4

Server.py

```
import socket
import pickle
from crypto.AsymmetricCipher import RSA
from KeyManagement import KeyManager
from crypto.SymmetricCipher import Encryptor, AESEncryption, DESEncryption
from Authenticate import Authenticator

class Server:
    def __init__(self) -> None:
        # Create a socket object
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Get local machine name
        self.host = socket.gethostname()
        self.port = 12345
        self.session_key = None
        self.session_iv = None
        self.session_encryptor = None
        self.auth = Authenticator()


    def server_up(self):
        # Bind to the port
        self.server_socket.bind((self.host, self.port))
        # Listen for incoming connections
        self.server_socket.listen(10)

        while True:
            # Establish connection with client
            client_socket, addr = self.server_socket.accept()
            print('Got connection from client at: ', addr)

            # 1. generate public and private key pairs and send the public key
to the client
            key_manager = KeyManager()
            public_key_pem, private_key_pem =
key_manager.get_key_pair("server")
            rsa = RSA()
            private_key = key_manager.pem_to_rsa_private_key(private_key_pem)
            client_socket.send(pickle.dumps({"public_key":public_key_pem}))
            print("Sent public key to the client.")

            # 2. recieve the session random key and session symmetric
encryption algorithm
            data = client_socket.recv(4096)
```

```
            if not data:
                print("Error! Client didnt send session random key.")
            else:
                # Unpickle received data
                received_dict = pickle.loads(data)
                received_dict["mode"] = rsa.decrypt(received_dict["mode"],
private_key).decode()
                self.session_key = received_dict["key"] =
rsa.decrypt(received_dict["key"], private_key)
                self.session_iv = received_dict["iv"] =
rsa.decrypt(received_dict["iv"], private_key)
                print("Received dictionary:", received_dict)
                # set the session encryptor
                if received_dict["mode"][:3] == "AES":
                    self.session_encryptor = Encryptor(AESEncryption())
                elif received_dict["mode"][:3] == "DES":
                    self.session_encryptor = Encryptor(DESEncryption())
                print("Finished Handshake. Waiting for client request.")


            data =client_socket.recv(4096)
            while(data):
                data =  pickle.loads(data)
                # decrypt
                data_recieved =
{"mode":self.session_encryptor.decrypt_text(data["mode"],self.session_key,self
.session_iv).decode(),
                            "username":
self.session_encryptor.decrypt_text(data["username"],self.session_key,self.ses
sion_iv).decode(),
                            "password" :
self.session_encryptor.decrypt_text(data["password"],self.session_key,self.ses
sion_iv).decode()}
                print(data_recieved)
                # serve
                if data_recieved["mode"] == "SignIn":
                    result =
self.auth.authenticate_user(data_recieved["username"],data_recieved["password"
])
                    result = "Login status " + str(result)
                elif data_recieved["mode"] == "SignUp":
                    result = self.auth.add_new_user(data_recieved["username"],
data_recieved["password"])
                # respond
                print(result)
                client_socket.send(pickle.dumps(self.session_encryptor.encrypt
_text(result.encode(), self.session_key, self.session_iv)))
                data =client_socket.recv(4096)
```

```
        # Close the server socket
        self.server_socket.close()



if __name__ == "__main__":
    server = Server()
    server.server_up()
```

client.py

```
import socket
import pickle
from crypto.AsymmetricCipher import RSA
from crypto.SymmetricCipher import AESEncryption, Encryptor
from KeyManagement import KeyManager



class Client:
    def __init__(self) -> None:
        # Create a socket object
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Get local machine name
        self.host = socket.gethostname()
        self.port = 12345

        # generate iv and random key
        aes_strategy = AESEncryption()
        self.iv = aes_strategy.generate_iv()
        self.key256 = aes_strategy.generate_key(32)
        self.encryptor = Encryptor(aes_strategy)


    def client_up(self):
        # Connect to the server
        self.client_socket.connect((self.host, self.port))

        # 1. Receive the public key from the server
        response = self.client_socket.recv(4096)
        if not response:
            print("Error! Server did not send its public key.")
        else:
            print("Recieved public key from the server.")
            # 2. Generate a random session key and iv, and send it along with
the encryption scheme to the server
            # get public key object
            rsa = RSA()
```

```
            rsa_public_key =
KeyManager().pem_to_rsa_public_key(pickle.loads(response)["public_key"])

            # encrypt the random key and iv with the servers public key
            data_to_send = {"mode":
rsa.encrypt("AES256".encode(),rsa_public_key),
                            "key":rsa.encrypt(self.key256, rsa_public_key),
                            "iv":rsa.encrypt(self.iv, rsa_public_key)}

            # send the encrypted message
            self.client_socket.send(pickle.dumps(data_to_send))
            print("Key : ",self.key256,"\niv : ", self.iv)
            print("Sent session key and iv to server.")

        # prompt the user to login then encrypt all communication using key256
        user_request = input("Sign in --> 0\nSign up --> 1\n")
        while user_request:
            username = input("Username: ")
            password = input("Password: ")
            if user_request == "0":
                mode = "SignIn"
            elif user_request == "1":
               mode = "SignUp"
            else:
                break
            data_to_send =
{"mode":self.encryptor.encrypt_text(mode.encode(),self.key256,self.iv),
                            "username":
self.encryptor.encrypt_text(username.encode(),self.key256,self.iv),
                            "password" :
self.encryptor.encrypt_text(password.encode(),self.key256,self.iv)}
            self.client_socket.send(pickle.dumps(data_to_send))
            result = pickle.loads(self.client_socket.recv(4096))
            print(self.encryptor.decrypt_text(result, self.key256,
self.iv).decode())
            user_request = input("Sign in --> 0\nSign up --> 1\nExit -->
Enter\n")

        # Close the connection
        self.client_socket.close()


if __name__=="__main__":
    client = Client()
    client.client_up()
```

