02232 Applied Crypto Course
Lab Sheet #1

Establishing Secure Message Exchange

Handout Date: Thursday 10th September 2020
Deadline: Thursday 24th September 2020 at 23:59pm via DTU Inside

1. Scope and Purpose

1.1 Purpose

The purpose of this lab is to provide hands on experience with **simple cryptographic primitives** needed to establish **secure communication channels**. The lab is based on a Chat Application providing communication capabilities between clients and a server and between the clients themselves. The software application has been created following the standards adopted by other commodity chat applications so as to get better experience with experimenting with near real-world code primitives towards analysing the security goals that need to be met within the application, identifying where in the code appropriate crypto algorithms need to be applied and also decide what cryptographic algorithms are best suited for meeting these goals.

1.2 Goals

The Chat Client and Server used throughout most of the lab exercises provides **no communication security**; all messages transmitted between agents in the system are in **plaintext**, which means that they can be easily intercepted by adversaries. In order to provide **confidential communication**, we need to be able to **encrypt these messages**. The messages will be encrypted via a symmetric (secret) key, which needs to be either shared or agreed between communicating clients. In this exercise, you will explore both the integration of encryption functions based on pre-existing keys but also the key agreement phase based on the use of Diffie Hellman. For the former, you are welcome to use any of the existing functions provided by the Java Cryptography API. For the latter, you will need to implement the Diffie-Hellman protocol (*no need to currently include public key crypto*) for establishing a secret key between communication clients that can be later used for exchanging encrypted messages.

Overall, there are three communication channels that need to be secure:

- **Client to Server.** Messages that are sent from a client to the server need to be encrypted. For this mode, you can consider pre-existing keys that have already been established and the focus is on identifying appropriate cryptographic primitives to employ for providing strong confidentiality. *In this case, you are welcome to hard code the same key between the server and the client.*
- **Server to Client.** Once a "public message" is being sent by a client to the server, this implies that the message will be broadcasted (i.e., forwarded) by the server to all connected clients. **This broadcast operation needs to also be secure.** This means that assuming the existence of secret keys between the server and all the clients, each message that is sent to the server needs to: (i) first be decrypted using the secret key established with the initial sender client, and (ii) encrypted using all the keys being established with the other clients and then the resulting cipher texts are transmitted to the respective clients. *In this case, you are also welcome to hard code the secretekeys between the server and all the clients.* Particular focus needs to be given to the correct encryption of the message based on the different keys with all the clients.
- **Client to Client (peer to peer).** This mode of communication allows the "private exchange of messages" between two clients without the need to go through the server. For this case, you will need to implement

the **Diffie-Hellman protocol for having the clients agreeing on the secret key** to be used for the later encryption and also provide the encryption process based on this established key.

More information on the Chat Application software can be found in the next section whereas specific on the problem definition can be found in Section 3.

2. The Chat Application

Socket programming is used for developing client-server applications in Java. Sockets allows two or more devices to communicate with each other over the network using TCP/IP protocol. The provided Chat Application is based on the use of Sockets and multi-threaded client-server architecture for achieving the following three main functionalities:

- Clients can send messages to the server and all other clients (**public chat**);
- Clients can communicate with each other through the server (**private chat** – not the focus of this exercise);
- Clients can communicate with each other without the server (**P2P communication**).

