Authorization In PostgreSQL

Managing User Privileges In The Database

Charles Clavadetscher

Swiss PostgreSQL Users Group

Punto d'incontro, Zurich, 17.09.2015

Outline

1. Introduction
2. Roles and Groups
3. Privileges on Objects
4. Organize Access Control
5. Conclusion

Authentication and authorization

Authentication is the process used to verify the identity of a user. It is a step that precedes authorization.
Most typical techniques:

- Username and password
- Public Key Infrastructure
- LDAP

Authorization is the process that takes place each time an authenticated user requests resources from a system. It is also known as access control.

- Roles and privileges
- The focus of this presentation

Basic Structure of PostgreSQL

- Server
- Cluster
- Database
- Schema
- Database objects
  - Tables, views and sequences
  - Functions
  - Others: language, type, domain, large object, foreign data wrapper, tablespace[1]

[1] Operations in normal environment usually do not require deeper knowledge of these objects.

Outline

1 Introduction

2 Roles and Groups

3 Privileges on Objects

4 Organize Access Control

5 Conclusion

Roles and Groups - 1

Since PostgreSQL 8.1 the concepts of `user` and `group` were unified into the single object named `role`.
Roles:

- Exist at the level of the cluster
- Can represent a user or a group
- Can have options, configuration parameters and privileges

A superuser or a role with the option `CREATEROLE` are allowed to create new roles and modify existing ones.

```
CREATE ROLE name [[WITH] option [...]]
```

Roles and Groups - 2

The options that we will look at more closely are the following[2] (underlined options are the defaults if not specified):

- [SUPERUSER|NOSUPERUSER]
- [LOGIN|NOLOGIN]
- [CREATEDB|NOCREATEDB]
- [CREATEROLE|NOCREATEROLE]
- [INHERIT|NOINHERIT]
- [PASSWORD '...']

[2] You can see a full list of options using the command `\h CREATE ROLE` in psql or in the PostgreSQL official documentation.

Privileges - 1

- Privileges specify actions allowed on objects
- Privileges can be granted or revoked to roles
- The type of privileges available depends on the object

The general syntax for granting and revoking privileges is as follows:

```
GRANT {ALL|privilege(s)} ON [object type] object(s) TO {role(s)|PUBLIC}

REVOKE {ALL|privilege(s)} ON [object type] object(s) FROM {role(s)|PUBLIC}
```

It is also possible to grant or revoke a role to another role.

```
GRANT role(s) TO role(s)

REVOKE role(s) FROM role(s)
```

Privileges - 2

The key word `PUBLIC` can be used to grant or revoke privileges (not roles) to all existing and future roles.

```
GRANT privilege ON object TO PUBLIC

REVOKE privilege ON object FROM PUBLIC
```

Notice that the owner of an object implicitly has all privileges on the object owned.

Superusers are not subject to the authorization mechanism of the database. Eventually it is the same as if a superuser had all privileges granted to all objects available in the cluster.

ACL Acronyms - 1

Many objects have an ACL associated with it that follows a specific format.

`grantee=acl/grantor`	for privileges granted to a role
`=acl/grantor`	for privileges granted to `PUBLIC`, i.e. to all roles

The meaning of the specific acronyms used to designate privileges are as follow:

```
r  SELECT ("read")
w  UPDATE ("write")
a  INSERT ("append")
d  DELETE
```

ACL Acronyms - 2

D	TRUNCATE
x	REFERENCES
t	TRIGGER
X	EXECUTE
U	USAGE
C	CREATE
c	CONNECT
T	TEMPORARY

Examples

```
admin=arwdDxt/admin
andrew=rw/postgres
=UC/postgres
```

Default Privileges

PostgreSQL sets privileges on some objects to `PUBLIC` by default. You can find them in the official documentation in the notes on the SQL statement `GRANT`.

Here for your convenience the list of default privileges:

- `CONNECT` and `TEMPORARY` on databases.
- `EXECUTE` on functions.
- `USAGE` on languages.

Besides, in a newly created database `PUBLIC` will have `USAGE` and `CREATE` privileges on the public schema.

