

Remote Password Manager

Cybersecurity project

Alessandro Noferi, Fabio Condomitti

June 29, 2018

Contents

Introduction	2
System architecture	3
Key Establishment Protocol	4
Observations	5
M2	5
M4	5
Ban Logic Analysis	6
Real Protocol	6
Idealized Protocol	6
Hypothesis	6
Key Assumption	6
Freshness	6
Trust	6
Objectives	6
Postulates application	7
Requests protocol	8
Add	8
Get	8
Delete	9
Update	10
Implementation	11
Client	11
Server	13
Certification Authority	14
Threat Model	15
Server Point of View	15
Password length	15
Conclusion	16

Introduction

In this project we propose a remote password manager based on the client-server paradigm. Through the use of a desktop application, the user can connect to the server in order to safely manage his passwords. By using this connection, the user will be able to add, get, change and remove passwords that are securely stored in a database on the Server.

All the communications among Client and Server take place under a secure channel protected with a shared symmetric key, K_{cs} . In order to establish this key, there is a protocol based on public key scheme encryption, which will allow the two parties to exchange data securely, allowing the Client to safely store his personal and private information.

This application can be useful because it allows the user, storing only a fairly complex password, to obtain and manage a huge number of passwords different from each other. In this way we can offer the customer a higher level of security, because if one of them is compromised, the others, being completely different, would be safe.

Moreover, this automatic mechanism allows the user to memorize complex passwords, instead of simple and short ones that are easy to memorize by a human being, but not very suitable to guarantee security against bruteforce or guessing attacks. The next sections will explain the operation in detail.

System Architecture

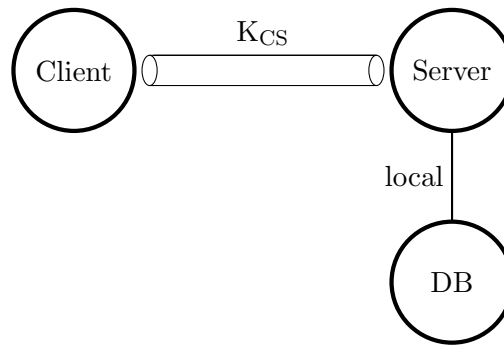
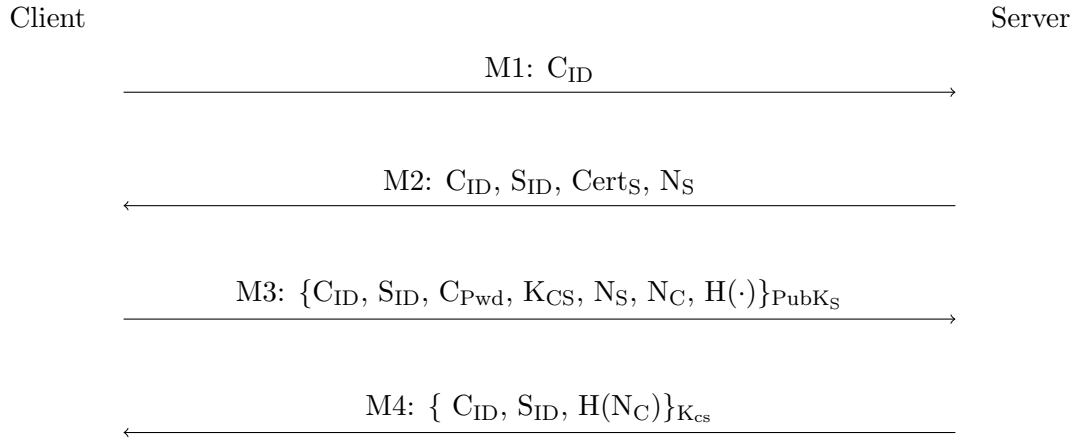


Figure 1: System Architecture

In this system the Client will use a Graphical User Interface (GUI) in order to establish a secure channel with the Server, which will lead to the creation of a new shared symmetric key K_{cs} valid only for one session. Through its use, Client and Server can communicate securely to manage user's passwords.

