Highly Dependable Systems
Project - 2st Stage
Report

João Tomázio, ist178039
João Leite, ist177907
António Freire, ist177969

Design

Our Password Manager is divided in 4 major projects: The Client, the Server, the KeyStore and the Lib. It is a RESTful system, so the Client and the Server communicate over HTTP, exchanging JSON Objects. For this process, we are using the Unirest Framework API in the Client-side to send requests, and the Spark Framework API on the Server-side to answer them. Both client and server depend on the Lib Project, that is a library that contains cryptographic methods and other shared code between the two sides of the system, and on the Keystore Project, an independent Key Manager, that generate and provides the client and the server a pair of Public/Private Keys.

Our system use a N clients to N Byzantine Atomic register servers. When a user requests a INIT, REGISTER, PUT or GET, requests are sent to all servers present on the system (identified by a different port: starting on 8080). We implement a 2f+1 faults system (minimum of 3 active servers for 1 possible fault). The client sends a password signature (with asymmetric digital signature) on every PUT and verify that signature on GETs. For all requests, we have the consensus number of $\frac{2N-2}{3}+1$ for N of total replicas.

We tried to implement a secure system, so we used Asymmetric Cryptography to encrypt sensitive data exchanged between clients and servers. We also used some hash methods, digital signature with asymmetric keys and HMAC.

- For Encryption, RSA in ECB Mode with No Padding on domains and usernames and PKCS1Padding to passwords, using Public/Private Keys of 2048 bits long. The No Padding choice was because we needed a deterministic output of the encryption given the same Key and PlainText, to use as identifier on the server's database entries (domain/username);
- For Hash, the safer available, SHA-512;
- Every message sent by the clients on our system are signed with Digital Signature (RSA over SHA-512). The server verifies the signature for sensitive requests like PUTs and GETs.
- The server signes sensitive responses (like REGISTERs and GETs) with HMAC. A key is shared on INIT between servers and a client.

Implementation

In a first step, a pair of asymmetric keys must be generated on the independent generator - KeyStore - for both of sides. These keys will be used on the pure RSA Encryption and Digital Signature signing on the client.

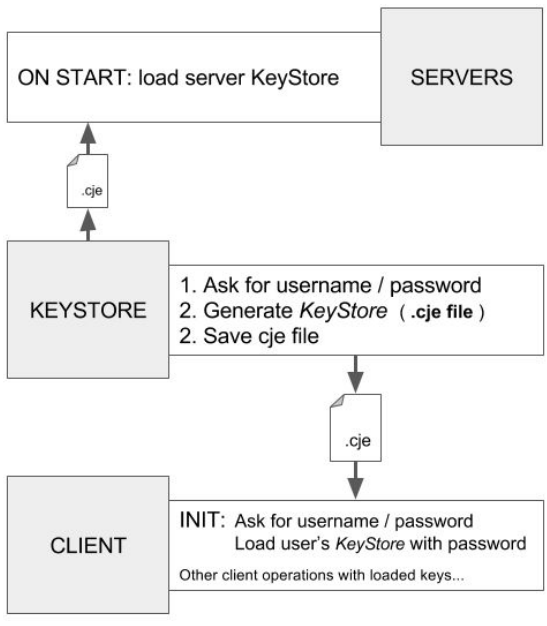

Fig. 1: Using KeyStore

For starting our app, we have two scenarios: The cold-start scenario, the first time the user launches the Password Manager, and the reboot scenario, when a registered user uses the app. In the first case, we have the following exchange of messages:

- **INIT**: This is an automatic request sent by the client to all servers after login on the app. It sends a JSON Object with the client public key (CPk) on the body and the signature on the header. When the message reaches the servers, they will check if the user exists by searching for its CPk on the database. Because it's the first time, the servers return an error reporting the client is not registered;

- **REGISTER**: The client sends now the same JSON to the right endpoint of the servers (/register). The servers receive it and check if the user is already registered. Due to first time, they register the user, set their SeqNumber to 0 and create a HMAC Manager for that client. The response of the server is a JSON with client Seq and HMAC Key create for that specific;

  From this point, the client can now start to save passwords.

