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 [2], similar hint was found also here [3].
- **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 and security concerns 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 [9]. 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 [4] 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 [12] and [7]. 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
```

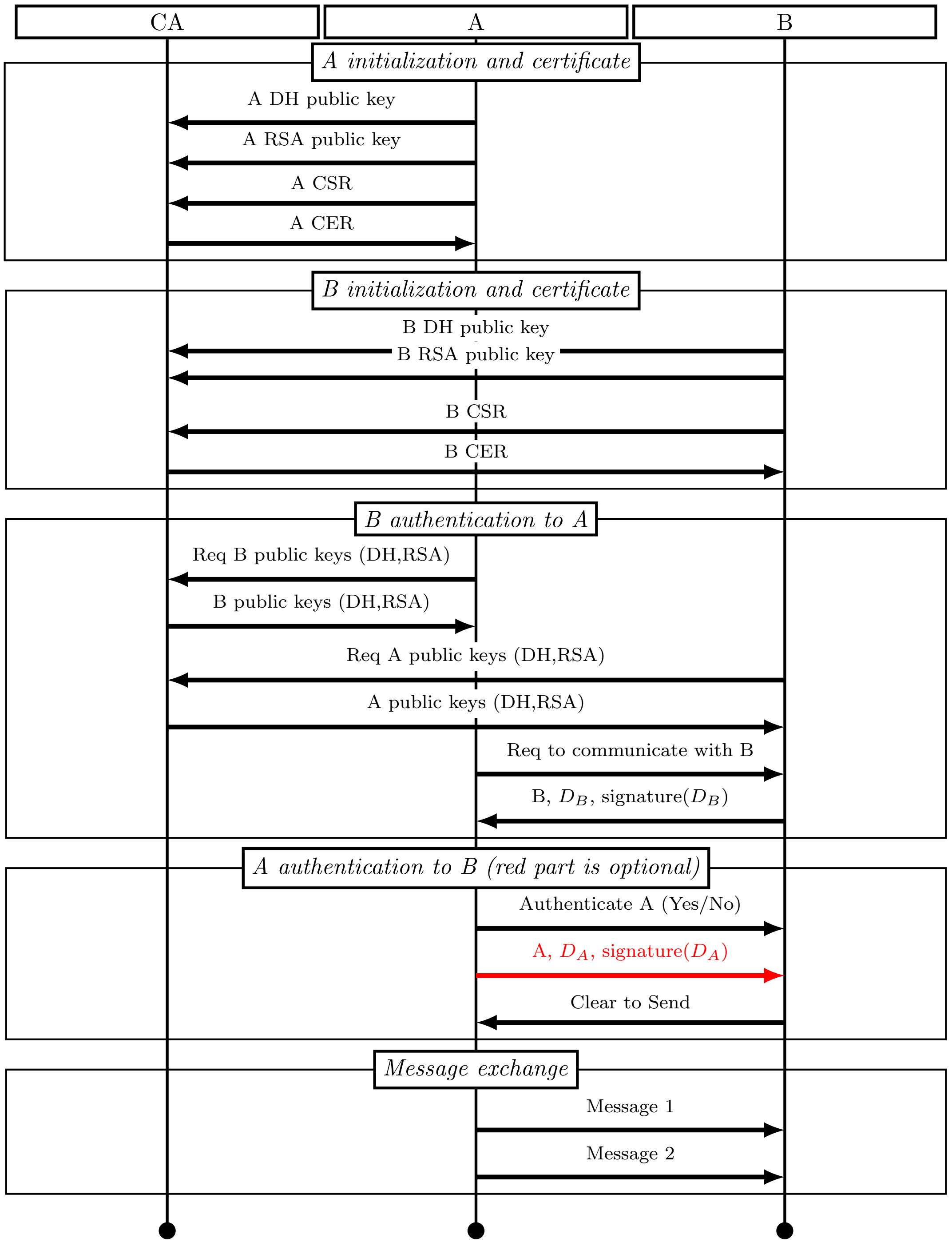
To generate the certificate, it is necessary to have:

- 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).
The complete protocol diagram can be seen on figure 1; the different phases are represented separately contained into rectangles, the only part to pay particular attention is in the red arrow: that part is sent to B only if A decides to authenticate itself to B, otherwise the previous message "Authenticate A" will contain "No" and B will send a clear to send to A and the message exchange can start.



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 [10]

3 Protocol Proof of Concept

This work contains also a small implementation of a proof of concept of the algorithm: it uses a basic topology of a network; it can be visualized in fig. 2.

