Afonso Santos – FC56368

Raquel Domingos – FC56378

Miguel Faísco – FC56954

Segurança Aplicada – 2023/24

Project – Phase 1 "Build It" – Group 02

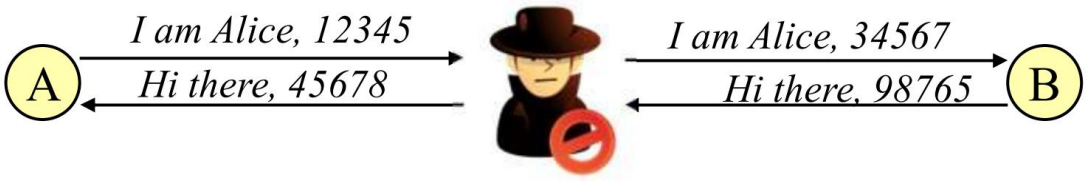
Design Document

Protocol between ATM and Bank

The communication between the ATM and Bank is started using Mutual Authentication with Public Key, followed by the use of Diffie-Hellman to establish a session key.

In the process of Mutual Authentication, the ATM begins by letting the Bank know it wishes to begin the communication process (i.e. the operation that the client using the ATM wishes to start is sent to the bank). The Bank then sends a nonce which is encrypted with the client's Public Key, which the latter has to decrypt with his Private Key. After decrypting the nonce and sending the answer back to the Bank, the ATM also sends his own nonce encrypted with the Bank's Public Key for mutual authentication to be established. Once the Bank sends back the decrypted nonce (decrypted using his Private Key), both parties have been successfully authenticated.

Then, before the ATM sends the value of the operation to the bank (The value the client wishes to deposit or withdraw), a DH session key is established. To prevent any attackers from attempting to establish a Man-In-The-Middle connection during the DH process (like in the image below), the Ya and Yb values exchanged in the



protocol are hashed, digitally signed and sent after their plaintext counterparts. This way, if the attacker was to send a different public key to B, it would also need to find a way to digitally sign it as A.

After both parties have been authenticated and a session key is established, all messages exchanged within the communication make use of HMACs to ensure Integrity and Authenticity. Upon receiving a message and its MAC, the receiving party confirms the MAC and aborts the operation if the confirmation fails. The messages are also encrypted with the session key to offer confidentiality.

Furthermore, all messages also make use of a sequence number, to prevent replay attacks.

Now, a more detailed explanation of the communication steps for every operation will be presented:

Create Account

Starts with the ATM sending the create_account operation to the bank. The ATM will then send its public key to the bank and the mutual authentication process mentioned beforehand will begin, followed by the DH session key establishment. After both the ATM and the Bank have a shared key, it is used for the encryption and MAC of the following messages, namely when the ATM sends its account name and value to deposit to the bank and the bank sends the response back. All the messages before the establishment of the session key are also encrypted with the public key of the recipient.

Deposit and Withdraw

These two operations function in the exact same way. The operation is first sent to the bank (deposit/withdraw), followed by the account name of the client using the ATM. The mutual authentication process begins which allows the bank to confirm that the card file being used belongs to the user. Then, a session key is established via the DH protocol as explained beforehand. The value to deposit/withdraw is then sent to the bank, encrypted with the session key, as well as the MAC of the message. The bank sends its response back.

Get Balance

Starts in the exact same way as the create_account operation or the deposit/withdraw, by a process of mutual authentication and the establishment of a session key with DH. With the session key established, the bank uses it to encrypt the result of the operation and if successful, the balance of the account, and sends it to the ATM (Of course, every message is followed by its MAC).

Implemented Protections

Confidentiality Attacks

Since the start of the communications between the ATM and the Bank, the messages are encrypted using the recipient's public key to provide confidentiality until a session key is established using Diffie-Hellman.

After the Diffie-Hellman shared key is established between the ATM and the Bank, all the messages are encrypted with this key. Since there is a different key generated for every session, even if an attacker can break a previous key, he still won't be able to decrypt the messages sent in future (or past) sessions.

