

# A34 MessagIST Project Report

## Team

Number	Name	User	E-mail
ist1112265	Daniela Camarinha	<a href="https://github.com/DanielaDoesCode">https://github.com/DanielaDoesCode</a>	<a href="mailto:daniela.camarinha@tecnico.ulisboa.pt">mailto:daniela.camarinha@tecnico.ulisboa.pt</a>
ist1104195	Sofia Du	<a href="https://github.com/SofiDu">https://github.com/SofiDu</a>	<a href="mailto:sofia.du@tecnico.ulisboa.pt">mailto:sofia.du@tecnico.ulisboa.pt</a>
ist1103793	Tomás Gouveia	<a href="https://github.com/tomas7770">https://github.com/tomas7770</a>	<a href="mailto:tomas.gouveia@tecnico.ulisboa.pt">mailto:tomas.gouveia@tecnico.ulisboa.pt</a>

## Index

- 1. [Introduction](#)
  - 1.1. [Main Components](#)
  - 1.2. [Project Architecture](#)
    - 1.2.1. [Server](#)
    - 1.2.2. [Database](#)
    - 1.2.3. [Client](#)
    - 1.2.4. [Message](#)
    - 1.2.5. [Operation Codes \(Opcode\)](#)
- 2. [Project Development](#)
  - 2.1. [Secure Document Format](#)
    - 2.1.1. [Design](#)
    - 2.1.2. [Implementation](#)
    - 2.1.3. [Public/Private key generation and exchange](#)
  - 2.2. [Security Challenge](#)
    - 2.2.1. [Challenge Overview](#)
    - 2.2.2. [Attacker Model](#)
- 3. [Conclusion](#)
- 4. [Bibliography](#)

## 1. Introduction

A messaging software called **MessagIST** was designed for IST students to communicate effectively and securely within the IST community. One of the primary goals of this application is to ensure privacy and security in every communication between users, while complying with GDPR regulations. The application must safeguard these messages from unauthorized access and tampering, ensuring that only the intended sender and receiver can access the content.

To ensure the **confidentiality**, **integrity**, **authentication** and **availability** of the messages exchanged through the platform, we need to design and implement robust security mechanisms alongside a distributed system within our application.

### 1.1. Main Components

- **Secure Documents**: a cryptographic library used to protect the documents (messages);

- **Infrastructure**: a virtual environment consisting of networks and machines, configured with adequate firewall rules, and running application and database servers.
- **Security Challenges**: a point to point encryption mechanism, **Security Challenge A**.

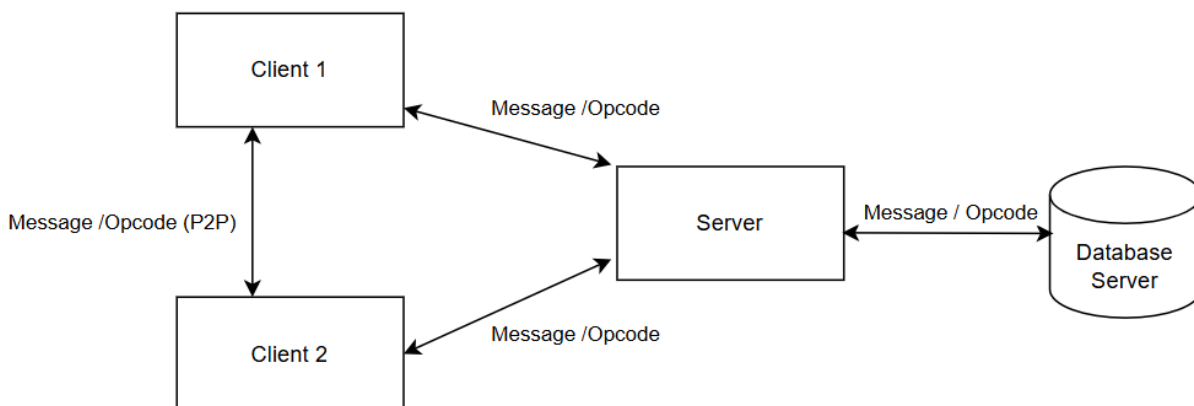
## 1.2. Project Architecture

This project combines a **client-server** architecture with a secure communication channel that enables **peer-to-peer (P2P)** interactions. The **server** acts as a central hub for data storage, messages exchange and coordination, while the **secure channel** ensures direct and encrypted communication between clients.

All communication is done over **TCP** sockets. **Client-server** and **server-database** communication use **SSL**. Each component (server, database, and client) maintains its own dedicated **Keystore** and **Truststore**, which are loaded and initialized to configure **SSL** settings for secure communication.

- **Keystore**: a storage mechanism used to store cryptographic keys and certificates.
- **Truststore**: a storage used to store trusted certificates, including root certificates and other public certificates, that a system uses to verify the authenticity of remote servers or clients.

Direct communication between clients does not use **SSL**, so that clients don't need to have their own certificates. However, message contents are still encrypted, and a **token** generated by the server provides authentication for clients. See the **Security Challenge** section below for more details.



### 1.2.1. Server

The server is a **trusted** and **multi-threaded** system that leverages **TLS** to ensure secure **client-server** and **server-database** communication.

