Secure Communication Protocol Implementation

Assignment 2

Submitted to:

Mam Urooj Ghani

Organization Name:

FAST National University of Computer and Emerging Sciences

Report Prepared by:

Faizan Pervaz, Daniyal Imran

Date of Submission:

27 October, 2023

Contact Information:

I200565@nu.edu.pk

I200940@nu.edu.pk

Roll No:

20I-0565, 20I-0940

Confidentiality:

The report is not confidential and can be freely shared.

Disclaimer:

This report is intended for educational and moral purposes only. Unauthorized use of the information contained here is strictly prohibited.

Table Of Contents

The implementation is done in Python, a versatile and widely used programming language known for its readability and a rich ecosystem of cryptographic libraries.

1. Introduction

This code illustrates the basic implementation or establishment of a secure client-server communication protocol, focusing on data security when sending data. It uses SSL/TLS, DDH and PKI key exchange, and AES, RSA to create an encrypted tunnel between the server and client to ensure secure exchange of messages being hashed through the SHA-256.

The structure of the code ensures that it generates and allows secure transmission of encryption keys.

Such keys are used as the basis for encrypting and decrypting data transferred between the client and the web server. In this regard, We use Diffie-Hellman key exchange or PKI, which allows us to generate a public key while still protecting the private key from being leaked during the communication process.

2. Handshake Protocol (SSL/TLS configuration)

Both the client and server use SSL/TLS for secure communication, and the server provides digital certificates for authentication,

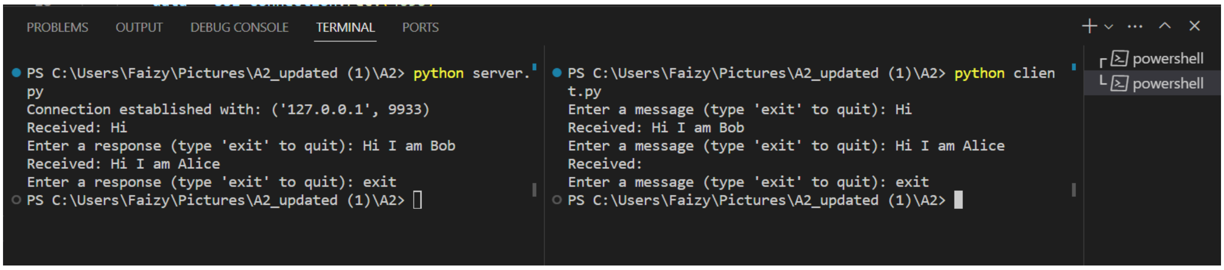
This configures the server's SSL context using the file "server.crt" that contains the server's certificate and the file "server.key" that contains the server's private key.

However, the client uses the CA certificate file to check its server's certificate.

3. Data encryption and decryption

Data exchanged between server and client is encrypted using AES or RSA encryption:

- When the server receives data from the client, it uses the shared secret key to decrypt the data.
- Additionally, just as the server uses a shared key to encrypt its data before sending it to the client, the client uses the same key to decrypt the data it receives from the server.



4. Communication process

The code orchestrates a simplified communication process to ensure secure data exchange:

- The server listens on the specified host and port, waiting for incoming client connections.
- Based on the connection request, the server and client establish an SSL/TLS connection to ensure the security and integrity of data transmission.
- Data is exchanged in encrypted form between client and server, ensuring data confidentiality and security.

- Both clients and servers are able to send and receive messages securely.

