Applied cryptography

Secure Bank Application

Adrien M.

June 2023

Contents

1 General introduction

The aim of the project was to create a "Secure Bank Application" (SBA): a client-server application that allows users to issue operations on their own bank account. Where each user is modeled by a username and a password whereas each bank account by an accountID and a balance. For simplicity we assume that a user owns a single account and that an account is owned by a single owner. The project was realized in c/cpp using **TCP/IP sockets** to create an online connexion.

1.1 Start

The directory containing the project is composed of 9 programming files and 3 directories: The directory *bank*, we assume it is private and only accessible by the bank. It is where the bank's keys are stored. The directory *clients*, divided into two sub-directories, *accounts* where the encrypted accounts' file of the clients are stored and the sub-directory *passwords*. In this directory the bank stores the **accountID**, the **public keys** and the **passwords and the salt** of each client. To start the application, you just need to open a terminal, go to the global directory, where all the files are located, and enter in the terminal **make clean** and then **make -s**. Then the application starts and asks the user to choose between **logging in (L)** or to **enroll (E)**.

1.2 Enrollment

Once the user has pressed "E" to enroll in the bank, he is asked for a **username** and a **password**. The enrollment is performed off-line (as if the client went to the bank office to create his account from the bank's computers).

1.2.1 Username and password verification

The username will be rejected if it is already used by another user (as we assumed that a single user owns only one bank account) and the password will be rejected if it is less than 5 characters, for security reasons. (Please take under consideration that username and password should not contain spaces).

1.2.2 AccountID generation

Then if username and password are accepted, the algorithm generate a random accountID for the user. The accountID is a 10 digits number between 10000000000 and 18999999999. The accountID is stored in the file *clients/passwords/userID.txt*.

1.2.3 Account file creation

Then the account file is created: It contains the **balance** of the client and his **history** (his last transactions): *clients/accounts/accountID.txt.enc*, where accountID is the actual ID generated for the client and the .enc suffix means the file is stored encrypted. For each new client the bank grants 1000€ on the account, but it does not count as a transaction.

1.2.4 Keys generation and storage

A pair of keys is then generated and stored in the directory *users/username*, where the username is the actual username of the client. We assume that this directory is "fictitious" and exists only for the simplicity of use for the exercise: a given client can access only his user directory. In the same directory the bank's public key is copied (*users/username/public_key_bank.txt*) and also a copy of the client's public key is stored at *clients/passwords/keys/username.txt*, so that the bank can have the public key of each client.

1.3 Login

Then once the client is registered he can access his account. So he need to launch again the application (make clean, make -s) and now the client can log in by pressing "L". In this section we will not talk about the authentication protocol, for more details go to section 3.

1.3.1 Functions available

Once the user is authenticated, he can access a menu, where he has 4 choices: (1) **check his balance**, (2) **make a transfer**, (3) **check his history**, (4) **leave the application**.

1.3.1.1 Balance: If the user presses "1", the bank sends the **accountID** and the current **balance**.

1.3.1.2 Transfer: If the user presses "2", the bank asks to choose a beneficiary and an amount of money to send to this beneficiary. If the transfer is feasible, i.e if the beneficiary exists and the amount is not higher than the current balance of the client, the transfer is made and the bank sends **Success!** otherwise it sends **Fail.** If the transfer is effectively done, history and balance of both client and beneficiary are updated.

1.3.1.3 History: If the user presses ”3”, the bank sends the **list of the transactions**, the list of either received or send money. It has the following shape:

- In case of debit (cash in): sender’s username – +amount – date.
- In case of credit (cash out): receiver’s username – -amount – date.

If the client has not been part of any transaction yet, the bank sends ”No transaction yet.”.

1.3.1.4 Leave the application: If the user presses ”4”, the client is **disconnected** from the application.

1.4 Test

There are already two clients registered, with the following usernames and passwords: Adrien – MYX123 and Benoit – MYX123. These accounts can be used as tests or new ones can be created to try all the functionalities of the application. (We can notice and verify in the file *clients/passwords/passwords.txt*, that even if the passwords are the same, the stored passwords are not the same due to the use of salt)

2 Long term private and public keys

The bank and each client own a pair of keys.

2.1 ECC keys

I chose to use Elliptic-curve cryptography (ECC). The curve chosen is the P-256 curve. This elliptic curve uses the finite prime body of size 256 bits and is defined in the ANSI X9.62 standard (in the code: $NID\_X9\_62\_prime256v1$). It is widely used and considered secure in many applications.

2.2 Comparison with RSA keys

I decided to use ECC (Elliptic Curve Cryptography) keys over RSA (Rivest-Shamir-Adleman) for many reasons:

- Key size: ECC keys are **much shorter** than RSA keys for a **similar level of security**. For example, a 256-bit ECC key provides a comparable level of security to a 2048-bit RSA key. This means that encryption, decryption, and **key generation operations are faster** with ECC as they require fewer computational resources. And thus ECC keys **tend to require less storage space**. In fact, a 2048-bit RSA key means that the prime numbers used in the key generation process are on the order of 2^{2048}.

Therefore, longer RSA keys require more storage space. And for the bank application, for each new client in total 2 keys are created and 2 keys are copied. So each client requires to store 4 keys and thus storage could be an issue for a big number of clients.

- Resistance to cryptographic attacks: ECC algorithms are considered more resistant to certain cryptographic attacks, particularly those based on integer factorization used to break RSA. The mathematical structure of elliptic curve cryptography makes ECC keys less vulnerable to these attacks.

3 Authentication protocol

The aim of this protocol is to **verify the identity of the client and the server** and to **initiate a secure channel**.

3.1 Generation of Diffie-Hellman (DH) ephemeral keys

To have a secure channel, the client and the server must have a shared secret, the **session key**. The session key is renewed at each connexion. To agree on a secret key without any previous shared secret before it is common to use the DH protocol: First of all, g and p parameters are fixed and common to both server and client. p, the modulo is a randomly generated large prime number (480 bits). g, a generator of the subgroup of the set of integers modulo p, is fixed at 2. This is a classic value, as it's a simple and efficient generator. These values are public parameters and do not constitute a previous shared secret.
Then the client randomly chooses an integer b and computes $Y_B \equiv g^b \pmod{p}$. Similarly, the server randomly chooses an integer a and computes $Y_A \equiv g^a \pmod{p}$. Then they exchange Y_A and Y_B and they compute the **session key**: K_{CS}, where $K_{CS} \equiv {Y_B}^a \pmod{p}$ for the server and $K_{CS} \equiv {Y_A}^b \pmod{p}$ for the client (which is the same).

3.2 Authentication of the server

In order to certify that the client is in communication with the bank, the bank must be authenticated. To authenticate the server, I have chosen to use digital signatures. Each client owns the ECC public key of the bank associated with the private key. In addition to sending Y_A to the client, the bank sends the encrypted digital signature of Y_B concatenated with Y_A, denoted as $\{< Y_B || Y_A >_{priv_{Kb}}\}_{K_{CS}}$, signed with its private key using the Elliptic Curve Digital Signature Algorithm (ECDSA) and encrypted with the session key. Upon receiving $\{< Y_B || Y_A >_{priv_{Kb}}\}_{K_{CS}}$, the client verifies that the signature has been performed by the bank by using the bank's public key stored in *users/username/public_key_bank.txt*. If either the private key used to sign is not the bank's one, or the public key used to verify the signature is not the bank's public key, the communication is shut down. Otherwise, the signature is verified, and the client can be sure that he is in communication with the bank, and the protocol can continue.

3.3 Authentication of the client

Then the bank has to verify the identity of the client it is in communication with. So once the authenticity of the bank has been proved and the DH protocol has been performed to the point that both parties have the session key, the channel between the bank and the unknown client is secured. The client is then asked to send his username and his password encrypted by the mean of the session key. If the username is not known by the bank, the communication shuts down and the client is asked to enroll. If the username is known, user sends then his encrypted password to the bank, the bank decrypts it, concatenate it with the salt corresponding to the username (found in the file *clients/passwords/salts.txt*), hashes the data and search in the file *clients/passwords/passwords.txt* if this unique digest exists. If it exists, user is authenticated otherwise he has 2 other attempts. If user fails to authenticate within these 3 attempts, he is disconnected.

