Workbook – Database Server Management

This workbook is designed for students to explore PostgreSQL security and access control in a fun, themed context inspired by The Hitchhiker's Guide to the Galaxy. All exercises are performed on a dedicated workspace database called galaxydb. The tasks cover creating roles and users, managing privileges via ACLs and DCL, and implementing fine-grained attribute-based access control (ABAC) using row-level security (RLS).

1 Workspace Preparation

1.1 Create a Dedicated Workspace Database

Set up your working environment by creating a dedicated PostgreSQL database named galaxydb. This database will be used exclusively for all exercises in this workbook. This step ensures that all operations, including role and table creation, are isolated from any production data.

```sql
CREATE DATABASE galaxydb;
```

Reflection Questions:

- Why is it important to use a dedicated database for experiments?
- How does isolating your work in **galaxydb** help protect production systems?

2 Role-Based Access Control (RBAC)

2.1 Create Basic Roles and Users for the Hitchhiker's Guide Scenario

Using the Hitchhiker's Guide to the Galaxy theme, create the following roles and users in the **galaxydb** database:

- Roles
  - **hitchhiker** (to represent readers with read-only access)
  - **guide_editor** (to represent authors who can add or edit guide entries)
  - **galactic_president** (to represent the all-powerful role with full privileges)
  - **vogon** (to represent a role with no access privileges)
- Users
  - **arthur** (assigned the role **hitchhiker**)
  - **ford** (assigned both **hitchhiker** and **guide_editor**)
  - **trillian** (assigned the role **guide_editor**)

  - **zaphod** (assigned the role **galactic_president**)

The goal is to create these entities and then assign the roles to the respective users.

```
-- Connect to galaxydb

-- Create roles
CREATE ROLE hitchhiker;
CREATE ROLE guide_editor;
CREATE ROLE galactic_president;
CREATE ROLE vogon;

-- Create users with sample passwords
CREATE USER arthur WITH PASSWORD 'secret';
CREATE USER ford WITH PASSWORD 'secret';
CREATE USER trillian WITH PASSWORD 'secret';
CREATE USER zaphod WITH PASSWORD 'secret';

-- Assign roles to users according to the theme
GRANT hitchhiker TO arthur;
GRANT hitchhiker, guide_editor TO ford;
GRANT guide_editor TO trillian;
GRANT galactic_president TO zaphod;
```

Reflection Questions:

- How does grouping privileges into roles (such as **hitchhiker** or **guide_editor**) simplify permission management?

Discussion Points:

- Discuss the principle of least privilege and why it is essential to assign only the necessary roles.

2.2 Implement Role Inheritance for Read-Only Access

Create an additional role called **read_only** that grants SELECT privileges on all tables within the public schema. Then, assign (inherit) the **read_only** privileges to the **hitchhiker** role. This task reinforces the concept of role inheritance so that all users with the **hitchhiker** role automatically gain read-only access to database objects.

```
-- Create a read_only role that grants SELECT privileges on all tables in the public schema
CREATE ROLE read_only;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only;

-- Allow the hitchhiker role to inherit the read_only privileges
GRANT read_only TO hitchhiker;
```

Reflection Questions:

- What are the benefits of role inheritance when managing a large number of users?
- How does granting **read_only** privileges to the **hitchhiker** role reduce administrative overhead?

Discussion Points:

- Consider potential issues that might arise if inherited privileges need to be revoked later.
- Discuss scenarios where dynamic privilege updates via inheritance can be particularly useful.

2.3 Verify Roles and Memberships

Ensure that all roles and user memberships are correctly established in the **galaxydb** database by querying PostgreSQL's system catalogs. Verify that users such as **arthur**, **ford**, **trillian**, and **zaphod** are properly assigned to their respective roles (and inherited roles like **read_only**).

```
-- Using psql meta-command to list roles and memberships:
\du

-- Alternatively, query system catalogs directly:
SELECT rolname FROM pg_roles;
SELECT * FROM pg_auth_members;
```

Reflection Questions:

- How can these queries help verify that role inheritance is working as intended?

Discussion Points:

- Compare the benefits of using built-in psql commands versus direct SQL queries for auditing purposes.
- Discuss the importance of regularly reviewing role memberships for maintaining security.

3 ACL, DCL, and ABAC (RLS) Exercises

3.1 Create Tables and Set Object-Level Privileges

Within the **galaxydb** database, create two tables with a Hitchhiker's Guide flavor:

- **GuideEntries**: to store general guide information.
- **Planets**: to store planetary data (e.g., names and descriptions).

