Security Project Report

Francisco Cunha, nº 76759
João Maia, nº 76364

December 31, 2017

Abstract

This document focus on the description and implementation of the security features of a project for the discipline of Security from the course of Computer and Telematics Engineer from University of Aveiro, which consists in a secure messaging repository system.

In this document, the reader will find all the studies performed, the alternatives considered, the decisions taken, the implemented functionalities, the problems and deficiencies of the project, the project's structure and the goals achieved and not achieved.

This report as the project itself is to be delivered and evaluated on the established deadline, i.e 31 of December of 2017.

Chapter 1

Introduction

The project described in this report was proposed by the teachers of the Security discipline, from the course Telematics and Computer Engineer from University of Aveiro, and has a number of prerequisites which will be mentioned along the report.

The project's main goal is the implementation of a secure messaging repository system which would be used to exchange messages between two users without the message's contents being eavesdropped or intercepted.

To achieve the main goal, a number of supporting objectives were created to guide the project's implementation. The achieving of these goals will be described in the next chapter. These goals are:

- Implementation of a complete, non-secure client application to communicate with the server;
- Coding of security features on messages;
- Coding of authentication features;

Chapter 2

Achieving the Goals

2.1 Implementation of a complete, non-secure client application to communicate with the server

A basic implementation of a server-client communication in Python version 3 was used to achieve this objective, a client sends messages to the server which responds and performs the operations asked by the client. The messages destined to users are stored in a folder named message box which only the server accesses.

The user interacts with the application using a command based interface, i.e., the user inserts a command into the terminal/command line and the client will run the function associated with the command. This implementation was built to run in Linux, Mac and Windows operating systems.

The messages, which are sent to a socket in which the user is connected, are in bytes enconded in utf-8 format, converted from a Javascript Object Notation (JSON) format. Each message has certain parameters which are stored in a JSON dictionary, some have to be converted to a string format to be stored in the JSON dictionary.

The messages are divided in the following types: create, list, new, all, send, recv, receipt, status (these type of messages were a prerequisite for the project), recPK, login and update (these type of messages were later added to the project to ease the communication between the client and the server). Each message type was created with a goal:

- **CREATE:** create a new account. The parameters for this message are a Universal User Identification (uuid), calculated using a digest function with the user's public key, extracted from the Citizen's Card (CC), a password, to later login in the account, the user's CC public key and its certificates;
- **LIST:** list the users with a message box. The parameter for this message is a user id, and in case the user wants a list of all users with the message box, the user id should be equal to zero;
- **NEW:** list all new messages in a user's message box. The parameter for this message is the user id;
- **ALL:** list all messages in a user's message box. The parameter for this message is the user id;
- **SEND:** send a message to a user's message box. The parameters for this message are the sender user id, the receiver user id, the ciphered message written by the sender, a ciphered copy only deciphered by the sender, a ciphered symmetric key, a *nonce* for Counter (CTR) mode and the sender's signature;
- **RECV:** receive a message from a user's message box. The parameters for this message are the receiver's user id and the message's id;
- **RECEIPT:** automatically sent after receiving and validating a message from a message box. The parameters for this message are the received message's sender's user id, the message id and a signature over the deciphered message;

- **STATUS:** checks the reception status of a sent message, in other words, checks if there's a receipt and it it's valid. The parameters for this message are the message's sender's id and the message's id;
- **LOGIN:** login into an existing account. The parameters for this message are the same as the create message;
- **UPDATE:** updates the server's databases with the new public keys and session keys. The parameters for this message are the user's user id, client's public key and the client's signature;

When the server receives a message, it first checks the type of message, indicated in the message, and then runs the appropriate function to handle the message.

2.2 Coding of security features on messages

The solution found to protect the messages sent between the client and the server was ciphering the messages with a session key, using a symmetric algorithm named Advanced Encryption Standard (AES), which only these two could have access. Each time a client connects to a server, the process to establish a session key is initiated, thus a session key is different in every session and for every client, making difficult to decipher the messages' contents. However, this doesn't protect against Man-in-the-Middle (MitM) attacks and if the session key is discovered, then the messages' contents are exposed and the communication is compromised. So to solve this problem, the solution found was the signing of the messages exchanged between the server and the client using their respective asymmetric private keys, notifying the receiver when a signature is invalid meaning there's a possibility of an MitM attack is occurring.

Although the messages are ciphered with the session key, it's not enough to protect the messages' contents, since in case the session key is discovered and the messages deciphered, the messages can be eavesdropped and tempered with. To prevent this situation, an hybrid cryptographic algorithm is used, i.e., two algorithms are used to cipher the messages, a symmetric algorithm (the chosen algorithm was the 256-bit AES in CTR mode) and a asymmetric algorithm (the chosen algorithm was the 2048-bit Rivest–Shamir–Adleman (RSA)). It's important to inform that the messages ciphered with this hybrid algorithm are not the JSON dictionaries mentioned before, but the messages written by the user. The reason behind this choice was the program's efficiency and performance, seeing that ciphering all the contents would decrease the performance, the most important parameters were selected to be ciphered by the mentioned hybrid algorithm.

2.2.1 Ciphering using the Session Key

All the messages exchanged between the server an the client are ciphered, using the provided functions from the Python library cryptography version 2.1.4, with the created session key (which creation is explained in the section 3.1.2) using the symmetric algorithm 256-bit AES in CTR mode, for the reason that it will cipher all the messages and it will be used very frequently, so it has to be fast in ciphering the messages and have a low memory requirements, which the AES fulfills. Other advantages of the algorithm are taking a long time to decrypt the message using brute force attacks and not having any known practical attacks that allow to decipher the message without knowing the key. The algorithm's disadvantage is its vulnerability to side-channel attacks.

The CTR mode was chosen considering it's cryptographically strong, transforms a block cipher in a stream cipher, easing and speeding the ciphering process, and eliminates the error propagation. The disadvantage of this mode is the use of *nonces*, a number used once, so it must be sent to the server, along with the ciphered message, to decipher the message and a new *nonce* must be generated every time a message is sent, otherwise the message would be deciphered more easily.

When the server receives a message, it separates the byte string received in two, the ciphered message and the *nonce* encoded in bytes, then deciphers the message and checks its type.

2.2.2 Ciphering with the Hybrid algorithm

First, when the client is initiated, a symmetric key and a pair of asymmetric keys are generated, then the client's public key and the server's public key are exchanged (this process is described in the section 3.1.1) and finally the session key is generated.

To protect a message destined to another user, the message written by the sender is ciphered with the symmetric key. However, the receiver doesn't know the symmetric key so it can't decipher the message, thus it must be sent as well with the ciphered message. In a case where the session key is discovered and the message compromised, this solution is useless, considering the attacker has the means to decipher the encrypted message. To prevent this from happening, the symmetric key is ciphered with the receiver's public key, so that only the receiver can decipher it with its private key.

This ensures integrity, but doesn't ensure source authentication. So the written message is signed with the sender's private key, ensuring it was the client to send the message.

Initially, the solution created for this problem was the ciphering of the ciphered symmetric key with the client's private key, resulting in a doubled ciphered symmetric key, this idea was retrieved from a book [1]. Even if the first layer was deciphered with the public key, seeing that any user has access to the public key, the written message would remain secure, because it would still need to be deciphered with the receiver's private key. However, this is not efficient compared to signing the message and the available Python libraries don't provide the possibility to implement this feature, so the solution was redesigned to the presented one, although both are similar in terms of source authentication. The CC is not used to sign the message, the reason for this choice is explained in the next section.

The ciphering is implemented using the provided function from the Python library cryptography version 2.1.4.

2.3 Coding of authentication features

One of the project's prerequisites was the use of the CC capabilities to authenticate the user and sign messages. In this implementation, the CC is used to create and to login to an account, so the user does not require the CC to send messages. The reason for this choice was a practical one from the user's point of view, in other words, the user only needs to use the CC to prove their identity, and after its use, the user can save the card. The disadvantages of this approach are stated in the section 4.

The first messages, which establish the session, are signed by their sender, the client/server, using their respective private keys. The following messages are authenticated through the session key, since only the respective client and the server have access to it. However the more sensible messages, the create, the login and the send messages, are signed by its sender and are verified by the receiver.

2.3.1 Creation and Login in an Account

Create Message

The create message is signed by the CC's authentication private key using the available functions from the Python libraries PyKCS11 version 1.4.4 and pyOpenSSL version 17.5.0. The data that is signed is the concatenation within the user's id (uuid) with the user's password, and this new string is then signed with the key referred above. In this message it's also sent the authentication public key certificate, in order to construct the respective certificate and save it in the server. After the certificate is saved, the server will try to decipher the signed data with the public key extracted from the certificate, and if it can decipher, the signature is valid.

Login Message

The login message is signed by the CC's private key using the available functions from the Python libraries PyKCS11 version 1.4.4 and pyOpenSSL version 17.5.0. The login authentication process is similar to the one previously described, the signed data is the same, however the certificate attributes are not sent, because the server already has the certificates for that user. When the server checks the signed data, it will try to decipher the data with the public key extracted from the certificate, and if it's successful, the signature is valid.

2.3.2 Send a Message to a User

The send message is signed by the sender's private key using the available functions from the Python library cryptography version 2.1.4. This solution ensures it was the client to send the message, not the user, and also ensures if there is an MitM attack occurring or the message was tempered, the receiver is notified.

The signature is generated by a cryptographic hash function, in this case a Secure Hash Algorithm 256 bits (SHA-256) function. The reason for choosing this hash function is for being the only 256-bit hash function compatible with the CC. The advantages of this hash function is being totally collision-resistant and stronger than SHA-1 family. There are no known vulnerabilities for the SHA-256 algorithm, the attacks performed, preimage, length extension and collision attacks, have only break a few *rounds* from the algorithm, but not all of them.

Chapter 3

Project's Structure

In this chapter, a description of how the client and the server communicate can be found. The description will consist in these steps: establishing the session, creating and logging into an account, sending and receiving messages, status of a message, saving messages and updating databases.

3.1 Establishing the Session

Establishing the session starts immediately after the client connects with the server. It consists in a two step process: exchange public keys and generate a session key.

3.1.1 Public Keys Exchange

Both the client and the server when are initiated, generate a pair of asymmetric RSA keys.

After a client connects with the server, this one sends its public key in a PEM format, signed with its private key and awaits the client response. The client receives the server's public key and verifies the signature, if it's invalid the client program terminates, otherwise, it responds with its public key in PEM format and signed with its private key. The server receives the public key and verifies the signature, disconnecting the client in case it is invalid.

After receiving each others public keys, both the client and the server store them. The server stores the public key in a temporary database, described in section 3.6, and the client stores in a variable.

3.1.2 Session Key establishment

The session key is generated right after the exchange of the public keys and the algorithm used to create the session key is the Elliptic-Curve Diffie Hellman Ephemeral (ECDHE) algorithm.

The private and the public components of the ECDHE algorithm are calculated, using the functions provided by the Python library cryptography version 2.1.4, in both the client and the server, however it's the server that first sends its ECDHE's public component, which is signed by the server's RSA private key to the client and awaits the client's response. The client verifies the signature and if the signature is invalid, the client is terminated, otherwise the client sends its ECDHE's public component signed with its RSA private key. Then the server verifies the signature and if the signature is invalid, the connection is terminated, otherwise the session key is calculated using its ECDHE private component and the client's ECDHE public component. The same calculation occurs in the client, although the client uses its ECDHE private component and the server's ECDHE public component, the session keys will be the same and can only be accessed by the client and the server.

