SSL framework

The SSL framework provides support for securing the following Neo4j communication channels using standard SSL/TLS technology:

- `bolt` (port - `7687`)
- `https` (port - `7473`)
- `cluster` (ports - `6000`, `7000`, and `7688`)
  Note that port `5000` is no longer used from Neo4j 2025.01 onwards.
- `backups` (port - `6362`)

This page describes how to set up SSL within your environment, how to view, validate, and test the certificates.

SSL Providers

The secure networking in Neo4j is provided through the Netty library, which supports both the native JDK SSL provider as well as Netty-supported OpenSSL derivatives. Each version of Neo4j ships with a version of Netty and Netty requires a specific version of the `netty-tcnative` library for compatibility, which can be found in the `lib/` directory of the Neo4j installation. 2025.01.0 ships with *Netty 2.0.69.Final* and requires *netty-tcnative-boringssl-static-2.0.69.Final*.

NOTE

Using dynamic versions of tcnative requires installation of platform-specific dependency libraries as described in https://netty.io/wiki/forked-tomcat-native.html →.

In most use cases, the statically linked `boringssl` variant of `netty-tcnative` is sufficient to enable SSL encryption.

Follow these steps to use OpenSSL:

1. Install a suitable `netty-tcnative` dependency into the `plugins/` directory of Neo4j. Which `netty-tcnative` version you need depends on your OS and architecture.

AI search

2. Set `dbms.netty.ssl.provider=OPENSSL` . Using OpenSSL can significantly improve performance, especially for AES-GCM-cryptos, e.g. TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256.
3. Restart Neo4j.

Certificates and private keys

Certificates

The SSL configuration requires SSL certificates to be issued by a Certificate Authority (CA). All certificates must follow the X.509 standard and be saved in a `PEM-encoded` file.

TIP

Valid trusted certificates can be generated for free using non-profit CAs such as Let's Encrypt.

Example public.crt file

```
-----BEGIN CERTIFICATE-----
MIIDojCCAoqgAwIBAgIBATANBgkqhkiG9w0BAQsFADBhMQswCQYDVQQGEwJTRTEQ
     ...
xsUBvcQuyxewlvWRS18YB51J+yu0Xg==
-----END CERTIFICATE-----
```

The instructions on this page assume that you have already obtained the required certificates from the CA and added them to the *public.crt* file. To achieve this, you should concatenate each PEM-encoded certificate, starting from the leaf certificate and moving up the chain toward the root.

TIP

If the same certificates are used across all instances of the cluster, ensure that the DNS names of all cluster instances are included in the certificates when generating them. Multi-host and wildcard certificates are also supported.

TIP

If setting up intra-cluster encryption as part of a cluster configuration, ensure that the certificates used on the cluster endpoint support server and client usage. This is because when connecting the servers for clustering, each server uses its own certificate to authenticate as a client on the connection to another server.

This could be verified from within the certificate details:

```
openssl x509 -in public.crt -noout -text
```

You should see that the X509v3 Extended Key Usage section shows both the usages listed:

```
X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Authentication
```

Transformations

Neo4j requires all SSL certificates to be in the `PEM` format. If your certificate is in the binary `DER` format, you must transform it into `PEM` format.

Transform DER format certificate to PEM format

```
openssl x509 -in cert.der -inform der -outform pem -out cert.crt
```

Private keys

Private keys must be in the standard format PKCS #8 and saved as a `PEM-encoded` file.

Example private.key file

```
-----BEGIN PRIVATE KEY-----
MIICdQIBADANBgkqhkiG9w0BAQEFAASCAl8wggJbAgEAAoGBAN5D0I4bgdQK4In6
     ...
oaMe91ZPQ1JI
-----END PRIVATE KEY-----
```

Private keys can also be encrypted with a passphrase according to the PKCS #5 standard.