The Server interacts with a Database that stores all the data in an encrypted format.

Key Establishment Protocol



Let us now analyze the evolution of the protocol used to obtain a share symmetric key between the two parts.

Messages exchanged:

- M1: the Client sends an identifier, C_{ID} , to the Server, which corresponds to the chosen username. This message indicates that a user wants to open a new connection via a socket
- M2: the Server uses this C_{ID} , and replies a message containing also:
 - S_{ID} , to prove his identity
 - *Certificate*, provided by a Certification Authority (CA)
 - N_S , a truly random quantity
- M3: the Client uses this *Certificate* to verify the Server, and to extract the $PubK_S$. After that he can respond with their identifiers, and also:
 - C_{Pwd} , the password user has entered to perform login/registration
 - K_{CS} , truly random quantity on 32 bytes
 - N_S , nonce used to guarantee freshness for the Server
 - N_C , nonce used to guarantee freshness for the Client, after M4
- M4: the Server uses this session key K_{CS} to encrypt this message. In addition to the identifiers, it also contains:
 - $H(N_C)$, the hash of the N_C that the Client will use to verify M4's freshness

Observations

M2

After this message the Client must verify and authenticate the Server. Using the $Cert_S$, the application extracts the distinguished name of the server to make sure about the identity of the server.

Then the Client checks the validity of the $Cert_S$ and that it has not been revoked using the CRL (Certificate Revocation List obtained by the trusted *Certification Authority*). Finally, if all the previous steps went well, the Client can extract from the $Cert_S$ the Server's public key.

M4

In this message we used an $H(N_C)$, at SHA-256 Hash, to assure the Client that the message M4 is new and fresh, because it was just generated by himself in M3.

Without this hash an attacker, who has become aware of a momentary session key K_{CS} , would also be aware of all fields except the client password Pwd_C of the M2 message. If the RSA encryption algorithm is not secure, since that the password space is relatively small, an attacker could initiate an offline attack on the password by encrypting messages with the public key of the server (public known) and check if the relative cipher text is the same of M2.

Ban Logic Analysis

Real Protocol

M1 $C \rightarrow S$: C_{ID}
 M2 $S \rightarrow C$: $C_{ID}, S_{ID}, Cert_S, N_S$
 M3 $C \rightarrow S$: $\{C_{ID}, S_{ID}, C_{Pwd}, K_{CS}, N_S, N_C\}_{PubK_S}$
 M4 $S \rightarrow C$: $\{C_{ID}, S_{ID}, H(N_C)\}_{K_{CS}}$

Idealized Protocol

M3 $C \rightarrow S$: $\{<N_C, C \xleftrightarrow{K_{CS}} S, N_S>_{C_{Pwd}}, C_{ID}, S_{ID}\}_{PubK_S}$
 M4 $S \rightarrow C$: $\{H(N_C), C \xleftrightarrow{K_{CS}} S\}_{K_{CS}}$

Hypothesis

Key Assumption

- $C \models (C \xleftrightarrow{K_{CS}} S)$
- $C \models (C \xrightarrow{PubK_S} S)$
- $S \models (C \xleftarrow{C_{Pwd}} S)$

Freshness

- $C \models \#(N_C)$
- $S \models \#(N_S)$

Trust

- $S \models C \Rightarrow K_{CS}$

Objectives

- 1 $C \models (C \xleftrightarrow{K_{CS}} S)$
- 2 $S \models (C \xleftrightarrow{K_{CS}} S)$
- 3 $C \models S \models (C \xleftrightarrow{K_{CS}} S)$
- 4 $S \models C \models (C \xleftrightarrow{K_{CS}} S)$

Where 1 and 2 provide *key authentication* and 3 and 4 *key confirmation*.

An important observation to do is relative the public key and certificate. Since ban logic does not say nothing about the time validity of the certificate, the assumption to do is that the user before starting the protocol verifies the validity of the certificate.