5. Certificates & Keys Generated:

- **server.key (private key)**


```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC8bkNqHd4AyEEe
Y/A7NTBg+Spx0Clg+bC9FZ502EDeJKhZOxBt5D6yNoJ0oIh+R5+VmbL9ZCcprrDy
NBQJ9UJl4ML2K3jDsvti/gdxpoQGdgwFQSCblkPZsqyHoE0GZWXp96SraHApbQTP
+JaGAG5mjv8Gw0nWeq+vP99ukABgSjLp4v8ybBhUaaBRota/lgSO9TnKatBVZoQs
RVyUX85McPSkDqPx5Wcv/tIqVx4/Vvl30bvmUcONQ/I6Y2WF2ntjlcnOeYT3EaHf
jCCZOy416WkK/akqnNwrjA5EfpulmNAlzFPwlzMk/iVssApo3Le1T55a6SWfTwXC
wVpnfafvAgMBAAECggEAQzwS2PP0SZIqziW5y96UNx3hG8A7a4tcmXNz2VxqdPle
ioxNNfDpwBw8G1JGscxrsG0p1Uldwf0/zQRxHAaD3SdAWac0TZv5IB71WBOi601G
x78v2+RXrH+ZGUHFCiG3Ji9DdhfrYZSxUWXxi7nuqrCeZfLbz2hydJVVa7V9NN3X
3odFX53FB/B6BPJtZt4+1VL9YxY1Tzx1zG78HSHHrDYZDH011a/kceHjx1Tlb15w
DBHdUWiHqd8C8BRQP4GNu/TyPIsuNAg0TIrUWCzeoVurfewt1jK8F/bmcYSgEAwK
38dzB6uG67sJx6cH6WmWQuOwDCzFnmnRbxgyfF3xjQKBgQDpCleW+ISgDU+weO9v
+xl7azWKVwy5lGl8MOg5x3vUu3VH+dVO3WtU/oLcpD6IXfxKr/saMBoeWzXBHfff
WiC9pqmDOKe4zvLfnGbKEHXHW+9gPHn9GRKSdgEl0qsOJTWU1JCIAV7wwYdAkeCX
Fa6qUh74L61X8m1wnbwyqnydWwKBgQDO/srdbNBIFbvcVWMU9EacqeJN5FEHnTst
56krtdS4xix1dMU0OldryLPo2u/IENtC2ZL+iDDKmpzDIHInl6+L5lRgzIe5emQb
bSTsWuvDfzxWz8FICax2a/Wm3tcOFoqT3hRfHwmYTnqKEjY8Mb8ZJ/ynrl6JT+Zi
fIKwPbJ//QKBgQCImS5UaSHca1ENwSWgfxAlvuboSzRDR5I5YLWOwLZ+MM+DPBdj
nfg/Htx4FrIs3uJ2qQbIB/AXYSF2LGnR+xN790gficMOWggVOKkrwlA3ZlU/FNPw
npS7Uv5DPGRV60uoDJ0Xi64p81aja565ENWMMozCr7es+IZb36mkDTj0RwKBgDjr
HRN4CWnY+BhlLmKjrQsFN+JdRt7GIHDGA+GuFTld3PnLSzLKXCGaRcZg9ZBY+kHO
nDn7bxc3HqYVNO6oKjBY/JjFhQi+m+piv8VyVuQiB5CDfk11w4OouhrRqecI0cBJ
T+a8HSJRaiavTVSOBVNAiJv/OaWeX+ZzAGi//mZtAoGAdbZ9ja7j/hFSFlUBBqkN
vXEFD2LZHNinVIOeyKFVU/b1uoMcbYKA7fy2RcrWxWp0Sao/9ZK//IOhTKtwtcvH
zQ1YvVGGl2/70AGD1e56/2hoWgygw0dGpEEdF7vIRFfiq/be35IwsVIH5tIBgkrp
3FjapqThCcv9SytOyLP9kOU=
-----END PRIVATE KEY-----
```


- **server.crt (server's digital certificate)**

```
-----BEGIN CERTIFICATE-----
MIIDOTCCAiECFD1Kl5WtvsoaW9mondZH4aLl95h6MA0GCSqGSIb3DQEBCwUAMFkx
CzAJBgNVBAYTAkFVMRMwEQYDVQQIDApTb21lLVN0YXRlMSEwHwYDVQQKDBhJbnRl
cm5ldCBXaWRnaXRzIFB0eSBMdGQxEjAQBgNVBAMMCWxvY2FsaG9zdDAeFw0yMzEw
MjUxODI0MjFaFw0yMzExMjQxODI0MjFaMFkxCzAJBgNVBAYTAkFVMRMwEQYDVQQI
DApTb21lLVN0YXRlMSEwHwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQx
EjAQBgNVBAMMCWxvY2FsaG9zdDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoC
ggEBALxuQ2od3gDIQR5j8Ds1MGD5KnHQKWD5sL0VnnTYQN4kqFk7EG3kPrI2gnSg
iH5Hn5WZsv1kJymusPI0FAn1QmXgwvYreMOy+2L+B3GmhAZ2DAVBIJuWQ9myrIeg
TQZlZen3pKtocCltBM/4loYAbmaO/wbDSdZ6r68/326QAGBKMuni/zJsGFRpoFGi
1r+WBI71Ocpq0FVmhCxFXJRfzkxw9KQOo/HlZy/+0ipXHj9W+XfRu+ZRw41D8jpj
ZYXae2OVyc55hPcRod+MIJk7LjXpaQr9qSqc3CuMDkR+m6WY0CXMU/CXMyT+JWyw
Cmjct7VPnlrpJZ9PBcLBWmd9p+8CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAlb4a
IMQYNok2+lD5FheIYBvdZMbwOa69Q3zWtw+ejXdSZJmgPjlCOe0w3PbO9XkKQdr5
i0JLw/QcdwGcJuKDAP7Oc9oE4GiAzsTUhTN71Qckc8cPdKMLNePrtrx9jLIg/1nu
cYzrpYVQzf+PTnZn+CxcwHdW7BTT9U166KFiQhjhbdVcGjtBfJuZZ4RXv/fkggwD
Jez3R/yzsUCHQvbygCsA2537Ofq41zHjS3foADjkpjsBw/Mkk72g0//heLJ5ZRiA
mQnCzt2UwxeBeo2y/WpitQEs4ALKRZJJag4Dd0xg5kcI2j+1Wak95dnhmt1Jr/tq
MkbwfzZPGcj+gaeLwQ==
-----END CERTIFICATE-----
```