The server stores all the calculated session keys in a database described in section 3.6 and the client in a variable.

3.2 Creating and Logging into an Account

3.2.1 Creating an Account

When an user intends to create an account, the CC must be connected to the terminal, so its certificates can be extracted and sign messages.

The user's uuid will be created using a digest function with the CC's public key and it will be asked to the user to create a password, which is hashed with the BLAKE2s algorithm, choose for being immune to length-extension attacks and the SHA-2 family being a bad choice to hash passwords. Next, the CC's certificates are extracted, the message is signed and sent to the server which will verify the validation of the message. If the message isn't valid, the process is canceled, or else, a message box, a receipt box and a "saved_messages" directories are created, in which the message box will contain the received messages, the respective symmetric key, *nonce* and signature stored in a file, and the user's description, i.e., the user's uuid and password, the receipt box will store all the received receipts and ciphered copies of the sent messages, and the "saved_messages" will store the saved messages.

3.2.2 Logging in a Account

As with the create message case, the CC must be connected to the terminal if the user intends to login into their account, seeing that it must sign the message, proving it was the user to send it.

When a user logins into a account, the databases are updated with the new session and public keys.

3.3 Sending and Receiving Messages

3.3.1 Sending Messages

To send a message, the user must first provide the receiver's user id then it can write the message. After finishing writing the message, the client will run the process described in section 2.2 and send the message to the server, which will store the message, with its symmetric key, *nonce* and signature, in the corresponding message box and the copy in the respective receipt box.

Each message when sent is given an id to identify it. The file storing the message is given the same name as the message id, where the file storing the symmetric key, the *nonce* and the signature is given the name of the message id followed with a "scs" acronym, which stands for "Symmetric key, Counter and Signature".

The copy also receives an id and its file has the same name as the copy's id.

3.3.2 Receiving Messages

The user isn't notified when they receive a message, instead they have to send a NEW message so that the server checks for new messages in the message box. If new messages exists, then the server returns a list of all new messages, if not, the list returned is empty.

To receive a message the user must first provide the message's id. An important note is when a message is read, its added to the file's name an underscore at the beginning, for example, the message "2_1" was sent to an user, after this user reads the message, i.e., receives the message, the message's name becomes "_2_1", so in case a user wants to read again the message, during the same session it was sent and read, it has to provide the new message id, "_2_1".

After the server receives the recv message, it searches for the message and its "scs" file in the receiver's message box and sends to the client.

The client then verifies the signature, if it's invalid it notifies the user, if not, it deciphers the message using the symmetric key, after deciphering the symmetric key with its private key, and finally presents the message in plaintext. When the message is shown, the client asks the user if the message is to be saved, the reason for this implementation is explained in section 3.5.

3.4 Status of a Message

3.4.1 Receipts

After the user receives a message, the client automatically sends a receipt to the server, so the user doesn't need to insert any command.

The receipt consists in a signature over the deciphered message, meaning the receiver collected and validated the message, and the receipt's sender is not notified that the receipt was sent.

3.4.2 Status

As said in the previous subsection, a receipt is sent automatically after the recv message, signifying the receiver has read the message. The goal of the status, which is also described in section 2.1, is to check for receipts and if there are any receipts for a message, as one message can have many receipts, the server will send them to the client.

The receipts suffer from the same problem as the messages. Since in every session a new pair of asymmetric keys is generated, the verification of old receipts received in previous sessions will be always invalid. Because the receipt is ciphered the same way as copies, the copies suffer the same problem.

3.5 Saving Messages

After the message is deciphered and shown to the user, the client asks the user if the message is to be saved, considering that in every new session a new pair of asymmetric keys is generated, there is no possibility to decipher the received messages after the end of a session, in other words, messages from previous sessions can't be deciphered. The reason for this implementation is the increase in security, so if an attacker possesses a pair of asymmetric keys, few messages are deciphered and the attacker won't be able to decipher anymore messages if the client is restarted.