Under this, we can say that Client C believes that Pub_{K_s} is the actual public key of the server.

Postulates application

After the decryption of message M3, the server sees: $S \triangleleft \langle N_C, C \xrightarrow{K_{CS}} S, N_S \rangle_{C_{Pwd}}$, so:

$$S \models (C \xrightarrow{C_{Pwd}} S), S \triangleleft \langle N_C, C \xrightarrow{K_{CS}} S, N_S \rangle_{C_{Pwd}}$$

applying the first postulate:

$$S \models C \sim (N_C, C \xrightarrow{K_{CS}} S, N_S)$$

since that, $S \models \#(N_S)$, by applying the second postulate:

$$S \models C \models C \xrightarrow{K_{CS}} S$$

and finally, by the assumption $S \models C \Rightarrow K_{CS}$ and the application of the third postulate, we finally obtain:

$$S \models C \xrightarrow{K_{CS}} S$$

After M4, the client sees: $S \triangleleft \{H(N_C), C \xrightarrow{K_{CS}} S\}_{K_{CS}}$, so:

$$C \models (C \xrightarrow{K_{CS}} S), S \triangleleft \{H(N_C), C \xrightarrow{K_{CS}} S\}_{K_{CS}}$$

applying the first postulate:

$$C \models S \sim (H(N_C), C \xrightarrow{K_{CS}} S)$$

observing that $C \triangleleft N_C$, we get:

$$C \models S \sim (N_C, C \xrightarrow{K_{CS}} S)$$

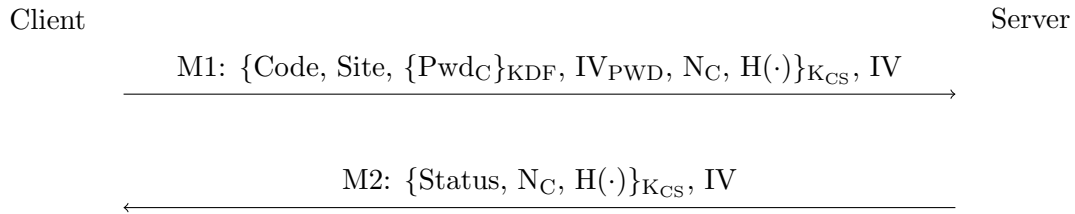
knowing $C \models \#(N_C)$, applying the second postulate we obtain:

$$C \models S \models (N_C, C \xrightarrow{K_{CS}} S)$$

Requests protocol

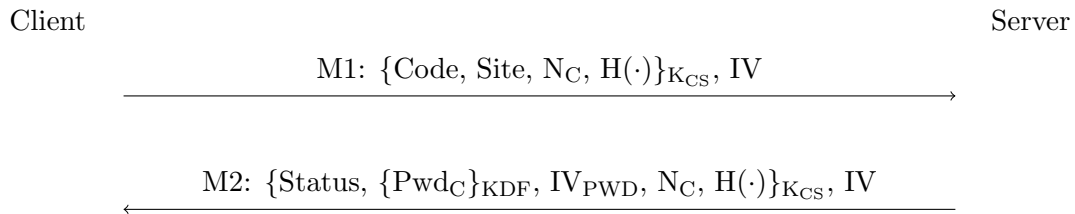
In the first message the Client sends an opcode related to the type of operation he wants to perform:

