MyBank

Cryptographic System

Author: Abhi Bhise | Security Architect

Table of Contents

1. Introduction

Banking systems are prime targets for cyber criminals due to the sensitive financial data they handle. A successful attack can lead to significant financial losses, identity theft, and loss of customer trust. This report details my implementation of "MyBank" security. This includes a secure banking system that protects customer data and transactions using modern cryptographic techniques.

The system addresses five key security requirements:

1. Data encryption for protecting client data and transaction details
2. Secure communication between the bank and its clients
3. Key management for generating, distributing and storing cryptographic keys
4. User authentication through multi-factor methods
5. Transaction integrity to prevent unauthorised modifications

MyBank supports three user roles with different permissions:

1. Clients can view accounts, check transaction history and make transfers
2. Bank employees can process deposits/withdrawals and access client information
3. System administrators can create users and manage access rights

While this implementation is primarily a demonstration model, it includes fundamental security principles that would be necessary in a real banking environment. This system shows how layered defences work together to protect sensitive financial operations.

The system aims to achieve four primary security goals:

1. Confidentiality: Ensuring sensitive data remains private through encryption
2. Integrity: Guaranteeing transactions cannot be altered once authorised
3. Authentication: Verifying user identities through multiple factors
4. Non-repudiation: Creating audit trails that link actions to specific users

2. Design

The top-level flow of how the application will work is shown in the Fig. 1.

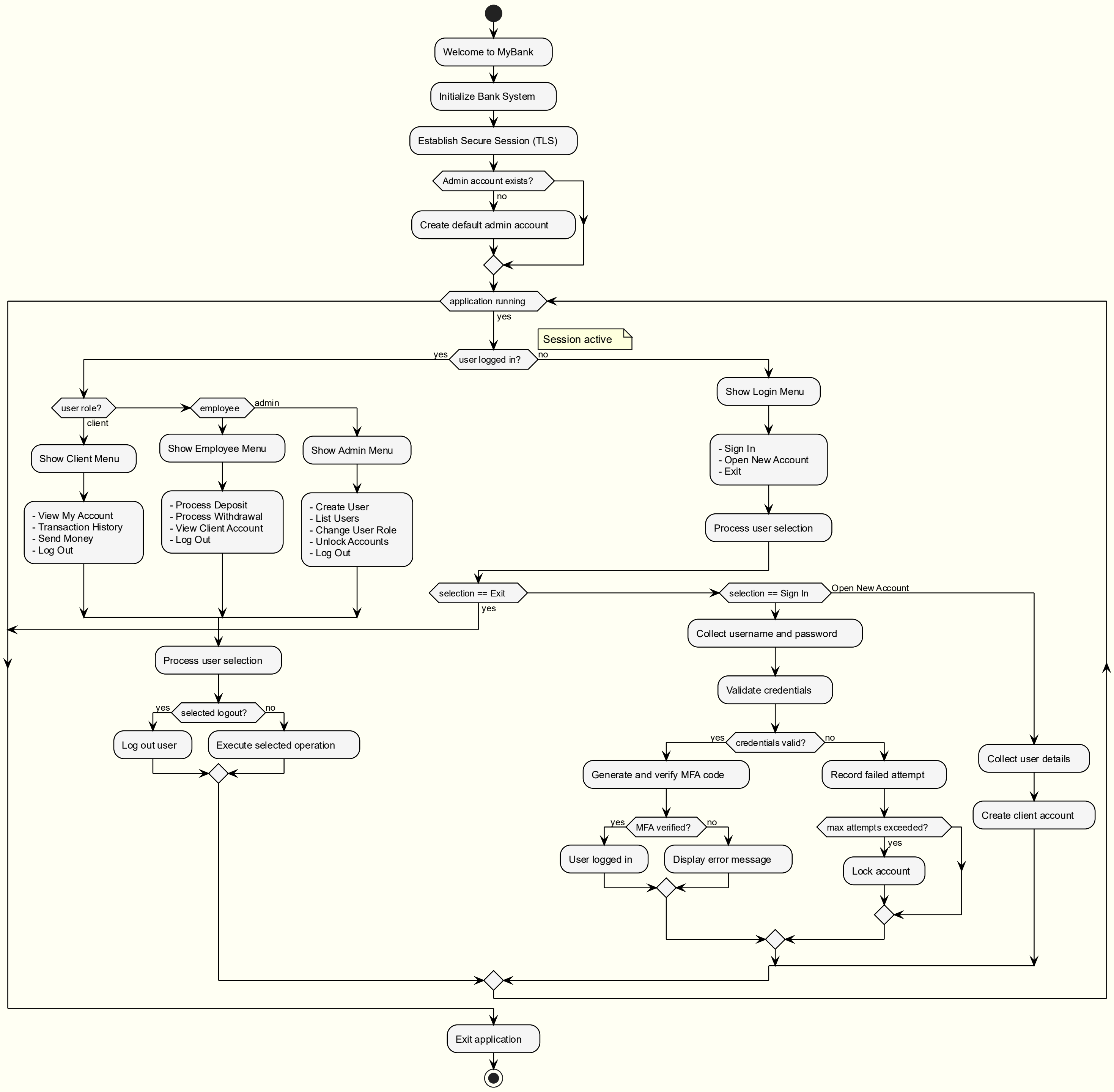


Fig. 1 Top level application flow

Fig. 2 shows how architecture and flow of the security methods and classes.

