Homework #6
Message Exchange Protocol Design

CNS Course Sapienza

Riccardo Prinzivalle, 1904064

December 11, 2020

1 Homework Goal

This homework contains a basic design and implementation of a message exchange *secure* protocol. The idea is taken from different protocols learned from the slides and the lessons of the course. The implementation is done using the **Kathara** framework to virtualize a small network, while the protocol is written using bash scripting exploiting the **OpenSSL** library on Linux.

2 Protocol Design

The protocol can be divided into three main functionalities: *confidentiality*, *message authentication* and *entity authentication*. Each functionality is developed as follows:

- **Confidentiality** is achieved by using symmetric encryption with AES in CBC mode with key length of 256 bits, which is the standard for confidentiality for NSA [1], similar hint was found also here [2].
- **Message Authentication** is performed with SHA-256 using public key signature (OpenSSL supports directly HMAC but the version installed on the docker image on the background of Kathara is older than the minimum version implementing HMAC).
- **Entity Authentication** is based on a weaker version of the X.509 protocol, which introduces some vulnerabilities but simplifies the implementation, since this part is done by hand since OpenSSL contains only the primitives and not a complete authentication protocol.

Let's analyze every part more in the details; weaknesses will be analyzed in section 4.
Encryption is the simpler; it is just based upon the primitive from OpenSSL, and it uses the primitive as follows:

```
openssl aes-256-cbc -in message/to/encrypt -out
    encrypted/message.enc -pass file:key/file
```

The key used is based on a shared secret from a DH key exchange phase performed previously, and the instructions needed and used in this work can be found in [6]. The secret is then hashed in order to obtain the desired length of 256 bits for the encryption key. The instruction used is:

```
openssl dgst -sha256 -out hashed/key.sha256
    shared/secret.bin
```

To decrypt the message, it is easy as the encryption, it is just necessary to have the shared secret, derive the key by using hashing (as seen upside) and then decrypt with the following command:

```
openssl aes-256-cbc -d -in encrypted/message.enc
    -out decrypted/message -pass file:key/file
```

As already said, OpenSSL implements different primitives for authenticating messages, the problem resides in the version shipped with the docker image used in the practical experiment: it is older, so it does not support these new addition to the library. Due to this inconvenience, I found [3] that it is possible to use message digest with SHA256 with a public key pair for digital signature of a message using OpenSSL, the command is the following:

```
openssl dgst -sha256 -sign private/key.pem -out
    message/signature.sha256 encrypted/message.enc
```

Then both the encrypted message and its signature are sent to the recipient of the message.
The most complicated part is the entity authentication: the idea is taken from the X.509 protocol, with the idea to simplify it and, in an undesired way, weaken it, this is due to the lack of time to build the protocol from scratch. This part is based on a fusion of [9] and [4]. An entity starting a message exchange must verify its interlocutor and can optionally ask to its interlocutor to verify its identity (sender identity). To perform entity authentication we must define a third party entity, which both the interlocutors trust, which can guarantee the identity of interlocutors on the net: this new entity is called **Certificate Authority (CA)**. First of all, to define a CA it is necessary to define a master key, as follows:

```
openssl genrsa -aes256 -out CA_key.private -
    passout pass:password 4096
```

The key is generated with RSA of length 4096 bits and then encrypted with AES @ 256 bits. It is useful to use a large number of bits for this key since most of the security of the authentication resides on it, we are going to generate a root CA so it is necessary to adopt all security precautions that are possibles. Once a master key for the CA is created, it is possible generate a root certificate for the CA: in real world implementations, it is used a chain of trusted CA, and the root CA are only few and "bulletproof", and there exists a hierarchy of CA; in this case we'll use a single CA that acts as root. To generate the root certificate, it is used the following instruction:

```
openssl req -x509 -new -nodes -key CA_key.private
    -sha256 -days 1825 -out CA_root.pem -passin
   pass:password
```