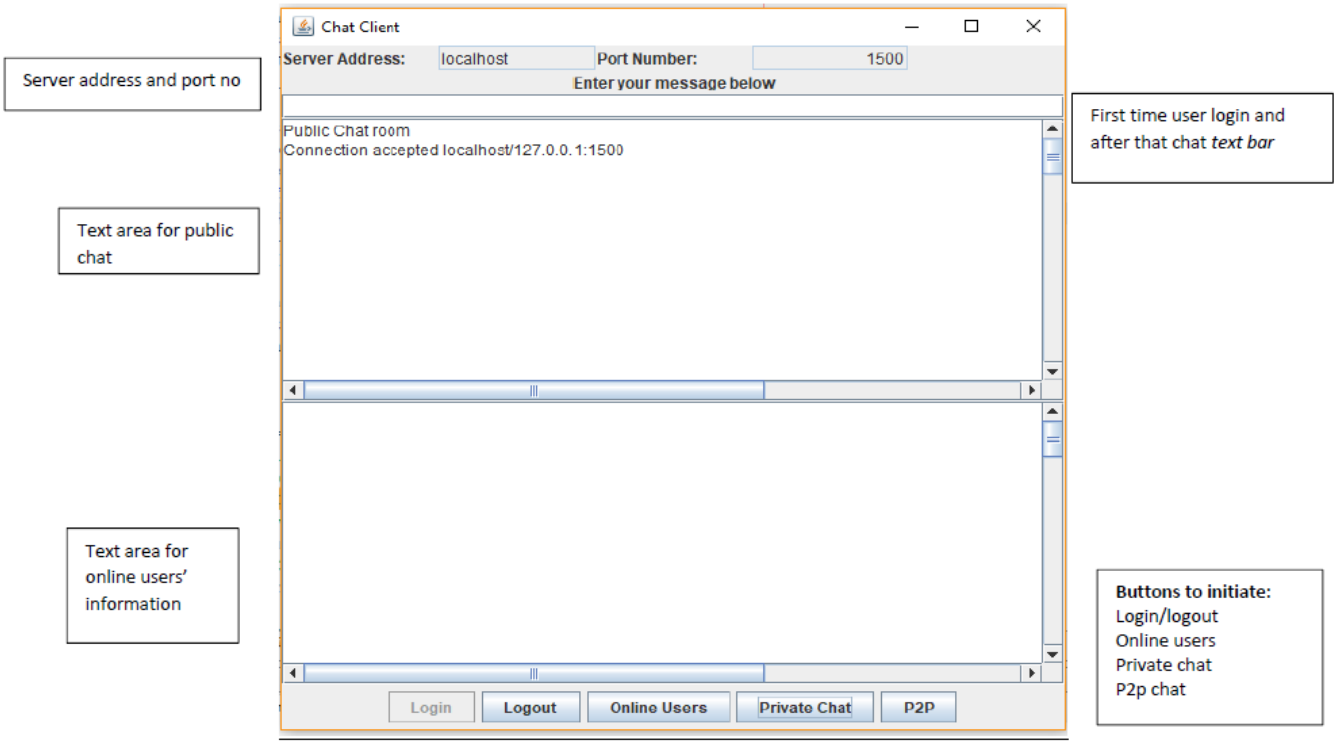
The Chat Application is composed of two parts; one that is running the Server and is responsible for setting up and handling a multi-threaded Socket Server and another that is running on any Client trying to connect to the server (currently through a non-secure socket). Once the socket connection has been established, both parties can start exchanging messages.

The software has been implemented following the **singleton pattern** that restricts the instantiation of a class or a system component to one "single" instance. This is particularly useful for applications based on the use of multiple components each responsible for a specific functionality. All these components are then coordinated by one object (the **ComponentManager class**) that is responsible for handling the graceful boot up and shut down of all other system components responsible for setting up the socket server, the client socket, etc.

The application has **7 packages** out of which two are the ones that provide the core messaging functionalities and you need to focus on; namely **the ServerSocketEngine** and **ClientSocketEngine**.

