# Authorization In PostgreSQL

## Managing User Privileges In The Database

Charles Clavadetscher

KOF, ETH Zurich

VII PgCuba, La Habana, Cuba, October 2015

## Outline

## Charles Clavadetscher

- Senior DB Engineer at KOF ETH Zurich
  - Migration and reengineering of legacy databases to PostgreSQL
  - Maintenance of all databases at KOF: PostgreSQL, Oracle, MySQL and MSSQL Server
  - Support in business process reengineering
- Co-founder and treasurer of the SwissPUG, the Swiss PostgreSQL Users Group
- Member of the Swiss PostreSQL Conference organization committee

## KOF ETH

- KOF is the Center of Economic Research of the
- ETHZ the Swiss Institute of Technology in Zurich, Switzerland

## 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

## 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

## Outline

## 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.

## 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.

## Roles and Groups - 3

Examples

Create a superuser:

```
CREATE ROLE charles SUPERUSER LOGIN PASSWORD '...';
```

Create a user that can create databases and roles, but is not a superuser:

```
CREATE ROLE boss CREATEDB CREATEROLE LOGIN PASSWORD '...';
```

Create a user that can login using the login mechanisms of the database:

```
CREATE ROLE andrew LOGIN PASSWORD '...';
```

Create a user that can login using database external mechanisms (e.g. LDAP)

```
CREATE ROLE steve LOGIN;
```

Create a group:

```
CREATE ROLE sales NOLOGIN;
```

## 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.

## Privileges - 3 - ACL Acronyms

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
```

## Privileges - 4 - ACL Acronyms (Continued)

| | |
|---|---|
| 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
```

## Databases - 1

- Databases are objects contained in a cluster
- A connection to PostgreSQL is always established to a database
- Databases contain schemas
- Databases can be created by superusers or users with the option `CREATEDB`

```
CREATE DATABASE dbname [[WITH] options]
```
[3]

The owner of the database is its creator or the role set in the option `OWNER` at creation time or later on using

```
ALTER DATABASE dbname OWNER rolename
```

[3]The command `\h CREATE DATABASE` displays all options

## Databases - 2

For the access control to the database the only option that may be of interest is `OWNER`. Notice that in order to give ownership to another user, the user creating the database must be a superuser or a user that is a member of the role it is giving ownership to.

Privileges that you can grant on a database:

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

## Databases - 3

Examples

Create a database with the current user as owner

```
CREATE DATABASE accounting;
```

Create a database with a different owner

```
CREATE DATABASE sales OWNER sales_manager;
```

Grant access to a database

```
GRANT CONNECT ON DATABASE accounting TO andrew;
```

Revoke access to a database

```
REVOKE CONNECT ON DATABASE accounting FROM andrew;
```

## Outline

## Schemas - 1

A schema can be considered as a section of a database with a unique name within that database. The official documentation of PostgreSQL lists some of the most typical use case scenarios for schemas:

- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Althought these statements point out the separation, schemas can be very useful to store shared objects between users and applications.

## Schemas - 2 - The public schema

- A newly created database always has a schema called public[4]
- 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

[4] As long as no changes have been made to the template1 database

## Schemas - 3

Users having the `CREATE` privilege on the database (along with superusers and the owner of the database) are allowed to create schemas in it[5].

```
CREATE SCHEMA schema_name [ AUTHORIZATION user_name ]
```

The clause `AUTHORIZATION` allows the creator to specify a different owner if he is a superuser or a member of the grantee role.

Example

```
admin@uci.localhost=> CREATE SCHEMA admin;
CREATE SCHEMA
admin@uci.localhost=> \dn+ admin
                 List of schemas
 Name  | Owner | Access privileges | Description
-------+-------+-------------------+-------------
 admin | admin |                   |
```

[5] It is also possible to create, with a single statement, objects in the schema. For details look at the official documentation.

## Schemas - 4 - Privileges

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 - 5 - 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.

## Schemas - 6 - search_path

- The content of search_path tells PostgreSQL in which schemas it must look for objects
- Objects created without schema qualification will be created in the first available schema on the list, provided the role has `CREATE` privilege on the schema
- The usage of search_path is convenience. Get used to always qualify the object with its schema in your code

### See your current search_path

```
admin@uci.localhost=> SHOW search_path;
   search_path
-----------------
 "$user", public
```

## Schemas - 7 - search_path

Modify your search_path in the current session.

```
SET search_path=public;
```

Reset your search_path to the value configured in your role.

```
RESET search_path;
```

Modify search_path permanently. Note each user can perform this action on his own search_path.

```
ALTER ROLE rolename [IN DATABASE database_name] SET search_path=public;
```

See search_path of another user.

```
SELECT rolname, rolconfig FROM pg_roles WHERE rolname = 'user1';
 rolname |                rolconfig
---------+-----------------------------------
 user1   | {"search_path=\"$user\", public"}
```

## Outline

## Tables, Views and Sequences

- Tables are the data containers within a database. This is probably the best known object within a database.
- From a user perpective, views are like tables. In fact they are statements that put together columns of one or more tables, along with additional values that may be computed from them. They mostly are used to hide complexity and to customize data selection to users' needs. In PostgreSQL views can be updatable if the underlying structures are trivial.
- Secuences are related to tables inasmuch as they are used to generate values that are unique across transactions in the context of the database. Thanks to this characteristic sequences may supply values for the primary key of a table.

## Tables and Views privileges - 1 - Basics

Tables and views are contained in a schema. Notice that privileges granted, only can be consumed if the grantee also has been granted `USAGE` privilege on the containing schema. Obviously a role should also have `CONNECT` privilege to the database. If granted to views the following privileges apply on the underlying table.

- `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.

## Tables and Views privileges - 2 - 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.

## Tables and Views privileges - 3 - Dependencies

Notice that error messages can be quite confusing. Consider the following example.

```
admin=> CREATE SCHEMA test CREATE TABLE test (id INTEGER);
admin=> GRANT INSERT, UPDATE ON test.test TO user1;

user1=> INSERT INTO test VALUES (9);
ERROR:  relation "test" does not exist

user1=> INSERT INTO test.test VALUES (9);
ERROR:  permission denied for schema test

admin=> GRANT USAGE ON SCHEMA test TO user1;

user1=> INSERT INTO test.test VALUES (9);
INSERT 0 1
user1=> UPDATE test.test SET id = 10 WHERE id = 9;
ERROR:  permission denied for relation test

user1=> SELECT has_table_privilege('test.test','update');
 has_table_privilege
---------------------
 t
user1=> SELECT has_table_privilege('test.test','select');
 has_table_privilege
---------------------
 f
```

## Tables and Views privileges - 4

Tables know some additional privileges.

- `TRUNCATE` allows emptying the whole table with the SQL statement of the same name.
- `REFERENCES` allows the grantee to create a foreign key in the table using the column(s) he's been granted the privilege. The user also must have the same privilege on the column(s) of the referenced table.
- `TRIGGER` is needed in order to create a trigger on the table. Notice that this has nothing to do with the privilege of writing or using a trigger function.

## Tables and Views privileges - 5 - 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
```

## Tables and Views privileges - 6

As shown in the previous slide the psql command `\dp` can be used to diplay the ACL of the objects in the schemas. There are some alternatives.

- The psql command `\z` is equivalent to `\dp`
- For single objects and privileges you can use the access privilege inquiry functions.

```
has_<object_type>_privilege([rolename,] object_name, [column_name,] privilege)
```

We have used one of these functions to investigate the privileges of a table and the error messages issued by PostgreSQL.

## Outline

## Functions - 1

- Functions extend the capability of the database.
  - Encapsulate complex operations difficult to achieve with plain SQL.
  - Reduce the amount of network traffic.
  - Automate tasks, e.g. with triggers.
- They run on the server and are called with a `SELECT` statement.
- Functions only know the privilege `EXECUTE`.
- `PUBLIC` is granted `EXECUTE` on functions at creation time by default, but. . .
- The user must have `USAGE` privilege on the containing schema and all necessary privileges on the objects manipulated in the function.

## Functions - 2 - View Privileges

In order to see which privileges are set on a function you have two choices.

- Use the access privilege inquiry function for functions if you need to know the privileges of a specific role. You may also use `public` to check if a function can be executed by every role.

```
has_function_privilege(rolename, function_name, privilege)
```

- You can also query the PostgreSQL catalog or the information_schema.

```
SELECT proname, proacl FROM pg_proc WHERE proname = '<function_name>';

SELECT grantor, grantee, routine_name, privilege_type
FROM information_schema.routine_privileges
WHERE routine_name = '<function_name>';
```

## Other Objects - 1

- `LANGUAGE`: Privilege `USAGE` allows the grantee to write functions in that language. Untrusted languages can only be used by superuser. The privilege is granted to `PUBLIC` by default.
- `TYPE`: `USAGE` is granted by default to `PUBLIC` and allows grantees to use the type in the creation of objects such as tables and functions.
- `DOMAIN`: Is an existing `TYPE` with some constraints. Grantees can receive the `USAGE` privilege to use the `DOMAIN`.
- `LARGE OBJECT`
  - `SELECT` read the large object.
  - `UPDATE` modify, i.e. write to the large object.

## Other Objects - 2

- `FOREIGN DATA WRAPPER`: Privilege `USAGE` allows the grantee to create a link to an external database Server using the data wrapper.
- `TABLESPACE`: Only knows the privilege `CREATE` to allow the creation of databases by default in the given `TABLESPACE` (eventually specific space on disk).

The objects that we have seen in this last section and their privileges are usually not relevant for normal daily operations. You can find more details on their usage and characteristics in the official PostgreSQL documentation.

## Outline

## Default Privileges - 1

PostgreSQL set default 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.

## Default Privileges - Modify

Due to security requirements you may want to modify one or more of the default privileges, e.g. allowing only certain roles to connect to a database or to execute some functions.

It is also possible to change the default privileges or create your own ones. For that purpose you can use the PostgreSQL statement that follows in a simplified form (the complete command can be seen in the official documentation).

```
ALTER DEFAULT PRIVILEGES FOR ROLE rolename [IN SCHEMA schema_name]
{GRANT|REVOKE} privilege(s) ON {TABLES|SEQUENCES|FUNCTIONS|TYPES}
{TO|FROM} rolename(s)
```

If you want to “delete” a default privileges setting you must use the same statement above using exactly the opposite values for `{GRANT|REVOKE}` and `{TO|FROM}`.
Notice that the scope of default privileges is the current database. The settings will not have any impact on other databases.

## Default Privileges - 3 - View

You can view the existing default privileges using the psql command `\ddp`. If the list is empty then the default privileges mentioned in the first slide of this section apply.

Examples

```
admin=> ALTER DEFAULT PRIVILEGES FOR ROLE admin GRANT select ON TABLES TO user1;
admin=> \ddp
          Default access privileges
 Owner | Schema | Type  |  Access privileges
-------+--------+-------+--------------------
 admin |        | table | admin=arwdDxt/admin+
       |        |       | user1=r/admin
```

This is equivant to say: Each time that admin creates a table in any schema in this database, grant read access to user1. You can remove the default privilege as follow.

```
admin=> ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE select ON TABLES FROM user1;
admin=> \ddp
         Default access privileges
 Owner | Schema | Type | Access privileges
-------+--------+------+------------------
(0 rows)
```

## Outline

## Introduction

We have seen how to grant and revoke privileges to roles. Depending on the complexity of the database design and on the numbers of users in your cluster, this could become difficult to manage.

In this section we will see some techniques that can be used in order to simplify to some extent privileges management.

In particular how to use roles to centralize privileges to a few manageable units, how to restrict access to tables using views and how to grant access indirectly through functions.

## Groups and Inheritance - 1

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.

## Groups and Inheritance - 2

Example:

```
CREATE ROLE accountants NOLOGIN;
GRANT SELECT, INSERT, UPDATE, DELETE ON accounts TO accountants;

GRANT accountants TO jonathan;
```

- We create a group named accountants and grant it privileges on table accounts.
- Then we grant the group accountants to a user jonathan thus enabling him to manipulate data of the accounts table.
- The big advantage of this approach becomes visible if you have more than one user that needs this set of privileges.
  - If privileges for all accountants change over time you only need to change the privileges of the group.
  - If an accountant leaves the company or a new one is hired you only need to revoke the group membership from the leaving one and grant it to the new one.

## Groups and Inheritance - 3 - 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.

## Groups and Inheritance - 4 - 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
```

## Groups and Inheritance - 5 - 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.

## Groups and Inheritance - 6 - Group Members

You can find which users are in a group querying the catalog. For convenience we restrict the query to those users that have directly or indirectly `CONNECT` privilege on the database uci. You may modify the query to search for all members of a group or all groups a user belogs to. However, in this form, you don't see indirections through group levels.

```
SELECT rg.rolname AS group_name,
       rr.rolname AS role_name,
       gr.rolname AS grantor
FROM pg_auth_members m
JOIN pg_roles rg ON (rg.oid = m.roleid)
JOIN pg_roles rr ON (rr.oid = m.member)
JOIN pg_roles gr ON (gr.oid = m.grantor)
WHERE has_database_privilege(rr.rolname,'uci','connect')
ORDER BY rg.rolname, rr.rolname;
```

You can also get a similar information using the psql command `\du`.

