Security Report

Security

Luís Silva, Pedro Santos

universidade
de aveiro

Security Report

DETI

Security

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

December 31, 2017

Contents

Chapter 1

Introduction

Nowadays computers are everywhere, and the subject of security is more important that ever. People are more concerned about their private data and how this data is shared, many times without them knowing it and are demanding more safe applications so they can enjoy all the benefits of technology without compromising their data.

For the final project of the Security class the challenge was to implement a secure application for exchanging asynchronous messages. This project enabled the exploration of different methodologies, and security mechanisms that would make our application to be more secure.

Within this report we will first begin to address the overall architecture of the application and also how the communication is done; next we will explain the details of the implementation for all the functionalities implemented; Finally but not least we will expose the algorithms and other tools that have been chosen to the implementation and why; After all this we have a small Conclusion with some final thoughts about the project and the subject of Security.

Chapter 2

Architecture

Our application consists of a messaging system, an asynchronous one. The system will have mainly two components. The server that provides the service and the clients that interact with 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 connected. The server cannot be trusted. The server will be responsible for saving a list of message boxes, and the respective cryptographic material to interact with their owners.

In a abstract way we can think of 2 security layers that exist in our project, the first is a layer of security between the communications of two user. In this layer, we have security that keeps a message sent, and stored in the server secure only to the supposed receiver. No other person can either see or tamper with the messages, receipts and user descriptions that are stored in the server. This is done through cyphering of the contents that are stored in the server. Another layer of security that we added was the communication between the clients and the server. As we will see the client and the server exchange pre-formated messages that enable the user to do some things, this messages are exchange securely through an encrypted channel. We will discuss this in more detail in the next sections.

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. As previous said the server cannot be trusted, so, information posted and retrieved from there need to be protected from tampering.

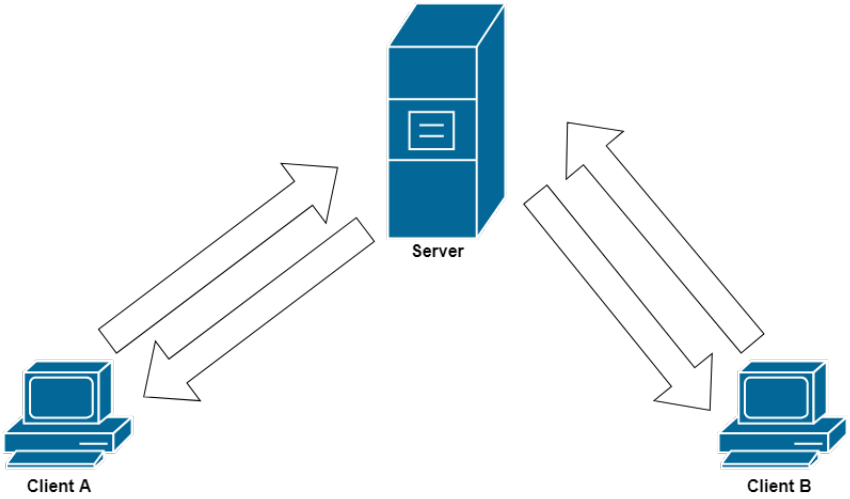

Figure 2.1: Client Server Diagram

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 boxes 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 Client

The client application will interact with the repository server exchanging JSON packets with it. The application when started will verify the presence of a Portuguese Citizen Card and if not detected will not continue. If a Citizen Card is present then, the user will be warned that a secure connection is going to be established with the server. After this connection is established, the user can create a new message box if he hasn't one or we can continue as an existing user. Then the user will be prompted to load his RSA keys to the system, and have the option to create new ones. Finally if all went well the user can now exchange securely messages with the server, having the opportunity to send and receive messages from and to other users

2.3 Client/Server Messages

As previous explained, all the messages between the server and the client will be exchange through a secure channel, this means, that before exchanging any message, the server and user establish a session. After that every packet exchange will be encrypted with the session credentials.

2.3.1 SESSION

This Messages are the first one to be exchanged and have the purpose of establishing a session key between the client and the server.

```
{
"type": "session",
"pubk": <pubk>,
"cert": <certificate>,
"signature":<signature>,
}
```

The **type** field is always a string "session" indicating this a session establishment packet.

The **pubk** field contain the public values for Diffie Hellman session. The **certificate** field contains a signed certificate.

The **signature** field which contains the signature of the packet.

