Ciências
ULisboa

Faculdade
de Ciências
da Universidade
de Lisboa

Academic Year
2022/2023

Project Report of "Secure P2P Messaging App"

Privacidade e Segurança de Dados

Work developed by:

Bruno Cotrim, nº54406

Diogo Novo, nº60400

Miguel Carvalho, nº54399

INDEX

I – Overview

NOTE: the following report contains nine pages, from which two of them are the cover and index list of the report. From the remaining seven pages, we consider that the images we present in the report are equivalent to the content of two pages (we had the option of annexing these images to the report, but for understandability and clarity reasons we decided to insert them into the main body of the report), making the report present content equivalent to the intended size of five pages. We ask the teacher to understand our decision and that he considers our report as of acceptable size.

The developed project consists of a messaging app that allows users to send messages in a protected manner. We ensure that the communication is end-to-end encrypted, the privacy of messages is guaranteed and that concepts such as key storage, privacy, availability, and encrypted message searching are employed in order to provide a higher level of security. To help us achieve this, we use a Java Framework called SpringBoot so that it is possible to abstract the communication configuration for HTTPS, database access and related management, server deployment and related configuration, and many other features that help us separate all the existing components. Relative to these components, we generated three main components that combined help us achieve our end goal of security. Hereby follows a small overview of each one:

- **Client:** Component responsible for the authentication with the Public Key Generator (PKG), communication between users (including group/topic conversations), message replication and its encryption, key storage, and replication (via Shamir Secret Sharing). The client component allows users to search keywords in the Network Attached Storage (NAS cloud) in a way that these queries remain private, and their results are encrypted. Finally, it is also responsible for encapsulating secret keys through an IBE Scheme, this being the first backup of a user friend associated to its shares and sent messages.
- **Server** (works as PKG): Component responsible for representing the main server of the system's architecture. It is also responsible for signing up and authenticating clients that desire to use services provided by the defined system. Its most significant role is its behavior as a Public-Key Generator that generates public and private keys for client-side encryption. For this, we use encryption schemes namely Identity-based encryption (for the communication between 2 users), and Key-policy Attribute-based encryption (for group/topic chats).
- **Cloud** (works as a NAS): Component responsible for remote storage, specifically shares associated to Shamir Secret Sharing, Encrypted Message Storage, and Index Storage (related to the Cash'14 Dynamic Searchable Encryption Scheme). To note that it is directly linked to the search capability of the system because of the ability of its shared storage to validate a token from the PKG that triggered a storage action. Finally, it also generates KPABE and encapsulates secrets keys with it so that users can successfully create group chats.

Important notes

- We created the Cloud and Server components to make the system highly scalable and allow for multiple instances to be easily added (only needing for an individual common database). In the project delivery we only used one instance for each component.
- To be possible for the persisting of our data in a local database (that uses the project file scope to store messages), the used database connection can be easily changed by altering the configuration file and specifying the information needed for the connection, like URL, user info, certificates, etc.
- For visualizing results and the overall functionality of the system, we provided clients with a REST interface that allows an easy implementation for a local frontend. To visualize and use the interface, we will later provide a brief tutorial on how to use the application through an API that clients offer while also providing a Postman collection.

