

**Project Title:** IMPLEMENTATION OF RSA OVER TCP & CREDIT CARD VAULT PROJECT

**Group Work**

**Name:** Roy Tanga **Student ID:** 666080

**Name:** Nellie Nduati **Student ID:** 6658003

**Name:** Hongki Kim **Student ID:** 659070

**Name:** Jodick Ndayisenga **Student ID:** 666225

**Name:** Monyjang Manyuat **Student ID:** 666238

**Course Name:** Cryptography and Network Security (APT 3090-A)

**Instructor:** Prof. Githinji, Stanley; PhD

**Semester:** Summer 2024

**Date:** July 2024

**Table of Contents**

[**ABSTRACT** 3](#_Toc173086975)

[**PART 1: RSA over TCP** 4](#_Toc173086976)

[**Introduction** 4](#_Toc173086977)

[**Implementation of RSA over TCP** 4](#_Toc173086978)

[**Overview** 4](#_Toc173086979)

[**How it is implemented in Python** 5](#_Toc173086980)

[**Results and Discussions** 6](#_Toc173086981)

[**Output:** 9](#_Toc173086982)

[**Discussions** 11](#_Toc173086983)

[**PART 2: Credit Card Vault System** 11](#_Toc173086984)

[**Background** 11](#_Toc173086985)

[**System Requirements:** 11](#_Toc173086986)

[**Database Schema:** 11](#_Toc173086987)

[**USER ACCESS LEVELS** 12](#_Toc173086988)

[**Implementation of the Credit Card Vault System** 13](#_Toc173086989)

[**Creating the Database** 13](#_Toc173086990)

[**Implementing Encryption of the Data and Decryption** 13](#_Toc173086991)

[**User Credential Authentication** 13](#_Toc173086992)

[**Discussions of the Output for Credit Card System:** 14](#_Toc173086993)

[**Outputs of Database Implementation for Card Vault System** 15](#_Toc173086994)

[**References** 17](#_Toc173086995)

[**Appendix: The credit card vault schema** 18](#_Toc173086996)

[**Appendix: Structured Query Language (SQL Syntax)** 32](#_Toc173086997)

# **ABSTRACT**

The two projects implemented in this lab report are the creation of a Credit Card Vault system for the safe management and storage of client credit card information and the use of the RSA algorithm over TCP for secure communication between Bob and Alice. The Credit Card Vault system focusses on the safe storage and retrieval of sensitive credit card data via access control and encryption, while the RSA project showcases the real-world application of public-key cryptography in creating secure communication channels between two parties. The use of Python for the RSA implementation and the integration of a database management system for the Credit Card Vault system are highlighted in the study, which offers thorough descriptions of the techniques, implementations, and outcomes of both projects.

# **PART 1: RSA over TCP**

# **Introduction**

The objective of this group project is to implement the RSA algorithm with a key size of 1024 bits and demonstrate the encryption and decryption process using a TCP client-server model. The implementation consists of two main scripts, Alice and Bob, where Alice generates RSA keys and sends the public key to Bob over a TCP connection. Bob uses the public key to encrypt a message and sends the cipher text back to Alice for decryption.

Data storage and safe communication are critical in today’s digital environment. A popular public-key cryptosystem, the RSA algorithm uses asymmetric encryption to provide secure communication over unreliable networks. In contrast, the Credit Card Vault solution guarantees the security and integrity of the data by addressing the requirement for the safe administration and storage of sensitive credit card information.   
  
The implementation specifics of both projects are presented in this lab report, which focusses on safe system design and the real-world use of cryptographic algorithms. In the RSA project, secure channels are established using Python and TCP connections; in the Credit Card Vault system, we have integrated systems that we are very familiar with and as the database management system, and access control and encryption are implemented.

# **Implementation of RSA over TCP**

## **Overview**

A crucial component of modern cryptography is the RSA algorithm, which functions as a public-key cryptosystem to enable safe communication between two parties across an unreliable network. RSA was created in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman hence the acronym “RSA” from the names of its creators. It works by using the mathematical characteristics of big prime numbers to produce a pair of keys: a private key that is kept secret and a public key that may be distributed widely. Because it is still computationally impossible to factor the product of two huge prime numbers with present technology, the security of RSA depends on this practical problem. This key combination makes it possible to encrypt and decrypt messages, guaranteeing the safe transmission of sensitive data.

Confidentiality and integrity are ensured when a communication is encrypted using the recipient’s public key and only the matching private key may decrypt it. RSA can also be used for digital signatures, in which the validity and non-repudiation of a communication are guaranteed by allowing anyone with the matching public key to verify a message signed with a private key.

### **How it is implemented in Python**

**ALICE:**

1. **Key Generation**

Generate two large prime numbers, p and q, each with a bit length of 512 using python modules or other modules depending on the programming language of choice.

* Compute, n=p\*q; then compute ϕ (n) = (p−1)\*(q−1)
* Choose a public exponent e (commonly 65537).
* Compute the private exponent d such that d\*e=1 mod ϕ (n)

1. **Communication:**

* Send the public key (n, e) to Bob over a TCP connection.
* Receive the cipher text from Bob.
* Decrypt the cipher text using the private key d.

**BOB:**

* **Receive Public Key:**

Connect to Alice and receive the public key of Alice (n, e).

* **Encrypt Message:**

Encrypt the plaintext message using the public key and send the cipher text back to Alice.

# **Results and Discussions**

**Code for Alice:**

*import socket*

*import rsa*

*def generate\_keys():*

*(public\_key, private\_key) = rsa.newkeys(1024)*

*return public\_key, private\_key*

*def main():*

*public\_key, private\_key = generate\_keys()*

*# Send public key to Bob*

*with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*

*s.connect(('localhost', 12345))*

*s.sendall(public\_key.save\_pkcs1())*

*# Receive ciphertext from Bob*

*with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*

*s.bind(('localhost', 12346))*

*s.listen()*

*conn, addr = s.accept()*

*with conn:*

*ciphertext = conn.recv(1024)*

*plaintext = rsa.decrypt(ciphertext, private\_key)*

*print("Decrypted message:", plaintext.decode('utf-8'))*

*if \_name\_ == "\_main\_":*

*main()*

**Brief Explanations of the Code:**

*def generate\_keys():*

*(public\_key, private\_key) = rsa.newkeys(1024)*

*return public\_key, private\_key*

Two RSA encryption keys are created using the function generate\_keys() that accepts no arguments.   
The function call rsa.newkeys(1024) from the RSA module creates a fresh set of RSA keys.   
The key length in bits is specified by the argument 1024.   
A tuple with two keys—a public key and a private key—is returned by the function.   
The tuple provided by rsa.newkeys(1024) is unpacked into two variables, public\_key and private\_key, using the function (public\_key, private\_key).   
The created public and private keys are returned as a tuple by the function return public\_key, private\_key.   
The generate\_keys method returns a pair of keys upon calling it. Data is encrypted using the public key and decrypted with the private key.

