CASSANDRA SECURITY

Abstract

Cloud security is a must ask for any enterprise. When data stored in cloud, it is even critical to ensure end to end security. This is an attempt to document security features for CASSANDRA in cloud.

Table of Contents

Cassandra Security

To secure Cassandra, **Enterprise** internal guidelines must be met. At minimum following needs to be ensured for securing Cassandra in cloud environment.

1. Secured Data in motion (SSL connection)
   a. Internode communication
   b. Client-server communication
   c. Spark Cassandra connector for SSL

2. Network Security Group (NSG)

3. Secured Data at rest

4. Authentication and Authorization

5. CQLSH SSL connection

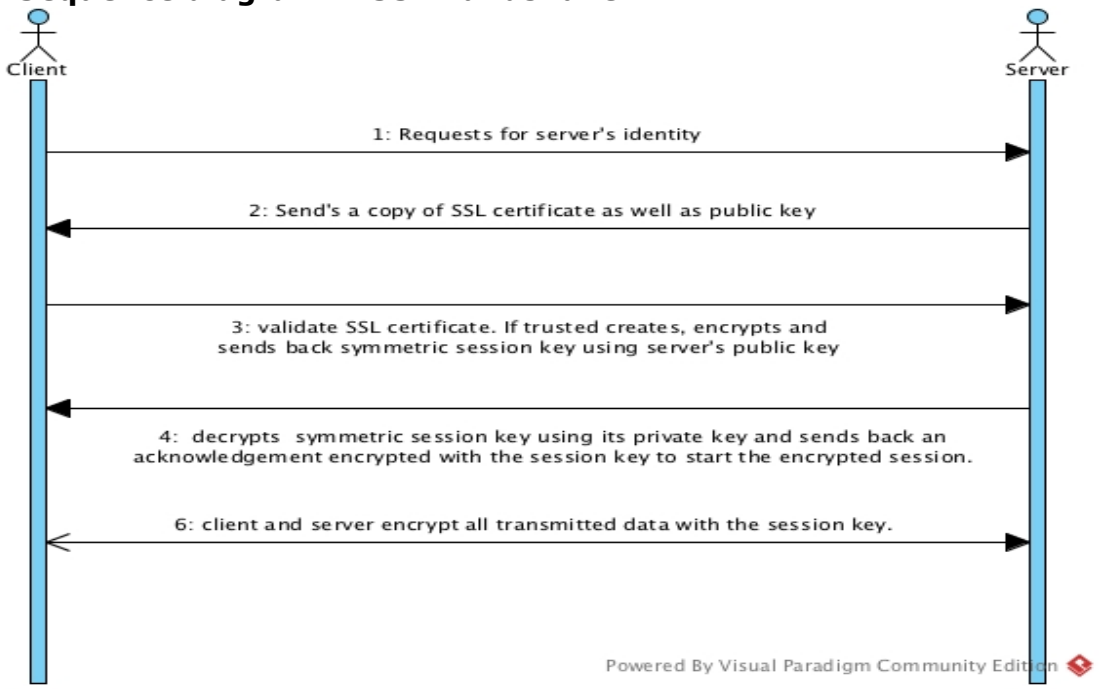
1.1 SSL

SSL (Secure Sockets Layer) is a standard security technology for establishing an encrypted link between a server and a client—typically a between two servers in a cluster or a web browser (client) to a server. SSL is a protocol that determines variables of the encryption for both the link and the data being transmitted.

1.1.1 SSL Handshake

Three keys are used to set up SSL connection. Public, private and session keys. Here are sequences of SSL handshake.

Fig 1: Sequence diagram -- SSL handshake



1.2 RSA algorithm in SSL

RSA algorithm involves four steps.

a. Key generation
b. Key distribution
c. Encryption
d. Decryption

1.3 Certificate Management Utility in Java

Java **Keytool** is a key and certificate management utility. It allows users to manage their own public/private key pairs and certificates. Java **Keytool** stores the keys and certificates in what is called a keystore. By default, the Java keystore is implemented as a file.