3.6 Updating Databases

The server manages many clients and establishes many sessions with them, thus it requires a place to store the keys of each client and session. The server possesses three databases: a public key database, a session key database and a arrival database.

- **Arrival Database:** The arrival database is a simple dictionary that stores the number of arrival of each client. Its keys are an object *socket* and it's updated when a client connects or disconnects with the server.
- **Public Key Database:** The public key database is also a simple dictionary that stores the client's public keys and its indexes are the number of arrival of each client. When a user logins into an account, the database updates the public key of the corresponding client. This database is the only one that it's stored in a file, seeing that it must be access by more than one script, it's more efficient to store in a document which the scripts can have access to, although it costs more CPU resources, since the file is updated every time a client connects with the server.
- **Session Key Database:** The session key database is also a simple dictionary that stores the client's session keys and its indexes are the number of arrival of each client. When a user logins into an account, the database updates the session key of the corresponding client.

Chapter 4

Problems

The project suffers some deficiencies, which are next numerated:

1. **No permissions to access the files:** each user has a message box and when using the application, a user can only access their respective message box. However, if the server is compromised, the attacker can access these message boxes and other important files, such as the file storing the public keys, and temper with them. The solution for this problems is to give permissions to read and write, only to one user from the terminal in which the server application is running.

2. **No use of *chroot* mechanism:** in case the server is compromised, the attacker shouldn't leave the repository system storing the server application, denying the access to the remaining repositories of the terminal. This is achieved using a *chroot* mechanism or jail, which reduces the visibility of a file system.

3. **Invalid message signatures and receipts:** this problem was addressed in sections 3.4.2 and 3.5. When a session is closed or terminated, all the received messages and receipts will be invalid, since a new pair of asymmetric keys will be generated. The solution found for this problem is asking the user if they wish to save the message in a directory, and the message won't be ciphered if saved. This proves the user read the message and can later have access to it, but the sender won't be able to check whether the receiver read it or not.

4. **Saved Messages are in Plaintext:** the messages stored in the user's reserved directory are in plaintext, in other words, they aren't ciphered. This presents a problem since other users may read and temper the file. It's the user's responsibility to change the file's permissions so the file can only be read and written by the user.

5. **Vulnerable to Side-Channel Attacks:** the messages sent between the server and the client are vulnerable to side-channel attacks if a session lasts to long. A solution to counter such attacks is to increase noise and decrease leakage.

6. **Validattion of the certificate chain:** To add a user on the create's message, after the certificate is sent and the signature is validated, the validation of the certicate chain should be correcty verified, however that does not happen. The validation of the certificate chain in our implementation is set to be correct even if it is not.

Requirements

1. **pykcs11 1.4.4**
2. **getpass**
3. **pyOpenSSL 17.5.0**
4. **pyjks 17.1.1**
5. **pycrypto 2.6.1**
6. **getch 1.0** - for Windows users

Acronyms

AES Advanced Encryption Standard

CC Citizen's Card

CTR Counter

ECDHE Elliptic-Curve Diffie Hellman Ephemeral

JSON Javascript Object Notation

MitM Man-in-the-Middle

RSA Rivest–Shamir–Adleman

SHA-256 Secure Hash Algorithm 256 bits

uuid Universal User Identification

Bibliography

[1] Security Computing, Fourth Edition, by Charles P. Pfleeger, Shari Lawrence Pfleeger (**Note:** the taken suggestion is located in page 116, in the Key Exchange subsection of The Uses of Encryption section)

[2] PyOpenSSL documentation (https://pyopenssl.readthedocs.io/em/latest/api/)

[3] PyKSC11 documentation (https://pkcs11wrap.sourceforge.io/api/)

[4] Cryptography Library documentation (https://cryptography.io/en/latest/)

[5] Python 3 documentation (https://docs.python.org/3/)

[6] StackOverflow (https://stackoverflow.com/)

[7] Wikipedia (https://wikipedia.org/)