Cloudera Streams Messaging Operator for Kubernetes 1.6.0

Schema Registry Security

Date published: 2024-06-11
Date modified: 2026-01-27

CLOUDERA

https://docs.cloudera.com/

Legal Notice

Contents

Channel encryption (TLS) in Schema Registry

Get started with TLS encryption in Schema Registry. Learn which types of traffic you can secure with TLS, corresponding configuration properties, and recommended certificate management practices.

You can protect sensitive data and ensure secure access to Schema Registry by configuring TLS. TLS can be configured for the following types of traffic:

- **TLS for application traffic (backend/server-side)** – Secures traffic between the external access point (`LoadBalancer` or `Ingress`) and Schema Registry. Ensures that even internal traffic within the Kubernetes cluster remains encrypted, protecting against potential threats inside the cluster network.
- **TLS for client traffic** – Secures connections from end users (clients) to Schema Registry. Configured on the Kubernetes `Service` that provides external access (`LoadBalancer` or `Ingress`). Ensures that all data exchanged with external clients are encrypted.
- **TLS for OAuth** – Secures connections between Schema Registry and your OAuth provider. Typically, you set this up when you configure OAuth authentication.

Cloudera recommends that you enable TLS for all of the above listed categories. This ensures that all traffic handled by Schema Registry is secure.

Configuring TLS channel encryption

Configure TLS with TLS-related properties in your custom values file. TLS for application, client, and OAuth traffic is configured separately.

TLS for application traffic

TLS for application traffic is configured with the tls.* properties. For example:

```
#...
tls:
  enabled: true
  keystore:
    secretKeyRef:
      name: [***KEYSTORE SECRET NAME***]
      key: [***KEYSTORE SECRET KEY***]
    password:
      secretKeyRef:
        name: [***KEYSTORE SECRET NAME***]
        key: [***KEYSTORE PASSWORD SECRET KEY***]
    type: PKCS12
```

- tls.enabled – Enables or disables TLS for Schema Registry.
- tls.keystore.secretKeyRef.* – The `Secret` name and `Secret` key that contain the keystore.
- tls.keystore.password.secretKeyRef.* – The `Secret` name and `Secret` key that contain the keystore password.

TLS for client traffic

TLS for client traffic is configured on the `Service` (`Ingress` or `LoadBalancer`) that provides access to Schema Registry.

For `Ingress`, you configure ingress.tls.* properties. For example:

```
#...
ingress:
  enabled: true
  className: "nginx"
```

```
  rules:
    path: "/"
    host: "my-domain.example.com"
  tls:
    enabled: true
    secretRef: [***INGRESS TLS CERT SECRET***]
  extraAnnotations:
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
```

TLS for `Ingress` is typically configured when you set up external access with `Ingress`. For a step-by-step guide see *Configuring external access with Ingress*.

For `LoadBalancers`, the actual load balancer is provisioned and managed by your cloud or infrastructure provider. As a result, TLS settings and certificate management may vary depending on the platform. Refer to vendor-specific documentation for detailed guidance on configuring TLS.

TLS for OAuth traffic

TLS for OAuth traffic is configured using the authentication.oauth.jwks.tls.* properties. These properties reference a Kubernetes `Secret` containing the truststore (PKCS12) of the root Certificate Authority (CA) of the OAuth certificate chain. You typically configure these properties when you set up OAuth, see *Authentication in Schema Registry*.

Related Information

Configuring external access with Ingress

Authentication in Schema Registry

Managing certificates

Learn about managing TLS certificates for Schema Registry. You can manage certificates manually or use cert-manager to automate certificate management. Cloudera recommends automatic certificate management.

TLS certificates for Schema Registry are stored in various `Secrets`. The `Secrets` are specified by the following properties:

- tls.keystore.secretKeyRef.name – The name of the `Secret` containing the TLS keystore used by Schema Registry.
- ingress.tls.secretRef – The name of the `Secret` containing Ingress TLS certificates.
- authentication.oauth.jwks.tls.truststore.secretKeyRef.name – The name of the `Secret` that contains the truststore for accessing the JWKS endpoint.

These `Secrets` must contain a valid certificate and private key. Cloudera recommends that you use cert-manager to manage the `Secrets` and the certificates that they store. Alternatively, you can choose to manage them manually.

Automatic certificate management with cert-manager

cert-manager is a popular Kubernetes add-on for automating the management and issuance of TLS certificates. In order to manage the certificates used by Schema Registry with cert-manager, you need the following:

- A cert-manager instance in your Kubernetes cluster.
- An `Issuer` deployed for cert-manager.

The management of the `Secrets` that Schema Registry uses to store certificates differs.

- tls.keystore.secretKeyRef.name and authentication.oauth.jwks.tls.truststore.secretKeyRef.name – The `Certificate` resource for these `Secrets` must be created manually. When creating a `Certificate` resource, set spec.secretName to the names specified in these properties. This way cert-manager saves the certificates and private keys into the appropriate `Secrets` that Schema Registry expects.

- ingress.tls.secretRef – The `Certificate` resource for this `Secret` is created automatically when the cert-manager.io/issuer: *[***ISSUER NAME***]* annotation is set in the ingress.extraAnnotations property.

  Specifically, the `Ingress` requests a certificate from cert-manager using the `Issuer` name from the annotation. This triggers the creation of the `Certificate` resource and saves the certificate file and private key to the `Secret` defined in ingress.tls.secretRef.

Manual certificate management

When managing certificates manually, you must create the `Secrets` that contain the certificates and private keys manually. Ensure that you create and update the appropriate `Secrets`.

Authentication in Schema Registry

Get started with OAuth authentication in Schema Registry. OAuth is the only supported authentication mechanism in Schema Registry.

Schema Registry supports OAuth authentication to integrate with an external identity provider. When OAuth is enabled, clients connecting to Schema Registry must present a valid Bearer JSON Web Token (JWT) for access. Incoming JWTs are verified by Schema Registry using a JSON Web Key Set (JWKS) which provides public keys required to validate signatures.

OAuth is enabled by default and is the only supported authentication mechanism. As a result, you must configure OAuth properties during or after installation unless you choose to explicitly disable OAuth. However, Cloudera does not recommend that you deploy Schema Registry with authentication disabled.

Tokens and authorization

OAuth JWTs presented by clients are also used for authorization. Schema Registry extracts the principal (the username) from a configured JWT claim, validates the token audience, and enforces authorization using a built-in authorizer. Because of this, configuring OAuth authentication is required for authorization. For more information, see *Authorization in Schema Registry*.

Related Information

Authorization in Schema Registry

Configuring OAuth authentication

Learn how to configure OAuth authentication in Schema Registry.

Before you begin

- An OAuth server is available that has TLS enabled.
- The server is accessible from the Kubernetes cluster where Schema Registry is deployed.
- At least one client must be configured in your realm that supports Client Credentials flow (sometimes referred to as Machine-to-Machine (M2M), Service Account, or Application Permissions).
- Identify if your OAuth server issues tokens that contain a value in the aud claim. If a value is present, note it down as you will need to provide it in your configuration. Referred to as *[***OAUTH EXPECTED AUDIENCE***]* in the following steps.
- Get the JWKS endpoint URL of your OAuth server. You will need to provide it in your configuration. Schema Registry requires this endpoint to validate the signatures of incoming tokens. Referred to as *[***OAUTH JWKS URL***]* in the following steps.

Procedure

1. Generate a Java truststore (PKCS12) containing the TLS certificate of the root Certificate Authority (CA) of the OAuth certificate chain.

```
keytool -import -trustcacerts -file [***OAUTH ROOT CA***] \
  -keystore [***TRUSTSTORE NAME***] \
  -storepass [***TRUSTSTORE PASSWORD***] \
  -storetype PKCS12
```

2. Create a `Secret` containing the truststore and its password.

```
kubectl create secret generic [***OAUTH TRUSTSTORE SECRET NAME***] \
  --namespace [***NAMESPACE***] \
  --from-file=[***OAUTH TRUSTSTORE SECRET KEY***]=[***TRUSTSTORE NAME***]
 \
  --from-file=[***OAUTH TRUSTSTORE PASSWORD SECRET KEY***]=[***PATH TO
 TRUSTSTORE PW FILE***]
```

   Take note of *[***OAUTH TRUSTSTORE SECRET NAME***]*, *[***OAUTH TRUSTSTORE SECRET KEY***]*, and *[***OAUTH TRUSTSTORE PASSWORD SECRET KEY***]*.

