Security Report

Security

Luís Silva, Pedro Santos

universidade de aveiro

Security Report

DETI

Security

Luís Silva, Pedro Santos
lfssilva@ua.pt,pedroamaralsantos@ua.pt

November 10, 2017

Contents

Chapter 1

Overview

This report serves the purpose of giving an overview of our project implementation. The project consists of a system that let his users exchange messages asynchronously.

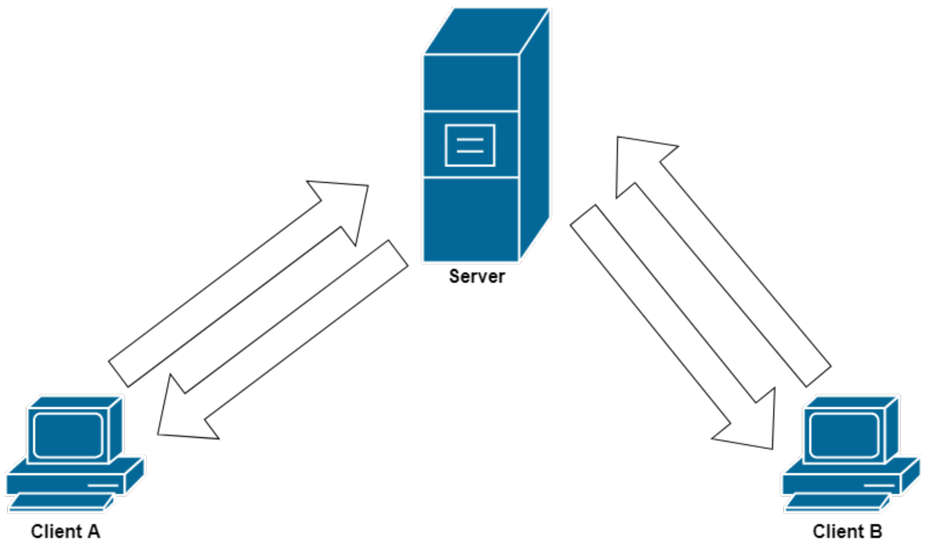
Chapter 2

Architecture

The system will have mainly two components that are the clients and the server. The clients of the system will not interact directly with each other, therefore the server will work like a rendezvous point where all the clients are connect. The server will be responsible for saving a list of message boxes, and the respective cryptographic material to interact with their owners.

2.1 Repository Server

The server to exchange messages with the clients will use TCP/IP. This protocol uses the client/server model of communication, accordingly clients never interact directly, they only interact with the server.



2.1.1 Clients list

All the users of the system will be at a list in the server that will contain the following information of each.

Information	**Description**
id	Short reference identifier that will be defined by the server. It is a integer value
uuid	Long term identifier unique for each user. The digest of the public key of the user will be used to compute the value.
mbox	Path to user's message box.
rbox	Path to a user's receipt box.
secdata	Path to the client's certificate.

Table 2.1: Table with the infomation that the list contains about each user

2.1.2 Message Box and Receipt Box

The server will have a directories that will be used to message box and a receipt box for each user. As mentioned before the path to each of this boxs will be at the list that the server has with all of the information that needs per client. The message box will have a different file per message and will be never deleted. The receipt box will have a different file per sent message and 0 or more read receipts. A receipt is a confirmation that a message was received and properly decoded by a receiver. Messages in the receipt box are encrypted in a way that only the sender could see it, thus they are not exact copies of the messages delivered to users. This allows a message sender to get access to the messages it has send and to check which messages were received and acknowledged by receivers. These files are never deleted. Both directories names of receipt box and message box will derived from user id and located below directories mboxes and receipts.

2.1.3 Messages

Messages inside message box are identified by a name formed by a sequence of number **U_S** where **U** is the identifier of the sender and **S** is a sequence number for that sender, starting at 1. Messages in messages boxes already read have a name prefixed with _.

2.1.4 Receipts

Receipt boxes are also identified by a name formed by a sequence of numbers **U_S**, however **U** is the identifier of the receiver and **S** is the sequence number above referred. Receipts have an extra extension with the date when the receipt was received by the server.

2.2 Processes

The server will have processes so users can interact with it and receive a response. The processes are run according to the message that the client sent.

Message	**Process Description**	**Server Response**
CREATE	Creates a user message box.	The internal identifier given to the user.
LIST	Lists users' messages boxes	A message of same type containing a list of users with a message box
NEW	Lists new messages received by a user.	A list of messages not yet read by the user.
ALL	List all messages received by a user.	A list of all messages received by the user.
SEND	When client wants to send a message to a user.	The identifiers of both the message sent and the receipt.
RECV	When Client wants to receive a message from a user message box.	The identifier of the message sender and the message contents
RECEIPT	When Client wants to send a receipt to a user, in order to acknowledge a received message.	
STATUS	When Client wants to check the status of a sent message.	An object containing the sent message and a vector of receipt objects, each containing the receipt data (when it was received by the server), the identification of the receipt sender and the receipt itself.

