Secure Communication

Yohannes Dawit Kassaye

yd.kassaye@stud.uis.no

249238

Abstract

In this paper we will go through the tasks and results of the second assignment in DAT510 - Security and Vulnerability in Networks. In this assignment we were introduced to the world of secure communication, trough the Public-key cryptography paradigm, together with a cryptographically-secure pseudo random number generator (CSPRNG) and the use symmetric ciphers to encrypt, decrypt and send messages.

1 Introduction

In the world of cryptography and security we have a model known as the CIA-triad. This model is used to describe the three main goals of security, which are Confidentiality, Integrity and Availability. Confidentiality is the principle that says that only authorized parties should be able to access information sent over public channels. Integrity on the other hand is the principle that says that the information sent over a network should not be altered in any way. This assignment has these two goals, namely ensuring confidentiality and integrity of the messages sent over the network. As mentioned this was done using the public-key cryptography paradigm, together with a CSPRNG and symmetric ciphers. The public-key cryptography paradigm is a paradigm that uses two keys, a public key and a private key. Now you might wonder why we need two keys, and the answer is that anyone with a public key can encrypt a message, but only the owner of the private key can decrypt the message. This is because the private key is used to decrypt the message, and the public key is used to encrypt the message. The public key is public, and can be shared with anyone, while the private key is kept secret. This ensures the CIA-triad goals, as only the intended recipient can decrypt the message, and the message is not altered in any way without it being detected. The CSPRNG is used to generate a random key that is used to encrypt the message, and the symmetric cipher is used to encrypt the message using the random key.

2 Design and Implementation

The assignment was split into two parts, the first part was to implement a key exchange protocol - Diffie Hellman - and combine this with a cryptographically secure pseudo random number generator (CSPRNG) to generate a even stronger encryption key before encrypting the message. In the second part of the assignment we were asked to implement a real-world scenario using secure communication. More specifically we were asked to implement a web server and a client that would communicate over a secure channel by using the different methods we implemented in the first part.

2.1 Part I

As mentioned the first part of the assignment was to implement a key exchange protocol, more specifically Diffie Hellman. Diffie Hellman is a key exchange protocol for exchanging cryptographic keys over a public channel. This is done by having two parties, Alice and Bob, that want to exchange a key. The key exchange protocol works by having Alice and Bob agree on a prime number, a base and a secret number. The secret number is a random number that is kept secret. The secret number is then used to calculate the public key. The public key is then sent over the public channel, and the other party can use the public key to calculate the shared key. The shared key is then used to encrypt the message. The shared key is calculated by using the secret number and the public key.

We were also asked to implement a CSPRNG, which is a cryptographically secure pseudo random number generator. This is a generator that generates random numbers that are not predictable. This is done by using a seed, which is a number that is used to generate the random numbers. There are a lot of different ways to generate a CSPRNG, but the one used in this assignment was the Blum Blum Shub algorithm. This algorithm uses a seed, and then generates a random number by squaring the seed and then taking the remainder of the square divided by a prime number. This is then used as the new seed, and the process is repeated. This is done until a random number is generated.

Lastly we had to encrypt the message using a symmetric cipher. This was done by using the shared key, from the previous two steps, and encrypt the message. The symmetric cipher used in this assignment was a custom cipher from the first assignment in the course called KES. This was a product cipher, which means that the encryption and decryption is done by multiple ciphers, namely the Caesar cipher and the Vigenere Cipher.

These three steps were done by using three classes, namely DH, KES and SecureCommunication.

The first two classes had the responsibility of dealing with the key exchange protocol, generating CSPRNG - and implementing the encryption and decryption process, respectively. While the SecureCommunication class had the responsibility of keeping track of a single users state, and combining the other two classes to ensure secure communication.

```python
class SecureCommunication:
    def __init__(self, user: Person, dh: DH, kes: KES):
        self.user = user
        self.dh = dh
        self.kes = kes

    def generate_shared_keys(self):
        shared_key = self.dh.generate_shared_key(self.user.private_key, self.user.
    counter_part_public_key, self.user)
        self.user.shared_key = self.dh.BBS(shared_key, 8, self.user)
        self.dh.log(f"Shared key for {self.user.name}: {self.user.shared_key}")

    def store_counterpart_public_key(self, counter_part_public_key):
        self.user.counter_part_public_key = counter_part_public_key

    def Encrypt(self, message):
        ceasar_key = self.dh.generate_psuedo_random_ceasar_key()
        return self.kes.KES_cipher(message, ceasar_key, self.user.shared_key)

    def Decrypt(self, message):
        ceasar_key = self.dh.generate_psuedo_random_ceasar_key()
        return self.kes.KES_cipher(message, ceasar_key, self.user.shared_key, mode="
    decrypt")

    # factory method to create a Communication object
    @classmethod
    def createCommunication(cls, user: Person):
        return cls(user, DH.createDH(30000000091, 40000000003), KES.createKES())
```