Code from previous phases

KeyManagement.py

```
import os
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from crypto.AsymmetricCipher import RSA

class KeyManager:
    def __init__(self, ):
        self.key_dir = "./keys"
        if not os.path.exists(self.key_dir):
            os.mkdir(self.key_dir)

    def get_key_pair(self, email):
        user_key_dir = self.key_dir+ "/"+ email
        # check if the email already has key pair then return them
        if os.path.exists(user_key_dir):
            with open((user_key_dir+"/"+'public.pem'), 'rb') as f:
                public_key = f.read()
            with open((user_key_dir+"/"+'private.pem'), 'rb') as f:
                private_key = f.read()
        else:
            # 1. create a key pair
            rsa = RSA()
            public_key, private_key = rsa.generate_key_pair()
            # 2. create a directory by the email and store the key pair there
            os.mkdir(user_key_dir)
            # 3. store the key pair
            with open((user_key_dir+"/"+'private.pem'), 'wb') as f:
                private_key = private_key.private_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PrivateFormat.PKCS8,
                    encryption_algorithm=serialization.NoEncryption()
                )
                f.write(private_key)
            with open((user_key_dir+"/"+'public.pem'), 'wb') as f:
                public_key = public_key.public_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PublicFormat.SubjectPublicKeyInfo
                )
                f.write(public_key)
        return public_key, private_key

    def pem_to_rsa_public_key(self, public_pem):
        # Load the PEM-encoded public key
        public_key = serialization.load_pem_public_key(
            public_pem,
```

```
            backend=default_backend()
        )
        return public_key

    def pem_to_rsa_private_key(self, private_pem):
       # Load the PEM-encoded private key
        private_key = serialization.load_pem_private_key(
            private_pem,
            password=None,  # Provide the password if the key is encrypted
            backend=default_backend()
        )
        return private_key

if __name__=="__main__":
    key_manager = KeyManager()
    public, private = key_manager.get_key_pair("admin")
    print("Public key: ",public,"\nPrivate Key: ",private)
```

Authenticate.py

```
import psycopg2
from Hash.SHA import hash

class Authenticator:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super(Authenticator, cls).__new__(cls, *args,
**kwargs)
        return cls._instance

    def __init__(self) -> None:
        self.conn = psycopg2.connect(host="localhost",
dbname="postgres",user="postgres", password="admin@1234", port="5432")
        self.cur = self.conn.cursor()

    def __del__(self):
        self.cur.close()
        self.conn.close()

    def authenticate_user(self, username, password):
        password_hash = hash(password).hex()
        # Execute the query to fetch all users
        self.cur.execute(""" SELECT username, password_hash FROM
public."UA_DB" """)

        # Fetch all rows
        rows = self.cur.fetchall()
```

```
        # check if the user is in records
        for row in rows:
            db_username, db_password_hash = row
            if username == db_username and
password_hash==db_password_hash[2:]:
                return True
        return False

    def add_new_user(self, username, password):
        # 1. check that the username doesnt already exists
        # Execute the query to fetch all users
        self.cur.execute(""" SELECT username FROM public."UA_DB" """)

        # Fetch all rows
        rows = self.cur.fetchall()
        for row in rows:
            if row[0] == username:
                return "Username Already Exsists! Try using another username."
        # 2. add the username and hash of the password to the database
        try:
            # Execute the SQL statement to insert the user into the table
            self.cur.execute(""" INSERT INTO public."UA_DB" (username,
password_hash) VALUES (%s, %s)""", (username, hash(password)))

            # Commit the transaction
            self.conn.commit()
            return "User added successfully."

        except psycopg2.Error as e:
            print("Error:", e)


if __name__ == "__main__":
    auth = Authenticator()
    print(auth.authenticate_user("admin", "admin@1234"))
    print(auth.add_new_user("admin", "admin@1234"))
    print(auth.authenticate_user("admin", "admin@1234"))
    print(auth.authenticate_user("admin", "admin@_1234"))
    print(auth.add_new_user("admin2", "admin"))
```