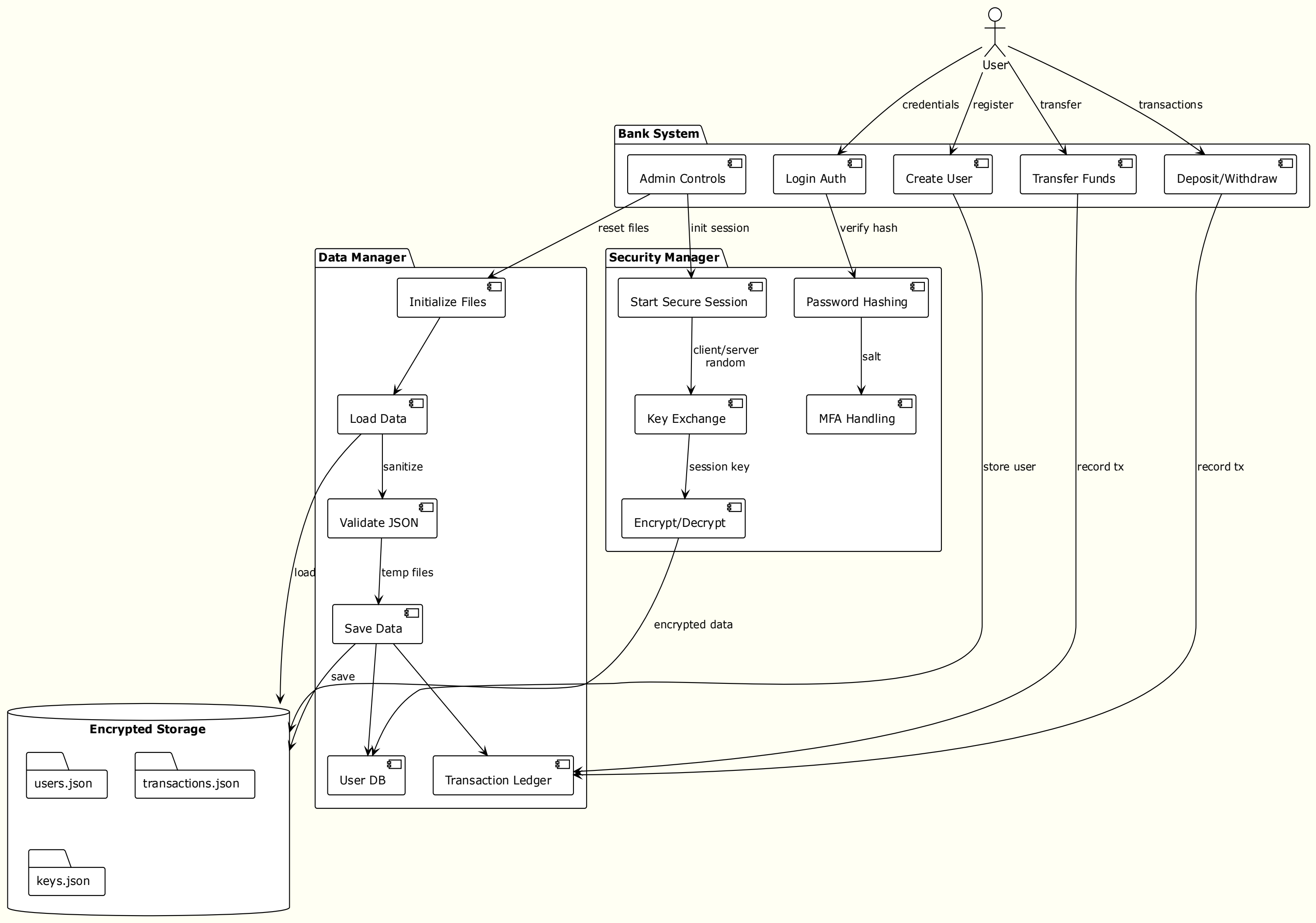


Fig. 2 Security Architecture

2.1 Architectural Overview

Designed MyBank security has a modular security architecture that separates responsibilities between three main components:

1. SecurityManager: Handles all cryptographic operations
2. DataManager: Manages secure data storage and retrieval
3. BankSystem: Controls access and coordinates system components

This separation of duties makes the code easier to understand and test, while also creating clear security boundaries between different system functions.

2.2 Cryptographic Method Selection

Selected cryptographic methods based on their security properties and suitability for a banking application are explained below.

1. Simulated TLS

The system establishes secure sessions through a TLS-like handshake process. This approach simulates the essential concepts of TLS without implementing the full complexity of certificate validation and authentication, which would be beyond the scope as we are not connecting to the internet and everything is being tested locally.

Here's how it works:

Steps:

a. The client and server each generate their own random 16-byte values
b. These random values are combined and fed through a key derivation function
c. The resulting key is used to encrypt all subsequent communications

2. Symmetric Encryption

For encrypting data, I've chosen the Fernet encryption scheme. This was done for the following reasons:

a. Speed (Symmetric encryption is much quicker than asymmetric alternatives)
b. Simplicity
c. It verifies message authenticity with included HMAC ensures encrypted messages haven't been tampered with

3. Password Security

For storing user passwords, I've implemented salted hashing using SHA-256. Each user has their own unique salt value, which prevents attackers from using rainbow tables attack.

4. Account Lockout Protection

The system triggers an account lockout mechanism after three failed login attempts. This helps protect against brute-force attacks by temporarily disabling accounts that might be under attack. Only administrators can unlock these accounts, adding an additional layer of security oversight.

5. Key Derivation Functions

For generating encryption keys from passwords, I'm using PBKDF2HMAC with a high iteration count. The high iteration count (100,000) is a deliberate design choice that makes the key derivation process computationally expensive. This should slow down brute-force attacks.

6. Multi-Factor Authentication

The system implements a simplest 4-digit code system for multi-factor authentication. This is just to demonstrate the concept of "something you know" (password) combined with "something you have" (the MFA code).

7. Transaction Integrity

To ensure transactions aren't modified after creation, I've implemented a simple integrity verification using SHA-256. By sorting the transaction items before hashing, it ensures consistent hash generation regardless of how the data is saved or loaded. Any modification to the transaction details would result in a different hash, allowing the system to detect tampering.

2.3 Security Trade-offs and Limitations

As this is a basic demonstration model for understanding the security principles, there are several practical trade-offs in this implementation that needs to be acknowledged:

1. Simple MFA
2. No minimum difficulty criteria for passwords: To make testing and understanding easy
3. Limited Session Management: No session timeouts for uninterrupted testing
4. Insecure Key Storage
5. Simplified TLS

3. Implementation

Firstly, this section explains the three main classes that form the foundation of the MyBank system.

1. SecurityManager Class

The SecurityManager class serves as the cryptographic core of the system, handling all security-related operations. This component encapsulates several critical security functions such as:

1. Secure Session Management: Establishes a simulated TLS connection by generating client and server random values, then deriving a session key through PBKDF2HMAC with SHA-256. (docs.python, n.d.) (cryptography.io, n.d.)
2. Encryption Operations: Implements Fernet symmetric encryption for protecting sensitive data during transmission and storage. The encrypt/decrypt methods handle data type conversion automatically, supporting both string and binary data formats. (cryptography.io, n.d.)
3. Authentication Services: Provides salted password hashing with SHA-256 and verification functions. Each user receives a unique salt value to prevent rainbow table attacks.
4. Access Control: Implements account lockout protection by tracking failed login attempts and blocking accounts after three failures. This significantly increases the system's resilience to brute force attacks.
5. Multi-Factor Authentication: Generates and validates one-time 4-digit verification codes, simulating an SMS/email second factor authentication mechanism.

2. DataManager Class

The DataManager class handles all persistent storage operations, creating a clear separation between security logic and data management: The detailed functions are:

1. File Initialization: Automatically creates necessary data files (users.json, transactions.json, keys.json) if they don't exist, ensuring the system can start without manual configuration.
2. Data Persistence: Provides methods for loading and saving structured data to JSON files, with basic error handling to maintain system stability.
3. User Management: Offers specialized methods for user operations such as checking existence, retrieval, and storage functions.
4. Transaction Recording: Maintains a complete audit trail of all financial operations with integrity protection through cryptographic hashing.
5. Key Storage: Handles the secure storage and retrieval of user encryption keys, maintaining a separate storage location from user credentials.

3. BankSystem Class

The BankSystem class integrates security and data components to provide complete banking functionality, such as:

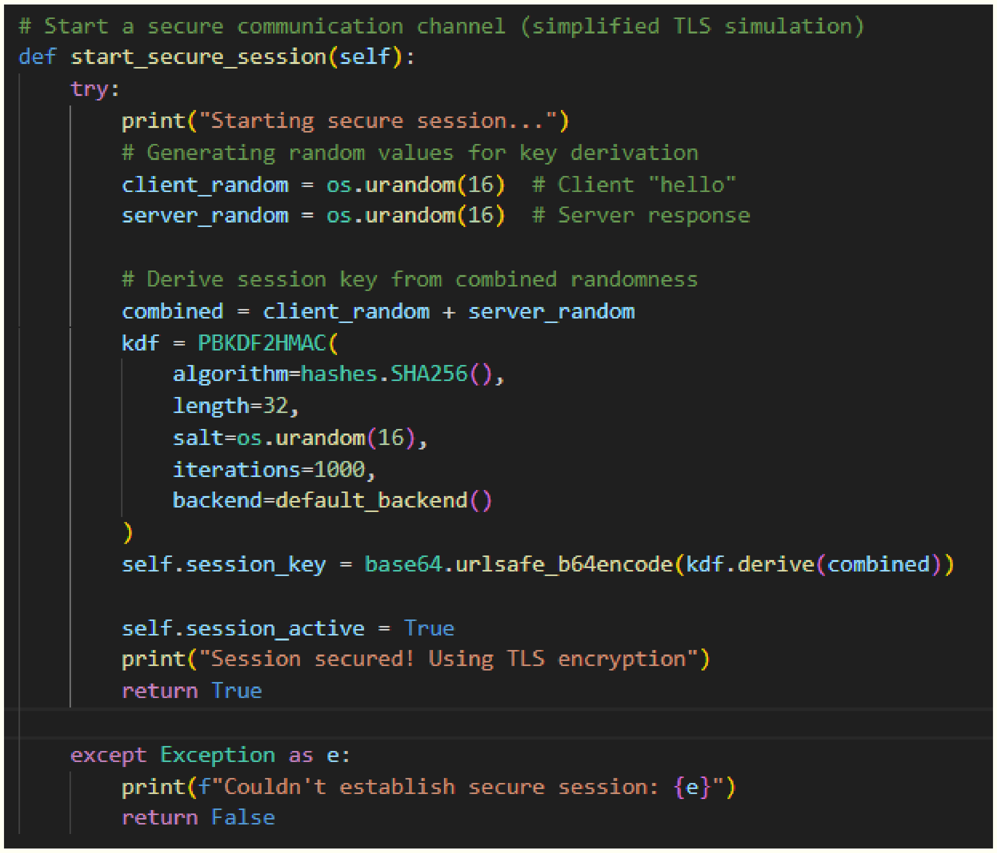
1. Role-Based Access Control: Implements distinct menus and capabilities for clients, employees, and administrators, enforcing the principle of least privilege.
2. User Authentication Flow: Coordinates the multi-step login process, including password verification, MFA validation, and account lockout checking.
3. Financial Operations: Provides methods for transfers, deposits, and withdrawals with appropriate validation checks and transaction recording.
4. User Administration: Offers capabilities for account creation, role management, and account recovery when users are locked out.
5. Menu Interface: Presents appropriate options based on user role, ensuring users only see and access functions they are authorized to use.

This modular architecture creates clear security boundaries between system components, making the code easier to understand, test, and maintain. Each class has specific responsibilities, working together to create a complete banking solution with layered security controls.

Now let me explains how the different cryptographic components are implemented in the codebase.

3.1 Secure Communication Workflow

The SecurityManager class establishes a simplified TLS-like secure channel:



```python
# Start a secure communication channel (simplified TLS simulation)
def start_secure_session(self):
    try:
        print("Starting secure session...")
        # Generating random values for key derivation
        client_random = os.urandom(16)  # Client "hello"
        server_random = os.urandom(16)  # Server response

        # Derive session key from combined randomness
        combined = client_random + server_random
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=os.urandom(16),
            iterations=1000,
            backend=default_backend()
        )
        self.session_key = base64.urlsafe_b64encode(kdf.derive(combined))

        self.session_active = True
        print("Session secured! Using TLS encryption")
        return True

    except Exception as e:
        print(f"Couldn't establish secure session: {e}")
        return False
```

Fig. 3 TLS-like method implementation

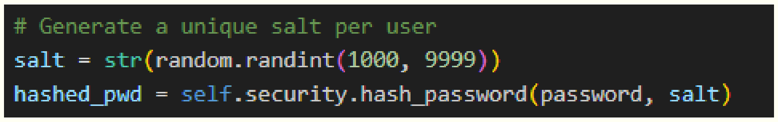
This function uses “os.urandom()” to generate cryptographically secure random values, which is important because predictable "random" values would compromise the entire encryption scheme.

This is a very limited simulation of a secure communication. As the entire system is offline and being run locally, there is no certificate verification mechanism.

3.2 Authentication Mechanisms

1. Password Hashing

The system creates a unique salt for each user and combines it with their password before hashing:

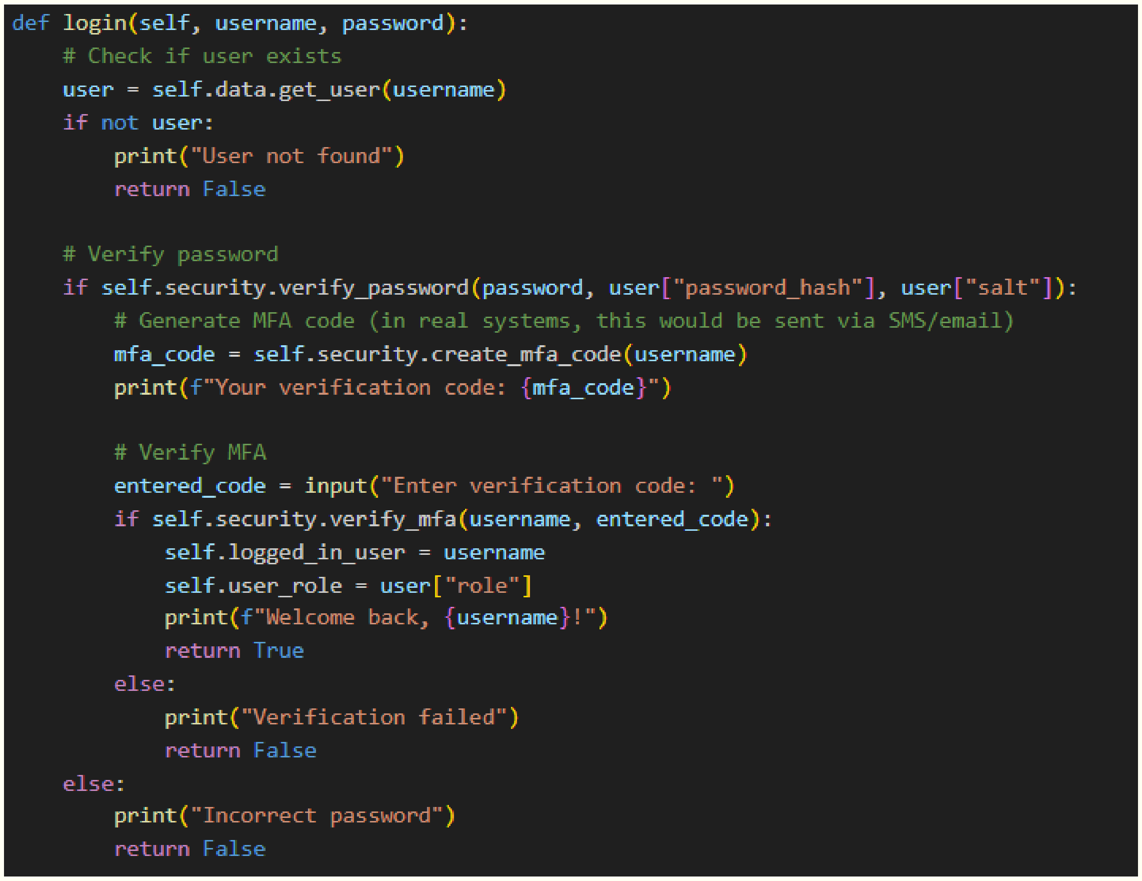


```
# Generate a unique salt per user
salt = str(random.randint(1000, 9999))
hashed_pwd = self.security.hash_password(password, salt)
```

Fig. 4 Hashing code snippet

2 Multi-Factor Authentication

The login process implements a basic MFA approach:



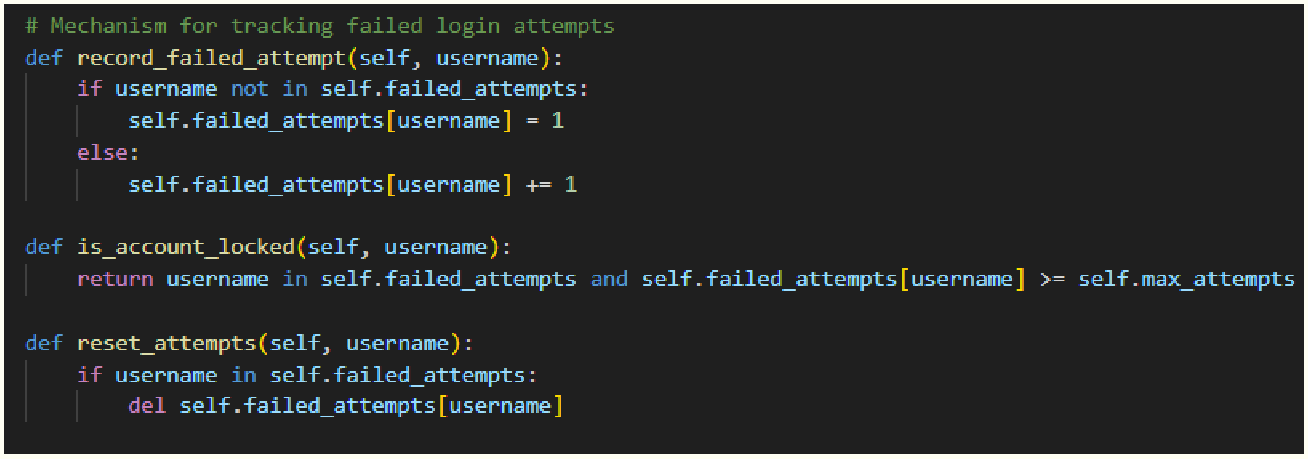
```
def login(self, username, password):
    # Check if user exists
    user = self.data.get_user(username)
    if not user:
        print("User not found")
        return False

    # Verify password
    if self.security.verify_password(password, user["password_hash"], user["salt"]):
        # Generate MFA code (in real systems, this would be sent via SMS/email)
        mfa_code = self.security.create_mfa_code(username)
        print(f"Your verification code: {mfa_code}")

        # Verify MFA
        entered_code = input("Enter verification code: ")
        if self.security.verify_mfa(username, entered_code):
            self.logged_in_user = username
            self.user_role = user["role"]
            print(f"Welcome back, {username}!")
            return True
        else:
            print("Verification failed")
            return False
    else:
        print("Incorrect password")
        return False
```

Fig. 5 MFA implementation

In a real system, the MFA code would be sent through a separate secure channel like SMS or email. The current implementation directly displays the code, which is suitable only for our purposes.

3. Account Lockout Protection

The system tracks failed login attempts and locks accounts after three failures:



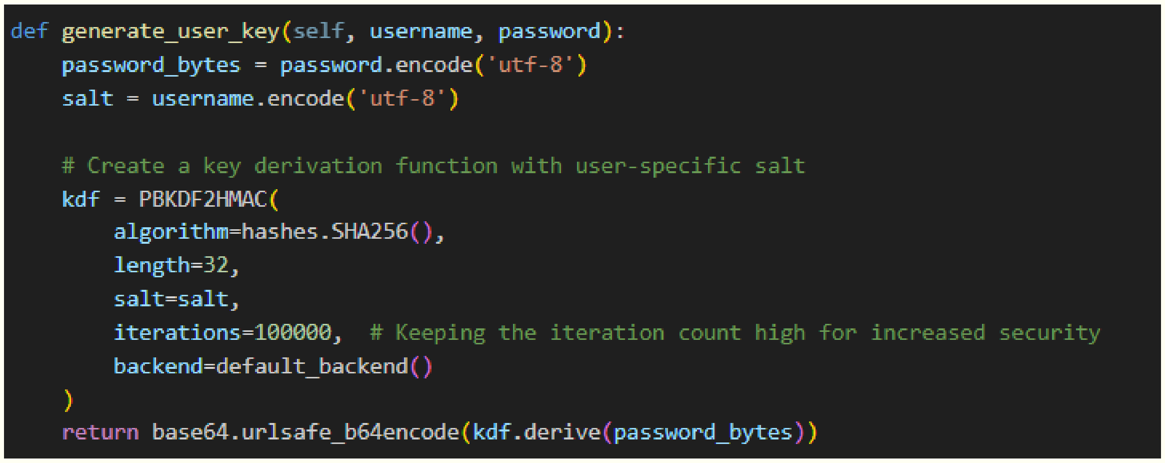
```python
# Mechanism for tracking failed login attempts
def record_failed_attempt(self, username):
    if username not in self.failed_attempts:
        self.failed_attempts[username] = 1
    else:
        self.failed_attempts[username] += 1

def is_account_locked(self, username):
    return username in self.failed_attempts and self.failed_attempts[username] >= self.max_attempts

def reset_attempts(self, username):
    if username in self.failed_attempts:
        del self.failed_attempts[username]
```