Table 2.2: Table with the description of the server's processes.

2.3 Client/Server Messages

The format of the exchanged messages between the server and clients will be Json. The binary content will be converted to Base-64.

2.3.1 Server response messages

The server will respond with

```
{
<command-specific attributes>
}
```

upon a successful. Otherwise, it will respond with

```
{
"error":"error message"
}
```

2.3.2 CREATE

Message type used to create a message box for the user in the server:

```
{
"type": "create",
"uuid": <user uuid>,
"certificate": <certificate>,
"pubkeygenerated":<public key>,
}
```

The **type** field is always create.

The **uuid** field contain a unique (and not yet known to the server) user unique identifier. It is used the digest of a user's public key certificate, extracted from his/her Citizen Card.

The **certificate** field contains Citizen card certificate.

The **pubkeygenerated** field which contains the public key generated by the client.

The certificate and the public key are sign by the client with his Citizen Card.

The server will respond with the id (an integer) given to the user:

```
{
"result": <user id>
}
```

2.3.3 LIST

Sent by a client in order to list users with a message box in the server:

```
{
"type": "list",
"id": <optional user id>
}
```

The **type** field is a constant with value list.
The **id** field is an optional field with an integer identifying a user to be listed.
The server reply is a message containing in a result field all information regarding a single user or all users. This information corresponds to the creation message, excluding the type field:

```
{
"result": [{user-data}, ...]
}
```

2.3.4 NEW

Sent by a client in order to list all new messages in a user's message box:

```
{
"type": "new",
"id": <user id>
}
```

The **type** field is a constant with value new.

The **id** field contains an integer identifying the user owning the target message box.

The server reply is a message containing in a result field an array of message identifiers (strings). These should be used as given to have access to messages' contents.

```
{
"result": [<message identifiers>]
}
```

2.3.5 ALL

Sent by a client in order to list all messages in a user's message box:

```
{
"type": "all",
"id": <user id>
}
```

The **type** field is a constant with value all.

The **id** field contains an integer identifying the user owning the target message box

The server reply is a message containing in a result an array containing two arrays: one with the identifiers of the received messages, and another with the identifiers of the sent messages.

```
{
"result": [[<received messages' identifiers>][sent messages' identifiers]]
}
```

2.3.6 SEND

Sent by a client in order to send a message to a user's message box:

```
{
"type": "send",
"src": <source id>,
"dst": <destination id>,
"msg": <JSON or base64 encoded>,
"copy": <JSON or base64 encoded>
}
```

The **type** field is a constant with value send.

The **src** and **dst** field contain the identifiers of the sender and receiver identifiers, respectively.

The **msg** field contains the encrypted and signed message to be delivered to the target message box; the server will not validate the message. The **copy** field contains a replica of the message to be stored in the receipt box of the sender. It should be encrypted in a way that only the sender can access its contents, which will be crucial to validate receipts.

The server reply is a message containing the identifiers of both the message sent and the receipt, stored in a vector of strings:

```
{
"result": [<message identifier>,<receipt identifier>]
}
```

2.3.7 RECV

Sent by a client in order to receive a message from a user's message box. "type": "recv", "id": <user id>, "msg": <message id>

The **type** field is a constant with value recv.

The **id** contains the identifier of the message box.

The **msg** contains the identifier of the message to fetch.

The server will reply with the identifier of the message sender and the message contents:

```
{
"result": [<source id,<base64 encoded message>]
}
```

2.3.8 RECEIPT

Receipt message sent by a client after receiving and validating a message from a message box:

```
{
"type": "receipt",
"id": <user id of the message box>,
"msg": <message id>,
"receipt": <signature over cleartext message>
}
```

The **type** field is a constant with value receipt.

The **id** contains the identifier of the message box of the receipt sender.

The **msg** contains the identifier of message for which a receipt is being sent.

The **receipt** field contains a signature over the plaintext message received, calculated with the same credentials that the user uses to authenticate messages to other users. Its contents will be stored next to the copy of the messages sent by a user, with a extension indicating the receipt reception date.

The server will not reply to this message, nor will it validate its correctness.

2.3.9 STATUS

Sent by a client for checking the reception status of a sent message (i.e. if it has or not a receipt and if it is valid):

```
{
"type": "status",
"id": <user id of the receipt box>
"msg": <sent message id>
}
```

The **type** field is a constant with value status.

The **id** contains the identifier of the receipt box.

The **msg** contains the identifier of sent message for which receipts are going to be checked.

