47232 - Security

Secure Instant Messaging System

Guilherme Cardoso - 45726
gjc@ua.pt

Diogo Cunha - 67408
diogocunha294@ua.pt

October 2016

Abstract

In this report we describe how a security layer was implemented in a simple 1:N client end-to-end client communication, using a server as a relay.

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.

2 Instructions

2.1 Requirements

To run *ChatSecure* it's necessary to use *Python 2.7* and to have the following libraries installed:

Library	Minimal version	Description	pip
cryptography	1.5.2	Provides interfaces for cryptography cryptography operations	cryptography

Table 1: Required external libraries

The above requirements can be installed using pip[1] as follows:

```
$ pip2 install cryptography
```

Or using system package manager. Although our implementation runs on *Python* it is only compatible with *UNIX* systems, since *select*[2] that waits for both *sockets* and user input is not compatible with *Windows* due to a *Python* limitation.

2.2 Running the server and client

To run *ChatSecure* it is necessary to first start the server as follow:

```
$ python2 server.py
```

And then start as many clients as needed by running:

```
$ python2 ChatSecure.py [debug]
```

Command-line "debug" flag is optional and prints client activity during the work flow.

Clients have an command line interface and after connecting there's a prompt available that has a few commands:

Command	Description
help	Displays help
list	Displays connected users
exit	Close connection and exit closes the client

Table 2: Available commands in ChatSecure

Client to client communication uses the syntax destination followed by a colon. For example in Adam sending a message to Eve:

```
[00:00:01] [Adam-5] Eve: hello
```

Adam's ChatSecure client inits an end-to-end connection to Eve and the message is sent.

3 CSCrypto Library

To facilitate the usage of all cryptographic operations needed to provide our secure communication between clients and server, we created our own library, called CSCrypto, sustained in the *cryptography.io* library.

This module is shared both with client and server and includes all the operations necessary to deal with the ciphered data, abstracting it from the server and the client modules. To make it easier to distinguish from this library usage from the native *Python* libraries we prefixed every function signature with *csc*. It has the following methods:

3.0.1 csc_generate_aPrivKey()

Generates a random RSA Private key.

3.0.2 csc_load_aPrivKey_from_file(filename, password=None)

Loads an asymmetric key from a file.

3.0.3 csc_pickle_asymEnc_b64enc(aPubKey, data)

Pickle dumps the data, performs asymmetric cipher with the given public asymmetric key and encodes the result in base64.

3.0.4 csc_b64dec_asymDec_pickle(aPriKey, data)

Base 64 decodes the data, decipher with private asymmetric key and pickle loads.

3.0.5 csc_get_aPubKey_from_pem(pem)

Generate asymmetric key from pem text data.

3.0.6 csc_get_pem_from_aPriKey(aPriKey)

Generates the public bytes (in PEM format) of the private asymmetric key.

3.0.7 csc_generate_symKey(password, iv)

Uses the given password and iv to generate a symmetric AES256 key.

3.0.8 csc_b64dec_symDec_pickle(key, data)

Base 64 decodes the data, decipher with symmetric key and pickle loads. This includes the unpadding process.

3.0.9 csc_pickle_symEnc_b64enc(key, data)

Pickle dumps an object, pads, ciphers and base64 encodes the data.

3.0.10 csc_HMAC_update_finalize(key, data)

Finalize the current context and return the message digest as bytes.

3.0.11 csc_HMAC_update_verify(key, hmacdata, data)

Finalize the current context and return the message digest as bytes.

4 Security Implementation

Our security layer implementation consists in three phases and its protocol is similar when a client connects to server and when connects to another client.

It requires to the user to know the *public key* of the server so we provide an easy way to confirm the public key in our client implementations.

The following notions are used in our architecture:

4.1 Asymmetric Key Pair

An asymmetric key pair is a pair of keys that are not identical (therefore asymmetric) to each other. One of the keys is public and the other is private.

This means that one of the keys (private) is kept secret, and it's only known by the entity that created it. The other key (public) may be released to everyone. If the public key is used to cipher, only the corresponding private key can be used to decipher.

Our current implementation uses RSA as an asymmetric key and for ciphering uses *OAEP* (Optimal Asymmetric cipher Padding) with *SHA256* both for masking and padding data.

4.2 Session Key

In order to avoid some limitations of the asymmetric keys (speed and block size limit) whenever a message is sent as that would make the messages much bigger than they need to be, we establish a session key whenever a client connects to the server.

The client generates a symmetric key upon connection and sends that key to the server ciphered with the server public key.

