Lab 8.1: Kafka SSL

Welcome to the session 8 lab 1. The work for this lab is done in `~/kafka-training/lab8.1`. In this lab, you are going to setup Kafka SSL support.

Please refer to the Kafka course notes for any updates or changes to this lab.

Find the latest version of this lab here.

Lab 8: Kafka Security

Kafka provides *authentication* via ***SASL*** (Simple Authentication and Security Layer) and ***SSL*** (Secure Sockets Layer) for encryption. Kafka also provides Authorization (pluggable) and encryption via SSL/TLS (in transit).

Authentication

Kafka Broker supports *authentication* in producers and consumers, brokers, tools with methods SSL and SASL.

Kafka supports the following SASL mechanisms:

SASL/GSSAPI Kerberos (GSSAPI - Generic Security Services Application Program Interface - offers a data-security layer)
SASL/PLAIN (Simple cleartext password mechanism)
SASL/SCRAM-SHA-256 (SCRAM - Salted Challenge Response Authentication Mechanism - modern challenge-response scheme based mechanism with channel binding support)
SASL/SCRAM-SHA-512 (SCRAM - Salted Challenge Response Authentication Mechanism - modern challenge-response scheme based mechanism with channel binding support)

You also can use ZooKeeper Authentication (brokers to ZooKeeper).

Java SSL performance is not always that great. There is a Kafka performance degradation when SSL is enabled.

Encryption and Authorization

Kafka provides encryption of data transferred (using SSL) via brokers, producers, and consumers.

The *authorization* provided in Kafka occurs in read/write operations, you can also use integration with 3rd party providers for pluggable authorization.

Kafka and SSL

SSL/TLS Overhead

SSL/TLS have some overhead, especially true in JVM world which is not as performant for handling SSL/TLS unless you are using Netty/OpenSSL integration.

Understanding SSL/TLS support for Kafka is important for developers, DevOps and DBAs.

If possible, use no encryption for cluster communication, and deploy your Kafka Cluster Broker nodes in a private subnet, and limit access to this subnet to client transport. Also if possible avoid using TLS/SSL on client transport and do client operations from your app tier, which is located in a non-public subnet.

However, that is not always possible to avoid TLS/SSL. Regulations and commons sense:

- U.S. Health Insurance Portability and Accountability Act (HIPAA),
- Germany's Federal Data Protection Act,
- The Payment Card Industry Data Security Standard (PCI DSS)
- U.S. Sarbanes-Oxley Act of 2002.
- Or you might work for a bank or other financial institution.
- Or it just might be a corporate policy to encrypt such transports.

Kafka has essential security features: authentication, role-based authorization, transport encryption, but is missing data at rest encryption, up to you to encrypt records via OS file systems or use AWS KMS to encrypt EBS volume

Encrypting client transports

Data that travels over the client transport across a network could be accessed by someone you don't want accessing said data with tools like wire shark. If data includes private information, SSN number, credentials (password, username), credit card numbers or account numbers, then we want to make that data unintelligible (encrypted) to any and all 3rd parties. Encryption is especially important if we don't control the

network. You can also use TLS to make sure the data has not been tampered with while traveling the network. The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols are designed to provide these features (SSL is the old name for what became TLS, but many people still refer to TLS as SSL). Kafka is written in Java. Java defines the JSSE framework which in turn uses the Java Cryptography Architecture (JCA). JSSE uses cryptographic service providers from JCA.

If any of the above is new to you, please take a few minutes to read through the TLS/SSL Java guide and Java Cryptography Architecture (JCA) Reference Guide

Avoid Man in the middle attacks

Set the Kafka Broker Config setting `ssl.endpoint.identification.algorithm=HTTPS` The default `ssl.endpoint.identification.algorithm` is `null` which is not a secure default. HTTPS better option as this forces producers and consumers to verify server's fully qualified domain name (FQDN) against Common Name (CN) or Subject Alternative Name (SAN).