- **server.cr**

```
-----BEGIN CERTIFICATE REQUEST-----
MIICnjCCAYYCAQAwWTELMAkGA1UEBhMCQVUxEzARBgNVBAgMClNvbWUtU3RhdGUx
ITAfBgNVBAoMGEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDESMBAGA1UEAwwJbG9j
YWxob3N0MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvG5Dah3eAMhB
HmPwOzUwYPkqcdApYPmwvRWedNhA3iSoWTsQbeQ+sjaCdKCIfkeflZmy/WQnKa6w
8jQUCfVCZeDC9it4w7L7Yv4HcaaEBnYMBUEgm5ZD2bKsh6BNBmVl6fekq2hwKW0E
z/iWhgBuZo7/BsNJ1nqvrz/fbpAAYEoy6eL/MmwYVGmgUaLWv5YEjvU5ymrQVWaE
LEVclF/OTHD0pA6j8eVnL/7SKlceP1b5d9G75lHDjUPyOmNlhdp7Y5XJznmE9xGh
34wgmTsuNelpCv2pKpzcK4wORH6bpZjQJcxT8JczJP4lbLAKaNy3tU+eWukln08F
wsFaZ32n7wIDAQABoAAwDQYJKoZIhvcNAQELBQADggEBAHwzGtTSOUbsH1IpJQMB
xJh9tLaRuMEq8833DW7768zHg3S6OIW8681hLScZjwq2BPdvVdZY8yqpq1xt0L1K
CUVBtw2QGMBuesYRLrM21LljXIxjjO7jt7ckpMnRwH5QKk4XsBYTZYETngEezfMz
IJYKyzjagVFmMb7KHzbnr6bwlQxODP6OIRGsBw4wqaOor/SXnWaLzjI1l3lh8/ob
Zhlfx3BqWu/7V/np1PPxB6944zdI5cUmZrmJ/j/XhXHJXOWswtr7sooZQNpIu8sc
GN4C4TvPJaPqMdZ9zlsKxiRRCG1iXqGCS/Bdaki+4A1di18hvtpZBoNMpyi8aaPR
Wko=
-----END CERTIFICATE REQUEST-----
```

- **ca.key (private key)**

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDke/6TUbB6PLbd
sG60wbG4IXaqSlrYsdisqTk092Q45vBVUB+oN5pQ7QFSzvxPPjGXzeeIZXZ/jk5N
I10zYCyqrmPcP++p4oCugoaJDPgC5v12h7vc25aDcx+XeQkrrFH74aFioThJhIN9
jUl3lgxa7lxBpfhuLQYoffIwEtUk0Jbu6TAni4mz5bMv2PWKmNjChiKRjqiOEag1
Hug5MH7OYsTq0IfQzTF2DyZ1N6WCiC4VFRgjbhbtybpbVE6nYne/3PY1AlcMXWla
9+/IP472DSHpGW42WKuVP8ZhtpOZJ+WqAGhcmd7XX3d/ynsV7kIO2cr0Gqa74+gq
Xc1+09IzAgMBAAECggEAAJKaGROc21zVmA4Ovk8Wtpv8yb7F471fQxlNL14Ys5SM
vzZwAzVK1sv5kiusV8YO1fwo+59P2biWN6d/43Mc9pVw8O3131LfdBCEy/POjdhY
Xep24Tk9AXKrt0QtsO2NrR/nPa9EDnFWZdi3cCnZL+5XwC+OkP5ZTZDAlGKi9Slc
8d4lwqfUlVQyggdtRdvtgs6CD6mP2/j/k9c1HR2JgsYQVc3Un1rkYVrQBiL+kHTO
Ab20HCRC0cs+Bs2XqFH8no1jygBzM6RJQrVWffQUupLKej+3M21FOwi9IG2sjIzh
nRTs2JGLdhkHsw+GytCMpmuVHS8otGnJx4yg643KkQKBgQD/CFO1EVrJ0EsgaGCO
2ob8tVNK4itlTig9chRyZy2R0YaATFIHyK2TsiaJ5ojNu1Kxj1aizVWTDxoP4MvA
xTu6d2sVUnAkX9kZHi++bEX3R79SgJ21W2XsyjFTi/7AEsgtN1ue0pocII3XtLyH
oobMb9usPXtdUcGr2NCjWFjJewKBgQDlWeKoiF5h12t+vjiNFuhMvGo7ONXlIvvW
x++h79InyNKS+0jnWsnZS/LuUF8h058XAOnsxl2N4r70HqfQkuAQxo36E6qzmUkC
5cjraD4kixdbJn5tRZPyGIKf4vVkzz3tnDrrzqhDpXyRgn3CFfAsPsC8fJbquTIn
w0OWRONwqQKBgQDJQcL9EvL4bB29N2FrmlwINPfEEnN4gJMU2l/0YCP//L8NpCzf
46fG9EGhdYHkCL2bTahq1WuAn9xLR3TDbWJTJkYlD2db7R+fyMiY100ndUEwVDsg
kT3//Rer/MBOxwOdWyA8V6oyJCaLYR+eUc3aqzNJ20LtQ8V4XaNYJDH83QKBgE8G
99jGlG72QW38sZO8DvNSAPDDFsDLmydY4TNVZX6b7iMDPw2o9OBRETYYr48CUlEk
2XXirB3VwaJwZbayxU5CfG1tFWapLMU41FB5LOB+pN+dlfalAONsmqXpGFFSL66O
JKdYIBafERs6cYbM9GLqhJLuAzqB8cxNth4zQoNxAoGAQ56rODO6DUc3VFFFcmpH
oOeQdMS+/8aWLluyRdvZBh89Uz2TGsoDuhJ2eiduHdvjSh6orw0MnBJRj2ePGxzS
mTeAvjVKQoyERxBFISwoSyDW9La/+GOLsThNapVF4PPNBFHsC53oXnW0z1BahwyM
dvnFXaF+gnIAocLXDiCRMgg=
-----END PRIVATE KEY-----
```


- **ca.crt (request of digital certificate)**