The server will then decipher the key using its own private key and store the session key for that client. From then on, every communication to or from a client will be ciphered using that client's session key.

Currently we support AES-128 and AES-256 using Cipher Block Chaining (CBC) and HMAC (Hash-based message authentication code) using *SHA256*

as a integrity validation with cipher then MAC strategy. Server and client must negotiate the preferred mode as described in the next section.

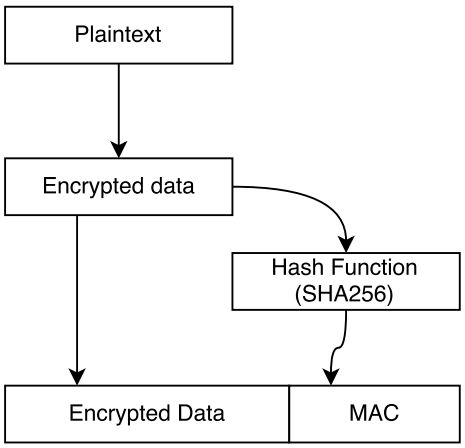

Figure 1: Encrypt then MAC for ciphered data

4.3 Client to Server

The client is required to register at the server using the *connect* message type, this allows to the server and user to negotiate and establish a secure connection.

In our security approach, server proposes a public key (exported in *PEM* format) (that should be known to the client), and that key is used to negotiate the symmetric key for the session between the client and the server.

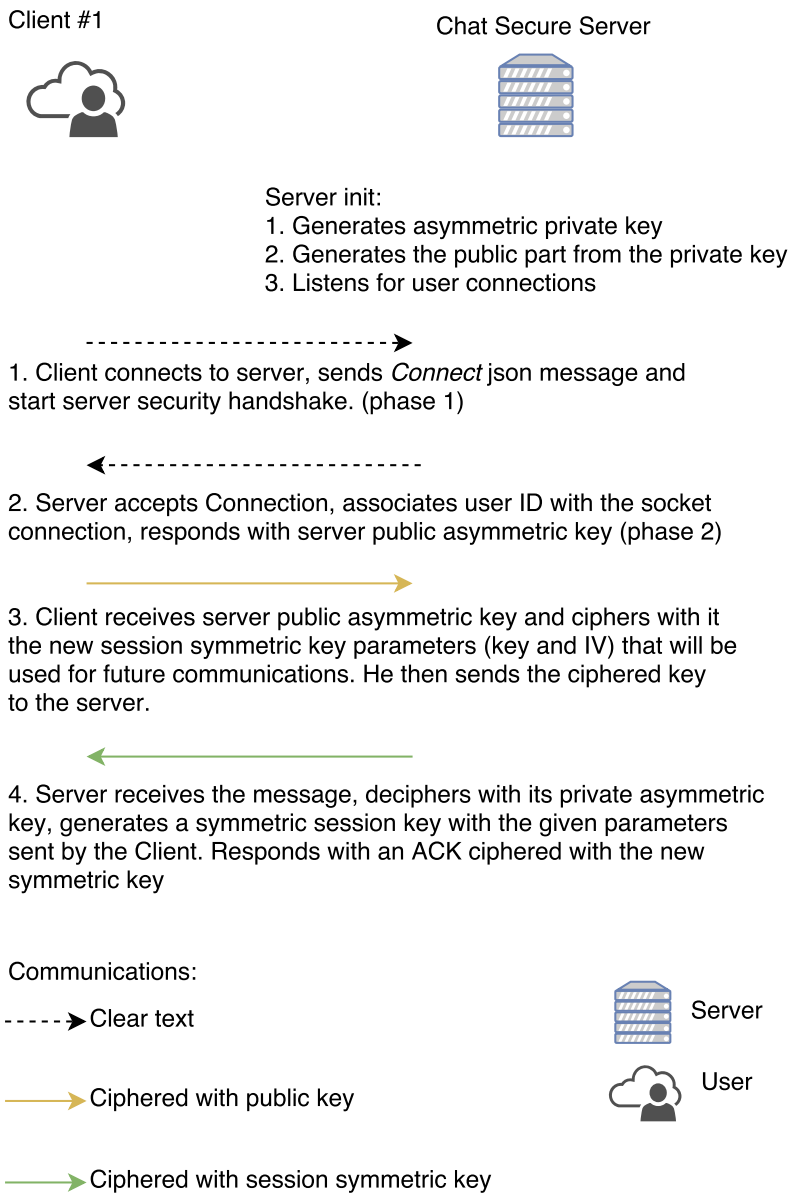

Figure 2: Client to server security handshake

In this process the client establishes a unique secure session with the server with a key that will be shared across all the following communications between that client and the server.