## Groups and Inheritance - 7 - Wrap up

Working with non inheriting users will add on your application developers an additional layer of complexity. They will have to perform a `SET ROLE` to the group in the session after user login. You can also see this as an asset and configure individual logins for users and group privileges for applications. This will be reflected in the variables `SESSION_USER` (the real user) and `CURRENT_USER` (the impersonated user).

More important than this, make sure that you can trust your users. If this is not the case, then you probably have a bigger problem anyway.

## Views and Security Definer Functions

So far we have seen the basics of the authorization system of PostgreSQL. We have seen how to grant and revoke privileges to objects, which privileges can be granted to specific objects and how a privilege may require granting others in order to be consumed.

Now we will show two practices that can be used to solve other problems:

- How can you extend the control over a table at the row level?
- How can you allow users to make only specific changes to a table without granting direct access to it?

## Views - 1

- Views are a way to manipulate what users can see. Besides defining which columns the view returns, you can also select a subset of rows based on a `WHERE` clause in the statement definition.
- Views that don’t add columns to a table are writable. Since `UPDATE` and `DELETE` require the `SELECT` privilege, users end up being able to modify only rows that they can see.
- Since version 9.4 views can have a `CHECK OPTION` that prevent users from changing fields that would make the row invisible to them. This can happen if a column that a user can modify is in any condition of the `WHERE` clause.

## Views - 2

- Now, if you create a view on a table that should deliver different result sets to different users, you may apply the authorization rules on the view instead of on the table.
- Privileges granted on a view extend to the underlying table, but only through the view and not directly.

## Security Definer Functions - 1

In some occasions a DBA must grant privileges that create unexpected threats. In a previous exercise we created a table for logging user actions in a table and filled the table using a trigger and a trigger function.

In order to make it work we had to grant `INSERT` privilege to the users on the log table. This also allows users to insert records directly into the log table, which is probably not what was intended.

A way to avoid this situation is using security definer functions.

## Security Definer Functions - 2

- A security definer function can be implemented exactly in the same way as any other function. The difference is that you must declare it as such with the keyword `SECURITY DEFINER`.

```
CREATE OR REPLACE FUNCTION operations.log_changes()
RETURNS TRIGGER
SECURITY DEFINER
AS $$
[...]
```

- A security definer function is executed with the privileges of the user that created it[6].
- This gives you flexibility, but as usual you must take some additional security measures.

[6] This is comparable to the SUID bit on X-like systems.

## Security Definer Functions - 3

The PostgreSQL official documentation gives some important points to consider when writing security definer functions.

- Avoid having writeable schemas by anyone in the search_path, in particular pg_temp (implicit at the start of search_path). Set the schema at the end of search_path in the function declaration.

```
[...]
$$ LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = admin, pg_temp;
```

- Revoke `EXECUTE` from `PUBLIC` and grant it to your users explicitly.
- Create the function and set the privileges in a single transaction.

```
BEGIN;
CREATE FUNCTION ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION ... FROM PUBLIC;
GRANT EXECUTE ON FUNCTION ... TO ...;
COMMIT;
```

## Security Definer Functions - 4

It is now clear that if a user can execute a function with the privileges of the creator and assuming that the creator has the necessary privileges to perform the actions listed in the function, that user will be able to perform actions for which he may not have the necessary privileges directly.

In our log table example that means recreating the trigger function as security definer, granting `EXECUTE` on it to legitimate users and revoke the `INSERT` privilege from everybody on the log table.

## Row Level Security - 1

- 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`.

## Row Level Security - 2 - Policies

- 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`.

## Row Level Security - 3 - Policies

- `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

- The use case implemented in the previous section using a view, could be done with a policy as follows.

```
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

## Resources

- Sources & more: http://www.schmiedewerkstatt.ch/uci
- 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