Network and Computer Security (SIRS)

Project Report - MessagIST

Group A07
ist1112191 - Xiting Wang
ist1112269 - Laura Cunha
ist1112270 - Rodrigo Correia

20/12/2024

1 Index

2 Introduction

2.1 Business Scenario: Messaging App (MessagIST)

MessagIST is an **instant messaging application** designed to ensure secure communication within the IST community. This application facilitates the secure exchange of text messages, with each message consisting of the following components:

- **Sender IST ID:** Identifies the sender.
- **Receiver IST ID:** Identifies the receiver.
- **Timestamp:** Indicates when the message was sent.
- **Content:** Contains the text of the message.
- **Counters:** Sent and receive message counters.

To meet the strict **protection needs** of secure communication, our implementation guarantees that all messages are confidential, authenticated, and maintain integrity. To achieve this, the application adheres to the following security requirements:

- **[SR1: Confidentiality]:** Only the sender and receiver can view the content of messages.
- **[SR2: Integrity 1]:** The receiver must verify the authenticity of the sender.
- **[SR3: Integrity 2]:** Missing or out-of-order messages must be detectable.
- **[SR4: Authentication]:** Messages must only be delivered to authenticated recipients.

These requirements, along with those related to the challenges outlined in **challenge A** (topic *"New Requirements"* on section 3.3), were carefully considered to ensure security.

2.2 Main Components

To address the business scenario effectively, the project is divided into three main components:

Secure Documents

A custom **cryptographic library** used by the clients and the server, that can also be run in an isolated way, designed to ensure the security of the messages (secure document), aligning with the defined protection needs and challenge needs.

The library contains methods to **protect**, **unprotect**, and **check** the integrity of files, as well as, methods to generate **symmetric and asymmetric keys**, **hashes of passwords**, and **encrypt/decrypt keys**.

As a brief overview, we used the **ChaCha20** stream cipher with **256-bit** keys in combination with the **Poly1305** hash function. The justification for choosing these algorithms is detailed in the *"Challenges Faced and Overcome"* subsection of section 3.1.2.

Infrastructure

The infrastructure focuses on the configuration of the necessary machines, including servers and environment, to support the client-server application, so this includes:

- Establishing a virtual environment with **virtual machines**, consisting of 2 clients, 1 server, and 1 database server, all isolated from the internet (*"Machines Setup"* in section 3.2.1).
- For the **networks**, the 4 machines are distributed along 2 networks (explained in detail on the *"Machines Setup"* in section 3.2.1).

- To secure the machines, **firewalls** were set up on the client, server, and PostgreSQL database machines. We also **secured communication** between all machines using the SSL/TLS protocol (*"Secure Communication"* in section 3.2.2).
- The infrastructure is built on a client-server application, incorporating all the design considerations outlined in (*"Built Infrastructure"* in section 3.2.1), the chosen technologies (*"Technologies Choice"* in section 3.2.1), and the security measures (discussed in section 3.2.2).

For more detailed information, check the section attached to each topic.

Security Challenge

For the security challenge, we decided to implement **challenge A** (section 3.3), which addresses advanced security requirements. This involves implementing a robust **end-to-end encryption mechanism** to meet the following confidentiality needs, as well as allow **availability** to the users:

- **[SRA1: Confidentiality]:** Ensures only the sender and receiver can read the message.
- **[SRA2: Confidentiality]:** Establish a secure protocol for key exchange between students, assuming the availability of a side channel.
- **[SRA3: Availability]:** Users must be able to recover message history even after losing their devices.

2.3 Structural Diagram

Below is the **UML structural deployment diagram** of our solution, illustrating all the **main concepts** and the **business scenario**. This architecture will be explained in detail in the section 3.2.

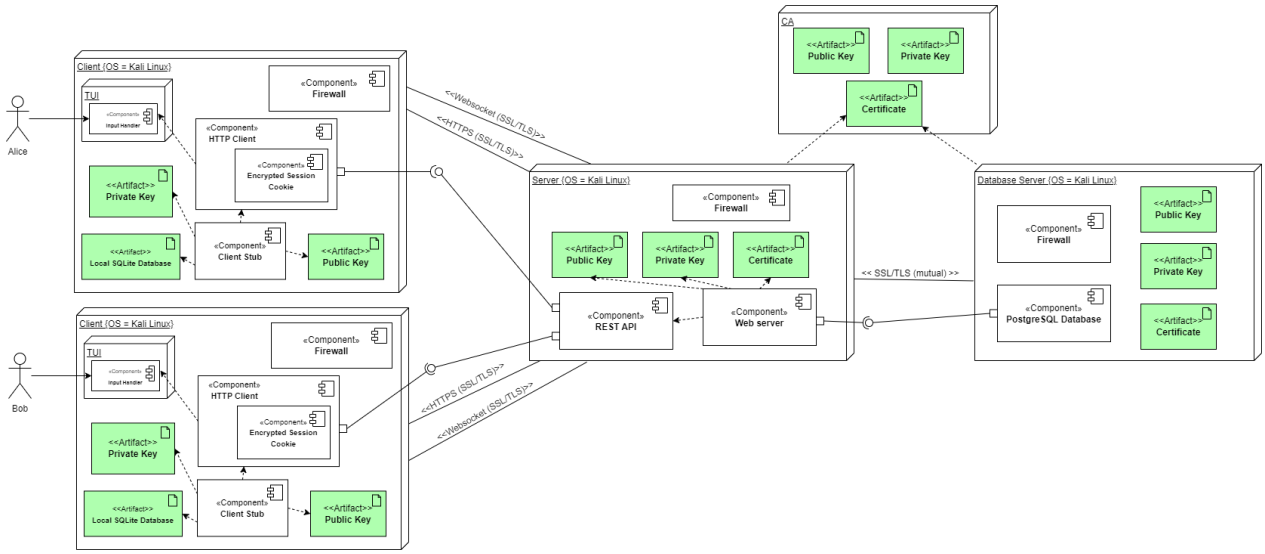

Figure 1: Structural deployment UML diagram of the business scenario showing the architecture of the main concepts.

3 Project Development

3.1 Secure Document Format

3.1.1 Design

1. **Document Format**

   According to our scenario, the **format of the Secure Document** is a JSON-formatted message with the following structure:

```
{
   "sender_istid": "ist1123123",
   "receiver_istid": "ist1321564",
   "timestamp": "2022-01-01T12:00:00Z",
   "content": "Hi! Do you know the solution for the SIRS exercise?",
   "sent_counter": 1,
   "receive_counter": 0,
}
```

   We decided to partially maintain the given message format, but instead of sending a list of messages, we now send one message at a time. This approach is more convenient, as the user sends only one message at a time, which is processed by the server and forwarded to the receiver.

   Additionally, we **added two counters** to the message, one to track the number of messages sent, and the other for the received ones. Then, when a client sends a message, the counter increments, enabling the receiving client to **detect if any messages are out of order or missing**.

2. **Library Functionalities**

   To provide cryptographic security for the format document, we make use of the functions of the **Cryptolib**, our custom cryptographic library, that includes the following functionalities:

   - **help** - displays the menu of available functions
   - **protect** - adds security to a document (confidentiality, integrity and authenticity)
   - **check** - verifies the security of a document (checks the integrity of the cyphertext)
   - **unprotect** - removes security from a document (checks the integrity and decrypt)
   - **hash-password** - generates a hash of a given password
   - **verify-password** - checks if the password corresponds to the given hash
   - **gen-secret-key** - generates a secret key with *ChaCha20Poly1305* algorithm
   - **gen-rsa-keypair** - generates an RSA key pair (public and private keys)
   - **encrypt-key-with-pub-key** - encrypts a key with a public key
   - **decrypt-key-with-priv-key** - decrypts a key with a private key

   We decided to provide not only the **protect**, **check**, and **unprotect** functions but also other useful functions for generating **symmetric and asymmetric keys**, **password hash generation and verification**, and **key protection using asymmetric keys**. These functions are essential for the implementation of the infrastructure, and their usage will be explained in more detail in the topic below and in the section 3.1.2.

3. **Role of Cryptolib in the Security of the Format Document**

   When the client writes the message, the JSON of it, which is the formatted secure document, is filled by populating the sender and receiver IDs, a timestamp of the sending moment, the content, and the counters, as mentioned earlier, are used to detect out-of-order or missing messages, addressing the **SR3** integrity challenge.

   The **protect** function from **Cryptolib** is then applied to this JSON, which will encrypt the data with a symmetric secret key using the stream cipher algorithm *ChaCha20Poly1305*, that merges confidentiality with integrity and authenticity, and uses a randomly generated *nonce*. The symmetric key is then encrypted with **assymetric encryption** using the sender and receiver public key, preventing the server from seeing the message content or even the counters, solving the **SR1** and **SRA1** confidentiality challenges, and ensuring end-to-end encryption, and the **SR2** of integrity.

   We also decided to use **symmetric encryption** because it involves a shared secret between both clients, and due to performance reasons as is much faster than asymmetric encryption, thus the algorithms require fewer computational resources to encrypt large amounts of data, making it more efficient, faster, and simpler.

