COE817 – Lab 3

Mai Abdelhameed – 500769087

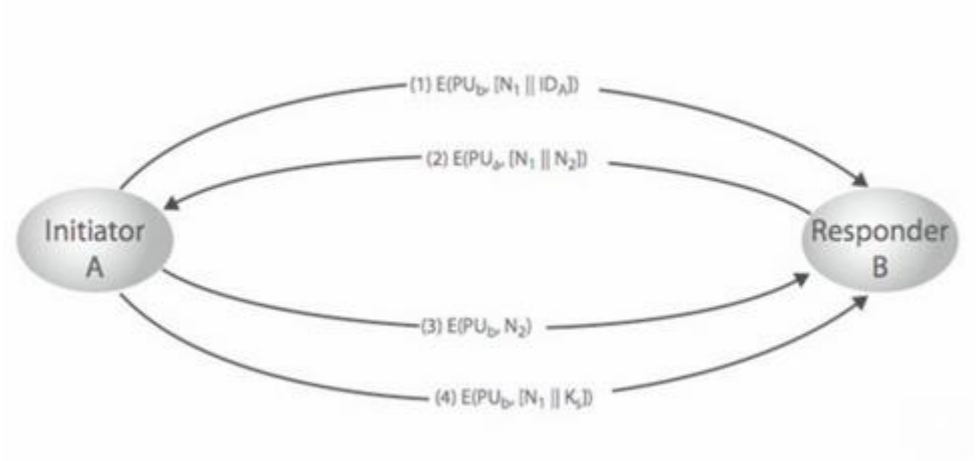

Figure 1: Client/Server Authentication Process Diagram

The application consists of a client and a server that establishes a chat session and uses a session key to send encrypted chat texts and images to each other.

The chat application flow is as follows:

1. The server is started and listens for incoming client connections.
2. The client is started and connects to the server.
3. The server and client establish a secure channel via the authentication process as shown in Figure 1. The UI output is displayed in Figure 2.
4. Once the channel is established, chatting commences.
    a. Text chat messages are shown in Figure 3.
    b. Image chat messages are shown in Figure 4.

The lab manual was not clear about a few things, so I made my assumptions:

- The process of establishing a secure channel should happen behind the scenes and not produce an output as shown in Figure 2. I used print statements to display the process for marking purposes only.
- The public keys were announced via public announcement from client to server and server to client.
- The manual never specified the number of members in the chat, so I assumed one to one only and do not have a group chat capability.
- The session key plaintext is hardcoded for demonstration purposes in this lab only. For proper implementation, it should be a randomly generation session key that is unique to each session.
- The lab manual never specified the method of image transferring so I implemented a working model with the following restrictions:

    - Image must be 100x100. I picked this size for faster image download times.
    - All images must be placed in a local repository directory.
    - Only the most recent image received from the end user will be saved, as all subsequent images will rewrite the locally stored downloaded image.
- Image sending works as follows, which is shown in Figure 4:
    - The server has a local repository of stored images, in the `serverImageRepo` directory, the client has a similar repository named `clientImageRepo`.
    - When the user wants to send an image, they enter `i` into the chat window that the application reads as a command and will list the images currently available in their respective repositories.
    - The user selects an image, and the image gets encrypted and sent over the secure TCP channel to the end user.
    - The end user gets the image downloaded into their repository for viewing.

```
codespace ➜~/workspace/COE817-labs/lab3 (main X) $ python3 server.py
Accepted client connection
Establishing secure channel...

Received client public key through public announcement
Sending server public key through public announcement
Received encrypted client ID as:
b'\x087Z\x19\x81\xf79lF3\x1eV\xd8\x04\xcb0'
Decrypted client ID to:
b'client ID'
Received encrypted nonce as:
b'&\n\xc5`\xa4*\xffp'
Decrypted nonce to:
b'N1'
Generating and encrypting nonce 2
Sending the encrypted nonces to the client
Received encrypted message:
b'\xe7M\x81+\xdf\x83b\x7f'
Decrypted message to:
b'N2'
Received encrypted message:
b'&\n\xc5`\xa4*\xffp'
Decrypted message to:
b'N1'
Received encrypted message from client:
b'\xc9y\xb7\x8b\x1f\x9c\xd3\x86U\x12\xa2\x84\xf6\x85\xa4\xb7'
Decrypted message into:
b'sessionk'

Established secure channel, start chatting!
```

Figure 2a: Server UI Output - Authentication

```
codespace ➜~/workspace/COE817-labs/lab3 (main X) $ python3 client.py
Established connection to the server
Establishing secure channel...

Sending public key to server
Waiting for the server public key
Received the server public key
Generating the client ID and nonce 1
Sent client ID and nonce 1, encrypted with the server public key
Received encrypted messages:
b'l\xc1\xb6\x8e\x03Jl\x7f'
b'2\xe0\x0bv\x91\xe2\x95\x93'
Decrypted messages into:
b'N1'
b'N2'

Established secure channel, start chatting!
```

Figure 2b: Client UI Output - Authentication

```
Established secure channel, start chatting!
(Type 'i' to send an image from your image repository 'serverImageRepo'
(Type 'q' at any time to quit)
Client Encrypted: b'L\x9c\xe7\x8du\xd5B,'
Client Decrypted: Hello!
Hey! What's up?
```

Figure 3a: Server UI Output – Text Chat Messages

```
Established secure channel, start chatting!
(Type 'i' to send an image from your image repository 'clientImageRepo'
(Type 'q' at any time to quit)
Hello!
Server Encrypted: b'\xfe\x82\x1a\x80@X\xd93\xfbP!\x0f\xb2\x0f\rL'
Server Decrypted: Hey! What's up?
```

Figure 3b: Client UI Output – Text Chat Messages

```
Client Encrypted: b'\xaa\xfd\xb5\x8dj,Q\xac$\xa7\xaf]\xf8\x00\xb55\xd8\xfd\xfdeo\xa4`
E\x9f\xb0\xf0Q\xad\t}\x86'
Client Decrypted: Wanna see a picture of a dog?
Sure!
Client wants to send an image
Downloading image...
Incoming image size in bytes is: 30008
Image data not displayed due to it's length
Decrypting image...
Download complete! View image at ./serverImageRepo/download.jpg
```

Figure 4a: Server UI Output – Image Chat Messages

```
Wanna see a picture of a dog?
Server Encrypted: b'\x1f\xd4$\xd4H\xec\xacU'
Server Decrypted: Sure!
i
Your image repo contains the following images:
0: cat.jpg
1: dog.jpg
Select an image by typing the associated number: 1
Sending dog.jpg to server...
Image sent successfully!
```

Figure 4b: Client UI Output – Image Chat Messages

Handling Replay Attacks

The server can only have one client connected at a time, and once a client is connected, it must follow the steps as shown in Figure 1. Using nonces, it can detect when/if a client is trying to send spam messages, and when detected, the server will sever the connection, preparing for another client connection. These two features allow for a robust defence against replay attacks.