- The ServerSocketEngine initially fires up a Socket Server that listens to a specified port for any new connections. It holds and manages a *configurable* pool of connection handlers that wait until there is a client to serve. When a new connection is established, a connection handler is assigned to it for handling any message between the server to the remote client.
- Once the Socket Server is operational, the ClientSocketEngine running on a client sets up and configures a socket connection to the specified port and hostname of the server. If the connection is successful, it then manages the necessary **write and read streams** for sending and receiving messages, respectively.
- The same approach is been followed and handled by the ClientSocketEngine when it comes to establishing private client to client to connections: In this case, each Client sets up its own Socket Server where other clients can then connect and start exchange messages.
- Communication streams from server to the client is done through string objects while messages from client to server are implemented as an object of ChatMessage. ChatMessage has a type (type of message) and a message that contains the actual chat string (ChatMessage.java).

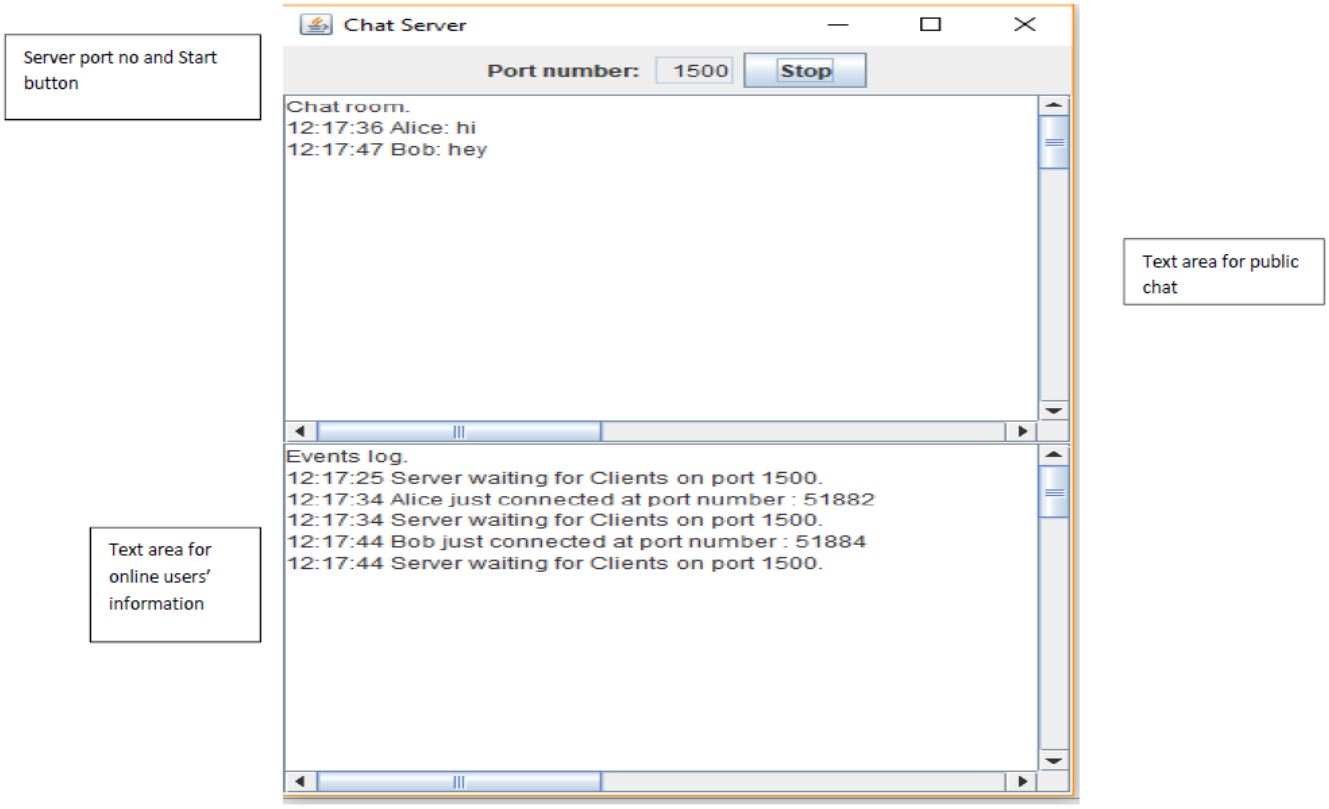
The **ClientSocketEngine has 4 files**: the ClientEngine.java, the ClientSocketGUI.java, the ListenFromServer.java and the P2PClient.java. The ClientEngine is responsible for setting up the required socket to the server (once the ClientSocketGUI has finished create the Chat Application Client Window) and the underlying thread for handling messages transmitted by the server (ListenFromServer.java).



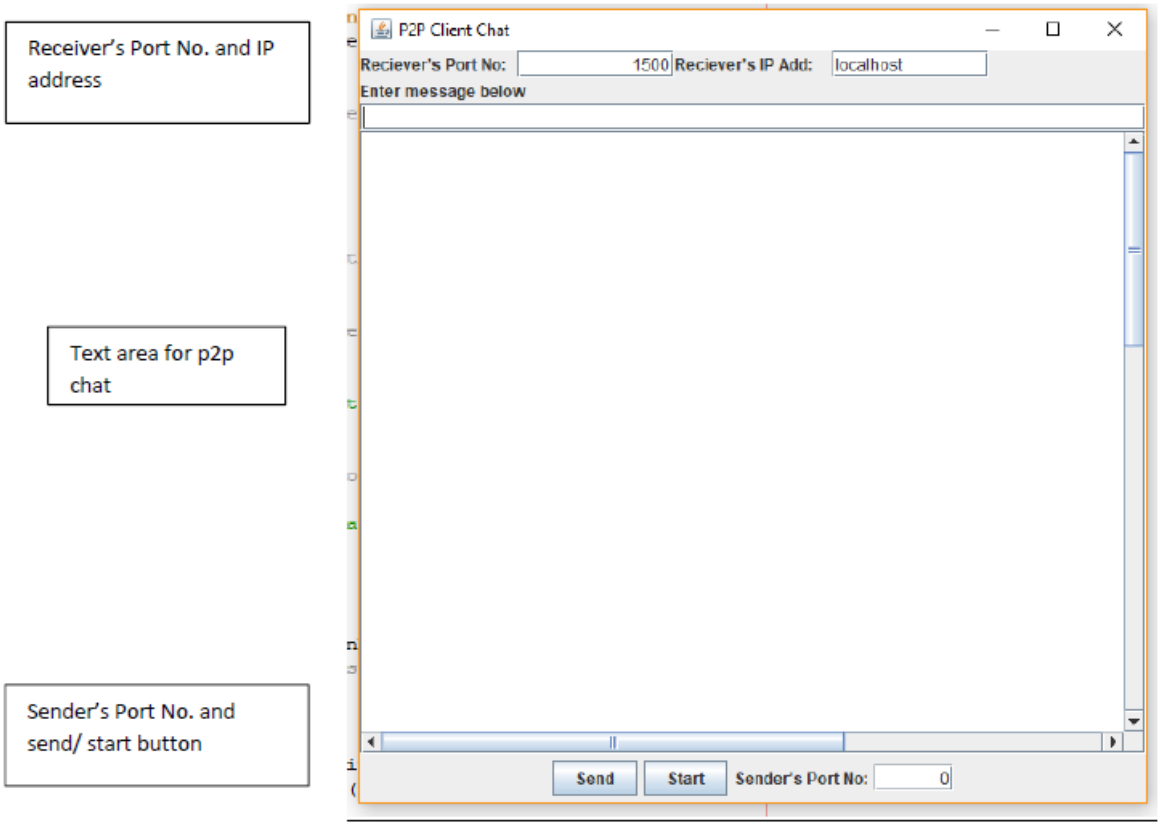
Two java applications can communicate with each other via sockets. A TCP connection is established via socket in which stream of bytes are transferred to and from different instances or applications. Java provides a facility of 'Java objects' as it performs the serializing and deserializing of the objects automatically. User just have to create an ObjectInputStream and an ObjectOutputStream from the Socket InputStream and the Socket OutputStream. These classes mostly focus on how to create sockets and how to read and write to/from them in order to communicate with the server. It listens from the server. As this application is multithreaded, several clients run simultaneously as threads i.e. all client threads use the same piece of code.

