Securing Your PostgreSQL Data: A Comprehensive Guide to Protecting Your Database Assets

Hettie Dombrovskaya
Database Architect
SCALE 2024
Who am I

Hettie Dombrovskaya
Database Architect at DRW
Local Organizer of Chicago PostgreSQL User Group

PG Day Chicago is on April 26, 2024!
Why this talk?
- We live in an age of data breaches
- Securing data is a high priority
- PostgreSQL has everything
- Still...
What will be covered

• Common security challenges
• The necessity of standardization and its role in solving these challenges
• Adopted security models and their practical implementations
• Addressing a wide spectrum of access control needs
• Using automation to streamline security
• Ongoing issues and future prospects
Challenge #1: PostgreSQL does not force you to create roles and schemas in order to start.

And all examples in documentation create objects in PUBLIC schema!
As a result...

- Applications are developed using postgres user
- When they move to production, developer either forget to change the user or run into permissions problems they do not have time to fix
does not know how to fix
- When an application uses connection pools different application users can connect as the same database user
Challenge #2: The wonders of inheritance

- Starting from PG 7.3, there is no distinction between users and roles (user=role+login)
  
  create role role1;
  create role role2 login password 'pwd';
  create user user1 password 'pwd';

- All of the grants below will work:
  
  grant role1 to role2;
  grant role2 TO user1;
  grant user1 to role2;

  ... and if later you will execute
  
  create role role3;
  grant role3 to role1 ---will be inherited
Challenge #3: You think you created a role for a database? 
Think again!

- Roles are created on the instance level, not the database level
- If there are several databases on one instance, all users will have access to all databases, because...
  By default, all user have CONNECT privilege to all databases on the instance
- Until PG 15, all users could create objects in PUBLIC schema. That includes public schema in all databases on the same instance.
- If a customer requested a superuser privilege, this superuser will be able to do everything on all databases on that instance.
Trying to do things the right way!

**Grouping (objects and users):**
- Using schemas for access control: all objects in each schema have the same set of privileges
- Granting privileges to groups (nologin roles) only. And granting roles to users

```sql
create schema orders owner orders_owner;
grant orders_owner to orders_admin;
create role orders_read_write;
create role orders_read_only;
grant select on all tables in schema orders to orders_read_only;
grant select, insert, update, delete on all tables in schema orders to orders_read_write;
```

**What is not going to work?**
Challenge #4: Default privileges

- Yes, you also need to **grant usage**!
  grant usage on schema orders to orders_read_write, orders_read_only

- What else?
  alter default privileges in schema orders grant select on tables to orders_read_only;
  alter default privileges in schema orders grant select, insert, update, delete on tables to orders_read_write;

Now:
create table orders.customer (
    customer_id int primary key,
    customer_name text);

- Why were default permissions not applied?!
  alter default privileges in schema orders for role orders_owner grant select, insert, update, delete on tables to orders_read_write;
Challenge #5: The wonders of ownership!

- When you run:

create schema orders owner orders_owner;

It created a lot of privileges for orders_owner user:

grant all on schema orders to orders_owner

- But what happens when you execute

alter schema orders owner new_orders_owner;

Does anything change with permissions?
Challenges #6, #7, #8... Lots of weird things!

grant select orders.sales_points to role1;
grant insert, update, delete on orders.sales_points to role2;
grant role1 to user1;
grant role2 to user1;
revoke delete on orders.sales_points from user1;

Will this work?
- It won’t, and moreover, errors won’t be reported:
  REVOKE of permissions which are not granted
  GRANT permissions which are already granted except for roles
- You can’t drop user that has any privileges
- You can’t drop role cascade
- And there is no easy way to see what permissions a given user has!
Now imagine you have not five, not ten, but 280 databases, and new requests are coming each day!
We want to be isolated!

A separate instance for each new project – possible, but expensive.

What are the alternatives?
Security Models Overview
Principles and implementation
Basic principles

The only security model to support multi-tenancy within one PostgreSQL database

Principle of least privilege

- A user is given the minimum levels of access needed to perform their job functions.

Durability

- Non-superuser users do not have a way to bypass security settings

Flexibility

- One package supports four security models with different permissions hierarchy.
Key features

Event trigger
- Forces all objects in each schema to be owned by the schema owner role and assigns default privileges

Security levels matrix
- Schema owner TRUE/FALSE
- Account owner TRUE/FALSE

Database level security
- Security modal is set up on the database level

Security-definer functions
- Schemas and roles creation/deletion are performed using security definer functions
Enabling security model

- Deploy the package
- If the package was previously deployed, the previous security settings will be used:
  you can’t change the existing settings for a database
- If that’s the first deployment run
  `select * from grant_create_schema_users(Boolean, Boolean)`
  This will
  - record security setting in the database
  - enable event trigger
  - grant execute on all security-definer functions to the database owner role
**Security matrix**

