# SIMS

## Secure Instant Messaging System

## Milestone 1

Security 2016

p2g2

Hugo Fragata 73875

Gonçalo Grilo 72608

# 1.Introduction

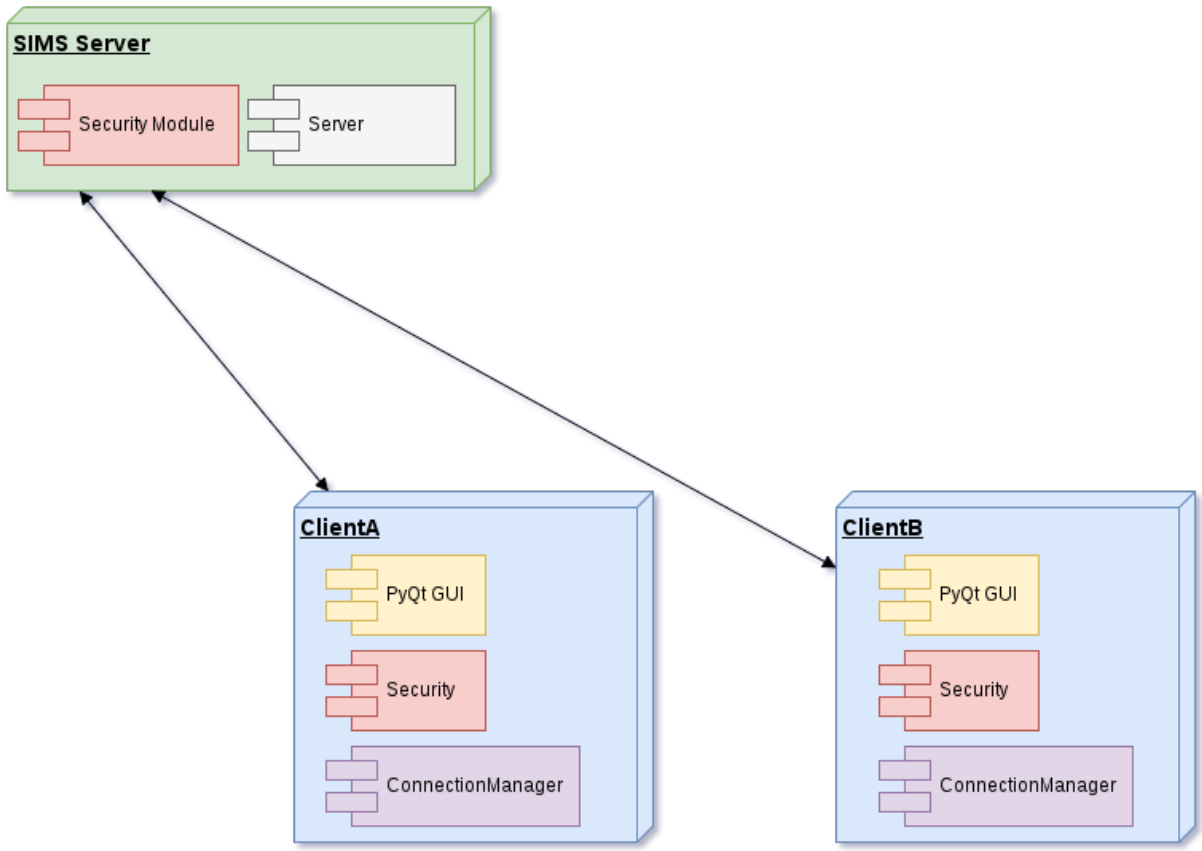
This report describes the first milestone of the Secure Instant Messaging System project for the Security class of 2016 of the Degree in Computer and Telematics Engineering at the University of Aveiro.

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, the cryptography.io python library and PyQt4.