   Then, on the receiver's side, the function **unprotect** is used, which also internally calls the **check** function, both from **Cryptolib**, to verify the integrity and decrypt the message, allowing the receiver to see its content.

   The decisions regarding the algorithms used to implement the security of the formatted document, as well as other related security layers, are explained in detail in section 3.1.2.

4. **Format Document Example**

   Below is a complete example of the **data with the referenced protections** (section 3.1.2) that is transmitted.

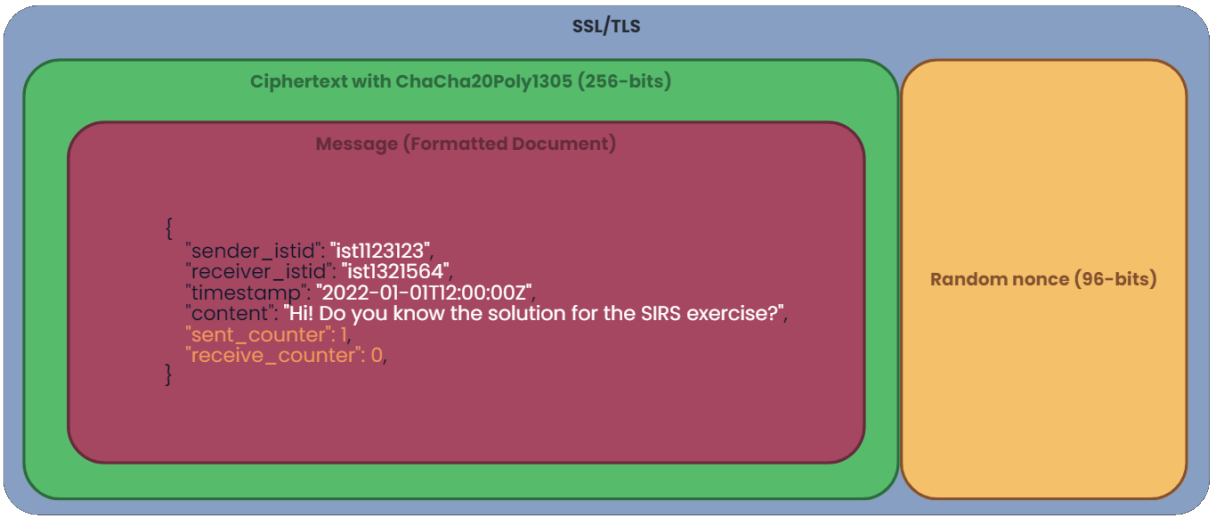

Figure 2: Secure formatted document, encrypted with the *ChaCha20Poly1305* algorithm that uses a key of 256-bits. The random nonce of 96-bits is also appended to the ciphertext, and all of this is transmitted through an SSL/TLS-secured channel.

3.1.2 Implementation

To implement our architectural solution, we utilized **Rust** version **1.83** (2021 edition).

For the implementation of the **Cryptolib**, we used the **RustCrypto library**, which contains methods for **encryption/ decryption**, **password hashing**, and a *Crate* for the algorithm that combines *ChaCha20* stream cipher with the *Poly1305* universal hash function, the **ChaCha20Poly1305** algorithm.

1. **Library Command-line**

   Our **Cryptolib** provides the following commands, that match with the design shown in *"Library Functionalities"* in the section 3.1.1.

```
> help
> protect <input file> <key file> <output file>
> check <doc> <key file>
> unprotect <input file> <key file> <output file>
> hash-password <password>
> verify-password <password> <hash>
> gen-aes-key <output file>
> gen-hmac-key <output file>
> gen-rsa-keypair <bits> <priv key output file> <pub key output file>
> encrypt-key-with-pub-key <key file> <pub key file> <output file>
> decrypt-key-with-priv-key <key file> <priv key file> <output file>
```

   The **help** command displays a list of available commands and their arguments. In case any argument is missing or invalid, it shows an error message with a summary and additional details if exists.

2. **Security Document Implementation**

   To implement the security of the formatted document, as outlined in section 3.1.1, we selected algorithms that are explained in the following topic of the current section, *"Challenges Faced and Overcome"*.

   The document is encrypted trough the **protect** function of the **Cryptolib**, using **ChaCha20** of **256-bits** with a random nonce, ensuring **confidentiality**, while **integrity** is maintained through the **Poly1305** hash function. The **authenticity** of the ciphertext is guaranteed through the combination of both, as the encryption key is used on the hash functions, working as an HMAC.

   Additionally, the sent and receive **counters**, included in the message, will be tracked locally on the client side using an SQLite database, as the sent counter will be incremented on each transmission from the client.

   Since the counters are encrypted on the formatted document (message), the server cannot access its value. However, the receiving client can detect the last message's counter and **verify whether the new message is in the correct sequence or if any messages are missing**.

   For the other client to retrieve the received message, they simply need to use the **unprotect** method of the **Cryptolib**. This method verifies the integrity of the ciphertext and decrypts the message with the help of the nonce, which is sent along with the message. Once decrypted, the client can check if the counter matches the last one stored in their local database and choose to reply if desired.

3. **Challenges Faced and Overcome**

During the design of the document format's security (section 3.1.1), we encountered several **key questions**:

- *How many bits should we use for symmetric encryption?*
- *How is the nonce generated?*
- *Which is the best symmetric cipher mode to our scenario?*
- *Should we use an HMAC, an algorithm that simultaneously encrypts and guarantees integrity and authenticity, or should we opt for signing the data instead?*

Then, below are the solutions that we achieved to address these implementation questions.

(a) **256-bit Key Length**

When generating a new symmetric key for **encrypting each message individually**, it is not necessary to use an extremely large key, such as a 512-bit key, since the key will not be reused to protect all messages in the channel.

However, thus the messages will remain **protected in the server database for extended periods**, it is important not to use a weak key, such as a 128-bit key, as a weaker key is less robust and more susceptible to being compromised over time.

Therefore, a balance must be struck between security and computational cost, that's why we ended up choosing a **256-bit key** [1, 2].

Also, rotating the secret key for each message ensures that, in the event of a key compromise, **only a single message is affected** rather than the entire conversation.

(b) **ChaCha20Poly1305 with 256-bit Key**

To encrypt the messages, we used **ChaCha20** with a **256-bit key**.

Since there is an implementation of **ChaCha20-Poly1305** in a **reliable Rust library**, we decided to use it. This choice combines the *ChaCha20* stream cipher with the *Poly1305* universal hash function, ensuring **confidentiality, integrity, and authenticity in a single operation**.

As a **stream cipher**, it does not rely on block structures, meaning an **error in one bit affects only that bit**. This contrasts with block ciphers, as *GCM* or *AES*, where an error can corrupt an entire block. Additionally, it is **resistant to side-channel attacks**, such as timing or cache analysis, without requiring specialized hardware [1, 3].

Furthermore, stream ciphers have **lower latency compared to block ciphers**, as there is **no need for padding** to fit fixed block sizes. This is particularly useful when the length of the message varies, and consequently, the length of the data to be encrypted [1, 3].

These characteristics make this algorithm ideal for **high-performance, real-time systems**, such as those requiring instant messaging.

(c) **Discarded Algorithms**

AES with CTR mode:

Initially, we considered using AES with CTR mode, which converts a block cipher into a stream cipher and combines it with a randomly generated IV for each message to ensure uniqueness. Since the blocks are independent of each other, an **error in one block does not affect subsequent blocks**, making it more efficient.

Additionally, CTR mode does **not require padding**, reducing overhead and making it suitable for messages of varying lengths. However, CTR mode **requires additional mechanisms for integrity and authenticity, such as HMAC**, which adds complexity and potential vulnerabilities, due to both operations need to be implemented manually,

increasing the risk of errors and reducing performance [3].

AES-GCM:

This algorithm proved to be more promising than *CTR* mode for our implementation, as it offers the same benefits but with **faster performance** and **built-in integrity and authenticity** [3]. By **eliminating the need for an *HMAC***, *GCM* reduces the complexity and potential vulnerabilities associated with separate *HMAC*-based implementations, which require two distinct steps. Instead, *GCM* achieves these objectives in a single operation through a reliable library. However, as detailed earlier, *GCM* is still based on **AES**, which is a **block cipher**, and thus retains some inherent disadvantages.

ECB:

Rejected due to its vulnerability to **pattern** recognition, making it insecure [1, 3].

CBC:

Not suitable because a single **corrupted bit** can affect an entire block and one bit in the next block. Additionally, it is not parallelizable, limiting performance [1, 3].

OFB:

Discarded due to **error propagation**, as the cipher output depends on the previous block [1, 3].

(d) **Generation of nonce**

The nonce is **randomly generated** for each cipher, using a **cryptographic random number generator** based on the Operating System, ensuring no reuse of it, which could compromise security.

Since *ChaCha20* uses the nonce along with the key to encrypt the message, the randomness of the nonce ensures each encrypted message is unique.

