UNIVERSITÀ DI PISA

Computer Engineering

Foundations Of Cybersecurity

Academic Year: 2023/2024

BBS: Bulletin Board System

Authors:

Saverio Mosti
Gabriele Pianigiani

Index

1 Introduction 3
1.1 An application overview 3
1.2 The source directory 3
2 Client 5
3 Server 5
4 The protocol 5
4.1 Key Exchange 6
4.2 Registration 7
4.3 Challenge 7
4.4 The session 7
4.4.1 List 7
4.4.2 Get 7
4.4.3 Add 8
4.4.3 Logout 8
5 Packets 8
5.1 Simple Message 8
5.2 Content Message 9
5.2.3 The cryptogram 9
5.3 RSAE Message 10

1 Introduction

The project regards a cryptographically secure application called BBS, developed in C++ for Linux.

A Bulletin Board System (BBS) is a distributed service where users can read messages and add their own, every user is identified by a nickname and a password.

A message is composed by the following fields: identifier, title, author, and body. The identifier field uniquely identifies the message within the BBS, while the author field specifies the nickname of the user who added the message to the BBS.

1.1 An application overview

The application is composed of a BBS Server, which stores all the BBS messages shared by users, and one or more clients that can communicate with the server via TCP protocol, each client can use five available functionalities:

- **Register** to the Server;
- **Login** to the Server;
- **Read** the last n messages available;
- **Download** a specified message;
- **Add** a new message;

1.2 The source directory

The project has been divided into six directories for readability and convenience. The following image represents the directory scheme of the project.

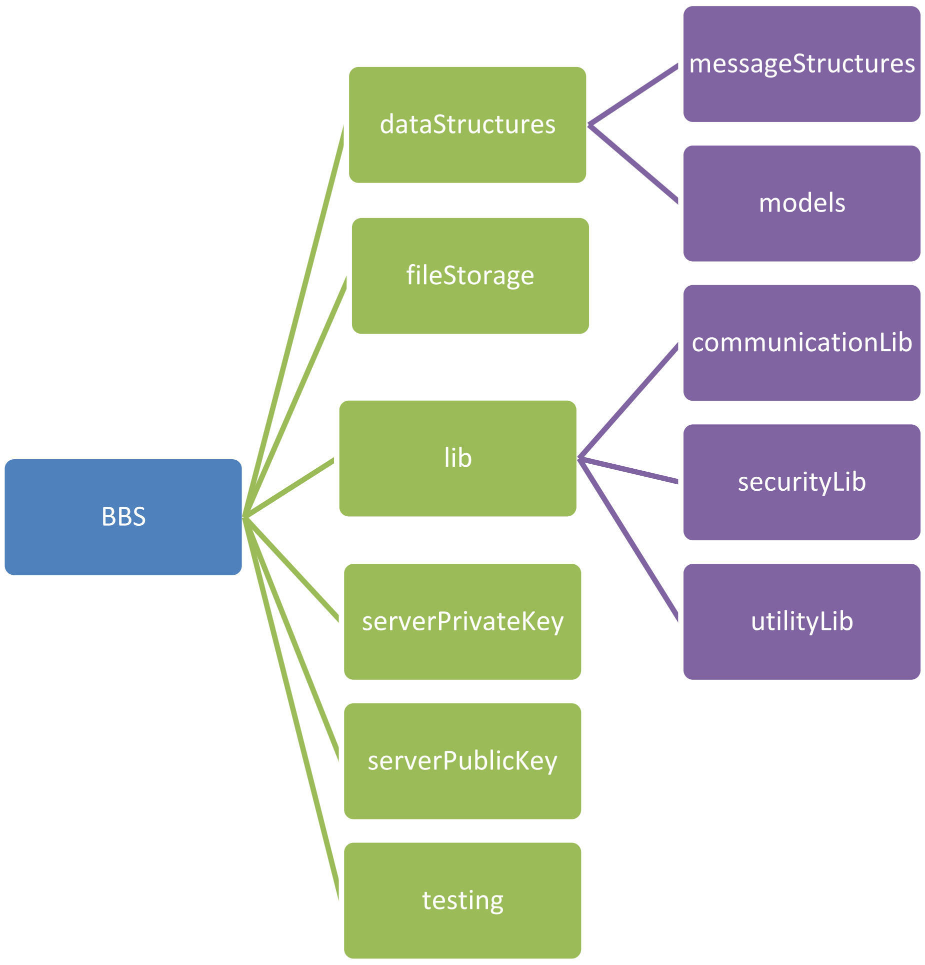


Figure 1 - The project directory diagram.

