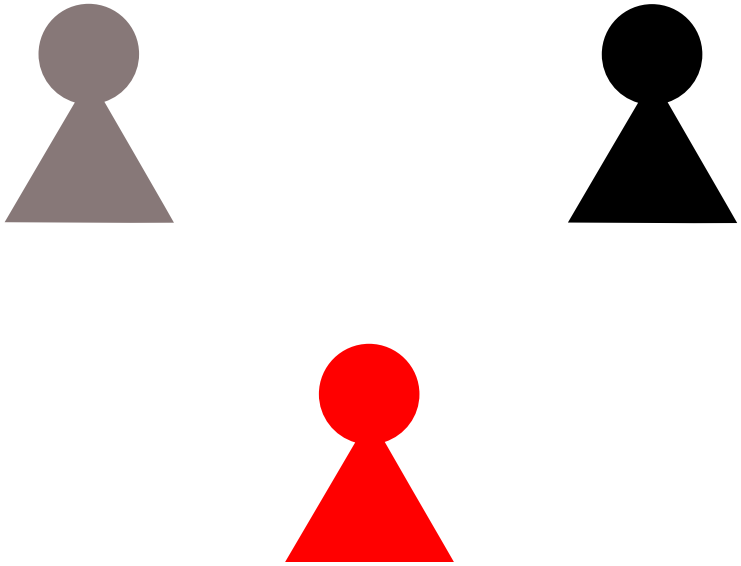
Security In Networked Systems
Project's Report

David Costa
Giuliano Peraz



Part I
Introduction

1 Scenario (in Italian)

Si consideri un'applicazione distribuita di tipo cliente-servitore in cui ciascun processo possiede una coppia di chiavi pubblica e privata.
Si assuma che il servitore conosca la chiave pubblica di ogni suo cliente (i certificati non sono necessari).
Si specifichi, si analizzi, si progetti ed, infine, si implementi un protocollo crittografico che soddisfa i seguenti requisiti:

- al termine dell'esecuzione del protocollo, viene stabilita una chiave di sessione tra cliente e servitore;
- al termine dell'esecuzione del protocollo, il cliente ritiene che il servitore dispone della chiave di sessione e viceversa;

La specifica del protocollo deve mettere chiaramente in evidenza le ipotesi sotto le quali il protocollo funziona correttamente.
L'implementazione deve comprende la realizzazione di un prototipo in cui il server ed il cliente si scambiano del materiale (testo o binario) cifrato con la chiave di sessione .

Le attività di specifica, analisi e progetto dovranno essere documentate da una concisa relazione scritta.

Part II
Protocol design & analysis

2 Real protocol

In this section we explain the main idea, the main features and the objectives of the protocol, including a description of each message of the real protocol.

2.1 Protocol Idea

We thought about a protocol that doesn't need to send the key over the unsecure channel. The idea is: the two main actors, the Client A and the Server B, have to create indipendentely the same session key which is the combination of two secret fresh random numbers (nonces N_a and N_b) being generated by the actors of the protocol.

In order to avoid leakage of information that makes the key easy to compute, the generation method is based on the hash of the two generated secrets. A cryptographic hash function is used because of its avalanche effect: if one or more bits of a message m are changed, then the output of $hash(m)$ will substantially change, i.e. small differences in the original message lead to big and unpredictable differences in the hash.

The avalanche effect plays an important role in the protocol because it makes sufficient to generate only one fresh nonce each session, instead of two, in order to create a completely new session key. What does this mean from a practical standpoint? If either the client or the server implementation is poorly written in the sense that a lazy programmer has implemented a nonce generation function which returns the same nonce each time, the key isn't compromised due to the fact that each bit of it depends on the other (hopefully) random nonce, so when the two nonces will be combined, the avalanche effect will create a key which will be unrelated to any other one and is still "difficult" to bruteforce.