- Uses the **SSLServerSocket**, (serverSocket), to accept client connections, spawning a new **MessageHandlerThread** for each client.
- The **MessageHandlerThread** class authenticates user credentials (username and password), utilizing a **salt** generated when the user is created for the first time, to protect against **rainbow table attacks**.
- The **SessionManager** class tracks connected users and their sockets. It allows checking user statuses (online/offline) and retrieving the client IP for peer-to-peer communication.
- The **Session** class handles each client connection and manages database server requests using defined operation codes.

### 1.2.2. Database

The database used in this project is also **trusted** and is an **in-memory system** that should be replicated across multiple machines for improved reliability and availability (note that the provided implementation does not implement replication, relying on a single database server instead).

To ensure the integrity and persistence of data, a **scheduled backup service** is employed. The backup process is managed using a **ScheduledExecutorService** that performs periodic backups of the entire database. Every 14 days, the service triggers a backup operation, storing a copy of the database to the **./backup** directory. This guarantees that all data stored in the database is periodically backed up and can be restored if necessary. The database is exclusively responsible for handling **server-side requests** and managing data storage.

- **messages** Table: stores all the messages exchanged between clients.

Column Name	Data Type	Description
<b>content</b>	TEXT	The message content (in JSON format, encrypted).
<b>sender</b>	TEXT	The name of the client sending the message.
<b>receiver</b>	TEXT	The name of the client receiving the message.
<b>timestamp</b>	TIMESTAMP	The timestamp when the message was sent.

- **clients** Table: stores each client's authentication and cryptographic information.

Column Name	Data Type	Description
<b>name</b>	TEXT	The username of the client (unique).
<b>password</b>	TEXT	The hashed password with salt associated with the client.
<b>salt</b>	TEXT	The salt added to the hashed password.
<b>contacts</b>	TEXT	A string representing the list of contacts for the client.
<b>pubkey</b>	TEXT	The client's public key for encryption purposes.
<b>privkey</b>	TEXT	The client's private key, encrypted with a secret key generated with the recovery password.

1.2.3. Client

The client interface is console-based, offering options to interact with our system.

Command	Description
<b>a</b>	Add contact.
<b>s</b>	Send message in client-server mode.
<b>e</b>	Send message in peer-to-peer mode.
<b>q</b>	Quit.
<b>h</b>	Help.

Each client's username is its own IST ID number (which must start with `ist` followed by numbers), and the user must choose a strong passwords (one for authentication and one for recovering the authenticator). When users want to start a communication with others (`s` or `e`), they must first add the client to their contact list using the `a` command.

1.2.4. Message

The Message class is responsible for creating and serializing `messages`. Each message is represented as a `JsonObject` and the content field contains an encrypted message.

1.2.5. Operation Codes (Opcode)

`Opcodes` are the API of the system, specifying how clients, server, and database communicate with each other. Opcodes are sent through TCP sockets as follows:

`OPCODE PARAMETER1 PARAMETER2 ...`

Sometimes one or more return values may be sent back.

When a client first connects to the server, the server expects `username` and `hashedPassword` without the need of an opcode.

COMMON

Opcode	Parameters	Return	Description
OK	N/A	N/A	Operation successful
ERROR	N/A	N/A	Operation failed

AUTHENTICATION

Opcode	Parameters	Return	Description
REGISTER_SUCCESS	N/A	N/A	Register successful
RETURNING_USER	N/A	N/A	Login successful
INVALID_CREDENTIALS	N/A	N/A	Invalid credentials trying to login
ERROR_REGISTERING	N/A	N/A	Error registering user
NO_USERS	N/A	N/A	Current user has no contacts, or no contacts are available to add
QUIT	N/A	N/A	Logout

CONTACTS

Opcode	Parameters	Return	Description
ADD_CONTACT	<code>contact_username</code>	OK/ERROR	Add a new contact

Opcode	Parameters	Return	Description
GET_POSSIBLE_CONTACTS	N/A	contacts_list/NO_USERS	Get list of possible contacts to add

MESSAGES

Opcode	Parameters	Return	Description
SEND_MESSAGE	message_json	OK/ERROR	Send a message
SEND_E2E_MESSAGE	contact	(contact_ip auth_token)/ERROR	Establish a peer-to-peer connection for end-to-end encrypted messages
GET_MESSAGES	contact_username	message_count message_list	Retrieve all messages between the current user and another user

E2E

Opcode	Parameters	Return	Description
VALIDATE_TOKEN	auth_token	VALID_TOKEN/INVALID_TOKEN	Request server to validate auth token
INVALID_TOKEN	N/A	N/A	Token is invalid
VALID_TOKEN	N/A	N/A	Token is valid

DATABASE

Opcode	Parameters	Return	Description
REGISTER	username saltedPassword salt	REGISTER_SUCCESS	Register a new user
CHECK_CREDENTIALS	username hashedPassword	true/false	Check user credentials
GET_USERS	N/A	users_list	Retrieve a list of all users
GET_CONTACTS	username	contacts_list/NO_USERS	Retrieve the contacts list of a user
UPDATE_CONTACTS	username contacts_list	N/A	Update the contacts list of a user