Assign privileges using GRANT and REVOKE commands:

- For **GuideEntries**:

  - Grant SELECT to the **hitchhiker** role
  - Grant SELECT, INSERT, and UPDATE to the **guide_editor** role
  - Grant ALL privileges to the **galactic_president** role
  - Revoke all privileges from the **vogon** role
- For **Planets**:
  - Grant SELECT to both hitchhiker and **guide_editor** roles
  - Grant ALL privileges to the **galactic_president** role

```sql
-- Create the GuideEntries table for the guide's content
CREATE TABLE GuideEntries (
  id SERIAL PRIMARY KEY,
  title TEXT,
  content TEXT
);

-- Create the Planets table for planetary data
CREATE TABLE Planets (
  id SERIAL PRIMARY KEY,
  name TEXT,
  description TEXT
);

-- Set privileges on GuideEntries
GRANT SELECT ON GuideEntries TO hitchhiker;
GRANT SELECT, INSERT, UPDATE ON GuideEntries TO guide_editor;
GRANT ALL PRIVILEGES ON GuideEntries TO galactic_president;
REVOKE ALL ON GuideEntries FROM vogon;

-- Set privileges on Planets
GRANT SELECT ON Planets TO hitchhiker, guide_editor;
GRANT ALL PRIVILEGES ON Planets TO galactic_president;
```

Reflection Questions:

- How do GRANT and REVOKE commands help enforce security on specific database objects?
- What are the implications of revoking all privileges from a role like **vogon**?

Discussion Points:

- Discuss the importance of fine-grained control using object-level privileges.
- Reflect on how these controls can prevent unauthorized data access in a live environment.

3.2 Implement ABAC Using Row-Level Security (RLS) with the Hitchhiker Theme

Design a table called **SecretGuideEntries** that contains sensitive guide entries, including a region attribute (for example, "Betelgeuse" or "Vogsphere"). Enable Row-Level Security (RLS) on this table and create a policy so that only rows where the region matches a session variable (**my.region**) are visible. This simulates attribute-based access control (ABAC) within the database.

- **Table Name**: SecretGuideEntries
- **Columns**: id, title, content, region
- RLS Policy: Users can only view rows where **region = current_setting('my.region', true)**

```
-- Create the Table
CREATE TABLE SecretGuideEntries (
  id SERIAL PRIMARY KEY,
  title TEXT,
  content TEXT,
  region TEXT
);

-- Enable Row Level Security
ALTER TABLE SecretGuideEntries ENABLE ROW LEVEL SECURITY;

-- Create the RLS Policy
CREATE POLICY region_policy ON SecretGuideEntries
    USING (region = current_setting('my.region', true));
```

Test the RLS Policy:

```
-- Set the session variable to a specific region, e.g., 'Betelgeuse'
SET my.region = 'Betelgeuse';

-- Insert sample data
INSERT INTO SecretGuideEntries (title, content, region) VALUES
('Entry A', 'Secrets of the Universe', 'Betelgeuse'),
('Entry B', 'Guide to Interstellar Travel', 'Vogsphere');

-- Query the table (only rows with region 'Betelgeuse' will be visible)
SELECT * FROM SecretGuideEntries;
```

Reflection Questions:

- How does RLS implement ABAC compared to traditional table-level ACLs?

- What advantages does filtering rows based on a session attribute offer?

Discussion Points:

- Discuss how this RLS policy could be expanded to support multiple attributes (for example, combining department and region).
- Compare the security benefits of RLS with the flexibility of traditional ACL methods.

4 Database Authentication and Local Configuration

4.1 Configure Local Access Without a Password

Configure PostgreSQL so that the specific user **zaphod** can connect to the dedicated database **galaxydb** without a password. This task requires modifying the **pg_hba.conf** file to allow local connections. The configuration will use either the **trust** method for TCP/IP connections or the **peer** method for Unix-domain socket connections. This setup is intended for development purposes where convenience is key, while restricting the rule only to **zaphod** and the **galaxydb** database.

Description of the **pg_hba.conf**:

- **TYPE:** Specifies the connection type.
  - **local** – for Unix-domain socket connections.
  - **host** – for TCP/IP connections.
- **DATABASE:** Indicates the database that the rule applies to. In this case, it is set to **galaxydb** to limit the rule to our workspace.
- **USER:** Specifies the PostgreSQL user for which the rule applies. Here it is set to **zaphod** so that only this user is affected.
- **ADDRESS:** For **host** rules, this defines the allowed client IP address range in CIDR notation (e.g., **127.0.0.1/32**). For **local** connections, this field is not used.
- **METHOD:** Specifies the authentication method.
  - **trust** – no password is required.
  - **peer** – uses the operating system's username for authentication.

Example Configuration:

```
# For TCP/IP connections from localhost:
host    galaxydb       zaphod         127.0.0.1/32         trust

# Alternatively, for Unix-domain socket connections using peer authentication:
local   galaxydb       zaphod                             peer
```

After editing **pg_hba.conf**, reload the PostgreSQL configuration to apply the changes.

Reflection Questions:

- How does restricting the rule to the **galaxydb** database and the user **zaphod** enhance security compared to a global rule?
- What are the benefits and potential risks of using the **trust** authentication method for local connections?

Discussion Points:

- Discuss scenarios in which allowing password-less access for a specific user (e.g., **zaphod**) is acceptable, and when it might introduce security risks.
- Compare the trade-offs between using **trust** versus **peer** authentication for local development environments.