# 2.Architecture



## 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>*
''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*

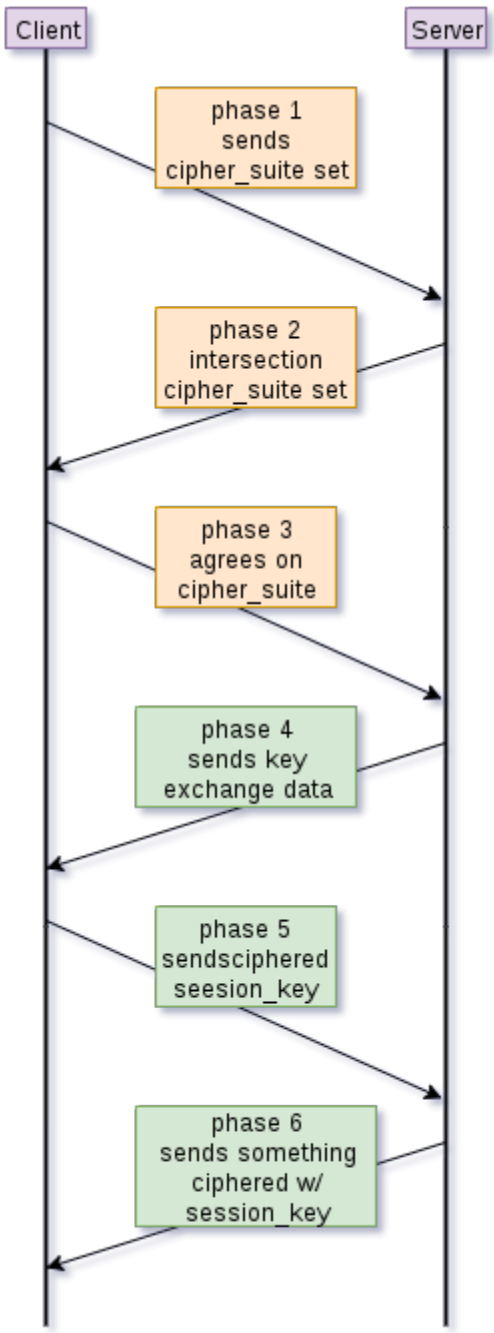
Acknowledgment message sent after every other message is received to the peer.