2.3.2 PAYLOAD

After the session is established all the packets exchanged will be a payload packet that will contain the instructions encrypted in his payload field.

```
{
"type": "payload",
"payload" : <payload>,
"iv": <IV>,
"nonce": <nonce>,
"mac": <HMAC>
}
```

The **type** field is always a string "payload" indicating this is an instruction encrypted. The **payload** field is the the instruction to the server encrypted with half the key generated in the process before The **iv** field refers to the IV necessary to decrypt the message The **nonce** field is a random number with 64 bits to keep the messages in the correct order Finally but not least the **mac** field contais the HMAC generated with the other half of the session key.

2.4 Client/Server Actions

This are the messages that go encrypted to the server

2.4.1 Server response messages

The server will respond with

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

upon a successful. Otherwise, it will respond with

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

2.4.2 CREATE

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

```
{
"type": "create",
"uuid": <user uuid>,
"pubk":<pubk>,
"cert": <certificate>,
"signature":<sign>,
}
```

The **type** field is always create.

The **uuid** field contain a unique (and not yet known to the server) 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 **pubk** field which contains the RSA public key generated by the client.

The **signature** field contain the signature over the certificate, pubk and uuid signed by the client with his Citizen Card.

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

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

2.4.3 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}, ...]
}
```

2.4.4 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>]
}
```

2.4.5 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]]
}
```

2.4.6 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": <base64 encoded>,
"copy": <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 message 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 is encrypted in a way that only the sender can access its contents, which will be crucial to validate receipts. This process will be better explained later in the report.

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>]
}
```

2.4.7 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>]
}
```

2.4.8 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>,
"nonce": <nonce>

}
```

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 **nonce** contains 128 bits randomly generated by the server when client read the message. The server concatenate the nonce in the message and then sends the message to the user. The user when sends the receipt will send the read nonce to prove that he received the message and the server will compare the nonces.

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

2.4.9 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 uses a Cryptographic hash function and it is very important to ensure security in the 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 this system the most critical property will be collision resistance.

3.1.1 SHA256

SHA256 is 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. We'll use the authenticated version.

3.3 Symmetric Cipher

The system uses 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 is used a **256 bit key** because it is only used to cipher plain text and it provides extra security in detriment of using a smaller key. To ensure confidentiality the mode of operation that it is used is **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 it is necessary to do padding when needed. Integrity is also desired, so this mode is 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.

3.4.1 HMACSHA-256

The HMACSHA-256 is 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 uses 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 OEAEP

There are not a lot of options. The used algorithm will be RSA OAEP.

Chapter 4

Implementation

In this chapter will be presented the approach used to implement and develop the application with all the security features. First we begin to implement a non-secure application were we could send all the different Json packets and ensure they were all working properly, only then we star to discuss the security functionalities and finally implement them.

4.1 Establish Session

In the application all the messages between client and the server are encrypted using **AES** cipher algorithm and also to provide integrity protection all the messages have a calculated mac.

To achieve this the client and the server have to agree in one key that will be divided by two. We used a variant of Diffie Hellman key exchange protocol, in our implementation besides the exchanged key, signed certificates are also exchanged and verified, this is done to prevent Man In The Middle Attacks. For this we had to create our own CA that signs the Server certificate. And we assumed that our client trusted the CA. The process begins by the client issuing a session request to the server exchanging his Public Diffie Hellman values and his Citizen Card Certificate, then, after verifying the certificate the server exchange his certificate signed by the trusted CA and his Public Diffie Hellman Value with the client exchange his Public Diffie Hellman values. These certificates are verified and if all is well the Diffie Hellman process concludes and a Key is generated for that session.

The key generated is a large one so the we can divide it and use it both for encrypting messages and generating the HMAC.

4.1.1 Sending and Processing Payloads

As been established that the communication between the server and the client is encrypted and so for that to happen the messages that the client, or the server, wants to send must be encrypted we call this the processing payload to send, for processing the payload to send first thing is done is encrypt the payload with a random IV vector for each message and using the key that was agreed before. Then a Mac is calculated using the encrypted message. This is done in order not to spend unnecessarily resources or time, the message is first encrypted and only then is the Mac calculated thus the receiver checks the MAC first, only if it is valid, it decrypts the message.

On the other end when a payload is received we then talk about process a received payload, and for that since the session is established the key that was agreed is used the one used to encrypted all the messages and for each it is calculated the mac. So all that is done is reverse the previous process if the Mac checks out, then we use the agreed key and the IV that arrived to decrypt the payload.

