Dependable Public Announcement Server

Stage 1 Report

Highly Dependable Systems

Group 16 - Alameda

Rodrigo Lima 83559

João Martinho 86454

Henrique Sousa 94126

1. Introduction

The goal of this first stage of the project was to create a public announcement system that allows users to post, track and verify public information. The system was implemented using Java 8 and the communication is done using SOAP-based java web services (JAX-WS).

2. Overall System Architecture

The system has a client-server architecture where each client and server has a Public key pair. Those key pairs are RSA keys that were generated using openssl and later converted to PKCS12 format.
At the start of the system, there is an authentication mechanism that allows to ensure that the registering of a publicKey is actually done by the owner of that publicKey. The client must provide its client id and password in order to authenticate. The passwords were previously generated using a secure password generator available online, to ensure some randomness (The client ids and password can be found in the end of the readme file).
The client module holds a digest for the corresponding password and checks whether the given input password, once hashed, matches the one that is hard coded. The message digest algorithm used was SHA-256, due to the fact that it is not yet broken and is one of the most recommended.

3. KeyStore

Due to the need of protecting our key pairs (especially the private keys), we implemented a keystore that holds securely all private keys and associated self-signed public key certificates (from which we can retrieve the public key).
The keystore is a java keystore and holds PKCS12 type files (each file has the private key and a certificate of the corresponding public key). The keystore is protected by a password as well as the files themselves. The password is unique, was generated using an online password generator to ensure randomness and can be found at the end of the readme file.

4. Persistence

In order to prevent the loss of information in case any crash fault happens, we decided to save (using java serialization) the server state in two files, at the end of every register, post and post general operation (the other operations, read and read general, do not change the server state, therefore we don't need to save the state after those operations are performed). The second file exists to serve as a backup if the first one, for some reason, becomes corrupted.

In case of server crash, once that module is executed again, it will ask if we want to recover the past server state. At this step, if the answer is yes, we try to load the state from the primary file. In case this one is corrupted we will receive an IOException, that is treated by trying to load the server state from the backup file.
That solution works because the save operation is sequential, meaning that in case of failure it will only corrupt the primary file.

5. Secure Communication Protocol

In order to protect the system and its users against potential threats, there is a custom secure communication protocol implemented at application level that follows the image below.

5.1. Integrity

To ensure integrity of the messages sent, we hash the message data, as well as extra data (mentioned in 5.3 and 5.4). If the hash received does not match the hash of the message received, we know there was some tampering of the message during transit.
Hashes were built using the java.security.MessageDigest with the SHA-256 algorithm.

5.2. Non-repudiation

To hold the clients accountable for their register (equivalent to the non-repudiation property in security) both post and postGeneral requests have their hash signed for the request they make. The server only accepts a request if the signature properly signs a correct hash of the request with the client's private key (and vice-versa for the responses in the client).
Signatures were built using the java.security.Cipher with RSA-2048 algorithm.

5.3. Replay and drop attack prevention

In the hash of the post and postGeneral requests and responses, we put a synchronized sequence number (SN) that uniquely identifies the request-response exchange. This protects the client and server protocol against replay attacks, as in the server a request with a specific SN will only be executed once.
The way we implement the SN also helps to prevent drops of messages in transit. We describe two undesirable situations: request is dropped (1) and response is dropped (2). The way we increase the SN in server and client prevent both situations from being catastrophic in our system:

- in situation (1), the server doesn't get any request and the client times out, so neither SN is increased, and so the system is safe.
- in situation (2), the server executes as normal and increments the SN but the client times out. The server SN is ahead by 1 now. When a client does a new request, the hash will be wrong (cause SNs mismatch). However, the server tests the hash with SN-1. If the hash is correct, a drop probably happened, and the server notifies the client of it, who then updates its SN.

5.4. Surreptitious forwarding and message stealing prevention

In the hash of the register, post and postGeneral requests and responses, we put the client and servers ids. Since the requests destination is always the server and the source is the client whose public key goes in the request (and vice versa for the responses) we do not need to send them in clear. By being in the hash, they prevent surreptitious forwarding and message stealing, since a man-in-the-middle (MITM) is unable to change the contents without knowing the client/server private keys.

Final Note: Our server module holds a test folder (announcementServer-ws/src/test) where we have some tests to prove that security requirements are working. We suggest the reading of those.

References

1. Java WebServices (JAX-WS)
2. Java Development Kit 8
3. Java Crypto API
4. Apache-Maven 3.6.3
5. JUnit 4