PostgreSQL: Security

Hans-Jürgen Schönig

Introduction

PostgreSQL security

- PostgreSQL allows to configure various layers of security
- The permission system has been improved over the years

Network configuration

- Adjust listen_addresses in postgresql.conf to turn the network on / off
- Add your OWN IP addresses
- listen_adresses defines, which of your interfaces PostgreSQL will consider
- A restart is required

Adjusting pg_hba.conf

- pg_hba.conf defines the authentication method required
- Not every IP range might have the same security requirements
- Many authentication methods available
  - trust, reject, md5, password, gss, sspi, ident, peer, pam, ldap, radius, cert
- The first rule that matches will decide on the authentication method

pg_hba.conf: An example

```
host   all   all    192.168.1.0/24          md5
host   all   all    192.168.1.43/32         reject
```

- In this case 192.168.1.43 will be allowed in with a password

Instance level

Users and roles

- PostgreSQL used to have “user”, “group”, “world”.
- Some years ago a role-based system has been introduced.
- Users, groups, and roles are more or less the same
- NOTE: Users live on the instance and not on the database level

LOGIN vs. NOLOGIN

- To log into the instance LOGIN permissions are needed.
- NOLOGIN roles are utilized to inherit permissions.
- Example:

```
CREATE ROLE warehouse NOLOGIN;
CREATE ROLE paul LOGIN;
GRANT warehouse TO paul;
```

Inheriting permissions

- In this example “paul” can do everything “warehouse” can do
- “paul” is allowed to log into the instance
- Users cannot log in as “warehouse”

Superusers

- During the setup process a superuser is created.
- The name of the superuser is not necessarily “postgres”.
- During initdb the UNIX user is cloned and used as name for the superuser
- However, it is a good idea to have a superuser called “postgres” (many people will rely on that)
- The SUPERUSER flag is boolean:
  - There are no “almost” superusers.
  - Simple permissions cannot be revoked from a superuser.

Database and schema

Database permissions

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP }
    [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

- CREATE: Allows the creation of schemas
- CONNECT: Allows to establish connections

Schema permissions

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

- CREATE: Allows the creation of objects inside a schema
- USAGE: Allows to use objects inside a schema.

Table and column permissions

Table permissions (1)

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
    REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

Table permissions (2)

- SELECT: Allows users to read
- INSERT: Allows insertions (does not imply SELECT)
- UPDATE: Allows updating
- DELETE: Allows the deletion of rows
- TRUNCATE: A separate permission is available (because of the table lock needed)
- REFERENCE: Needed to create a foreign key constraint
- TRIGGER: Allows the creation of a trigger

Displaying permissions (1)

- \dp displays permissions in psql

```
rolename=xxxx -- privileges granted to a role
        =xxxx -- privileges granted to PUBLIC
            r -- SELECT ("read")
            w -- UPDATE ("write")
            a -- INSERT ("append")
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
```

Displaying permissions (2)

```
                  X -- EXECUTE
                  U -- USAGE
                  C -- CREATE
                  c -- CONNECT
                  T -- TEMPORARY
            arwdDxt -- ALL PRIVILEGES (for tables,
               varies for other objects)
                  * -- grant option for preceding privilege

              /yyyy -- role that granted this privilege
```

Making it work

- To use permissions successfully, run the following commands:

```
REVOKE ALL ON SCHEMA public FROM public;
REVOKE ALL ON DATABASE test FROM public;
```

- Otherwise everybody can connect and everybody can create objects inside the public schema.

Additional levels of security

Security barrier views

- PostgreSQL is allowed to reorder restrictions during executions
- If views are used to manage permissions, this is not always possible
- security_barrier can help to avoid security leaks

The core problem (1)

```
CREATE TABLE person (id int, gender boolean);
CREATE VIEW girls AS SELECT *
    FROM     person
    WHERE    gender = 'f';
SELECT * FROM girls WHERE func(id) = 10;
```

The core problem (2)

- func(id) = 10 is the better filter than gender = ‘f’
- PostgreSQL will use the more selective filter first
- What if the procedure yields a debug message containing data?
- If func(id) = 10 is called for a man, this returns secret data
- This will fix the leak:

```
CREATE VIEW girls WITH (security_barrier = true)
    AS SELECT ...
```

Future: Row Level Security

- In 9.5 PostgreSQL will support RLS (Row Level Security)
- It allows to hide rows from a user

Practice

- Slides: https://goo.gl/xdizp9
- Create 2 users that are able to log in (u1, u2).
- Revoke permissions from PUBLIC on database and schema "public".
- Create schema s1 and allow access with admin option to u1.
- Log in as u1 and create table s1.t1.
- Log in as u2 and verify that can't read s1.t1
- Grant access to u2 on s1 and s1.t1, check that it works.