In essence the SecureCommunication class ensures the CIA - principles. Once the class is initiated with a user as a parameter, we will generate public and private keys through the DH class, and store them on the user. Now say we have two user, Alice and Bob, the next step would be to share their public keys with each other. Once the public keys are shared, we can generate the shared key through the DH class and the BBS - CSPRNG, and use that key to encrypt or decrypt the message. This is a great way of implementing a secure communication, as we will see later, given that the class will live on the client side of a exchange.

2.2 Part II

As mentioned earlier, in the second part of the assignment we were asked to implement a real-world scenario using secure communication. More specifically we were asked to implement a web server and a client that would communicate over a secure channel by using the different methods we implemented in the first part. The web server was implemented using Flask and Flask-SocketIO. Flask is a web framework for Python, and Flask-SocketIO is a Flask extension that adds support for Web-Socket communication. The client was implemented using python-socketio, and was a simple console application.

The web server consisted of 6 endpoints, each with a specific purpose and responsibility. The first endpoint was the join endpoint, which in was used to store a mapping between the user and its session id. This was done by using a dictionary, where the key was the user, and the value was the session id. The dictionary was used to send messages to a specific user.

The second endpoint was the getOnlineUsers endpoint, which was used to get a list of all the users that were currently online. Once the client were able to see a list of user that were online, they could choose a user to communicate with. In essence this endpoint was used to share the public keys between the users. Without this endpoint the message that were sent from one client to another, would be unreadable. This as mentioned earlier ensures confidentiality.

Lastly once the users had chosen a user to communicate with, they could send messages to each other, and this was done by using the sendmessage endpoint.

Now in the case of the client, this was implemented as a console application were the user were prompted with multiple options and was able to chat with other users. The user was prompted with a total of 4 options, namely show Online Users, Start Chat Session With User, Send Message and Exit.

Once the user had seen a list of online users, then could choose to start a chat session with a user. This would the prompt the user to enter the name of the user they wanted to chat with. This in essence was used as we user-friendly was of sharing the public-keys between the users. Now that the user had the public key of the user they wanted to chat with, they could send messages to each other. This was done through the sendmessage endpoint, where the server would send the message to the correct session, given the nature of the web-sockets and its bidirectional capabilities.

The entire process is shown in the following diagram:

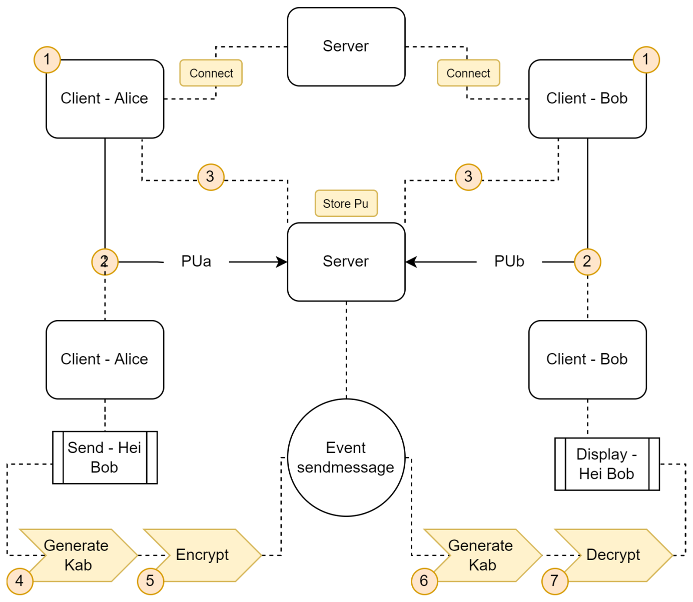


Figure 1: Secure Communication Flow

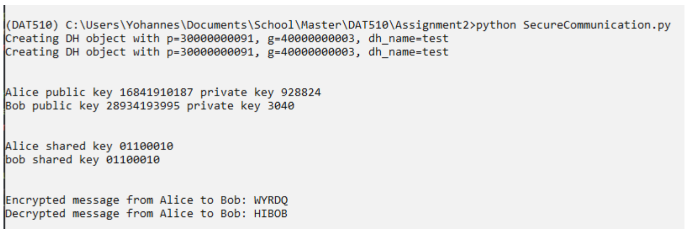
3 Test Results

Now that we have an understanding on how the program works, we will turn our attention to the test results. We will go through some of the outputs and results from each of the parts of the assignment.

