PostgreSQL Security Best Practices

—

Whitepaper

2ndQuadrant®
PostgreSQL

PostgreSQL Security Best Practices

Introduction

The following practices are well proven in the field. We present them as an introduction into the minimum concepts that are needed for any system. This is not an exhaustive list. It is provided as a compilation of field experience related to security, and avoids the most common pitfalls of security configuration.

- Keep your system updated to the latest minor version. The PGDG provides interim security patches as part of the minor release cycle.
- Do not expose PostgreSQL directly to the general internet on a public port.
- Disallow superuser login from ipv4 connections.
- Confine access to the configuration files for PostgreSQL to the system root user. (most distributions do this anyway, so don't circumvent it.)
- SCRAM
- Certificates

Contents

Connecting Securely

libPQ is a library that is provided by the PostgreSQL Global Development Group (PGDG). This library provides secure authentication methods and implements the communications protocol for PostgreSQL. Many different application development platforms provide wrappers around this library to provide secure communications to PostgreSQL.

Language	Library
Python	psycopg2
PHP	php-pg5
Ruby	pg
Perl	DBI
node.js	node-pg-native
...many...	...more...

The code wrappers which provide access to a library are called "bindings". You will find a binding for every language that is commonly available for PostgreSQL. Many of them are listed by the PGDG here:https://wiki.postgresql.org/wiki/List_of_drivers.

libPQ provides secure file access for local storage of a password or SSL certificate access. All of the languages that use libpq via local bindings inherit this ability, as it is done at the library level.

Providing a plain text password on the command line is heavily discouraged.

Passphrase Strength

Let's start with a basic understanding of password strength in the face of a brute force attack. We'll start with just using the 26 characters of the English alphabet, and the passphrase is only one character long. Of course this is not a passphrase by anybody's definition at this point, but we need to start calculating somewhere.

This would give us 26 possible passphrases. Obviously not very secure, and easily guessable within a few milliseconds. So, we'll add another character to the passphrase. This would make 26 * 26 possible passphrases, which is 676. Still not very many possibilities, and probably still less than one second of guessing using even very modest hardware. But we see a simple trend here. For each character of passphrase that we add, the complexity goes up by an exponent. By adding a third character, we get 26*2*626 possible passphrases, which is 17576.

So we see that the complexity relationship in the passphrase is related to the number of possible characters in the phrase, exponentiated by the length of the phrase. If we add capital letters to the mix, we now have 52 characters instead of 26, so the same 3 letter passphrase has 140,608 permutations.

Our original example was 26^3 for a total of 17576, and now with a larger character set, it is 52^3, which is 140608. Imagine that we add numbers to this, and we get an additional 10 characters, $62^3 = 238328$. So by increasing the character set (entropy), we get a complexity multiplier (ratiometric), but when we add a character to the length of the password, we get an exponential rise in complexity.

This means that a good passphrase is a combination of the largest character set that we can reasonably come up with on a standard keyboard (or even some other characters, if you want to be really secure), and a length that represents something close to the bit strength of the algorithm that we are using to encrypt/decrypt the messages.

In public key cryptography, the ideal strength is one where the password is as strong as the key that we are securing with it. Any further strength is wasted, as it would become more cryptographically expedient to attack the key than the password. The formula for that is $key\_size\_in\_bits/8\ \ bytes/entropy$. For a 4096 bit key, this would be $4096/8/62 = 8.25$. So, at this point, a nine character/number password would be about as strong as what it's protecting.

PostgreSQL, on the other hand, is not a public key cryptography system. The passphrase itself is symmetric, so the longer it is, the more secure it is. The upper limit of a password storage in PostgreSQL is a text column (2^{31}) bytes. Of course, this is after SCRAM encoding (you **are** using encrypted passwords, right?). The practical limit, of course, is a lot lower, and has more to do with the amount of time you are willing to sacrifice to encryption and decryption during a service call. This should stay somewhere in balance with the length of time it takes to perform an "average" query for your application needs. If you are deploying an OLTP solution where the average database operation takes a few milliseconds, a passphrase of 20 or so bytes may be a reasonable compromise. For a data warehouse or operational data store, 100 characters or so is probably more appropriate. It is not unusual in very well tuned systems for the cryptographic handshake to take 5 to 10 times longer than the actual data service request. Your performance **will** vary, based on available hardware and application needs.

These suggested numbers are not absolutes. They are experience values in 2019, and should be treated as such. They are here to help the beginner to intermediate security administrator get a "feel" for what is reasonable at the moment, and adjust accordingly in the future.

Trying to concatenate simple dictionary words together in order to create a longer passphrase actually reduces the passphrase strength. This method reduces the randomness of the passphrase, so the brute force attack does not have to go through all of the permutations of the characters, but simply through all of the permutations of words in a common usage dictionary. This approach can be strengthened quite dramatically by adding symbols and numbers at random locations in the phrase.

