Security Guide for DataStax Distribution of Apache Cassandra 3.11 Latest DDAC patch: 5.1.17

Updated: 2020-02-15Z
Contents

Chapter 1. Securing the database

Chapter 2. Internal authentication
  About Internal authentication
  Configuring authentication
  Using cqlsh with authentication

Chapter 3. Internal authorization
  Object permissions
  Configuring internal authorization

Chapter 4. JMX authentication and authorization
  JMX Authentication and Authorization
  Enabling JMX authentication and authorization
  Using nodetool with authentication
  Using jconsole with authentication

Chapter 5. SSL encryption
  Encrypting database connections with SSL
  Installing Java Cryptography Extension (JCE) Files
  Preparing server certificates for development
  Preparing SSL certificates for production
  Node-to-node encryption
  Client-to-node encryption
  Using cqlsh with SSL
  Using nodetool (JMX) with SSL encryption
  Using jconsole (JMX) with SSL encryption

Chapter 6. Configuring firewall port access
Chapter 1. Securing the database

DataStax offers subscription-based support (Luna) for open-source Cassandra. Learn more.

Security features include:

The DataStax Distribution of Apache Cassandra™ (DDAC) uses the security capabilities present in Apache Cassandra™, not DSE Advanced Security.

- **Authentication based on internally controlled rolename/passwords**
  Authentication is roles-based and stored internally in tables. Administrators can create, alter, drop, or list roles using CQL commands. Roles have superuser, login, and password properties.

- **Authorization based on object permission management**
  Authorization provides access control on database resources to authenticated roles. Authorization can grant permission to access the entire database or restrict a role to individual table access. Roles can be granted to roles to create a permissions hierarchy and separate login roles from sets of permissions. Use the CQL GRANT and REVOKE commands to manage authorization.

- **Authentication and authorization based on JMX username/passwords**
  JMX (Java Management Extensions) technology provides a simple and standard way of managing and monitoring resources related to an instance of a Java Virtual Machine (JVM). This is achieved by instrumenting resources with Java objects known as Managed Beans (MBeans) that are registered with an MBean server. JMX authentication stores username and associated passwords in two files, one for passwords and one for access. JMX authentication is used by `nodetool` and external monitoring tools such as `jconsole`.

- **SSL encryption**
  Secure communication between a client and a database cluster, and between nodes in a cluster. Enabling SSL encryption ensures that data in flight is not compromised and is transferred securely. Client-to-node and node-to-node encryption are independently configured. Tools (cqlsh, nodetool) can be configured to use SSL encryption. The DataStax drivers can be configured to secure traffic between the driver and the database.
Chapter 2. Internal authentication

The database authentication uses login roles and stores the passwords internally. Administrators can create, alter, drop, or list roles using CQL commands. Roles have superuser, login, and password properties.

About Internal authentication

Internal authentication provides the following types of roles:

- **Login roles**: Validate a user’s identity (password) and control access to database resources
- **Permission set (non-login) roles**: Allows assignment of the same privileges to one or more users (login roles). A role access permissions are inherited but the role options (superuser, login and password) are not. The database supports nesting roles to create a permission hierarchy.

The diagram below shows four login accounts with four group roles.

Figure 1: Roles assigned to individuals and functions

Setting up the administrator accounts

Use the following steps to set up user logins alice and fred and grant them administrator access to the cycling keyspace.

1. Create login accounts for each user. For example, *alice* is created with the ability to login and given a password:

```
CREATE ROLE alice WITH PASSWORD = 'enjoyLife' AND LOGIN = true;
```
CREATE ROLE fred WITH PASSWORD = 'enjoyLife' AND LOGIN = true;

The user enters the role name and password to log into the database. However at this point the role only has default access to some system tables.

Roles options include superuser, login, and password options. A superuser can perform any database operations.

2. Create a role with access to all the functionality for the cycling keyspace. The first command creates the role and the second assigns the privileges:

CREATE ROLE cycling_admin
   WITH PASSWORD = '1234abcd';

   GRANT ALL PERMISSIONS
   ON KEYSPACE cycling
   TO cycling_admin;

When the cycling_admin role is assigned to a login role, all the permissions are inherited. This gives any user that is a member of the cycling_admin role full access to the cycling keyspace.

3. Assign the cycling_admin role to alice and fred:

GRANT cycling_admin TO alice;
GRANT cycling_admin TO fred;

When alice is granted the role cycling_admin, alice is now granted all permissions on the keyspace cycling. Individual user can be granted any number of roles, just as any functional role can be granted another role's permissions.

In this example, the role cycling_analyst has the ability to select data, and then gains the ability to select data in another table hockey when the role hockey_analyst is granted.

CREATE ROLE cycling_analyst WITH PASSWORD = 'zyxw9876';
CREATE ROLE hockey_analyst WITH PASSWORD = 'Iget2seeAll';

GRANT SELECT ON TABLE cycling.analysis TO cycling_analyst;
GRANT SELECT ON TABLE hockey.analysis TO hockey_analyst;
GRANT hockey_analyst TO cycling_analyst;
GRANT cyclist_analyst TO jane;

If a user then is granted the role of cycling_analyst role, that user is able to select data in the additional table hockey:

Permissions and SUPERUSER status are inherited, but the LOGIN privilege is not.

An important change that roles-based access control also introduces is that the need for SUPERUSER privileges in order to perform user/role management operations is removed. A role can be authorized to create roles or be authorized to grant and revoke permissions:

// Give cycling_accounts the right to create roles
GRANT CREATE ON ALL ROLES TO cycling_accounts;
// Give cycling_accounts the right to create or revoke permissions
GRANT AUTHORIZE ON KEYSSPACE cycling TO cycling_accounts;
GRANT cyclist_accounts TO jane;
GRANT cyclist_accounts TO john;

Internal authentication and authorization information is stored in the following tables:
**Internal authentication**

**system_auth.roles**
Table that stores the role name, whether the role can be used for login, whether the role is a superuser, what other roles the role may be a member of, and a bcrypt salted hash password for the role.

**system_auth.role_members**
Table that stores the roles and role members.

**system_auth.role_permissions**
Table that stores the role, a resource (keyspace, table), and the permission that the role has to access the resource.

**system_auth.role_permissions_index**
Table that stores the role and a resource that the role has a set permission.

The database has a default superuser role and password pair of **cassandra/cassandra** by default. Only use this role to create your own superuser account because all request from the default account run with a consistency level of **QUORUM**. To secure the system, delete the default account after creating a superuser account.

Once roles and passwords have been set, the database can be configured to use authentication in the **cassandra.yaml** file.

After authentication is enabled, specify a username and password when using database tools and drivers. For details refer to the following sections:

- Using cqlsh with authentication
- Using nodetool with authentication
- Using jconsole with authentication
- DataStax drivers - produced and certified by DataStax to work with DataStax Distribution of Apache Cassandra™ (DDAC).

## Configuring authentication

Steps for configuring authentication in the DataStax Distribution of Apache Cassandra™ (DDAC).

1. Change the authenticator option in the **cassandra.yaml** file to **PasswordAuthenticator**:

   ```yaml
   authenticator: PasswordAuthenticator
   ```

   By default, the authenticator option is set to **AllowAllAuthenticator**.

2. Restart the database.

3. Start cqlsh using the default superuser name and password:

   ```bash
   $ cqlsh -u cassandra -p cassandra
   ```

4. To ensure that the keyspace is always available, increase the replication factor for the **system_auth** keyspace to 3 to 5 nodes per datacenter (recommended):

   ```cql
   ALTER KEYSPACE "system_auth"
   WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 4};
   ```

   The **system_auth** keyspace uses a QUORUM consistency level when checking authentication for the default **cassandra** user. For all other users created, superuser or otherwise, a LOCAL_ONE consistency level is used for authenticating.

   Datacenter names are case sensitive. Verify the case of the using utility, such as **nodetool status**.
Internal authentication

If the single replica of the keyspace goes down, using the default replication factor of 1 set for the `system_auth` keyspace results in denial of access to the cluster. For multiple datacenters, be sure to set the replication class to `NetworkTopologyStrategy`.

5. After increasing the replication factor of a keyspace, run `nodetool repair` to make certain the change is propagated:

```bash
$ nodetool repair system_auth
```

6. **Restart the database.**

7. Start `cqlsh` using the superuser name and password:

```bash
$ cqlsh -u cassandra -p cassandra
```

8. To prevent security breaches, replace the default superuser, `cassandra`, with another superuser with a different name:

```bash
CREATE ROLE <new_super_user> WITH PASSWORD = '<some_secure_password>'
    AND SUPERUSER = true
    AND LOGIN = true;
```

The default user `cassandra` reads with a consistency level of QUORUM by default, whereas another superuser reads with a consistency level of LOCAL_ONE.

9. Log in as the newly created superuser:

```bash
$ cqlsh -u <new_super_user> -p <some_secure_password>
```

10. To neutralize or delete the default account:

    - Neutralize the account by changing the password to something long and incomprehensible, and alter the user’s status to NOSUPERUSER:
      ```bash
      ALTER ROLE cassandra WITH PASSWORD='SomeNonsenseThatNoOneWillThinkOf'
      AND SUPERUSER=false;
      ```

    - Delete the account by logging in with the new super account created in the previous step and then running DROP ROLE:
      ```bash
      DROP ROLE cassandra;
      ```

11. Once you create some new roles, you can **authorize those roles** to access database objects.

12. Fetching role authentication can be a costly operation. To decrease the burden, adjust the validity period for role caching with the [https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra.yaml.html#configCassandra.yaml__roles_validity_in_ms](https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra.yaml.html#configCassandra.yaml__roles_validity_in_ms) option in the `cassandra.yaml` file (default 2000 milliseconds):

    ```bash
    roles_validity_in_ms: 2000
    ```

    To disable, set this option to 0. This setting is automatically disabled when the authenticator is set to `AllowAllAuthenticator`.

configCassandra_yaml.html#configCassandra_yaml__roles_update_interval_in_ms option in the cassandra.yaml file (default 2000 ms):

roles_update_interval_in_ms: 2000

If roles_validity_in_ms is non-zero, this setting must be set.

The credentials are cached in their encrypted form.

14. Fetching credentials authentication can be a costly operation. To decrease the burden, adjust the validity period for credential caching with the https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__credentials_update_interval_in_ms option in the cassandra.yaml file (default 2000 ms):

credentials_validity_in_ms: 2000

To disable, set this option to 0. This setting is automatically disabled when the authenticator is set to AllowAllAuthenticator.

15. To set the refresh interval for credentials caches, use the https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__credentials_update_interval_in_ms option (default 2000 ms):

credentials_update_interval_in_ms: 2000

If credentials_validity_in_ms is non-zero, this setting must be set.

16. To disable configuration of authentication and authorization caches (credentials, roles, and permissions) via JMX, uncomment the following line in the jvm.options file:

#-Dcassandra.disable_auth_caches_remote_configuration=true

After setting this option, cache options can only be set in the cassandra.yaml file. To make the new setting take effect, restart the database.

Using cqlsh with authentication

Typically, after configuring authentication, logging into cqlsh requires the -u and -p options to the cqlsh command. To set credentials for use when launching cqlsh, create or modify the .cassandra/cqlshrc file. When present, this file passes default login information to cqlsh. The cqlshrc.sample file provides an example.

1. Create or modify the cqlshrc file that specifies a role name and password.

[authentication]
username = fred
password = !!bang!!$

Additional settings in the cqlshrc file are described in Creating and using the cqlshrc file.

2. Save the file in home/.cassandra directory and name it cqlshrc.
Internal authentication

3. Set permissions on the file to prevent unauthorized access, as the password is stored in plain text. The file must be readable by the user that starts cassandra.

   ```bash
   $ chmod 440 home/.cassandra/cqlshrc
   ```

4. Check the permissions on `home/.cassandra/cqlshrc_history` to ensure that plain text passwords are not compromised.
Chapter 3. Internal authorization

Authorization provides access control to database resources with authenticated roles. Authorization can grant permission to access the entire database or restrict a role to individual table access. Roles can be granted to roles to create a permissions hierarchy and separate login roles from sets of permissions. Use the CQL GRANT and REVOKE commands to manage authorization.

Object permissions

Object permissions may be assigned using the authorization mechanism for the following objects:

- keyspace
- table
- function
- aggregate
- roles
- MBeans

Authenticated roles with passwords stored in the database are authorized selective access. The permissions are stored in tables.

Permission is configurable for CQL commands CREATE, ALTER, DROP, SELECT, MODIFY, and DESCRIBE, which are used to interact with the database. The EXECUTE command may be used to grant permission to a role for the SELECT, INSERT, and UPDATE commands. In addition, the AUTHORIZE command may be used to grant permission for a role to GRANT, REVOKE, or AUTHORIZE another role's permissions.

Read access to these system tables is implicitly given to every authenticated user or role because the tables are used by most tools:

- system_schema.keyspaces
- system_schema.columns
- system_schema.tables
- system.local
- system.peers

Configuring internal authorization

CassandraAuthorizer is one of many possible IAuthorizer implementations. Its advantage is that it stores permissions in the system_auth.permissions table to support all authorization-related CQL statements.

To configure authentication, see Internal authentication.

1. In the cassandra.yaml file, change the authorizer setting to CassandraAuthorizer.

   ```yaml
   authorizer: CassandraAuthorizer
   ```

   You can use any authenticator except AllowAll.
Internal authorization

2. Increase the replication factor for the `system_auth` keyspace if not already configured.

3. Fetching role permissions can be a costly operation. Role permissions can be cached to decrease the burden. Adjust the validity period for permission caching by setting the [https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__permissions_validity_in_ms](https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__permissions_validity_in_ms) option in the `cassandra.yaml` file. The default value is 2000 milliseconds. The caching can be disabled by setting the option to 0. This setting is disabled automatically if the authorizer is set to `AllowAllAuthorizer`.

```yaml
permissions_validity_in_ms: 2000
```

4. A refresh interval for role caches can also be configured by setting the [https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__permissions_update_interval_in_ms](https://docs.datastax.com/en/ddac/doc/datastax_enterprise/config/configCassandra_yaml.html#configCassandra_yaml__permissions_update_interval_in_ms) option in the `cassandra.yaml` file. The default value is the same value as the `permissions_validity_in_ms` setting. If `permissions_validity_in_ms` is non-zero, this setting must be set.

```yaml
permissions_update_interval_in_ms: 2000
```

CQL supports these authorization statements:

- `GRANT`
- `LIST PERMISSIONS`
- `REVOKE`
Chapter 4. JMX authentication and authorization

JMX (Java Management Extensions) technology provides a simple and standard way of managing and monitoring resources related to an instance of a Java Virtual Machine (JVM). This is achieved by instrumenting resources with Java objects known as Managed Beans (MBeans) that are registered with an MBean server. JMX authentication stores username and associated passwords in two files, one for passwords and one for access. JMX authentication is used by `nodetool` and external monitoring tools such as `jconsole`.

JMX Authentication and Authorization

JMX authentication and authorization allows selective users to access JMX tools and JMX metrics. If usernames and passwords exist and the database is configured to use authentication and authorization, JMX tools must be executed with authentication and authorization options.

- `nodetool` with authentication
- `jconsole` with authentication

Enabling JMX authentication and authorization

By default, JMX security is disabled and accessible only from `localhost` without authentication as shown in the following lines from the `cassandra-env.sh` file:

```bash
if [ "$LOCAL_JMX" = "yes" ]; then
    JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT"
```

Configuring JMX authentication and authorization can be accomplished using local password and access files to set the usernames, passwords and access permissions.

These two methods work for remote authentication and authorization; the difference is just the location of the configuration settings in the `cassandra-env.sh` file. Local configuration is placed within the `if [ "$LOCAL_JMX" = "yes" ]; then` block in the file, whereas remote configuration is placed with the `else` block.

Authentication and authorization using local files

- By default, JMX security is disabled and accessible only from `localhost` as shown in the following lines from the `cassandra-env.sh` file:

  ```bash
  if [ "$LOCAL_JMX" = "yes" ]; then
    JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT"
  ```

Authentication and authorization with an internal login role

- By default, JMX security is disabled and accessible only from `localhost` as shown in the following lines from the `cassandra-env.sh` file:

  ```bash
  if [ "$LOCAL_JMX" = "yes" ]; then
    JVM_OPTS="$JVM_OPTS -Dcassandra.jmx.local.port=$JMX_PORT"
  ```

- Comment out the existing line and add or uncomment the following lines in either the local or remote block in the `cassandra-env.sh` file:

  ```bash
  JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.authenticate=true"
  ```
JMX authentication and authorization

- And comment out the following lines in the `cassandra-env.sh` file:

```bash
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.password.file=/etc/cassandra/jmxremote.password"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.access.file=/etc/cassandra/jmxremote.access"
```

- Change authentication in the `cassandra.yaml` file to `PasswordAuthenticator`.

```yaml
authenticator: PasswordAuthenticator
```

- Change authorization in the `cassandra.yaml` file to `CassandraAuthorizer`.

```yaml
authorizer: CassandraAuthorizer
```

- Restart the database to make the change effective.

- Check that `nodetool status` requires the username and password in order to execute. The command should fail without authentication if everything is configured correctly.

```bash
$ nodetool -u cassandra -pw cassandra status
```

- Configure **Authorization** to grant and revoke permissions to database objects, including MBeans.

- Change `$LOCAL_JMX` to `no`. Add the following lines in the remote block in the `cassandra-env.sh` file:

```bash
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.authenticate=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.password.file=/etc/cassandra/jmxremote.password"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.access.file=/etc/cassandra/jmxremote.access"
```

- Create a password file and add the user and password for JMX-compliant utilities, specifying the credentials for your environment. The default location of the password file in the `cassandra-env.sh` is `/etc/cassandra/jmxremote.password`.

```
cassandra cassandra
<new_superuser> <new_superuser_password>
<some_other_user> <some_other_user_password>
c controlRole someOtherHardToRememberPassword
```

The default superuser account is a security hazard! This account is used only for the purposes of illustration.

- The password file must be secured from unauthorized readers. Change the ownership of the `jmxremote.password` file to the user who starts `cassandra` and change permissions to read only:

```bash
$ chown cassandra:cassandra /etc/cassandra/jmxremote.password
$ chmod 400 /etc/cassandra/jmxremote.password
```

This example presumes that `cassandra` is run by the default user `cassandra`. 
JMX authentication and authorization

• Create an access file and enter the following information. The default location of the access file in the Cassandra environment file is `/etc/cassandra/jmxremote.access`.

```bash
cassandra readwrite
<new_superuser> readwrite
<some_other_user> readonly
controlRole readwrite
create javax.management.monitor.,javax.management.timer.
unregister
```

The default superuser account is a security hazard! This account is used only for the purposes of illustration.

The `readonly` permission allows the JMX client to read an MBean's attributes and receive notifications. The `readwrite` permission allows the JMX client to set attributes, invoke operations, and create and remove MBeans, in addition to reading an MBean's attributes and receiving notifications.

• The access file must be secured from unauthorized readers. Change the ownership of the `jmxremote.access` file to the user who starts Cassandra and change permissions to read only:

```bash
$ chown cassandra:cassandra /etc/cassandra/jmxremote.access
$ chmod 400 /etc/cassandra/jmxremote.access
```

This example presumes that Cassandra is run by the default user `cassandra`.

• Restart the database to make the change effective.

• Check that `nodetool status` requires the username and password in order to execute. The command should fail without authentication if everything is configured correctly.

```bash
$ nodetool status
```

• Run `nodetool status` with the `cassandra` user and password.

```bash
$ nodetool -u cassandra -pw cassandra status
```

Specifying JMX authentication on command line

• Generally, JMX settings are inserted into the Cassandra environment file. However, these options can be specified at the command line:

```bash
cassandra -Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.password.file=/etc/cassandra/jmxremote.password
```

Enabling SSL for use with JMX authentication

• Ensure the mandatory properties are set for SSL in the Cassandra environment file.

For one-way SSL:

```bash
JVM_OPTS="-Djvm_opts -Dcom.sun.management.jmxremote.ssl=true"
JVM_OPTS="-Djvm_opts -Dcom.sun.management.jmxremote.ssl.need.client.auth=false"
JVM_OPTS="-Djvm_opts -Djavax.net.ssl.keyStore=/path/to/keystore"
```
JMX authentication and authorization

JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.keyStorePassword=keystore-password"

For two-way SSL:

JVM_OPTS="${JVM_OPTS} -Dcom.sun.management.jmxremote.ssl=true"
JVM_OPTS="${JVM_OPTS} -Dcom.sun.management.jmxremote.ssl.need.client.auth=true"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.keyStore=/path/to/keystore"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.trustStore=/path/to/truststore"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.trustStorePassword=truststore-pass"

For more, see Using nodetool (JMX) with SSL encryption and Using jconsole (JMX) with SSL encryption.

If you run nodetool status without user and password when authentication and authorization are configured, you'll see an error similar to:

```
Exception in thread "main" java.lang.SecurityException: Authentication failed!
Credentials required
```
1. Run `nodetool` using a pre-configured JMX username and password for `<username>` and `<password>`:

   ```
   $ nodetool -u <username> -pw <password>
   ```

### Using jconsole with authentication

After configuring JMX authentication, using `jconsole` requires a username and password to complete the remote connection to the cluster. Use an appropriate username/password combination.

1. Start `jconsole` using a pre-configured JMX username and password for `<username>` and `<password>`:
Chapter 5. SSL encryption

The database provides secure communication between a client and a database cluster, and between nodes in a cluster. Enabling SSL encryption ensures that data in flight is not compromised and is transferred securely. Client-to-node and node-to-node encryption are independently configured. Tools (cqlsh, nodetool) can be configured to use SSL encryption. The DataStax drivers can be configured to secure traffic between the driver and the database.

Encrypting database connections with SSL

The Secure Socket Layer (SSL) is a cryptographic protocol used to secure communications between computers. For reference, see SSL in wikipedia. Data is encrypted during communication to prevent accidental or deliberate attempts to read the data.

Briefly, SSL works in the following manner. Two entities, either software or hardware, that are communicating with one another. The entities can be a client and node or peers in a cluster. These entities must exchange information to set up trust between them. Each entity that will provide such information must have a generated key that consists of a private key that only the entity stores and a public key that can be exchanged with other entities. If the client wants to connect to the server, the client requests the secure connection and the server sends a certificate that includes its public key. The client checks the validity of the certificate by exchanging information with the server, which the server validates with its private key. If a two-way validation is desired, this process must be carried out in both directions. Private keys and certificates are stored in the keystore and public keys are stored in the truststore. For systems using a Certificate Authority (CA), the truststore can store certificates signed by the CA for verification. Both keystores and truststores have passwords assigned, referred to as the keypass and storepass.

Use SSL encryption for:

- **Node-to-node encrypted communication**
  - Secures data passed between nodes in a cluster.

- **Client-to-node encrypted communication**
  - Secures data passed between a client program, such as cqlsh and nodetool, and the nodes in the cluster.

Installing Java Cryptography Extension (JCE) Files

Starting with Java 1.8.0_162, JCE unlimited policy is enabled by default. You no longer need to install the policy file in the JRE or set the security property crypto.policy.

Installing the JCE Unlimited Strength Jurisdiction Policy Files can ensure support for all encryption algorithms when using Oracle Java with SSL on Apache Cassandra™, and it highly recommended. The files must be installed on every node in the cluster.

Some of the cipher suites in the default cassandra.yaml are included only in the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. To ensure support for all encryption algorithms, install the JCE Unlimited Strength Jurisdiction Policy Files.

Install the JCE files using the appropriate method for your database installation:

Installing the JCE on RHEL-based systems

1. If necessary, install the EPEL repository:

   ```bash
   $ sudo yum install epel-release
   ```

2. Installing the JCE using the Oracle JAR files:
SSL encryption

a. Download the Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from Oracle Java SE download page under Additional Resources.

b. Unzip the downloaded file.

c. Copy local_policy.jar and US_export_policy.jar to the $JAVA_HOME/jre/lib/security directory to overwrite the existing jar files.

Installing the JCE on Debian-based systems
Install JCE using webupd8 PPA repository:

```bash
$ sudo apt-get install oracle-java8-unlimited-jce-policy
```

Preparing server certificates for development

To use SSL encryption for client-to-node encryption or node-to-node encryption, SSL certificates must be generated using keytool. If you generate the certificates for one type of encryption, you do not need to generate them again for the other; the same certificates are used for both. All nodes must have all the relevant SSL certificates on all nodes. A keystore contains private keys. The truststore contains SSL certificates for each node. The certificates in the truststore don’t require signing by a trusted and recognized public certification authority.

- Generate a private and public key pair on each node of the cluster. Use an alias that identifies the node. After executing the following command, you are prompted for the keystore password, dname (first and last name, organizational unit, organization, city, state, country), and key password. The dname should be generated with the CN value as the IP address or FQDN for the node.

  ```bash
  $ keytool -genkey -keyalg RSA -alias node0 -validity 36500 -keystore keystore.node0
  ```

  In this example, the value for `--validity` gives this key pair a validity period of 100 years. The default validity value for a key pair is 90 days.

  ```bash
  $ keytool -genkey -keyalg RSA -alias node0 -keystore keystore.node0 -storepass cassandra -keypass cassandra -dname "CN=172.31.10.22, OU=None, O=None, L=None, C=None"
  ```

- Export the public part of the certificate to a separate file.

  ```bash
  $ keytool -export -alias cassandra -file node0.cer -keystore .keystore
  ```

- Add the `node0.cer` certificate to the node0 truststore of the node using the keytool -import command.

  ```bash
  $ keytool -import -v -trustcacerts -alias node0 -file node0.cer -keystore truststore.node0
  ```

- cqlsh does not work with the certificate in the format generated. Use openssl to generate a PEM file of the certificate with no keys, `node0.cer.pem`, and a PEM file of the key with no certificate, `node0.key.pem`. First, the keystore is imported in PKCS12 format to a destination keystore, `node0.p12`, in the example. This is followed by the two commands that extract the two PEM files.

  ```bash
  $ keytool -importkeystore -srckeystore keystore.node0 -destkeystore node0.p12 -deststoretype PKCS12 -srcstorepass cassandra -deststorepass cassandra && openssl
  ```
SSL encryption

- For client-to-remote-node encryption or node-to-node encryption, use a copying tool such as `scp` to copy the `node0.cer` file to each node. Import the file into the truststore after copying to each node. The example imports the certificate for node0 into the truststore for node1.

  ```
  $ keytool -import -v -trustcacerts -alias node0 -file node0.cer -keystore truststore.node1
  ```

- Make sure keystore file is readable only to the database daemon and not by any user of the system.

- Check that the certificates exist in the keystore and truststore files using `keytool -list`. The example shows checking for the node1 certificate in the keystore file and for the node0 and node1 certificates in the truststore file.

  ```
  $ keytool -list -keystore keystore.node1
  $ keytool -list -keystore truststore.node1
  ```

- Import the user's certificate into every node's truststore using `keytool`:

  ```
  $ keytool -import -v -trustcacerts -alias username -file certificate_file -keystore .truststore
  ```

Preparing SSL certificates for production

To use SSL encryption for client-to-node encryption or node-to-node encryption, SSL certificates must be generated using `openssl` and `keytool`. To validate the certificates, a self-signed Certificate Authority (CA) can be generated for production use. The certificates generated using these instructions can be used for both internode and client-to-node encryption. For internode encryption, all nodes must have the truststore that provides the chain of trust for the CA. The certificates in the truststore can either be signed by the self-signed certificate authority used here or by a trusted and recognized public certificate authority.

Create a root CA certificate and key

1. Create the root CA certificate and key using `openssl req` using the certificate configuration file `gen_rootCa_cert.conf`.

   ```
   $ openssl req -config gen_rootCa_cert.conf -new -x509 -nodes -subj /CN=rootCa/OU=TestCluster/O=YourCompany/C=US/ -keyout rootCa.key -out rootCa.crt -days 365
   ```

   ```
   # gen_rootCa_cert.conf
   [ req ]
   distinguished_name = req_distinguished_name
   prompt = no
   output_password = myPass
   default_bits = 2048
   [ req_distinguished_name ]
   C = US
   O = YourCompany
   OU = TestCluster
   ```
### SSL Encryption

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-config</td>
<td>Configuration file to use</td>
</tr>
<tr>
<td>-new</td>
<td>Generate new certificate</td>
</tr>
<tr>
<td>-x509</td>
<td>Outputs a self-signed certificate</td>
</tr>
<tr>
<td>-nodes</td>
<td>If a private key is created, it is not encrypted</td>
</tr>
<tr>
<td>-subj</td>
<td>Set subject name when processing</td>
</tr>
<tr>
<td>-keyout</td>
<td>Specify the private key filename to write</td>
</tr>
<tr>
<td>-out</td>
<td>Specify the certificate filename to write</td>
</tr>
<tr>
<td>-days</td>
<td>Specify the number of days for which to certify the certificate</td>
</tr>
</tbody>
</table>

The resulting files are the rootCA certificate and the rootCa private key.

An example of the rootCa.crt file:

```
-----BEGIN CERTIFICATE-----
MIIDAQCgCCQWlFMAoGA1UEChMINzExFDQgCCQgGDAcGBMwCAYGZRSANBgkqhkiG9w0BAQsFADB
MCQwDQYJKoZIhvcNAQEFBQADgYDVQQIEwpMAQICMA0GA1UEChMKMTAeFw0yMDEyMDEyMDA1
MDAwWhcNMjIwMTExMTQwMDQxWhcNMzIwMTExMTQwMDQxIjAaMCcGA1UEBhMCV3NjQxMCcGA1UE
BhMCV2hlbGwCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMK4gjAeU9p4w1qVnW
fiDnVXBeY1J6QVvKuZyCXMm7A5b6X26m03DmQ2yO784v9BQ8LSa5uF0AYJx1SswR0jSTaBZ
wPi9a/N6T5xXeMTejx5tBbxhCGuw/N7QyDw72Bf6P5wUy0UbT883s0zU9xsBb3I0j01v7H
/hDkA3m4Gj2Jc2tLl3Oykz32ucxKza8u3xn4vQV9fT1zOGwCNSZj2GC1FlhP4U4V5zvIDz
ETs8oj4Zqz82hO3v8RE4ZQ18AzJ0hZU5D0sZ63X171Ik9vW/3OCx3X070+zK0iD8kZQ7O
nR9xT8496Jy40VtGmhoqJjealy2Eylj/03X3lns+8/2Ij72uyN4rsc5k+JLH7z7K7s+/z8
Nvak3cGwh+fT3qG9YJQkg1kxu+VvP+65N/T+Bqnu/pmtvAiu794OLEnLU70o230JInbX3
9w/hSfQ1iUg7H2lCw/A1pS5Wk75coM91iZ7Z1E10zF1nPLo8X6Qd1+8/509/E87xwOo
jgsym4oTzK7UDc8Zi0dvP0wKm3CQh8p3vQ0ZxZ9/vr98f/c2uJ89ul651Gp5N6cUoQ9a
A1yCjWvYF4fRCg1d7C2r54snpdq20yM5OOGdddO9jF4f2yMCsZyuBnGwO1tGG8BY5B+n
x+9q4ZKbX6zTY0nGDjvghC22n82B5tIj6Bh0oEf0c5L66eM87tC9mJSlMg6m0B00Vo3
p72L/4E+0X5848Z/TfDxgVNHoOmiO1m+PAAw==
-----END CERTIFICATE-----
```

An example of the rootCa.key file:

```
-----BEGIN PRIVATE KEY-----
MIIEvAgIBAgIBAQBGAQFASCBKwgqSKAgEAACIBAQZMxOOhFmnnZqzzmK+sKrRjZ8BLykSNLqf
XKUNbJ0MzN7+gYVHo6Ww7NIKyteGuFKBPd/QWlij13IlB7y5/eF8Fw99JX0nax3sBkQECfUj
KlC0Y1ezWwKmQg2QRWd5Y0a/9j9tMT045W2i2aCUjXyxH2pPw5h5iExZ6fI5IhITxZ+XMF5a
fWqIS16QHq7J3/0/J90n1Lnlf43uATjX7Qw2Q9QGb/0QXv0q58St2W9MzLwEH7M6oQ8ZNSn
XFeP7Y5QWoSrX16Q0N+9z2P20zT1jibxbXo7MTX2Q4P+ke+VLlmEqSsLwqjMq+JFPP0U18p8
CyVpAeDsnf/T7/RaeUz+WjeSmP5lerv7Cv9A6te/5kFKiBzZY1iMsTCFOY6rbk1KdcvDyvhi
co1bSh5vShLZ4x3nd64wMxNdLi7YrqnLAL9nms9Z6F1k1w95P1g6dB0U5t62W311IyPzBd0v
WB0pU0CAwEAATA3Enbkx1G9w0BQAQFAAOCAQEApmh6xkoai7yv4XQO24w4DFSN1pAlPlu7t
7vqzazaPz221op8e3/34wRwzdzLn3U4lcpmumo9bJ1/WQxnqoFI3yyJwD7Z2U6VjLx53sD0
y10As+JHlN9Qa9TdJ/s/eGh85MFj7BYB7DjX68sIrwyqWFBN6uu2ud2/9kL1ts1isiFLWKN
vExY*x*np+ja4CZ2nbQ/G0T2hd2D/177gB9l1hjQ5KjV/MoVhVNoDIdb6cuqefx0OOGKqk
Xev29NO2zjQ1TRW0f1N0u45DRHL25PAJ8+s1UnZvI4Adbe3P+5sFeI6AE7nAdnCnA+W0f
Hi35oQxxePO9VqWQ8u8==
-----END PRIVATE KEY-----
```

---

Security Guide for DataStax Distribution of Apache Cassandra 3.11 Latest DDAC PATCH: 5.1.17

Page: 5.1.17

Page 21
SSL encryption

eohMfvgbQKBgFHIaJSUYjJiMoQkeUqTDsx2qq4SmRVCK/llle25MGrgECXMUs3eg
OXbZb7ChPkiJnO5s4mPXi1h3bieZzHAKW83i3p20f6G07bHhhcvRrhngvKuxjA
gkFyGCyVpFWEsGN1DHj49pt/d0a304m16EHDf4/x5vAF/wmITJF2D42A0GBALbB
jpwJUNXKNEU7jxjLGMITVNUdMqEPAgsS6LCrFJzu0PL46YFKW9R1V5faJ15xQxada
x5iLPE4M4l4aysGaiCTTyax2yLAdTH9hT0JmXk75iH4QYCYV48JpaCCaBCTp3pf
ogf EQPjpd3VwgoNDp1tuInrzwzi21sG4t5WRaOGBAKdyp15znZgtg1U0oaazS
CXFQsIT7g4y0LvoL9+EqdPko98l2cB6MD1M9UgFhoyh65xE73ESsGqDyyqGFGE
HS/sM9P4f4tfWRqEgJe61JdKXliYuHF+1kKg4mormriOSSm5z4Z7Q5RxEVcMq
arXACLJvFKL90AxQRWw/dv
-----END PRIVATE KEY-----

Verify the rootCa certificate

2. Verify the rootCa certificate.

$ openssl x509 -in rootCa.crt -text -noout

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-in</td>
<td>Specify the certificate filename to verify</td>
</tr>
<tr>
<td>-text</td>
<td>Print out the certificate in text form including the public key, signature algorithms, issuer and subject names, serial number any extensions present and any trust settings</td>
</tr>
<tr>
<td>-noout</td>
<td>Prevents output of the encoded version of the request</td>
</tr>
</tbody>
</table>

This command prints output to the console that is similar to this example:

Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 10851234054762703412 (0x969753e168c0aa34)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C=US, O=YourCompany, OU=TestCluster, CN=rootCa
    Validity
      Not Before: Sep 10 03:59:38 2016 GMT
      Not After : Sep 10 03:59:38 2017 GMT
    Subject: C=US, O=YourCompany, OU=TestCluster, CN=rootCa
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      Public-Key: (2048 bit)
      Modulus:
        d6:0c:9a:00:cc:cd:af:e8:3f:54:7a:2c:5d:67:96:3c:
        b3:34:82:2b:5e:3e:1:4a:04:f7:7f:59:09:62:8e:8:
        2d:08:05:8b:7d:0f:de:32:37:3f:9a:28:e5:ee:
        17:84:3e:91:ef:95:2c:cb:9b:84:81:2b:0a:3c:08:6e:
        d7:43
      Exponent: 65537 (0x10001)
      Signature Algorithm: sha256WithRSAEncryption
      9a:88:7a:6c:4a:1e:ef:5c:ae:57:12:4e:3b:6e:30:0d:f4:8d:
### Generate public/private key pair and keystore for each node

3. Repeat this command for each node. The files can be generated on a single node and distributed out to the nodes after the entire process is completed.

```
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-genkeypair</code></td>
<td>Command to generate a public/private key pair</td>
</tr>
<tr>
<td><code>-keyalg</code></td>
<td>Specify the key algorithm</td>
</tr>
<tr>
<td><code>-alias</code></td>
<td>Assign an unique alias by which keystore entry is accessed</td>
</tr>
<tr>
<td><code>-keystore</code></td>
<td>Specify the keystore filename</td>
</tr>
<tr>
<td><code>-storepass</code></td>
<td>Specify the keystore password</td>
</tr>
<tr>
<td><code>-keypass</code></td>
<td>Specify the private key password</td>
</tr>
<tr>
<td><code>-validity</code></td>
<td>Specify the number of days for the keystore certificate validity</td>
</tr>
<tr>
<td><code>-keysize</code></td>
<td>Specify the size of the generated key</td>
</tr>
<tr>
<td><code>-dname</code></td>
<td>Specify the X.500 Distinguished Name to be associated with the value of alias</td>
</tr>
</tbody>
</table>

In this example, the node IP address is `10.200.175.15` and the keystore filename incorporates that IP address with a suffix of `.jks` (Java KeyStore). While the keystore can be named with any convention, these examples use the IP address to map the files to the nodes. The `dname` sets the `CN` value to the node's IP address or FQDN as well. The `storepass` and `keypass` must be the same value.

### Check certificates

4. The certificates can be checked once generated:

```
$ keytool -list -keystore 10.200.175.15.jks -storepass myKeyPass
```

An example keystore file:

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry
```
Export certificate signing request (CSR) for each node

5. Once the node certificate and key are generated, a certificate signing request (CSR) is exported. The CSR is signed with the rootCa certificate to verify that the node's certificate is trusted.


<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-certreq</td>
<td>Command to export a CSR</td>
</tr>
<tr>
<td>-file</td>
<td>Specify the CSR filename</td>
</tr>
<tr>
<td>-alias</td>
<td>Assign an unique alias by which keystore entry is accessed</td>
</tr>
<tr>
<td>-keystore</td>
<td>Specify the keystore filename</td>
</tr>
<tr>
<td>-storepass</td>
<td>Specify the keystore password</td>
</tr>
<tr>
<td>-keypass</td>
<td>Specify the private key password</td>
</tr>
<tr>
<td>-dname</td>
<td>Specify the X.500 Distinguished Name to be associated with the value of alias</td>
</tr>
</tbody>
</table>

An example CSR file:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICvDCCAaQCAQAwRzELMAkGA1UEBhMCVVMxDDAKBgNVBAoTA0xMUDESMBAGA1UECxMJTExQMDkw
NzE2MRYwFAYDVQFDEw0xMC4yMDEwNzhmYjU0MTFmZmUtNzlmMTg5MWRkM2U4M2YyOTA3MDYyMIIBIjAN
BgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhUJWFQ+9Xgs8KfhLYCBlb1io9sY44YkCFBaz3j5h6g80w
CqBQVi7YcNwI+3rgAFL6hAibtC3xGqBTNanzxyUu8M7ocfw+dVeT4YQJXEO7NAnlij+x+y12IKCFTuON3
NmGvHVp1DyNzv2YbB7jA3fGucSSX2pDxSTKXNozeIxnnLRv2VKkp8edOX0B13Q7xWgwzQ0/7ZyNaxl11B2YKz
Y9s1BnoEOeRHDFeQ2R7F9UQYBWIKscX3cFsJEPJnpKgbZYLFRpMgMrR7ynIx6MGVDUlfrXp/yFyuHmAM
M8OCvGVi1jwr/ona1DYF5Re5duYBM3VTVwC9n9jwQIDAAoBoDaAwlgYJKoZIhvcNAQKOMSEWhzAd
BgNVHQ4EFgQUQBSrkacD4XCIaJQrX+7/en3Z8r8wDQYJKoZIhvcNAQELBQADggbEBAxU2FbQxy21
EHcnfcC4YETDQvXuwr4QGqscT61fazAEjmmnZ9ekm6cVxVS1NPyjxzi1ZEO1Jtwh94U5ZItKi1YKw
tp5wUGQoYhdyqNqO6wunyVRJAmidoZMAJKj6bXxWqNgq1KL481GD8znquVUM35kWTDmlnwj4Yel
ssScnnBk4d12AzIzFh+ixy3mpyCr/MjMCODLF7VVMRHS5F6PQZO+uGA1gQFz8F8eDgxsOe4ktzqF3xkUB
RDKkfrUC8Z61gLK3LplLZ77a1okP3cNkcVStVgbhL9gwnhCORNHY+NYzLmLa+hS4CAJzRK1C
nsdwUTp+HXUtyNLd7GJHLPu0YY=
-----END NEW CERTIFICATE REQUEST-----
```

Sign node certificate with rootCa for each node

6. The CSR is input, signed with the rootCa certificate and a signed node certificate is created.

$ openssl x509 -req -CA rootCa.crt -CAkey rootCa.key -in 10.200.175.15.csr -out 10.200.175.15.crt_signed -days 365 -CAcreateserial -passin pass:myPass

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-req</td>
<td>Specify that the input file is a CSR</td>
</tr>
<tr>
<td>-CA</td>
<td>Identify the rootCa certificate</td>
</tr>
<tr>
<td>-CAkey</td>
<td>Identify the rootCa key</td>
</tr>
</tbody>
</table>

Page 24

Security Guide for DataStax Distribution of Apache Cassandra 3.11 Latest DDAC

patch: 5.1.17
SSL encryption

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-in</td>
<td>Specify the input filename from which to read a certificate</td>
</tr>
<tr>
<td>-out</td>
<td>Specify the output filename for the signed certificate</td>
</tr>
<tr>
<td>-CAcreateserial</td>
<td>Specify that a CA serial number file is created if it does not exist</td>
</tr>
<tr>
<td>-days</td>
<td>Specify the number of days for which to certify the certificate</td>
</tr>
<tr>
<td>-passin</td>
<td>Specify the key password source</td>
</tr>
</tbody>
</table>

An example of the signed certificate:

```
-----BEGIN CERTIFICATE-----
MIIDBTCCAe0CCQDBKbNGSE8C9DANBgkqhkiG9w0BAQsFADBCMQswCQYDVQQGEwJV
UzEMMAoGa1UECWoMDQsGAR8wDQYJKoZIhvcNAQEBBQADggEPADCCAQEEAwIbADCC
AQEEBQADggEPADCCAQEEAwIbADCCAQEEBQADggEPADCCAQEEAwIbADCCAQEEBQAD
ggEPADCCAQEEAwIbADCCAQEEBQADggEPADCCAQEEAwIbADCCAQEEBQADggEPAD
CCAQEEAwIbADCCAQEEBQADggEPADCCAQEEAwIbADCCAQEEBQADggEPADCCAQEE
AwIbADCCAQEEBQADggEPADCCAQEEAwIbADCCAQEEBQADggEPADCCAQEEAwIbADCC
AQEEBQADggEPADCCAQEEAwIbADCCAQEEBQADggEPADCCAQEEAwIbADCCAQEEBQAD
-----END CERTIFICATE-----
```

Verify the signed certificate for each node

7. Check the signed certificate by designating the rootCa certificate and the signed certificate to verify:

```
$ openssl verify -CAfile rootCa.crt 10.200.175.15.crt_signed
```

If the verification succeeds, a console message is returned:

```
10.200.175.15.crt_signed: OK
```

Import rootCa certificate to each node keystore

8. Use keytool -importcert to import the rootCa certificate into each node keystore:

```
keytool -importcert
    -keystore 10.200.175.15.jks
    -alias rootCa
    -file rootCa.crt
    -noprompt
    -keypass myKeyPass
    -storepass myKeyPass
```

The -noprompt option allows the command to use the specified values rather than prompting for input.

The keystore file now has two entries, one for the rootCa certificate and one for the node certificate:

```
Keystore type: JKS
```

SSL encryption

Keystore provider: SUN

Your keystore contains 2 entries

rootca, Sep 10, 2016, trustedCertEntry, Certificate fingerprint (SHA1):
10.200.175.15, Sep 10, 2016, PrivateKeyEntry, Certificate fingerprint (SHA1):

Import node’s signed certificate into node keystore for each node

9. Import node’s signed certificate into node keystore:

$ keytool -importcert -keystore 10.200.175.15.jks -alias 10.200.175.15 -file 10.200.175.15.crt_signed -noprompt -keypass myKeyPass -storepass myKeyPass

The resulting file appears similar to the result from the previous step, but the node certificate originally created is replaced with the signed node certificate.

Create a server truststore

10. Use a server truststore to establish a chain of trust between the nodes of the cluster.


The resulting truststore file can be inspected using the keytool -list command:

$ keytool -list -keystore generic-server-truststore.jks -storepass truststorePass

Example of the truststore file includes a rootCa certificate entry:

Keystore type: JKS
Keystore provider: SUN
Your keystore contains 1 entry

rootca, Sep 10, 2016, trustedCertEntry, Certificate fingerprint (SHA1):

Copy the truststore file to each node

11. The truststore file must be copied to each node. If you use a node to generate the file, copy the file to a location of choice and name the file with a standard format, such as server-truststore.jks. This example shows the copy command for a Linux server:

$ cp ~/generic-server-truststore.jks /usr/local/lib/cassandra/conf/server-truststore.jks

Copy the each node keystore file to each node

Page  Security Guide for DataStax Distribution of Apache Cassandra 3.11 Latest DDAC 26
SSL encryption

Each node file must have a copy of its keystore file. If you use a node to generate the file, copy the file to a location of choice. This example shows the secure remote copy commands, where the certificates were generated on a single node. The file is stored in the configuration directory:

```bash
$ scp -r 10.200.175.150.jks /usr/local/lib/cassandra/conf/10.200.175.150.jks
```

Node-to-node encryption

Node-to-node encryption protects data transferred between nodes in a cluster, including gossip communications, using SSL (Secure Sockets Layer).

**Prerequisites:** Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development.

To enable node-to-node SSL, you must set the `server_encryption_options` in the `cassandra.yaml` file.

**Enable server_encryption_options on each node**

1. Modify the `cassandra.yaml` file with the following settings:

### Production clusters

```yaml
server_encryption_options:
  internode_encryption: all
  keystore: /usr/local/lib/cassandra/conf/server-keystore.jks
  keystore_password: myKeyPass
  truststore: /usr/local/lib/cassandra/conf/server-truststore.jks
  truststore_password: truststorePass
  # More advanced defaults below:
  protocol: TLS
  algorithm: SunX509
  store_type: JKS
  cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
  require_client_auth: true
```

This file uses the certificates prepared with a self-signed CA.

- `cipher_suites` can be configured for FIPS-140 compliance if required.

### Development clusters

```yaml
server_encryption_options:
  internode_encryption: all
  keystore: /conf/keystore.node0
  keystore_password: cassandra
  truststore: /conf/truststore.node0
  truststore_password: cassandra
  # More advanced defaults below:
  protocol: TLS
  algorithm: SunX509
  store_type: JKS
  cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]
```
SSL encryption

**require_client_auth: true**

This file uses the **certificates prepared for development**.

Internode encryption can be set to four different choices:
- **all**
  - All traffic is encrypted.
- **none**
  - No traffic is encrypted.
- **dc**
  - Traffic between datacenters is encrypted.
- **rack**
  - Traffic between racks is encrypted.

Set appropriate paths to the *keystore* and *truststore* files. Set the passwords to the passwords set during *keystore* and *truststore* generation. If two-way certificate authentication is desired, set *require_client_auth* to *true*.

**Restart the database.**

2. Restart the database to make changes effective:

   ```
   $ kill -9 cassandra_pid
   $ cassandra
   ```

3. Check the logs to discover if SSL encryption has been started. Use the *grep* command:

   ```
   $ grep SSL install_location/logs/system.log
   ```

   The resulting line is similar to this example:

   ```
   INFO [main] 2016-09-12 18:34:14,478 MessagingService.java:511 - Starting Encrypted Messaging Service on SSL port 7001
   ```

**Client-to-node encryption**

Client-to-node encryption protects data in flight from client machines to a database cluster using SSL (Secure Sockets Layer). It establishes a secure channel between the client and the coordinator node.

**Prerequisites:** Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development.

To enable client-to-node SSL, set the `client_encryption_options` in the *cassandra.yaml* file.

On each node under `client_encryption_options`:

1. Enable encryption.
2. Modify the cassandra.yaml file with the following settings:

**Production clusters**

```
client_encryption_options:
```
SSL encryption

enabled: true
# If enabled and optional is set to true encrypted and unencrypted connections are handled.
optional: false
keystore: /usr/local/lib/cassandra/conf/server-keystore.jks
keystore_password: myKeyPass

require_client_auth: true
# Set trustore and truststore_password if require_client_auth is true
truststore: /usr/local/lib/cassandra/conf/server-truststore.jks
truststore_password: truststorePass
protocol: TLS
algorithm: SunX509
store_type: JKS
cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]

This file uses the certificates prepared with a self-signed CA.

Development clusters

client_encryption_options:
  enabled: true
  # If enabled and optional is set to true encrypted and unencrypted connections are handled.
  optional: false
  keystore: conf/keystore.node0
  keystore_password: cassandra

  require_client_auth: true
  # Set trustore and truststore_password if require_client_auth is true
  truststore: conf/truststore.node0
  truststore_password: cassandra
  protocol: TLS
  algorithm: SunX509
  store_type: JKS
  cipher_suites: [TLS_RSA_WITH_AES_256_CBC_SHA]

This file uses the certificates prepared for development.

Set appropriate paths to the keystore and truststore files. Set the passwords to the passwords set during keystore and truststore generation. If two-way certificate authentication is required, set require_client_auth to true. Enabling two-way certificate authentication allows tools to connect to a remote node. For local access to run cqlsh on a local node with SSL encryption, set require_client_auth to false.

Enabling client encryption encrypts all traffic on the native_transport_port (default: 9042). If both encrypted and unencrypted traffic is required, in the cassandra.yaml set native_transport_port_ssl for encrypted traffic (default: 9142) and native_transport_port for traffic that is not encrypted to different ports.

Restart the database.

3. Restart the database to make changes effective.

$ kill -9 cassandra_pid

$ cassandra
Using cqlsh with SSL

Using a cqlshrc file is the easiest method of getting cqlshrc settings. The cqlshrc.sample provides an example that can be copied as a starting point.

Prerequisites: Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development. Additionally, configure client-to-node encryption.

1. To run cqlsh with SSL encryption, create a ~/.cassandra/cqlshrc file in with the following settings:

```plaintext
[authentication]
username = fred
password = !!bang!!$

[connection]
hostname = 127.0.0.1
port = 9042

[ssl]
certfile = ~/keys/cassandra.cert
validate = false ;; Optional, true by default. See the paragraph below.

[certfiles] ;; Optional section, overrides the default certfile in the [ssl] section.
10.209.182.160 = /etc/dse/cassandra/conf/dsenode0.cer
10.68.65.199 = /etc/dse/cassandra/conf/dsenode1.cer
```

When validate is enabled, to verify that the certificate is trusted, the host in the certificate is compared to the host of the machine to which it is connected. Note that the rootCa certificate and key are supplied to access the trustchain. The SSL certificate must be provided either in the configuration file or as an environment variable. The environment variables (SSL_CERTFILE and SSL_VALIDATE) override any options set in this file.

Additional settings in the cqlshrc file are described in Creating and using the cqlshrc file.

An optional section, [certfiles], overrides the default certfile in the [ssl] section. The use of the same IP addresses in the [certfiles] section, as is used to generate the dname of the certificates, is required for two-way SSL encryption. Each node must have a line in the [certfiles] section for client-to-remote-node or node-to-node. Using certfiles is more common for development clusters.

2. Start cqlsh with the --ssl option for cqlsh to local node encrypted connection:

```bash
$ install_location/bin/cqlsh --ssl
```

3. A username and password can also be supplied at cqlsh startup. This example provides the username cassandra with password cassandra:

```bash
$ install_location/bin/cqlsh --ssl -u cassandra -p cassandra
```

CQLSH can read the username and password from the cqlshrc file at start up.
4. For a remote node encrypted connection, start cqlsh with the --ssl option and an IP address:

$ install_location/bin/cqlsh --ssl 172.31.10.22

**Using nodetool (JMX) with SSL encryption**

Using nodetool with SSL requires some JMX setup. Changes to cassandra-env.sh are required, and a
configuration file, ~/.cassandra/nodetool-ssl.properties, is created.

**Prerequisites:** Configure the following options before setting up nodetool to run with SSL and authentication:

- Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for development.
  Additionally, configure client-to-node encryption

- Enabling JMX authentication and authorization if authentication and authorization are required.

1. To run nodetool with SSL encryption, additional changes are required to cassandra-env.sh. Add the
following settings to the file. Use the file path to the keystore and truststore, and appropriate passwords for
each file. Make these changes on all nodes in the cluster.

### Production

```bash
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.need.client.auth=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.registry.ssl=true"
#JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.enabled.protocols=<enabled-protocols>"
#JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.enabled.cipher.suites=<enabled-cipher-suites>"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStore=/usr/local/lib/cassandra/conf/server-keystore.jks"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStorePassword=myKeyPass"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStore=/usr/local/lib/cassandra/conf/server-truststore.jks"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStorePassword=truststorePass"
```

### Development

```bash
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.need.client.auth=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.registry.ssl=true"
#JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.enabled.protocols=<enabled-protocols>"
#JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl.enabled.cipher.suites=<enabled-cipher-suites>"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStore=keystore.node0"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.keyStorePassword=cassandra"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStore=truststore.node0"
JVM_OPTS="$JVM_OPTS -Djavax.net.ssl.trustStorePassword=truststorePass"
```

Where the settings implement the following SSL options:

- (Required) Enable SSL for JMX by setting com.sun.management.jmxremote.ssl to true
SSL encryption

- Use a two-way certificate authentication by setting
  `com.sun.management.jmxremote.ssl.need.client.auth` to `true`

- Create an RMI registry protected by SSL that is created and configured by
  the out-of-the-box management agent when the Java VM is started by setting
  `com.sun.management.jmxremote.registry.ssl` to `true`

- Enable client authentication over SSL to have full security by setting
  `com.sun.management.jmxremote.ssl.need.client.auth` to `true`

- (Required) Set appropriate paths to the `keystore` and `truststore` files.

- (Required) Set the passwords to the passwords set during keystore and truststore generation.

2. Restart the database.

3. To run `nodetool` with SSL encryption, create a `.cassandra/nodetool-ssl.properties` file in your home or
   client program directory with the following settings on the same system that runs `nodetool`.

   **Production**

   ```
   -Dcom.sun.management.jmxremote.ssl=true
   -Dcom.sun.management.jmxremote.ssl.need.client.auth=true
   -Dcom.sun.management.jmxremote.registry.ssl=true
   -Djavax.net.ssl.keyStore=/usr/local/lib/cassandra/conf/server-keystore.jks
   -Djavax.net.ssl.keyStorePassword=myKeyPass
   -Djavax.net.ssl.trustStore=/usr/local/lib/cassandra/conf/server-truststore.jks
   -Djavax.net.ssl.trustStorePassword=truststorePass
   ```

   **Development**

   ```
   -Djavax.net.ssl.keyStore=keystore.node0
   -Djavax.net.ssl.keyStorePassword=cassandra
   -Djavax.net.ssl.trustStore=truststore.node0
   -Djavax.net.ssl.trustStorePassword=cassandra
   -Dcom.sun.management.jmxremote.ssl.need.client.auth=true
   -Dcom.sun.management.jmxremote.registry.ssl=true
   ```

4. Start `nodetool` with the `--ssl` option for encrypted connection for any `nodetool` operation.

   ```
   $ install_location/bin/nodetool -ssl info
   ```

5. Start `nodetool` with the `--ssl` option for encrypted connection and a username and password for
   authentication and authorization for any `nodetool` operation.

   ```
   $ install_location/bin/nodetool -ssl -u cassandra -pw cassandra status
   ```

**Using jconsole (JMX) with SSL encryption**

Using jconsole with SSL requires the same JMX changes to `cassandra-env.sh` as with `nodetool`. See using
`nodetool (JMX) with SSL encryption`. You do not need to create `nodetool-ssl.properties`, but you must use
the same JVM keystore and truststore options on the Jconsole command line.

**Prerequisites:** Prepare SSL certificates with a self-signed CA for production, or prepare SSL certificates for
development. Additionally, configure client-to-node encryption.
1. Copy the keystore and truststore files created in the prerequisite to the system where jconsole is launched. In this example, the files are `server-keystore.jks` and `server-truststore.jks`.

2. Run `jconsole` using the JVM options:

```
jconsole -Djavax.net.ssl.keyStore=server-keystore.jks
-Djavax.net.ssl.keyStorePassword=myKeyPass
-Djavax.net.ssl.trustStore=server-truststore.jks
-Djavax.net.ssl.trustStorePassword=truststorePass
```

If no errors occur, `jconsole` launches a session with the node. If connecting to a remote node, enter the hostname and JMX port in `Remote Process`. If using authentication, enter the username and password.
Chapter 6. Configuring firewall port access

The following ports must be open to allow bi-directional communication between nodes, including certain DataStax Distribution of Apache Cassandra™ (DDAC) ports. Configure the firewall running on nodes in your cluster accordingly. Without open ports as shown, nodes will act as a standalone database server and will not join the Cassandra cluster.

Table 1: Public port

<table>
<thead>
<tr>
<th>Port number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>SSH port</td>
</tr>
</tbody>
</table>

Table 2: Cassandra inter-node ports

<table>
<thead>
<tr>
<th>Port number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>Cassandra inter-node cluster communication.</td>
</tr>
<tr>
<td>7001</td>
<td>Cassandra SSL inter-node cluster communication.</td>
</tr>
<tr>
<td>7199</td>
<td>Cassandra JMX monitoring port.</td>
</tr>
</tbody>
</table>

Table 3: Cassandra client ports

<table>
<thead>
<tr>
<th>Port number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9042</td>
<td>Cassandra client port.</td>
</tr>
<tr>
<td>9160</td>
<td>Cassandra client port (Thrift).</td>
</tr>
<tr>
<td>9142</td>
<td>Default for <code>native_transport_port_ssl</code>, useful when both encrypted and unencrypted connections are required</td>
</tr>
</tbody>
</table>