FIT5136 – Information Computer Security.

Group_StudentID: {34648933 – Christopher (crystal) Bailon/ GIT Saint-Smooth, 33578273 - Mohamad Abdul Karim Jabri/ GIT KarimJabri01, 33786054 – Behnam Tafreshnia / GIT BenT108}.
GIT REPO: https://github.com/KarimJabri01/FIT50003 /
History: https://github.com/KarimJabri01/FIT50003/commits/main/

Topic 12: "EMV *card payment protocol. Implement a demo of the Eurocard / Mastercard / Visa (EMV) credit chip card payment protocol.* "

Design: When creating a protocol, such as EMV speed and size needs to be prioritised. This can be attained by languages that are statically typed prioritising speed at the cost of possible mistakes. EMV protocols will often run on a variety of computers, mainly small Linux machines without a Graphic User Interface that often does not have any memory to spare. Thus, C++ becoming our language of choice
EMV, which stands for "Eurocard, MasterCard, and Visa,". At its core, EMV defines the communication rules between a card, terminal, and bank or ATM. The technology, now commonly associated with embedded chips in debit and credit cards, allows for cashless payments, including contactless methods. Given the exchange of sensitive information, EMV protocols require robust security across multiple systems, all working together to verify and authorise transactions in everyday payments.
As described by Basin, Sasse, and Toro-Pozo (2021, p. 3-6), An EMV protocol works in 4 different stages:

- Initialisation
- Off-line Data Authorisation (ODA)
- Cardholder authentication.
- Transaction Authorisation.

Explaining these stages further, at the initialisation phase many things occur. At first the terminal will issue a select command to agree on what application to use (contact, contactless). Secondly, a series of data will be exchanged. The card will respond to the select command with a list of Application Identifiers (AIDs). It may also request the Processing Data Object List (PDOL), which includes terminal-sourced data like transaction *amount, currency, date, and a terminal-generated random number* (Unpredictable Number, TUN).
Thirdly, The terminal sends a GET PROCESSING OPTIONS command along with PDOL data *if requested*. The card responds with the Application Interchange Profile (AIP) and the Application File Locator (AFL). **Thereafter,** the terminal reads the necessary records from the card, which may include the Primary Account Number (PAN), expiration date, certificates, Card Risk Management Data Object Lists (**CDOL1 and CDOL2**), and the list of supported Cardholder Verification Methods **(CVMs**). Finally, **Certificate Validation**. The terminal retrieves the Certificate Authority's (CA) public key, verifies the bank's certificate, and then acquires and verifies the card's public key certificate.
In stage two, the authentication method is handled: We have decided to create our EMV protocol using dynamic data authentication. This means that the terminal will send a unique challenge *to the card,* which responds with a signed version of such. We have decided on **Dynamic Data Authentication (DDA)** which entails the terminal sending a unique challenge to the card, which responds with a signed dynamic data object (SDAD), including transaction details, preventing cloning and data modification. We would have decided on **Combined Data authentication**, which adds an

extra layer of information to **DDA** with shop-specific data. But it is difficult to simulate in a local demonstration.
Across stage three, the cardholder's ***legitimacy is proven.*** Legitimacy is proven by presenting the cardholder with a variety of challenges. In this case, we have offered two options: 1) a 4-digit integer or PIN, and 2) an 8-16 long Password.
Finally, at stage four, validation is handled: Validation occurs when the terminal decides whether to:

- Decline the transaction offline,
- Approve the transaction offline,
- Request online authorization from the bank.

Once it's finished, the card generates an Application Cryptogram (AC) based on this decision:

- Application Authentication Cryptogram (AAC): If the transaction is declined.
- Transaction Cryptogram (TC): If the transaction is approved offline.
- Authorization Request Cryptogram (ARQC): If online authorization is needed.

Due to the lack of an online authorisation component in our demonstration, ARQC will not be included, as to potentially be able to run all the programs in one device.
Across these ***four stages***, our protocol allows the communication of the bank, the card, and the terminal, which all work in unison to challenge, verify, and store information.

Implementation:

Our implementation follows a series of steps to ensure a robust and secure EMV protocol. To achieve both security and ease of use, we adopted an object-oriented programming approach, utilising four key classes: `UserData`, `Card`, `Bank`, and `Terminal`. These classes work together to handle the core functionality of the protocol. Additionally, we defined global variables and functions that provide general utility throughout the program, supporting the seamless operation of the system.

Class `UserData`:

The `UserData` class is primarily responsible for storing and verifying user data. While its main functionality supports the bank and potential future enhancements, it plays a relatively minor role in the current implementation. Since the program focuses on the payment protocol and interactions between the three key entities—Bank, Terminal, and Card—the `UserData` class is not extensively utilised.
This class takes three arguments—`firstName`, `lastName`, and `address`—as input and generates an account number (`acc_num`) upon instantiation. The account number is derived from the last account number stored in the database, ensuring uniqueness.

Class `Card`:

The `Card` class is responsible for creating a card and validating stored data. It primarily acts as a data storage unit, but it also ensures the validity of the input before saving it. To instantiate a `Card`, several arguments are required: `in_card_number`, `in_cvv`, `in_exp_date`, `in_currency`, and `in_account_nb`. Each of these values are validated using various class methods, as described below:

- **`void validateCVV(int cvv)`**: This method takes an integer (CVV) as input and checks if it contains exactly 3 digits, following the criteria for Visa and MasterCard/EuroCard. If the CVV does not meet this requirement, it throws an error: "CVV code is incorrect".

- **`void validateExpirationDate(const std::string &exp_date)`**: This method accepts a string representing the expiration date and checks if the card has expired by comparing the input date with the current date. If the card is expired, it throws an error: "Your card is expired".
- **`void check()`**: This method validates the card number by performing several checks:
    1. Ensuring the card number is not empty.
    2. Verifying the card number's length is between 13 and 19 digits.
    3. Checking for invalid characters (non-digits) in the card number.
    4. Validating the card number using the Luhn Algorithm.
    5. If any of these checks fail, appropriate error messages are raised.

Additionally, the `Card` class provides several getter methods that allow other classes to access its private data members. These methods return the requested values encrypted using RSA encryption with the `BANK_PUBLIC_KEY`, ensuring that only the bank can decrypt them using its private key.

Class `Bank`:

The `Bank` class provides core functionality for encrypting, decrypting, authenticating, and validating data. It contains several private variables for security purposes, such as `publicKey` and `privateKey` for RSA encryption. The `publicKey` is also stored in a global variable, `BANK_PUBLIC_KEY`, to facilitate secure communication between the bank and other components. Other private members include `secretKey` and `iv`, which are used for symmetric AES encryption to secure the database. The encrypted database, `encDB`, stores card details in the form of a vector of `CardDetails` structs. The class methods are publicly accessible, allowing other classes to perform validation and authentication. Upon instantiation, the following sequence of actions occurs:

- RSA keys are generated using the `generateBankRSAKeys()` method.
- The `publicKey` is stored in a global variable via the `GetPublicKey()` method.
- The `bankEncryptCSV()` function is invoked to encrypt the database (stored as a CSV for demonstration purposes) using AES encryption with the bank's `secretKey` and `iv`.

Methods:

- **`bool cardAuthentication(Card &card)`**: This method takes a `Card` object as input, checks if the card exists in the encrypted database, decrypts the card's authentication method and credentials (PIN or password), and passes them to the `Authenticate()` method.
- **`bool Authenticate(const std::string& authMeth, const std::string& decPass)`**: This method authenticates the card by asking the user for the PIN or password. The user has three attempts to input the correct value, which is validated against the stored credentials in the database.
- **`void balanceChange(Card &card, double cost)`**: This method checks if a card exists in the database and ensures that it has sufficient balance to cover a transaction. If so, the transaction amount is deducted from the balance, and the updated balance is encrypted and saved back into the database (`encDB`).

- **`bool checkCardDetail(const Card& card)`**: Validates the card details against the data stored in the encrypted database.
- **`CryptoPP::RSA::PublicKey GetPublicKey()`**: Returns the bank's public RSA key.
- **`CryptoPP::SecByteBlock GetIV()`**: Returns the AES initialization vector (`iv`).
- **`void printKeys()`**: Prints the `publicKey` and `privateKey` for demonstration purposes. Before printing, the keys are encoded in Base64 format using the helper methods `std::string encodeKeyToBase64(const CryptoPP::RSA::PublicKey& key)` and `std::string encodeKeyToBase64(const CryptoPP::RSA::PrivateKey& key)`.

