Cloudera Streams Messaging Operator for Kubernetes 1.5.0

Kafka Security

Date published: 2024-06-11
Date modified: 2025-10-30

https://docs.cloudera.com/

Legal Notice

Contents

Channel encryption (TLS)

Learn how to configure channel encryption (TLS) for Kafka clusters. You have multiple options for configuring TLS. You can use auto-generated and self-signed certificates, use a custom external certificates, or use an external certificate authority (CA) certificate, but have broker certificates automatically generated by the Strimzi Cluster Operator.

Using auto-generated self-signed certificates

When the tls property is set to true on one of the Kafka listeners, the Strimzi Cluster Operator creates self-signed certificates. In this case, the Strimzi Cluster Operator automatically sets up and renews certificates.

You can add a TLS-enabled listener by configuring spec.kafka.listeners in your `Kafka` resource.

```
#...
kind: Kafka
spec:
  kafka:
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
```

Related Information

Secrets generated by the operators | Strimzi

Using external certificates

It is possible to pass externally issued certificates as secrets to the Strimzi Cluster Operator, however there's no way to request new certificates automatically, they have to be prepared ahead of time.

The spec.kafka.listeners[n].configuration.brokerCertChainAndKey.secretName property specifies to the secret containing the broker certificate.

```
#...
kind: Kafka
spec:
  clusterCa:
    generateCertificateAuthority: false
  clientsCa:
    generateCertificateAuthority: false
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        configuration:
          brokerCertChainAndKey:
            secretName: cluster-cert
            certificate: tls.crt
```

```
            key: tls.key
```

When using externally created certificates, the spec.clusterCa.generateCertificateAuthority and spec.clientsCa.generateCertificateAuthority properties have to be set to false to avoid generating self-signed CAs.

The Strimzi Cluster Operator expects the CA certificates to be in specific Kubernetes secrets and specific structure. For a cluster with name my-cluster, the following commands can be used to create those secrets for the Strimzi Cluster Operator when the CA is provided externally.

```
kubectl create secret generic my-cluster-cluster-ca-cert -n kafka \
  --from-file="ca.p12" \
  --from-file="ca.crt" \
  --from-file="ca.password"
```

```
kubectl create secret generic my-cluster-clients-ca-cert -n kafka \
  --from-file="ca.p12" \
  --from-file="ca.crt" \
  --from-file="ca.password"
```

```
kubectl create secret generic my-cluster-cluster-ca -n kafka \
  --from-file="ca.key"
```

```
kubectl create secret generic my-cluster-clients-ca -n kafka \
  --from-file="ca.key"
```

```
kubectl label secret my-cluster-cluster-ca-cert -n kafka \
  "strimzi.io/kind=Kafka" "strimzi.io/cluster=my-cluster
```

```
kubectl label secret my-cluster-clients-ca-cert -n kafka \
  "strimzi.io/kind=Kafka" "strimzi.io/cluster=my-cluster"
```

```
kubectl label secret my-cluster-cluster-ca -n kafka \
  "strimzi.io/kind=Kafka" "strimzi.io/cluster=my-cluster"
```

```
kubectl label secret my-cluster-clients-ca -n kafka \
  "strimzi.io/kind=Kafka" "strimzi.io/cluster=my-cluster"
```

It is also possible to only create the CA and let the Strimzi Cluster Operator use that to provision certificates. In that case skip the broker and client certificate creation and do not specify the brokerCertChainAndKey” field on the listeners.

Authentication

Learn how to configure Authentication for Kafka. Multiple authentication mechanisms are supported.

Configuring mTLS authentication

Learn how to enable mTLS authentication on broker listeners with or without an external certificate.

To enable mTLS authentication on any of the broker listeners, set the spec.kafka.listeners[n].authentication.type property to tls.

```
#...
```

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
```

To use mTLS authentication using an external certificate, you need to set the type field in the `KafkaUser` resource to tls-external. A secret and credentials are not created for the user:

```
#...
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
```

Related Information

Installing your own CA certificates | Strimzi

Configuring OAuth authentication

Learn how to configure OAuth authentication for Kafka. OAuth is configured by creating a Kubernetes Secret for the Oauth certificate, configuring an OAuth in your Kafka resource, and adding your OAuth server to the allowed list of OAuth URLs in the KafkaNodePool resource.

Before you begin

- An OAuth server is available. The server is accessible from the Kubernetes environment.
- Both Kafka brokers and clients are able to access the OAuth server.
- The TLS certificates of the OAuth server must be available in PEM format.
- The following attributes of the OAuth environment must be determined:
  - userNameClaim – the claim name which contains the client ID. Typically this is asub, but its OAuth provider dependent.
  - validIssuerUri – it must point to the URL that clients can use to connect to the OAuth server. The value can be obtained from the well-known endpoint of the OAuth server or a JWT token.

To set up OAuth, create a Kubernetes `Secret` for the OAuth certificate. The Strimzi Cluster Operator will mount and use the `Secret` when configuring the listener.

```
kubectl create secret generic [***SECRET NAME***] \
  --namespace [***NAMESPACE***] \
  --from-file=[***OAUTH SERVER CERT.PEM FILE***]
```

The following snippet configures a Kafka cluster with an OAuth authenticated listener on port 9093. Notice that the authentication section in the listener config contains all OAuth specific settings.

```
#...
kind: Kafka
```

```
spec:
  kafka:
    listeners:
      - name: oauth
        port: 9093
        type: internal
        tls: false
        authentication:
          type: oauth
          jwksEndpointUri: [***JWKS ENDPOINT URI***]
          tlsTrustedCertificates:
            - secretName: [***SECRET NAME***]
              certificate: [***OAUTH SERVER CERT.PEM FILE***]
          userNameClaim: [***USER NAME CLAIM***]
          validIssuerUri: [***ISSUER URI***]
          maxSecondsWithoutReauthentication: 3600
---
#...
kind: KafkaNodePool
spec:
  jvmOptions:
    javaSystemProperties:
      - name: org.apache.kafka.sasl.oauthbearer.allowed.urls
        value: [***OAUTH SERVER URL 1***],[***OAUTH SERVER URL 2***]
```

Note: If maxSecondsWithoutReauthentication is not set, authenticated sessions remain open even after token expiry.

Related Information

Using OAuth 2.0 token-based authentication | Strimzi

Configuring LDAP authentication

Learn how to configure LDAP authentication for Kafka. LDAP is configured by creating a Kubernetes Secret that stores your LDAP truststore, configuring your Kafka resource to include a custom type listener for LDAP, and adding your LDAP server to the allowed list of LDAP URLs in the KafkaNodePool resource.

Before you begin

- An LDAP server is available. The server is accessible from the Kafka Kubernetes environment.
- You have access to a truststore that contains the CA certificate of the LDAP server.

To set up LDAP, create a `Secret` from the truststore. The Strimzi Cluster Operator will be able to mount the `Secret` for the brokers.