Fig. 6 Account locking mechanism

This mechanism helps prevent brute force attacks by detecting and blocking suspicious login activity. After an account is locked, only system administrators can unlock it through a dedicated menu option.

3.3 Key Management

User-specific encryption keys are generated based on their username and password:



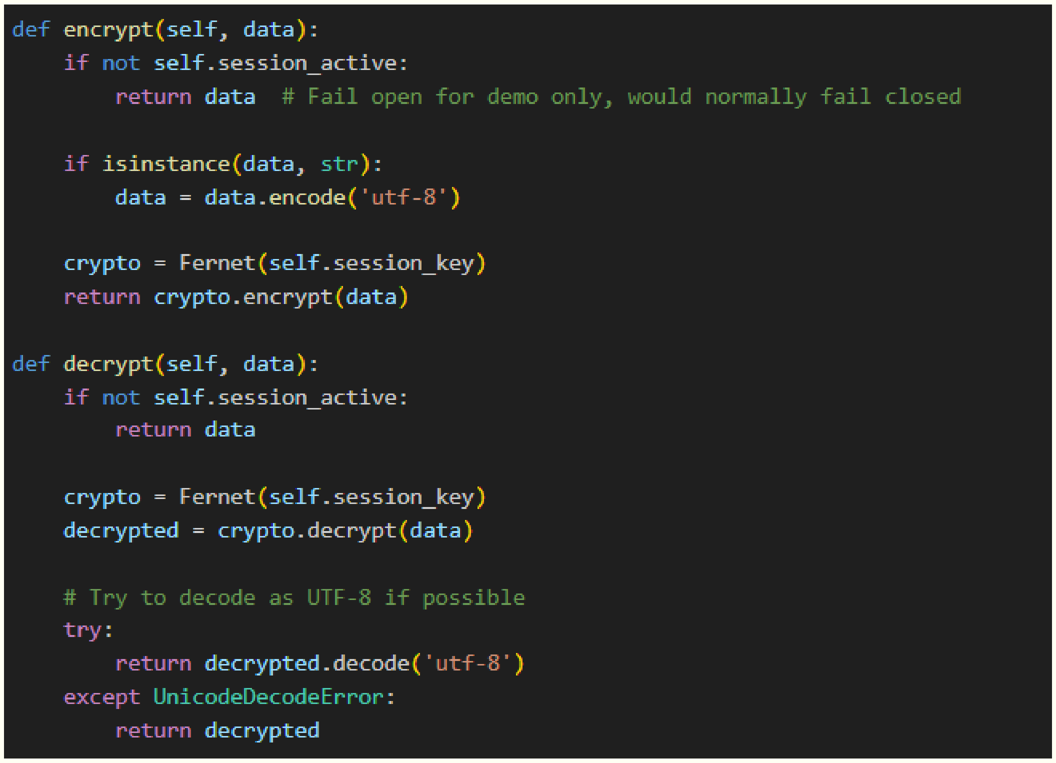
```python
def generate_user_key(self, username, password):
    password_bytes = password.encode('utf-8')
    salt = username.encode('utf-8')

    # Create a key derivation function with user-specific salt
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,  # Keeping the iteration count high for increased security
        backend=default_backend()
    )
    return base64.urlsafe_b64encode(kdf.derive(password_bytes))
```

Fig. 7 Key generation mechanism

These keys are stored in a separate .json files to be verified. These files are created automatically at the start of the code if not found already.

3.4 Data Encryption and Decryption

The system uses Fernet for symmetric encryption:



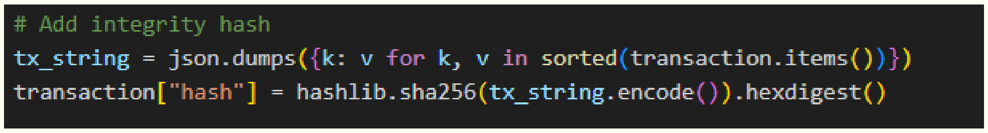
```python
def encrypt(self, data):
    if not self.session_active:
        return data  # Fail open for demo only, would normally fail closed

    if isinstance(data, str):
        data = data.encode('utf-8')

    crypto = Fernet(self.session_key)
    return crypto.encrypt(data)

def decrypt(self, data):
    if not self.session_active:
        return data

    crypto = Fernet(self.session_key)
    decrypted = crypto.decrypt(data)

    # Try to decode as UTF-8 if possible
    try:
        return decrypted.decode('utf-8')
    except UnicodeDecodeError:
        return decrypted
```

Fig. 8 Encryption and Decryption mechanism

A notable thing here is the "fail open" behaviour – if no secure session is active, the system returns unencrypted data rather than failing securely. This is a bad practice and was done here for testing and demonstration purpose only.

3.5 Transaction Integrity

Each financial transaction includes a cryptographic hash to detect tampering:



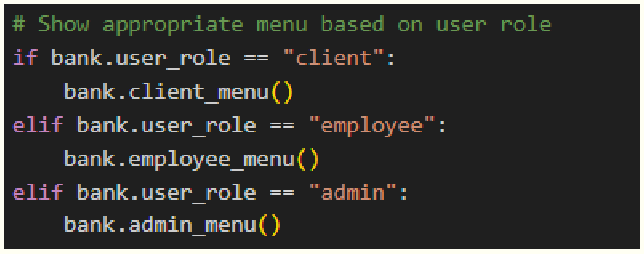
```python
# Add integrity hash
tx_string = json.dumps({k: v for k, v in sorted(transaction.items())})
transaction["hash"] = hashlib.sha256(tx_string.encode()).hexdigest()
```

Fig. 9 Transaction integrity check

This approach creates a digital fingerprint of the transaction data. If someone tries to modify the transaction details (like changing the amount), the hash would no longer match, revealing the tampering attempt.