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

# 4.Protocols

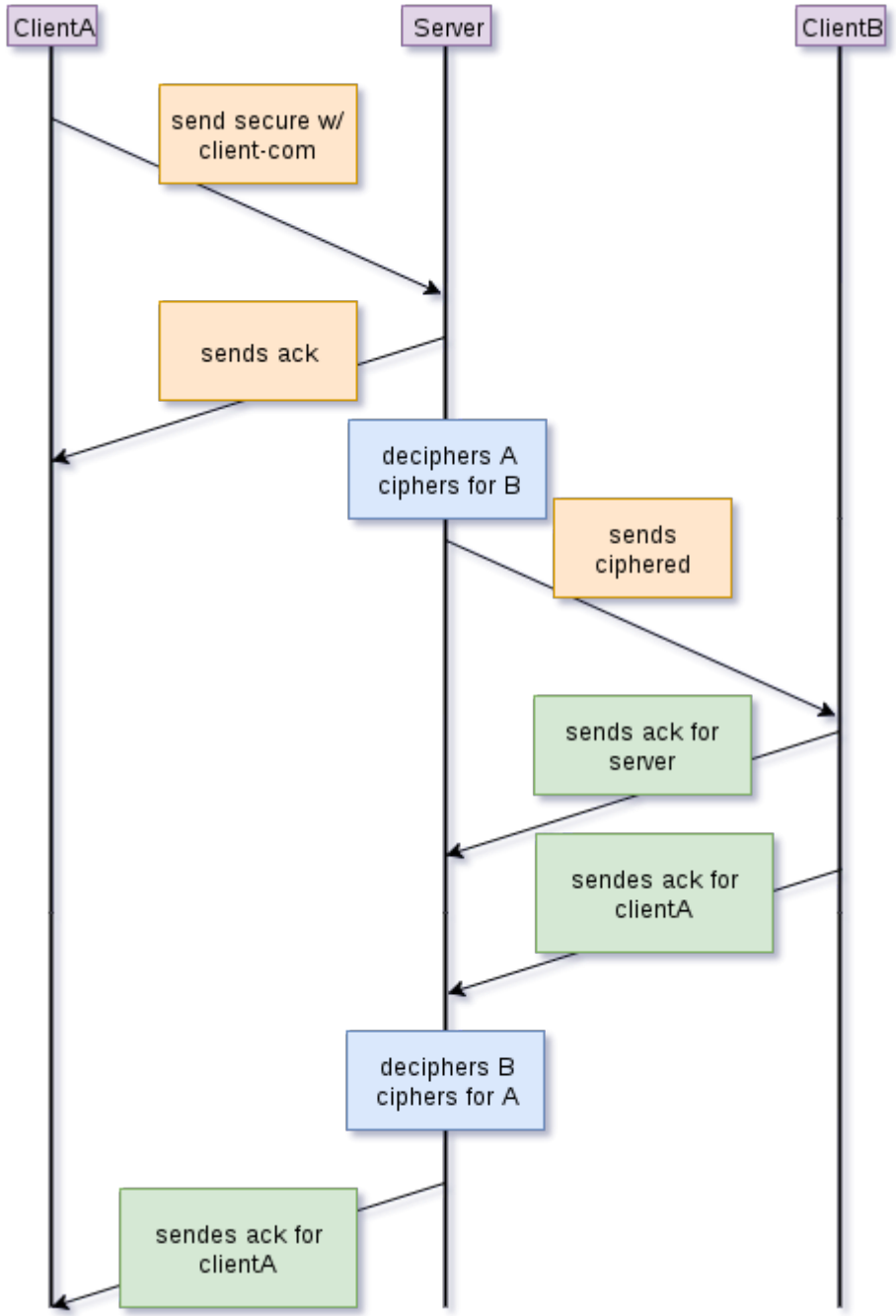
## 4.1 Client - Server Handshake

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



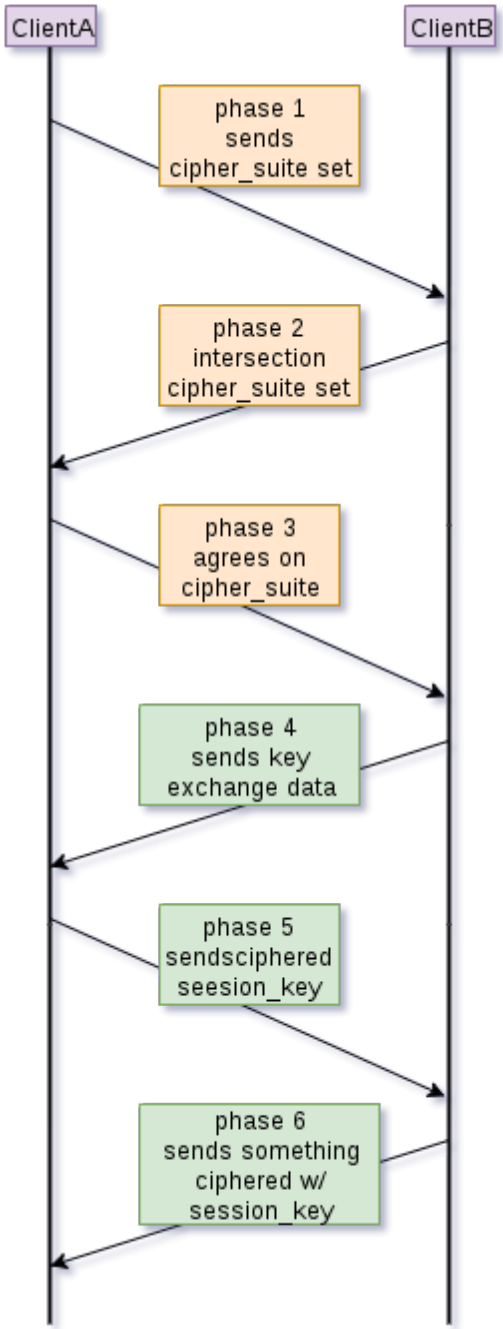
## 4.3 Client - Client Server Redirection

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 RSA Key Exchange

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. For padding we use both OAEP with the masking generation function MGF1 both of these use the hashing algorithm SHA256.