Example private.key file with passphrase encryption

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIICojAcBgoqhkiG9w0BDAEEMA4ECL3eSAoRlJ18AgIIAASCAoCj7WDyjsgcawdv
     ...
lYeSjVah
-----END ENCRYPTED PRIVATE KEY-----
```

Transformation

If the private key is encoded with the old PKCS #1 format, the file will typically start with the line:

```
-----BEGIN RSA PRIVATE KEY-----
```

You can convert it to PKCS #8 format with the following command:

Convert a PKCS #1 key to a PKCS #8 key

```
openssl pkcs8 -topk8 -in pkcs1.key -out pkcs8.key
```

An unencrypted private key could be PKCS #1 or PKCS #8. It can be encrypted with the following command:

Convert an unencrypted key to an encrypted PKCS #8 key using 256bit AES in cipher-block-chaining (CBC) mode

```
openssl pkcs8 -topk8 -v2 aes-256-cbc -v2prf hmacWithSHA512 -in pkcs1or8.key -out
pkcs8.encrypted.key
```

Supported encryption arguments to openssl are:

- `-v2 aes-128-cbc -v2prf hmacWithSHA1`
- `-v2 aes-128-cbc -v2prf hmacWithSHA224`
- `-v2 aes-128-cbc -v2prf hmacWithSHA256`

- `-v2 aes-128-cbc -v2prf hmacWithSHA384`
- `-v2 aes-128-cbc -v2prf hmacWithSHA512`
- `-v2 aes-256-cbc -v2prf hmacWithSHA224`
- `-v2 aes-256-cbc -v2prf hmacWithSHA256`
- `-v2 aes-256-cbc -v2prf hmacWithSHA384`
- `-v2 aes-256-cbc -v2prf hmacWithSHA512`

NOTE

Versions before Neo4j 5.0 allow keys to be stored with the old PKCS #1 standard. You can identify them by the line `-----BEGIN RSA PRIVATE KEY-----` at the beginning of the file. While Neo4j 5.0 can load and use those keys, they are considered deprecated and will be removed in a future version.

Validate the key and the certificate

If you need, you can validate the key file and the certificate as follows:

Validate the key

```
openssl rsa -in private.key -check
```

Validate certificate in the PEM format

```
openssl x509 -in public.crt –text –noout
```

Network connectors

Before enabling SSL support, you must ensure the following network connector configurations to avoid errors:

- Set `server.https.enabled` to `true` when using HTTPS.
- Set `server.bolt.tls_level` to `REQUIRED` or `OPTIONAL` when using Bolt.

For more information on configuring network connectors, see Configure network connectors.

Configuration

The SSL policies are configured by assigning values to parameters of the following format:

