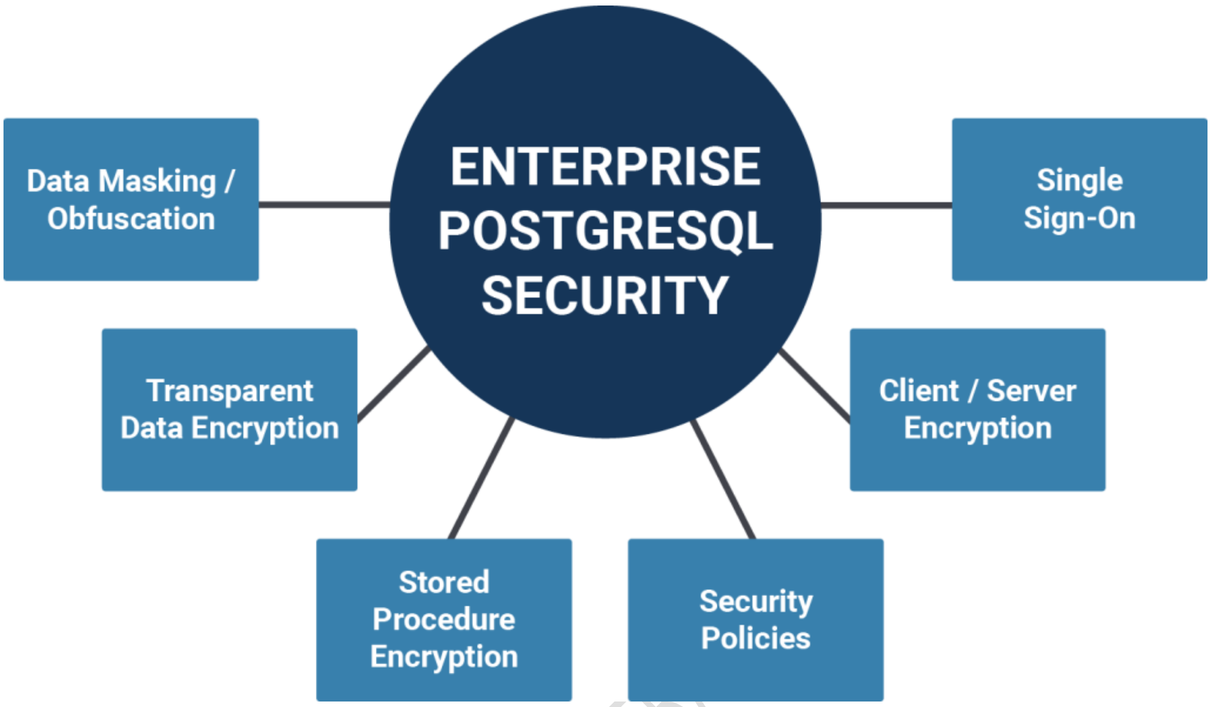
Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR

DAY – 7

TOPIC: Postgres Security Hardening and Role Permissions


ENTERPRISE POSTGRESQL SECURITY
Data Masking / Obfuscation
Single Sign-On
Transparent Data Encryption
Client / Server Encryption
Stored Procedure Encryption
Security Policies

PostgreSQL security hardening and role permissions are crucial for protecting a database from unauthorized access and data breaches. This involves a multi-layered approach that includes network security, database configuration, data security, and fine-grained access control. 🛡️

Purpose of PostgreSQL Security

The primary goal of PostgreSQL security is to ensure **confidentiality**, **integrity**, and **availability** of data.

- **Confidentiality** prevents unauthorized disclosure of information.
- **Integrity** ensures data is accurate and not tampered with.
- **Availability** guarantees the database is accessible to authorized users when needed.

This is achieved by implementing the **principle of least privilege**, which means users and applications are granted only the minimum permissions required to perform their tasks.

Network Security

Network security focuses on controlling who can connect to the PostgreSQL server and how they connect.

pg_hba.conf

The pg_hba.conf file is PostgreSQL's primary configuration file for client authentication. It controls which hosts can connect, which users they can connect as, and the authentication method used.

- **Purpose:** To define a policy for client connections.
- **Methods:** You specify connection entries with fields like type, database, user, address, and method.
  - **type**: Specifies the connection type (e.g., host for TCP/IP, local for Unix domain sockets).
  - **address**: The IP address or range of clients allowed to connect.
  - **method**: The authentication method (e.g., md5 for password-based, trust for no password, cert for SSL certificates).
- **Checking with psql**:
  - You can't directly check the file's contents with psql. You must access the file on the server.
  - However, you can check active connections and their details with:

```
SELECT * FROM pg_stat_activity;
```

- **Checking with a GUI (like pgAdmin)**:
  - In pgAdmin, navigate to the **Tools** menu and select **Server Configuration**. From there, you can view and edit the pg_hba.conf file.

