47232 - Security

Secure Instant Messaging System M2

Diogo Cunha - 67408
diogocunha294@ua.pt

Guilherme Cardoso - 45726
gjc@ua.pt

18th December 2016

Abstract

This report describes the changes made in order to implement smart card authentication support (using the Portuguese Citizen Card) in the chat architecture, in which is included the server, clients and communication support 1:N clients.

Contents

1 Introduction

In this report we describe how the structure of a message relay server that registers and relays user messages was implemented, including the security layer and the whole interface for clients.

The goal is to have secure 1:N end-to-end communication between clients, where the server acts as relay of the messages, provides client listing and intermediates client connection.

All communications between peers are ciphered using symmetric or asymmetric keys (between clients and server and clients to clients) and their implementation is described later in this report.

In addition to M1, M2 will feature an authentication verification system through the Portuguese Citizen Card. All messages, either from client to server or from client to client will be signed before being sent and verified when they reach the destination.

2 Architectural changes

To accommodate the changes needed in order to implement citizen card support and client authentication, we had to add one new module and two new dependencies. We took this opportunity to also improve overall project structure, enabling inheritance in order to reduce redundant code. All changes are described in sections bellow.

There are now new required libraries, that are:

Library	Minimal version	Description	pip
cryptography	1.7.1	Interfaces for cryptography operations	cryptography
OpenSSL	16.2.0	x509 chain validation and revoke testing	pyOpenSSL
PyKCS11	1.3.3	Interface for smart card operations	PyKCS11
pem	16.1.10	Splits multiple PEMs in a single file	pem

Table 1: List of all required external libraries

3 Changes

Added changes in this milestone (M2) include PKCS11 support and a few features included when using x509 certificates: signature validation and chain

validation as well as certificate validation against the Certificate Revocation Lists (CRL).

A few new fields were added when a client connects to another client, server or when sending a message. The public key field will contain the sender's public key signed with the Citizen Card as well the certificate chain.

The new Certificate field will contain the necessary certificate to properly identify the sender.

The Certificate Chain field will contain the certificate chain necessary to validate the sender's certificate excluding the root certificate. This means that the sender will only send the certificate chain contained inside the Citizen Card. The root certificate should already be contained in the destination system and will ensure that at least the root certificate (self signed) is trusted. In this case, the root certificate will be the *Baltimore CyberTrust Root*.

3.1 PKCS11 Support

To facilitate the usage of smart cards (in this case the Portuguese citizen card) we created a module that abstracted the usage of the PyPCKS11 library[2]. Note that this library already provides the abstraction of using and invoking the card operations, therefore, our abstraction module, *PKCS11 Wrapper*, abstracts the usage of signing keys, retrieving certificates and maintaining an card session.

By default Portuguese Citizen Card provides all certificates in the chain up to the root and two keys: one for signing and other for authentication. In this milestone we're using authentication key in order to validate the authentication when connecting to the server and other clients.

These functions provide the requirements needed by ChatSecure so that a client or server can authenticate another client. In order to do this the followed methods are provided:

3.1.1 smart_card_detected()

This function provides an interface to know if the system is ready for usage, including if the smart card reader is detected and if the card is inserted. This validation is needed in order to initiate a new ChatSecure client.

In order to accomplish this, we try to initialize a session with the card. If it isn't possible, either the smart card reader is not supported or the card is not inserted. When running the ChatSecure in debug mode, an error is shown accordingly.

3.1.2 get_available_certs_as_list()

This function lists all available certificates stored in the card. This allows us to know the name of the certificates in order to retrieve them if necessary.

PKCS11 Wrapper lists objects available in the card and filters where the class (*CKA_CLASS*) is a certificate (*CKO_CERTIFICATE*) as accordingly to the PKCS11 standard[1].

3.1.3 get_available_keys_as_list()

This function lists all available keys that are used to sign data. This keys are private and stored only in the smart card and cannot be retrieved since they are private and not exportable.

Analogously to *get_available_certs_as_list()* the object class is (*CKA_CLASS*) and the attribute is *CKO_PRIVATE_KEY* instead of *CKO_CERTIFICATE*[1].

