APCM Exam Project : Green Chat

Valentina Astore
Giacomo Fregona

13 January 2023

1 Description of the chat

Interface description

The visual representation of our chat has been divided into two main stages: the welcome stage and the chat stage. The welcome stage allows the user to enter the needed parameters to fully complete the login procedure. He will be asked to provide an ID, whose validity is checked by controlling that the given ID empty string and that it is not already in use.
When this procedure is completed, the user is redirected to the chat stage. The stage is organized as follows:

- a title, displaying the user's ID;
- a *Quit* `Button` that the user can use to logout;
- a `ScrollPane`, containing a `VBox` where the messages of the currently displayed chat are shown. An indicator of the current chat can be seen in the right column;
- a *Group chat* `Button`, which is visible unless we are displaying that chat. Whenever it is clicked, it shows the group chat messagaes in the `ScrollPane` by updating its `VBox`;
- a *Show users* `Button`, that opens a new stage when it is clicked. The new stage contains the list of the online users represented by `Label`s containing their IDs, organized in a `TilePane`. The user can decide to click on a `Label` to visualize its private chat with that participant in the chat stage;
- a *Refresh* `Button`, that displays eventual notifications.
  If the group chat is displayed, the new group messages are added to the `ScrollPane` along with the notifications of new users' accesses. Moreover, if private messages have been received, an highlighted border will appear around the *Show users* `Button`. If clicked, then it will show which participants have shared something by hihlighting their corresponding `Label`s.
  If a private chat is displayed, its new notifications are shown in the `ScrollPane` and both the *Group chat* and *Show users* `Button`s will be highlighted if needed;

- a `TextArea`, where the user can write his messages;
- a *Send* `Button`, with which the user can send his message to same participant as that whose chat is currently displayed, or to the group chat accordingly.

Communication structure description

Our chat allows different users, whose devices are denoted as *Clients*, to securely communicate with each other by exploiting a third trusted entity, denoted as *Server*. Each Client gets in touch with the Server through a socket. In particular, the connection is established when the Client sends a connection request on the gate 2023 of the Server. After that, the Server initializes an object of the class `ServerThread` running in an independent thread and dedicated to the specific Client.
In order to start the secure message flow between Client and Server, they exchange a secret AES128 key. To do that, the Server needs a pair of private and public keys, so to share the public key with each user who wants to connect. The key generation procedure constructs them and returns two files, *PrivateKey.key* and *PublicKey.key*, which contain the information needed to retrieve the corresponding keys. Of course the PrivateKey.key must be secretly kept by the Server. We refer to the next section about further details regarding the asymmetric scheme which we have chosen to adopt for this phase.
At this point, any further communication between Client and Server will start from the Client. This setting turned out to be fundamental to ensure that the encryption and decryption parameters of the two entities remain coordinated any time. The encryption mode that we have chosen to use, described in the next section, requires that Server and Client cannot encrypt or decrypt messages at the same time and therefore we decided to ensure that the Server will not do anything until it is asked for. The Client, then, can either ask for information or it can spread a new message to be delivered to other users. In the case of an information request, the `Update` function of the homonymous class will be invoked. Such function delivers the notification query to the Server and processes the received updates. In order to make this system work, we have decided to build an encoding procedure to both label the type of notification which is received and the Client's actions towards the Server. To do so, a further byte is added at the beginning of each shared message, where such encoded value will be stored. Analogously, the server will attach the encoding byte at the beginning of each information forwarded to the Client to indicate the type of the message's content. The encoding convention is displayed in the table below.

Client		Server	
Encoding Byte	Action	Encoding Byte	Notification Type
$0x00$	Access	$0x00$	New user access
$0x01$	Update		
$0x02$	Send group message	$0x02$	Group message received
$0x03$	Send private message	$0x03$	Private message received
$0xFF$	Logout	$0xFF$	A user logged-out

Whenever a new user enters the chat the `Update` function will return this event to the Client that sent the refresh request. Consequently, the chat's participants list will be updated. In our implementation, we decide to store two distinct lists in the Server and in the Client. In the first case, each participant is identified by an array of bytes, which is the byte representation of the `String` name with which that user has logged-in. On the contrary, in the Client an `ArrayList` of text `Label`s is stored, allowing to depict the list in the Client's interface fast. This list is update whenever the Client's user asks for an update or he asks to see the current participants.

2 Security

AES choices

In order to guarantee the secrecy of the communication between client and server, the shared messages in our chat are encrypted using AES128. A crucial point in our work has been choosing the most suitable usability mode for this cipher according to our needs. Our main goal was finding a fast encryption modality, for which the AES128 security level was guaranteed. We have immediately thought about the so-called CFB mode, which we display in the schemes below.
As the previous scheme shows, this mode is based on the use of an initialization vector (iv) and one of its main features is that it can be employed for encrypting long files. In order to adapt this mode for our needs, we had to understand how to update the iv for each new message that is exchanged, remembering that each iv must be used only once with the same key, otherwise some information could leak. Indeed, encrypting two distinct messages m_1 and m_2 with the same key and iv obtaining the ciphertexts c_1 and c_2, would allow an attacker to easily derive the first block of $m_1 \oplus m_2$ by simply computing the first block of $c_1 \oplus c_2$. In the context of a Choosen Plaintext Attack, an attacker could further iterate this procedure and exploit it to retrieve the overall message.
Bearing in mind this awareness, we came up with the idea of choosing the iv so that the overall encryption procedure acts as if the various messages that are exchanged were a unique long text. More precisely, in order to achieve this feature, we employ a *current vector* (cv) of 16 bytes, which will take place of the iv from the encryption of the second message on. At the end of the encryption of each new message, the cv is updated using the last block of ciphertext which has just been obtained. In this way, the next message is encrypted in the same way as if it was attached to the previous one.
Clearly this encryption modality requires Server and Client to be synchronized. This is the reason why an exchange of information between the two parties is always triggered by the Client, even in the case of a new that message must be delivered to it.
Another issue we needed to deal with is that the AES cipher can be used only for messages that are multiples of 16 bytes long. Consequentially, we designed a simple padding procedure to adapt our messages length. Each message exchaged trought the channel is padded in the following way: $l \,|\, r \,|\, m$ where:

- l is the three bytes representation of the message length,

- r is an array of pseudo random bytes returned by an instance of the *SecureRandom* Java class with suitable length,
- m is the byte representation of the message

and | denotes byte arrays concatenation.
It is worthwhile to highlight that among AES128's features, it has been crucial for us the fact that it provides *Indistinguishability under chosen-plaintext attack (IND-CPA)*. Indeed, as depicted above, when a message is sent through our chat, the real content of the shared information includes 3 bytes for the message's length in the beginning. Due to this fact, it is highly probable that the first byte or even first two bytes of the shared plaintext are equal to $0x00$. The IND-CPA property, then, ensures that an attacker will not be able to gain any further information to decrypt the shared ciphertext by only exploiting this characteristics of our cipher

The AES related Java methods are collected in the `AESCipher` class. In order to exploit this code for encryption and decryption, an object belonging to this class must be initialized and the method `.Setup()` must be invoked. In particular, the session key is passed in input to the method and it will be associated to the instance of the cipher. The principal methods of the `AESCipher` class are native and written in C. They communicate with the Java code through the *Java Native Interface*. The *cv* array and the key are grouped in a structure and stored in an array by the C procedures. In particular, each `AEScipher` instance running on the machine is associated with an array index given by the Java constructor. The AES core has been written during the course lectures and its source code can be found in *AESLib.c*, that we have expanded including our implementation of the CFB mode of operation.

Key sharing choices

From the project specifications, Client and Server must agree on an AES session key. In our concept of the chat, everybody can enter the chat providing a username that will be used for the session, so there is no need for the user to authenticate and to associate its id with a physical identity. On the other side, we would like the Server to be authenticated, so that a user can trust it and be sure that the sent messages are properly redirected to the desired recipients. In this context, we discarded bidirectional authentication protocols and considered Key Encapsulation Mechanisms which represent a better fit for our purpose. Hence the Server will be associated to a pair of private and public keys, that in our case are stored in the two *.key* files. The Client will generate the AES session key and send its encapsulation across the channel as the first step of the connection protocol. The Server can then retrieve the AES secret using the private key. From that moment on, the AES key is shared between the two parties and it will be used to encrypt any further communication.
Regarding the particular key sharing scheme, we were allowed to choose it among the protocols made available by Bouncy Castle or Oracle/Sun providers. We decided to use Bouncy Castle that we had already met during the course lectures. In our

code, we work with Bouncy Castle 1.72 which is currently the latest release [1].
The implemented KEMs in the chosen provider are all candidates for the NIST Post Quantum standardization process. Among them, we have been looking for a cryptographic primitive that matches the same security level provided by our symmetric encryption. In particular, the NIST competition winner CRYSTALS-Kyber stood out among the other algorithms. The version Kyber-512 ”aims at security roughly equivalent to AES-128” [2], therefore we considered it to be a suitable choice for our purpose.
The Java functions used by Client and Server to execute the KEM are static methods of the file `KeyGeneration` and they are called respectively `AESencapsulation` and `AESdecapsulation`.

Final considerations

During the implementation of our chat we understood that some optimizations are still possible, more significantly in the case where a lot of users join the chat. For instance, the Server and Client storage structures could be better organized (e.g., passing from lists to ordered lists or binary graphs) to speed up the research phase. Other optimizations may regard the bound between the graphical interface and the communication on the Client's end, that in our work are mostly considered as separate issues but that could be partially merged to gain some upgrades.
At the same time, we feel like our chat performance is satisfying for our purposes and it matches the project specifications.

References

[1] *Bouncy Castle Releases Page.* URL: `https://www.bouncycastle.org/latest_releases.html` (visited on 01/13/2023).

[2] *CRYSTALS Kyber Page.* URL: `https://pq-crystals.org/kyber/index.shtml` (visited on 01/13/2023).

[3] *Kyber Bouncy Castle Page.* URL: `https://www.bouncycastle.org/docs/docs1.8on/index.html` (visited on 01/13/2023).