IM Coursework 2 Report

u5539006

Database Design and Security

Table Design

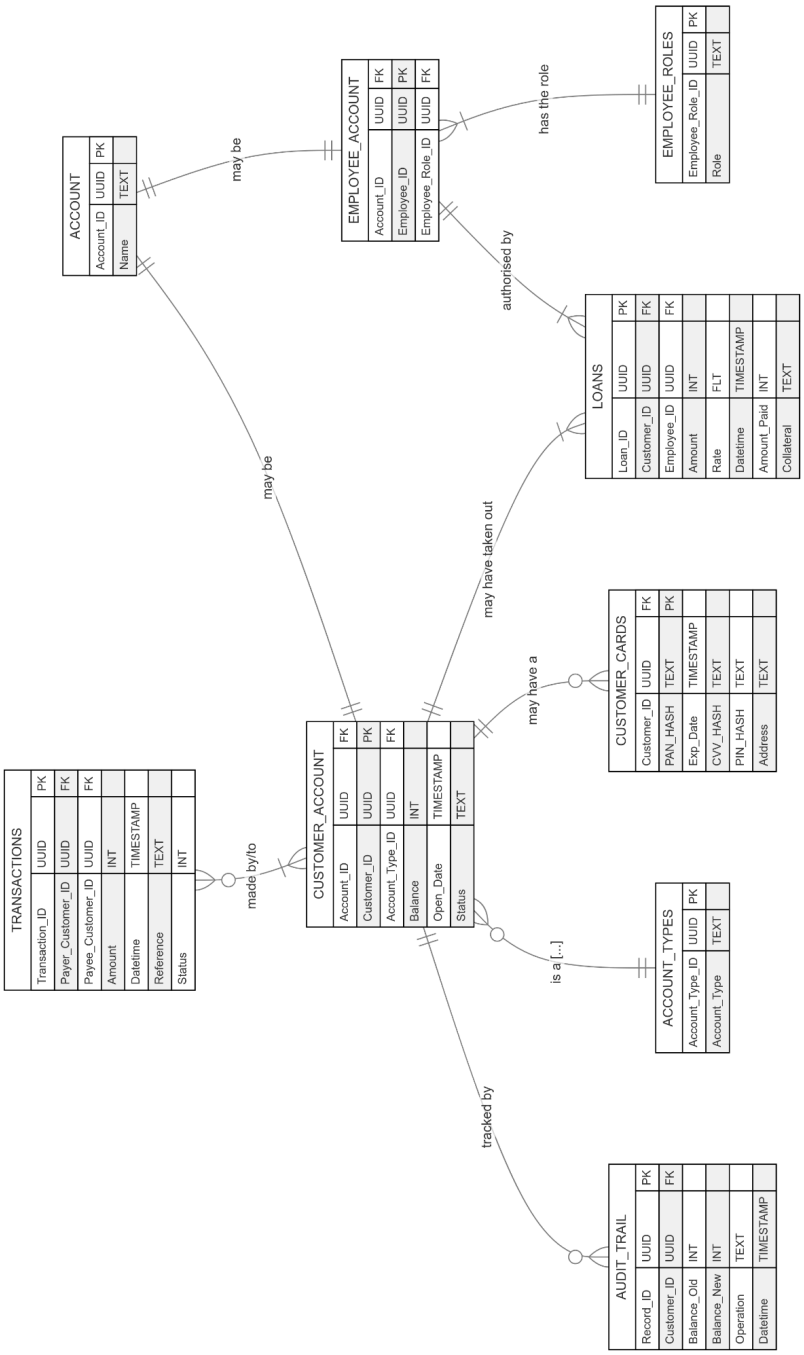

Figure: an ERD of the proposed tables contained within the database, including relations and data types.

Table	Details	Primary Key	Foreign Key(s)
account	Parent "class" for all accounts that gives them a unique ID and stores information that isn't exclusive to the type of account	Account_ID	
employee_roles	Stores the different roles of employees in the database. These are implemented using group roles, but are also stored for clarity.	Employee_Role_ID	
employee_account	Covers the details of each individual employee, and their role. Each person will have a unique record.	Employee_ID	Account_ID, Employee_Role_ID
account_types	Covers the possible types of accounts for customers.	Account_Type_ID	
customer_account	A bank account record. Referred to by cards, loans and transactions. Stores the balance and type of the account etc.	Customer_ID	Account_ID, Account_Type_ID
customer_cards	Stores information about a payment card associated with a customer account. Sensitive information is hashed upon entry to the database, and it is possible to compare data using a function.	PAN_HASH	Customer_ID
transactions	Stores the history of all transactions that have taken place between members of the system.	Transaction_ID	Payer_Customer_ID, Payee_Customer_ID
loans	Loans can only be created by Loan Officers. This table stores important information about the loan, with money stored in pence, and rate stored as an integer percentage.	Loan_ID	Customer_ID, Employee_ID
audit_trail	Records all changes to customer balances, and the operations through which they happened.	Record_ID	Customer_ID

Identify Security Requirements

As a bank, storing financial and sensitive data is necessary. It is therefore of the utmost importance that adequate security measures are implemented to mitigate against cyber attacks and data theft. Using the CIA triad (confidentiality, integrity, and availability), key measures to protect the data and systems will be discussed.
Confidentiality covers the protection of sensitive, personal data under GDPR (UK Government 2016), and also covers payment card information under PCI-DSS.
Integrity will be maintained through strict access control measures on all parts of the database, implemented using the least privilege principle - employees will only have access to information they *have* to in order to complete their job. Regular auditing also allows for identifying any event which might compromise integrity.
Availability is also somewhat covered by strict access controls, allowing only certain users to perform certain operations on the data. Anything more than this is out of scope for this assessment.

The following are basic, overarching security requirements for the whole application, which will be covered in more detail later in the document.

Requirement	**Description**
Encryption	Sensitive information should be encrypted during transit and at rest in the database.
Access Control	Access control should be implemented for employees according to the least-privilege principle, and users should only be able to access details about themselves. Employees should also be restricted from their accounts during non-working hours.
Hashing of financial details	Financial details (i.e. card details) may only authenticate cards using hashed data, and this should ideally not be transmitted.
Database logging	Changes to certain records i.e. customer balance, transaction history should be logged, along with identifying data, such as timestamp, IP address, etc.
Compliance with standards and regulations	The storage of this data should comply with relevant standards and regulations such as GDPR, PCI-DSS, and ISO 27001.

Standards and Regulations

In order to ensure compliance of the proposed database with GDPR (UK Government 2016), and with the Payment Card Industry Data Security Standard (PCI-DSS) regulations, ensuring

cardholder detail security, the following security measures must be met (PCI Security Standards Council 2018).

PCI-DSS Number	Requirement	Implementation
3	*Protect Stored Cardholder Data*	Hashed payment card details within the database, and the database will be encrypted.
4	*Encrypt transmission of cardholder data across open, public networks*	Use an SSL key and certificate to encrypt any communication with clients.
7	*Restrict access to cardholder data by business need to know*	Allow access to cardholder data according to the least-privilege principle using roles of employees.
8	*Identify and authenticate access to system components*	Log IP addresses of clients in the audit trail, and require users to enter a password to access their account.
10	*Track and monitor all access to network resources and cardholder data*	Log all access to the database and any changes in the audit trail. No users will be permitted to edit data within the audit trail.

Implementation of Security Measures

Encryption at Rest and in Transit

There is no functionality with Postgres to support encryption at rest, but is more reliable simply within full-disk encryption. This can be achieved by storing the postgres configuration and data within an encrypted partition. A script to demonstrate the achievement of this on Linux is below.

```
$ encryption.sh
#!/bin/sh

# this script moves the postgres directory to a new encrypted partition in order to enable encryption at rest

sudo -u postgres psql -c "SHOW data_directory;"; # shows the data directory for postgres

sudo cryptsetup luksFormat /dev/sdb1 # replace /dev/sdb1 with the target partition for the encryption


sudo mkfs.ext4 /dev/mapper/pgsql_crypt # format the partition

sudo mkdir /mnt/pgsql # create a directory to mount the partition inside

sudo mount /dev/mapper/pgsql_crypt /mnt/pgsql # mount the partition

sudo systemctl stop postgresql # shuts down running postgres instances

sudo mv /var/lib/postgresql /mnt/pgsql/ # replace /var/lib/postgresql with the data_directory
# moves the data directory to the encrypted partition

sudo ln -s /mnt/pgsql/postgresql /var/lib/postgresql # create a symbolic link to the new directory

sudo systemctl start postgresql # restart postgres
```

Figure: a shell (*.sh*) script to move postgres data to a newly created encrypted partition

Encryption in transit, however, is achieved through the use of SSL certificates and keys. I have set up the database to require SSL through the file *postgresql.conf*:

```
config > postgresql.conf
ssl = on  # enables ssl
ssl_ca_file = 'root.crt' # defines ssl root ca file
ssl_cert_file = 'server.crt' # defines ssl certificate file
ssl_key_file = 'server.key'  # defines ssl key file
ssl_ciphers = 'HIGH:MEDIUM:+3DES:!aNULL'  # defines ssl ciphers
```

Figure: SSL configuration for the postgres database.

I have also created an SSL root and server certificate, and key for the database. This causes all transmissions with the database to be encrypted.

Access Control

This is covered in lots of detail in the next section, and as such, I won't cover it here.