```
-----BEGIN CERTIFICATE-----
MIIDizCCAnMCFFO2CTwfcDb84vSQtJWqvyp+NHooMA0GCSqGSIb3DQEBCwUAMIGB
MQswCQYDVQQGEwJQSzEPMA0GA1UECAwGUHVuamFiMQwwCgYDVQQHDANJc2IxDTAL
BgNVBAoMBEZhc3QxDTALBgNVBAsMBEZhc3IxDTALBgNVBAMMBEZhc3QxJjAkBgkq
hkiG9w0BCQEWF2ZhaXphbi5wZXJ2YXpAZ21haWwuY29tMB4XDTIzMTAyNTE3NTQ0
MVoXDTIzMTEyNDE3NTQ0MVowgYExCzAJBgNVBAYTAlBLMQ8wDQYDVQQIDAZQdW5q
YWIxDDAKBgNVBAcMA0lzYjENMAsGA1UECgwERmFzdDENMAsGA1UECwwERmFzcjEN
MAsGA1UEAwwERmFzdDEmMCQGCSqGSIb3DQEJARYXZmFpemFuLnBlcnZhekBnbWFp
bC5jb20wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDke/6TUbB6PLbd
sG60wbG4IXaqSlrYsdisqTk092Q45vBVUB+oN5pQ7QFSzvxPPjGXzeeIZXZ/jk5N
I10zYCyqrmPcP++p4oCugoaJDPgC5v12h7vc25aDcx+XeQkrrFH74aFioThJhIN9
jUl3lgxa7lxBpfhuLQYoffIwEtUk0Jbu6TAni4mz5bMv2PWKmNjChiKRjqiOEag1
Hug5MH7OYsTq0IfQzTF2DyZ1N6WCiC4VFRgjbhbtybpbVE6nYne/3PY1AlcMXWla
9+/IP472DSHpGW42WKuVP8ZhtpOZJ+WqAGhcmd7XX3d/ynsV7kIO2cr0Gqa74+gq
Xc1+09IzAgMBAAEwDQYJKoZIhvcNAQELBQADggEBANep+2tP0c5wGx7JSK6X9sGK
SjddJsOCUy6p2Z/5yb9sDY+nTqrzCl1rRXKVzvdAWssePuLhrYcjCzT1T2EJkbvo
v3cCezCr9erVLg/oIdzn3zIzTBOLwPHAs336Ck6U60iXKn/4qYn+IObXNDnjA1sz
4bu9rqOg/tHxq9NjfwzH6gxJ71rnZl7YqBv2L7My0iYHXLyw0pT1oN2cKKCOCNIs
6YfTqXqJvPOOpzDNJ0hs6K0T7woF0IgCx2aPWDHswZOd1q59cU9ApRBZf9j0Mh5h
4gHMmxW9zOQUtIUlsgRmZfBv3S+iiK8egurPs/OMj8U3P72SFKH7B+V6xbopDq4=
-----END CERTIFICATE-----
```


- **ca.cr**