**Code for Bob:**

*import socket*

*import rsa*

*def main():*

*# Receive public key from Alice*

*with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*

*s.bind(('localhost', 12345))*

*s.listen()*

*conn, addr = s.accept()*

*with conn:*

*public\_key\_data = conn.recv(1024)*

*public\_key = rsa.PublicKey.load\_pkcs1(public\_key\_data)*

*# Encrypt the message*

*message = "Hello, Alice!"*

*ciphertext = rsa.encrypt(message.encode('utf-8'), public\_key)*

*# Print the encrypted message*

*print("Encrypted message:", ciphertext)*

*# Send ciphertext to Alice*

*with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*

*s.connect(('localhost', 12346))*

*s.sendall(ciphertext)*

*if \_name\_ == "\_main\_":*

*main()*

**Code Explanation:**

**Code:**

*def main():*

*# Receive public key from Alice*

*with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*

*s.bind(('localhost', 12345))*

*s.listen()*

*conn, addr = s.accept()*

*with conn:*

*public\_key\_data = conn.recv(1024)*

*public\_key = rsa.PublicKey.load\_pkcs1(public\_key\_data)*

**Explanation:**

Here the main () function sets up a server to receive a public key from a client (in this case, “Alice”) over a network connection.

**Function Definition:**

*def main()*: declares a new function: main which takes no arguments i.e. non-parameterised.

**Socket Creation:**

***with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:*** creates a new socket using the Internet address family (***AF\_INET***) and the TCP protocol (***SOCK\_STREAM***). The ***with*** statement ensures that the socket is properly closed when the block is exited.

**Binding the Socket:**

***s.bind((‘localhost’, 12345))*** binds the socket to the local address localhost on port 12345. This means the server will listen for incoming connections on this address and port.

**Listening for Connections:**

***s.listen()*** puts the socket into listening mode, ready to accept incoming connection requests.

**Accepting a Connection:**

***conn, addr = s.accept()*** waits for an incoming connection and, when one is made, accepts it. This returns a new socket object (conn) to communicate with the client and the address of the client (addr).

**Handling the Connection:**

***with conn:*** creates a context for the accepted connection socket (conn). This ensures that the connection is properly closed when the block is exited.

**Receiving Data:**

***public\_key\_data = conn.recv(1024)*** receives up to 1024 bytes of data from the client. This data is the public key sent by Alice.

**Loading the Public Key:**

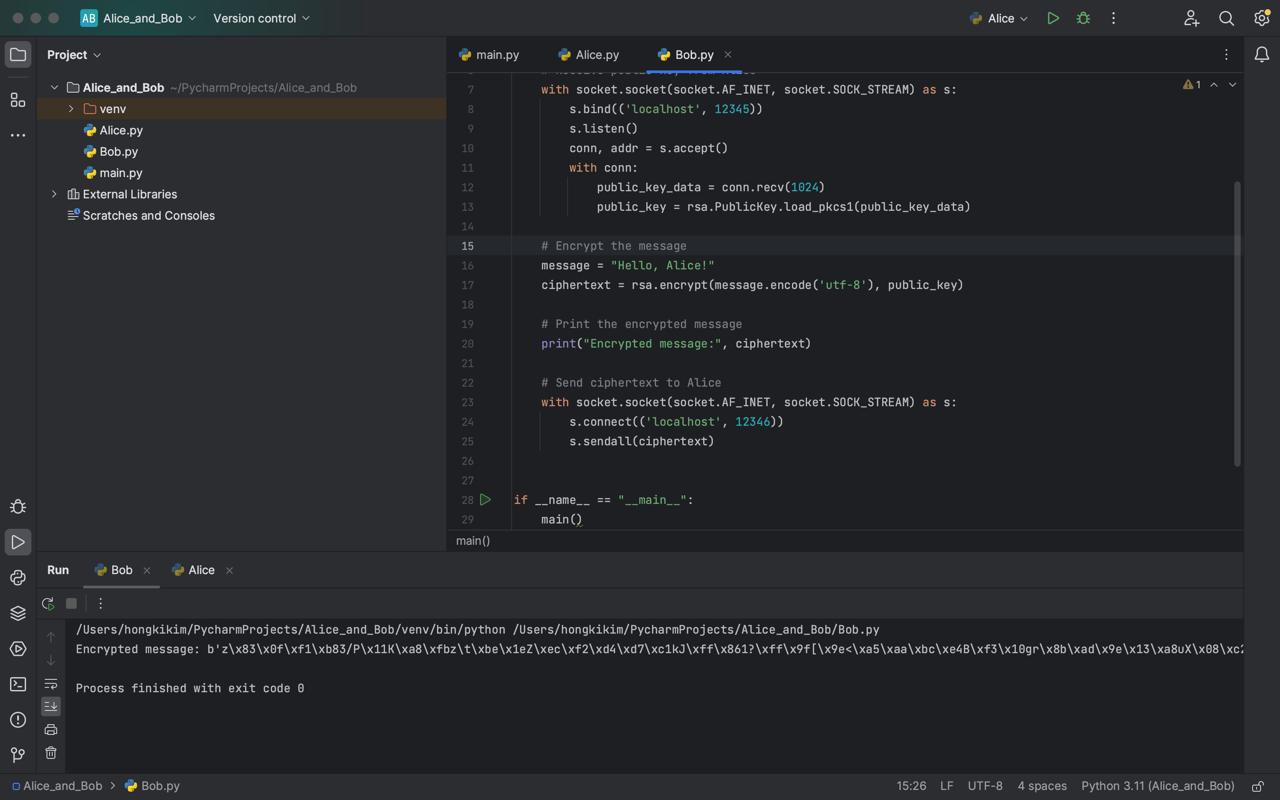
***public\_key = rsa.PublicKey.load\_pkcs1(public\_key\_data)*** loads the public key from the received data. The ***load\_pkcs1*** method is used to de-serialize the public key data, assuming it’s in PKCS#1 format.

