

PostgreSQL Access Controls (AuthN, AuthZ, Perms)

Presented to: PgCon 2010
Date: May 19th, 2010

Introduction

- Stephen Frost
 - System Architect/Designer
 - DBA, Unix Administrator
 - PostgreSQL/PostGIS Hacker
 - Added Roles in 8.1, Column-level Privs in 8.4
- Noblis, Inc.
 - Nonprofit science, technology and strategy organization
 - <http://www.noblis.org>

Agenda

- Server Configuration / Where is everything?!
- Authentication – pg_hba.conf
- Roles
- Authorization – GRANT system
- SE-PostgreSQL – Quick look (time permitting)

- Notes:
 - Newer options (8.3, 8.4, 9.0) will generally be marked using '(X.Y)' notation
 - 9.0 has not been released yet- anything marked (9.0) could be changed by the time it's out!

Server Configuration

- Server config files are typically in the 'data' directory
 - Whatever you used when you did 'initdb -D <dir>'
- May be under /etc on some systems (eg: /etc/postgresql on Debian/Ubuntu)
- PG Config files:
 - postgresql.conf – Main server config file
 - pg_hba.conf – PG host-based authentication file
 - pg_ident.conf – PG User-Mapping file
- Use 'show' to find them from in PG!
 - show data_directory; show config_file;

Server Configuration - postgresql.conf

- authentication_timeout
 - Max time to complete authentication
 - Default 1 minute
 - Server disconnects if time is exceeded
- ssl – Enables SSL connections, default off
- ssl_ciphers – Lists allowed ciphers for SSL
- password_encryption
 - Default password storage (md5'd or not)
 - Default on
 - 'ALTER USER WITH ENCRYPTED PASSWORD'

Server Configuration – File Updates

- postgresql.conf
 - Will be re-read on a 'reload', but
 - Only some parameters can be changed with 'reload'
 - Certain parameters require a full 'restart'
- pg_hba.conf and pg_ident.conf
 - Will be re-read and all entries updated on 'reload'
 - Existing connections will not be affected
 - If the reload isn't successful, will continue using the old file data!
- Monitor your logs!

Authentication - Sub-Agenda

- Authentication process
- Basics of pg_hba.conf
- Connection Types
- Options for Database
- Options for User / Role
- Auth Methods
- Authentication Map File
- Kerberos Configuration/Integration
- Server Configuration

Authentication Process

- Client connects to database server
- Client provides
 - User/Role connecting as (role must have LOGIN priv)
 - PG database name ('postgres' is default)
- Server looks in pg_hba.conf for:
 - Connection type
 - PG database name
 - User/Role name
 - IP address (if applicable)
 - First matching record found is used (no fall-thru)
 - If no record is found (or 'reject' record)- access denied

Auth Process (cont'd)

- Once an auth method is found, PG checks:
 - Client credentials, eg:
 - Asks for password (password or md5)
 - Looks up unix user connecting (ident)
 - Verifies certificate (SSL) or ticket (GSS/SSPI)
 - User/Role access:
 - LOGIN privilege for User/Role
 - CONNECT privileges for User/Role for Database
- Each back-end process connects to only one database and serves only one connection at a time

pg_hba.conf Basics

- Set of records which consist of:
 - Connection type
 - Database name
 - Or: sameuser, samerole, all
 - User / Role
 - IP range (for host, hostssl, hostnossl records), in either format:
 - Specified in CIDR notation (200.2.2.0/24)
 - Specified in Netmask (200.2.2.0 255.255.255.0)
 - OR: samehost (match server's IP)/samenet (server's net) (9.0)
 - Auth Method
 - Auth options (if any)
- Eg (DON'T USE THIS- will tell you why):

```
host    mydb    myuser    0.0.0.0/0    trust
```

Connection Types

- local
 - Matches Unix-domain socket connections
 - Without a 'local' entry, Unix socket connections will be denied
 - Usually used with 'ident' auth method
- host
 - Matches all TCP/IP connections (ssl or nonssl)
 - Only if IP-range supplied matches
- hostssl
 - Matches TCP/IP SSL connections
- hostnossl
 - Matches TCP/IP non-SSL connections
 - Could be used with 'reject' to prevent non-SSL for some ranges

Options for Database

- Name of Database being connected to
- Special values:
 - all: Matches any database name
 - sameuser: Database name equals user name
 - samerole: Requested user must be a member of role (directly or indirectly) which shares the name of the requested database
 - replication: matches for replication (9.0!)
 - comma-separated-list of database names
 - @filename: file of database names

Options for User / Role

- User / Role connecting as
- Special values:
 - all: Matches any user name
 - +role: Match if user name is member of role, either directly or indirectly
 - comma-separated-list of roles
 - @filename: file of role names

Auth Methods - Simple

- trust: Allow connection with no other checks
 - Will allow user to connect as any user!
- reject: Reject connection
- md5: Require client to provide md5-hashed password
 - Prevents sniffer from seeing password on network
- password: Require client to provide cleartext password
 - Strongly discouraged! Do not use!
- ident: Compare OS user name of the client (obtained from OS for Unix Sockets, or through ident server for tcp/ip) to those allowed to connect as user requested through map file
 - Only secure for Unix-socket connections
 - auth-option: map=<file>

Auth Methods - Kerberos-based

- gss, krb5 (deprecated)
 - With Kerberos authentication (RFC2743, 1964)
 - Principal format: `servicename/hostname@realm`
 - auth-options:
 - map: Map between principal or user and database user
 - include_realm: Send entire principal to mapping system
 - krb_realm: Requires principal to be in realm specified
- sspi
 - Windows-based; Uses Negotiate with Kerberos & NTLM
 - Uses Kerberos if possible, falls back to NTLM otherwise
 - auth-options: same as gss

Auth Methods – LDAP (Changed in 8.4)

- Attempts to validate against LDAP directory (or Active Directory)
- PG Server binds to LDAP directory to check user/pass
 - Constructs DN as prefix username suffix
 - Binds with user-provided password
- auth-options
 - ldapserver: LDAP server to connect to
 - ldaprefix: Prepend to username to form DN
 - ladsuffix: Postfix to username to form DN
 - ldapport: Connect to LDAP server on this port
 - ldaptls: Use TLS to connect to LDAP server
- Eg:
`ldapserver=ldap.example.net prefix="cn=" suffix="dc=example, dc=net"`

Auth Method – cert (8.4), pam

- cert
 - Uses SSL certificates for authentication
 - SSL certificate must be valid
 - cn attribute of certificate compared to user name
 - auth-options:
 - map: Map between cn attribute and database name
- pam
 - Authenticates using Pluggable Auth Modules
 - If PAM needs /etc/shadow or another protected file authenticating may fail since PG runs non-root
 - auth-options:
 - pamservice: PAM service name

Auth Method – radius (9.0!)

- radius
 - Uses an external RADIUS server for authentication
 - No redundancy support yet (maybe in 9.1..)
 - Recommend building PG with OpenSSL if using RADIUS

The encryption vector used will only be cryptographically strong if PostgreSQL is built with support for OpenSSL. In other cases, the transmission to the RADIUS server should only be considered obfuscated, not secured, and external security measures should be applied if necessary.
 - auth-options:
 - radiusserver: Name or IP of RADIUS server to connect to (required)
 - radiussecret: RADIUS shared secret, at least 16 characters (required)
 - radiusport: Port on RADIUS server to use (default: 1812)
 - radiusidentifier: NAS identifier in RADIUS requests (default: postgresql)

Authentication – Map file (Changed in 8.4)

- Allows “name” presented by authentication system to be mapped to a database user
- “name” can be:
 - Certificate “cn” attribute
 - Kerberos (gss, sspi, krb5) principal
 - LDAP distinguished name
 - Ident/OS username
- Map file layout:
- map-name system-username database-user
- Eg:
mymap myuser dbuser

Authentication – Map file (cont'd)

- map-name: Arbitrary name of the map, can be specified as authentication option
- With map-name, multiple maps can be defined in single file
- system-username:
 - Explicit name
 - /regexp: regular expression used to transform system name to db user
- database-user:
 - Explicit name
 - \1: result of substitution from regular expression
- Eg:
mymap /^(.*)@mydomain\.com\$ \1