(e) **Message Signing and Authentication**

We determined that **signing the messages was unnecessary** because the **ChaCha20-Poly1305** algorithm already guarantees *integrity and authenticity*.

While digital signatures offer **additional properties** such as non-repudiation, anti-forgery, and non-reusability, these were not required for our use case. Implementing them would only add **unnecessary complexity and computational cost** without providing significant benefits [3].

(f) **Packet Freshness**

We chose not to implement additional mechanisms for freshness because all communications occur over **SSL/TLS**, which inherently ensures packet freshness [1].

Additionally, the **nonce** sent with the formatted document already ensures freshness, as it is a large enough random value to prevent repetition.

3.2 Infrastructure

3.2.1 Network and Machine Setup

1. **Machines Setup**

To build the infrastructure we need to make a setup of a set of **separate virtual machines, with network isolation**. All of the machines are running the Linux 64-bit, KALI 2024.4 OS:

- **Client 1: Alice**: 192.168.0.1 - client to access the application server
- **Client 2: Bob**: 192.168.0.2 - client to access the application server
- **Server**: 192.168.0.3 and 192.168.1.1 - the application server exposing the *REST API*
- **Database**: 192.168.1.2 - the *PostgreSQL* database server

The 4 machines are distributed across **2 networks**, as shown in figure 3.

The clients are in the common subnet **192.168.0.0/24**, as we assume that both clients will operate over the IST network. However, this is not a strict requirement, as they can still function with any other IPs. The clients are connected via their *eth0* interfaces to the server's *eth1* interface through a switch, corresponding to the first network connected to ***SW1***.

The server application and the database server, connected to ***SW2***, belong to the second network **192.168.1.0/24**.

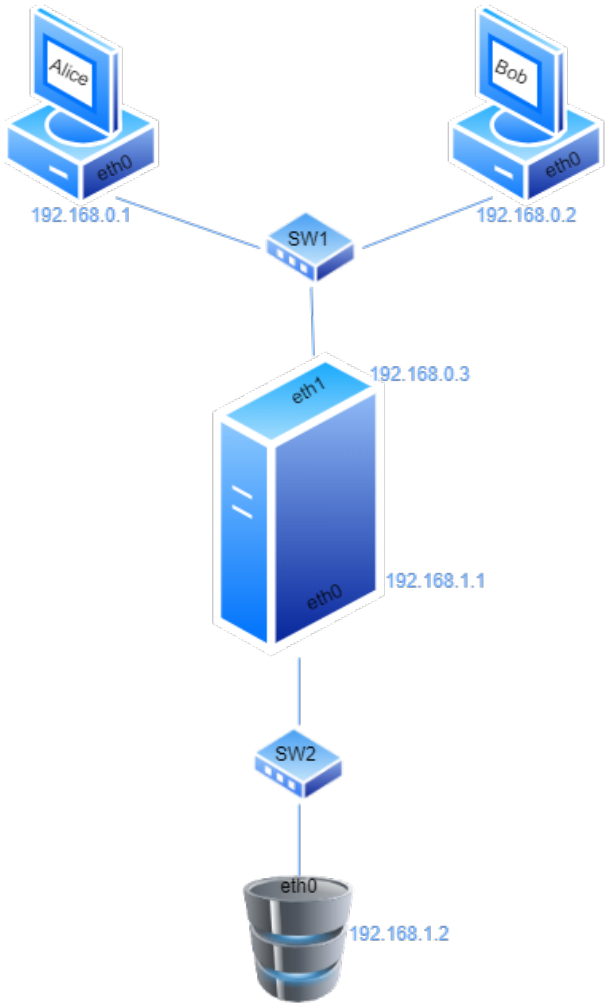

Figure 3: Diagram with 2 clients, 1 server application and 1 database, with the corresponding network interfaces and networks (**192.168.0.0/24** and **192.168.1.0/24**).

2. **Built Infraestructure**

For the client-server application infrastructure, considering all the requirements and the chosen challenge, we implemented the following solution, which aligns with the structural diagram of figure 1.

It is important to highlight that all the machines have a **firewall** set up (more information in *"Secure Network"* in section 3.2.2).

(a) **Client**

Connected to the server using **SSL/TLS**, and contains a local database, a TUI (Terminal User Interface), and has a side channel with a websocket with the server:

- **TUI:**

  Each client has a TUI built with the **Ratatui** library in Rust, enabling the user to perform login, register, logout, send and view messages, view and add contacts, and view their own profile.

  The requests sent to the server are performed using a **HTTP client** made with the **reqwest** crate from Rust.

- **Local SQLite Database:**

  It's a **SQLite** database designed to reduce the volume of requests to the server by acting as a **cache on the client**. Instead of the client requesting synchronization and the server sending all messages, the server now only sends messages created after the latest date recorded in this database.

  It contains:

  - A ***Contacts*** table with the fields *id*, *name*, and *public_key*.

  - A ***Stored Messages*** table with the received and sent messages with the fields *id*, *sender_istid*, *receiver_istid*, *timestamp*, *content*, *secret_key*, *receive_counter*, *sent_counter* and *server_id*.

- **WebSocket Connection:**

  The client establishes a **side-channel connection with the server** through a WebSocket. This acts as a **notification** mechanism, similar to a **webhook**, eliminating the need for the client to continuously poll the server for new messages. Whenever a message arrives for this client, the server sends a notification through the connection.

(b) **Server**

Connected to the **clients** and to a **PostgreSQL** database via **SSL/TLS**.

Accesses the PostgreSQL database and performs **CRUD** operations (Create, Read, Update, Delete) for secure documents, users, messages, and the message history.

Handles client requests via a **REST API** (application programming interface) made with the **Rocket** library.

(c) **PostgreSQL Database**

A PostgreSQL connected via **SSL/TLS** to the server.

It includes:

- A ***Users*** table with the fields *id*, *name*, *password_hash* and *public_key*.

- A ***Incoming Messages*** table for the messages received by the server from the clients, with the fields *id*, *user_id*, *content*, *secret_key*.

- A ***Outcoming Messages*** table for the messages sent from the server to the clients, with the fields *id*, *user_id*, *content*, *secret_key*.

3. **Technologies Choice**

As outlined in the topics above, we used several technologies to build the infrastructure, including:

- **PostgreSQL 17.2 Database:** Chosen due to prior **familiarity** and its ability to meet the application's requirements, thus it supports **SSL/TLS connections in both directions** (server -> database and database -> server).

- **SQLite Database:** Allow us to have a local SQL database, on a **lightweight** solution. Moreover, through the **SQLCipher** extension (explained in detail in *"Adicional Security Measures"* in section 3.2.2), it enables the **encryption of the file of the database**, which was essential for our solution.

- **SSL/TLS:** Used instead of SSH because, although both provide secure communication, SSL/TLS is more **widely supported by libraries**, making integration easier. Also, the fact that is specifically designed to secure **application layer** protocols, such as HTTP, APIs, and real-time data transfer, makes it more appropriate than SSH which is primarily developed for secure remote access and file transfer. Additionally, SSL/TLS is natively **compatible with WebSockets**, whereas SSH requires alternative solutions to operate in such scenarios.

- **WebSocket:** Runs over **SSL/TLS**, ensuring secure communication, and enables the creation of a **channel** between the client and server for efficient real-time message delivery.

- **Rust 1.83:** Offers numerous security advantages, such as **preventing memory related problems** common in C/C++, it's efficient **without relying on a garbage collector**, in contrast with Java, and supports **safe concurrency** due to its language design. Rust is also a **low-level language**, comparable to C, but versatile enough as is a **multi-paradigm language**.

- **Rust Crypto:** A Rust library for **cryptographic methods** which contains methods for **encryption/ decryption**, **password hashing**, and a *Crate* for the ***ChaCha20Poly1305*** algorithm that combines stream cipher with a hash function.

- **Ratatui:** Facilitates the creation of **interactive and stylized TUI** interfaces in **Rust**, which enhances user interaction on the client-side.

- **Rocket:** A feature-rich library that accelerates the implementation of a **RESTful API**. It also provides **session cookie management**, which was crucial for enabling user login functionality in our application (explained in detail in *"Adicional Security Measures"* in section 3.2.2).

- **OpenSSL:** Provides commands for **generating cryptographic RSA asymmetric keys**, and the associated **certificates** (detailed in *"Existing Keys and Distribution"* in section 3.2.2).

- **KALI 2024.4 distribution:** A Linux Derbian-based Linux distribution used on each virtual machine (detailed in *"Machines Setup"* in section 3.2.1).

- **Reqwest crate from Rust:** For the **HTTP client** to perform requests to the server.

- **iptables:** To set rules on the **firewalls** of the involved machines.

3.2.2 Server Communication Security

1. **Secure Communication**

   To ensure secure communication across all machines in the infrastructure, we implemented the following method to enable secure communications:

   - **SSL/TLS:**

     SSL/TLS guarantees **confidentiality**, **integrity**, and **authenticity** by authenticating the server and, optionally, the client.