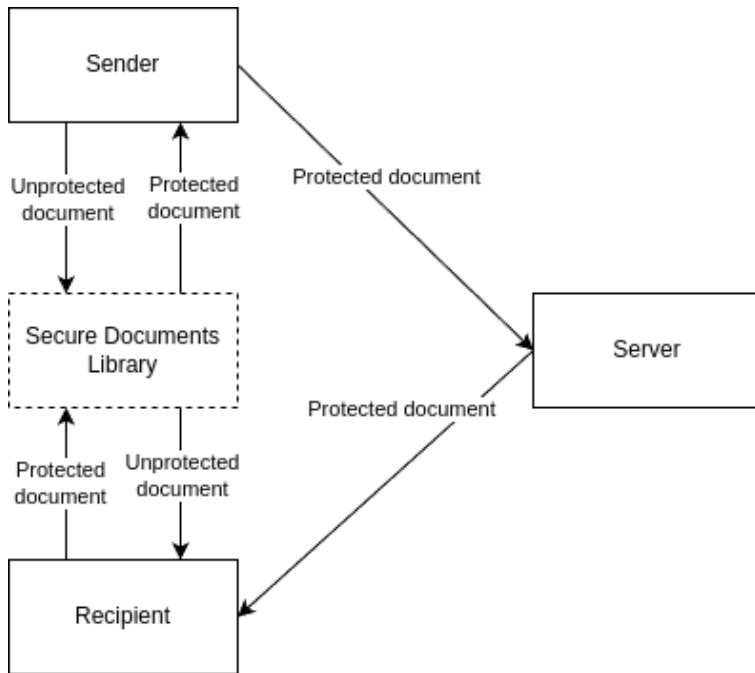
Opcode	Parameters	Return	Description
GET_MESSAGES_FROM_USER_TO_USER	username1 username2	message_count message_list	Retrieve all messages between two users
PUT_MESSAGE	message_json	true/false	Insert a message into the database
CHECK_USER	username	true/false	Check if a user exists
GET_SALT	username	salt	Retrieve the salt for a user
UPDATE_PUBKEY	username pubkey	N/A	Update the user's public key
GET_PUBKEY	username	pubKey	Retrieve the public key of a user
UPDATE_PRIVKEY	username privKey	N/A	Update the user's private key
GET_PRIVKEY	username	privKey	Retrieve the private key of a user

## 2. Project Development

### 2.1. Secure Document Format

#### 2.1.1. Design

To implement encryption and prevent the server from being able to read messages, a library that provides cryptographic security to messages ("documents") is used.



When a client wants to send a message, it starts by protecting it, then sends it through the server (or directly to the recipient, if the peer-to-peer option is used), and the recipient unprotects it.

The library provides the following operations:

- `protect(PublicKey recipientKey, PrivateKey senderPrivateKey, PublicKey senderPublicKey)` - Adds security to a document.
- `check(PublicKey senderKey)` - Verifies if the message was written by the sender and not tampered by someone else (returns `true` if the check passes, `false` otherwise).
- `unprotect(PrivateKey recipientKey, PublicKey senderKey, boolean useSenderKey)` - Removes security from a document. A validity check is also performed (similar to the `check` operation), and the operation fails if the check fails. If `useSenderKey` is `true`, the `recipientKey` is assumed to be the sender's private key. This is used when the message is read by its sender.

Message protection is provided by encrypting its contents with an **AES secret key** only known by the sender and recipient. This ensures that only these clients can read the contents (**SR1**).

Each message has its own secret key, randomly generated upon protecting it, to prevent contents of other messages from being compromised if a single message's key is compromised. Because of this, secret keys are sent and stored along with the message. They are encrypted using the recipient's public key, so that only the recipient can decrypt them (**SR4**), and therefore decrypt the message. A copy of the key encrypted with the sender's public key is also stored, to allow the sender to read messages they sent.

In addition, the encrypted message is **digitally signed** with the sender's private key (using the `SHA256withRSA` algorithm), verifying that the received message was indeed written by the sender (**SR2**).

The format of an unprotected document is as follows (JSON):

```

{
  "message": {
    "sender": "ist1123123",
    "receiver": "ist1321564",
    "timestamp": "2022-01-01T12:00:00Z",
  }
}
  
```

```
    "content": "Hi! do you know the solution for the SIRS exercise?"
  }
}
```

And a protected document:

```
{
  "message": {
    "sender": "ist1123123",
    "receiver": "ist1321564",
    "timestamp": "2022-01-01T12:00:00Z",
    "content": "MIQfb5dLFkroJnfeP2lMrp80hegs...",
    "keyForReceiver": "eQr+qAtLjZcdusC2Qd3Y/...",
    "keyForSender": "ldtZhtjVpwReY351mffAav/...",
    "signature": "YRsoxCAS9RrLSFcYJAu/..."
  }
}
```

Note that the encrypted content, keys, and signature are encoded in Base64 to facilitate storing them in a text format.

### 2.1.2. Implementation

The Secure Document library is implemented in Java, in a single class `SecureDocument`. Java's built-in `java.security` and `javax.crypto` are used for cryptography. Google `GSON` is used for reading and modifying messages in JSON.

A secure document is created using the `SecureDocument(String message)` constructor, where `message` is a message in the format specified above, unprotected or protected. The resulting secure document object can then be used to protect, check, or unprotect messages, by calling the respective methods. After protecting/unprotecting, the modified message can be obtained by calling `getMessage()`, which returns a `String` of the message in JSON.

