Computer and Network Security

Project Report

Abdelrahman Osama 34-5226 T-16
Hady Yasser 34-3051 T-08
Mark Ragaee 34-3956 T-10
Islam Elgohary 34-5639 T-08
Mohamed Mohy Elgamal 34-14295 T-09

Summary

For this project, we implemented a chat application that supports peer to peer messaging. The application is a hybrid between client server and peer to peer applications. The application uses a server that allows users to register and updates the user list for every registered users. The users can then use this users list to directly message other registered users. The application offers the following security measurements: Non-repudiation , authentication and confidentiality.

Motivation

Chat applications are everywhere and have a huge set of users. Unfortunately only recently these applications started addressing security concerns. The purpose of this project, which implements a simple yet secure chat application, is purely academic. We hope that it could carry on to future researchers in hopes it will guide research towards more secure chat applications.

Implementation

In this section, we briefly discuss how each component of the project is implemented.

Server

The server listens on a port until a client tries to connect. Once a client tries to connect, the server accepts the connection then generates a response based on the message type. Here is a list of allowed message types and how the server responds to them:

1) **REG**: If the username is not already taken by another client, the server adds the user to its list of users and send them a list of all other registered user.
2) **LOGIN**: If the username and password, sent in this type of message, match those of a user stored, the client is logged and sent a message containing ta list of all currently registered users.
3) **FTCH**: Sends a list of all users to the client.
4) **BYE**: Closes the connection with the client, who sent the message and replies with a message of the same type, so the client also closes their connection.

Client:

Once a client attempts to connect, it is prompted to either start with a message of type **LOGIN** or **REG** it then can send messages of other types. The different types as well as what they represent, are listed below:

1) **REG**: Indicates that a client is trying to register fo the application.
2) **LOGIN**: Indicates that a client is trying to login to the application.
3) **FTCH**: Indicates that a registered client is trying to obtain the list of currently registered clients.
4) **BYE**: Indicates that a user is trying to log off the application.

Design choice:

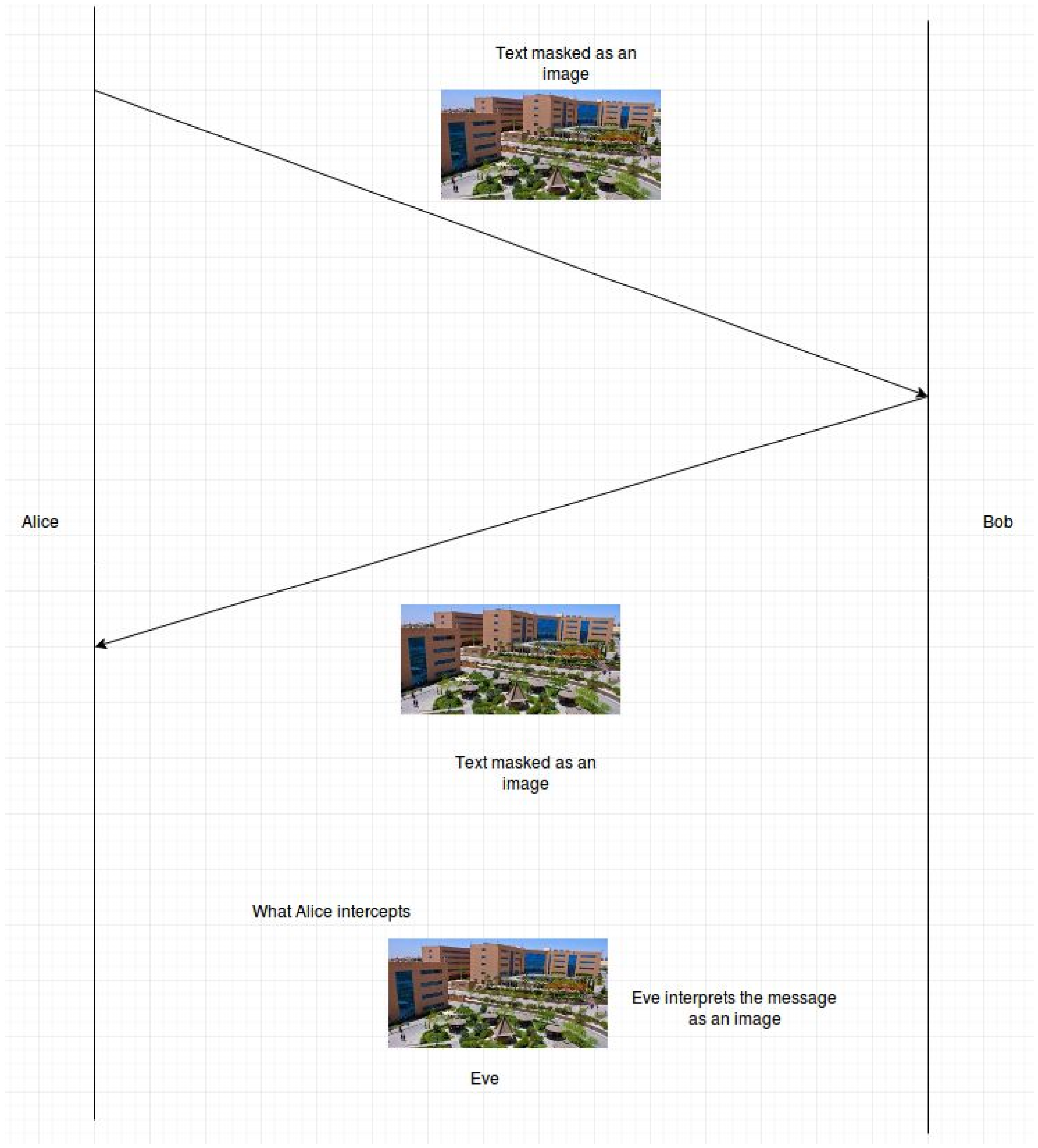
The main encryption technique used in this project was **Steganography.** Despite the fact that there are a lot of techniques that can be used to encrypt messages, **Steganography** is one of the few really interesting techniques. It masks the messages as an image, so even if the message was intercepted, the person intercepting the message might be tricked into believing that the intended message was the image they intercepted. We used **Steganography** in this project to encrypt messages being shared between clients in peer to peer connection, to achieve confidentiality between users. It is worth mentioning that using **Steganography** can actually be risky since the technique is know, which is why different images must be used for encryption to make it less suspicious and the pool of images available needs to be updated frequently. More details about **Steganography** are discussed in the next section.

Attack scenarios

If Alice and Bob are trying to exchange messages, Imagine the following flow:

1) Alice and Bob register/login to the server
2) Alice tries to send Bob a Message
3) A “man” in the middle Eve tries to intercept the message.
4) Eve is deceived into thinking the intended message was an image and keeps eavesdropping as she believes that Bob and Alice are exchanging images.

As we can clearly see, Eve might think she was successful however the users were actually exchanging text messages disguised in images and Eve did not really get the interpret the intended messages. This is still dangerous because if eve suspects they are using **Steganography**, she could easily decode it by guessing the encoding algorithm.



Steganography

`encode_text`

The function takes the text string as an input

The algorithm calculates the length of the string

This Length is coded on 2 bytes so the text size can be up to 65536 bytes long, the function takes an int as input and return the binary value of it as a byte

The output of the above function is then passed to another function to Put text length coded on 4 bytes into the image by getting the pixel value as a list, and loop over the input bits one by one and checks if the bit is equal to 1 then the corresponding element in the list to the current channel will be equal to the value of that element bitwise ANDed with the maskOne value or ORed with the maskZero value and then move to the next slot and start over

All pixels of the image are scanned to select a pixel having a remainder of 1 when each RGB value is divided by eight. Change the color of the selected pixel so that there is no more than one.

After that, The algorithm gets the Unicode code point for a one-character string for each character in the text and use this unicode and get the binary value of it as a byte then puts this byte in the image using the same technique

`decode_text`

The function reads the bits in the 2 bytes specified on by one

It gets the value of the image at the current width, height and channel, convert it to int, then and it with the maskOne value and return 1 if the value is larger than 0 and 0 otherwise

The value then is converted to binary number, this bits are used to get all the bytes of the text

For each byte read, the algorithm gets the Unicode string with ordinal and then concatenate it to a string and then return this string after converting all the bytes

Original image

Text length 11

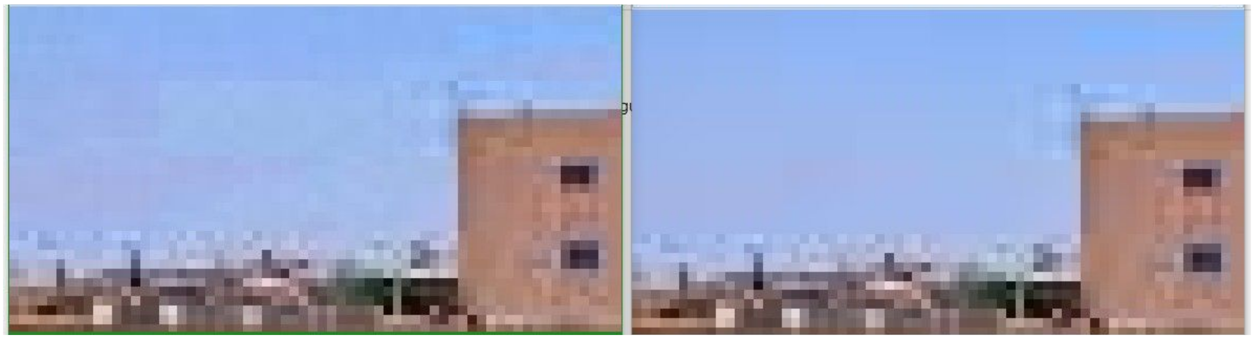
Text length 456

Text length 18339

Text length 55017

Capacity:
The length was coded on 2 bytes results in a string of maximum length of **65536 bytes**
Capacity = (usable %age) X (encoded bits) x (no.pixels)
Capacity = 318*159*3*4 = 606,744 bits

Fidelity:
We tested the algorithm on 4 sample texts of lengths 11, 456, 18339, 55017 characters respectively and noticed almost no difference for the first 3 samples and a slight noise started to show in the sample number 4 as shown in the image below



----Original image on right and the encoded text image on left----

Advantage and disadvantage of Steganography

Steganography is beneficial for securely storing sensitive data, such as hiding system passwords or keys within other files as if the communication is comprised the attacker won't detect the text at first place as he will see only an image. However, it can also pose serious problems because it's difficult to detect and it is based on the algorithm so if the algorithm is known to the attacker, it will be so easy to decode and get the plain text message.

Using RSA for Confidentiality

RSA Is an asymmetric Encryption algorithm in which each participant has both a public and a private key. To use **RSA** for confidentiality, we wanted to encrypt each message by the receiver’s public key so only they can decrypt and read the content of the message using their private key. However, doing that directly with the **RSA** would require a very large key size or else the plain text would be too large to be encrypted by the algorithm but using large keys would cause huge delay between sending and receiving each message. A possible solution was to use the **RSA** to encrypt a session key generated by another symmetric algorithm. We tried using **AES** to encrypt messages and decrypt the **AES** session key itself using **RSA**, however, that would be a huge overhead as we would need to implement sessions between the clients and handle cases when session key changes.

The first library that we tried to use for the **RSA** was **PyCrypto**. However, we realized that it was outdated so we moved to another fork of the repo which is **PyCryptodomex.**

We used **AES**, **PKCS1_OAEP** and **RSA.**
To generate a session key, we used **Cryptodome.Random.get_random_bytes.**

Functions:
RSA.generate: creates a new RSA key object.
Crypto.Cipher.PKCS1_OAEP.new: takes an RSA key object an creates a cipher object that is used to perform PKCS#1 OAEP encryption or decryption.
PKCS1OAEP_Cipher.encrypt: encrypts a byte string message whose length is not longer than the RSA modulus (in bytes) minus 2, minus twice the hash output size.
AES.new: Creates a new AES cipher.
encrypt_and_digest: Encrypts a message using AES.
PKCS1OAEP_Cipher.decrypt: decrypts a message using RSA.
Decrypt_and_verify: decrypts a message using AES.

Refrences
N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," in *Computer*, vol. 31, no. 2, pp. 26-34, Feb. 1998.

https://realpython.com/python-sockets/