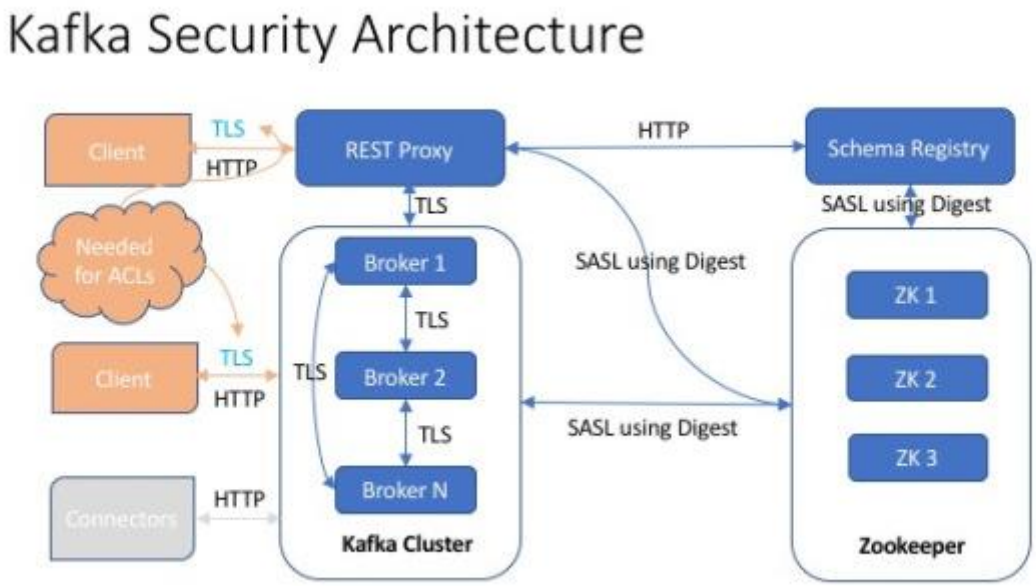
Kafka Security using SSL

Architecture

Following diagram explains how the architecture looks for the PoC.



We are going to use SSL for authenticating to the cluster as well as inter-broker communication.

Details

Secure Zookeeper

Since we are going to implement authorization, it's imperative that we secure zookeeper first. This is where Kafka stores all ACLs (Access Control Lists) and we need to ensure only an Administrator can create/modify/delete them. If we don't do this step, we defeat the purpose of securing the cluster, when it comes to Authorization.

ZooKeeper provides SASL based authentication mechanism that we're going to use to secure the cluster. Confluent distribution provided startup scripts for all the services that we'll modify to suit our security needs (such as passing JAAS file as an argument).

```
# tail -2 /bin/zookeeper-server-start-secure

exec $base_dir/kafka-run-class $EXTRA_ARGS -
Djava.security.auth.login.config=/etc/kafka/zookeeper_server_jaas.properties
org.apache.zookeeper.server.quorum.QuorumPeerMain "$@"
```

JAAS file will contain user credentials.

```
# cat /etc/kafka/zookeeper_server_jaas.properties
/*****
* user/password is used for connections between ZK instances.
* user_zk_admin defines a user that client uses to authenticate with this ZK instance.
*****/

Server {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   user="zk_admin"
   password="zk_admin_secret"
   user_zk_admin="zk_admin_secret";
};
```

Finally, configure Zookeeper to use SASL mechanism for authentication in it's properties file.

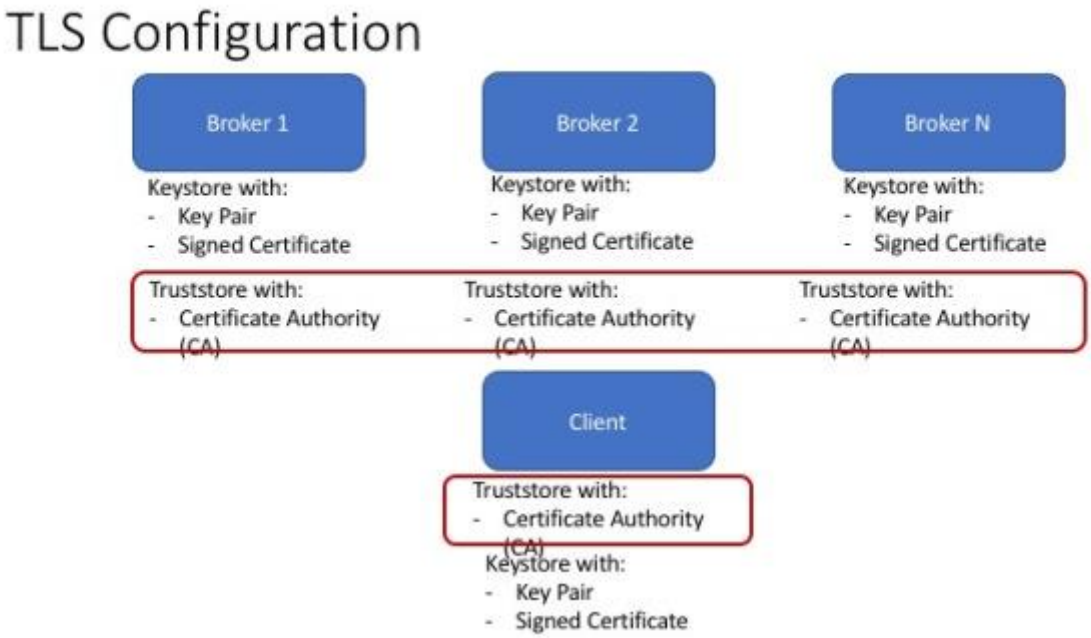
```
# SASL configuration
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
requireClientAuthScheme=sasl
```

