Red Hat Streams for Apache Kafka 2.7

Using the Streams for Apache Kafka Proxy

Enhancing Kafka with specialized features

Last Updated: 2024-07-03

Red Hat Streams for Apache Kafka 2.7 Using the Streams for Apache Kafka Proxy

Enhancing Kafka with specialized features

Abstract

Streams for Apache Kafka Proxy is an Apache Kafka protocol-aware proxy designed to enhance Kafka-based systems through its filter mechanism. In this preview, Streams for Apache Kafka Proxy includes a Record Encryption filter, which provides encryption at rest for data stored in a Kafka cluster.

Table of Contents

PREFACE

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation.

To propose improvements, open a Jira issue and describe your suggested changes. Provide as much detail as possible to enable us to address your request quickly.

Prerequisite

- You have a Red Hat Customer Portal account. This account enables you to log in to the Red Hat Jira Software instance.
  If you do not have an account, you will be prompted to create one.

Procedure

1. Click the following: Create issue.
2. In the **Summary** text box, enter a brief description of the issue.
3. In the **Description** text box, provide the following information:
   - The URL of the page where you found the issue.
   - A detailed description of the issue.
     You can leave the information in any other fields at their default values.
4. Add a reporter name.
5. Click **Create** to submit the Jira issue to the documentation team.

Thank you for taking the time to provide feedback.

TECHNOLOGY PREVIEW

The Streams for Apache Kafka Proxy is a technology preview.

Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about the support scope, see Technology Preview Features Support Scope.

CHAPTER 1. STREAMS FOR APACHE KAFKA PROXY OVERVIEW

Streams for Apache Kafka Proxy is an Apache Kafka protocol-aware proxy designed to enhance Kafka-based systems. Through its filter mechanism it allows additional behavior to be introduced into a Kafka-based system without requiring changes to either your applications or the Kafka cluster itself.

Functioning as an intermediary, the Streams for Apache Kafka Proxy mediates communication between a Kafka cluster and its clients. It takes on the responsibility of receiving, filtering, and forwarding messages.

1.1. RECORD ENCRYPTION FILTER

Streams for Apache Kafka Proxy's Record Encryption filter enhances the security of Kafka messages. The filter uses industry-standard cryptographic techniques to apply encryption to Kafka messages, ensuring the confidentiality of data stored in the Kafka Cluster. Streams for Apache Kafka Proxy centralizes topic-level encryption, ensuring streamlined encryption across Kafka clusters.

The filter uses envelope encryption to encrypt the records with symmetric encryption keys.

Envelope encryption

Envelope encryption is an industry-standard technique suited for encrypting large volumes of data in an efficient manner. Data is encrypted with a Data Encryption Key (DEK). The DEK is encrypted using a Key Encryption Key (KEK). The KEK is stored securely in a Key Management System (KMS).

Symmetric encryption keys

AES(GCM) 256 bit encryption symmetric encryption keys are used to encrypt and decrypt record data.

The process is as follows:

1. The filter intercepts produce requests from producing applications and encrypts the records.
2. The produce request is forwarded to the broker.
3. The filter intercepts fetch responses from consuming applications and decrypts the records.
4. The fetch response is forwarded to the consuming application.

The filter encrypts the record value only. Record keys, headers, and timestamps are not encrypted.

The entire process is transparent from the point of view of Kafka clients and Kafka brokers. Neither are aware that the records are being encrypted, nor do they have any access to the encryption keys or have any influence on the ciphering process to secure the records.

The filter integrates with a Key Management Service (KMS), which has ultimate responsibility for the safe storage of key material. Currently, the filter integrates with HashiCorp Vault as its KMS, though further supported KMS integrations are planned.

1.1.1. How the filter encrypts records

The filter encrypts records from produce requests as follows:

1. Filter selects a KEK to apply.

2. Requests the KMS to generate a DEK for the KEK.
3. Uses an encrypted DEK (DEK encrypted with the KEK) to encrypt the record.
4. Replaces the original record with a cipher record (encrypted record, encrypted DEK, and metadata).

The filter uses a DEK reuse strategy. Encrypted records are sent to the same topic using the same DEK until a time-out or an encryption limit is reached.