Add



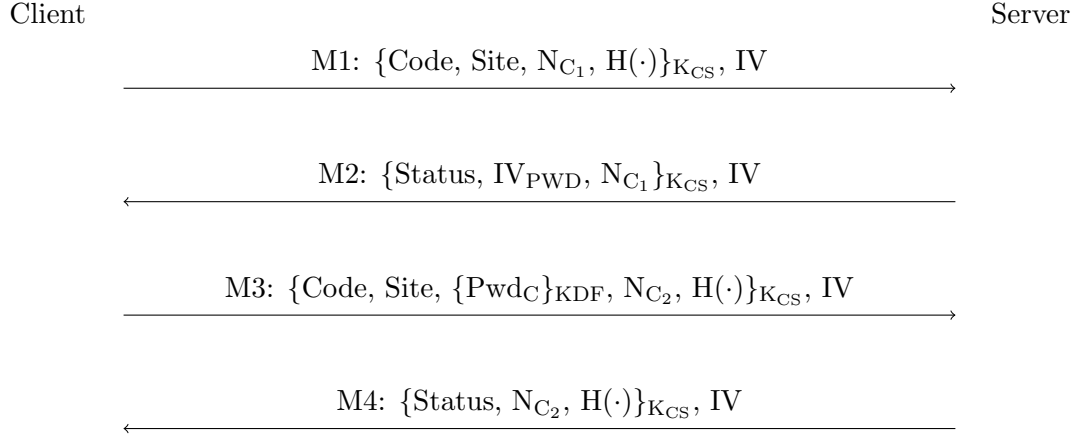
- M1: the Client sends the opcode *ADD* to the Server, then the *website* he wants to insert in the database, and the password $\{Pw_d_C\}_{KDF}$ he has chosen for it, in addition to the IV_{PWD} he will need to decrypt it, then a nonce N_C and the hash $H(\cdot)$ of the entire message
- M2: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not, the nonce N_C received before and the hash $H(\cdot)$ of the entire message

Get



- M1: the Client sends the opcode *GET* to the Server, the *website* of which he wants to obtain the password, a nonce N_C and the hash $H(\cdot)$ of the entire message
- M2: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not, the password $\{Pw_d_C\}_{KDF}$ he has chosen for the *website*, the Initialization Vector IV_{PWD} needed to decrypt it, then the nonce N_C received before and the hash $H(\cdot)$ of the entire message

Delete



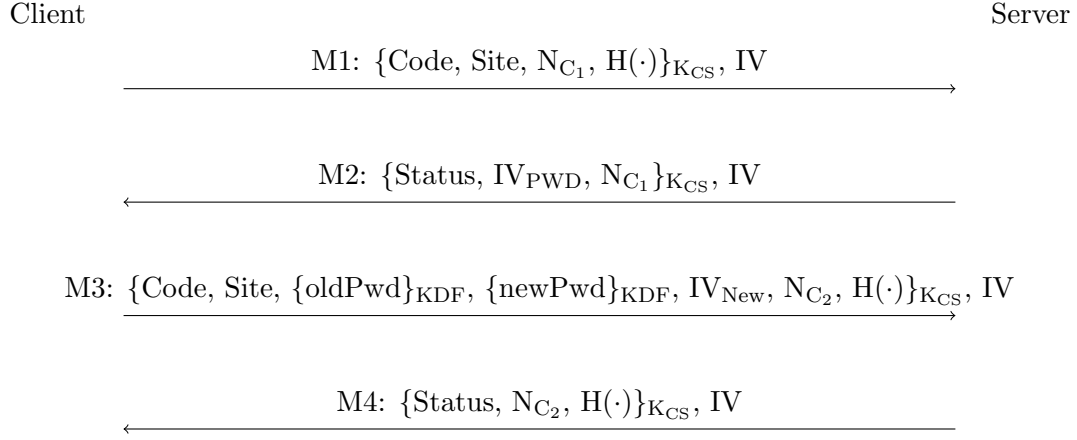
- M1: the Client sends the special opcode IV to the Server, the *website* of which he wants to delete the password, a nonce N_{C_1} and the hash $H(\cdot)$ of the entire message
- M2: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not, the Initialization Vector IV_{PWD} needed to the Client to encrypt the password with the right IV, in that way the Server can check if the stored one is equal to it, the nonce N_{C_1} received before and the hash $H(\cdot)$ of the entire message
- M3: the Client sends the special opcode UPD to the Server, the *website* of which he wants to delete the password, the current $\{Pwd\}_{K_{DF}}$, a nonce N_{C_2} and the hash $H(\cdot)$ of the entire message
- M4: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not and the nonce N_{C_2} received before, then the hash $H(\cdot)$ of the entire message

Observation

The use of the nonces in this case is to prevent 'non voluti' replay attacks. We could imagine the client that first adds a password for a site and then he remove it from the DB. An adversary could resend the first message and add again the password on the DB.

