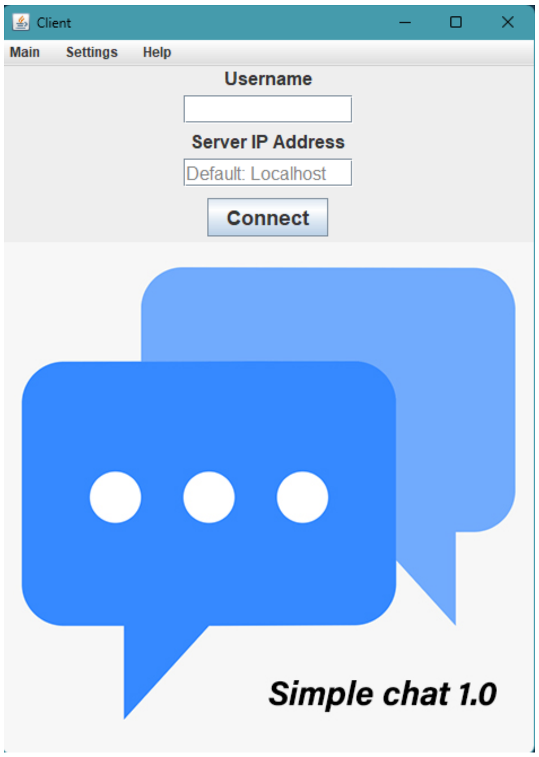
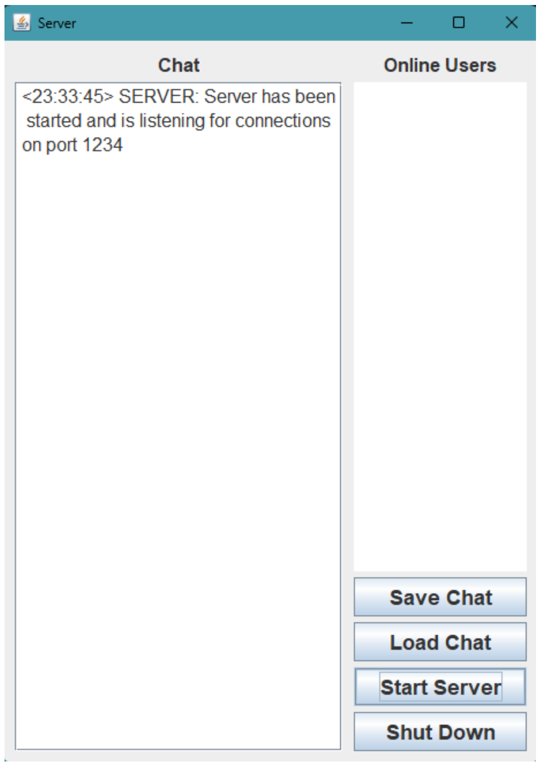
Encrypted Java Client-Server Chat Application

Mirco Ghadri & August Holm

June 6, 2024





```
<23:33:45> SERVER: Server has been started and is listening for connections on port 1234
```

Abstract

This project report describes the design and implementation of encryption into an existing Java client-server chat application. The project consisted of designing the encryption solution, implementing it in the existing codebase using Java Cryptographic libraries and APIs and finally testing the solution using Wireshark and Java print statements.

Contents

1 Introduction

Today most communication is digital and in digital communication there are a lot of security concerns. When communicating online you want to be sure your messages don't get intercepted and can be read by an unauthorized party.[2]

1.1 Background

In today's digital age, where more and more people have become connected to the internet, ensuring the security and privacy of communications has become paramount. With the increasing prevalence of cyber threats and privacy concerns, there is a growing demand for secure messaging solutions that offer encryption. Without encryption your private messages can be accesed by third parties so encrypting messages is important to keep your personal data and your private conversations confidential [2].

1.2 Purpose

This report aims to go through the implementation and evaluation of a chat application with encrypted messages to ensure the users privacy.

1.3 Goal

The goal of this project is to develop an encrypted Java Chat application that provides users with a secure platform for communication. By leveraging cryptographic APIs available in Java, we aim to implement robust encryption mechanisms to safeguard the confidentiality of messages exchanged between users.

1.4 Limitations

The project will focus on encrypting messages between the client and server. However the messages will not be end-to-end encrypted. This means that the server can also read all messages between the clients.

2 Theoretical background

Symmetric key uses the same key to encrypt and decrypt messages, so that anyone with the key can both encrypt and decrypt messages. Asymmetric key was created to solve the issue of sharing a key which you need to do with a symmetric key. It uses a public key to encrypt and a private key to decode so that anyone can encrypt messages with the public key but only the intended recipient with the private key can decode and read the message[1].

2.1 AES encryption

AES or Advanced Encryption System is a symmetric encryption algorithm that was made to replace an older algorithm called DES. They devoleped this to improve on DES main weakness which was the short encrytion key length making AES less vulnerable to brute force.

2.2 RSA encryption

RSA or Rivest–Shamir–Adleman is a common assymetric key algorithm named after the engineers that developed it. This algorithm is slower compared to AES.

