SEC Report - Group 21

Diogo Dias 90792

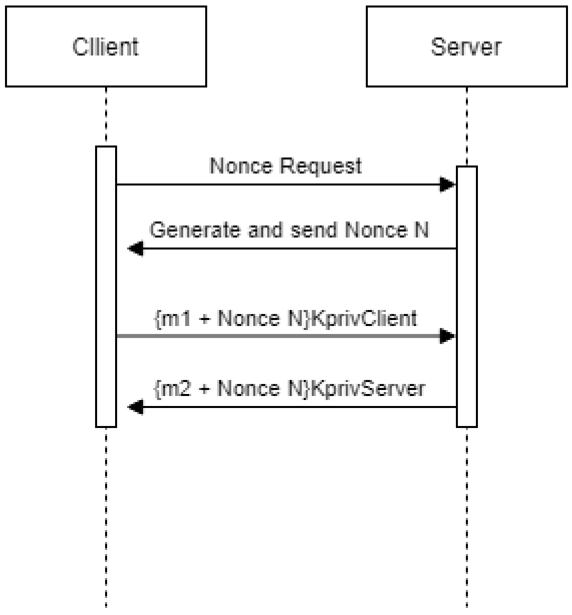
Gonçalo Velhinho 90718

Miguel Mota 90964

Communication

The communication between Server and Client is implemented using JSON strings and Java Sockets.

Java objects were converted to JSON using the GSON library. The messages are represented as JSON objects and then are digitally signed by the sender. A random number is requested from the receiver, which is added to the JSON object before signing. This number is generated from Java's SecureRandom class. The receiver checks if the signing corresponds with the public key and checks if the random number is the same as the one provided. This guarantees freshness, preventing replay attacks.


Cllient
Server
Nonce Request
Generate and send Nonce N
{m1 + Nonce N}KprivClient
{m2 + Nonce N}KprivServer

This communication protocol ensures that only the people with the private key can sign messages, and it ensures integrity of the message since the whole message is signed. The signing of a requested random number constitutes a challenge-response mechanism which ensures freshness. This set of guarantees ensures non-repudiation.

Private and public key pairs are stored in a secure password-protected JCA KeyStore generated by each client using Java's keytool. Private keys are generated using 2048 bit

RSA. We also assume prior distribution of the public keys, so every client already knows each other's and the server's public keys.

If an attacker drops, manipulates or duplicates messages the server rejects these messages, thus creating a potential denial of service. We do not address this problem since it seems outside the scope of this project. We however address some aspects of Denial of Service, by the server having absolutely no state for a client, which makes the use by a large number of clients possible. It does this by when executing commands it creates a new socket and closes it when the response from that command has been received.

Persistence and Atomicity

The data is recorded in an SQLite database called SEC.db. Persistence is ensured since the data is written to a file, so in the event of a server crash the data is still safely stored. Data consistency is also guaranteed from the SQLite transactions, which make sure that in the event of a server crash only the committed data persists in the file or is rolled back.. The transactions also ensure atomicity during concurrent writes/reads.

Pros and Cons

- Using JSON as a message format has memory and performance issues but it’s simple to use and also facilitates debugging since everything is a human-readable string.
- Using a challenge-response mechanism for freshness is also simple but it doubles the amount of messages required for communication.
- SQLite easily ensured two of the design requirements (persistence and atomicity) but compared to other databases it has worse performance. However SQLite is used alot by Server-Side Databases.

Considering the scope of the project and the time limit for submission, we decided to favour simplicity over performance. For this reason, the choice made above seemed reasonable.