Table 1.1: KeyStore and TrustStore in keytool

Subject	**Keystore**	**Truststore**
Context	Keystore and truststore are used in context to setting up SSL connection among clients and server.	
Construct	TrustStore and keyStore are very much similar in terms of construct and structure as both are managed by keytool command	
Certificates	Keystore is used to store public certificates for SSL connection	TrustStore is used to store private certificates for SSL connection
Handshaking	Keystore is used to provide credentials for handshaking	TrustStore is used to verify credentials during handshake
Contains	keyStore in Java stores private key and certificates corresponding to their public keys and require if SSL Server or SSL requires client authentication	TrustStore stores public key or certificates from third party, Java application communicate or certificates signed by CA (certificate authorities like Verisign, Thawte, Geotrust or GoDaddy) which can be used to identify third party
Manager	Is managed by KeyManager in java	Is managed by TrustManager in java and determines whether remote connection is trusted or not.
Access path in api	Djavax.net.ssl.keyStore to specify path for keyStore	Djavax.net.ssl.trustStore to specify path for trustStore
Password in api	Djavax.net.ssl.keyStorePassword to specify path for keyStorePass	Djavax.net.ssl.trustStorePassword to specify path for trustStorePass
File Management	For manageability and maintainability, it is good to manage separate files for keystore and truststore. But it is possible to combine into one file .	

2.0 Secured Data in motion (SSL connection)

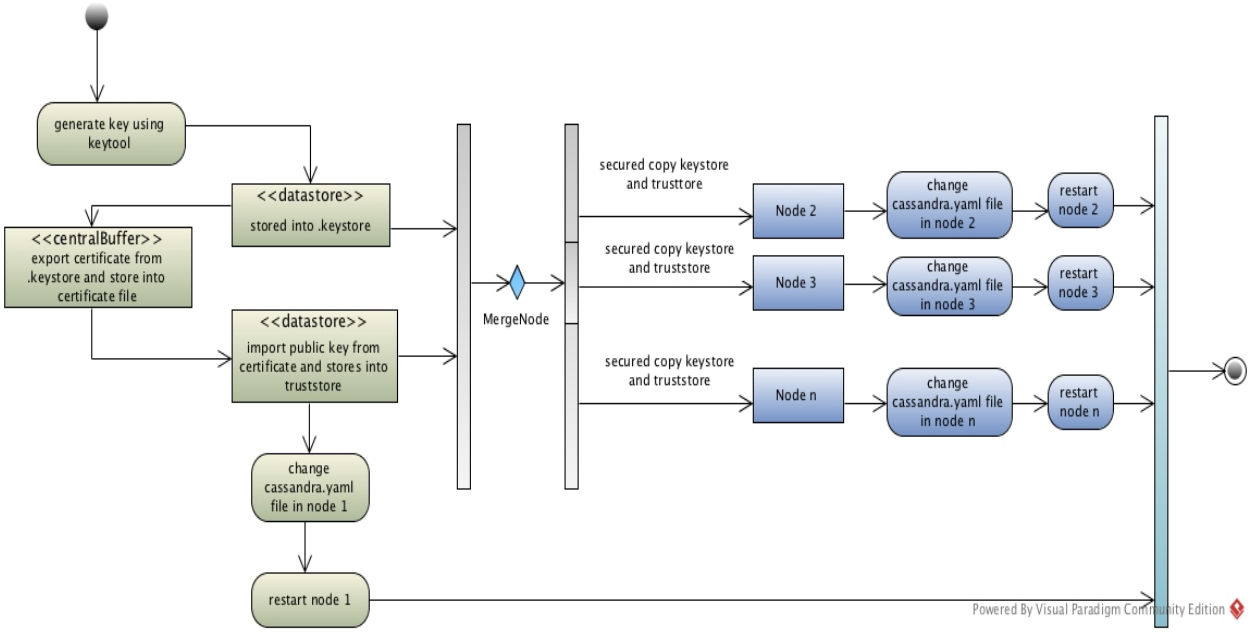
2.0.1 Internode Communication:

Cassandra cluster contains nodes and in its distributed architecture need to Gossip and replicate data among nodes. SSL integration among Cassandra nodes is a way for securing Internode communication.

In order to achieve SSL connectivity across all nodes in Cassandra cluster following steps need to be performed.

1. For symmetric key encryption create certificate and public, private key pair in one of the nodes using java keytool certificate management utility.
2. Secured copy (scp) of public, private key pair in all the nodes for symmetric key encryption.
3. Change cassandra.yaml file with properties related to server_encryption_options in all the nodes
4. Restart all the nodes as root.