Hash.SHA.py

```
from cryptography.hazmat.primitives import hashes


def hash(text):
    # Choose a hash algorithm (SHA-256 in this example)
```

```
    algorithm = hashes.SHA256()
    hasher = hashes.Hash(algorithm)
    hasher.update(text.encode('utf-8'))
    # Finalize the hash to obtain the digest (hash value) of the data
    return hasher.finalize()

if __name__ == '__main__':
    plainText = "Hello, world!".encode()
    hashed_data = hash(plainText)
    print("Text: ",plainText,"\nHash: ",hashed_data)
```

crypto.AsymmenticCipher.py

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes


class RSA():
    def generate_key_pair(self):
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
            )
        public_key = private_key.public_key()
        return public_key, private_key

    def encrypt(self, text, public_key):
        ciphertext = public_key.encrypt(
                    text,
                    padding.OAEP(
                        mgf=padding.MGF1(algorithm=hashes.SHA256()),
                        algorithm=hashes.SHA256(),
                        label=None
                    )
        )
        return ciphertext

    def decrypt(self, text, private_key):
        decrypted_text = private_key.decrypt(
                text,
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None
                )
        )
        return decrypted_text
```

```python
    def generate_digital_signature(self, message, private_key):
        signature = private_key.sign(
                message,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
        )
        return signature

    def verify_digital_signature(self, message, signature, public_key):
        try:
            public_key.verify(
                signature,
                message,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            return True
        except:
            return False


if __name__ == '__main__':

    plainText = "Hello, world!".encode()

    rsa_object = RSA()
    public_key, private_key = rsa_object.generate_key_pair()

    # Encryption
    ciphertext = rsa_object.encrypt(plainText, public_key)
    deciphered_text = rsa_object.decrypt(ciphertext, private_key)
    print("\nText: ",plainText,"\n\nCiphertext: ",ciphertext,"\n\nDeciphered
Text: ",deciphered_text, end="\n\n")

    # Digital Signature
    digital_signature = rsa_object.generate_digital_signature(plainText,
private_key)
    print("Digitally signed message: ", digital_signature)
    validate = rsa_object.verify_digital_signature(plainText,
digital_signature, public_key)
    print("\nDigital Signature Validation state: ",validate,end="\n\n")
```