The password is needed to decrypt the private key which was previously encrypted; then it is necessary to specify the number of days of validity for the certificate. We must keep this certificate under high security layers, since as for the master key, the trust of entity authentication is based on this certificate. Once the root certificate is generated, then the CA can start signing certificates for entities who asked for: the entity asked signing by sending a **Certificate Signing Request (CSR)** that the CA can use to generate the related certificate:

```
openssl x509 -req -in user.csr -CA CA_root.pem -
   CAkey CA_key.private -passin pass:password -
   CAcreateserial -out user.cer -days 1 -sha256
```

To generate the certificate, it is necessary to have:

- A CSR from the entity who wants a certificate.
- The certificate of the CA, in this case the root CA.
- The private master key of the CA.
- The password to decrypt the private key if it was encrypted previously

It is needed also to specify a number of days for the validity of the signed certificate.
A entity on the net can generate a CSR by executing the following command:

```
openssl req -new -key user.key -out user.csr
```

So the user needs only a private key to generate a CSR, no more than that. Once a user has obtained a certificate, the entity authentication can be performed: the initiator asks the recipient to send him its certificate, a special data structure D_r and the signature of D_r. The data structure D_r contains a timestamp, the identity of the sender and a publicly encrypted shared secret between the two entities (the shared secret is computed with a public key retrieved from the certificate authority).

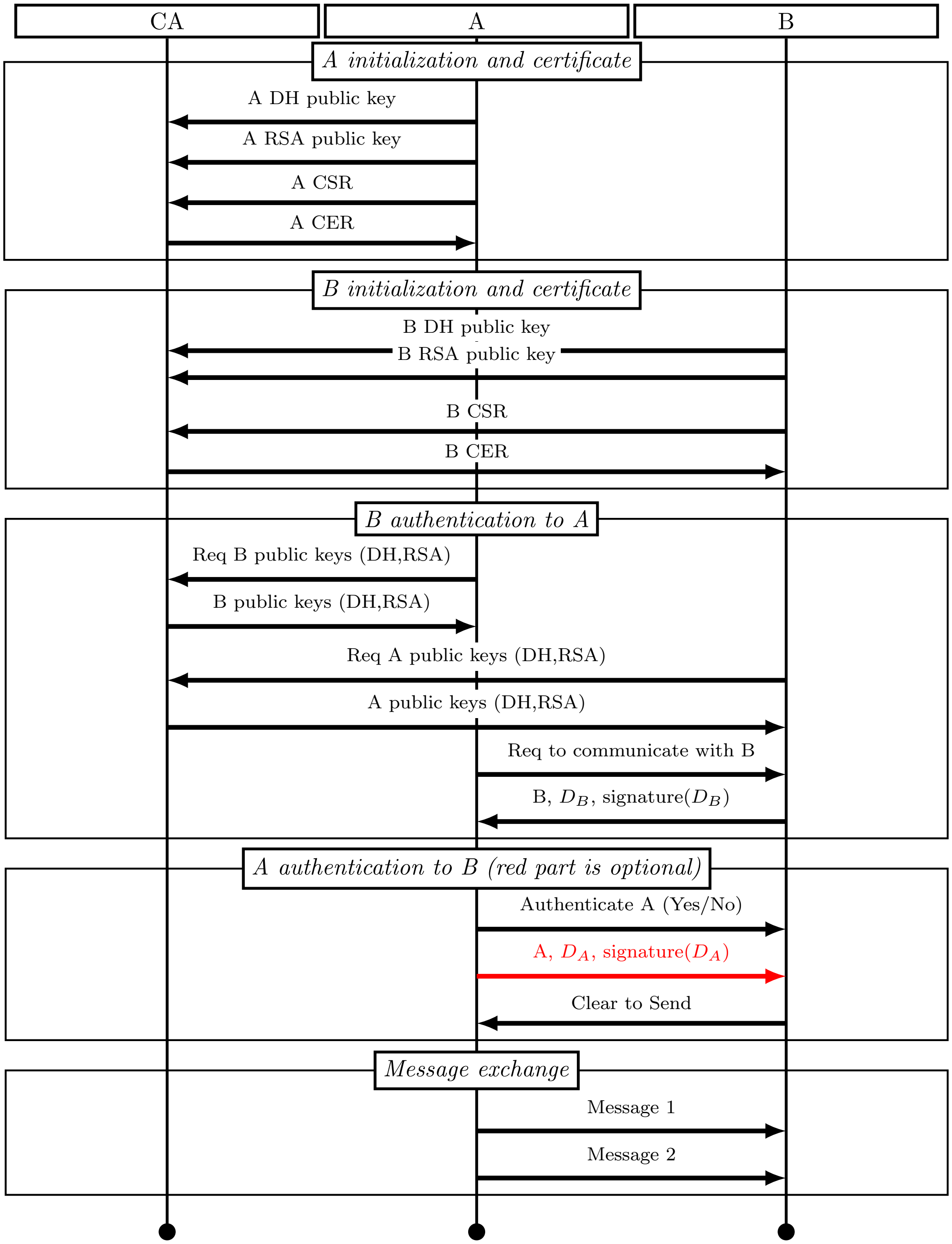
CA
A
B
A initialization and certificate
A DH public key
A RSA public key
A CSR
A CER
B initialization and certificate
B DH public key
B RSA public key
B CSR
B CER
B authentication to A
Req B public keys (DH,RSA)
B public keys (DH,RSA)
Req A public keys (DH,RSA)
A public keys (DH,RSA)
Req to communicate with B
B, D_B, signature(D_B)
A authentication to B (red part is optional)
Authenticate A (Yes/No)
A, D_A, signature(D_A)
Clear to Send
Message exchange
Message 1
Message 2

Figure 1: Communication Protocol Diagram, based on [8]

3 Protocol Proof of Concept

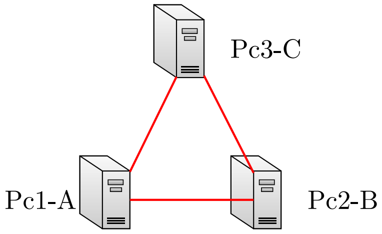

Figure 2: Proof of Concept Network Topology, based on[5]

This work compares the proposed implementation of RSA with a common library implementation of RSA itself and AES, the last one just to have an idea of how symmetric ciphers are in general more computational efficient than asymmetric ones. To have a fair comparison, AES uses 128 bit key, so as stated in table 1, it is necessary to have 3072 bit key for RSA. In the implementation proposed, since the key must have a length of 3072 bit, both p and q have a length of 1536 bit so their product has the correct length for the resulting key, as stated in section 7.6 of [7].

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer Factorization	RSA	1024	3072	7680	15360
Discrete Logarithm	DH, DSA, Elgamal	1024	3072	7680	15360
Elliptic Curves	ECDH, ECDSA	160	256	384	512
Symmetric key	AES, 3DES	80	128	192	256

Table 1: Key length comparison in public key and symmetric key algorithm

The 3072 bit key allows to encrypt a maximum of $3072 \div 8 = 384$ bytes, this is due to the characteristics of RSA itself, otherwise the modulo reduction will collapse the oversize message inside the modulo domain and it will be impossible to recover the original message after decryption. To overcome this limit, an idea could be to implement operation mode on asymmetric ciphers, but it is not feasible due to the fact that these ciphers are so much slower with respect to symmetric ones, and it is better to use the latter to encrypt larger messages. For

these reasons, the message to be encrypted has been chosen of dimension around 313 bytes. The proposed implementation needs to preprocess data to be sure to give to the encryption phase a integer number, this is done by the instruction `binarybuffer = ''.join(format(ord(x), 'b') for x in buffer)` which produces an output in binary form, which is then converted into base ten numbers by using the instruction `int(binarybuffer, 2)` whose output is used in the encryption phase.
For **RSA** comparison, both the key generation and message encryption are measured separately; the key generation is called once for every test since it requires more time with respect to the encryption phase, while the message encryption can be called more times by specifying the number of rounds when calling the function. Since the key generation phase requires different time depending on the test performed, I decided to perform 4 times the tests in different times of the day, and the median value are represented in tab. 2; encryption and decryption value are much more stable in different tests (All the original tests value can be found in the output.txt file attached with this report).

RSA	key generation	encryption throughput	decryption throughput
PyCryptoDome	11.68 sec	52,7 KB/s	6,37 KB/s
Proposed implementation	6.73 sec	0,825 KB/s	0.807 KB/s

Table 2: RSA time and throughput comparison

It is necessary to add that *PyCryptoDome* RSA implements a padding scheme, while the proposed implementation does not, which offer less security and maybe requires less computation effort. The strange thing is that the proposed key generation algorithm needs less time on average with respect to the *PyCryptoDome* one: as stated in [**?**], os.urandom, which is used here to generate the prime random numbers, uses on windows (on which I performed the tests) the function CryptGenRandom [**?**], which I was not able to understand if it waits effectively for the entropy pool to be full enough, but since os.urandom uses /dev/urandom on Linux machines, which does not check if the entropy pool is filled up, probably neither the windows function does it, so here the reason for which the proposed implementation needs less time on average.
Instead, as it was easily predictable, the library implementation has much more throughput, both in encryption and decryption, with slower decryption, while the proposed implementation has much more similar time for both phases (it uses the same function for both), probably the library implementation has some tricks to speed up the process which are possible only on one way in encryption.
To obtain a comparison with AES, I have chosen to use the ECB mode with padding; this is just for educational purposes, it is already known that symmet-

ric ciphers are faster than public keys ones. The comparison is represented in tab. 2.

AES vs RSA	Encryption	Decryption
AES ECB mode	1,49 MB/s	1,49 MB/s
Proposed RSA	0,825 KB/s	0.807 KB/s

Table 3: RSA vs AES comparison

Here, the time used by the key generation phase in RSA is not represented since the comparison this time is just on the pure encryption and decryption phase assuming we already have the keys for both algorithms.

4 Security Analysis

Figure 3: Improved version of Square And Multiply

5 Conclusion

In this homework, as a difference with respect to the AES implementation, the code is based mainly on existent libraries, with the exception of few functions, whose implementation is based on pseudocode of already known algorithms, with small improvements in order to speed up the computations. The bigger improvement is the introduction of the modulo reduction on every computation of SAM, without it the proposed implementation didn't work as the result grew too much and it remained stuck after some iterations. As expected, also in this case the work is slower than the library implementation, with the exception of the key generation phase, which needs much more digging to reach the real cause of its larger throughput.

References

[1] Commercial national security algorithm suite and quantum computing faq. `https://cryptome.org/2016/01/CNSA-Suite-and-Quantum-Computing-FAQ.pdf`, 2016.

[2] Aes quick hints on stack exchange answer. `https://askubuntu.com/a/60713`, 2017.

[3] How to use openssl: Hashes, digital signatures, and more. https://opensource.com/article/19/6/cryptography-basics-openssl-part-2, 2019.

[4] L. Dentella. Una certificate authority con openssl. https://www.lucadentella.it/2017/10/26/una-certificate-authority-con-openssl/, 2017.

[5] C. Fiandrino. Example: Network topology. https://texample.net/tikz/examples/network-topology/, 2014.

[6] S. Gordon. Diffie hellman secret key exchange using openssl. https://sandilands.info/sgordon/diffie-hellman-secret-key-exchange-with-openssl, 2013.

[7] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[8] Sari. Network communication diagram help. https://tex.stackexchange.com/q/460964, 2018.

[9] B. Touesnard. How to create your own ssl certificate authority for local https development. https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/, 2020.