Authentication – Kerberos Config

- Using Kerberos (GSS or SSPI-based) can allow secure integration with existing Kerberos or Active Directory installations
- PG service principal: `servicename/hostname@realm`
- hostname is the fully-qualified server domain name
- postgresql.conf Kerberos options
 - `krb_srvname`: Specify servicename; defaults to postgres but Active Directory requires POSTGRES, can be done in `pg_hba.conf` too
 - `krb_server_keyfile`: Specify the path to the keytab for PG
 - `krb_server_hostname`: Override fqdn used
 - `krb_caseins_users`: Treat usernames case-insensitively?
- realm is the preferred realm of the server machine

Authentication – Kerberos (cont'd)

- Client principal must have DB user first or be using a map file
- Realm is not verified!
 - Use maps if cross-realm is used
 - Enable `include_realm`
- `mod_auth_kerb`
 - Can be used to authenticate web users to the database as themselves
 - No common web user account required
 - No authentication required in web code

Authentication – Kerberos - Keytabs

- Generating a keytab using MIT Kerberos:
 - `kadmin: ank – randkey postgres/my.server.com`
 - `kadmin: ktadd -k /etc/postgres/krb5.keytab postgres/my.server.com`
- Generating a keytab using Windows 2008:
 - Use Active Directory create a user account for the PG service, eg: named 'postgres'
 - `ktpass /princ POSTGRES/myserver.mydomain.com@MYDOMAIN.COM /mapuser postgres@mydomain.com /pass mypass /crypto AES256-SHA1 /ptype KRB5_NT_PRINCIPAL /out krb5.keytab`
 - Note that older versions of Windows only support poor encryption types when creating external keytabs or setting up cross-realm trust
- Cross-realm can also be done between Active Dir and MIT Kerberos

Roles - Sub-Agenda

- Role basics
- Inheritance and the role system
- Best practices (and why..)
- Security Definer functions (and other magic)
- Owner rights, and multiple Owners

Roles - Role Basics

- Users are Roles
- Groups are Roles
- Roles can be members of other roles
- To login, you have to have a role of some kind
- By default, a single role is created and it is a super-user
- Default role is the OS user who ran 'initdb' (eg: sfrost)
- What roles do you have?
 - In psql, use “\du” to see the roles (“\dg” is the same thing)
 - You can also “select * from pg_authid;”
 - To see memberships- “select * from pg_auth_members;”
 - To see who you are “select current_user;”
 - More on this later...

Roles - Role Options

- Superuser – Can override all access restrictions
- CreateDB – Can create new databases
- CreateRole – Can create, alter, and drop roles
- Inherit – Do access rights inherit automatically?
- Login – Allowed to log in?
- Connection Limit – Maximum number of concurrent connections
- Password / Encrypted Password – Sets password (md5'd or not)
- Valid Until – Date/Time when role is no longer valid
- Role-specific run-time configuration options:
 - Many things in postgresql.conf can be set per-role!
 - search_path, statement_timeout, date_style, etc
 - Use ALTER ROLE to make these 'permanent' for a role!

Roles – Role Basics (cont'd)

- Roles are created using 'CREATE ROLE' (or 'CREATE USER'- adds LOGIN)
- Role settings can be changed with 'ALTER ROLE'
- Roles can be dropped with 'DROP ROLE'
 - Roles are OIDs and can not be dropped if they are still referenced anywhere, except role memberships which will be automatically cleaned up
 - REASSIGN OWNED and DROP OWNED can handle any situations where the role owns objects
- The GRANT command is used to make roles members of other roles
- Eg:
GRANT mygroup TO myuser;
- By default, myuser will now automatically have all the rights of both myuser and mygroup
- If mygroup is used as a group in pg_hba.conf, myuser will now be included for matching

Roles - Inheritance

- Roles can either automatically inherit rights or not
- If `noinherit` is set, role must use 'set role' to gain rights of another role
- Inherit is the default (similar to old group system)
- Inheritance is a graph; firewalls can be set up
- User 'A' has inherit and is granted 'B', and 'C'
- User 'B' has `noinherit` and is granted 'D'
- User 'A' has rights of 'A', 'B', and 'C' immediately
- User 'A' can 'set role' to 'D'