3.1 Part I

```python
def test_communication(comm1: SecureCommunication, comm2: SecureCommunication):
    # Alice and Bob generates a private key and a public key
    _,_ = comm1.dh.generate_keys(comm1.user), comm1.dh.generate_keys(comm2.user)

    print("\n")
    print("Alice", "public key", comm1.user.public_key, "private key", comm1.user.
    private_key)
    print("Bob", "public key", comm2.user.public_key, "private key",comm2.user.
    private_key)
    print("\n")

    # Alice and Bob share their public keys
    comm1.store_counterpart_public_key(comm2.user.public_key)
    comm2.store_counterpart_public_key(comm1.user.public_key)

    # Alice and Bob generate their shared key
    comm1.generate_shared_keys()
    comm2.generate_shared_keys()

    print("Alice", "shared key", comm1.user.shared_key)
    print("bob", "shared key", comm2.user.shared_key)
    print("\n")

    # Alice encrypts the message with the shared key, and sends it to Bob
    # Bob decrypts the message with the shared key, and reads the message
    alice_message = "Hi bob"
    alice_encrypted_message = comm1.Encrypt(alice_message)
    bob_decrypted_message = comm2.Decrypt(alice_encrypted_message)
    print("Encrypted message from Alice to Bob:", alice_encrypted_message)
    print("Decrypted message from Alice to Bob:", bob_decrypted_message)
```

The code above is a simple test in which we create two users, Alice and Bob, and we test the secure communication between the two users. We firstly generate the public and private keys for both users, and then we share the public keys with each other. Once the public keys are shared, we can generate the shared key, and use that key to encrypt and decrypt messages. As we can see in the output, the message is encrypted and decrypted.

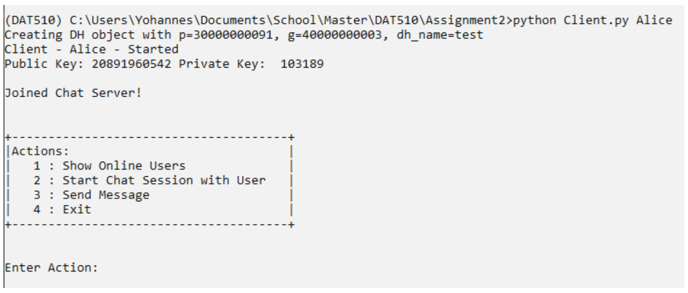


```
(DAT510) C:\Users\Yohannes\Documents\School\Master\DAT510\Assignment2>python SecureCommunication.py
Creating DH object with p=30000000091, g=40000000003, dh_name=test
Creating DH object with p=30000000091, g=40000000003, dh_name=test


Alice public key 16841910187 private key 928824
Bob public key 28934193995 private key 3040


Alice shared key 01100010
bob shared key 01100010


Encrypted message from Alice to Bob: WYRDQ
Decrypted message from Alice to Bob: HIBOB
```

Figure 2: Secure Communication test

3.2 Part II

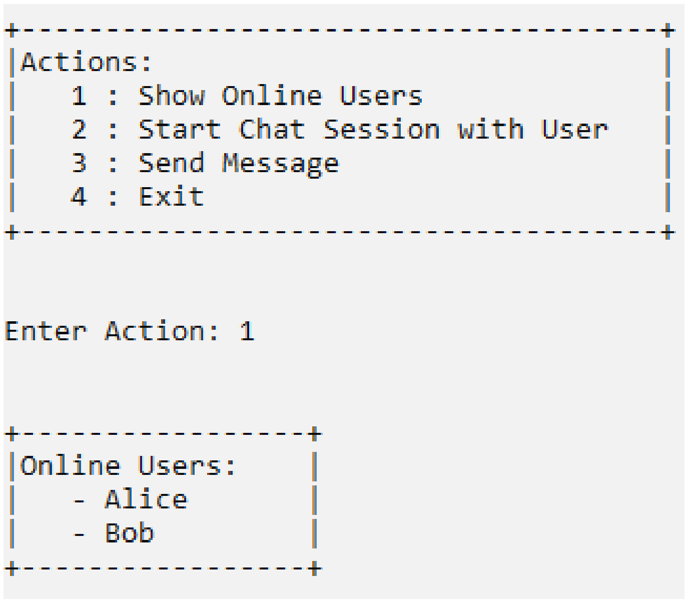
Now for the second part of the assignment, we will look at the test results from the communication between the clients and the server. So when testing the program we had one server running and two clients, one for Alice and one for Bob. The first thing that happens when the client spins up, is that the client will connect the server, and generate a public and private key.



```
(DAT510) C:\Users\Yohannes\Documents\School\Master\DAT510\Assignment2>python Client.py Alice
Creating DH object with p=30000000091, g=40000000003, dh_name=test
Client - Alice - Started
Public Key: 20891960542 Private Key:  103189

Joined Chat Server!


+-------------------------------------+
|Actions:                             |
|    1 : Show Online Users            |
|    2 : Start Chat Session with User |
|    3 : Send Message                 |
|    4 : Exit                         |
+-------------------------------------+


Enter Action:
```

