SIMS

Secure Instant Messaging System

Milestone 1

Security 2016

p2g2

Hugo Fragata 73875

Gonçalo Grilo 72608

1.Introduction

This report describes the first milestone of the Secure Instant Messaging System project for the Security class of 2016 of the Degree in Computer and Telematics Engineering at the University of Aveiro.

Secure Instant Messaging System is a, as the name states, a instant messaging tool which has certain security properties in communications.

Our goal is to provide the end user with secrecy, integrity and authentication for their messaging, sending and receiving.

We use Python 2.7 64bit and the cryptography.io python library and PyQt4.

2.Architecture

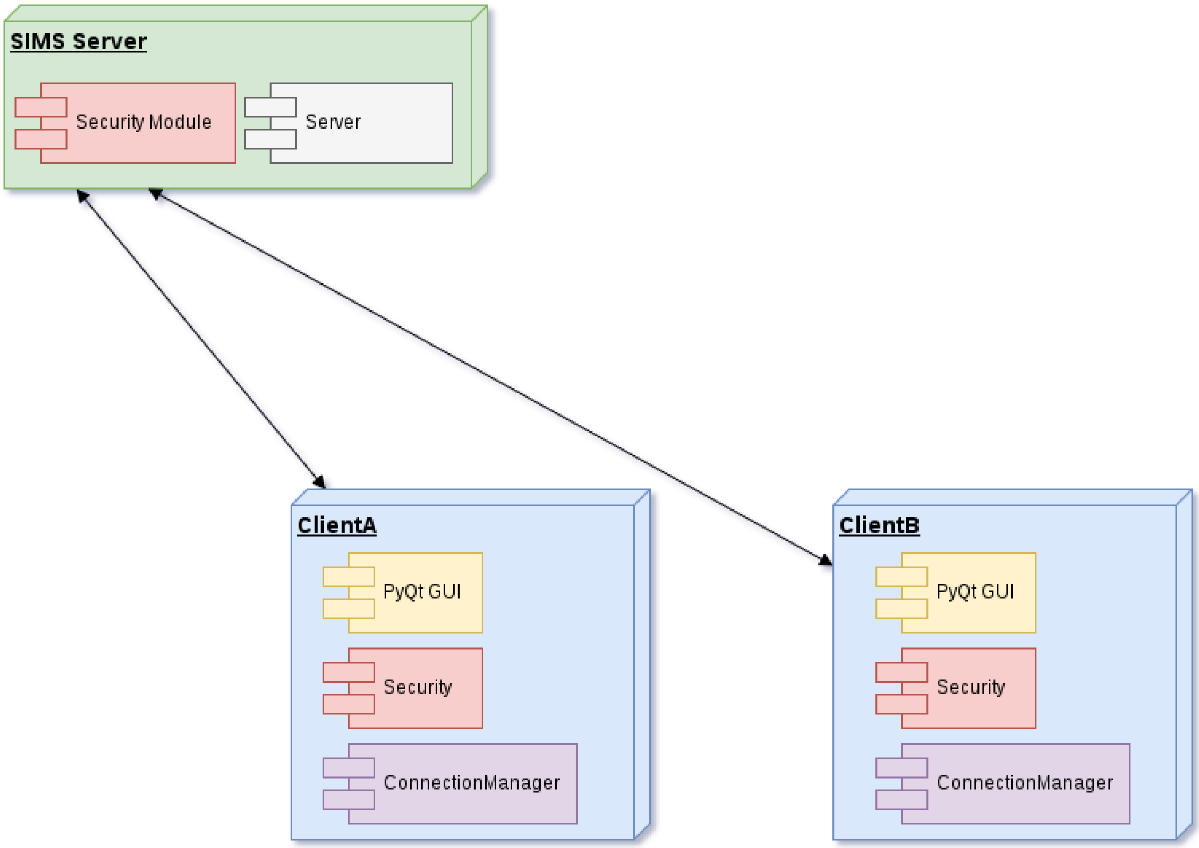


Fig1. SIMS M1 System Architecture Diagram

2.1 SIMS Server

The SIMS server is composed by two modules: Security and the actual server.

Security module is the same as the client's security module. And the server is an modification to our needs of the provided professor's server.

2.2 SIMS Client

The clients are comprised of the the security module and the PyQt GUI module, which we're going to explain in another section of this document.

The connection manager module does all the dirty work of handshaking the server and their peer client and forming and ciphering the messages and giving them to the GUI.

3.Messages

In this section we explain the json structure of the messages. Subtitles with a "*" at the end mean that they were not present or altered in the original professor's description of the SIMS.