Outline

1 Introduction

2 Roles and Groups

3 Privileges on Objects

4 Organize Access Control

5 Conclusion

Databases

Privileges that you can grant on a database:

- `CONNECT`
- `[TEMP|TEMPORARY]`
- `CREATE`

Schemas

There are two privileges that can be granted or revoked on a schema.

`CREATE` allows to create objects like tables, views, etc. in the schema.

`USAGE` allows using objects available in the schema. Notice that this alone is not enough to use the objects in the schema. Those have their own privileges that must be granted for specific usage. On the other hand, a role with privileges granted on an object in the schema will not be able to use it unless it has been granted `USAGE` on the containing schema as well.

Schemas - The public schema

- A newly created database always has a schema called public
- The owner by default is postgres, but you can change the ownership, if you have superuser access
- Using the command `\dn+` in psql you can see which schemas are available in the database you are connected to

```
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
```

- The public schema can be useful, but you can also drop it, if you don't have a specific use for it or change its ACL according to your needs

Visibility

In this context visibility refers to the need to qualify the name of the objects in a database with their schema. In general an object is fully qualified in the following form.

```
[database.]schema.object[parameters]
```

Because you must be connected to a database for interacting with PostgreSQL, its name is always implicit. Qualifying objects with their schema is a good habit and will spare you many problems in the future. If you use objects in the database without qualifying them with their schema, PostgreSQL will search for it in the schemas listed in the user's property search_path.

Tables and Views

Tables and views are contained in a schema.

- `SELECT` allows reading rows from a table. It can be granted on all columns or a subset of them.
- `INSERT` is needed to add rows to a table. It can be granted on all column or a subset of them.
- `UPDATE` enables the grantee to modify existing records of a table. Again the grant can extend to all columns or be limited to a number of them.
- `DELETE` allows removing records from a table.

Dependencies

Besides privileges on the container objects, some of the table privileges have dependencies on other objects' privileges at the same level and even between them.

- `INSERT` requires `USAGE` on all sequences used by `DEFAULT` clauses to provide column values in the table.
- `UPDATE` needs to search for the record(s) to modify and requires therefore `SELECT` on the same table.
- `DELETE` is similar to update and also requires `SELECT` on the same table.
- All privileges granted on objects in a schema require the role to also have `USAGE` on the containing schema.

Columns

`SELECT`, `INSERT`, `UPDATE` AND `REFERENCES` can be granted to a subset of the table's columns. In order to achieve this target you cannot revoke privileges on single columns after you have granted them to the whole table. Instead you must revoke the privilege on the table and then grant it to the columns you want to allow the access to.

```
admin=> GRANT INSERT ON test.test TO user1;
admin=> REVOKE INSERT (id) ON test.test FROM user1;
admin=> \dp
 Schema | Name | Type  |    Access privileges    | Column access privileges
--------+------+-------+-------------------------+-------------------------
 test   | test | table | charles=arwdDxt/charles+|
        |      |       | user1=a/charles         |

admin=> REVOKE INSERT ON test.test FROM user1;
admin=> GRANT INSERT (id) ON test.test TO user1;
admin=> \dp
 Schema | Name | Type  |    Access privileges    | Column access privileges
--------+------+-------+-------------------------+-------------------------
 test   | test | table | charles=arwdDxt/charles | id:                    +
        |      |       |                         |   user1=a/charles
```

Outline

1 Introduction

2 Roles and Groups

3 Privileges on Objects

4 Organize Access Control

5 Conclusion

Groups and Inheritance

In PostgreSQL users and groups are the same object `role`. A group is a role that does not have the `LOGIN` attribute set. The basic idea behind using groups is that instead of granting or revoking privileges to every single user, you customize those privileges for a group or a small set of group, depending on your requirements and then grant the group (i.e. the role) to all users that share the same privileges.

What makes this possible is the inheritance of the privileges from the granted role to the grantee. The grantee receives through the membership in a role, all its privileges as if these were granted directly to him.

Careful

This flexible approach requires some caution when implemented.