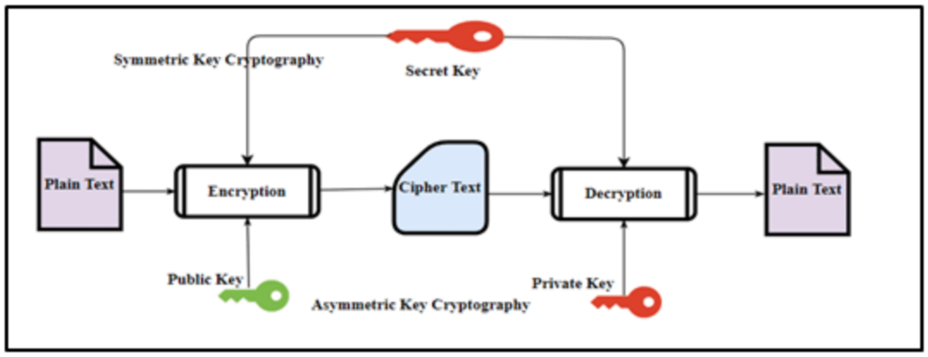


Figure 1: Symmetric and asymmetric key encryption.

3 Method

Given the Java chat app we Implemented encryption for the app by encrypting each message sent from clients to servers.First the client starts by generating a public and private RSA key pair and the server creates a symmetric AES key or our session key. Then the client starts by sending the public key to the server, then the server sends back the symmetric AES key encrypted with the public key so that only the client who sent the public key can decrypt it. Now each message sent by the clients is safe from third parties intercepting and reading it since it is sent using the symmetric session key for encryption and decryption.

3.1 Design

Before we wrote any code in Java, we carefully designed our solution. We wanted to use a symmetric session key for all communication between clients and the server. This session key would be generated by the server and needed to be securely shared with the clients. In order to securely share the session key, we used an RSA key exchange scheme. The steps were as follows:

- **Step 1) Key generation:** Before any encrypted communication could take place, we needed to generate the required encryption/decryption keys.
  - Client generates an assymetric private and public RSA keypair
  - Server generates a symmetric AES session key

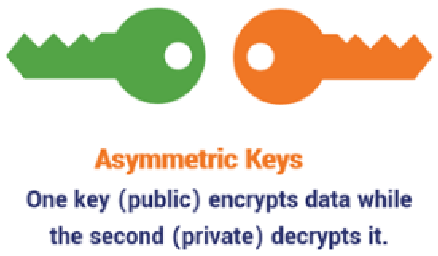

Asymmetric Keys
One key (public) encrypts data while the second (private) decrypts it.

Figure 2: RSA keypair generated by Client

Symmetric Key
A single private key shared by both parties that can encrypt and decrypt data.

Figure 3: AES key generated by Server

- **Step 2) Secure key exchange:** The server needed to securely share their session key with the client.
  1. Client sends their public RSA key to the server in unencrypted form. This is acceptable since the public RSA key can only be used for encryption.

2. The Server encrypts their session key with the clients public RSA key.
3. The Server sends their encrypted session key to the client
4. The Client uses their private RSA key to decrypt the encrypted session key

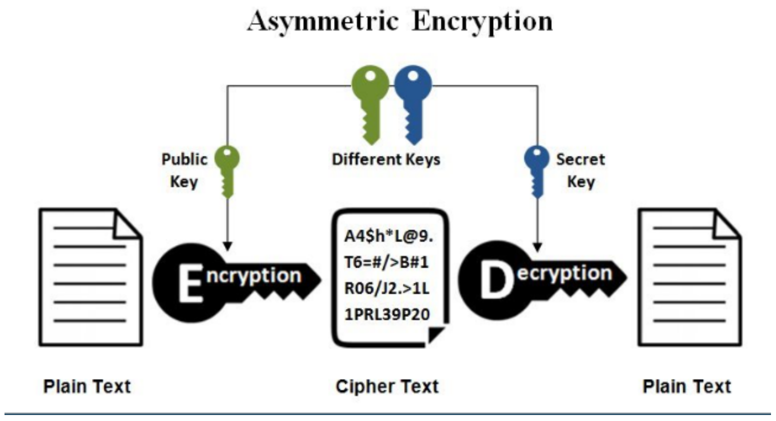


Figure 4: RSA is used for the secure key exchange

- **Step 3) Encrypted communication:** Once the client had obtained and decrypted the session key, they would use it to encrypt/decrypt all further communication with the server.

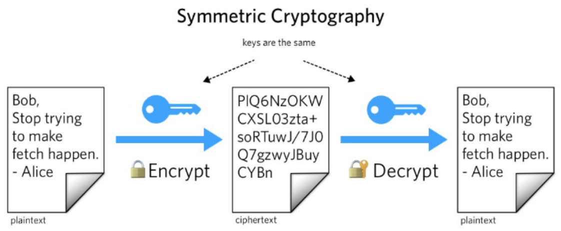


Figure 5: AES session key is used for encrypting/decrypting all messages to/from server

3.2 Implementation

In this section, we aim to document our actual implementation of encryption into the Chat application using the Java programming language and Cryptographic APIs. We will document the classes that we changed and the changes, including methods and instance fields, that we made.
To understand the changes we made, it is also important to have some basic understanding of how the existing chat application worked. The Chat application was built around the MVC(Model-View-Controller) architecture. As such, it made it easier to implement encryption since we only needed to worry about changing the model classes which handled all data and logic in the application. There were 3 modules in the chat application:

client module: This module contained the Client classes which were ClientController, ClientModel and ClientView. For implementing encryption in this project, we only modified the ClientModel.

server module: This module contained the Server classes which were ServerController, ServerModel, ServerView and ClientHandler. For implementing encryption in this project, we only modified the ServerModel and ClientHandler. ClientHandler was also a model class, it could be seen as an extension of ServerModel used to handle each individual client thread.

message module: This module contained a single class named Message, which represented a message sent between client and server. This class was modified to provide the option for encrypting and decrypting the message.