Figure 3: Client Options Prompt

Once that is done, the client will send a request to see a list of online users, and the server will respond with a list of online users.

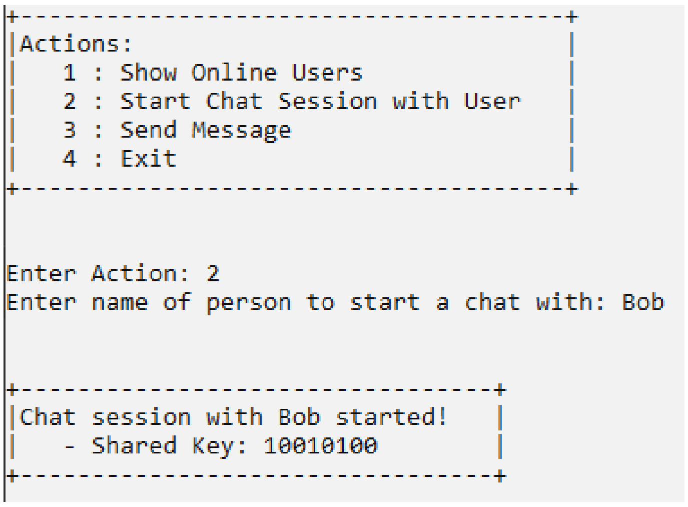


```
+-------------------------------------+
|Actions:                             |
|    1 : Show Online Users            |
|    2 : Start Chat Session with User |
|    3 : Send Message                 |
|    4 : Exit                         |
+-------------------------------------+


Enter Action: 1


+-----------------+
|Online Users:    |
|    - Alice      |
|    - Bob        |
+-----------------+
```

Figure 4: Online Users

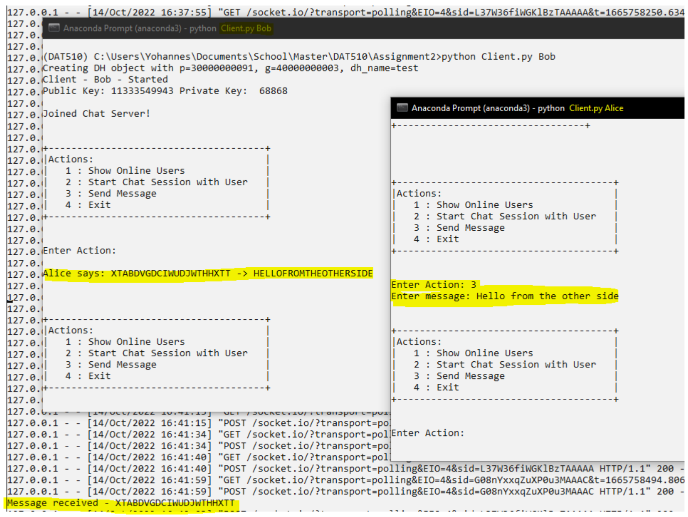
Then the user will choose a user to chat with, and the server will respond with the public key of the user they want to chat with, and the shared key is generated.