After cipher suite agreement, the server/responder peer generates an ephemeral RSA key pair with 4096 bits and sends the public key to the initiator. The initiator generates a session key, and encrypts it with the received public key, using OAEP with SHA256 for padding, the resulting encrypted text is sent to the server/responder. After the reception of this packet the peer can decrypt its contents with the private key generated earlier. After this step both parties have a shared secret that will be used to encrypt (see 5.3) the rest of the communications in this session.

## 5.2 Elliptic Curve Diffie Hellman

Our security module allows for ECDH key exchange. We have a Key Pair generating function and a Get Shared Secret function. We use a NIST P-384 elliptic curve to generate the key pairs. To generate the session key we apply a concatenation key derivation function to bring the key to the desired size.

The key exchange is similar to 5.2, but each peer generates an ephemeral key pair and sends the public component to the other peer. After receiving the public key from the peer each client will generate the shared secret using his private key and the received key. The key exchange is explained in 4.1 and/or 4.3.

## 5.3 Symmetric Encryption and Decryption

We use Fernet from cryptography.io for all symmetric cryptography. Fernet does encryption and authentication. The reason we choose Fernet is to mitigate cryptographic implementation mistakes. It is 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 Communications forward secrecy

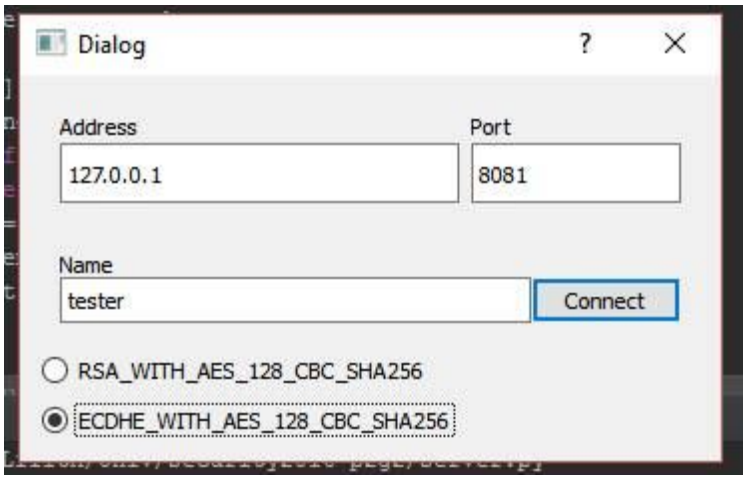
In order to provide forward secrecy we use ephemeral keys for every session, so we have a different encryption/decryption Fernet Key every time the client connects.