```
-----BEGIN CERTIFICATE REQUEST-----
MIIC8TCCAdkCAQAwgYExCzAJBgNVBAYTAlBLMQ8wDQYDVQQIDAZQdW5qYWIxDDAK
BgNVBAcMA0lzYjENMAsGA1UECgwERmFzdDENMAsGA1UECwwERmFzcjENMAsGA1UE
AwwERmFzdDEmMCQGCSqGSIb3DQEJARYXZmFpemFuLnBlcnZhekBnbWFpbC5jb20w
ggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDke/6TUbB6PLbdsG60wbG4
IXaqSlrYsdisqTk092Q45vBVUB+oN5pQ7QFSzvxPPjGXzeeIZXZ/jk5NI10zYCyq
rmPcP++p4oCugoaJDPgC5v12h7vc25aDcx+XeQkrrFH74aFioThJhIN9jUl3lgxa
7lxBpfhuLQYoffIwEtUk0Jbu6TAni4mz5bMv2PWKmNjChiKRjqiOEag1Hug5MH7O
YsTq0IfQzTF2DyZ1N6WCiC4VFRgjbhbtybpbVE6nYne/3PY1AlcMXWla9+/IP472
DSHpGW42WKuVP8ZhtpOZJ+WqAGhcmd7XX3d/ynsV7kIO2cr0Gqa74+gqXc1+09Iz
AgMBAAGgKjATBgkqhkiG9w0BCQIxBgwERmFzdDATBgkqhkiG9w0BCQcxBgwEMDkw
MDANBgkqhkiG9w0BAQsFAAOCAQEAsFpvRtVgeGveqW5/d8m08mTzaMW+Ti3Ejsww
ctOEAQERwsRWBRfRYKCLbTqZtCkJWcRxF93qOqTyHkiujecPQzTq3MZv8kRVx0bh
irLeZ29IncbHQfZaAH58MOzsIbUTG127gos92e2mjZxnzAdH2ryWPiZDFVtEGpAZ
TcObO20LBda0g9+rnXwsJhrD9gM91SAa2uvcm9aPwdaAuvvktilfCQ4tpgiK+60Z
3a9hlvBHfG4tcoarlFcgNYccuIcBe6nUMifUw+4W/HII0/Shs+HZJNGl/JgEL3K7
T5Ov+jw5tqQM1IYoomNgoikNb7QxwfBYx7inZo7VTstTk7OCNQ==
-----END CERTIFICATE REQUEST-----
```


- **client_public_key.pem**

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuIEKdIiKSUAh1IFFUvdX
LEVidpVYZPX+QJf4zW4F53VQuREhk4melZeA9CQfxmueF+L1t/MJbvFci837cAHT
6JE+T5jGnl5OS2DuqISJ4l6M4ckKdsgmV57sudI93orE7gTb71iD/8U/SOcSnBiw
wSSbA4UbOD6wvHeDVureJ+J6j28YTAraAZiWqCLsOOjj4uFHU7/PhRxLEaRWOFB5
093X1cbCN2Pm1ZUOKfpicJOGqH0adYIdc7VIYepijgYQcB5WJSOYLyGfC94zWr+Q
1HGLyzxUoi7UGvkUob47T5LlqpbuD0XeUn6bm/ZMWYY4K/hOYbhSzBCoISY3mPox
MQIDAQAB
-----END PUBLIC KEY-----
```

- **server_public_key.pem:**

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqOAOJ+KIzAdghq5539h+
TRq6WbVd2wWk5WWw/rohzAwk465K0aswMuxaMLY0bZ/ZuJRecKNmTwVzK2/3UT5U
0a6bdDdEmu/n9AuxiQUZgeh8N1ih4JzGKetuakDL95RVM6/zKpcSZ57a4jb6/4KW
yOHpWzjXTpznAfMahqsRAvLBCm0Bwu6dydVPyTb0BnhUWdsdLPzpe7Q+L7JnH30s
Sff2UJ9TKILqUZzXCqQr+Jfg+ZygaH9RP57Fc96f975Nnm705x+rGJVBkYarSYhy
PSGyF4t99OwDu7pZlfcjFYETeoJP6scTIZ1wGVnScOkYGngnp1mR4FFxf/GGlHUx
hwIDAQAB
-----END PUBLIC KEY-----
```

6. Documentation:

1. **Symmetric SSL/TLS Communication with AES, SHA-256, and DH - Server and Client Code**
    a. **Server.py**

First, it imports some necessary modules. Set up an SSL/TLS based secure socket server for encryption and decryption.

```
import socket
import ssl
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os
```

This is a function with a DH parameter. It creates a DH parameter set with a 2048-bit key size and returns privkey and pubkey.

```
def generate_dh_parameters():
    parameters = dh.generate_parameters(generator=2, key_size=2048)
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    return private_key, public_key
```

DH key exchange between client and server is performed using this function. The shared key is generated by calculating the client's public key and the server's private key. This is the shared key they will use for encryption.

```
def perform_dh_key_exchange(client_public_key, server_private_key):
    shared_key = server_private_key.exchange(client_public_key)
    return shared_key
```

This function reads DH parameters from the "dhparams.pem" file. These parameters are required for the Diffie-Helman key exchange protocol.

The load_dh_parameters() function retrieves them from a file named "dhparams.pem".

```
def load_dh_params():
    with open("dhparams.pem", "rb") as file:
        dh_params = serialization.load_pem_parameters(file.read())
    return dh_params
```

This is used to encrypt a given message using a shared key. It uses CFB mode with AES encryption and random IV generation. For a block size of 16 bytes, the PKCS7 padding method is used to pad the message. The encrypted message is preceded by an IV.

```python
def encrypt_message(shared_key, message):
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(shared_key), modes.CFB(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    padded_message = message + b' ' * (16 - len(message) % 16)  # PKCS7 padding
    encrypted_message = encryptor.update(padded_message) + encryptor.finalize()
    return iv + encrypted_message
```

This function decrypts the message. It takes the IV and encoded message from the input data and then decodes the message using shared key, AES and CFB modes.

```python
def decrypt_message(shared_key, encrypted_data):
    iv, encrypted_message = encrypted_data[:16], encrypted_data[16:]
    cipher = Cipher(algorithms.AES(shared_key), modes.CFB(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_message = decryptor.update(encrypted_message) + decryptor.finalize()
    return decrypted_message
```

The main() function initializes the server socket at localhost 12345. It also opens ports to listen for connections and accept incoming connections.