3.4 Respect of the perfect forward secrecy (PFS)

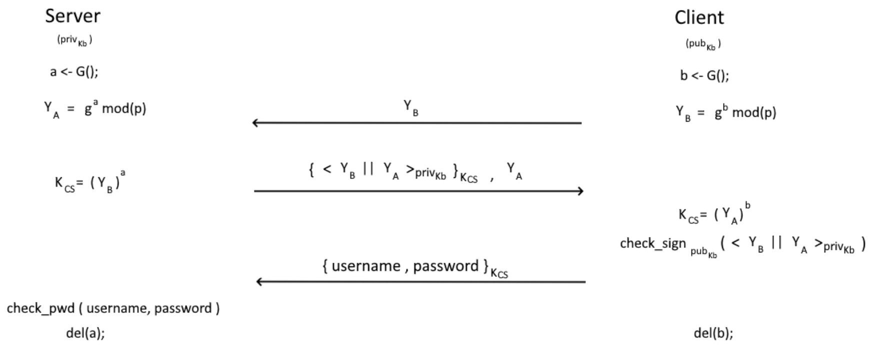
In this protocol, there is no previous shared secret between the client and the user, DH parameters a and b are ephemeral, computed "on the fly". The security issues regarding this protocol could be that either **bank's private key is compromised** and thus the client is fooled, he thinks he is talking to the bank whereas he is in communication with a malicious server. Or either the **password of a given user is compromised** and thus an adversary can look at the balance, the transactions of this user and even make transfers from this account. But in fact:

- If we assume that the keys are safely stored, the security of ECC relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP) and form a public key it is computationally infeasible to determine the corresponding private key. So the best-known algorithm to solve the ECDLP is through brute force.

- Passwords never travel in the clear as it is encrypted by the mean of the session key. Moreover, passwords are stored safely in the bank's directories (for more information about password storage go to section 4.1).

3.5 Sequence diagram of the authentication protocol

Here is the sequence diagram of the authentication protocol:


Server
(priv_Kb)
a <- G();
Y_A = g^a mod(p)
Client
(pub_Kb)
b <- G();
Y_B = g^b mod(p)
Y_B
K_CS = (Y_B)^a
{ < Y_B || Y_A >priv_Kb }_KCS , Y_A
K_CS = (Y_A)^b
check_sign pub_Kb (< Y_B || Y_A >priv_Kb)
{ username , password }_KCS
check_pwd (username, password)
del(a);
del(b);

4 Encryption of stored files

The bank should guarantee confidentiality on users' information. So even if an adversary obtains by any means the *clients/passwords/passwords.txt* file, he should not be able to read it. Similarly, if an adversary obtain by any means the account file of any user, he should not be able to read it.

4.1 SHA-256 with salt to store the passwords

In order to safely store all the passwords, I chose to use hash functions and especially **SHA-256**. Hash functions are **one-way functions**, so it means that it is easy to encrypt but almost impossible to decrypt. So for an adversary, if he obtains the file containing all the passwords, he would not be able to retrieve the passwords. Moreover I use salting: during the enrollment phase a randomly generated 128 bits salt is computed for each new user and stored in a different file than the passwords to make sure dictionary and rainbow attacks are infeasible. In order for the bank to verify the password matches the username, upon receiving it, it is concatenated with the salt, hashed and then the bank verifies the resulting tag exists in the passwords file.

4.2 AES-128 CBC to store the "account" files

To safely store the account files, I chose to encrypt the files with **AES-128 CBC**, by the means of the bank's private key. In fact, this time I did not choose hash functions to store the files because as they are one-way functions they do not provide **data retrievability** and it is an issue as the bank has to retrieve the history and the balance of a given user each time he asks. I could not choose either the ECB mode, where each block of data is encrypted independently, because it could have been easy for an adversary to create two accounts and analyze the data pattern of the accounts, that are similar.
I also chose the block cipher AES over DES or 3DES because AES is the standard: it is considered **more secure**, **faster** and **computationally more efficient**.