A command-line tool known as "Secure Document Tool" is included with the project to easily test interaction with the library. It takes message JSON files as input, and outputs the modified message after protecting/unprotecting, or the result of a `check` operation. More detailed instructions are included in a help message within the tool.

### 2.1.3. Public/Private key generation and exchange

Although generation of clients' public/private keys and their exchange is not part of the Secure Document library, it's required for it to be properly used.

When logging in for the first time, the user is prompted to enter a recovery password. Using `PBKDF2WithHmacSHA256`, a secret key is derived from this password and the user salt. A random `RSA` key pair is also generated. The private key is encrypted with the secret key, and the key pair is sent to the server to be stored. For a given password, the secret key is always the same, so that the user can recover messages (**SRA3**). Although storing the private key in the server (even encrypted) weakens security, it improves usability for this application. Originally, we considered generating an `RSA` key pair directly from the password, however we found out it wasn't feasible with the libraries used.



Whenever a client needs to obtain the public key of another client, they simply need to retrieve it from the server, which we assume to be trustworthy enough to authenticate users.

## 2.2. Security Challenge

### 2.2.1. Challenge Overview

#### Challenge A - security requirements

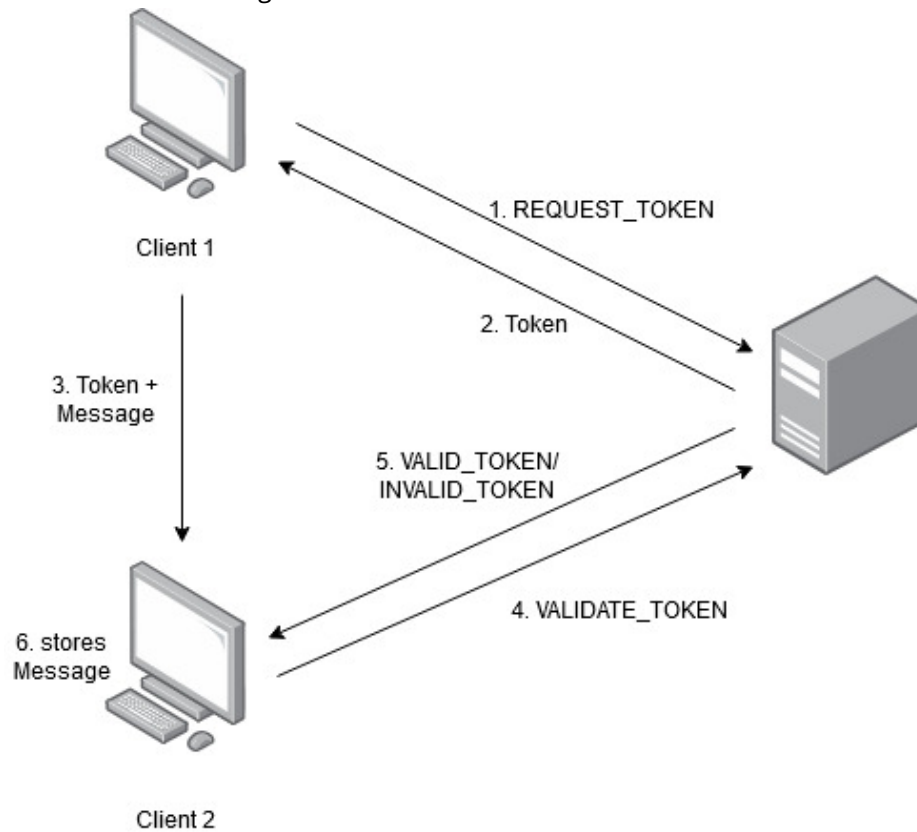
- **[SRA1: Confidentiality]** : This was solved by using the `SecureDocument` class (Explained above).
- **[SRA2: Confidentiality]** : For this requirement, we assumed that a side channel would be a new connection directly between two clients. Neither of the clients are authenticated by a CA (Certification Authority) so without any more mechanisms we cannot for sure now if they are to be trusted. To solve this we decide to implemented a `Token Service` System: Whenever a clients wants to send a `E2E Connection` (End-to-End), it requests a Token from the Server, which is the trusted authority in the system, serving the server as a broker of Tokens. These are in the format of `JWT` - JSON Web Tokens. And these have the goal of asserting claims, in our case we want the receiving client to know the sender is an authenticated user of the MessagIST system. To do this, the sender requests a Token to the server, and the sender sends that token, alongside the actual message to the receiver. After receiving the token, the receiver sends it to the server for validation. If the token is valid the server validates it and the receiver stores the message sent by the sender.

The Token has the following properties:

```
return Jwts.builder()
    .setSubject("E2E-connection")
    .claim("sender", sender)
    .claim("receiver", receiver)
    .setIssuedAt(new Date())
    .setExpiration(new Date(System.currentTimeMillis() + 60000))
// 1 min expiry
    .signWith(config.getJWT_KEY())
    .compact();
```

-- Where the claims `sender` and `receiver` are their respective IDs. -- The `.setExpiration` is set to deal with replay attack, by setting a time of validity of the Token, it shortens the window of vulnerability of the system, so that attackers are not able to use the Token and pretend to be an authorized communication entity. -- The token is then signed with a Key from the Server, and this is what makes it possible for the server to validate the token. To accept peer-to-peer connections we also implemented the following classes: `E2ECommunicationHandler.java`, `E2EConnectioListener.java`. And to store the messages: `ClientLocalStorage.java`

Flow of execution diagram:



- **[SRA3: Availability]** : For this requirement, we are only storing data locally for the E2E communication (from the last challenge point). And this is done through a local instance of the database that stores the message as they are sent out and as they arrive. Since it is a Peer-to-Peer type of communication with an authenticator entity (the server) we don't see much sense in having a recover mechanism for those messages. We have, however, implemented a **PasswordDerivationService**, that generates a key from a password (written by the client on login), This key is used to get the message history of the client and without it the client cannot retrieve it.