The specification of our protocol clearly states that a random generation of the two nonces is needed, in order to increase the level of security, but the protocol doesn't fall even if one of the actors makes use of a wrong implementation, be either the server or the client. Of course, if both implementations are wrong then someone doesn't actually use our protocol.

The on-the-fly generation of the key by each part of the protocol permits also to reach the key authentication objective after only 2 messages and key confirmation objective after 4.

2.2 Protocol Objectives

By the specification of the project explained on the preceding page we can extract (and express in BAN logic) these two objectives for our protocol:

- Key Authentication:
  $A \models A \overset{K}{\leftrightarrow} B, \quad B \models A \overset{K}{\leftrightarrow} B$
- Key Confirmation:
  $A \models B \models A \overset{K}{\leftrightarrow} B, \quad B \models A \models A \overset{K}{\leftrightarrow} B$

Key Authentication means that each partecipant must know that the key K is the current session key, this objective is mandatory. Key confirmation means that each partecipant knows that the other part truly has the same session key for communicating.

The structure of our protocol reaches also another objective that isn't required by the scenario: Key Freshness. This achievement is met by involving quantities that are believed to be fresh by both the actors A and B.

2.3 Real protocol messages

$M1 \quad A \to B: \quad A,\ \{\{N_a\}_{e_A^{-1}}\}_{e_B}$
$M2 \quad B \to A: \quad B,\ \{N_a,\ \{N_b\}_{e_B^{-1}}\}_{e_A}$

The key is a function of N_a and N_b. We define $K = \langle K_{AB} \rangle_{N_a, N_b}$ for the sake of clarity.
The way N_a and N_b are combined to create K is explained on page 8

$M3 \quad A \to B: \quad \{\mathrm{H}(N_b)\}_K$
$M4 \quad B \to A: \quad \{\mathrm{H}(N_a)\}_K$

H is an hash function whose output size is appropriate (see Protocol implementation).

2.4 Protocol actions

A wants to communicate with B. In order to do this a session key must be estabilished by executing the following actions:

1. A generates a random number N_a, signs it with its own key and encrypts it for B using the public key e_B. This quantity is then sent to B in a message together with the name of A in plaintext. The presence of the name of A unencrypted in the message allows a faster verification of the message for B because it doesn't need to check against all the public keys it knows. Once received the message B decrypts the message and verifies the signature.

2. B generates a random number N_b and signs it with its own key. The signed $N_b$and plain $N_a$are then sent encrypted for A using the public key e_A together with the name of B in plaintext for the same reason of the previous point. A decrypts the message and verifies the signature.

3. A and B now calculate the session key by applying a function that ensures that all the bits of the key are dependent on each secret N_a and N_b, i.e. an hash. The session key is generated and it never shows up as a piece of information in a protocol message so a verification is needed to check whether both partecipants has the same key; this is done by issuing two challenges.

4. A proves to B that it knows the session key by hashing N_b and sending it to B encrypted with the session key K. B decrypts it and verifies the hash.

5. B does the same, hashing $N_a$and sending it to A encrypted with the session key K. A must then decrypt the message and verify the hash.

If any verification fails during the execution of the protocol, no further messages should be sent and the execution should be stopped (e.g. by closing the connection).

3 Idealized Protocol

The idealized protocol is a translation of each crypted part of the real protocol messages in an appropriate form that explicits the meaning of each message rather than what the message transport through the network. In this section we will show which are the messages of the idealized protocol, including the implicit information and which is the meaning of each message.