In conclusion, I chose AES-128 CBC because it is an easy way to store information in a file while **allowing easy and efficient retrieval** and providing **data confidentiality**.

5 Encryption of communications

The only thing left to do is to encrypt all the communications between the bank and the client.

5.1 AES-128 Galois/Counter Mode (GCM)

I chose to use **GCM** with the session key, as it is widely adopted for its performance. The GCM algorithm provides both data **integrity** and **confidentiality** but it also fulfills **no-replay** and **non-malleability**.

5.1.1 Confidentiality

GCM uses **symmetrical encryption, AES-128**. This means that data is securely encrypted, preventing any adversary from accessing the content of the data. For each encrypted message, a different initialization vector is randomly chosen, adding an extra layer of security to prevent the plaintext being obtained even if the same data is encrypted several times.

5.1.2 Integrity

Data integrity is guaranteed by the use of GCM, which incorporates message authentication in addition to encryption. In fact, GCM combines symmetrical encryption with message authentication. During encryption, a **message authentication code (MAC) is generated** and attached to the encrypted data. The MAC is calculated using the session key, the plaintext and the IV. During decryption, the MAC is checked for data corruption. If the MAC does not match, this indicates that the data has been altered, and decryption fails.

5.1.3 No-replay

Instead of using a randomly generated IV, I chose to **generate a nonce for each encryption**, to add an extra security measure: to prevent replication of encrypted data. By including a nonce, as it is unique, we ensure that each set of encrypted data is associated with a unique value at the beginning of the ciphertext. So it effectively prevents an adversary from reproducing a previous encryption with the same nonce.

5.1.4 Non-malleability

Thanks to **the use of the MAC**, any modification of the encrypted data, even a minor alteration, will be detected when the MAC is checked during decryption.

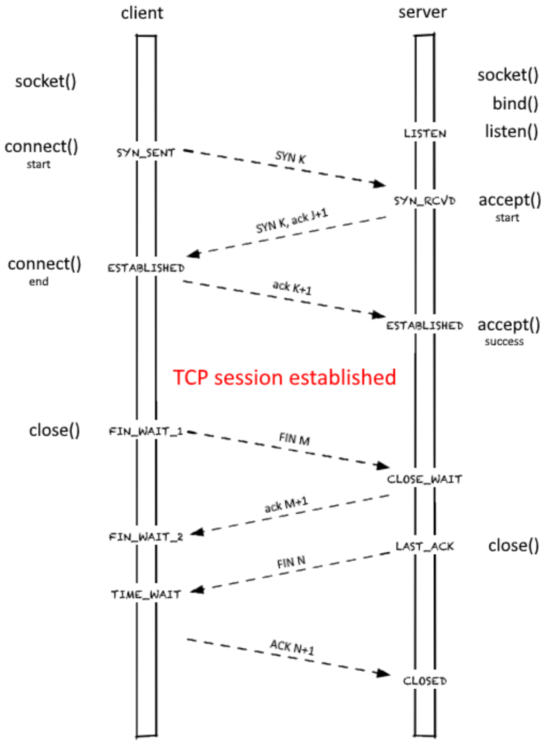
If the data has been manipulated, the MAC will not match and decryption will fail. This ensures data integrity and prevents unauthorized manipulation or alteration.

5.2 Comparison with AES-128 CBC

I chose this time GCM over CBC because CBC is a basic encryption mode that provides confidentiality but not integrity as it is malleable, while GCM offers both data confidentiality and integrity. To store the files I chose CBC because as the files are never moved, they are never transmitted in the network, so it is not necessary to verify their authenticity.

6 Sequence Diagram of TCP/IP

To create the SBA, I used the TCP/IP communication protocol and this is its sequence diagram at the application layer [1]:



[1] source: graffletopia.com/stencils/1560