```python
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("localhost", 12345))
    server.listen(1)

    connection, address = server.accept()
    print("Connection established with:", address)
```

This code sets the server's SSL context. The statement said that the server certificate and private key were loaded from the files "server.crt" and "server.key".

```python
    context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    context.load_cert_chain(certfile="server.crt", keyfile="server.key")
```

The dh parameters are loaded from "dhparams.pem". The server generates its dh parameters and performs the key exchange algorithm with the client using the Perform_dh_key_exchange function. Compute the shared secret.

```python
    dh_params = load_dh_params()

    private_key, public_key = generate_dh_parameters()

    # Use the server's private key for the key exchange
    shared_key = perform_dh_key_exchange(public_key, private_key)

    ssl_connection = context.wrap_socket(connection, server_side=True)
```

Essentially, once the SSL context overrides it, it becomes a secure SSL/TLS connection.\

```
while True:
    data = ssl_connection.recv(4096)
    if not data:
        break
```

To do this, the server enters a loop to read the data sent by the client through the SSL connector. If there is no data, the loop is interrupted.

```
    decrypted_data = decrypt_message(shared_key, data)
    print("Received:", decrypted_data.decode())
```

The received data is then decrypted using the shared key, printing out the final result.

```
    response = input("Enter a response (type 'exit' to quit): ")
    if response == "exit":
        break
```

Afterwards, the server asks the user for a reply. If "exit", the loop exits.

It then sends the encrypted message back to the client over a Secure Socket Layer or SSL connection.

```
    # Encrypt and send the response
    encrypted_response = encrypt_message(shared_key, response.encode())
    ssl_connection.send(encrypted_response)
```

When the loop ends, the session will stop and the ssl connection and server socket will be closed.

```
ssl_connection.close()
```

b. Client.py

Similar to the server script, the client script begins by importing the required modules.

```
import socket
import ssl
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os
```

The client script performs a similar role by generating DH parameters, as well as private key and public key for the client.

```
def generate_dh_parameters():
    parameters = dh.generate_parameters(generator=2, key_size=2048)
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    return private_key, public_key
```

In addition, the client-side DH key exchange function can calculate the shared key by making use of the server's public key and a client private key.

```
def perform_dh_key_exchange(client_public_key, server_private_key):
    shared_key = server_private_key.exchange(client_public_key)
    return shared_key
```

This function fetches DH parameters from a "dhparams.pem" file.

```
def load_dh_params():
    with open("dhparams.pem", "rb") as file:
        dh_params = serialization.load_pem_parameters(file.read())
    return dh_params
```

Client's encrypt function creates an initial vector (IV), performs aes encryption in cfb mode and applies pkcs7 padding onto the message. An additional IV is inserted into the start of the encrypted text.

```
def encrypt_message(shared_key, message):
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(shared_key), modes.CFB(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    padded_message = message + b' ' * (16 - len(message) % 16)  # PKCS7 padding
    encrypted_message = encryptor.update(padded_message) + encryptor.finalize()
    return iv + encrypted_message
```

Just like decrypt function on a server, that of the client has the same principles.

```
def decrypt_message(shared_key, encrypted_data):
    iv, encrypted_message = encrypted_data[:16], encrypted_data[16:]
    cipher = Cipher(algorithms.AES(shared_key), modes.CFB(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_message = decryptor.update(encrypted_message) + decryptor.finalize()
    return decrypted_message
```

The client script's main() function uses sockets to establish a connection with the server running on localhost / port 12345.

```
    context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
    context.load_verify_locations(cafile="server.crt")
```

For server authentication, an SSL context is created for the client. It states that "ca.crt" has been loaded with the trusted CA certificate.

```
def main():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(("localhost", 12345))
```

DH parameters of the client are loaded from "dhparams.pem". It then generates the DH parameters it possesses.

```
    private_key, public_key = generate_dh_parameters()
```

During DH key exchange, the client employs its private key. The shared key is calculated. Wrapped in an SSL/TLS, encapsulate the CLIENT socket using the provided content. Server verication is done using the server hostname "localhost".

```
    while True:
        message = input("Enter a message (type 'exit' to quit): ")
        if message == 'exit':
            ssl_connection.send(message.encode())
            break
```

The client sends messages through a loop. The prompt the input if the input is "exit" the loop break.
The message if encrypted and then sent over to the server.
Over the SSL connection, the server sends data to the client. The loop gets broken if there isn't any data.
Decrypted information is displayed and printed.

```
    # Encrypt and send the message
    encrypted_message = encrypt_message(shared_key, message.encode())
    ssl_connection.send(encrypted_message)
```

```python
while True:
    message = input("Enter a message (type 'exit' to quit): ")
    if message == 'exit':
        ssl_connection.send(message.encode())
        break

    server_public_key = serialization.load_pem_public_key(ssl_connection.getpeercert(binary_form=True))
    shared_key = private_key.exchange(server_public_key)

    # Encrypt and send the message
    encrypted_message = encrypt_message(shared_key, message.encode())
    ssl_connection.send(encrypted_message)

    response = ssl_connection.recv(4096)

    shared_key = private_key.exchange(server_public_key)
    decrypted_response = decrypt_message(shared_key, response)
    print("Received:", decrypted_response.decode())
```

Upon completion of the loop, the SSL connections and the client socket will be terminated.