Hashing of Financial Details

To store cardholder details in compliance with PCI-DSS, the PAN, PIN, and CVV of the card all need to be hashed. The following Postgres function demonstrates the implementation of such a process.

setup.sql

```
1   CREATE OR REPLACE FUNCTION hash_details()
2   RETURNS TRIGGER AS $$
3       BEGIN
4           NEW.PAN_HASH = digest(NEW.PAN_HASH, 'sha256');
5           NEW.CVV_HASH = digest(NEW.CVV_HASH, 'sha256');
6           NEW.PIN_HASH = digest(NEW.PIN_HASH, 'sha256');
7           RETURN NEW;
8       END $$
9   LANGUAGE plpgsql;
10
11  CREATE TRIGGER card_details_update_insert_trigger
12  BEFORE INSERT OR UPDATE ON customer_cards
13  FOR EACH ROW EXECUTE FUNCTION hash_details();
```

Figure: a Postgres function that causes the hashing of the PAN, PIN and CVV

This function is triggered when a record is added or changed in the card_details table, hashing the PAN, PIN and CVV using the sha256 algorithm. A card shouldn't need to be updated - only added or removed, but it's just to cover all possibilities.

Database Logging

Database logging is enabled to capture access to the database using postgres' built-in functionality. This is enabled in *postgresql.conf*:

```
9   logging_collector = on  # enable logs
10  log_directory = '/logs/'  # set log directory
11  log_filename = 'postgresql.log'  # set log filename convention
12  log_connections = on  # log connections to the database
13  log_disconnections = on  # log disconnections from the database
14  log_statement = 'all' # log all SQL querues
15  log_checkpoints = on  # Log all checkpoints
```

Figure: the enabling and configuration of logging settings within the file.

This records the IP addresses of the connections, and between this and the audit_trail table, a comprehensive log of transactions and access is achieved.

Compliance with standards and regulations

GDPR

Regulation: Integrity & Confidentiality (Articles 5 & 32).
Requirement: Ensure data security.
Implementation: Encryption in transit and at rest, and hashing of critical financial information.

Regulation: Accountability and Governance (Article 30).
Requirement: Maintain records of data processing.
Implementation: Logging access to the database, and any modifications made to personal data, using the audit_trail table.

PCI-DSS

Requirement: Protect Stored Cardholder Data.
Implementation: Payment card details are hashed before being stored in the database, so no plaintext details are ever stored.

Requirement: Encrypt transmission of cardholder data across open, public networks.
Implementation: Force the database to use SSL and provide an SSL certificate and key to allow communication.

Requirement: Restrict access to cardholder data by business need to know.
Implementation: Detailed below in Access Control, only certain roles can access information, through specified functions.

Requirement: Identify and authenticate access to system components.
Implementation: Postgres logging is enabled to track access to the database, and any modifications can be found in the audit_trail. Users also require a password to log in.

Requirement: Track and monitor all access to network resources and cardholder data.
Implementation: All changes recorded in the audit_trail table which is read-only to all users. Cardholder data can also only be accessed through specific functions.

Payment Security Enhancement

All functions are in a *Security Defined* state, meaning they execute with *postgres* superuser privileges. This means that they will execute flawlessly, even if the executing user does not have the privilege to perform the given operations on the respective table. This allows for complete control over user control of certain operations.

Registering a new customer

The function below allows a new customer to be registered, and their account to be opened. There are no authorisation checks for this function, since it is likely possible for users to create an account themselves, perhaps online.

```
15  CREATE OR REPLACE FUNCTION create_account(
16      name varchar(255), account_type TEXT)
17      RETURNS UUID LANGUAGE plpgsql AS
18      $$
19          DECLARE
20              userid UUID;
21              roleid UUID;
22          BEGIN
23              SELECT account_type_id FROM account_types WHERE account_type = account_type INTO roleid;
24              IF roleid IS NULL THEN
25                  RAISE EXCEPTION 'Account Type does not exist.';
26              END IF;
27              INSERT INTO account(name) VALUES (name) RETURNING account_id INTO userid;
28              INSERT INTO customer_account(account_id, account_type_id) VALUES (userid, roleid);
29              RETURN userid;
30          END
31      $$ SECURITY DEFINER;
```

Figure: Postgres function to create a new customer account.

Registering a new employee

This function allows a new employee to be registered, with their role.

```
CREATE OR REPLACE FUNCTION create_employee(
    name varchar(255), employee_type_input TEXT)
    RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            userid UUID;
            roleid UUID;
        BEGIN
            SELECT employee_role_id FROM employee_roles WHERE role = employee_type_input INTO roleid;
            IF roleid IS NULL THEN
                RAISE EXCEPTION 'Account Type does not exist.';
            END IF;
            INSERT INTO account(name) VALUES (name) RETURNING account_id INTO userid;
            INSERT INTO employee_account(account_id, employee_role_id) VALUES (userid, roleid);
            RETURN userid;
        END
    $$ SECURITY DEFINER;
```

Figure: Postgres function to register a new employee.