1.1.2. How the filter decrypts records

The filter decrypts records from fetch responses as follows:

1. Filter receives a cipher record from the Kafka broker.
2. Reverses the process that constructed the cipher record.
3. Uses KMS to decrypt the DEK.
4. Uses the decrypted DEK to decrypt the encrypted record.
5. Replaces the cipher record with a decrypted record.

The filter uses an LRU (least recently used) strategy for decrypted records. Decrypted DEKs are kept in memory to reduce interactions with the KMS.

1.1.3. How the filter uses the KMS

To support the filter, the KMS provides the following:

- A secure repository for storing Key Encryption Keys (KEKs)
- A service for generating and decrypting Data Encryption Keys (DEKs)

KEKs stay within the KMS. The KMS generates a DEK (which is securely generated random data) for a given KEK, then returns the DEK and an encrypted DEK. The encrypted DEK has the same data but encrypted with the KEK. The KMS doesn't store DEKs; they are stored as part of the cipher record in the broker.

WARNING

The KMS must be available during runtime. If the KMS is unavailable, production and consumption through the filter become impossible until KMS service is restored. It is recommended to use the KMS in a high availability (HA) configuration.

1.1.4. What part of a record is encrypted?

The record encryption filter encrypts only the values of records, leaving record keys, headers, and timestamps untouched. Null record values, which might represent deletions in compacted topics, are transmitted to the broker unencrypted. This approach ensures that compacted topics function correctly.

1.1.5. Unencrypted topics

You may configure the system so that some topics are encrypted and others are not. This supports scenarios where topics with confidential information are encrypted and Kafka topics with non-sensitive information can be left unencrypted.

CHAPTER 2. PREPARING HASHICORP VAULT FOR THE RECORD ENCRYPTION FILTER

To use Vault with the Record Encryption filter in an OpenShift cluster, use the following setup for your Vault instance:

- Enable the Transit Engine as the Record Encryption filter relies on its APIs.
- Create a Vault policy specifically for the filter with permissions for generating and decrypting Data Encryption Keys (DEKs) for envelope encryption.
- Obtain a Vault token that includes the filter policy.

The deployment configuration for the proxy uses the URL for the Vault Transit Engine service.

Vault can be deployed as an existing instance, a cloud instance, or on OpenShift. With accessibility to the proxy, it can either be co-located with the Streams for Apache Kafka Proxy or deployed remotely.

For information on installing Vault on OpenShift and setting up access, refer to the HashiCorp Vault product documentation.

This procedure outlines two options for preparing Vault:

- Deploying Vault to the OpenShift cluster using Helm with an example ephemeral deployment configuration provided with Streams for Apache Kafka Proxy.
- Updating your existing Vault instance.

When you have prepared a Vault instance, you must then create a Vault policy and token for the Record Encryption filter.

WARNING

The example deployment configuration is not suitable for production environments.

Streams for Apache Kafka includes example installation artifacts in the **examples/proxy/record-encryption/vault** folder, which contains pre-configured Vault deployment files compatible with the proxy and Record Encryption filter.

- **amqstreams_proxy_encryption_filter_policy.hcl** defines a Vault policy for the Record Encryption filter
- **helm-dev-values.yaml** specifies the Helm deployment configuration for Vault

These installation files offer a quick setup for trying out the proxy.

Prerequisites

- Installation requires an OpenShift user with **cluster-admin** role, such as **system:admin**.

- The **oc** command-line tool is installed and configured to connect to the OpenShift cluster with admin access.
- The **helm** command line tool is installed and configured to connect to the OpenShift cluster with admin access.
- An OpenShift project namespace called **proxy**, which is the same namespace where the proxy is installed by default.

For information on the **oc** and **helm** command line options used in this procedure, check the **--help**.

Deploying Vault using the example Helm deployment configuration

1. Download and extract the Streams for Apache Kafka Proxy installation artifacts.
   The proxy is available from Streams for Apache Kafka software downloads page .

   The files contain the deployment configuration required for deploying Vault.

2. Create a root token and make a note of it:

   ```
   cat /dev/urandom | LC_ALL=C tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1 > vault.root.token
   export VAULT_TOKEN=$(cat vault.root.token)
   ```

3. Install Vault using Helm:

   ```
   helm repo add hashicorp https://helm.releases.hashicorp.com
   helm install vault hashicorp/vault \
     --create-namespace --namespace=vault \
     --version <helm_version> \
     --values vault/helm-dev-values.yaml \
     --set server.dev.devRootToken=${VAULT_TOKEN} \
     --wait
   ```

   The root token is used for the Vault instance.

