P2P-Messaging App Report – Phase 1

Henrique Marques (57153), Gonçalo Fernandes (58194), Miguel Pena (64446)

This report analyzes the P2P Messaging System developed for the class of Data Privacy and Security in the java programming language. The system implements a peer-to-peer messaging framework that allows users to send and receive messages directly without a centralized server. This report will provide an overview of the code structure, functionality, and the security measures implemented to ensure secure communication among peers.

The codebase consists of several key components organized into different files and directories. The primary classes of interest include:

- **Main:** This class facilitates the initial setup and configuration of the peer, including the loading of keyStores and trustStores for secure communication. It's responsible for user interaction and managing the overall lifecycle of a peer.
- **Peer:** This class encapsulates the core functionality for peer-to-peer communication. It facilitates the connection, message sending and receiving processes, while ensuring that all communications are secure through the use of encryption. This class makes use of the **ConnectionManager** class for establishing secure communication channels, handling socket connections, and facilitate the sending and receiving of encrypted packets.
- **HybridEncryption:** This class was designed to facilitate hybrid encryption. It combines symmetric encryption (AES) and asymmetric encryption (RSA) to provide confidentiality and integrity for messages exchanged between peers.

Functionality

The system provides the following core functionalities:

- **Conversation Listing/Opening:** a peer can see any conversation they have had with another peer and any conversation can be opened and its contents read;
- **Secure Message Sending (E2EE):** any message sent is end-to-end encrypted.

Solution - Overview

In order to start a peer there are some requirements: a keyStore containing the public/private key pair, a trustStore that contains all the certificates of known peers and the IP and port of the peer the user wants to connect to. The IP and port of that peer need to be known prior to establishing that connection because there is no system in place that allows for automatic peer discovery.

When a peer starts it has two main threads, one responsible for reading user input and another responsible for reading incoming messages. Both threads make use of TLS sockets for secure communication.

The main thread always shows the available conversations, enabling the user to open them or to try and talk to another peer that is connected at the moment. When sending a message to another peer a packet (Packet.java) is created containing the name of the sender, the message to send, a timestamp (that currently is not being used) and the type of packet (in this case the type is MSG). After the packet is created it is then encrypted by making use of the methods in the HybridEncryption class, using the receivers' public key and a symmetric key created only for that message. After sending the encrypted packet (EncryptedPacket.java) the sender awaits for another packet (with message type ACK) before writing the message to file and continuing execution (this is a point where the program might remain blocked if it does not receive the ACK from the other peer).

Security Analysis - Security Guarantees Offered

The following security guarantees are provided by **TLS sockets**:

Confidentiality: the underlying protocols used by TLS sockets encrypt any message sent through the socket, preventing malicious actors from finding its contents;

Integrity: each message transmitted includes a message integrity check using an authentication code. It prevents undetected loss or alteration of the data during transmission;

Authenticity: the use of "trusted certificates" (it falls to each peer to decide which connections are safe) and asymmetric keys prevents any malicious peers from impersonating legitimate entities, introducing tampered data, or hijacking communications.

Resilience against Eavesdropping: the use of encryption not only protects message content but also against eavesdropping on communication channels. Even if data packets are captured, they will be meaningless without the corresponding decryption keys.

The following security guarantees are provided by the **hybrid encryption algorithm** used on top of the TLS sockets:

Confidentiality: AES-GCM Encryption ensures the packet data remains confidential by encrypting it with a randomly generated 256-bit AES key, making unauthorized access difficult. RSA Encryption for AES Key protects the key itself by encrypting (wrapping) it with the receiver's public RSA key, allowing only the corresponding private key to decrypt it.

Data Integrity: AES-GCM Authentication Tag Provides a 128-bit authentication tag that verifies data integrity, ensuring the packet data has not been tampered with during transmission.

Authenticity: by using the RSA-OAEP scheme with SHA-256 and MGF1, the encrypted AES key is strongly resistant to manipulation, providing confidence in the authenticity of the encryption key and, therefore, the packet's origin.

Replay Protection: each packet is encrypted with a unique 12-byte IV generated using a secure random number generator, ensuring that the encryption is unique for each packet, protecting against replay attacks.

Implementation Considerations and Shortcomings

- **Peer creation**: because of the way the project is structured, the addition of new peers is more troublesome than it should be. For a new peer to be added, a keystore needs to be created, and the certificates of all the peers it wants to talk to need to be in its truststore, before running the program. Ideally this exchange would occur while the program is running that way the network could have as many peers as possible.
- **Keystore password saved in a variable in code**: this is an obvious security risk waiting to happen. Unfortunately this password is needed to retrieve asymmetric keys from the keystore and the alternative would be to save the keys instead of the password, which would result in the same risk.
- **Key Management:** Proper key management is crucial. The security of the messaging system is directly linked to the management of encryption keys. If keys are compromised, the entire security structure may fail.
- **Username flexibility:** the user name of the peer must correspond to the alias present in the trustStore, otherwise the program will throw an exception.