Highly Dependable Systems Project - 1st 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.

We tried to implement a secure system, so we used Asymmetric Cryptography to encrypt sensitive data exchanged between client and server. We also used some hash methods and digital signature.

- 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;
- For Digital Signature, RSA over SHA-512. Every message exchanged on our system is signed and verified on the other end.

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 on the client and in Digital Signature on both sides.

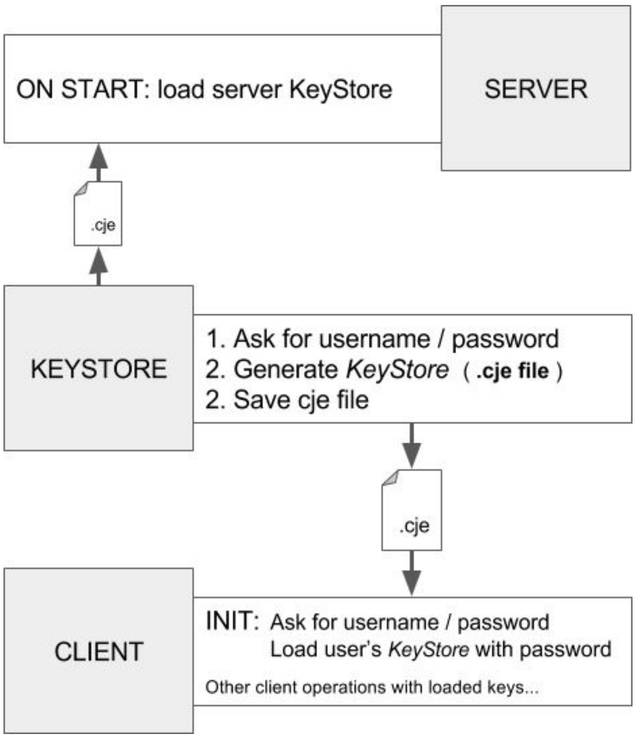
ON START: load server KeyStore
SERVER
.cje
KEYSTORE
1. Ask for username / password
2. Generate *KeyStore* (**.cje file**)
2. Save cje file
.cje
CLIENT
INIT: Ask for username / password
Load user's *KeyStore* with password
Other client operations with loaded keys...

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 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 server, it will verify the signature and check if the user exists by searching for its CPk on the database. Because it's the first time, the server returns an error reporting the client is not registered;
- REGISTER: The client sends now the same JSON to the right endpoint of the server (/register). The server receives it and checks if the user is already registered. Due to first time, it registers the user, sets its SeqNumber to 0 and creates a Digital Signature Manager for that client. The response of the server is a JSON with client Seq and Server Public Key (SPk);