4. Check the status of the deployment:

   ```
   oc get pods -n vault
   ```

   Output shows the deployment name and readiness

   ```
   NAME                    READY  STATUS   RESTARTS
   vault-0               1/1    Running  0
   ```

   A pod ID identifies the pod created.

   With the default deployment, you install a single proxy pod.

   READY shows the number of replicas that are ready/expected. The deployment is successful when the STATUS displays as Running.

5. Create a Vault address (**VAULT_ADDR**) environment variable to point to the new Vault instance:

   ```
   export VAULT_ADDR=$(oc get route -n vault vault --template='https://{{.spec.host}}')
   ```

6. Login to Vault as an administrator and enable the Vault Transit secrets engine:

   ```
   vault secrets enable transit
   ```

   If the secrets engine is already enabled, ignore the error.

7. Create an environment variable to point to the Vault Transit address:

   ```
   export VAULT_TRANSIT_URL=${VAULT_ADDR}/v1/transit
   ```

   The address is used in the proxy deployment configuration.

8. Create a Vault policy and token .

Configuring your own Vault instance

If you already have a Kafka instance installed, you can update it to use it with Streams for Apache Kafka Proxy.

1. Create a Vault address environment variable (**VAULT_ADDR** and **VAULT_NAMESPACE**, if using Enterprise) to point to the Vault instance:

   ```
   export VAULT_ADDR=https://<vault server>:8200
   export VAULT_NAMESPACE=<namespaces>
   ```

2. Login to Vault as an administrator and enable the Vault Transit secrets engine:

   ```
   vault secrets enable transit
   ```

   If the secrets engine is already enabled, ignore the error.

3. Create an environment variable to point to the Vault Transit address:

   ```
   export VAULT_TRANSIT_URL=${VAULT_ADDR}/v1/${VAULT_NAMESPACE}/transit
   ```

   The address is used in the proxy deployment configuration.

4. Update the proxy deployment configuration to refer to your Vault instance:

   ```
   sed -i "s/\(vaultTransitEngineUrl:\).*$/\1 ${VAULT_TRANSIT_URL}/" */proxy/proxy-
   config.yaml
   ```

5. Create a Vault policy and token .

Creating a Vault policy and token

With the Vault instance set up, create a Vault policy and token for the Record Encryption filter.

1. Create a Vault policy:

   ```
   vault policy write amqstreams_proxy_encryption_filter_policy
   vault/amqstreams_proxy_encryption_filter_policy.hcl
   ```

   Write the policy to Vault using the HashiCorp policy definition file (**.hcl**) provided with Streams for Apache Kafka Proxy. The policy is named **amqstreams_proxy_encryption_filter_policy**.

2. Create a Vault token:

```
vault token create \
  -display-name "amqstreams-proxy encryption filter" \
  -policy=amqstreams_proxy_encryption_filter_policy \
  -no-default-policy \
  -orphan \
  -field=token > vault.encryption.token
```

   The command creates a token with the specified policy, and with no associated parent token or default policies.

3. Create a secret containing the token:

```
oc create secret generic proxy-encryption-vault-token \
  -n proxy \
  --from-file=encryption-vault-token.txt=vault.encryption.token \
  --dry-run=client \
  -o yaml > base/proxy/proxy-encryption-vault-token-secret.yaml
```

   The command stores the Vault token in the secret and creates the secret as a YAML file in the **proxy** namespace.

   The **proxy-encryption-vault-token-secret.yaml** secret is applied to the OpenShift cluster when deploying Streams for Apache Kafka Proxy with the Record Encryption filter.

TIP

Rotate keys periodically to minimize the impact of compromised keys. When using a Key Management System (KMS), such as HashiCorp Vault, rotate the Key Encryption Key (KEK) stored in the KMS. Streams for Apache Kafka Proxy manages DEK rotation automatically. Occasional restarts may be necessary for the proxy to pick up the new key. Additionally, encrypted messages should include key version metadata to indicate key rotation.

