TUTORIAL

How to Secure PostgreSQL Connections with SSL and TLS

Learn how to secure PostgreSQL connections with SSL and TLS by preparing certificates, configuring server and client settings, validating encryption, and operating the setup safely in production.

Databases - August 04, 2026

What securing PostgreSQL connections actually changes

Plain PostgreSQL connections leave authentication and query traffic exposed to interception on the network path. In practice, that creates two operational problems: credentials can be captured if they are reused or poorly protected, and application traffic can be observed or modified by an attacker who can place themselves between the client and the server.

This tutorial shows how to secure PostgreSQL connections with SSL and TLS so you can require encrypted sessions, verify server identity, and confirm that clients are actually using encryption. By the end, you will know how to prepare certificates, enable encrypted connections on the server, configure clients to trust the right CA, validate the result, and check the setup before production use.

What you should have before you start

Goal

Confirm that the environment is ready for encrypted PostgreSQL traffic before you change server settings.

Action

Check these prerequisites:

- You have administrative access to the PostgreSQL server configuration and data directory.
- You know the hostname clients use to reach the database.
- You can restart PostgreSQL during a maintenance window.
- You can distribute CA certificates and client files to application hosts.
- You know whether clients connect directly, through a load balancer, or through a proxy that may terminate TLS.

Expected output

You can identify where certificates will live, how clients resolve the server name, and whether TLS will be terminated by PostgreSQL itself or by another component.

Validation

Confirm the connection path end to end. If an intermediate proxy or pooler terminates TLS, PostgreSQL server-side SSL settings alone will not protect the wire between the client and that intermediary.

Common failure

A frequent mistake is enabling TLS on the database server while clients continue connecting to a proxy, pooler, or alternate address that does not use encryption.

Stop here if

Stop here if you do not control the full connection path or cannot distribute trust material safely. In that case, fix the network design first, because certificate configuration alone will not protect a plaintext segment in the middle.

Choose the certificate model

Goal

Decide how the server certificate will be issued and what clients will trust.

Action

Use one of these common models:

- Internal CA: best for private infrastructure and controlled fleets.
- Public CA: useful when clients are external or already trust a public PKI.
- Self-signed server certificate: acceptable for isolated labs, but usually not ideal for production because trust distribution is manual and easy to mismanage.

For production, a CA-signed server certificate is the practical default. If clients must verify the server identity, make sure the certificate includes the DNS name or names they actually use.

If your access model also depends on privilege boundaries after the connection is established, combine transport encryption with explicit authorization controls such as PostgreSQL Role-Based Access Control Setup and Management Tutorial and, where row filtering matters, PostgreSQL Row-Level Security: Enforce Least-Privilege Access.

Expected output

You have a clear trust model: who signs the server certificate, where the CA certificate will be stored, and what hostname clients will validate.

Validation

Check that the server certificate subject alternative name matches the exact DNS name clients will use. If the certificate only contains a short host name but clients connect by FQDN, hostname verification can fail even though the server is encrypted.

Common failure

The most common certificate mistake is a mismatch between the certificate names and the actual connection string. Another common issue is distributing the server certificate without the CA certificate, which allows encryption but not identity verification.

Enable SSL/TLS on the PostgreSQL server

Goal

Make PostgreSQL present a certificate and accept encrypted connections.

Action

Update the server configuration so it can use TLS, then restart or reload as required by your version and packaging.

Typical configuration items include:

Place the files where PostgreSQL can read them, and make sure the private key is readable only by the database service account and protected with restrictive file permissions.

If your deployment uses certificate chains, include the full chain in the correct file as required by your certificate format and PostgreSQL version. Verify the required file names and supported formats for your specific release, because behavior can vary by version.

Expected output

PostgreSQL starts successfully with SSL enabled and can present a certificate to clients.

Validation

After restart, check the server log for TLS initialization errors. Then query the server to confirm that SSL is enabled:

You can also connect and inspect the session state:

A successful connection alone is not enough; you still need to verify whether the session is encrypted.

Common failure

Common failures include wrong file ownership, unreadable private keys, unsupported key formats, or a certificate chain that does not match the key. PostgreSQL will usually fail fast at startup if the TLS files are invalid, so server logs are the first place to look.

Require encryption for client connections

Goal

Prevent clients from falling back to plaintext connections.

Action

Adjust pg_hba.conf so the access rules require SSL or prefer it explicitly. Use the hostssl entry type when you want a rule to apply only to encrypted sessions.

Example:

If you currently have generic host rules that allow plaintext, decide whether they should be removed or narrowed so that unencrypted traffic is no longer accepted.

