OpenSCG PostgreSQL Operational Procedures

Security Hardening PostgreSQL

Version 2.0

Scott Mead
Sr. Architect, OpenSCG
Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-07-25</td>
<td>Scott Mead</td>
<td>1.0</td>
<td>Initial Document</td>
</tr>
<tr>
<td>2012-08-15</td>
<td>Scott Mead</td>
<td>1.5</td>
<td>Updated security privileges</td>
</tr>
<tr>
<td>2013-04-02</td>
<td>Scott Mead</td>
<td>2.0</td>
<td>More Detail on RBAC</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>4</td>
</tr>
<tr>
<td>Audience</td>
<td>4</td>
</tr>
<tr>
<td>Overview</td>
<td>4</td>
</tr>
<tr>
<td>Security Mechanisms</td>
<td>5</td>
</tr>
<tr>
<td>Role Based Access Control</td>
<td>5</td>
</tr>
<tr>
<td>Users</td>
<td>5</td>
</tr>
<tr>
<td>Groups</td>
<td>5</td>
</tr>
<tr>
<td>Roles</td>
<td>5</td>
</tr>
<tr>
<td>Privileges</td>
<td>6</td>
</tr>
<tr>
<td>Best Practice</td>
<td>8</td>
</tr>
<tr>
<td>Authentication</td>
<td>8</td>
</tr>
<tr>
<td>Password Authentication</td>
<td>8</td>
</tr>
<tr>
<td>LDAP Authentication</td>
<td>8</td>
</tr>
<tr>
<td>Kerberos Authentication</td>
<td>9</td>
</tr>
<tr>
<td>Best Practices</td>
<td>9</td>
</tr>
<tr>
<td>Host Based Access</td>
<td>9</td>
</tr>
<tr>
<td>Best Practices</td>
<td>10</td>
</tr>
<tr>
<td>Data Layout</td>
<td>10</td>
</tr>
<tr>
<td>Physical Separation</td>
<td>10</td>
</tr>
<tr>
<td>Best Practices</td>
<td>11</td>
</tr>
<tr>
<td>Logical Separation</td>
<td>11</td>
</tr>
<tr>
<td>Best Practices</td>
<td>11</td>
</tr>
<tr>
<td>Database hosts</td>
<td>12</td>
</tr>
<tr>
<td>Operating system</td>
<td>12</td>
</tr>
<tr>
<td>Best Practices</td>
<td>12</td>
</tr>
<tr>
<td>Physical Access</td>
<td>12</td>
</tr>
</tbody>
</table>
Purpose

This document is designed to give the reader a set of standard procedures to follow when hardening PostgreSQL for security purposes in a production environment. This document follows basic best practices in use through the computer industry with regards to security as well as things typically done in production PostgreSQL database environments. Data security is a larger topic that is out of scope here, although will be touched on briefly.

Audience

This document is designed to be consumed by individuals responsible for:

- Infrastructure management
- Computer security
- Data security
- Systems administration
- Database administration

The focus of this document is to bring technical solutions to common systems security issues in the context of a production PostgreSQL infrastructure.

Overview

Database systems live at the heart of critical infrastructure. They are responsible for consuming and producing data for every aspect of an organization, and are typically accessed by many different applications and individuals.

PostgreSQL is the world's most advanced open-source RDBMS (relational database management system) and provides a number of security mechanisms in order to effectively secure and manage the access to the system. PostgreSQL also provides modules like pg_crypto, which provide in-database encryption mechanisms for protecting individual data. This topic is out of scope for this document.
Security Mechanisms

Role Based Access Control

PostgreSQL has a built-in RBAC system for managing access not only to the database system, but to the database constructs as well.

Users

A postgresql user is a role that has the CONNECT privilege associated with it.

Groups

A group is a role that has other roles as members. If a role 'securitygroupa' exists and has member roles 'scott' and 'joe', then 'scott' and 'joe' will have the privileges assigned to 'securitygroupa'.

Roles

A role is a database construct that provides a list of specific security privileges and applies them, based on which role is currently logged in, to database constructs (listed in hierarchical order):

- Instance
- Database
  - Schema
    - Table
    - Sequence
    - View
    - Procedure

Roles are available at the database 'instance' level, this makes them assignable across all databases in the instance (or cluster), and can have attributes assigned to any of the sub-objects within a database.

Roles can also be assigned certain, server-side parameters to be enforced on login, these settings can be used to change everything from the types of query plans that can be executed to memory parameters available to the user. Custom parameters can even be defined and set. For a full list of these options, login to a PostgreSQL database and run:

```
SELECT * FROM pg_settings where context='user';
```
Privileges

The list of privileges change with every PostgreSQL release and can be found at (http://www.postgresql.org/docs/current/static/sql-grant.html). The list here is associated with the PostgreSQL 9.2 major release.

**SELECT**

Allows `SELECT` from any column, or the specific columns listed, of the specified table, view, or sequence. Also allows the use of `COPY` TO. This privilege is also needed to reference existing column values in `UPDATE` or `DELETE`. For sequences, this privilege also allows the use of the `currval` function. For large objects, this privilege allows the object to be read.

**INSERT**

Allows `INSERT` of a new row into the specified table. If specific columns are listed, only those columns may be assigned to in the `INSERT` command (other columns will therefore receive default values). Also allows `COPY` FROM.

**UPDATE**

Allows `UPDATE` of any column, or the specific columns listed, of the specified table. (In practice, any nontrivial `UPDATE` command will require `SELECT` privilege as well, since it must reference table columns to determine which rows to update, and/or to compute new values for columns.) `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege on at least one column, in addition to the `SELECT` privilege. For sequences, this privilege allows the use of the `nextval` and `setval` functions. For large objects, this privilege allows writing or truncating the object.

**DELETE**

Allows `DELETE` of a row from the specified table. (In practice, any nontrivial `DELETE` command will require `SELECT` privilege as well, since it must reference table columns to determine which rows to delete.)

**TRUNCATE**

Allows `TRUNCATE` on the specified table.

**REFERENCES**

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced columns. The privilege may be granted for all columns of a table, or just specific columns.
TRIGGER
Allows the creation of a trigger on the specified table. (See the CREATE TRIGGER statement.)

CREATE
For databases, allows new schemas to be created within the database.
For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.
For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT
Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by pg_hba.conf).

TEMPORARY / TEMP
Allows temporary tables to be created while using the specified database.

EXECUTE
Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE
For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.
For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to "look up" objects within the schema. Without this permission, it is still possible to see the object names, e.g. by querying the system tables. Also, after revoking this permission, existing backends might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access.
For sequences, this privilege allows the use of the currval and nextval functions.
For types and domains, this privilege allow the use of the type or domain in the creation of tables, functions, and other schema objects. (Note that it does not control general"usage" of the type, such as values of the type appearing in queries. It only prevents objects from being created that depend on the type. The main purpose of the privilege is controlling which users create dependencies on a type, which could prevent the owner from changing the type later.)
For foreign-data wrappers, this privilege enables the grantee to create new servers using that foreign-data wrapper.
Security Hardening PostgreSQL – Scott Mead

For servers, this privilege enables the grantee to create, alter, and drop his own user's user mappings associated with that server. Also, it enables the grantee to query the options of the server and associated user mappings.

ALL PRIVILEGES

Grant all of the available privileges at once. The PRIVILEGES key word is optional in PostgreSQL, though it is required by strict SQL.

Best Practice

Best practices for RBAC are as follows:

1. A database user (role with CONNECT) should be assigned to each type of application that is logging in to the database. This allows for privilege separation by login role when necessary.

2. Database objects (databases, tables, schemas, etc...) should be owned by a role that does not commonly modify the data in those tables. In other words, the application and ad-hoc users that are writing INSERT / UPDATE / DELETE / SELECT statements against the objects should not own them. This prevents against accidental or nefarious changes to the data model and structure.

3. Access roles should be created based on application access patterns to provide access to individual database objects, assigning access directly to logon roles should be avoided when possible.

4. Assign logon roles to the access roles in order to provide them access to a set or group of database objects.

Authentication

PostgreSQL provides multiple methods of role-authentication in addition to its own internal password system.

Password Authentication

The default method for authenticating to a PostgreSQL database is the internal password authentication system. This system stores only hashes of the role password in the database and validates that the hash provided by the inbound authentication request matches the hash as stored locally.

In this system, users and passwords are managed by the database administrator in the postgres database. Options are available to set password expiration.
LDAP Authentication

LDAP authentication is available in PostgreSQL, LDAP authorization is not. Specifically, LDAP can be used to manage passwords and password policies as well as denying access to logon. LDAP cannot be used to control access to specific database-level objects however, this is still accomplished using the built-in PostgreSQL RBAC.

In order for an LDAP user to have access to a postgres instance, the user must have a role installed by the same login attribute (typically uid= or cn=) that LDAP is using. In this scenario, the RDBMS would make a call to the LDAP system for authentication.

Kerberos Authentication

Kerberos authentication is available in PostgreSQL, as with LDAP, authorization is not. This means that the role names used in the kerberos infrastructure will also need to exist in the postgres database. The RDBMS would call to the ticketing system to ascertain authentication.

Best Practices

1. For stringent SLAs and a low number of roles, use internal authentication
   ○ When using external systems, the uptime of your database becomes tied directly to the external authentication system in use.
2. For less stringent SLAs, use external authentication
3. When data analysis of high-SLA systems is required, requiring many logon roles
   ○ Use ETL / replication strategies to bring data out of critical-path systems for the SLA and into less critical systems that use external authentication.
   ○ Provide frequent snapshots of data in less critical systems using external authentication

Host Based Access

The PostgreSQL host based access control system (pg_hba) provides a method to perform pre-authentication reject / accept of requests based on a set of firewall-like rules. This allows for protection of the database server above and beyond RBAC and any authentication methods.
Best Practices

1. Remove all 'trust' lines from pg_hba.conf
2. Add per-host lines for each application host in use
   - If this is becomes unwieldy, using per-subnet rules is possible
3. Allow only necessary users to have network access from necessary network segments
4. Allow users and network segments to connect only to the databases that they need access to, avoid using 'all' whenever possible
5. Use md5 as the method for comparing the hashes across the wire
6. Add a default 'all all 0.0.0.0/0 REJECT' rule to the bottom of the file
   - Since pg_hba.conf is processed top to bottom, the first rule that matches is the rule that is used. If a reject rule is set at the bottom, this covers all non-defined traffic

Data Layout

PostgreSQL provides both physical and logical separation of data.

Physical Separation

In a hierarchical PostgreSQL system, you have an database instance. This instance will be comprised of one or more physical databases.

Whenever logging into a PostgreSQL instance, you are logged into a specific database, for the life of that session, you only have physical access to the data inside of that database. A role may or may not have logon privileges to multiple databases within an instance, but once logged in to a database, the user will be locked there for the life of the session. In order to switch connections, the user must disconnect from a physical database and reconnect to another. These connections are controlled both by RBAC and pg_hba.

The lines between these physical databases cannot be blurred, in other words, if a database instance contains two databases, dbA and dbB:
Security Hardening PostgreSQL – Scott Mead

Database: dbA
Schema:   bi
Table:       reporting_customer_a

Database: dbB
Schema :   bi
Table:       reporting_customer_b

The data in these two tables **cannot** be seen simultaneously, even if the role has access to both systems. There is a physical layer between the two databases that cannot be circumvented unless data is physically copied from one database to another.

**Best Practices**

1. Any data that needs to be kept apart should be separated into different physical databases
   - This is only recommended where the two types of data will not need to be seen as a singular unit (i.e. SQL joins)

2. `pg_hba` and RBAC should be used to control access to these physically separate databases

**Logical Separation**

PostgreSQL allows for logical separation of data at the 'Schema' layer within a physical database. Schemas provide for logical groupings of relations (tables, sequences, indexes, etc...). Schemas can also be used in conjunction with RBAC to protect access to that logical grouping of tables when required. Data can be joined between schemas without any special modifications, the role in use only need have access to the schema and relation in order to perform the select.

**Best Practices**

1. Data should be separated logically into different schemas based on application access patterns to the data.

2. If multiple copies of the same datamodel need to be installed within a physical database, but require that they be 'join-ed' between, create these in two distinct schemas in the same database, and do the following:
   - RoleA: Has access to schema A and all relations stored within, does not have access to schema B.
Security Hardening PostgreSQL – Scott Mead

- RoleB: Has access to schema B and all relations stored within, does not have access to schema A
- RoleReporting: Has access to both schema A and B, and only has select privileges on the 4 tables that are required to fulfill the reporting requirement.

Database hosts

Even with proper database-level access / authorization control, access to the physical systems hosting the database must be restricted.

Operating system

PostgreSQL has a security requirement when running on a system. It may not run as the root user, it must have a non-privileged account that owns all running database processes. This account must have full access to the data-directory on disk, but the data directory must be locked down to the point of allowing ONLY the postgres server user access to it. In other words, the data directory is only viewable by super-users on the machine and the postgres user. No group based access may be used to the file system. If the permissions are not set in this way, the database engine will refuse to start.

Best Practices

1. Allow only required personnel to have host login rights
2. Provide 'postgres' service account only to those who need access to control the postgres processes on the system, and only after they have authenticated to the box using their own account (i.e. 'sudo').
3. Allow a limited set of ports to have network access to the database server
   - The database port (only from known hosts)
   - Any required management ports
   - Lock down all other network access wherever possible
4. Ensure pg_hba allows only the required database accounts have database login from the local (unix-socket) system.

Physical Access

Physical access to database hosts need to be tightly controlled. Each organization will have different capabilities in this regard and should follow any procedures that are in-place.