If you are interested in the proof of the absence of problem into this protocol using BAN logic please go to section Proof by BAN logic

3.1 Idealized Messages

- $M1 \quad A \to B: \quad \{\{N_a,\ A \overset{N_a}{\rightleftharpoons} B\}_{e_A^{-1}}\}_{e_B}$
- $M2 \quad B \to A: \quad \{N_a,\ \{N_b,\ A \overset{N_b}{\rightleftharpoons} B\}_{e_B^{-1}}\}_{e_A}$
- $M3 \quad A \to B: \quad \{\mathrm{H}(N_b),\ \mathrm{A} \overset{\mathrm{K}}{\leftrightarrow} \mathrm{B}\}_{K=\langle K_{AB}\rangle_{N_a,N_b}}$
- $M4 \quad B \to A: \quad \{\mathrm{H}(N_a),\ \mathrm{A} \overset{\mathrm{K}}{\leftrightarrow} \mathrm{B}\}_{K}$

3.2 Idealized message meaning

- **M1**: A says to B that the nounce N_a A has just generated is good for generating the key.
- **M2**: B says to A that the nounce N_b B has just generated is good for generating the key.
- **M3**: A says to B that $K = \langle K_{AB}\rangle_{N_a,N_b}$is the session key for this session.
- **M4**: B says to A that K is the session key also for him.

4 Proof by BAN logic

Initial suppositions

According to the scenario: $B \models \overset{e_A}{\longmapsto} A$.

Since there are no certificates involved, the only feasible ways for the client to authenticate the server is via a shared secret or by letting the client know the server's public key.

We chose for the latter thus $A \models \overset{e_B}{\longmapsto} B$ must hold in order for the protocol to work correctly. Without this assumption our protocol (like other protocols) is subject to Man-In-The-Middle attack.

Also, since B is considered a server from the A's point of view, and since we use two shared secrets for building the session key, one from the client and one from the server, we may suppose that $A \models B \Rightarrow (A \overset{N_b}{\rightleftharpoons} B)$, that is the client trusts the server in generating shared secret, but not viceversa.

After message M1

A generated the secret $N_a$thus:

$$A \models \#(N_a),\ A \models A \overset{N_a}{\rightleftharpoons} B.$$

From B's point of view (applying the Message Meaning Rule):

$$\frac{B \models \overset{e_A}{\longmapsto} A, \quad B \triangleleft \{A \overset{N_a}{\rightleftharpoons} B\}_{e_A^{-1}}}{B \models A \mid\sim A \overset{N_a}{\rightleftharpoons} B}$$

After message M2

B generated the secret N_b thus:

$$B \models \#(N_b),\ B \models A \overset{N_b}{\rightleftharpoons} B$$

From A's point of view (applying the Message Meaning Rule):

$$\frac{A \models \overset{e_B}{\longmapsto} B, \quad A \triangleleft \{A \overset{N_b}{\rightleftharpoons} B\}_{e_B^{-1}}}{A \models B \mid\sim A \overset{N_b}{\rightleftharpoons} B}$$

Since

$$\frac{A \models \#(N_a)}{A \models \#(N_a, N_b, A \overset{N_b}{\rightleftharpoons} B)}$$

then, for the Nonce Verification Rule:

$$\frac{A \models \#(N_a, N_b, A \overset{N_b}{\rightleftharpoons} B), \quad A \models B \mid\sim (N_a, N_b, A \overset{N_b}{\rightleftharpoons} B)}{A \models B \models (N_a, N_b, A \overset{N_b}{\rightleftharpoons} B)}$$

applying the postulates seen during the class we obtain:

$$\frac{A \models B \models (N_a, N_b, A \overset{N_b}{\rightleftharpoons} B)}{A \models B \models A \overset{N_b}{\rightleftharpoons} B}$$

and, for the Jurisdiction Rule:

$$\frac{A \models B \models A \overset{N_b}{\rightleftharpoons} B, \quad A \models B \Rightarrow (A \overset{N_b}{\rightleftharpoons} B)}{A \models A \overset{N_b}{\rightleftharpoons} B}$$

B believes in the freshness of (N_a, N_b) because $\frac{B \mid\equiv \#(N_b)}{B \mid\equiv \#(N_b, N_a)}$ and so does A (already proved).

At this point, both the actors can use (N_a, N_b) to generate the session key $K = \langle K_{AB} \rangle_{N_a, N_b}$ indipendently.