3.2.1 Message.java

The message class was the first class that we edited to implement encryption. We added 2 new constructors to it, `Message(String message, PublicKey publicKey)` and `Message(String Message, SecretKey secretKey)`. The first constructor was used to create a Message that would be encrypted by a public key. This would be used one time by the server to encrypt the session key. The other constructor was used to create a Message that was encrypted with a session key. This would be used for all messages the client sent to the server. To implement these constructors, we needed to create 2 helper methods, `public static String encrypt(String message, PublicKey publicKey)` and `public static String encrypt(String message, SecretKey secretKey)`. These helper methods were used to encrypt the messages.

We also needed to create 2 new decryption methods, so that the messages could be decrypted. To this end, we created `public String decrypt(PrivateKey privateKey)` and `public String decrypt (SecretKey secretKey)`. The first method was used once by the client to decrypt the encrypted session key sent from the server. The client would use their private RSA key as argument to the decrypt method. The second method was used to decrypt regular encrypted messages sent from the server to the clients using the AES session key.

3.2.2 ClientModel.java

Here, we generated a private and public RSA keypair using a `KeyPairGenerator` and stored them in private instance fields so that they could not be accessed outside of the program. When the client connected to the server, we sent the RSA key unencrypted to the server. The server would respond with the encrypted session key which we decrypted using private RSA key and stored in a private instance field. The session key was then used to encrypt all messages sent to the server and decrypt all messages sent from the server to the client.

3.2.3 ServerModel.java

Here, we generated a 256 bit AES session key using `KeyGenerator`. The session key was stored in a private instance field and had a getter method which allowed the ClientHandler class to acquire it.

3.2.4 ClientHandler.java

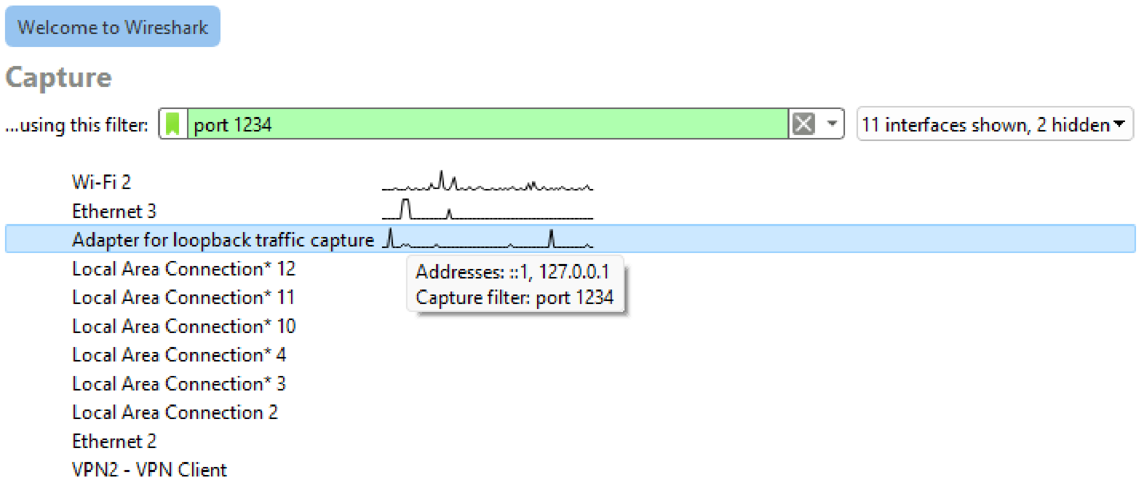
The client handler handled all communication between the server and an individual client thread. When the client sent their public key to the client handler, it would encrypt the ServerModel session key with the clients public key and send it back to the client as an encrypted message.

4 Results

The results were successful. We managed to implement encryption into the chat app and the encrypted as well as unencrypted version of the app can be found here: Github. To test and verify that the app actually works, we used Wireshark and Java print statements

4.1 Wireshark

Wireshark is a network traffic analyzer that allows you to monitor network traffic and inspect packets sent over the network[3]. To test our app, we started the Server locally on our computer and opened the client. We conneced to the server(localhost) and then we started Wireshark and filtered for traffic on port 1234 on the loopback interface. We tested both the unencrypted version of the chat app and the encrypted version of the chat app. Once connected as client, we sent the message "hello". In the unencrypted version of the chat app, we could see this message in plain text in wireshark. In the encrypted version of the chat app, we could also intercept the message but we could not read it in plain text. It was instead a message cipher that was impossible to break since it was generated with a 256 bit AES session key.



