UNIVERSITÀ DI PISA

Scuola di Ingegneria
Department of Information Engineering
MSc in Artificial Intelligence and Data Engineering

Foundation of Cybersecurity Project
Secure Bank

Adelmo Brunelli

Academic Year 2022/2023

Indice

Chapter 1

Introduction

Security is a fundamental aspect of a banking application, and implementing robust security measures is essential to protect users' sensitive data. OpenSSL is a common tool used to ensure security in communications. It can be used for data encryption, key generation, certificate management, and much more. In the context of a client-server banking application, OpenSSL can be employed to establish a secure channel between the client and the server. The challenge pertains to developing a secure banking application (*SBA*) that enables users to perform operations on their bank accounts in a client-server architecture. To maintain **Perfect Forward Secrecy** (*PFS*), which ensures

that the encryption of previous communications cannot be compromised even if the private keys are later disclosed, the following functionalities have been implemented in this project:

- **Use of robust cryptographic algorithms**: **AES** (*Advanced Encryption Standard*) has been employed, providing both strong security and high performance.
- **Secure key exchange**: To establish a secure connection, a mechanism is required for exchanging cryptographic keys between the client and the server. A common approach is to utilize the **Diffie-Hellman** (*DH*) protocol to generate a shared session key without revealing the private keys.
- **Adoption of digital certificates**: Digital certificates are used to authenticate the server's identity and can also be used for client authentication. OpenSSL supports the X.509 format for managing digital certificates and provides functions for certificate generation, signing, and verification.
- **Protection against Man-in-the-Middle (*MITM*) attacks**: To prevent *MITM attacks*, where an attacker intercepts or manipulates communications by inserting themselves between the client and the server, it is essential to verify the authenticity of the server's certificate. OpenSSL provides functions for certificate verification, including validating the certificate chain up to the root certification authority.

Chapter 2

Design Choiches

2.1 Connection

Firstly, a user needs to log in or register within the application. Then, a **TCP connection** is established: this protocol was chosen over UDP because it ensures a reliable connection. TCP incorporates *flow control*, *congestion control*, and *re-transmission* mechanisms to guarantee error-free and in-order data delivery. These features make TCP suitable for applications that require data reliability, such as file transfer. The connection remains active throughout the user's logged-in session.

2.2 Authentication

To ensure the *authenticity* and *integrity* of communications in a secure banking application, it is necessary to establish a secure channel between the client and the server. In this regard, **self-signed digital certificates** can be used. Self-signed certificates are generated and signed by the same server without involving an external certificate authority. This approach can be useful for development, testing, or simulating secure communication between components of a system.
To generate a self-signed certificate, OpenSSL provides a set of functions. A self-signed certificate can be created using the server's private key and adding information such as the server name, organization, expiration date, and other relevant details. Once the certificate is created, it can be used to authenticate the server's identity.
However, it is important to note that self-signed certificates are not reliable in production environments or situations that require high security. This is because self-signed certificates have not been verified by a recognized and trusted certificate authority. In real-world contexts, it is crucial to obtain certificates signed by an external certificate authority to ensure the authenticity and integrity of the certificates.
In the context of this project, server authentication is achieved by verifying the self-signed certificate. The client uses OpenSSL functions to verify the integrity and authenticity of the server's certificate. This involves verifying the digital signature of the certificate using the public key associated with the

certificate. If the verification is successful, the client can be confident about the server's identity.
Once the server's identity is verified, it sends its public key to the client. Similarly, the client sends its public key to the server. This exchange of public keys represents the first step in establishing a secure connection between the client and the server. The public keys will be used later for decrypting the data exchanged during communication.

2.3 Keys Exchange

After verifying the server's identity, the client and server proceed with the exchange of a symmetric key through the **Diffie-Hellman** protocol. The Diffie-Hellman protocol allows both parties to independently generate a shared secret without transmitting the key over the communication channel. In essence, the client and server exchange a series of messages containing public parameters and utilize their private secrets to generate a common value known as the shared secret.

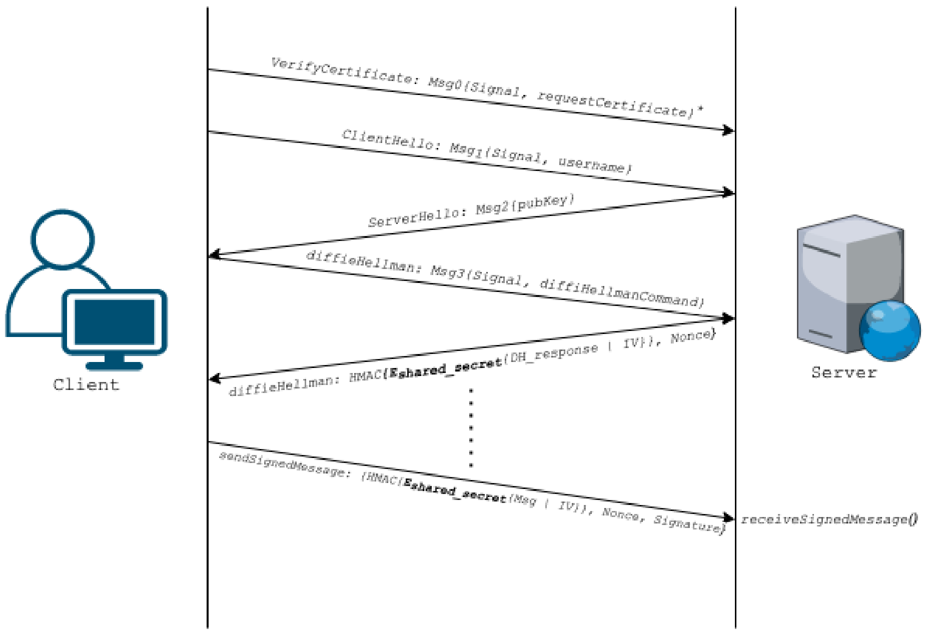
VerifyCertificate: Msg0(Signal, requestCertificate)*
ClientHello: Msg1(Signal, username)
ServerHello: Msg2(pubKey)
diffieHellman: Msg3(Signal, diffiHellmanCommand)
diffieHellman: HMAC(Eshared_secret(DH_response | IV)), Nonce)
sendSignedMessage: (HMAC(Eshared_secret(Msg | IV)), Nonce, Signature)
receiveSignedMessage()
Client
Server

Figure 2.1: Handshake between client and server

Once the shared secret is established, communications can be securely encrypted using symmetric algorithms like AES (Advanced Encryption Standard). However, simple data encryption alone is not sufficient to ensure complete system security. [1]

[1]*: The 'verifyCertificate()' function does not actually represent a message, but rather a local identity verification by the user. Ideally, it would have been preferable to obtain the certificate data from a Certificate Authority (CA) and then perform the verification. However, due to technical limitations, this option is not possible.

To properly handle cryptographic keys, it would have been ideal to use a *KeyStore*. A KeyStore is a secure entity that provides a protected environment for managing cryptographic keys. However, for illustrative purposes and to focus on secure communication, it was decided to store the initial shared secret in a file called *'key.txt'*. This file serves the purpose of enabling proper decryption of transactions and information stored on disk, which will be encrypted using the shared secret calculated in the initial communication.
After the initial shared secret is saved, the user data just entered is encrypted and saved on both the server and client sides. The use of the shared secret ensures the confidentiality and integrity of the data during its storage on disk.
The approach taken in the project to store the initial shared secret in a file was chosen for illustrative purposes only, to demonstrate the basic principles of encryption and to simplify the development process. In real-world environments, it is crucial to adopt appropriate security practices, such as using a KeyStore, to ensure the protection of cryptographic keys and sensitive data.

2.4 Bootstrap

At startup, each user will have a predefined folder structure. Within this structure, we will find various encrypted files and folders containing sensitive information.
In particular, the *'info.txt'* file will store the user's private information, encrypted to ensure its confidentiality. The *'key.txt'* file will contain the initially generated key for encrypting the data. Additionally, separate files will be present for the user's public key and private key.
All of these files and folders, including transaction files, will be stored in a shared folder between the user and the server. This folder has been specifically designed to be secure and impervious to external or unauthorized access. It ensures that only the user and the server have access to the files and information within.
The sharing of this folder between the user and the server is crucial to enable the proper functioning of the system. Through this sharing, the server can access the user's encrypted files to process transactions, update information, and provide the requested services to the user. Simultaneously, the user can send encrypted data to the server to ensure the confidentiality and integrity of their information.

Chapter 3

Messages Format

Now we can examine the communication method used for exchanging messages and notifications between the client and the server. In this context, a message called *'SIGNAL'* is always sent, which informs the server about the nature of the request the client is about to make.

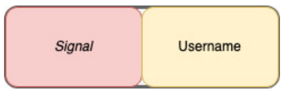

Figure 3.1: Structure of the message sent by the client to initiate communication

The client sends signals to the server using the *'encryptMessage'* encryption function. This function generates a random **Initialization Vector** (*IV*) and a **random nonce**, and encrypts the message text. The IV is generated using a cryptographically secure pseudorandom number generator (*CSPRNG*), which relies on cryptographic algorithms, making the generated random numbers extremely difficult to predict or reconstruct. The use of the IV is crucial to ensuring **data confidentiality**, as it allows for generating a different sequence of cipher bits for each message, even if the message itself is the same.
Subsequently, an **HMAC** (*Hash-based Message Authentication Code*) is calculated for the encrypted message. The HMAC is computed by taking the encrypted message as input and using the *HMAC-SHA256* function, which employs the SHA-256 hash algorithm to produce a 256-bit (32-byte) hash. The use of HMAC ensures **data integrity**, as any modifications made to the message would be detected by the server during HMAC verification.
Furthermore, the system also utilizes a **random nonce**, which is generated during the message encryption phase. The nonce is a unique and unpredictable value that is used to prevent **replay attacks**. Each encrypted message is associated with a different nonce, and the server verifies that the nonce has not been used previously to prevent an attacker from **replaying** a previous message.
Through the use of **digital signatures** and HMAC, along with the random IV and nonce, the system guarantees confidentiality (data confidentiality), integrity (data integrity), no-replay (prevention of replay attacks), and non-malleability (prevention of data manipulation) during the communication

between the client and the server.
All the messages in a session are encrypted with the session key and authenticated using the authentication encryption. In particular, all the messages are been encrypted using AES with a key length of 128 bits.

$$calculate\_hmac(data, data\_len, key, key\_len, hmac)$$

The encrypted message, along with the random IV, nonce, and HMAC, is passed to the digital signature generation function. The digital signature is a cryptographic mechanism that provides authentication and data integrity verification during communication. It is generated using a digital signature algorithm such as RSA.
In the context of communication between the client and the server, the digital signature plays a fundamental role. It is used to ensure that the message indeed originates from the stated sender and has not been altered during transfer. In other words, the digital signature allows the server to verify the **authenticity** of the message and its **integrity**.
The verification of a digital signature is performed using the public key associated with the sender. The server possesses the client's public key, which has been previously exchanged or securely shared. The verification process involves computing the hash of the received message using the same hash algorithm employed to generate the digital signature. Subsequently, the public key is used to decrypt the digital signature and obtain the original hash. If the computed hash matches the original hash, the signature is considered valid, and the message is accepted as authentic and unaltered.
In summary, the digital signature provides a secure mechanism for authenticating and verifying the integrity of messages exchanged between the client and the server. It allows the server to confirm the authenticity of the sender and detect any data manipulations during communication.

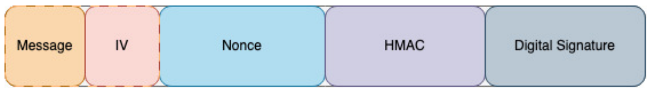

Figure 3.2: Structure of the message exchanged between client and server