Security Best Practices for Postgres

June 13, 2016

EnterpriseDB Corporation, 34 Crosby Drive, Suite 201, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 589 5701 **E** info@enterprisedb.com www.enterprisedb.com

Table of Contents

1 Executive Summary

This white paper presents a framework and a series of recommendations to secure and protect a Postgres database. We discuss a layered security model that addresses physical security, network security, host access control, database access management, and data security. While all of these aspects are equally important, the paper focuses on Postgres-specific aspects of securing the database and the data.

For our discussion of the specific security aspects relating to the database and the data managed in the database, we use an AAA (*Authentication, Authorization, and Auditing*) approach common to computer and network security.

Most of the recommendations in this paper are applicable to PostgreSQL (the community edition) and to EDB Postgres™ Advanced Server (Advanced Server), the enterprise-class, feature-rich commercial distribution of PostgreSQL from EnterpriseDB® (EDB™). Advanced Server provides additional relevant security enhancements, such as `edb_audit`, SQL/Protect and Virtual Private Database (VPD) that are not available in the same form in PostgreSQL.

In this paper, we use the following conventions:

- PostgreSQL refers specifically to the community edition of Postgres.
- EDB Postgres Advanced Server (Advanced Server) refers to EDB's powerful feature-rich commercial distribution of PostgreSQL.
- Postgres refers to PostgreSQL, the Community Edition, and to EDB Postgres Advanced Server.

1.1 Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a *term* refers to any word or group of words that are language keywords, user-supplied values, literals, etc. A term's exact meaning depends on the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- `Fixed-width (mono-spaced) font` is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, `SELECT * FROM emp`;
- *`Italic fixed-width font`* is used for terms for which the user must substitute values in actual usage. For example, `DELETE FROM` *`table_name`*;
- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
- Square brackets [ ] denote that one or none of the enclosed terms may be substituted. For example, `[ a | b ]` means choose one of "`a`" or "`b`" or neither of the two.
- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, `{ a | b }` means exactly one of "`a`" or "`b`" must be specified.
- Ellipses . . . denote that the preceding term may be repeated. For example, `[ a | b ]...` means that you may have the sequence, "`b a a b a`".

2 Introduction

We can think of security in layers, and advise a strategy of granting the least access necessary for any job or role, blocking unnecessary access at the earliest opportunity.

- First, and perhaps most important, is to secure physical access to the host.
- Next is to limit access to your corporate network in general.
- Next is to limit access to the database host.
- Next is to limit access to the database application.
- Next is to limit access to the data contained within.

In this white paper, we will discuss the last two items: limiting access to the database, and to the data. While physical, network, and host system security are extremely important to the security of your data, they are beyond the scope of this paper.

General Recommendations

- Keep your system and your database patched. EDB's support subscriptions provide timely notifications of security updates and appropriate patches for Postgres.
- Don't put a postmaster port on the internet, unless it is truly vital to your business. Firewall this port appropriately; if that's not possible, make a read-only standby database available on the port, instead of a read-write master. Network port forwarding with auditing of all connections is a valid alternative.
- Isolate the database port from other network traffic.
- Grant users the minimum access they require to do their work, nothing more; reserve the use of superuser accounts for tasks or roles where it is absolutely required.
- Restrict access to configuration files (`postgresql.conf` and `pg_hba.conf`) and log files (`pg_log`) to administrators.
- Disallow host system login by the database superuser roles (`postgres` on PostgreSQL, `enterprisedb` on Advanced Server).
- Provide each user with their own login; shared credentials are not a recommended practice and they make auditing more complicated. Alternatively, use the `edb_audit_tag` capability (available in EDB Postgres Advanced Server only) to allow applications to add more audit information to sessions resulting from application-level connections.

- Don’t rely solely on your front-end application to prevent unauthorized access to your database; integrate database security with enterprise level authentication and authorization models, such as `LDAP/AD` or `Kerberos`.
- Keep backups, and have a tested recovery plan. No matter how well you secure your system, it is still possible for an intruder to get in and delete or modify your data.

It may be helpful to think of security in terms of the AAA model developed for network and computer security. AAA stands for *Authentication, Authorization,* and *Auditing*.

- *Authentication:* verify that the user is who he or she claims to be.
- *Authorization:* verify that the user is allowed access.
- *Auditing (or Accounting):* record all database activity, including the user name and the time in the log files.