```python
ssl_connection.close()
```

2. Asymmetric SSL/TLS Communication with RSA, SHA-256, and PKI - Server and Client Code

a. Server.py

This section contains all required imports required for socket communication, SSL/TLS encryption, RSA and X25519 key exchange, and other cryptographic functions.

```python
import socket
import ssl
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import x25519
from cryptography.hazmat.primitives.asymmetric.x25519 import X25519PrivateKey
from cryptography.hazmat.primitives.asymmetric.x25519 import X25519PublicKey
```

This function generates an RSA private key. This algorithm returns a public key with exponent 65537 and a private key 2048 bits long. It reads a byte array of x25519 public key.

This operation converts the x25519 public key into bytes and sends it to the client.

In this section, you create a server socket and bind it to localhost at port 12345. It starts listening for at most one incoming connection, with a backlog of 1 connection. After receiving an incoming connection from the client, the server acknowledges receipt by printing its address.

```python
def Gen_RSA():
    p_key = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())
    return p_key

def load_x25519_pubkey(data):
    return X25519PublicKey.from_public_bytes(data)

def exchange_x25519_pubkey(client_socket, pub_key):
    pub_key_bytes = pub_key.public_bytes(Encoding.Raw, PublicFormat.Raw)
    client_socket.send(pub_key_bytes)
```

In this section, the client authenticates to the SSL/TLS context. SSL/TSL communicates using certificates and private keys loaded from "server.crt" and "server.crt". key".

Server authentication and secure key exchange are accomplished by generating an RSA private key and an x25519 key pair. The main communication loop starts here. If the client does not send anything to the server over the SSL connection, the loop will break.

At this stage, the server uses its x25519 key to exchange keys with the client's public key. The data is encrypted, then transmitted and stored in a buffer for later decoding.

The server reads the message, decrypts it using its own RSA private key, and prints it to the console.

```python
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("localhost", 12345))
    server.listen(1)

    Connection, addr = server.accept()
    print("Connection being established with: ", addr)

    context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    context.load_cert_chain(certfile="server.crt", keyfile="server.key")

    client_pb_key = Gen_RSA()
    pr_key = Gen_RSA()
    pr_key_x25519 = X25519PrivateKey.generate()
    pr_key_pub_x25519 = pr_key_x25519.public_key()

    ssl_connection = context.wrap_socket(Connection, server_side=True)

    while True:
        data = ssl_connection.recv(4096)
        if not data:
            break

        # Key exchange using x25519
        x25519_pub_key = load_x25519_pubkey(data)
        shared_key = pr_key_x25519.exchange(x25519_pub_key)

        de_data = b''
        while True:
            data = ssl_connection.recv(4096)
            if not data:
                break
            de_data += data

        if not de_data:
            break
```

The user enters a response to the server. If the answer is "exit", the loop will stop. It either returns "Yes" or "No" depending on whether the condition is met.

The server responds by encrypting the message using the shared x25519 key and sending it back to the client.

```
    # Decrypt the message
    cipher = serialization.load_pem_private_key(de_data, password=None, backend=default_backend())
    decrypted_message = cipher.decrypt(de_data, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
    print("Received: ", decrypted_message.decode())

    res = input("Reply to client (type 'exit' to exit): ")

    if res == 'exit':
        ssl_connection.send(res.encode())
        break

    # Encrypt the response message
    x25519_pub_key = pr_key_pub_x25519.public_bytes(Encoding.Raw, PublicFormat.Raw)
    pub_key = X25519PublicKey.from_public_bytes(x25519_pub_key)
    shared_key = pr_key_x25519.exchange(pub_key)
    ciphertext = shared_key.encrypt(res.encode(), padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
    ssl_connection.send(ciphertext)

ssl_connection.close()
```

b. Client.py

Client.py also has a similar structure, so we will discuss only some of the changes and the client-specific parts in detail.