### 2.2.2. Attacker Model

#### Trusted Authorities

- **MessageIST Server** The Server has its own certificate signed by a CA (Certification Authority) generated by the **generate\_certs\_stores** script. Its certificate is put into every client's TrustStore, making it a trusted entity by all clients in the network.
- **Database Server** The database is also a trusted entity, but it's only connected directly to the Server, and so the server is the only one that has the database certificate in its TrustStore.

#### Partially Trusted Authorities

In a Peer-to-Peer connection between clients, none of them are trusted unless they have successfully sent out a valid token, and they only become trusted for the purpose of accepting their message. After that, the client who sends the message has to generate another token and repeat the process to become trusted by the receiver again. Hence, it being Partially Trusted.

#### Untrusted Authorities

Any machine that sends out an invalid token or that is not part of the permitted IPs by the iptable Firewall rules.

### Attacker Power

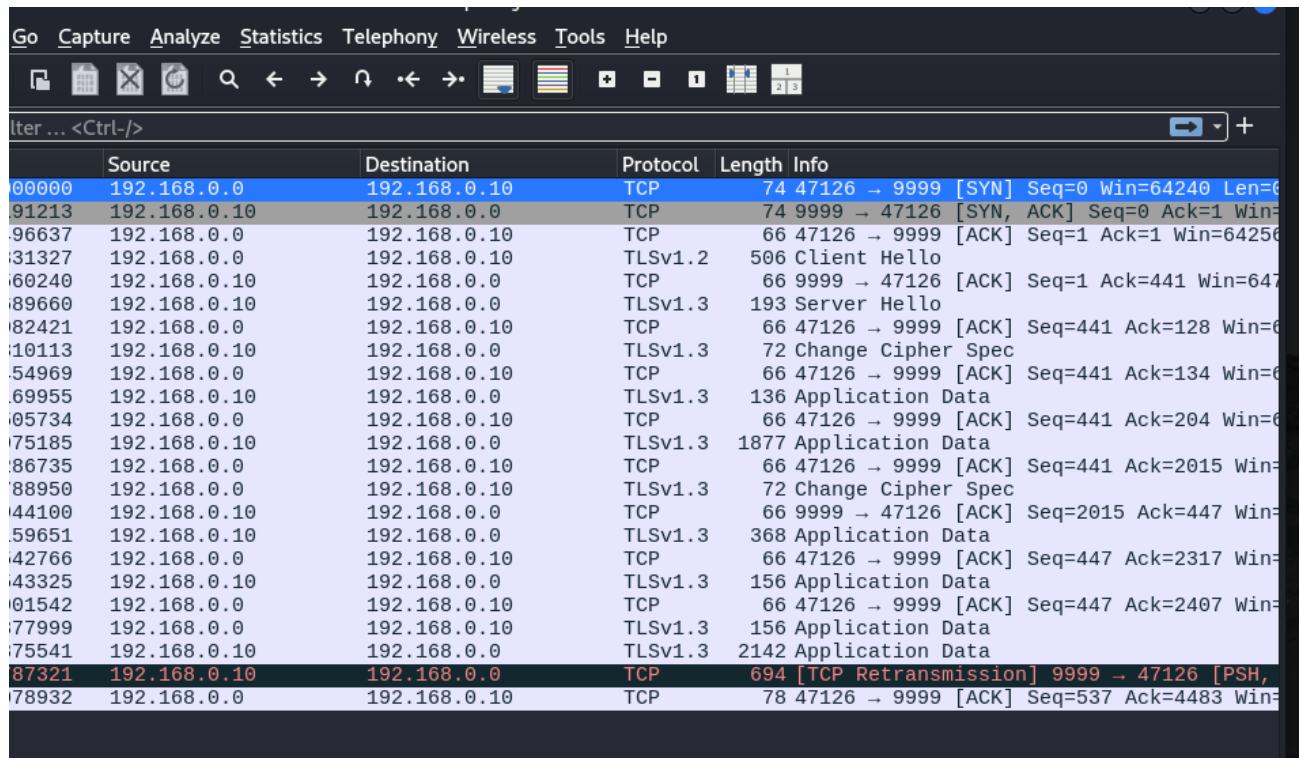
- **Things the attacker know how to do:** We assume the attacker know how to use Wireshark to try to snoop packets. That they also know how to setup a that can connect to the network and try to connect to our system.
- **Things the attacker does not know how to do:** We assume the attack does not have access to our system directly, that he is not capable of interfering with its runtime and therefore not be able to see the flow o execution nor the in-memory data that is being used by the database.

### Possible attacking scenarios

-- Wireshark on VM4 we turned on an instance of Wireshark to check if any attacker could see the contents of the message we were sending and the login process.

### Login Process

- A clients starts its instance and the SSL handshake starts:



No.	Source	Destination	Protocol	Length	Info
000000	192.168.0.0	192.168.0.10	TCP	74	47126 → 9999 [SYN] Seq=0 Win=64240 Len=0
000001	192.168.0.10	192.168.0.0	TCP	74	9999 → 47126 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0
000002	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=1 Ack=1 Win=64256 Len=0
000003	192.168.0.0	192.168.0.10	TLSv1.2	506	Client Hello
000004	192.168.0.10	192.168.0.0	TCP	66	9999 → 47126 [ACK] Seq=1 Ack=441 Win=64256 Len=0
000005	192.168.0.10	192.168.0.0	TLSv1.3	193	Server Hello
000006	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=441 Ack=128 Win=64256 Len=0
000007	192.168.0.10	192.168.0.0	TLSv1.3	72	Change Cipher Spec
000008	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=441 Ack=134 Win=64256 Len=0
000009	192.168.0.10	192.168.0.0	TLSv1.3	136	Application Data
000010	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=441 Ack=204 Win=64256 Len=0
000011	192.168.0.10	192.168.0.0	TLSv1.3	1877	Application Data
000012	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=441 Ack=2015 Win=64256 Len=0
000013	192.168.0.10	192.168.0.0	TLSv1.3	72	Change Cipher Spec
000014	192.168.0.10	192.168.0.0	TCP	66	9999 → 47126 [ACK] Seq=2015 Ack=447 Win=64256 Len=0
000015	192.168.0.10	192.168.0.0	TLSv1.3	368	Application Data
000016	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=447 Ack=2317 Win=64256 Len=0
000017	192.168.0.10	192.168.0.0	TLSv1.3	156	Application Data
000018	192.168.0.0	192.168.0.10	TCP	66	47126 → 9999 [ACK] Seq=447 Ack=2407 Win=64256 Len=0
000019	192.168.0.10	192.168.0.0	TLSv1.3	156	Application Data
000020	192.168.0.10	192.168.0.0	TLSv1.3	2142	Application Data
000021	192.168.0.10	192.168.0.0	TCP	694	[TCP Retransmission] 9999 → 47126 [PSH, Seq=537 Ack=4483 Win=64256 Len=0
000022	192.168.0.0	192.168.0.10	TCP	78	47126 → 9999 [ACK] Seq=537 Ack=4483 Win=64256 Len=0

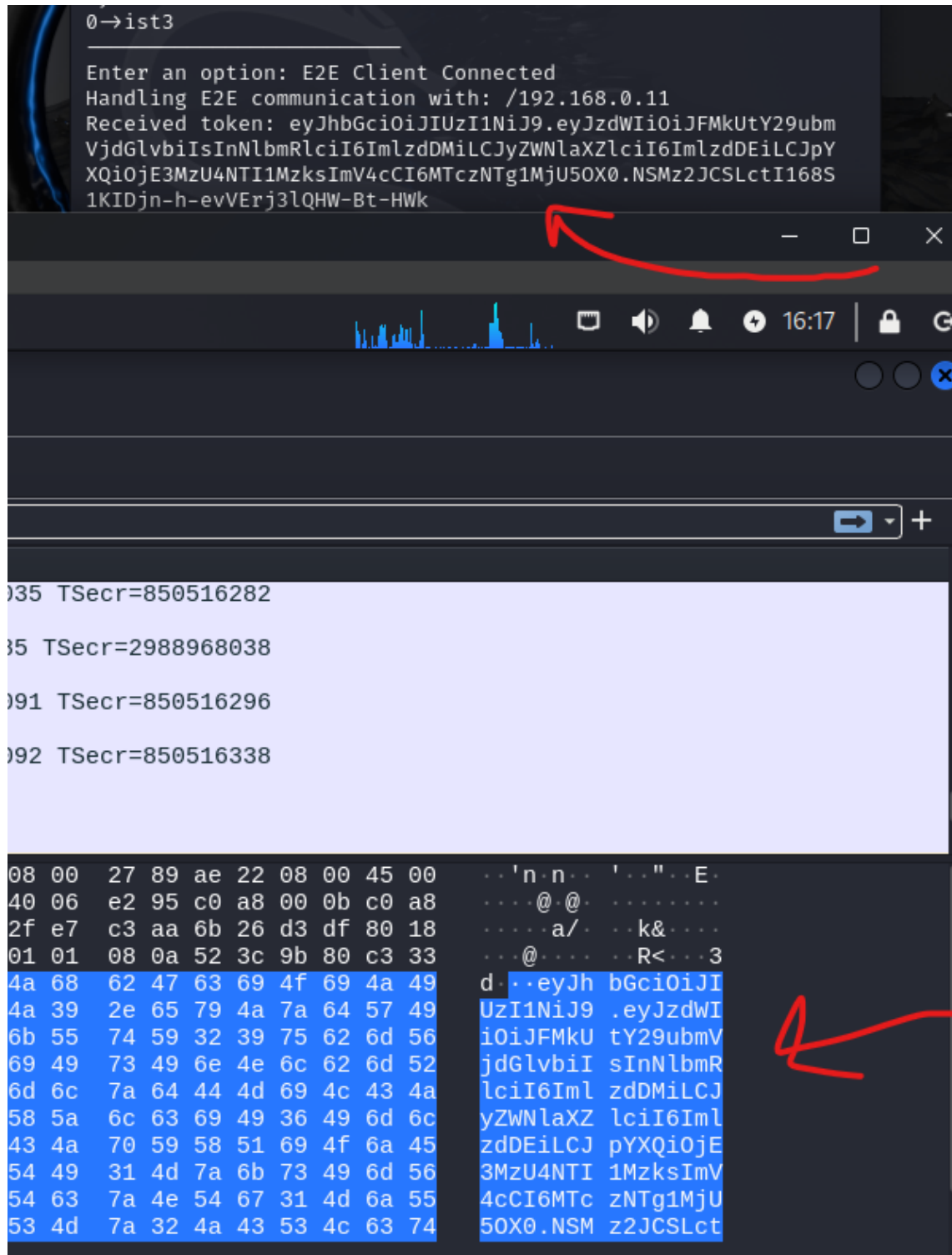
- The client types its username - we can see that since it's a TLS connection, the payload is encrypted:

The image shows a Wireshark packet capture of a TLS connection. The top pane displays a list of packets, with packet 24 highlighted. The middle pane shows the details of packet 24, which is a TLSv1.3 record. The 'Application Data' field is expanded, showing the encrypted payload. A red arrow points to the 'Application Data' field in the details pane. The bottom pane shows the packet bytes in hexadecimal and ASCII. The status bar at the bottom indicates 'eth0: <live capture in progress>' and 'Packets: 31'.

- Another 2nd client connects and disconnects:

The image shows a Wireshark packet capture of a second client connecting and disconnecting. The top pane displays a list of packets, with packet 93 highlighted. The middle pane shows the details of packet 93, which is a TCP reset (RST) packet. The 'Application Data' field is expanded, showing the encrypted payload. A red arrow points to the 'Application Data' field in the details pane. The bottom pane shows the packet bytes in hexadecimal and ASCII. The status bar at the bottom indicates 'eth0: <live capture in progress>' and 'Packets: 31'.

- On a E2E Communication we can see the clear Token (since this is not a secure connection):



```
0→ist3
Enter an option: E2E Client Connected
Handling E2E communication with: /192.168.0.11
Received token: eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJFMkUtY29ubmVjdGlvbiIsInNlbnRlciI6ImlzdDMiLCJyZWNaXZlciI6ImlzdDEiLCJpYXQiojE3MzU4NTI1MzksImV4cCI6MTczNTg1MjU5OX0.NSMz2JCSLctI168S1KIDjn-h-evVERj3lQHW-Bt-HWk

035 TSecr=850516282
035 TSecr=2988968038
091 TSecr=850516296
092 TSecr=850516338

08 00 27 89 ae 22 08 00 45 00  ..'n.n.. '..."..E.
40 06 e2 95 c0 a8 00 0b c0 a8  ....@.@.....
2f e7 c3 aa 6b 26 d3 df 80 18  ....a/. ..k&....
01 01 08 0a 52 3c 9b 80 c3 33  ....@.....R<...3
4a 68 62 47 63 69 4f 69 4a 49  d...eyJh bGci0iJI
4a 39 2e 65 79 4a 7a 64 57 49  UzI1NiJ9 .eyJzdWI
6b 55 74 59 32 39 75 62 6d 56  i0iJFMkU tY29ubmV
69 49 73 49 6e 4e 6c 62 6d 52  jdGlvbiI sInNlbnR
6d 6c 7a 64 44 4d 69 4c 43 4a  lciI6Iml zdDMiLCJ
58 5a 6c 63 69 49 36 49 6d 6c  yZWNaXZ lciI6Iml
43 4a 70 59 58 51 69 4f 6a 45  zdDEiLCJ pYXQiojE
54 49 31 4d 7a 6b 73 49 6d 56  3MzU4NTI 1MzksImV
54 63 7a 4e 54 67 31 4d 6a 55  4cCI6MTc zNTg1MjU
53 4d 7a 32 4a 43 53 4c 63 74  5OX0.NSM z2JCSLct
```