Welcome to Wireshark

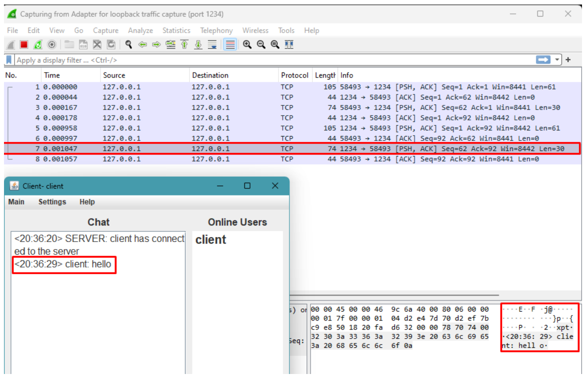
Capture

...using this filter: port 1234 — 11 interfaces shown, 2 hidden

- Wi-Fi 2
- Ethernet 3
- Adapter for loopback traffic capture
- Local Area Connection* 12
- Local Area Connection* 11
- Local Area Connection* 10
- Local Area Connection* 4
- Local Area Connection* 3
- Local Area Connection 2
- Ethernet 2
- VPN2 - VPN Client

Addresses: ::1, 127.0.0.1
Capture filter: port 1234

Figure 6: Wireshark loopback capture on port 1234



Capturing from Adapter for loopback traffic capture (port 1234)

```
No.  Time      Source     Destination  Protocol Length Info
  1 0.000000  127.0.0.1  127.0.0.1    TCP      105 58493 → 1234 [PSH, ACK] Seq=1 Ack=1 Win=8441 Len=61
  2 0.000044  127.0.0.1  127.0.0.1    TCP       44 1234 → 58493 [ACK] Seq=1 Ack=62 Win=8442 Len=0
  3 0.000167  127.0.0.1  127.0.0.1    TCP       74 58493 → 1234 [PSH, ACK] Seq=62 Ack=1 Win=8441 Len=30
  4 0.000178  127.0.0.1  127.0.0.1    TCP       44 1234 → 58493 [ACK] Seq=1 Ack=92 Win=8442 Len=0
  5 0.000958  127.0.0.1  127.0.0.1    TCP      105 1234 → 58493 [PSH, ACK] Seq=1 Ack=92 Win=8442 Len=61
  6 0.000997  127.0.0.1  127.0.0.1    TCP       44 58493 → 1234 [ACK] Seq=92 Ack=62 Win=8441 Len=0
  7 0.001047  127.0.0.1  127.0.0.1    TCP       74 1234 → 58493 [PSH, ACK] Seq=62 Ack=92 Win=8442 Len=30
  8 0.001057  127.0.0.1  127.0.0.1    TCP       44 58493 → 1234 [ACK] Seq=92 Ack=92 Win=8441 Len=0
```

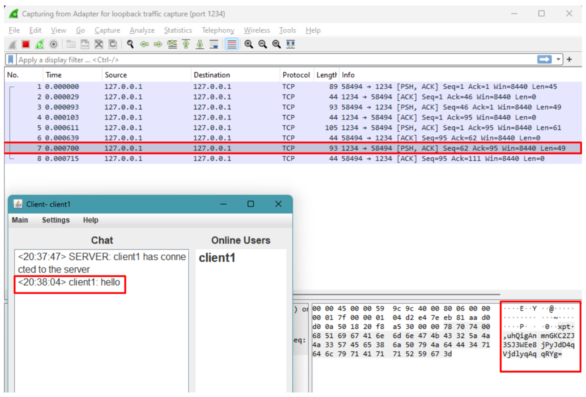
Client- client

Chat
<20:36:20> SERVER: client has connected to the server
<20:36:29> client: hello

Online Users
client

```
00 00 45 00 00 46  9c 6a 40 00 80 06 00 00   ····E··F ·j@·····
00 01 7f 00 00 01  04 d2 e4 7d 70 d2 ef 7b   ········ ···}p··{
c9 e8 50 18 20 fa  d6 32 00 00 78 70 74 00   ····P·  ·2··xpt·
32 30 3a 33 36 3a  32 39 3e 20 63 6c 69 65   ·<20:36: 29> clie
3a 20 68 65 6c 6c  6f 0a                     nt: hell o·
```

Figure 7: Message "hello" intercepted as plain text on Wireshark using unencrypted chat app



Capturing from Adapter for loopback traffic capture (port 1234)

```
No.  Time      Source     Destination  Protocol Length Info
  1 0.000000  127.0.0.1  127.0.0.1    TCP       89 58494 → 1234 [PSH, ACK] Seq=1 Ack=1 Win=8440 Len=45
  2 0.000029  127.0.0.1  127.0.0.1    TCP       44 1234 → 58494 [ACK] Seq=1 Ack=46 Win=8440 Len=0
  3 0.000093  127.0.0.1  127.0.0.1    TCP       93 58494 → 1234 [PSH, ACK] Seq=46 Ack=1 Win=8440 Len=49
  4 0.000103  127.0.0.1  127.0.0.1    TCP       44 1234 → 58494 [ACK] Seq=1 Ack=95 Win=8440 Len=0
  5 0.000611  127.0.0.1  127.0.0.1    TCP      105 1234 → 58494 [PSH, ACK] Seq=1 Ack=95 Win=8440 Len=61
  6 0.000639  127.0.0.1  127.0.0.1    TCP       44 58494 → 1234 [ACK] Seq=95 Ack=62 Win=8440 Len=0
  7 0.000700  127.0.0.1  127.0.0.1    TCP       93 1234 → 58494 [PSH, ACK] Seq=62 Ack=95 Win=8440 Len=49
  8 0.000715  127.0.0.1  127.0.0.1    TCP       44 58494 → 1234 [ACK] Seq=95 Ack=111 Win=8440 Len=0
```