Class `Terminal`:
The `Terminal` class primarily functions as a user interface, displaying data and collecting user inputs to be sent to the bank for processing. It has two key components:

- **Struct `Transaction`:** This structure holds the following information related to each transaction:
  - `TUN` (Terminal Unique Number)
  - `currency`
  - `data` (transaction-related information)
  - `transaction_type` (contact or contactless)
  - `transaction_date`
- **`Transaction_data`:** A list of `Transaction` values storing the details of each transaction.

Methods:

- **`void validate_authentication(Bank &bank, Card &card)`**: This method takes a `Bank` object and a `Card` object as inputs. It calls the bank's `cardAuthentication()` method to authenticate the card being used for the transaction.
- **`void add_transaction(const std::string& curr, const std::string& data, std::string& date)`**: This method initiates the transaction process. It takes the transaction currency, data, and date (current date and time) as inputs. A random TUN is generated using the `generateTUN()` function, and the `selectTransactionType()` method is called to determine whether the transaction is cardless or contact-based.
- **`std::string selectTransactionType()`**: This method prompts the user to choose between two transaction types: `1` for contactless and `2` for contact. Based on the input, it sets the transaction type.
- **`void display_transactions()`**: This method displays all transaction data after a transaction has been successfully completed.

There are some global functions including security-related ones, which are included in the Security Analysis part of this report, and GetCurrentDateTime(), which returns the current date and time.

“Security Analysis”

We have implemented several security measures in this system to guarantee the integrity and confidentiality of sensitive cards and transaction data. Our main focus is on using encryption algorithms, secure authentication methods, and safe handling of user inputs and sensitive user information. Security Components implemented in the code:

Encryption Algorithms:

RSA Encryption:

- Used for asymmetric encryption, and to encrypt sensitive card details such as the card number, CVV, account number, expiration date, and currency.
- Uses a public key for encryption and a private key for decryption. In our case the bank holdes the private key, while the public key is used by the system to encrypt card data.
- Ensure secure data transmission, only the bank can decrypt the given information using teh private key.
- Prevent unauthorised users from accessing sensitive card details.

AES Encryption:

- Used for symmetric encryption of transaction related data (card balance)
- Uses a secret key and an initialization vector (iv) for encryption and decryption.
- All sensitive card-related fields like the balance, authentication method, and authentication credentials are encrypted using AES before storage in the encrypted database.

Secure Authentication Mechanism:

- Authentication Based on Card Details:
    - The System validates the card details by decrypting them and comparing them against the stored values
    - Protects the authentication process by ensuring that only authorized users can access the card
- Password Hashing Using SHA-384:
    - Hashes passwords using the SHA-384 algorithm to transform them into a secure hash
    - Ensures that the passwords are stored securely in their hashed form to provide protection if the database is breached

Encrypted Database:

- All the sensitive card details information are stored in an encrypted database.
- Before being added to the database, the data is encrypted using AES with a secret key and an initialization vector.
- This action prevents unauthorised parties from accessing the database.

Secure Key Management:

- **Private/Public Key Management for RSA:**
    - RSA private key is stored in the bank and used to decrypt sensitive card details.
    - Prevent unauthorised access to the cardholder private information
- **Secret Key and IV for AES**
    - AES encryption depends on securely managing the secret key and initialization vector (iv).
    - These must be protected from exposure to prevent unauthorised decryption of the database or card data.

Safe Input Handling:

- The implementation of input validation in order to prevent a buffer overflow attack or injection attacks, during user authentication and transaction handling.

- When selecting a transaction type, the system ensures that only valid inputs are accepted, and any invalid inputs are handled properly.

Reasons for these encryptions and libraries:

- Why RSA: Ideal for security exchanging information because only the holder has the private key (in our case the bacnk) and use it to decrypt the data.
- Why AES:Efficient and secure for large amounts of data because the same key is used for encryption and decryption which is efficient for large amounts of data to encrypt the database.
- Why Crypto++ Library: Robust cryptographic library, reliable and has good performance.

References:

Basin, D. A., Sasse, R., & Toro-Pozo, J. (2021). The EMV standard: Break, fix, verify. *Proceedings of the IEEE Symposium on Security and Privacy*, 1766–1781. https://doi.org/10.1109/SP.2021.00109