```
kubectl create secret generic [***SECRET NAME***] \
  --namespace [***NAMESPACE***] \
  --from-file=[***LDAP SERVER TRUSTSTORE FILE***]
```

Afterward, configure your `Kafka` and `KafkaNodePool` resources to include the LDAP configuration.

```
#...
kind: Kafka
spec:
  kafka:
    listeners:
      - name: ldap
        port: 9094
        type: internal
        tls: false
```

```
        authentication:
          type: custom
          sasl: true
          listenerConfig:
            plain.sasl.server.callback.handler.class: org.apache.kafka.comm
on.security.ldap.internals.LdapPlainServerCallbackHandler
            plain.sasl.jaas.config: 'org.apache.kafka.common.secur
ity.plain.PlainLoginModule required ssl.truststore.password="[***SSL
 TRUSTSTORE PASSWORD***]" ssl.truststore.location="/mnt/[***SECRET
 NAME***]/[***LDAP SERVER TRUSTSTORE FILE***]" ldap_url="ldaps://[***LDAP
 SERVER URL***]:[***PORT***]" user_dn_template="cn={0},ou=users,dc=ldap-dc,d
c=ldap";'
            sasl.enabled.mechanisms: PLAIN
    template:
      pod:
        volumes:
          - name: [***VOLUME NAME***]
            secret:
              secretName: [***SECRET NAME***]
      kafkaContainer:
        volumeMounts:
          - name: [***VOLUME NAME***]
            mountPath: /mnt/[***SECRET NAME***]
---
#...
kind: KafkaNodePool
spec:
  jvmOptions:
    javaSystemProperties:
      - name: com.cloudera.kafka.ldap.allowed.urls
        value: [***LDAP SERVER URL 1***],[***LDAP SERVER URL 2***]
```

Apply the configuration changes to the Kafka resource and wait for the Strimzi Cluster Operator to reconcile the cluster.

Configuring SCRAM-SHA-512 authentication

Learn how to enable SCRAM-SHA-512 authentication and generate SCRAM credentials for your clients.

To enable SCRAM-SHA-512 authentication, you can specify a listener in your Kafka resource that has authenticati on.type set to scram-sha-512. Additionally, you create a KafkaUser resource to generate SCRAM credentials for your clients.

```
#...
kind: Kafka
metadata:
  name: my-cluster
  namespace: kafka
spec:
  kafka:
    listeners:
      - name: scram
        port: 9093
        type: internal
        tls: false
        authentication:
          type: scram-sha-512
```

To generate SCRAM credentials that your clients can use to access Kafka, you create a `KafkaUser` resource that has spec.authentication.type set to scram-sha-512. For example:

```
#...
kind: KafkaUser
metadata:
  name: my-user
  namespace: kafka
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
```

When the user specified by the `KafkUser` resource is created, the Strimzi User Operator creates a new secret with the same name as the `KafkaUser` resource. The secret contains the generated password (`data.password`) as well as a JAAS configuration string (`data.sasl.jaas.config`). The password and JAAS are encoded with Base64. As a result, they must be decoded when you retrieve them for use.

Using kubectl, you can extract both the password and JAAS. However, when configuring your clients, you typically want to extract the JAAS, as this is the string that you add to your client's configuration. Specifically, the JAAS string you extract is the value you set for sasl.jaas.config in your Kafka client configuration. The following command example prints the full JAAS configuration generated for a user.

```
kubectl get secret [***SECRET NAME***] \
  --namespace [***NAMESPACE***] \
  --output jsonpath='{.data.sasl\.jaas\.config}' \
| base64 -d
```

Configuring PLAIN authentication

Learn how to configure PLAIN (basic) authentication by applying a custom authentication configuration for Kafka on an exposed listener.

To set up PLAIN, create a secret that contains the jaas.conf with the username-password configuration.

```
echo -n 'org.apache.kafka.common.security.plain.PlainLoginModule required us
er_kafka="password";' > kafka-jaas.conf
```

```
kubectl create secret -n kafka generic my-kafka-secret-name --from-file=kafk
a-jaas.conf
```

Next, a `Role` and a `RoleBinding` is needed to be able to use the kafka-jaas.conf secret:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kafka-configuration-role
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["my-kafka-secret-name"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kafka-configuration-role-binding
subjects:
```

```
- kind: ServiceAccount
  name: my-cluster-kafka
  namespace: kafka
roleRef:
  kind: Role
  name: kafka-configuration-role
  apiGroup: rbac.authorization.k8s.io
```

Finally, the Kafka listener can be configured. By setting the spec.kafka.listeners[n].authentication.sasl to true, the Strimzi Cluster Operator will configure SASL protocol for the listener.

```
#...
kind: Kafka
spec:
  kafka:
    listeners:
      - name: plain
        port: 9093
        type: internal
        tls: true
        authentication:
          type: custom
          sasl: true
          listenerConfig:
            plain.sasl.server.callback.handler.class: org.apache.kafka.comm
on.security.plain.internals.PlainServerCallbackHandler
            sasl.enabled.mechanisms: PLAIN
            plain.sasl.jaas.config: ${secrets:kafka/my-kafka-secret-name:k
afka-jaas.conf}
    config:
      config.providers: secrets
      config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretConfi
gProvider
```

Related Information

Using RBAC Authorization | Kubernetes

Authorization

Learn how to configure authorization for Kafka. Multiple types of authorization are available.

Simple ACL authorization

Learn how to configure Simple ACL authorization, ACL rules, and well as super users.

Configuring simple ACL

Learn how to enable and configure simple ACL authorization for Kafka.

Simple ACL authorization is enabled by setting spec.kafka.authorization.type to simple in your `Kafka` resource. Additionally, to manage user (client) access, you create `KafkaUser` resources that have a matching authorization type configured. `KafkaUser` resources configure authorization rules for users that require access to your cluster.

The following is an example `Kafka` resource with simple ACL and mTLS authentication enabled.

```
#...
kind: Kafka
```

```
metadata:
  name: my-cluster
spec:
  kafka:
    authorization:
      type: simple
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
```

Following the configuration of the `Kafka` resource, you create `KafkaUser` resources, which define the access control rules for the users (clients) accessing Kafka. When creating a `KafkUser` resource for simple authorization, you set spec.authorization.type to simple (matching the authorization configuration of Kafka) Additionally, you define the rules for the user with the acls property. Each rule is defined as an array.

