Università di Pisa

Department of Engineering

Secure Banking Software
Foundations of Cyber Security

Supervisor:
Prof. Gianluca Dini

Presenters:
Ali Mahdavi
Jaffar Ali

Academic Year:
2022-2023

Contents

Introduction

This Project intends to build a client-server application with built in security for exchanged messages. Using This application, both client and server must be able to authenticate each other, establish a session key via which they can have secure communication. the following operations have been already implemented and tested.

- login to server via username and password
- get balance of logged in user
- get list of encrypted transaction details for the logged in user
- send a transaction from logged in user to any other username in the system

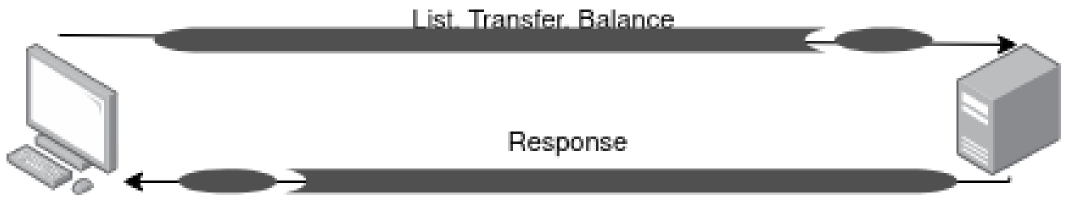

Figure 1.1: secure channel of communication between client and SBA server

The SBA application is hosted by a server which maintains users and accounts. Users interact with the SBA server through a secure channel that must be established before issuing operations.

1.1 Assumptions

Clients would have the following in their possession:

- root CA certificate
- RSA key pair

Server would have the following in its possession:

- server certificate, signed by root CA
- RSA Key Pair
- username & password of all the clients
- public RSA Key of each client
- balance of each client
- list of encrypted Transactions for each client

The key exchange must provide perfect secrecy in order to encrypt and authenticate each session. Also the session must be protected against replay attacks. In this paper, we first present the different protocols implemented in this application. We will then explain the global structure of the implementation. Eventually we will conclude about the safety of our application and the further progress that could be made.

Protocols

2.1 Initial Handshake

To establish authenticated channel between server and the client we are using ECDH algorithm to derive shared secret with the following steps would be taken:

- client would send their ephemeral ECC public key combined with a fresh nonce to server.

  - **ephemeral ECC Key Pair:** for every connection to be established client generates a temporary key using OpenSSL library on *NID_X9_62_prime256v1* curve

  - **nonce:** 16 random bytes are generated using OpenSSL PRNG to ensure unpredictability and these bytes are concatenated with a timestamp which ensures uniqueness.

- server would respond with it's signed certificate.

  - **certificate:** server has EC Key certificate signed by CA that is sent to client

  - **session key:** client and server can derive a session key at this point using the client's temporary key and server's certificate. we also **SHA256** the derived key and use the out put as session key for better entropy.

- server would also send the received nonce from client combined with another nonce it generated plus counter all encrypted with the shared key calculated.

- to conclude the connection has been established client would send server nonce combined with counter+1 back to server

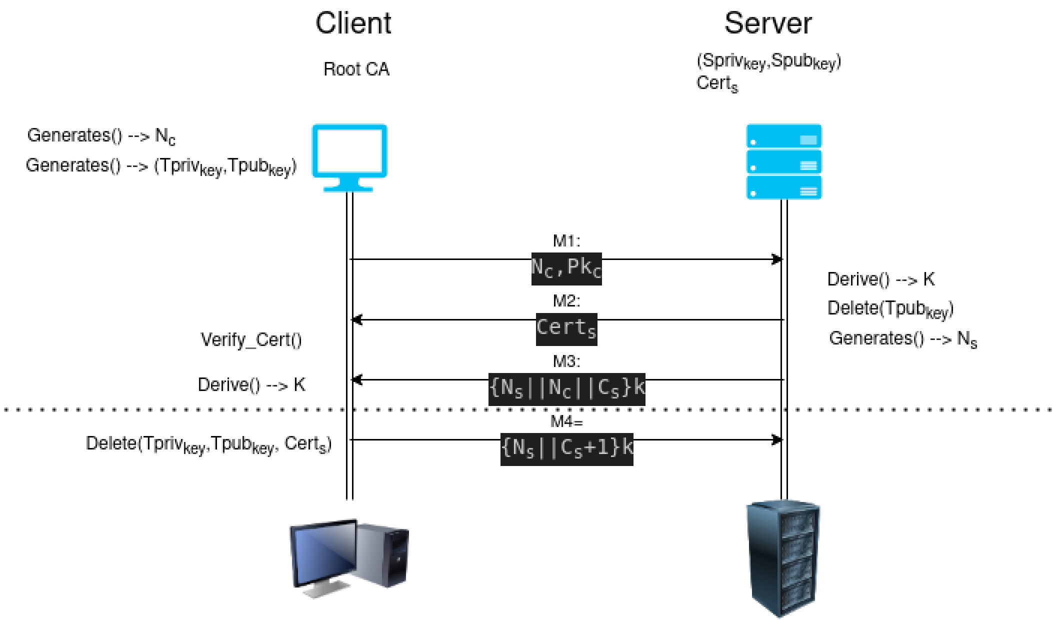

Figure 2.1: Sequence Diagram of ECDH Key Exchange

2.2 Client-Server communication

Generally Client-Server communication has specific structure for requests and responses.

- Request

```
--> commandCode(0-3):args[0]:args[1]:...
```

- Response

```
--> result(0|1):error:message
```

After the Initial handshake between server and client is completed, anytime one wants to send a message, we go through the following steps:

- add counter end of message **payload** = **plain-text** ||**counter**
- **select a random IV** and encrypt message with **aes_256_cbc(key,payload)**
- create the message digest with **digest** = **HMAC(key,payload)**
- bundle every thing and send it as **(IV** ||**cipher-text** ||**digest)**

and on the receiving side following operations would be performed:

- split the received message into **cipher and digest parts**.
- decrypt cipher to obtain the payload.
- calculate digest and check if they match the attached digest other wise ignore.
- extract the message counter and see if it's valid other wise ignore.

Here we have naturalized replay attacks by the use of counter. furthermore a received message is always authenticated a received message is always authenticated with hamc, and a random IV that was created using OpenSSL is being used in process of decryption/encryption. Each digest is created out of the authentication key, the counter and the plain-text such that reordering attacks are also not possible by the attacker.

Implementation

3.1 DataStore

we need to store the information relevant to each client such as username,password,balance, and transactions. we are using SQLite for this purpose as it's a light weight database which we can restrict access to. we are using the following datastructres within SQLite database:

- **clients**
  - *id*
  - *username*
  - *password*: **SHA256(salt, user provided Password)**
  - *pubkey*: **RSA public key of the client**
  - *Balance*
- **transactions**
  - *id*
  - *user_id*
  - *enc_transaction* : **RSA Encrypted Transaciton details**

3.2 Operations

The list of operations that are performed by the Client and the Server with respective Sequence diagrams are represented in this section: server can reply for each operation with 0 or 1 status indicating failure or success of operation respectively. each of operations have their own specific code which is as follows:

- Login (**0**)

  To Login user must provide valid username and password. client would encrypt and send these info to server after which server will look for the user and calculate the password hash and check it against db record if they match server will set session user and reply with 1 indicating success-full login.

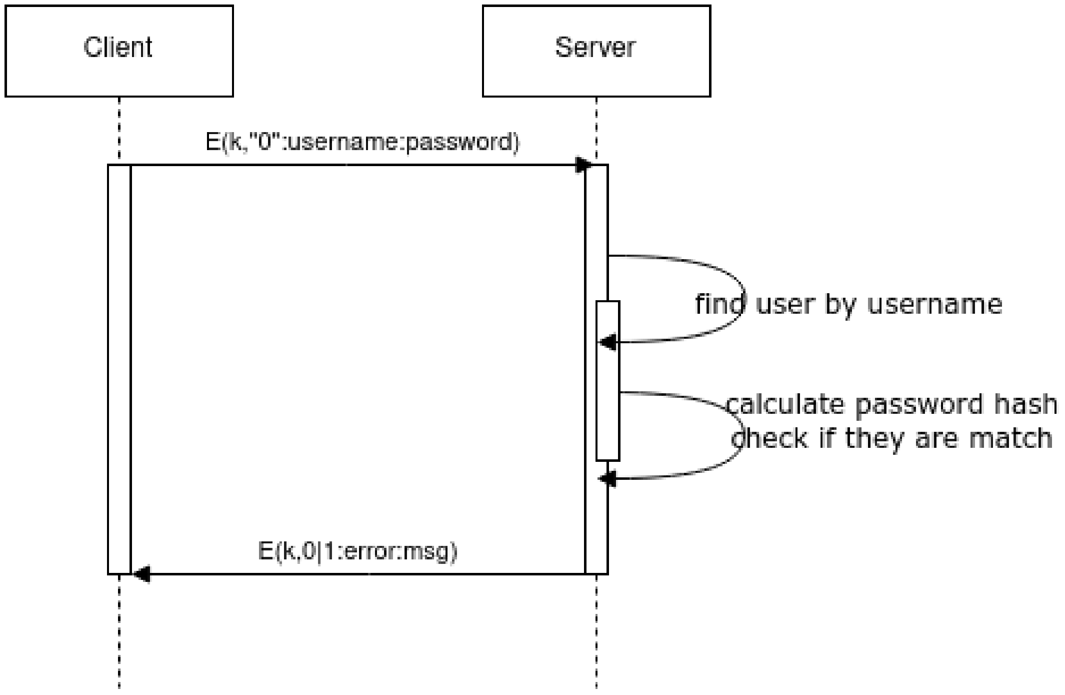

Figure 3.1: Sequence Diagram of Login OP

- Balance (**1**)

  To receive users balance user should be already logged in. if so client can request server for the balance and server would check the db records and reply with amount of balance in user's records.

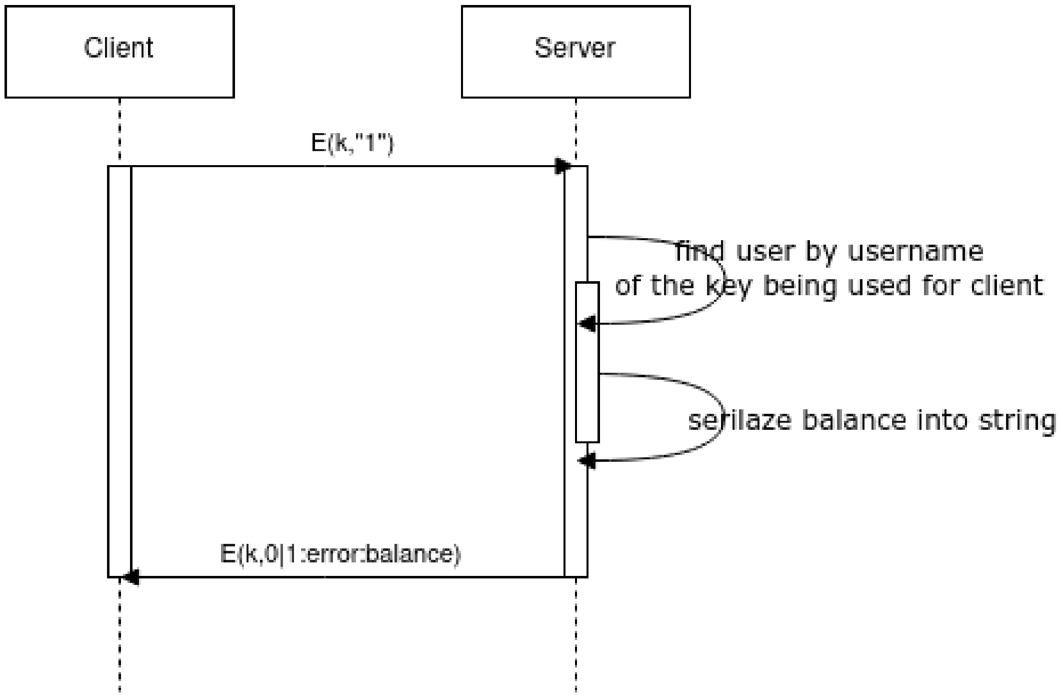
Client
Server
E(k,"1")
find user by username of the key being used for client
serilaze balance into string
E(k,0|1:error:balance)

Figure 3.2: Sequence Diagram of Balance OP

- Transfer (**2**)

  To Transfer user should be logged in already. client can send request for transfer by providing the amount and receiver to the server. server then would do some checks regarding validity of the transactions and if all the checks pass server will encrypt and save transaction details in db deduct from sender add to receiver and return with success(1) response. otherwise failure(0) will be sent with an error message indicating what went wrong.

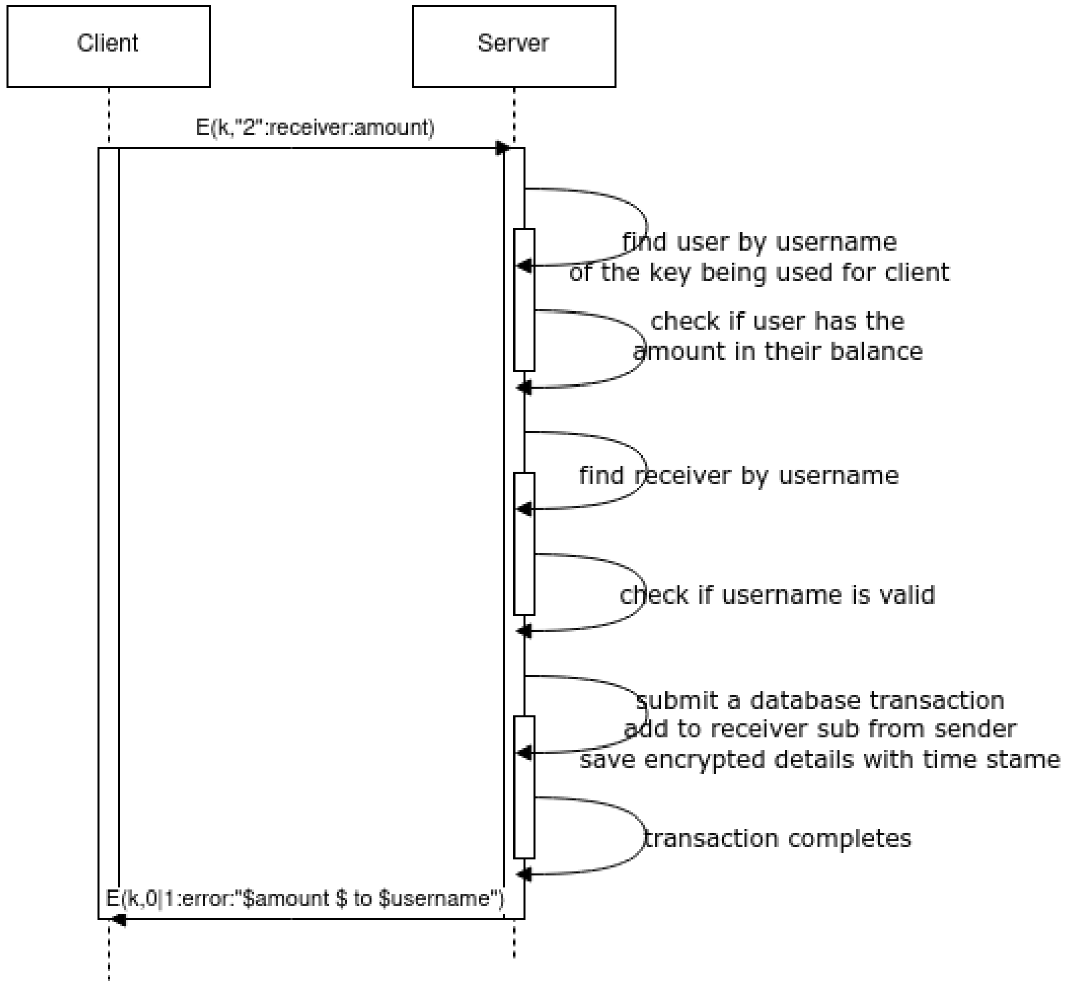
Client
Server
E(k,"2":receiver:amount)
find user by username
of the key being used for client
check if user has the
amount in their balance
find receiver by username
check if username is valid
submit a database transaction
add to receiver sub from sender
save encrypted details with time stame
transaction completes
E(k,0|1:error:"$amount $ to $username")

Figure 3.3: Sequence Diagram of Transfer OP

- List of Transactions (**3**)

  To list users Transaction, user should be logged in already. once server receives the request server will get last 10 encrypted transactions of the user and will send them to client, client will extract encrypted transaction and decrypt everyone of them and print them out.

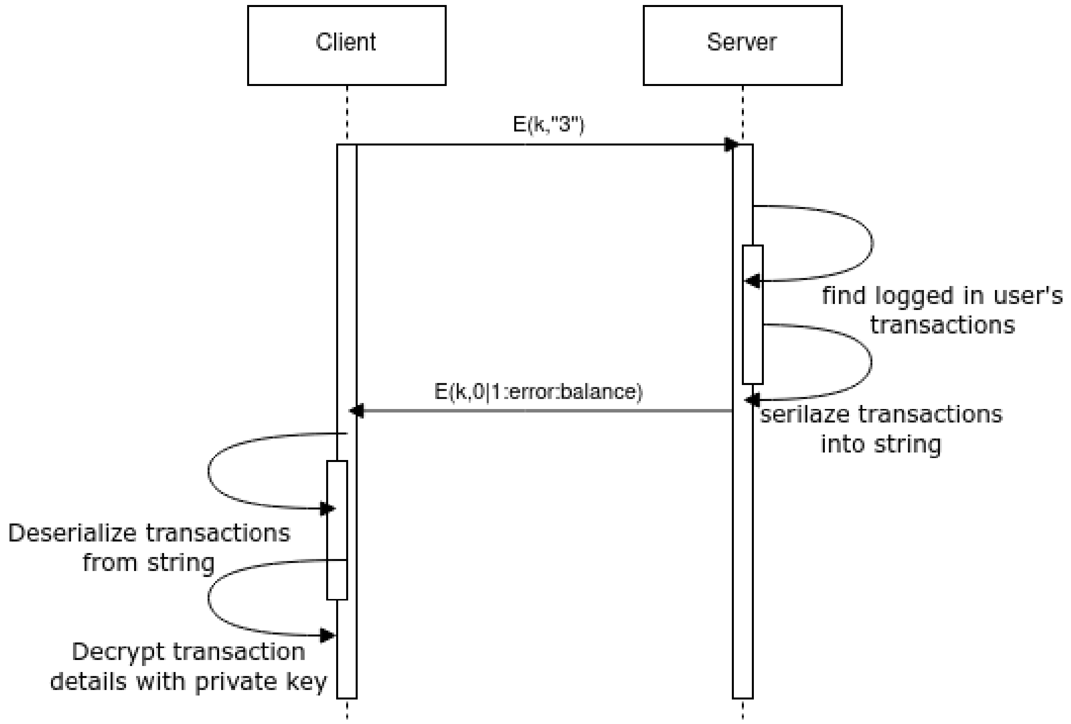
Client
Server
E(k,"3")
find logged in user's transactions
serilaze transactions into string
E(k,0|1:error:balance)
Deserialize transactions from string
Decrypt transaction details with private key

Figure 3.4: Sequence Diagram of List OP

Conclusion

What we have done so far in this project is:

created a working client - server application that secures the communications from any passive adversary. which even if they are able to retrieve a session key they are only able to compromise the corresponding session. we have support for all the operations requested and have secured access only to logged in users information.

What else can be done:

we could update session key per operation success or failure which would require more bandwidth for our application. also it might be better idea to use aes in gcm mode rather than cbc mode as it provides built in message authentication therefore we may or maynot need HMAC digest used in this program.