Checking an employee role

This function allows for the checking for the employee role of the caller, against a specified role.

```
CREATE OR REPLACE FUNCTION check_employee_role(
    caller UUID, proposed_role TEXT)
    RETURNS BOOLEAN LANGUAGE plpgsql AS
    $$
        DECLARE
            role TEXT;
        BEGIN
            SELECT employee_roles.role FROM employee_roles INNER JOIN employee_account ON
            employee_account.employee_role_id=employee_roles.employee_role_id
            WHERE employee_account.account_id = caller INTO role;

            IF role IS NULL THEN
                RETURN FALSE;
            END IF;

            RETURN role = proposed_role;

        EXCEPTION
            WHEN OTHERS THEN
                RAISE WARNING 'Error in check_employee_role: %', SQLERRM;
                RETURN FALSE;

        END
    $$ SECURITY DEFINER;
```

Figure: Postgres function to get the role of an employee

Paying a customer

This method attempts to transfer an amount of money between two accounts. The payee is identified by the Sort Code and Account Number associated with their account, and it is assumed that the person calling the function is the one paying. The amount is deducted from the payer account, assuming their balance is at least as much, and added to the payee account.

```
59  CREATE OR REPLACE FUNCTION pay_customer(
60      caller UUID, sort_code TEXT, account_num TEXT, cost INT, reference TEXT)
61      RETURNS UUID LANGUAGE plpgsql AS
62      $$
63          DECLARE
64              transactionID UUID;
65              payee_CustomerID UUID;
66              payer_CustomerID UUID;
67              rowsAffected INT;
68          BEGIN
69              SELECT customer_ID FROM customer_account WHERE sort_code = sort_code AND
70              account_num = account_num INTO payee_CustomerID;
71              SELECT CustomerID FROM custmer_account WHERE account_ID = caller INTO payer_CustomerID;
72
73              INSERT INTO transactions(payer_customer_id, payee_customer_id, amount, reference, status)
74              VALUES (payer_CustomerID, payee_CustomerID, cost, reference, 0)
75              RETURNING transaction_id INTO transactionID;
76
77              UPDATE customer_account SET amount = amount - cost WHERE customer_ID = payer_CustomerID
78              AND balance >= cost;
79
80              GET DIAGNOSTICS rowsAffected = ROW_COUNT;
81              IF rowsAffected = 0 THEN
82                  UPDATE transactions SET status = 2 WHERE transaction_id = transactionID;
83                  RAISE EXCEPTION 'Balance is not sufficient for this transaction.';
84              END IF;
85
86              UPDATE customer_account SET amount = amount + cost WHERE CustomerID = payee_CustomerID;
87              UPDATE transactions SET status = 1 WHERE transaction_id = transactionID;
88
89              RETURN transactionID;
90
91          EXCEPTION
92              WHEN OTHERS THEN
93                  UPDATE transactions SET status = 2 WHERE transaction_id = transactionID;
94                  RAISE WARNING 'Error in pay_customer: %', SQLERRM;
95                  RETURN NULL;
96          END
97      $$ SECURITY DEFINER;
```

Figure: Postgres function to pay another customer

This function uses a status code system to keep track of the status of the payment:

Status code of transaction	State
0	Incomplete
1	Completed

2	Failed

Check payment details

This function allows for the payment details of a user to be checked, without transmitting them to another server, for whether or not they are valid, and whether the funds are available to cover the transaction.

```
CREATE OR REPLACE FUNCTION check_payment_details(
    PAN VARCHAR(16), Exp_Date TIMESTAMP, CVV VARCHAR(4), Name TEXT, cost FLOAT)
    RETURNS BOOLEAN LANGUAGE plpgsql AS
    $$
        DECLARE
            PAN_HASH VARCHAR(64);
            CVV_HASH VARCHAR(64);
            CustomerID UUID;
            AccountName VARCHAR(255);
            ValidAccounts INT;
        BEGIN
            PAN_HASH = digest(PAN, 'sha256');
            CVV_HASH = digest(CVV, 'sha256');

            SELECT customer_id FROM customer_cards WHERE pan_hash = PAN_HASH AND exp_date = Exp_Date
            AND cvv_hash = CVV_HASH INTO CustomerID;

            SELECT account.name FROM account INNER JOIN customer_account
            ON account.account_id = customer_account.account_id
            WHERE customer_account.customer_id = CustomerID INTO AccountName;

            IF Name = AccountName THEN
                SELECT 1 FROM customer_account WHERE customer_id = CustomerID
                AND balance >= CAST(cost * 100 AS INTEGER) INTO ValidAccounts;
                IF ValidAccounts = 1 THEN
                    RETURN TRUE;
                ELSE
                    RAISE WARNING 'Payment Failed: Insufficient balance to cover transaction.';
                    RETURN FALSE;
                END IF;
            ELSE
                RAISE WARNING 'Payment Failed: Account details do not match.';
                RETURN FALSE;
            END IF;

            EXCEPTION
            WHEN OTHERS THEN
                RAISE WARNING 'Error in check_payment_details: %', SQLERRM;
                RETURN NULL;

        END
    $$ SECURITY DEFINER;
```

Figure: Postgres function to check payment details.

