Encryption and Authentication using SSL¶

Apache Kafka allows clients and brokers to communicate over SSL using a dedicated port, although this is not enabled by default.

Overview¶

Each broker and logical client will need its own key and certificate. The key is used for encryption, and the certificate is used for identification. Each broker and logical client can also be configured with a truststore, which is used to determine which certificates (broker or logical client identities) to trust (authenticate).

The truststore can be configured in many ways. To understand the truststore, consider the following two examples: (1) the truststore contains one or many certificates; or (2) it contains a certificate authority (CA). In (1), with a list of certificates, the broker or logical client will trust any certificate listed in the truststore. In (2), with a CA, the broker or logical client will trust any certificate that was signed by the CA in the truststore.

Using the CA (2) is more convenient, because adding a new broker or client doesn't require a change to the truststore. However, Kafka does not conveniently support blocking authentication for individual brokers or clients that were previously trusted via this mechanism (certificate revocation is typically done via Certificate Revocation Lists or the Online Certificate Status Protocol), so one would have to rely on authorization to block access. In (1), blocking authentication would be achieved by removing the broker or client's certificate from the truststore.

The CA case (2) is outlined in this diagram.

Generate SSL key and certificate for each Kafka broker¶

The first step of deploying SSL is to generate the key and the certificate for each machine in the cluster. You can use Java's `keytool` utility to accomplish this task. We will generate the key into a temporary keystore initially so that we can export and sign it later with CA.

```
keytool -keystore kafka.server.keystore.jks -alias localhost -validity {validity} -genkey
```

You need to specify two parameters in the above command:

keystore: the keystore file that stores the certificate. The keystore file contains the private key of the certificate; therefore, it needs to be kept safely.
validity: the valid time of the certificate in days.
Ensure that common name (CN) matches exactly with 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.

Creating your own CA¶

After the first step, 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 likes a government that issues passports — the government stamps (signs) each passport so that the passport becomes difficult to forge. Other governments verify the stamps 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.

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

The generated CA is simply a public-private key pair and certificate, and it is intended to sign other certificates.

The next step is to add the generated CA to the **clients' truststore** so that the clients can trust this CA:

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

Note: If you configure the Kafka brokers to require client authentication by setting `ssl.client.auth` to `requested` or `required` on the broker config then you must also provide a truststore for the Kafka brokers and it should have all the CA certificates that clients keys were signed by.

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

In contrast to the keystore, which stores each machine's own identity, the truststore of a client stores all the certificates that the client 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.

Signing the certificate¶

The next step is to sign all certificates in the keystore with the CA we generated. First, you need to export the certificate from the keystore:

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

Then sign it with the CA:

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

Finally, you need to import both the certificate of the CA and the signed certificate into the keystore:

```
keytool -keystore kafka.server.keystore.jks -alias CARoot -import -file ca-cert
keytool -keystore kafka.server.keystore.jks -alias localhost -import -file cert-signed
```

The definitions of the parameters are the following:

keystore: the location of the keystore
ca-cert: the certificate of the CA
ca-key: the private key of the CA
ca-password: the passphrase of the CA
cert-file: the exported, unsigned certificate of the server
cert-signed: the signed certificate of the server
confluent-platform-security-tools.git has a script that can generate truststores and keystores.

Configuring Kafka Brokers¶

Kafka Brokers support listening for connections on multiple ports. We need to configure `listeners` and optionally `advertised.listeners` in `server.properties`, each of which contains one or more comma-separated values.

If SSL is not enabled for inter-broker communication (see below for how to enable it), both `PLAINTEXT` and `SSL` ports are necessary. The `PLAINTEXT` port is also necessary if you intend to use any client that does not support SSL.

```
listeners=PLAINTEXT://host.name:port,SSL://host.name:port
# The following is only needed if the value is different from ``listeners``, but it should contain
# the same security protocols as ``listeners``
advertised.listeners=PLAINTEXT://host.name:port,SSL://host.name:port
```

Note that `advertised.host.name` and `advertised.port` configure a single `PLAINTEXT` port and are *incompatible* with secure protocols. Please use `advertised.listeners` instead.

The following SSL configs are needed on the broker side:

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

Since we are storing passwords in the broker config, it is important to restrict access via file system permissions.

Optional settings that are worth considering:

ssl.client.auth = none (“required” => client authentication is required, “requested” => client authentication is requested and client without certs can still connect. The usage of “requested” is discouraged as it provides a false sense of security and misconfigured clients will still connect successfully.)
ssl.cipher.suites = A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. (Default is an empty list)
ssl.enabled.protocols = TLSv1.2,TLSv1.1,TLSv1 (list out the SSL protocols that you are going to accept from clients. Do note that SSL is deprecated in favor of TLS and using SSL in production is not recommended)
ssl.keystore.type = JKS
ssl.truststore.type = JKS
If you want to enable SSL for inter-broker communication, add the following to the broker properties file (it defaults to `PLAINTEXT`):

```
security.inter.broker.protocol = SSL
```

Due to import regulations in some countries, the Oracle implementation limits the strength of cryptographic algorithms available by default. If stronger algorithms are needed (for example, AES with 256-bit keys), the JCE Unlimited Strength Jurisdiction Policy Files must be obtained and installed in the JDK/JRE. See the JCA Providers Documentation for more information.

Once you start the broker you should be able to see in the `server.log`:

```
with addresses: PLAINTEXT -> EndPoint(192.168.64.1,9092,PLAINTEXT),SSL -> EndPoint(192.168.64.1,9093,SSL)
```

To verify if the server’s keystore and truststore are setup correctly you can run the following command:

```
openssl s_client -debug -connect localhost:9093 -tls1
```

Note: TLSv1 should be listed under `ssl.enabled.protocols`.

In the output of this command you should see the server's certificate:

```
-----BEGIN CERTIFICATE-----
{variable sized random bytes}
-----END CERTIFICATE-----
subject=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=Joe Smith
issuer=/C=US/ST=CA/L=Santa Clara/O=org/OU=org/CN=kafka/emailAddress=test@test.com
```

If the certificate does not show up or if there are any other error messages then your keystore is not setup correctly.

Configuring Kafka Clients¶

SSL is supported only for the new Kafka Producer and Consumer (Kafka versions 0.9.0 and higher), the older APIs are not supported. The configs for SSL will be the same for both the producer and consumer.

If client authentication is not required by the broker, the following is a minimal configuration example:

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

If client authentication is required, then a keystore must be created for each client, and the brokers' truststores must trust the certificate in the client's keystore. This may be done using commands that are similar to what we used for the broker keystore. The script in confluent-platform-security-tools.git can also generate client keystores that the broker truststore will trust.

And the following must also be configured:

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

Since we are storing passwords in the client config, it is important to restrict access via file system permissions.

Other configuration settings that may also be needed depending on our requirements and the broker configuration:

ssl.provider (Optional). The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.
ssl.cipher.suites (Optional). A cipher suite is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.
ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1. It should list at least one of the protocols configured on the broker side
ssl.truststore.type=JKS
ssl.keystore.type=JKS
Examples using console-producer and console-consumer:

```
bin/kafka-console-producer --broker-list localhost:9093 --topic test --producer.config client-ssl.properties
bin/kafka-console-consumer --bootstrap-server localhost:9093 --topic test --new-consumer --consumer.config
client-ssl.properties --from-beginning
```

Enabling SSL Logging¶

In Kafka 0.9, you can enable SSL debug logging at the JVM level by starting the Kafka broker and/or clients with the `javax.net.debug` system property. For example:

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

You can find more details on this in the Oracle documentation on debugging SSL/TLS connections.