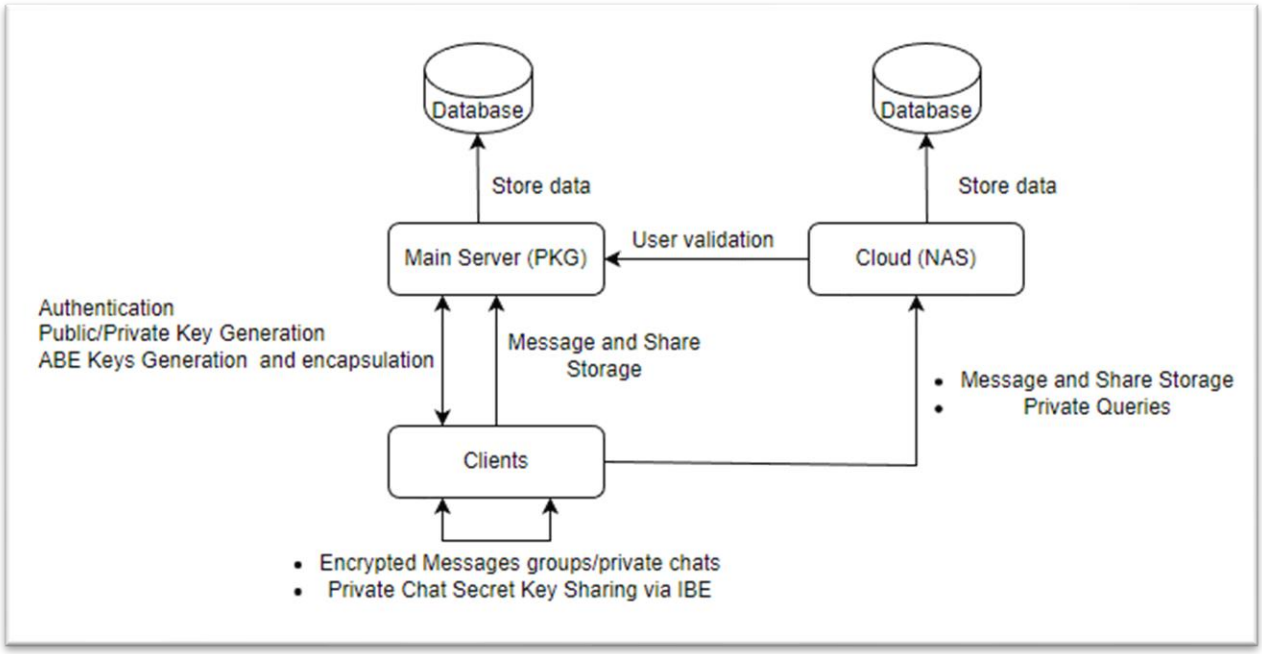

Figure 1 - Diagram overview of the system's components and interactions

II – Authentication

The existing authentication provided by the system is based on JSON web tokens (JWT). Step-by-step we will introduce how this was done and what security guarantees it provides. As described in the official JTW website [1]: "JSON Web Tokens are an open, industry-standard RFC 7519 method for representing claims securely between two parties". Taking this into consideration, a JWT is composed of three parts, namely a header segment, a payload segment, and a signature. The header presents the algorithm used for the signature (in this case, "HMAC-SHA256"), the payload contains the UserId, issue date, and expiration date, and lastly the signature contains the utilized signature, defined as following: HMACSHA512(base64UrlEncode(header) + "." + base64UrlEncode(payload), 512-bit-secret). This last process is supported by an HMAC algorithm that uses a SHA hash function (with 512 bits) to improve the strength of the signature. To note that the signature will provide the integrity of tokens and user authentication.

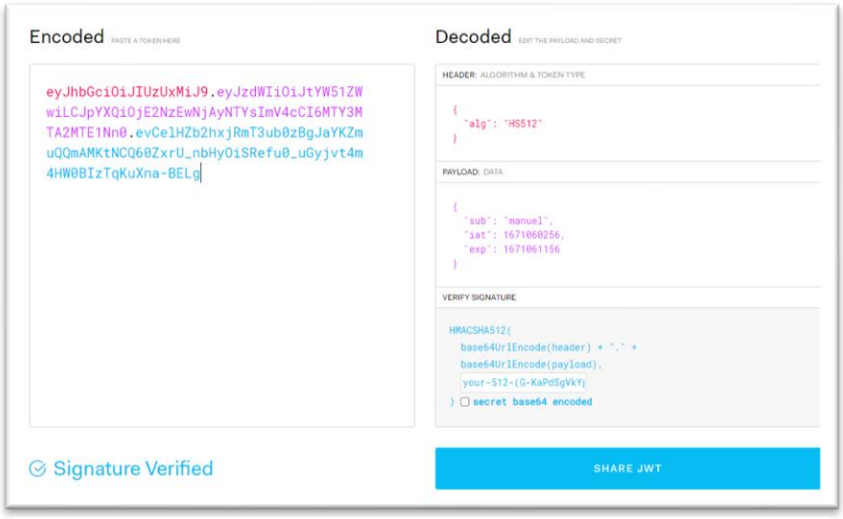

Figure 2 - JWT exemple

Analyzing how the system's flow operates

In first place, the client signs up to the PKG server providing his username, password, and address. Then, the server uses a library called Bcrypt that provides powerful cryptography primitives so hashing and salting of passwords can be made in the most secure way. These last two characteristics are based on the Blowfish cipher.