Now that the shared key is generated, the user can send messages to the other user, and the server will send the message to the correct session. This is know end-to-end encrypted, as the message only exists in a readable format on the client sides.



```
+---------------------------------------+
|Actions:                               |
|   1 : Show Online Users               |
|   2 : Start Chat Session with User    |
|   3 : Send Message                    |
|   4 : Exit                            |
+---------------------------------------+


Enter Action: 2
Enter name of person to start a chat with: Bob


+--------------------------------+
|Chat session with Bob started!  |
|   - Shared Key: 10010100       |
+--------------------------------+
```

Figure 5: Started Chat Session



```
Anaconda Prompt (anaconda3) - python Client.py Bob

(DAT510) C:\Users\Yohannes\Documents\School\Master\DAT510\Assignment2>python Client.py Bob
Creating DH object with p=30000000091, g=40000000003, dh_name=test
Client - Bob - Started
Public Key: 11333549943 Private Key:  68868

Joined Chat Server!


+---------------------------------------+
|Actions:                               |
|   1 : Show Online Users               |
|   2 : Start Chat Session with User    |
|   3 : Send Message                    |
|   4 : Exit                            |
+---------------------------------------+


Enter Action:

Alice says: XTABDVGDCIWUDJWTHHXTT -> HELLOFROMTHEOTHERSIDE



+---------------------------------------+
|Actions:                               |
|   1 : Show Online Users               |
|   2 : Start Chat Session with User    |
|   3 : Send Message                    |
|   4 : Exit                            |
+---------------------------------------+
```

```
Anaconda Prompt (anaconda3) - python Client.py Alice
+--------------------------------+



+---------------------------------------+
|Actions:                               |
|   1 : Show Online Users               |
|   2 : Start Chat Session with User    |
|   3 : Send Message                    |
|   4 : Exit                            |
+---------------------------------------+


Enter Action: 3
Enter message: Hello from the other side


+---------------------------------------+
|Actions:                               |
|   1 : Show Online Users               |
|   2 : Start Chat Session with User    |
|   3 : Send Message                    |
|   4 : Exit                            |
+---------------------------------------+


Enter Action:
```

```
127.0.0.1 - - [14/Oct/2022 16:37:55] "GET /socket.io/?transport=polling&EIO=4&sid=L37W36fiWGKlBzTAAAAA&t=1665758250.634
127.0.0.1 - - [14/Oct/2022 16:41:15] "POST /socket.io/?transport=po
127.0.0.1 - - [14/Oct/2022 16:41:34] "GET /socket.io/?transport=pol
127.0.0.1 - - [14/Oct/2022 16:41:34] "POST /socket.io/?transport=po
127.0.0.1 - - [14/Oct/2022 16:41:40] "GET /socket.io/?transport=pol
127.0.0.1 - - [14/Oct/2022 16:41:40] "POST /socket.io/?transport=polling&EIO=4&sid=L37W36fiWGKlBzTAAAAA HTTP/1.1" 200 -
127.0.0.1 - - [14/Oct/2022 16:41:59] "GET /socket.io/?transport=polling&EIO=4&sid=G08nYxxqZuXP0u3MAAAC&t=1665758494.806
127.0.0.1 - - [14/Oct/2022 16:41:59] "POST /socket.io/?transport=polling&EIO=4&sid=G08nYxxqZuXP0u3MAAAC HTTP/1.1" 200 -
Message received - XTABDVGDCIWUDJWTHHXTT
```

Figure 6: Message Flow

4 Discussion

This assignment was a great and practical introduction to secure communication using public-key cryptography. When looking at the code implementation, good coding practises such as SOLID was used.

This gave me a lot of freedom when implementing the second part of the assignment. The SecureCommunication class is the heart of the program, and it being a combination of key exchange protocols, encryption and decryption algorithm, and a state machine for the users data, makes it very easy to create multiple users and clients. This means that the application is not limited to just

Alice and Bob, but can be used for any number of users.

5 Conclusion

In conclusion, this assignment was a great introduction to secure communication, and I learned a lot about public-key cryptography, and how it can be used to create a secure communication between two parties.

In addition to that i learned a lot about web sockets, and how to develop an event driven application using Flask and Flask-SocketIO. I would have liked to make it possible for a user to chat with multiple user at the same time, and again given the nature of the SecureCommunication class, this would have been relatively easy to implement.

I would also have liked to implement a group chat feature, where a user could create a group, and invite other users to join the group. Besides that the last improvement i would have liked to make, is a better user interface. The current user interface is very basic, given that it is a console application, and i would have liked to make a more user friendly interface, such as a web application.

References

[1] A cryptographically secure random number generator. URL `https://www.johndcook.com/blog/2017/09/21/a-cryptographically-secure-random-number-generator/`.

[2] Flask-socketio. URL `https://flask-socketio.readthedocs.io/en/latest/`.

[3] Public-key cryptography. URL `https://en.wikipedia.org/wiki/Public-key_cryptographyl`.

[4] Python-socketio. URL `https://python-socketio.readthedocs.io/en/latest/`.

[5] Understand diffie-hellman key exchange. URL `https://www.infoworld.com/article/3647751/understand-diffie-hellman-key-exchange.html`.