Not all features fit neatly into these categories, but the AAA model offers a useful framework for this discussion.

3 Applying Postgres Security Features to the AAA Framework

3.1 Authentication

The `pg_hba.conf` (Postgres host-based access) file restricts access based on user name, database, and source IP (if the user is connecting via TCP/IP). Authentication methods are assigned in this file as well. The authentication method (or methods) you choose depends on your use case.

SSPI

> Use this if you are on a Windows system and would like to implement Single Sign-On (SSO) authentication.

LDAP and RADIUS

> `ldap` and `radius` are useful in situations where you have large numbers of users and need to manage passwords from a central location. This centralization has the advantage of keeping your `pg_hba.conf` file small and more manageable, and gives your users a "unified password experience" across your infrastructure. They will appreciate that. Both `LDAP` and `RADIUS` require solid infrastructure, as you are relying on the service and connectivity to that service to access your database.

Kerberos

> Postgres supports `GSSAPI` with `Kerberos` authentication according to RFC 1964. `GSSAPI` provides automatic authentication (single sign-on) for systems that support it. The authentication itself is secure, but data sent over the database connection is unencrypted unless `SSL` is in use.

md5

> If you have a very small number of trusted users, you may want to use one of the built-in authentication methods instead. Generally, md5 is the preferred option as users connect with hashed passwords; `password` passes the credentials in clear text.

reject

> Use this method to reject specific users, connections to specific databases, and/or specific source IPs.

It's imperative that you have a full understanding of the ramifications of each authentication method. See the Postgres documentation for a more detailed study of these and other authentication methods:

https://www.postgresql.org/docs/current/static/auth-methods.html

As mentioned in the Introduction, access to the `pg_hba.conf` file should be restricted to administrators. Try to keep this file properly pruned; larger, more complicated files are harder to maintain and more likely to contain incorrect or outdated entries. Review this file periodically for unnecessary entries.

3.2 Authorization

Once the user has been properly authenticated, you must grant permissions to view data and perform work in the database. As previously advised, grant only those privileges required for a user to perform a job and disallow shared (group) login credentials.

Manage users and groups in Postgres via role assignments. A role may refer to an individual user or a group of users. In Postgres, roles are created at the cluster (database server) level. This means roles are applied to all databases defined for the cluster/database server; it is very important to limit role permissions appropriately. Accomplish this by careful assignment of roles and privileges, and by restricting access in the `pg_hba.conf` file. Assigned privileges and caveats are outlined in the Postgres `CREATE ROLE` documentation:

https://www.postgresql.org/docs/9.5/static/sql-createrole.html

Revoke `CREATE` privileges from all users and grant them back to trusted users only.

Don't allow the use of functions or triggers written in untrusted procedural languages.

`SECURITY DEFINER` functions allow users to run functions at an elevated privilege level in a controlled way, but a carelessly written function can inadvertantly reduce security. Review the documentation (section *Writing Security Definer Functions Safely* of `CREATE FUNCTION`) for more details:

https://www.postgresql.org/docs/9.5/static/sql-createfunction.html

Objects should not be owned by a superuser unless absolutely necessary. Be especially diligent about this for `SECURITY DEFINER` functions.

Be aware that when log_statement is set to `'ddl'` or higher, changing a role's password via the `ALTER ROLE` command will result in password exposure in the logs.

When storing authentication in clear text in a table, use of statement logging can expose that information, even if the table is nominally secure. Similarly, if sensitive information is used in queries (such as a SSN or TIN as a key); those parameters can be exposed by statement logging.

3.3 Auditing

Advanced Server provides the capability to produce audit reports. Database auditing allows database administrators, auditors, and operators to track and analyze database activities in support of complex auditing requirements. These audited activities include database access and usage along with data creation, change, or deletion. The auditing system is based on configuration parameters defined in the configuration file.

We recommend that you audit, (listed by increasing level of scrutiny):

- User connections
- DDL changes
- Data changes
- Data views

Highly detailed levels of scrutiny can result in a lot of log messages; log only at the level you need. With Postgres, you can adjust logging levels on a per-user and per-database basis; that is an option if required.

Review your audit logs frequently for anomalous behavior. Establish a chain of custody for your logs.

3.4 Data Security

Additionally, there's the question of security of the data itself: preventing intentional or unintentional data views by users. The simplest way to limit visibility of data to certain groups of users is by creating a `VIEW` of a table and limiting permissions for that `VIEW`. Users can find a way around a restriction; Postgres versions 9.2 and higher provide the option to `CREATE VIEW WITH` (*`security_barrier`*), if extra precaution is deemed necessary.