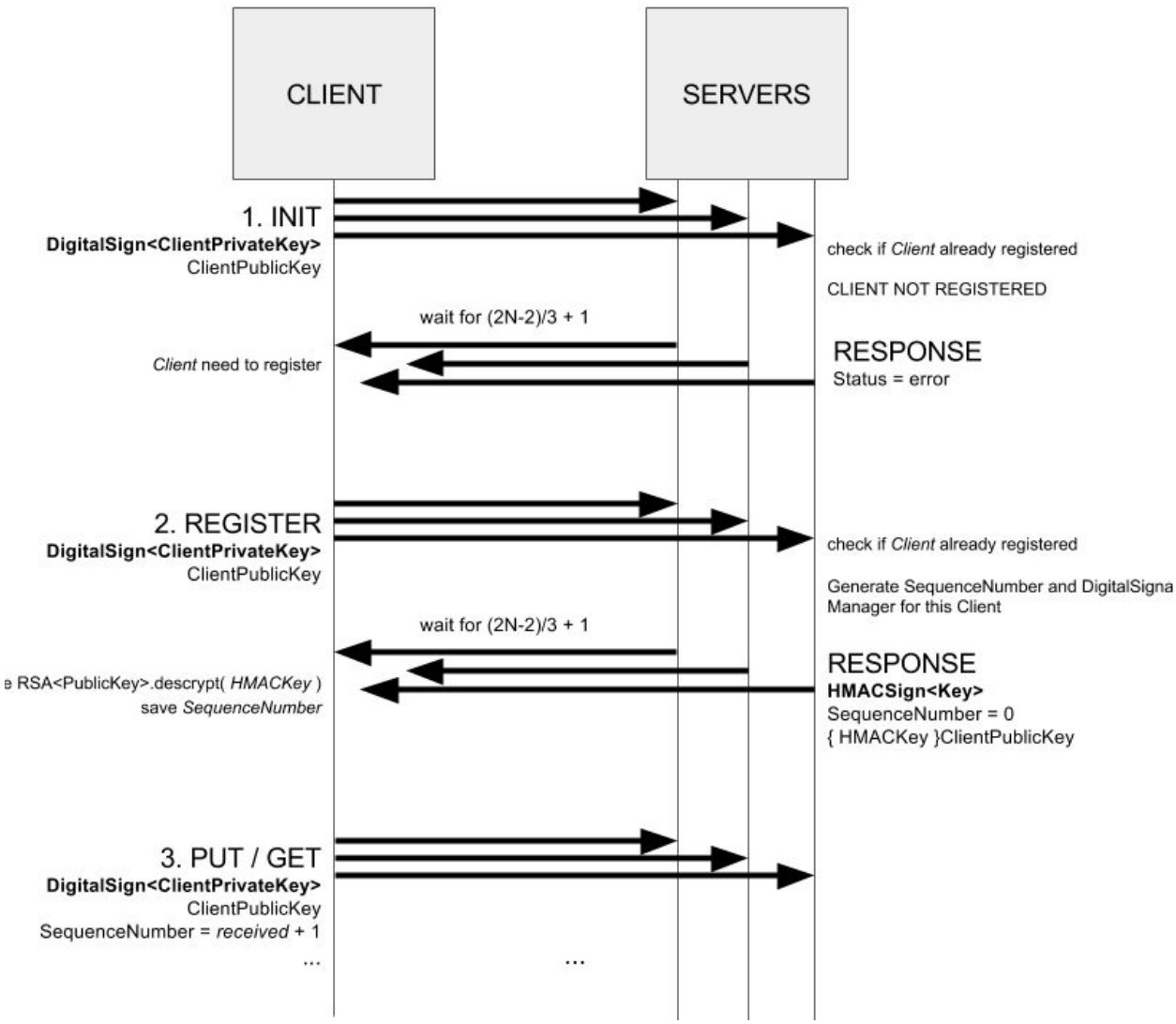

Fig. 2: Cold-start scenario

When the app loads from a reboot, with the user already registered:

- **INIT**: The client sends a request with CPk to the servers. The client is already registered so the response contains the Seq of the last request of the client (before reboot) and the SPk. The client need to save a different Seq Number for every server;

  And now the client can continue to use the app.