### 5.5.1 Client-Client session key rotation (not implemented)

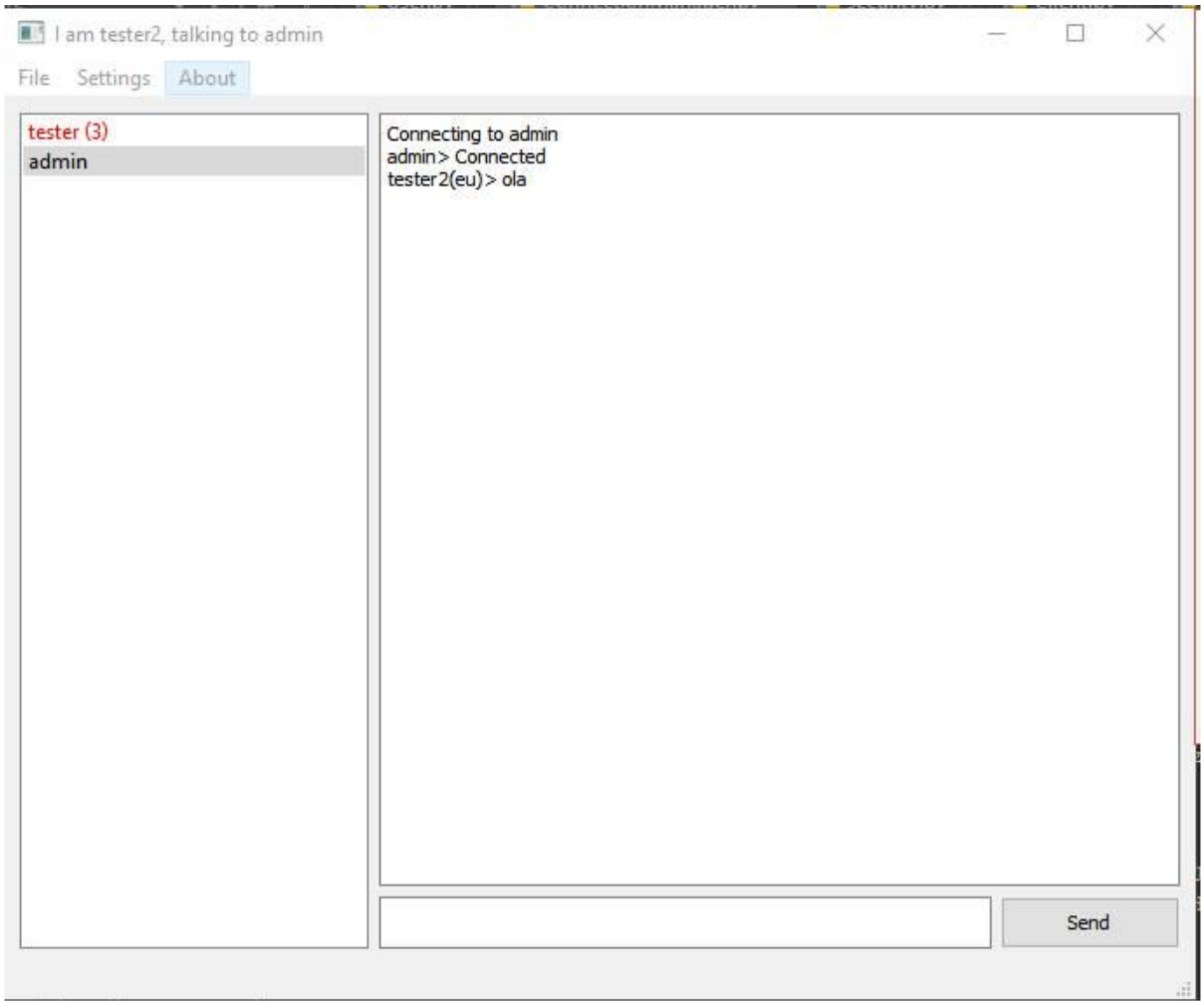
In order to provide forward secrecy Clients perform a key exchange using their agreed upon algorithm every session and the session key is never reused. Each session lasts until the client disconnects from the server or from other client.

We plan on implement a system to change this key more often (every roundtrip using ECDH or on each 100 messages) in order to provide perfect forward secrecy.

# 6.Interface




Choosing a desired Cipher Suite, Port, IP and Name.




Messaging interface. Left box lists the connected users, the ones in red already tried to contact this user. The app window title denotes who we are and who we are talking to.

# 7. References

https://github.com/fernet/spec/blob/master/Spec.md
https://cryptography.io/en/latest/
https://tools.ietf.org/html/rfc2104
https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf