PostgreSQL Audit Extension User Guide

Version 1.0.4

Open Source PostgreSQL Audit Logging

Crunchy Data Solutions, Inc.

June 24, 2016

Table of Contents

1 Introduction

The PostgreSQL Audit Extension (pgAudit) provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

The goal of the pgAudit is to provide PostgreSQL users with capability to produce audit logs often required to comply with government, financial, or ISO certifications.

An audit is an official inspection of an individual's or organization's accounts, typically by an independent body. The information gathered by pgAudit is properly called an audit trail or audit log. The term audit log is used in this documentation.

2 Why pgAudit?

Basic statement logging can be provided by the standard logging facility with `log_statement = all`. This is acceptable for monitoring and other usages but does not provide the level of detail generally required for an audit. It is not enough to have a list of all the operations performed against the database. It must also be possible to find particular statements that are of interest to an auditor. The standard logging facility shows what the user requested, while pgAudit focuses on the details of what happened while the database was satisfying the request.

For example, an auditor may want to verify that a particular table was created inside a documented maintenance window. This might seem like a simple job for grep, but what if you are presented with something like this (intentionally obfuscated) example:

```
BEGIN
   EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

Standard logging will give you this:

```
LOG: statement: DO $$
BEGIN
   EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

It appears that finding the table of interest may require some knowledge of the code in cases where tables are created dynamically. This is not ideal since it would be preferable to just search on the table name. This is where pgAudit comes in. For the same input, it will produce this output in the log:

```
AUDIT: SESSION,33,1,FUNCTION,DO,,,"DO $$
BEGIN
   EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;"
AUDIT: SESSION,33,2,DDL,CREATE TABLE,TABLE,public.important_table,CREATE TABLE
   important_table (id INT)
```

Not only is the `DO` block logged, but substatement 2 contains the full text of the `CREATE TABLE` with the statement type, object type, and full-qualified name to make searches easy.

When logging `SELECT` and `DML` statements, pgAudit can be configured to log a separate entry for each relation referenced in a statement. No parsing is required to find all statements that touch a particular table. In fact, the goal is that the statement text is provided primarily for deep forensics and should not be required for an audit.

3 Usage Considerations

Depending on settings, it is possible for pgAudit to generate an enormous volume of logging. Be careful to determine exactly what needs to be audit logged in your environment to avoid logging too much.

For example, when working in an OLAP environment it would probably not be wise to audit log inserts into a large fact table. The size of the log file will likely be many times the actual data size of the inserts because the log file is expressed as text. Since logs are generally stored with the OS this may lead to disk space being exhausted very quickly. In cases where it is not possible to limit audit logging to certain tables, be sure to assess the performance impact while testing and allocate plenty of space on the log volume. This may also be true for OLTP environments. Even if the insert volume is not as high, the performance impact of audit logging may still noticeably affect latency.

To limit the number of relations audit logged for `SELECT` and `DML` statements, consider using object audit logging (see OBJECT AUDITING). Object audit logging allows selection of the relations to be logged allowing for reduction of the overall log volume. However, when new relations are added they must be explicitly added to object audit logging. A programmatic solution where specified tables are excluded from logging and all others are included may be a good option in this case.

4 Installation

Please refer to section 4.8 of the Crunchy Certified PostgreSQL Secure Installation and Configuration Guide.

5 Settings

Settings may be modified only by a superuser. Allowing normal users to change their settings would defeat the point of an audit log.

Settings can be specified globally (in `postgresql.conf` or using `ALTER SYSTEM ... SET`), at the database level (using `ALTER DATABASE ... SET`), or at the role level (using `ALTER ROLE ... SET`). Note that settings are not inherited through normal role inheritance and `SET ROLE` will not alter a user's pgAudit settings. This is a limitation of the roles system and not inherent to pgAudit.

The pgAudit extension must be loaded in SHARED_PRELOAD_LIBRARIES. Otherwise, an error will be raised at load time and no audit logging will occur.

5.1 pgaudit.log

Specifies which classes of statements will be logged by session audit logging. Possible values are:

* **READ**: `SELECT` and `COPY` when the source is a relation or a query.

* **WRITE**: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY` when the destination is a relation.

* **FUNCTION**: Function calls and `DO` blocks.