As for the lines of code relevant to this attack, they can be encountered throughout different files, namely AtmStub.java and BankThread.java. One example are these lines from AtmStub.java:

```
108             //Client sends account and value encrypted to server
109             MessageSequence requestMessageSequence = new MessageSequence(Utils.serializeData(requestMessage), messageCounter);
110             encryptedBytes = EncryptionUtils.aesEncrypt(Utils.serializeData(requestMessageSequence), secretKey);
111             outToServer.writeObject(encryptedBytes);
112             messageCounter++;
```

As seen in the code, the actual encryption is done at the EncryptionUtils.java file.

```
83      public static byte[] aesEncrypt(byte[] data, SecretKey key) {
84          byte[] encryptedBytes = null;
85          try {
86              Cipher cipher = Cipher.getInstance("AES");
87              cipher.init(Cipher.ENCRYPT_MODE, key);
88              encryptedBytes = cipher.doFinal(data);
89          } catch (Exception e) {
90              System.exit(RETURN_VALUE_INVALID);
91          }
92          return encryptedBytes;
93      }
```

Integrity Attacks

To protect against integrity attacks and to ensure authenticity for each of these, a MAC is sent right after every message. This way the recipient can calculate the MAC himself and check if the message has been altered.

Again, the relevant lines of code relevant to this protection can be seen both at AtmStub.java and BankThread.java. One example are these lines from AtmStub.java:

```
114             // Create HMAC and send it to bank
115             byte[] hmacBytes = EncryptionUtils.createHmac(secretKey, Utils.serializeData(requestMessage));
116             MessageSequence hmacMessageSequence = new MessageSequence(hmacBytes, messageCounter);
117             encryptedBytes = EncryptionUtils.aesEncrypt(Utils.serializeData(hmacMessageSequence), secretKey);
118             outToServer.writeObject(encryptedBytes);
119             messageCounter++;
```

An example of the verification of the MAC can be seen in these lines of AtmStub.java:

```
217             //Client receives and confirms HMAC of operation result
218             byte[] hmacMessageEncrypted = (byte[]) inFromServer.readObject();
219             hmacMessageSequence = (MessageSequence) EncryptionUtils.aesDecryptAndDeserialize(hmacMessageEncrypted, secretKey);
220             if(hmacMessageSequence.getCounter() != messageCounter)
221                 return RETURN_VALUE_INVALID;
222             messageCounter++;
223
224             hmacBytes = EncryptionUtils.createHmac(secretKey, resultMessageSequence.getMessage());
225             if(!Arrays.equals(hmacBytes, hmacMessageSequence.getMessage())) return RETURN_VALUE_INVALID;
```

Man-In-The-Middle

One protection for MITM attacks is employed during the exchange of the DH session key. The DH public key is sent in the clear since the message size was too big to be encrypted using the recipient's public key. This way, a MITM could possibly tamper with the message and send a different DH public key to the other party. So, as explained beforehand, after the DH public key is sent, the sender sends a digital signature of that DH public key. This prevents the MITM from tampering with it since even if the MITM sends a different DH public key to the recipient, he would not be able to digitally sign it with the sender's private key.

One example can be seen in AtmStub.java where the ATM receives the DH public key from the Bank and checks its signature:

```
473             //Client receives publicKey DH of server
474             MessageSequence receivedPublicKeyDHmessage = (MessageSequence) inFromServer.readObject();
475             if (receivedPublicKeyDHmessage.getCounter() != messageCounter) return null;
476             messageCounter++;
477             byte[] bankDHPublicKey = receivedPublicKeyDHmessage.getMessage(); //DH public key of the bank
478             byte[] dhPubKeyHash = EncryptionUtils.createHash(bankDHPublicKey);
479
480             //Receive signed hash of the server's DH public key
481             MessageSequence dHPubKeyHashMessage = (MessageSequence) inFromServer.readObject();
482             if (dHPubKeyHashMessage.getCounter() != messageCounter) return null;
483             messageCounter++;
484             byte[] bankDHPublicKeySignedHash = dHPubKeyHashMessage.getMessage();
485
486             //Check if it matches the signature from the bank
487             if (!EncryptionUtils.verifySignature(dhPubKeyHash, bankDHPublicKeySignedHash, bankPublicKey)) return null;
```

Replay Attacks

The messages sent between the Bank and the ATM always include a sequence number. These sequence numbers prevent replay attacks during sessions, since if an attacker intercepts and tries to manipulate the messages by delaying or reordering them, the sequence number can reveal this tampering. Before sending a message to the Bank, the ATM randomly generates a sequence number that is sent along with the first message. The following messages will increase the sequence number.

This can be seen throughout the source code, the generation of the sequence number can be seen in the AtmStub.java file, at the beginning of every operation's function. One example is in the following lines of the AtmStub.java file.

```
53          byte[] nonce = EncryptionUtils.generateNonce(8);
54          ByteBuffer bb = ByteBuffer.allocate(nonce.length);
55          bb.put(nonce);
56          bb.flip();
57          messageCounter = bb.getLong();
```

Then, the message counter is incremented after every message sent or received (One example of this can be seen at the image in the Confidentiality Attacks section).

Message Format

For the messages sent between the Bank and the ATM, a MessageSequence object was created. This object has two fields, the actual content of the message in bytes and the sequence number of the message. These messages are mostly encrypted, the exception being when Public Keys are sent as the content of the message, since Public Keys have a bigger size than the one accepted for the encryption. In all the other cases, the messages are encrypted with the recipient's public key or a session key if one has been established. This MessageSequence object is used for every message exchanged.

Card and Auth File Explanation

As mentioned in the requirements, the bank has an auth file that will be shared with the ATM. In our implementation, the auth file contains a serialized PublicKey object. This way, upon startup, the ATM has access to the bank's public key and can use it

during the authentication phase, making the bank decrypt a challenge that was encrypted using the public key in the auth file.

As for the card file, it is generated upon the create_account operation and it is a serialized KeyPair object. This way, the client can send its PublicKey to the bank (that the bank will store) and decrypt using his private key. When the ATM is started again, the client uses his card file and therefore has access to his keys and can decrypt the messages. If a different card file is provided, then the client won't be able to authenticate and the communication will fail (As it is supposed since that, per requirement, the card file must be associated with the account performing the operation).

Failed / Partial Implementations

Diffie-Hellman Calculations Exchange

Regarding the Diffie-Hellman protocol and the establishment of a session key, the group intended to encrypt the Y values (the DH public keys) with the other user's public key for confidentiality, but this strategy was eventually put to the side due to encryption protocol limitations since the message that the group was trying to encrypt was too big.

Timestamps in Authentication

The use of timestamps was thought about extensively for the communication between the Bank and the ATM, but it ended up not being implemented due mostly to the amount of benchmarking that would have to be made to ensure the accepted delay was large enough to accept correct messages and small enough to discard tampered ones), but also due to the fact that it would be highly hardware dependent, and the calculated delay in a machine could not be appropriate for other. Regardless, the group wished to have this feature implemented to better avoid replay attacks and MITM attacks.

How to run

The project was implemented using Java and therefore we provide two .jar files. One jar file to run the Bank and one jar file to run the ATM. They are named Bank and ATM respectively.

To run the Bank, use:

java -jar Bank.jar -p <port> -s <auth-file>

(The port and the auth-file are optional arguments as said in the project description).

To run the ATM, use:

java -jar ATM.jar -a <account>

(The account parameter is required, other parameters described in the project description can be used following the same flag logic).

The necessary jar files (Bank.jar and ATM.jar) are in the root directory of the project. A parameter for the operation is always necessary, as per the project description.

After running the Bank.jar file. The generated auth file should be placed in the directory where the ATM.jar will be run.