Cryptographic Banking System for MyBank

Author: Leen El Mir

Date: 03 – 03 - 2025

Contents

1. Introduction

This cryptosystem is designed for MyBank, a financial institution offering an online banking system to customers. The banking system grants customers access to sensitive financial operations such as managing their accounts and transferring funds. Therefore, this cryptosystem is built with confidentiality, integrity, and authenticity of information in mind.

The cryptographic features include a secure session established with the company's server, allowing clients to manage their finances. A strong user authentication process confirms the identity of clients accessing the system. A key management system was implemented for generating, storing and destroying cryptographic keys.

The system recognizes three roles for accounts: *user*, *administrator*, and *employee*. Role-based access control was implemented to grant each role its supported functions. For example, a user can monitor only their transaction history, while an employee can monitor all transactions.

2. Design Justification

The system design is divided into a client and server component communicating over a socket. The client handles all inputs from users, such as signing up, while the server handles all backend requests. The encryptions include:

2.1 Client storing data at rest: 256-bit KDF-derived key

Data stored on the client side was encrypted using a Key Derivation Function derived from the user's password. Specifically, PBKDF2, which is common for passwords, was used to generate a 256-bit encryption key (salted with a unique salt). A KDF was used to avoid storing the encryption key locally, reducing the risk of offline attacks. When the encryption key was needed, the KDF function was called to recompute it. The large key size was selected to make it harder for attackers to brute-force the key, given the high associated computational effort. The salt was added to reduce risk of brute-force attacks and rainbow table attacks.

2.2 Server storing data at rest: 256-bit AES-GCM Key

On the server, data at rest is protected using AES-GCM encryption to generate a master 256-bit AES key. This key was used to encrypt data, such as other encryption keys or client information. AES-GCM was selected because it is efficient and verifies integrity, preventing data tampering. It also ensures data stored on the server's database is authenticated, offering data confidentiality, authenticity, and integrity.

This AES master key is then encrypted by another 256-bit random AES-GCM key which is stored securely in a Hardware Security Module (HSM). Since this encrypting key can decrypt highly sensitive data, placing it in an HSM ensures data cannot be accessed even if the database is compromised. This second key was used to store the master key in the database, adding another layer of protection. Therefore, even if attackers access the encrypted master key, they cannot decrypt the database's content. Actually, an attacker would require both keys to decrypt the database, reducing attack risk. Again, the large key sizes were selected to reduce brute-force attacks, given the high associated computations.

Using AES in GCM mode was very convenient; the nonce used in the encryption guarantees each encryption is unique, even if plaintexts were identical. The mode's authentication tag ensured ciphertext is authentic and untampered with, protecting the database from data modifications or snooping.

2.3 Client-Server data exchange: TLS, KDF

The client and server communications were split into initiating the connection at sign up and performing data exchanges later. Asymmetric cryptography, as outlined by TLS 1.2, was used at sign up, and symmetric KDF-derived cryptography was used for communications after users were authenticated. This design was chosen to protect communications from eavesdropping and tampering.

2.3.1 Initiating Client-Server Connection (at Sign up)

The system was designed by the guidelines of TLS 1.2 (Transport Layer Security). TLS 1.2 suggests using RSA public-private key encryption to securely exchange a shared secret key (Jahan, 2021). This key is salted with random values from the client and server, then input into a KDF function to derive the symmetric encryption key on both sides locally. Switching from asymmetric to symmetric encryption reduces the overhead of executing costly asymmetric cryptographic operations. This two-step approach guarantees both the client and the server are authenticated and securely exchange data.

Initially, the client generates a 2048-bit RSA public-private pair. Client encrypts its private key with the pre-derived KDF function, and server secures its private key with its master key. To establish the connection, the client sends its public key and a random salt. Server responds with a digitally signed certificate containing its own public key, and a random salt. Using a digital certificate reduces the risk of impersonation attacks. The client then generates a random shared secret, encrypts it with the server's public key, signs it with the user's private key and sends it. The server decrypts the shared secret using its private key and verifies the secret is untampered with and correctly sourced from the user using the signature. Both derive the shared encryption key by applying the shared secret and

combined salts to a KDF, offering forward secrecy. This key is used to generate an AES-GCM session key for encrypted communication on the socket. These steps allow mutual authentication, eavesdropping prevention, and secure data transmission (reducing risk of Man-In-the-Middle attacks).

While the user is created on the server, the user's password is hashed using *bcrypt*. *Bcrypt* was used because of its computationally expensive function. Its slow hashing process makes it harder for attackers to brute force attacks. Its built-in salt, created for each password uniquely, increases password security.

2.3.2 Client-Server Communication (at login)

Before the user authenticates its credentials, the login request with the user's credentials is sent encrypted with the server's public key. This ensures only the server could decrypt the user's credentials. After the server authenticates the user using their password, MFA is implemented using OTP verification. The user receives an OTP on their email, adding a second layer of security in verifying their identity.

Now, both the client and server use the unique session key to communicate securely. Since each user has its own session key, attackers cannot gain access to all communications, even if one user was compromised.

3. Implementation

The system is divided into client and server, which communicate over a secure socket. All their communications are encrypted using a symmetric key. This session key is an AES-GCM encryption key stored with an IV.

3.1 Account Sign up

When a user signs up, it creates its own public and private key locally. The user stores its public key locally, unencrypted since it has no sensitive information. It stores its sensitive private key using the encrypted key derived from its password. The salt is stored along with the password to allow for decryption.

To initiate the connection, the user and the server exchange public keys and random salt values, as per TLS1.2. The server sends the user its public key in a digital certificate, signed by the server's private key. The user decrypts the certificate, extracts the public key, and verifies the signature using the server's public key. This ensures the user can authenticate the server's identity.

On sign up, the client sends the account information to the server. Before it is sent to the server, the client encrypts the data using the server's public key and signs it with its private key. The session key is not used because the client has not completed the TLS handshake, so no session key exists. The server decrypts the account information using its private key,

verifies the signature using the client's public key, and creates the account. This account is encrypted with the server's master key and stored on the server's database. The user's password is first hashed using *bcrypt* to ensure secrecy.

To complete the connection, the user sends a random shared secret to the server. The shared secret is encrypted with the server's public key and signed with the client's private key. The server decrypts the secret using its private key and verifies using the user's public key. Now, the server and the client use KDF to derive the shared AES-GCM session key. The function inputs are the shared secret as the keying material and client and server salts as salt. The user stores its session key (and IV), encrypted with the password-derived key derived earlier. The server stores its session key (and IV) encrypted with the 256-bit AES-GCM master key. The user and server can now use this symmetric session key to exchange data securely.

3.2 Account Login

On login, the user already possesses an assigned session key but cannot access it till the server authenticates the user. The login credentials are sent to the server encrypted with its public key. The server decrypts the credentials using its private key and returns a successful user login. The user now accesses its session key by decrypting it using the encryption key derived from the user's password using KDF. The salt, stored unencrypted, is also entered into the KDF to generate the encryption key. Now, the user and the server use this session key to communicate.

Since Role-based management was implemented, each user is granted function rights depending on its role. The functionalities include:

3.3 User Functions

A bank customer is allowed to apply for a loan, check account balance, pay bills, and set recurring payments. The implemented functionalities are performing a transaction, messaging the company, and getting the user's transaction history.

The implemented functions were encrypted similarly, so the transaction function is used as an example. To perform a transaction, the user inputs transaction details, encrypts it using the pre-derived AES session key and sends it to the server. After login, the server has already authenticated the user, so the server uses that user's session key to decrypt the information. The server then completes the request and returns the response to the user, encrypted with the session key.

3.4 Admin Functions

Administrators manage user roles, implement security measures, and handles cryptographic key management. Admins cannot access user information besides usernames and passwords, which are necessary for IT management. Like the user

functions, the admin's communications are encrypted with the shared session key when in transit to and from the server.

3.5 Employee Functions

Employees must respond to user messages, make client transactions, access/update client account information, and monitor transactions. Employees are granted access to all the user's information. When requesting access from the server, the server ensures user is an employee (using login credentials) and decrypts the received request using the session key. The response is encrypted with the session key which the employee decrypts. This ensures only servers and employees access the sensitive user data.

4. Testing

All encryption/decryption functions were tested on simple data to ensure their functionality. Then, functions that integrate these functions were tested.

4.1 Account Sign Up

Several accounts were created to ensure accounts were created correctly, on client and server side. The tested accounts include:

- Accounts with passwords that don't meet requirements, resulting in denied sign-up requests
- Accounts with varying roles (user, employee, admin), resulting in successful sign-up requests

To verify functionality, the user's generated public/private keys were checked to be stored properly on the client side. On the server, the user's stored information was decrypted and checked to be matching the sent data. The validity of the derived session keys was also checked on both parties by encrypting on one party and decrypting on the other.

4.2 Account Login

All accounts created during sign up were logged into to test functionality. All accounts successfully logged in and retrieved their session keys. The session keys on the client and server sides were compared.

The tested accounts include:

- Accounts with incorrect passwords, OTPs, or usernames, resulting in denied login requests
- Accounts with varying roles (*user, employee, admin*), resulting in successful login requests

To verify functionality, the user's credentials on the client side were matched with the user's decrypted credentials on the server side. The server's functionality to verify hashed passwords was also checked to ensure it worked.

4.3 User Functions

Each of the implemented user functions was tested individually using logged in users.

4.3.1 Transfer Money

To test the function, different receiver usernames, amounts, and user accounts were used to transfer money. The function communicates with the server using the session key, so the validity of the encrypted/decrypted parameters was also checked.

4.3.2 Get Transaction History

To test the function, different usernames were used to retrieve a user's transaction history. It was verified that the current user can only retrieve transactions that include them as sender or receiver. They cannot access other user's transactions.

4.3.3 Communicate with bank representative

This function encrypts its parameters before sending to the server. Therefore, it was verified that the message on the client matches the decrypted message on the server.

4.4 Employee Functions

Each of the implemented employee functions was tested individually using logged in employees.

4.4.1 Receive user messages

This function receives all the messages on the server, encrypted using the current employee's session key. Due to the sensitivity of the data, it was checked that only employees can request to receive all messages on the server. It was also tested that the received messages include all the messages, regardless of their type. The message content was also matched on both the client and server sides to ensure the messages are being stored correctly.

4.4.2 Receive user transactions

This function receives all the transactions stored on the server, encrypted using the current employee's session key. Like the previous function, the data in transit is sensitive. So, it was checked that only employees can request to receive all transactions on the server. To ensure transactions integrity and proper storage, the message content was matched on both client and server sides.

4.5 Admin Functions

Like the user and employee functions, the admin functions were checked to be working properly.

4.6 Logout Function

This function ensures the user's username, session key, and public key are removed from the server once the user logs out. This was checked by verifying all the aforementioned variables were cleared on the server side once a user logs out.

5. Evaluation

5.1 Meeting Requirements

The system meets the required level of defense. It implements Defense in Depth, whereby multiple layers of security protect the data stored. This includes using TLS 1.2 with RSA exchange to secure client-server authentication, relying on AES-GCM encryption to secure data in transit and using *bcrypt* hashing to secure passwords. The use of password-derived KDFs also reduces the number of keys stored locally, reducing the attack space too. The system also relies on the Least Privilege principle, which is implemented through role-based access control (RBAC). The use of RBAC ensures that users, employees, and admins have just enough access to complete their necessary functions. This limits the potential damage from accidents and reduces the malicious attack surface. In addition, the session key being derived from a combination of shared secrets and unique salts ensures each user's data is uniquely encrypted, reducing replay attack risks.

5.2 Limitations

The most critical limitation is the server's master key being used to encrypt all data in the database; if the encryption key stored on the HSM was exposed, the server's database will be revealed. However, this risk is low since the master key's encryption key is stored on a secure HSM. Additionally, the use of asymmetric key exchange, with the large RSA key size, causes great computational overhead during the TLS handshake. It slows down connections on resource-limited devices. Another issue is the client side's reliance on a user's password to perform KDF. This implies a user forgetting their password can no longer access their encrypted data, so recovering the account is practically impossible.

6. Conclusion

In conclusion, the designed system for MyBank is implemented with confidentiality, integrity, and authentication in mind. It incorporates several encryption and hashing techniques to ensure sensitive data is being handled securely. It also offers three account modes: *user, admin,* and *employee* to suit the system's specified requirements.

The challenges faced included determining how to store the keys securely, how to transfer data securely between the client and server, and decrypting received data. Implementing proper key management in a manner that balances security and computational efficiency.

The system could be improved by upgrading TLS1.2 to TLS1.3. TLS1.3 mitigates TLS 1.2's weaknesses, including the SHA-1 dependency, and ensures a more efficient handshakes (with forward secrecy). Another improvement is introducing multiple master keys for the server to reduce the risk of a single point of failure. Each server-client connection could be encrypted with its own unique master key, securing the system.

7. References

Jahan, Ishrat. (2021). Everything about TLS handshake. 10.13140/RG.2.2.22636.08324.