The following is a `KafkaUser` example configured for simple authorization that includes a few example rules.

```
#...
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Read
          - Describe
      - resource:
          type: topic
          name: "*"
          patternType: literal
        type: allow
        host: "*"
        operations:
          - Read
      - resource:
          type: group
          name: my-group
          patternType: prefix
        operations:
          - Read
```

Note: The `KafkaUser` resource specifies the username in the metadata.name property. The username must follow the Kubernetes rules for the metadata fields. So for example underscores (_) are not allowed. If you need to create a user with an incompatible name, disable the Strimzi User Operator and manage users directly in Kafka. In this case, limitations on naming imposed by Kubernetes do not apply.

Related Information

Object Names and IDs | Kubernetes

Configuring ACL rules

Learn how to configure ACL rules for simple ACL authorization.

ACL rules are specified in the acls property of the `KafkaUser` resource.

```
#...
kind: KafkaUser
spec:
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: "*"
          patternType: literal
        type: allow
        host: "*"
        operations:
          - Read
```

The properties you use to define an ACL rule are as follows.

resource

The resource property specifies the `Kafka` resource that the rule applies to. Simple authorization supports the following resource types, which are specified in the type property.

- topic
- group
- cluster
- transactionalId

For topic, group, and transactionalID type resources you can specify the name of the resource that the rule applies to in the name property. Resources of the cluster type do not have a name.

The name of the resource is either a literal or a prefix. This is specified in the value of the patternT ype property which can be either literal or prefix.

- Literal names (patterntype: literal) are interpreted as they are specified in name.
- Prefix names (patterntype: prefix) treat the value specified in name as a prefix. The rule is applied to all resources that have names starting with the prefix.

The name property accepts an asterisk (*) as a value. If name is set to * and patternType is literal, the rule applies to all resources.

```
#...
- resource:
  type: topic
  name: *
  patternType: literal
```

type

The type property specifies the type of the rule. This is an optional property, the rule type is set to allow by default if it is not specified.

Important: While the type property accepts both allow and deny as values, deny rules are not supported.

host

You use the hostproperty to restrict the rule to apply to a specified remote host. If set to *, the rule is applied to all hosts. This is an optional property, the default value is *.

operations

The operations property specifies a list of operations for the rule. Supported operations are Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, Describe Configs.

Some operations are not valid on some resources. See the Apache Kafka documentation for a comprehensive matrix regarding operations and their supported resources.

Related Information

AclRule schema reference | Strimzi API reference

Operations and Resources on Protocols | Apace Kafka

Configuring super users

In addition to creating users with KafkaUser resources that have specific access restrictions defined, you can choose to designate super users in your Kafka cluster. Super users have unlimited access, regardless of access restrictions.

To designate super users for a Kafka cluster, add a list of user principals to the spec.kafka.authorization.superUsers property in your `Kafka` resource.

```
#...
kind: Kafka
spec:
  kafka:
    authorization:
      type: simple
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
```

If a user uses mTLS authentication, the username is the common name from the TLS certificate subject prefixed with CN=. If you are not using the Strimzi User Operator and using your own certificates for mTLS, the username is the full certificate subject.

A full certificate subject can have the following fields.

```
CN=user,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=my_country_code
```

Omit any fields that are not present.

Apache Ranger authorization

Learn how to integrate an Apache Ranger service running in a Cloudera Base on premises cluster with an Apache Kafka cluster that is deployed using Cloudera Streams Messaging Operator for Kubernetes.

Apache Kafka clusters deployed with Cloudera Streams Messaging Operator for Kubernetes can integrate with Apache Ranger. Ranger is a framework to enable, monitor, and manage comprehensive data security. Specifically, you can use Ranger to authorize access requests made to Kafka. The Ranger service that you integrate with Kafka must run in a Cloudera Base on premises cluster.

To provide authorization for various services, Ranger uses a plugin architecture. Ranger Plugins are lightweight Java plugins developed for specific components and services that run as part of the target component's JVM process.

Ranger plugins pull policies from Ranger, evaluate incoming requests providing authorization, and also capture and push requests as audit events to different audit destinations. To provide authorization for Kafka, Ranger uses the Ranger Kafka plugin. The Ranger Kafka plugin is shipped as part of the Cloudera Runtime parcel.

Integrating your Cloudera Streams Messaging Operator for Kubernetes Kafka clusters with Ranger requires that you complete multiple configuration tasks. The following provides an overview of the process.

1. Creating a custom Kafka image that includes the Ranger Kafka plugin and ZooKeeper

   This step involves building a new Kafka image that includes the Ranger Kafka plugin and ZooKeeper. Your new image will be based on the default Kafka image shipped with Cloudera Streams Messaging Operator for Kubernetes. The Kafka Ranger plugin and ZooKeeper are extracted from a Cloudera Runtime parcel.
2. Configuring Ranger

   This step involves creating a user in the Kerberos Key Distribution Center (KDC) of the Cloudera Base on premises cluster as well as various configuration tasks that you complete in the Ranger Admin Web UI.
3. Creating Ranger plugin configuration files

   This step involves creating various configuration files required for the Ranger Kafka plugin to function. These files store settings such as how to reach Ranger, HDFS, the TLS truststore, and authentication credentials.
4. **Optional:** Configuring Ranger group authorization

   Ranger group authorization enables you to add groups to policies in Ranger. Users that are part of a group can be authorized based on the permissions of the group. User group memberships are defined in an LDAP server. This step involves setting up LDAP group mapping properties for the Ranger Kafka plugin.
5. Deploying a Ranger-integrated Kafka cluster

   This step involves deploying a new Kafka cluster using the custom image you built as well as deploying the various configuration files you created for the Ranger Kafka plugin.

Limitations

- Configuring Ranger authorization for Kafka requires that additional persistent storage is attached to each Kafka broker.

  These volumes must be unique per broker. As a result of how these volumes can be defined in Cloudera Streams Messaging Operator for Kubernetes, you must create a separate `KafkaNodePool` for each broker. With this limitation, scaling of `KafkaNodePools` will not work. This is because each `KafkaNodePool` must be limited to a single replica. If you want to scale your Kafka cluster, you must define another `KafkaNodePool` for the new broker.
- Ranger authorization does not work with KRaft combined mode.

Supported Cloudera Base on premisesversions

Integration with Ranger is only supported for specific Cloudera Base on premises versions. Supported versions of Cloudera Base on premises are as follows.

Table 1: Supported Cloudera Base on premisesversions for Ranger authorization in Cloudera Streams Messaging Operator for Kubernetes

Cloudera Base on premises cluster version	Ranger Kafka plugin version
7.3.1 (any SP or CHF)	7.3.1.0
7.1.9 (any SP or CHF)	7.1.9.1023

Prerequisites

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.

- You have a Cloudera Base on premises cluster with the following.
  - The cluster version is supported.
  - The cluster must be secure. Both TLS/SSL (channel encryption) and Kerberos (authentication) must be enabled.
  - **Optional:** If you use HDFS or Solr to store Ranger audit data, HDFS or Solr must be installed on the cluster.

    Note: The following instructions and configuration examples configure both HDFS and Solr auditing.
  - Access to a registry where you can upload a container image is required. The registry must also be accessible by your Kubernetes cluster.
  - Java 8 or higher installed on the host which is used to generate the plugin configuration files.

Related Information

Cloudera Private Cloud Base Release Notes

Creating a custom Kafka image that includes the Ranger Kafka plugin and ZooKeeper

The Ranger Kafka plugin performing the authorization in Kafka's JVM needs multiple JARs in order to be able to function correctly. As a result, you need to download a Cloudera Runtime parcel, extract the Ranger Kafka plugin and ZooKeeper, and build a custom Kafka image that contains both. The image that you create is used to deploy your Kafka cluster that integrates with Ranger.

Before you begin

Access to docker or equivalent utility that you can use to build, pull, and push images is required. The following steps use docker. Replace commands where necessary.

Procedure

1. Download a supported version of the Cloudera Runtime RHEL9 parcel.

   You can download the parcel from the Cloudera Archive.

   ```
   https://archive.cloudera.com/p/cdh7/[***VERSION***]/parcels/
   ```

   Downloading the parcel requires authentication. Use your Cloudera credentials. For more information, see *Cloudera Runtime Download Information*.
2. Extract the Ranger Kafka plugin and ZooKeeper from the downloaded parcel.

   ```
   tar xf [***PARCEL FILE***] 'CDH-*/lib/ranger-kafka-plugin/' 'CDH-*/jars' \
   && cp -RL CDH-*/lib/ranger-kafka-plugin CDH-*/jars/zookeeper-*.jar .
   ```
3. Verify that the Ranger Kafka plugin symlinks are resolved successfully.

   ```
   [[ "$(find ranger-kafka-plugin/ -type l | wc -l)" -eq "0" ]] && echo "Ra
   nger plugin symbolic links resolved successfully." || echo "Error! Symbo
   lic link resolution failed while extracting Ranger plugin."
   ```
4. Create a file named Dockerfile with the following contents.

   ```
   FROM container.repository.cloudera.com/cloudera/kafka:0.47.0.1.5.0-b123-
   kafka-4.0.1.1.5
   USER root
   COPY --chmod=644 ranger-kafka-plugin/lib/ranger-*.jar /opt/kafka/libs/
   COPY --chmod=644 ranger-kafka-plugin/lib/ranger-kafka-plugin-impl /opt/kaf
   ka/libs/ranger-kafka-plugin-impl/
   COPY --chmod=644 zookeeper-*.jar /opt/kafka/libs/
   COPY --chmod=644 zookeeper-*.jar /opt/kafka/libs/ranger-kafka-plugin-impl/
   ```

```
RUN find /opt/kafka/libs/ranger-kafka-plugin-impl/ -type d -exec chmod +x
 {} \;
RUN ln -s /mnt/ranger-kafka-plugin/conf /opt/kafka/libs/ranger-kafka-plug
in-impl/conf
USER kafka
```

5. Build and tag the image using the Dockerfile you created.

```
docker build --tag [***YOUR REGISTRY***]/[***IMAGE NAME***]:[***TAG***] .
```

6. Push the image to a registry.

```
docker image push [***YOUR REGISTRY***]/[***IMAGE NAME***]:[***TAG***]
```

Take note of the registry, image name, and image tag. These are needed later on when deploying your Kafka cluster in Kubernetes.

Results

A custom Kafka image that contains an appropriate version of the Ranger Kafka plugin is available in a registry of your choosing. You can use this image to deploy a Kafka cluster that integrates with Ranger.

Related Information

Cloudera Runtime Download Information

Configuring Ranger

To enable an external Kafka cluster to work with Ranger, you must set up various users and policies in Ranger and in your Cloudera Base on premises cluster.

Procedure

1. Create a Ranger Kafka plugin user.

   The external Ranger Kafka plugin must be able to authenticate to various Cloudera services. Therefore, a user must exists in the Kerberos Key Distribution Center (KDC) of the Cloudera Base on premises cluster that the Ranger Kafka plugin can use to authenticate itself.

   Create a Kerberos principal and generate a keytab for it if it does not yet exist. These instructions refer to primary part of this principal as *[***PLUGIN PRINCIPAL***]*. The realm of this principal is referred to as *[***KERBEROS REALM***]* where needed.
2. Ensure that the *[***PLUGIN PRINCIPAL***]* user exists in Ranger.

   Verification is required because users that you create in the KDC might not get synched immediately to Ranger.
3. Log in to the Ranger Admin Web UI.
4. Create resource-based service in Ranger for the Kafka cluster.

   Although it is possible to reuse an existing Kafka resource-based service, Cloudera recommends creating a new one. Configure the required properties as follows.

   - Service Name # The name of the resource-based service in Ranger. Arbitrary string, must be defined in the plugin config *[***KAFKA SERVICE NAME***]* in the ranger-kafka-security.xml. It also appears in the audit log, distinguishing the external Kafka from, for example, the Cloudera Private Cloud Base Kafka if installed.
   - Username # This is a legacy parameter. It is added by default to the policies, but otherwise has no effect.
   - Password # This is a legacy parameter. It has no effect.
   - ZooKeeper Connect String # This is a legacy parameter. It has no effect. Leave as default.
   - Add the following name and value pairs to Add New Configurations.
     - policy.download.auth.users: *[***PLUGIN PRINCIPAL***]*
     - tag.download.auth.users: *[***PLUGIN PRINCIPAL***]*

5. Create users for Strimzi users.

   The Strimzi operators as well as Kafka itself have users which are unknown to the Cloudera Base on premises cluster. *[***KAFKA CLUSTER NAME***]* is the name of your Kafka cluster running in Kubernetes. The name is an arbitrary string and is specified later on, when creating your `Kafka` resource.

   a. CN=cluster-operator,O=io.strimzi
   b. CN=*[***KAFKA CLUSTER NAME***]*-kafka,O=io.strimzi
   c. CN=*[***KAFKA CLUSTER NAME***]*-entity-user-operator,O=io.strimzi
   d. CN=*[***KAFKA CLUSTER NAME***]*-entity-topic-operator,O=io.strimzi
   e. CN=*[***KAFKA CLUSTER NAME***]*-kafka-exporter,O=io.strimzi
   f. CN=*[***KAFKA CLUSTER NAME***]*-cruise-control,O=io.strimzi

6. Set up Kafka policies.

   a) Grant each Strimzi user permissions to all Kafka resources and actions in all Kafka policies.