Given a message M, an asymmetric private key *(Kx-)* and a public key *(Kx+)*, the connect works as follows:

1. Client tries to establish a socket connection with the server, in which

is included the client's user name, compatible ciphering algorithms and initializes the security process handshake.

2. Server responds to the user, agreeing with registration of his user name and sending an asymmetric public key *(Kx+)* and the interception of the compatible cipher methods between the user and the server, with the connect message response.

3. The server's public key *(Kx+)* fingerprint is presented to user, which can be accepted or not. Th user is expected to be able to verify the server asymmetric public key fingerprint.

4. If the client accepts the server asymmetric public key *(Kx+)*, a symmetric AES-CBC key (K) will be generated. The client then ciphers the key generation details (key password and IV) using the server's asymmetric key and sends it to the server.

5. Server receives the message, deciphers with private asymmetric key *(Kx-)*, and generates an equal symmetric AES-CBC key and associates this key to the client. From now on, all messages to client will be ciphered and deciphered using this symmetric key

Established the connection all messages between client and the server must be ciphered and deciphered, using the *payload* field of the *secure json* message type.

Note that in this process server and client negotiate the compatible ciphering algorithms between the two. In our case we support key exchange using RSA, supported session keys are AES-256-CBC and AES-128-CBC, both using HMAC as integrity.

Same goes for every connected user at the server. All keys are unique to each user and valid only for the session. After a client disconnection, the server deletes the used key and it's parameters from memory and a new session will require a new security handshake.

Although the client to server communication only requires one cipher and decipher, when two clients are trading messages and when the server is working as a relay, it must decipher the message and cipher with the destination user key, as described in the section bellow.

