Confluent Security

This exercise provides an example of how to enable security on Confluent Platform.

Overview

This exercise provides a step-by-step example to enable SSL encryption, SASL authentication, and authorization on Confluent Platform with monitoring using Confluent Control Center. Follow the steps to walk through configuration settings for securing ZooKeeper, Apache Kafka® brokers, Kafka Connect, and Confluent Replicator, plus all the components required for monitoring, including the Confluent Metrics Reporter and Confluent Monitoring Interceptors.

Prerequisites

You should understand why it is critical to secure Confluent Platform and have a conceptual understanding of how encryption, authentication, and authorization work.

Before proceeding with this exercise:

- Use the Quick Start for Confluent Platform to bring up Confluent Platform *without* security enabled

Creating SSL Keys and Certificates

Each machine in the cluster has a public-private key pair, and a certificate to identify the machine. The certificate, however, is unsigned, which means that an attacker can create such a certificate to pretend to be any machine.

Therefore, it is important to prevent forged certificates by signing them for each machine in the cluster. A certificate authority (CA) is responsible for signing certificates. CA works like a government that issues passports - the government validates the identity of the person applying for the passport and then provides a passport in a standard form that is difficult to forge. Other governments verify the form is valid to ensure the passport is authentic. Similarly, the CA signs the certificates, and the cryptography guarantees that a signed certificate is computationally difficult to forge. Thus, as long as the CA is a genuine and trusted authority, the clients have high assurance that they are connecting to the authentic machines.

The keystore stores each machine's own identity. The truststore stores all the certificates that the machine should trust. Importing a certificate into one's truststore also means trusting all certificates that are signed by that certificate. As the analogy above, trusting the government (CA) also means trusting all passports (certificates) that it has issued. This attribute is called the chain of trust, and it is particularly useful when deploying SSL on a large Kafka cluster. You can sign all certificates in the cluster with a single CA, and have all machines share the same truststore that trusts the CA. That way all machines can authenticate all other machines.

Important

OpenSSL certificates may include an Extended Key Usage extension (`extendedKeyUsage`) to control the purpose for which the certificate public key can be used. If this field is empty, there are no restrictions on usage, but if any usage is specified, valid SSL implementations must enforce the restrictions.

Extension key usages are relevant for client and server authentication. Kafka brokers require both client and server authentication for intracluster communication because every broker is both the client and the server for the other brokers. Some corporate CAs may have a signing profile for web servers †hat is used for Kafka as well and only include the `serverAuth` usage value, causing the SSL handshake to fail.
To deploy SSL, the general steps are:

- Generate the keys and certificates
- Create your own Certificate Authority (CA)
- Sign the certificate

The steps to create keys and sign certificates are enumerated below. You may also adapt a script from confluent-platform-security-tools.git.

The definitions of the parameters used in the steps are as follows:

1. keystore: the location of the keystore
2. ca-cert: the certificate of the CA
3. ca-key: the private key of the CA
4. ca-password: the passphrase of the CA
5. cert-file: the exported, unsigned certificate of the server
6. cert-signed: the signed certificate of the server

Configuring Host Name Verification

Host name verification of servers is enabled by default for client connections as well as inter-broker connections to prevent man-in-the-middle attacks. Server host name verification may be disabled by setting `ssl.endpoint.identification.algorithm` to an empty string. For example,

```
ssl.endpoint.identification.algorithm=
```

For dynamically configured broker listeners, hostname verification may be disabled using `kafka-configs`. For example,

```
./bin/kafka-configs --bootstrap-server localhost:9093 --entity-type brokers --enti
ty-name 0 --alter \
--add-config "listener.name.internal.ssl.endpoint.identification.algorithm="
```

Configuring Host Name In Certificates

If host name verification is enabled, clients will verify the server's fully qualified domain name (FQDN) against one of the following two fields:

- Common Name (CN)
- Subject Alternative Name (SAN)

Both fields are valid, however RFC-2818 recommends the use of SAN. SAN is also more flexible, allowing for multiple DNS entries to be declared. Another advantage is that the CN can be set to a more meaningful value for authorization purposes. To add a SAN field, append the argument `-ext SAN=DNS:{FQDN}` to the keytool command:

```
keytool -keystore server.keystore.jks -alias localhost -validity {validity} -genke
y -keyalg RSA -ext SAN=DNS:{FQDN}
```

The following command can be run afterwards to verify the contents of the generated certificate:

```
keytool -list -v -keystore server.keystore.jks
```

Generate the keys and certificates

You can use Java's `keytool` utility for this process.

Generate the key and the certificate for each Kafka broker in the cluster. Generate the key into a keystore called `kafka.server.keystore` so that you can export and sign it later with CA. The keystore file contains the private key of the certificate; therefore, it needs to be kept safely.

```
# With user prompts
keytool -keystore kafka.server.keystore.jks -alias localhost -keyalg RSA -genk
ey

# Without user prompts, pass command line arguments
keytool -keystore kafka.server.keystore.jks -alias localhost -keyalg RSA -vali
dity {validity} -genkey -storepass {keystore-pass} -keypass {key-pass} -dname
{distinguished-name} -ext SAN=DNS:{hostname}
```

Ensure that the common name (CN) exactly matches the fully qualified domain name (FQDN) of the server. The client compares the CN with the DNS domain name to ensure that it is indeed connecting to the desired server, not a malicious one. The hostname of the server can also be specified in the Subject Alternative Name (SAN). Since the distinguished name is used as the server principal when SSL is used as the inter-broker security protocol, it is useful to have hostname as a SAN rather than the CN.

Create your own Certificate Authority (CA)

2. Generate a CA that is simply a public-private key pair and certificate, and it is intended to sign other certificates.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days {validity}
```

3. Add the generated CA to the **clients' truststore** so that the clients can trust this CA:

```
keytool -keystore kafka.client.truststore.jks -alias CARoot -importcert -file
ca-cert
```

4. Add the generated CA to the **brokers' truststore** so that the brokers can trust this CA.

```
keytool -keystore kafka.server.truststore.jks -alias CARoot -importcert -file
ca-cert
```

Sign the certificate

To sign all certificates in the keystore with the CA that you generated:

5. Export the certificate from the keystore:

```
keytool -keystore kafka.server.keystore.jks -alias localhost -certreq -file ce
rt-file
```

6. Sign it with the CA:

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -da
ys {validity} -CAcreateserial -passin pass:{ca-password}
```

7. Import both the certificate of the CA and the signed certificate into the broker keystore:

```
keytool -keystore kafka.server.keystore.jks -alias CARoot -importcert -file ca
-cert
```

```
keytool -keystore kafka.server.keystore.jks -alias localhost -importcert -file
cert-signed
```

Summary

Combining the steps described above, the script to create the CA and broker and client truststores and keystores is as follows:

```
keytool -keystore kafka.server.keystore.jks -alias localhost -keyalg RSA -validity
{validity} -genkey
openssl req -new -x509 -keyout ca-key -out ca-cert -days {validity}
keytool -keystore kafka.client.truststore.jks -alias CARoot -importcert -file ca-c
ert
keytool -keystore kafka.server.truststore.jks -alias CARoot -importcert -file ca-c
ert
keytool -keystore kafka.server.keystore.jks -alias localhost -certreq -file cert-f
ile
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days {
validity} -CAcreateserial -passin pass:{ca-password}
keytool -keystore kafka.server.keystore.jks -alias CARoot -importcert -file ca-cer
t
keytool -keystore kafka.server.keystore.jks -alias localhost -importcert -file cer
t-signed
```

Configure ZooKeeper

For broker to ZooKeeper communication, we will configure optional DIGEST-MD5 authentication.

1. Set the authentication provider:

```
authProvider.sasl=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

2. Configure the `zookeeper_jaas.conf` file as follows. Note the two semicolons.

```
Server {
       org.apache.zookeeper.server.auth.DigestLoginModule required
       user_super="admin-secret"
       user_kafka="kafka-secret";
};
```

3. When you start ZooKeeper, pass the name of its JAAS file as a JVM parameter:

```
export KAFKA_OPTS="-Djava.security.auth.login.config=etc/kafka/zookeeper_jaas.
conf"
bin/zookeeper-server-start etc/kafka/zookeeper.properties
```

.

Configure Brokers

Administrators can configure a mix of secure and unsecured clients. This tutorial ensures that all broker/client and inter-broker network communication is encrypted in the following manner:

- All broker/client communication use `SASL_SSL` security protocol, which ensures that the communication is encrypted and authenticated using SASL/PLAIN
- All inter-broker communication use `SSL` security protocol, which ensures that the communication is encrypted and authenticated using SSL
- The unsecured `PLAINTEXT` port is not enabled

The steps are as follows:

1. Enable the desired security protocols and ports in each broker's `server.properties`. Notice that both `SSL` and `SASL_SSL` are enabled.

```
listeners=SSL://:9093,SASL_SSL://:9094
```

2. To enable the brokers to authenticate each other (two-way TLS/SSL authentication), you need to configure all the brokers for client authentication (in this case, the requesting broker is the "client"). We recommend setting `ssl.client.auth=required`. We discourage configuring it as `requested` because misconfigured brokers will still connect successfully and it provides a false sense of security.

```
security.inter.broker.protocol=SSL
ssl.client.auth=required
```

3. Define the SSL truststore, keystore, and password in the `server.properties` file of every broker. Since this stores passwords directly in the broker configuration file, it is important to restrict access to these files using file system permissions.

```
ssl.truststore.location=/var/ssl/private/kafka.server.truststore.jks
ssl.truststore.password=test1234
ssl.keystore.location=/var/ssl/private/kafka.server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
```

4. Enable SASL/PLAIN mechanism in the `server.properties` file of every broker.

```
sasl.enabled.mechanisms=PLAIN
```

5. Create the broker's JAAS configuration file in each Kafka broker's config directory, let's call it `kafka_server_jaas.conf` for this example.

- Configure a `KafkaServer` section used when the broker validates client connections, including those from other brokers. The properties `username` and `password` are used by the broker to initiate connections to other brokers, and in this example, `kafkabroker` is the user for inter-broker communication. The set of properties `user_{userName}` defines the passwords for all other clients that connect to the broker. In this example, there are two users `kafkabroker` and `client`.
- Configure a `Client` section used when the broker connects to ZooKeeper. There is a single username and password that must match the ZooKeeper jaas configuration.

Note

Note the two semicolons in each section.

```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   username="kafkabroker"
   password="kafkabroker-secret"
   user_kafkabroker="kafkabroker-secret"
   user_kafka-broker-metric-reporter="kafkabroker-metric-reporter-secret"
   user_client="client-secret";
};

Client {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   username="kafka"
   password="kafka-secret";
};
```

6. If you are using Confluent Control Center to monitor your deployment, and if the monitoring cluster backing Confluent Control Center is also configured with the same security protocols, you must configure the Confluent Metrics Reporter for security as well. Add these configurations to the `server.properties` file of each broker.

```
metric.reporters=io.confluent.metrics.reporter.ConfluentMetricsReporter
confluent.metrics.reporter.security.protocol=SASL_SSL
confluent.metrics.reporter.ssl.truststore.location=/var/ssl/private/kafka.serv
er.truststore.jks
confluent.metrics.reporter.ssl.truststore.password=test1234
confluent.metrics.reporter.sasl.mechanism=PLAIN
confluent.metrics.reporter.sasl.jaas.config=org.apache.kafka.common.security.p
lain.PlainLoginModule required \
   username="kafka-broker-metric-reporter" \
   password="kafka-broker-metric-reporter-secret";
```

7. To enable ACLs, we need to configure an authorizer. Kafka provides a simple authorizer implementation, and to use it, you can add the following to `server.properties`:

```
authorizer.class.name=kafka.security.auth.AclAuthorizer
```

8. The default behavior is such that if a resource has no associated ACLs, then no one is allowed to access the resource, except super users. Setting broker principals as super users is a convenient way to give them the required access to perform inter-broker operations. Because this tutorial configures the inter-broker security protocol as SSL, set the super user name to be the `distinguished name` configured in the broker’s certificate. (See other authorization configuration options).

```
super.users=User:<DN of broker1>;User:<DN of broker2>;User:<DN of broker3>;Use
r:kafka-broker-metric-reporter
```

Combining the configuration steps described above, the broker’s `server.properties` file contains the following configuration settings:

```
# Enable SSL security protocol for inter-broker communication
# Enable SASL_SSL security protocol for broker-client communication
listeners=SSL://:9093,SASL_SSL://:9094
security.inter.broker.protocol=SSL
ssl.client.auth=required

# Broker security settings
ssl.truststore.location=/var/ssl/private/kafka.server.truststore.jks
ssl.truststore.password=test1234
ssl.keystore.location=/var/ssl/private/kafka.server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
sasl.enabled.mechanisms=PLAIN

# Confluent Metrics Reporter for monitoring with Confluent Control Center
metric.reporters=io.confluent.metrics.reporter.ConfluentMetricsReporter
confluent.metrics.reporter.security.protocol=SASL_SSL
confluent.metrics.reporter.ssl.truststore.location=/var/ssl/private/kafka.server.t
ruststore.jks
confluent.metrics.reporter.ssl.truststore.password=test1234
confluent.metrics.reporter.sasl.mechanism=PLAIN
confluent.metrics.reporter.sasl.jaas.config=org.apache.kafka.common.security.plain
.PlainLoginModule required \
   username="kafka-broker-metric-reporter" \
   password="kafkabroker-metric-reporter-secret";

# ACLs
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:<DN of broker1>;User:<DN of broker2>;User:<DN of broker3>
```

Note

This is not the full Broker configuration. This is just the additional configurations required to enable security on a known working Kafka cluster of brokers that is already successfully monitored via Confluent Control Center.
Start each Kafka broker. Pass the name of the JAAS file as a JVM parameter:

```
export KAFKA_OPTS=-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.c
onf
bin/kafka-server-start etc/kafka/server.properties
```

Alternatively, you can modify the configuration file with the following:

```
listener.name.sasl_ssl.plain.sasl.jaas.config=org.apache.kafka.common.security.pla
in.PlainLoginModule required \
   username="admin" \
   password="admin-secret" \
   user_admin="admin-secret" \
   user_kafkabroker1="kafkabroker1-secret";
```

Configure Clients

Common Configuration

Any component that interacts with secured Kafka brokers is a *client* and must be configured for security as well. These clients include Kafka Connect workers and certain connectors such as Replicator, Kafka Streams API clients, ksqlDB clients, non-Java clients, Confluent Control Center, Confluent Schema Registry, REST Proxy, etc.

All clients share a general set of security configuration parameters required to interact with a secured Kafka cluster:

1. To encrypt via SSL and authenticate via SASL, configure the security protocol to use `SASL_SSL`. (If you wanted SSL for both encryption and authentication without SASL, the security protocol would be `SSL`).

   ```
   security.protocol=SASL_SSL
   ```

2. To configure TLS/SSL encryption truststore settings, set the truststore configuration parameters. In this tutorial, the client does not need the keystore because authentication is done via SASL/PLAIN instead of two-way TLS/SSL.

   ```
   ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
   ssl.truststore.password=test1234
   ```

3. To configure SASL authentication, set the SASL mechanism, which in this tutorial is `PLAIN`. Then configure the JAAS configuration property to describe to connect to the Kafka brokers. The properties `username` and `password` are used to configure the user for connections.

   ```
   sasl.mechanism=PLAIN
   ```

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule requi
red \
   username="client" \
   password="client-secret";
```

Combining the configuration steps above, the client's general pattern for enabling SSL encryption and SASL/PLAIN authentication is to add the following to the client's properties file.

```
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
\
    username="client" \
    password="client-secret";
```

What differs between the clients is the specific configuration prefix that precedes each configuration parameter, as described in the sections below.

Configure Console Producer and Consumer

The command line tools for console producer and consumer are convenient ways to send and receive a small amount of data to the cluster. They are clients and thus need security configurations as well.

1. Create a `client_security.properties` file with the security configuration parameters described above, with no additional configuration prefix.

```
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule requi
red \
    username="client" \
    password="client-secret";
```

2. Pass in the properties file when using the command line tools.

```
kafka-console-producer --broker-list kafka1:9094 --topic test-topic --producer
.config client_security.properties
kafka-console-consumer --bootstrap-server kafka1:9094 --topic test-topic --con
sumer.config client_security.properties
```

Configure ksqlDB and Stream Processing Clients

Enabling ksqlDB and stream processing clients for security is simply a matter of passing the security configurations to the relevant client constructor.

Take the basic client security configuration:

```
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
\
    username="client" \
    password="client-secret";
```

And configure the application for the following:

- Top-level, with no additional configuration prefix
- Confluent Monitoring Interceptor producer used for Confluent Control Center streams monitoring, with an additional configuration prefix `producer.confluent.monitoring.interceptor.`
- Confluent Monitoring Interceptor consumer used for Confluent Control Center streams monitoring, with an additional configuration prefix `consumer.confluent.monitoring.interceptor.`

Combining these configurations, ksqlDB configuration for SSL encryption and SASL/PLAIN authentication is the following. You may configure them by either loading the properties from a file, as shown below, or by setting the properties programmatically.

```
# Top level
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
username="client" password="client-secret";

# Embedded producer for streams monitoring with Confluent Control Center
producer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gProducerInterceptor
producer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
producer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
producer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
producer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
producer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required username="client" password="client-secre
t";

# Embedded consumer for streams monitoring with Confluent Control Center
consumer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gConsumerInterceptor
```

```
consumer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
consumer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
consumer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
consumer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
consumer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required username="client" password="client-secre
t";
```

Note

This is not the full configuration. This is just the additional configurations required to enable security on a known working ksqlDB or Java application that is monitored via Confluent Control Center.

Configure Kafka Connect

From the perspective of the brokers, Kafka Connect is another client, and this tutorial configures Connect for SSL encryption and SASL/PLAIN authentication. Enabling Kafka Connect for security is simply a matter of passing the security configurations to the Connect workers, the producers used by source connectors, and the consumers used by sink connectors.

Take the basic client security configuration:

```
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
\
    username="client" \
    password="client-secret";
```

And configure Kafka Connect for the following:

- Top-level for Connect workers, with no additional configuration prefix
- Embedded producer for source connectors, with an additional configuration prefix `producer.`
- Embedded consumers for sink connectors, with an additional configuration prefix `consumer.`
- Confluent Monitoring Interceptor producer for source connectors used for Confluent Control Center streams monitoring, with an additional configuration prefix `producer.confluent.monitoring.interceptor.`
- Confluent Monitoring Interceptor consumer for sink connectors used for Confluent Control Center streams monitoring, with an additional configuration prefix `consumer.confluent.monitoring.interceptor.`

Combining these configurations, a Kafka Connect worker configuration for SSL encryption and SASL/PLAIN authentication is the following. You may configure these settings in the `connect-distributed.properties` file.

```
# Connect worker
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
\
   username="connect" \
   password="connect-secret";

# Embedded producer for source connectors
producer.security.protocol=SASL_SSL
producer.ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
producer.ssl.truststore.password=test1234
producer.sasl.mechanism=PLAIN
producer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \
  username="connect" \
  password="connect-secret";

# Embedded consumer for sink connectors
consumer.security.protocol=SASL_SSL
consumer.ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
consumer.ssl.truststore.password=test1234
consumer.sasl.mechanism=PLAIN
consumer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \
  username="connect" \
  password="connect-secret";

# Embedded producer for source connectors for streams monitoring with Confluent Co
ntrol Center
producer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gProducerInterceptor
producer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
producer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
producer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
producer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
producer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required \
  username="connect" \
  password="connect-secret";

# Embedded consumer for sink connectors for streams monitoring with Confluent Cont
rol Center
consumer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gConsumerInterceptor
consumer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
consumer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
consumer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
consumer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
consumer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required \
  username="connect" \
  password="connect-secret";
```

Note

This is not the full Connect worker configuration. This is just the additional configurations required to enable security on a known working Kafka Connect cluster that is already successfully monitored via Confluent Control Center.
Pass in the properties file when starting each Connect worker.

```
bin/connect-distributed etc/kafka/connect-distributed.properties
```

Replicator

Confluent Replicator is a type of Kafka source connector that replicates data from a source to destination Kafka cluster. An embedded consumer inside Replicator consumes data from the source cluster, and an embedded producer inside the Kafka Connect worker produces data to the destination cluster.

Take the basic client security configuration:

```
security.protocol=SASL_SSL
ssl.truststore.location=/var/ssl/private/kafka.client.truststore.jks
ssl.truststore.password=test1234
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
\
    username="client" \
    password="client-secret";
```

And configure Replicator for the following:

- Top-level Replicator consumer from the origin cluster, with an additional configuration prefix `src.kafka.`
- Confluent Monitoring Interceptor consumer from the origin cluster used for Confluent Control Center streams monitoring, with an additional configuration prefix `src.consumer.confluent.monitoring.interceptor.`

Combining the configuration steps described above, the Replicator JSON properties file contains the following configuration settings:

```
{
  "name":"replicator",
  "config":{
    ....
    "src.kafka.security.protocol" : "SASL_SSL",
    "src.kafka.ssl.truststore.location" : "var/private/ssl/kafka.server.truststore
.jks",
    "src.kafka.ssl.truststore.password" : "test1234",
    "src.kafka.sasl.mechanism" : "PLAIN",
    "src.kafka.sasl.jaas.config" : "org.apache.kafka.common.security.plain.PlainLo
ginModule required username=\"replicator\" password=\"replicator-secret\";",
    "src.consumer.interceptor.classes": "io.confluent.monitoring.clients.intercept
or.MonitoringConsumerInterceptor",
    "src.consumer.confluent.monitoring.interceptor.security.protocol": "SASL_SSL",
    "src.consumer.confluent.monitoring.interceptor.ssl.truststore.location": "/var
/ssl/private/kafka.client.truststore.jks",
    "src.consumer.confluent.monitoring.interceptor.ssl.truststore.password": "conf
luent",
    "src.consumer.confluent.monitoring.interceptor.sasl.mechanism": "PLAIN",
    "src.consumer.confluent.monitoring.interceptor.sasl.jaas.config": "org.apache.
kafka.common.security.plain.PlainLoginModule required username="client" password="
client-secret";",
    ....
  }
}
```

Note

This is not the full Replicator configuration. Rather, it shows the additional configurations required to enable security on a known working Replicator connector that is already successfully monitored using Confluent Control Center.
After Kafka Connect is started, you can add the Confluent Replicator:

```
curl -X POST -H "Content-Type: application/json" --data @replicator_properties.jso
n http://connect:8083/connectors
```

Confluent Control Center

Confluent Control Center uses Kafka Streams for stream processing, so if all the Kafka brokers in the monitoring cluster backing Control Center are secured, then the Control Center application, another client, also needs to be secured.

Take the basic client security configuration and add the configuration prefix `confluent.controlcenter.streams.` Make all the following modifications in the `etc/confluent-control-center/control-center.properties` file:

```
confluent.controlcenter.streams.security.protocol=SASL_SSL
confluent.controlcenter.streams.ssl.truststore.location=/var/ssl/private/kafka.cli
ent.truststore.jks
confluent.controlcenter.streams.ssl.truststore.password=test1234
confluent.controlcenter.streams.sasl.mechanism=PLAIN
confluent.controlcenter.streams.sasl.jaas.config=org.apache.kafka.common.security.
plain.PlainLoginModule required \
  username="confluent" \
  password="confluent-secret";
```

Note

This is not the full Confluent Control Center configuration. Rather, it shows the additional configurations required to enable security on a known working Confluent Control Center deployment.
Start Confluent Control Center.

```
bin/control-center-start etc/confluent-control-center/control-center.properties
```

Confluent Metrics Reporter

If you are using Confluent Control Center to monitor your deployment, the Confluent Metrics Reporter is a client as well. If the monitoring cluster backing Confluent Control Center is also configured with the same security protocols, then configure the Confluent Metrics Reporter for security in each broker's `server.properties` file. The configuration prefix is `confluent.metrics.reporter.` and is described above.

Confluent Monitoring Interceptors

Configure security for the components that are using Confluent Monitoring Interceptors to report stream monitoring statistics in Confluent Control Center:

```
# Embedded producer for streams monitoring with Confluent Control Center
producer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gProducerInterceptor
producer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
producer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
producer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
producer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
producer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required username="client" password="client-secre
t";

# Embedded consumer for streams monitoring with Confluent Control Center
consumer.interceptor.classes=io.confluent.monitoring.clients.interceptor.Monitorin
gConsumerInterceptor
consumer.confluent.monitoring.interceptor.security.protocol=SASL_SSL
consumer.confluent.monitoring.interceptor.ssl.truststore.location=/var/ssl/private
/kafka.client.truststore.jks
consumer.confluent.monitoring.interceptor.ssl.truststore.password=test1234
consumer.confluent.monitoring.interceptor.sasl.mechanism=PLAIN
consumer.confluent.monitoring.interceptor.sasl.jaas.config=org.apache.kafka.common
.security.plain.PlainLoginModule required username="client" password="client-secre
t";
```

Authorization and ACLs

Use Centralized ACLs to add, remove, or list ACLs when using RBAC. The most common tasks for ACL management are adding or removing a principal as a producer or consumer. For example, to add a client called `client-1` as a producer and consumer of a topic called `test-topic`, execute the following:

```
confluent iam acl create --allow --principal User:client-1 --operation write --top
ic test-topic --kafka-cluster-id <kafka-cluster-id>
confluent iam acl create --allow --principal User:client-1 --operation read --topi
c test-topic --kafka-cluster-id <kafka-cluster-id>
```

For additional CLI details, refer to the ACL subcommands in confluent iam. If you are not running RBAC, then refer to Authorization using ACLs.

Troubleshooting

In cases where the configuration does not work on the first attempt, debugging output is a helpful way to diagnose the cause of the problem:

1. Validate the keys and certificates in the keystores and truststores in the brokers and clients.

   ```
   keytool -list -v -keystore /var/ssl/private/kafka.server.keystore.jks
   ```

2. Enable Kafka authorization logging by modifying the `etc/kafka/log4j.properties` file. Change the log level to DEBUG, and then restart the brokers.

   ```
   log4j.logger.kafka.authorizer.logger=DEBUG, authorizerAppender
   ```

3. Enable SSL debug output by using the `javax.net.debug` system property, which requires a restart of the JVM.

   ```
   export KAFKA_OPTS=-Djavax.net.debug=all
   ```

4. Enable SASL debug output using the `sun.security.krb5.debug` system property, which requires a restart of the JVM.

   ```
   export KAFKA_OPTS=-Dsun.security.krb5.debug=true
   ```