After the client has successfully signed up, he is now capable of logging in to the system. If this occurs, the server then creates two JWTs from two different master keys for generation and validation. One of the JWTs is called the *access token* and the other is called the *refresh token*. The *access token* allows clients to access functionalities and services of the system (if the server correctly validates the token) and the *refresh token* allows clients to refresh *access tokens* from time to time. This is very useful because if attackers capture the client token, he only presents a minute time window to exploit the system. Since the token expires at every seven minutes, it can be revoked from the defined database, making attackers that manage to capture *refresh tokens* (and get access to other tokens) unable to infiltrate the system: when this type of activity is detected, that specific *refresh token* is revoked by removing it from the database. The *access token* also allows users to identify themselves across the system, something important for share retrievals and key sharing because the entity that is receiving the request always validates the token (with the help of the PKG) to prove the requester's identity. To note that, minute by minute, the *access token* is refreshed, changing its timer to a value between the range of one to six minutes.

Figure 3 - Signup/Signin Request

Figure 4 - Signin Response

III – Private Messaging (chat between two users)

To develop this feature, we strongly relied on the usage of the Identity-based encryption scheme. Its functionality is described as next:

- After the client has successfully authenticated, he probes the server to check for its current state. If everything is deemed as normal, the client generates a request to retrieve the public key and private key generated from the PKG based on his ID and content of its *access token*. Following this, the client will present, on his side, the requested pair of keys. It is important to mention that since IBE requires a secure channel for key sharing, we use HTTPS (a secure protocol) that runs on top of TLS and thus ensures a protected communication channel.
- When a user (U1) opens a chat with another user (U2), two things happen:
    - In a standard case, U1 requests U2 for an encapsulated secret key. If U2 has it, he will send it to U1 so that he can decapsulate it (using his IBE private key) and extract the key. Then, U1 will use Shamir secret sharing scheme to split the key into three different shares (U2, PKG, NAS) and send the respective keys to the shareholders.
    - If U2 does not have the encapsulated secret key, U1 will try to retrieve it from the secret key shares from NAS and PKG so that it is possible to reconstruct the secret key.
    - If U1 cannot retrieve the necessary key shares, he will encapsulate a new secret key by using U2 ID together with its public key, so it's possible to encrypt the resulting object using the IBE scheme. It is important to mention that, at the end of this process, U1 will try to retrieve the encrypted messages from U2 (if not successful, it will retrieve them from NAS).
    - To note that more NAS can be added, being only necessary to change the number of shares and, if needed, the defined threshold.