      To do this, go to Service Manager Resource Policies *[***KAFKA SERVICE NAME***]* and add the Strimzi users to the general policies with all action types granted.

   b) Set up permissions for your Kubernetes Kafka cluster's end users.

      These users might be managed outside of the Cloudera Private Cloud cluster and you might have to define them in Ranger. Add these users to the Kafka policies according to your requirements.

7. Ensure to configure HDFS policies in Ranger so that the *[***PLUGIN PRINCIPAL***]* user has read, write, and execute permission to the *[***AUDIT PATH***]* or one of its parent folders.

   The audit path is defined later as the value of xasecure.audit.destination.hdfs.dir in ranger-kafka-audit.xml, but it is possible that in Ranger you can not choose the specific folder, only its parent folder.

8. Ensure to configure Solr policies in Ranger so that the *[***PLUGIN PRINCIPAL***]* user has query and update permissions to the Ranger audit Solr collection.

   The Ranger audit Solr collection is referred to as *[***SOLR AUDIT COLLECTION***]*. It is defined later as xase cure.audit.destination.solr.collection in ranger-kafka-audit.xml.

Related Information

Configure a resource-based service: Kafka
Adding a user
Configure a resource-based policy: Kafka
Configure a resource-based policy: HDFS
Configure a resource-based policy: Solr
Get or create a Kerberos principal for each user account

Creating Ranger Kafka plugin configuration files

The Ranger Kafka plugin requires a number of configuration files which store settings such as how to reach Ranger, HDFS, the TLS truststore, authentication credentials, and so on. Some of these files can be saved from the Cloudera Base on premises cluster, some must be created manually.

Save or create the following configuration files to a subfolder named conf.

Important: File names must be an exact match with the names listed here.

ranger_truststore.jks

Save the truststore of the Ranger Admin server in JKS format.

krb5.conf

Save the Kerberos client config from a Cloudera Base on premises cluster host.

kafka_plugin.keytab

Save the keytab of the previously created *[***PLUGIN PRINCIPAL***]* user.

jaas.conf

Save the example below and edit the principal property.

```
ranger.KafkaServer {
   com.sun.security.auth.module.Krb5LoginModule required
   doNotPrompt=true
   useKeyTab=true
   storeKey=true
   keyTab="/mnt/ranger-kafka-plugin/conf_sensitive/kafka_plugi
n.keytab"
   principal="[***PLUGIN PRINCIPAL***]@[***KERBEROS REALM***]";
};
```

ranger-kafka-security.xml

Save the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- Required user defined configuration block start -->
  <property>
    <name>ranger.plugin.kafka.policy.rest.url</name>
    <value>[***RANGER REST URL***]</value>
  </property>
  <property>
    <name>ranger.plugin.kafka.service.name</name>
    <value>[***KAFKA SERVICE NAME***]</value>
  </property>
  <property>
    <name>ranger.plugin.kafka.access.cluster.name</name>
    <value>[***KAFKA ACCESS CLUSTER NAME***]</value>
  </property>
  <!-- Required user defined configuration block end -->
  <!-- Required hardcoded configuration block start -->
  <property>
    <name>ranger.plugin.kafka.policy.cache.dir</name>
    <value>/mnt/ranger-kafka-plugin/cache</value>
  </property>
  <property>
    <name>ranger.plugin.kafka.policy.source.impl</name>
    <value>org.apache.ranger.admin.client.RangerAdminRESTClient</
value>
  </property>
  <property>
    <name>ranger.plugin.kafka.policy.rest.ssl.config.file</name>
    <value>/opt/kafka/libs/ranger-kafka-plugin-impl/conf/ranger-
kafka-policymgr-ssl.xml</value>
  </property>
  <property>
    <name>ranger.plugin.kafka.disable.cache.if.servicenotfound</
name>
    <value>false</value>
  </property>
  <!-- Required hardcoded configuration block end -->
</configuration>
```

Substitute the variables in this example as follows.

- *[***RANGER REST URL***]* # Ranger Admin server base URL. A comma separated list of Ranger Admin base URLs can be provided here if Ranger Admin High Availability is configured. You can also configure a load balancer URL if a load balancer was installed with Ranger Admin High Availability.
- *[***KAFKA SERVICE NAME***]* # The Kafka service name as defined earlier in Ranger Admin UI.

- *[***KAFKA ACCESS CLUSTER NAME***]* # Cluster name to be displayed in Ranger audit logs. This is an optional config and can be removed. It has no effect other than a field in Ranger audit logs. Cloudera recommends that you set this to *[***KAFKA CLUSTER NAME***].[***NAMESPACE***]*.

ranger-kafka-policymgr-ssl.xml

Save the following example. This file only contains hardcoded properties. You do not need to make any changes.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- Required hardcoded configuration block start -->
  <property>
    <name>xasecure.policymgr.clientssl.truststore</name>
     <value>/opt/kafka/libs/ranger-kafka-plugin-impl/conf/range
r_truststore.jks</value>
  </property>
  <property>
    <name>xasecure.policymgr.clientssl.truststore.credential.fil
e</name>
    <value>jceks://file/opt/kafka/libs/ranger-kafka-plugin-impl/
conf/rangerpluginssl.jceks</value>
  </property>
  <!-- Required hardcoded configuration block end -->
</configuration>
```

ranger-kafka-audit.xml

Save the following example.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <!-- Required user defined configuration block start -->
  <property>
    <name>xasecure.audit.is.enabled</name>
    <value>true</value>
  </property>
  <!-- HDFS audit block start -->
  <property>
    <name>xasecure.audit.destination.hdfs</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.destination.hdfs.dir</name>
    <value>hdfs://[***ACTIVE HDFS NAMENODE HOST]:[***HDFS
 NAMENODE PORT***]/[***AUDIT PATH ON HDFS***]</value>
  </property>
  <!-- HDFS audit block end -->
  <!-- Solr audit block start -->
  <property>
    <name>xasecure.audit.destination.solr</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.destination.solr.zookeepers</name>
    <value>[***ZOOKEEPER CONNECTION STRING***]</value>
  </property>
  <property>
    <name>xasecure.audit.destination.solr.collection</name>
    <value>[***RANGER AUDIT SOLR COLLECTION***]</value>
  </property>
  <property>
```

```
    <name>xasecure.audit.jaas.Client.option.principal</name>
    <value>[***PLUGIN PRINCIPAL***]@[***KERBEROS REALM***]</va
lue>
  </property>
  <!-- Solr audit block end -->
  <!-- Required user defined configuration block end -->
  <!-- Required hardcoded configuration block start -->
  <!-- HDFS audit block start -->
  <property>
    <name>xasecure.audit.destination.hdfs.batch.filespool.enable
</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.destination.hdfs.batch.filespool.dir</na
me>
    <value>/mnt/ranger-kafka-plugin/audit/hdfs/spool</value>
  </property>
  <!-- HDFS audit block end -->
  <!-- Solr audit block start -->
  <property>
    <name>xasecure.audit.destination.solr.force.use.inmemory.j
aas.config</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.loginModuleName</name>
    <value>com.sun.security.auth.module.Krb5LoginModule</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.loginModuleControlFlag</nam
e>
    <value>required</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.option.useKeyTab</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.option.storeKey</name>
    <value>false</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.option.serviceName</name>
    <value>solr</value>
  </property>
  <property>
    <name>xasecure.audit.jaas.Client.option.keyTab</name>
    <value>/mnt/ranger-kafka-plugin/conf_sensitive/kafka_plugi
n.keytab</value>
  </property>
  <property>
    <name>xasecure.audit.destination.solr.batch.filespool.enable
</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.destination.solr.batch.filespool.dir</na
me>
    <value>/mnt/ranger-kafka-plugin/audit/solr/spool</value>
  </property>
  <!-- Solr audit block end -->
  <!-- Required hardcoded configuration block end -->
```

```
  <!-- Recommended but not required configuration block start -->
  <property>
    <name>xasecure.audit.provider.summary.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.destination.metrics</name>
    <value>true</value>
  </property>
  <!-- Recommended but not required configuration block end -->
</configuration>
```

Substitute the variables in this example as follows.

- *[***ACTIVE HDFS NAMENODE HOST***]* # The hostname of the active HDFS NameNode.
- *[***HDFS NAMENODE PORT***]* # The port where the NameNode runs the HDFS protocol. You can find port in Cloudera Manager HDFS Configuration NameNode Port .
- *[***AUDIT PATH ON HDFS***]* # The HDFS directory where audit logs should be stored.
- *[***ZOOKEEPER CONNECTION STRING***]* # Zookeeper connection string as defined by ZooKeeper Sessions. The chroot suffix should be the Znode of the Solr service. You can find the Znode in Cloudera Manager Solr Configuration ZooKeeper Znode . For example, host1:port1, host2:port2/solr-infra.
- *[***RANGER AUDIT SOLR COLLECTION***]* # The Solr collection where Ranger audit logs are stored. Optional, defaults to ranger_audits if not configured.
- *[***PLUGIN PRINCIPAL***]* # The primary part of the Ranger Kafka plugin principal.
- *[***KERBEROS REALM***]* # The Kerberos realm of the Ranger Kafka plugin principal.

core-site.xml

Save the following example. This file only contains hardcoded properties. You do not need to make any changes.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <!-- Required hardcoded configuration block start -->
  <property>
    <name>hadoop.security.authentication</name>
    <value>kerberos</value>
  </property>
  <property>
    <name>hadoop.rpc.protection</name>
    <value>privacy</value>
  </property>
  <!-- Required hardcoded configuration block end -->
</configuration>
```

rangerpluginssl.jceks and hadoop-credstore-password

Get the Ranger Admin server truststore password. Define and export the following shell variables so that the commands to be run can access them:

- TRUSTSTORE_PASSWORD: The truststore password obtained above
- HADOOP_CREDSTORE_PASSWORD: A user defined password which will be used to encrypt the truststore password

```
java -cp "ranger-kafka-plugin/install/lib/*" org.apache.ranger.c
redentialapi.buildks create sslTrustStore -value "${TRUSTSTORE_P
ASSWORD}" -provider "jceks://file/$(pwd)/conf/rangerpluginssl.jc
eks" -storetype "jceks"
```

```
echo -n $HADOOP_CREDSTORE_PASSWORD > conf/hadoop-credstore-passw
ord
```

Related Information

Configuring Apache Ranger High Availability

Configuring Ranger group authorization

Ranger makes it possible to authorize users based on the group that they are in. If you want to use group authorization, you must create a Secret containing LDAP data and extend your core-site.xml with required properties. This task is optional when configuring Ranger authorization for Kafka.

About this task

Ranger group authorization can be useful if you have a number of users but do not want to add them one-by-one to the policies. In this case you might want to utilize group authorization, which enables you to add groups to policies in Ranger.

If a user is a member of a group, it can be authorized based on the permissions of its group. The Ranger Kafka plugin running inside the Kafka broker pod is responsible for determining which groups the user belongs to. When configuring Ranger authorization, you can optionally set up LDAP group mapping, in which case the Ranger Kafka plugin connects to an LDAP server, looks up the end user that made the request to Kafka, and finds the groups that the user is a member of.

This is an optional step when configuring Ranger authorization. If you do not configure group mapping, user permissions can not be determined based on group membership in Ranger, but other authorization methods (such as user and role authorization) will still work as expected. Other group mappings are not supported in Cloudera Streams Messaging Operator for Kubernetes.

The following example demonstrates how you can configure your Kafka cluster to use LDAP group mapping.

Before you begin

- A running LDAP server is required.
- The LDAP TLS certificate is available to you.
- A bind user and password that Kafka can use to connect to the LDAP server for user and group lookup is available to you.

Procedure

1. Create additional LDAP resources in Kubernetes.

   Connection to LDAP requires some additional information (including sensitive data), such as a bind user password, a truststore, and a truststore password. You must create a Kubernetes `Secret` that contains this data. You mount this `Secret` to Kafka broker pods in a later step. Mounting the `Secret` makes the data available to the Kafka Ranger plugin. The plugin uses this data to connect to the LDAP server.

   a) Create the following three files containing LDAP data.

      - *[***LDAP TRUSTSTORE FILE***]* # Truststore file containing the LDAP server certificate in JKS format.
      - *[***LDAP TRUSTSTORE PASSWORD FILE***]* # File containing the password of the LDAP truststore.
      - *[***LDAP BIND USER PASSWORD FILE***]* # File containing the password of the bind user, which is used to connect to the LDAP server.

   b) Create a `Secret` using the files you created.

```
kubectl create secret generic [***LDAP CONFIG SECRET***] \
  --namespace [***NAMESPACE***] \
  --from-file=[***LDAP TRUSTSTORE FILE***] \
  --from-file=[***LDAP TRUSTSTORE PASSWORD FILE***] \
```

```
    --from-file=[***LDAP BIND USER PASSWORD FILE***]
```

2. Configure core-site.xml.

   The Ranger plugin searches for LDAP group lookup specific configuration in the core-site.xml file. Extend your core-site.xml file with LDAP related properties.

   Important:

   The following snippet is incomplete and only contains common configurations across LDAP installations, such as LDAP URL, bind user, and TLS configuration. It will not work out of the box.

   This is because each LDAP setup is different and you can have differences in, for example, where users or groups are looked for in the directory hierarchy, what attribute is checked for group memberships, and so on. As a result of this, you must extend this example based on your specific setup.

   For more information, see the *LDAP Groups Mapping* and *core-site.xml parameter reference* in the Apache Hadoop documentation. Search for the hadoop.security.group.mapping.ldap prefix in the parameter reference to find relevant parameters.

```
<property>
    <name>hadoop.security.group.mapping</name>
    <value>org.apache.hadoop.security.LdapGroupsMapping</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.url</name>
    <value>[***LDAP URL***]</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.bind.user</name>
    <value>[***LDAP BIND USER***]</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.bind.password.file</name>
    <value>/mnt/ldap/[***LDAP BIND USER PASSWORD FILE***]</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.ssl</name>
    <value>true</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.ssl.truststore</name>
    <value>/mnt/ldap/[***LDAP TRUSTSTORE FILE***]</value>
  </property>
  <property>
    <name>hadoop.security.group.mapping.ldap.ssl.truststore.password.fil
e</name>
    <value>/mnt/ldap/[***LDAP TRUSTSTORE PASSWORD FILE***]</value>
  </property>
```

Related Information

Adding a group

Hadoop Groups Mapping | Hadoop

core-site.xml parameter reference | Hadoop

Deploying a Ranger-integrated Kafka cluster

Once your custom image is ready and all configuration files are prepared, you can deploy a Kafka cluster that can integrate with Ranger. To do this, you deploy the Ranger Kafka plugin configuration files together with your Kafka cluster. The configuration files are mounted in the Kafka pods as volumes making them available to Ranger Kafka plugin. The files containing sensitive data are deployed as a Secret. Other files are deployed as a ConfigMap.

Procedure

1. Create a `ConfigMap` using the Ranger Kafka plugin configuration files that do not include any sensitive data.

```
kubectl create configmap [***KAFKA CLUSTER NAME***]-ranger-plugin-config \
  --namespace [***NAMESPACE***] \
  --from-file=conf/core-site.xml \
  --from-file=conf/jaas.conf \
  --from-file=conf/krb5.conf \
  --from-file=conf/ranger-kafka-audit.xml \
  --from-file=conf/ranger-kafka-policymgr-ssl.xml \
  --from-file=conf/ranger-kafka-security.xml \
  --from-file=conf/ranger_truststore.jks \
  --from-file=conf/rangerpluginssl.jceks
```

2. Create a `Secret` using the Ranger Kafka plugin configuration files that include sensitive data.

```
kubectl create secret generic [***KAFKA CLUSTER NAME***]-ranger-plugin-c
onfig-sensitive \
  --namespace [***NAMESPACE***] \
  --from-file=conf/hadoop-credstore-password \
  --from-file=conf/kafka_plugin.keytab
```

3. Create a YAML configuration containing your `PersistentVoulemClaim` and `KafkaNodePool` resources.

   In order to be able to function correctly in case of temporary connectivity issues, the Ranger Kafka plugin needs persistent storage. Each Kafka broker requires three distinct `PersistentVolumeClaims` (PVC) for this purpose. The name of the PVCs should match the mounted PVC names in `KafkaNodePool` resources.

   The required size of these volumes depend on your environment.

   - The policy cache size depends on how many Ranger policies, roles, and tags you defined.
   - The audit file spool serves the purpose of storing the audit logs locally if the Ranger Kafka plugin can not reach the audit destination.
   - The file spool volume size depends on how many audit logs are generated and how long you want to retain audit logs in the Kafka pod in case of connectivity issues to the audit target.

   The following snippet contains the necessary Kubernetes resource configurations (PVC and `KafkaNodePool`) for a single broker. If you have more than one broker, define a similar configuration for each one.

   

   Warning: As a result of the unique PVCs per broker requirement, a different `KafkaNodePool` needs to be defined for each broker. This also means that scaling of `KafkaNodePools` will not work as each `KafkaNodePool` must be limited to a single replica. If you want to scale your Kafka cluster, you have to define another `KafkaNodePool` (and PVCs) for the new broker.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-policy-cache
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: [***REQUIRED SIZE***]
    limits:
      storage: [***MAXIMUM SIZE***]
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-hdfs-filespool
spec:
```

```
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: [***REQUIRED SIZE***]
    limits:
      storage: [***MAXIMUM SIZE***]
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-solr-filespool
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: [**REQUIRED SIZE***]
    limits:
      storage: [**MAXIMUM SIZE***]
---

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]
  labels:
    strimzi.io/cluster: [***KAFKA CLUSTER NAME***]
spec:
  replicas: 1
  roles:
    - broker
...
  template:
    pod:
      volumes:
        - name: ranger-plugin-conf
          configMap:
            name: [***KAFKA CLUSTER NAME***]-ranger-plugin-config
        - name: ranger-plugin-sensitive
          secret:
            secretName: [***KAFKA CLUSTER NAME***]-ranger-plugin-config-se
nsitive
        - name: ranger-plugin-policy-cache
          persistentVolumeClaim:
            claimName: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-pol
icy-cache
        - name: ranger-plugin-hdfs-filespool
          persistentVolumeClaim:
            claimName: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-hdfs-
filespool
        - name: ranger-plugin-solr-filespool
          persistentVolumeClaim:
            claimName: [***KAFKA CLUSTER NAME***]-[***BROKER ID***]-solr-
filespool
    kafkaContainer:
      env:
        - name: HADOOP_CREDSTORE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: [***KAFKA CLUSTER NAME***]-ranger-plugin-config-sensit
ive
              key: hadoop-credstore-password
```