     This protocol prevents several kinds of attacks as: **man-in-the-middle (MITM)** attacks, **replay** attacks, **interception of passwords**, **data forgery**, **eavesdropping** (as the encryption prevents third parties from listening to or capturing the data exchanged between the parties), **dictionary** or **brute force** attacks on passwords, and **downgrade** attacks (where an attacker attempts to force the use of a weaker protocol version to exploit known vulnerabilities, so TLS ensures both parties use and commit to a secure version of the protocol) [1]. These protections secure communication at the application layer.

     The connection between the server and the PostgreSQL database is a **mutually authenticated SSL/TLS connection**, ensuring that both the server and the database authenticate each other.

     Connections between the server and the client are **one-way SSL/TLS connections**, where only the server is authenticated, ensuring client security while maintaining efficiency.

2. **Adicional Security Measures**

   Aside from the secure channels of communication, we need some other measures:

   - **Login and Session Cookie:**

     When a client registers, they send the password to the server, which hashes the password using **Argon2** from the *RustCrypto library*, and then stores it in the database. Afterward, the server returns a **session cookie** to the client.

     This **session cookie**, encrypted using Rocket's built-in library, is stored on the client side. It has a **TTL (time-to-live)**, which serves to validate the client's session, allowing them to perform further authenticated requests to the server without the need to resend their credentials, effectively addressing the challenge **SR4** of authorization.

   - **Encrypted Local Database File:**

     During login, the client's password undergoes a **key derivation function (KDF)** to generate a cryptographic key. This key is then used with **SQLCipher** to encrypt the entire SQLite database. Due to this, the encryption encompasses not only the data but also **metadata, indices, temporary logs, and system tables**, ensuring no sensitive information is exposed, even if an attacker gains access to the database file.

     By relying on SQLCipher, there is **no need to implement encryption and decryption routines manually** for database operations, and for the data itself, reducing the risk of errors and vulnerabilities.

   - **Database Sanitization:**

     All queries to the client's local and server databases include **input sanitization** to prevent SQL injection attacks.

3. **Existing Keys and Distribution**

Initially, we have the asymmetric *RSA* keys for the client, the server, the database, and the CA (a pair for each), along with the associated certificates, all generated using the following commands from **OpenSSL**:

- OpenSSL commands for the ***CA*** to generate a **private key and self signed certificate**:

```
openssl genrsa -out ca.key 4096
openssl req -x509 -days 365-key ca.key -out ca.crt -addext
    "basicConstraints=CA:TRUE"
```

- OpenSSL commands for the application **server** and **database** server to generate a **private key and a certificate sign request for the CA**:

```
openssl genrsa -out server.key 4096
openssl req -new -key server.key -out server.csr
openssl x509 -req -days 365-in server.csr -CA ca.crt -CAkey ca.key -out
    server.crt
```

The **symmetric secret keys** for encrypting the messages **are generated each time a message is sent**.

To facilitate the **distribution** of all of these keys, we designed the following architecture:

- **Certificate Authority (CA):**

  A emulation of CA with a **self-signed certificate** which is generated and distributed to the PostgreSQL database, the server, and all clients.

  The CA **signs the certificates** of the server and database. Since all components trust the CA, they inherently trust any certificates signed by it.

- **Message Sending Process and Key Distribution:**

  When a **client registers for the first time**, they **send their public key** to the server. Upon logging in, when the client wants to **send a message**, they **encrypt the symmetric encryption key** (used for secure messages) with their **own** public key. The client then requests the public key of the other client to the server, and encrypts the symmetric key with the **received public key**, ensuring that only the other client can decrypt it with their private key. Then the client **sends both encrypted keys to the server**, along with a packet containing the **encrypted message** and the **nonce**, as shown in figure 2.

  If a client wants to send a message to another user for the first time, they can **manually add the recipient** through the interface by providing their **public key**, serving as **visual secure side-channel** for ensuring the authenticity of the key. This implementation solves the challenge **SRA2** of confidentiality.

  The **CA** also contributes to key distribution as it has its own public/private key pair and certificate. Using its private key, it **signs** the certificates of the server and the database.

4. **Secure Network**

Since the infrastructure relies on the use of multiple machines, it is essential to configure them properly to ensure their protection on the internet.

- **Firewall (iptables):**
  All machines have a firewall implemented using **iptables** commands with rules:

- **PostgreSQL Server Database Machine:** Blocks all incoming and outgoing traffic by default, allows **loopback** communication, permits established connections, and enables new **TCP connections to the PostgreSQL** server on port **5432** from the server's IP 192.168.1.1.

- **Server Machine:** Blocks all incoming and outgoing traffic by default, allows **loopback** communication, permits established connections, enables new **TCP connections to the application server** on port **8000**, and allows **outgoing connections to the database** on port **5432** from the IP 192.168.1.2.

- **Client Machine:** Blocks all incoming traffic by default, **allows all outgoing traffic**, permits **loopback** communication, and allows input only from **established connections** initiated by itself.

5. **Verify Traffic**

To verify the security of the communication channels, we employed **Wireshark**, a tool that enables traffic interception and analysis. This allowed us to conduct thorough checks and **confirm the payload encryption**.

Below are the screenshots showing the **encrypted data** in the hex viewer for the following connections:

- Between the **server** (192.168.0.3) and the **Alice** client (192.168.0.1) (see Figure 4).
- Between the **server** (192.168.1.1) and the **database** (192.168.1.2) (see Figure 5).

In both cases, it is observable that the connections are encrypted under the **TLSv1.2** protocol.

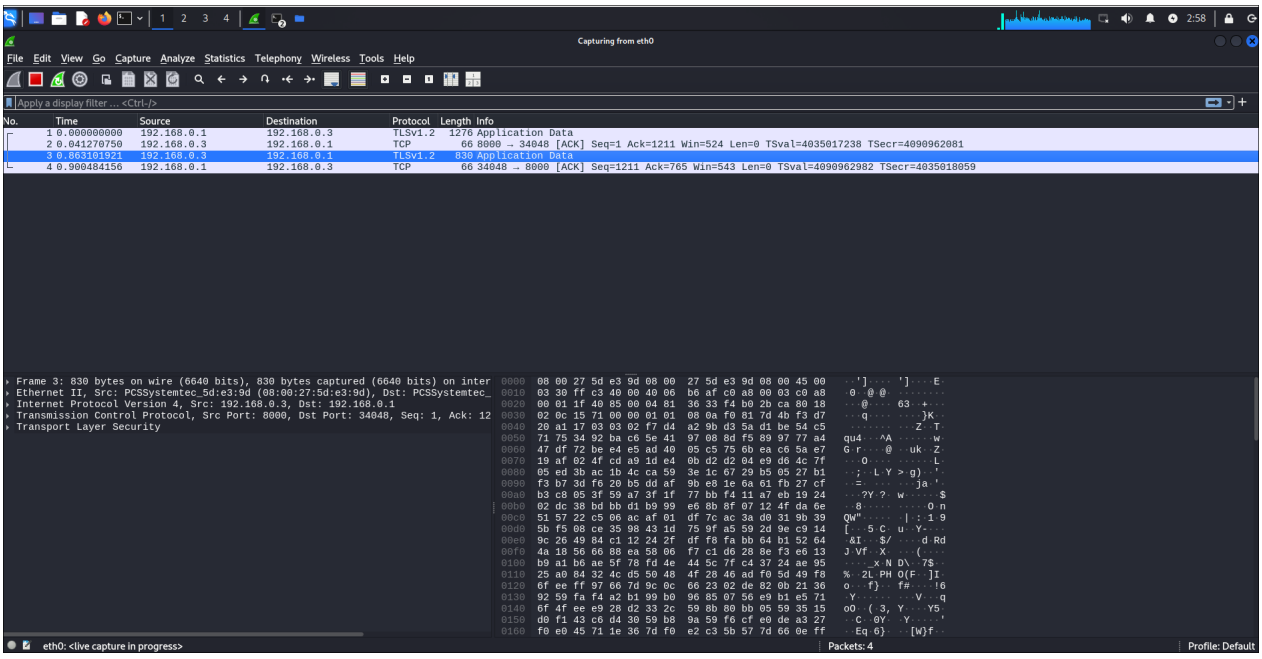

Figure 4: TLS connection between the **server** (192.168.0.3) and the **Alice** client (192.168.0.1) showing on the hex editor the encrypted data.