Consider for a moment how easily guessable this passphrase would be with dictionary attack:

"beam me up"

But with a few simple changes, it becomes more of a random guessing game:

"Scotty#, you stink1 at beaming 8 an9body anywhere."

Because of the addition of punctuation, capitalization, numbers and symbols, the strength of this phrase is much higher. Also, the spacing changes, and the placement of the numerals is somewhat random. Additionally, we added a word that is not in the common English dictionary (albeit a rather weak one). These changes in combination make a dictionary attack improbable, enforcing a true brute force approach.

Common phrases or quotes fall into the same category of general weakness, and in fact may be weaker than much shorter lengths of random characters. A good phrase dictionary may have a few hundred thousand common quotes, and probably much less.

Here is a simple bash one-liner that generates cryptographically reasonable passwords.

```
tr -dc 'A-Za-z0-9!@#$%^&*()' < /dev/urandom | head -c 32 #<--however long you want it.
```

This method generates gibberish with letters, numbers, capitalization and some symbols. This would be 72^{32} bits long, which is fairly strong.

There are many more variations of command line password generators. Here are a few examples:

```
# Fairly safe to quote
openssl rand -hex 32
# A bit more readable
openssl rand -base64 32
```

Of course, if you would like something a bit more pronouncable, a dictionary with about 125,000 words and a diceware password generator are supplied with this article.

- Passphrase Recommendations

Sanitizing Inputs

The PostgreSQL Global Development Group is highly aware of this application level issue, and provides a solution that can be easily integrated into all development systems. These are in the form of functions called format() , quote_literal(), and quote_ident().

These functions wrap inputs in appropriate quotation marks and escape sequences, so that SQL injection is not possible. The "embedded" SQL will simply be transformed into harmless parameter text.

Consider the following function:

```
CREATE OR REPLACE FUNCTION get_stock_item(match text, criteria text) RETURNING
article
AS $$
DECLARE query TEXT := 'SELECT * FROM article WHERE stock > 0';
BEGIN
   IF match IS NOT NULL THEN
      query := query || ' AND ' || match || ' = $a$' || criteria || '$a$;';
   END IF;
   RETURN QUERY EXECUTE query;
END;
$$ LANGUAGE plpgSQL
STRICT
VOLATILE;
```

This function is vulnerable to SQL injection for both the column name and the criteria. To rewrite it better is fairly simple:

```
CREATE OR REPLACE FUNCTION get_stock_item(match text, criteria text) RETURNING
article
AS $$
DECLARE query TEXT := 'SELECT * FROM article WHERE stock > 0';
BEGIN
   IF match IS NOT NULL THEN
      query := query || ' AND ' || quote_ident(match) || ' = ' || quote_literal(criteria) || ';';
   END IF;
   RETURN QUERY EXECUTE query;
END;
$$ LANGUAGE plpgSQL
STRICT
VOLATILE;
```

This also has the added benefit of not dealing with nested quotes in the literal criteria.

Parameterized Queries

PostgreSQL also provides a way to sanitize inputs via parameterized queries. These queries are planned when "prepared", and the plan is cached by libpq. In the PHP documentation there is an example:

```
<?php
// Connect to a database named "mary"
$dbconn = pg_connect("dbname=mary");

// Prepare a query for execution
$result = pg_prepare($dbconn, "my_query", 'SELECT * FROM shops WHERE name = $1');
```

```
// Execute the prepared query.  Note that it is not necessary to escape
// the string "Joe's Widgets" in any way
$result = pg_execute($dbconn, "my_query", array("Joe's Widgets"));

// Execute the same prepared query, this time with a different parameter
$result = pg_execute($dbconn, "my_query", array("Clothes Clothes Clothes"));

?>
```

This does more than just take care of quotation issues in the input. It also internally calls quote_literal() function, which will effectively strip any embedded SQL commands from the input.

There is a variation of this parameterization in virtually every language that implements the libpq API for PostgreSQL access, and will be exposed to the developer as a set of data access functions.

Principle of least privilege

It is tempting as a beginner, or with a new system, to make a single user with open privileges, and use that role for everything. The general idea is that when "we go to production," the security will be tightened up at deployment time. The issue with this assumption is that it is very hard to disentangle the required permissions and roles at that point. More often than not, this leads to very lazy security practices in the field.

Let's turn that concept on its' head. Instead of granting universal rights to a single user, we'll start with a concept of 5 roles. These roles are:

1. Maintenance operations
2. Structural changes
3. Data manipulation
4. Application administrator
5. Extract, Transform and Load
6. Read-only

Many more finely grained roles are possible, of course. These roles are experience values that have held up well in practice. They are a balance of general needs of the application and administrative needs of the organization. They are also a much better starting point for further role refinement if the organization chooses to move in that direction.

These roles tend to grant the minimum privileges necessary for any given role, rather than the maximum. This is called the principle of least privilege. Grant only what is necessary for a given operation, and decide whether roles can or should be combined for simplicity later.