Start zookeeper ensemble with this configuration.

```
# /bin/zookeeper-server-start-secure -daemon /etc/kafka/zookeeper.properties
```

Secure Brokers using TLS

First thing we need to do is to generate a key and certificate for each of the brokers and each client in the ecosystem.The common name (CN) of the broker certificate must match the fully qualified domain name (FQDN) of the server, since during authentication, client will compare the CN with the DNS domain name to ensure that it is connecting to the desired broker. For the purpose of PoC, we used self signed certificates. Following picture shows the setup that we'll use for the setup.



You may use scripts to automate the most of the TLS setup.

Generate CA and Truststore

Generate public-private key pair and certificate for the CA and add the same CA certificate to each broker's truststore.

```
#!/bin/bash

PASSWORD=test1234
CLIENT_PASSWORD=test1234
VALIDITY=365

# Generate CA (certificate authority) public-private key pair and certificate, and it is intended to sign other certificates.
openssl req -new -x509 -keyout ../ca/ca-key -out ../ca/ca-cert -days $VALIDITY -passin pass:$PASSWORD -passout pass:$PASSWORD -subj "/C=US/ST=CA/L=San Jose/O=Company/OU=Org/CN=FQDN" -nodes

# Add the CA to the servers' truststore
keytool -keystore /etc/kafka/ssl-server-steps/ca/kafka.server.truststore.jks -alias CARoot -import -file /etc/kafka/ssl-server-steps/ca/ca-cert -storepass $PASSWORD -keypass $PASSWORD
```

Keystore stores each application's identity, the truststore stores all the certificates that the application should trust. Importing a certificate into one's truststore also means trusting all certificates that are signed by that certificate. This attribute is called the chain of trust, and it is particularly useful when deploying TLS 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 contains the CA

certificate. That way all machines can authenticate all other machines. This is as depicted in the picture above.

Generate Certificate/key pair for brokers

Next, generate a certificate/key pair for each of the brokers using this script.

```
#!/bin/bash

PASSWORD=test1234
VALIDITY=365

if [ $# -lt 1 ];
then
  echo "`basename $0` <host fqdn|user name|app name>"
  exit 1
fi

CNAME=$1
ALIAS=`echo $CNAME|cut -f1 -d"."`


# Generate keypair, ensure CN matches exactly with the FQDN of the server.
keytool -noprompt -keystore kafka.server.keystore.jks -alias $ALIAS -keyalg RSA -validity
$VALIDITY -genkey -dname "CN=$CNAME,OU=BDP,O=Company,L=San Jose,S=CA,C=US"
-storepass $PASSWORD -keypass $PASSWORD

# The next step is to sign all certificates in the keystore with the CA we generated in another step
# First, you need to export the certificate from the keystore
keytool -keystore kafka.server.keystore.jks -alias $ALIAS -certreq -file cert-file -storepass
$PASSWORD

# Then, sign it with the CA that was generated earlier
openssl x509 -req -CA /etc/kafka/ssl-server-steps/ca/ca-cert -CAkey /etc/kafka/ssl-server-
steps/ca/ca-key -in cert-file -out cert-signed -days $VALIDITY -CAcreateserial -passin
pass:$PASSWORD

# Finally, you need to import both the certificate of CA and the signed certificate into the
keystore
keytool -keystore kafka.server.keystore.jks -alias CARoot -import -file /etc/kafka/ssl-server-
steps/ca/ca-cert -storepass $PASSWORD
keytool -keystore kafka.server.keystore.jks -alias $ALIAS -import -file cert-signed -storepass
$PASSWORD
```

Save above commands in a file and run it with broker FQDN as argument to generate the key and certificate for that broker.

```
# ./generate_keystore.sh <FQDN>
```

Copy the broker keystore to each broker in a desired location. In addition, we need to copy same trust store that has CAs certificate to each broker as well.

Configure brokers

We will run the brokers that'll server both SSL as well as non SSL traffic, since not every client may support SSL (or want to). The configuration looks like this for the brokers.

```
listeners=PLAINTEXT://<FQDN>:9092,SSL://<FQDN>:10092
security.inter.broker.protocol=SSL
zookeeper.set.acl=true
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer

# By default, if a Resource R has no associated ACLs, no one other than super users is allowed to access R.
# We want the opposite, i.e. if ACL is not defined, allow all the access to the resource R
allow.everyone.if.no.acl.found=true

################ Kafka SSL #####################
ssl.keystore.location=/etc/kafka/ssl/kafka.server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
ssl.truststore.location=/etc/kafka/ssl/kafka.server.truststore.jks
ssl.truststore.password=test1234
ssl.client.auth=requested

# In order to enable hostname verification
ssl.endpoint.identification.algorithm=HTTPS
```

Finally, when we use REST client to authenticate with REST Proxy, it passes the principal name to the brokers, which by default is DN (Distinguished Name) in clients' certificate. We use a custom function to translate that into something that looks familiar. Following is the setting for it.

```
# Custom SSL principal builder
principal.builder.class=customPrincipalBuilderClass
```

This function breaks up the DN ***"CN=$CNAME,OU=BDP,O=Company,L=San Jose,S=CA,C=US"*** and translated principal name to ***"User=$CNAME".*** This is useful since Kafka ACLs follow this format, that we'll show later. For this to work, just drop the jar file containing ***customPrincipalBuilderClass*** into /usr/share/java/kafka directory.

Start brokers

Since we have secured zookeeper, we need to start brokers with a JAAS file that contains credentials to interact with zookeeper ensemble. Again, we copy the startup script to suit our need here and use it.

```
# tail -2 /bin/kafka-server-start-secure

exec $base_dir/kafka-run-class $EXTRA_ARGS -Djava.security.auth.login.config=/etc/kafka/zookeeper_client_jaas.properties io.confluent.support.metrics.SupportedKafka "$@"
```

Contents of the JAAS file are as follows:

```
# cat /etc/kafka/zookeeper_client_jaas.properties
Client {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   username="zk_admin"
   password="zk_admin_secret";
};
```
Start broker using this script and earlier mentioned configuration.

```
# /bin/kafka-server-start-secure -daemon /etc/kafka/server.properties
```

Configure Schema Registry and Start

Although it's possible to use SSL for communication between brokers and schema registry, we didn't see a need for it. Schema registry stores metadata about topic structures that are used by HDFS connectors to sink topic level data to Hive tables. Since this data is not super confidential, we decided to go with simpler configuration for it. However, since this component interacts with secure Zookeeper, we need to ensure we pass right credentials to the daemon.

Key configuration file parametes:

```
kafkastore.bootstrap.servers=PLAINTEXT://<Broker FQDN>:9092
zookeeper.set.acl=true
```
Here's how the startup script looks like:

```
# tail -2 /bin/schema-registry-start-secure

exec $(dirname $0)/schema-registry-run-class ${EXTRA_ARGS} -
Djava.security.auth.login.config=/etc/schema-registry/schema_registry_client_jaas.properties
io.confluent.kafka.schemaregistry.rest.SchemaRegistryMain "$@"
```
Contents of the JAAS file:

```
# cat /etc/schema-registry/schema_registry_client_jaas.properties
/* Zookeeper client */
Client {
   org.apache.zookeeper.server.auth.DigestLoginModule required
   username="zk_admin"
   password="zk_admin_secret";
};
```
Start the schema registry, like so:

```
# /bin/schema-registry-start-secure -daemon /etc/schema-registry/schema-registry.properties
```

Configure REST

Typically, you would need to run 2 separate instances of REST Proxy, one to server SSL traffic (which is a requirement for Kafka ACLs) and other one for non SSL traffic. We'll discuss SSL configuration here. Here's how the configuration file looks like:

```
listeners=https://0.0.0.0:8083
schema.registry.url=http://localhost:8081
zookeeper.connect=FQDN1:2181,FQDN2:2181,FQDN3:2181
bootstrap.servers=SSL://FQDN1:10092

# ********************************
# Kafka security
# ********************************

kafka.rest.resource.extension.class=io.confluent.kafkarest.security.KafkaRestSecurityResourceExtension

# Principal propagation for the incoming requests is determined by following - Only SSL allowed and is mandatory
confluent.rest.auth.propagate.method=SSL

# Configuration Options for HTTPS
ssl.client.auth=true
ssl.keystore.location=/etc/kafka/ssl/kafka.server.keystore.jks
ssl.keystore.password=test1234
ssl.key.password=test1234
ssl.truststore.location=/etc/kafka/ssl/kafka.server.truststore.jks
ssl.truststore.password=test1234

# Configuration Options for SSL Encryption between REST Proxy and Apache Kafka Brokers
client.security.protocol=SSL
client.ssl.key.password=client1234
client.ssl.keystore.location=/etc/kafka/ssl/kafka.rest.keystore.jks
client.ssl.keystore.password=client1234
client.ssl.truststore.location=/etc/kafka/ssl/kafka.server.truststore.jks
client.ssl.truststore.password=test1234
```

Keystore file (kafka.rest.keystore.jks) that is needed to communicate with brokers is generated in same way as brokers (using generate_keystore.sh script). One difference is it'll use a user name as alias instead of server name for the CN. Ensure that the certificate is signed by same CA certificate that was generated earlier and which was used to sign broker certificates. We generated the keystore to have 2 certificates, one generated for a user called **secure_user** and another one called **inscure_user**. These user names are used as CN for the certificates and stored in the same key store.

```
# ./client.sh secure_user
# ./client.sh insecure_user
```

Finally, notice that we use same trust store as the brokers.

Create topic level ACLs

Create a topic called test_secure_topic and add ACLs to it.

```
User:secure_user has Allow permission for operations: Read from hosts: <REST Proxy host>
User:secure_user has Allow permission for operations: Describe from hosts:<REST Proxy host>
User:secure_user has Allow permission for operations: Write from hosts:<REST Proxy host>
User:* has Allow permission for operations: Describe from hosts: *
```

Last ACL (anyone can describe this topic from anywhere) is needed to get a meaningful error message when someone who has no access tries an operation on the topic. If this ACL is absent, he'll get UNKNOWN_TOPIC_OR_PARTITION error, which is cryptic.

Test Java client

We use console producer and consoler programs to test the ACLs against the topic.

```
kafka-avro-console-producer --broker-list <Broker FQDN>:10092 \
                        --topic test_secure_topic \
                        --producer.config client_ssl.properties \
                        --property
value.schema='{"type":"record","name":"test","fields":[{"name":"name","type":"string"},{"name
":"salary","type":"int"}]}' <<EOF
{"name": "console", "salary": 2000}
EOF
```

Producer config file contents are as follows:

```
security.protocol=SSL
ssl.keystore.location=kafka.client.keystore.jks
ssl.keystore.password=client1234
ssl.key.password=client1234
ssl.truststore.location=kafka.server.truststore.jks
ssl.truststore.password=client1234
```

kafka.client.keystore.jks is generated just like what we did for REST Proxy, except it uses certificate for just ***secure_user*** alias (generated by ***./client.sh secure_user*** command) .We also use same trust store that's used by everybody else.
Let's consume that record that we ingested like below:

```
kafka-avro-console-consumer --bootstrap-server <Broker FQDN>:10092 --topic
test_secure_topic --consumer.config client_ssl.properties --from-beginning --new-consumer

{"name":"console","salary":2000}
```

Let's try the same with a certificate that's generated for ***insecure_user*** alias (generated by ***./client.sh insecure_user*** command).

```
[2017-11-09 22:57:51,435] ERROR Error processing message, terminating consumer
process:  (kafka.tools.ConsoleConsumer$:105)
org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized to access topics:
[test_secure_topic]
```

Test REST client

We'll use **curl** for testing ACLs via REST. Since curl doesn't understand java key store, we need to extract and store certificate and keys for the client in PEM (Privacy Enhanced Mail) format. We used a script to automate that portion.

```
#!/bin/bash
```

```
PASSWORD=client1234
if [ $# -lt 1 ];
then
  echo "`basename $0` <alias>"
  exit 1
fi

ALIAS=$1

rm -f /etc/kafka/ssl-client-steps/certs/kafka.client.keystore.p12

# Convert JKS keystore into PKCS#12 keystore, then into PEM file:
keytool -importkeystore -srckeystore kafka.client.keystore.jks \
  -destkeystore kafka.client.keystore.p12 \
  -srcalias $ALIAS \
  -destalias $ALIAS \
  -srcstoretype jks \
  -deststoretype pkcs12 \
  -srcstorepass $PASSWORD \
  -deststorepass $PASSWORD

openssl pkcs12 -in kafka.client.keystore.p12 -chain -name $ALIAS -out ${ALIAS}.pem -passin
pass:$PASSWORD -nodes
```

For example, to extract keys and certificate for ***secure_user***, run the script with ***secure_user*** alias as the argument. This will create ***secure_user.pem*** file, that'll contain keys and certificate for that user.

```
# ../convert.sh secure_user
MAC verified OK
```

Whenever we make a call to REST Proxy using curl, we need to send the CA certificate and PEM file as arguments.

```
# Produce a message with JSON data

curl --tlsv1.0 --cacert <Location of CA Cert> -E secure_user.pem \
    -X POST -H "Content-Type: application/vnd.kafka.avro.v2+json" \
    -H "Accept: application/vnd.kafka.v2+json" \
    --data @test.data \
     https://<FQDN>:8083/topics/test_secure_topic
```

test.data file has the following:

```
{"value_schema_id": 21,"records": [ {
      "value": {
        "name":"REST API",
        "salary":1000
      }
    }
  ]
}
```

Note that we have registered that schema in schema registry earlier:

```
curl localhost:8081/schemas/ids/21
```

```
{"schema":"{\"type\":\"record\",\"name\":\"test\",\"fields\":[{\"name\":\"name\",\"type\":\"string\"}
,{\"name\":\"salary\",\"type\":\"int\"}]}"}
```
Consume the message that was created

```
# Create a consumer for AVRO data, starting at the beginning of the topic's
# log and subscribe to a topic. Then consume some data using the base URL in the first response.
# Finally, close the consumer with a DELETE to make it leave the group and clean up
# its resources.

curl --tlsv1.0 --cacert <Location of CA Cert> -E secure_user.pem \
    -X POST -H "Content-Type: application/vnd.kafka.v2+json" \
    --data '{"name": "my_consumer_instance", "format": "avro", "auto.offset.reset": "earliest"}' \
     https://<REST Proxy FQDN>:8083/consumers/secure_topic_group

curl --tlsv1.0 --cacert <Location of CA Cert> -E secure_user.pem \
    -X POST -H "Content-Type: application/vnd.kafka.json.v2+json" \
    --data '{"topics":["test_secure_topic"]}' \
    https://<REST Proxy
FQDN>:8083/consumers/secure_topic_group/instances/my_consumer_instance/subscription

curl --tlsv1.0 --cacert <Location of CA Cert> -E secure_user.pem \
    -X GET -H "Accept: application/vnd.kafka.avro.v2+json" \
     https://<REST Proxy
FQDN>:8083/consumers/secure_topic_group/instances/my_consumer_instance/records

curl --tlsv1.0 --cacert <Location of CA Cert> -E secure_user.pem \
    -X DELETE -H "Content-Type: application/vnd.kafka.v2+json" \
    https://<REST Proxy
FQDN>:8083/consumers/secure_topic_group/instances/my_consumer_instance
```
Read operation will produce output like below:

```
[{"topic":"test_secure_topic","key":null,"value":{"name":"REST
API","salary":1000},"partition":0,"offset":0}]
```
When we try the same operation with ***insecure_user*** account that doesn't have privileges for this topic (you'll have to pass right PEM file in the curl call), Read operation will produce output like below:
```
{"error_code":50002,"message":"Kafka error: Not authorized to access group:
secure_topic_group"}
```
Configure Connector

Connector is just another producer/consumer to a topic. Connectors connect to the cluster as ANONYMOUS, so we need to provide producer/consumer access to that role. We configure connectors to brokers using PLAINTEXT, since there's no need to do SSL for this.

```
name=hdfs_sink_test_secure_topic
bootstrap.servers=PLAINTEXT://<Broker FQDN>:9092
```
Source connectors must be given WRITE permission to any topics that they need to write to. Similarly, sink connectors need READ permission to any topics they will read from. They also need Group READ permission since sink tasks depend on consumer groups internally. Connect

defines the consumer group.id conventionally for each sink connector as connect-{name} where {name} is substituted by the name of the connector.
As an example, when we are running HDFS connector, we need following ACLs to the topic (for a connector that's named as above).

```
Current ACLs for resource `Group:connect-hdfs_sink_test_secure_topic`:
  User:ANONYMOUS has Allow permission for operations: Read from hosts: <Connector Host>

Current ACLs for resource `Topic:test_secure_topic`:
  User:ANONYMOUS has Allow permission for operations: Read from hosts:<Connector Host>
```