3.1 CONNECT

Message type used to initialize a new session between client and server.

```
{
"type": "connect",
"phase": <int>,
"name": "client name",
"id": <message id>,
"ciphers": [<cipher combination to use>, ]
"data": <JSON or base64 encoded if binary (optional)>
}
```

3.2 SECURE

Message type used to encapsulate data sent securely between client and server upon their initial connection.

```
{
"type": "secure",
"sa-data": <JSON or base64 encoded if binary>,
"payload": <JSON Sub Message>
}
```

3.3 DISCONNECT*

Message type used to disconnect the session between client and server.

```
{
"type": "connect",
"name": "client name",
"id": <message id>
}
```

3.4 Encapsulated Messages

These messages are sent between the client and server, using the secure session already established, encapsulated as a payload of a SECURE message. Therefore, these messages are never transmitted in clear text.

3.4.1 LIST

List clients connected to the server.

```
{
"type": "list"
"data": [{client-data}, ...]
}
```

3.4.2 CLIENT-CONNECT

Message used for a client to start a secure session with another client.

```
{
"type": "client-connect",
"src": <id_source>,
"dst": <id_destination>,
"phase": <int>,
"ciphers": [<cipher combination to use>, ],
"data": <JSON or base64 encoded>
}
```

3.4.3 CLIENT-DISCONNECT

Sent by a client in order to tear down the end-to-end session with another client.

```
{
"type": "client-disconnect",
"src": <id_source>,
"dst": <id_destination>,
"data": <JSON or base64 encoded>
}
```

3.4.4 CLIENT-COM*

Message sent between clients with an actual text message to be presented to users.

```
{
"type": "client-com",
"src": <id_source>,
"dst": <id_destination>,
“verification”:<HMAC of message expect this field>*
“salt”:<salt used in this message’s kdf>*
"data": <JSON or base64 encoded>
}
```

3.4.5 ACK

Acknowledgment message sent after every other message is received to the peer.

```
{
"type": "ack",
"src": <id_source>,
"dst": <id_destination>,
"data": <hashed data>
}
```

3.5 ACK*

Acknowledgment message sent after every other message is received to the peer.

```
{
"type": "connect",
"name": "client name",
"id": <message id>
}
```

4.Protocols

4.1 Client - Server Handshake

Using CONNECT (3.1) messages client and server agree on a supported cipher suite and trade securely a session_key.