The fact that B generated the key leads to $B \mid\equiv A \overset{K}{\leftrightarrow} B$. Same thing applies to A, but it has a stronger confidence in $A \overset{K}{\leftrightarrow} B$ because of the jurisdiction rule applied before.

Considering that $A \mid\equiv A \overset{K}{\rightleftharpoons} B$ and $A \mid\equiv A \overset{K}{\rightleftharpoons} B$ the Key Authentication objective is reached.

After message M3

By applying the Nonce Verification Rule:

$$\frac{B \mid\equiv \#(A \overset{K}{\leftrightarrow} B),\ B \mid\equiv A \mid\sim A \overset{K}{\leftrightarrow} B}{B \mid\equiv A \mid\equiv A \overset{K}{\leftrightarrow} B}$$

After message M4

By applying the Nonce Verification Rule:

$$\frac{A \mid\equiv \#(A \overset{K}{\leftrightarrow} B),\ A \mid\equiv B \mid\sim A \overset{K}{\leftrightarrow} B}{A \mid\equiv B \mid\equiv A \overset{K}{\leftrightarrow} B}$$

After M3 and M4 we obtain the required key confirmation objective. As a side effect, key freshness is also obtained thanks to the fact each part has generated the session key in the current session of the protocol.

Part III
Protocol implementation

In this part we will explore the implementation details and the specifications of the project in order to implement correctly our protocol.

5 Protocol Specification

The protocol will be based on the follwing specifications, some of them are mandatory because of the scenario, some others instead have been added by ourself.

Public/private keys: we recommend Elliptic Curve Cryptography, because it is faster and has smaller key size than RSA but with the same level of confidentiality. However, due to some bugs in the OpenSSL library, in the implementation we was forced to use RSA criptographic and signing algorithms (see 6.2 on page 10 for more details).

Nounces: N_a and N_b must be **RANDOM** 128-bit numbers.

Identifiers: the length of the identifiers is not fixed. We recommend to use as the small size as possible in order to use the ID as the hash of a table for retrieving the correct public key (server side).

Hash Functions: we recommend the last standard CRHF (Collision Resistant Hash Function). In particular we suggest **SHA-256** or any equivalent hash function with 256-bit output.

Salt: the salt for the key generation must be $S = "FzHp3CbMao"$.

Key Generation Function: the key is generated by hashing the concatenation of the nonces and the salt:

$$K = H(N_a|N_b|"FzHp3CbMao")$$

where $H(\cdot)$ is an hash function with output of 256-bit and so is the key.

Initialization Vectors: They must be random numbers.

6 Implementation details

In this section we show the directory structure of the project, what the files contains and some considerations about the implementation of the project, that is which problems we encountered when developing the project itself and which solutions we adopted.

6.1 Structure of the project

The project is composed by two little programs: the client and the server. In order to maintain an ordered structure we have created some directories which store files divided by extension: the header *.h files are all stored in ${PROJ_DIR}/include directory, source code *.c files are in the ${PROJ_DIR}/src directory instead.

Since this is a demo project, that is we have to show our protocol works as expected, we saved the private and the public key, both client and server, into the ${PROJ_DIR}/key directory. We know that this is a security threat and a very big mistake, so we recommend putting correctly those files into a more secure structure such as a key database, i.e. GnuPG database, if someone would use our client and server source files for his future projects.

The project has been built using the cmake tool and the Gnu C Compiler, so there is a CMakeLists.txt file into ${PROJ_DIR} and the instructions for building the programs are in the README.txt file.

6.1.1 Source code files

We decided to implement separately the client and the server parts, so one of us wrote the client and the other one the server. We wroteeverything using the C programming language, as requested by the professor, and we used the OpenSSL library as said by the project specifications.