Authorising a loan

It is important for loan officers to be able to authorise loans through the bank. The below function allows them to do that:

```
142  CREATE OR REPLACE FUNCTION authorise_loan(
143      caller UUID, sort_code TEXT, account_num TEXT, amount FLOAT, rate FLOAT, collateral TEXT)
144      RETURNS UUID LANGUAGE plpgsql AS
145      $$
146          DECLARE
147              customerID UUID;
148              employeeID UUID;
149              loanID UUID;
150          BEGIN
151              SELECT customer_id FROM customer_account WHERE sort_code = sort_code
152              AND account_num = account_num INTO customerID;
153
154              SELECT employee_id FROM employee_account WHERE account_id = caller INTO employeeID;
155
156              IF customer_id IS NOT NULL AND employee_id IS NOT NULL THEN
157                  INSERT INTO loans (customer_id, employee_id, amount, rate, collateral)
158                  VALUES (customerID, employeeID, CAST(amount * 100 AS INTEGER),
159                  CAST(rate * 100 AS INTEGER))
160                  RETURNING loan_id INTO loanID;
161              ELSE
162                  RAISE EXCEPTION 'One or more IDs were not recognised.';
163              END IF;
164
165              RETURN loanID;
166
167          EXCEPTION
168              WHEN OTHERS THEN
169                  RAISE WARNING 'Error in authorise_loan: %', SQLERRM;
170                  RETURN NULL;
171          END
172      $$ SECURITY DEFINER;
```

Figure: a Postgres function allowing a Loan Officer to authorise a loan

Access Control

Access control is really important in a database that stores lots of personal information, that would breach GDPR if it was accessed by just anyone.
In this scenario, access control can be managed through group roles, column-level security, and row-level security.
Group roles permit different types of employees or customers to run operations on specific tables. This may limit them to only being able to SELECT, or only INSERT, for example.
Column-level security will allow for users to only view columns that are relevant and needed. For example, a customer does not need to see the hashes of their PAN, PIN and CVV in the customer_cards table, but they should be able to view (and edit!) their billing address.
Finally, row-level security permits users to only view records that are relevant e.g. customers viewing only their own record and no-one else's record.

Access control permissions are decided based on the following assumptions:

- A teller has access to do anything a customer can do, for that customer
- A loan officer can ONLY authorise loans
- A bank manager has access to more of the system, allowing them to do anything any other role can do, and more. This allows them to scale the system according to the business growth.

Group Role Creation

Roles need to be set up for each type of employee, and for customers, to restrict what functions they can run, and what operations they can perform on certain tables.

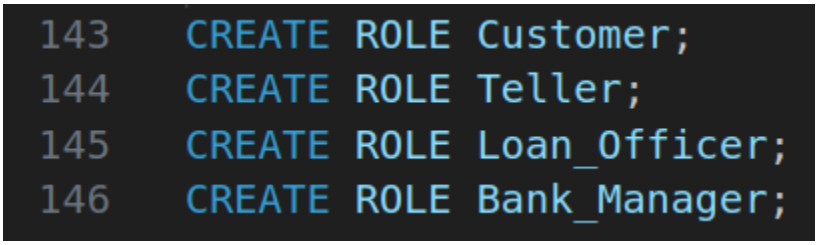

Figure: the Postgres commands to create the roles for each type of employee and customers.

Granting Privileges

Each role group needs to be granted the relevant permissions for their level of access. These are detailed in the below table, in the form [Create/Read/Update/Delete]:

	Customer	Teller	Loan Officer	Bank Manager

account	C,R,U	C,R,U,D	R,U	C,R,U,D
employee_roles	-	R	R	C,R,U,D
employee_account	-	R	R	C,R,U,D
account_types	R	-	-	C,R,U,D
customer_account	R	R	R	C,R,D
customer_cards	R, U	R, U	-	C,R,D
transactions	R	R	R	R
loans	R	R	C,R,U	C,R,U,D
audit_trail	-	-	-	R

These permissions are granted with the presumption that a teller can do anything a customer can do online, so they can help anyone who uses the bank physically. A Loan Officer is allowed to view the financial history of customers in order to ascertain their suitability for a loan, and a Bank Manager can manage and scale the system as much as they want, e.g. by introducing different types of accounts. They also have read access to the audit_trail table in case any incident occurs.

Column-Level Security

In the following tables, it will be specified which types of users can access which columns in each table. This is primarily for viewing, and as such only affects SELECT statements.

Table: account

	Customer	Teller	Loan Officer	Bank Manager
account_id	x	x	x	x
name	x	x	x	x

Table: employee_roles

	Customer	Teller	Loan Officer	Bank Manager

employee_role_id	-	x	x	x
role	-	x	x	x

Customers are not permitted to view any information about employees.