* **ROLE**: Statements related to roles and privileges: `GRANT`, `REVOKE`, `CREATE/ALTER/DROP ROLE`.

* **DDL**: All `DDL` that is not included in the `ROLE` class.

* **MISC**: Miscellaneous commands, e.g. `DISCARD`, `FETCH`, `CHECKPOINT`, `VACUUM`.

Multiple classes can be provided using a comma-separated list and classes can be subtracted by prefacing the class with a - sign (see SESSION AUDIT LOGGING).

The default is `none`.

5.2 pgaudit.log_catalog

Specifies that session logging should be enabled in the case where all relations in a statement are in pg_catalog. Disabling this setting will reduce noise in the log from tools like psql and PgAdmin that query the catalog heavily.

The default is `on`.

5.3 pgaudit.log_level

Specifies the log level that will be used for log entries (see MESSAGE SEVERITY LEVELS for valid levels) but note that `ERROR`, `FATAL`, and `PANIC` are not allowed). This setting is used for regression testing

and may also be useful to end users for testing or other purposes.

The default is `log`.

5.4 pgaudit.log_parameter

Specifies that audit logging should include the parameters that were passed with the statement. When parameters are present they will be included in `CSV` format after the statement text.

The default is `off`.

5.5 pgaudit.log_relation

Specifies whether session audit logging should create a separate log entry for each relation (`TABLE`, `VIEW`, etc.) referenced in a `SELECT` or `DML` statement. This is a useful shortcut for exhaustive logging without using object audit logging.

The default is `off`.

5.6 pgaudit.log_statement_once

Specifies whether logging will include the statement text and parameters with the first log entry for a statement/substatement combination or with every entry. Disabling this setting will result in less verbose logging but may make it more difficult to determine the statement that generated a log entry, though the statement/substatement pair along with the process id should suffice to identify the statement text logged with a previous entry.

The default is `off`.

5.7 pgaudit.role

Specifies the master role to use for object audit logging. Multiple audit roles can be defined by granting them to the master role. This allows multiple groups to be in charge of different aspects of audit logging.

There is no default.

6 Session Audit Logging

Session audit logging provides detailed logs of all statements executed by a user in the backend.

6.1 Configuration

Session logging is enabled with the PGAUDIT.LOG setting.

Enable session logging for all DML and DDL and log all relations in DML statements:

```
set pgaudit.log = 'write, ddl';
set pgaudit.log_relation = on;
```

Enable session logging for all commands except MISC and raise audit log messages as NOTICE:

```
set pgaudit.log = 'all, -misc';
set pgaudit.log_level = notice;
```

6.2 Example

In this example session audit logging is used for logging DDL and SELECT statements. Note that the insert statement is not logged since the WRITE class is not enabled

SQL:

```
set pgaudit.log = 'read, ddl';

create table account
(
   id int,
   name text,
   password text,
   description text
);

insert into account (id, name, password, description)
          values (1, 'user1', 'HASH1', 'blah, blah');

select *
   from account;
```

Log Output:

```
AUDIT: SESSION,1,1,DDL,CREATE TABLE,TABLE,public.account,create table account
(
```

```
   id int,
   name text,
   password text,
   description text
);
AUDIT: SESSION,2,1,READ,SELECT,,,select *
   from account
```

7 Object Audit Logging

Object audit logging logs statements that affect a particular relation. Only SELECT, INSERT, UPDATE and DELETE commands are supported. TRUNCATE is not included in object audit logging.

Object audit logging is intended to be a finer-grained replacement for `pgaudit.log = 'read, write'`. As such, it may not make sense to use them in conjunction but one possible scenario would be to use session logging to capture each statement and then supplement that with object logging to get more detail about specific relations.

7.1 Configuration

Object-level audit logging is implemented via the roles system. The PGAUDIT.ROLE setting defines the role that will be used for audit logging. A relation (TABLE, VIEW, etc.) will be audit logged when the audit role has permissions for the command executed or inherits the permissions from another role. This allows you to effectively have multiple audit roles even though there is a single master role in any context.

Set PGAUDIT.ROLE to `auditor` and grant SELECT and DELETE privileges on the `account` table. Any SELECT or DELETE statements on the `account` table will now be logged:

```
set pgaudit.role = 'auditor';

grant select, delete
  on public.account
  to auditor;
```

7.2 Example

In this example object audit logging is used to illustrate how a granular approach may be taken towards logging of SELECT and DML statements. Note that logging on the `account` table is controlled by column-level permissions, while logging on the `account_role_map` table is table-level.

SQL:

```
set pgaudit.role = 'auditor';

create table account
(
   id int,
   name text,
   password text,
   description text
);
```

```
grant select (password)
  on public.account
  to auditor;

select id, name
 from account;

select password
 from account;

grant update (name, password)
  on public.account
  to auditor;

update account
  set description = 'yada, yada';

update account
  set password = 'HASH2';

create table account_role_map
(
   account_id int,
   role_id int
);

grant select
  on public.account_role_map
  to auditor;

select account.password,
     account_role_map.role_id
 from account
     inner join account_role_map
         on account.id = account_role_map.account_id
```

Log Output:

```
AUDIT: OBJECT,1,1,READ,SELECT,TABLE,public.account,select password
 from account
AUDIT: OBJECT,2,1,WRITE,UPDATE,TABLE,public.account,update account
  set password = 'HASH2'
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account,select account.password,
     account_role_map.role_id
 from account
     inner join account_role_map
         on account.id = account_role_map.account_id
```

```
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account_role_map,select account.
   password,
     account_role_map.role_id
 from account
     inner join account_role_map
         on account.id = account_role_map.account_id
```

8 Format

Audit entries are written to the standard logging facility and contain the following columns in comma-separated format. Output is compliant CSV format only if the log line prefix portion of each log entry is removed.

* **AUDIT_TYPE** - `SESSION` or `OBJECT`.

* **STATEMENT_ID** - Unique statement ID for this session. Each statement ID represents a backend call. Statement IDs are sequential even if some statements are not logged. There may be multiple entries for a statement ID when more than one relation is logged.

* **SUBSTATEMENT_ID** - Sequential ID for each sub-statement within the main statement. For example, calling a function from a query. Sub-statement IDs are continuous even if some sub-statements are not logged. There may be multiple entries for a sub-statement ID when more than one relation is logged.

* **CLASS** - e.g. `READ`, `ROLE` (see PGAUDIT.LOG).

* **COMMAND** - e.g. `ALTER TABLE`, `SELECT`.

* **OBJECT_TYPE** - `TABLE`, `INDEX`, `VIEW`, etc. Available for `SELECT`, `DML` and most `DDL` statements.

* **OBJECT_NAME** - The fully-qualified object name (e.g. public.account). Available for `SELECT`, `DML` and most `DDL` statements.

* **STATEMENT** - Statement executed on the backend.

* **PARAMETER** - If `pgaudit.log_parameter` is set then this field will contain the statement parameters as quoted CSV.

Use LOG_LINE_PREFIX to add any other fields that are needed to satisfy your audit log requirements. A typical log line prefix might be `'%m %u %d: '` which would provide the date/time, user name, and database name for each audit log.

9 Caveats

* Object renames are logged under the name they were renamed to. For example, renaming a table will produce the following result:

```
ALTER TABLE test RENAME TO test2;

AUDIT: SESSION,36,1,DDL,ALTER TABLE,TABLE,public.test2,ALTER TABLE test RENAME TO
   test2
```

* It is possible to have a command logged more than once. For example, when a table is created with a primary key specified at creation time the index for the primary key will be logged independently and another audit log will be made for the index under the create entry. The multiple entries will however be contained within one statement ID.

* Autovacuum and Autoanalyze are not logged.

* Statements that are executed after a transaction enters an aborted state will not be audit logged. However, the statement that caused the error and any subsequent statements executed in the aborted transaction will be logged as ERRORs by the standard logging facility.

10 PostgreSQL Audit Log Analyzer

The PostgreSQL Audit extension (pgAudit) provides detailed session and/or object audit logging via the standard PostgreSQL logging facility. However, logs are not the ideal place to store audit information. The PostgreSQL Audit Log Analyzer (pgAudit Analyze) reads audit entries from the PostgreSQL logs and loads them into a database schema to aid in analysis and auditing.

10.1 Installation

Please refer to section 4.9 of the Crunchy Certified PostgreSQL Secure Installation and Configuration Guide.

10.2 pgAudit Schema

The `pgaudit_analyze` processes monitors the PostgreSQL `CSV` log output and loads all log and audit events into the `pgaudit` schema. Session and statement errors set the `state` column in the `session` and `audit_statement` tables respectively.