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

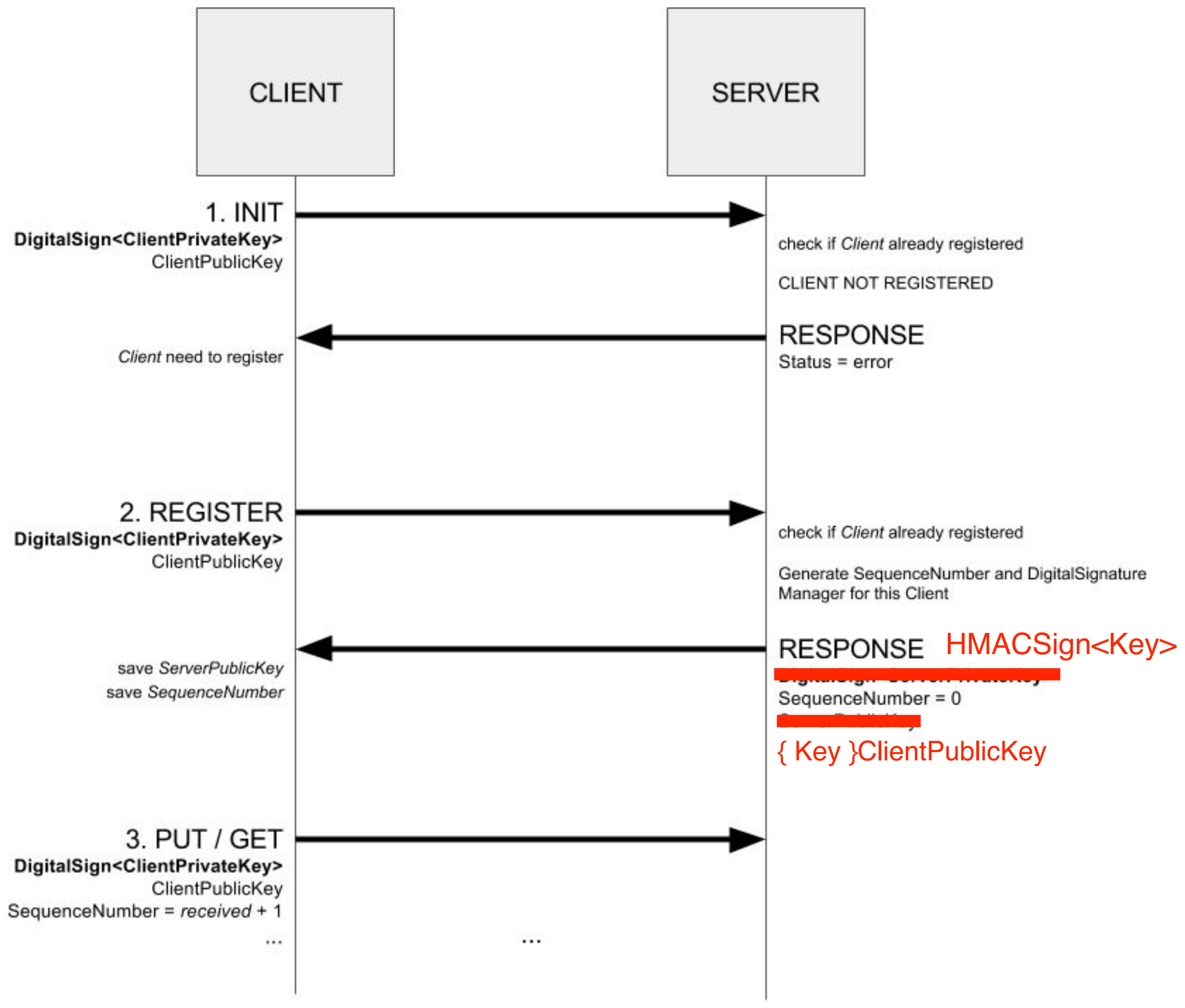

Fig. 2: Cold-start scenario

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

- INIT: The client sends a request with CPk to the server. The client is already registered so the response contains the Seq of the last request of the client (before reboot) and the SPk;

    And now the client can continue to use the app.

