Ciências ULisboa

Segurança Aplicada (SA)

2024/2025

Project 1 Report

Students
João Santos 56380
Rafael Ferreira 57544
Tiago Santos 64586

Date
6/04/2025

Table of Contents

1. Objectives 1
2. Initialization 1
3. Protocol Description 2
3.1. Message Format 2
3.2. Authentication Files 2
3.3. Transactions Flow 2
3.4. Security Protections 3
4. Specific Attacks and Countermeasures 4
4.1. Man-in-the-middle (MITM) / Eavesdropping 4
4.2. Replay Attack 5
4.3. Unauthorized Account Access 5
4.4. Message Tampering 5
4.5. Denial of Service (DoS) 5

1. Objectives

This project was developed by group 3 composed of João Santos (56380), Rafael Ferreira (57544) and Tiago Santos (64586), in the scope of the Segurança Aplicada (SA) course. This report's goal is to explain the decisions behind our design and implementation of an ATM communication protocol ensuring security by addressing different threats.

2. Initialization

1 - Install JDK (this project was tested using the JDK 21.0.4) -> https://www.oracle.com/java/technologies/downloads/;

2 - Open your CLI and enter into the project directory;

3 - Compile all the java's files:

Windows: javac -cp ".;./bcprov-lts8on-2.73.7.jar" *.java

Linux: javac -cp ".:./bcprov-lts8on-2.73.7.jar" *.java

3 - Run the Bank server first:

Windows: java -cp ".;./bcprov-lts8on-2.73.7.jar" BankServer.java

Linux: java -cp ".:./bcprov-lts8on-2.73.7.jar" BankServer.java

4 - After running the server Bank, it is possible to make operations in the ATM client:

Windows: java -cp ".;./bcprov-lts8on-2.73.7.jar" ATMClient.java <arguments>

Linux: java -cp ".:./bcprov-lts8on-2.73.7.jar" ATMClient.java <arguments>

3. Protocol Description

3.1. Message Format

Each message exchanged between the components (ATM and Bank) has the following structure:

- **Message**:
  - **Sequence Number (4 bytes):** Incremental value used to prevent replay attacks.
  - **Payload:** in JSON format, encrypted with AES-256 in GCM mode.
    Example content: {"account": "bob","deposit": 199.57}
- **MAC:**
  - 16 bytes computed with HMAC-SHA256 over the plaintext payload.
  - The total structure is encrypted with a symmetric key, with the MAC calculated on payload in plaintext:
  - The final format will be this:
    MessageWithSequenceNumber = E_s (Message) + MAC (Message).

3.2. Authentication Files

- **Auth File:**
  - Generated by the Bank at application startup.
  - It contains the Bank RSA Public key used to authenticate the Bank and allow the ATM to send its Public key in a secure way (encrypted with the Bank Public key, so only the Bank has the means to decrypt it).
  - Shared with the ATM through a secure and trusted channel unavailable to the attacker.
- **Card File:**
  - Created by the ATM during account creation (with -n parameter).
  - It contains a random 16-byte PIN unique to each account.

3.3. Transactions Flow

- **Initialization:**
  - The Bank generates the *auth file*.
  - ATM reads the auth file and establishes a connection with the Bank.
  - Bank and ATM agree on a session key using the Diffie-Hellman-Merkle algorithm to communicate.
- **Authentication:**
  - The Bank is authenticated at the moment that the ATM can see the contents of the *auth file* (we can also suppose that ONLY the Bank could of written on that file thus from now on is authenticated), encrypt its Public key with the key that's inside of the file and obtains an answer from the Bank.

- The ATM is authenticated when the Bank receives a communication encrypted with its Public Key. Only the ATM could have access to its Public key that is stored inside the auth file.

- **Transaction:**
  - ATM sends an encrypted request (e.g.: deposit, withdrawal).
  - Bank processes the order, updates the balance and responds with encrypted confirmation.
  - Both parties print a transaction summary in JSON.
    Example: {"account": "bob", "deposit": 100.00}

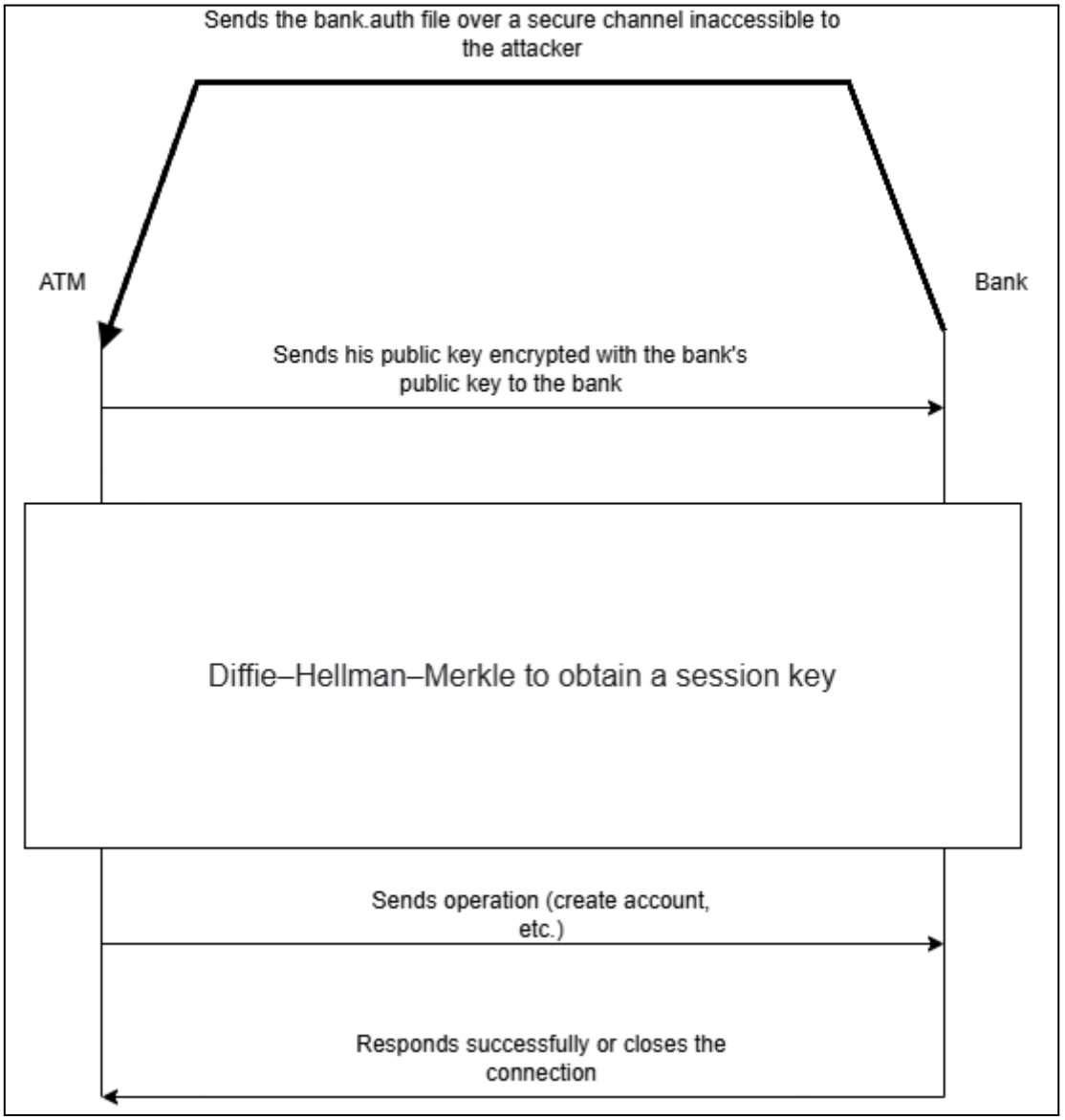

Figura 1: Communication flow diagram.

3.4. Security Protections

- **Confidentiality:** Each communication between the parties is encrypted with AES-256-GCM.

- **Integrity:** The ATM and Bank verify the data received using HMAC-SHA256 to check whether or not the data was corrupted or tampered with.

- **Authentication:** Through the inaccessible auth file we simulate the distribution of the Bank's Public key. After this key is read by the ATM it sends its public key encrypted with the Bank Public key. This way only the Bank could have access to the ATM's Public key as it and only it can have the Private key that is able to decrypt the communication.
  This protocol behaves the same way as a Certificate AUthority would with the tweak that we have a "perfect" file (inaccessible and inviolable) to establish the first (normally insecure) connection properly without the worry that it might be the wrong key.
  An intruder can't impersonate an ATM without compromising the auth file, because this is the proof that the ATM is legitimate.

- **Replay Prevention:** We use a secure generation of a 6 digit code that will be used to be the starting sequence number for every communication that the Bank does. Upon receiving the sequence number sent by the ATM it checks to see whether or not it is the corresponding order, thus preventing a replay attack. This number is also concealed within the message structure that is encrypted.

