Project Report:
Privacy Preserving eCommerce System

CS6349 : Network Security
Fall ‘23

Layeeq Ahmed (LXA230013)
Tejaswini Indala (TXI220003)
Ojal Bhatnagar (OXB210000)

Problem statement

This project report presents the design and implementation of a secure Privacy-Preserving eCommerce System. The system is made up of three primary sections: a Customer application, a Merchant application, and a Broker application. Within this system, the Merchant sells a range of eProducts to Customers, while the Broker's key job is to facilitate communication between Customers and the Merchant while protecting the Customers' privacy.

Initial Assumptions:

- There is no existing session for communication between either of the parties.
- Everyone has their own public/private key pairs.
- Customers have the public key of the Broker as well as the Merchant.
- Broker has the public key of the Customers as well as the Merchant.
- Merchant has the public key of the Broker but not the Customers.
- Broker has the usernames/passwords of all the Customers.

Workflow:

We assume:
P_c , K_c = Public key and Private key of the customer respectively.
P_B , K_B = Public key and Private key of the Broker respectively.
P_M , K_M = Public key and Private key of the Merchant respectively.

At the outset, Socket programming is utilized to establish a secure connection among the customers, broker and the merchant.

STEP 1: Authentication:

Broker-Merchant authentication:

When the merchant starts the terminal, a connection is established between them and the broker.

- The broker initiates a challenge-response authentication process by encrypting in merchants public key (P_M).
- This is done by generating a Random number($R1_B$) and encrypting it in merchants public key (P_M). This encrypted challenge is sent to the Merchant.

$$P_M\{\text{“Broker”},R1_B\}$$

- The Merchant, upon receiving the encrypted challenge, uses their private key (KM) to decrypt the message.
- The merchant then generates their own Random Value (R_M) and adds it to the received challenge number($R1_B$).
- The merchant then sends (R1B) + (RM) to the broker, encrypted in the brokers' public key(PB).

PB{"Merchant", (R1B +RM)}

- Broker decrypts the message , and retrieves R1B from it. They extract the challenge sent by the merchant.
- Since only the merchant can decrypt and obtain $R1_B$, merchant authentication is achieved.
- The broker now encrypts the received challenge(RM) with the merchant's public key and sends it back to the merchant
- If the response received at the merchant's end is RM , we can say that the merchant authenticated the broker.
- Thus, mutual authentication is achieved through this challenge-response process.

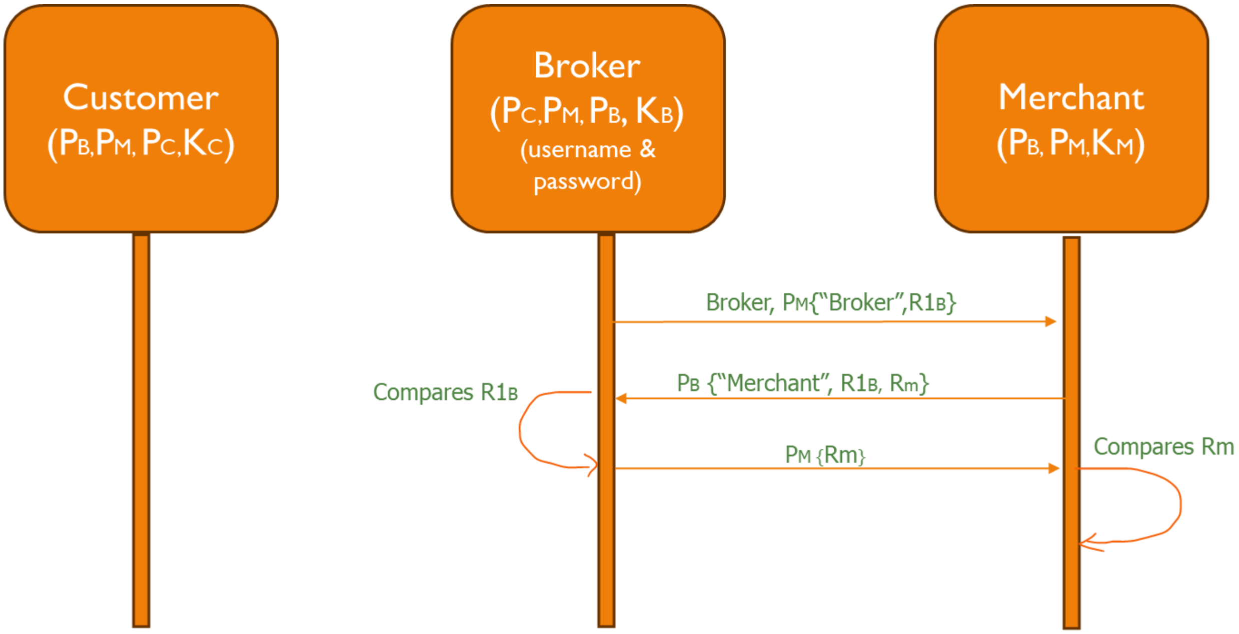

Figure 1: Broker-Merchant Authentication

Broker-Customer Authentication:

Initially, the customer does login using their Username and Password in the customer application.

- The customer application sends a random number(R_C), username and hash(SHA 256) of the password encrypted in the broker's public key (P_B)

Customer1, P_B\{ username , H(password), R_C \}

- The Broker decrypts the message using their Private key(K_B).
- The Broker compares the username and hash of password with the data available in the database.
- If the username and password match, then the broker sends the challenge back to the customer for authentication verification.
- $P_C\{R_c$,(Success/Failure)\}
- Depending upon the message reply from the broker we get:
  - Login Success- Both customer and broker receive verification replies.
    - `Random challenge verified; Username and password verified`
    - `success`
  - Login Failed- If either wrong challenge response was obtained at the clients end or the broker did not find the obtained username-hash(password) combination in the database.
    - `Did not find a matching username and password`
    - `Incorrect challenge reply`

- The diagram below shows the message flow during Customer Authenticat

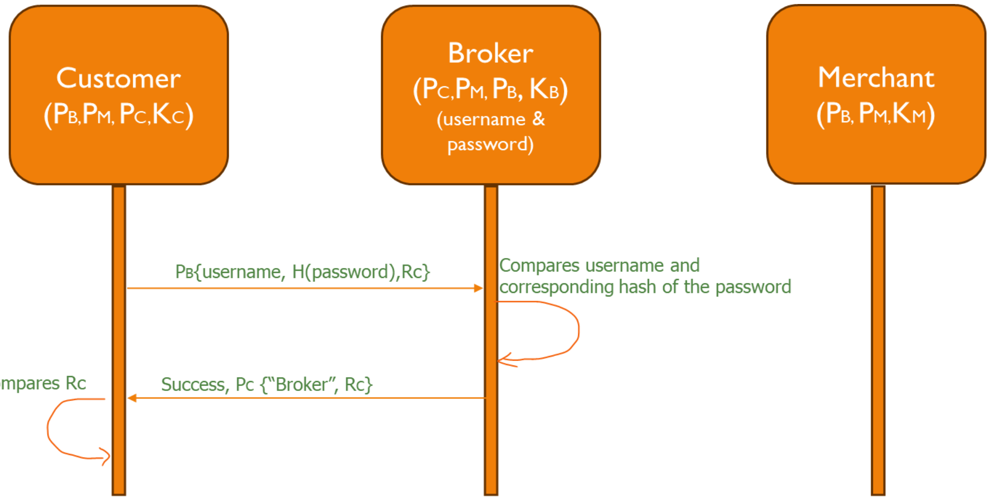
Customer
(P_B,P_M, P_C,K_C)
Broker
(P_C,P_M, P_B, K_B)
(username &
password)
Merchant
(P_B, P_M,K_M)
P_B\{username, H(password),Rc\}
Compares username and
corresponding hash of the password
Compares Rc
Success, Pc \{"Broker", Rc\}

Figure 2: Broker-Customer Authentication

STEP 2: Establishment of Session keys

For generating the session keys between the three parties, we use Diffie Hellman key exchange protocol.

The Diffie-Hellman key exchange relies on the prime number 'P' and the generator 'G' to enable two parties to compute a shared secret. Through modular exponentiation with these values, each party generates public keys ('A' and 'B') that, when exchanged and combined with their private keys, result in a shared secret key for secure communication.

- We generate three session keys respectively:
    a. Customer — Broker (S1) (authenticated DH)
    b. Broker — Merchant (S2) (authenticated DH)
    c. Customer — Merchant (S3)
- Types:
    - Encryption key
    - MAC / Authentication key

Session Keys -S1, A1: Customer & Broker

The customer application chooses P1, G1 and a secret number N1 to generate their public key C1.
Similarly, C2 is generated using P1,G1 and a new secret number A1.

- The customer application now sends (P1, G1,N1,A1) along with (C1,C2) to the Broker by encrypting them with the broker's RSA public key(P_B).
- The broker generates their own DH public key(B1) using a secret number of their choice(N2).
- Using a new secret number A2, the broker generates a MAC session key (B2).
- Now the broker sends B1 and B2 back to the customer application by encrypting it using the customer's RSA Public key(P_C).
- Both the customer and broker now have DH encryption key(S1) and MAC session key(A1).

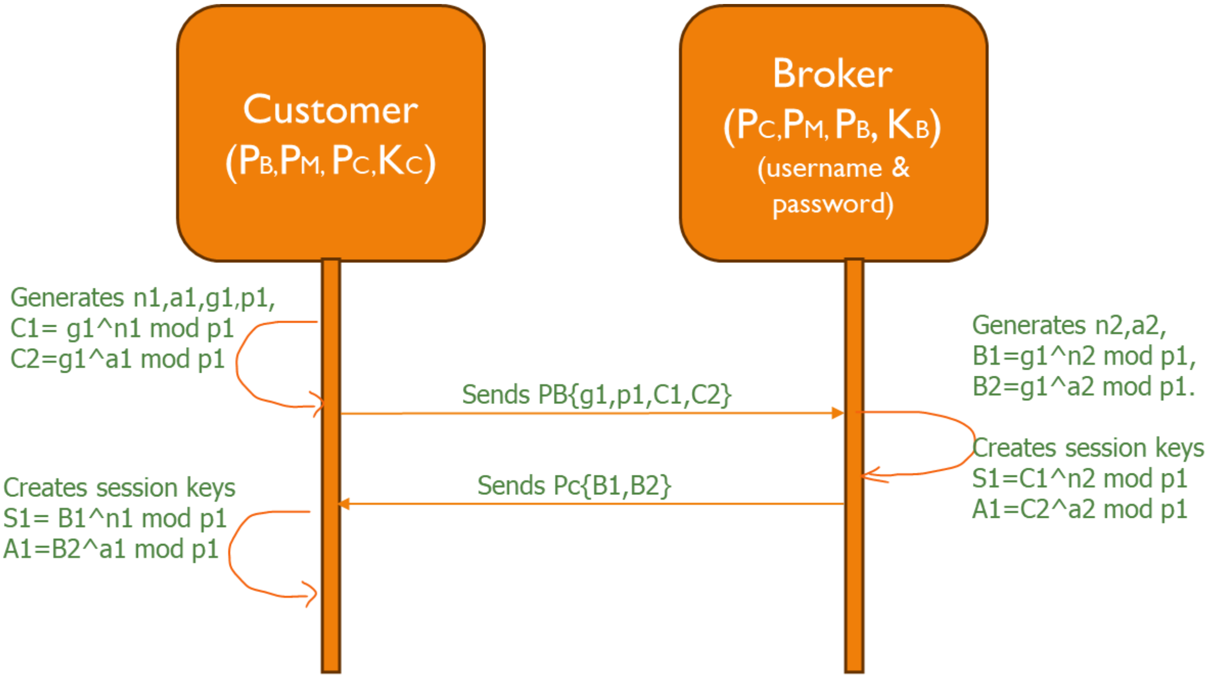

Figure 3: Establishment of S1,A1

Session keys S2, A2 - Broker & Merchant:

Similar to S1, A1 generation above, DH encryption key (S2) and MAC session key (A2) between broker and merchant servers are established.

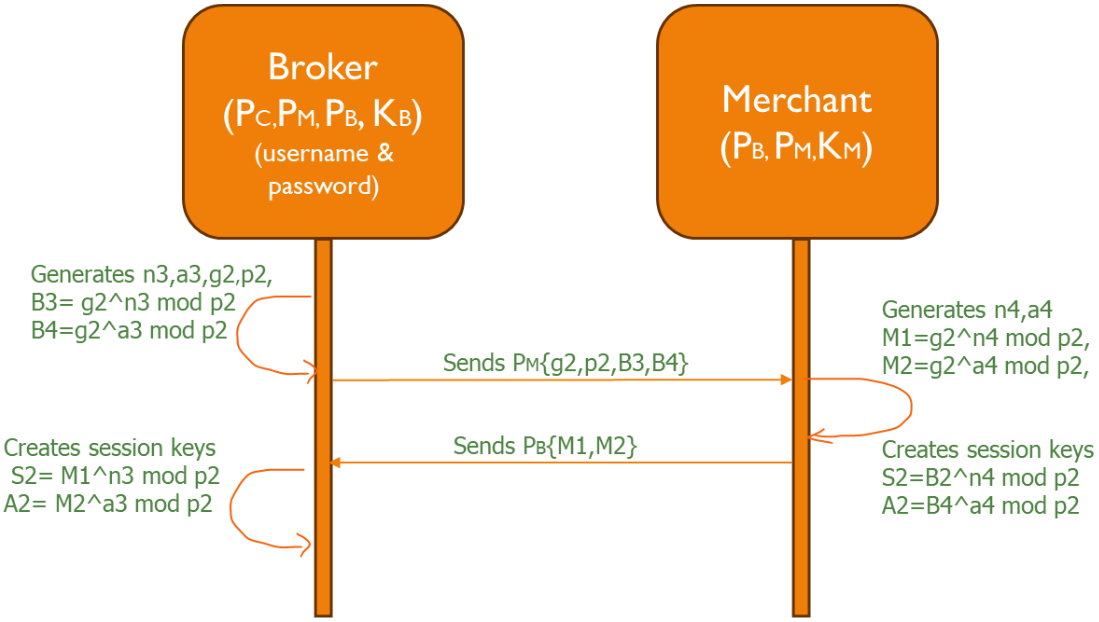

Figure 4: Establishment of S2,A2

Session Keys S3, A3: Customer & Merchant

After establishing S1, S2, A1, A2 keys, the customer application now proceeds to establish session keys with the merchant server.

- The message transmission follows a similar process as described earlier, with the merchant's response to the client being the only distinguishable variation in the method.
- Since the merchant will not have the client's RSA public key, the merchant will send back its DH public key without any RSA encryption.
- The assurance arises from the fact that the broker is unable to acquire the session keys (S3, A3) since the DH public keys communicated by the customer application are encrypted, without which the broker cannot generate the session keys between client and merchant.

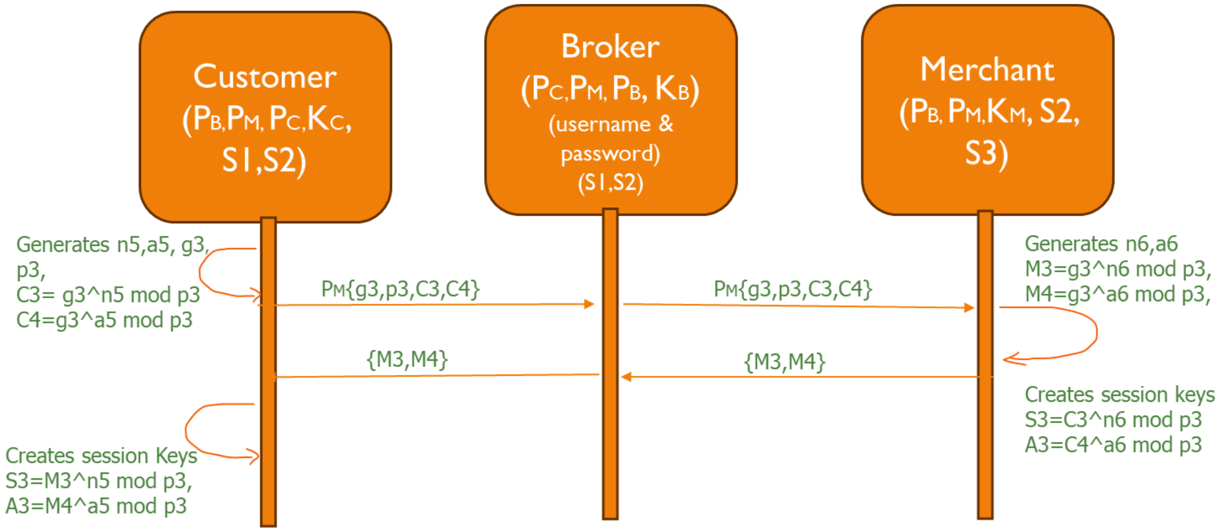

Figure 5: Establishment of S3,A3

STEP 3: eProduct Browsing:

Once the session keys are established, customer requests for the merchant's list of products to browse from.

- The broker relays the request to the merchant.
- The merchant sends the list of products to the customer using S3,and thus the broker would not know the contents of this list.
- The customer browses through this list and selects a product to purchase.

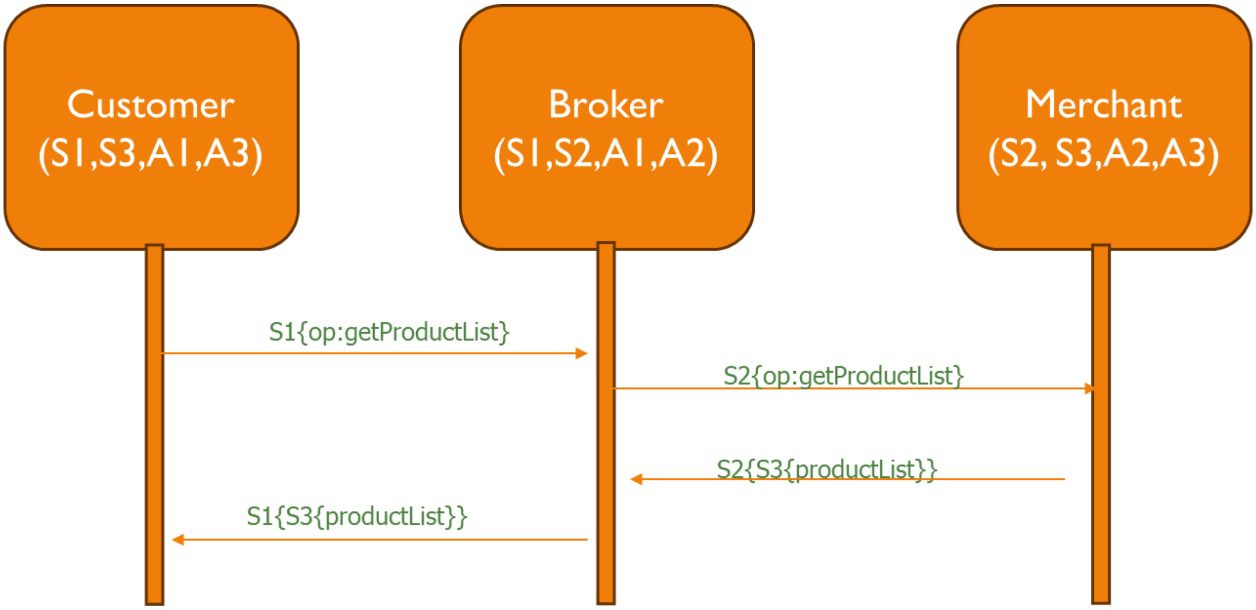

Figure 6: Transfer of eProducts list

STEP 4: Product Checkout

- The customer application sends the selected product to the merchant.
- Merchant creates an order object which contains an unique order_id,selected product, total payable amount and status .
- Merchant sends the order object to both the customer and broker without the selected product details.
- The order details are displayed to the customer.
- This marks the end of product checkout.

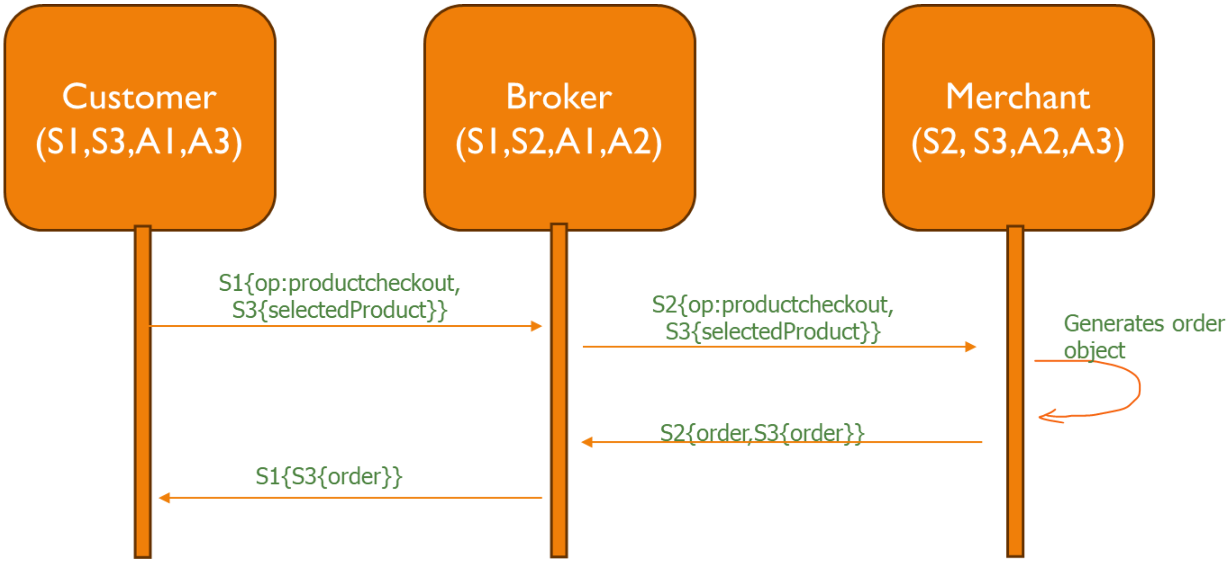

Figure 7: Product checkout process

STEP 5: Process Payment:

- The customer can choose to accept or decline the order.
- If the customer chooses to proceed with payment, their account details are taken as input and sent to the broker with the Order_Id.
- The broker validates the Order_Id and account details received.
- Here, we assume that the total payment includes the fees to be remitted to the broker.
  - Total payment due = Price of the products + Broker charges
- Broker simulates the payment transaction as follows:
  - Subtract the total amount from the customer's account.
  - Add the merchant's portion of the funds to the merchant's account.
  - Add the broker's share to their own account.
- Once the operations are completed, the Broker sends a 'Payment successful' message to both the parties.
- In case if the customer's account is not found or the customer's account has insufficient balance, the broker sends "Payment Failed" message to both the parties.