```
dbms.ssl.policy.<scope>.<setting-suffix>
```

- `scope` is the name of the communication channel, such as `bolt`, `https`, `cluster`, and `backup`.
- `setting-suffix` can be any of the following:

Setting suffix	Description	Default value
	Basic	
`enabled`	Setting this to `true` enables this policy.	`false`
`base_directory`	The base directory under which cryptographic objects are searched for by default.	`certificates/<scope>`
`private_key`	The private key used for authenticating and securing this instance.	`private.key`
`private_key_password`	The passphrase to decode the private key. Only applicable for encrypted private keys.	
`public_certificate`	A public certificate matching the private key signed by a CA.	`public.crt`
`trusted_dir`	A directory populated with certificates of trusted parties.	`trusted/`
`revoked_dir`	A directory populated with certificate revocation lists (CRLs).	`revoked/`
	Advanced	

1. In Neo4j 2025.01, the default value is changed from `false` to `true`.

Setting suffix	Description	Default value
`verify_hostname` [1]	This setting turns on client-side hostname verification. After receiving the server's public certificate, the client compares the address it uses against the certificate Common Name (CN) and Subject Alternative Names (SAN) fields. If the address does not match those fields, the client disconnects.	`true`
`ciphers`	A comma-separated list of ciphers suites allowed during cipher negotiation. Valid values depend on the current JRE, SSL provider, and TLS version. For Ciphers supported by the Oracle JRE, see the Oracle official documentation → .	Java platform default allowed cipher suites.
`tls_versions`	A comma-separated list of allowed TLS versions. By default only TLSv1.2 and TLSv1.3 are allowed. To use both TLSv1.2 and TLSv1.3 versions, you must specify which ciphers to be enforced for each version. Otherwise, Neo4j could use every possible cipher in the JVM for those versions, leading to a less secure configuration.	`TLSv1.2` `TLSv1.3`
`client_auth`	Whether or not clients must be authenticated. Setting this to `REQUIRE` enables mutual authentication for servers. Other possible values are `NONE` and `OPTIONAL`.	`OPTIONAL` for `bolt` and `https`; `REQUIRE` for `cluster` and `backup`.

1. In Neo4j 2025.01, the default value is changed from `false` to `true`.

Setting suffix	Description	Default value
`trust_all`	Setting this to `true` results in all clients and servers to be trusted and the content of the `trusted_dir` directory to be ignored. Use this only as a mean of debugging, since it does not offer security.	`false`

1. In Neo4j 2025.01, the default value is changed from `false` to `true`.

NOTE

For security reasons, Neo4j does not automatically create any of these directories. Therefore, the creation of an SSL policy requires the appropriate file system structure to be set up manually. Note that the existence of the directories, the certificate file, and the private key are mandatory. Ensure that only the Neo4j user can read the private key.

Each policy needs to be explicitly enabled by setting `dbms.ssl.policy.<scope>.enabled=true`.

Configure SSL over Bolt

Bolt protocol is based on the PackStream serialization and supports the Cypher type system, protocol versioning, authentication, and TLS via certificates. For Neo4j clusters, Bolt provides smart client routing with load balancing and failover. When server side routing is enabled, an additional Bolt port is open on `7688`. It can be used only within the cluster and with all the same settings as the external Bolt port.

Bolt connector is used by Cypher Shell, Neo4j Browser, and by the officially supported language drivers. Bolt connector is enabled by default but its encryption is disabled. To enable the encryption over Bolt, create the folder structure and place the key file and the certificates under those. Then, you need to configure the SSL Bolt policies in the *neo4j.conf* file.

1. Enable the Bolt connector to enable SSL over Bolt:

```
server.bolt.enabled=true (default is true)
```

2. Set up the *bolt* folder under *certificates*.

    a. Create a directory *bolt* under *<NEO4J_HOME>/certificates* folder:

```
mkdir certificates/bolt
```

    b. Create a directory *trusted* and *revoked* under *<NEO4J_HOME>/certificates/bolt* folder:

```
mkdir certificates/bolt/trusted
mkdir certificates/bolt/revoked
```

3. Place the certificates *private.key* and the *public.crt* files under *<NEO4J_HOME>/certificates/bolt* folder:

```
cp /path/to/certs/private.key certificates/bolt
cp /path/to/certs/public.crt certificates/bolt
```

4. Place the *public.crt* file under the *<NEO4J_HOME>/certificates/bolt/trusted* folder.

```
cp /path/to/certs/public.crt certificates/bolt/trusted
```

5. (Optional) If a particular certificate is revoked, then place it under *<NEO4J_HOME>/certificates/bolt/revoked* folder.

```
cp /path/to/certs/public.crt certificates/bolt/revoked
```

The folder structure should look like this with the right file permissions and the groups and ownerships:

Path	Directory/ File	Owner	Group	Permissio n	Unix/Linu x View
/data/neo4j/certificates/bolt	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/bolt/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/bolt/private.key	File	`neo4j`	`neo4j`	`0400`	`-r--------`
/data/neo4j/certificates/bolt/trusted	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/bolt/trusted/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/bolt/revoked	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

TIP

The owner/group should be configured to the user/group that will be running the `neo4j` service. Default user/group is neo4j/neo4j.

6. Set the Bolt SSL configuration in *neo4j.conf*.

   a. Set the SSL Bolt policy to `true`:

```
dbms.ssl.policy.bolt.enabled=true
```

   b. Set the appropriate certificates path and the right key and cert files:

```
dbms.ssl.policy.bolt.base_directory=certificates/bolt
dbms.ssl.policy.bolt.private_key=private.key
dbms.ssl.policy.bolt.public_certificate=public.crt
```

TIP

TIP

If the certificate is a different path outside of NEO4J_HOME, then set the absolute path for the certificates directory.

c. Set the Bolt client authentication to `NONE` to disable the mutual authentication:

```
dbms.ssl.policy.bolt.client_auth=NONE
```

d. Set the Bolt TLS level to allow the connector to accept encrypted and/or unencrypted connections:

```
server.bolt.tls_level=REQUIRED (default is DISABLED)
```

TIP

`REQUIRED` means the connector accepts only encrypted client connections and reject the unencrypted ones. `OPTIONAL` means the connector accepts either encrypted or unencrypted client connections.

7. Test the SSL connection to the specified host and Bolt port and view the certificate:

```
openssl s_client -connect my_domain.com:7687
```

Connect with SSL over Bolt

Each of the `neo4j` and `bolt` URI schemes permit variants that contain extra encryption and trust information. The `+s` variants enable encryption with a full certificate check. The `+ssc` variants enable encryption with no certificate check. This latter variant is designed specifically for use with self-signed certificates.

URI Scheme	Routing	Description
`neo4j`	Yes	Unsecured
`neo4j+s`	Yes	Secured with full certificate

URI Scheme	Routing	Description
`neo4j+ssc`	Yes	Secured with self-signed certificate
`bolt`	No	Unsecured
`bolt+s`	No	Secured with full certificate
`bolt+ssc`	No	Secured with self-signed certificate

Once SSL is enabled over Bolt, you can connect to the Neo4j DBMS using `neo4j+s` or `bolt+s`:

Cypher Shell

```
cypher-shell -a neo4j+s://<Server DNS or IP>:<Bolt port>

or

cypher-shell -a bolt+s://<Server DNS or IP>:<Bolt port>
```

Configure SSL over HTTPS

HTTP(s) is used by the Neo4j Browser and the HTTP API. HTTPS (secure HTTP) is set to encrypt network communications. To enable the encryption over HTTPS, create the folder structure and place the key file and the certificates under those. Then, you need to configure the SSL HTTPS policies in the *neo4j.conf* file and disable the HTTP connector.

NOTE

The HTTPS configuration requires that Bolt is also set. Refer to Configure SSL over Bolt for more instructions.

1. Enable the HTTPS connector to enable SSL over HTTPS:

```
server.https.enabled=true (default is false)
```

2. Set up the *https* folder under *certificates*.

a. Create a directory *https* under *<NEO4J_HOME>/certificates* folder:

```
mkdir certificates/https
```

b. Create a directory *trusted* and *revoked* under *<NEO4J_HOME>/certificates/https* folder:

```
mkdir certificates/https/trusted
mkdir certificates/https/revoked
```

3. Place the certificates *private.key* and the *public.crt* files under *<NEO4J_HOME>/certificates/https* folder:

```
cp /path/to/certs/private.key certificates/https
cp /path/to/certs/public.crt certificates/https
```

4. Place the *public.crt* file under the *<NEO4J_HOME>/certificates/https/trusted* folder.

```
cp /path/to/certs/public.crt certificates/https/trusted
```

5. (Optional) If a particular certificate is revoked, then place it under *<NEO4J_HOME>/certificates/https/revoked* folder.

```
cp /path/to/certs/public.crt certificates/https/revoked
```

The folder structure should look like this with the right file permissions and the groups and ownerships:

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/https	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/https/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/https/private.key	File	`neo4j`	`neo4j`	`0400`	`-r--------`
/data/neo4j/certificates/https/trusted	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/https/trusted/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/https/revoked	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

TIP

The owner/group should be configured to the user/group that will be running the `neo4j` service. Default user/group is neo4j/neo4j.

6. Set the HTTPS SSL configuration in *neo4j.conf*.

a. Set the SSL HTTPS policy to `true`:

```
dbms.ssl.policy.https.enabled=true
```

b. Set the appropriate certificates path and the right key and cert files:

```
dbms.ssl.policy.https.base_directory=certificates/https
dbms.ssl.policy.https.private_key=private.key
dbms.ssl.policy.https.public_certificate=public.crt
```

TIP

If the certificate is a different path outside of NEO4J_HOME, then set the absolute path for the certificates directory.

c. Set the HTTPS client authentication to `NONE` to disable the mutual authentication:

```
dbms.ssl.policy.https.client_auth=NONE
```

d. Disable HTTP connector:

```
server.http.enabled=false
```

7. Test the SSL connection to the specified host and HTTPS port and view the certificate:

```
openssl s_client -connect my_domain.com:7473
```

Configure SSL for intra-cluster communications

Intra-cluster encryption is the security solution for the cluster communication. The Neo4j cluster communicates on 3 ports:

- 6000 - Transactions
- 7000 - Raft communications
- 7688 - Server side routing

To set up intra-cluster encryption, on each server create the folder structure and place the key file and the certificates under those. Then, you need to configure the SSL cluster policies in the *neo4j.conf* file and test that the intra-cluster communication is encrypted.

1. Set up the *cluster* folder under *certificates*.

a. Create a directory *cluster* under_<NEO4J_HOME>/certificates_ folder:

```
mkdir certificates/cluster
```

b. Create a directory *trusted* and *revoked* under *<NEO4J_HOME>/certificates/cluster* folder:

```
mkdir certificates/cluster/trusted
mkdir certificates/cluster/revoked
```

2. Place the certificates *private.key* and the *public.crt* files under *<NEO4J_HOME>/certificates/cluster* folder:

```
cp /path/to/certs/private.key certificates/cluster
cp /path/to/certs/public.crt certificates/cluster
```

3. Place the *public.crt* file under the *<NEO4J_HOME>/certificates/cluster/trusted* folder.

```
cp /path/to/certs/public.crt certificates/cluster/trusted
```

TIP

If each server has a certificate of its own, signed by a CA, then each server's public certificate has to be put in the *trusted* folder on each instance of the cluster. Thus, the servers are able to establish trust relationships with each other.

4. (Optional) If a particular certificate is revoked, then place it under *<NEO4J_HOME>/certificates/cluster/revoked* folder.

```
cp /path/to/certs/public.crt certificates/cluster/revoked
```

The folder structure should look like this with the right file permissions and the groups and ownerships:

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/cluster	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/cluster/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/cluster/private.key	File	`neo4j`	`neo4j`	`0400`	`-r--------`
/data/neo4j/certificates/cluster/trusted	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/cluster/trusted/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/cluster/revoked	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

TIP

The owner/group should be configured to the user/group that will be running the `neo4j` service. Default user/group is neo4j/neo4j.

5. Set the cluster SSL configuration in *neo4j.conf*.

a. Set the cluster SSL policy to `true`:

```
dbms.ssl.policy.cluster.enabled=true
```

b. Set the appropriate certificates path and the right key and cert files:

```
dbms.ssl.policy.cluster.base_directory=certificates/cluster
dbms.ssl.policy.cluster.private_key=private.key
dbms.ssl.policy.cluster.public_certificate=public.crt
```

TIP

If the certificate is a different path outside of NEO4J_HOME, then set the absolute path for the certificates directory.

c. Set the cluster client authentication to `REQUIRE` to enable the mutual authentication, which means that both ends of a channel have to authenticate:

```
dbms.ssl.policy.cluster.client_auth=REQUIRE
```

NOTE

The policy must be configured on every server with the same settings. The actual cryptographic objects installed will mostly be different since they do not share the same private keys and corresponding certificates. The trusted CA certificate will be shared however.

d. Verify that the intra-cluster communication is encrypted. You may use an external tooling, such as Nmap (https://nmap.org/download.html →):

```
nmap --script ssl-enum-ciphers -p <port> <hostname>
```

NOTE

The hostname and port have to be adjusted according to your configuration. This can prove that TLS is enabled and that only the intended cipher suites are enabled. All servers and all applicable ports should be tested. If the intra-cluster encryption is enabled, the output should indicate the port is open and it is using TLS with the ciphers used.

TIP

For more details on securing the comunication between the cluster servers, see Intra-cluster encryption.

Configure SSL for backup communication

In a single instance, by default the backup communication happens on port `6362`. In a cluster topology, it is possible to take a backup from any server, and each server has two configurable ports capable of serving a backup. These ports are configured by `dbms.backup.listen.address` (port `6362`) and `server.cluster.listen_address` (port `6000`) respectively. If the intra-cluster encryption is enabled and the backup communication is using port `6000`, then your communication channels are already encrypted. The following steps assumes that your backup is set up on a different port.

To set up SSL for backup communication, create the folder structure and place the key file and the certificates under those. Then, you need to configure the SSL backup policies in the *neo4j.conf* file.

1. Set up the *backup* folder under *certificates*.

a. Create a directory *backup* under *<NEO4J_HOME>/certificates* folder:

```
mkdir certificates/backup
```

b. Create a directory *trusted* and *revoked* under *<NEO4J_HOME>/certificates/backup* folder:

```
mkdir certificates/backup/trusted
mkdir certificates/backup/revoked
```

2. Place the certificates *private.key* and the *public.crt* files under *<NEO4J_HOME>/certificates/backup* folder:

```
cp /path/to/certs/private.key certificates/backup
cp /path/to/certs/public.crt certificates/backup
```

3. Place the *public.crt* file under the *<NEO4J_HOME>/certificates/backup/trusted* folder.

```
cp /path/to/certs/public.crt certificates/backup/trusted
```

4. (Optional) If a particular certificate is revoked, then place it under *<NEO4J_HOME>/certificates/backup/revoked* folder.

```
cp /path/to/certs/public.crt certificates/backup/revoked
```

The folder structure should look like this with the right file permissions and the groups and ownerships:

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/backup	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

Path	Directory/ File	Owner	Group	Permission	Unix/Linux View
/data/neo4j/certificates/backup/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/backup/private.key	File	`neo4j`	`neo4j`	`0400`	`-r--------`
/data/neo4j/certificates/backup/trusted	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`
/data/neo4j/certificates/backup/trusted/public.crt	File	`neo4j`	`neo4j`	`0644`	`-rw-r--r--`
/data/neo4j/certificates/backup/revoked	Directory	`neo4j`	`neo4j`	`0755`	`drwxr-xr-x`

TIP

The owner/group should be configured to the user/group that will be running the `neo4j` service. Default user/group is neo4j/neo4j.

5. Set the backup SSL configuration in *neo4j.conf*.

a. Set the backup SSL policy to `true`:

```
dbms.ssl.policy.backup.enabled=true
```

b. Set the appropriate certificates path and the right key and cert files:

```
dbms.ssl.policy.backup.base_directory=certificates/backup
dbms.ssl.policy.backup.private_key=private.key
dbms.ssl.policy.backup.public_certificate=public.crt
```

TIP

If the certificate is a different path outside of NEO4J_HOME, then set the absolute path for the certificates directory.

c. Set the backup client authentication to `REQUIRE` to enable the mutual authentication, which means that both ends of a channel have to authenticate:

```
dbms.ssl.policy.backup.client_auth=REQUIRE
```

Other configurations for SSL

Using encrypted private key

To use an encrypted private key, configure the following settings. The private key password must be clear text format without any quotes.

Bolt

```
dbms.ssl.policy.bolt.private_key_password=<clear text password>
```

HTTPS

```
dbms.ssl.policy.https.private_key_password=<password>
```

Intra-cluster encryption

```
dbms.ssl.policy.cluster.private_key_password=<password>
```

Backup

```
dbms.ssl.policy.backup.private_key_password=<password>
```

If hardcoding of clear text private key password is not feasible due to security constraints, it can be set up to use dynamic password pickup by following these steps:

1. Create a file containing the `cleartext` password for the private key password and encrypt it with the certificate (assuming private key for cert has password set and certificate is in `pwd`):

```
echo "password123" > passwordfile

base64 -w 0 certificate.crt | openssl aes-256-cbc -a -salt -in passwordfile -out
password.enc -pass stdin
```

NOTE

Delete the password file and set file permissions for `password.enc` to `400` (e.g. `chmod 400 password.enc`).

2. Verify that encrypted password can be read from password.enc:

```
base64 -w 0 certificate.crt | openssl aes-256-cbc -a -d -in password.enc -pass stdin
```

3. Set the neo4j.conf `dbms.ssl.policy.<type>.private_key_password` to be able to read out encrypted password. To adjust paths to cert and encrypted password file, use full paths:

```
dbms.ssl.policy.bolt.private_key_password=$(base64 -w 0 certificate.crt | openssl aes-256-
cbc -a -d -in password.enc -pass stdin)
```

NOTE

Using a dynamic command requires Neo4j to be started with the `--expand-commands` option. For more information, see Command expansion.

Using specific cipher

There are cases where Neo4j Enterprise requires the use of specific ciphers for encryptions. One can set up a Neo4j configuration by specifying the list of cipher suites that will be allowed during cipher negotiation. Valid values depend on the current JRE and SSL provider. For Oracle JRE here is the list of supported ones - https://docs.oracle.com/en/java/javase/21/docs/specs/security/standard-names.html#jsse-cipher-suite-names →.

Note that CBC (cipher block chaining)-based ciphers (RFC 8447), used in TLS v1.2 network encryption, are not supported in 2025.01. See Changes, deprecations, and removals in Neo4j 2025.x for more information.

Bolt

```
dbms.ssl.policy.bolt.ciphers=TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

HTTPS

```
dbms.ssl.policy.https.ciphers=TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

Intra-cluster encryption

```
dbms.ssl.policy.cluster.ciphers=TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

Backup

```
dbms.ssl.policy.backup.ciphers=TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

Using OCSP stapling

Neo4j supports OCSP stapling, which is implemented on the server side, and can be configured in the *neo4j.config* file. OCSP stapling is also available for Java Bolt driver and HTTP API.

On the server side in the *neo4j.conf* file, configure the following settings:

1. Set the SSL Bolt policy to `true`:

```
dbms.ssl.policy.bolt.enabled=true
```

2. Enable the OCSP stapling for Bolt:

```
server.bolt.ocsp_stapling_enabled=true (default = false)
```

SSL logs

All information related to SSL can be found in the *debug.log* file. You can also enable additional debug logging for SSL by adding the following configuration to the *neo4j.conf* file and restarting Neo4j.

```
server.jvm.additional=-Djavax.net.debug=ssl:handshake
```

This logs additional information in the *neo4j.log* file. In some installations, for example, RPM-based installs, *neo4j.log* is not created. To get the contents of this, since *neo4j.log* just contains `STDOUT` content, look for the `neo4j` service log contents using `journalctl`:

```
neo4j@ubuntu:/var/log/neo4j$ journalctl -u neo4j -b > neo4j.log
neo4j@ubuntu:/var/log/neo4j$ vi neo4j.log
```

WARNING

Beware that the SSL debug option logs a new statement every time a client connects over SSL, which can make *neo4j.log* grow large reasonably quickly. To avoid that scenario, make sure this setting is only enabled for a short term duration.

Certificates rotation

Introduced in 2025.03

It is considered best practice to use certificates with reasonably short duration. This, however, requires the periodic rotation of certificates whereby old certificates are removed and the new ones are installed. Previous versions of Neo4j required a database restart for changes to be applied. Starting from 2025.03, new certificates can be rotated in, and SSL configuration can be updated without requiring a restart. This reduces undesirable effects of transient loss of cluster members.

The following steps outline the process for certificates rotation.

1. Enable the dynamic reloading of certificates on all cluster members. It is best to do this when the cluster is deployed as changing this configuration requires a restart:

```
dbms.security.tls_reload_enabled=true (default is false)
```

2. Replace old certificates either by overwriting them on the filesystem or by copying them to a new location.

   Keep in mind that if you choose to copy the certificates to a new directory or use different filenames, you must dynamically update the SSL policy settings. If you are overwriting the certificates in place and not changing anything else, there is no need to dynamically update the SSL policy settings.

   New and old certificates may co-exist on the filesystem, but only one can be referenced in the configuration.
   It is required to copy new certificates to all cluster members.

3. Make necessary changes to any of the SSL configuration and/or replace certificates for affected scopes.
4. Connect to each cluster member in turn with Cypher Shell using a `bolt` URI scheme and run the reload procedure:

```
dbms.security.reloadTLS()
```

5. New settings will take effect immediately; however, existing connections will not be preemptively terminated. This means that while new connections will use new certificates, the existing connections (established before the update) will continue using the old certificates. Even if a certificate expires, active connections remain unaffected because the certificates are only used during the initial connection handshake.
6. Verify that the intra-cluster communication is still encrypted using external tooling, such as Nmap, described in Configuring SSL for intra-cluster communications.

Terminology

The following terms are relevant to SSL support within Neo4j:

Certificate Authority (CA)

A trusted entity that issues electronic documents that can verify the identity of a digital entity. The term commonly refers to globally recognized CAs, but can also include internal CAs that are trusted inside of an organization. The electronic documents are digital certificates. They are an essential part of secure communication, and play an important part in the Public Key Infrastructure.

Certificate Revocation List (CRL)

In the event of a certificate being compromised, that certificate can be revoked. This is done by means of a list (located in one or several files) spelling out which certificates are revoked. The CRL is always issued by the CA which issues the corresponding certificates.

cipher

An algorithm for performing encryption or decryption. In the most general implementation of encrypted communications, Neo4j makes implicit use of ciphers that are included as part of the Java platform. The configuration of the SSL framework also allows for the explicit declaration of allowed ciphers.

communication channel

A means for communicating with the Neo4j database. Available channels are:

- Bolt client traffic
- HTTPS client traffic
- intra-cluster communication
- backup traffic

cryptographic objects

A term denoting the artifacts private keys, certificates and CRLs.

configuration parameters

These are the parameters defined for a certain ssl policy in *neo4j.conf*.

certificate

SSL certificates are issued by a trusted certificate authority (*CA*). The public key can be obtained and used by anyone to encrypt messages intended for a particular recipient. The certificate is commonly stored in a file named *<file name>.crt*. This is also referred to as the public key.

SAN

SAN is an acronym for *Subject Alternative Names*. It is an extension to certificates that one can include optionally. When presented with a certificate that includes SAN entries, it is recommended that the address of the host is checked against this field. Verifying that the hostname matches the certificate SAN helps prevent attacks where a rogue machine has access to a valid key pair.

SSL

SSL is an acronym for *Secure Sockets Layer*, and is the predecessor of TLS. It is common to refer to SSL/TLS as just SSL. However, the modern and secure version is TLS, which is also the default in Neo4j.

SSL policy

An SSL policy in Neo4j consists of a a digital certificate and a set of configuration parameters defined in *neo4j.conf*.

PKCS #1

PKCS #1 is the first family of standards called Public-Key Cryptography Standards (PKCS). It provides the basic definitions and recommendations for implementing the RSA algorithm for public-key cryptography. It defines the mathematical properties of public and private keys, primitive operations for encryption and signatures, secure cryptographic schemes, and related ASN.1 syntax representations.

PKCS #5

PKCS #5 contains recommendations for implementating password-based cryptography, covering key derivation functions, encryption schemes, message authentication schemes, and *ASN.1* syntax, identifying the techniques.

PKCS #8

PKCS #8 is a standard syntax for storing private key information. The PKCS #8 private key may be encrypted with a passphrase using the PKCS #5 standards, which support multiple ciphers. The main difference from PKCS #1 is that it allows more algorithms than RSA and supports stronger encryption of the private key.

private key

The private key ensures that encrypted messages can be deciphered only by the intended recipient. The private key is commonly stored in a file named *<file name>.key*. It is important to protect the private key to ensure the integrity of encrypted communication.

Public Key Infrastructure (PKI)

A set of roles, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption.

public key

The public key can be obtained and used by anyone to encrypt messages intended for a particular recipient. This is also referred to as the certificate.

TLS protocol

The cryptographic protocol that provides communications security over a computer network. The Transport Layer Security (TLS) protocol and its predecessor, the Secure Sockets Layer (SSL) protocol, are both frequently referred to as "SSL".

TLS version

A version of the TLS protocol.

X.509

X.509 is an International Telecommunication Union (ITU) standard defining the format of public key certificates.

Prev
‹ Securing extensions

Next
Configuring SSL for FIPS 140-2 compatibility ›

Contents