If you are using Advanced Server version 9.1 or higher, we recommend you use our `VPD` (Virtual Private Database) feature to further limit a user's access to data.

Postgres offers encryption at several levels, and provides flexibility in protecting data from disclosure due to database server theft, unscrupulous administrators, and insecure networks:

- Password storage encryption
- Encryption for specific columns
- Data partition encryption
- Encrypting passwords across a network
- Encrypting data across a network
- SSL host authentication
- Client-side encryption

You can read more about these options in the Postgres documentation:

https://www.postgresql.org/docs/9.5/static/encryption-options.html

If you are concerned about data being sniffed during transfer between a client and the database, enable SSL in the `postgresql.conf` file. Don't do this unless absolutely necessary; it adds some overhead, and certificate management can be a bit tricky.

You can also encrypt data within the database, or at the filesystem level (one or the other). See more about Transparent Data Encryption for Postgres on EDB's blog. With this encryption option, the data is decrypted as it is read from the filesystem, so DBAs can view data; it's imperative to have roles and privileges locked down.

Use the `pgcrypto contrib` module to encrypt data on a per-column basis. There are a few drawbacks to this method:

- There's a potential performance hit, depending on the size of the table.
- The encrypted fields can't be searched or indexed.
- The encryption must be applied at table creation time, requiring advanced planning.

Additionally, your application must handle the encryption/decryption so that each exchange with the database remains encrypted to prevent an unscrupulous DBA from viewing data.

3.5 SQL Injection Attacks

A SQL injection attack is an attempt to compromise a database by running SQL statements the results of which provide clues to the attacker as to the content, structure, or security of that database. Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

The standard method to prevent SQL injection attacks is to use parameterized queries. However, versions of Postgres prior to 9.2 used a generic query plan for parameterized queries, which can cause performance problems. If you are using an earlier version, benchmark your system to find out if this is the correct solution for you. A good alternative is to use Postgres' built-in quoting functions. We do not recommend writing your own quoting functions.

If you are using Advanced Server, we recommend you use the EDB Postgres SQL/Protect module to protect against SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL profiles. SQL/Protect gives control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

4 EDB Postgres Advanced Server Security Features

Advanced Server provides several additional security features beyond those available with community edition PostgreSQL.

4.1 Auditing

The `edb_audit` configuration parameters in the `postgresql.conf` file allow you to easily control logging of connection attempts and record DDL or DML changes. Events are logged in `.csv` or `.xml` format for easy parsing or storing in a database for later perusal. See *Section 2.2 "Controlling the Audit Logs"* of the *EDB Postgres Advanced Server Guide* for details, available at:

http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition

4.2 SQL/Protect

SQL/Protect protects against SQL Injection attacks and records attack attempts. Once SQL/Protect has learned which queries are acceptable, you can run the utility in passive mode, to be warned of potential attacks, or active mode, which blocks potential injection attacks. SQL/Protect also maintains statistics about suspicious queries for later review by an administrator. See *Section 3 "Security"* of the *EDB Postgres Advanced Server Guide* for details (link above). Please note that SQL/Protect is also available for community edition PostgreSQL as part of EDB Postgres Standard.

4.3 Virtual Private Database (VPD)

Since version 9.1, Advanced Server also supports Virtual Private Database (VPD) technology to provide row-level security: you can apply a security policy to a table, and the policy will prevent users from viewing or altering rows on which they don't have permissions. This functionality is included in the EDB DBMS_RLS package. See *Section 7.7* of *EDB's Database Compatibility for Oracle® Developer's Guide* for details, and this blog article for an example of VPD use.

4.4 Password Profiles

Starting with version 9.5, Advanced Server supports *password profiles*. A password profile is a named set of password attributes that allow a DBA to easily manage a group of roles that share comparable authentication requirements. Each profile can be associated with one or more users. When a user connects to the server, the server enforces the profile that is associated with the login role. Profiles can be used to:

- specify the number of allowable failed login attempts
- lock an account due to excessive failed login attempts
- mark a password for expiration
- define a grace period after a password expiration
- define rules for password complexity; and
- define rules that limit password re-use.

See *Section 2.3 "Profile Management"* of *EDB's Database Compatibility for Oracle® Developer's Guide* for more information, available at:

http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition