Universidade de Aveiro

Secure Messaging Repository System

Departamento de Electrónica, Telecomunicações e Informática

Security

Diogo Ferreira, Luís Leira
76504, 76514
diogodanielsoaresferreira@ua.pt, luisleira@ua.pt

December 31, 2017

Contents

Chapter 1

Introduction

On this report we will present our solution for a secure messaging repository system. The project was made in the Security course, in Departamento de Electrónica, Telecomunicações e Informática in Universidade de Aveiro. The main goal of the project was to develop a system to exchange messages asynchronously between clients and a central repository. The three main security features of the project are:

- Message confidentiality, integrity and authentication;
- Message delivery confirmation;
- Identity preservation.

On the second chapter of this report, we will explain the security functionalities that were made to implement the security policies defined, and why we choose them. On the third chapter we will go into further detail about some relevant parts of the implementation, such as the caches created, and also the known problems and limitations.

On the last chapter, we will conclude by summing up the most important features and implemented functionalities of the project.

Chapter 2

Security functionalities

2.1 Setup of a session key between a client and the server prior to exchange any command/ response

2.1.1 Certificates exchange

Initially, the client will start the communication by identifying himself. For that, he will send the Citizen Authentication Certificate, present in the citizen card, along with the certificate chain. The server will validate the certificate chain and it will send to the client its certificate chain, and also a challenge (a 32-byte random integer) to assure that the client signs it with the private key, proving that he is the owner of the corresponding private key.

The client will validate the certificate chain and it will sign the challenge sent by the server (the signed challenge will be the response) with the Authentication Private Key. It will also generate a new challenge and send it, along with the response, to the server.

The server will receive the packet and check if the response matches with the challenge sent previously using the client's Authentication Public Key. If it does, the server calculates the response to the challenge received, by signing the challenge with his private key and sends the response to the client. The client will check if the response matches the challenge using the server's public key. If it does, it is ready to initiate the session key establishment.

2.1.2 Establishing a session key

After the certificates and public keys exchange, a session key can be established. For that, we used *Elliptic-Curve Diffie-Hellman* (ECDH). We chose it instead of regular *Diffie-Hellman* mainly because it is faster generating the keys and provides better secrecy for the same number of bits. The client will generate a key pair, it will store temporarily the private key and it will send the public key to the server. The public key will be signed with the client's Authentication Private Key to avoid man-in-the-middle attacks. The server will also generate a key pair, and it will calculate the session key with the public key received (after validating the signature). Then, it will send its public key to the client signed with its own private key. Finally, the client will receive the server public key, validate its signature and calculate the session key. At this time, client and server will have both the same session key.

The signatures will be made using the *SHA-256* algorithm. The session key will be derived after every message exchanged to minimize the probability of brute force attacks and enable forward secrecy. The key derivation function used will be *PBKDF2HMAC* with the hash algorithm being *SHA-256* with 100 000 iterations. The derived key will have 32 bytes and the salt used will be generated randomly by the client, and sent to the server on the same message that the client sends the response to the server challenge, ciphered with the server public key. In that way, only the client and the server can know the salt used for deriving the session key.

2.2 Authentication (with the session key) and integrity control of all messages exchanged between client and server

After the session key is established, every message between the client and the server has to be authenticated and allow integrity control. For that, we decided to cipher every message after the session key is established with *AES-256* algorithm in *GCM* (Galois Counter Mode). The key used will be the session key, and the IV will be sent along with the message.

It is not asked that the client-server communications need to be confidential. However, improves the robustness of the solution by making it harder for a man-in-the-middle attack or a brute force attack (for example, an attacker can't see the response to "list" packets, now knowing which users are registered in the server; an attacker also can't see the id of the messages

received and sent by other users). Other alternatives taken into account for integrity control without confidentiality were signing all messages or include message authentication codes (*HMAC*). Signing all messages would have a higher time and processor cost, but it would also add the non-repudiation property to all messages; message authentication codes would be a faster solution.

When the server receives a packet with an *ID*, it will check if the *ID* received in the packet matches the *ID* of the client bounded to the session. The client bounded to the session is calculated by doing a digest of the client's Authentication Public Key (exchanged in the beginning of the communication) and checking if there is any user with that digest as *UUID*. If that does not happen, the user has not been registered, and the server will answer with an error. If the user exists, but the packet id does not match the user *ID*, the server will also answer the client with an error.

2.3 Add to each server reply a genuineness warrant

Each packet sent by the client, after the public key of the server is received, will contain a nonce (number used only once) that will be a 32-byte random integer ciphered with the public key of the server. That number will be deciphered only by the server on the response to the message. In that way, we can be sure that the sender of the message has the correspondent private key. It also prevents replay attacks, since each message has a random generated *ID*.