The server will reply with an object containing the sent message and a vector of receipt objects, each containing the receipt data (when it was received by the server), the identification of the receipt sender and the receipt itself:

```
{
"result":{
"msg": <base64 encoded sent message>,
"receipts": [
{
"data":<date>,
"id":<id of the receipt sender>,
"receipt":<base64 encoded receipt>
},
...]
}
}
```

Chapter 3

Ciphers and Mechanisms

3.1 Hash Function

The system will use a Cryptographic hash function and it will be very important to ensure security in our system. The most important properties a hash function has to have are:

- **Pre-image resistance**
- **2nd pre-image resistance**
- **Collision Resistance**
- **Random oracle property**

In our system the most critical property will be collision resistance.

3.1.1 SHA256

SHA256 will be the algorithm used to calculate hash values due to the compromise between the size of the output and the likelihood of collisions.

3.2 Key Agreement and Key Distribution

The client needs to agree with a key with the server to exchange messages in a efficient way by using symmetric cipher algorithms. This key is named session key in our system.

3.2.1 Diffie–Hellman

Diffie-Helmman is a well known algorithm to key agreement. Using it separately is necessary to take into account the Man-In-Middle attack.

3.3 Symmetric Cipher

The system will make use of a symmetric block cipher and this cipher will be very important to ensure security in it.

3.3.1 AES

AES is a block cipher algorithm and it provides the security that our system needs. Until today there aren't any relevant reports about successful attacks to it and it is a algorithm widespread used for example in TLS and IPsec. Symmetric cipher algorithms are fast, therefore it will be used a **256 bit key** because it will be only to cipher plain text and it will provide extra security in detriment of using a smaller key. To ensure confidentiality the mode of operation will be **CBC**. This mode of operation is probably the most widely used and apart from the key, it is also necessary an Initialization Vector (IV). This IV will be random and independent for each message. The same key/IV pair will not be used to process more than one message. Using this mode we will have to do padding when needed. Integrity is also desired, so this mode will be used together with a secure MAC, that will be discuss in the next section.

3.4 Message Authentication Codes

The main goal of use MACs is to provide integrity protection. To calculate the MAC it is necessary to give a key, as well as the message that we want to guarantee the integrity and authenticity of the same. The used key it will be the session key.

3.4.1 HMACSHA-256

The HMACSHA-256 will be the choice because SHA-1 collision resistance is already broken and in this way it is ensure a good level of security in the system. In terms of performance it is not a critical situation because it is not a real time system and it is a system where messages are sent asynchronously and it is only to be applied in plain text.

3.5 Key Encapsulation Mechanisms

Symmetric keys are fast, but they alone don't provide the needed security. On the other hand, public cryptography is very secure, but they are not efficient. So, the system will use both in the best convenient way. This approach is called hybrid Encryption.

3.5.1 RSA

The algorithm will use a section N >= 2432 and a public exponent of e > 65536. This values can suffer changes if they are compromising system performance.

3.6 Signatures

To the system be able to ensure the authentication of some messages will be necessary create digital signatures.

3.6.1 RSA PKCS#1

There are not a lot of options. The used algorithm will be RSA PKCS#1.

Chapter 4

Implementation

4.1 Setup of a session key between a client and the server prior to exchange any command/response

We will use the authenticated variant of Diffie-Hellman so we establish a session key.
The packets exchanged are sign. The client signs with his citizen card and the server with his private key.

4.2 Authentication and integrity control of all messages exchanged between client and server

The messages exchanged will have a MAC calculated with the session key.

4.3 Add to each server reply a genuineness warrant

The client challenges the server with a random value that needs to be present in the response

4.4 Register relevant security-related data in a user creation process

When issuing a new user create we need to provide the server with all the cryptographic elements that will be needed in posterior communications. For instances, send the CC certificate and the public key (of the generated pair).

4.5 Involve the Citizen Card in the user creation process

Ensure that a new client can only be created with a valid CC. In the process of creation of a new client we need to sign the message with the CC private key and provide a valid certificate.

4.6 Encrypt messages delivered to other users

The messages delivered to other users will be encrypt through an hybrid method, will be generated a random key that will be the symmetric key to encrypt the message to the the user, then will encrypt this key with the public key (generated) of the receiver.

4.7 Signature of the messages delivered to other users and validation of those signatures

All the messages delivered to other users will be signed with the CC and the server will be responsible for validating those signatures before storing the messages.

4.8 Encrypt messages saved in the receipt box

The copy element of the send message will be a copy of the plain text sent encrypted with the Pub key of the sender. So only him can access its content.

4.9 Send a secure receipt after reading a message

Send a receipt that will consist of the encryption of the hash of the received plain text with the public key of the message original sender.

4.10 Check receipts of sent messages

Check if the message have a receipt and if it is a valid one, one that actually means the user read the message. The decryption of the receipt must reveal the hash of the original plain text .

4.11 Proper checking of public key certificates from Citizen Cards

Server validates the entire certification chain.

4.12 Prevent a user from reading messages from other than their own message box

Already ensured by the encryption of the sent text messages.

4.13 Prevent a user from sending a receipt for a message that they had not read

When the user is read the server adds a random element that must be present when the user claims to have read the message and wants to send the receipt.