Roles – Inheritance (cont'd)

- Role trees can have multiple levels
 - GRANT A to B;
 - GRANT B to C;
 - GRANT C to D;
 - D has rights of D, C, B, and A
 - C has rights of C, B, and A
- Loops are not permitted
 - GRANT D to A;
 - ERROR: role “D” is a member of role “A”

Roles – Best Practices - Admins

- Minimize privilege escalation
- Require Admins to explicitly request superuser
- Allow Admins to be in regular “groups”
- Create a noinherit role “admin”
- Grant postgres to admin
- Grant admin to administrator roles
- Requires Admins to “set role postgres” to get superuser privileges
- Minimizes need to log in directly as “postgres”
- pg_dump support for setting role in 8.4

Roles – Best Practices – App Devs

- Database privileges can be used to enforce permissions even for web users
- Option #1:
 - Kerberos-enabled app users
 - Credentials are passed through application code authenticates to DB as user (can be web-based using mod_auth_kerb)
- Option #2:
 - Username/Password users
 - Credentials passed through application code authenticates to DB as user
- Option #3:
 - Application handles authentication
 - Application code uses common account to authenticate to server, but common account has no rights in database except to be able to 'set role' to users
 - Application code uses 'set role' to change to user

Roles – Security Definer Functions

- Numerous objects in PG can change role
- Functions which are set as “security definer” run as the role which owns the function
 - Be very careful with security definer functions owned by postgres!
 - Pay attention to `search_path` and any other user-settable variables in security definer functions
- Views are run as the owner of the view
 - User running query might have rights to a table the view uses, but the view owner might not!

Roles – Multiple Owners

- Some PG commands require user be the owner:
 - Truncate (prior to 8.4)
 - Vacuum
 - Analyze
 - Alter
- Using Roles, an object can have multiple owners!
- Role 'mygroup' owns table 'mytable'
- Roles 'A', 'B', and 'C' have been granted 'mygroup'
- 'A', 'B', and 'C' have owner-level rights on 'mytable'

Roles – Where are they (Changed in 9.0!)?

- Roles are stored in `pg_authid` (not `public`!)
- Memberships are in `pg_auth_members`
- In 8.4 and below:
 - Flat-files used in 8.4 and below, which are created from catalog tables
 - Allows auth system to verify passwords and role memberships without opening the database
 - Updates to certain catalogs (`pg_authid`, `pg_auth_members`, `pg_databases`) result in new flat-files being written (only 8.4 and below)
 - Implemented as triggers on system catalogs
- In 9.0 and going forward:
 - Flat-files no longer used!
 - Adding/removing/dealing with large # of users much faster!

Viewing Roles

- \du in psql lists roles, non-default attributes and memberships (Changed in 8.4, and again in 9.0!)
- pg_roles view also provides list of roles
 - Allowed to public since password is '*'d out

List of Roles		
Role Name	Attributes	Member of
sfrost	Superuser : Create role : Create DB	{}
test1		{}
t2	Cannot login	{t1}

Per-Database User Names

- Temporary measure
- Requires enabling `db_user_namespace` in `postgresql.conf`
- Users are created as “`username@dbname`”
- Username provided by client has `@` and the database name appended to it
- The result is then looked up by the server
- Global users created by “`username`”
- Clients authenticate to global user using “`username@`”
- `md5` cannot be used with this
- Authentication checks always done with server's user name

Authorization - Sub-Agenda

- PG 8.3 vs. 8.4 vs. 9.0
- General GRANT/REVOKE syntax for Privileges
- Database Privileges, Tablespace Privileges
- Schema Privileges, Table Privileges
- Sequence Privileges, Function Privileges
- Language Privileges
- Foreign Data Wrapper / Foreign Data Server Privs
- Column Privileges
- Controlling Access

Privileges Available in 8.3

- Database
 - Create, Connect (default), Temporary (default)
- Tablespace
 - Create
- Schema
 - Create, Usage
- Table
 - Select, Insert, Update, Delete, References, Trigger
- Function – Execute (default)
- Language – Usage (default)
- Sequence – Usage, Select, Update

Privilege Changes in 8.4

- Table
 - Truncate permission added
 - Non-owners can be granted truncate!
 - Not transaction safe, doesn't fire triggers, etc.
- Foreign Data Wrapper Added
 - Usage
- Foreign Data Server Added
 - Usage
- Column Permissions

Privilege Changes in 9.0

- Ability to 'GRANT ... ON ALL <type> IN SCHEMA'
- Type can be:
 - TABLES (also includes views)
 - SEQUENCES
 - FUNCTIONS
- Adds ALTER DEFAULT PRIVILEGES (for new objects)
 - Allows setting default privileges for the above object types per role / per schema
 - Schema not required- then globally affects role default privs
 - Does not affect existing objects
 - Target roles must have 'create' rights on schema
 - View with \ddp in psql

GRANT syntax for Privileges

- All grant syntax for privileges follow the same general structure
- GRANT { { PRIVILEGES } [,...] | ALL [PRIVILEGES] }
 - ON { { OBJECT TYPE } *object* [, ...]
 - | ALL { OBJECT TYPE } IN SCHEMA *schema_name* [, ...] }
 - TO { *rolename* | PUBLIC } [, ...]
 - [WITH GRANT OPTION]
- The default 'object type' is TABLE
- Functions include the argument types in the '*object*'
- PRIVILEGES are only those which are applicable to the object type being modified
- WITH GRANT OPTION is included then the grantee will be able to grant the same privilege to others
- Granting to PUBLIC gives all users the privilege

Revoke Syntax for Privileges

- All grant syntax for privileges follow the same general structure
- REVOKE [GRANT OPTION FOR]
 - { { PRIVILEGE } [, ...] | ALL [PRIVILEGES] }
 - ON { { OBJECT TYPE } *object* [, ...]
 - | ALL { OBJECT TYPE } IN SCHEMA *schema_name* [, ...] }
 - FROM { *rolename* | PUBLIC } [, ...]
 - [CASCADE | STRICT]
- The default 'object type' is TABLE
- Functions include the argument types in the '*object*'
- PRIVILEGES are only those which are applicable to the object type being modified
- GRANT OPTION FOR will only remove the grant option
- CASCADE will revoke any privileges granted by the user through this granted privilege: STRICT will fail the action if such grants exist: default is STRICT

Database Privileges

- CREATE
 - Allows user to create schemas inside the database
- CONNECT
 - Allows user to connect to the database
 - Granted to public by default
- TEMPORARY/TEMP
 - Allows user to create temporary tables while using the database
 - Granted to public by default
- ALL – All of the above

Tablespace Privileges

- CREATE
 - Allows tables, indexes, and temporary files to be created in the tablespace
 - Databases created are also allowed to have tablespace as the default tablespace
 - Revoking this does not move any existing objects even if the user can no longer create new objects in the tablespace
- ALL – All of the above

Schema Privileges

- CREATE
 - Allows any new object to be created in the schema
 - To rename an object, the user must be considered an owner of the object and have create rights on the schema
- USAGE
 - Allows the user to access the objects in the schema
 - Permissions on the object itself will also be checked
 - Revocation may not be immediate (if other backends have existing statements which have already looked up the schema)
 - Does not prevent users from seeing object names/definitions
- ALL – All of the above

Table Privileges

- SELECT
 - User can pull/query all columns from the table (also allows COPY TO)
- INSERT
 - User can add new data to all columns (also allows COPY FROM)
- UPDATE
 - User can change all columns of the table (requires SELECT also generally)
- DELETE
 - Removes full rows from the table (requires SELECT also generally)
- TRUNCATE (8.4)
 - Remove all data very quickly (ON DELETE triggers not called!)
 - Not MVCC safe!
- REFERENCES – Required to create foreign keys referenced by or referencing
- TRIGGER – Required to create triggers on the table

Sequence Privileges

- USAGE
 - Allows the user to use currval() and nextval() functions which return/update the sequence
 - Note that SELECT is not required if user has USAGE
- SELECT
 - Allows the user to use currval() to query the current value of the sequence
- UPDATE
 - Allows the user to use nextval() and setval() to update and forcibly set the sequence
- ALL – All of the above

Function Privileges

- Object Name for Functions
 - Defined by both their name and their argument types
 - Multiple functions with the same name but different argument types can co-exist
- EXECUTE
 - Allows the user to run the function
 - Granted to PUBLIC by default
 - Be cautious to revoke EXECUTE from public in the same transaction as a security definer function is created if it should not be available to all users
- ALL – All of the above

Language Privileges

- USAGE
 - Allows the user to create functions with this language
 - Granted to PUBLIC by default
- Regular users can not create functions using untrusted languages regardless of the privileges granted
- ALL – All of the above

Foreign Data Wrapper / Data Server (8.4)

- Data Wrapper
 - USAGE
 - Allows the user to create new servers using this foreign data wrapper
- Data Server
 - USAGE
 - Allows the user to query the options of the server and associated user mappings.
- ALL – All of the above

Column Privileges (8.4)

- SELECT
 - If less than the full table is granted, “select *” won't be permitted!
- INSERT
 - Using: INSERT table (col1,col2) values (1,2);
 - unreferenced columns will get NULL or default
 - Error for unreferenced columns when no default and not null
- UPDATE
 - Allows UPDATE SET col1 = x;
 - WHERE clause in UPDATE requires SELECT! (Just like Table)
- REFERENCES
 - Permits use as a Foreign Key referenced column
- ALL - All of the above

Assigning Privileges

- New Syntax added and documented under GRANT command:
- GRANT { { SELECT | INSERT | UPDATE | REFERENCES } (column [, ...])
 - [,...] | ALL [PRIVILEGES] (column [, ...]) }
 - ON [TABLE] tablename [, ...]
 - TO { [GROUP] rolename | PUBLIC } [, ...] [WITH GRANT OPTION]
- To illustrate a bit more clearly:
- GRANT SELECT (col2, col3) ON mytable TO role1;
 - Grants select on columns “col2” and “col3” in table “mytable” to “role1”
- GRANT INSERT(col1), UPDATE (col2) ON mytable TO role2;
 - Grants insert on “col1”, update on “col2” in table “mytable” to “role2”
- GRANT SELECT, UPDATE(col3) ON mytable TO role3;
 - Grants select on the table, and update on “col3” for “mytable” to “role3”

Revoking Privileges

- New syntax added and documented under REVOKE
- REVOKE [GRANT OPTION FOR]
 - { { SELECT | INSERT | UPDATE | REFERENCES } (column [, ...])
 - [,...] | ALL [PRIVILEGES] (column [, ...]) }
 - ON [TABLE] tablename [, ...]
 - FROM { [GROUP] rolename | PUBLIC } [, ...]
 - [CASCADE | RESTRICT]
- Illustration:
 - REVOKE SELECT (col1) ON tab1 FROM user1;
- Effect of table-level revokes
 - REVOKE SELECT ON tab1 FROM user1;
 - This will also remove SELECT privileges from all columns of tab1 for user1!

Usage - Select

- create table tab1 (col1 int, col2 int, col3 int);
- grant select (col1) on tab1 to user1;
- (note: No table privileges granted or required! Not like a schema.)
- As user1:
 - select * from tab1; table tab1;
 - ERROR: permission denied for relation tab1
 - select col1 from tab1; -- works!
 - create table sneaky (col2 int);
 - select sneaky.col2 from tab1 natural join sneaky;
 - ERROR: permission denied for relation tab1
 - select tab1.col1, sneaky.col2 from tab1 join sneaky ON (tab1.col1 = sneaky.col2); -- works!

Usage - Update

- create table tab1 (col1 int, col2 int, col3 int);
- grant update (col2) on tab1 to user1;
- As user1:
 - update tab1 set col2 = 1 where col3 = 2;
 - ERROR: permission denied for relation tab1
 - What happened?
- grant select (col3) on tab1 to user1;
- As user1:
 - update tab1 set col2 = 1 where col3 = 2; -- works!
 - create table mine (col3 int);
 - update tab1 set col2 = 1 from mine where tab1.col3 = mine.col3; -- works!

Usage - Insert

- create table tab1 (col1 int, col2 int, col3 int);
- grant insert (col1) on tab1 to user1;
- As user1:
 - insert into tab1 values (1,NULL,NULL);
 - ERROR: permission denied for relation tab1
 - insert into tab1 values (1); -- works! But why?
- revoke all on tab1 from user1;
- grant insert (col2) on tab1 to user1;
- As user1:
 - insert into tab1 values (1);
 - ERROR: permission denied for relation tab1
 - insert into tab1 (col2) values (1); -- works!

Usage - References

- create table tab1 (col1 int primary key, col2 int unique, col3 int);
- grant references (col1) on tab1 to user1;
- As user1:
 - create table mine (col1 int references tab1, col4 int); -- works!
 - create table mine (col2 int, col4 int, foreign key (col2) references tab1 (col2));
 - ERROR: permission denied for relation tab1
 - select col1 from mine natural join tab1;
 - ERROR: permission denied for relation tab1
 - Still need select rights to use table in a join

Usage - Copy

- COPY command is also supported
- With SELECT privileges on col1 and col2
 - COPY tab1 (col1, col2) TO stdout;
- With INSERT privileges on col3
 - COPY tab1 (col3) FROM stdin;
- Using queries under COPY require regular SELECT privileges; eg:
 - COPY (select col1, col2 FROM tab1) TO stdout;

Special cases

- create table tab1 (col1 int, col2 int, col3 int);
- grant select (col1,col2,col3) on tab1 to user1;
- grant select (col2) on tab1 to user2;
- As user1:
 - select * from tab1; -- works!
 - select tab1 from tab1; -- works! (what's that?)
- alter table tab1 add column col4 int; (or what if a column is removed?)
- As user1:
 - select * from tab1;
 - ERROR: permission denied for relation tab1;
- Be careful with those select *'s!
- As user2: select mine.col1 from mine, tab1; -- works! (but why?)
 - Need column privileges on one column to include table in a join

Viewing Privileges

- postgres=# \dp
-
- Access Privileges
- Schema | Name | Type | Access Privileges | Column Access Privileges
- -----+-----+-----+-----+-----
- public | tabl | table | sfrost=arwdDxt/sfrost | col1:
- : user3=arw/sfrost : user1=x/sfrost
- : col2:
- : user2=arwx/sfrost
-
- What's it all mean?!

Viewing Privileges (cont'd)

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

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

Viewing Privileges (cont'd)

- Stored as *acl columns in most catalogs
- pg_class, relacl
 - {sfrost=arwdDxt/sfrost,t1=r/sfrost}
 - sfrost has all, granted by sfrost
 - t1 has select, granted by sfrost
- pg_attribute, attacl
- pg_database, dataacl
- Etc...

Controlling Access

- Levels of access control
- Containers
 - Database (control if user can connect)
 - Schema (control if user can use objects inside)
- Object-level (tables, columns, views, etc)
 - Grant only privileges necessary
 - Property of least privilege
- Complex controls (Views, Functions)

Using Database Container

- Option #1:
 - Disallow access to database with `pg_hba.conf`
 - Requires config file management
 - Requires postgresql reload to update
- Option #2:
 - Disallow `CONNECT` access to database
 - Must revoke `CONNECT` privilege from public
 - Should revoke `TEMPORARY` priv from public
 - Controls access in database instead of file

Using Schema Container

- Only controls access to use objects
- Create on public schema granted to public!
 - Recommend revoking unless necessary
- Object definitions still visible to all users which can connect to database
 - What objects (tables, views, etc) exist and their definitions
 - Table definitions (the columns)
 - View definitions
 - Function definitions

Using Views

- Views run with privileges of View owner
- Can be used to implement row-level security
- CREATE VIEW only_mine AS
 - SELECT * FROM all_data
 - WHERE user_allowed = CURRENT_USER;
- GRANT SELECT ON only_mine TO public;
- only_mine owned by user with rights to all_data
- If rights on all_data revoked from view owner, view will be useless and always return perm denied