The project code is divided into these parts:

server.c It is the main file which contains the main() of the server, its internal data structures and the whole code the server has to run in order to work properly. Remember that a server can't brutally exit like a client program can do, that is the server must recognize all errors and must consequently act in order to close the communication if something goes wrong during the data exchange.

client.c This file contain the main() function of the client. The code permits the client to communicate with the server starting with the D&G protocol, then the client will send a file and will wait the server response which will consist of another file, then the client will exit.

protocol.c The code of the protocol is written here. The protocol doesn't only mean the creation of the messages client and server exchange, but also some functions being used for the message verifing step. If one single protocol message fails, then the protocol is aborted, so the client program will exit and the server will close that connection.

utils.c This file contains all the utility functions both client and server share. Here there are encryption, decryption, signing and verifying by means of public key, decryption and encryption by means of symmetric key, digest creation using SHA256 algorithm and some other functions that permits to concatenate bit strings in one single buffer and viceversa.Please add the part of the concatenating messages, sending a buffer etc etc etc.

We designed this project in order to simplify reading, mantaining and to make easier extending features for whom will able to do that.

We have developed the code using as much standards as possible, an example of this is the using the std* libraries.

6.1.2 Header files

Each header file, with the same name of the .c file, contains the definitions of the publicly available functions and also their documentation. We have documented the code following the doxygen documentation tool language, that is a javadoc-style markdown language, but in our opinion it is more flexible and powerful than the standard documentation tool of Java language.

There is also a header file called common.h which contains all the needed include directives and some useful macros which are used by both client and server.

6.2 OpenSSL problems

During the development of this project, we encountered some bugs and some limitations of the OpenSSL library. We want to develop a system which will use the Elliptic Curve Cryptography, but during the project development we noticed that some of the OpenSSL sign verification functionalities didn't work well because, when testing the program, the return value of the signature verification function wasn't what expected: sometimes the signature verification function failed, sometimes it succeded but always on the same test designed for passing the verification. That problem has been solved by changing the EC keys to RSA keys. Only this change have made the same test to pass every time as expected without source code changes.

We saw another "strange" behaviour of OpenSSL library functions: when using EVP_PKEY_Encrypt* we noticed that we couldn't encrypt more than public_key_size/8 - 11 bytes. After some time googling in the net, we learnt that this limit is by PKCS#1 standard, so, in our case, we couldn't encrypt more than 501 bytes. We tried using a smaller nonce size, but the RSA signature occupies 512 bytes by itself, so we couldn't encrypt anything without exceeding RSA limitations.

Since we didn't want to modify the protocol by moving the encrypted sign of the nonce outside the encrypted block, we moved to use the digital envelopes OpenSSL provides. This is useful for exchanging files or everything greater than 501 bytes with few changes to the code. Digital envelopes solved the problem, allowing exchange of an arbitrary amount of data.

6.3 Using standards

We have developed the project thinking about portability, so we have included and used only standard libraries, i. e. stdint.h in order to make as easier as possible a port to another UNIX-like operating system, or Windows-like but with MinGW software compilation installed.

6.4 Extract and concatenate messages

A problem we have encountered during the development was the recognizing of which bytes belong to one field of one message and which not so. We have

solved this problem by means of extr_msgs() which uses a variable number of struct message_data (more details in the documentation of utils.h), in order to extract the field of a message, that is this function puts into those structs the information about a field by knowing the field's length.

Another similar problem was to create a single message, a. k. a. bitstream, from the separated field. This is the opposite problem wrt the above one. We have solved this by developing the conc_msgs() function which is the exact opposite of extr_msgs(), that is conc_msgs() accept a sorted variable list of message_data structs and puts them in a single bitstream. For example, since M1 is composed by 5 parts in the implementation of the protocol, when it received received M1, the server must extract each part from a single bit stream. The extraction has been made by the extr_msgs() function. But the server must also create M2 which is also composed by 5 parts. In order to create M2, the server creates each parts separately, then it uses conc_msgs() in order to produce a bit stream to be sent to the client. More details are in the documentation of the project.

6.5 Sending Buffers

recv() and send() could not send all the bytes user wants them to send, so it can be needed to call those system calls more times for sending all the data the user wants to. In order to solve this problem we have implemented sendbuf() function that is in charge of continuously try to send the buffer passed by parameter until all bytes has been sent.

6.6 Implemented protocol

After the development of this project, client and server will send these messages:

$M1 \quad A \rightarrow B: \quad A,\ IV_{env},\ E_{kenv},\ \{\{N_a\}_{e_A^{-1}}\}_{e_B}$

$M2 \quad B \rightarrow A: \quad B,\ IV,\ IV_{env},\ E_{kenv},\ \{N_a,\ \{N_b\}_{e_B^{-1}}\}_{e_A}$

$M3 \quad A \rightarrow B: \quad \{\mathrm{H}(N_b)\}_K$

$M4 \quad B \rightarrow A: \quad \{\mathrm{H}(N_a)\}_K$

Where letters subscripted with env means that they are parameters of the OpenSSL's digital envelopes.

7 Reusable parts of the project

This project wasn't been thought to be extendend in future projects, since the client.c and server.c describes only an example of what a program must do for using our protocol correctly. In our opinion the reusable code is written inside protocol.c and utils.c. The first file has been thought for users which want our protocol to use, when the latter file is a collection of functions with the same purpose of protocol.c, but they are more general, so we can use them, and anyone can do that, in any other project we will design and implement in the future. An example of this is conc_msgs() and extr_msgs() functions which will accept a struct message_data, explained in utils.h with the documentation, in order to make easier both the concatenation of single parts of a message in a single one and the extraction of the interesting parts from a single large message received by the client or the server.

Code is deployed under the GPL v3.0 License, whose full text is available in the repository.

Part IV
BAN Logic in Latex

This report is written using a LaTeX frontend called LyX.

Since LaTeX lacks of support for writing BAN logic, some custom commands were written by us to assist in the writing of this report. The commands' sources can be found in the preamble of this file; an example of what the commands can do is shown below:

Simple predicates

- $A \models B$, A \believes B
- $A \mid\sim B$, A \said B
- $A \overset{K}{\leftrightarrow} B$, A \key{K} B
- $A \overset{K}{\rightleftharpoons} B$, A \secret{K} B
- $P \triangleleft Q$, P \sees Q
- $\{X\}_K$, \encrypted{K}{X}
- $\{X\}_{e_A}$, \encryptedfor {A}{X}
- $\{X\}_{e_A^{-1}}$, \signed{A}{X}
- $\#(N)$, \fresh{N}
- $\overset{e_Q}{\longmapsto} Q$, \publickey{Q}
- $\Rightarrow N_a$, \authority{N_a}
- $\langle X \rangle_Y$, \combined{X}{Y}

Postulates

1. Message Meaning
   (a) $\dfrac{P \models Q \overset{K}{\leftrightarrow} P,\quad P \triangleleft \{X\}_K}{P \models Q \mid\sim X}$, \meaningI{P}{Q}{X}{K}
   (b) $\dfrac{P \models \overset{e_Q}{\longmapsto} Q,\quad P \triangleleft \{X\}_{e_Q^{-1}}}{P \models Q \mid\sim X}$, \meaningII{P}{Q}{X}
   (c) $\dfrac{P \models Q \overset{Y}{\rightleftharpoons} P,\quad P \triangleleft \langle Y \rangle_X}{P \models Q \mid\sim X}$, \meaningIII{P}{Q}{X}{Y}
2. Nonce Verification
   (a) $\dfrac{P \models \#(X),\ P \models Q \mid\sim X}{P \models Q \models X}$, \nonce{P}{Q}{X}
3. Jurisdiction rule

(a) $\dfrac{P \mid\!\!\models Q \mid\!\!\models X,\ P \mid\!\!\models Q \Rightarrow X}{P \mid\!\!\models X}$, \jurisdiction{P}{Q}{X}

4. Other postulates

(a) $\dfrac{P \mid\!\!\models \#(X)}{P \mid\!\!\models \#(X, Y)}$, \freshcouple{X}{Y}