Highly Dependable Systems

Universidade de Lisboa - Instituto Superior Técnico
Mestrado Bolonha em Engenharia Informática e de Computadores 2021/2022

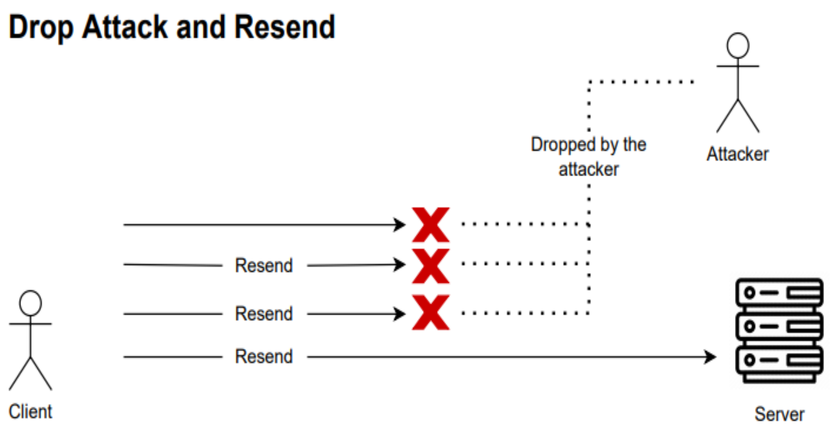
Bruno Silva 102172
Guilherme Saraiva 93717
Maria Gomes 102203

Attacks and Security

The following are possible attacks and how we protect against them:

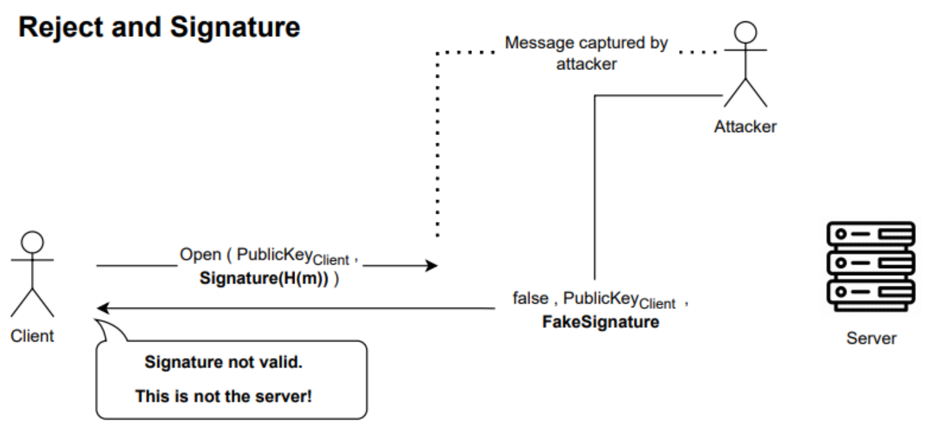
- **Dropping**: If an attacker is able to intercept a message between client and server, he might capture and drop it causing confusion amongst them.

  Solution: Because it is very difficult to guarantee that a message cannot be dropped, our solution to this attack is to resend the message until the receiver gets it. The attacker can continue dropping the messages, but our system will continue resending them until it has confirmation that they arrived.



- **Rejecting:** An attacker can receive a message and pretend to be the receiver, sending a reply that rejects the request made by the sender.

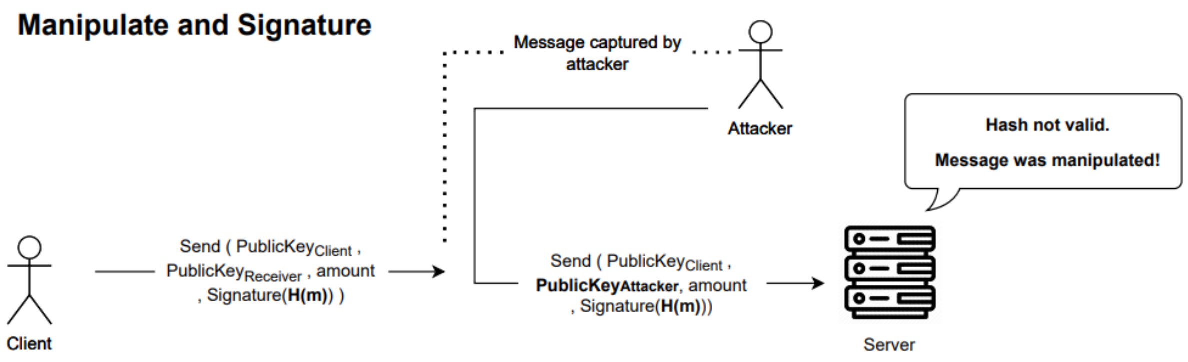
  Solution: To guarantee that the attacker cannot reject messages, every message sent includes a signature of the respective sender. This signature is signed with the sender's private key, which guarantees that the user created that message. On the receiver's side, the signature is verified using the respective public key. If the public key does not open the signature, then it can be assumed that the message was not created by the "real" sender.