CHAPTER 3. DEPLOYING STREAMS FOR APACHE KAFKA PROXY WITH THE RECORD ENCRYPTION FILTER

Streams for Apache Kafka Proxy is designed to seamlessly integrate with Kafka clusters managed by Streams for Apache Kafka. Additionally, it offers compatibility with all types of Kafka instances, irrespective of their distribution or protocol version. Whether your deployment involves public or private clouds, or if you are setting up a local development environment, the instructions in this guide are applicable in all cases.

In this procedure, the Streams for Apache Kafka Proxy is deployed with the Record Encryption filter for use with a Kafka instance managed by Streams for Apache Kafka on OpenShift.

Streams for Apache Kafka provides example installation artifacts with the necessary configuration for the Streams for Apache Kafka Proxy to connect to the Kafka cluster in the **examples/proxy/record-encryption** folder.

Using the example configuration files, deploy and expose the proxy with the following types of listener:

- **cluster-ip** type listener using per-broker **ClusterIP** services to expose the proxy within the OpenShift cluster
- **loadbalancer** type listener using per-broker **loadbalancer** services to expose the proxy outside the OpenShift cluster

For each option, the following files are provided:

- **kustomization.yaml** specifies the Kubernetes customization for deploying the proxy
- **proxy-config.yaml** specifies the **ConfigMap** resource configuration for the proxy
- **proxy-service.yaml** specifies the **Service** resource configuration for the proxy service

The **ConfigMap** resource provides the configuration for setting up virtual clusters and filters. Virtual clusters represent the Kafka clusters you wish to use with the proxy.

Prerequisites

- An OpenShift cluster running a supported version.
- A Kafka cluster managed by Streams for Apache Kafka running on the OpenShift cluster.
- Kafka binaries installed locally to verify a proxy setup for external access through a loadbalancer. The Kafka binaries are included with the installation artifacts for Streams for Apache Kafka on RHEL from the Streams for Apache Kafka software downloads page .
- A config map that includes the configuration for creating virtual clusters and filters.
- The **oc** command-line tool is installed and configured to connect to the OpenShift cluster with admin access.
- The **helm** command line tool is installed and configured to connect to the OpenShift cluster with admin access.
- HashiCorp Vault is set up for the proxy and is accessible from the Streams for Apache Kafka Proxy.
  Make sure the Vault instance is set up for the Record Encryption filter.

For information on the **oc** and **helm** command line options used in this procedure, check the **--help**.

In addition to the files to install the Streams for Apache Kafka Proxy, Streams for Apache Kafka Proxy also provides pre-configured files to install a Kafka cluster. The installation files offer the quickest way to set up and try the proxy, though you can use your own deployments of a Kafka cluster managed by Streams for Apache Kafka and Vault.

In this procedure, we are connecting to a Kafka cluster named **my-cluster** that is deployed to the **kafka** namespace. To deploy the proxy to the same namespace as the cluster managed by Streams for Apache Kafka, change the **namespace** setting in the **kustomization.yaml** file. The proxy is deployed to the **proxy** namespace by default. If you keep this setting, the Streams for Apache Kafka Operator must be installed cluster-wide.

Procedure

1. Download and extract the Streams for Apache Kafka Proxy installation artifacts.
   The artifacts are included with installation and example files available from the Streams for Apache Kafka software downloads page.

   The files contain the deployment configuration required for connecting through a **cluster-ip** or **loadbalancer** type listener.

2. Create a topic in the Kafka cluster:

   ```
   oc run -n <my_proxy_namespace> -ti proxy-producer \
     --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
     --rm=true \
     --restart=Never \
     -- bin/kafka-topics.sh \
     --bootstrap-server proxy-service:9092 \
     --create -topic my-topic
   ```

   In this example, we create a topic named **my-topic** through an interactive pod container.

3. Create a key for **my-topic** in Vault:

   ```
   vault write -f transit/keys/KEK_my-topic
   ```

4. Edit the **ConfigMap** that provides the filter configuration for the proxy.
   The Record Encryption filter **config** requires credentials for the HashiCorp Vault KMS.

   #### Example Record Encryption filter configuration

   ```
   filters:
     - type: RecordEncryption
       config:
         kms: VaultKmsService (1)
         kmsConfig:
           vaultTransitEngineUrl: http://vault.vault.svc.cluster.local:8200/v1/transit (2)
           vaultToken:
             passwordFile: /opt/proxy/encryption/token.txt (3)
         selector: TemplateKekSelector (4)
   ```

```
      selectorConfig:
        template: "KEK_${topicName}" (5)
  # ...
```

(1) The type of KMS (key management system) used. In this case, HashiCorp Vault.

(2) The URL of the Vault Transit Engine service.

(3) The file containing the token required to access the Vault service. If this location changes, equivalent changes are required in the proxy deployment configuration.

(4) The Key Encryption Key (KEK) selector to use. The **${topicName}** is a literal understood by the proxy.

(5) The template for deriving the KEK, based on a specific topic name.

5. Deploy Streams for Apache Kafka Proxy with the Record Encryption filter and the appropriate listener to your OpenShift cluster:

   Deploying the proxy with a cluster-ip listener

   ```
   cd /examples/proxy/record-encryption/
   oc apply -k cluster-ip
   ```

   Deploying the proxy with a loadbalancer listener

   ```
   cd /examples/proxy/record-encryption/
   oc apply -k loadbalancer
   ```

6. If you are using a **loadbalancer** listener, update the proxy configuration to use the address of loadbalancer service that was created.

   a. Get the external address of the proxy service:

      ```
      LOAD_BALANCER_ADDRESS=$(oc get service -n <my_proxy_namespace> proxy-service --template='{{(index .status.loadBalancer.ingress 0).hostname}}')
      ```

   b. Update the **brokerAddressPattern** property in the proxy service configuration to use the broker address:

      ```
      sed -i "s/\(brokerAddressPattern:\).*$/\1 ${LOAD_BALANCER_ADDRESS}/" load-balancer/proxy/proxy-config.yaml
      ```

   c. Apply the change to the proxy configuration and restart the proxy pod.

      ```
       oc apply -k load-balancer && oc delete pod -n <my_proxy_namespace> --all
      ```

7. Check the status of the deployment:

   ```
   oc get pods -n <my_proxy_namespace>
   ```

   Output shows the deployment name and readiness

   –

```
NAME                    READY  STATUS   RESTARTS
my-cluster-kafka-0        1/1    Running  0
my-cluster-kafka-1        1/1    Running  0
my-cluster-kafka-2        1/1    Running  0
my-cluster-proxy-<pod_id> 1/1    Running  0
```

my-cluster-proxy is the name of the proxy.

A pod ID identifies the pod created.

With the default deployment, you install a single proxy pod.

READY shows the number of replicas that are ready/expected. The deployment is successful when the STATUS displays as Running.

8. Verify that the encryption has been applied to the specified topics by producing messages through the proxy and then consuming directly and indirectly from the Kafka cluster.

3.1. VERIFYING THE PROXY WHEN USING THECLUSTER-IP TYPE LISTENER

Verify the proxy is working when using the **cluster-ip** type listener by running interactive pod containers for Kafka producers and consumers within the OpenShift cluster.

1. Produce messages from the proxy:

Producing messages through the proxy

```
oc run -n <my_proxy_namespace> -ti proxy-producer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never \
  -- bin/kafka-console-producer.sh \
  --bootstrap-server proxy-service:9092 \
  --topic my-topic
```

2. Consume messages directly from the Kafka cluster to show they are encrypted:

Consuming messages directly from the Kafka cluster

```
oc run -n my-cluster -ti cluster-consumer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never
  -- ./bin/kafka-console-consumer.sh  \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 \
  --topic my-topic \
  --from-beginning \
  --timeout-ms 10000
```

3. Consume messages from the proxy to show they are decrypted automatically:

Consuming messages directly from the Kafka cluster

```
oc run -n <my_proxy_namespace> -ti proxy-consumer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never \
  -- ./bin/kafka-console-consumer.sh \
  --bootstrap-server proxy-service:9092 \
  --topic my-topic --from-beginning --timeout-ms 10000
```

3.2. VERIFYING THE PROXY WHEN USING THE LOADBALANCER TYPE LISTENER

Verify the proxy is working when using the **loadbalancer** type listener by running a Kafka producer and consumer through the proxy locally.

