Design and implement a protocol

Homework6 - CNS Sapienza

Martina Evangelisti 1796480

11/12/2020

1 Introduction

This report describes a protocol to secure messaging, in particular it allows two entities `client` and `server` to communicate providing them:

- end-to-end encryption
- message authentication
- entity authentication

Last security service is obtained trusting a third party called `C`.

This protocol has three main stages:

1. PK key generation and exchange
2. Entity authentication
3. Secure communication session

2 Key generation and exchange

In the very first step of this phase a private and a public key are generated for each entity [`client,server,C`]. These keys are 2048 bits long and are generated using RSA key generation.
Both `client` and `server` send their identifier and the public key to `C`, we assume that in this step an attacker can't intercept or manipulate this messages. `C` stores the public keys in order to provide them when asked.

After that initialization step the `server` listens for upcoming connections.

3 Entity authentication

The mutual authentication protocol follows *Needham-Schroeder public-key protocol.*
When the `client` wants to communicate with the server it has to ask `C` for the public key of the server. So, a message with his identifier and the identifier of the server [`C,S`] is sent to C.
When it receives the request, C, sends back a message containing: the identifier of the server, the server's public key and the sign of the key plus the identifier. The digital signature scheme used is RSA PKCS#1 using SHA256 to generate the hash of the message to sign. In particular the scheme used is RSASSA-PSS, which stands for *Signature Scheme with Appendix based on a Probabilistic Signature Scheme using encoding EMSA-PSS.*
After receiving the message, the client can verify the signature of `C` using C's public key, under the assumption that each entity knows it. If the signature is correct, he knows he has the correct public key of the server and can start the communication with it.

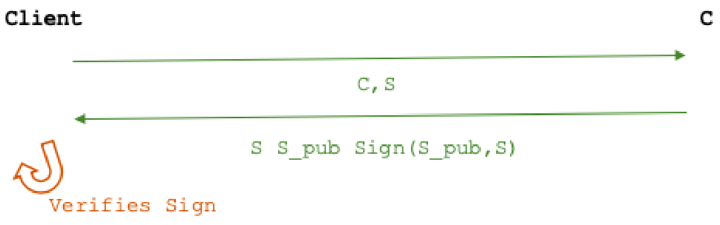

Figure 1: Client gets Server public key

Client generates a nonce of 16 random bytes, encrypts it, with his identifier appended, with the public key of the Server and sends it to the server `S` `[Kpub_S(N1,C)]`.
When the server receives the message wants to verify the identity of the client, so, in the same way as the client did, it gets `C_pubkey` from `C`.

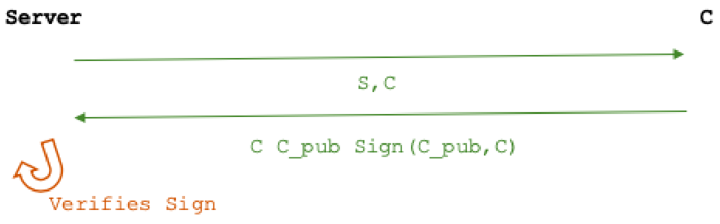

Figure 2: Server gets Client public key

After checking the signature, the server generates a new nonce of 16 random bytes and sends it, appended to the his identifier and to the previous nonce generated by the client and encrypted with the client public key `[Kpub_C(S,N1,N2)]`.
The client checks the nonce and sends back the new nonce encrypted with the server public key `[Kpub_S(N2)]`.
If, at the server side, the last nonce is verified, the two entities are now authenticated.

4 Communication session: confidentiality and message authentication

After the entity authentication phase, the Server generates a random key of 256 bits that will be used as a symmetric key between Client and Server. This key is sent to the client encrypted with the client's public key.
Now the real communication session can start.
The cipher used in order to guarantee confidentiality is AES-256 in ECB mode of operation.
The symmetric key is slightly different for each message exchanged. Every time a message is sent, a little key made of 6 random bytes is generated and sent with the message.
At the receiver side, when the little key is received the new symmetric key is the previous one with the last 6 bytes changed:
`sym_key=sym_key[:-6] +lit_key`.

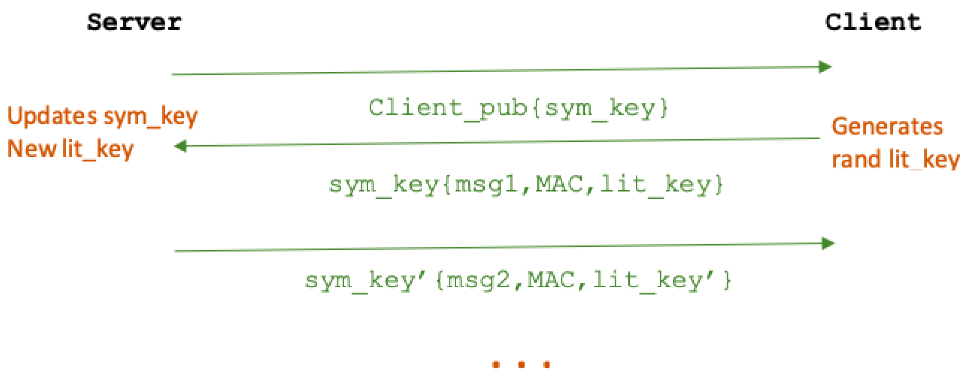

Figure 3: Encrypted communication

Each message is authenticated with MAC, *Message Authentication Code* following the MAC-then-Encrypt `MtE` approach.