The **ServerSocketEngine has 3 files**: the SocketConnectionHandler.java, the SocketServerEngine.java and the SocketServerGUI.java. The SocketServerEngine is responsible for setting up the Socket Server which listens for connections from online clients (once the SocketServerGUI has finished the creation of the respective Chat Application Server Window). Each new incoming connection is being handled by a SocketConnectionHandler threat that sets up all the necessary input and output streams.

The **P2PClient.java** deals with how two clients can communicate with each other without the server knowing; they establish a socket connection of their own. Both clients should know the IP address and port number of the sender and receiver node i.e. their own address and the client's address they want to talk to. As in this scenario, there is no other way for them to know about the existence of each other than with the help of server, so first they exchange IP address (which is localhost for simplicity) and port numbers on a private message with server knowing. Then they can establish an independent connection of their own.



Each client first sets up its own Socket Server where the other remote clients can connect to before starting exchanging messages. The ListenFromClient class (within the P2PClient.java) file is responsible for handling the incoming messages on this server.



2.1 Running Instructions

The current version of the Chat Application contains the following files that you will need for successfully running the app:

- **ChatApplication-Server.jar:** Package holding all class files needed to run either the Server or the Client.
- **chatapplication.properties:** Configuration file holding all the necessary properties that need to be loaded in the Chat Application server and/or client. These **configuration properties** mainly comprise the port number of the server (ServerPort), the address (i.e., IP) of the server (ServerAddress), the maximum number of parallel client connections that the server can support (ConnectionHandlers) and the username of the client (ClientUsername). If one wishes to change a defined property he/she *must change only the property value* and not the name since this is used by the system to read this specific property value. The stored properties in the chatapplication.properties file will be loaded automatically to the Chat Application once it is started. It must resides in the same directory as the Chat Application Program Code, which is, by default, where the app will go to look for this properties file upon start up.

After having successfully set up the configuration properties of the application, you need to start it up. This can be done through the command terminal as seen below:

Go to the directory where the ChatApplication-Server.jar file resides and type, in a simple command line (Windows) or terminal (Linux), one of the following commands:

	Command	Possible Response (s)
Configuration file in same directory	**FOR THE Chat Application Server...** java –cp ChatApplication-Server.jar chatapplication_server.ChatApplicationServerEngine Mode=Server **FOR A CHAT APPLICATION CLIENT...** java –cp ChatApplication-Server.jar chatapplication_server.ChatApplicationServerEngine Mode=Client	**BOOT UP successfully OR FATAL EXCEPTION + (kind of exception)**

This command, by default, checks for the configuration file in the same directory where the package holding the .class files resides. The user has to give the mode of operation (Server or Client) as an argument in order for the app to boot up correctly.

You should install a Java platform for loading the created app. You are encouraged to use NetBeans as this is the platform that was used for the compilation of the code and, thus, it follows the respective structure.

Run as many instance of the Client as you want, but run at least two or three so that you can test the P2P connection. Once you have fired up a client, write a username and press 'Login' to connect to the server. Once you are logged in, you can write messages in the tab which will be visible to server and all the clients (Broadcast Operation). If you want to establish a connection to a specific client, first press the button 'Online Users' and find out the port number of that client and then press 'private chat', enter the port number of relevant user and send a message. Note that this communication is done via server i.e. server knows about the communication.

If you want to establish a P2P channel, it means both of the clients need a listening port, so one client will send his new (random) port number to the desired user via 'private chat' and if that user wants to communicate then it will share his port number too (for simplicity the IP address is the localhost, but it can be run on different machines, in that case IP address needs to be shared too). To establish P2P connection, click 'P2P' and put the socket to listening state by pressing 'Start' after entering your port number.