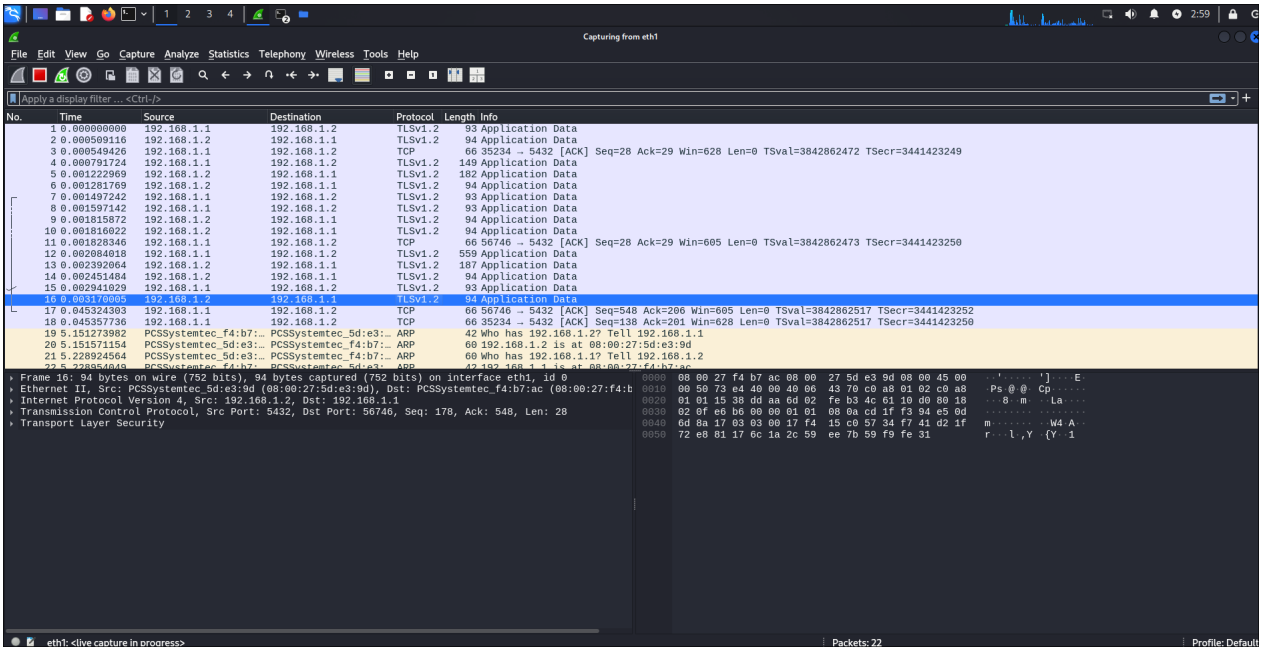

Figure 5: TLS connection between the **server** (192.168.1.1) and the **database** (192.168.1.2) showing on the hex editor the encrypted data.

3.3 Security Challenge

3.3.1 Challenge Overview

1. **New Requirements**
   Given our decision to address **challenge A**, like said in the section 2.1 the following additional requirements were added:

   - **[SRA1: Confidentiality]:** Ensure that only the sender and receiver can access the message content.
   - **[SRA2: Confidentiality]:** Establish a secure protocol for exchanging encryption keys between clients, assuming the availability of a side channel.
   - **[SRA3: Availability]:** Enable users to recover their message history even after losing their devices.

2. **Impact in the Original Design**

   From the beginning, we envisioned a **solution that addressed both** the security needs and the requirements of challenge A, which is the architecture explained throughout the previous sections.

   However, an **original previous design without considering challenge A** would still encompass everything explained so far, with the **only difference being the absence of asymmetric encryption on the client**, as it is now unnecessary.

   As a result, the only **impact** the original design would have is that, with the challenge, the **client needs to have a key pair for asymmetric encryption** of the keys it sends to the server.

3.3.2 Attacker Model

1. **Trust Levels**

   According to our solution, there are various levels of trust:

   - **Fully Trusted**
     - **Client:** Once the client is **registered and logged in**, it becomes authenticated by the server, being fully trusted. There are no other ways to verify if the device has been compromised, that's why from the server's perspective, the client remains fully trusted. The client, although vulnerable to local security risks, uses message counters to ensure they arrive in the correct order, helping detect any issues related to message delivery and integrity.
     - **Certificate Authority (CA):** The CA is fully trusted as it **signs its own certificate and signs certificates** of the server and the database, enabling secure communication based on public-key infrastructure (PKI). Since all parties trust the CA, it plays a central role in establishing and maintaining the trust chain.

   - **Partially Trusted**
     - **Server:** The server is partially trusted because the **client can choose not to trust the public key sent by the server**, which corresponds to another client, and instead **manually add the user** by using the secure code and inserting the public key. Additionally, the server could be compromised if an **exploit or backdoor** allows unauthorized access.
     - **PostgreSQL Database:** While SSL/TLS protects the communication between the server, it is only **partially trusted**, this because the data stored in it is **end-to-end** encrypted, so unauthorized access to the database cannot tamper with the contents otherwise the client will detect it, but that database is somewhat trusted in deciding the messages that belong to each user since it stores tables the received and sent messages of each user of the application. Another relevant factors that contribute to the database not being fully trusted is, that it stored **passwords hashes** instead of them in plaintext, so an attacker cannot see the passwords of each user. It also **does not link users with messages**, since the messages are duplicated across two tables (receive and sent) so an **attacker cannot know that a user messaged another user**.
   - **Untrusted**
     - **External Attacker:** Any **external entity attempting to intercept or alter communication or data** is considered untrusted. This includes attackers trying to perform man-in-the-middle attacks, brute-force attacks on keys or login passwords, or attempting unauthorized access to sensitive data.
     - **Unverified Clients:** **Clients that are not authenticated** are considered untrusted, as until the verification process is complete, the system cannot fully trust their identity or integrity.

2. **Attacker Capabilities and Limitations**

   An attacker is an unauthorized entity or a malicious actor seeking to compromise a system's security without legitimate access.

   - **Capabilities**
     - **Brute Force Attacks:** Attackers can **attempt to guess passwords or encryption keys**, especially if weak credentials are in use.

- **Exploitation of Vulnerabilities:** Takes advantage of software **vulnerabilities**, such as unpatched security flaws or deprecated components, to gain unauthorized access, bypassing some of the security measures, allowing them to try to send messages to clients.

- **Phishing Attack:** By **mimicking legitimate systems or services**, attackers can trick users into disclosing sensitive information, such as the user's password.

- **Ping Flood, DDoS, DoS, Spamming Attacks:** Attackers can disrupt **server availability** by overwhelming it with traffic or resource exhaustion.

- **Access to Encrypted Data on Client's Local Database:** If the attacker successfully **cracks the login password**, which is used to protect the local database file via a key derivation function (KDF), they can derive the encryption key. This would enable them to decrypt and access the entire history of messages sent and received if they have access to the file stored locally in the client's device.

- **Modify or Delete Data:** If attackers **gain access to the server**, they can alter or delete sensitive data on the database. Once they have control over the machine, they can perform any actions that the server itself is authorized to execute.

- **Physical Device Theft:** If an attacker **physically steals a client's device, they may gain access to the private key** stored locally. In this scenario if the attacker is able to also crack the user's password it can totally impersonate the user, doing actions in his behalf and reading the entire history of all messages.

- **Limitations**
  - **Probing Attacks**: The attacker can also perform attacks such as sniffing or port scanning. However, even if the attacker intercepts the traffic, the data is **encrypted**. As for port scanning, it is ineffective since all **ports** on the database machine are **closed** by default, only allowing connections from the server machine to the port **5432**, an the server has all ports also closed, only allowing connections to the web server at port **8000**. Additionally, an **IDS** could help to detect such behaviour and alarm the security team of potential threats.

  - **Man-in-the-Middle (MITM) Attacks:** Attackers **cannot impersonate the server or establish an SSL/TLS connection with the client without the server's private key**. Even attempts to create an insecure connection (e.g., HTTP) will fail because the client only connects via HTTPS, so performing this type of connection could only be possible by having the server's private key.

  - **Intercepting Connections:** Even if traffic is intercepted, attackers **cannot decrypt SSL/TLS communication without the private key** of the server.

  - **Exploits:** As we **implemented the system in Rust**, the language itself provides **memory safety**, reducing the risk of successful exploits targeting memory related vulnerabilities.

  - **Encrypted and Secure Communication:** Attackers cannot **decrypt end-to-end encrypted messages or forge valid ones without access to the private key** of the client. And even if SSL/TLS is compromised, the content of the messages remains secure

due to an additional layer of encryption provided by the end-to-end encryption.

- **Bypassing Security Mechanisms:** Even if an attacker cracks a client's password, it will be **able to send invalid messages to other clients**. However, these messages will be ignored due to end-to-end encryption, since the attacker lacks the client's private keys, the messages they send will be meaningless, as they cannot be decrypted for the other user. Additionally, any messages they receive cannot be read, as the attacker won't have the private key of the impersonated client, nor can they decrypt the encryption key, which is encrypted with the public key of the impersonated client.

- **Replay Attacks:** The **SSL/TLS** protocols prevent replay attacks, where intercepted packets are resent to gain unauthorized access. Additionally, our implementation includes sent and receive **counters** within the transmitted messages, enabling us to detect and mitigate such attacks.