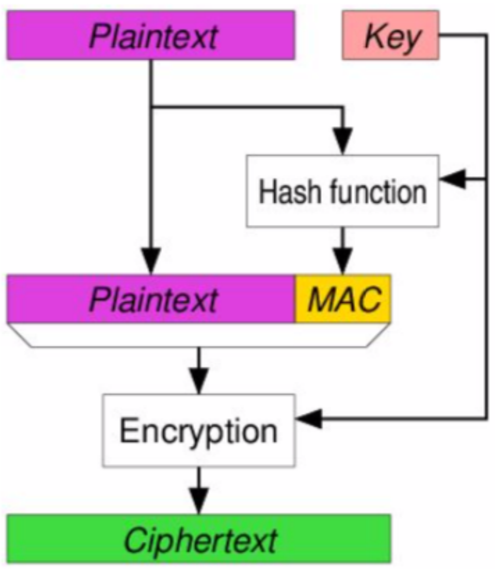

Figure 4: MAC then Encrypt
[2]

The Hash algorithm used is SHA-256 and the key used is the symmetric key. So,for each message integrity and origin are verified and, thanks to the encryption algorithm, confidentiality is guaranteed.

5 Implementation

The protocol described above is implemented in python[1] using *PyCryptodome* library. There are three main files: `C.py`, `Client.py`, `Server.py` and a startup script `start.sh` that will launch the programs.
The first to be launched is `C` that has to be always listening for incoming connections from the client and the server, after it `server` will start, and at the end the client.
The communication between these processes is realized using stream sockets with TCP protocol.
To run the code `start.sh` should be executed and Client and Server will automatically generate and exchange the keys and perform entity authentication. The communication session is automatically started with the exchange of two messages:

"HI,i'm the client C", from the Client
"HI,i'm the server S", from the Server

After these two messages are exchanged, the following messages can be written when *"insert the message: "* is asked.

[1]python 2.7

The connection can be closed by the client inserting the command *"exit"*. When the connection with the client is closed, the `client` program can be restarted in order to start a new communication session with the server `[cmd: python client.py]`.

Note that this protocol is designed to have a communication in which at each message from the client correspond a reply message from the server.

6 Security analysis

In order to analyze the security of the protocol previously described we have to analyze all its components.
The asymmetric encryption is performed using RSA and keys of 2048 bits. RSA PK scheme is based on the integer factorization problem and, using a large key space and a standard implementation, PKCS#1, we don't suffer the textbook implementation problems and, even if it still can be attacked we can consider it almost secure.

When we first communicate the public keys to `C` we're under the assumption that an attacker can't intercept or modify the message and, in all the phases of the protocol we assume that we can trust the entity `C`.

A typical attack in the Entity Authentication phase is man in the middle where a third entity *Trudy* authenticates as the client with the server and as the server with the client. Following Needham-Schroeder protocol in the fixed version, as described in Chapter 3, this attack can't be done.

Confidentiality is guaranteed by AES-256. The mode of operation used is ECB, *Electronic Code Block*, this mode of operation is chosen because it's the fastest and because this protocol is meant for a messaging application and so, we expect to have very short messages that, in most cases, won't even exceed the length of the block.
The drawbacks of this choice is that being independent, the blocks can be replaced and manipulated. But, in order to prevent this, each message has its MAC appended. Another problem of ECB is that the encryption of the same message twice will generate the same ciphertext. In this protocol the key is chosen randomly each time the client connects with the server, and so, this can happen only in the same communication session because if the client closes the connection and then opens another connections the key will be different.
In order to avoid the problem of having the same ciphertext if the content of the message is the same, a `lit_key` which represents the 4 last bytes

of the symmetric key is generated randomly for each message, and so the symmetric key will always be different.

About the message authentication the protocol uses HMAC-SHA-256. The approach MAC-then-Encrypt does not provide integrity on the ciphertext but provides integrity of the plaintext. However is not provably secure.

Note that in the implementation of the protocol that i realized the trusted entity C can only serve the entities one by one, so in this implementation a Denial of Service attack could be performed against C and the client and the server wouldn't be able to authenticate. In general we're under the assumption that the entity C is secure.

The protocol explained in that report, even if it guarantees confidentiality, message authentication and entity authentication, can't be considered always secure, although no protocol should be considered as such, because it still suffers some attacks but it tries to balance efficiency and security.

References

[1] PyCryptodome documentation
`https://pycryptodome.readthedocs.io/en/latest/src/api.html`

[2] Slides of the course: Computer and Network Security.
Emilio Coppa

[3] Understanding Cryptography by Christof Paar and Jan Pelzi

[4] Wikipedia