- `dataStructures`: contains all the class definitions of the used objects, in particular the message categories and the user definition.
- `lib`: contains many libraries:
  - `communicationLib`: contains utility functions for the client-server communication.
  - `securityLib`: for manage all the operations related to the security (*AES, RSA, HMAC, SHA, ...*) or random generations.
  - `utilityLib`: contains many different types of function, used to manage strings, conversion from a type to another and functions to store/load something in/to a file.
- `serverPrivateKey`: contains the RSA encrypted private key of the server, used during the key exchange phase.
- `serverPublicKey`: contains the RSA encrypted public key of the server with its certificate, used during the key exchange phase.
- `testing`: contains a testing script that test the core functionalities of the project.

2 Client

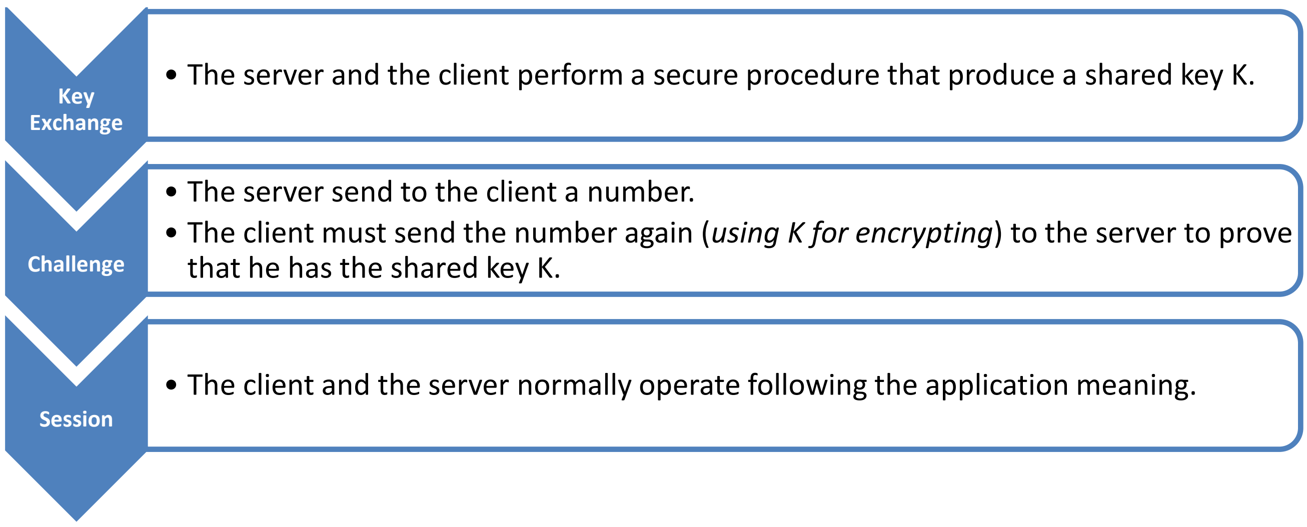
The client since it establishes a connection with the server, asks the user to specify an action between registration and login, after this phase, the client starts the authentication and the session key exchange procedures. Then the user can login or register and starts to use the application by reading, downloading and adding messages.

3 Server

The server exploits the multithreading advantages, when a new client connects to the server, this one creates a new thread to interact with that client, the thread execution terminates when its assigned client disconnects. The server stores all the information about users and all the messages in the BBS and replies to the clients with the information requested by users.

4 The protocol

The protocol is slightly different if the user wants to login or wants to register. In particular, the registration process has an additional part; for this reason, we will only illustrate the protocol in the case of a registration.



- The server and the client perform a secure procedure that produce a shared key K.

- The server send to the client a number.
- The client must send the number again (*using K for encrypting*) to the server to prove that he has the shared key K.

- The client and the server normally operate following the application meaning.

All those 3 phases must fulfill these requirements:

1. Confidentiality.
2. Integrity.
3. No-replay of the communications.
4. Non-malleability of the communications.
5. Grant the perfect-forward-secrecy.

4.1 Key Exchange

This phase must fulfill the 5° requirement. For this reason, we have chosen **RSAE** (Ephemeral RSA) to perform the key exchange.

The RSAE procedure is based on 3 messages (M1, M2, M3).

M1 C → S: “reg” , R

The client randomly generates a 64-bit unsigned integer called $\boldsymbol{R}$ and send it to the server in the clear.

M2 S → C: $T_{pk}, < R \mid T_{pk} >_{S_k}, Cert_S$

The server randomly generates a couple of RSA keys $\{T_{pk}, T_{sk}\}$ called “*temporary RSA keys*”. The server sends to the client:

- The temporary public key T_{pk}.
- A string that contains R concatenated with T_{pk} all digitally signed with the long-term private RSA key of the server S_k.
- The certificate of the long-term public key of the server P_k.

The client receives the message M2, extracts the public key P_k from the certificate and verify the digital signature (*the digital signature grants the* ***authenticity*** *of the message M2, and the fact that the sign contains R assure* ***freshness*** *to the message*).

Now the client generates a random secure string $\boldsymbol{K}$ that will be used as session key.

M3 C → S: $E_{T_{pk}}(K)$

After receiving this message, the server deletes $\{T_{pk}, T_{sk}\}$.

PFS is granted because: If the adversary finds K or T_{sk} only this session will be compromised, because other sessions (*between the same client and the same server*) will use different keys.

4.2 Registration

M4 C → S: $IV, E_K(registration\ details), HMAC$

4.3 Challenge

The challenge is a quite simple phase; it consists of 3 messages (M5, M6, M7).

The server generates a random 16-bit unsigned integer $\boldsymbol{C}$ and send it to the client via mail.

M5 S → Client Mail Server: C

The client reads from his/her mail the number and send it to the server.

M6 C → S: $IV, E_K(C), HMAC$

The server receives the message and send to the client the result of the registration process and of the challenge.

Keep in mind that in the login procedure, the only difference is the absence of M5 and M6. M7 always contains the result of the login procedure.

This message is important and will be cited after.

M7 S → C: $IV, E_K(result), HMAC$

4.4 The session

A session consists of an unknown number of messages, but a message in the session can be one between the following 4 types:

1. List
2. Get
3. Add
4. Logout

4.4.1 List

M8 C → S: $IV, E_K(command\ details), HMAC$

M9 S → C: $IV, E_K(wanted\ result), HMAC$

4.4.2 Get

M10 C → S: $IV, E_K(command\ details), HMAC$

M11 S → C: $IV, E_K(wanted\ result), HMAC$

4.4.3 Add

The add message is even simpler:

M12 C → S: $IV\ , E_K(add\ details), HMAC$

M13 S → C: $result$

Where result con be 1 or 0 and it is sent in the clear.

4.4.3 Logout

The logout message is the simplest.

M14 C → S: $IV\ , E_K(\text{logout}), HMAC$

After receiving it, the thread that was managing the connection simply dies after closing the socket.

5 Packets

Each type of used packet has a relative C++ class in the directory `dataStructures/messageStructures`.

Must be noticed that each class has two special methods:

- `concatenateFields`: concatenate the dimensions and the fields of the message in a string.
- `deconcatenateAndAssign`: from the concatenated string, reconstruct the object.

These two methods permit to the client and the server to exchange objects through strings. The string can be obviously encrypted before sending.

In particular we can use the following 3 types of messages.

5.1 Simple Message

This type of message is the simplest one and it is used to transmit structured string in the clear without any further check.

Messages that follows this schema are: M1
The message structure is like:

Content	ContentDim

Figure 2 - The simple message structure.

Where ContentDim is the number of characters of the content string. The ContentDim is necessary because very often the content of a message is composed by many fields concatenated, so it is necessary to have the capability to distinguish the entire message from the fields.

5.2 Content Message

This type of message is the one used in the session and in every message that requires authentication and any other check.

Messages that follows this schema are every message after M4.
The message structure is like:

IVdim	Cdim	HMACdim	IV	C	HMAC

Figure 3 - The content message structure.

Where the three dimensions are the number of characters of the relative field.

The HMAC is calculated like: $HMAC = HMAC_{IV}(C)$, in other words follows the "**Encrypt then MAC"** paradigm. For produce the HMAC we use SHA-256 as hash function.

The encrypted text is calculated like: $C = E_K(message)$ using AES-256 as symmetric cipher.

The IV is a 16 characters random string generated before sending the message, so for each message a new IV is generated.

5.2.3 The cryptogram

$\boldsymbol{C}$ deserves some other explanation: In the case of the session messages (*list, get, add and logout*) the format of the plaintext is the following:

REQUEST_TYPE	REQUEST_DETAIL	Counter

Figure 4 - The session message structure.

- **REQUEST_TYPE**: identifies the type of request (*list, get, add and logout*).
- **REQUEST_DETAIL**: specify the details of the request (*in the case of the list, how many message the client want to se*e).
- **Counter**: is a 64-bit number (*converted in string*) that is used to identify a message of a user; this number is kept by the server and sent to the client after the login procedure (*in the result message M7 sent by the server after the login procedure*).

The counter is used to **grant non-replicability of a session**, if the counter value sent by the client is not the counter expected by the server the session will be closed by the latter.

The counter is sent to the client only after a successful login procedure (*at the registration phase, it is zero*).

5.3 RSAE Message

This message is used only one time during the RSAE procedure (M2).

TPKdim	DSdim	CERTdim	T_{pk}	Digital Signature	Certificate

Figure 5 - The RSAE message structure.

Each one of the content fields are sent as strings (*then converted to* EVP_PKEY*, or* $X509$ *by the server or by the client after receiving them*).

The RSAE Message class permits to the client to use methods for:

- Verify the digital signature.
- Extract the public key from the certificate
- Verify the validity of the certificate.