Reject and Signature
Message captured by attacker
Attacker
Open (PublicKey_Client , Signature(H(m)))
false , PublicKey_Client , FakeSignature
Client
Server
Signature not valid.
This is not the server!

- **Manipulating**: An attacker might intercept messages and perform modifications to it, changing its content and sending it again.

  Solution: To protect against manipulation, our system guarantees integrity by signing every message (digital signature using the sender's private key) with a hash of its content. Because the attacker cannot replicate the signature, he might alter the content that comes on the message, but the content inside the signature will remain unaltered. Upon receiving the message, the receiver verifies the signature.

  If the hashes do not match, it assumes that the message was altered and rejects it.


Manipulate and Signature
Message captured by attacker
Attacker
Hash not valid.
Message was manipulated!
Client
Send (PublicKey_Client , PublicKey_Receiver , amount , Signature(H(m)))
Send (PublicKey_Client , PublicKeyAttacker, amount , Signature(H(m)))
Server

- **Replay**: An attacker can observe requests between the client and server and duplicate them, resending and possibly causing the request to occur multiple times.

  Solution: To prevent duplication, requests are sent with a nonce and a timestamp (and a signature with a hash of the nonce and timestamp inside to prevent modification).

  The nonce can only be used once so any messages with the same nonce and timestamp will be rejected. Using timestamps with the nonces avoids eventual nonce collision and makes it faster and more manageable for the server to detect duplicates.

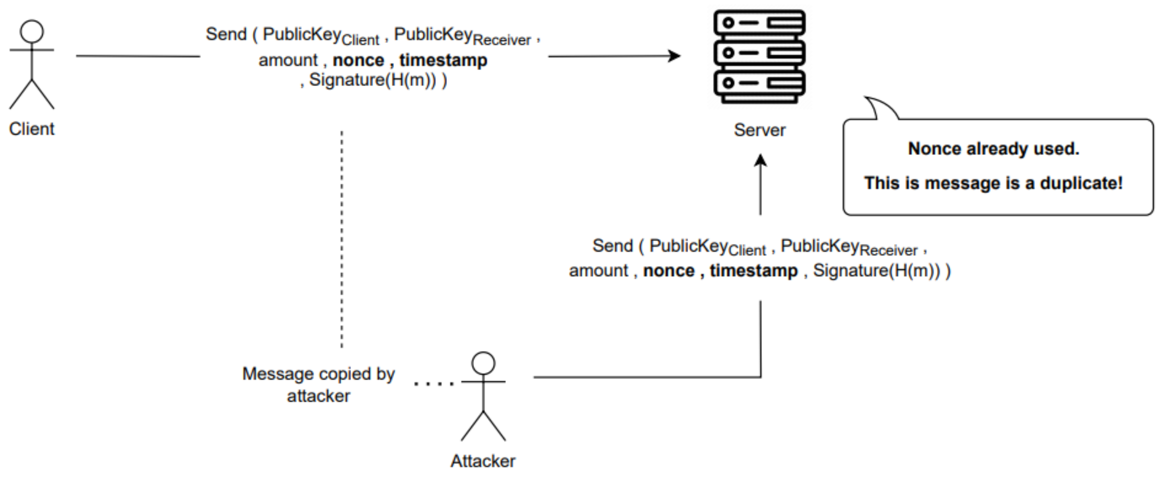
  On the server side, if the timestamp is due, the server rejects the message. There is a chance for clock desynchronization between client and server so, for this project, we added a period of 10 minutes in which the sender must send the request (in real life this period would be bigger).

  On the client side, if there are nonces involved in the request, the client checks if the nonce sent by the server matches its nonce+1 and the timestamp is correct. In requests where there is no nonce the server:

  - sends a timestamp **OR**
  - sends the public key of the sender

  And the client checks for validation.

Duplication and Nonce


Client
Send (PublicKey$_{Client}$, PublicKey$_{Receiver}$, amount , **nonce** , **timestamp** , Signature(H(m)))
Server
Nonce already used.
This is message is a duplicate!
Send (PublicKey$_{Client}$, PublicKey$_{Receiver}$, amount , **nonce** , **timestamp** , Signature(H(m)))
Message copied by attacker
Attacker

Integrity

Our system provides integrity by signing every message (both client and server) with digital signatures, hashed with the message's content.

To ensure that the message was not altered, both client and server need to hash the message's content and compare it to the signature's content. If the hash is the same, it means no modifications were made, since the attacker cannot replicate the signature of the client or server.

Dependability

To guarantee that the requests are being made by a bank client, the client must log in using an username and password.

The system offers non-repudiation by signing most functionalities with the sender's digital signature. Because only the sender is able to have its own private key, it is guaranteed it was he who signed those messages. Only two requests do not use digital signatures (check and audit on the client side) because these functionalities can be made by any client of the bank. To ensure that it was a client of the bank that made these requests, our system requests that the user logs in upon opening the application.

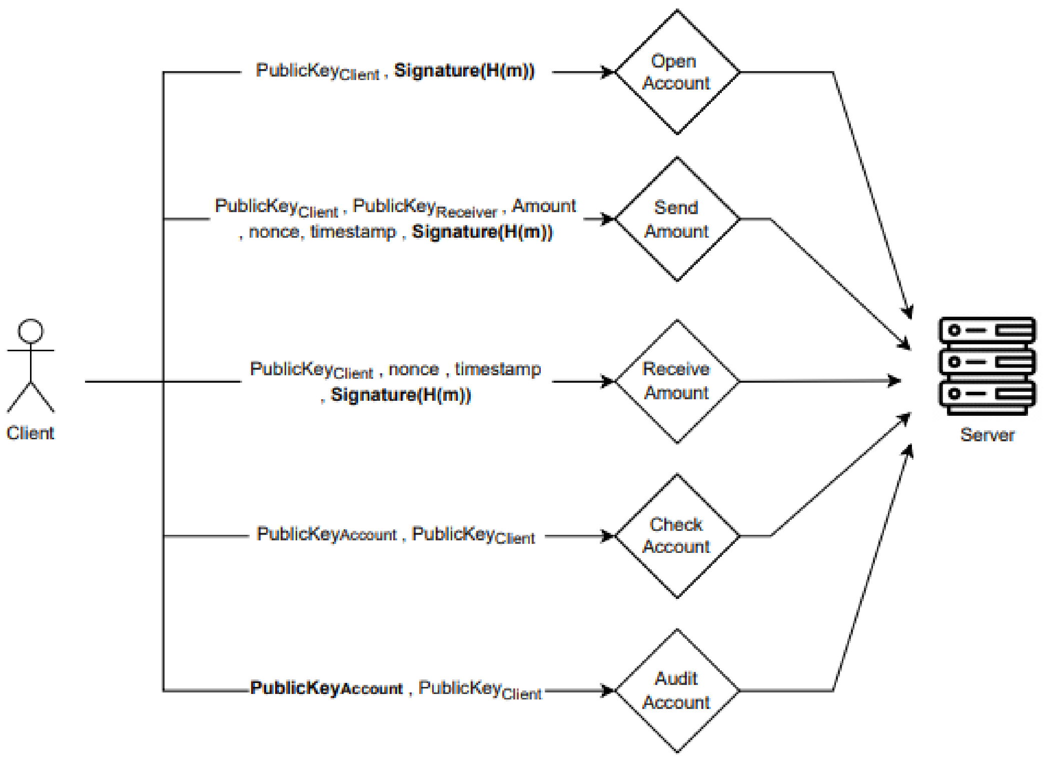
The system does not offer confidentiality since any client can check and audit other client's accounts.

Besides that, the system is stateful meaning that if the server is rebooted, it will keep all the information about its clients. This was implemented by doing atomic writes on the file that stores the data.

The system

Below are two schemes of the whole system - Client Requests and Server responses.

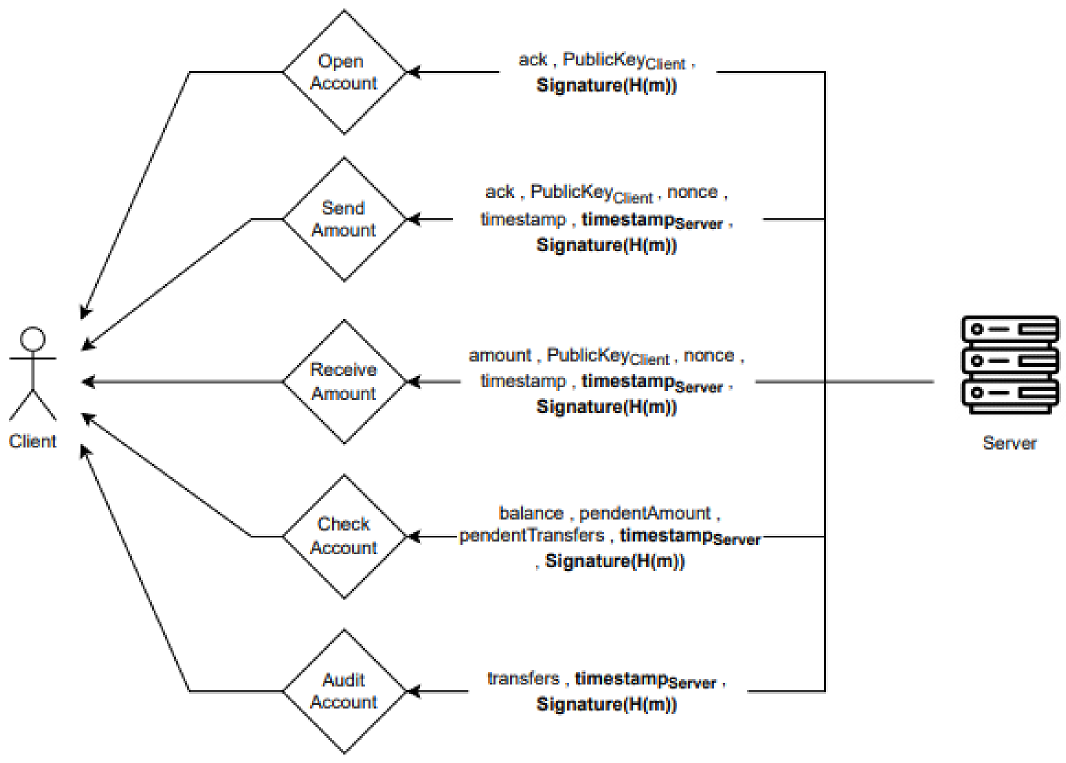
Client Requests


PublicKey$_{Client}$, **Signature(H(m))**
Open Account
PublicKey$_{Client}$, PublicKey$_{Receiver}$, Amount , nonce, timestamp , **Signature(H(m))**
Send Amount
Client
PublicKey$_{Client}$, nonce , timestamp , **Signature(H(m))**
Receive Amount
Server
PublicKey$_{Account}$, PublicKey$_{Client}$
Check Account
PublicKey$_{Account}$, PublicKey$_{Client}$
Audit Account

Signature(H(m)) - represents an Hash of the message cyphered with the client's private key
The messages are the following:
Open - PublicKey$_{Client}$
Send - PublicKey$_{Client}$, PublicKey$_{Receiver}$, Amount , nonce, timestamp
Receive - PublicKey$_{Client}$, nonce , timestamp

PublicKey$_{Account}$ - in check account, PublicKey$_{Account}$ is the public key of the account the client wants to check

Server Responses


Open Account
ack , PublicKey_Client , Signature(H(m))
Send Amount
ack , PublicKey_Client , nonce , timestamp , timestamp_Server , Signature(H(m))
Client
Receive Amount
amount , PublicKey_Client , nonce , timestamp , timestamp_Server , Signature(H(m))
Server
Check Account
balance , pendentAmount , pendentTransfers , timestamp_Server , Signature(H(m))
Audit Account
transfers , timestamp_Server , Signature(H(m))

Signature(H(m)) - represents an Hash of the message cyphered with the server's private key
The messages are the following:
Open - ack , PublicKey$_{Server}$
Send - ack , PublicKey$_{Server}$, nonce, timestamp , serverTimestamp
Receive - balance , PublicKey$_{Server}$, nonce , timestamp , serverTimestamp
Check - balance , pendentAmmount , pendingTransfers , serverTimestamp
Receive - transfers , serverTimestamp

timestamp$_{\textbf{Server}}$ - represents a timestamp created by the server to ensure that the server has responded within the timeframe