3. Configure OAuth properties in a custom values file (values.yaml).

```
#...
authentication:
  oauth:
    enabled: true
    jwt:
      expectedAudience: [***OAUTH EXPECTED AUDIENCE***]
    jwks:
      url: [***OAUTH JWKS URL***]
      tls:
        truststore:
          secretKeyRef:
            name: [***OAUTH TRUSTSTORE SECRET NAME***]
            key: [***OAUTH TRUSTSTORE SECRET KEY***]
          password:
            secretKeyRef:
              name: [***OAUTH TRUSTSTORE SECRET NAME***]
              key: [***OAUTH TRUSTSTORE PASSWORD SECRET KEY***]
          type: PKCS12
```

   - authentication.oauth.enabled – Enables OAuth authentication for the Schema Registry server.
   - authentication.oauth.jwt.expectedAudience – The expected audience value. If the JWT token contains an aud claim, it must match this value, otherwise the token is considered invalid.
   - authentication.oauth.jwks.url – The URL to the JWKS endpoint.
   - authentication.oauth.jwks.tls.truststore.secretKeyRef.name – The name of the `Secret` that contains the truststore for accessing the JWKS endpoint. Configure this property if the backend of your JWKS has self-signed certificates.
   - authentication.oauth.jwks.tls.truststore.secretKeyRef.key – The key in the `Secret` specified by authenticati on.oauth.jwks.tls.truststore.secretKeyRef.name that contains the truststore for accessing the JWKS endpoint.
   - authentication.oauth.jwks.tls.truststore.password.secretKeyRef.name – The name of the `Secret` that contains the truststore password for accessing the JWKS endpoint.
   - authentication.oauth.jwks.tls.truststore.password.secretKeyRef.key – The key in the `Secret` specified by authentication.oauth.jwks.tls.truststore.password.secretKeyRef.name that contains the truststore password for accessing the JWKS endpoint.

4. Apply your configuration.

```
helm upgrade SCHEMA-REGISTRY [***CHART***] \
```

```
  --namespace [***NAMESPACE***] \
  --values [***VALUES.YML***] \
  --reuse-values
```

Results

OAuth authentication is enabled. Users are required to present a valid token to Schema Registry for access.

Authorization in Schema Registry

Get started with Schema Registry authorization. Learn how to enable authorization and configure users, and how to configure which JWT (JSON Web Token) claim the server uses for principal mapping.

Schema Registry includes a built-in basic authorizer that enforces access using OAuth and JWTs. The authorizer defines two user types: admin (full privileges) and read-only (read-only privileges). By default Schema Registry uses the sub JWT claim as the principal. The server reads the configured principal claim from each token and compares its exact string value to entries in authorization.simple.adminUsers and authorization.simple.readOnlyUsers, so the claim value must exactly match a configured entry.

- If the principal is in adminUsers, all access is granted.
- Else if the principal is in readOnlyUsers, read access is granted.
- Else access is denied.

Requirements

- OAuth authentication must be enabled. See Authentication in Schema Registry on page 6.

Configuring authorization and users

Configure authorization and authorized users using authorization.simple.* properties. Set authorization.simple.enabled to true and provide lists for authorization.simple.adminUsers and authorization.simple.readOnlyUsers.

```
#...
authorization:
  simple:
    enabled: true
    adminUsers:
      - 1234567890
    readOnlyUsers:
      - 0987654321
```

- authorization.simple.enabled – Enables or disables authorization.
- authorization.simple.adminUsers – A list of admin usernames. Admin users can perform any operation in Schema Registry.
- authorization.simple.readOnlyUsers – A list of read-only usernames. Read-only users can only perform read operations in Schema Registry.

Configuring principal mapping

Configure the principal claim used for authorization with authentication.oauth.jwt.principalClaimName

By default Schema Registry uses the sub JWT claim as the principal. The default sub claim often contains an opaque identifier (numeric id or UUID). If your principals are provided in a different claim, set authentication.oauth.jwt.pri

ncipalClaimName to that claim (for example, email or preferred_username) and ensure those claim values appear exactly in authorization.simple.adminUsers or authorization.simple.readOnlyUsers.

```
#...
authentication:
  oauth:
    jwt:
      principalClaimName: email

authorization:
  simple:
    enabled: true
    adminUsers:
      - ALICE@EXAMPLE.COM
    readOnlyUsers:
      - BOB@EXAMPLE.COM
```

- authentication.oauth.jwt.principalClaimName – JWT claim name used to identify the principal. Default: sub.
- authorization.simple.adminUsers – A list of admin usernames. Admin users can perform any operation in Schema Registry.
- authorization.simple.readOnlyUsers – A list of read-only usernames. Read-only users can only perform read operations in Schema Registry.