4.2 Login/Registration in the application using Citizen Card

After the user starts the client application and after having established the session a uuid is calculated by hashing the authentication certificate of his citizen card. For most of the cryptographic operations present in this application the client besides his CC certificate also needs to have a RSA Pair of keys, because we can utilize the CC to cipher information.

So, the next step is to generate or read from a file those RSA keys. This keys are used in the process of ciphering and deciphering messages.

After the keys are established two things can happen the user can say he already has an "account" in the server, and if so, our application lists (LIST) the users of the server and continues if in fact already is present a user with that uuid. If the user claims that he is new and wants to create an "account", then a CREATE is issued with all the information required. For a new user is also sent a LIST, this list, besides the objective described above to an existing user, this LIST will also create a object UserDescription named currUser in the client build form the information gathered so far about the curr User, that is also the information that the server has about the user with exception of the RSA private key. After this steps, the client did everything need to login/register in the application.

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.

When the clients exchange a message to the server he generates a 64 bit nonce that he sends with the payload, and from that moment he is expecting a response that also has that 64 bit nonce in it. The server when receives a message from a client copies the nonce to the answer and sends it to the client. So with this we can always know that the answer we are receiving is for the packet we sent.

4.4 Encrypt messages delivered to other users

The messages delivered to other users will are encrypted 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. To send encrypted messages to the users, because of the non-trustworthy server, we need to encrypt the text sent to them.

For this an hybrid method was used, this method consists of generating a symmetric key that is used to encrypt the text, and the encrypt this symmetric key with the Private Key of the user. With this method we do not compromise the speed of the system. Private key encryption of a large text could take a long time, so by encrypting only a small symmetric key we ensure the overall speed of the process.

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

Encryption is not the only security feature protecting the messages sent to other users. Besides this, the messages that are stored in the server for the other users are also signed. And are verified upon reading.

All the messages delivered to other users will be signed with the CC Authentication certificate. This signatures are validated before a client reads a message he has received.

4.6 Proper checking of public key certificates from Citizen Cards

The clients when receive information that needs to be validated they start by retrieving from the server the CC certificate from the server issuing a LIST, after that, in java with the aid of the classes a *java.security.cert.PKIXBuilder*, *java.security.cert.CertPathBuilder* and *java.security.cert.CertPathValidator* we can build a certification path and then validate it with the parameters we want, in our case besides the Date Validation of the certificate an OCSP query is made to ensure each certificate in the path as not been revoked

4.7 Encrypt messages saved in the receipt box

Like the messages that are sent, to be stored in the receiver user message box, the message that is stored in the sender receipt box is also encrypted but this ones are encrypted with the sender credentials, meaning that only him can access it.

4.8 Send a secure receipt after reading a message

The user when sending the receipt will send the corresponding signature of the message made with his CC and the respective Nonce. When the server receivs the receipt will check if it was received a recv for that message by checking the name of the file of the message (the name of read messages starts by _), but to ensure that the user receive the message the server also will check if the received nonce it's the same as in the message. If these conditions are verified the receipt is saved.

4.9 Check receipts of sent messages

Previously when a user sent a message he also stored the message encrypted with his credentials. So Checking the receipts consist in retrieving this copy and verify if the receipt corresponds of the signature of that clear text.

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

When a session is created, the server also knows who is talking too in that session because he receives the client uuid. When the server receives an order to retrieve a message from a user message box he verifies if the uuid of the current User corresponds to the message box the client is trying to access, if so all goes well.

4.11 Prevent a user from sending a receipt for a message that they had not read/received

To send a receipt the user must have received the message. To ensure that this happens when the user sends a recv to receive the message the server creates a nonce and appends to the message. When the user receives the message in addition to doing what is necessary to decipher the message will save that nonce in a data structure with the corresponding message id. In addition, in order to ensure that the user have read the message, a signature of the plain text of the message made with the user's cc is also stored in a data structure. Whenever the user reads the message a new nonce is created.

Chapter 5

Conclusion

This project has shown us the difficulties that one encounters while developing a secure application. Today, security is a theme more important than ever and developers need to be aware of the dangers so they can implement solutions to solve them.

Other important aspect to be in consideration is how contra-productive security measures can be, it is most likely that a standard user would not understand why the application is constantly asking for a password, or the necessity of RSA keys, but is something that needs to be instilled in the users mentality so we can have a more secure experience in informatics. In the end, a good application was developed. Security issues were a concerned and both passive and active attackers were taken in consideration when thinking of solutions.