In addition, logon information for users on each audited database is collected and stored.

10.2.1 Tables

10.2.1.1 session table

Contains information about PostgreSQL sessions.

pgAudit tracks whether connections succeed or fail and the result is stored in the `state` column. Additional information about the error can be found by joining to the `log_event` table on the `session_id` column.

Column	Type	Nullable	Description
session_id	text	N	PostgreSQL session identifier.
session_start_time	timestamp	N	Timestamp of session start.
user_name	text	N	PostgreSQL user name.
application_name	text	Y	Session application name.
connection_from	text	Y	IP/Host that the session was initiated from.
state	text	N	Was the connection successful: `OK` or `ERROR`.

10.2.1.2 logon table

Tracks information about user logons.

Column	Type	Nullable	Description
user_name	text	N	PostgreSQL user name.
last_success	timestamp	Y	Timestamp of last successful logon.
current_success	timestamp	Y	Timestamp of current successful logon.
last_failure	timestamp	Y	Timestamp of last logon failure.
failures_since_last_success	int	Y	Number of failed logon attempts since the last successful logon.

10.2.1.3 log_event table Contains PostgreSQL `CSV` log entries. Information from the `CSV` log is loaded into this table and the `session` table.

If a statement fails then the `state` column will be set to `error`. Additional information about the error can be found by joining to the `log_event` table on the `session_id` and `session_line_num` columns.

Column	Type	Nullable	Description
session_id	text	N	PostgreSQL session identifier.
session_line_num	numeric	N	Sequential statement id generated by PostgreSQL.
log_time	timestamp(3)	N	Timestamp with milliseconds.
command	text	Y	Command tag - type of session's command.
error_severity	text	Y	Type of error (e.g. `ERROR`, `PANIC`).
sql_state_code	text	Y	SQLSTATE error code.
virtual_transaction_id	text	Y	Virtual transaction ID (backendID/localXID).
transaction_id	bigint	Y	Transaction ID (0 if none is assigned).
message	text	Y	Text of the log message.
detail	text	Y	Additional detail for the log message.
hint	text	Y	A hint for interpreting the log message.
query	text	Y	Text of the query.
query_pos	int	Y	On error, position in the query where the error occurred.
internal_query	text	Y	Text of the internal query.
internal_query_pos	int	Y	On error, position in the internal query where the error occurred.
context	text	Y	Additional information about where the query was executed.
location	text	Y	On error, location in the PostgreSQL source code where the error occurred.

10.2.1.4 audit_statement table

Contains the top-level statement information. A statement may contain numerous sub-statements which are tracked in the `audit_substatement` table.

Column	Type	Nullable	Description
session_id	text	N	PostgreSQL session identifier.
statement_id	numeric	N	Sequential statement ID generated by pgAudit.
state	text	N	Final state of the statement: `ok` or `error`
error_session_line_num	bigint	Y	If `state` is `error` then this column contains the PostgreSQL session line number where error details can be found.

10.2.1.5 audit_substatement table

Contains the sub-statements that make up statement.

There are many cases where a user statement can contain multiple sub-statements. For example, if the user statement contains a `DO` block or a function call then multiple sub-statements can be generated.

Column	Type	Nullable	Description
session_id	text	N	PostgreSQL session identifier.
statement_id	numeric	N	Sequential statement ID generated by pgAudit.
substatement_id	numeric	N	Sequential sub-statement ID generated by pgAudit.
substatement	text	Y	Sub-statement executed.
parameter	text[]	Y	Parameters passed to the sub-statement.

10.2.1.6 audit_substatement_detail table

Contains individual sub-statement audit events.

There may be multiple auditable events in a single sub-statement. For example, a query may have a number of tables that are being individually audit logged. In that case, an `audit_substatement_detail` record will be created for each table.

Column	Type	Nullable	Description
session_id	text	N	PostgreSQL session identifier.
statement_id	numeric	N	Sequential statement ID generated by pgAudit.
substatement_id	numeric	N	Sequential sub-statement ID generated by pgAudit.
session_line_num	numeric	N	Sequential statement id generated by PostgreSQL.
audit_type	text	N	Type of audit record generated: `SESSION` or `OBJECT`.
class	text	Y	Logging class of the audit record: `READ`, `WRITE`, `FUNCTION`, `ROLE`, `DDL`, `MISC`
command	text	Y	SQL command executed (e.g. `ALTER TABLE`, `SELECT`).
object_type	text	Y	SQL object type (e.g. `TABLE`, `INDEX`, `VIEW`).
object_name	text	Y	The fully-qualified object name (e.g. public.account).

