Cloudera Streams Messaging Operator 1.0.0

Kafka Security

Date published: 2024-06-11
Date modified: 2024-06-11

CLOUDERA

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 and configuring OAuth for a listener in your Kafka resource.

Before you begin

Enure that you have the following:

- An OAuth server running that is accessible from the Kafka Kubernetes environment.
- Both Kafka brokers and clientsare 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 \
  -n kafka generic <oauth-server-cert-secret> \
  --from-file=<oauth-server-cert.pem>
```

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: <uri-from-kafka-brokers-to-oauth-server>
          tlsTrustedCertificates:
            - secretName: <oauth-server-cert-secret>
              certificate: <oauth-server-cert.pem>
          userNameClaim: <user-name-claim>
          validIssuerUri: <uri-from-kafka-clients-to-oauth-server>
          maxSecondsWithoutReauthentication: 3600
```

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 and configuring your Kafka resource to include a listener that has LDAP enabled.

Before you begin

Ensure that you have the following:

- An LDAP server running that is accessible from the Kafka Kubernetes environment.
- A truststore container that contains the CA certificate of the LDAP server (ldap.truststore.jks).

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

```
kubectl create secret -n kafka generic ldap-truststore --from-file=ldap-trus
tstore.jks
```

Afterward, modify the Kafka resource configuration 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.security.plai
n.PlainLoginModule required ssl.truststore.password="<ssl-truststore-passwor
d>" ssl.truststore.location="/opt/kafka/custom-authn-secrets/custom-listener
```

```
-ldap-9094/ldap-truststore/ldap-truststore.jks" ldap_url="ldaps://<ldap-serv
er-url:port>" user_dn_template="cn={0},ou=users,dc=ldap-dc,dc=ldap";'
            sasl.enabled.mechanisms: PLAIN
          secrets:
            - key: ldap-truststore.jks
              secretName: ldap-truststore
```

Note: By convention, the Strimzi Cluster Operator mounts custom listener secrets to /opt/kafka/custom-authn-secrets/custom-listener-<listener name>-<listener port>/<secret name>/<secret key>.

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

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.

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
```

Pod security

Learn how to run the Strimzi Cluster Operator and Kafka cluster pods with a restricted profile.

Running the Strimzi Cluster Operator with a restricted profile

You run the Strimzi Cluster Operator with a restricted profile by configuring the podSecurityContext Helm property.

By default, the Strimzi Cluster Operator runs with the baseline profile. However, the Helm templates allows customizing the security context of the Strimzi Cluster Operator with the podSecurityContext property. You run the Strimzi Cluster Operator with a restricted profile by specifying appropriate privileges during installation. For example, the helm install command you run would be similar to the following.

```
helm install csm-operator [***HELM CHART***] --namespace [***NAMESPACE***] \
  --create-namespace \
  --set watchAnyNamespace=true
  --set securityContext.allowPrivilegeEscalation=false \
  --set securityContext.capabilities.drop={ALL} \
  --set securityContext.runAsNonRoot=true \
  --set securityContext.seccompProfile.type=RuntimeDefault
```

Running Kafka clusters with restricted profile

You run your Kafka cluster with a restricted profile by either setting the security context manually in the Kafka resource with spec.*.template.pod.securityContext for each Kafka cluster component. Alternatively, you can use a pod security provider to set security context across all pods.

Setting the security context manually

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:
        # ...
```

Tip: The following additional resources have configurable pod and container templates: ZooKeeper, Strimzi Entity Operator, Kafka Connect, Kafka Exporter, Mirror Maker, Kafka NodePool.

Using security providers

Pod Security Providers allow you to manage the security context for all pods and containers managed by the Strimzi Cluster Operator from a single location. That is, a Security Provider defines the default security context of the pods and containers that the Strimzi Cluster Operator creates and manages. The following two providers are available.

Baseline

The Baseline Provider is based on the Kubernetes baseline security profile. This is a minimally restrictive profile that prevents privilege escalations and defines other standard access controls and limitations.

Restricted

The Restricted Provider is based on the Kubernetes restricted security profile. This is a highly restrictive profile that is aimed for use in environments where high levels of security is critical.

By default, the Strimzi Cluster Operator uses the Baseline Provider. To use the Restricted Provider, set the STRI MZI_POD_SECURITY_PROVIDER_CLASS environment variable of the Strimzi Cluster Operator to restricted. This is done during installation. For example:

```
helm install csm-operator [***HELM CHART***] --namespace [***NAMESPACE***] \
  --create-namespace \
  --set extraEnvs[0].name=STRIMZI_POD_SECURITY_PROVIDER_CLASS \
  --set extraEnvs[0].value=resticted
```

Related Information

Pod Security Standards | Kubernetes

Inter-broker and ZooKeeper security

Learn about inter-broker and ZooKeeper security.

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.

ZooKeeper security

Communication between the ZooKeeper servers on all ports, as well as between clients and ZooKeeper, is encrypted using TLS. Communication between Kafka brokers and ZooKeeper servers is also encrypted.

When both a keystore and a truststore are configured for both Kafka and ZooKeeper, both components use mTLS. There is no separate flag or configuration property you can use. This is enabled by default.

ZooKeeper uses ACLs to restrict access to Znodes. The ACL usage (zookeeper.set.acl) is not configurable, as it is managed by the Strimzi Cluster Operator itself.

Related Information

Using RBAC Authorization | Kubernetes