Client- client1

Chat
<20:37:47> SERVER: client1 has connected to the server
<20:38:04> client1: hello

Online Users
client1

```
00 00 45 00 00 59  9c 9c 40 00 80 06 00 00   ····E··Y ··@·····
00 01 7f 00 00 01  04 d2 e4 7e eb 81 aa d0   ········ ···~····
d0 0a 50 18 20 f8  a5 30 00 00 78 70 74 00   ····P·  ·0··xpt·
68 51 69 67 41 6e  6d 6e 47 4b 43 32 5a 4a   ,uhQigAn mnGKC2ZJ
4a 33 57 45 65 38  6a 50 79 4a 64 44 34 71   3SJ3WEe8 jPyJdD4q
64 6c 79 71 41 71  71 52 59 67 3d            VjdlyqAq qRYg=
```

Figure 8: Message "hello" intercepted as ciphered text on Wireshark using encrypted chat app

4.2 Java Print statements

To verify that what Wireshark was capturing was indeed the encrypted messages, we used `System.out.println()` statements inside our code. We used print statements when the client received a message as well as when the server received a message to print out the message(in encrypted string form) before decrypting it. The results showed us that the encrypted messages that the server and client printed out were identical and also matched the encrypted strings that Wireshark capture was showing.

5 Discussion

5.1 Sending Encrypted Message as String

Since we sent all encrypted messages as encrypted strings, this presented some bugs and challenges along the way. One of the bugs occured in the encrypt helper methods when we tried to convert the encrypted message bytes to a string using the String constructor. This did not work and the reason was that when the message bytes were encrypted, the encrypted bytes could no longer be represented using the standard character encoding of the String Class. The String constructor interprets the bytes as characters according to a specific character encoding. If the byte array does not represent a valid sequence of characters in the specified encoding (or the default encoding), the resulting string may contain invalid or unreadable characters. This is because encrypted byte data does not necessarily correspond to valid character data.

The solution was to use the `Base64` encoder class to encode the encrypted bytes to an ASCII string without losing or corrupting data in doing so. The Base64 encoder can represent the encrypted bytes as valid ASCII string sequence. Base64 encoding ensures that any binary data (including encrypted bytes) is converted into a text representation using only printable ASCII characters (A-Z, a-z, 0-9, +, /). This avoids the issues of invalid characters and data corruption/loss during transformation of bytes to string.

So instead of doing `return new String(encryptedBytes)` we did `return Base64.getEncoder().encodeToString(encryptedBytes)`.

However, in the decrypt methods, when we returned a string, we used the String Constructor. Here it was the opposite. If we used `Base64` encoder to encode the decrypted bytes to a string, it would cause the program to crash.