Using Security-Definer Functions

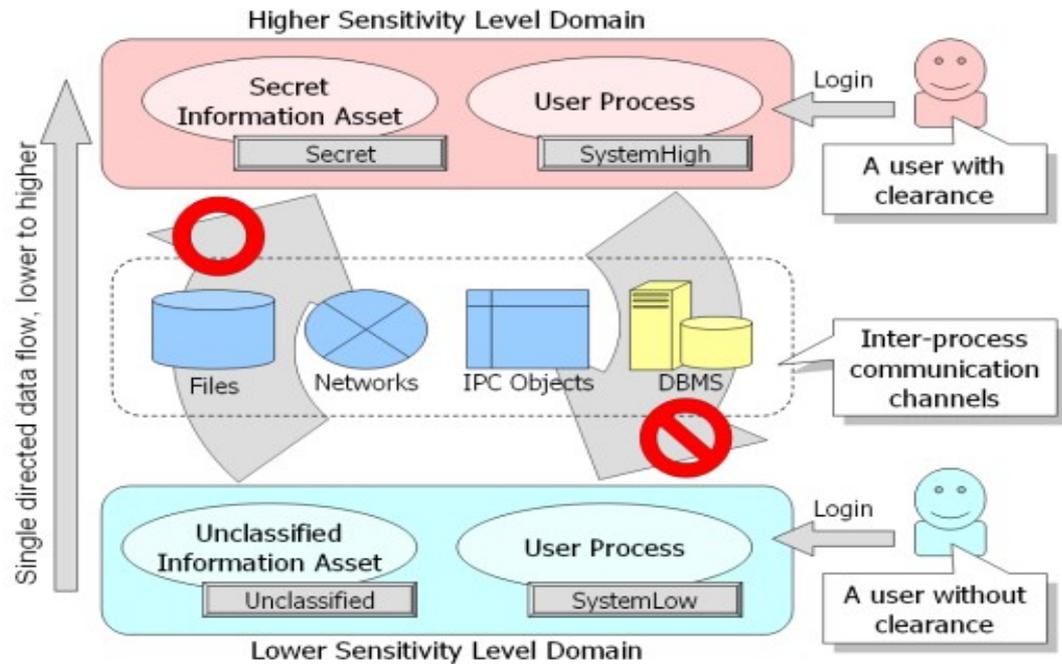
- Security-Definer Functions run as Function owner
- Can be used to implement complex security
- CREATE FUNCTION get_auser_data()
 - RETURNS integer
 - LANGUAGE SQL
 - SECURITY DEFINER
 - AS \$\$
 - select mynum from myschema.user_data where user_name = 'auser'; \$\$;
- REVOKE EXECUTE ON FUNCTION get_auser_data FROM public;
- GRANT EXECUTE ON FUNCTION get_auser_data TO auser;

SE-PostgreSQL

- Not in PostgreSQL core
- <http://wiki.postgresql.org/wiki/SEPostgreSQL>
- Provides MAC (mandatory access control)
 - No superuser with ability to do anything
 - Similar to/based on SELinux
- Provides row-level access control
 - Also exists under Oracle, SQL Server, etc.
 - Does so using labels
- Access control consistency- Network, OS, DB

SE-PostgreSQL (continued)

- Combined with other pieces, can track labels from user's desktop to database
- Kernel/Network
 - Linux
 - IPSEC
 - Filesystem/NFS
 - IPC
- Database
 - SE-PostgreSQL
- Web Server
 - Apache
 - mod_selinux
 - PHP/SELinux
- Transitions between labels must be specifically allowed by central policy



GUI Tools

- pgAdmin3
 - <http://www.pgadmin.org/>
- phpPgAdmin
 - <http://phppgadmin.sourceforge.net/>

Looking to 9.1

- Allow SSL over unix domain sockets?
- Support GSS/Kerberos-based encryption?
- Better CRL support? Support for OCSP?
- SE-PostgreSQL?
- Row-Level security?
- RADIUS redundant server support?
- Thoughts?

Questions?