Toronto Metropolitan University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Network Security
Course Number:	COE817
Semester/Year	W2025

Instructor:	Dr. Truman Yang
Teaching Assistant:	Aman Yadav

Assignment/Lab Number:	Final Project
Assignment/Lab Title:	Secure Banking System

Submission Date:	April 12, 2025
Due Date:	April 12, 2025

Student Last Name	Student First Name	Student Number	Signature	Student Last Name
Fredette	Benjamin	500947897	BF	Fredette
Muzzo	Eric	501019745	EM	Muzzo
Ashraf	Rafa	501044626	RA	Ashraf
Siddiqui	Noor-e-Sahar	501040733	NA	Siddiqui

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at:
http://www.ryerson.ca/senate/current/pol60.pdf

Secure Banking System Project Report

Introduction

The purpose of this project was to design and implement a secure banking system that facilitates financial transactions through a client-server model. The central goal was to ensure the confidentiality, integrity, and authentication of communication between ATM clients and the bank server. To achieve this, we developed a multi-layered security protocol that includes authenticated key distribution, key derivation for encryption and message authentication, and secure logging of transactions.

This project was developed as part of the COE817 course requirements. It is designed to simulate real-world secure financial systems and to provide practical experience in implementing cryptographic principles in a full-stack Python-based application using Flask, FastAPI, WebSockets, and standard cryptography libraries. The system supports registration, login, balance inquiry, deposit, and withdrawal operations.

Scope

- Develop a secure communication protocol between ATM client and bank server.
- Support account registration, login, and transaction services.
- Ensure confidentiality and integrity through AES encryption and HMAC verification.
- Maintain secure audit logs on the bank server.
- Provide a GUI for the ATM client and bank server.

Limitations

- Only a single pre-shared key (PSK) is supported at setup.
- The system runs locally; no cloud deployment is included in this version.
- Authentication is based on username and password, without two-factor authentication.

Design

Overview

The system consists of two major components:

1. **ATM Client** – Implemented using Flask, this GUI-based client allows users to register, login, and perform transactions.

2. **Bank Server** – Implemented using FastAPI, it exposes secure REST endpoints to handle all banking operations and manage secure communication.

Technologies Used

- Python for all backend and frontend scripting.
- Flask for GUI development of ATM client.
- FastAPI for RESTful services on the bank server.
- WebSockets for full-duplex communication.
- Cryptography (via `cryptography` module) for AES-CBC and HMAC.
- HKDF (HMAC-based Key Derivation Function) to derive encryption and MAC keys.

Architecture Diagram

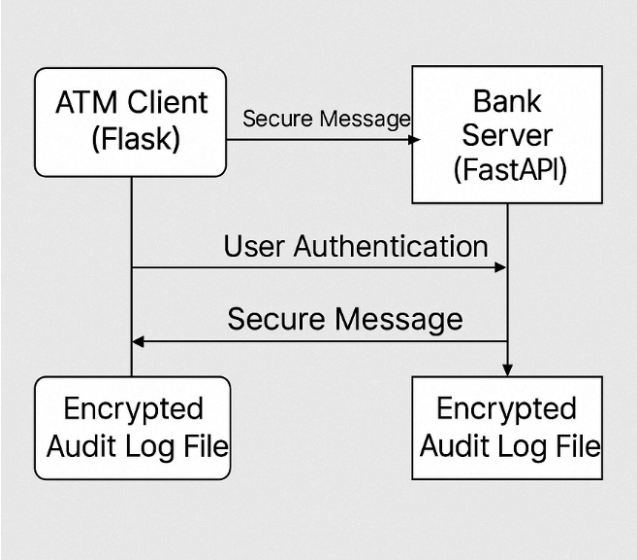

Figure 1: Architecture Diagram

Implementation Details

Authentication and Key Distribution Protocol

The system utilizes a dual-phase approach to establish a secure session between the ATM client and the Bank server. The protocol begins with an authentication phase that leverages a pre-shared key known to both parties. This pre-shared key is stored securely and used to encrypt the initial messages that contain user credentials. The client first generates a random nonce using a cryptographically secure random generator. This nonce is included in the authentication request along with the user's credentials and a timestamp. Including a timestamp and a nonce provides protection against replay attacks because it ensures that any authentication request is unique and time-bound.

Upon receiving the client's request, the Bank server decrypts the message using the same pre-shared key. The server then validates the credentials and generates its own random nonce. The use of dual nonces, one from the client and one from the server, serves two key purposes. The first purpose is to guarantee the freshness of each session. The second purpose is to contribute to the derivation of the Master Secret that will be used to secure subsequent communications. The client computes the Master Secret by applying an HMAC function to the concatenation of the client nonce and the bank nonce, using the pre-shared key as the HMAC key. This computation not only ensures that both parties have contributed randomness to the session but also binds the Master Secret to the specific authentication attempt.

The next step involves deriving the session keys that will be used for encryption and message integrity. The system employs a HMAC-based Key Derivation Function known as HKDF. HKDF is used to derive two separate keys from the Master Secret. One key is designated for encrypting transaction data using the AES algorithm in CBC mode with PKCS7 padding. The other key is used for generating a Message Authentication Code to verify the integrity of the communication. The separation of these keys plays an essential role in mitigating the risk of certain attacks, as the compromise of one key does not automatically compromise the other.

The following list outlines the key steps of the protocol:

1. The client initiates the process by generating a client nonce and prepares a JSON message containing login or registration credentials along with the nonce and a timestamp.
2. The message is encrypted using AES-CBC with the pre-shared key and sent to the Bank server over a secure connection.
3. The Bank server decrypts the message, validates the credentials, and generates a bank nonce.
4. The Master Secret is computed by the client through an HMAC operation on the concatenation of the client nonce and the bank nonce. This computation ensures that both parties contribute to the session keys.
5. HKDF is applied to the Master Secret to derive two separate keys. One key is used for encrypting further communications, and the other key is used for generating MACs to verify message integrity.
6. The Bank server sends a response that includes the bank nonce and relevant account data. This response is encrypted using the pre-shared key, ensuring that the exchange is secure.

The design of the protocol offers strong protection against replay attacks because of the inclusion of random nonces and timestamps in every authentication attempt. An attacker attempting to replay a valid authentication message would not succeed since the server expects the client nonce to be fresh and the timestamp to be within an acceptable range. This approach ensures that each session is unique and that stale messages are rejected. Moreover, the mutual contribution of randomness to the Master Secret calculation helps to ensure authenticity. Both the client and the Bank server must possess the pre-shared key and must generate their respective nonces to compute the Master Secret. The adherence to this strict protocol procedure prevents impersonation and ensures that only legitimate parties can derive the correct session keys.

Overall, the authentication and key distribution protocol ensures that the exchange of sensitive information is protected through encryption and integrity verification. The design choices made in this protocol, such as the use of nonces and HKDF-derived keys, provide strong guarantees against replay attacks and unauthorized access while enabling a secure communication channel for subsequent transactions.

The following method is used to generate the master_secret key shared between the ATM Client and Bank Server

```
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes

def kdf_derive(master_secret, info):
    return HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=info.encode(),
    ).derive(master_secret)
```

Secure Data Transaction Protocol

The system secures every transaction between the ATM client and the Bank server by encapsulating each operation in a secure message envelope. The secure message envelope is constructed using symmetric key cryptography and a message authentication code to ensure both confidentiality and integrity. The process begins once a secure session has been established through the key distribution protocol and separate session keys have been derived using HKDF.

When a transaction is initiated, the client first converts the transaction details into a JSON format and then encodes the JSON string into bytes. This plaintext is padded according to the PKCS7 standard so that its size is compatible with the block size requirements of the AES algorithm. The client then generates a random initialization vector for each transaction to ensure that even identical messages will yield different ciphertext outputs. With the session-specific encryption key, the client encrypts the padded plaintext using AES in CBC mode. The result is the ciphertext that hides the original message.

After encrypting the message, the client computes a Message Authentication Code by applying HMAC with the session-specific MAC key over the ciphertext. The integrity of the ciphertext is thereby guaranteed, as any attempt to modify the encrypted data would require recomputing the MAC without knowledge of the secret MAC key. The complete secure envelope is then formatted as a JSON object that includes three fields. The first field is the initialization vector which is presented as a hexadecimal string. The second field contains the ciphertext in hexadecimal format and the third field carries the MAC also encoded as a hexadecimal string. The structure of the envelope ensures that every transaction message has the necessary elements to be securely transmitted and validated by the receiving party.

The secure message is sent over the WebSocket connection to the Bank server. Upon receipt of the secure envelope, the server performs a series of steps to validate and decrypt the message. The server first extracts the initialization vector, ciphertext, and MAC from the secure envelope by converting the corresponding hexadecimal strings back into bytes. To verify the integrity of the message, the server computes the HMAC over the received ciphertext using its session-specific MAC key and compares this value with the received MAC. If the two values do not match, the server rejects the message and raises an error, indicating that the message may have been tampered with.

Once the MAC has been successfully validated, the server proceeds with decryption. Using the session-specific encryption key and the extracted initialization vector, the server decrypts the ciphertext using AES in CBC mode. The decryption process produces the padded plaintext, which is then unpadded according to the PKCS7 standard to retrieve the original JSON-formatted message. Finally, the unpadded plaintext is decoded from bytes to a string and parsed to reconstruct the structured transaction data. This thorough process of encryption, integrity verification, and decryption ensures that every transaction is confidential, protected against unauthorized modifications, and resistant to replay attacks.

Encrypted Audit Logs

All transactions are kept on the server in encrypted format. Audit entries have this structure:

```
user_id | ACTION | TIMESTAMP
```

These logs are written using AES encryption to prevent tampering or unauthorized disclosure.

Example:

```
1 | BALANCE_INQUIRY | 2025-04-06T16:23:12Z
```

JSON Communication Format

All API communications use JSON with encrypted payloads. Example server response after registration:

```
{
 "status": "success",
 "message": "John Doe has been registered",
 "bank_nonce": "abcdefg",
 "data": {
  "user_id": 1,
```

```
  "username": "johndoe",
  "name": "John Doe",
  "account_id": 1000,
  "balance": 0.0
 }
}
```

Sensitive fields are encrypted using AES-CBC and authenticated with HMAC.

GUI Design

The ATM client uses Flask's render_template() to serve HTML pages with forms for:

- Login
- Registration
- Deposit/Withdraw
- Balance Inquiry

The ATM Client interface is implemented as a web application that provides a simple and intuitive graphical user interface. Users first access a login and registration page where they can either enter their credentials to log in or provide the required details to register a new account. The login page, designed using Bootstrap, presents two side-by-side sections for login and registration. In the login section, users enter their username and password, while the registration form requires a username, full name, and password. Error messages are displayed within the same page if the authentication fails.

Once authenticated, the user is redirected to a dashboard that displays a personalized greeting along with the user's username and account number. The dashboard is organized into distinct panels that allow users to perform banking operations. The deposit panel provides a form to input an amount for deposit and submit the request. Similarly, the withdrawal panel lets the user specify an amount to withdraw, and a separate section is dedicated to checking the current balance. Each of these transaction forms is designed to send a POST request to the corresponding route on the ATM Client backend.

Results

The system was tested with two ATM clients simultaneously. Each client could:

- Successfully register and log in

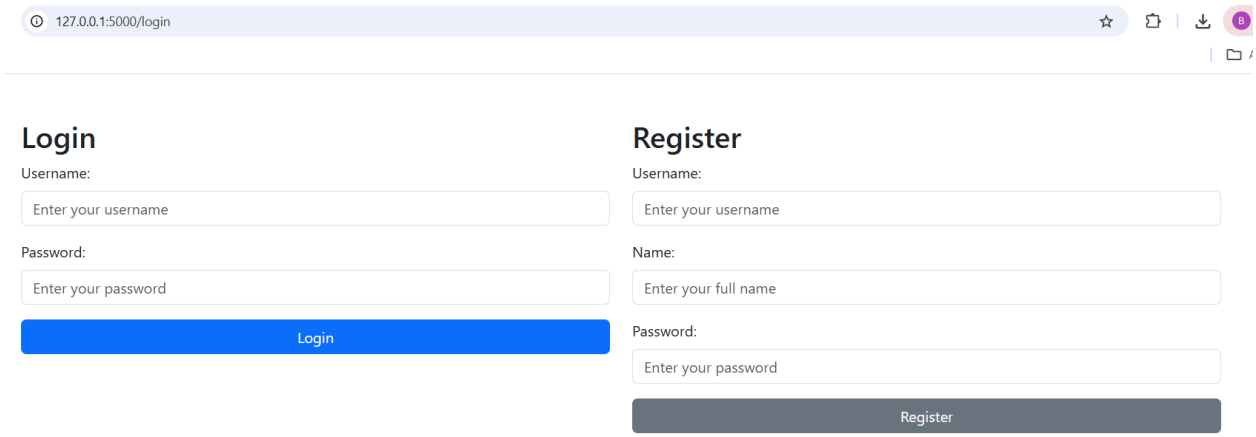

Figure 2: User accesses the ATM Client application through their web browser

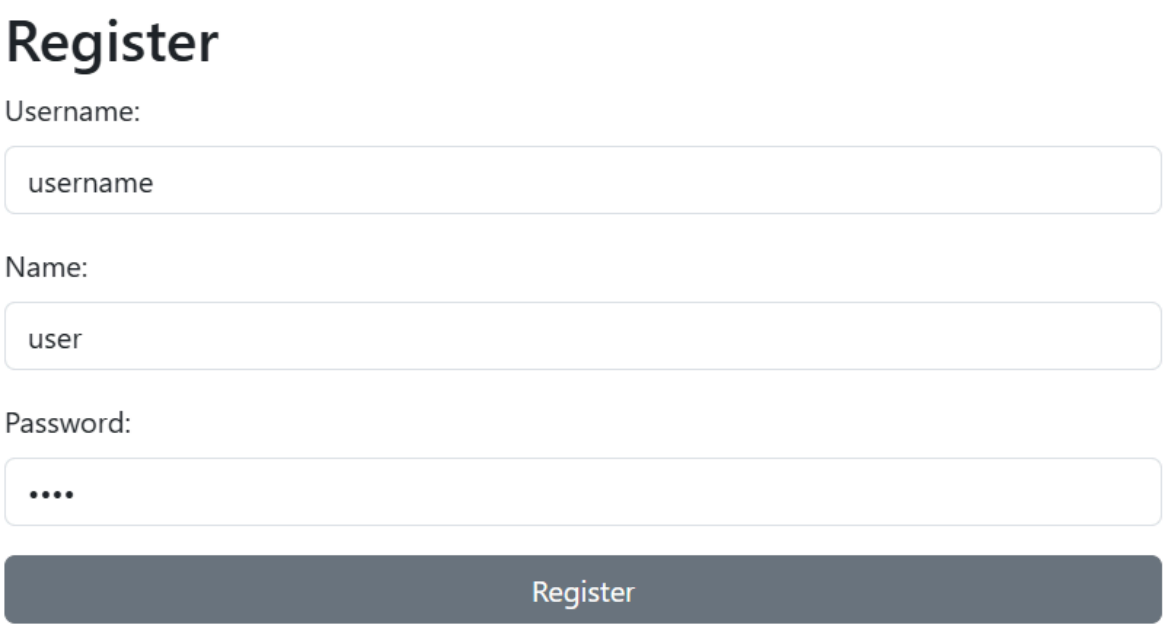

Figure 3: User enters their information into the 'Register' fields

Screenshots of the GUI interfaces and logs:

1. ATM Dashboard Screen

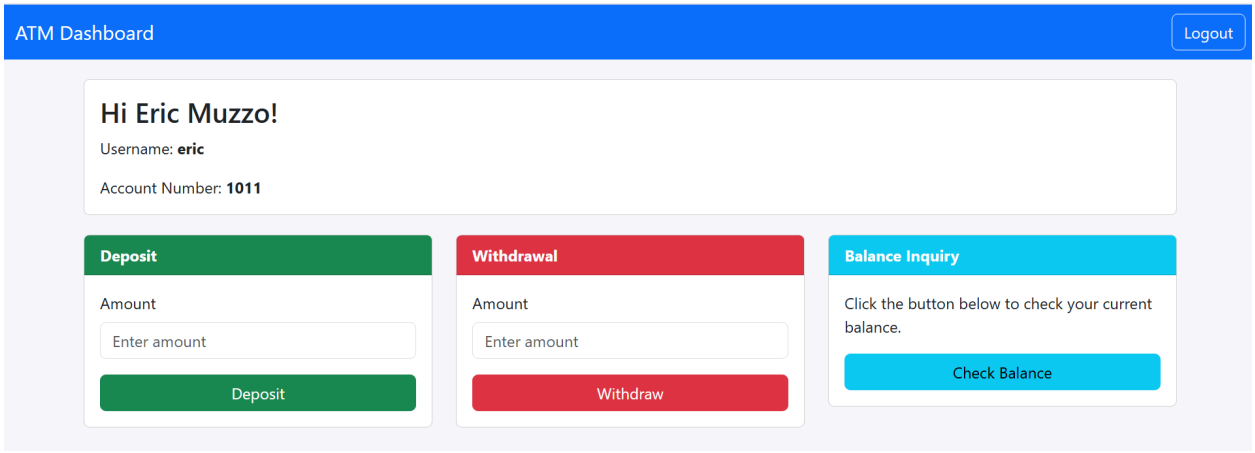

Figure 4: Screenshot of the user dashboard

2. Deposit Form and Confirmation Message

```
Decrypted bank server response for deposit: {"status": "success", "message": "deposit transaction
 was successful", "balance": 475.0}
```

Client Side

```
[CONNECTION 9814c497-79b6-4106-94c3-37943d1d73c5] Client requested transaction: deposit
User 6 requested deposit for 100.0
[CONNECTION 9814c497-79b6-4106-94c3-37943d1d73c5] Client deposit transaction was successful
```

Server Side

3. Balance Inquiry Output

```
Decrypted bank server response for balance inquiry: {"status": "success", "message": "balance tra
nsaction was successful", "balance": 375.0}
```

Client Side

```
[CONNECTION dbe50dea-e546-4046-85ec-cb2b4ecb147c] Client requested transaction: balance
User 6 requested balance
[CONNECTION dbe50dea-e546-4046-85ec-cb2b4ecb147c] Client balance transaction was successful
```

Server Side

4. Encrypted Audit Log Sample

All of the transactions were checked to be secure and logged correctly.

Conclusion

This project provided valuable insights into practical applications of cryptography in secure communication in real world scenarios. The implementation required a deep understanding of encryption, authentication, integrity verification, and client-server architecture. The use of Flask and FastAPI facilitated fast development and easy integration.

Key Takeaways

- Importance of layered security: using the combination of PSK, nonces, and derived keys to implement security in the system.
- Managing encryption and padding correctly for AES-CBC.
- Structuring JSON APIs to balance security and usability.
- Role of encrypted audit logs to ensure a secure system design.

Team Contributions

- **Eric Muzzo** (Lead Developer): Implemented the bank server using FastAPI and WebSocket integration. Developed key exchange and secure protocol features.
- **Benjamin Fredette** (Client Developer): Developed the Flask-based GUI, integrated encryption modules, and managed user interface.

- **Noor-e-Sahar Siddiqui** (Security Analyst): Developed key derivation and HMAC verification code, tested protocol edge cases, and designed the audit logging system.

- **Rafa Ashraf** (Tester & Documenter): Performed integration testing, wrote setup instructions and the README, and facilitated in writing this report.

This project gave us a better appreciation of the complexities of real-world secure financial systems, and the importance of detailed testing and design implementation in deploying cryptographic software.