Table of Contents ⊞

5.4. Intra-cluster encryption Enterprise Edition

This chapter describes how to secure the cluster communication between server instances.

This section includes:

- Introduction
- Example deployment
  - Generate and install cryptographic objects
  - Create an SSL policy
  - Configure Causal Clustering with the SSL policy
  - Validate the secure operation of the cluster

5.4.1. Introduction

The security solution for cluster communication is based on standard SSL/TLS technology (referred to jointly as SSL). Encryption is in fact just one aspect of security, with the other cornerstones being authentication and integrity. A secure solution will be based on a key infrastructure which is deployed together with a requirement of authentication.

The SSL support in the platform is documented in detail in Section 9.2, “SSL framework” (../../security/ssl-framework/). This section will cover the specifics as they relate to securing a cluster.

Under SSL, an endpoint can authenticate itself using certificates managed by a Public Key Infrastructure (*PKI*) (../../security/ssl-framework/#term-ssl-pki).

It should be noted that the deployment of a secure key management infrastructure is beyond the scope of this manual, and should be entrusted to experienced security professionals. The example deployment illustrated below is for reference purposes only.

5.4.2. Example deployment

The following steps will create an example deployment, and each step is expanded in further detail below.

- Generate and install cryptographic objects (../../security/ssl-framework/#term-ssl-cryptographic-objects)
- Create an SSL policy
- Configure Causal Clustering with the SSL policy
- Validate the secure operation of the cluster

5.4.2.1. Generate and install cryptographic objects

The generation of cryptographic objects is for the most part outside the scope of this manual. It will generally require having a PKI with a Certificate Authority (*CA*) (../../security/ssl-framework/#term-ssl-certificate-authority) within the organization and they should be able to advise here. Please note that the information in this manual relating to the PKI is mainly for illustrative purposes.

When the certificates and private keys have been obtained they can be installed on each of the servers. Each server will have a certificate of its own, signed by a CA, and the corresponding private key. The certificate of the CA is installed into the `trusted` directory, and any certificate signed by the CA will thus be trusted. This means that the server now has the capability of establishing trust with other servers.

Please be sure to exercise caution when using CA certificates in the `trusted` directory, as any certificates signed by that CA will then be trusted to join the cluster. For this reason, never use a public CA to sign certificates for your cluster. Instead, use an intermediate certificate or a CA certificate which originates from and is controlled by your organization.

In this example we will deploy a mutual authentication setup, which means that both ends of a channel have to authenticate. To enable mutual authentication the SSL policy must have `client_auth` set to `REQUIRE` (which is the default). Servers are by default required to authenticate themselves, so there is no corresponding server setting.

Neo4j is able to generate self-signed certificates but a deployment using these should generally be regarded as a special case. It can make sense in some circumstances and even a mutual authenticated setup can be created by sharing the self-generated certificates among instances.

If the certificate for a particular server is compromised it is possible to revoke it by installing a Certificate Revocation List (*CRL*) (../../security/ssl-framework/#term-ssl-certificate-revocation-list) in the `revoked` directory. It is also possible to redeploy using a new CA. For contingency purposes, it is advised that you have a separate intermediate CA specifically for the cluster which can be substituted in its entirety should it ever become necessary. This approach would be much easier than having to handle revocations and ensuring their propagation.

Example 5.7. Generate and install cryptographic objects

In this example we assume a plan to name the SSL policy `cluster`, and that the private key and certificate file are named *private.key* and *public.crt*, respectively. If you want to use different names you may override the policy configuration for the key and certificate names/locations. We want to use the default configuration for this server so we create the appropriate directory structure and install the certificate:

```
$neo4j-home> mkdir certificates/cluster
$neo4j-home> mkdir certificates/cluster/trusted
$neo4j-home> mkdir certificates/cluster/revoked

$neo4j-home> cp $some-dir/private.key certificates/cluster
$neo4j-home> cp $some-dir/public.crt certificates/cluster
```

5.4.2.2. Create an SSL policy

An SSL policy utilizes the installed cryptographic objects and additionally allows parameters to be configured. We will use the following parameters in our configuration:

Table 5.2. Example settings

Setting suffix	Value	Comment
`client_auth`	`REQUIRE`	Setting this to `REQUIRE` effectively enables mutual authentication for servers.
`ciphers`	`TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`	We can enforce a particular single strong cipher and remove any doubt about which cipher gets negotiated and chosen. The cipher chosen above offers Perfect Forward Secrecy (PFS) which is generally desirable. It also uses Advanced Encryption Standard (*AES*) for symmetric encryption which has great support for acceleration in hardware and thus allows performance to generally be negligibly affected.
`tls_versions`	`TLSv1.2`	Since we control the entire cluster we can enforce the latest TLS standard without any concern for backwards compatibility. It has no known security vulnerabilities and uses the most modern algorithms for key exchanges, etc.

In the following example we will create and configure an SSL policy that we will use in our cluster.

Example 5.8. Create an SSL policy

In this example we assume that the directory structure has been created, and certificate files have been installed, as per the previous example. We wish to create a cluster SSL policy called `cluster`. We do this by defining the following parameter.

```
dbms.ssl.policy.cluster.base_directory=certificates/cluster
```

Since our policy is named `cluster` the corresponding settings will be available to configure under the `dbms.ssl.policy.cluster.*` group. We add the following content to our *neo4j.conf* file:

```
dbms.ssl.policy.cluster.tls_versions=TLSv1.2
dbms.ssl.policy.cluster.ciphers=TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
dbms.ssl.policy.cluster.client_auth=REQUIRE
```

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

5.4.2.3. Configure Causal Clustering with the SSL policy

There is no default SSL policy for Causal Clusters. This means that by default, cluster communication is unencrypted. To configure a Causal Cluster to encrypt its intra-cluster communication, set `causal_clustering.ssl_policy (../../reference/configuration-settings/#config_causal_clustering.ssl_policy)` to the name of a valid SSL policy.

Example 5.9. Configure Causal Clustering with the SSL policy

In this example we assume that the tasks in the previous two examples have been performed. We now configure our cluster to use the SSL policy that we have named `cluster`.

```
causal_clustering.ssl_policy=cluster
```

Any user data communicated between instances will now be secured. Please note that an instance which is not correctly setup would not be able to communicate with the others.

5.4.2.4. Validate the secure operation of the cluster

To make sure that everything is secured as intended it makes sense to validate using external tooling such as, for example, the open source assessment tools `nmap` or `OpenSSL`.

Example 5.10. Validate the secure operation of the cluster

In this example we will use the `nmap` tool to validate the secure operation of our cluster. A simple test to perform is a cipher enumeration using the following command:

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

The hostname and port have to be adjusted according to our configuration. This can prove that TLS is in fact enabled and that the only the intended cipher suites are enabled. All servers and all applicable ports should be tested.

For testing purposes we could also attempt to utilize a separate testing instance of Neo4j which, for example, has an untrusted certificate in place. The expected result of this test is that the test server is not able to participate in replication of user data. The debug logs will generally indicate an issue by printing an SSL or certificate-related exception.