1. Produce messages from the proxy using the loadbalancer address:

   Producing messages through the proxy

   ```
   kafka-console-producer \
   --bootstrap-server <load_balancer_address>:9092 \
   --topic my-topic
   ```

2. Consume messages directly from the Kafka cluster using an interactive pod container to show they are encrypted:

   Consuming messages directly from the Kafka cluster

   ```
   oc run -n my-cluster -ti cluster-consumer \
     --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
     --rm=true \
     --restart=Never
     -- ./bin/kafka-console-consumer.sh  \
     --bootstrap-server my-cluster-kafka-bootstrap:9092 \
     --topic my-topic \
     --from-beginning \
     --timeout-ms 10000
   ```

3. Consume messages from the proxy to show they are decrypted automatically:

   Consuming messages directly from the Kafka cluster

   ```
   kafka-console-consumer \
   --bootstrap-server <load_balancer_address>:9092 \
   --topic my-topic \
   --from-beginning \
   --timeout-ms 10000
   ```

CHAPTER 4. CONFIGURING STREAMS FOR APACHE KAFKA PROXY

Fine-tune your deployment by configuring Streams for Apache Kafka Proxy resources to include additional features according to your specific requirements.

4.1. EXAMPLE STREAMS FOR APACHE KAFKA PROXY CONFIGURATION

Streams for Apache Kafka Proxy configuration is defined in a **ConfigMap** resource. Use the **data** properties of the **ConfigMap** resource to configure the following:

- Virtual clusters that represent the Kafka clusters
- Network addresses for broker communication in a Kafka cluster
- Filters to introduce additional functionality to the Kafka deployment

In this example, configuration for the Record Encryption filter is shown.

Example Streams for Apache Kafka Proxy configuration

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: proxy-config
data:
  config.yaml: |
    adminHttp: (1)
      endpoints:
        prometheus: {}
    virtualClusters: (2)
      my-cluster-proxy: (3)
        targetCluster:
          bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093 (4)
          tls: (5)
            trust:
              storeFile: /opt/proxy/trust/ca.p12
              storePassword:
                passwordFile: /opt/proxy/trust/ca.password
        clusterNetworkAddressConfigProvider: (6)
          type: SniRoutingClusterNetworkAddressConfigProvider
          Config:
            bootstrapAddress: mycluster-proxy.kafka:9092
            brokerAddressPattern: broker$(nodeId).mycluster-proxy.kafka
        logNetwork: false (7)
        logFrames: false
        tls: (8)
          key:
            storeFile: /opt/proxy/server/key-material/keystore.p12
            storePassword:
              passwordFile: /opt/proxy/server/keystore-password/storePassword
filters: (9)
```

```
- type: EnvelopeEncryption (10)
  config: (11)
    kms: VaultKmsService
    kmsConfig:
      vaultTransitEngineUrl: https://vault.vault.svc.cluster.local:8200/v1/transit
      vaultToken:
        passwordFile: /opt/proxy/server/token.txt
      tls: (12)
        key:
          storeFile: /opt/cert/server.p12
          storePassword:
            passwordFile: /opt/cert/store.password
          keyPassword:
            passwordFile: /opt/cert/key.password
          storeType: PKCS12
    selector: TemplateKekSelector
    selectorConfig:
      template: "${topicName}"
```

1. Enables metrics for the proxy.
2. Virtual cluster configuration.
3. The name of the virtual cluster.
4. The bootstrap address of the target physical Kafka Cluster being proxied.
5. TLS configuration for the connection to the target cluster.
6. The configuration for the cluster network address configuration provider that controls how the virtual cluster is presented to the network.
7. Logging is disabled by default. Enable logging related to network activity (**logNetwork**) and messages (**logFrames**) by setting the logging properties to **true**.
8. TLS encryption for securing connections with the clients.
9. Filter configuration.
10. The type of filter, which is the Record Encryption filter in this example.
11. The configuration specific to the type of filter.
12. The Record Encryption filter requires a connection to Vault. If required, you can also specify the credentials for TLS authentication with Vault, with key names under which TLS certificates are stored.

4.2. CONFIGURING VIRTUAL CLUSTERS

A Kafka cluster is represented by the proxy as a virtual cluster. Clients connect to the virtual cluster rather than the actual cluster. When Streams for Apache Kafka Proxy is deployed, it includes configuration to create virtual clusters.

A virtual cluster has exactly one target cluster, but many virtual clusters can target the same cluster. Each virtual cluster targets a single listener on the target cluster, so multiple listeners on the Kafka side are represented as multiple virtual clusters by the proxy. Clients connect to a virtual cluster using a **bootstrap_servers** address. The virtual cluster has a bootstrap address that maps to each broker in the target cluster. When a client connects to the proxy, communication is proxied to the target broker by rewriting the address. Responses back to clients are rewritten to reflect the appropriate network addresses of the virtual clusters.

You can secure virtual cluster connections from clients and to target clusters.

Streams for Apache Kafka Proxy accepts keys and certificates in PEM (Privacy Enhanced Mail), PKCS #12 (Public-Key Cryptography Standards), or JKS (Java KeyStore) keystore format.

4.2.1. Securing connections from clients

To secure client connections to virtual clusters, configure TLS on the virtual cluster by doing the following:

Obtain a CA (Certificate Authority) certificate for the virtual cluster from a Certificate Authority. When requesting the certificate, ensure it matches the names of the virtual cluster's bootstrap and broker addresses. This might require wildcard certificates and Subject Alternative Names (SANs).

Specify TLS credentials in the virtual cluster configuration using **tls** properties.

Example PKCS #12 configuration

