Project - Build It Phase

Diogo Maia Fc54449
João Galveias Fc54459
Rodrigo Martins Fc54479 Group 8

[1]Faculdade de Ciências da Universidade de Lisboa

Segurança Aplicada - 2022/2023

April 16, 2023

1. Our approach

We decided to approach this problem using Java 17, where we created 3 main files: MBeC.java, Bank.java, Store.java, these files have the function to represent each of the endpoints.

The communication is done using TCP sockets, we wanted to use TLS for better security but due to the file limitations it wouldn't be possible, so we decided to use the standard sockets. The bank works only as a server, while the store works as a server (for the MBeC) and as a client (for the bank), the MBeC works only as a client. To start the system we have to run the bank, and then we can run the store and the clients. When started the bank generates its keys, the public key and the private key, the public key is placed in the default bank.auth file (or whatever you decide when you run the bank), and that file is shared with the client and the store. In our implementation we have only one file, there is not even an .auth file in each folder, just for simplicity and because we can consider that this file cannot be the victim of an attack by someone with malicious intentions.

When the client executes an operation either directly to the bank or to the store, these two are always running and waiting for connections, after the MBeC executes this operation a socket is created between the MBeC and the server (either store or bank) and the bilateral authentication process is initiated.

First step The client starts a conversation with the server, sending him is id, an auto-generated challenge and its public key (the client public key), all encrypted with **Bank's public key**. **Second step** The server gets the client public key, and after solving the challenge previously sent, it will send to the client another auto-generated challenge along with the solution of the client's challenge for the bank, all encrypted with **Client's public key**. **Third step** In this step, the client will solve the challenge sent by the Bank, and respond to him, this solution, encrypted with **Bank's public key**. **Last step** if the solution is correct, the Bank will create a symmetric key to be used in future communications in this client's session and will send it to the client, encrypted with **Client's public key**. In all these steps, all the messages were sent along with its **hash** in order to guarantee integrity.

This process is performed so that the client knows that it is communicating with the bank, using the public key in the .auth file, and also proving to the bank that we are the user we say we are, thus preventing impersonation and replication attacks due to the use of the symmetric key exchanged by both and the nonces.

The messages between the various endpoints (MBeC, Bank and Store) are always arrays of bytes, these contain the information needed for each operation. This information is a string orginally, which is divided by a specific group of characters.

After authentication is performed and making use of the symmetric key generated from these authentication steps, this key is used for the rest of the communication so that the operation, nor any of its constituents, ever passes through the channel without any cipher.

Additionally, to maintain the integrity of the messages exchanged, we use hash mechanisms that use SHA-256 to generate a hash of the message that will be sent, this hash is sent inside the ciphered product, so that when received and after decrypting the payload received, the hash is performed, with the components that are shared by the sender in the same message, of the received message (excluding the hash and the components for the hash function that come from the message) and after that compared to the received one, so any attempt to alter messages over the network, by malicious users is blocked because if the messages do not have the same hash an error is raised and the execution is terminated.

1.1. File Explanation:

- **Auth file:** The authentication file as mentioned before is created every time the bank is started, and an .auth file is never overwritten, i.e. if the file already exists the bank throws an error and terminates execution. This file in our approach serves for the bank to share its public key with the machines it knows are safe and not malicious, this public key is later used for our authentication mechanism already explained above, so that the endpoints can get a symmetric key for their communication.
- **User file:** The user file is created on execution of the "-n" operation, where MBeC creates a client in the bank, this operation creates a new user file with the number defined in the execution command, if it doesn't exist yet, otherwise it returns an error. This file is used to store the public and private keys of each user, so that they can be used in each operation after the creation of the user.
- **Vcc file:** The VCC file in our approach does not need to store any information, it is a file that merely represents the existence of a card, this occurs because the server stores all user cards, so only with the name of the card or even just the MBeC number it is possible to see which is the only card that is active, that is the card with the user number and the largest sequence number (since it is only possible to have one active card at any time).
  We understand that this file could be important in case we want to validate if the user who uses it actually has the permission to do so, in our approach we did not take into account this security problem, but we think it would be a good step to take in future iterations in order to be sure that each card was only used by the user who owns it, this could be done with pins or passwords.
  In our approach we just check that the user using it has the same number that is in the given vcc.
  The vcc file is created when the "-c" command is executed, there is no initial check to verify that the user has enough balance to cover the value of the card, we chose to let the user create the card with the value they want regardless of their balance,

and then only when the card is used in a store is the customer's balance verified in order to understand if the operation in the store can be done or not, a bit like how an MBway works.

2. The four specific vulnerabilities we defend

2.1. Personification (Man-in-the-middle) (Implemented)

Authentication using public and private keys, also known as public key cryptography, is a secure way to establish the identity of a user or system in a communication process. This process prevents personification, which is when an attacker pretends to be someone else in a communication, by using a unique combination of public and private keys.

The authentication process begins with the bank generating a public-private key pair when started, where the public key is shared with the other parties involved in the communication via the auth file. The public key is used to encrypt messages that are sent to the user, and the private key is used to decrypt these messages.

When the user needs to authenticate themselves to another party and vice versa, they send a message encrypted with the public key from the auth file. The receiving party then uses the his private key to decrypt the message, which verifies that the message was indeed sent by the user and not by an attacker.

We also implemented a set of challenges so that the MBeC can be sure the bank is really the bank, since the bank will solve the challenge and then send the solution ciphered with his private key, so then the MBeC can Decrypt it with the public key from the auth file.

To further prevent personification, the parties can use this authenticated communication to securely exchange a symmetric key, which can be used for future communication between them. This symmetric key generated by the server can be encrypted using the private key of the server, ensuring that only the receiving party with the auth file can decrypt the key and use it for future communication.

Once the symmetric key is exchanged, the parties can communicate using symmetric encryption, which is faster and more efficient than public key encryption. By using this process of authentication and symmetric key exchange, the parties can establish a secure and trusted communication channel, preventing personification by attackers.

Overall, authentication using public and private keys, combined with the secure exchange of a symmetric key, is a reliable way to prevent personification and establish secure communication channels in a wide range of applications, from online banking to secure messaging platforms.

You can see the authentication code in the following lines of the ClientThread.java from bank, from line 63 to line 205, where you can see the described autentication protocol, with the 4 phases:

1. Gather the public Key
2. Creates challenge, solves it and send the challenge to the other party
3. Receives the solution and compares it so his solution, if equals jumps to phase 4
4. Creates a symmetric key and shares it using his public key to chipher

2.2. Integrity (Implemented)

In our approach we implemented the mechanism of sending a hash of a message along with the message itself, and then encrypting the combination, this a common way to protect against integrity problems in secure communication. This approach ensures that any tampering with the message is immediately detected by the receiver.

A hash is a fixed-length digital representation of a message, which is calculated using a hashing algorithm. This algorithm generates a unique hash for each message, and even small changes to the message will result in a completely different hash. By sending the hash along with the message, the receiver can calculate the hash of the received message and compare it to the original hash to verify the integrity of the message.

The cipher used either with Public-private keys or symmetric keys, ensures that only the intended receiver can access the message and its hash, preventing any unauthorized tampering with the message.

In this way, sending a hash of the message along with the message itself, and then encrypting the combination, provides a high level of protection against integrity problems. Any attempt to alter the message, either intentionally or accidentally, will be detected by the receiver due to the change in the hash value. Additionally, encrypting the combination of the message and its hash ensures that only the intended receiver can access the message, ensuring confidentiality and further protecting against unauthorized tampering.

We used always the same function to generate the hash that can be found in all the files, in the bank (in the file ClientThread) the same lines previously mentioned, where all these protections end up occuring at the same time.

```
public static String getHash(byte[] inputToHash) throws NoSuchAlgorithmException, IOException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");

        for(byte b: inputToHash) {
                md.update(b);
        }

        byte[] bytes = md.digest();

        StringBuilder sb = new StringBuilder();
        for(int i=0; i< bytes.length ;i++)
        {
                sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
        }

        return sb.toString();
}
```

The function uses the SHA-256 hashing algorithm, which is a strong cryptographic hash function. In addition, uses a salt to the input data, which makes it less vulnerable to brute-force attacks.

However, the way the function updates the message digest is not secure as we know it, the for-loop in the function updates the message digest one byte at a time. This

approach is vulnerable to length extension attacks, where an attacker can extend the hash with additional data without knowing the original data. This can be used to generate fraudulent messages that have the same hash as the original message. Due to lack of time we did not have time to correct.

You can see the implementation in the ClientThread.java from bank, in for example lines 80 to 88, and in line 447 of the mbec.java.

2.3. Brute-force attack (Implemented)

In our approach we developed a function that generates a public-private key pair using the RSA algorithm with a key size of 4096 bits. The RSA algorithm is based on the mathematical problem of factoring large prime numbers, which is believed to be a computationally difficult problem. In other words, the RSA algorithm relies on the fact that it is very hard to find the prime factors of a large number.

The key size of 4096 bits used in this code is considered to be extremely secure and is currently recommended for applications that require strong security, such as online banking and e-commerce. The number of possible key combinations for a 4096-bit key is so large that it is practically impossible to brute force the key within a reasonable timeframe, even with the most powerful computers available today.

Additionally, the SecureRandom class is used to generate a random seed for the key generation process, which adds an additional layer of randomness to the key generation process. This makes it extremely difficult for an attacker to predict the key or to reproduce the same key that was generated using the same algorithm and seed. You can find the code that generates those keys in the file bank.java (called in line 140), store.java (called in line 50) and MBeC.java (called in line 182), like you can see here:

```
177         public static KeyPair generateKeyPair()
178                 throws NoSuchAlgorithmException, NoSuchProviderException {
179
180             KeyPairGenerator keyGen = KeyPairGenerator.getInstance(ALGORITHM);
181
182             SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
183
184             // 512 is keysize
185             keyGen.initialize(4096, random);
186
187             KeyPair generateKeyPair = keyGen.generateKeyPair();
188             return generateKeyPair;
189         }
```

The symmetric key is also secure against brute force using AES key with 256 bits,since enerating an AES key with a length of 256 bits using a secure random number generator is considered secure against brute-force attacks.

A brute-force attack involves trying every possible key until the correct one is found. However, with a key length of 256 bits, the number of possible keys is so large that it would take an infeasible amount of time and computing power to try them all.

The number of possible keys with a length of 256 bits can be calculated as 2^{256}, which is approximately 1.15×10^{77}. This number is so large that it is practically impossible to try every possible key using a brute-force attack, even with the most powerful computers or specialized hardware. as we can see in the ClientThread.java from the bank:

```
392         public static SecretKey createAESKey()
393                 throws Exception
394             {
395
396                         String AES = "AES";
397
398                 // Creating a new instance of
399                 // SecureRandom class.
400                 SecureRandom securerandom
401                     = new SecureRandom();
402
403                 // Passing the string to
404                 // KeyGenerator
405                 KeyGenerator keygenerator
406                     = KeyGenerator.getInstance(AES);
407
408                 // Initializing the KeyGenerator
409                 // with 256 bits.
410                 keygenerator.init(256, securerandom);
411                 SecretKey key = keygenerator.generateKey();
412                 return key;
413             }
```

2.4. Timing modification (Not Implemented)

Timing attacks are a type of side-channel attack that can be used against various encryption algorithms, including RSA. These attacks are based on measuring the amount of time that certain cryptographic operations take to be performed. By analyzing these measurements, an attacker may be able to infer information about the secret keys being used in the encryption process, potentially leading to a security breach.

One way to prevent timing attacks in RSA encryption, and the way our approach uses is by using timeouts to limit the amount of time a cryptographic operation can take to complete. By enforcing a fixed maximum time for cryptographic operations, the timing attack is prevented because the snooper is unable to measure the time differences needed to infer the encryption key.

For example, consider an RSA decryption operation that takes longer to complete when the private key has a certain value. A snooper could measure the time it takes for this operation to complete and use this information to make educated guesses about the private key. By setting a timeout for the decryption operation, regardless of the key value, the snooper cannot use timing analysis to deduce information about the private key.

2.5. Sequence modification/Replay attack (Implemented)

Using sequence numbers in ciphered messages is a common technique used to protect against sequence modification attacks. These attacks involve an adversary intercepting and modifying the order of messages, with the goal of disrupting the integrity and authenticity of the communication.

By assigning a unique sequence number to each message, the receiver can detect any modification or reordering of messages that may have occurred during transmission. This technique is often used in conjunction with encryption to provide additional security.

When sending a message, the sender first assigns a sequence number to the message and then encrypts the message along with its sequence number. When the receiver receives the message, they first decrypt the message and extract the sequence number. The receiver then compares the received sequence number to the expected sequence number to verify that the message was not modified or reordered during transmission, or not or that the same sequence number has not yet been received (Replay attack).

If the received sequence number does not match the expected sequence number or the or sequence number has already been received, then the receiver knows that the message was either modified or reordered, or replayed. The receiver can then take appropriate action, such as requesting a retransmission of the message or terminating the communication.

We had already implemented some things necessary for this protection, but it was not possible to finish in time for the first delivery, so we can show the next function that would be used to generate the nonce from the timestamp (There may be better options). Then a nonce would be added to each message sent, the endpoints would store nonces already received from each other party and if a duplicate was received an error would be generated.

```java
public static byte[] generateNonce() {
    long timestamp = System.currentTimeMillis();
    byte[] nonce = new byte[8];
    nonce[0] = (byte) (timestamp >>> 56);
    nonce[1] = (byte) (timestamp >>> 48);
    nonce[2] = (byte) (timestamp >>> 40);
    nonce[3] = (byte) (timestamp >>> 32);
    nonce[4] = (byte) (timestamp >>> 24);
    nonce[5] = (byte) (timestamp >>> 16);
    nonce[6] = (byte) (timestamp >>> 8);
    nonce[7] = (byte) timestamp;
    return nonce;
}
```

Update: After the first delivery, we apply this protection as we have described, the nonces are created by the client and placed in the operation message after the authen-

tication steps, thus preventing the possibility of replaying the messages that change the state. On the bank/store side, each received nounce is stored, and when a new message is received, it is checked if that nounce has already appeared in some communication.

The nounces are created using the function present in lines 655 to 667, the nounce is added to the message in the operation as visible in line 607. After that the nonce is checked in line 206 of the bank .