10.2.2 Views

10.2.2.1 vw_audit_event view Produces a view of the `pgaudit` schema that brings together all the major information needed for auditing an analysis.

View definition (join criteria removed for brevity):

```
create view pgaudit.vw_audit_event as
select session.session_id,
       log_event.session_line_num,
       log_event.log_time,
       session.user_name,
       audit_statement.statement_id,
       audit_statement.state,
       audit_statement.error_session_line_num,
       audit_substatement.substatement_id,
       audit_substatement.substatement,
       audit_substatement_detail.audit_type,
       audit_substatement_detail.class,
       audit_substatement_detail.command,
       audit_substatement_detail.object_type,
       audit_substatement_detail.object_name
  from pgaudit.audit_substatement_detail
       inner join pgaudit.log_event
           [...]
       inner join pgaudit.session
           [...]
       inner join pgaudit.audit_substatement
           [...]
       inner join pgaudit.audit_statement
           [...]
```

10.2.3 Functions

10.2.3.1 logon_info() function Allows the user to query logon information for the current database. The function returns the following values:

Column	Type	Nullable	Description
last_success	timestamp	Y	Timestamp of last successful logon.
last_failure	timestamp	Y	Timestamp of last logon failure.
failures_since_last_success	int	Y	Number of failed logon attempts since the last successful logon.

10.3 Caveats

* The pgaudit.logon table contains the logon information for users of the database. If a user is renamed they must also be renamed in this table or the logon history will be lost.

* Reads and writes to the pgAudit schema by the user running pgAudit Analyze are never logged.

11 PostgreSQL set_user Extension Module

11.1 Syntax

```
set_user(text rolename) returns text
reset_user() returns text
```

11.2 Inputs

rolename is the role to be transitioned to.

11.3 Requirements

* Add set_user to shared_preload_libraries in postgresql.conf.

* Optionally, the following custom parameters may be set to control their respective commands:

* set_user.block_alter_system = off (defaults to 'on')

* set_user.block_copy_program = off (defaults to 'on')

* set_user.block_log_statement = off (defaults to 'on')

11.4 Description

This PostgreSQL extension allows privilege escalation with enhanced logging and control. It provides an additional layer of logging and control when unprivileged users must escalate themselves to superuser or object owner roles in order to perform needed maintenance tasks. Specifically, when an allowed user executes **set_user('rolename')**, several actions occur:

* The current effective user becomes **rolename**.

* The role transition is logged, with specific notation if **rolename** is a superuser.

* log_statement setting is set to 'all', meaning every SQL statement executed while in this state will also get logged.

* If set_user.block_alter_system is set to 'on', **ALTER SYSTEM** commands will be blocked.

* If set_user.block_copy_program is set to 'on', **COPY PROGRAM** commands will be blocked.

* If set_user.block_log_statement is set to 'on', **SET log_statement** and variations will be blocked.

When finished with required actions as **rolename**, the **reset_user()** function is executed to restore the original user. At that point, these actions occur:

* Role transition is logged.

* log_statement setting is set to its original value.

* Blocked command behaviors return to normal.

The concept is to grant the EXECUTE privilege to the **set_user()** function to otherwise unprivileged postgres users who can then escalate themselves when needed to perform specific superuser actions. The enhanced logging ensures an audit trail of what actions are taken while privileges are escalated.

Once one or more unprivileged users are able to run **set_user()**, the superuser account (normally **postgres**) can be altered to NOLOGIN, preventing any direct database connection by a superuser which would bypass the enhanced logging.

Naturally for this to work as expected, the PostgreSQL cluster must be audited to ensure there are no other PostgreSQL roles existing which are both superuser and can log in. Additionally there must be no unprivileged PostgreSQL roles which have been granted access to one of the existing superuser roles.

Note that for the blocking of **ALTER SYSTEM** and **COPY PROGRAM** to work properly, you must include **set_user** in shared_preload_libraries in postgresql.conf and restart PostgreSQL.