Maintenance operations

This user should have a fairly narrow task of refreshing materialized views, vacuuming and reindexing operations. These tasks typically take a fairly large amount of memory. One of the advantages of this role is that a large maintenance_work_mem setting can be allocated for these operations, and restricted to this specialized user. This allows for a very low maintenance_work_mem per application or data manipulation user, freeing up memory that may be used as work_mem, or enabling more connections to the database via max_connections.

Structural Changes

This user is restricted to CREATE and ALTER statements. Typically, this user will also have the ability to INSERT data, in order to perform structural migration tasks. These statements are a subset of data definition language (DDL) commands. Only the ones strictly required for schema migration need to be allowed.

This includes CREATE EXTENSION and other commands that provide extended structure.

Data Manipulation

This user has the ability to use INSERT, UPDATE, and DELETE statements. This is typically an application level user, the abilities of which are limited by the application operation as well as the database constraints. This is also the category that is most likely to be sub-divided in the future into even less privileged roles on a per object basis.

It is certainly not a superuser, and should not have replication, TRUNCATE , or COPY privileges.

Application administrator

This user typically has TRUNCATE, COPY and possibly other DML privileges required for seeding lookup tables, performing data retention tasks such as partitioning operations, and bulk operations for application maintenance. This role is similar in privileges to the ETL user outlined below, but is intended for tasks which require knowledge of the application architecture.

Extract, Transform and Load

Our final user is restricted to bulk operations. Typically, these operations include exporting data to reporting servers and warehouses, as well as importing data for enrichment such as maps, reviews, or finances. For the most part, this is the function of the COPY command.

Read-only

Some authorized users may not need to make any modifications at all. Creating a role that does not allow for modifications, and use it to prevent the connection from being able to take any malicious action.

Host Based Authentication

PostgreSQL determines the login mechanism based on a rule set established in the pg_hba.conf file. This rule set includes 4 identifiers for the connection, the authentication mechanism and parameters for the authentication. A typical line of the form:

```
Host    postgres    all     127.0.0.1/32    SCRAM-SHA-256
```

would require that users wishing to connect over TCP/IP on the local machine would use the SCRAM authentication method.

These rules are evaluated in the order in which they appear in the file. The first rule that matches the 4 criteria is used to determine the authentication method. Any number of authentication methods may be configured. If the request does not match any of the provided methods, the connection will simply be rejected.

The evaluation of the rules in a determined order allows the principle of least privilege to be applied to the connection authentication process. Consider for a moment the following rules in order:

```
Local   postgres    all                 peer
Host    postgres    all     0.0.0.0/8       reject
Local   all         all                 md5
Host    all         all     127.0.0.1/32    reject
Host    all         all     0.0.0.0/8       SCRAM-SHA-256
```

This rule set determines that the postgres user can only authenticate from a local connection over unix sockets on the same machine as a unix user of the same name. Furthermore, postgres will then be forced to use SCRAM authentication to the PostgreSQL service. This practice goes a long way in preventing brute force attacks on the superuser account.

Additionally, any other user of the local machine is also forced to use unix sockets to connect, and will be denied TCP/IP access from the local machine. This improves concurrency efficiency a bit, while at the same time providing an authentication mechanism for automated tasks. In this case, we are relying on the host security as well as PostgreSQL security, so the automation can be more efficient, and require a bit less resources on the part of the PostgreSQL server authentication system.

Finally, we have a rule for all other users that are arriving from external TCP/IP origination points. The most secure method that PostgreSQL offers internally for these

direct connections is SCRAM-SHA-256. Additional methods may include kerberos, ldap, active directory, pam, radius and a host of other methods that can be plugged into the pluggable authentication module (PAM) or integrated into PostgreSQL directly.

Internal methods the PostgreSQL provides directly are described in the documentation.

Convenience vs. Security (Roles)

The roles that were outlined in the previous section represent a model of the **minimum** number of roles that should be created for even the most simplistic installation. The principle of least privilege typically demands several more. This flies in the face of the natural human tendency to simplify for the sake of convenient understanding. In this arena, simplification tends toward intrusion.

Resist the natural urge to simplify security roles. Rather, find the time to get comfortable with the added complexity, and learn the security features of the product well. We will be assisting in that endeavor by publishing more articles and books to clarify the rules.

Your system will ultimately only be as secure as your understanding of the underlying principles.

About 2ndQuadrant

2ndQuadrant provides solutions for mission critical PostgreSQL databases. Our solutions ensure high availability, disaster recovery, backup & failover management & more.

We provide tools to make your deployment process simple and reliable. 2ndQuadrant is a member of the Cloud Native Computing Foundation (CNCF) and the first Kubernetes Certified Service Provider (KCSP) for PostgreSQL. With over 19 years of hands-on experience through client engagements, our engineers ensure your PostgreSQL solutions are fully supported in their production environment.