3.6 Access Control

The system implements role-based access control through separate menu systems:



```python
# Show appropriate menu based on user role
if bank.user_role == "client":
    bank.client_menu()
elif bank.user_role == "employee":
    bank.employee_menu()
elif bank.user_role == "admin":
    bank.admin_menu()
```

Fig. 10 Access control menu

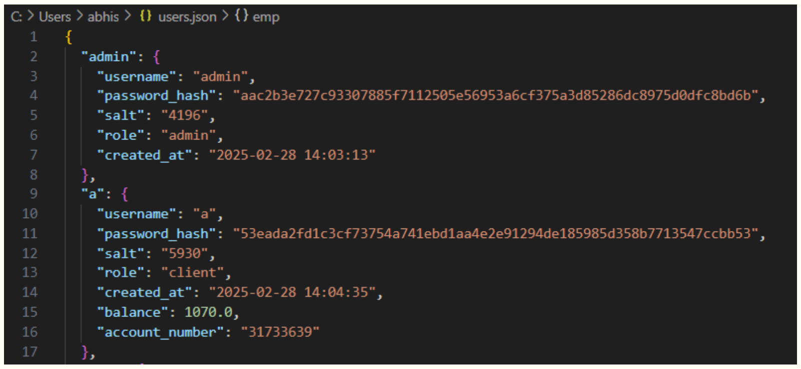
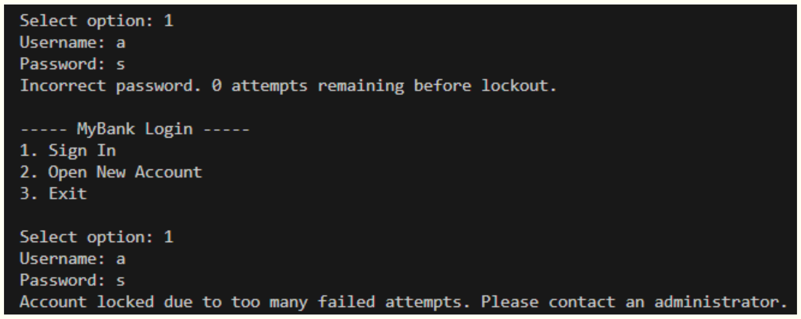
Each menu only provides access to operations appropriate for that role. For example:

1. Clients can view accounts, check transaction history, and make transfers
2. Bank employees can process deposits/withdrawals and view client information
3. System administrators can create users, manage roles, and unlock locked accounts

Another design choice was not to implement Deposit and Withdraw function in the client menu. As the system represents an app and not an ATM, it does not make sense to add those features. Only employee has the options to perform these actions.

4. Testing

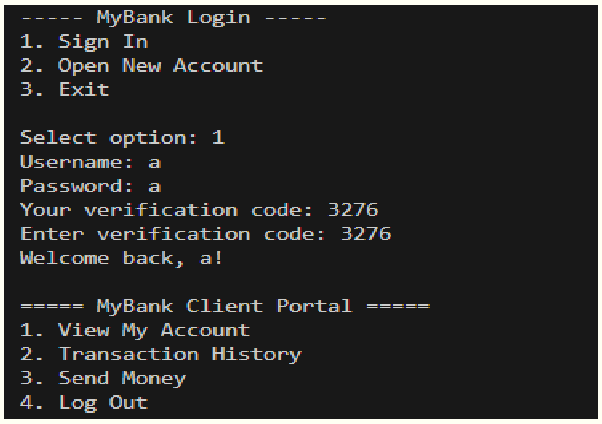
4.1 Core Security Testing

Test Scenario	**Test Description**	**Expected Outcome**
Password Security	Create user and verify password storage	Passwords are stored as salted hashes, not plaintext  *Fig. 11 Password hash storage with salt and other details*
Account Lockout Protection	Enter incorrect password 3 times	Account is locked after 3 failed attempts and requires admin to unlock  *Fig. 12 Account locking after 3 unsuccessful attempts*
Two-Factor Authentication	Login with valid password and verify MFA flow	User must enter correct one-time code to complete authentication

```
C: > Users > abhis > {} users.json > {} emp
{
  "admin": {
    "username": "admin",
    "password_hash": "aac2b3e727c93307885f7112505e56953a6cf375a3d85286dc8975d0dfc8bd6b",
    "salt": "4196",
    "role": "admin",
    "created_at": "2025-02-28 14:03:13"
  },
  "a": {
    "username": "a",
    "password_hash": "53eada2fd1c3cf73754a741ebd1aa4e2e91294de185985d358b7713547ccbb53",
    "salt": "5930",
    "role": "client",
    "created_at": "2025-02-28 14:04:35",
    "balance": 1070.0,
    "account_number": "31733639"
  },
```

```
Select option: 1
Username: a
Password: s
Incorrect password. 0 attempts remaining before lockout.

----- MyBank Login -----
1. Sign In
2. Open New Account
3. Exit

Select option: 1
Username: a
Password: s
Account locked due to too many failed attempts. Please contact an administrator.
```

Test Scenario	**Test Description**	**Expected Outcome**

		
		```
		----- MyBank Login -----
		1. Sign In
		2. Open New Account
		3. Exit

		Select option: 1
		Username: a
		Password: a
		Your verification code: 3276
		Enter verification code: 3276
		Welcome back, a!

		===== MyBank Client Portal =====
		1. View My Account
		2. Transaction History
		3. Send Money
		4. Log Out
		```
		 Fig. 13 Workings of authentication using MFA code

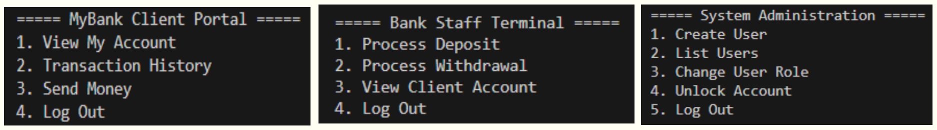
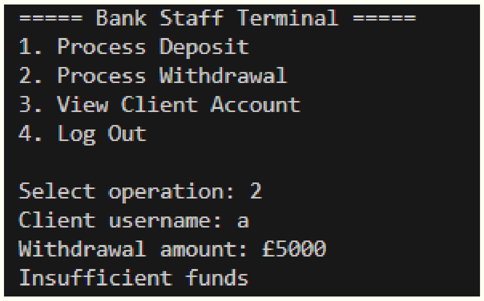
Role-Based Access Control	Log in with different user roles	Each role can only access appropriate menus and operations 
		```
		===== MyBank Client Portal =====
		1. View My Account
		2. Transaction History
		3. Send Money
		4. Log Out
		```
		```
		===== Bank Staff Terminal =====
		1. Process Deposit
		2. Process Withdrawal
		3. View Client Account
		4. Log Out
		```
		```
		===== System Administration =====
		1. Create User
		2. List Users
		3. Change User Role
		4. Unlock Account
		5. Log Out
		```
		 Fig. 14 Client, Employee and Admin menu respectively