- Nevertheless, when the client gets validated the content of the message is still encrypted:

```

66 9999 → 46136 [ACK] Seq=4694 Ack=3158 Win=62720 Len=0 TSval=2988968035 TS
198 Application Data
00 01 01 08 0a 52 3c 9b 80 c3 33 ...A.... ..R<...3
22 6d 65 73 73 61 67 65 22 3a 7b d. {"me ssage":{
69 76 65 72 22 3a 22 69 73 74 31 "receive r":"ist1
6d 65 73 74 61 6d 70 22 3a 22 32 ", "times tamp":"2
31 2d 30 32 54 31 36 3a 31 35 3a 025-01-0 2T16:15:
30 38 35 37 35 35 35 22 2c 22 63 39.53085 7555", "c
74 22 3a 22 63 4c 65 6f 6a 58 53 ontent": "cLeojXS
34 4d 74 4c 46 6c 62 74 67 51 3d 6PMiF4Mt LFlbtgQ=
65 6e 64 65 72 22 3a 22 69 73 74 =", "send er":"ist
65 79 46 6f 72 52 65 63 65 69 76 3", "keyF orReceiv
4d 54 76 4a 69 65 50 75 43 32 44 er": "MTv JiePuC2D
78 4d 73 6e 38 6d 4e 61 4f 57 42 CRbPlxMs n8mNa0WB
63 34 70 72 57 6a 58 6f 31 33 2f rMmqc4p rWjXo13/
51 45 31 30 42 4f 31 51 32 43 4d cLzlnQE1 0B01Q2CM
50 48 30 46 71 70 6a 63 50 62 31 7gZkQPH0 FqnpjcPb1
43 75 5a 53 6a 58 63 6b 37 45 74 wWbRyCuZ SjXck7Et
77 51 6a 37 4f 61 77 34 4a 38 4c DaTD7wQj 70aw4J8L
37 56 35 4c 4b 65 51 2f 66 39 48 f4wBe7V5 LKeQ/f9H
4e 53 4a 52 42 68 70 46 41 64 47 AdwRtNSJ RBhpFAdG
78 66 45 74 36 6e 63 59 53 4c 36 ClWaxxFE t6ncYSL6
34 53 35 4b 72 41 41 42 34 50 47 3NUVr4S5 KrAAB4PG
7a 64 48 4d 41 50 33 63 59 68 4a i+WlYzdH MAP3cYhJ
6e 2f 65 64 6f 63 33 48 37 68 57 hUlAdn/e doc3H7hW
34 30 72 46 4d 72 5a 77 42 48 69 7YA1R40r FMrZwBHi
47 73 6a 74 36 6a 43 30 4e 51 78 CRGV9GsJ t6jC0NQx
30 56 65 56 59 30 38 73 71 30 49 q0+dX0Ve VY08sq0I
4c 57 43 79 70 7a 6d 46 37 66 45 g8bW0LWC ypzmf7fE
4e 4c 50 4b 6f 34 56 67 4a 59 57 KII0QNLP Ko4VgJYW
75 45 43 47 44 4c 78 33 4f 54 4b DGj0nuEC GDlx30TK
31 76 6d 44 68 30 61 63 42 44 35 G6Sxz1vm Dh0acBD5
4d 62 37 75 6f 62 59 55 66 49 51 c0jZEMb7 uobYUfIQ
62 6e 35 5a 63 67 3d 3d 22 2c 22 z0FiSbn5 Zcg=="
72 53 65 6e 64 65 72 22 3a 22 54 keyForSe nder": "T
6f 67 65 46 6c 32 78 75 55 6f 2b 8JwrNoge Fl2xuUo+
61 75 54 50 78 4c 55 71 75 30 6a FdZz5auT PxLUqu0j
33 63 46 4f 4e 54 50 6e 46 73 2f rGddb3cF ONTPnFs/
44 4c 6e 4d 76 4b 33 6c 6b 55 7a zA90mDLn MvK3lkUz
35 62 4f 2b 44 32 5a 52 34 66 65 h2GyF5b0 +D2ZR4fe
2f 45 31 7a 2b 61 75 4c 64 61 54 kxG7+/E1 z+auLdaT
74 45 46 4a 72 42 63 4a 38 31 68 URWZxtEF JrBcJ81h
48 41 47 43 2b 47 48 37 42 41 76 XzGTYHAG C+GH7BAv
46 53 73 32 34 33 58 55 52 42 59 gdBfCFsS 243XURBY

```

## Firewall

To minimize the access of the malicious actors into our network we implemented **iptables** firewall rules, that are designed to only permit the minimum IPs possible for the system to function without failure. [Check them here](#)

## 3. Conclusion

To conclude, we were able to establish SSL/TLS connections safely between clients and server, server and database. We were able to securely encrypt the content of the messages with the use of the **Secure Document** tool, authenticate sender and receiver and check the integrity of the messages. We were able to establish peer-to-peer connections over an insecure network with the validation of an authenticated entity.



We were able to store keys and certificates safely with keyStores and trustStores

**Requirements fulfilled:**

- [SR1: Confidentiality] Only the sender and receiver of a message can see its content. **YES** - SecureDocument Class
- [SR2: Integrity 1] The receiver of a message can verify that the sender did wrote the message. **YES** - SecureDocument Class
- [SR3: Integrity 2] It must be possible to detect if there is a missing message or if they are out of order. **YES** - By using TLS under TCP, it allows us to assume no messages will be discarded
- [SR4: Authentication] Messages are only send to their authenticated recipients. **YES** - By securing that clients are authenticated through the TLS connection.

Security Challenge:

- [SRA1: Confidentiality] Only sender and receiver can see the content of the messages. **YES** Secure Document Class
- [SRA2: Confidentiality] There must be a protocol that allows two students to exchange a key (in a secure way). You can assume the existence of a side channel for this. **YES** TokenService and `E2ECommunicationListener` and `E2ECommunicationHandler`
- [SRA3: Availability] If a user loses their phone, they must be able to recover the message history. More or Less (there is no local message history for normal chats, but the recovery mechanism is there 😊)

**Possible enhancements:**

One of the possible enhancements would be the addition of the local message logs for each client of all of the message types. This would allow to minimize traffic to the server to receive the chat history everytime a user logs in or requests to send a message.

This project allowed us to develop our critical analysis of system regarding how secure they are, and broaden our knowledge of the different possibilities to make a system secure.

## 4. Bibliography

Alex Yu, System Design Interview,ByteByteGo, 2020 [here](#)

---

END OF REPORT