2.4 Register relevant security-related data in a user creation process

In the user creation process, the user needs to send to the server the public key to cipher messages, as well as a signature of the public key made with the authentication private key. If the signature is valid, the server can be sure that the public key was sent by the user. The *UUID* is internally calculated by the server, as a digest of the Authentication Public Key, using the algorithm *SHA-256*.

2.5 Involve the Citizen Card in the user creation

When the client establishes a secure connection with the server, if it is not registered in the server, it will send a "Create" message with his Authentication Public Key and a signature of it made with his Authentication Private Key. This public key will be used to generate a digest using the algorithm *SHA-256* and it will be the client's *UUID*.

2.6 Signature of the messages delivered to other users and validation of those signatures

When a client sends a message to other users, it needs to sign it to assure that he sent it. The signature is made using the Authentication Public Key, with the algorithm *SHA-256*. The signature made will also concatenate the timestamp to the message ciphered (message|timestamp), so the receiver can be sure that the received timestamp is the timestamp sent by the sender client. It does not prevent that the client sends a wrong timestamp, but it assures that the timestamp sent from the client was not changed on its way.

When a client requests a message to be read, the server will send the ciphered message, along with the public key of the sender and a signature of that public key, made by the server. The client will validate the signature made to the public key. Then, it will verify the signature of the message with the public key received.

2.7 Encrypt messages saved in the receipt box

When a client sends a message to another user, it will need to send the message to his receipt box, allowing only him to decipher the message. To do that, the field "copy" contains the message ciphered with *AES-256* on *GCM* (Galois Counter Mode) to cipher and authenticate the message (*GCM* ensures integrity control). The key and the IV will be randomly generated by the client and will be ciphered with the client public key, and concatenated to the "copy" sent to the server. The server will receive the copy and save it in the receipt box of the user.

2.8 Encrypt messages delivered to other users

The messages sent to other users are ciphered using end-to-end encryption. The client sending the message will check if it has the public key for ciphering messages of the receiver saved on his cache. If it does not have it, it will send a packet *getPublicKey* to the server, with the *ID* of the receiver client. The server will reply with the public key of the receiver client, signed by him. The client will validate the signature and store the public key of the client on cache.

The messages will be ciphered using a hybrid cipher, with algorithm *AES-256* in *CTR* mode, because it allows random access and can be parallelized. It is not needed to use an algorithm with integrity control, because the signature of the messages already guarantees integrity control, authentication and non-repudiation. The key and the IV will be a 32-byte and 16-byte integer, respectively, generated randomly. They will both be ciphered with the public key of the receiver and sent concatenated with the ciphered message.

2.9 Send a secure receipt after reading a message

When the client sends a request for reading a message, the server will also send a nonce signed with his private key. That nonce will be verified by the client to check if the message was sent by the server. After reading the message, the client will send a receipt for the message. The receipt is a signature with the authentication private key to the message, appended with the timestamp sent (message|timestamp). In that way, the signature assures that the message was read by a client and that the timestamp received was the timestamp sent from that client.

2.10 Check receipts of sent messages

When a client sends a "Status" packet, it will receive a list of receipts of the "copy" field of his sent messages. To get the message in plaintext, the client will have to decipher the "copy", by deciphering the key and the IV with his private key (not the authentication private key, but the private key for ciphering and deciphering messages). Then, the key and the IV deciphered will be used to decipher the message received. With the message in plaintext, the client has to verify if the signature of the message is valid. Because the signature was made with the receiver authentication public key,

that is sent by the server, the client has to validate its signature with the public key of the server. If the signature is valid, the client loads the public key received and validates the receipt of the message and the timestamp with the public key. If the signature is valid, the client can be sure that the receipt was made by that client.

2.11 Proper checking of public key certificates

In the beginning of the communication between client and server, both exchange the certificate and the certificate path. Each entity will validate the certificate path received.

For that, each entity will have a file with all the trusted certificates. When it receives the certificate chain, the entity will create a path from the certificate of the other entity to the root certificate, that is self-signed. Then, it will try to validate the certificate. To do that, it will first check if the key usage is correct, which means for the entity certificate to check if the certificate can sign digital documents and to check if the certificate can be used for key agreement. For the other certificates, it will check if the key can be used to sign certificates and *CRL*s.

For all certificates, it is needed to check if they have expired or if they are revoked. To check if they are revoked, we decided to make a cache for *CLR* files. For each certificate, we will ask the cache for all *CRL*s in that certificate, and the cache will return the valid *CRL*s. Then, it is needed to check if the *CRL* signature is valid with the public key present on the certificate. If it is valid, the entity can finally check if the certificate serial number is present in the *CRL*. If it is, the certificate has been revoked.

The entity also has to check if the public key of the *CA* validates the signature made to the certificate.

The certificate chain is valid if all the points are true:

- The entity certificate can be used for digital signature and key agreement;
- All the other certificates can be used for certificate signing and *CRL* signing;
- All certificates have not expired;
- All certificates have not been revoked;
- All the certificates are validated by a CA, and the root certificate by itself;

- The root certificate is stored as a trusted certificate.

If one or more of the points are not true, the certificate chain is not valid.

2.12 Prevent a user from reading messages from other than their own message box

When a client requests for reading a message, the server will check if the client *ID* matches the message destination. If it is valid, the server will send the message to the client along with the one-time ciphered symmetric key. The message sent by the server is ciphered using a hybrid cipher. The symmetric key used to cipher the message content will be ciphered with the Authentication Public Key of the receiver. This guarantees that only the receiver will be able to decipher the message because only he has the Authentication Private Key that deciphers the received symmetric key, which deciphers the message content.

2.13 Prevent a user from sending a receipt for a message that they had not read

Since the receipt of a client is a digital signature of the message, the client will need to read the entire message to digitally sign it correctly. It is also not possible to send a receipt for a message that has not been requested, because each message sent by the server will have a nonce (number used only once) that will be a 32-byte random integer signed by the server. That number will be validated and sent by the client on the "receipt" packet to assure that the receipt corresponds to a specific packet, and also to prevent replay attacks.

2.14 Authentication of users in the client software

The client entity will have a folder *users* which contains a folder for each registered user on the client. Each user is identified by the *username*, the *UUID* (a digest calculated with the Authentication Public Key from the citizen card), a digest of the password and the *ID* returned from the user. Those elements will be stored inside a file called *description* on the folder of the user. Each user will have a *keystore* with the public and private keys to

cipher messages. The private key is ciphered, with the key being the *password* of the user.

Each time a user runs the application, it has a menu where it can choose to register himself as a new user or to login with a previous registered user. If a user chooses to register himself, it will be asked to enter the *username* and the *password*. The application will check if the *username* doesn't exist in the users' file. If it doesn't exist, it will get the Authentication Public Key, calculate the *UUID* and check if the *UUID* already exists on the users' saved. If it does not exist, it will generate a pair of keys for that user to cipher and decipher messages, and it will derive the password.

The derivation of the password is done using *PBKDF2HMAC* with the hash algorithm being *SHA-256* with 100 000 iterations. The derived key will have 32 bytes and the salt used will be randomly generated. The number of iterations increases the time needed to calculate the derived *password*, which can reduce the probability of brute-force attacks. Before saving the user description and files on memory, the client will try to register the user on the server, with the "create" packet. If the register fails, the user will not be registered on memory. Otherwise, the client will save the derived *password* concatenated with the IV, the public key, the *UUID* and the private key, ciphered with the password entered by the user.

If a client already has an account and wishes to login, it will be asked to insert the *username* and the *password*. If the derived of the *password* matches with the derived password saved on the folder of that user, and if the *UUID* of the actual smart card matches the *UUID* saved, the client is logged in.

2.15 Authentication of server

The server will have a unique *password* for start the communications. That password will be stored using the same mechanisms as the users' *passwords*, with a derivation of the inserted *password*. That *password* will assure that only the ones with access to the *password* will manage to start the server. Besides that, the *password* will cipher the private key of the server.

Chapter 3

Implementation details

3.1 How to run the server and the client

The server already has a *password*: labcom. After the first time, the user can change the *password* of the server, by deleting the folder "description". If the user deletes the folder, the server will print the instructions about the new pair of public keys and certificate that need to be created.

Initially, the client starts without any registered user. To register a user, start a client and on the first menu select the option "2 - Register a user". It will be asked to insert a *username* and a *password*. If the *username* does not exist in the client, the *username* and *password* are saved and it is generated a new key pair for the user. During this phase, the user needs to have the citizen card connected to the client software because it will be needed to calculate the *UUID* of the client, that is a digest of the authentication public key. After the registry of the user, the citizen card is bounded to that particular user, which means that the user will not be able to login without that citizen card, or other user will not be able to register itself in the system with other users' citizen card.

After the registry of the user is done, the client will try to register that user on the server. If it is successful, it will be presented a menu to the user with six options:

- List users' messages boxes - List all the users registered in the server, with its *ID*, authentication public key and public key for encrypting messages;
- List all new messages received - Lists the *ID* of all new messages received (messsages that were not seen);

- List all messages received/sent - Lists the *ID* of all messages received and sent;
- Send message to a user - Sends a message to a user message box;
- Receive a message - Receives a message sent to himself;
- Check the reception status of a sent message - Checks if a message that was sent to other user was already seen.

At the end of every command, the menu will be printed again to the user. Due to the asynchronous nature of the messages received, it is likely that there are logs after the printed menu in the command line.

3.2 Cache for client public keys

The clients have a circular cache for public keys, with the maximum size being 32. The cache will have the public keys of the clients for cipher messages sent to them. All the public keys are stored only in the cache itself, and are not save permanently in memory. The format of the cache is a dictionary, with tuples (*ID* of the client, public key).

3.3 Cache for CRLs

The cache for *CRL*s is used to save the most used *CRL*s and validate the expiring timestamps of each *CRL*. The only method accessed by outside functions will be *getCRLS* that will return all the *CRL*s valid for a certificate. For that, first it will parse the *URL* of the *CRL* present in the certificates. Then, it will check if those *CRL*s exist in memory. If they do, and if the date for next update is after the current timestamp, it will return the *CRL*s. Otherwise, it will download the new *CRL*s, save them in memory and return them.

It exists a global *CRL* description file, that is a dictionary with the *URL*s of the *CRL*s, and the timestamp when they are no longer valid. Every time a *CRL* is updated with a new one, his *timestamp* is also updated on the description.

The cache is not responsible for validating the signature of the *CRL*s with the *CA* certificate. That is done by outside methods.

3.4 Trusted certificates and keys storage

The server and the client will both have a folder called "trusted_certs". That folder will contain a *keystore* with all the certificates needed to validate the authentication certificate and its certificate chain. The *makefile* present in the folder was provided in the practical classes, downloads the certificates needed for the citizen card, and stores them in a Java *keystore*. The other certificate present in the trusted certificates in the certificate of the *CA* used to sign the certificate of the server.

On the client, each user will have a key pair, saved on the folder of the user, on the folder *keystore*. The private key will be ciphered with the client password. On the server-side, the server will also have a *keystore* with the private and public keys, and the private key will be ciphered with the server *password*.

3.5 Requirements

The following packages were installed using *pip3*, with the respective versions:

- python==3.5.2
- pyOpenSSL==17.3.0
- criptography==2.1.1
- pycrypto==2.6.1
- pyjks==17.1.0
- requests==2.9.1
- PyKCS11==1.4.4
- python-pkcs11==0.4.0
- asn1crypto==0.23.0

It was also used the version 1.61.0-1640 of the Citizen Card middleware.

3.6 Known problems and deficiencies

We could only test with the version 1.61.0 of the Citizen Card middleware, so the software can have a strange behavior on other versions of the middleware, mainly on getting the certificates from the *smart card*, and signing and verifying signatures. We also only tested with *smart cards* where the root certificate was from *Baltimore Cyber Trust Root*.

As new *Citizen Cards* are being created, the root certificates will naturally change, and our software does not have an automatic way of detecting that, so the solution is to update manually the trusted certificates.

Because our implementation only checks if the certificate is revoked with *CRL*s and delta *CRL*s, if a certificate does not use *CRL* and uses *OCSP*, the software will not check if the certificate is revoked.

Each packet send by an entity will have a timestamp, marked by the sender of the packet. When the server or the client send a packet and expect a reply, the timestamp of the packet reply should be 10 seconds or less after the initial packet was sent. This was made to prevent late responses to requests, but it has three downfalls. First, the sender of the reply can forge the timestamp. Second, it is needed that both client and server use the same clock. In this project, both are using the system clock, but in a real world system, they would need to use a synchronized clock between them. The last downfall is that if a client or a server take too long to process a packet, the reception of the packet will be discarded. For that, we have made a global variable "CLIENT_MAX_RESPONSE_TIME" in the client and in the server software (in seconds), that can be changed if the messages are being rejected because of late timestamps.

If the server leaks his private key, it is possible to impersonate the server and send wrong public keys of other clients when a client wants to sent a message. Only in this case it is possible to mislead the user, ciphering the message with the wrong public key. If the client leaks his private key for ciphering messages, it is still not possible to impersonate the user, because it also needs the authentication private key to confirm its identity. On the other hand, if a client leaks his authentication private key, it is possible to impersonate that person and send messages to other users, but it is not possible to read messages from other users, because those messages need to be deciphered with the private key for decipher messages.

Chapter 4

Conclusion

As far as security concerns, all the features expected were implemented, assuring the main goals of the project:

- Message confidentiality, integrity and authentication;
- Message delivery confirmation;
- Identity preservation.

With this project we have showed that it is possible to maintain a communication between clients using a central repository, with end-to-end encryption. It can be noted that in no moment the server can read any message exchanged between clients, because they are always ciphered with the private keys of the client. However, the server is an important intermediate to guarantee that the clients are trusty and the communication is made correctly, mainly by exchanging the public keys of the clients between them.