**Usage of the Main Function in this Context:**

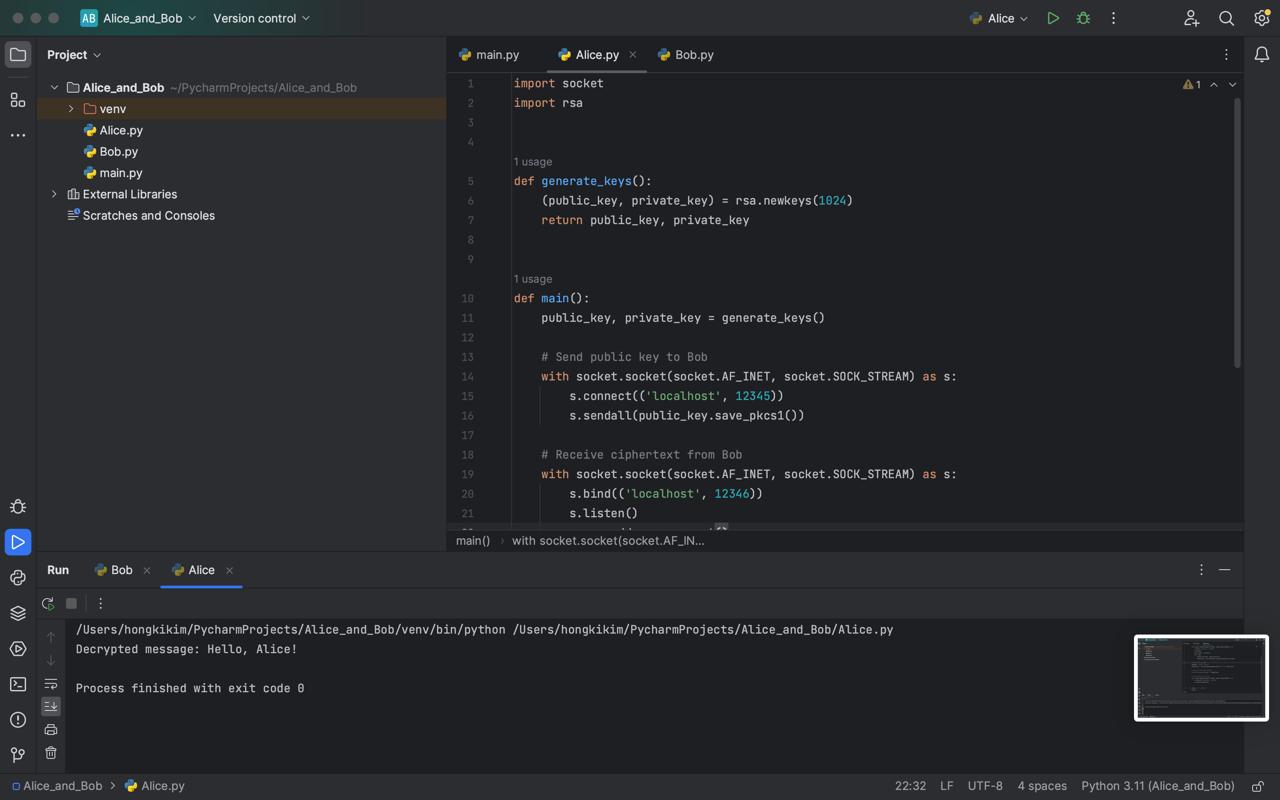
* Set up a server socket to listen for incoming connections on localhost: 12345.
* Accept a connection from a client (Alice).
* Receive up to 1024 bytes of data from Alice, which is expected to be her public key.
* De-serialize the received data to create an RSA public key object.

### **Output:**

After running Bob’s script and then Alice’s the following output can be seen:



**Output Alice’s Side:**



## **Discussions**

The project effectively illustrates how to use the RSA algorithm to enable secure communication between two people, in this case Bob And Alice. The solution creates secure communication channels over TCP sockets, and reliably produces RSA key pairs (public and private keys) that encrypts and decrypts communications.   
The group work emphasises the significance of safe private key management. Additional safeguards like digital signatures (use of Hash Functions) and certificate authorities would be used in the real world to guarantee the integrity and authenticity of the public keys.  
The “pycryptodome” package offers a safe and effective method of encryption and decryption while streamlining the RSA algorithm’s implementation. To preserve the system’s security, it is imperative to keep the library current and follow accepted cryptography best practices.

# **PART 2: Credit Card Vault System**

# **Background**

With the increasing need for secure storage, especially when it comes to financial information stored in banks; the management of the secure storage of customer credit card information is so very important for any financial organisation, if that information is to be protected from unauthorized access and to ensure compliance with industry standards such as the European GDPR and the Kenya Data Protection Act, recently enacted.

## **System Requirements:**

* Using appropriate encryption standards to ensure secure storage of credit card information.
* It should be able to allow user authentication and access based on their (user) respective roles.
* Must have an integration to a Database.

## **Database Schema:**

To guarantee data integrity and reduce redundancy, the Credit Card Vault system’s database schema is created in **Third Normal Form** (3NF). The tables that comprise the schema are as follows:

1. **‘Users’**

* ID (Primary Key)
* Username
* Hashed Password
* Role/duty

1. **Customers**

* ID (Primary Key)
* User\_ID (Foreign Key referencing users ID
* Name
* Email

1. **Credit\_cards**

* ID (Primary Key)
* Customer\_ID(Foreign Key referencing customers ID)
* EncryptedCard\_Number
* Encrypted\_CVV
* Expiration\_Date

1. **Transactions:**

* ID (Primary Key)
* Credit\_card\_id (Foreign Key referencing credit cards ID)
* Amount
* Timestamp

## **USER ACCESS LEVELS**

**Admin:**

* Managing user accounts and assigning of roles.
* Has full total access to the entirety of the database.
* Can delete accounts.

**Manager:**

* Viewing and retrieving of credit card information for transaction processing.
* Doesn’t have the capability to delete a record.

**Customer Service Representative:**

* Using customer information and credit card details for support purposes.
* Can’t modify or delete records.

# **Implementation of the Credit Card Vault System**

## **Creating the Database**

PostgreSQL is the database management system used by the Credit Card Vault system. The SQL statements given in appendix are used to build the database. Connecting the Python program to the PostgreSQL database requires the use of the ‘***psycopg2***’ package.

## **Implementing Encryption of the Data and Decryption**

Using Python’s cryptography library, information retrieved is encrypted at REST i.e. before it’s written into the disk in the database and read from it.

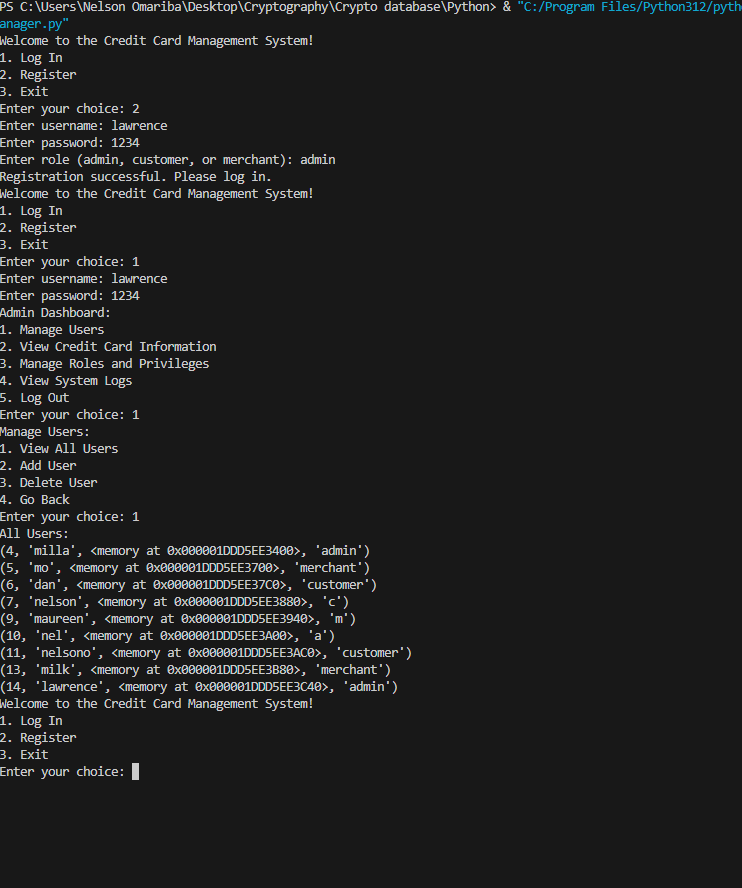
**Data encryption and decryption:** Using a 256-bit key in Galois/Counter Mode (GCM) and the Advanced Encryption Standard (AES), the ‘***encrypt\_data ()***’ function encrypts the data. The function returns the encrypted data together with the nonce that was used for encryption after receiving as inputs the plaintext data and a secret key.   
The encrypted data is decrypted using the same secret key and nonce by the ‘***decrypt\_data()***’ function. It returns the decrypted plaintext data after receiving the encrypted data and the secret key as input.