- The granted role inherits the privileges directly if the grantee role has the `INHERIT` attribute set (which is the default), but can inherit also the attributes of the granted role if the grantee issues a `SET ROLE rolename` command.
- As an example: If the granted role has the attribute `CREATEROLE`, the grantee can impersonate the granted role using `SET ROLE` and manipulate database users.
- A user can grant himself to another user, thus granting access to all objects he has privileges on. If the granted role is the owner of objects, this extends to their destruction.

Evil on Earth

Never do that on your systems!

```
admin=> CREATE ROLE evil LOGIN PASSWORD 'xxx';
admin=> \du
 Role name |                   Attributes                   | Member of
-----------+------------------------------------------------+-----------
 admin     | Create role, Create DB                         | {}
 evil      |                                                | {}

admin=> CREATE DATABASE test;
CREATE DATABASE

admin=> \c uci evil
FATAL:  permission denied for database "uci"
admin=> GRANT admin TO evil;
admin=> \c uci evil
You are now connected to database "uci" as user "evil".
evil=> SET ROLE admin;
evil=> DROP DATABASE test;
DROP DATABASE
evil=> ALTER ROLE admin NOLOGIN;
ALTER ROLE
evil=> \c - admin
FATAL:  role "admin" is not permitted to log in
evil=> GRANT charles TO evil;
ERROR:  must be superuser to alter superusers
```

What can you do?

- First of all make sure that you have a working policy for backup and restore of your database.
- Never grant superuser roles or roles with special attributes or object owners to others, unless you have a very good reason for it.
- If you cannot trust your users not to grant themselves to other, you may do the following.

1. Create all users with `NOINHERIT`. Let groups inherit privileges instead.
2. Don't grant any privileges directly to users, only through groups.
3. Don't grant the `CONNECT` privilege on databases to the groups, only to individual users.

Row Level Security

- RLS is a new feature introduced with PostgreSQL version 9.5.
- The documentation is available in the development section: http://www.postgresql.org/docs/devel/static/ddl-rowsecurity.html
- In order to use RLS you need to perform two steps.

1. Create one or more policies for a table.
2. Enable row level security on that table using `ALTER TABLE`.

Policies - 1

- A policy is created using the new command `CREATE POLICY`.

```
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

- A policy is per table an must have a unique name among that table policies. Different tables may have policies with the same name.
- `FOR` indicates the commands subject to the policy. The default is `ALL`.
- `TO` specifies for which role(s) the policy must be applied to. If omitted the default is `PUBLIC`.

Policies - 2

- `USING` is used to specify a condition to apply for filtering rows from the table while applying the command mentioned in `FOR`. The condition or set of conditions must return a boolean. If policies are defined for individual commands, this condition can be specified for all commands but `INSERT`.
- `WITH CHECK` is used when an `INSERT` or an `UPDATE` tries to create a new row. Therefore this clause can only be applied to those commands.
- When using `FOR ALL` the condition is applied to all commands, which can be handy at times, but also could lead to unwanted results.

Row Level Security - 4 - Policies

- An example.

```
CREATE POLICY ON operations.catalogue
FOR ALL
TO uci_users
USING (name = SESSION_USER)
WITH CHECK (name = SESSION_USER);

ALTER TABLE operations.catalogue ENABLE ROW LEVEL SECURITY.
```

- The policies can be seen using `\dp`.

```
    Name   |                        Policies
-----------+-----------------------------------------------------
 catalogue | uci_users_policy:                                  +
           |    (u): (responsible = ("session_user"())::text)+
           |    (c): (responsible = ("session_user"())::text)+
           |   to: uci_users
```

Outline

1 Introduction

2 Roles and Groups

3 Privileges on Objects

4 Organize Access Control

5 Conclusion

Resources

- Sources & more: http://www.swisspug.org/Agenda
- Online documentation: http://www.postgresql.org/docs/9.4/interactive/index.html

Contact

- SwissPUG: clavadetscher@swisspug.org
  http://www.swisspug.org
- Work: clavadetscher@kof.ethz.ch
  http://www.kof.ethz.ch
- Private: charles@schmiedewerkstatt.ch
  http://www.schmiedewerkstatt.ch

Thank you

Thank you very much for your attention!

Feedback

Q&A