3.1.4 get_certificate_pem()

This function returns a certificate using its name. By default it retrieves the most used key in our project, the authentication key.

In order to retrieve the certificate, the PyPCKS11 library is used. We retrieve the object where the *CKA_CLASS* is *CKO_CERTIFICATE* and its *CKA_LABEL* is the name of the certificate (e.g. for the authentication key, the certificate name is *"CITIZEN AUTHENTICATION CERTIFICATE"*.

Since the retrieved certificate is in *der* format we convert it to *PEM* for easier serialization and less incompatibility of binary data when sending it over network. This also allows better compatibility between the cryptography.io and PyOpenSSL libraries.

3.1.5 sign()

This function signs data using and Citizen Card key. For this we need to find the correct signing key and to create the according mechanism.

We search for the objects where the *CKA_LABEL* is the given key name, its *CKA_CLASS* is *CKO_PRIVATE_KEY* and the *CKA_KEY_TYPE* is *CKK_RSA*. This allows us to be sure that we're using the correct key to sign.

As for the mechanism we preferred to use *CKM_SHA256_RSA_PKCS* where

[1]Note that all CKA, CKO objects are provided by PCKS11 RSA Standard as referenced by [1]

the data to be signed is automatically hashed using SHA256 and signed accordingly.

3.2 x509 Support

For x509 management we continued to use cryptography.io library that provides all the needed functionality (signing verification, certificate importing and exporting using PEM format, retrieving details, revoke testing, etc)

To do this, we've completed the *CScrypto* module (that contains all the needed cryptographic functions) with the following new methods:

Function	Description
generate_x509_from_pem	Generates a x509 object from the given serialized certificate in PEM format
validate_x509_signature	Validates a message and its signature according to a given certificate
validate_chain	Validates a chain up to CA root
pretty_cert_text	Prints in a pretty way the certificate data using corresponding *NameOID*s of the certificate

Table 2: New functions included in *CScrypto* module

All the needed functions to deal with the new requirements of this milestone are provided by the methods described in the table above. Note that when signing we're using *PKCS1v15* for padding and SHA256 for hashing and the only dependency of *PyOpenSSL* is in chain validation, since cryptography.io is yet to support chain validation and OpenSSL's store feature is required.

3.2.1 x509 signature validation strategy

Our message authenticity relies in the signature of each message, signed with an asymmetric private key (K-), unique to each user.

The public asymmetric key (K+) is signed with the Citizen Cart Authentication Key, with provides a sub level of trust that requires using the Citizen Card just one time.

When sending a message, the data to sign is signed using the private key (K-), and attached to the message we include the following data:

This way, to validate the correctness of a message and its authenticity, the signature can be verified against the given public key (K+). The public key signature can be verified against the citizen card certificate (x509 field) and the certificate can be validated up to the root using the given x509 chain (explained in the next section).

Data	Description
data	ciphered data to send
signature	signature of the ciphered data using the private key (K-)
public key	public key (K+) of the asymmetric key used to sign the data and produce the signature
public key signature	signature data of the public key
x509 certificate	certificate of the smart card used to sign the public key
x509 chain	certificate chain from the x509 certificate used to sign the public key (K+) up to the root

Table 3: Data attached to each sent message

Using this strategy as identity preservation we created a new sub-level of trusted asymmetric keys that can be verified up to the root, allowing us to require the smart card just once when the asymmetric keys are generated and signed and full message confidentiality, integrity and authentication.

3.2.2 x509 chain validation

Signing each encrypted data for each message sent and validating every message with a signature only guarantees the encrypted data correctness and that the message was signed using the given key. It is also necessary to verify if the certificate belongs to a trusted chain, by verifying each signature and each issue up to a root.

For each message sent we also send the certificate chain. This makes it possible to the receiver (server or another user) to validate the full correctness of the cryptographic data.

Once the message is received the following steps are verified:

1. Verify if the signature of the data complies to the public key (K+)
2. Verify if the public key is signed using the provided x509 certificate
3. Verify if the x509 certificate is valid, which includes validating if the certificate:
   (a) Is in a valid trusted chain
   (b) Is revoked
   (c) Produced signature for the given public key (K+) is valid

If everything is correct, the message is valid. For the chain validation of the x509 certificates up to the root we've used OpenSSL store feature, that allows us to load trusted certificates and to validate a chain of certificates against he trusted CA.

