Computer security

Assignment 2

Océane Guennec, Camille Kerhervé, Malaury Auberge

1 - Introduction

The objective of this project is to design a secure file transfer system that ensures the confidentiality of data transmitted between a client and a server. To achieve this, the system leverages cryptographic primitives such as RSA, ECDHE, and AES to prevent any exploitable interception by an attacker. The client and server first establish a secure session key using asymmetric encryption, followed by a handshake to negotiate the encryption algorithm. Files are stored in plaintext on the server but are encrypted before transmission using symmetric encryption. The client receives the encrypted file, decrypts it, and saves it in plaintext on disk. The entire communication is based on network sockets, ensuring a strict separation between client and server processes. This project provides an opportunity to implement fundamental applied cryptography concepts and assess their effectiveness in a secure file transfer scenario.
The attack model corresponding is man-in-the-middle attack. This kind of attack occurs when an attacker intercepts and can alter communication between two parties without their knowledge.

2 - Methods

Some parts of the implementation are interesting and important. Here are some examples below.

The *receive_encrypted_file* function in the **ft_client** class:

- The purpose of this function is to receive an encrypted file from the server and decrypt it using AES in GCM mode.
- The client receives a nonce and a tag of 16 bytes each. The nonce guarantees the uniqueness of the encryption and the tag is used to verify the integrity and authenticity of the data during decryption.
- The client reads 4 bytes representing the size of the encrypted file. It permits to know how much data to receive, avoiding incomplete or excessive readings.
- The data is read in 4096-byte chunks until it reaches the expected size, ensuring complete reception without overloading memory.
- Using the AES key and the nonce, the client decrypts the data and checks the tag. If the tag fails, the data is corrupted; otherwise, the original text is recovered.
- The decrypted content is saved as OUTPUT_FILE, providing the user with the original, unaltered version of the transmitted file.

The *encrypt_and_send_*file function in the **ft_server** class:

- The purpose of this function is to encrypt a file using AES in GCM mode and send it to the client.
- The server loads its RSA private key from a file to decrypt the data sent by the client, including the encrypted AES key used for secure file transmission.
- The server creates a TCP socket, binds it to the specified address and listens for incoming connections. This makes it possible to manage several clients waiting for a connection on the defined port.
- As soon as a client connects, the server receives its RSA public key, generates a random AES key, encrypts it with the client's key and sends it to the client. Once the key has been exchanged, the server encrypts the file to be transmitted and sends it to the client.
- The server creates a 32-byte AES key for symmetric encryption. This key is then encrypted with the client's public key using RSA-OAEP and sent, ensuring that only the client can decrypt it.
- The server reads the file, encrypts it with AES in GCM mode to ensure confidentiality and integrity, then sends the nonce, the authenticity tag, the size of the encrypted data and the encrypted file in succession. This allows the client to decrypt and verify the file without corruption.

The server generates an RSA key pair and distributes its public key to the client during the initial exchange over the TCP connection. This allows the client to securely encrypt the symmetric AES key. For added security, the public key can be distributed beforehand via a trusted channel (e.g. a certification authority or secure manual transfer), thus avoiding potential man-in-the-middle attacks during the handshake.

The message format consists of two main phases: handshake and file transfer. During the handshake, the client retrieves the server's RSA public key, generates a random AES key, encrypts it with RSA, and sends it to the server, which decrypts it for secure symmetric encryption. In the file transfer phase, the client requests a file, and the server encrypts it using AES-GCM, sending the ciphertext, nonce, and authentication tag. The client decrypts the file using the shared AES key and verifies its integrity with the GCM tag. This structure ensures confidentiality, security, and data authenticity.

The implementation combines several cryptographic algorithms:

- RSA: Used to securely encrypt the AES key.
- AES-GCM (Advanced Encryption Standard-Gaulois Counter Mode): Provides authenticated encryption, ensuring both confidentiality and integrity of the file.

In our system file transfer system, we chose to use RSA for key exchange and AES –GCM for symmetric encryption. While ECDH and ChaCha20 are modern alternatives with strong security guarantees, we chose RSA and AES-GCM due to their simplicity, broad support, and ease of implementation. Also, the project does not require digital signature, RSA provides a simpler solution by directly encrypting the AES key for secure exchange.

The program's organization can be represented with a figure:

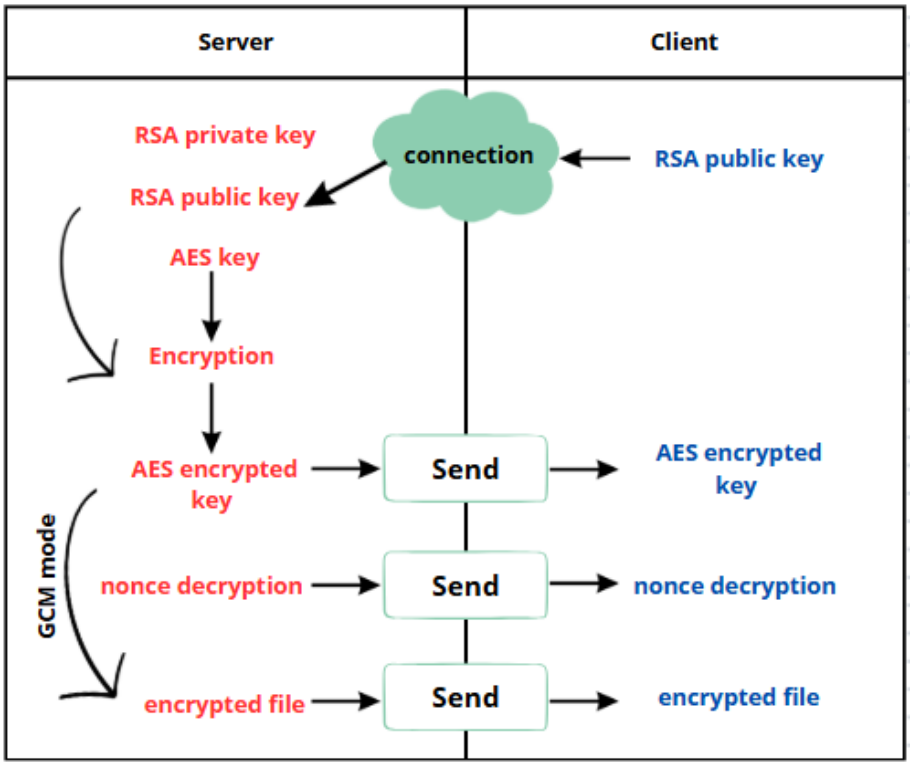

Figure 1 : Program organization

The implementation uses Python for its readability and rich library support:

- `Socket` library: Manages TCP communication.
- `PyCryptodome`: Provides cryptographic functions (RSA, AES).
- Secure random generators and error handling ensure robust encryption and prevent vulnerabilities.

3 - Results

The secure file-sharing system effectively integrates AES-GCM encryption, ensuring that files are both confidential and tamper-proof during transmission. By using RSA for securely exchanging the AES encryption keys, the system prevents unauthorized access and ensures that only the intended recipient can decrypt the files. A suite of automated tests, implemented using *unittest*, evaluates the system's functionality and security. These tests validate secure key exchange, encryption and decryption integrity, and file authenticity. Specific tests, such as verifying that the transferred file remains unaltered using SHA-256 hashing, confirm that data integrity is maintained. The test suite also includes scenarios for potential failures, such as issues with key exchange or corrupted transfers, ensuring that the system consistently meets security requirements.

Additionally, the AES-GCM encryption itself provides automatic verification of data integrity. During decryption, the client checks the authenticity of the data using the included authentication tag. If any tampering or modification occurs during transmission, the tag will fail, and the decryption process will not succeed. This built-in check ensures that corrupted files are detected and rejected, showcasing the system's robustness against potential attacks.

Regarding performance, cryptographic operations do introduce some delay in file transfers, though the overhead remains manageable for typical file sizes. Larger files result in slightly longer transmission times due to the complexity of the encryption process, but the system performs effectively within expected limits.

4 - Evaluation / Discussions

In this file-sharing system, RSA encrypts the symmetric AES key to prevent unauthorized access. In addition, AES-GCM guarantees data confidentiality and integrity, making intercepted data unreadable and detecting any modification. Together, these mechanisms protect against man-in-the-middle attacks by securing key exchange and verifying the authenticity of transmitted files.

This system currently ensures confidentiality using AES-GCM and this mode includes authentication which ensures confidentiality and integrity. The ciphertext and authentication tag are verified on the client side which prevent tampering. But there is no protection about metadata like RSA key exchange and handshake messages, also an attacker could re-send an old, encrypted file. For additional integrity we could use a hash function on the public key, like digital signature, that will ensure messages have not been altered. To prevent replay attacks we could add timestamps and nonces in messages, so the client sends a random nonce and the server signs it using its private key.
Currently, the client blindly accepts any RSA public key received from the server. A man-in-the-middle attacker could inject their own key by intercepting the connection. To prevent this happening and improve our code we could use a digital signature or pre-shared certificate. With digital signature, the server signs its public key with a pre-distributed trusted key and the client verifies the signature before proceeding.

We made choices during this assignment regarding performance, security, and implementation simplicity. For example, since our group is familiar with Java for software development, we initially hesitated to use the Bouncy Castle dependency, which is the equivalent of the PyCryptodome library in Python. It also allows the use of KeyGenerator and CipherText. However, even though we hadn't worked with Python for a long time, we wanted to follow the recommendations and requirements of this assignment while learning even more. In the end, the implementation wasn't too much of a challenge thanks to the documentation.
Next, we chose to use AES-GCM instead of AES-CBC (Cipher Block Chain) because it provides integrity protection, and AES is the fastest for file encryption.
Furthermore, to simplify the implementation and since it was not a requirement, our protocol does not verify the server's authenticity, making it easier to implement but less secure against man –in-the-middle attacks. The file size is sent before the encrypted data, ensuring that the client reads the exact amount but does not provide forward secrecy.

5 – Conclusion

This project aimed to develop a secure file transfer system that ensures data confidentiality, protecting against man-in-the-middle attacks. To achieve this, we implemented a cryptographic scheme using RSA for secure key exchange and AES-GCM for authenticated encryption.

Through extensive testing under different conditions, the implementation proved effective, with files being securely transmitted and decrypted without corruption. The use of AES-GCM provided built-in integrity checks, preventing unauthorized modifications, and RSA ensured that the encryption keys remained confidential.

However, we identified potential areas for improvement, particularly in verifying the authenticity of the server and protecting the integrity of the handshake messages. Solutions such as digital signatures, certificates, and nonce-based authentication could further enhance security, preventing attacks like key injection and replay attacks.

Overall, despite some possible improvements like digital signatures or certificates, the project has succeeded in securing file transfers using encryption and integrity checks, thus preventing data interception and falsification.

6 – References

[1]
"Welcome to PyCryptodome's documentation — PyCryptodome 3.21.0 documentation." Accessed: February 28, 2025. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/

[2]
"unittest — Unit testing framework," Python documentation. Accessed: February 28, 2025. [Online]. Available: https://docs.python.org/3/library/unittest.html

[3]
"Introduction to Computer Security: Pearson New International Edition." Accessed: February 28, 2025. [Online]. Available: https://viewer-ebscohost-com.mime.uit.no/EbscoViewerService/ebook?an=1418217&callbackUrl=https%3a%2f%2fresearch.ebsco.com&db=nlebk&format=EB&profId=ehost&lpid=&ppid=&lang=fi&location=https%3a%2f%2fresearch-ebsco-com.mime.uit.no%2fc%2fq3feil%2fsearch%2fdetails%2fjba24ekbin%3fdb%3dnlebk&isPLink=False&requestContext=&profileIdentifier=q3feil&recordId=jba24ekbin

[4]
Mablr, mablr/BPPA. (January 17, 2025). Accessed: February 28, 2025. [Online]. Available: https://github.com/mablr/BPPA