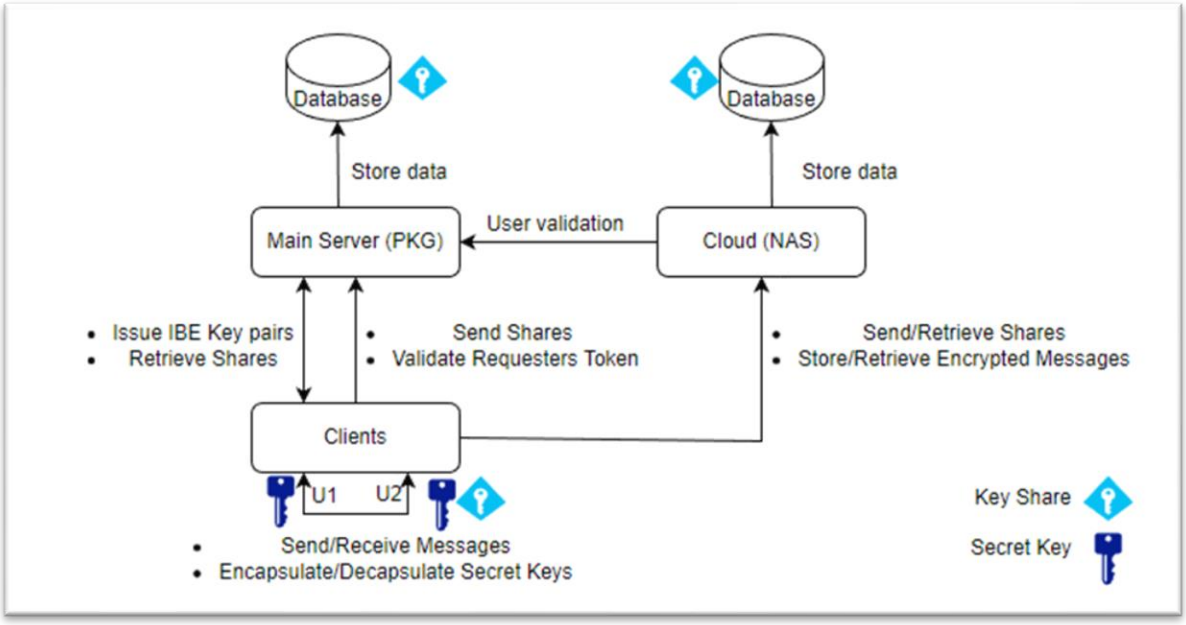

Figure 5 - IBE Scheme and system workflow

- Relatively to the Messaging part of this process, when U1 wants to send a message to U2, he encrypts the message using symmetric encryption (AES/GCM/NoPadding) and a 16-byte initialization vector (IV) and sends it to U2 and NAS. If at that moment U2 is offline, U1 only sends the message to NAS and U2 will later be capable of retrieving the message from this entity. To note that AES in Galois/Counter mode provides authentication, integrity, and privacy, possible because of the usage of AES block cipher in counter mode along with a polynomial MAC based on Galois field multiplication. By using this type of encryption algorithm, it makes it possible for U2 to, when he receives a message and tries to decrypt it, always know its authenticity by analyzing the MAC (assuming that U1 is the only one with the secret key). After receiving/sending a message, the user tokenizes the string into keywords and updates the Dynamic SSE scheme maps to be able to search them in the future.

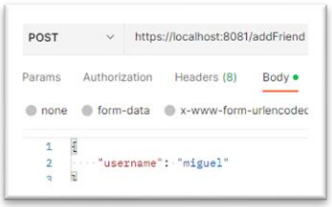

Figure 6 - Add a friend body request message

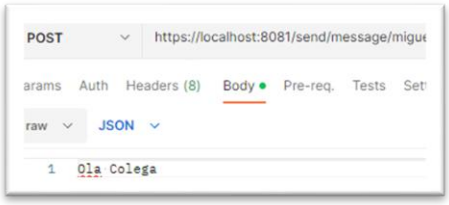

Figure 7 - Send a message to a friend body request message

Figure 8 - Check all messages with friend response message

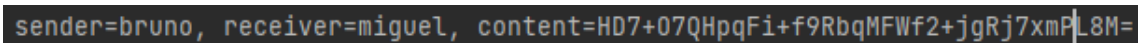

Figure 9 - Message information and encrypted content

IV - Topic/Group Chat

To develop this concrete feature, we used Key-Policy Attribute-based encryption. Its functionality is described as next:

- First, the server starts by generating a master and public key for the algorithm. The private keys from the clients will contain the policy that defines the attributes the cipher must present in order to the possible for the key to decrypt it. Since we used the cloud crypto library, the attributes and policy must exclusively be integers. For that, we hashed our string attributes in a defined integer interval.

To note that with this impletentation collisions can occur regarding this solution. For the key policy, we opted to use the hash of the topic category (hash(topic_category)) and the hash of the participantID (hash(participantID)). Our cypher-text will present attributes related to hash(topic_category) as well as all the hashed names for all the participants belonging to the topic.

- The normal execution is given by following:
  - A user requests the PKG to create a topic/group, by providing the topic name and participants;
  - The PKG will prepare an encapsulated key for all users containing the topic and all the usersIds that will participate in the chat. After this, the users that participate in the group must subscribe to the topic (including its creator);
  - The PKG then validates that the user has access to the topic and, if successful, builds his private key with the predefined policy. Next, this key is returned along with the encapsulated secret key as well as all the participant addresses (for communication).
  - Every time a user subscribes to a topic, he will need to request the NAS for any messages that exist. This way, if other people send messages before he joined, he has the capability of never losing them.
  - When the user wants to send a message to a specific topic, he encrypts it the same way as the private chats, and sends the encrypted message to every online person on the topic. When a user receives/sends a message, he then tokenizes the string into keywords and updates the Dynamic SSE scheme maps in order to be able to search them in the future. This will be explored in detail later.

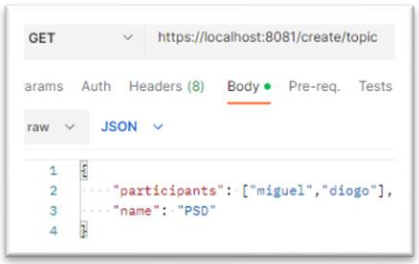

Figure 10 - Create topic response message

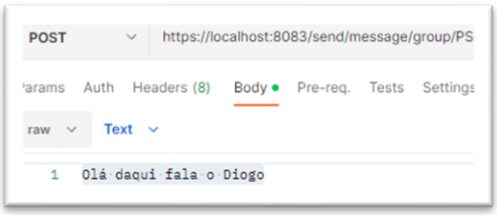

Figure 11 – Send message to group

Figure 12 - Response message of "https://localhost:8083/messages/group/PSD"

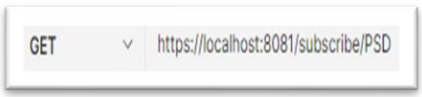

Figure 13 – URL used to subscribe to a specific topic

V - Dynamic Searchable Encryption

This functionality is present in the NAS and Client components previously discussed. To develop this feature, we were inspired by the Cash'14 Dynamic Searchable Encryption Scheme. Now we present how the scheme helps the system to function properly:

- The scheme starts with the NAS creating an empty index map in the database. The client then creates a counterMap that stores keywords and their count. When a client receives/sends a message, he tokenizes it into keywords;
- For each keyword, the client will create k1 and k2 from a PRF(K,w||"i"). Then he creates the index label from a PRF(k1||c) where c is the current keyword count. We constructed the PRF through the use of the" HmacSHA256" algorithm;
- After this, the client creates the index value, which is the encrypted MessageID with K2 using the "AES/GCM/NoPadding" and a random IV. Then, the client sends the encoded indexLabel and indexValue to the NAS that will store them in the DB. To execute the search, the client generates K1 and K2 and sends them along with the keyword to search. Then, the NAS starts with a counter set to

0 and he applies PRF(k1|C) and while he finds indexLabels for increments of C, he will decrypt the index value with K2 and will store the message in a response list. After the client has no longer found index labels, he will stop searching and send the encrypted messages to the client;

- Although we could achieve forward privacy with re-encryption, we opted to use a non-optimal solution that uses different keys for the same keyword based on the conversation that the keyword and document are related to.

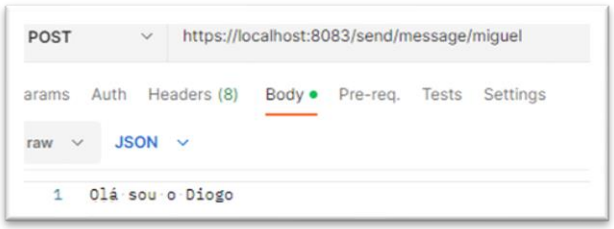

Figure 14 - Sending of a message to a specific user

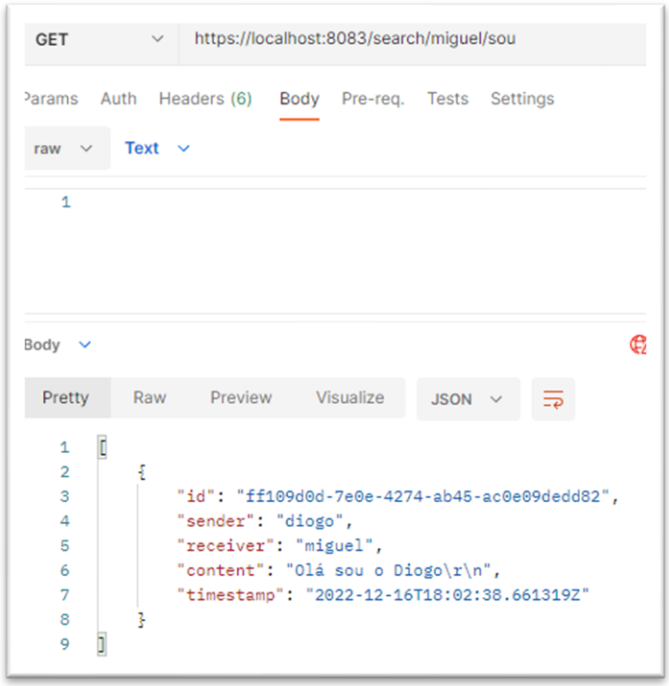

Figure 15 - Search for keywords in messages with a specific user

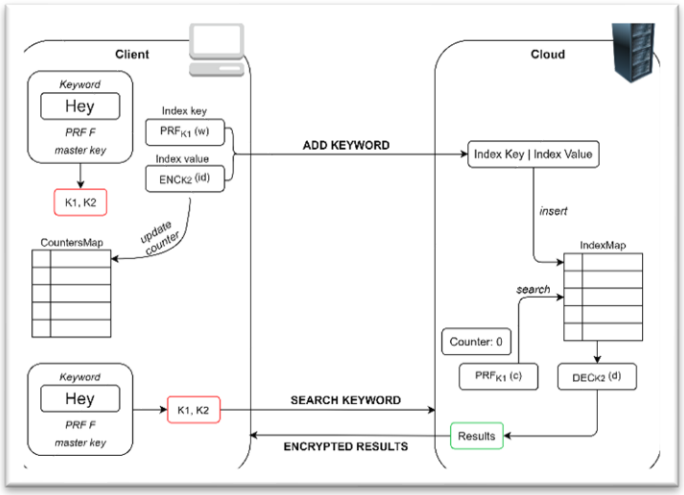

Figure 16 – SSE architecture

VI – Considerations

In order to be able to interactively observe the operation and application of the discussed algorithms and associated processes, we assume that whenever a client shuts down its application, the memory associated with it is lost, making our operations strictly recover the previous state. Thus, whenever a client goes offline (disconnects), the addFriend and Subscribe functions of a topic work as if it were an "open chat" button. Therefore, these functions are not just about setting up new chats, but also chats that have existed in the past, working to restore the state that was defined in a previous interaction.

In terms of performance, our system has a considerably low overhead, since the normal operation presents only two main operations, a request and a response message (with the exception of operations that need to restore state). For example, if you need to fetch shares of the secret key or restore messages, something that involves a slightly higher number of communications than normal, both the clients and the server have low processing times in relation to the response time. This value can increase if we use non-local databases, since the server is dependent on the communication time with this entity. With regard to storage, the only necessary information that must be kept is: the messages on the client and on the NAS; the N shares of the secret key; user and group chat info; and authentication tokens. In more critical operations, such as exchanging secret keys between users, the user who receives the request also receives the JWT from the other user, validating the authenticity of the request with the server. This way, we have one more layer of security for critical operations at the cost of a query made to the PKG.

VII – Project Execution

To execute the developed system, the correct order to initialize everything is to firstly launch the server, then the cloud, and lastly the clients. For this procedure, we can use “mvn clean package” and execute the jar that is created under the target file, from root directory. For demonstration purposes, we have three clients, Bruno, Diogo and Miguel, that need to be executed in conjunction with the environment variable SPRING_PROFILES_ACTIVE={profile name, like: bruno, diogo or miguel}, and in case the client is being executed through the command line with jar, the target file path and the following argument must be provided:

--spring.profiles.active=username --spring.config.location=classpath:application-usenrame.yml

, where the name of the user should be changed for the one desired.

Alongside with the project code, we provide a Postman collection that contains a demonstration flow with client requests to demonstrate all existing project features.