VectraOps

Knowledge Library

TUTORIAL

How to Secure PostgreSQL Connections with SSL/TLS

This tutorial shows how to secure PostgreSQL client and server connections with SSL/TLS, verify certificate trust, and validate that encryption is actually in use before production rollout.

Databases - July 15, 2026

Why this matters operationally

PostgreSQL connections often carry credentials, application data, and administrative traffic across networks that cannot be treated as inherently trusted. Without SSL/TLS, a client may still connect successfully, but the session can be exposed to interception, tampering, or credential theft on the path between the application and the database.

In this tutorial, you will learn how to secure PostgreSQL connections with SSL/TLS, validate that encryption is active, and confirm that certificate trust is working as expected before you rely on it in production. The finished state is a PostgreSQL deployment where clients connect over TLS, certificate verification is enforced where appropriate, and you have checks to prove the setup is doing what you intended.

Before you start: prerequisites and stop-here checks

Goal

Make sure the environment can support a safe TLS rollout before you change server or client settings.

What to verify

- You control the PostgreSQL server configuration and can restart or reload it.
- You have a server certificate, private key, and certificate chain that clients can trust.
- You know the DNS name or IP address clients will use, because certificate validation depends on identity matching.
- You can update client connection strings, drivers, or connection pools.
- You know whether any replication, proxy, or pooler traffic also needs TLS. If you are securing replication traffic too, coordinate the certificate plan with How to Configure PostgreSQL Streaming Replication Securely.

Stop-here-if warnings

Stop here if any of the following are true:

- You only have a self-signed certificate but no plan for distributing its trust anchor to clients.
- The server name clients use does not match the name on the certificate.
- You cannot confirm which application drivers support certificate verification mode.
- You are about to enable TLS without a rollback plan for clients that may not trust the new certificate chain.

If you proceed without these checks, the most common outcome is not failure to connect, but insecure fallback behavior or validation errors that break production traffic.

What a secure PostgreSQL TLS setup should look like

Goal

Define the target state so you can verify it later.

Expected output

A secure PostgreSQL connection path usually has these properties:

- The server presents a valid certificate.
- The client verifies the certificate chain against a trusted CA.
- The client checks the server identity against the hostname it connected to.
- Plaintext connections are either disabled or intentionally restricted.
- Application connection strings explicitly request TLS and, where appropriate, require verification.

Validation

You should be able to answer these questions after implementation:

- Does the connection negotiate TLS?
- Does the client verify the certificate chain?
- Does the server certificate match the hostname in use?
- Are any paths still allowing unencrypted access?

Common failure

A common mistake is assuming that ?using SSL? means the connection is secure. In practice, a TLS connection without certificate validation still protects against passive sniffing but can remain vulnerable to man-in-the-middle attacks.

Prepare the server certificate and trust chain

Goal

Provide PostgreSQL with a certificate and key that clients can validate.

Action

Use a certificate issued by an internal or public CA that your clients can trust. The certificate should include the DNS names clients actually use, usually in the Subject Alternative Name field.

At minimum, the server needs:

- a certificate file
- a private key file
- any intermediate CA certificates required to build the chain

A common server-side configuration pattern is:

The exact file locations and reload behavior depend on your packaging and version, so verify the configuration path used by your deployment.

Expected output

PostgreSQL can present a certificate to clients and, if configured, validate client certificates as well.

Validation

- Confirm the certificate subject and SAN entries match the hostnames clients will use.
- Confirm file permissions protect the private key from non-privileged reads.
- Confirm the certificate chain is complete enough for client validation.

Common failure

The most frequent issue is a hostname mismatch. If the certificate is issued to db.internal.example but clients connect to 10.10.10.15, strict verification will fail unless the IP is also included in the certificate SAN and the client is configured accordingly.

Enable SSL on the PostgreSQL server

Goal

Make the server advertise and accept TLS for client sessions.

Action

Set SSL to enabled in postgresql.conf or your managed configuration equivalent, then load the certificate and key paths. After changing the configuration, restart or reload as required by the specific setting and your PostgreSQL version.

For example:

If you are using a packaged service or container, ensure the files are mounted where PostgreSQL can read them and the service user has appropriate access.

Expected output

The server will accept TLS-capable client connections and present its certificate during the handshake.

Validation

Check the server logs for startup errors related to certificates, keys, or file permissions.
If the server fails to start after enabling SSL, the likely causes are:

- incorrect file ownership or permissions on the private key
- an unreadable certificate path
- an invalid certificate or key pair
- a missing intermediate chain

Common failure

A private key that is too permissive is a common startup blocker. PostgreSQL expects the key to be protected; if permissions are wrong, it may refuse to use it.

Require TLS at the access-control layer

Goal

Prevent accidental plaintext connections from being accepted for roles or networks that should only use encrypted transport.

Action

Use pg_hba.conf entries that require SSL for the relevant users and networks. A typical pattern is to use hostssl rather than host for accounts that must always connect over TLS.