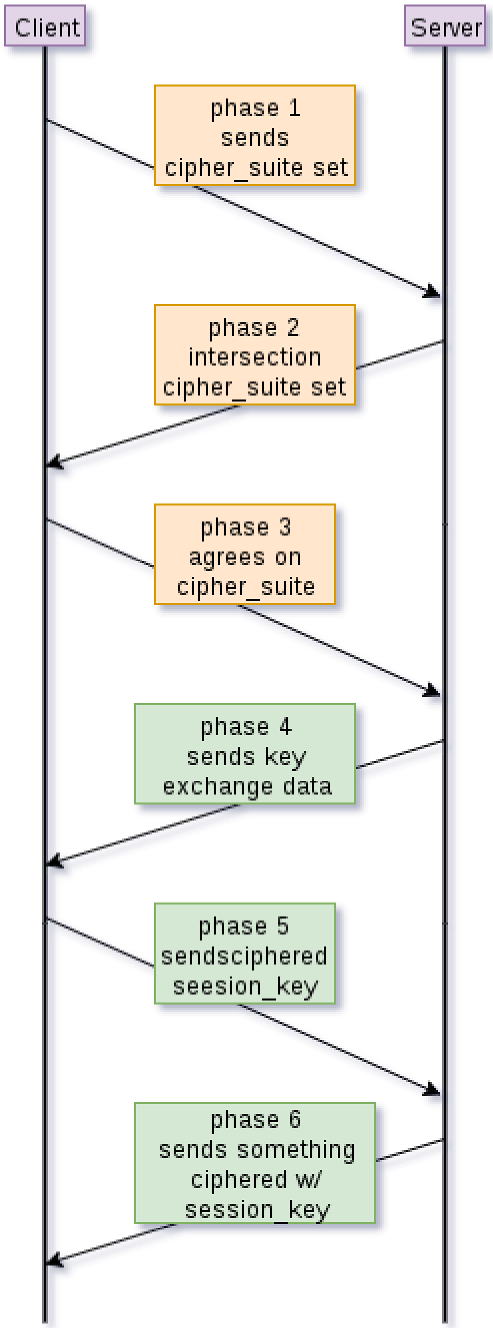
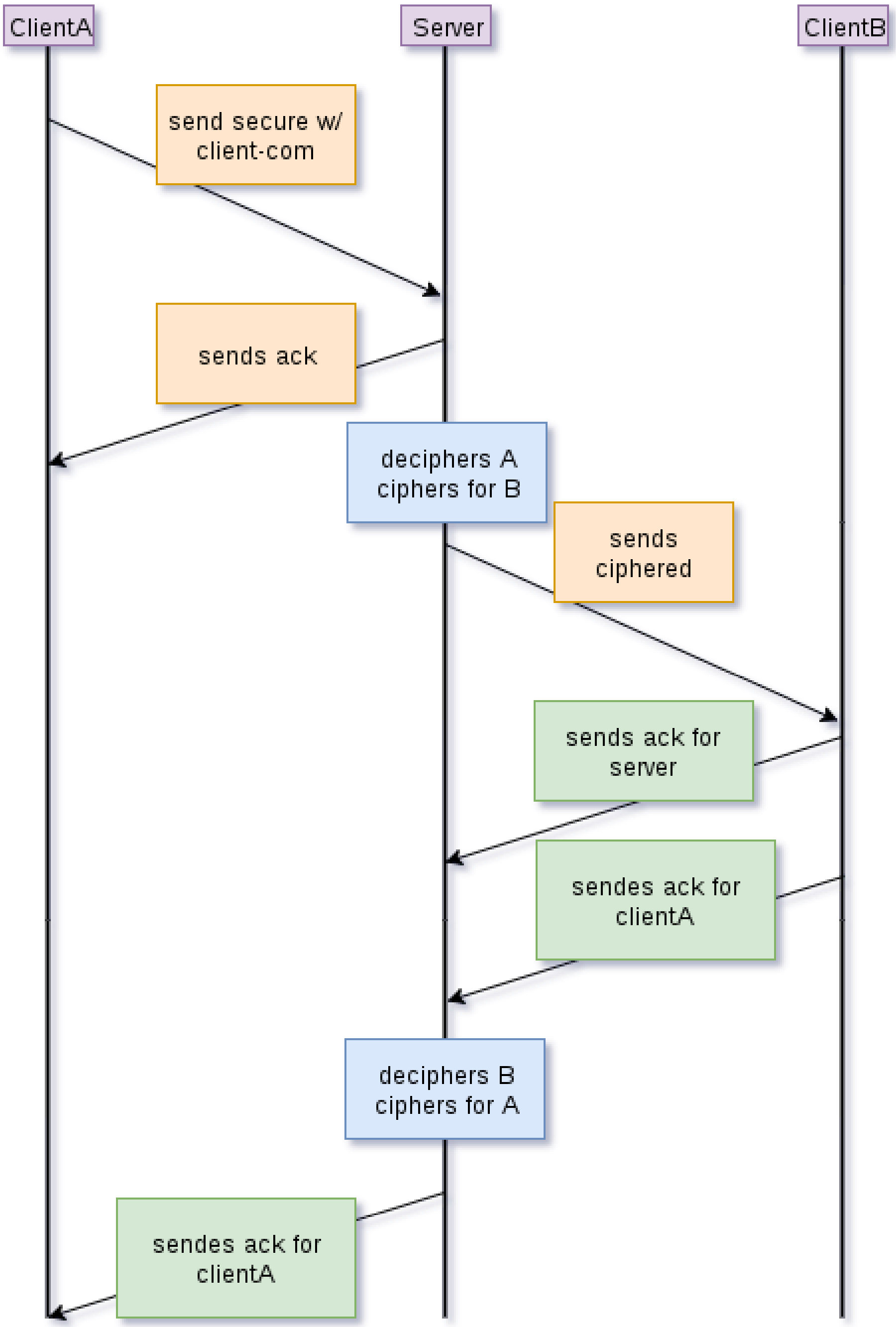