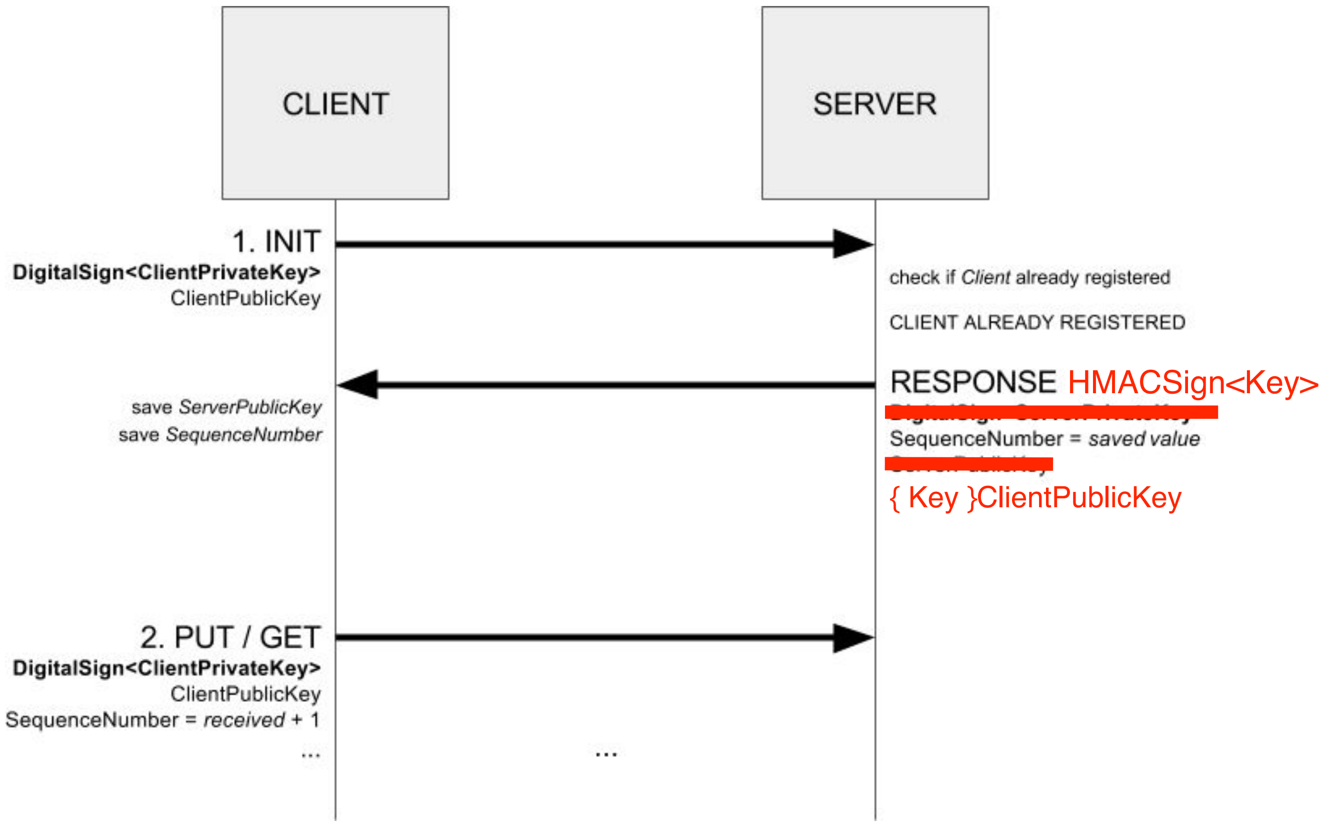
CLIENT
SERVER
1. INIT
DigitalSign<ClientPrivateKey>
ClientPublicKey
check if Client already registered
CLIENT ALREADY REGISTERED
RESPONSE HMACSign<Key>
[redacted]
SequenceNumber = saved value
[redacted]
{ Key }ClientPublicKey
save ServerPublicKey
save SequenceNumber
2. PUT / GET
DigitalSign<ClientPrivateKey>
ClientPublicKey
SequenceNumber = received + 1
...
...

Fig. 3: Reboot scenario

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 server creates an entry on that client database for that domain/username id and stores the password. Then returns 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.