- **Database Security:** The database **firewall only accepts connections from the server**, blocking unauthorized access attempts. Additionally, the end-to-end encryption ensures that even if attackers gain access to the *PostgreSQL* database, the content remains unreadable. Also, the **database users are limited in privileges**, reducing the risk of misuse, thus we have created a separate, that's not the *PostgreSQL* admin, called *messagist_server* that only has access to the database *MessagIST*. Also, all databases, both client-side and server-side, handle inputs by **sanitizing** them to prevent **SQL injection** attacks. Another relevant point is that even if an attacker gains access to the database, it **cannot know what messages are sent between which users** as the database does not store relations between two users and the message.

- **Access Control:** The **logging** can help to avoid attacks from internal attackers.

- **Multi-layered Security:** A multi-layered defense implemented by **firewalls** in all the machines.

3.3.3 Solution Design and Implementation

1. **Redesign of the Solution**

Having in mind the **initial design without the challenge**, we need to perform the following redesign to meet the final design that we implemented, the one that addressed the requirements of the challenges:

- **[SRA1: Confidentiality]:** For this requirement we need end-to-end encryption. In the initial design, it was not required to encrypt the encryption key with the client's public key before sending it to the server, allowing the server to access and view the key in plaintext. By modifying the design to **encrypt the key with the public keys of both clients before sending them to the server** (2 encrypted keys are sent to the server), we prevent the server from accessing these keys, as only the client, possessing the matching private key, will be able to decrypt and access them.

- **[SRA2: Confidentiality]:** In the original design, without considering the challenge, there was no way to address this requirement. To solve this, we introduced an option that allows a client to **add another client by manually inserting their public key**. This process acts as a visual secure code, functioning as a side channel for verification, ensuring that the client can securely confirm the identity of the other client.

- **[SRA3: Availability]:** In the original design, there was no way to address this requirement. In a way to solve this, we introduced a public/private key pair for the client. Now, since the **client's private key is stored locally**, and assuming they have a backup copy, they will never lose access to their messages. Even if the device is stolen, the client can still decrypt the messages. As mentioned in **SRA1**, the symmetric key for each message is encrypted with the client's public key. Therefore, the client can decrypt it and, use the symmetric keys to recover the entire message history.

2. **Redesign of Key Distribution**

   In the original design, without the challenge, **key distribution was handled by the server**. Clients would send their symmetric encryption key to the server, and then the other client, with whom the first client was communicating, would request the key from the server in order to decrypt the received message.

   However, to address the challenge, the solution involves **encrypting the key with the public keys of both clients** involved and sending them to the server. The receiving client requests the encryption from the server, which then provides the key encrypted with the receiver client's public key. The receiving client decrypts the key using their matching private key, obtaining the encryption key necessary to decrypt the message itself.

   Essentially, the **distribution method remains the same**, but now instead of sending just **1 key in plaintext, 2 encrypted keys are sent to the server**. This shift occurs because **asymmetric** cryptography is now being used to enhance security.

3. **Exchange of messages**

   To exchange messages there are several operations performed by the system that involves some **entities** such as the **user**, his **client application** that contains a **local database**, the **server** of the application, that establishes a connection with a **PostgreSQL** database, and then the **other client application**.

   These interactions encompass a wide range of processes, from user **registration**, which creates a new user in the application, to **login** by the user, **message storage** in both the general database and the clients' local databases, **notifications** when the destination client is online and receives a new message, **synchronization** of the client's local database, which occurs each time the client logs in, or when the client requests messages from the server that it does not yet have, ordering them by message ID, and finally, the **delivery** of the message to the recipient's client application.

   The sequence of these interactions is illustrated in the following **sequence** diagrams, which depicts the *MessagIST* flow.

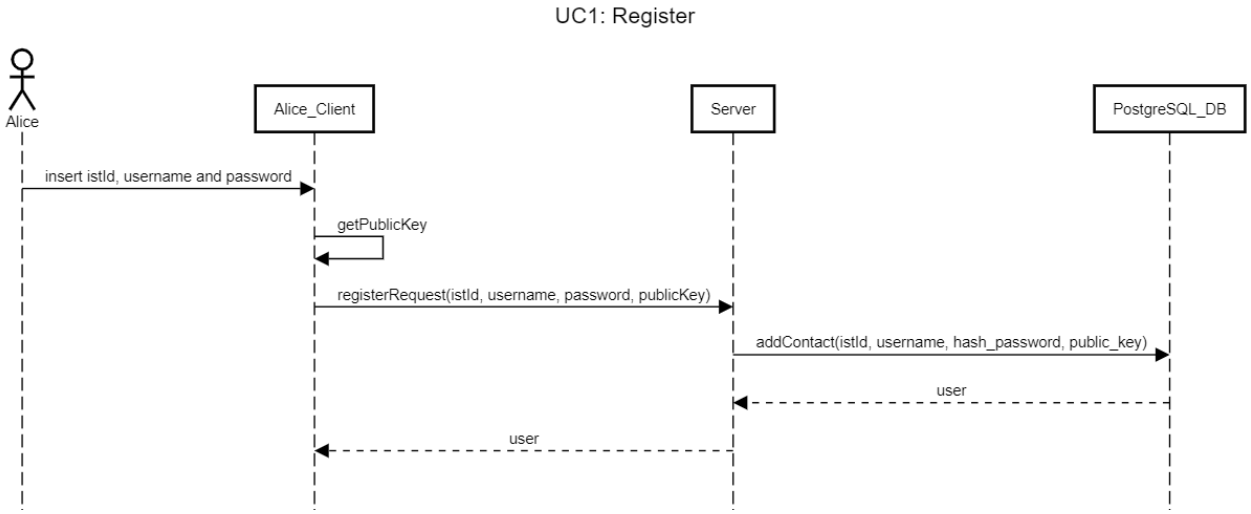

Figure 6: Sequence diagram for the UC of **registration**.

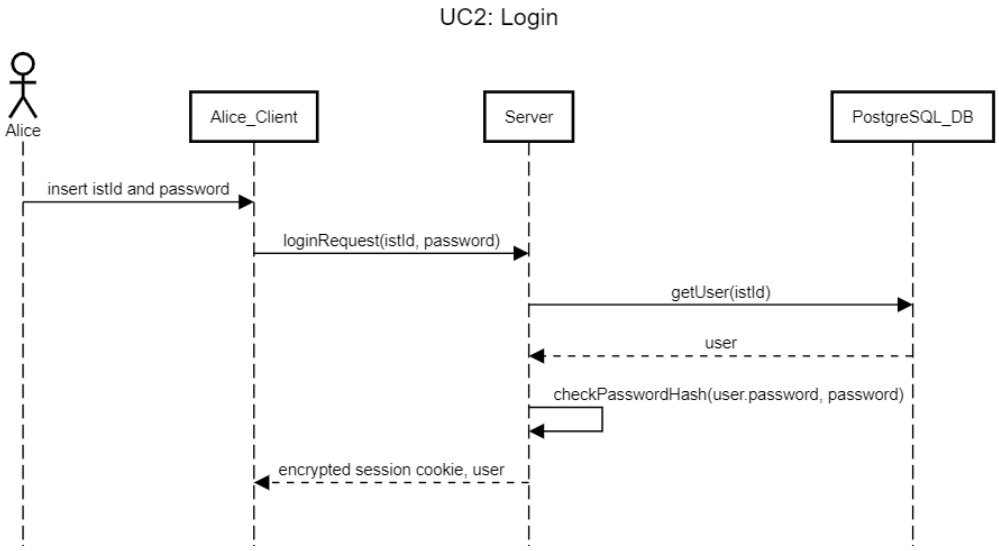

Figure 7: Sequence diagram for the UC of **login**.

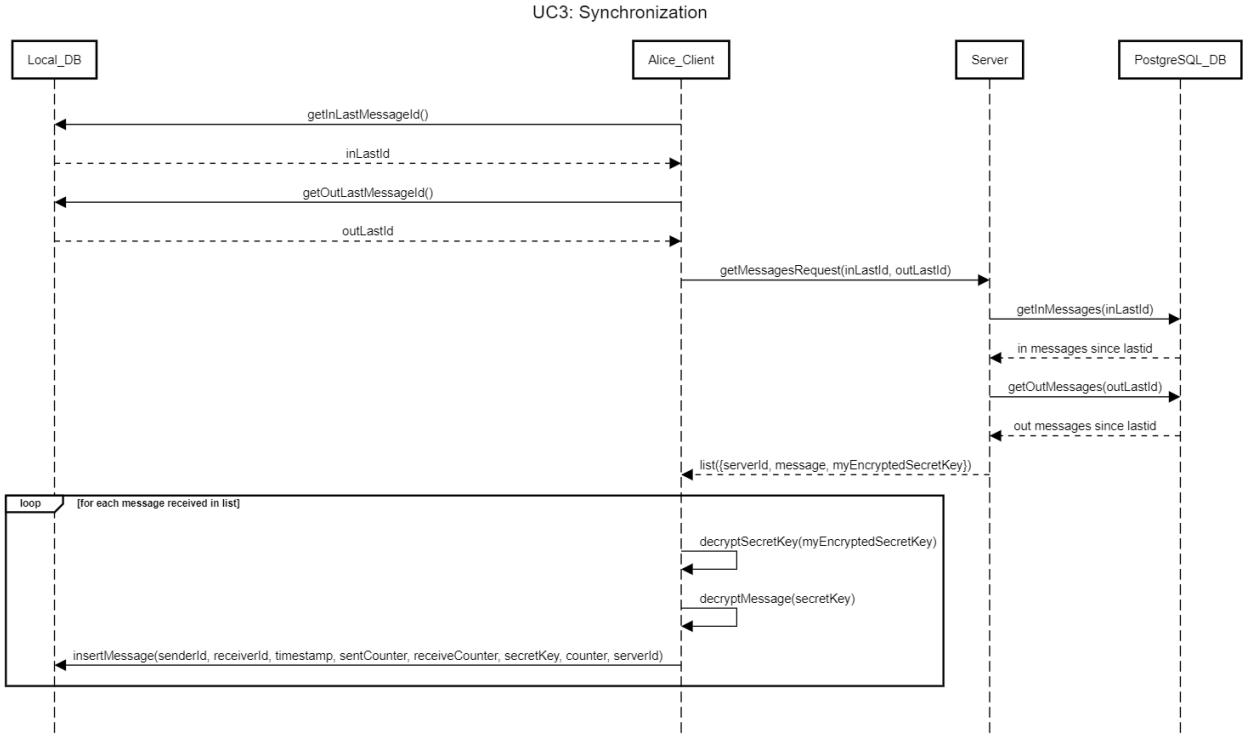

Figure 8: Sequence diagram for the UC of **synchronization** with the local database.

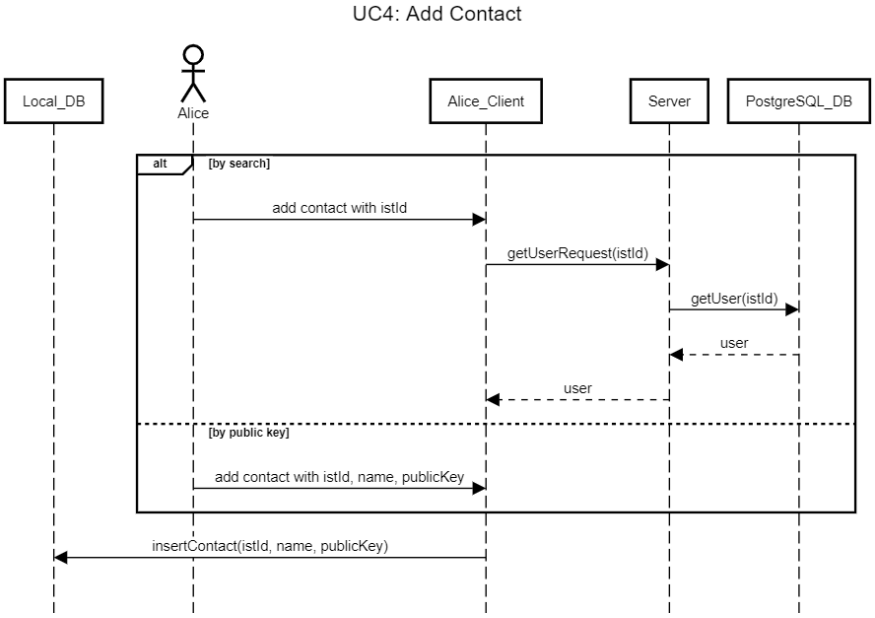

Figure 9: Sequence diagram for the UC of **add contact**.

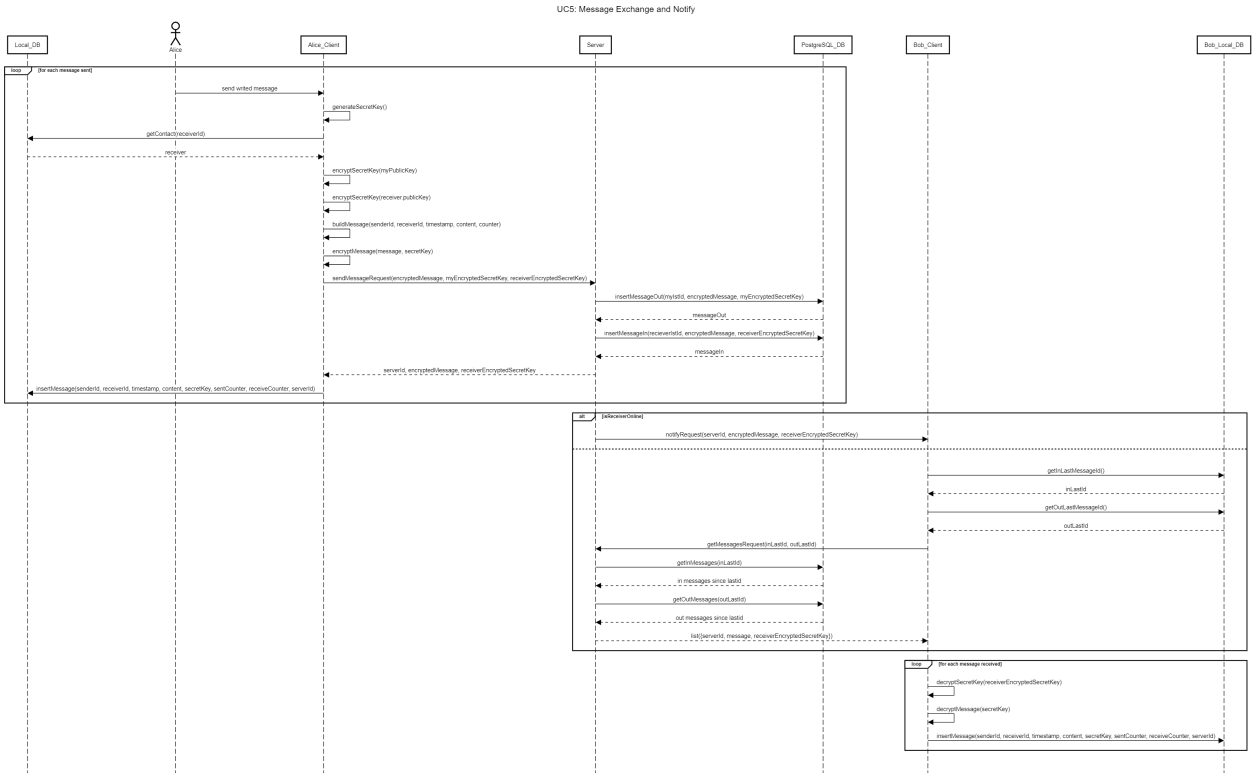

Figure 10: Sequence diagram for the UC of **exchange messages** and **notification**.

4 Conclusion

4.1 Main Achievements

Through the completion of this project, we accomplished several main achievements, which can be highlighted as follows:

- **End-to-End Encryption, Integrity, and Authenticity Check:** We successfully implemented a secure communication system that guarantees the confidentiality of messages, **preventing unauthorized access, even by the server**. The system also ensures the integrity and authenticity of the messages using the ***ChaCha20*** cipher algorithm combined with the ***Poly1305*** hash function, that uses the encryption key. Additionally, message **counters** are employed to ensure that messages are neither tampered with, replayed, nor missing.
- **User Authentication:** We implemented a **password-based authentication** process using **session cookies**, ensuring secure and verified user access, being the messages only sent to authenticated recipients.
- **Scalable Key Management:** We developed a **secure key exchange protocol** leveraging public key encryption and a trusted certificate authority (**CA**), allowing seamless and secure distribution of encryption keys.
- **Data Recovery Mechanism:** We found a way to enable users to securely recover message history, even in cases of device loss, without compromising confidentiality, through the use of the **private key of the client**.
- **Secure Code:** We implemented a secure side channel using a **secure visual code** to **manually add another client** by inserting their **public key**, enhancing confidentiality.

- **Local Data Security:** We implemented *SQLite* local database, that acts as a **client's cache**, with **SQLCipher** for client-side local databases, encrypting the entire database file to protect sensitive data against unauthorized access.
- **Secure Server-Client Communication:** We established **SSL/TLS connections** between all infrastructure components (clients, server, database), ensuring confidentiality, integrity, and authentication in real-time communication.

4.2 Satisfied Requirements

With our implementation, we successfully **satisfied all the proposed challenges**. Below, we explain how these challenges were resolved, consolidating the solutions into a single explanation since they have already been mentioned in various sections (3.1.1 3.2.2, and 3.3.3):

1. **Protection Needs**

   - **[SR1: Confidentiality]:** To guarantee confidentiality, we implemented **end-to-end encryption** for all communication. Messages are encrypted using ***ChaCha20***, and the encryption keys are securely exchanged between users, by being encrypted with the recipient's public key, certifying that only the intended receiver can decrypt them using their private key. This ensures that no intermediary, including the server, can access the message contents.

   - **[SR2: Integrity 1]:** Message integrity is ensured using the ***Poly1305*** hash function, which is combined with the *ChaCha20* algorithm. When a message is received, we decrypt the ciphertext, if an error occurs during that, the ***ChaCha20-Poly1305*** algorithm, which **performs both decryption and integrity verification simultaneously**, indicates that the integrity check has failed. This ensures that the message has not been tampered with during transmission.

   - **[SR3: Integrity 2]:** We further enhance integrity by using **message counters**. Each message includes two counters, a sent, and a receive, preventing replay attacks and ensuring that messages are processed in the correct order. These counters are "maintained" and checked by the clients, and encrypted in the body of the message, having the server no information about it.

   - **[SR4: Authentication]:** Authentication is achieved through:

     - Session Cookies: This requirement is mainly addressed during the client's password login, where the server validates the user's credentials and issues an encrypted session cookie. This **cookie authenticates subsequent client requests**, ensuring secure interactions.

     - Public/Private Key Pair: Each client is issued a public/private key pair. Messages exchanged between clients are encrypted using keys, that are also encrypted with the recipient's public key, then **only the receiver can decrypt it with his private key**. Additionally, messages are also authenticated with the **ChaCha20-Poly1305**, ensuring that only verified clients can participate in the communication.

2. **Challenge A**

   - **[SRA1: Confidentiality]:** We implemented **end-to-end encryption**, ensuring that only the sender and receiver can access the messages. **Keys for encryption are encrypted using the public keys of both clients** that are communicating, and before being sent to the server. This guarantees that only the intended client, possessing the matching private key, can decrypt and access the keys, preventing server-side access.

   - **[SRA2: Confidentiality]:** For secure key exchange between students, we added the ability for users to **manually input a peer's public key and verify it through a visual secure code**, serving as a side channel for authentication. This ensures that even in the absence of direct server involvement, users can establish trust securely.

   - **[SRA3: Availability]:** To recover message history in case of device loss, we **rely on the client's locally stored private key**, as we assumed that they always have a backup copy of it. The server stores encrypted messages and keys, which the client can retrieve and decrypt using their private key, ensuring that the users can retain access to their data, even if they lose their original device.

4.3 Future Enhancements

For future enhancements, we can consider the following improvements:

- **Challenge B: Group Conversations with Admin Access Control**

  The implementation of **challenge B** could be a valuable improvement for our system. For a messaging app, it's essential to enable **private group conversations** where only group members can access the messages, and the **system admin** would have access to the database but would not be able to read the contents of the messages.

  To implement this **key distribution** for a group of users we could use the **Double Ratchet Algorithm**, which encrypts each message with a different symmetric key, and for the distribution of the keys [4], the **Messaging Layer Security (MLS)** protocol, which makes use of **key tree** to ensure the security and confidentiality of group communications [5].

- **Privacy-Enhancing Technologies**

  To allow private communication on the system we could use techniques like **onion routing** used in networks like **Tor**, to anonymize communication [6], making it harder to trace users, ensuring that third parties (like attackers or authorities) are unable to monitor users' activities or conversations.

  Another option could be to use the **Homomorphic Encryption** technique that allows **computation on encrypted data** without needing to decrypt it first [7]. This is useful on the server, as it can process sensitive data without accessing plaintext, ensuring users' data remains private and secure.

- **Improved Key Management and Automated Key Revocation**

  To improve the **keys management**, we could implement a **Public Key Infrastructure (PKI)**, that simplifies the management of encryption keys and automates many processes such as **key generation**, **distribution**, **storage**, **rotation**, and **revocation**. It utilizes **public and private key pairs** in asymmetric encryption, which helps maintain the system's security over time.

  To allow an **automated key revocation**, a process that is essential for invalidating keys when they are compromised, outdated, or no longer needed, can be performed by the PKI.

Other option could be, where a **suspicious activity** is detected, such as the use of a key in an unexpected location or manner, the system can automatically revoke access to the compromised key, by implementing and maintaining a **revocation list**, or notifying all users or systems, to make a particular key invalid.

- **2FA Authentication**

  To avoid the weakness given by the brute force of the password at the client's login, we could implement a 2FA which is a security process that enhances traditional authentication, which usually relies on a password, by requiring the user to provide **two distinct forms of verification** before accessing an account.

  This additional step makes unauthorized access much harder, as attackers would need to compromise **two factors** instead of just one, which may include a **First Factor (Something You Know)** like the user's password, and a **Second Factor (Something You Have)**, as a code generated by an authentication app such as **Google Authenticator** or **Authy**, or a code sent via **SMS** [1, 8].

- **Robust Gateway Architecture (DMZ, Firewall, and Routing)**

  To have a better network security architecture, we could implement a **gateway** that can be used to control traffic between a trusted internal network, like client machines and the server, and the external world, such as the internet.

  This architecture will have a **DMZ (Demilitarized Zone)**, which is a network segment that is separate from the internal network, where some accessible services, like servers are publicly placed, to add an extra layer of security, making it harder for attackers to access the internal network [1, 3].

  So, we'll have a **Simple Gateway (Dual-Homed Gateway)** that is a simple, resource-saving implementation, that avoids extra delays on private network computers.

  This setup consists of a bastion host with an application gateway, connecting the server to the clients and the database while restricting direct communication between the clients and the database, effectively acting as a **router** [1].

  This low-cost implementation only involves adding **one** more **virtual machine** to our design, the bastion host, that will be located at the DMZ along with the server and will be connected to the clients, the server and the database, routering and blocking the respective connections between the machines.

  However, if the **bastion host (gateway)** is compromised, it could expose the entire network to security risks. Additionally, the gateway could become overwhelmed with traffic, potentially affecting overall performance.

- **IDS (Intrusion Detection System)**

  Including an **IDS** like **Snort** on the application server and the database to **detect unauthorized accesses**, allowing us to receive notifications for various types of attacks such as ping/syn floods and port scanning, alerting when the rules are triggered by sending a specific message. The following rules are some examples that could be placed in the machines to help identify potential threats:

```
alert tcp any any -> 192.168.0.0/24 [1:1024](msg: "Port Scan Detected";
    detection_filter: track by_dst, count 5,seconds 60; sid: 1000001; rev: 1;)

alert tcp any any -> 192.168.0.0/24 any (msg: "SYN flood detected";
    detection_filter: track by_dst, count 20,seconds 10; flags:S; sid: 1000002;
    rev: 1;)

alert icmp any any -> 192.168.0.0/24 any (msg: "PING flood detected";
    detection_filter: track by_dst, count 20,seconds 10; sid: 1000003; rev: 1;)
```

Here is a quick explanation of all the rules:

1. This rule is used to detect a **port scan** in the services reserved ports (<= 1024), it alerts when a TCP packet is sent to any of these ports 5 times in 60-seconds by the same address.
2. This rule is used to detect **SYN floods**, it shows an alert when 20 TCP packets with the syn flag are sent in a 10-second interval from the same address.
3. Finally the last rule is used to detect **PING floods**, similarly to the SYN flood it shows an alert when receiving 20 ICMP packets in a 10-second interval by the same address.

4.4 Project Experience

This project has been a rewarding and insightful experience, providing a solid foundation for implementing **security protocols**, **encryption methods**, and **network architecture** for **secure communications**.

The practical challenges of ensuring **confidentiality, integrity, authentication, and avaliability** in real-time systems gave us a deep understanding of the intricacies involved in protecting user data and communication.

Through the hands-on implementation of **end-to-end encryption, user authentication, and scalable key management systems**, we learned the importance of having a robust security infrastructure and the complexities of managing sensitive information across distributed systems.

Ultimately, this project has not only enriched our technical skills but also underscored the **importance of layered security** approach in creating systems that protect both user data and the integrity of the network infrastructure. The knowledge and experience acquired here will be invaluable as we continue to tackle more complex security-related challenges in the future.

5 Bibliography

[1] Lawrie Brown William Stallings. Computer security: Principles and practice, 2018. Accessed: 2024-12-22.

[2] Allen Roginsky Elaine Barker. Transitioning the use of cryptographic algorithms and key lengths. https://doi.org/10.6028/NIST.SP.800-131Ar2, 2019. Accessed: 2024-12-22.

[3] André Zúquete. Segurança em redes informáticas, 2018. Accessed: 2024-12-22.

[4] Moxie Marlinspike Trevor Perrin. The double ratchet algorithm. https://kr-labs.com.ua/books/doubleratchet.pdf, 2016. Accessed: 2024-12-22.

[5] Abdullah Bahashwan1and and et. al. A brief review of messaging protocol standards for internet of things (iot). https://doi.org/10.13052/2245-1439.811, 2018. Accessed: 2024-12-22.

[6] Akshaya Mani and et.al. Understanding tor usage with privacy-preserving measurement. https://doi.org/10.1145/3278532.3278549, 2018. Accessed: 2024-12-22.

[7] Abbas Acar and et.al. A survey on homomorphic encryption schemes: Theory and implementation. https://doi.org/10.1145/3214303, 2018. Accessed: 2024-12-22.

[8] Thanasis Petsas and et. al. Two-factor authentication: is the world ready?: quantifying 2fa adoption. https://doi.org/10.1145/2751323.2751327, 2015. Accessed: 2024-12-20.