Expected output

Clients that do not negotiate TLS are rejected, while encrypted clients can authenticate normally.

Validation

Test both paths intentionally:

- A client configured for SSL should connect successfully.
- A client forced to use plaintext should fail if the server is locked down correctly.

This is the most important operational validation step, because encryption enabled on the server is not the same thing as encryption being required.

Common failure

A broad host all all 0.0.0.0/0 rule can silently permit plaintext if it appears before stricter TLS-only rules. Rule order matters, so review the effective matching sequence carefully.

Configure clients to verify the server

Goal

Make clients not only encrypt the session, but also verify they are talking to the correct server.

Action

Distribute the CA certificate to the client host and configure the connection string or client settings to require verification.

A libpq-style connection string may look like this:

Use the strictest mode that fits your trust model:

- require encrypts traffic but does not verify the server name.
- verify-ca verifies the certificate chain.
- verify-full verifies the chain and the hostname.

For production systems, verify-full is usually the correct choice when DNS names are stable and certificate names are managed properly.

Expected output

The client validates the server certificate and rejects unexpected identities.

Validation

Use the client?s own SSL reporting or connection verbosity to confirm the negotiated TLS session and verification mode. If hostname validation is enabled, a name mismatch should cause a connection failure rather than a warning.

Common failure

A frequent mistake is using sslmode=require and assuming that means the server identity is verified. It does not. That setting encrypts traffic, but it still allows a man-in-the-middle with a trusted path to impersonate the server if the client does not validate the certificate name.

Validate that sessions are actually encrypted

Goal

Prove that live sessions use TLS and not a fallback path.

Action

Check server-side session metadata. A simple query can show whether the current connection is using SSL:

If your PostgreSQL version or view columns differ, verify the exact column names for your release. Some deployments also use external monitoring or logs to confirm cipher negotiation and handshake details.

Expected output

You should see an active SSL session with a negotiated protocol version and cipher.

Validation

Confirm that the result is not null or empty and that the session is associated with the current backend process. If ssl is false, the connection is not encrypted even if authentication succeeded.

Common failure

If the query returns no row, the session may not be connected to the expected backend or the privileges may be insufficient to inspect the view in your environment. If the values indicate a weak or unexpected protocol, review server and client library versions.

Test certificate failure cases before production

Goal

Ensure the system fails safely when trust is broken.

Action

Perform controlled negative tests in a non-production environment:

- Connect with the wrong CA file.

- Use a hostname that does not match the certificate.
- Attempt a plaintext connection against a TLS-only rule.

These tests are especially useful after rotation changes, because the failure mode often reveals whether clients are truly validating the server or only encrypting opportunistically.

Expected output

The connection should fail clearly and predictably for each invalid case.

Validation

Document the exact error messages your clients return so operators can distinguish certificate trust problems from network reachability problems.

Common failure

A setup that ?works? even when the CA file is wrong often means the client is not verifying the server identity. That is acceptable only if you deliberately chose a lower-trust mode, which is uncommon for production databases.

Operate the setup safely over time

Goal

Keep TLS working after certificate renewals, restarts, and client changes.

Action

Add the following operational checks to your maintenance process:

- Track certificate expiration dates.
- Test renewal in staging before production.
- Keep private key permissions restrictive.
- Reconfirm client hostname validation after DNS or load balancer changes.
- Review pg_hba.conf after access-rule changes so plaintext rules do not reappear.

If you use connection pooling, verify whether the pooler terminates TLS, re-encrypts traffic to the server, or passes plaintext to PostgreSQL. Transport security is only as strong as the weakest segment.

Expected output

TLS remains enforced after routine operational changes, and certificate rotation does not break applications unexpectedly.

Validation

After every certificate or hostname change, run a real application connection test and inspect the negotiated SSL state again. Do not rely solely on service restarts or log messages.

Common failure

The most common lifecycle issue is expired or replaced certificates that no longer match client trust stores. The second most common is assuming that a network device still preserves the same security behavior after a topology change.

What a finished secure setup looks like

A completed configuration has three properties:

- PostgreSQL can present a valid server certificate.
- The server rejects plaintext connections where encryption is required.
- Clients verify the CA and, ideally, the hostname with verify-full or an equivalent setting.

At that point, you have not only enabled SSL/TLS, but also confirmed that the deployment actually enforces encrypted, identity-checked PostgreSQL connections in the path your applications use. That is the difference between a checkbox configuration and a secure operational state.

Use this guidance together with NoSQL security to connect the workflow with related operational context already available on the site.