```
context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
context.load_verify_locations(cafile="server.crt")
ssl_connection = context.wrap_socket(client, server_hostname="localhost")
```

For server authentication, the client generates its SSL/TLS context. This context loads the trusted CA certificate from the "server.crt" file and then wraps the socket to establish an SSL/TLS connection with the server.

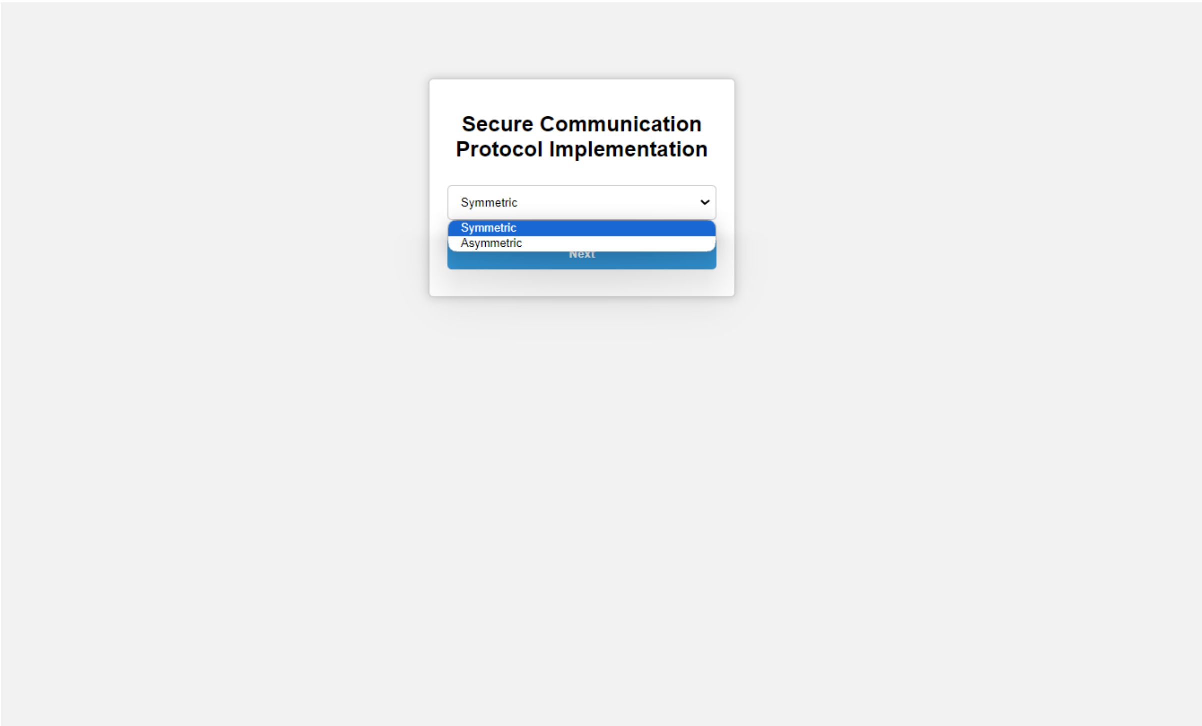
```
pr_key = Gen_RSA()
pr_key_x25519, pub_key_x25519 = Gen_x25519_key()
exchange_x25519_pubkey(ssl_connection, pub_key_x25519)
```

In this case, the client generates its RSA private key x25519 key pair, which is sent to the server to initiate key exchange.

The main communication loop is similar to the server, with the following tasks:

- Message prompts and "exit" command handlers.
- Send the encrypted message to the server in the form of x25519.
- Accepts the server's response, decrypts it, and prints it to the console.
- After the loop ends, the SSL connection is severed due to client termination.

13.User Interface



14.Security Points to Remember

1. Vulnerabilities in Security

a. Lack of Detailed PKI Implementation:

Although PKI is mentioned in the server code, it is not fully functioning. PKI is necessary to manage certificates securely and guarantee communication authenticity.

b. Key Size and Algorithm Selection:

Although the code makes use of respectable key sizes (2048 bits for Diffie-Hellman and RSA), it is imperative to take future-proofing into account and modify key sizes in accordance with industry norms. It's also critical to assess the application of algorithms for any potential vulnerabilities.

c. Lack of Error Handling:

In the event of an exception or problems during execution, the code may not handle errors sufficiently, opening the door to possible security risks.

d. Limited Input Validation:

If malicious data is entered, the code's lack of comprehensive input validation may result in security flaws.

2. Reductions in Threats

a. Detailed PKI Implementation:

To increase the authenticity of communications, implement a complete PKI system to handle certificates, signatures, and verifications.

b. Regular updates:

Stay up to date on security guidelines and modify key sizes and algorithms as needed.

c. Powerful Error Handling:

Use error handling to handle errors and problems in a friendly way while avoiding possible security threats.

d. Input Validation:

To reduce the risks associated with malicious input data, apply comprehensive input validation.

15. Testing:

a) Test Cases

The security and correctness of the implementation have been verified using a comprehensive test set. Test cases contain different use cases designed to check the reliability of the code.

b) Variable Encryption Strength and Key Length

The protocol's flexibility and capacity to manage a range of security requirements are tested using varying encryption strengths and key lengths.

16.Conclusion:

Finally, it successfully demonstrates the simplest yet powerful secure channel between server and client via SSL/TLS handshake, Diffie-Hellman key exchange, and AES data encryption. This maintains data integrity and confidentiality as it ensures messages are only delivered through secure channels. To further enhance the code, it is recommended to consider the following points:

- Handle errors, exceptions and logging.
- Enhance scalability by adding more features, such as handling multiple clients simultaneously.
- Make sure to implement proper authentication and authorization on the server and client so that both components have true identities to each other.
- Stress-test the code under different scenarios to confirm its robustness and efficiency.

- Its code is a good foundation for reliable communication, and one can extend and personalize it accordingly to meet other needs.

17.References

https://www.youtube.com/watch?v=ZkL10eoG1PY
https://crypto.stackexchange.com/questions/9509/aes-encryption-using-a-diffie-hellman-key-exchange
https://www.quickprogrammingtips.com/python/aes-256-encryption-and-decryption-in-python.html
https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2