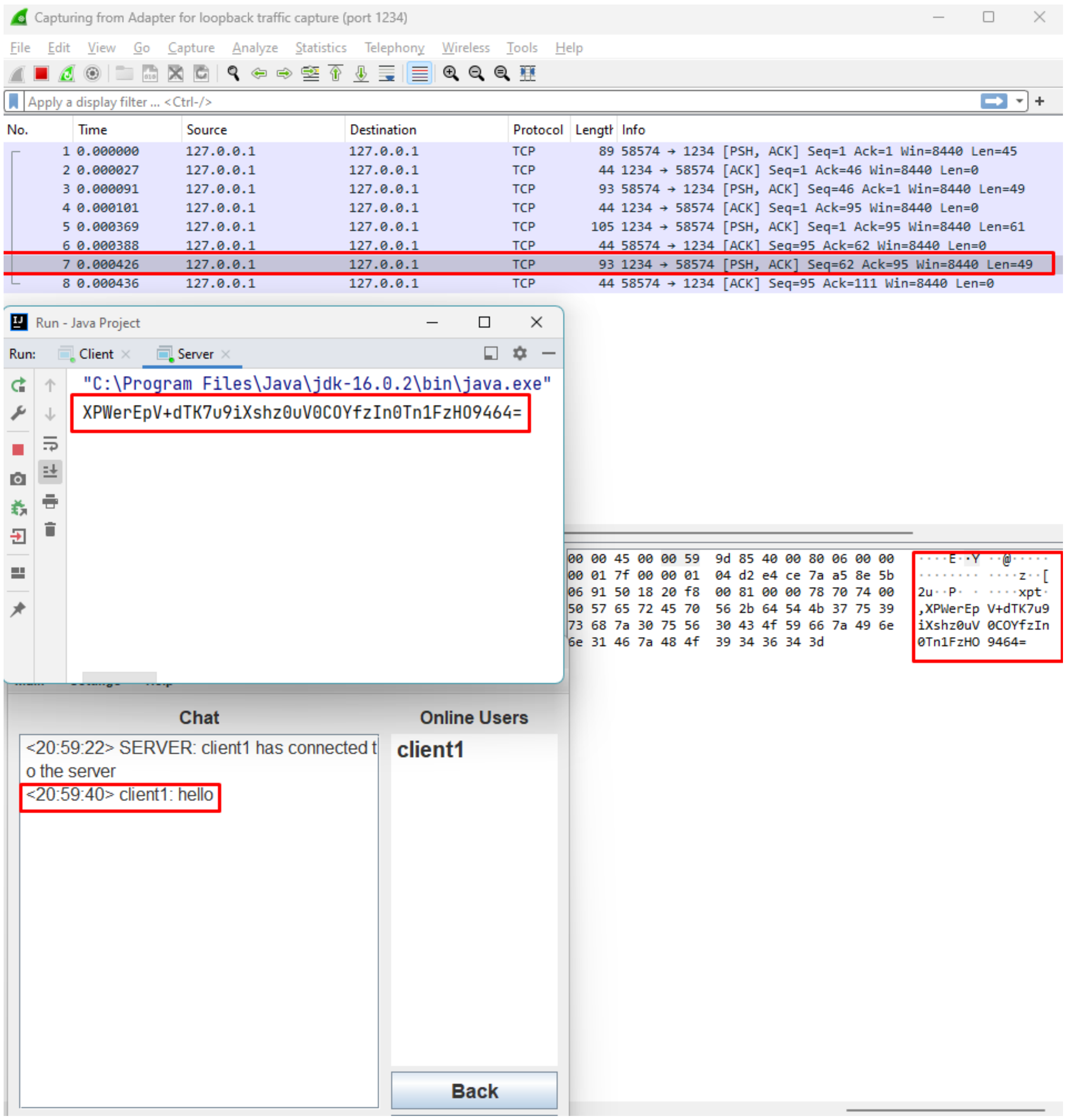
5.2 End-to-End encryption

The original goal of our chat application was to implement end-to-end encryption. End-to-end encryption ensures that only the clients who are communicating with each other can read each others messages, so that not even the server or service provider has access to the conversations[5]. End-to-end encryption could be seen as the standard of security today in messaging solutions, since not only specialized privacy-oriented messaging apps such as Signal use it, but also major mainstream message applications such as WhatsApp, IMessage, Telegram, Google Messages and Facebook Messenger[6].

However, this was harder to implemenet and required more extensive design approach. Therefore we did not implement end-to-end encryption. This meant that the Server could read all messages between the clients. But if the Server ran locally on one of the clients computers, this would be equivalent to end-to-end encryption between the clients. However if the server was hosted on a third party, this would mean that the third party, such as hosting provider, would have access to all conversations between clients.

5.3 Key Exchange algorithm

We used a RSA key exchange algorithm so that the server could securely share their session key with the clients. Another possible and more common approach would be to use the diffie-hellman key exchange algorithm[8]. This algorithm is the standard key exchange algorithm used today and is used in security


Capturing from Adapter for loopback traffic capture (port 1234)
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time      Source     Destination  Protocol Length Info
  1 0.000000  127.0.0.1  127.0.0.1    TCP       89 58574 → 1234 [PSH, ACK] Seq=1 Ack=1 Win=8440 Len=45
  2 0.000027  127.0.0.1  127.0.0.1    TCP       44 1234 → 58574 [ACK] Seq=1 Ack=46 Win=8440 Len=0
  3 0.000091  127.0.0.1  127.0.0.1    TCP       93 58574 → 1234 [PSH, ACK] Seq=46 Ack=1 Win=8440 Len=49
  4 0.000101  127.0.0.1  127.0.0.1    TCP       44 1234 → 58574 [ACK] Seq=1 Ack=95 Win=8440 Len=0
  5 0.000369  127.0.0.1  127.0.0.1    TCP      105 1234 → 58574 [PSH, ACK] Seq=1 Ack=95 Win=8440 Len=61
  6 0.000388  127.0.0.1  127.0.0.1    TCP       44 58574 → 1234 [ACK] Seq=95 Ack=62 Win=8440 Len=0
  7 0.000426  127.0.0.1  127.0.0.1    TCP       93 1234 → 58574 [PSH, ACK] Seq=62 Ack=95 Win=8440 Len=49
  8 0.000436  127.0.0.1  127.0.0.1    TCP       44 58574 → 1234 [ACK] Seq=95 Ack=111 Win=8440 Len=0
```

```
"C:\Program Files\Java\jdk-16.0.2\bin\java.exe"
XPWerEpV+dTK7u9iXshz0uV0COYfzIn0Tn1FzHO9464=
```

```
00 00 45 00 00 59  9d 85 40 00 80 06 00 00   ····E··Y ··@·····
00 01 7f 00 00 01  04 d2 e4 ce 7a a5 8e 5b   ········ ····z··[
06 91 50 18 20 f8  00 81 00 00 78 70 74 00   2u··P·   ····xpt·
50 57 65 72 45 70  56 2b 64 54 4b 37 75 39   ,XPWerEp V+dTK7u9
73 68 7a 30 75 56  30 43 4f 59 66 7a 49 6e   iXshz0uV 0COYfzIn
6e 31 46 7a 48 4f  39 34 36 34 3d            0Tn1FzHO 9464=
```