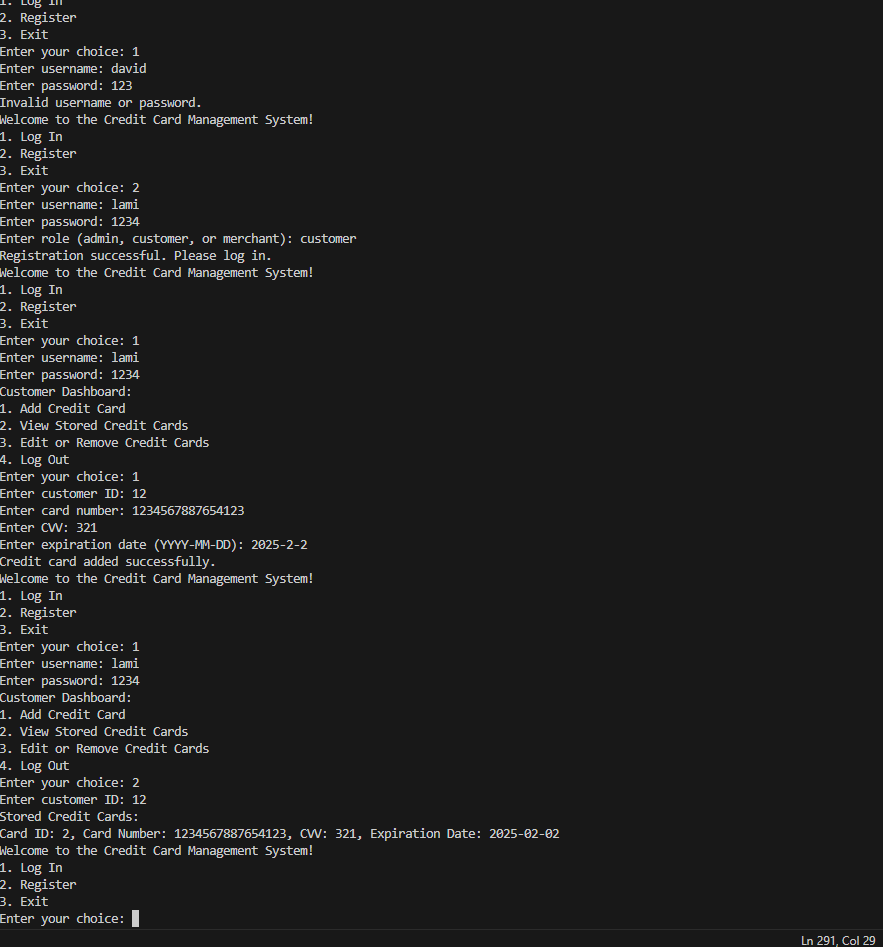
## **User Credential Authentication**

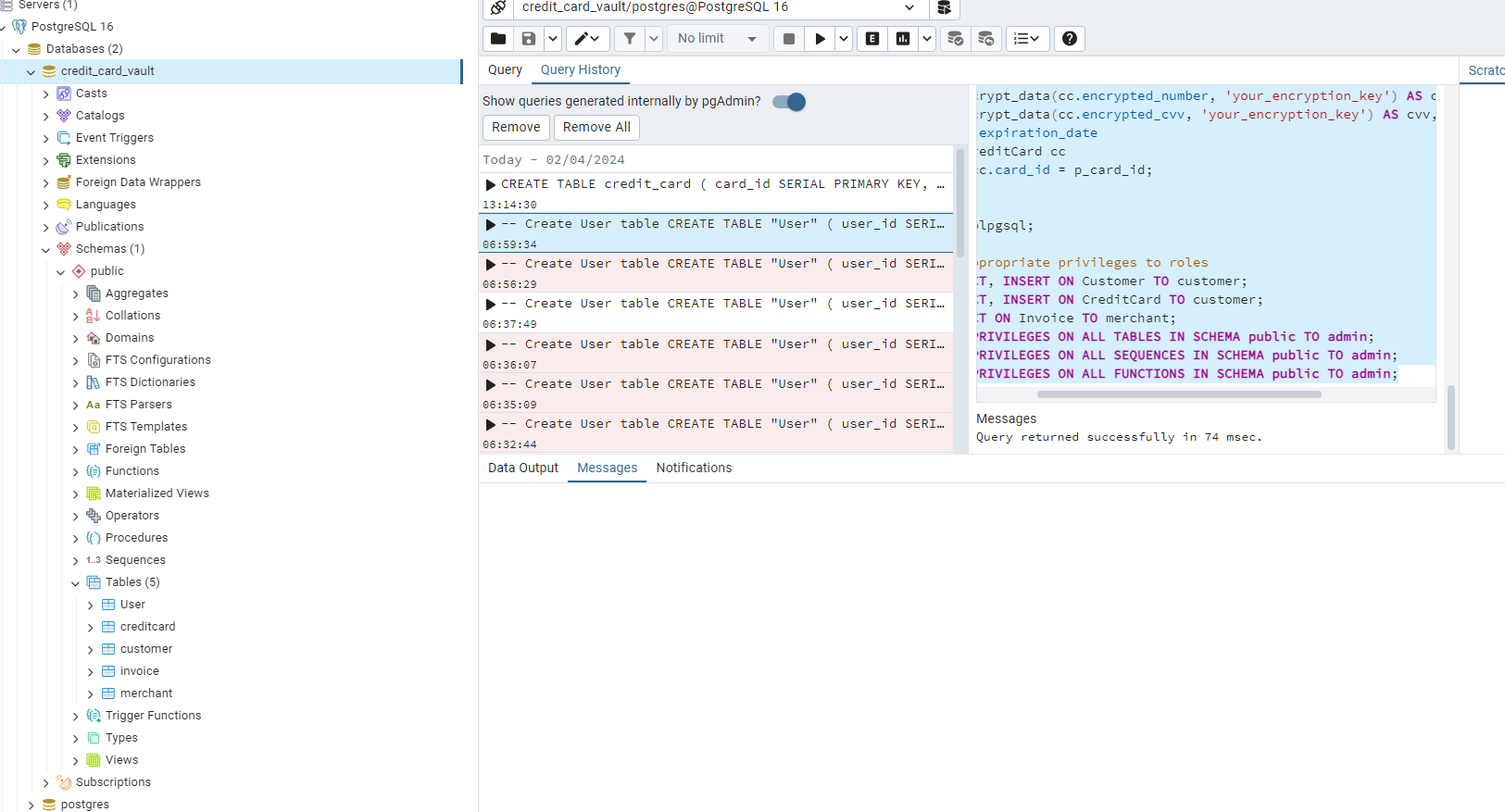
The PostgreSQL database’s users table would hold user credentials, such as a hashed password and username. According to the ***password\_hash*** column, passwords are hashed for security reasons rather than being kept in plain text.   
**Password Hashing:** Before a new user’s password is entered into the database, it is hashed using a secure hashing technique. This use the cryptographic hash function ***SHA-256***.  
**User Registration:** To create a new user record in the users database, a registration function would gather user data, hash the password, and store it.   
**User Login:** This function would obtain the user record from the database, hash the password that was entered, and then compare it with the hash that was previously recorded. The user is authenticated if the hashes match.   
**Session Management:** A session is started to keep the user signed in after a successful authentication. Python’s built-in ***http.cookies*** module or an outside library like ***Flask-Session*** are used to handle this.   
**Access Control:** To ascertain a user’s level of access and permissions within an application, the system would consult their role as recorded in the users database.

**Discussions of the Output for Credit Card System:**The Credit Card Vault system securely stores and manages customer credit card information using encryption for data confidentiality and access controls to prevent unauthorized access. PostgreSQL is used as the database management system, providing reliable and scalable storage, with a 3NF schema ensuring data integrity and minimal redundancy. The Django web framework facilitates development with built-in database management, user authentication, and web application features. While the system functions effectively, additional security measures and optimizations, including regular audits and adherence to PCI DSS standards, are necessary for production deployment, but given that this is entirely an academic undertaking to fulfill the requirements for Cryptography and Network Security (APT3090A) we shall venture no further.

## **Outputs of Database Implementation for Card Vault System**







# **References**

* Cohen, A. E., & Parhi, K. K. (2011). Architecture optimizations for the RSA public key cryptosystem: A tutorial. *IEEE Circuits and Systems Magazine*, *11*(4), 24-34. Retrieved from: <https://ieeexplore.ieee.org/abstract/document/6035849>
* Crouch, P. A., & Davenport, J. H. (2001, December). Lattice Attacks on RSA-encrypted IP and TCP. In *IMA International Conference on Cryptography and Coding* (pp. 329-338). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from: <https://link.springer.com/chapter/10.1007/3-540-45325-3_29>
* Kamiński, K., & Mazurczyk, W. (2023). RSA Keys Quality in a Real-world Organizational Certificate Dataset: a Practical Outlook. *International Journal of Electronics and Telecommunications*. Retrieved: DOI:10.24425/ijet.2023.147704
* Soram, R., & Meitei, E. S. (2015, September). On the performance of RSA in virtual banking. In *2015 International Symposium on Advanced Computing and Communication (ISACC)* (pp. 352-359). IEEE. Link: <https://ieeexplore.ieee.org/abstract/document/7377369>
* Szala, K. (2014). Encrypted TCP chat using RSA and AES algorithm. Retrieved from: <https://www.theseus.fi/bitstream/handle/10024/72709/szala_krzysztof.pdf?sequence=1>

# **Appendix: The credit card vault schema**

**Language of Implementation: Python**

import psycopg2

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

from cryptography.hazmat.primitives import hashes

import os

# Encryption and decryption functions

def encrypt\_data(plaintext, key):

    aesgcm = AESGCM(key)

    nonce = os.urandom(12)

    ciphertext = aesgcm.encrypt(nonce, plaintext.encode(), None)

    return nonce + ciphertext

def decrypt\_data(encrypted\_data, key):

    aesgcm = AESGCM(key)

    nonce = encrypted\_data[:12]

    ciphertext = encrypted\_data[12:]

    plaintext = aesgcm.decrypt(nonce, ciphertext, None)

    return plaintext.decode()

# Password hashing function

def hash\_password(password):

    digest = hashes.Hash(hashes.SHA256())

    digest.update(password.encode())

    return digest.finalize()

# Database connection parameters

db\_params = {

    'dbname': 'credit\_card\_vault',

    'user': 'postgres',

    'password': '654123',

    'host': 'localhost',

    'port': 5432,

}

# Encryption key (should be kept secret and secure)

encryption\_key = os.urandom(32)

# Function to authenticate user

def authenticate\_user(username, password):

    hashed\_password = hash\_password(password)

    with psycopg2.connect(\*\*db\_params) as conn:

        with conn.cursor() as cur:

            cur.execute(

                """

                SELECT role

                FROM "User"

                WHERE username = %s AND password\_hash = %s

                """,

                (username, hashed\_password)

            )

            user = cur.fetchone()

            if user:

                return user[0]

            else:

                return None

# Function to register new user

def register\_user(username, password, role):

    hashed\_password = hash\_password(password)

    with psycopg2.connect(\*\*db\_params) as conn:

        with conn.cursor() as cur:

            cur.execute(

                """

                INSERT INTO "User" (username, password\_hash, role)

                VALUES (%s, %s, %s)

                """,

                (username, hashed\_password, role)

            )

            conn.commit()

            return True

# Function to manage users (Admin Dashboard)

def manage\_users():

    print("Manage Users:")

    print("1. View All Users")

    print("2. Add User")

    print("3. Delete User")

    print("4. Go Back")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        view\_all\_users()

    elif choice == '2':

        add\_user()

    elif choice == '3':

        delete\_user()

    elif choice == '4':

        admin\_dashboard()

    else:

        print("Invalid choice.")

def view\_all\_users():

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve all users from the database

        cur.execute("SELECT \* FROM \"User\"")

        users = cur.fetchall()

        # Display users

        print("All Users:")

        for user in users:

            print(user)

    except psycopg2.Error as e:

        print("Error fetching users:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

def add\_user():

    username = input("Enter username: ")

    password = input("Enter password: ")

    role = input("Enter role (admin, customer, or merchant): ").lower()

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Insert new user into the database

        cur.execute("INSERT INTO \"User\" (username, password\_hash, role) VALUES (%s, %s, %s)", (username, password, role))

        conn.commit()

        print("User added successfully.")

    except psycopg2.Error as e:

        print("Error adding user:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

def delete\_user():

    username = input("Enter username to delete: ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Delete user from the database

        cur.execute("DELETE FROM \"User\" WHERE username = %s", (username,))

        conn.commit()

        print("User deleted successfully.")

    except psycopg2.Error as e:

        print("Error deleting user:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to view credit card information (Admin Dashboard)

def view\_credit\_card\_info():

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve credit card information from the database

        cur.execute("SELECT \* FROM credit\_card")

        credit\_card = cur.fetchall()

        # Display credit card information

        print("Credit Card Information:")

        for card in credit\_card:

            print(card)

    except psycopg2.Error as e:

        print("Error fetching credit card information:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to manage roles and privileges (Admin Dashboard)

def manage\_roles\_privileges():

    print("Manage Roles and Privileges:")

    print("1. Assign Role to User")

    print("2. Modify Privileges")

    print("3. Go Back")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        assign\_role()

    elif choice == '2':

        modify\_privileges()

    elif choice == '3':

        admin\_dashboard()

    else:

        print("Invalid choice.")

def assign\_role():

    username = input("Enter username: ")

    role = input("Enter role (admin, customer, or merchant): ").lower()

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Update user's role in the database

        cur.execute("UPDATE \"User\" SET role = %s WHERE username = %s", (role, username))

        conn.commit()

        print("Role assigned successfully.")

    except psycopg2.Error as e:

        print("Error assigning role:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

def modify\_privileges():

    print("Modify Privileges functionality under construction.")