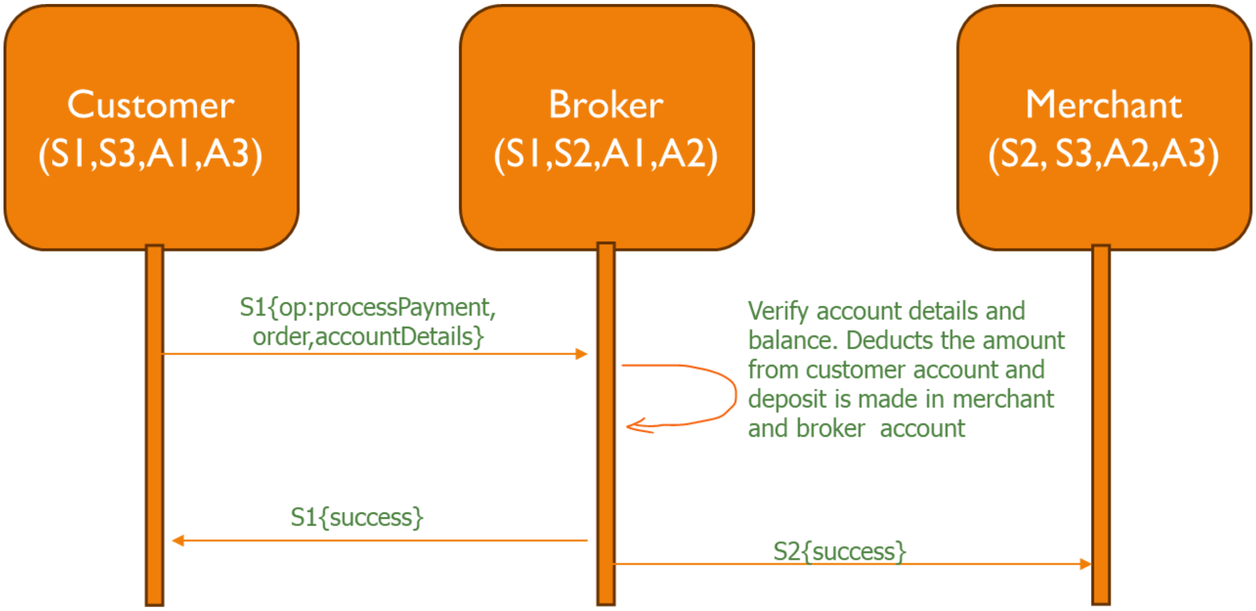
Customer (S1,S3,A1,A3)
Broker (S1,S2,A1,A2)
Merchant (S2, S3,A2,A3)
S1{op:processPayment, order,accountDetails}
Verify account details and balance. Deducts the amount from customer account and deposit is made in merchant and broker account
S1{success}
S2{success}

Figure 8: Payment Process

STEP 6: eProduct Delivery:

- After a successful payment, the customer sends the order ID along with a request for the delivery of the eProduct.
- The merchant verifies the payment status against the order_Id and reads the selected product.
- Merchant then proceeds to apply padding to the content to hide product details from the broker.
- The merchant dispatches the eProduct and its hash.
- The customer receives the eProduct and verifies the hash for integrity and authenticity check.

Diagram showing a successful delivery:

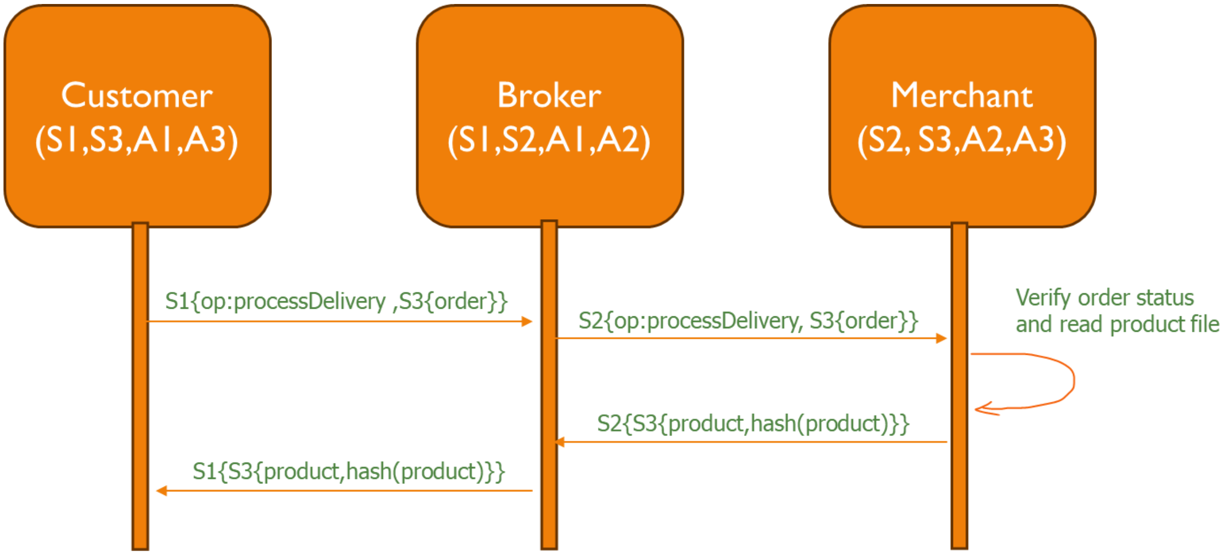
Customer
(S1,S3,A1,A3)
Broker
(S1,S2,A1,A2)
Merchant
(S2, S3,A2,A3)
S1{op:processDelivery ,S3{order}}
S2{op:processDelivery, S3{order}}
Verify order status
and read product file
S2{S3{product,hash(product)}}
S1{S3{product,hash(product)}}

Figure 9: eProduct delivery

Once the transaction is done and the customer receives the products,this marks the end of the eProduct purchase journey .

Security Requirements :

We obtain confidentiality, authenticity and integrity at each applicants side as follows:
For encryption, we use AES in CFB mode. For hash, we use SHA256. For MAC, we use SHA256 based HMAC.

Authenticity:

- **Step-1**
  - Broker authentication at customer’s end is achieved through challenge-response message transfer,i.e, through verifying random numbers sent and received.
  - Customer authentication is done by comparing the hash of the password received with the stored hashes present in the database.
  - Mutual authentication is achieved through challenge response message transfer between broker and merchant.
- **Steps 3-6**
  - DH encryption session keys at each end ensures the authenticity of the sender of each encrypted message.

Confidentiality:

- After session keys are created at each end, each message is encrypted with their respective DH encryption keys. This provides confidentiality.

Integrity:

- For each session encrypted message, the message is appended with its keyed hash (using MAC session key of that session). When received, this hash is compared with the hash of the received message and validated. This hash comparison will fail if the message was modified. SHA256 hash is used for this.
- While delivering the product, its content undergoes an appending process with a SHA256 hash, which is then transmitted. Upon reception by the customer app, this hash is cross-checked against the product content's hash to validate and ensure the integrity of the received product.

Setup/ReadMe

Folder Structure:

- Keys: within each entity there is a keys folder having corresponding RSA keys.
- Broker : Contains account details, passwords, orders.
    - **Accounts**: account details of each entity.
    - **Orders:** for each order placed, a file with order details is saved.
    - **Passwords:** json file with usernames and corresponding hash of passwords.

- Customer:
    - **Purchased:** a folder for each user containing the purchased product.

- Merchant:
    - **Products:** contains available products
    - **Orders:** contains files for each order.

Libraries used

- Rsa
- Secrets : module is used for generating cryptographically strong random numbers
- hashLib: for SHA256
- Hmac: for keyed hash
- Math : to obtain gcd
- Crypto.cipher:for AES encryption/ decryption
- Socket
- Threading
- Uuid: unique order id generation
- Json
- Argparse
- config Parser
- Base64
- Os

Build

Built with python version: Python 3.10.12
Install additional dependencies with pip3 -r requirements.txt

RUN AS BROKER:

```
python3 sockets.py --mode broker
```

RUN AS CLIENT:

```
python3 sockets.py --mode client
```

RUN AS MERCHANT:

```
python3 sockets.py --mode merchant
```

Configuring defaults:

Use config file to define default IP and ports for each mode. Also provides many runtime configuration values that can be changed:

- broker share/commission percentage
- broker account number
- merchant account number (at broker)
- different password hash file
- file padding length
- product amount

References:

- https://realpython.com/python-sockets/
- https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python
- https://docs.python.org/3/howto/sockets.html
- https://www.markdownguide.org/cheat-sheet/
- https://realpython.com/intro-to-python-threading/

Python Sockets Simply Explained - YouTube
File Transfer via Sockets in Python - YouTube
AES — PyCryptodome 3.19.0 documentation
secrets — Generate secure random numbers for managing secrets — Python 3.12.1 documentation
base64 — Base16, Base32, Base64, Base85 Data Encodings — Python 3.12.1 documentation
hmac — Keyed-Hashing for Message Authentication — Python 3.12.1 documentation
rsa · PyPI
hashlib — Secure hashes and message digests — Python 3.12.1 documentation
uuid — UUID objects according to RFC 4122 — Python 3.12.1 documentation

Contribution:

Group tasks: implementation of RSA,AES encryption using respective modules, report.
Layeeq Ahmed : Keyed Hash,multithreading, socket implementation
Tejaswini Indala: Message sequencing and payment processing
Ojal Bhatnagar: DH implementation and file padding