4.4 Client To Client

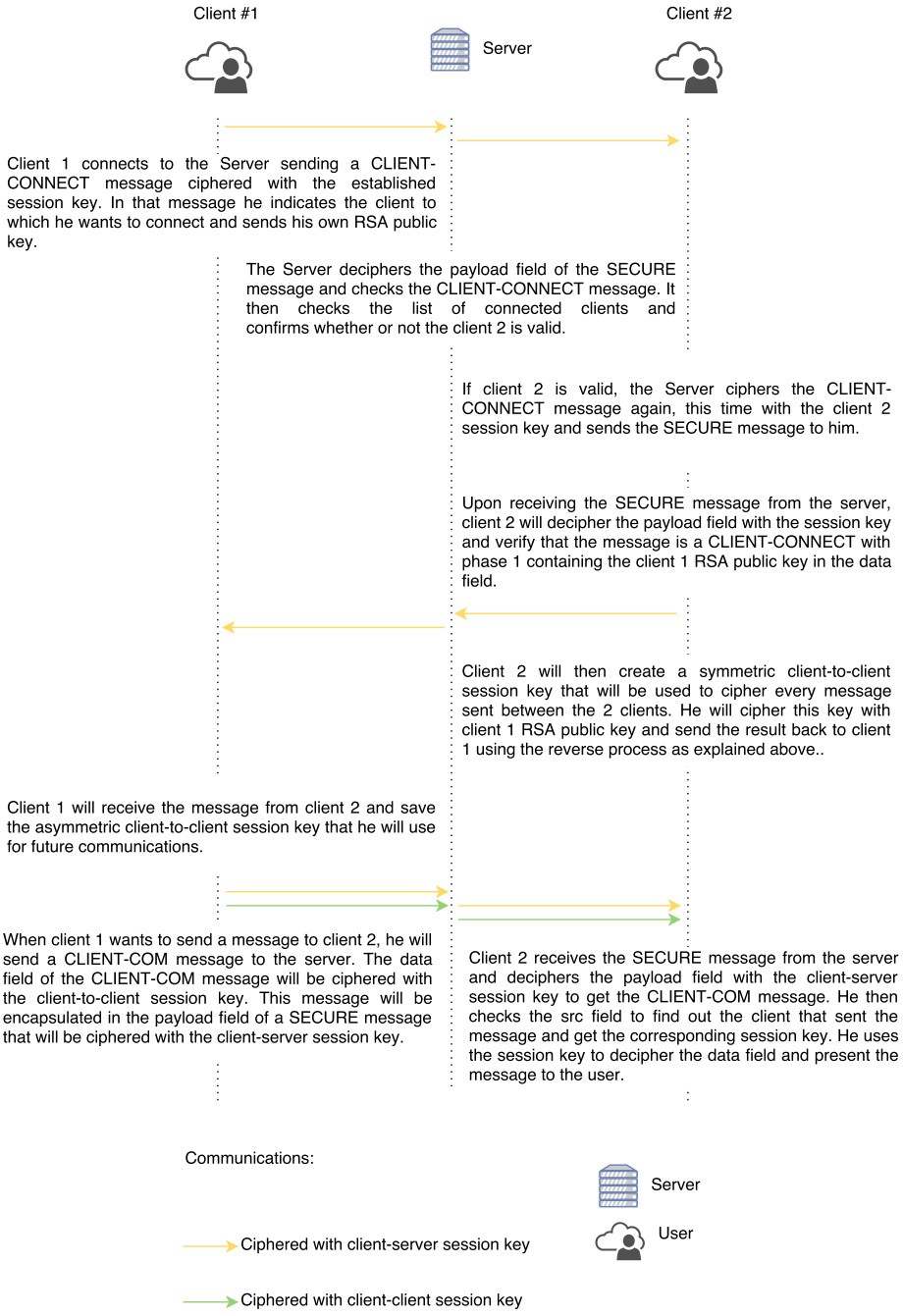

Figure 3: Client to client security handshake

When the clients want to send messages to each other, each message will always be relayed through the server. To allow this connection to happen in a secure way, we use a security handshake as explain in the image above.

On a higher level, the payload field of each SECURE message from client to server will be ciphered using the previously established session key. The server will receive the message, decipher it using the session key and assert the destination of the message. Then it will cipher the message again with the destination client session key and send it.

The client will receive the message and create a client-to-client session key that he will cipher with the first client public key and send to him. On future communications the clients will use that key to cipher the secret message from the users. Even if the server tried to read the secret message, it wouldn't be able to do so as the message itself is ciphered with the client-to-client session key that only the two clients know.

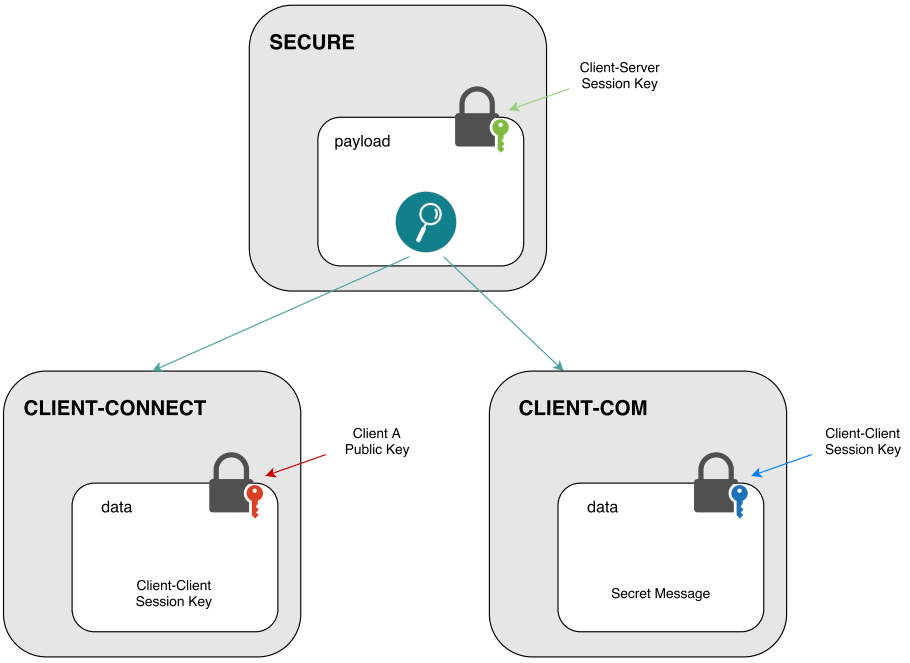

Figure 4: Secure payload and client-connect/client-com encapsulation

5 Conclusion

Currently there are a few limitation in our secure communications layer, that are

- Our current implementation requires the user to validate and verify the public prompted public key to the user. This can be solved by signing

public key and have an chain of trust.

- There's no message signature that guarantees that the message is coming from the right source. This can be solved by signing every message.
- Receipts are not implemented. We expect to implement message delivery and read receipts in the next *ChatSecure* implementation

References

[1] https://pypi.python.org/pypi

[2] https://docs.python.org/2/library/select.html - "Note that on Windows, it only works for sockets"

[3] https://cryptography.io/