**Integrated MSc Course on Informatics Engineering, DI/FCT/UNL**

**Computer Networks and Systems Security / Semester 1, 2019-2020**

**WORK-ASSIGNMENT #2 REPORT for Evaluation**

Messaging System

REPORT

**Authors:**

**Tomás Pessanha (**[**t.pessanha@campus.fct.unl.pt**](mailto:t.pessanha@campus.fct.unl.pt)**)**

***Summary***

The project is a server/client secure Messaging System a lite version of Gmail where each user has an inbox with messages for him, the goal is to protect the messages from an attacker both in transport and in the way the message is saved in the server.

To secure the data TLS and several cryptographic constructions were used, TLS for the transport and other constructions like signatures and encryptions.

It also allows the receiver to acknowledge that he received the message.

**1. Introduction**

In any message system especially in a message system with a server involved users expect confidentiality and the assurance the system is working correctly like for instance when u send the message ‘hello’ you expect the other user to receive ‘hello’ and not anything else, this project.

The objective is to support the following base security features

* Message confidentiality, integrity and authentication following X.800 typology.
* Secure message delivery confirmation

The project is divided into 2 (two) main parts client and server.

The server is a REST (<https://en.wikipedia.org/wiki/Representational_state_transfer>) based server and Spring [0] was used as the chosen framework for the implementation.

The client is a simple java terminal application

During the development many features of a real-world message application were cut in order to prioritize the actual security aspect of the application since that is the main goal here more about this will be discussed later in the report.

**2. System model and architecture**

**2.1 System model**

**A screenshot of a cell phone

Description automatically generated**

(Solid lines mean real connections, dotted lines mean not actually implemented connections)

All clients communicate only with the server (indicated by the solid lines) every request is made to the server and optionally a response is returned to the client no client to client interaction occurs, example if a user wants to send a message to another user it has to send the message to the server and then the other user will check its messages.

Both the server and the clients are need the “Root CA” to validate its own authenticity, this is an imaginary pseudo Certification Authority, in a real world application it would be its own server, in this project however we just pretend that the CA exists and create all the needed certificates ourselves (All CA related files are in the folder “RootCA”).

This CA belong to our trust base, anything signed by this CA we can trust.

**2.2 Architecture**

## Server

The Spring [0] server provides an API that clients can use to use the system a full list of the provided functionality will be provided has an attachment.

At the moment most requests use ***http post***, but it would make sense to use ***get/put*** on some requests, this allows for a more ***restful* *API***.

Using Spring [0] as the framework for the server makes this process relatively painless and Spring is a very well documented and tested framework with support for protocols like TLS so it was a good choice for the project at hand.

## Client

For the client I was not aware of any major framework to ease the work, so the client was made using normal means of a terminal application with normal java.

**2.3 Threat model**

The implementation is supposed to resist all attacks in the **X.800** threats typology, this means assurances of confidentiality, integrity and authentication.

We are also in the presence of a Honest But Curious (HBC) system administrator in the server side, so we need to be able to hide the private data.

Presented next is a table mapping the security mechanism used to protect against each attack considered.

|  |  |  |  |
| --- | --- | --- | --- |
| **S. Mechanisms / Attacks** | **Unwanted release of message contents** | **Modification of messages** | **Masquerade** |
| **Encipherment** | x |  |  |
| **Data Integrity** |  | x |  |
| **Digital signature** |  |  | x |

**3 Implementation details**

## TLS

The use of TLS is fully supported in the client and server, and fully configurable, since the server is using Spring [0] there was no need to reinvent the wheel and the TLS configurations can be made using the file “application.yml” with Spring [0] configurations can be seen here ([https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#server-properties](https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html%23server-properties)).

The TLS in the client side can be configured much the same using the file “application.yml”, this however is a custom application and no standard can be applied the will be comments on the file showing what are valid inputs on non-obvious configurations.

Both the client and the server have a trust store and a key store file, the trust store contains the RootCA certificate, and the key store contains the server/client certificate signed by the RootCA this is what identifies the user (Note: all passwords are set to “password” for ease of testing).

Finally, the server decides witch ciphersuites to use among the enabled ciphersuites on both sides, this was made this way because changing the server is easier than all the clients.

## HBC and unwanted use protection

On user register the user sends to the server its public key, and private/public Diffie hellman (DH) params (the private param is encrypted with a Password)

The users uuid is an hash of its public key.

The server provides API to retrieve any registered users keys or DH params.

All messages send are encrypted with an AES key, this key is obtained with a DH key exchange, between the sender and the receiver (Note: this feature is disabled and the DH secret is constant in the current implementation because of an unfixed but however all code is provided for this feature to work).

**4. Work Evaluation and Validation**

The server was evaluated with the testssl tool given in the labs, TLS was tested with wireshark and several manual tests were done to ensure the correct functionality of the project.

**5. Conclusion**

The initial expectations for this work were not meet and many of the requirements could not be completed but still much was learned from this work, if I could suggest that the teacher adds more examples of how to turn byte arrays into keys in the labs your next semesters of SRSC, because I was stuck many times trying to get a key for something,

**References**

Put your cited references here, ex:

[0] Spring Framework, <https://spring.io> (available and retrieved in November/2019).

[1] Paul Sklenar, Securing REST APIs With Client Certificates, DZone Tutorial and implementation, <https://dzone.com/articles/securing-rest-apis-with-client-certificates> (available and retrieved in October/2019).

[2] [baeldung](file:///C:\Users\Tomas-PC\Downloads\baeldung), Jackson JSON Tutorial, <https://www.baeldung.com/jackson> (Last modified: December 7, 2019)

[3] Open source, https://testssl.sh (Last modified: December 7, 2019)

# Attachment 1

List of endpoints the API supports:

* /create – creates a user message box
* /list – lists users registered in the system
* /new – shows new messages
* /all – shows all messages
* /send – sends a message to a user
* /recv – receive a message
* /receipt – create a receipt for the message
* /status – show the receipt status of a message
* /pubKey/{id} – returns the public key of an user
* /pubDH/{id} – returns the public DH param of a user
* /prvDH/{id} – returns the private DH param of a user (encrypted by a password)