LEAKPEAKPROJECTREPORT

GROUP MEMBERS:

ABDUL HADI KHAN (22K-4724)

ZEHRA QURESHI (22K-4744)

MUHAMMAD SOHAIB (22K-4751)

Project Overview:

Introduction

The "Leak Peak" project is a comprehensive web application designed to manage user information, subscriptions, and monitor data breaches. It aims to provide a secure platform for users and administrators, allowing them to track user activities, manage subscriptions, and identify potential security threats. This project leverages modern web technologies and follows best practices in security and database management.

Objectives

- To create a user-friendly interface for users and administrators to manage accounts and subscriptions.
- To implement robust authentication and authorization mechanisms using JWT and session management.
- To track user activities through logging and provide detailed reports for security audits.
- To monitor and manage subscription plans, ensuring users are informed about renewals and payment details.
- To maintain a record of compromised accounts and breaches, providing users with insights into dtheir security status.

Technology Stack

- Frontend: HTML, CSS (Tailwind CSS), JavaScript
- Backend: Node.js, Express.js
- Database: MySQL
- Authentication: JSON Web Tokens (JWT), Express-Session
- Email Notifications: Nodemailer
- Task Scheduling: Node-Cron

Key Features

1. User Management:
    - Registration and login functionality for users and administrators.
    - Role-based access control to differentiate between user and admin functionalities.
2. Subscription Management:
    - Ability to view, manage, and renew subscription plans.
    - Integration of payment details for premium users, including credit card information.
3. User Activity Logs:
    - Recording of user actions for auditing and monitoring purposes.
    - A dedicated page to display user logs, accessible by authorized personnel.
4. Breach Monitoring:
    - Tracking of compromised accounts and related breaches.
    - Users can check if their accounts have been involved in any data breaches.
5. Email Notifications:
    - Automated email reminders for subscription renewals.
    - Alerts for users regarding potential security threats related to their accounts.
6. Admin Management:
    - Admins can manage users, view logs, and handle administrative tasks such as adding or deleting accounts.

Database Design

The database for the "Leak Peak" project is designed using an Entity-Relationship (ER) model, which includes the following entities and relationships:

- Entities:
- Users: Stores user information including email, username, password, and user category.
- Admins: Contains administrator details with their role and credentials.
- User _Logs: Records actions performed by users along with timestamps.
- Subscriptions: Manages user subscriptions, including plan details and renewal dates.
- Breach_Types: Defines various types of data breaches.
- Compromised_Accounts: Records accounts that have been compromised.
- Subscription_Plans: Holds information about different subscription offerings.
- Breaches: Details specific breaches, including their type and source.
- Countries: Lists countries associated with users and credit cards.
- Credit_Cards: Manages credit card information linked to users.
- Relationships:
- Users can have multiple logs, subscriptions, and compromised accounts.
- Admins can manage multiple users and track deletion events.
- Subscriptions are linked to specific subscription plans, and breaches are categorized by type and source.

User Interface

The user interface is designed to be intuitive and responsive, utilizing Tailwind CSS for styling. Key pages include:

- Home Page: Introduction to the application and navigation to different sections. ● User Logs Page: Displays logs of user activities in a structured table format.
- Admin Management Page: Allows admins to manage user accounts and view logs.
- Subscription Management Page: Enables users to view and manage their subscription plans.

Security Considerations

The project implements several security measures to protect user data:.

- JWT is used for authentication, ensuring secure communication between the client and server.
- CORS is configured to restrict access to the API from unauthorized origins.
- Regular monitoring of user activities helps identify suspicious behavior.

Project Flow:

The flow of the "Leak Peak" project outlines the sequence of operations and interactions between users, administrators, and the system. This section provides a detailed description of how the application functions from user registration to breach monitoring, ensuring a clear understanding of the user journey and system processes.

1. **User Registration and Authentication**

    - User Registration:
    - Users visit the registration page and fill out a form with their email, username, and password.
    - Upon submission, the system validates the input and creates a new user entry in the database..
    - User Login:
    - Users access the login page and enter their credentials (email and password).
    - The system checks the credentials against the database. If valid, a JWT is generated and stored in local storage for session management.
    - Users are redirected to the dashboard or home page upon successful login.

2. **User Dashboard**

    - After logging in, users access their dashboard, which displays:
    - Overview of their account status.

- Subscription details, including the current plan and renewal dates.
- Links to view user logs and breach monitoring.

3. Subscription Management

- Viewing Subscriptions:
- Users can navigate to the subscription management section to view available plans and their features.
- Users can upgrade, downgrade, or renew their subscriptions as needed.
- Payment Processing:
- For premium subscriptions, users enter their credit card information securely.
- The system processes the payment and updates the subscription status in the database.

4. User Activity Logging

- Action Tracking:
- Every action performed by users (e.g., login, logout, subscription changes) is recorded in the User_Logs table.
- Each log entry includes the user ID, action type, and timestamp for auditing purposes.
- Viewing Logs:
- Users can view their activity logs on the user logs page, where actions are listed in a structured table format.

5. Breach Monitoring

- Breach Checks:
- Users can check if their accounts have been compromised by querying the Compromised_Accounts table.
- The system compares user email addresses with known compromised accounts and displays relevant information.
- Alerts and Notifications:
- If a breach is detected that affects a user, the system sends an alert via email, informing them of the potential risk and recommended actions.

6. Admin Management

- Admin Login:
- Administrators log in through a secure admin portal using their credentials.
- Admins receive a higher level of access to manage user accounts and view all user logs.
- User Management:
- Admins can view, edit, or delete user accounts as necessary.
- Admin actions are logged in the Admin_Deletion table to maintain a record of changes.
- Monitoring User Activity:
- Admins can access a comprehensive view of user logs to monitor user activities and identify any suspicious behavior.

7. Data Breach Management

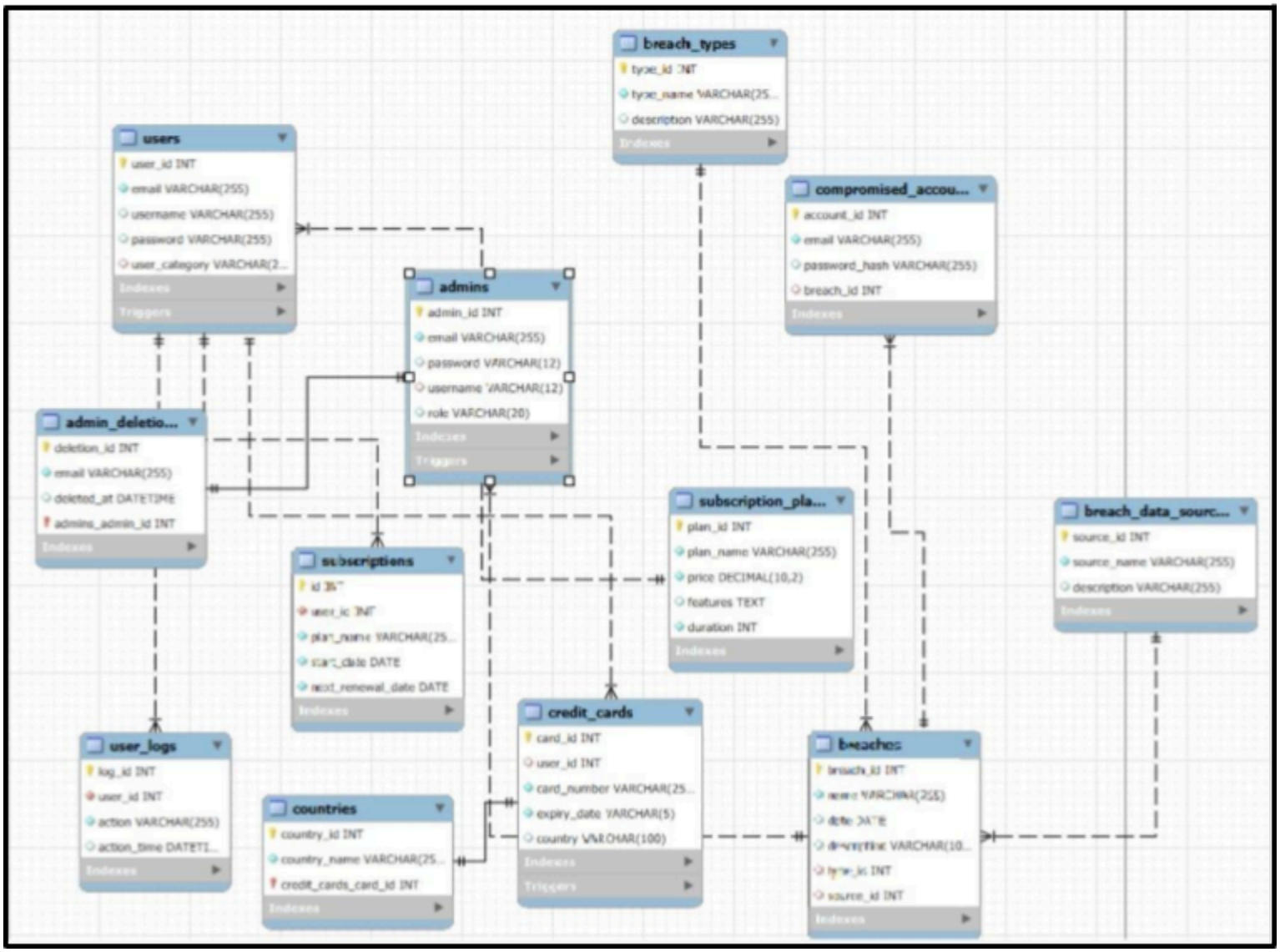
- Recording Breaches:

- When a data breach occurs, details are recorded in the Breaches table, including the type of breach, source, and affected accounts.
- The system categorizes breaches by type and source for easier analysis. ● Breach Reports:
- Admins can generate reports on breaches to assess the impact and take necessary actions to enhance security measures.

8. User Logout

- Users can log out from the application, which invalidates the JWT stored in local storage.
- The system clears user session data, ensuring that sensitive information is not accessible after logout.

ER Diagram:



1. First Normal Form (1NF)

```
CREATE TABLE `admin_deletions` (
  `deletion_id` INT NOT NULL AUTO_INCREMENT,
  `email` VARCHAR(255) NOT NULL,
  `deleted_at` DATETIME DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`deletion_id`)
```

);

2. Second Normal Form (2NF)

Using admin_deletions:

- The primary key is deletion_id.
- email and deleted_at depend entirely on deletion_id.

3. Third Normal Form (3NF)

In the schema:
The admin_deletions table has no transitive dependencies:

- email and deleted_at depend directly on deletion_id.

4. Boyce-Codd Normal Form (BCNF)

In the schema:

For admin_deletions, the only functional dependency is deletion_id → (email, deleted_at), where deletion_id is the superkey.

JOINS

```
const query = `
  SELECT
  compromised_account.email,
  compromised_account.password_hash,
  breaches.name AS breach_name,
  breaches.date AS breach_date,
  breaches.description AS breach_description,
  breach_types.type_name AS breach_type,
  breach_data_sources.source_name AS breach_source,
  COUNT(breaches.breach_id) AS breach_count
FROM compromised_account
JOIN breaches ON compromised_account.breach_id = breaches.breach_id
LEFT JOIN breach_types ON breaches.type_id = breach_types.type_id
LEFT JOIN breach_data_sources ON breaches.source_id = breach_data_sources.source_id
WHERE compromised_account.email = ?
GROUP BY breach_types.type_name, breaches.name, breaches.date, breaches.description, breach_data_sources.source_name, compromised_account.password
ORDER BY breach_count DESC;
`;
```

UPDATE

```
// Update user category
await connection.promise().query(
  `UPDATE users SET user_category = ? WHERE user_id = ?`,
  [user_category, userId]
);
```

READ

```
app.get('/user-logs', ensureAdmin, (req, res) => {
  const query = 'SELECT * FROM user_logs ORDER BY action_time DESC'; // Fetch logs ordered by timestamp

  connection.query(query, (err, results) => {
      if (err) {
          console.error('Error fetching user logs:', err);
          return res.status(500).json({ error: 'Internal server error' });
      }
      res.json(results); // Return the logs directly in the response
  });
});
```

CREATE

```
const createDatabaseQuery = "CREATE DATABASE IF NOT EXISTS db_project";


// Execute the query to create the database
connection.query(createDatabaseQuery, (err, results) => {
  if (err) {
    console.error('Error creating database:', err);
    return;
  }
  console.log('Database created or already exists:', results);


  // Switch to the newly created or existing database
  connection.changeUser({ database: 'db_project' }, (err) => {
    if (err) {
      console.error('Error switching to database:', err);
      return;
    }


    // SQL query to create a new table breach_data_sources
    const createTableBreachDataSources = `
      CREATE TABLE IF NOT EXISTS breach_data_sources (
        source_id INT AUTO_INCREMENT PRIMARY KEY,
        source_name VARCHAR(255) NOT NULL,
        description VARCHAR(255)
      )`;
```

TRIGGERS

```
DELIMITER $$

CREATE TRIGGER validate_card_expiry
BEFORE INSERT ON credit_cards
FOR EACH ROW
BEGIN
    -- Convert CURDATE() to MM/YY format
    DECLARE current_expiry_date VARCHAR(5);
    SET current_expiry_date = DATE_FORMAT(CURDATE(), '%m/%y');

    -- Compare the NEW.expiry_date with the current date
    IF NEW.expiry_date < current_expiry_date THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Card expiry date cannot be in the past.';
    END IF;
END$$

DELIMITER ;
```

```
////////////////////////////////////////////////////////////ADMIN DELETION//////////////////////////////////////////////////////////////////////////////
DELIMITER $$

CREATE TRIGGER log_admin_deletion
AFTER DELETE ON admins
FOR EACH ROW
BEGIN
    INSERT INTO admin_deletions (admin_id, deleted_at)
    VALUES (OLD.admin_id, NOW());
END$$

DELIMITER ;
```

```
////////////////////////////////////////////////////////////REGISTER USER//////////////////////////////////////////////////////////////////////////////
DELIMITER $$

CREATE TRIGGER after_user_insert
AFTER INSERT ON users
FOR EACH ROW
BEGIN
  INSERT INTO user_logs (user_id, action, action_time)
  VALUES (NEW.user_id, 'User Registered', NOW());
END$$

DELIMITER ;
```

PROCEDURE

```
function getUserCategoryByEmail(email) {
  return new Promise((resolve, reject) => {
    // Call the stored procedure
    connection.execute(
      'CALL GetUserCategoryByEmail(?)', [email],
      (err, results) => {
        if (err) {
          reject(err);
        } else {

          resolve(results[0]);
        }
      }
    );
  });
}
```

```
DELIMITER $$

CREATE PROCEDURE GetUserCategoryByEmail(IN user_email VARCHAR(255))
BEGIN
    SELECT user_category
    FROM users
    WHERE email = user_email;
END $$
SHOW PROCEDURE STATUS WHERE Name = 'GetUserCategoryByEmail';
```

TRANSACTION

```
app.post('/register-user', async (req, res) => {
  const { email, username, password, user_category, cardNumber, expiryDate, country } = req.body;

  if (!email || !username || !password || !user_category) {
    return res.status(400).json({ error: 'Email, username, password, and user category are required' });
  }

  if (user_category === 'premium' && (!cardNumber || !expiryDate || !country)) {
    return res.status(400).json({ error: 'Card details and country are required for premium users' });
  }

  try {
    // Begin the transaction
    await connection.promise().beginTransaction();

    // Insert into the `users` table
    const [userResult] = await connection.promise().query(
      `INSERT INTO users (email, username, password, user_category) VALUES (?, ?, ?, ?)`,
      [email, username, password, user_category]
    );

    const userId = userResult.insertId;

    if (user_category === 'premium') {
      // Insert into the `credit_cards` table for premium users
      await connection.promise().query(
        `INSERT INTO credit_cards (user_id, card_number, expiry_date, country) VALUES (?, ?, ?, ?)`,
        [userId, cardNumber, expiryDate, country]
      );
    }

    // Commit the transaction
    await connection.promise().commit();

    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    // Roll back the transaction in case of an error
    await connection.promise().rollback();
    console.error('Error during user registration transaction:', error);
    res.status(500).json({ error: 'Failed to register user' });
  }
```

```
CREATE TABLE user_logs (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    action VARCHAR(255) NOT NULL,
    action_time DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);
```