SSL/TLS Encryption

- **Purpose:** To encrypt data transmitted between the client and the server, protecting against eavesdropping.
- **Methods:** Configure ssl = on in postgresql.conf and provide the path to your server's certificate and key files.
- **Checking with psql**:
  - When connecting, use sslmode=require in your connection string to force an SSL connection.

- You can verify the connection status with:

SELECT ssl_is_used() FROM pg_stat_activity WHERE pid = pg_backend_pid();

- **Checking with a GUI**:
  - In pgAdmin, when creating a new server connection, you can specify the SSL mode in the connection details.

Core Concepts

Before we dive into the setup, let's establish a clear understanding of the core concepts underpinning PostgreSQL SSL authentication:

1. **SSL/TLS**: Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS), are cryptographic protocols that secure communication over a network. They use certificates and encryption keys to ensure that data transmitted between the PostgreSQL server and client remains confidential and tamper-proof.
2. **Server Certificates**: A server certificate is a digital document that verifies the identity of the PostgreSQL server. It acts like a digital passport, containing information such as the server's hostname and public key. Clients use this certificate to authenticate the server and establish a secure connection.
3. **Client Certificates (Optional)**: In addition to server certificates, PostgreSQL can also use client certificates to authenticate clients connecting to the database. This mutual authentication adds another layer of security by verifying the identity of both parties involved in the communication.
4. **Configuration Files**: PostgreSQL stores SSL/TLS settings in specific configuration files:
   - **postgresql.conf**: This file contains general PostgreSQL server settings, including whether SSL is enabled.
   - **pg_hba.conf**: The "Host Based Authentication" file defines which clients can connect to which databases and how they should authenticate. Here, you specify the SSL requirements for different clients and databases.

To securely enable SSL/TLS encryption for PostgreSQL, you must generate certificates, configure the server, and optionally configure client authentication. Here's a detailed, practical workflow, including GUI hints and psql commands:

1. Generate SSL/TLS Certificates

a. Generate a Self-Signed Server Certificate (CLI with OpenSSL)

For demonstration or internal tools, you can create a self-signed certificate. In production, use a trusted CA.

```
openssl req -new -x509 -days 365 -nodes -text \
  -out server.crt -keyout server.key \
  -subj "/CN=your-db-hostname"
```

- server.crt: Server certificate.
- server.key: Server private key.
  Permissions: Secure the private key.
- Then do:

```
chmod og-rwx server.key
chmod 600 server.key
```

Best practice: The CN (Common Name) must match your PostgreSQL server's hostname.

b. (Optional) Create and Use a Root CA Certificate

For mutual authentication or larger enterprises, create a root CA and sign both server and client certificates:

- root.crt — trusted root certificate.
- Use openssl to generate a root certificate and sign server.crt and client certificates.

To create a server certificate whose identity can be validated by clients, first create a certificate signing request (CSR) and a public/private key file:

```
openssl req -new -nodes -text -out root.csr \
 -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Then, sign the request with the key to create a root certificate authority (using the default OpenSSL configuration file location on Linux):

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

2. Place Certificates in PostgreSQL Data Directory

Copy server.crt and server.key (and optionally root.crt) to your PostgreSQL data directory (e.g., /var/lib/postgresql/data/ or as configured in your system).

Finally, create a server certificate signed by the new root certificate authority:

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

server.crt and server.key should be stored on the server, and root.crt should be stored on the client so the client can verify that the server's leaf certificate was signed by its trusted root certificate. root.key should be stored offline for use in creating future certificates.

It is also possible to create a chain of trust that includes intermediate certificates:

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

# intermediate

```
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt
```

# leaf

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

server.crt and intermediate.crt should be concatenated into a certificate file bundle and stored on the server. server.key should also be stored on the server. root.crt should be stored on the client so the client can verify that the server's leaf certificate was signed by a chain of certificates linked to its trusted root certificate. root.key and intermediate.key should be stored offline for use in creating future certificates.

3. PostgreSQL Server Configuration

a. Edit postgresql.conf

Set or verify these lines:

```
ssl = on

ssl_cert_file = 'server.crt'

ssl_key_file = 'server.key'

ssl_ca_file = 'root.crt'      # if using client verification
```

You can do this via:

- Terminal/text editor:

```
nano /var/lib/postgresql/data/postgresql.conf
```

- GUI tools: pgAdmin can be used to edit server config parameters: Navigate to your server > right-click > Properties > Parameters.

b. Edit pg_hba.conf

Control SSL enforcement and authentication. Example entries:

```
# Enforce SSL for local IPv4 clients using certificates

hostssl all all 127.0.0.1/32 cert clientcert=1

# Enforce SSL for a specific subnet, using password

hostssl all all 10.0.0.0/24 md5

# Allow only SSL connections from anywhere with password

hostssl all all 0.0.0.0/0 md5
```

Change host to hostssl for all connections you want to require SSL.

c. Reload/Restart PostgreSQL

Apply changes by restarting or reloading PostgreSQL:

```
sudo systemctl restart postgresql

# or reload

psql -c "SELECT pg_reload_conf();"
```

4. **Connect via SSL/TLS**

a. Using psql (CLI)

Typical connection string:

```
psql "host=your-db-host dbname=yourdb user=youruser sslmode=require"
```

Options:

- sslmode=disable|require|prefer|verify-ca|verify-full
- For stricter validation:

```
psql "host=your-db-host dbname=yourdb user=youruser \
 sslmode=verify-full \
 sslrootcert=root.crt \
 sslcert=client.crt \
 sslkey=client.key"
```

b. **Using GUI Tools (pgAdmin, DBeaver, DataGrip)**

- When adding a new connection, go to SSL or Security tab.
- Provide the path to server.crt, sslmode, and optional client and CA certificates.
- Example (pgAdmin):
  - Host: your-db-host
  - SSL Mode: require/verify-ca/verify-full
  - Client Cert/Key: Path to client.crt/client.key
  - Root Cert: Path to root.crt
- Save and test the connection.

5. **(Optional) Enforce SSL-Only**

To make SSL mandatory for all, remove any non-SSL host entries in pg_hba.conf and keep only hostssl. Reload the configuration after changes.

Notes and Tips:

- Always secure your server.key and client.key files (permissions: 600).
- The server certificate's CN should match the host clients use to connect.
- For production, certificates should be signed by a reputable CA.

- GUI tools make certificate entry easier but always verify the location and permissions of files specified.
- Most cloud-hosted PostgreSQL services allow SSL enforcement via dashboard/console settings.

This process enables you to set up PostgreSQL with robust SSL/TLS communication, securing all traffic between your server and clients—either using scripts (psql/CLI) or more visually in GUIs like pgAdmin. This protects passwords, queries, and data-in-transit from interception or tampering.

SSL Server File Usage

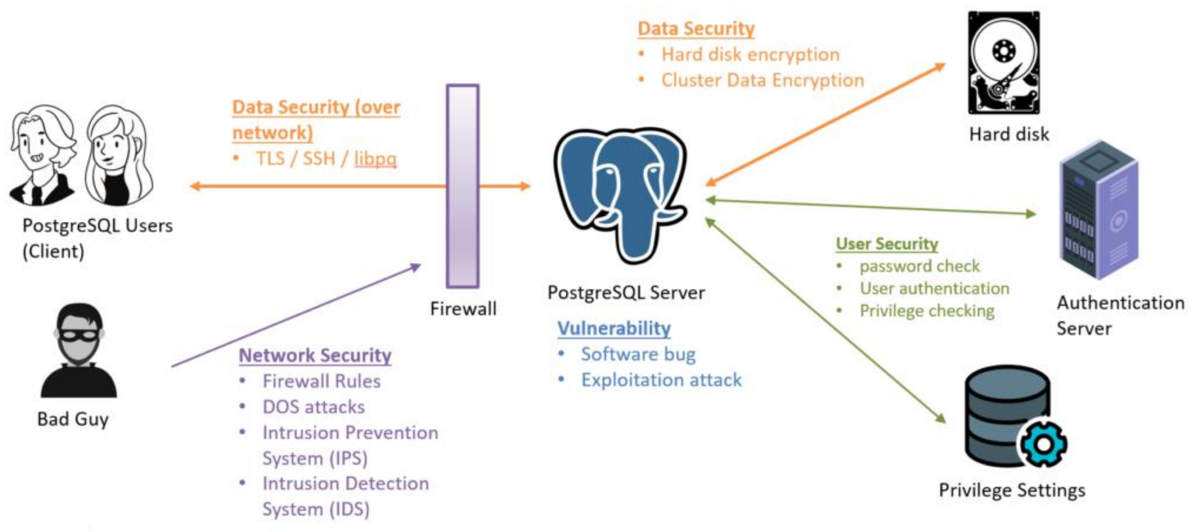
summarizes the files that are relevant to the SSL setup on the server. (The shown file names are default names. The locally configured names could be different.)

Table 18.2. SSL Server File Usage

File	Contents	Effect
ssl_cert_file ($PGDATA/server.crt)	server certificate	sent to client to indicate server's identity
ssl_key_file ($PGDATA/server.key)	server private key	proves server certificate was sent by the owner; does not indicate certificate owner is trustworthy
ssl_ca_file	trusted certificate authorities	checks that client certificate is signed by a trusted certificate authority
ssl_crl_file	certificates revoked by certificate authorities	client certificate must not be on this list

The server reads these files at server start and whenever the server configuration is reloaded. On Windows systems, they are also re-read whenever a new backend process is spawned for a new client connection.

If an error in these files is detected at server start, the server will refuse to start. But if an error is detected during a configuration reload, the files are ignored and the old SSL configuration continues to be used. On Windows systems, if an error in these files is detected at backend start, that backend will be unable to establish an SSL connection. In all these cases, the error condition is reported in the server log.



🔐 PostgreSQL SSL/TLS Authentication – Full Guide

Secure Sockets Layer (SSL) and its successor, Transport Layer Security (TLS), are essential for encrypting PostgreSQL connections to ensure **data-in-transit confidentiality and integrity**. This guide provides advanced techniques for secure deployments, focusing on both **server-side** and **client-side certificate enforcement**, along with **common pitfalls, practical application scenarios, and future trends.**

🔍 Common SSL/TLS Pitfalls in PostgreSQL (And How to Avoid Them)

1. Incorrect File Permissions

- **Problem**: PostgreSQL fails to start or accept SSL connections due to unreadable certificate files.
- **Cause**: The PostgreSQL service account (often postgres) lacks read permission on certificate/key files.
- **Solution**:

```
chown postgres:postgres server.key server.crt

chmod 600 server.key
chmod 644 server.crt
```

- server.key must be **600**, or PostgreSQL will **refuse to start** for security reasons.

---

2. Certificate Chain Mismatches

- **Problem**: Clients reject the server's certificate even though it is CA-signed.
- **Cause**: The server certificate is missing **intermediate certificates**.
- **Solution**:
  - Concatenate server certificate and intermediate(s) into server.crt:

```
cat your_domain.crt intermediate.crt > server.crt
```

  - Ensure the root CA is included on the client side in root.crt.

---

3. Improper pg_hba.conf Configuration

- **Problem**: Unintentional fallback to unencrypted or weaker authentication methods.
- **Cause**: host entries are used instead of hostssl, or sslmode=prefer on clients.
- **Solution**:
  - Use hostssl in pg_hba.conf to enforce SSL-only connections.
  - Example:

```
hostssl all all 0.0.0.0/0 md5
```

  - Set sslmode=require, verify-ca, or verify-full in clients.

---

4. Outdated OpenSSL Libraries

- **Problem**: PostgreSQL cannot use modern ciphers or is exposed to vulnerabilities.
- **Cause**: Legacy OpenSSL versions (e.g., <1.1.1).
- **Solution**:
  - Upgrade to a **maintained OpenSSL version** on both server and client:

```
openssl version
sudo apt-get upgrade openssl  # Debian-based
```

✅ Enforcing Client-Side SSL Authentication in PostgreSQL

Enforcing **mutual TLS (mTLS)** ensures **both server and client** must authenticate each other using digital certificates. This is stronger than just using SSL and passwords.

📘 Scenario: Centralized Analytics Platform

Imagine a shared PostgreSQL analytics environment. To prevent unauthorized access:

- Each analyst receives a **client certificate** issued by your internal or external CA.
- Access is granted only if the client presents a valid certificate.

🔧 Step-by-Step Implementation

Step 1: Enable SSL on Server

Edit postgresql.conf:

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
ssl_ca_file = 'root.crt'  # Optional but required for client verification
```

Restart PostgreSQL:

```
sudo systemctl restart postgresql
```

Step 2: Generate Client Certificates

Using your Certificate Authority (CA):

```
openssl genrsa -out client.key 2048
```

openssl req -new -key client.key -out client.csr -subj "/CN=analyst1"

openssl x509 -req -in client.csr -CA root.crt -CAkey root.key -CAcreateserial -out client.crt -days 365

Ensure:

- The Common Name (CN) matches the PostgreSQL username (analyst1).
- The client certificate is signed by the same CA used on the server.

Step 3: Configure pg_hba.conf for Client Authentication

Open pg_hba.conf and add:

# Enforce SSL + client cert validation

hostssl all all 0.0.0.0/0 cert clientcert=verify-ca

Optionally, use clientcert=verify-full to **verify CN against username**.

Step 4: Distribute Client Files Securely

Each client needs:

- client.key (Private Key)
- client.crt (Signed Certificate)
- root.crt (CA certificate used to sign server/client certs)

Ensure permissions:

chmod 600 client.key

Step 5: Connect Using Client Certificates

psql "host=db.yourdomain.com port=5432 user=analyst1 dbname=analytics sslmode=verify-ca sslcert=client.crt sslkey=client.key"

🔁 **Certificate Lifecycle Management & IAM Integration**

Challenges:

- Manually managing client certs can be complex as users join or leave.

- Revoking certs is critical to prevent unauthorized access.

Solutions:

- Use **Certificate Revocation Lists (CRL)**:

```
ssl_crl_file = 'revoked.crl'
```

- Enable **OCSP Stapling** (available in PostgreSQL 15+).
- Integrate with **IAM platforms**:
  - Use **OpenID Connect (OIDC)** or **Kerberos** for centralized identity.
  - Automatically issue and expire certificates using tools like HashiCorp Vault, Teleport, or Smallstep CA.