Certificate Authority

Each Kafka Broker in cluster has a public-private key pair, and a certificate to identify the broker.
To prevent forged certificates, you have to sign the certificates. Certificate authority (CA) signs the certificate and signed certificates are hard to forge. If you trust the CA, clients (producers, consumers, other brokers) can trust the authenticity of Kafka brokers.

Steps to use SSL for Consumers and Producers

Generate SSL key and certificate for each Kafka broker

ACTION - EDIT `bin/create-ssl-key-keystore.sh` and follow instructions

```
#!/usr/bin/env bash
set -e

# Common Variables for SSL Key Gen
CERT_OUTPUT_PATH="$PWD/resources/opt/kafka/conf/certs"
KEY_STORE="$CERT_OUTPUT_PATH/kafka.keystore"
TRUST_STORE="$CERT_OUTPUT_PATH/kafka.truststore"
PASSWORD=kafka123
KEY_KEY_PASS="$PASSWORD"
KEY_STORE_PASS="$PASSWORD"
TRUST_KEY_PASS="$PASSWORD"
TRUST_STORE_PASS="$PASSWORD"
CLUSTER_NAME=kafka
CERT_AUTH_FILE="$CERT_OUTPUT_PATH/ca-cert"
CLUSTER_CERT_FILE="$CERT_OUTPUT_PATH/${CLUSTER_NAME}-cert"
D_NAME="CN=CloudDurable Image $CLUSTER_NAME cluster, OU=Cloudurable, O=Cloudurable"
D_NAME="${D_NAME}, L=San Francisco, ST=CA, C=USA, DC=cloudurable, DC=com"
DAYS_VALID=365

mkdir -p "$CERT_OUTPUT_PATH"

echo "Create the cluster key for cluster communication."
keytool -genkey -keyalg RSA -alias "${CLUSTER_NAME}_cluster" \
-keystore "$KEY_STORE" -storepass "$KEY_STORE_PASS" \
-keypass "$KEY_KEY_PASS" -dname  "$D_NAME" -validity "$DAYS_VALID"

echo "Create the Certificate Authority (CA) file to sign keys."
openssl req -new -x509 -keyout ca-key -out "$CERT_AUTH_FILE" \
-days "$DAYS_VALID" \
-passin pass:"$PASSWORD" -passout pass:"$PASSWORD" \
-subj "/C=US/ST=CA/L=San Francisco/O=Engineering/CN=cloudurable.com"

echo "Import the Certificate Authority file into the trust store."
keytool -keystore "$TRUST_STORE" -alias CARoot \
-import -file "$CERT_AUTH_FILE" \
-storepass "$TRUST_STORE_PASS" -keypass "$TRUST_KEY_PASS" \
-noprompt

echo "Export the cluster certificate from the key store."
```

```
keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \
-certreq -file "$CLUSTER_CERT_FILE" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt

echo "Sign the cluster certificate with the CA."
openssl x509 -req -CA "$CERT_AUTH_FILE" -CAkey ca-key \
-in "$CLUSTER_CERT_FILE" -out "${CLUSTER_CERT_FILE}-signed" \
-days "$DAYS_VALID" -CAcreateserial -passin pass:"$PASSWORD"

echo "Import the Certificate Authority (CA) file into the key store."
keytool -keystore "$KEY_STORE" -alias CARoot -import -file "$CERT_AUTH_FILE" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt

echo "Import the Signed Cluster Certificate into the key store."
keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \
-import -file "${CLUSTER_CERT_FILE}-signed" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt
```