# Function to view system logs (Admin Dashboard)

def view\_system\_logs():

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve system logs from the database

        cur.execute("SELECT \* FROM system\_logs")

        logs = cur.fetchall()

        # Display system logs

        print("System Logs:")

        for log in logs:

            print(log)

    except psycopg2.Error as e:

        print("Error fetching system logs:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Customer dashboard functions

# Function to add a credit card (Customer Dashboard)

def add\_credit\_card():

    customer\_id = input("Enter customer ID: ")

    card\_number = input("Enter card number: ")

    cvv = input("Enter CVV: ")

    expiration\_date = input("Enter expiration date (YYYY-MM-DD): ")

    # Validate card number length (assuming it should be 16 digits)

    if len(card\_number) != 16:

        print("Invalid card number length. It should be 16 digits.")

        return

    # Validate CVV length (assuming it should be 3 digits)

    if len(cvv) != 3:

        print("Invalid CVV length. It should be 3 digits.")

        return

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Encrypt credit card information

        encrypted\_number = encrypt\_data(card\_number, encryption\_key)

        encrypted\_cvv = encrypt\_data(cvv, encryption\_key)

        # Insert credit card into the database

        cur.execute(

            """

            INSERT INTO credit\_card (customer\_id, encrypted\_number, encrypted\_cvv, expiration\_date)

            VALUES (%s, %s, %s, %s)

            """,

            (customer\_id, encrypted\_number, encrypted\_cvv, expiration\_date)

        )

        conn.commit()

        print("Credit card added successfully.")

    except psycopg2.Error as e:

        print("Error adding credit card:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to view stored credit cards (Customer Dashboard)

def view\_stored\_credit\_card():

    customer\_id = input("Enter customer ID: ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve credit cards for the customer from the database

        cur.execute("SELECT \* FROM credit\_card WHERE customer\_id = %s", (customer\_id,))

        credit\_card = cur.fetchall()

        # Display credit card information

        print("Stored Credit Cards:")

        for card in credit\_card:

            card\_id, customer\_id, encrypted\_number, encrypted\_cvv, expiration\_date = card

            card\_number = decrypt\_data(encrypted\_number, encryption\_key)

            cvv = decrypt\_data(encrypted\_cvv, encryption\_key)

            print(f"Card ID: {card\_id}, Card Number: {card\_number}, CVV: {cvv}, Expiration Date: {expiration\_date}")

    except psycopg2.Error as e:

        print("Error fetching credit cards:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to edit or remove credit cards (Customer Dashboard)

def edit\_or\_remove\_credit\_card():

    print("Edit or Remove Credit Cards:")

    print("1. Edit Credit Card")

    print("2. Remove Credit Card")

    print("3. Go Back")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        edit\_credit\_card()

    elif choice == '2':

        remove\_credit\_card()

    elif choice == '3':

        customer\_dashboard()

    else:

        print("Invalid choice.")

# Function to edit a credit card (Customer Dashboard)

def edit\_credit\_card():

    card\_id = input("Enter card ID to edit: ")

    card\_number = input("Enter new card number: ")

    cvv = input("Enter new CVV: ")

    expiration\_date = input("Enter new expiration date (YYYY-MM-DD): ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Encrypt credit card information

        encrypted\_number = encrypt\_data(card\_number, encryption\_key)

        encrypted\_cvv = encrypt\_data(cvv, encryption\_key)

        # Update credit card in the database

        cur.execute(

            """

            UPDATE credit\_card

            SET encrypted\_number = %s, encrypted\_cvv = %s, expiration\_date = %s

            WHERE card\_id = %s

            """,

            (encrypted\_number, encrypted\_cvv, expiration\_date, card\_id)

        )

        conn.commit()

        print("Credit card updated successfully.")

    except psycopg2.Error as e:

        print("Error updating credit card:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to remove a credit card (Customer Dashboard)

def remove\_credit\_card():

    card\_id = input("Enter card ID to remove: ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Delete credit card from the database

        cur.execute("DELETE FROM credit\_card WHERE card\_id = %s", (card\_id,))

        conn.commit()

        print("Credit card removed successfully.")

    except psycopg2.Error as e:

        print("Error removing credit card:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to view customer credit cards (Merchant Dashboard)

def view\_customer\_credit\_card():

    customer\_id = input("Enter customer ID: ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve credit cards for the customer from the database

        cur.execute("SELECT \* FROM credit\_card WHERE customer\_id = %s", (customer\_id,))

        credit\_card = cur.fetchall()

        # Display credit card information

        print("Customer Credit Cards:")

        for card in credit\_card:

            card\_id, customer\_id, encrypted\_number, encrypted\_cvv, expiration\_date = card

            card\_number = decrypt\_data(encrypted\_number, encryption\_key)

            cvv = decrypt\_data(encrypted\_cvv, encryption\_key)

            print(f"Card ID: {card\_id}, Card Number: {card\_number}, CVV: {cvv}, Expiration Date: {expiration\_date}")

    except psycopg2.Error as e:

        print("Error fetching customer credit cards:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to process transactions (Merchant Dashboard)

def process\_transactions():

    card\_id = input("Enter card ID: ")

    amount = float(input("Enter transaction amount: "))

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Insert transaction into the database

        cur.execute(

            """

            INSERT INTO transactions (credit\_card\_id, amount)

            VALUES (%s, %s)

            """,

            (card\_id, amount)

        )

        conn.commit()

        print("Transaction processed successfully.")

    except psycopg2.Error as e:

        print("Error processing transaction:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Function to view transaction history for merchants (Merchant Dashboard)

def transaction\_history\_merchant():

    merchant\_id = input("Enter merchant ID: ")

    try:

        # Connect to the database

        conn = psycopg2.connect(\*\*db\_params)

        cur = conn.cursor()

        # Retrieve transaction history for the merchant from the database

        cur.execute(

            """

            SELECT t.transaction\_id, t.credit\_card\_id, c.customer\_id, t.amount

            FROM transactions t

            INNER JOIN credit\_card c ON t.credit\_card\_id = c.card\_id

            WHERE c.merchant\_id = %s

            """,

            (merchant\_id,)

        )

        transactions = cur.fetchall()

        # Display transaction history

        print("Transaction History:")

        for transaction in transactions:

            print(transaction)

    except psycopg2.Error as e:

        print("Error fetching transaction history:", e)

    finally:

        # Close the connection

        if conn is not None:

            conn.close()

# Main function for admin dashboard

def admin\_dashboard():

    print("Admin Dashboard:")

    print("1. Manage Users")

    print("2. View Credit Card Information")

    print("3. Manage Roles and Privileges")

    print("4. View System Logs")

    print("5. Log Out")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        manage\_users()

    elif choice == '2':

        view\_credit\_card\_info()

    elif choice == '3':

        manage\_roles\_privileges()

    elif choice == '4':

        view\_system\_logs()

    elif choice == '5':

        print("Logged out.")

        return

    else:

        print("Invalid choice.")

        admin\_dashboard()

# Main function for customer dashboard

def customer\_dashboard():

    print("Customer Dashboard:")

    print("1. Add Credit Card")

    print("2. View Stored Credit Cards")

    print("3. Edit or Remove Credit Cards")

    print("4. Log Out")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        add\_credit\_card()

    elif choice == '2':

        view\_stored\_credit\_card()

    elif choice == '3':

        edit\_or\_remove\_credit\_card()

    elif choice == '4':

        print("Logged out.")

        return

    else:

        print("Invalid choice.")

        customer\_dashboard()

# Main function for merchant dashboard

def merchant\_dashboard():

    print("Merchant Dashboard:")

    print("1. View Customer Credit Cards")

    print("2. Process Transactions")

    print("3. Transaction History")

    print("4. Log Out")

    choice = input("Enter your choice: ").strip()

    if choice == '1':

        view\_customer\_credit\_card()

    elif choice == '2':

        process\_transactions()

    elif choice == '3':

        transaction\_history\_merchant()

    elif choice == '4':

        print("Logged out.")

        return

    else:

        print("Invalid choice.")

        merchant\_dashboard()

# Main function for the login system

def main():

    while True:

        print("Welcome to the Credit Card Management System!")

        print("1. Log In")

        print("2. Register")

        print("3. Exit")

        choice = input("Enter your choice: ").strip()

        if choice == '1':

            username = input("Enter username: ")

            password = input("Enter password: ")

            role = authenticate\_user(username, password)

            if role == 'admin':

                admin\_dashboard()

            elif role == 'customer':

                customer\_dashboard()

            elif role == 'merchant':

                merchant\_dashboard()

            else:

                print("Invalid username or password.")

        elif choice == '2':

            username = input("Enter username: ")

            password = input("Enter password: ")

            role = input("Enter role (admin, customer, or merchant): ").lower()

            if role in ['admin', 'customer', 'merchant']:

                if register\_user(username, password, role):

                    print("Registration successful. Please log in.")

                else:

                    print("Registration failed.")

            else:

                print("Invalid role.")

        elif choice == '3':

            print("Goodbye!")

            break

        else:

            print("Invalid choice.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# **Appendix: Structured Query Language (SQL Syntax)**

*CREATE TABLE credit\_card (*

*card\_id SERIAL PRIMARY KEY,*

*customer\_id VARCHAR(50),*

*encrypted\_number BYTEA,*

*encrypted\_cvv BYTEA,*

*expiration\_date DATE);*

*-- Create User table*

*CREATE TABLE "User" (*

*user\_id SERIAL PRIMARY KEY,*

*username VARCHAR(100) UNIQUE NOT NULL,*

*password\_hash BYTEA NOT NULL,*

*role VARCHAR(20) NOT NULL);*

*-- Create Customer table*

*CREATE TABLE Customer (*

*customer\_id SERIAL PRIMARY KEY,*

*user\_id INTEGER REFERENCES "User"(user\_id),*

*name VARCHAR(100) NOT NULL,*

*email VARCHAR(100) UNIQUE NOT NULL);*

*-- Create CreditCard table*

*CREATE TABLE CreditCard (*

*card\_id SERIAL PRIMARY KEY,*

*customer\_id INTEGER REFERENCES Customer(customer\_id),*

*encrypted\_number BYTEA NOT NULL,*

*encrypted\_cvv BYTEA NOT NULL,*

*expiration\_date DATE NOT NULL);*

*-- Create Merchant table*

*CREATE TABLE Merchant (*

*merchant\_id SERIAL PRIMARY KEY,*

*name VARCHAR(100) NOT NULL,*

*website VARCHAR(100),*

*contact\_info VARCHAR(100));*

*-- Create Invoice table*

*CREATE TABLE Invoice (*

*invoice\_id SERIAL PRIMARY KEY,*

*customer\_id INTEGER REFERENCES Customer(customer\_id),*

*merchant\_id INTEGER REFERENCES Merchant(merchant\_id),*

*amount DECIMAL(10, 2) NOT NULL,*

*date DATE NOT NULL);*

*-- Function to insert credit card information*

*CREATE OR REPLACE FUNCTION insert\_credit\_card(*

*p\_customer\_id INTEGER,*

*p\_card\_number VARCHAR(16),*

*p\_cvv VARCHAR(3),*

*p\_expiration\_date DATE)*

*RETURNS VOID AS $$*

*DECLARE*

*v\_encrypted\_number BYTEA;*

*v\_encrypted\_cvv BYTEA;*

*BEGIN*

*v\_encrypted\_number := encrypt\_data(p\_card\_number, 'your\_encryption\_key');*

*v\_encrypted\_cvv := encrypt\_data(p\_cvv, 'your\_encryption\_key');*

*INSERT INTO CreditCard (customer\_id, encrypted\_number, encrypted\_cvv, expiration\_date)*

*VALUES (p\_customer\_id, v\_encrypted\_number, v\_encrypted\_cvv, p\_expiration\_date);*

*END;*

*$$*

*LANGUAGE plpgsql;*

*-- Function to retrieve credit card information*

*CREATE OR REPLACE FUNCTION get\_credit\_card(p\_card\_id INTEGER)*

*RETURNS TABLE (*

*card\_id INTEGER,*

*customer\_id INTEGER,*

*card\_number VARCHAR(16),*

*cvv VARCHAR(3),*

*expiration\_date DATE*

*) AS $$*

*BEGIN*

*RETURN QUERY*

*SELECT*

*cc.card\_id,*

*cc.customer\_id,*

*decrypt\_data(cc.encrypted\_number, 'your\_encryption\_key') AS card\_number,*

*decrypt\_data(cc.encrypted\_cvv, 'your\_encryption\_key') AS cvv,*

*cc.expiration\_date*

*FROM CreditCard cc*

*WHERE cc.card\_id = p\_card\_id;*

*END;*

*$$*

*LANGUAGE plpgsql;*

*-- Grant appropriate privileges to roles*

*GRANT SELECT, INSERT ON Customer TO customer;*

*GRANT SELECT, INSERT ON CreditCard TO customer;*

*GRANT SELECT ON Invoice TO merchant;*

*GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin;*

*GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO admin;*

*GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO admin;*