```
virtualClusters:
  my-cluster-proxy:
    tls:
      key:
        storeFile: <path>/server.p12 (1)
        storePassword:
          passwordFile: <path>/store.password (2)
        keyPassword:
          passwordFile: <path>/key.password (3)
        storeType: PKCS12 (4)
      # ...
```

1. PKCS #12 store for the public CA certificate of the virtual cluster.
2. Password to protect the PKCS #12 store.
3. (Optional) Password for the key. If a password is not specified, the keystore's password is used to decrypt the key too.
4. (Optional) Keystore type. If a keystore type is not specified, the default JKS (Java Keystore) type is used.

NOTE

TLS is recommended on Kafka clients and virtual clusters for production configurations.

Example PEM configuration

```
virtualClusters:
  my-cluster-proxy:
    tls:
      key:
        privateKeyFile: <path>/server.key  1
        certificateFile: <path>/server.crt 2
        keyPassword:
          passwordFile: <path>/key.password 3
# …
```

1. Private key of the virtual cluster.
2. Public CA certificate of the virtual cluster.
3. (Optional) Password for the key.

If required, configure the **insecure** property to disable trust and establish insecure connections with any Kafka Cluster, irrespective of certificate validity. However, this option is not recommended for production use.

Example to enable insecure TLS

```
virtualClusters:
  demo:
    targetCluster:
      bootstrap_servers: myprivatecluster:9092
      tls:
        trust:
          insecure: true 1
      #...
# …
```

1. Enables insecure TLS.

4.2.2. Securing connections to target clusters

To secure a virtual cluster connection to a target cluster, configure TLS on the virtual cluster. The target cluster must already be configured to use TLS.

Specify TLS for the virtual cluster configuration using **targetCluster.tls** properties

Use an empty object (**{}**) to inherit trust from the OpenShift platform. This option is suitable if the target cluster is using a TLS certificate signed by a public CA.

Example target cluster configuration for TLS

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093
      tls: {}
      #...
```

If it is using a TLS certificate signed by a private CA, you must add truststore configuration for the target cluster.

Example truststore configuration for a target cluster

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093
      tls:
        trust:
          storeFile: <path>/trust.p12 1
          storePassword:
            passwordFile: <path>/store.password 2
          storeType: PKCS12 3
      #...
```

1. PKCS #12 store for the public CA certificate of the Kafka cluster.
2. Password to access the public Kafka cluster CA certificate.
3. (Optional) Keystore type. If a keystore type is not specified, the default JKS (Java Keystore) type is used.

For mTLS, you can add keystore configuration for the virtual cluster too.

Example keystore and truststore configuration for mTLS

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093:9092
      tls:
        key:
          privateKeyFile: <path>/client.key 1
          certificateFile: <path>/client.crt 2
        trust:
          storeFile: <path>/server.crt
          storeType: PEM
# ...
```

1. Private key of the virtual cluster.
2. Public CA certificate of the virtual cluster.

For the purposes of testing outside of a production environment, you can set the **insecure** property to **true** to turn off TLS so that the Streams for Apache Kafka Proxy can connect to any Kafka cluster.