- **Mutual Authentication:** Implemented via the Diffie-Hellman-Merkle protocol with symmetric key generation.

- **Perfect Forward Secrecy:** With the use of session keys that are thrown away after each connection, by only using the asymmetric key to establish the first connection and encrypting the Diffie-Hellman-Merkle public parameters with the respective Public keys (ensuring that only the rightful receptor can decrypt) we achieve Perfect Forward Secrecy. As such no decrypted message from the past has future implications (if any of the Privates keys are accessible by an attacker this isn't valid) in the communications.

4. Specific Attacks and Countermeasures

4.1. Man-in-the-middle (MITM) / Eavesdropping

Context: An attacker acts as an intermediary between the ATM and Bank. The objective of this attack is to intercept data while it is transmitted over the network and access sensitive data including account balances or transaction amounts.

To combat this, all the communication exchanged between the parties is encrypted with the AES-256-GCM algorithm (ECDHAESEncryption.java line 31), where the secret key is a session key (stored in the program's memory). Only legitimate ATM's have access to this file. Without this key, an eavesdropper or the Man-in-the-middle cannot decrypt the data, thus ensuring confidentiality of the data.
Also a MITM might try to access the exchange of the Diffie-Hellman public parameters, however these are encrypted with the public key of each of the receptors.
An eavesdropper can only infer information because it can't gain access to it.

4.2. Replay Attack

Context: An attacker retransmits a previously recorded legitimate message in an attempt to duplicate an operation, such as deposits or withdrawal.

To ensure that this does not happen, on every communication startup the Bank sends a sequence number that the ATM needs to increment (ATMClient.java line 87). The Bank will maintain a record of sequence numbers that each client should send. This method ensures that the communications aren't able to be repeated, thus only the right sequence is accepted to avoid a replay/reuse of previous messages.

4.3. Message Tampering

Context: An attacker has access to the message that the ATM is about to send to the Bank in order to modify the contents of a legitimate message, for example, changing the value of a withdrawal from 10€ to 1000€.

To avoid the tampering of a message, an HMAC-SHA256 is used, which is computed over the payload of each message (ECDHAESEncryption.java line 39). The recipient checks the HMAC of the unwrapped payload with the HMAC concatenated. Any discrepancy results in validation failure and the session is closed. This approach ensures the integrity of each message communicated and its contents.
Also if an attacker tries to tamper with the message blindly (switching bits/bytes at will) will also fail due to this validation of HMAC's.

4.4. Denial of Service (DoS)

Context: An intruder attempts to overload the system by sending messages repeatedly as an effort to jam or slow down the normal processing of the operations.

In a bid to counter this, the system was designed to compare the sequence number of the Bank with the received message. If the sequence number field of a message is not the corresponding one, the Bank will discard the rest of the operation.

To defend against this kind of attack we designed the system to automatically drop a message if ten or more messages are from the same IP address (In our interpretation a real ATM should never send more than 10 operations per minute), to prevent repetitive attacks from overwhelming the Bank (BankServer.java line 76). This approach promotes the availability and reliability of the system against automated attacks.
Even if an attacker tries to impersonate an ATM (in order to do this it needs to have access to the auth file and be connected from the same IP:port) and launch a DoS attack it will be blocked, because even a legitimate/corrupted ATM shouldn't send many requests per minute (>10).

4.5. Denial of Service (DoS)

Context: An ill intended individual tries to tamper with a card file in order to transfer or withdraw from legitimate accounts. To deny this threat, every card file contains a PIN. When a transaction is received, the Bank verifies if it is equal to the value previously stored for that account (BankServer.java line 291, 205, 318). This will prevent a tampered card file from being accepted and thus maintain existing users and authorizations secure.

5. Final Notes

In our implementation we don't allow card names to have the character "-", because it wouldn't allows us to distinguish a legitimate file from a legitimate operation, for example:
The file name "*this.file-name.pdf*" when it's being processed would flag the "-n" as an operation and it wouldn't be able to store the true file name as a file. In the end the operation would fail because the "*-name.pdf*" is not a valid new account operation.
If the use of spaces between arguments were enforced we would be able to correctly filter the "-", however we have to accept a command like "-n9999" and "-n 9999", for this reason we can't make the correct call every single time about the "-" being either a file name or a operation character.

A limitation of our project is that the IP passed in the arguments will always fail the validation. We only noticed it too close to the deadline.