Table 1 Core Security Testing

4.2 Client Operations Testing

Test Scenario	**Test Description**	**Expected Outcome**
Funds Transfer	Send money between two client accounts	Sender's balance decreases, recipient's increases, transaction is recorded with integrity hash
Deposit Processing	Deposit funds into client account	Balance increases and transaction is recorded correctly
Withdrawal - Sufficient Funds	Withdraw from account with adequate balance	Balance decreases and transaction is recorded correctly

Withdrawal - Insufficient Funds	Attempt to withdraw more than available	Operation fails with appropriate message, balance unchanged
		
		`===== Bank Staff Terminal =====`
		`1. Process Deposit`
		`2. Process Withdrawal`
		`3. View Client Account`
		`4. Log Out`
		
		`Select operation: 2`
		`Client username: a`
		`Withdrawal amount: £5000`
		`Insufficient funds`
		Fig. 15 Inefficient fund message

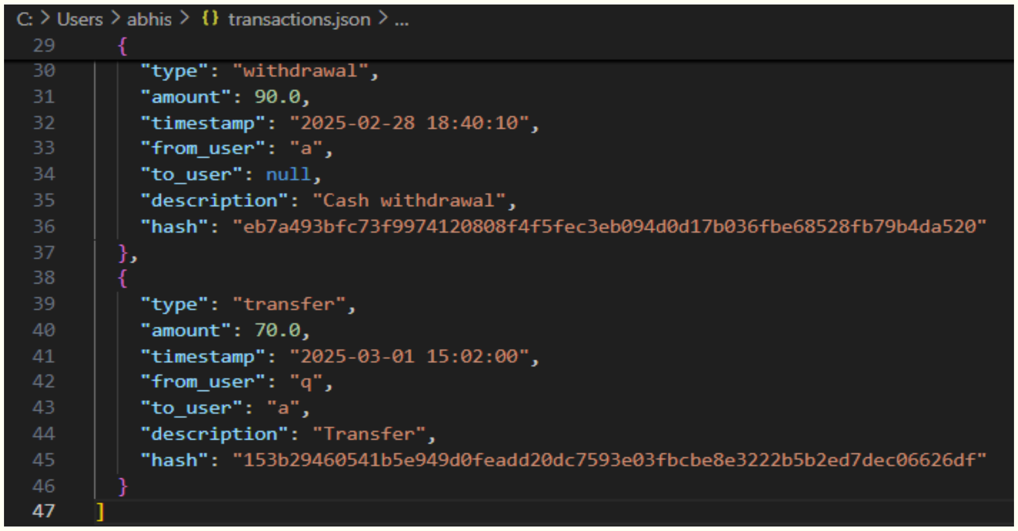
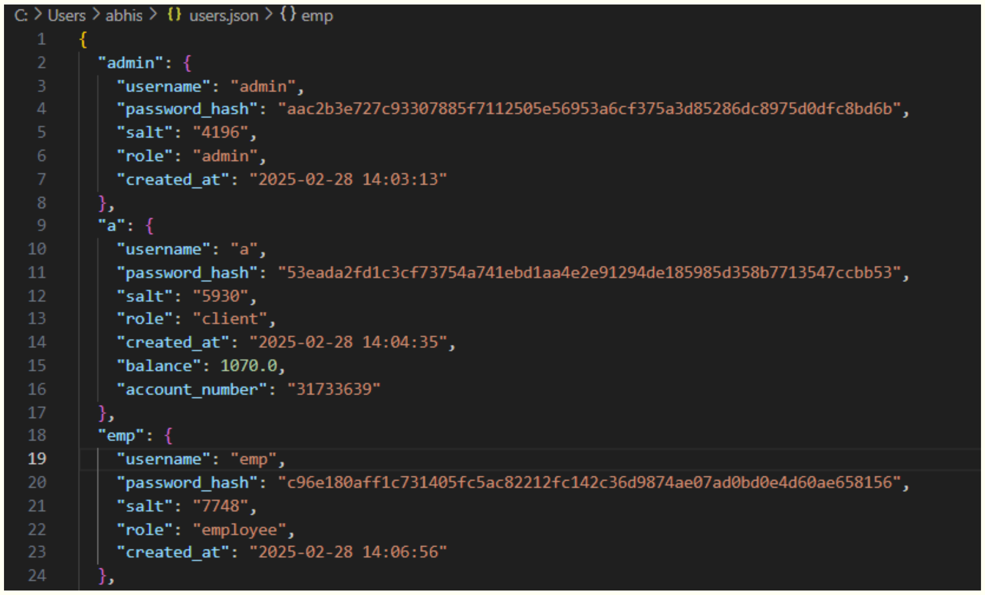
Transaction Integrity	View transaction history and verify hash protection	All transactions are shown with proper details and protected from tampering
		
		C: > Users > abhis > {} transactions.json > ...
		`    {`
		`      "type": "withdrawal",`
		`      "amount": 90.0,`
		`      "timestamp": "2025-02-28 18:40:10",`
		`      "from_user": "a",`
		`      "to_user": null,`
		`      "description": "Cash withdrawal",`
		`      "hash": "eb7a493bfc73f9974120808f4f5fec3eb094d0d17b036fbe68528fb79b4da520"`
		`    },`
		`    {`
		`      "type": "transfer",`
		`      "amount": 70.0,`
		`      "timestamp": "2025-03-01 15:02:00",`
		`      "from_user": "q",`
		`      "to_user": "a",`
		`      "description": "Transfer",`
		`      "hash": "153b29460541b5e949d0feadd20dc7593e03fbcbe8e3222b5b2ed7dec06626df"`
		`    }`
		`  ]`
		Fig. 16 Transaction details with hash

Table 2 Client Operations Testing

4.3 User Management Testing

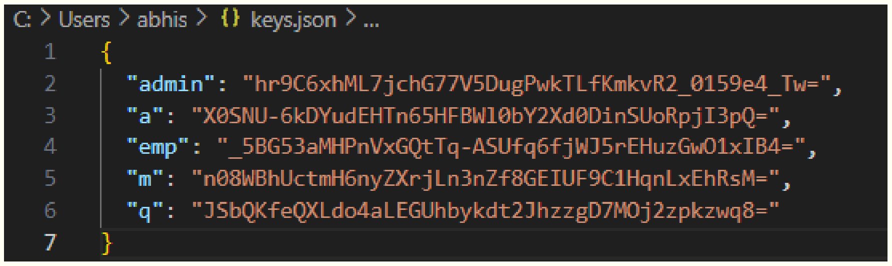
Test Scenario	**Test Description**	**Expected Outcome**
New User Registration	Create accounts for all three roles	Accounts are created with appropriate permissions and fields  *Fig. 17 Different types of accounts and their details in DB*
User Authentication	Log in with valid credentials	User is authenticated and shown appropriate menu
Account Recovery	Admin unlocks a locked user account	Account becomes accessible again after admin intervention
Role Modification	Change a user's role	User's access rights change according to new role

```
C: > Users > abhis > {} users.json > {} emp
{
  "admin": {
    "username": "admin",
    "password_hash": "aac2b3e727c93307885f7112505e56953a6cf375a3d85286dc8975d0dfc8bd6b",
    "salt": "4196",
    "role": "admin",
    "created_at": "2025-02-28 14:03:13"
  },
  "a": {
    "username": "a",
    "password_hash": "53eada2fd1c3cf73754a741ebd1aa4e2e91294de185985d358b7713547ccbb53",
    "salt": "5930",
    "role": "client",
    "created_at": "2025-02-28 14:04:35",
    "balance": 1070.0,
    "account_number": "31733639"
  },
  "emp": {
    "username": "emp",
    "password_hash": "c96e180aff1c731405fc5ac82212fc142c36d9874ae07ad0bd0e4d60ae658156",
    "salt": "7748",
    "role": "employee",
    "created_at": "2025-02-28 14:06:56"
  },
```

Table 3 User Management Testing

4.4 Data Management Testing

Test Scenario	**Test Description**	**Expected Outcome**
Data Persistence	Perform operations, restart application, verify data	All data (users, transactions, keys) persists correctly between sessions

User Key Management	Verify encryption key generation	Each user has a unique encryption key stored in keys.json
		
		C: > Users > abhis > {} keys.json > ...
		`{`
		`  "admin": "hr9C6xhML7jchG77V5DugPwkTLfKmkvR2_0159e4_Tw=",`
		`  "a": "X0SNU-6kDYudEHTn65HFBW10bY2Xd0DinSUoRpjI3pQ=",`
		`  "emp": "_5BG53aMHPnVxGQtTq-ASUfq6fjWJ5rEHuzGwO1xIB4=",`
		`  "m": "n08WBhUctmH6nyZXrjLn3nZf8GEIUF9C1HqnLxEhRsM=",`
		`  "q": "JSbQKfeQXLdo4aLEGUhbykdt2JhzzgD7MOj2zpkzwq8="`
		`}`
		Fig. 18 Stored keys in DB

Error Handling	Test with invalid inputs (text for numbers, etc.)	System handles errors without crashing

Table 4 Data Management Testing

5. Security Analysis

This section checks how well the implementation meets the original security requirements:

1. Data Encryption: Fully met through Fernet encryption.
2. Secure Communication: Partially met through the TLS-like session implementation, but lacks proper certificate validation.
3. Key Management: Fully met through key generation and storage (but keys are stored as plaintext for demo)
4. User Authentication: Fully met through password hashing and MFA (However due to its simple nature, it is vulnerable to the brute force attack. More robust mechanism is recommended for the actual production system)
5. Transaction Integrity: Successfully met through hash-based integrity verification.

6. Conclusion & Recommendations

MyBank successfully demonstrates and achieves all set security objectives including:

1. Confidentiality
2. Integrity
3. Authentication
4. Non-repudiation

6.1 Implementation Challenges

During development, I encountered several practical challenges:

1. Creating simulation of TLS principles without full certificate infrastructure
2. Ensuring data integrity across file operations
3. Managing encryption keys securely while maintaining legitimate access
4. Handling error recovery after data corruption

6.2 Recommendations

1. Fix the fail-open behaviour
2. Implement proper TOTP
3. Minimum password difficulty
4. Upgrade key management to use hardware security modules (HSMs)
5. Add session expiry
6. Replace JSON files with a proper database

Many of these improvements could have been integrated in the code, however, due to the limited time constrain I could not.

References

cryptography.io. (n.d.). *Fernet (symmetric encryption)*. Retrieved from https://cryptography.io/en/latest/fernet/

cryptography.io. (n.d.). *Key derivation functions*. Retrieved from https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/

docs.python. (n.d.). *hashlib*. Retrieved from https://docs.python.org/3/library/hashlib.html

Appendix

A1 Running instructions

Setup Instructions

Prerequisites:

Python 3.8+ installed on your system
OS: Windows, MAC, Linux

Setup Instructions:

Step 1: Install Required Libraries

Open a command prompt or terminal and install the necessary Python packages:

$$pip\ install\ cryptography$$

Step 2: Prepare the Files

1. Copy the Code.py file to your desired directory
2. Create an empty directory where you want to store the banking data

Step 3: First Run Setup

1. Open a command prompt or terminal
2. Navigate to the directory where you placed the Python file
3. Run the program with:

$$python\ "Code.py"$$

4. The system will automatically:
   a. Create required data files (users.json, transactions.json, keys.json)
   b. Set up a secure session
   c. Create a default admin account with credentials:

      Username: admin

      Password: admin123

Step 4: Logging In

1. Choose option 1 to sign in
2. Enter the default admin credentials:

   Username: admin

   Password: admin123
3. The system will generate and display a 4-digit MFA code
4. Enter the displayed MFA code to complete authentication

Step 5: Creating Additional Users

As an admin, you can:

1. Choose option 1 from the admin menu to create new users
2. Select the appropriate role:

   1 for Client (banking customers)

   2 for Employee (bank staff)

   3 for Admin (system administrators)

Important Notes

1. Data Security: The system stores data in three JSON files:
   a. users.json - User account information
   b. transactions.json - Transaction records
   c. keys.json - Encryption keys

2. Account Lockout: After 3 failed login attempts, accounts will be locked and require admin intervention to unlock.

3. Test Environment: This system is primarily for demonstration purposes and uses simulated security features.

Troubleshooting

1. "File not found" errors: Make sure you have the proper write permissions in the directory where the program is running.
2. Login issues: If your account gets locked, you'll need to have an admin unlock it using the "Unlock Account" option in the admin menu.
3. Encryption errors: If you move the program between machines with existing data files, encryption keys may not work properly. It's best to start with fresh data files on new machines.