Example configuration to turn off TLS

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
```

```
    bootstrap_servers: myprivatecluster:9092
    tls:
      trust:
        insecure: true
    #...
```

4.3. CONFIGURING NETWORK ADDRESSES

Virtual cluster configuration requires a network address configuration provider that manages network communication and provides broker address information to clients.

Streams for Apache Kafka Proxy has two built-in providers:

Broker address provider (PortPerBrokerClusterNetworkAddressConfigProvider)

The per-broker network address configuration provider opens one port for a virtual cluster's bootstrap address and one port for each broker in the target Kafka cluster. The ports are maintained dynamically. For example, if a broker is removed from the cluster, the port assigned to it is closed.

SNI routing address provider (SniRoutingClusterNetworkAddressConfigProvider)

The SNI routing provider opens a single port for all virtual clusters or a port for each. For the Kafka cluster, you can open a port for the whole cluster or each broker. The SNI routing provider uses SNI information to determine where to route the traffic.

Example broker address provider configuration

```
clusterNetworkAddressConfigProvider:
  type: PortPerBrokerClusterNetworkAddressConfigProvider
  config:
    bootstrapAddress: mycluster.kafka.com:9192 ①
    brokerAddressPattern: mybroker-$(nodeId).mycluster.kafka.com ②
    brokerStartPort: 9193 ③
    numberOfBrokerPorts: 3 ④
    bindAddress: 192.168.0.1 ⑤
```

① The hostname and port of the bootstrap address used by Kafka clients.

② (Optional) The broker address pattern used to form broker addresses. If not defined, it defaults to the hostname part of the bootstrap address and the port number allocated to the broker. The $(nodeId) token is replaced by the broker's **node.id** (or **broker.id** if **node.id** is not set).

③ (Optional) The starting number for broker port range. Defaults to the port of the bootstrap address plus 1.

④ (Optional) The maximum number of broker ports that are permitted. Defaults to 3.

⑤ (Optional) The bind address used when binding the ports. If undefined, all network interfaces are bound.

The example broker address configuration creates the following broker addresses:

```
mybroker-0.mycluster.kafka.com:9193
mybroker-1.mycluster.kafka.com:9194
mybroker-2.mycluster.kafka.com:9194
```

NOTE

For a configuration with multiple physical clusters, ensure that the **numberOfBrokerPorts** is set to (number of brokers * number of listeners per broker) + number of bootstrap listeners across all clusters. For instance, if there are two physical clusters with 3 nodes each, and each broker has one listener, the configuration requires a value of 8 (comprising 3 ports for broker listeners + 1 port for the bootstrap listener in each cluster).

Example SNI routing address provider configuration

```
clusterNetworkAddressConfigProvider:
  type: SniRoutingClusterNetworkAddressConfigProvider
  config:
    bootstrapAddress: mycluster.kafka.com:9192 (1)
    brokerAddressPattern: mybroker-$(nodeId).mycluster.kafka.com
    bindAddress: 192.168.0.1
```

1 A Single address for all traffic, including bootstrap address and brokers.

In the SNI routing address configuration, the **brokerAddressPattern** specification is mandatory, as it is required to generate routes for each broker.

CHAPTER 5. INTRODUCING METRICS

If you want to introduce metrics to your Streams for Apache Kafka Proxy deployment, you can configure an insecure HTTP and Prometheus endpoint (at **/metrics**).

Add the following to the **ConfigMap** resource that defines the Streams for Apache Kafka Proxy configuration:

Minimal metrics configuration

```
adminHttp:
  endpoints:
    prometheus: {}
```

By default, the HTTP endpoint listens on **0.0.0.0:9190**. You can change the hostname and port as follows:

Example metrics configuration with hostname and port

```
adminHttp:
  host: localhost
  port: 9999
  endpoints:
    prometheus: {}
```

The example files provided with the proxy include a **PodMonitor** resource. If you have enabled monitoring in OpenShift for user-defined projects, you can use a **PodMonitor** resource to ingest the proxy metrics.

Example **PodMonitor** resource configuration

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: proxy
  labels:
    app: proxy
spec:
  selector:
    matchLabels:
      app: proxy
  namespaceSelector:
    matchNames:
      - proxy
  podMetricsEndpoints:
  - path: /metrics
    port: metrics
```

APPENDIX A. USING YOUR SUBSCRIPTION

Streams for Apache Kafka is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing Your Account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a Subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading Zip and Tar Files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **Streams for Apache Kafka for Apache Kafka**entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired Streams for Apache Kafka product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

Revised on 2024-07-03 09:00:33 UTC