```
      volumeMounts:
        - name: ranger-plugin-conf
          mountPath: /mnt/ranger-kafka-plugin/conf
        - name: ranger-plugin-sensitive
          mountPath: /mnt/ranger-kafka-plugin/conf_sensitive
        - name: ranger-plugin-policy-cache
          mountPath: /mnt/ranger-kafka-plugin/cache
        - name: ranger-plugin-hdfs-filespool
          mountPath: /mnt/ranger-kafka-plugin/audit/hdfs/spool
        - name: ranger-plugin-solr-filespool
          mountPath: /mnt/ranger-kafka-plugin/audit/solr/spool
```

- *[***KAFKA CLUSTER NAME***]* # Kafka cluster name.
- *[***BROKER ID***]* # A unique broker ID within the same Kafka cluster.

4. If you configured Ranger group authorization, specify additional properties in your `KafkaNodePool` resources.

   If you completed configuration for Ranger group authorization, you must mount *[***LDAP CONFIG SECRET***]* as a volume and set the KAFKA_OPTS environment variable in your `KafkaNodePool` resources.

   Mounting *[***LDAP CONFIG SECRET***]* mounts the `Secret` in the Kafka pods and makes sensitive LDAP configuration available for the Ranger Kafka plugin. Setting the KAFKA_OPTS environment variable is required so that the group mapping functionality in the Ranger plugin works correctly with recent Java versions.

```
#...
kind: KafkaNodePool
spec:
  template:
    pod:
      volumes:
        - name: ldap-config
          secret:
            secretName: [***LDAP CONFIG SECRET***]
    kafkaContainer:
      env:
        - name: KAFKA_OPTS
          value: "--add-exports java.naming/com.sun.jndi.ldap=ALL-UNNAMED"
      volumeMounts:
        - name: ldap-config
          mountPath: /mnt/ldap
```

5. Deploy your `PersistentVolumeClaim` and `KafkaNodePool` resources.

   This creates the volumes and the `KafkaNodepools` that will be used by the Kafka cluster you deploy in a later step. If you created more than a single configuration file, ensure that you deploy all of them.

```
kubectl apply --file [***YAML CONFIG***] --namespace [***NAMESPACE***]
```

6. Ensure that the status of each PVC is bound

```
kubectl get pvc --namespace [***NAMESPACE***]
```

7. Create a YAML configuration that contains your `Kafka` resource.

```
#...
kind: Kafka
metadata:
  name: [***KAFKA CLUSTER NAME***]
  annotations:
    strimzi.io/node-pools: enabled
spec:
  kafka:
    image: [***YOUR REGISTRY***]/[***IMAGE***]:[***TAG***]
    jvmOptions:
```

```
      javaSystemProperties:
        - name: java.security.krb5.conf
          value: /opt/kafka/libs/ranger-kafka-plugin-impl/conf/krb5.conf
        - name: java.security.auth.login.config
          value: /opt/kafka/libs/ranger-kafka-plugin-impl/conf/jaas.conf
    authorization:
      type: custom
      authorizerClass: org.apache.ranger.authorization.kafka.authorizer
.RangerKafkaAuthorizer
    config:
      ranger.jaas.context: ranger
```

- *[***KAFKA CLUSTER NAME***]* # The name of the Kafka should be the one which was used during the service creation step in Ranger setup.
- *[***IMAGE REGISTRY***]/[***IMAGE NAME***]:[***TAG***]* #The location where you pushed the previously built Docker image, which contains Kafka and Ranger artifacts.

8. Deploy the `Kafka` resource.

```
kubectl apply --file [***YAML CONFIG***] --namespace [***NAMESPACE***]
```

Related Information

Persistent Volumes | Kubernetes

User management

Users are created and managed with the Strimzi Entity Operator and KafkaUser resources.

The Strimzi Entity Operator can set up external Kafka users with `KafkaUser` resources. In the`KafkaUser` resource, authentication can be configured with spec.authentication property and authorization can be configured using the spec.authorization.type property.

The following is an example of a `KafkaUser` resource that has tls authentication and simple authorization configured

```
#...
kind: KafkaUser
spec:
  authentication:
    type: tls
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - All
```

Inter-broker and metadata store security

Learn about security for inter-broker and broker to controller communications.

Inter-broker security

Kafka exposes ports 9090 and 9091 for inter-broker communication as well as communication with Cruise Control and the operators. These listeners are not configurable and use mTLS authentication by default. As a result, only clients that have access to the certificate secrets can access Kafka through these listeners. To protect these secrets, it is possible to further limit access to the cluster by using RBAC authorization to restrict namespace access to specific users.

By separating internal and external listeners, internal listener configurations can be simplified and kept secure when opening the cluster for access to external clients.

Broker to controller security

Kafka uses KRaft controllers (nodes with the controller role) to store and manage metadata. As with inter-broker communication, brokers talk to controllers over non#configurable listeners secured with mTLS authentication. This means the communication is encrypted by default.

If the `Kafka` resource is configured with the simple authorization plugin that is built into Kafka, then `org.apache.kafka.metadata.authorizer.StandardAuthorizer` is used.

To enable simple authorization, set the spec.kafka.authorization.type property to simple, and configure a list of super users. Super users are always allowed without querying Access Control List (ACL) rules. ACLs allow you to define which users have access to which resources at a granular level. Access rules can be specified for the `KafkaUser`.

```
#...
kind: Kafka
spec:
  kafka:
    authorization:
      type: simple
      superUsers:
        - CN=user-1
        - user-2
        - CN=user-3
```

Related Information

Using RBAC Authorization | Kubernetes

Setting the security context of Kafka cluster components

The `Kafka` resource allows users to specify the security context at the pod and container level with template properties.

The `Kafka` resource allows users to specify the security context at the pod and container level with template properties.

```
#...
kind: Kafka
spec:
  kafka:
    template:
      pod:
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop:
              - ALL
          runAsNonRoot: true
          seccompProfile:
```

```
            type: RuntimeDefault
      kafkaContainer:
        securityContext:
          # ...
  cruiseControl:
    template:
      pod:
        securityContext:
          # ...
      cruiseControlContainer:
        # ...
```

In addition to Kafka, you can also set the security context of other Kafka cluster components configured in the `Kafka` resource in the same way.

Related Information

Pod Security Standards | Kubernetes