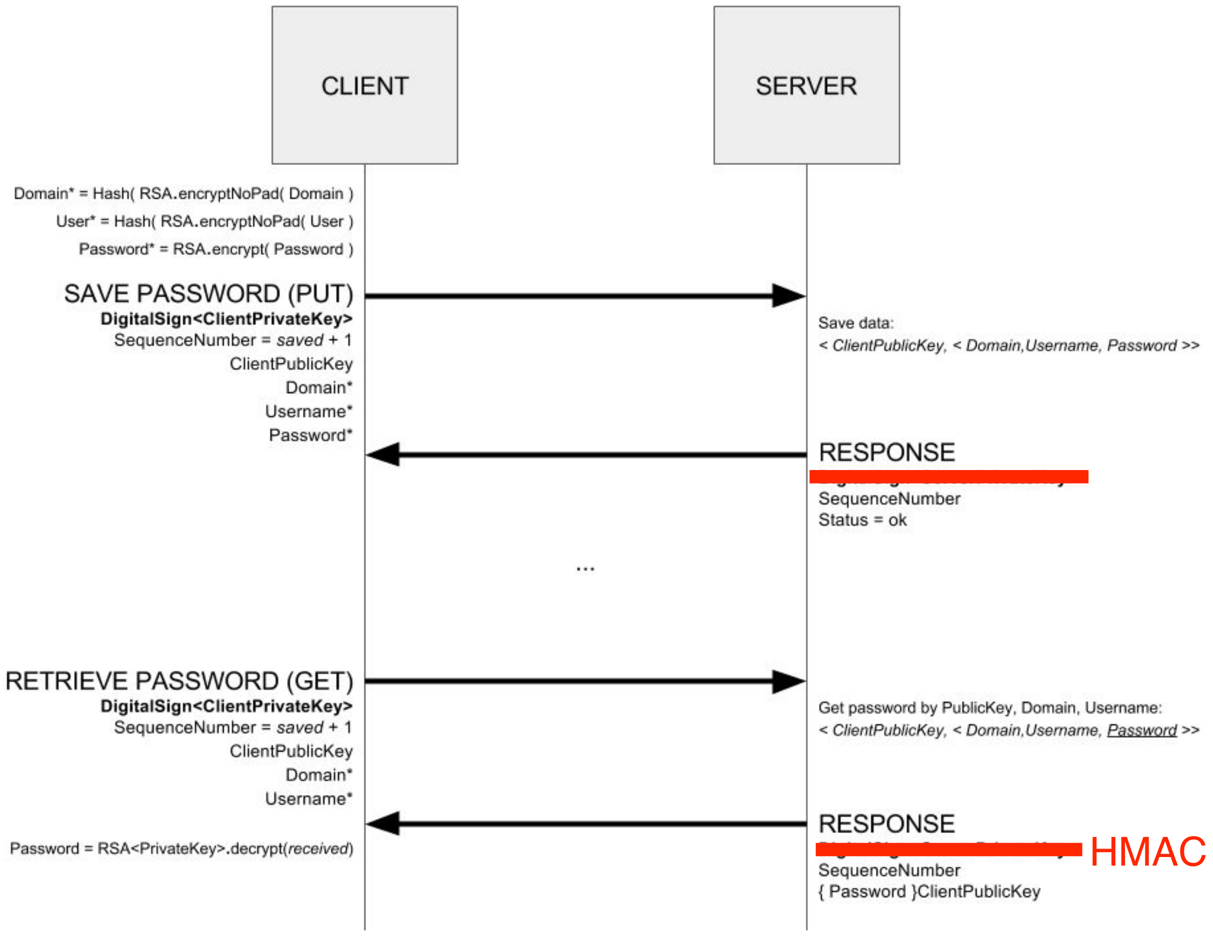
CLIENT
SERVER
Domain* = Hash(RSA.encryptNoPad(Domain)
User* = Hash(RSA.encryptNoPad(User)
Password* = RSA.encrypt(Password)
SAVE PASSWORD (PUT)
DigitalSign<ClientPrivateKey>
SequenceNumber = saved + 1
ClientPublicKey
Domain*
Username*
Password*
Save data:
< ClientPublicKey, < Domain,Username, Password >>
RESPONSE
[redacted]
SequenceNumber
Status = ok
...
RETRIEVE PASSWORD (GET)
DigitalSign<ClientPrivateKey>
SequenceNumber = saved + 1
ClientPublicKey
Domain*
Username*
Get password by PublicKey, Domain, Username:
< ClientPublicKey, < Domain,Username, Password >>
RESPONSE
[redacted]
HMAC
SequenceNumber
{ Password }ClientPublicKey
Password = RSA<PrivateKey>.decrypt(received)

Fig. 4: Save Password and Retrieve Password

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;
- 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