🚀 Future Trends: Zero Trust in PostgreSQL Security

Zero Trust Architecture (ZTA) is becoming the gold standard. In PostgreSQL context:

Feature	**Description**
Mutual TLS (mTLS)	Both client and server present valid X.509 certs
Just-In-Time Access	Grant access only when needed, and revoke quickly
Certificate Automation	Use short-lived certs with automatic renewal and revocation
Telemetry & Audit Logs	Track cert usage and connection attempts
Access Brokers (e.g., Teleport)	Provide per-query visibility and policy enforcement

---

🛠 Example: Using Teleport for PostgreSQL

Teleport supports:

- **Role-based access control (RBAC)**
- **Session logging**
- **Client certificate generation**
- **IAM federation**

🧠 Reflection Questions

- How frequently do you rotate server and client certificates?
- Are your SSL settings reviewed during every major patch or PostgreSQL upgrade?
- Can client-side certificate authentication be integrated with existing CI/CD or ITSM workflows?

📃 Summary

Component	**Best Practice**
File Permissions	600 for keys, 644 for certs, owned by postgres
SSL Modes	Use verify-ca or verify-full for strong validation
Client Auth	Use pg_hba.conf with clientcert=verify-ca
Automation	Integrate with IAM & Cert Management (e.g., Vault, Teleport)
Hardening	Enable OCSP, CRLs, enforce mTLS

Database and Server Security

Database Configuration

- **Purpose:** To enhance server security through various configuration parameters.
- **Methods:** Modify the postgresql.conf file.
  - **Port:** Change port = 5432 to a non-standard port to avoid common scanning attempts.
  - **Logging:** Enable detailed logging by setting log_statement = 'all' and log_connections = on.
  - **Permissions:** Ensure the data directory has restrictive permissions. The PostgreSQL user should be the owner, and no other users should have read/write access.
- **Checking with psql**:

  - You can check the value of any configuration parameter using the SHOW command:

SHOW port;

SHOW log_statement;

- **Checking with a GUI**:
  - In pgAdmin, you can navigate to the **Tools** menu, then **Server Configuration**, to see and modify the postgresql.conf file.

Detailed Guide to PostgreSQL Database Configuration

PostgreSQL is a highly configurable open-source database system, providing extensive options to tune performance, security, and functionality. This guide provides a thorough walkthrough of the main configuration methods, files, and the associated SQL commands, with practical examples.

1. Configuration File: postgresql.conf

Location and Purpose:

- Found in the PostgreSQL data directory (commonly /var/lib/pgsql/data/ or /var/lib/postgresql/XX/main/).
- Controls core server behavior, resource usage, logging, and more.

Key Parameters & Examples:

- Connections and Authentication:

```
listen_addresses = 'localhost'     # Listen only on localhost for security

port = 5432              # Default PostgreSQL port

max_connections = 100          # Number of simultaneous clients

authentication_timeout = 1min      # Max wait time for authentication
```

- Memory Allocation:

```
shared_buffers = 256MB          # Memory for caching data blocks

work_mem = 4MB              # Per-operation memory for sorting, hashing

maintenance_work_mem = 64MB       # For maintenance tasks like VACUUM
```

- Logging:

```
log_destination = 'csvlog'        # CSV log files
logging_collector = on            # Enable log file creation
log_filename = 'postgresql-%a.log'   # Log file naming convention
```

- WAL (Write-Ahead Log):

```
wal_level = replica               # Necessary for replication
max_wal_size = 1GB                # Max WAL size before checkpoint
checkpoint_timeout = 5min         # Frequency of checkpoints
```

- Query Planning:

```
effective_cache_size = 768MB       # Estimated OS disk cache available
geqo = on                          # Enable genetic query optimization
```

Applying Changes:

- After editing, reload or restart the service:

```
# Reload (no active connections dropped)
pg_ctl reload -D /path/to/data_dir
# OR using systemctl
sudo systemctl reload postgresql

# Restart (disconnects users)
pg_ctl restart -D /path/to/data_dir
# OR
sudo systemctl restart postgresql
```

2. Host-Based Authentication File: pg_hba.conf

Role:

- Located in the data directory, controls what hosts and users can connect and how they authenticate.
- Crucial for security.

Structure and Example:

```
# TYPE  DATABASE      USER        ADDRESS           METHOD
```

```
host   all          all          127.0.0.1/32         md5
host   app_db       app_user     192.168.1.0/24       scram-sha-256
local  all          postgres                          peer
```

- METHOD can be md5, scram-sha-256, peer, trust, etc.

Applying Changes:
Reload server for settings to apply:

```
pg_ctl reload -D /path/to/data_dir
sudo systemctl reload postgresql
```

3. SQL Commands for Dynamic Configuration

a. ALTER SYSTEM

- Changes configuration globally and writes to postgresql.auto.conf.
- Effective for all databases, overrides postgresql.conf.

Example:

```
ALTER SYSTEM SET log_min_duration_statement = 1000;  -- log queries longer than 1 second
-- Undo with:
ALTER SYSTEM RESET log_min_duration_statement;
```

- Reload required with:

```
SELECT pg_reload_conf();
```

b. ALTER DATABASE

- Sets parameters on a per-database basis.