For trusted authorities we've used *Mozilla CA Certificate Store* that are included in most operating systems. Once loaded, OpenSSL can solve the certificates graph and validate them using the loaded CA list.

In this case, the Portuguese Citizen Card is validated against the Baltimore CyberTrust CA[3], which is included in the Mozilla CA bundle,

4 Improved modules

For this milestone we took the opportunity to make overall improvements. Inheritance reduced the code base size and removed the duplicated code, and key derivation and message delivery reports provide more message integrity, security and control of communications flow.

4.1 Inheritance

In the previous milestone, server and ChatSecure client were two completely independent modules that had no shared code or connection, leading to duplicate source code. In this milestone we improved the core code and shared code across the modules.

Communication class is inherited by ChatSecure and Server and provides the following methods:

- flushOut()
- flushIn()
- handleRequest()

Connection class provides all information about a connection:

- Methods: parseReqs(), send(), close()
- Parameters: security level, id, bufin, bufout, message history, x509, x509 chain, etc

Communication class, that's shared between ChatSecure and Server, that provides network capabilities. Connection class (which was converted from a client class) that contains information related to a connection.

ChatSecure become it self a Connection and Communication and Server now handles connections and uses Communication class, as the diagram shows:

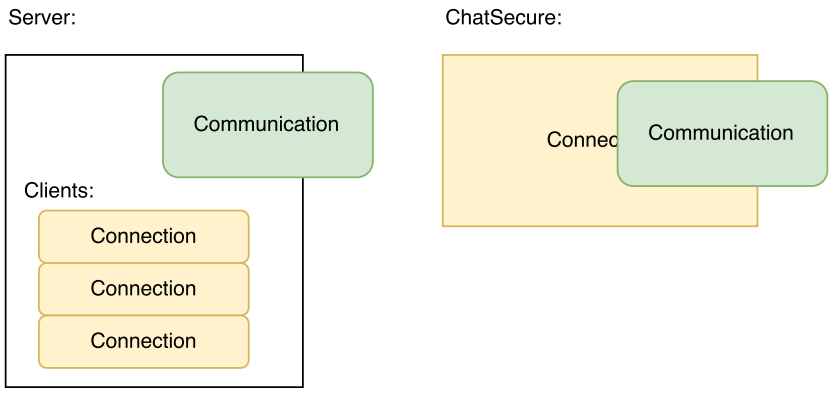

Figure 1: Classes inheritance diagram

Since *Connection* class itself stores all information related to a client (x509 certificate, cipher, etc) we used an instance of it for the ChatSecure itself.

There's also the Message class that provides the message delivery reports and message integrity, ordering and history.

4.2 CScrypto new functions

As described in M1 delivery, CScrypto (the module that includes all necessary cryptography functions for ChatSecure) was also improved and got new functions required for adding support for the client authentication process.

4.2.1 csc_aPrivate_sign()

Signs the data with a private key. This function is required as described in 3.2.1 section.

4.2.2 csc_aPublic_validate_signature()

Analogously to aPrivate_sign() this function validates if the signature is correct for a given data and if the data was signed with the given public key corresponding private key.

4.2.3 csc_derive_key()

Derives a key from the given master key and random. This is used to rotate a session key. In this phase of the project we rotate the key at every message received. More details in section 4.4

4.2.4 csc_generate_x509_cryptography_io_from_pem()

Generates a x509 from serialized certificate in PEM format in the cryptography.io format (the library used to handle cryptography in ChatSecure).

4.2.5 csc_validate_x509_signature()

Validates a message and its signature according to a given certificate.

4.2.6 csc_validate_chain()

Validates an chain up to CA root, including if the certificate is revoked. This solves the graph from the given certificates against a list of trusted CA certificates.

4.2.7 csc_pretty_cert_text()

Prints in a pretty way the certificate data. For this we get the data from the Name OIDs for the data we want to display in a readable format. In section 5 there's examples how the readable data looks when printed.

4.3 Message delivery reports

As an improvement we've also created a message class and all message texts sent between clients aren't just text. This class provides read receipts, ordering and uniqueness that are used for delivery reports. (message id are used for message acknowledge) by improving conversation consistency.

Now messages have an order that can be checked, are stored locally and it's status can be viewed and verified offline using the new *history* command.

4.4 Key derivation

Key derivation allows us to use a unique key on each message. Each key is derived from the session keys, either client-client or client-server, depending on the message being sent. On every sent message, a random is generated and used to derive one of the previous keys. That random is then added to the message on a new field and the derived key is used to cipher the message content.

When the receiver gets the message, he will use the random from the message to derive once again the session key and decipher the message with the key he derived.

This prevents an attacker from retrieving the session key as every message is ciphered with a key different than the previous one. Also because the derivation process is unidirectional in the way that you can't get the original key from the derived one, this method adds another security step to the overall implementation and provides full forward and backward secrecy.

4.5 Server's Participant Consistency

We've also implemented a way for the clients to identify if a user is using multiple devices to communicate (or different accesses), as the server now sends to the client the corresponding client-server connection unique identifier to the users who are communicating.

5 Examples

5.1 Whois

```
python ChatSecure.py
Name:  Diogo
Server cipher sha256 fingerprint is:
d913f65c02489bddc16e1845b09e4f7aed44c8a9329ccbf1a5032e7c8001fa98
Confirm (y/n)? y
[09:58:20] [Diogo-139934559672912] list
User list:

1. ID: Guilherme Level: 1
2. ID: Diogo Level: 1

[09:58:21] [Diogo-139934559672912] whois Guilherme
Nome           : GUILHERME JORGE CARDOSO
Organização    : Cartão de Cidadão
Identificação  : BI134612515
Instituicao    : Cidadão Português
Pais           : PT
Válido até     : 2021-01-03 23:59:59
Válido de      : 2016-01-04 16:59:29
```

In this example we can see that the user Diogo wanted to check information about Guilherme. Information from the Citizen Card is shown.

5.2 history

```
[09:58:36] [Diogo-139934559672912] history
History for Guilherme:
ID                  Message              Status
139934430395280     Bom dia!             READ
```

The user Diogo had previously started a connection to Guilherme and then typed the history command. A history of the communications between the two is shown.

5.3 client communication

```
Name:  Guilherme
Server cipher sha256 fingerprint is:
b05e0535485bfcce4d568a07c20feb722d64f39ec6f8e9ec6d128fd45ca4508c
Confirm (y/n)? y
Nome           : DIOGO CUNHA E SILVA
Organização    : Cartão de Cidadão
Identificação  : BI14617999
Instituicao    : Cidadão Português
Pais           : PT
Válido até     : 2021-03-01 12:59:59
Válido de      : 2016-05-07 15:59:39
[09:58:36] [Diogo]: Bom dia!
```

User Guilherme logs in and receives a communication from Diogo. The information of Diogo is shown along with the message.

6 Limitations

In this implementation milestone ChatSecure and its server throw an error and a debug message when an signature is invalid.

It also doesn't allow clients to authenticate to the server using revoked certificates, limiting the usage of clients that have citizen card expired.

This also greatly reduces the testing possibilities with *pki development kit*[4].

Therefore, ChatSecure secure expects a normal flow that everything is validated correctly and failing for all other usages where the integrity or anything else fails to validate.

7 Conclusion

With this project we were able to authenticate users using the Portuguese Citizen Card. We verified certificates with certificate chains and against certificate revocation lists.

Overall we were able to create a secure and authenticated channel of communication between users.

References

[1] RSA PKCS#11: Cryptographic Token Interface Standard - http://www.onefs.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm

[2] PyKCS11 - PKCS#11 Wrapper for Python - https://bitbucket.org/PyKCS11/pykcs11/overview

[3] https://www.digicert.com/digicert-root-certificates.htm

[4] Cartão de cidadão (Kit de desenvolvimento) - https://www.kitcc.pt/ccidadao/kits