```
<20:59:22> SERVER: client1 has connected t
o the server
<20:59:40> client1: hello
```

Figure 9: Wireshark and System.out.println() statements test

protocols such as SSH and TLS[7]. The main difference between RSA key exchange algorithm and Diffie-hellman is that in the latter, the shared key is calculated locally on the client/server side. So the shared key is not sent over the network at all. The only thing which is sent is the parameters, which include the generator and prime modulus[4].

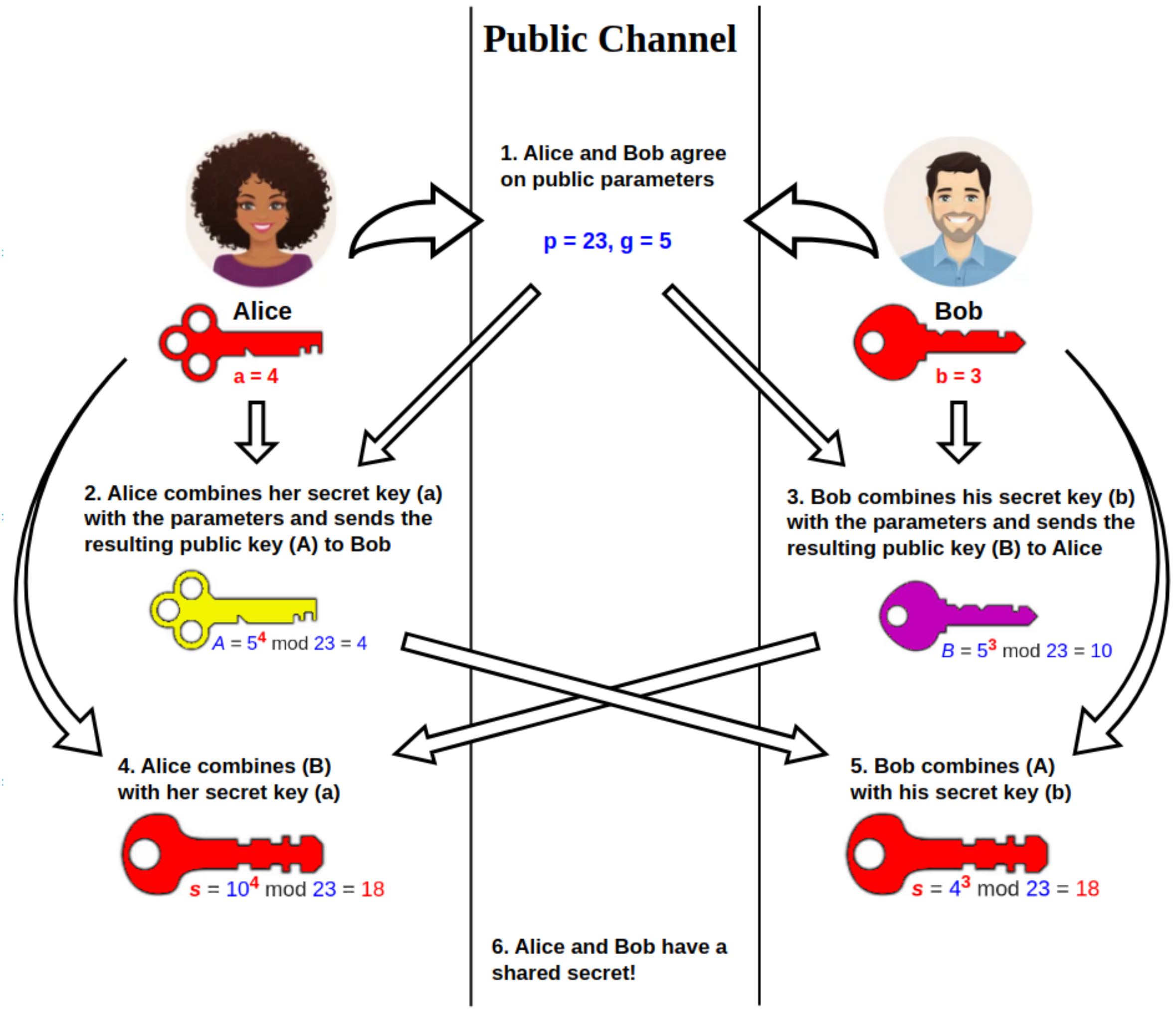

Public Channel
1. Alice and Bob agree on public parameters
p = 23, g = 5
Alice
a = 4
Bob
b = 3
2. Alice combines her secret key (a) with the parameters and sends the resulting public key (A) to Bob
A = 5^4 mod 23 = 4
3. Bob combines his secret key (b) with the parameters and sends the resulting public key (B) to Alice
B = 5^3 mod 23 = 10
4. Alice combines (B) with her secret key (a)
s = 10^4 mod 23 = 18
5. Bob combines (A) with his secret key (b)
s = 4^3 mod 23 = 18
6. Alice and Bob have a shared secret!

Figure 10: Diffie-Hellman Key Exchange algorithm

Although diffie-hellman is more common key exchange algorithm, we found no evidence that it would be more secure than RSA key exchange algorithm

6 Conclusion

In conclusion, our project has managed to succesfully implement encryption into an existing Java Client-Server chat application. The encryption ensured that only the clients and server had access to the communicated messages in plain text. We tested and verified that our encryption soluton worked using WireShark and Java print statements. The code, as well as encrypted and unencrypted chat applications, can be found on Github. A video tutorial of how the encryption was implemented can be found on Youtube.

For future considerations, we could try to implement end-to-end encryption for added security as well as try different key exchange algorithms.

References

[1] Aes and rsa - about. Accessed on May 22, 2024. URL: `https://preyproject.com/blog/types-of-encryption-symmetric-or-asymmetric-rsa-or-aes`.

[2] Encrypted - about. `https://securitysenses.com/posts/importance-encryption-messaging-apps`. Accessed on May 22, 2024.

[3] Wireshark - about. Accessed on May 22, 2024. URL: `https://www.wireshark.org/about.html`.

[4] How does the diffie-hellman key exchange work?, 2013. Accessed: May 22, 2024. URL: `https://youtu.be/M-0qt6tdHzk`.

[5] Cloudflare, Inc. What is end-to-end encryption?, 2024. Accessed: May 22, 2024. URL: `https://www.cloudflare.com/learning/privacy/what-is-end-to-end-encryption/`.

[6] Sliksafe. 16 apps that use end-to-end encryption, 2024. Accessed: May 22, 2024. URL: `https://www.sliksafe.com/blog/16-apps-that-use-end-to-end-encryption`.

[7] TechTarget. Diffie-hellman key exchange, 2024. Accessed: May 22, 2024. URL: `https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange`.

[8] Wikipedia contributors. Diffie–hellman key exchange, 2024. Accessed: May 22, 2024. URL: `https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange`.

A Code

This will go over the some of the code additions that were necessary to implement encryption in the Chat application. All code additions, such as the key exchange, have not been included since they were dependant on other parts of the code. For the full code additions and changes, see Github.

A.1 Message.java

This shows the changes made to message.java. 2 new constructors were added that supported encryption with session key and public key, as well as 2 new encryption helper methods and 2 decryption helper methods.