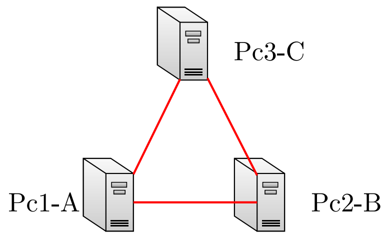


Figure 2: Proof of Concept Network Topology, based on[8]

The example implementation uses Katharà [1], a virtualization tool developed by Università Roma Tre to simulate, through Docker containers, the behavior of networks of various entity. To install this tool follow the instructions on [5], various OSes are supported. Once installed you need to go to ∼/.config/kathara.conf and modify the line about the docker image with:

```
"image": "marcospazianibrunella/network_infrastructures"
```

This allows the machine to have netcat command preinstalled, which will come in handy in this simulation. Once Katharà is working, go into the lab folder provided with this homework and execute the following command:

```
kathara lstart
```

To close the simulation environment it is necessary to perform a **kathara lclean**. We need a basic intro to how Katharà lab works to better understand the simulation: everything inside the pc* folder will be put inside the root of the correspondent pc when the machine is created and everything contained inside the pc*.startup will be executed during the startup of the machine. PC1 is the entity A, PC2 is the entity B and PC3 is the Certificate Authority. **A** will execute the script "A.sh", **B** the "B.sh" and **CA** the "server.sh", these scripts are located in /root/ of the corresponding machines. The programs are written using bash to exploit the power of OpenSSL (I know, maybe bash wasn't the best possible idea, I have over complicated my task probably). After some seconds of simulations, there must compare the messages foo1 and foo2 inside the folder /root/messages/ of PC2, and some text must be contained in them; if the messages don't appear after maximum ten second then restart the simulation with **kathara lrestart**, since sometimes the scripts get hanged in the startup of the machines due to different spawning times of the machines: this issue must have been mitigated by introducing the initial ping control but sometimes the script remains stuck in some point due to kathara problem with startup.
Basically the scripts contains the instructions previously explained in section 2 with the addition of extra instruction to synchronize the communication or

wait for some action from other entities to occur. An issue I have experienced is on the command used to communicate, which is *netcat*: basically all messages, even the text and control ones, are stored as files: this was chosen since all security related stuff, such as keys and certificates, are stored as files, so to level out everything and have a unique instruction to create the waiting part of the TCP communication (this is an implementation choice born after some days of googling the net). The problem is that, my idea was to use a single TCP connection for every channel (so one between A and CA, one between A and B and so on) and use it as a bidirectional channel, but the problem is that netcat (and all its variant) closes the connection when and EOF is sent over the channel, even when using the option -k, which is meant to keep the communication open when the client closes it to reopen it later, netcat raises some error, such as broken pipe or connection refused or similar, so I needed to change approach. The final idea is to put a netcat server on every machine which reopens the communication whenever an EOF closes it, and when a machine has to send something to another one then starts a new TCP session for every file. To manage file through netcat, I used a pipe with tar which takes care about file positioning on receiving machine [11].
To recap, all files are saved into /root/ of the machines, then the folder keys contains all security related files like keys and certificates while the folder messages contains the messages and the control communication like the CTS. Special mention goes to the file D_* which is created as a unique file with the three data seen in previous section, signed, sent and then disassembled at destination to get the three parts separated and ready to be used.
Probably the implementation has some error inside, so let me know if it does not work, I may record my screen and share the video as a proof (it's something like "it works on my machine", but the simulator is not that clear about the startup procedure and sometimes it doesn't work, the last runs on both my machines produce always a working example so I hope it works also during this evaluation).

4 Security Analysis

The implemented protocol for sure contains some, if not all possible, security vulnerabilities: the background layer is per se secure, since it uses a community driven application which is OpenSSL, yes, it can have some security issues but it is a community standard. The problem arises when we combine the OpenSSL primitives to build our protocol: here will come for sure some error, in addition this is an individual work so this adds some more problems since more than one pair of eyes sees batter than only one, and some errors can have been made by distraction or due to not completely clear concepts. Said that, let's start with the security analysis.
One basic assumption is that the first communication of all the entities on the net with the CA cannot be listened by an attacker and it is considered secure: this is not a real world hypothesis, but it simplifies the work of this homework.

Then all the entities which wants to talk with the CA already know the public key of CA and both shares the same parameters to build a Diffie Hellman key exchange, this is much unrealistic, and if we consider a real big network this will lead statistically to some couple of users having the same shared secret, so this assumption is suitable only for small networks; the size depends on the dimension of the shared parameters (in this case they are hard coded in the script).
The TCP ports used to communicate are all known except for the server port of A which is stored as a variable and sent as a first message with whatever entity A wants to communicate: in this way, it is possible to choose a random port for A just by writing some few lines of code, in the example proposed all the mechanism is implemented, also the exchange phase, except for the random generation. The fixed number for the TCP port can cause some network attack if the port are not well protected: in the example no firewall is present and whatever file sent through the port arrives on the destination machine due to the choices made to establish a communication channel. In addition to that, for simplicity (which is in general the cause of huge number of security threats) it is used the root user, which is the one logged in by default by Katharà at startup, to have a more secure way it is necessary to first create a new user at startup and then change to him and then start the scripts used here (changing root directories with the one from the user). These two action (root login plus the pipe of netcat and tar) allows anyone to send whatever file to every entity on the net which is a big security vulnerability. To counteract netcat+tar pipe, I haven't found a solution, it must be changed the base idea of how the protocol is implemented and abandon bash in favor of some higher level programming language.
Then, half of the OpenSSL instruction used needs an interaction with the user; to make everything automatic without the need to press a key, I used a pipe of **echo -e** and then the command which needed the user input, most of the argument of the echo are only "`\n`" to get default values, one of this example is on all the parameters needed to build the CA:

```
echo -e '\n\n\n\n\n\n\n' | openssl req -x509 -new -nodes
    -key CA_key.private -sha256 -days 1825 -out CA_root.
    pem -passin pass:8%0%Zef6kbBvG0g
```

This can be a security issues, if anyone is able to change the file with different values then everything stops working. Other instructions ignores any input and cannot be echoed in pipe so there exists a specific parameter: in general this happen when a user must type a password to decrypt a key, and this is done with the -passin or -passout parameters depending if the password is needed to encrypt or decrypt. This exposes to an issue: any password (generally any parameter) passed to OpenSSL (generally any CLI instruction) by pipe or by the specific parameter can be sniffed on the machine by any other process, as briefly explained in [6]; there it is said that files are more secure than simple pipe or plain text password, since processes can only grab the name of the file, but we are logged in as root user, so anyone can send a script file, make the

machine execute it and then sniff the file and access it since everything run will have root privileges.
As writing this report, I put an eye to the code to read the passages to made fig. 1, and I saw that, even if I shared, through DHKE, a common secret between A and CA, and B and CA, then no symmetric encryption is used for the communication between these entities, so to avoid this is just necessary to encrypt with AES the communication prior of sending anything on the channel as it is done between A and B, with the instruction previously seen 2.
Another unseen was the verification of the certificates of A and B in the authentication phase: so the authentication is based only on D_* and its signature. It is needed to implement another interaction with the CA for every certificate to verify and then execute the -verify option on openssl to verify the certificate by using the CA root key used to generate the certificate itself.
Regarding the communication between A and B, only the messages are fully encrypted, the other control messages are only encrypted in part, which is a stupid thing since both A and B ask CA to give them the public keys of the other entity, it could have been better to encrypt all the communication between the two, but initially I haven't seen it.
Another addition to this long list become clearer after a post on Piazza, where one colleague asked if we need to used different public keys for the communication, which is something obvious, but then I realized that any entity generates only one public key and uses it with both the other entities on the net; probably this would have been solved if I have realized that the communication with the CA was not encrypted.
Now a vulnerability I decided to implement to simplify the code: if using the two ways authentication, the D_* messages sent does not change, so no nonce is used and then this opens to possible replay attacks. Another problem linked to D_* is that I don't know how to implement the identity of a entity to be sent, so I decided to simply sent a file named with the entity name and containing the same inside. This causes no identity verification. As already said before, authentication is only based on the shared secret, its decryption and signature verification of D_*.
This poses an end to what I was able to find as security vulnerabilities of the proposed implementation.

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] Katharà framework. https://www.kathara.org/.

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

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

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

[5] Kathara install wiki page. https://github.com/KatharaFramework/Kathara/wiki/Linux, 2020.

[6] Caf. How to generate an openssl key using a passphrase from the command line? https://stackoverflow.com/a/4300425, 2010.

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

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

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

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

[11] Tom. Send multiple files with netcat. https://stackoverflow.com/a/44894223, 2017.

[12] 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.