Table: employee_account

	Customer	Teller	Loan Officer	Bank Manager
account_id	-	x	x	x
employee_id	-	x	x	x
employee_role_id	-	x	x	x

Customers are not permitted to view any information about employees.

Table: account_types

	Customer	Teller	Loan Officer	Bank Manager
account_type_id	x	x	x	x
account_type	x	x	x	x

Table: customer_account

	Customer	Teller	Loan Officer	Bank Manager
account_id	-	x	x	x
customer_id	-	-	x	x
account_type_id	-	-	-	x
balance	x	x	x	x
open_date	-	-	-	x
status	-	-	-	x
sort_code	x	x	x	x

account_num	x	x	x	x

Customers only really need to view their balance, and account details so they can share them with others. All other permissions are taken from the brief.

Table: customer_cards

	Customer	Teller	Loan Officer	Bank Manager
customer_id	-	x	-	x
pan_hash	–	-	-	-
exp_date	-	x	-	x
cvv_hash	-	-	-	-
pin_hash	-	-	-	-
address	x	x	-	x

It is not possible for any user (except some superuser) to view the hashed details of the issued bank cards, for security purposes. However, the customer can see their billing address, and the teller can also see their exp_date in order to troubleshoot problems.

Table: transactions

	Customer	Teller	Loan Officer	Bank Manager
transaction_id	x	x	x	x
payer_customer_id	x	x	x	x
payee_customer_id	x	x	x	x
amount	x	x	x	x
datetime	x	x	x	x
reference	x	x	x	x
status	x	x	x	x

It is useful for all users to be able to access a transaction log in order to view their history (if customers or tellers) or to make informed decisions about loans.

Table: loans

	Customer	Teller	Loan Officer	Bank Manager
loan_id	x	x	x	x
customer_id	x	x	x	x
employee_id	-	-	x	x
amount	x	x	x	x
rate	x	x	x	x
datetime	x	x	x	x
amount_paid	x	x	x	x
collateral	x	x	x	x

Table: audit_trail

	Customer	Teller	Loan Officer	Bank Manager
record_id	-	-	-	x
customer_id	-	-	-	x
balance_old	-	-	-	x
balance_new	-	-	-	x
operation	-	-	–	x
datetime	-	-	-	x

Only the Bank Manager has the authority to view the audit trail for the database.

Functions

	Customer	Teller	Loan Officer	Bank Manager
create_account()	x	x	-	x
create_employee()	-	-	-	x
check_employee_role()	-	-	-	x
pay_customer()	x	-	-	-
check_payment_details()	-	-	-	-
authorise_loan()	-	-	x	x

Row-Level Security

The idea of row-level security (RLS) is to limit the records a role can view based on certain conditions. It is not necessary to cover individual tables, but instead to just make the statement: **Customers will only have access to rows that are connected to their account.**
This means employees can view records about customers, but customers are limited to only see the information concerning them. This also affects other operations - customers will not be able to insert, update, delete any more rows than their own.

Implementation

Functions

```
324  GRANT EXECUTE ON FUNCTION create_account TO Customer, Teller, Bank_Manager;
325  GRANT EXECUTE ON FUNCTION create_employee TO Bank_Manager;
326  GRANT EXECUTE ON FUNCTION check_employee_role TO Bank_Manager;
327  GRANT EXECUTE ON FUNCTION pay_customer TO Customer;
328  GRANT EXECUTE ON FUNCTION authorise_loan TO Loan_Officer, Bank_Manager;
```

Figure: Granting permission to execute necessary functions as above.

Bank Manager

```
179  -- Bank Manager --
180  GRANT ALL PRIVILEGES ON account TO Bank_Manager;
181  GRANT ALL PRIVILEGES ON employee_roles TO Bank_Manager;
182  GRANT ALL PRIVILEGES ON employee_account TO Bank_Manager;
183  GRANT ALL PRIVILEGES ON account_types TO Bank_Manager;
184  GRANT SELECT, UPDATE, DELETE ON customer_account TO Bank_Manager;
185  GRANT SELECT(customer_id, exp_date, address), UPDATE, DELETE ON customer_cards TO Bank_Manager;
186  GRANT SELECT ON transactions TO Bank_Manager;
187  GRANT ALL PRIVILEGES ON loans TO Bank_Manager;
188  GRANT SELECT ON audit_trail TO Bank_Manager;
```

Figure: SQL statements to grant the necessary privileges to the bank manager, as discussed above.

Loan Officer

```
190  -- Loan Officer --
191  GRANT SELECT, UPDATE ON account TO Loan_Officer;
192  GRANT SELECT ON employee_roles TO Loan_Officer;
193  GRANT SELECT ON employee_account TO Loan_Officer;
194  GRANT SELECT(account_id, customer_id, balance, sort_code, account_num)
195      ON customer_account TO Loan_Officer;
196  GRANT SELECT ON transactions TO Loan_Officer;
197  GRANT SELECT, UPDATE, INSERT ON loans to Loan_Officer;
```

Figure: SQL statements to grant the necessary privileges to loan officers, as discussed above.

Teller

```
199  -- Teller --
200  GRANT SELECT, UPDATE, INSERT, DELETE ON account TO Teller;
201  GRANT SELECT ON employee_roles TO Teller;
202  GRANT SELECT ON employee_account TO Teller;
203  GRANT SELECT(account_id, balance, sort_code, account_num) ON customer_account TO Teller;
204  GRANT SELECT(customer_id, exp_date, address), UPDATE ON customer_cards TO Teller;
205  GRANT SELECT ON transactions TO Teller;
206  GRANT SELECT(loan_id, customer_id, amount, rate, datetime, amount_paid, collateral)
207      ON loans to Teller;
```

Figure: SQL statements to grant the necessary privileges to tellers, as discussed above.

Customer

```
209   -- Customer --
210   GRANT SELECT, INSERT, UPDATE ON account TO Customer;
211   GRANT SELECT ON account_types to Customer;
212   GRANT SELECT(balance, sort_code, account_num) ON customer_account to Customer;
213   GRANT SELECT(address), UPDATE ON customer_cards to Customer;
214   GRANT SELECT ON transactions to Customer;
215   GRANT SELECT(loan_id, customer_id, amount, rate, datetime, amount_paid, collateral)
216       ON loans to Customer;
```

Figure: SQL statements to grant the necessary privileges to customers, as discussed above.

The following queries set up row-level security on the account table, so that customers can only access their own record. However, other types of users are not restricted i.e. employees.

```
218   ALTER TABLE account ENABLE ROW LEVEL SECURITY;
219   ALTER TABLE account FORCE ROW LEVEL SECURITY;
220
221   CREATE POLICY account_rls
222   ON account
223   USING (
224       EXISTS (
225           SELECT 1 FROM customer_account
226           WHERE customer_account.account_id = account.account_id
227       )
228       AND name = CURRENT_USER
229   );
```

Figure: Postgres commands to enable and configure row-level security on the account table.

The following queries enable RLS on the customer_account table so customers can only access their own records.

```
221   CREATE POLICY customer_account_rls
222   ON customer_account
223   USING (
224       EXISTS (
225           SELECT 1
226           FROM account
227           JOIN customer_account ON account.account_id = customer_account.account_id
228           WHERE account.name = CURRENT_USER
229       )
230   );
```

Figure: Postgres commands to enable and configure RLS on the customer_account table.

```
246  ALTER TABLE customer_cards ENABLE ROW LEVEL SECURITY;
247  ALTER TABLE customer_cards FORCE ROW LEVEL SECURITY;
248
249  CREATE POLICY customer_cards_rls
250  ON customer_cards
251  USING (
252      EXISTS (
253          SELECT 1
254          FROM account
255          JOIN customer_account ON account.account_id = customer_account.account_id
256          WHERE account.name = CURRENT_USER
257          AND customer_account.customer_id = customer_cards.customer_id
258      )
259  );
```

Figure: Postgres commands to enable and configure RLS on the customer_cards table.

```
261  ALTER TABLE transactions ENABLE ROW LEVEL SECURITY;
262  ALTER TABLE transactions FORCE ROW LEVEL SECURITY;
263
264  CREATE POLICY transactions_rls
265  ON transactions
266  USING (
267      EXISTS (
268          SELECT 1
269          FROM account
270          JOIN customer_account ON account.account_id = customer_account.account_id
271          WHERE account.name = CURRENT_USER
272          AND (customer_account.customer_id = transactions.payer_customer_id OR
273          customer_account.customer_id = transactions.payee_customer_id)
274      )
275  );
```

Figure: Postgres commands to enable and configure RLS on the transactions table.

```
277  ALTER TABLE loans ENABLE ROW LEVEL SECURITY;
278  ALTER TABLE loans FORCE ROW LEVEL SECURITY;
279
280  CREATE POLICY loans_rls
281  ON loans
282  USING (
283      EXISTS (
284          SELECT 1
285          FROM account
286          JOIN customer_account ON account.account_id = customer_account.account_id
287          WHERE account.name = CURRENT_USER
288          AND customer_account.customer_id = loans.customer_id
289      )
290  );
```

Figure: Postgres commands to enable and configure RLS on the loans table.

Role-Based Security

Role-based Security (RBS) is the idea of securing asset/system access through roles. Each user of the system is allocated a, or multiple, roles. Each role is associated with different privileges which are then granted to the user which has the role. This structured security makes it very easy to maintain and verify, as well as modular: it is possible to change all the Teller's privileges at once, for example. When it is compared to other methods of access control, such as Mandatory or Discretionary Access Control, it balances well control over permissions and scalability.

In the banking institution's financial management system, it plays a key role in securing personal and financial information. For example, customers should not be able to view any information about employees - so any database tables that contain that information cannot be viewed, edited, appended to, or deleted by anyone with the customer role. Secondly, the database can set up a role for Loan Officers such that they can create, update, and read information about loans in the respective database table, and Tellers can read that information, but cannot perform any other operations on it. In this way, role-based security in the financial management system ensures confidentiality of necessary data, and also maintains the integrity of data within the database.

Advantages of Role-Based Security

Enhanced Security and Compliance

RBS ensures users comply with the least-privilege principle, limiting their access to only that which is needed. Secondly, strict access controls allows the organisation to meet regulatory requirements such as GDPR (UK Government 2016).

Simplified User Management

User access is much easier to manage through roles than individual permissions as access doesn't need to be changed on an individual basis, just for the role. It is also simple to assign users to roles e.g. if they were just hired, or if they changed permissions, and their privileges would automatically adjust.

Reduced Administrative Workload

Administrators don't need to waste time individually managing the permissions of each user of the system. Instead, the roles can be assigned manually or even automatically. This makes the system easily scalable as well, as new employees may be hired.

Minimised Risk of Error

If one were to manually assign permissions to each individual user of a system, there are massive opportunities for human error to affect certain people. However, user privileges are synced to role privileges, so they are the same for everyone with that particular role.

Minimising Risk of Unauthorised Access

In order to secure sensitive financial data, it is important to implement security controls both within the configuration of the database, and outside of the database.

Measure	Description
MFA	This ensures that user accounts cannot be accessed by unauthorised individuals, since several methods of authentication are necessary
Least Privilege Principle	Only allocate the absolutely necessary privileges to a role to allow them to complete their job. Excess privileges should be removed.
Encryption at rest	Encrypt the database when it is not in use, using a cryptographically secure algorithm.
Encryption in transit	If data needs to be transmitted, ensure it is accessed through HTTPS from the Internet, or encrypted using TLS1.3
Firewall	Implement a firewall between the database and any public networks.
Network Segmentation	Isolate the transmission of the sensitive information from general network traffic by segmenting the network, in orde to ensure privacy.
Security Audits / Vulnerability Assessments	Regularly hire outside security professionals to perform security audits or vulnerability assessments on the network and database, and ensure that any issues they find are patched ASAP.
Monitoring of access logs	Continuously monitor access logs and database activity.

Detect and Respond to Suspicious Activities

In order to detect and respond, implement an Intrusion Detection & Prevention System (IDPS) that uses signature- and heuristic-based algorithms to identify known attack patterns from malware or attacks. This should also include automated response mechanisms such as locking accounts after multiple failed password attempts, and blocking accounts trying to make repeated unauthorised changes.

Data Auditing

The *audit_trail* table in the database tracks changes to the *customer_account* table. Since transaction history can only be edited through functions, it doesn't need to be recorded. Updates are not possible to the table, beyond the function to handle the original transaction.

```
15  CREATE OR REPLACE FUNCTION audit_this()
16  RETURNS TRIGGER AS $$
17      BEGIN
18          INSERT INTO audit_trail(customer_id, balance_old, balance_new, operation)
19          VALUES (OLD.customer_id, OLD.balance, NEW.balance, TG_OP);
20          RETURN NEW;
21      END $$
22  LANGUAGE plpgsql;
23
24  CREATE TRIGGER customer_account_update_trigger
25  BEFORE UPDATE ON customer_account
26  FOR EACH ROW EXECUTE FUNCTION audit_this();
```

Figure: SQL function and trigger to capture changes to the customer_account table.

The above function and trigger will run whenever a record is updated in the *customer_account* table, capturing the original balance, time, and the new balance, and which customer it affected. This is sufficient to record all changes to the *customer_account* table.

Data Auditing and Logging in Security Breaches and Data Integrity

Detection of Security Breaches

Logging records all access to the database or system, and allows for checking of logins to search for unauthorised access to the database, particularly from unusual IP addresses, for example.
These logs may also reveal anomalies in database access, which may be the result of or part of a cyber-attack.
Causes the database system to meet regulatory requirements because of the audit trail, since logging is a necessary mechanism to prevent and detect breaches.

Ensuring Data Integrity

Auditing allows for the tracking of changes to a table, both authorised and unauthorised, and can be used to identify unauthorised changes.

It can also be used to rollback changes to tables before any incident by tracking the original values of the table prior to any breach or attack.

Login

User	**Password**
Customer	c
Teller	t
Loan_Officer	lo
Bank_Manager	bm

References

PCI Security Standards Council. 2018. “PCI-DSS v3.2.1 Quick Reference Guide.” https://listings.pcisecuritystandards.org/documents/PCI_DSS-QRG-v3_2_1.pdf.

UK Government. 2016. “Regulation (EU) 2016/679 of the European Parliament and of the Council.” https://www.legislation.gov.uk/eur/2016/679/contents.