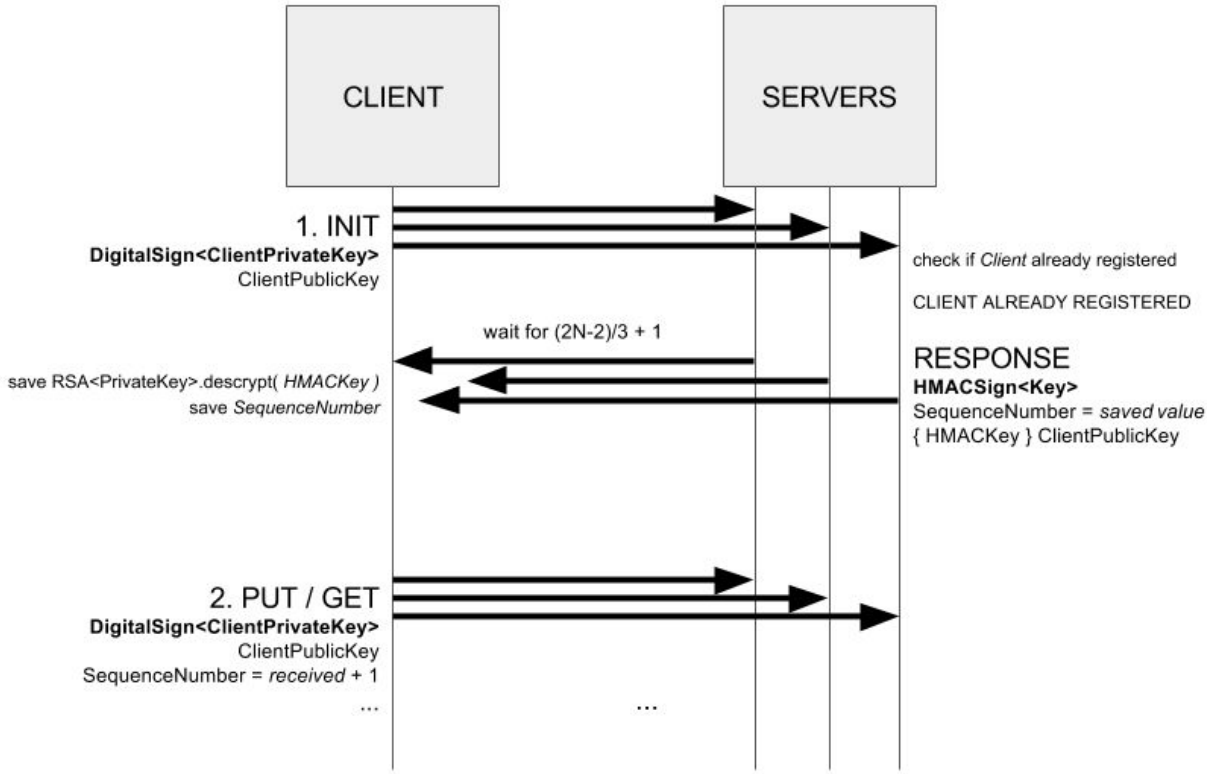

Fig. 3: Reboot scenario

Our project implements a (N,N) Atomic register. The client always make a PUT request to all servers after a GET request. In this way, the client help the servers synchronize the data with the most recent values. If the client wants to PUT a password to the system, it will need to make a GET request first to know the most recent timestamp. Them, it can make the PUT request with timestamp+1 and with a signature of the new password (for later checking in GET requests)

For storing passwords, the client sends a request with the Seq+1 (next request), CPk to identify the client, domain and user encrypted and hash over it, and the password encrypted. The requests contains also a timestamp and a signature of the password. The servers creates an entry on that client database for that domain/username id and stores the password, timestamp and signature. Then return an OK.

For retrieving passwords, the client sends the Seq, the CPk, and the domain/username (encrypted and hashed) that identifies the entry. The server searches on that client's database and returns the password associated. If it doesn't have that entry, it will return an error. The client decrypts the JSON and gets the password.

For all PUT and GET, the server return an HMAC with the key shared on the INIT Process.