Example:

The exact authentication method is separate from TLS, but the two should be designed together. If your organization already uses strong password auth or certificates, TLS protects those credentials in transit. If you are reviewing broader database hardening, row-level controls such as PostgreSQL Row-Level Security for Secure Multi-Tenant Access can complement transport security, but they solve a different problem.

Expected output

Clients connecting to protected databases or roles are forced onto TLS.

Validation

- Attempt a non-TLS connection and confirm it is rejected or does not match any allowed rule.
- Attempt a TLS connection and confirm it succeeds.
- Confirm the intended authentication method still works after switching to hostssl.

Common failure

An overlapping host rule can unintentionally permit plaintext access. Order matters in pg_hba.conf, so validate the first matching rule for each client path.

Configure clients to verify certificates

Goal

Make client connections both encrypted and authenticated.

Action

Update application connection strings or driver settings so they request TLS and verify the server certificate. The safest choice is usually a mode that both encrypts and verifies the hostname, often exposed as a ?verify-full? equivalent in drivers.

Typical connection properties to look for include:

- TLS/SSL enabled
- CA certificate path or trust store
- hostname verification enabled
- optional client certificate and key, if mutual TLS is required

Example connection URI pattern:

If your driver uses a different property name, map the same security requirements into that driver?s terminology rather than relying on defaults.

Expected output

The application negotiates TLS and refuses servers whose certificate chain or hostname does not validate.

Validation

- Confirm the application connects successfully when the certificate is trusted and the hostname matches.
- Rename or mismatch the hostname in a test environment and confirm validation fails.
- Confirm the application does not silently downgrade to an unverified TLS mode.

Common failure

Many drivers default to encryption without hostname verification, or to no encryption at all unless configured. Do not assume the driver?s default is safe; inspect it explicitly.

Verify encryption from the client and the server

Goal

Prove that the connection is actually using TLS and that the server sees it that way.

Action

Use both client-side and server-side checks so you are not relying on one layer of evidence.

On the client, you can often inspect the connection mode using the driver?s diagnostics or a simple session query. On the server, PostgreSQL exposes SSL status in session-level views.

Example session check:

You can also verify from an active session:

Expected output

You should see ssl = true for active encrypted sessions, along with the negotiated protocol version and cipher suite.

Validation

- Connect with the application or a TLS-capable client and confirm ssl = true.
- Compare a known TLS session with a deliberately non-TLS test path, if your configuration still allows one in a controlled environment.
- Confirm that the cipher and protocol version align with your organization?s policy.

Common failure

A session can appear to work while the client is not checking certificate identity. Encryption alone is not enough if the certificate is not validated.

Test certificate trust and hostname verification deliberately

Goal

Make sure failures happen when they should, so you know verification is real.

Action

Run at least one negative test in a non-production environment or during a maintenance window.

Useful tests include:

- connecting with the wrong hostname
- connecting with an untrusted CA
- connecting with a certificate that does not include the target SAN
- removing the root CA from the client trust store temporarily, then restoring it

Expected output

The client should reject the connection with a certificate or trust error, not silently continue.

Validation

A successful negative test produces evidence that your client settings are enforcing trust. If the connection still succeeds under a mismatch, the client is probably not verifying the server identity.

Common failure

The most dangerous false positive is a TLS connection that continues even when the hostname or CA is wrong. That usually means the client is using a permissive mode and not full verification.

Operational follow-up after rollout

Goal

Keep the TLS setup healthy after deployment.

Action

Add routine checks for certificate expiration, chain validity, and configuration drift. Monitor application logs for TLS negotiation failures, trust-store errors, and hostname mismatch errors after certificate rotation or infrastructure changes.

You should also plan for certificate renewal before expiry and test the renewal path in advance. If your database topology includes replication or other internal PostgreSQL-to-PostgreSQL traffic, keep the TLS rules consistent across those channels as well; secure transport problems in one path often surface during failover or replica promotion.

Expected output

Certificate rotation does not break application connectivity, and TLS remains enforced across the intended connection paths.

Validation

Before production sign-off, verify all of the following:

- the server certificate is current and trusted
- the client uses certificate verification mode
- pg_stat_ssl confirms encrypted sessions
- non-TLS paths are blocked or intentionally limited
- application pools reconnect successfully after a controlled restart or certificate refresh

Common failure

The most common post-rollout issue is expiration or chain drift: the server certificate is renewed, but clients still trust the old chain or were never configured to trust the issuing CA correctly.

Final check

A secure PostgreSQL TLS implementation is not complete until you have proven three things: the server presents a valid certificate, the client verifies that certificate, and the access rules prevent accidental plaintext use. If you can show those checks passing in your environment, you have moved from ?TLS is enabled? to ?PostgreSQL connections are actually secured.?

Use this guidance together with Oracle database auditing to connect the workflow with related operational context already available on the site.