<table>
<thead>
<tr>
<th>TRUE</th>
<th>FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>- db_owner creates accounts</td>
<td>- All schemas are created and owned by db_owner.</td>
</tr>
<tr>
<td>- account can create schemas</td>
<td>- Users are created/assigned roles by db_owner</td>
</tr>
<tr>
<td>- schemas are owned by account_owner</td>
<td>- Each schema has its own owner.</td>
</tr>
<tr>
<td>- Users are created/assigned roles by account_owner</td>
<td>- Users are created/assigned roles by db_owner</td>
</tr>
<tr>
<td>- Accounts are isolated</td>
<td>- Accounts are isolated</td>
</tr>
</tbody>
</table>
Security matrix

**FALSE**
- All schemas are created and owned by db_owner.
- Users are created/assigned roles by db_owner.
- Each schema has its own owner.
- Users are created/assigned roles by db_owner.
- Accounts are isolated

**TRUE**
- db_owner creates accounts
- account can create schemas
- schemas are owned by account_owner
- Users are created/assigned roles by account_owner
- Accounts are isolated

```
CREATE SCHEMA orders
AUTHORIZATION my_db_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER my_db_owner GRANT SELECT ON TABLES TO orders_read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER my_db_owner GRANT INSERT, UPDATE, DELETE, SELECT ON TABLES TO orders_read_write;

CREATE SCHEMA orders
AUTHORIZATION orders_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER orders_owner GRANT SELECT ON TABLES TO orders_read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER orders_owner GRANT INSERT, UPDATE, DELETE, SELECT ON TABLES TO orders_read_write;
```

```
CREATE SCHEMA orders
AUTHORIZATION my_account_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER my_account_owner GRANT SELECT ON TABLES TO orders_read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER my_account_owner GRANT INSERT, UPDATE, DELETE, SELECT ON TABLES TO orders_read_write;

CREATE SCHEMA orders
AUTHORIZATION orders_owner;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER orders_owner GRANT SELECT ON TABLES TO orders_read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA orders FOR USER orders_owner GRANT INSERT, UPDATE, DELETE, SELECT ON TABLES TO orders_read_write;
```
create_schema_roles

Input parameters:
• schema_name
• app_user_name (opt)
• app_user_password (opt)
• ddl_user_name (opt)
• ddl_user_password (opt)
• account_owner (optional. Default current user)

Actions:
• creates schema (ownership is driven by security settings)
• creates read_write role
• creates read_only role
• creates owner role (if applicable)
• creates/assigns app and owner users
drop_schema_roles

Input parameters:
• schema_name

Actions:
• revokes read_only role from all users
• revokes read_write from all users
• revokes owner role (if applicable)
• drops all associated roles
• drops schema
assign_schema_owner_user

Input parameters:
• schema_name
• ddl_user_name
• ddl_user_password (opt)

Actions:
• creates user ddl_user_name if it does not exist
• changes password if user exists & password provided
• grants schema owner role to ddl_user_name
assign_schema_app_user

Input parameters:
- schema_name
- app_user_name
- app_user_password (opt)

Actions:
- creates user app_user_name if it does not exist
- changes password if user exists & password provided
- grants schema read_write role to app_user_name
assign_schema_ro_user

**Input parameters:**
- schema_name
- ro_user_name
- ro_user_password (opt)

**Actions:**
- creates user ro_user_name if it does not exist
- changes password if user exists & password provided
- grants schema read_only role to ro_user_name
Revoke functions

- revoke_schema_owner_role
- revoke_schema_app_role
- revoke_schema_ro_role
Additional security definer functions

- `select_all_privileges()`: all privileges on the current db
- `blocking_processes()`: blocking query with superuser privileges
- `pg_stat_activity()`: `pg_stat_activity` with superuser privileges
Code details

Event trigger forces new object ownership and permissions to the schema owner

FOR v_obj IN
  SELECT * FROM pg_event_trigger_ddl_commands ()
  order by object_type desc
LOOP
  <fix perm>
END LOOP
Code details

Check whether the current_user has an ownership role for this schema (grant execute is not enough)

```sql
select exists (  
    with recursive x as  
    (  
        select member::regrole,  
            roleid::regrole as role  
        from pg_auth_members as m  
        union all  
        select x.member::regrole,  
            m.roleid::regrole  
        from pg_auth_members as m  
        join x on m.member = x.role  
    )  
    select 1  
    from x  
    where  
        (member::text = current_user  
        and role = (select nspowner::regrole from pg_namespace  
            where nspname=p_schema_name)  
        or current_user=(select  
            (nspowner::regrole)::text from pg_namespace  
            where nspname=p_schema_name)  
    )  
);```
Code details

Checking the execution stack inside security definer function

```sql
if not
    perm_check_stack(
        'dba_tools.perm_drop_schema_roles')
then
    raise exception 'You are not allowed
to drop schema %', p_schema_name;
end if;
```
Future work

- Reporting
- Unit tests
- Conversion automation
Q&A

Hettie Dombrovskaya  
Database Architect  DRW  
hdombrovsk@drwholdings.com  
www.drw.com