```java
public Message(String message, SecretKey sessionKey) throws
↪   IllegalBlockSizeException, NoSuchPaddingException, BadPaddingException,
↪   NoSuchAlgorithmException, InvalidKeyException {
    String time;
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("HH:mm:ss");
    LocalDateTime now = LocalDateTime.now();
    time = "<" + dtf.format(now) + "> ";
    this.message = encrypt(time + message + "\n",sessionKey);
}

public Message(String message, PublicKey publicKey) throws NoSuchPaddingException,
↪   IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException,
↪   InvalidKeyException {
    this.message = encrypt(message,publicKey);
}

public static String encrypt(String message, SecretKey sessionKey) throws
↪   IllegalBlockSizeException, BadPaddingException, InvalidKeyException,
↪   NoSuchPaddingException, NoSuchAlgorithmException {
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, sessionKey);
    byte[] encryptedBytes = cipher.doFinal(message.getBytes());

    return Base64.getEncoder().encodeToString(encryptedBytes);
}

public static String encrypt(String message, PublicKey publicKey) throws
↪   BadPaddingException, InvalidKeyException, NoSuchPaddingException,
↪   NoSuchAlgorithmException, IllegalBlockSizeException {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE,publicKey);
    byte[] encryptedBytes = cipher.doFinal(message.getBytes());

    return Base64.getEncoder().encodeToString(encryptedBytes);
}

public String decrypt(SecretKey sessionKey) throws NoSuchPaddingException,
↪   NoSuchAlgorithmException, InvalidKeyException, IllegalBlockSizeException,
↪   BadPaddingException {
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE,sessionKey);
    byte[] encryptedBytes = Base64.getDecoder().decode(this.message);
    byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

    return new String(decryptedBytes);
}

public String decrypt(PrivateKey privateKey) throws NoSuchPaddingException,
↪   NoSuchAlgorithmException, InvalidKeyException, IllegalBlockSizeException,
↪   BadPaddingException {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE,privateKey);
    byte[] encryptedBytes = Base64.getDecoder().decode(this.message);
    byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

    return new String(decryptedBytes);
}
```

A.2 ServerModel.java

This includes the code for generating the AES session key.

```
private SecretKey sessionKey;

public ServerModel(){
    KeyGenerator keyGenerator = null;
    try {
        keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(256);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    this.sessionKey = keyGenerator.generateKey();
}
```

A.3 ClientModel.java

This includes the code for creating the RSA keypair. The code for sending the public key to the server and receiving,decrypting and storing the session key sent from server has not been included.

```
private PublicKey publicKey;
private PrivateKey privateKey;
private SecretKey sessionKey;

private static KeyPair generateKeyPair(){
    KeyPairGenerator keyPairGenerator = null;
    try {
        keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return keyPairGenerator.generateKeyPair();
}

public ClientModel(){
    KeyPair keyPair = generateKeyPair();
    this.privateKey = keyPair.getPrivate();
    this.publicKey = keyPair.getPublic();
}
```