So the server before perform the operation checks if the nonces received is already present in a session nonces list.

Update



- M1: the Client sends the special opcode IV to the Server, the *website* of which he wants to update the password, a nonce N_{C_1} and the hash $H(\cdot)$ of the entire message
- M2: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not, the Initialization Vector IV_{PWD} needed to the Client to encrypt the old password with the right IV, in that way the Server can check if the stored one is equal to it, the nonce N_{C_1} received before and the hash $H(\cdot)$ of the entire message
- M3: the Client sends the special opcode UPD to the Server, the *website* of which he wants to update the password, the old $\{\text{oldPwd}\}_{KDF}$ and the new password $\{\text{newPwd}\}_{KDF}$, with the correspondent IV_{New} , a nonce N_{C_2} and the hash $H(\cdot)$ of the entire message
- M4: the Server replies with a *Status* parameter that defines if the previous operation succeeded or not and the nonce N_{C_2} received before, then the hash $H(\cdot)$ of the entire message

Implementation

Client

The Client was implemented by using Python 2.7 and some graphical libraries as *Tkinter*, and *Cryptography* for the secure primitives.

The Homepage offered to the user is shown in Figure 2.

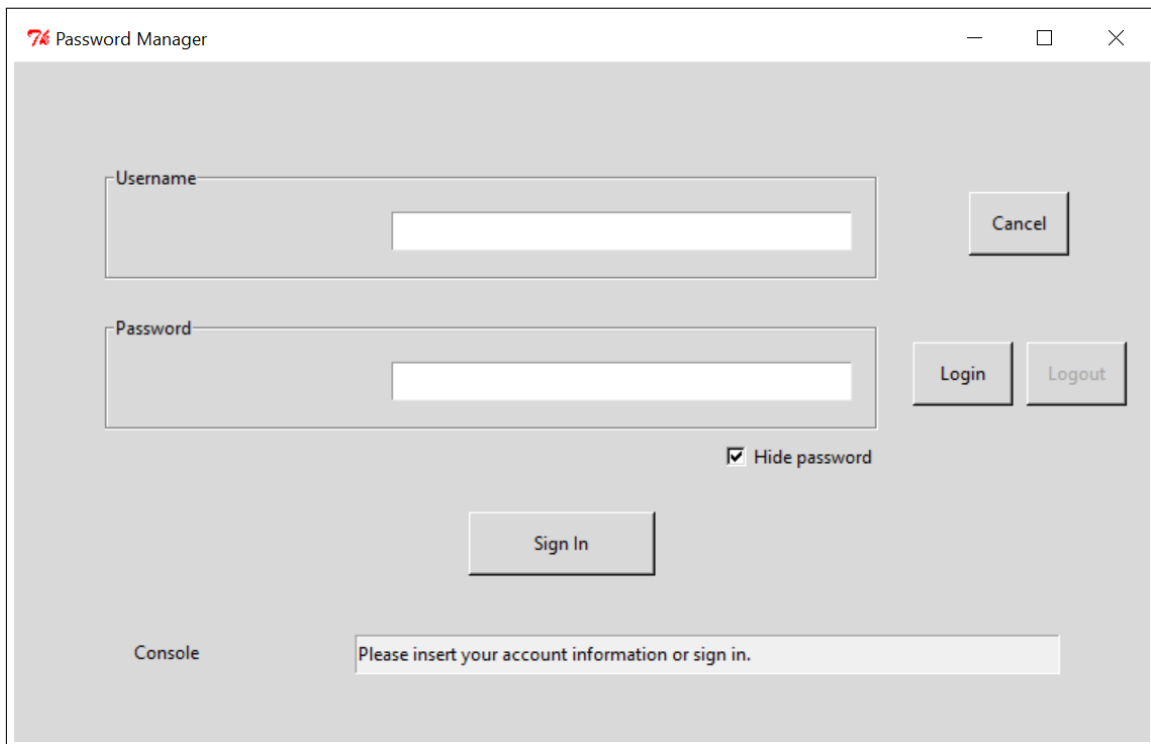
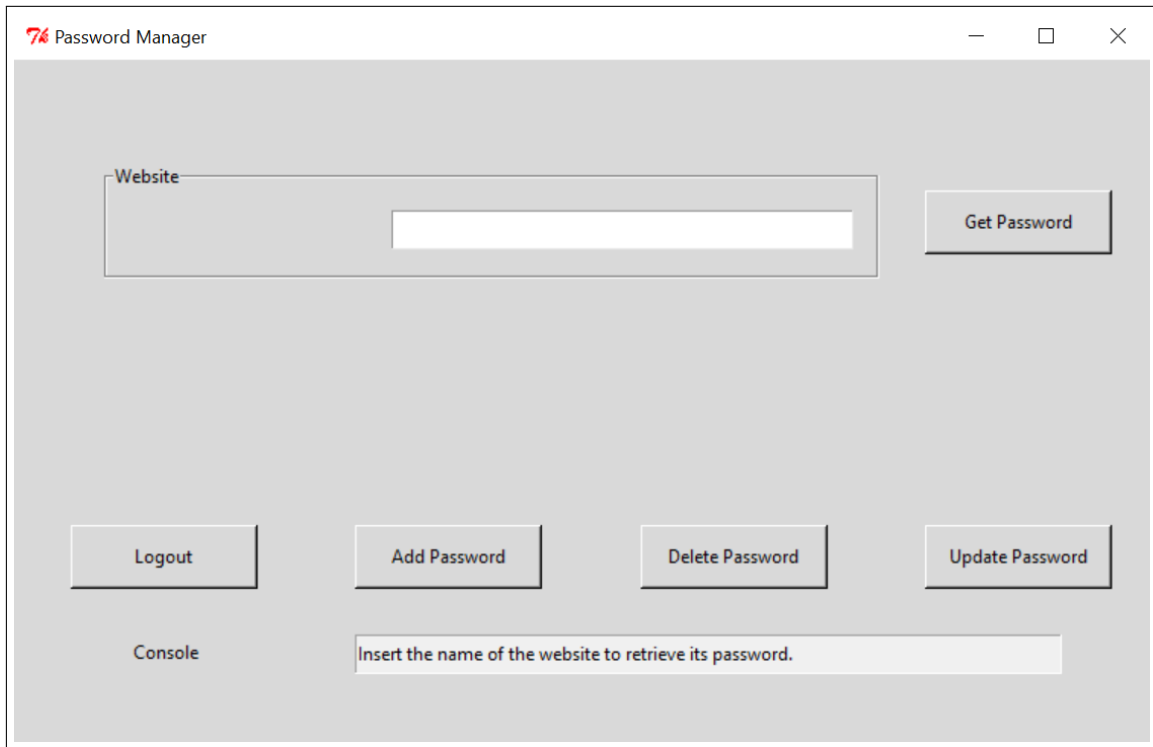


Figure 2: Homepage

The user can enter his personal information here to access the service. In addition, there is another page required to register.

Pressing the access button will initiate the Key Establishment Protocol described in section *****. In the event that the system does not detect errors, the Client is allowed to start using the application and to interact with the Server.

The user can now request for some password stored by the Server, as shown in the Figure 3.



The screenshot shows a window titled "Password Manager" with standard window controls (minimize, maximize, close). The main interface is a light gray panel. At the top, there is a "Website" label followed by a text input field and a "Get Password" button. Below this, there is a row of four buttons: "Logout", "Add Password", "Delete Password", and "Update Password". At the bottom, there is a "Console" label followed by a text input field containing the placeholder text "Insert the name of the website to retrieve its password."

Figure 3: GetPassword page

Using this form to request a password, or to delete, update or add one, the Client app will initiate the Request Protocol with the Server.