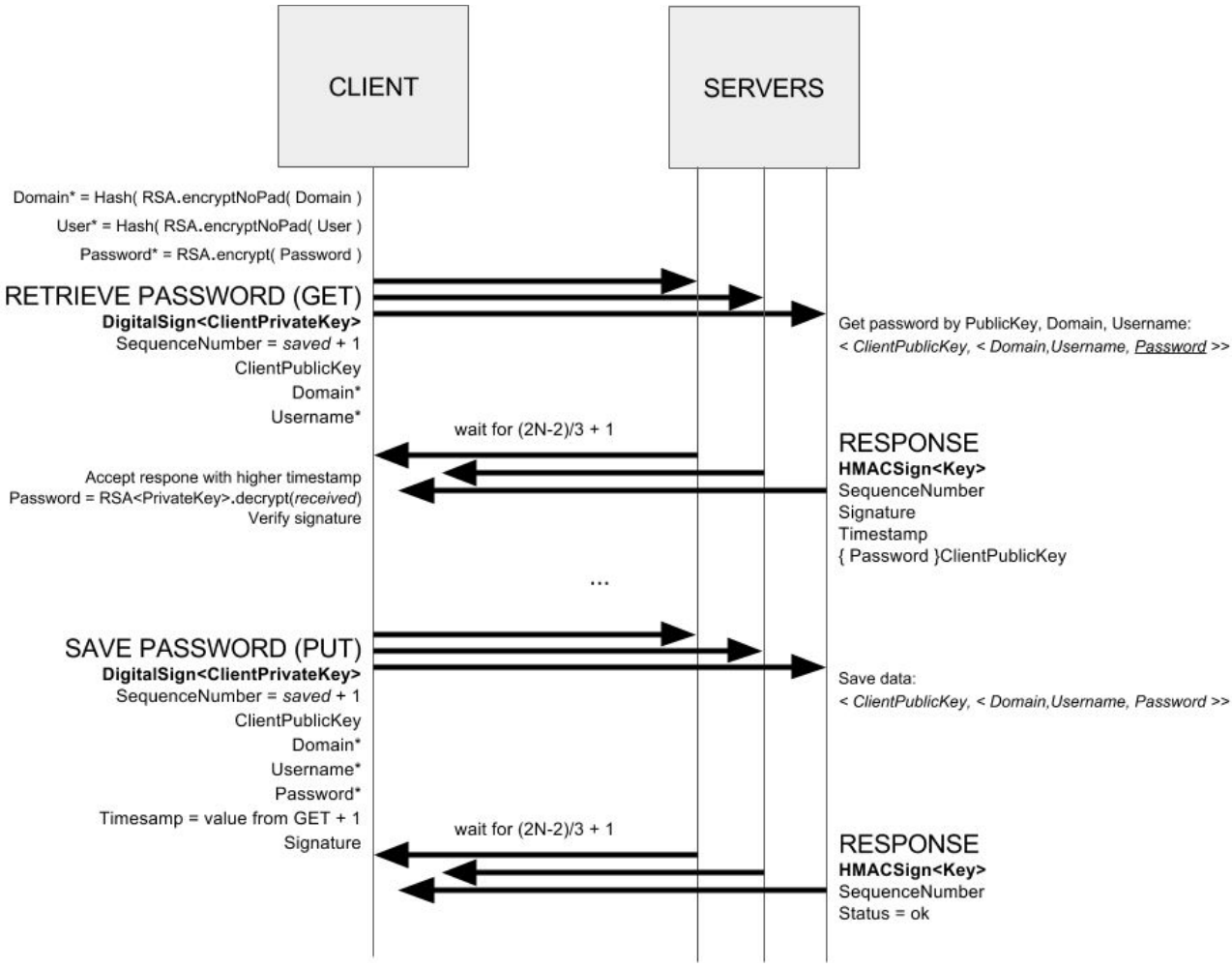

Fig. 4: Save Password and Retrieve Password

If the USER wants to GET a password, the CLIENT APPLICATION make this steps:

- GET requests to all servers by domain and username
- Waits for $\frac{2N-2}{3}+1$ for N Servers
- Selects the password with higher timestamp
- Verifies signature of the password
- Increments the timestamp
- makes PUT requests to all servers with the received password and signature

If the USER wants to PUT a password, the CLIENT APPLICATION make this steps:

- GET request to all servers by domain and username
- Waits for $\frac{2N-2}{3}+1$ for N Servers
- Selects the password with higher timestamp
- Verifies signature of the password
- Increments the timestamp
- Create a signature for the new password
- makes PUT requests to all servers with the new password and signature

If we have a USER on 2 concurrent CLIENT APPLICATIONS:

- The CLIENT APPLICATION will receive an error if it doesn't have the most recent Seq Number
- That error contains the most recent Seq Number so that it can repeat the request again

Security mechanisms against threats

We tried to cover all the possible attacks to our system:

- **Repudiation, Tampering and Masquerading** are not possible due to the Digital Signature, that is an hash of the JSON's body with RSA Private Key Encryption over it. Because it uses the Private Key, Non-Repudiation and Authenticity are secure. With the hash inside, we can check if the packet was modified in flight, so Integrity is also secure. The sensitive responses from the server are signed with HMAC;

- **Eavesdropping** is not possible because all the sensitive data is encrypted with client and server Public Keys and can only be decrypted with client/server Private Keys respectively;

- **Replay-attacks** are not possible as well because all the Save and Retrieve requests and responses are sent with a sequential number to each client, and receiving a JSON with a wrong Seq will discard the request/response.

Bibliography

http://unirest.io
http://sparkjava.com