Fig 2: Activity Diagram - internode communication using SSL



Here are step by step processes and details for securing internode communications using SSL connection.

Steps 1: Create ssl directory.

As root of each node create .ssl directory.

```
sudo -i
mkdir /etc/dse/cassandra/.ssl
```

Steps 2: Create certificate and public, private key pairs in one of the nodes.

```
cd /etc/dse/cassandra/.ssl
```

a. Generate key and stores into .keystore

```
keytool -genkey -alias dc0vm0 -keyalg RSA -dname "CN=Braja Das, OU=ABCCorp, O=BI, L=Seattle, C=US" -keystore .keystore -storepass Pass123 -keypass Pass123
```

b. Export certificate from keystore and store into certificate file.

```
keytool -export -alias dc0vm0 -file dc0vm0.cer -keystore .keystore -storepass Pass123 -keypass Pass123
```

c. Import public key from certificates and stores into .truststore.

```
keytool -import -v -trustcacerts -alias dc0vm0 -file dc0vm0.cer -keystore .truststore -storepass Pass123 -keypass Pass123 --noprompt
```

set appropriate permission for ssh user.

```
sudo -i
chown datastax:datastax *.cer
chown datastax:datastax .keystore
chown datastax:datastax .truststore
chown datastax:datastax /etc/dse/cassandra/.ssl
chmod 700 /etc/dse/cassandra/.ssl
```

Steps 3: Distribute public and private key among all the nodes.
From dc0vm0 node to dc0vm1 and other nodes (remote) use following to distribute. keystore and. truststore. Here are steps in dc0vm0.

```
scp /etc/dse/cassandra/.ssl/.truststore     /etc/dse/cassandra/.ssl/
scp /etc/dse/cassandra/.ssl/.keystore      /etc/dse/cassandra/.ssl/
```

Steps 4: Set permissions
As root, use following to set permissions among all nodes.

```
sudo -i
chown cassandra:cassandra *.cer
chown cassandra:cassandra .keystore
chown cassandra:cassandra .truststore
chown cassandra:cassandra /etc/dse/cassandra/.ssl
chmod 700 /etc/dse/cassandra/.ssl
```

Steps 5: Change cassandra.yaml file in server_encryption_options
change followings in cassandra.yaml file

```
server_encryption_options:
    internode_encryption: all
    keystore: /etc/dse/cassandra/.ssl/.keystore
    keystore_password: Pass123
    truststore: /etc/dse/cassandra/.ssl/.truststore
    truststore_password: Pass123
    # More advanced defaults below:
    protocol: TLS
    algorithm: SunX509
    store_type: JKS
    cipher_suites:
[TLS_RSA_WITH_AES_128_CBC_SHA,
TLS_DHE_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA]
    require_client_auth: true
```

Steps 6: Restart all the nodes as root
As root on each node use following commands to restart nodes.

```
nodetool -h localhost drain
sudo service dse stop
sudo service dse start
```

2.1.1 Secured Client Server Communication

In order to achieve SSL connectivity between client applications and cassandra cluster, following steps need to be performed.

1. For symmetric key encryption create certificate and public, private key pair in one of the nodes using java keytool certificate management utility.
2. Secured copy (scp) of public, private key pair in all the nodes for symmetric key encryption.
3. Change cassandra.yaml file with properties related to client_encryption_options in all the nodes
4. Restart all the nodes as root.

Step1 and steps 2 are similar to secured internode communication. Here different names can be maintained as keystore_client and truststore_client.

Step: Change in client_encryption_options in cassandra.yaml

Change followings in client_encryption_options.

client_encryption_options:
enabled: true
If enabled and optional is set to true encrypted and unencrypted connections are handled.
#optional: false
keystore: /etc/dse/cassandra/.ssl/.keystore
keystore_password: Pass123
require_client_auth: false
Set trustore and truststore_password if require_client_auth is true
truststore: /etc/dse/cassandra/.ssl/.truststore
truststore_password: Pass123
More advanced defaults below:
protocol: TLS
algorithm: SunX509
store_type: JKS
cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA]

2.1.1.1 OPEN SSL and PEM File

OpenSSL is the de-facto tool for SSL. It providers both the library for creating SSL sockets, and a set of powerful tools for administrating an SSL enabled website.