The Console Log will prompt the user the status of the system and the operations he performed.

After the push of Logout button, the Client application automatically deletes the information in memory about the user, such as the K_{CS} and the KDF .

Server

The server was implemented by using Python 2.7. It print on the terminal all the operations done by the user.

Two implement it we used *SocketServer* a library that provides a multithread architecture that starts a thread at every connection.

So the server is listening for users, when it gets a connection, the **key establishment protocol** starts. After sent the own certificate, the server decrypts the user requests; there are two types of requests:

- **Login request**

in this request there is the user's password and the server has to check the correctness. For security reasons, the server does not store the password, but the *hash* of it. To avoid rainbow table attacks, a salt is used, too. (Figure.4)

ID	hash_password	salt
alessandro	67c31b2f811e0b1c4a4b6f7c13513e2ec6	6dd6bd72b161d54f475a9784e2173f94
fabio	79e310d4887ddec8a1f63ba9a29c854615	6dade18c1a9e0cfb497fef2b83cde861

Figure 4: Users' table

The hash function used is a SHA256 and the salt is a random number of 16 Bytes. If the hash are the same then, the user has corrected logged in and the session key K_{cs} is established.

- **Sign in requests** This request refers to a new user that wants to use the service. In this case the server check if the Client ID is already used and if the response is negative proceeds to the registration by creating a new table for that user.

If the protocol finishes correctly the server starts to wait for operation requests like: add, remove, change and get passwords for a site.

For every request the server makes a query to the user's table named as *client_ID_table*. The password stored is encrypted with a symmetric key using a *cbc* cipher by the user so the *Initialization Vector* is also stored.

The structure of the user's table is shown in figure 5

sites	password	IV
amazon	527bd2f364b478824e5075642de89125	2023d546b91d2105a6eaa77ac211645d
ebay	c6da84b329c5c451ec0d2af4c1aeeb0e	10cd1587bf53fb58ced75194e2787efe
facebook	63d7ff75dfae6b9a833915c7fa66a024	a3489dd0140cec69269a5e96f2deabd7

Figure 5: User sites table

If the query returns a successful results, then code 100 is sent back, else the server sends the error code that will be managed by the client. Obviously if the user asks for a password, with the code 100 is sent even the password.

Certification Authority

In order to use asymmetric encryption we need to securely link the public key to the owner in order avoid attacks such as man in the middle (MiTM). To do this we used certificates. For academic purposes we set up an OpenSSL CA trusted by all clients that signs server's public key. In particular we created a server certificate with the *distinguished name* shown in figure 6.

$/C = IT/ST = Pisa/L = Pisa/O = Unipi/OU = CyberSecurity/CN = RPM_server$

Figure 6: Server Certificate: distinguished name

Threat Model

When we have to develop a secure system, we have to evaluate what kind of attacks an adversary can do.

Server Point of View

Since that the passwords on the database are encrypted by a 32 Bytes key and only the client knows is, an adversary that breaks the server securities can only see random bytes, therefore she is not able to retrieve any password.

This solution is valid even in the case, the server, for same reason became malicious and wants to steal the password.

SQL Injection

An other attack that could an adversary do is an SQL injection to the database. We avoided this kind of threat by using prepared statement in order to perform queries.

Password length

In the extreme case in which an attacker becomes aware of both the salt used by the Server to randomize the hash of the Client's password, and this hash, both saved in the database from the Server, we must ensure that the system still remains secure.

For this reason we advise the user to choose a password at least 10 bytes long. In this way, the attacker, while knowing this data, should save a rainbow table of the size of 2 to 80 (number of entries in the table)*26 bytes.

Another problem for the attacker is related to the necessary time. Due to the introduction of 16 bytes salt, the attacker can only generate the rainbow table as a result of its penetration into the Server database, to check if the stolen hash is contained in it.

In a real context we can assume that the Server becomes aware of the intrusion in a suitable time for which the attacker does not have time to generate all the possibilities.

Conclusion