Fig2. SIMS Client-Server Handshake

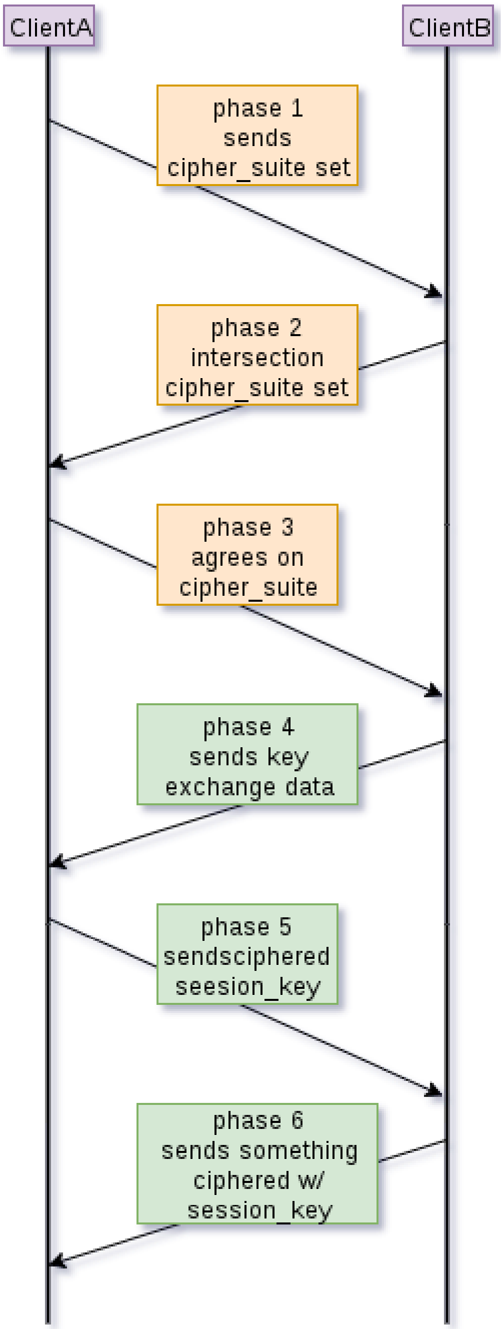
4.3 Client - Client Server Redirection

Two clients communicate between the server. Each of them have a session key for symmetric ciphering with the server. The server get a secure message from ClientA with the "payload" field ciphered with it's respective key. Then it deciphers it, looks at the type field, and if it's meant to go to another client, the server ciphers the "Secure"(3.2) message with ClientB's agreed symmetric key and sends it.



4.3 Client - Client Handshake

After two clients are connected to the server and able to trade messages with the server they perform a similar handshake to 4.1 in which they trade symmetric keys in order to cipher the "data" field and validate the "validation" field of Client-Com(3.4.4) messages. This entire handshake only happens after 4.3 is working and only inside Secure(3.2) messages.



5.Security

5.0 Cipher Suites

Our SIMS implementation supports two secure cipher suites: "RSA_WITH_AES_128_CBC_SHA256" and “ECDHE_WITH_AES_128_CBC_SHA256". They differ in the kind of key exchange protocol is used. Bulk symmetric encryption and HMACs are exactly the same. It supports different cipher suites between Client-Server and Client-Client.

5.1 Elliptic Curve Diffie Hellman

Our security module allows for ECDH. We have a Key Pair generating function and a Get Shared Secret function. We use the standards in the cryptography.io library. The key exchange is explained in 4.1 and/or 4.3.

5.2 RSA Key Exchange

The server has a fixed RSA Key Pair 4096 bit. We have functions for Generating and Loading RSA(Private/Public)Key objects. And most importantly Sign/Verify and Encrypt/Decrypt. The key exchange is explained in 4.1 and/or 4.3. The cryptography.io requires passing a hashing algorithm. We choose SHA256. For padding we use both OAEP and MGF1.

5.3 Symmetric Encryption, Decryption

We use Fernet from cryptography.io for all symmetric cryptography. Fernet does encryption and authentication. Built with AES128 in CBC mode, PKCS7 for padding and SHA256 for HMAC. The Key Format is (Signing-key || Encryption-key). The Token Format or Cipher Format is (Version || Timestamp || IV || Ciphertext || HMAC). For the HMAC Format is used SHA256(Version || Timestamp || IV || Ciphertext).

Bit length of fields:

Signing-key, 128 bits
Encryption-key, 128 bits
Version, 8 bits
Timestamp, 64 bits
IV, 128 bits
Ciphertext, variable length, multiple of 128 bits
HMAC, 256 bits

5.4 HMAC

Although Fernet provides a HMAC already it only verifies the 'data' field of a 'Client-Com'(3.4.4). There are other fields where injection or altering of the fields may happen. We therefore added our 'verification' field to 'Client-Com'(3.4.4) messages.

This 'verification' field contains a HMAC which is calculated using RFC2104's norms. We take the 'src', 'dst' and 'data' fields of a 'Client-Com'(3.4.4) and the secret shared session key agreed upon between the clients.

Then we calculate HMAC(k, m) = SHA256(k, SHA256(k + m)) where k=secret shared session key, "+" denotes concatenation and m=src+dst+data.

5.5 Client-Com's encryption key system

In order to provide more entropy we have a different encryption/decryption Fernet Key that we get by passing the field 'data' and the current client session key through cryptography.io's PBKDF2HMAC key derivation function using SHA256 for hashing and 100000 iterations.

5.6 Client-Client session key rotation

In order to provide forward and backwards secrecy Clients perform a key exchange using their agreed upon algorithm and the session key is updated and both ends and the old session key is discarded.

This happens every other (100?) messages traded.

6.Interface

7. References

https://github.com/fernet/spec/blob/master/Spec.md
https://cryptography.io/en/latest/
https://tools.ietf.org/html/rfc2104