Example:

```
ALTER DATABASE mydb SET work_mem = '16MB';
-- Reset for database:
ALTER DATABASE mydb RESET work_mem;
```

c. ALTER ROLE

- Sets parameters for specific users (roles).

Example:

```
ALTER ROLE dbuser SET maintenance_work_mem = '128MB';

-- Reset for role:

ALTER ROLE dbuser RESET maintenance_work_mem;
```

4. Complete Configuration/Deployment Process

Step 1: Installation

- Install via OS package (apt, yum, brew, etc.) or source.
- Pick data directory and superuser password:

```
# Example (on Linux):

sudo apt-get install postgresql

sudo -u postgres psql

\password postgres
```

Step 2: Initial Setup

- Locate config files with:

```
SHOW config_file;

SHOW hba_file;
```

Step 3: Editing Configurations

- Edit postgresql.conf and pg_hba.conf as described above.

Step 4: Applying and Validating

- Reload/restart as described.
- Validate settings:

```
SHOW work_mem;

SHOW log_min_duration_statement;
```

Step 5: Dynamic Changes

- Use ALTER SYSTEM, ALTER DATABASE, ALTER ROLE for adjustments without file edits.

5. Important Best Practices

- Back Up: Always back up configuration files before changes.
- Documentation: Consult [official PostgreSQL documentation]() for each parameter.

- Testing: Apply changes in a staging environment first for production systems.
- Reload vs. Restart: Some changes (like listen_addresses) require restart; others (like logging settings) only need a reload.

Example: Setting Up a Custom Work Memory Limit

Edit via SQL:

ALTER SYSTEM **SET** work_mem = '8MB';

SELECT pg_reload_conf();

Database-specific:

ALTER DATABASE mydb **SET** work_mem = '16MB';

Role-specific:

ALTER ROLE johndoe **SET** work_mem = '32MB';

Check Effective Setting:

SHOW work_mem;

By combining configuration file editing with dynamic SQL commands, PostgreSQL allows precise and powerful database configuration. Always tailor settings to your workload, and carefully monitor impact after each change for best results.

Data Security

- **Purpose:** To protect sensitive data at rest.
- **Methods:**
  - **pgcrypto**: This extension provides cryptographic functions for encrypting data within columns. For example, you can encrypt a password column using pgp_sym_encrypt.
  - **Row-Level Security (RLS)**: RLS policies provide fine-grained control over which rows a user can access. For example, a policy could ensure a sales agent can only see their own sales records.
- **Checking with psql**:
  - To see RLS policies on a table:

```
\d+ table_name;
```

- This command will show if Row-Level Security is enabled on the table.

- **Checking with a GUI**:
  - In pgAdmin, right-click on a table, go to **Properties**, and you can find the **Security** tab to manage RLS policies.

PostgreSQL's data security is a comprehensive system that protects your data from various threats. It combines **encryption**, **access control**, and **auditing** to create a robust defense.

Encryption 🛡️

Encryption is a foundational security measure that protects data in two primary states: when it's at rest and when it's in transit.

Data at Rest 💾

Encrypting data at rest means it's unreadable to anyone who might gain unauthorized access to the physical storage media. PostgreSQL doesn't have built-in Transparent Data Encryption (TDE) like some other databases. Instead, you can use:

- **Filesystem-level encryption:** Use tools like LUKS on Linux to encrypt the entire disk partition where your PostgreSQL data resides.
- **Extension-based encryption:** The **pgcrypto** extension provides cryptographic functions for encrypting specific data columns. This is useful for protecting highly sensitive data, like social security numbers or credit card details, within a table.
- **Third-party tools:** You can also use third-party solutions that integrate with PostgreSQL to provide TDE-like functionality.

Password Encryption

- **Storage Format**: User passwords are stored as cryptographic hashes, controlled by the password_encryption setting.
- **Supported Methods**:
  - **SCRAM-SHA-256**: Preferred method; follows IETF standards and offers strong security.
  - **MD5**: Legacy method; less secure and PostgreSQL-specific.

- **Client-Side Hashing**: With SCRAM or MD5, the client hashes the password before transmission, ensuring the plaintext password never touches the server.

Column-Level Encryption (pgcrypto)

- **Use Case**: Encrypt specific fields (e.g., SSNs, credit card numbers) without encrypting the entire database.
- **Module**: pgcrypto provides functions like pgp_sym_encrypt() and pgp_sym_decrypt().
- **Key Handling**:
    - The client supplies the decryption key.
    - Decrypted data and keys exist briefly on the server, posing a risk if the server is compromised.

Data in Transit 🌐

Data in transit refers to data as it moves between a client application and the PostgreSQL server. This is a common point of vulnerability for eavesdropping. PostgreSQL uses **SSL/TLS** to encrypt this communication, preventing man-in-the-middle attacks. To enable it, you must:

1. **Configure the server:** Set up SSL in the **postgresql.conf** file by specifying a certificate and private key.
2. **Configure clients:** Clients must be configured to connect using SSL, which can also be set to enforce that all connections must be encrypted.

SSL Host Authentication

- **Mutual Authentication**: Both client and server present SSL certificates.
- **Benefits**:
    - Prevents impersonation attacks.
    - Mitigates "man-in-the-middle" risks.
- **Setup**: Requires certificate configuration on both ends.

Client-Side Encryption

- **Use Case**: When the server admin cannot be trusted.
- **Mechanism**:
    - Data is encrypted before transmission.

  - Decryption occurs only on the client.

- **Benefit**: Unencrypted data never resides on the server.

Network Encryption

🔷 **SSL/TLS**

- Encrypts all traffic: passwords, queries, and results.
- Controlled via pg_hba.conf:
  - host: allows non-encrypted connections.
  - hostssl: enforces SSL.
- Clients can enforce SSL with sslmode=require.

🔷 **GSSAPI (Kerberos)**

- Encrypts all traffic without transmitting passwords.
- Controlled via pg_hba.conf:
  - hostgssenc: enforces GSSAPI encryption.
- Clients can enforce with gssencmode=require.

🔷 **SSH/Stunnel**

- External tunneling options for encrypting PostgreSQL traffic.

Access Control 🔒

Access control is about defining who can connect to the database and what they can do once they're in. PostgreSQL provides several layers of control to manage this.

Role-Based Access Control (RBAC) 🧑‍💼

PostgreSQL's RBAC system is a powerful tool for managing permissions. It lets you create **roles**, which can represent individual users or groups. You then grant specific **privileges** (like SELECT, INSERT, UPDATE, DELETE) on database objects (tables, views, functions) to these roles. This system adheres to the **principle of least privilege**, ensuring users only have the permissions they need to perform their jobs.

pg_hba.conf 📜

The **pg_hba.conf** (host-based authentication) file is the first line of defense for controlling who can connect to your database server. It defines rules based on:

- **Host:** The IP address or subnet from which the connection is originating.
- **Database:** The specific database the user is trying to connect to.
- **User:** The user role attempting to connect.
- **Authentication method:** The method used to verify the user's identity, such as scram-sha-256, cert, or ldap.

Row-Level Security (RLS) 📏

RLS offers the most granular level of control by restricting which rows a user can see or modify in a table. You define **policies** that filter data based on the current user or other criteria. For example, a policy could ensure that a user can only see data related to their own department.

Authentication and Auditing 🕵️

These two areas focus on verifying user identity and tracking database activity.

Authentication 🔐

PostgreSQL supports various authentication methods, but using strong, modern options is critical. **SCRAM-SHA-256** is the recommended method for password-based authentication, as it's more secure than older methods like MD5. Integrating with external systems like **LDAP** or **Kerberos** can also enable centralized user management and **multi-factor authentication (MFA)** for an added layer of security.

Auditing and Logging 📝

Regularly reviewing database activity is crucial for identifying security threats. PostgreSQL's logging features can be configured to capture detailed information, including:

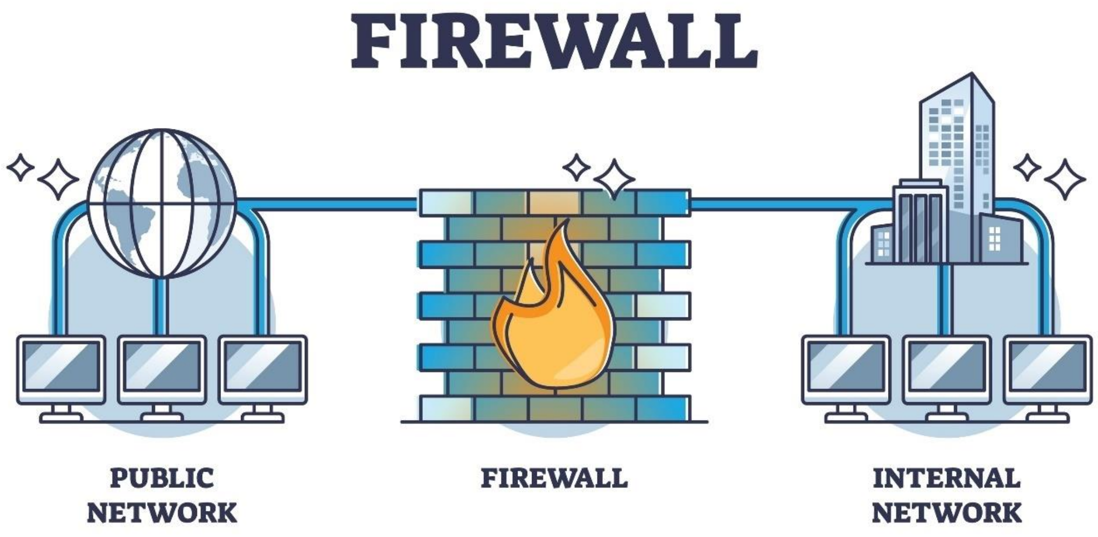
- **Statement logging:** Logs all SQL statements executed by clients.
- **Connection logging:** Records successful and failed connection attempts.

By analyzing these logs, you can detect unauthorized access, suspicious queries, or privilege escalation attempts.

Other Best Practices 🛠️

Beyond the core security features, several other practices are vital for a secure PostgreSQL environment.

- **Network security:** Place your PostgreSQL server behind a firewall and configure it to accept connections only from trusted hosts.



Licensed by Google

- **Regular patching:** Keep PostgreSQL and all its extensions up-to-date to ensure you have the latest security fixes.
- **Data masking:** In non-production environments, use tools or functions to obscure sensitive data, preventing accidental exposure.
- **Principle of least privilege:** Always grant roles the minimum privileges required to perform their duties. Never grant a user more access than they absolutely need.

🔐 What Is a Security Vulnerability in PostgreSQL?

A **security vulnerability** in PostgreSQL is defined as any issue that:

- Allows unauthorized access to data or privileges.
- Enables arbitrary code execution via PostgreSQL processes.
- Permits privilege escalation (e.g., an unprivileged user becoming a superuser).
- Results in denial-of-service (DoS) under certain conditions.

Notably, actions performed by a PostgreSQL superuser are **not** considered vulnerabilities unless they enable privilege escalation or unauthorized access.

📋 How Vulnerabilities Are Tracked

PostgreSQL maintains a centralized Security Information page that includes:

- A table listing known vulnerabilities.
- Affected major and minor versions.
- Fix versions.
- Whether a valid login is required for exploitation.
- CVSS (Common Vulnerability Scoring System) scores for severity assessment.

PostgreSQL is also a **CVE Numbering Authority (CNA)**, meaning it can assign CVE identifiers to its own vulnerabilities and related projects like pgAdmin, pgJDBC, and PgBouncer.

🛠 Recent and Notable Vulnerabilities

Here are some recent examples:

CVE ID	**Description**	**CVSS Score**	**Fixed In Version**
CVE-2023-5869	Integer overflow in array modification allows arbitrary code execution	8.8 (High)	PostgreSQL 16.5+
CVE-2023-39417	SQL injection via extension scripts with quoting constructs	7.5 (High)	PostgreSQL 16.5+
CVE-2023-5868	Memory disclosure via aggregate functions with 'unknown'-type arguments	4.3 (Medium)	PostgreSQL 16.5+
CVE-2024-0985	Privilege drop flaw in REFRESH MATERIALIZED VIEW CONCURRENTLY	8.0 (High)	PostgreSQL 16.5+
CVE-2024-10979	Environment variable manipulation leading to code execution or data leakage	8.8 (High)	PostgreSQL 16.5+3

🔄 Security Updates and Patch Releases

The PostgreSQL team regularly releases **minor version updates** that include:

- Fixes for security vulnerabilities.

- Bug patches.
- Performance improvements.

As of the latest update, supported versions include:

- PostgreSQL 17.1
- PostgreSQL 16.5
- PostgreSQL 15.9
- PostgreSQL 14.14
- PostgreSQL 13.17
- PostgreSQL 12.21

🧪 Reporting and Mitigation

If you discover a vulnerability:

- Email the PostgreSQL Security Team at **security@postgresql.org**.
- For non-security bugs, use the Report a Bug page.

Mitigation strategies include:

- Keeping PostgreSQL up to date.
- Restricting access to trusted users.
- Using secure extensions and avoiding unverified third-party scripts.

Role Permissions and Access Control

PostgreSQL uses a **Role-Based Access Control (RBAC)** model, where roles are used to manage privileges.

Mind Map of RBAC

Key Concepts

- **Roles**: Can be thought of as a user, a group of users, or a set of permissions.
- **Users**: A role with the LOGIN privilege.

- **Privileges**: Specific permissions to perform actions (SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT) on database objects (tables, schemas, databases, etc.).
- **PUBLIC Role**: A special, built-in role to which every user belongs. It's crucial to **revoke privileges** from this role to enforce granular security.

Privilege Management

- **Method:** Use the GRANT and REVOKE commands.
- **Granting Privileges**:

```
GRANT SELECT, INSERT ON employees TO app_role;
```

- **Revoking Privileges**:

```
REVOKE CONNECT ON DATABASE mydb FROM PUBLIC;
```

- **Role Inheritance**: Grant one role to another to create a hierarchy.

```
GRANT admin_role TO john;
```

This grants all privileges of admin_role to the user john.

Checking Permissions

- **Checking with psql**:
  - To see a list of all roles:

```
\du
```

  - To see privileges on a table:

```
\dp table_name
```

  - To see which roles a user is a member of:

```
\dg+ role_name
```

- **Checking with a GUI**:
  - In pgAdmin, navigate to **Login/Group Roles** under the server tree. You can view and edit roles, their permissions, and memberships.
  - Right-clicking on a table and selecting **Properties** will show you the privileges granted on that object.