11.5 Caveats

In its current state, this extension cannot prevent **rolename** from performing a variety of nefarious or otherwise undesireable actions. However, these actions will be logged providing an audit trail, which could also be used to trigger alerts.

Although this extension compiles and works with all supported versions of PostgreSQL starting with PostgreSQL 9.1, all features are not supported until PostgreSQL 9.4 or higher. The ALTER SYSTEM command does not exist prior to 9.4 and COPY PROGRAM does not exist prior to 9.3.

11.6 Installation

11.6.1 Requirements

* PostgreSQL 9.1 or higher.

11.6.2 Compile and Install

Clone PostgreSQL repository:

```
git clone https://github.com/postgres/postgres.git
```

Checkout REL9_5_STABLE (for example) branch:

```
git checkout REL9_5_STABLE
```

Make PostgreSQL:

```
./configure
make install -s
```

Change to the contrib directory:

```
cd contrib
```

Clone **set_user** extension:

```
git clone https://github.com/pgaudit/set_user
```

Change to **set_user** directory:

```
cd set_user
```

Build **set_user**:

```
make
```

Install **set_user**:

```
make install
```

11.6.3 Using PGXS

If an instance of PostgreSQL is already installed, then PGXS can be utilized to build and install **set_user**. Ensure that PostgreSQL binaries are available via the **PATH** environment variable then use the following commands.

```
make USE_PGXS=1
make USE_PGXS=1 install
```

11.7 Configure

The following bash commands should configure your system to utilize set_user. Replace all paths as appropriate. It may be prudent to visually inspect the files afterward to ensure the changes took place.

11.7.1 Initialize PostgreSQL (if needed):

```
initdb -D /path/to/data/directory
```

11.7.2 Create Target Database (if needed):

```
createdb {database}
```

11.7.3 Install set_user functions:

Edit postgresql.conf and add **set_user** to the shared_preload_libraries line, optionally also setting block_alter_sys
and/or block_copy_program.

First edit postgresql.conf in your favorite editor:

```
vi PGDATA/postgresql.conf
```

Then add these lines to the end of the file:

```
# Add set_user to any existing list
shared_preload_libraries = 'set_user'
# The following lines are only required to turn off the
# blocking of each respective command if desired
set_user.block_alter_system = off #defaults to 'on'
set_user.block_copy_program = off #defaults to 'on'
set_user.block_log_statement = off #defaults to 'on'
```

Finally, restart PostgreSQL (method may vary):

```
service postgresql restart
```

Install the extension into your database:

```
psql {database}
CREATE EXTENSION set_user;
```

11.8 GUC Parameters

* Block ALTER SYSTEM commands

* set_user.block_alter_system = on

* Block COPY PROGRAM commands

* set_user.block_copy_program = on

* Block SET log_statement commands

* set_user.block_log_statement = on

11.9 Examples

```
###################################
# OS command line, terminal 1
###################################
psql -U postgres {dbname}

---------------------------------
-- psql command line, terminal 1
---------------------------------
SELECT rolname FROM pg_authid WHERE rolsuper and rolcanlogin;
 rolname
----------
 postgres
(1 row)

CREATE EXTENSION set_user;
CREATE USER dba_user;
GRANT EXECUTE ON FUNCTION set_user(text) TO dba_user;

###################################
# OS command line, terminal 2
###################################
psql -U dba_user {dbname}

---------------------------------
-- psql command line, terminal 2
---------------------------------
SELECT set_user('postgres');
SELECT CURRENT_USER, SESSION_USER;
 current_user | session_user
--------------+--------------
 postgres  | dba_user
(1 row)

SELECT reset_user();
SELECT CURRENT_USER, SESSION_USER;
 current_user | session_user
```

```
-------------+-------------
 dba_user  | dba_user
(1 row)

\q

---------------------------------
-- psql command line, terminal 1
---------------------------------
ALTER USER postgres NOLOGIN;
-- repeat terminal 2 test with dba_user before exiting
\q

#################################
# OS command line, terminal 1
#################################
tail -n 6 {postgres log}
LOG: Role dba_user transitioning to Superuser Role postgres
STATEMENT: SELECT set_user('postgres');
LOG: statement: SELECT CURRENT_USER, SESSION_USER;
LOG: statement: SELECT reset_user();
LOG: Superuser Role postgres transitioning to Role dba_user
STATEMENT: SELECT reset_user();

#################################
# OS command line, terminal 2
#################################
psql -U dba_user {dbname}

---------------------------------
-- psql command line, terminal 2
---------------------------------
-- Verify there are no superusers that can login directly
SELECT rolname FROM pg_authid WHERE rolsuper and rolcanlogin;
 rolname
---------
(0 rows)

-- Verify there are no unprivileged roles that can login directly
-- that are granted a superuser role even if it is multiple layers
-- removed
DROP VIEW IF EXISTS roletree;
CREATE OR REPLACE VIEW roletree AS
WITH RECURSIVE
roltree AS (
  SELECT u.rolname AS rolname,
       u.oid AS roloid,
       u.rolcanlogin,
```

```
      u.rolsuper,
      '{}'::name[] AS rolparents,
      NULL::oid AS parent_roloid,
      NULL::name AS parent_rolname
 FROM pg_catalog.pg_authid u
 LEFT JOIN pg_catalog.pg_auth_members m on u.oid = m.member
 LEFT JOIN pg_catalog.pg_authid g on m.roleid = g.oid
 WHERE g.oid IS NULL
 UNION ALL
 SELECT u.rolname AS rolname,
      u.oid AS roloid,
      u.rolcanlogin,
      u.rolsuper,
      t.rolparents || g.rolname AS rolparents,
      g.oid AS parent_roloid,
      g.rolname AS parent_rolname
 FROM pg_catalog.pg_authid u
 JOIN pg_catalog.pg_auth_members m on u.oid = m.member
 JOIN pg_catalog.pg_authid g on m.roleid = g.oid
 JOIN roltree t on t.roloid = g.oid
)
SELECT
 r.rolname,
 r.roloid,
 r.rolcanlogin,
 r.rolsuper,
 r.rolparents
FROM roltree r
ORDER BY 1;

-- For example purposes, given this set of roles
SELECT r.rolname, r.rolsuper, r.rolinherit,
 r.rolcreaterole, r.rolcreatedb, r.rolcanlogin,
 r.rolconnlimit, r.rolvaliduntil,
 ARRAY(SELECT b.rolname
      FROM pg_catalog.pg_auth_members m
      JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid)
      WHERE m.member = r.oid) as memberof
, r.rolreplication
, r.rolbypassrls
FROM pg_catalog.pg_roles r
ORDER BY 1;
                          List of roles
 Role name |                Attributes                | Member of
-----------+------------------------------------------+------------

 bob       |                                          | {}
 dba_user  |                                          | {su}
```

```
 joe     |                                                    | {newbs}
 newbs   | Cannot login                                       | {}
 postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 su      | No inheritance, Cannot login                       | {postgres}

-- This query shows current status is not acceptable
-- 1) postgres can login directly
-- 2) dba_user can login and is able to escalate without using set_user()
SELECT
  ro.rolname,
  ro.roloid,
  ro.rolcanlogin,
  ro.rolsuper,
  ro.rolparents
FROM roletree ro
WHERE (ro.rolcanlogin AND ro.rolsuper)
OR
(
   ro.rolcanlogin AND EXISTS
   (
     SELECT TRUE FROM roletree ri
     WHERE ri.rolname = ANY (ro.rolparents)
     AND ri.rolsuper
   )
);
 rolname | roloid | rolcanlogin | rolsuper | rolparents
---------+--------+-------------+----------+--------------
 dba_user | 16387 | t     | f      | {postgres,su}
 postgres |  10 | t        | t      | {}
(2 rows)

-- Fix it
REVOKE postgres FROM su;
ALTER USER postgres NOLOGIN;

-- Rerun the query - shows current status is acceptable
SELECT
  ro.rolname,
  ro.roloid,
  ro.rolcanlogin,
  ro.rolsuper,
  ro.rolparents
FROM roletree ro
WHERE (ro.rolcanlogin AND ro.rolsuper)
OR
(
   ro.rolcanlogin AND EXISTS
   (
```

```
    SELECT TRUE FROM roletree ri
    WHERE ri.rolname = ANY (ro.rolparents)
    AND ri.rolsuper
   )
);
 rolname | roloid | rolcanlogin | rolsuper | rolparents
---------+--------+-------------+----------+------------
(0 rows)
```

11.10 Licensing

Please see the LICENSE file.