2.1.1.2 PEM File:

PEM files are standard format for openSSL and many other SSL tools. This format is designed to be safe for inclusion in ascii or even rich-text documents. This means that you can simple copy and paste the content of a pem file to another document and back.

Following is a sample PEM file containing a private key and a certificate. A few rules apply when copying a certificate around:

- A single key or certiciate must start with the appropriate header, such as "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----". Always copy the certificate with the header and footer notes.
- The number of dashs ("-----") is meaningful, and must be correct.

A single PEM file can contain a number of certificates and a key, for example, a single file with:

- Public certificate
- Intermidiate Certificate
- Root certificate
- Private key

PEM File Content:

```
cassandra@dc0vm4:/etc/dse/cassandra/.ssl$ more CQLSHcassandra1.pem
Bag Attributes
    friendlyName: client_key
    localKeyID: 54 69 6D 65 20 31 34 38 38 35 30 36 35 39 34 38 38 35
Key Attributes: <No Attributes>
-----BEGIN PRIVATE KEY-----
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAMFLyv8/EF/rjTA5
SXyBudUEiAg/y6ST4zfgnfRT80pzmiYG6z3ixT+ow63yS6Ro18CtvzafAl6dE4mZ
aXtZYiwo2r++yIPvnVLCLnPPaRzahnaCC6W2m+HkJuxWQ/UxDC6BEhhruv8JZ1i6
a7kNrrT6+YZX667IPGQbpWFV9befAgMBAAECgYEAlsAtyXO9qZFjw8Bp95iU/fVS
wlw+zlQoWWPszKjMjbwq9I1g2hsKCuPr+LWHGOpLmhHnlwncJz4KBr6G7ZSAYuLg
ZdcYzszjU/VcGrpK6d29QgoAkUjNcIwW0FIYOToLT4hePAIdnjRuh+CKgOoAQKME
39onesSci0wUNXYMToECQQD6xduplemHGalXN2BA/pH7j4NVkBPakhvtfpb2UaoC
Hh/GRDdja+QhVLdTnr1jcTIP8rQKJmAafkoZC1j/52+3AkEAxVM8CVgT/R570OZo
RRoaeXl+0UXca7tFiP4Eyq/sSfQj3h224NVBCrVooO3AinW2/cBTc1vSSNADSbn1
sV0nWQJBAI8NGS5XRxz7RX9sJmtNDOeMyWWgx8KSQH4tDV673RhSKNwIA/SiEkP1
OJLp5a15YA5667sygvX5/rjkoUNxuWcCQCTKRwhK9rcbxuQFAW3Y1xTM9TsZdmZT
rTxEeCo+MKT9Mu7HxYAX4p+XgSF4Eoz+M5cOE8EPXp6awfIuDMP75UkCQQCDguCb
2vfeZiC0Ag+KHHZq0C97pFIySKq9lKM+Rbl5nX/nELfLwUQKqoGNv18phkPfh9bA
FrGsUTeSmVf74g44
-----END PRIVATE KEY-----
Bag Attributes
    friendlyName: client_key
    localKeyID: 54 69 6D 65 20 31 34 38 38 35 30 36 35 39 34 38 38 35
subject=/C=US/L=Seattle/O=BI/OU=ABCCorp/CN=Braja Das
issuer=/C=US/L=Seattle/O=BI/OU=ABCCorp/CN=Braja Das
-----BEGIN CERTIFICATE-----
MIICQjCCAaugAwIBAgIEV6xODTANBgkqhkiG9w0BAQsFADBUMQswCQYDVQQGEwJV
UzEQMA4GA1UEBxMHU2VhdHRsZTELMAkGA1UEChMCQkkxEjAQBgNVBAsTCVN0YXJi
dWNrczESMBAGA1UEAxMJQnJhamEgRGFzMB4XDTE3MDMwMzAwMDgyM1oXDTE3MDYw
MTAwMDgyM1owVDELMAkGA1UEBhMCVVMxEDAOBgNVBAcTB1NlYXR0bGUxCzAJBgNV
BAoTAkJJMRIwEAYDVQQLEwlTdGFyYnVja3MxEjAQBgNVBAMTCUJyYWphIERhczCB
nzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAwUvK/z8QX+uNMDlJfIG51QSICD/L
pJPjN+Cd9FPzSnOaJgbrPeLFP6jDrfJLpGjXwK2/Np8CXp0TiZlpe1liLCjav77I
g++dUsIuc89pHNqGdoILpbab4eQm7FZD9TEMLoESGGu6/wlnWLpruQ2utPr5hlfr
rsg8ZBulYVX1t58CAwEAAaMhMB8wHQYDVR0OBBYEFPhNfAiYw8iRS6WDH7xf/yvD
m7MKMA0GCSqGSIb3DQEBCwUAA4GBAITtUFPEYayhsSyOuui7eZlYaV+85Lz4KaQZ
rfv0TPPVJV9+ZHa8NEb3FKdUknit6sj9mOTNV+hiOdvDIIAtvAxSYQabTx24Uijo
USxMJkJLcYbnJA7ZJOkxUQWIjlci5JoVnvlAZiWGms3mSMIaIrR5LIsYh8Gr65Pn
DDjYKHKI
-----END CERTIFICATE-----
```

3.0 CQLSH SSL Connection

Following steps need to be performed for CQLSH SSL connection.

Step 1: Exporting Private key from keytool keytool's proprietary format (JKS format) to PKCS12 format.

keytool -importkeystore –srckeystore .keystore_client -destkeystore local_user.p12 -deststoretype PKCS12

Step 2: Export unencrypted private key and certificate using open SSL:

openssl pkcs12 -in local_user.p12 -out CQLSHcassandra1.pem -nodes

Step 3: Set up permissions for CQLSHcassandra1.pem

chown cassandra:cassandra local_user.p12
chmod 400 local_user.p12

chown cassandra:cassandra CQLSHcassandra1.pem
chmod 444 CQLSHcassandra1.pem
chmod 755 /etc/dse/cassandra/.ssl

Step 4: Change cqlshrc file to use PEM file on host nodes.

As root or datastax admin (cassandra) create cqlshrc file in /.cassandra. Append following contents in cqlshrc file.

[ssl]
validate = false
certfile = /etc/dse/cassandra/.ssl/CQLSHcassandra1.pem

Step 5: Connect cqlsh using following.
cqlsh –ssl

4.0 Spark Cassandra Connector for SSL

https://github.com/datastax/spark-cassandra-connector/blob/master/doc/reference.md

Cassandra SSL Connection Options
Here is an option in datastax spark Cassandra connector for SSL integration.

Property Name	Default	Description
connection.ssl. clientAuth.enabled	false	Enable 2-way secure connection to Cassandra cluster
connection.ssl.enabled	false	Enable secure connection to Cassandra cluster
connection.ssl.enabledAlgorithms	Set (TLS_RSA_WITH _AES_128_CBC_SHA, TLS_RSA_WITH _AES_256_CBC_SHA)	SSL cipher suites
connection.ssl.keyStore.password	None	Key store password
connection.ssl.keyStore.path	None	Path for the key store being used
connection.ssl.keyStore.type	JKS	Key store type
connection.ssl.protocol	TLS	SSL protocol
connection.ssl.trustStore.password	None	Trust store password

connection.ssl.trustStore.path	None	Path for the trust store being used
connection.ssl.trustStore.type	JKS	Trust store type

All parameters should be prefixed with spark.cassandra.

4.0.1 Spark Cassandra Connector code

```
def sparkConfCassandraSSL(appName: String, host: String, userName: String, password: String, trustStorePwd: String, trustStorePath: String) : SparkConf = {

  val basicConf = sparkConfCassandra(appName, host, userName, password)
  val conf: SparkConf = basicConf
    .set("spark.cassandra.connection.ssl.enabled", "true")
    .set("spark.cassandra.connection.ssl.trustStore.password", trustStorePwd)

.set("spark.cassandra.connection.ssl.trustStore.path",trustStorePath)
    .set("spark.cassandra.connection.ssl.trustStore.type", "JKS")
  conf
}
```

In spark Cassandra connector, following properties are mandatory and important for SSL connection from client application to server(node).

a. Host name
b. User name
c. Password
d. trustStore password
e. trustStorePath
f. trustStoreType
g. sslEnabled = true

trustStorePath in Spark Cassandra connector points to trustStoreClient file. In container orchestration framework truststorePath can point to a location inside a container. Here are options of setting up trustStore files inside a container.

1. Fixed trustStorePath in a container and trustStoreClient file can be pushed as part of CD (continuous deployment).
2. Reading trustStore file from secured source (BLOB storage) and download into container directory during runtime and connect to Cassandra server
3. Key vault integration: Generate key from key vault and download into both Cassandra server and containers and connecting application to Cassandra server.

4.1 Cluster Builder Cassandra Driver Connector with SSL

https://docs.datastax.com/en/drivers/java/2.0/com/datastax/driver/core/Cluster.Builder.html

Here are API steps to be followed in order to establish secured SSL connection from client using cluster builder.

1. Get trustmanager from keystore
2. Create SSLContext from trustManager
3. Build SSLOptions.from sslcontext
4. Build secured cluster using SSLOptions, Cassandra credentials.

Here are snippets of code.

getTrustManagerFromKeyStore

```
def getTrustManagerFromKeyStore (truststorePath: String,
truststorePassword: String): Array[TrustManager] = {

  val ks = KeyStore.getInstance("JKS")
  val trustStore = new FileInputStream(truststorePath)
  ks.load(trustStore, truststorePassword.toCharArray())
  val tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm(
))
  tmf.init(ks)

  val tm: Array[TrustManager] = tmf.getTrustManagers()

  tm
}
```

getSSlContext

```
def getSSlContext(tm: Array[TrustManager]): SSLContext = {
  val sslcontext: SSLContext = SSLContext.getInstance("TLS");
  sslcontext.init(null, tm, null)

  sslcontext
```

```
}def buildSSlOptions(sslcontext: SSLContext): JdkSSLOptions = {
  val sslOptions = JdkSSLOptions.builder()
    .withSSLContext(sslcontext)
    .build();

  sslOptions
}
```

buildSecuredCluster

```
def buildSecuredCluster (CASSANDRA_DEFAULT_HOST: String,
CASSANDRA_DEFAULT_PORT: String,
                    userid: String, password: String, sslOptions:
JdkSSLOptions): Cluster = {

  val cluster: Cluster = Cluster.builder().addContactPointsWithPorts(
    new InetSocketAddress(
      sys.props.getOrElse("chost", CASSANDRA_DEFAULT_HOST),
      sys.props.getOrElse("cport",
CASSANDRA_DEFAULT_PORT).toString.toInt
    )
  ).withCredentials(userid, password)
    .withSSL(sslOptions)
    .build()

  cluster
}
```

getsecuredCluster

```
def getsecuredCluster (truststorePath: String, trustStoreName: String,
truststorePassword: String, host: String, port: String, userid: String,
password: String): Cluster = {
  val tm =
security.getTrustManagerFromKeyStore(truststorePath.concat(trustStoreNa
me), truststorePassword)
  val sslcontext = security.getSSlContext(tm)
  val sslOptions = security.buildSSlOptions(sslcontext)
  val cluster = security.buildSecuredCluster(host, port, userid, password,
sslOptions)

  cluster
```

}

5.0 Transparent Data Encryption (TDE) at Rest

Following steps have to be performed to ensure data encryption at rest.

1. Create system key in one of the node /etc/dse/conf/ directory.
2. Copy system key to all the nodes.
3. Set ownership of keys.
4. Bounce the cluster.

1. Create system key in one of the node /etc/dse/conf/ directory.

As root on one node run following.

dsetool createsystemkey 'AES/ECB/PKCS5Padding' 128 system_key

copy system key into /etc/dse/conf/

2. Copy system key on each node

As root on each node copy system key to /etc/dse/conf. create directory if doesn't exist.

scp /etc/dse/conf/system_key datastax@dc0vm1:/etc/dse/conf/

3. Set ownership of keys

As root on each node do followings.

cd /etc/dse/
chown –R Cassandra:Cassandra /etc/dse/conf
chmod 755 /et/dse/conf/
chmod 600 /etc/dse/conf/system_key

4. Bounce the cluster

As root on each node run followings.

nodetool –h localhost drain
sudo service dse stop
sudo service dse start

5.0.1 Create and Encrypted Table

As **datastaxadmin** on any node run followings.

```
CREATE  table pos.agg_store_qtrhr_netsales (
        storeid text,
        eventdate text,
        daypart text,
        hour int,
        qtrhr text,
        areaid text,
        dayofweek int,
        dayofyear int,
        districtid text,
        divisionid text,
        enterpriseid text,
        eventtime text,
        inserttime text,
        kpiname text,
        kpivalue text,
        period text,
        PRIMARY KEY ((storeid, storedrivethrough), eventdate, daypart, hour,
        qtrhr)
)
WITH CLUSTERING ORDER BY (eventdate DESC, daypart DESC, hour DESC,
qtrhr DESC)
AND compression = {'sstable_compression':
'EncryptingSnappyCompressor',
'cipher_algorithm': 'AES/ECB/PKCS5Padding',
'secret_key_strength': 128,
'chunk_length_kb': 128,
'system_key_file': 'system_key'}
;
```

5.0.2 Verify table is Encrypted.

```
CREATE TABLE pos.agg_store_qtrhr_netsales (
    storeid text,
    eventdate text,
    daypart text,
    hour int,
    qtrhr text,
    areaid text,
    dayofweek int,
    dayofyear int,
    districtid text,
    divisionid text,
    enterpriseid text,
    eventtime text,
    inserttime text,
    kpiname text,
    kpivalue text,
    period text,
    PRIMARY KEY ((storeid, storedrivethrough), eventdate, daypart, hour,
qtrhr)
) WITH CLUSTERING ORDER BY (eventdate DESC, daypart DESC, hour
DESC, qtrhr DESC)
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy
', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '128',
'cipher_algorithm': 'AES/ECB/PKCS5Padding', 'class':
'org.apache.cassandra.io.compress.EncryptingSnappyCompressor',
'secret_key_strength': '128', 'system_key_file': 'system_key'}
```

5.0.3 Rewrite SStables as Encrypted

For tables that already exists, alter tables to perform rewrite of all SSTables.

nodetool upgradesstables –include-all-sstables.

5.0.4 Verify system_key exists on each node

As admin run following.

cqlsh -u cassandra –p cassandra –ssl

cassandra@cqlsh> select * from dse_system.encrypted_keys;

```
 key_file   | cipher | strength | key_id                                | key
------------+--------+----------+---------------------------------------+----------
----------------------------------
 system_key |    AES |      128 | 2c1081d0-ff98-11e6-b8ea-
612efc0c5c21 |
cIInbehkM+9oyYNN5M5qJgVhFMxtTFVFzmGQmBlLRkI=
```

6.0 Authentication

Following steps need to be performed for password authentication.

1. **As root on each node, modify cassandra.yaml.**

   vi /etc/dse/cassandra/cassandra.yaml

   Comment out AllowAllAuthenticator and enable PasswordAuthenticator

   # dhc authenticator: AllowAllAuthenticator
   authenticator: PasswordAuthenticator

2. **Bounce the server.**

   As root on each node run followings.

   nodetool –h localhost drain
   sudo service dse stop
   sudo service dse start

7.0 Authorization

In Role based access control (RBAC), permissions have been granted to a role as they were granted to a user. Roles can be also granted to each other.

```
CREATE ROLE supervisor;
GRANT MODIFY ON pos.divisionloc TO supervisor;
GRANT SELECT ON pos.divisionloc TO supervisor;
```

For granting a role to database user, use followings.

```
CREATE ROLE divisionmgr with PASSWORD ='div' and LOGIN =true;
GRANT SUPERVISOR to  divisionmgr;
```

To list permissions of supervisor use followings.

```
cassandra@cqlsh:pos> LIST ALL PERMISSIONS OF  supervisor;
 role       | username   | resource                 | permission
------------+------------+--------------------------+------------
 supervisor | supervisor | <table pos.divisionloc> |     SELECT
 supervisor | supervisor | <table pos.divisionloc> |     MODIFY

cassandra@cqlsh:pos> LIST ALL PERMISSIONS OF divisionmgr;
 role       | username   | resource                 | permission
------------+------------+--------------------------+------------
 supervisor | supervisor | <table pos.divisionloc> |     SELECT
 supervisor | supervisor | <table pos.divisionloc> |     MODIFY
```

```
cqlsh -u appuser -p Wh236t75n?1d --ssl
cqlsh -u devopsuser -p zLX49md6isMXJg --ssl
cqlsh -u admin -p k4MyUcvK71456y --ssl
```

```
CREATE ROLE appuser with PASSWORD ='Wh236t75n?1d' and LOGIN =true;
CREATE ROLE devopsuser with PASSWORD ='zLX49md6isMXJg' and LOGIN
=true;
```

```
CREATE ROLE admin with PASSWORD ='k4MyUcvK71456y' and LOGIN =true
and superuser=true;

GRANT EXECUTE on INTERNAL SCHEME to appuser;
GRANT EXECUTE on INTERNAL SCHEME to devopsuser;
GRANT EXECUTE on INTERNAL SCHEME to admin;

GRANT ALL PERMISSIONS ON ALL KEYSPACES to admin;

GRANT CREATE ON KEYSPACE IOT to appuser;  // grant create table
privilege on IOT keyspace;
GRANT ALTER ON KEYSPACE IOT to appuser;
GRANT DROP ON KEYSPACE IOT to appuser;
GRANT SELECT ON KEYSPACE IOT to appuser;
GRANT MODIFY ON KEYSPACE IOT to appuser;



GRANT CREATE ON KEYSPACE IOT to devopsuser;  // grant create table
privilege on IOT keyspace;
GRANT ALTER ON KEYSPACE IOT to devopsuser;
GRANT DROP ON KEYSPACE IOT to devopsuser;
GRANT SELECT ON KEYSPACE IOT to devopsuser;
GRANT MODIFY ON KEYSPACE IOT to devopsuser;

GRANT AUTHORIZE ON KEYSPACE IOT to devopsuser;
```

8.0 Data Auditing

Audit logger logs information on the node sets up for logging. Node 0 can be turned on for auditing but node 1 does not. Issuing updates and other commands on node 1 doesn't usually show up on node 0's audit log. To get maximum information from data auditing, turn on data auditing from every node.

Audit-logs can be written to filesystem log files using log4j, or to a Cassandra table. Default logger for auditing is to log into log4j filesystem log files. Each node's log files are local to the machine, making it difficult to find out what is happening across the cluster.

Logging audit data to Cassandra table helps querying like any other table, making analysis easier and custom audit reports possible.
Here are steps to be followed in order to enable audit_logging_options in Cassandra.Modify following from dse.yaml file.

audit_logging_options:
enabled: true
logger: CassandraAuditWriter

cassandra_audit_writer_options:
mode: async
dropped_event_log: /var/log/cassandra/dropped_audit_events.log

Other optional setting contains included_categoeries or exclude_categories but not both.
Here are settings can be included.

Setting	**Logging**
ADMIN	Logs describe schema versions, cluster name, version, ring, and other administration events.
AUTH	Logs login events
DML	Logs insert, update, delete and other DML events
DDL	Logs object and user create, alter, drop, and other DDL events
DCL	Logs grant, revoke, create user, drop user, and list users events

9.0 Network Security Group (NSG)

Following Inbound Network Security Rules can be applied in NSG.

Priority	Name	Port	Protocol	Source	Destination	Action
100	SSH	22	TCP	Virtual Network	Virtual Network	Allow
400	Cassandra Client	9042	TCP	Virtual Network	Virtual Network	Allow
500	Cassandra Inter Node	7000	TCP	Virtual Network	Virtual Network	Allow
600	Cassandra Inter Node SSL	7001	TCP	Virtual Network	Virtual Network	Allow
700	Cassandra JMX	7199	TCP	Virtual Network	Virtual Network	Allow
800	Internode Message	8609	TCP	Virtual Network	Virtual Network	Allow
900	DSEThirft	9060	TCP	Virtual Network	Virtual Network	Allow
4096	DenyVnet	Any	TCP	Virtual Network	Virtual Network	Deny
65000	Allow VnetInbound	Any	TCP	Virtual Network	Virtual Network	ALLOW
65001	Allow LoadBalancer Inbound	Any	Any	Load Balancer	Virtual Network	Allow
65500	Deny All Inbound	Any	Any	Any	Any	Deny