3. Problem Definition

What you need to focus in this assignment is to provide security support for this Chat Application with a particular focus on supporting **secure messaging**. You need to identify the appropriate parts of the code that need to be updated towards allowing the clients and servers to communicate over secure connection. In this secure connection, the messages that are being sent should be encrypted (before transmission) and decrypted upon reception and before processing. Hence, this process should be *transparent* to the user of the chat application. **Both the clients and the server should encrypt all messages before transmission.** Thus, the security goals that you need to take into consideration are:

- **Confidentiality:** When messages are being passed between the server and the clients, third parties can view this data. However, you are responsible to encrypt these messages so that they remain confidential and cannot be deciphered by third parties.
- **Integrity:** You need to guarantee that the transmitted messages will not modified in transit by a third party.

As described in Section 1.2, you need to secure all 3 modes of message exchange between the clients and the server or the clients themselves. For the former, you can assume the existence of pre-established key, thus, the focus is on the pure encryption. For the latter, you will need to implement the Diffie Hellman key agreement protocol before you can start the encryption phase.

Details on the underpinnings of the Diffie Hellman protocol can be found on Slide 59 of the CryptoBasics(1) lecture.

IMPORTANT NOTE: There is no need to currently consider the use of public key crypto as part of the Diffie-Hellman

3.1 Diffie-Hellman

The Diffie-Hellman protocol is a method for two computer users to generate a shared private key with which they can then exchange information across an insecure channel. The important point for consideration here is

Personal information is not being shared during the key exchange; instead, both parties are creating a key together

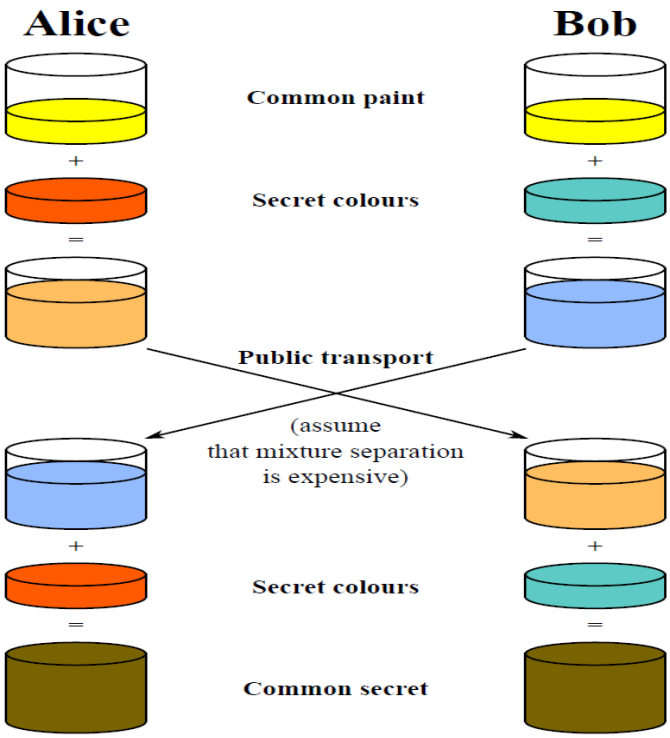
This technique is useful to generate a mutual shared key among interested parties, and then the rest of the communication (traffic) is encrypted with this key. The Diffie-Hellman protocol is described below

1. Alice comes up with a prime number **p** and a number **g**, which is coprime to p-1, and tells Bob what they are.
2. Bob then picks a secret number (**a**), compute $\mathbf{g}^{\mathbf{a}}$ *mod* **p** and send that result **A** to Alice (let's say- $\mathbf{g}^{\mathbf{a}}$ *mod* **p = A)**
3. Alice then picks a secret number (**b**), compute $\mathbf{g}^{\mathbf{b}}$ *mod* **p** and send that result **B** to Bob (let's say- $\mathbf{g}^{\mathbf{b}}$ *mod* **p = B)**
4. Bob computes $\mathbf{B}^{\mathbf{a}}$ *mod* **p** to get a key **K**
5. Alice computes $\mathbf{A}^{\mathbf{b}}$ *mod* **p** to get a key **K**

The value of **K** is same at step 4 and 5 due to the property of modulo exponents (i.e. you get the same answer no matter, which order you do the exponentiation in). Specifically:

$$(\mathbf{g}^{\mathbf{a}} \textit{ mod } \mathbf{p})^{\mathbf{b}} \textit{ mod } \mathbf{p} = \mathbf{g}^{\mathbf{ab}} \textit{ mod } \mathbf{p}$$
$$(\mathbf{g}^{\mathbf{b}} \textit{ mod } \mathbf{p})^{\mathbf{a}} \textit{ mod } \mathbf{p} = \mathbf{g}^{\mathbf{ba}} \textit{ mod } \mathbf{p}$$

Alice never knew what secret number Bob chose to get to the result and Bob never knew what number Alice picked, but both will get the same result. It can be simply illustrated as:



If intruder ever analyses the recorded traffic, there is absolutely no way to figure out what the key was, even though the exchanges that created it may have been visible. This is where perfect forward secrecy comes from. Nobody analysing the traffic can later break in because the key was never saved, never transmitted, and never made visible anywhere.

For more details, check: https://www.comparitech.com/blog/information-security/diffie-hellman-key-exchange/

To implement the Diffie-Helman protocol in a JAVA/C# project, there are several cryptographic libraries that can be used, which are well tested and documented. One of the most commonly used in Bouncy castle.

The Bouncy Castle Crypto API for Java provides a lightweight cryptography API that complements the default Java Cryptographic Extension (JCE). It provides a wide range of functions for

- Encryption/decryption
- Key management (generation, storage and distribution)
- Key exchange
- Public key Infrastructure
- Digital signatures
- Hashing and much more

Detailed description of every function is available in Javadoc's.
For installation and importing in a java project check: https://www.itcsolutions.eu/2011/08/22/how-to-use-bouncy-castle-cryptographic-api-in-netbeans-or-eclipse-for-java-jse-projects/

For detailed Java examples check: http://www.java2s.com/Tutorial/Java/0490__Security/Catalog0490__Security.htm

IMPORTANT NOTE: You are allowed to use such libraries provided by Java. However, more points will be awarded if you manually implement the process of the Diffie-Hellman so as to have both clients agreeing on a secret key.

4. What to Submit?

The lab should be documented by a short group report, explaining how the secure messaging was achieved. The report should include at least four sections: **Security Goals**, which analyses the security goals of the application, **Algorithm**, which discuss what cryptographic algorithms meet these goals, **Modes**, which discusses which cryptographic modes should be used, and **Implementation** that provides a simple overview of the implementation, e.g. through simple code snips of the modifications to the original chat application code.

The discussions of algorithms and modes should identify alternative solutions and compare and contrast these alternatives with respect to the security goals. The short report documenting the lab should be no longer than 5-8 pages.

You should also upload a .zip file of you updated Chat Application containing the (security) enriched code so that it can be run in the same way as described above. You application will them be examined and tested. Your .zip file should be GroupID_Lab1.zip where you can find your group id from the respective excel file on DTU Inside. Your code should also be well commented: You should always put your documentation comments inside the comment delimeters /** ... */ with one comment per class, interface, method and data structure.

https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

5. References

https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#Description
https://security.stackexchange.com/questions/45963/diffie-hellman-key-exchange-in-plain-english
https://www.itcsolutions.eu/2011/08/22/how-to-use-bouncy-castle-cryptographic-api-in-netbeans-or-eclipse-for-java-jse-projects/