```
    #Failed verification
    public_key2, private_key2 = rsa_object.generate_key_pair()
    validate2 = rsa_object.verify_digital_signature(plainText,
digital_signature, public_key2)
    print("\nDigital Signature Validation state: ",validate2,end="\n\n")
```

Crypto.symmetricCipher.py

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
import os


# Define the interface for encryption algorithm
class EncryptionStrategy:
    def encrypt(self, text, key, iv):
        pass
    def decrypt(self, text, key, iv):
        pass


class DESEncryption(EncryptionStrategy):
    def generate_key(self):
        return os.urandom(8)

    def generate_iv(self):
        return get_random_bytes(8)

    def encrypt(self, text, key, iv):
        cipher = DES.new(key, DES.MODE_CBC, iv)
        ciphertext = cipher.encrypt(pad(text, DES.block_size))
        return ciphertext

    def decrypt(self, text, key, iv):
        cipher = DES.new(key, DES.MODE_CBC, iv)
        plaintext = cipher.decrypt(text)
        return unpad(plaintext, DES.block_size)


# Implement encryption algorithms
class AESEncryption(EncryptionStrategy):
    def generate_key(self, size):
        return os.urandom(size)

    def generate_iv(self):
```

```
        return os.urandom(16)

    def pad(self, plaintext):
        padder = padding.PKCS7(128).padder()
        padded_data = padder.update(plaintext)
        padded_data += padder.finalize()
        return padded_data

    # Function to unpad the plaintext
    def unpad(self, padded_data):
        unpadder = padding.PKCS7(128).unpadder()
        unpadded_data = unpadder.update(padded_data)
        unpadded_data += unpadder.finalize()
        return unpadded_data

    def encrypt(self, text, key, iv):
        cipher = Cipher(algorithms.AES(key), modes.CTR(iv),
backend=default_backend())
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(self.pad(text)) + encryptor.finalize()
        return ciphertext

    def decrypt(self, text, key, iv):
        cipher = Cipher(algorithms.AES(key), modes.CTR(iv),
backend=default_backend())
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(text) + decryptor.finalize()
        return self.unpad(plaintext)


# Context class that uses the chosen encryption strategy
class Encryptor:
    def __init__(self, strategy:EncryptionStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy:EncryptionStrategy):
        self.strategy = strategy

    def encrypt_text(self, text, key, iv):
        return self.strategy.encrypt(text, key, iv)

    def decrypt_text(self, text, key, iv):
        return self.strategy.decrypt(text, key, iv)


# Example usage
if __name__ == "__main__":

    plainText = "Hello, world!".encode()
```

```
    print("\n\n",plainText,end="\n\n\n")

    aes_strategy = AESEncryption()
    iv = aes_strategy.generate_iv()

    # Example usage of AES128
    key128 = aes_strategy.generate_key(16)
    encryptor = Encryptor(aes_strategy)
    cipher_text_AES128 = encryptor.encrypt_text(plainText, key128, iv)
    deciphered_text_AES128 = encryptor.decrypt_text(cipher_text_AES128,
key128, iv)
    print("\tAES128\nCipherText: ",cipher_text_AES128,"\nDecipheredText:
",deciphered_text_AES128, end="\n\n")

    # Example usage of AES192
    key192 = aes_strategy.generate_key(24)
    encryptor = Encryptor(aes_strategy)
    cipher_text_AES192 = encryptor.encrypt_text(plainText, key192, iv)
    deciphered_text_AES192 = encryptor.decrypt_text(cipher_text_AES192,
key192, iv)
    print("\tAES192\nCipherText: ",cipher_text_AES192,"\nDecipheredText:
",deciphered_text_AES192, end="\n\n")

    # Example usage of AES256
    key256 = aes_strategy.generate_key(32)
    encryptor = Encryptor(aes_strategy)
    cipher_text_AES256 = encryptor.encrypt_text(plainText, key256, iv)
    deciphered_text_AES256 = encryptor.decrypt_text(cipher_text_AES256,
key256, iv)
    print("\tAES256\nCipherText: ",cipher_text_AES256,"\nDecipheredText:
",deciphered_text_AES256, end="\n\n")

    # Example usage of DES
    des_strategy = DESEncryption()
    key64 = des_strategy.generate_key()
    iv_8 = des_strategy.generate_iv()
    encryptor = Encryptor(des_strategy)
    cipher_text_DES = encryptor.encrypt_text(plainText, key64, iv_8)
    deciphered_text_DES = encryptor.decrypt_text(cipher_text_DES, key64, iv_8)
    print("\tDES\nCipherText: ",cipher_text_DES,"\nDecipheredText:
",deciphered_text_DES, end="\n\n")
```

certify.py

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from cryptography import x509
from cryptography.x509.oid import NameOID
from datetime import datetime, timedelta, timezone
```

```
from cryptography.hazmat.primitives import hashes



def create_selfsigned_certificate(private_key, email):
    # in self-signed certificate we will make the subject same as issuer
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "EGY"),
        x509.NameAttribute(NameOID.COMMON_NAME, f"{email}@ca_self-
signed.com"),
    ])
    # create the certificate
    ca = x509.CertificateBuilder().subject_name(
                                                        subject
                                                    ).issuer_name(
                                                        issuer
                                                    ).public_key(
                                                        private_key.public_key()
# Accessing the public key from private_key
                                                    ).serial_number(
                                                        x509.random_serial_number(
)
                                                    ).not_valid_before(
                                                        datetime.now(timezone.utc)
 # Use timezone.utc here
                                                    ).not_valid_after(
                                                        datetime.now(timezone.utc)
+ timedelta(days=365)  # Use timezone.utc here
                                                    ).sign(private_key,
hashes.SHA256())
    return ca


def verify_certificate(public_key, certificate_pem, email):
    try:
        # convert pem object to certificate object
        cert = x509.load_pem_x509_certificate(certificate_pem,
default_backend())

        # verifty that certificate public key is the same as shared one
        cert.public_key().verify(
            cert.signature,
            cert.tbs_certificate_bytes,
            padding.PKCS1v15(),
            cert.signature_hash_algorithm,
        )

        public_key.verify(
            cert.signature,
```

```
            cert.tbs_certificate_bytes,
            padding.PKCS1v15(),
            cert.signature_hash_algorithm,
        )

        # check the certificate issuer
        issuer_common_name =
cert.issuer.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
        expected_issuer_common_name = f"{email}@ca_self-signed.com"
        if issuer_common_name != expected_issuer_common_name:
            return False

        # check the certificate validity period
        not_valid_before = cert.not_valid_before_utc
        not_valid_after = cert.not_valid_after_utc
        now = datetime.now(timezone.utc)
        if now < not_valid_before or now > not_valid_after:
            return False

        return True
    except (ValueError, IndexError, x509.InvalidSignature):
        return False
```