Generate cluster certificate into a keystore use keytool

```
echo "Create the cluster key for cluster communication."
keytool -genkey -keyalg RSA -alias "${CLUSTER_NAME}_cluster" \
-keystore "$KEY_STORE" -storepass "$KEY_STORE_PASS" \
-keypass "$KEY_KEY_PASS" -dname  "$D_NAME" -validity "$DAYS_VALID"
```

keytool ships with Java used for SSL/TLS

`-genkey (generate key)`

`-keystore (location of keystore to add the key)`

`-keyalg RSA (use the RSA algorithm for the key)`

`-alias (alias of the key we use this later to extract and sign key)`

`-storepass (password for the keystore)`

`-keypass (password for key)`

`-validity (how many days is this key valid)`

Generate or use CA (Certificate Authority) use openssl

```
echo "Create the Certificate Authority (CA) file to sign keys."
openssl req -new -x509 -keyout ca-key -out "$CERT_AUTH_FILE" \
-days "$DAYS_VALID" \
-passin pass:"$PASSWORD" -passout pass:"$PASSWORD" \
-subj "/C=US/ST=CA/L=San Francisco/O=Engineering/CN=cloudurable.com"
```

X.509 certificate contains a public key and an identity (is hostname, or an organization, or an individual and is either signed by a certificate authority or self-signed)

`-req -new -x509 (create contains a public key and an identity)`

`-days (how many days is this certificate valid)`

`-passin pass / -passout pass (passwords to access the certificate)`

`-subj (pass identity information about the certificate)`

Import CA into Kafka's truststore use keytool

```
echo "Import the Certificate Authority file into the trust store."
keytool -keystore "$TRUST_STORE" -alias CARoot \
-import -file "$CERT_AUTH_FILE" \
-storepass "$TRUST_STORE_PASS" -keypass "$TRUST_KEY_PASS" \
-noprompt
```

`-import -file (is CA file we generated in the last step)`

`-keystore (location of trust keystore file)`

`–storepass (password for the keystore)`

`–keypass (password for key)`

Export and Sign cluster certificate with CA use openssl

```
echo "Export the cluster certificate from the key store."
keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \
-certreq -file "$CLUSTER_CERT_FILE" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt

echo "Sign the cluster certificate with the CA."
openssl x509 -req -CA "$CERT_AUTH_FILE" -CAkey ca-key \
-in "$CLUSTER_CERT_FILE" -out "${CLUSTER_CERT_FILE}-signed" \
-days "$DAYS_VALID" -CAcreateserial -passin pass:"$PASSWORD"
```

Export the CLUSTER_CERT_FILE from the first step from the keystore, then sign the CLUSTER_CERT_FILE with the CA

Import CA and signed cluster certificate into Kafka's keystore use keytool

```
echo "Import the Certificate Authority (CA) file into the key store."
keytool -keystore "$KEY_STORE" -alias CARoot -import -file "$CERT_AUTH_FILE" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt

echo "Import the Signed Cluster Certificate into the key store."
keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \
-import -file "${CLUSTER_CERT_FILE}-signed" \
-storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt
```

Import the CA file into keystore, it was already imported into the truststore. Import the signed version of the cluster certificate into the keystore. This was the file we create in the last step.

Run bin/create-ssl-key-keystore.sh and copy files to /opt/kafka

Running create-ssl-key-keystore.sh

You will want to run `create-ssl-key-keystore.sh` and then copy and/or move files so that each Broker, Producer or Consumer has access to `/opt/kafka/conf/certs/` .

Running create-ssl-key-keystore.sh

```
~/kafka-training/lab8.1/solution

$ bin/create-ssl-key-keystore.sh
Create the cluster key for cluster communication.
Create the Certificate Authority (CA) file to sign keys.
Generating a 1024 bit RSA private key
writing new private key to 'ca-key'
...
Certificate was added to keystore
Import the Signed Cluster Certificate into the key store.
Certificate reply was installed in keystore
```

ACTION - RUN `bin/create-ssl-key-keystore.sh`

Copying cert files to /opt/kafka/

```
$ sudo cp -R resources/opt/kafka/ /opt/
```

ACTION - COPY output of `bin/create-ssl-key-keystore.sh` to `/opt/kafka/`

ACTION - See files generated `ls /opt/kafka/conf/certs/` (5 files)

ca-cert - Certificate Authority file - don't ship this around
kafka-cert - Kafka Certification File - public key and private key, don't ship this around
kafka-cert-signed - Kafka Certification File signed with CA - don't ship this around
kafka.keystore - needed on all clients and servers
kafka.truststore - needed on all clients and servers

Configuring Kafka Servers

You will need to configure the listener's protocols for each server. In this example, we are using three servers. You will want to configure Kafka, so it is available on SSL and plaintext. The plaintext important for tools, and you could block Plaintext at firewalls or using routes.

You will need to pass in the truststore and keystore locations and passwords.

The setting `security.inter.broker.protocol=SSL` may not be needed if Kafka a cluster runs in a single private subnet. Remember that SSL makes it Kafka run slower, and adds extra CPU load on Kafka Brokers.

ACTION - EDIT `config/server-0.properties` and follow instructions

```
broker.id=0
listeners=PLAINTEXT://localhost:9092,SSL://localhost:10092
ssl.keystore.location=/opt/kafka/conf/certs/kafka.keystore
ssl.keystore.password=kafka123
ssl.key.password=kafka123
ssl.truststore.location=/opt/kafka/conf/certs/kafka.truststore
ssl.truststore.password=kafka123
ssl.client.auth=required
security.inter.broker.protocol=SSL

log.dirs=./logs/kafka-0
default.replication.factor=3
num.partitions=8
min.insync.replicas=2
auto.create.topics.enable=false
broker.rack=us-west2-a
queued.max.requests=1000
auto.leader.rebalance.enable=true

zookeeper.connect=localhost:2181
delete.topic.enable=true
compression.type=producer
message.max.bytes=65536
replica.lag.time.max.ms=5000
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connection.timeout.ms=6000
```

ACTION - EDIT `config/server-1.properties` and follow instructions

```
broker.id=1
listeners=PLAINTEXT://localhost:9093,SSL://localhost:10093
ssl.keystore.location=/opt/kafka/conf/certs/kafka.keystore
ssl.keystore.password=kafka123
ssl.key.password=kafka123
ssl.truststore.location=/opt/kafka/conf/certs/kafka.truststore
ssl.truststore.password=kafka123
ssl.client.auth=required
security.inter.broker.protocol=SSL

log.dirs=./logs/kafka-1
default.replication.factor=3
num.partitions=8
```

```
min.insync.replicas=2
auto.create.topics.enable=false
broker.rack=us-west2-a
queued.max.requests=1000
auto.leader.rebalance.enable=true

zookeeper.connect=localhost:2181
delete.topic.enable=true
compression.type=producer
message.max.bytes=65536
replica.lag.time.max.ms=5000
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connection.timeout.ms=6000
```

ACTION - EDIT `config/server-2.properties` and follow instructions

```
broker.id=2
listeners=PLAINTEXT://localhost:9094,SSL://localhost:10094
ssl.keystore.location=/opt/kafka/conf/certs/kafka.keystore
ssl.keystore.password=kafka123
ssl.key.password=kafka123
ssl.truststore.location=/opt/kafka/conf/certs/kafka.truststore
ssl.truststore.password=kafka123
ssl.client.auth=required
security.inter.broker.protocol=SSL

log.dirs=./logs/kafka-2
default.replication.factor=3
num.partitions=8
min.insync.replicas=2
auto.create.topics.enable=false
broker.rack=us-west2-a
queued.max.requests=1000
auto.leader.rebalance.enable=true

zookeeper.connect=localhost:2181
delete.topic.enable=true
compression.type=producer
message.max.bytes=65536
replica.lag.time.max.ms=5000
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
num.recovery.threads.per.data.dir=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connection.timeout.ms=6000
```

Configure Kafka Consumer

You will need to pass in truststore and keystore locations and passwords to the consumer.

ACTION - EDIT

`src/main/java/com/cloudurable/kafka/consumer/ConsumerUtil.java` and follow instructions in file.

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

import static java.util.concurrent.Executors.newFixedThreadPool;

public class ConsumerUtil {

    public static final String BROKERS = "localhost:10092,localhost:10093,localhost:10094";

    private static Consumer<String, StockPrice> createConsumer(
            final String bootstrapServers, final String clientId ) {

        final Properties props = new Properties();

        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                BROKERS);

        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
        props.put("ssl.truststore.location", "/opt/kafka/conf/certs/kafka.truststore");
        props.put("ssl.truststore.password", "kafka123");
        props.put("ssl.keystore.location", "/opt/kafka/conf/certs/kafka.keystore");
        props.put("ssl.keystore.password", "kafka123");

        //Turn off auto commit - "enable.auto.commit".
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, clientId);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                "StockPriceConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        //Custom Deserializer
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);

        // Create the consumer using props.
        return new KafkaConsumer<>(props);
    }
...
}
```

Configure Kafka Producer

You will need to pass in truststore and keystore locations and passwords to the producer.

ACTION - EDIT `src/main/java/com/cloudurable/kafka/producer/support/ProducerUtils.java` and follow instructions in file.

```
package com.cloudurable.kafka.producer.support;

import com.cloudurable.kafka.model.StockPrice;
import io.advantageous.boon.core.Lists;
import org.apache.kafka.clients.CommonClientConfigs;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class StockPriceProducerUtils {

    private static Producer<String, StockPrice> createProducer() {
        final Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                "localhost:10092,localhost:10093");

        props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
        props.put("ssl.truststore.location", "/opt/kafka/conf/certs/kafka.truststore");
        props.put("ssl.truststore.password", "kafka123");
        props.put("ssl.keystore.location", "/opt/kafka/conf/certs/kafka.keystore");
        props.put("ssl.keystore.password", "kafka123");

        props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceProducerUtils");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                StockPriceSerializer.class.getName());
        props.put(ProducerConfig.LINGER_MS_CONFIG, 100);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");

        return new KafkaProducer<>(props);
    }
...
}
```

Run the lab

ACTION - RUN ZooKeeper and three Kafka Brokers (scripts are under bin for ZooKeeper and Kafka Brokers).

ACTION - RUN ConsumerBlueMain from the IDE

ACTION - RUN StockPriceProducer from the IDE

Expected results

You should be able to send records from the producer to the broker and read records from the consumer to the broker using SSL.

Kafka Tutorial

This comprehensive *Kafka tutorial* covers Kafka architecture and design. The *Kafka tutorial* has example Java Kafka producers and Kafka consumers. The *Kafka tutorial* also covers Avro and Schema Registry.

Complete Kafka Tutorial: Architecture, Design, DevOps and Java Examples.

- Kafka Tutorial Part 1: What is Kafka?
- Kafka Tutorial Part 2: Kafka Architecture
- Kafka Tutorial Part 3: Kafka Topic Architecture
- Kafka Tutorial Part 4: Kafka Consumer Architecture
- Kafka Tutorial Part 5: Kafka Producer Architecture
- Kafka Tutorial Part 6: Using Kafka from the command line
- Kafka Tutorial Part 7: Kafka Broker Failover and Consumer Failover
- Kafka Tutorial Part 8: Kafka Ecosystem
- Kafka Tutorial Part 9: Kafka Low-Level Design
- Kafka Tutorial Part 10: Kafka Log Compaction Architecture
- Kafka Tutorial Part 11: Writing a Kafka Producer example in Java
- Kafka Tutorial Part 12: Writing a Kafka Consumer example in Java
- Kafka Tutorial Part 13: Writing Advanced Kafka Producer Java examples
- Kafka Tutorial 14: Writing Advanced Kafka Consumer Java examples
- Kafka Tutorial Part 15: Kafka and Avro
- Kafka Tutorial Part 16: Kafka and Schema Registry
- Kafka Tutorial
