Kafka Connect Security

Encryption

If you have enabled SSL encryption in your Apache Kafka® cluster, then you must make sure that Kafka Connect is also configured for security. Click on the section to configure encryption in Kafka Connect:

Encryption with SSL

Authentication

If you have enabled authentication in your Kafka cluster, then you must make sure that Kafka Connect is also configured for security. Click on the section to configure authentication in Kafka Connect:

- Authentication with SSL
- Authentication with SASL/GSSAPI
- Authentication with SASL/SCRAM
- Authentication with SASL/PLAIN

Separate principals

Within the Connect worker configuration, all properties having a prefix of `producer.` and `consumer.` are applied to all source and sink connectors created in the worker. The `admin.` prefix is used for error reporting in sink connectors. The following describes how these prefixes are used:

- The `consumer.` prefix controls consumer behavior for sink connectors.
- The `producer.` prefix controls producer behavior for source connectors.
- Both the `producer.` and `admin.` prefixes control producer and client behavior for sink connector error reporting.

You can override these properties for individual connectors using the `producer.override.`, `consumer.override.`, and `admin.override.` prefixes. This includes overriding the worker service principal configuration to create separate service principals for each connector. Overrides are disabled by default. They are enabled using the `connector.client.config.override.policy` worker property. This property sets the per-connector overrides the worker permits. The out-of-the-box (OOTB) options for the override policy are:

- **`connector.client.config.override.policy=None`**
  Default. Does not allow any configuration overrides.

- **`connector.client.config.override.policy=Principal`**
  Allows overrides for the `security.protocol`, `sasl.jaas.config`, and `sasl.mechanism` configuration properties, using the `producer.override.`, `consumer.override`, and `admin.override` prefixes.

- **`connector.client.config.override.policy=All`**
  Allows overrides for all configuration properties using the `producer.override.`, `consumer.override`, and `admin.override` pre

Expand Content

Tip

You can write your own implementation of the ConnectorClientConfigOverridePolicy class if any of the OOTB policies don't meet your needs.

If your Kafka broker supports client authentication over SSL, you can configure a separate principal for the worker and the connectors. In this case, you need to generate a separate certificate for each of them and install them in separate keystores.

The key Connect configuration differences are as follows, notice the unique password, keystore location, and keystore password:

```
# Authentication settings for Connect workers
ssl.keystore.location=/var/private/ssl/kafka.worker.keystore.jks
ssl.keystore.password=worker1234
ssl.key.password=worker1234
```

Connect workers manage the producers used by source connectors and the consumers used by sink connectors. So, for the connectors to leverage security, you also have to override the default producer/consumer configuration that the worker uses.

```
# Authentication settings for Connect producers used with source connectors
producer.ssl.keystore.location=/var/private/ssl/kafka.source.keystore.jks
producer.ssl.keystore.password=connector1234
producer.ssl.key.password=connector1234

# Authentication settings for Connect consumers used with sink connectors
consumer.ssl.keystore.location=/var/private/ssl/kafka.sink.keystore.jks
consumer.ssl.keystore.password=connector1234
consumer.ssl.key.password=connector1234
```

ACL Considerations

Using separate principals for the connectors allows you to define access control lists (ACLs) with finer granularity. For example, you can use this capability to prevent the connectors themselves from writing to any of internal topics used by the Connect cluster. Additionally, you can use different keystores for source and sink connectors and enable scenarios where source connectors have only write access to a topic but sink connectors have only read access to the same topic.

Note that if you are using SASL for authentication, you must use the same principal for workers and connectors as only a single JAAS is currently supported on the client side at this time as described here.

Worker ACL Requirements

Workers must be given access to the common group that all workers in a cluster join, and to all the internal topics required by Connect. Read and write access to the internal topics are always required, but create access is only required if the internal topics don't yet exist and Kafka Connect is to automatically create them. The table below shows each required permission and the relevant configuration setting used to define its value.

Operation(s)	Resource	Configuration Item
Create	Cluster	`config.storage.topic`
Create	Cluster	`config.storage.replication.factor`
Create	Cluster	`offset.storage.topic`
Create	Cluster	`offset.storage.partitions`
Create	Cluster	`offset.storage.replication.factor`
Create	Cluster	`status.storage.topic`
Create	Cluster	`status.storage.partitions`
Create	Cluster	`status.storage.replication.factor`
Read/Write	Topic	`config.storage.topic`
Read/Write	Topic	`offsets.storage.topic`
Read/Write	Topic	`status.storage.topic`
Read	Group	`group.id`

See Adding ACLs for documentation on creating new ACLs from the command line.

Connector ACL Requirements

Source connectors must be given `WRITE` permission to any topics that they need to write to. Similarly, sink connectors neec `READ` permission to any topics they will read from. They also need Group `READ` permission since sink tasks depend on consumer groups internally. Connect defines the consumer `group.id` conventionally for each sink connector as `connect-{name}` where `{name}` is substituted by the name of the connector. For example, if your sink connector is named "hdfs-logs" and it reads from a topic named "logs," then you could add an ACL with the following command:

```
bin/kafka-acls --authorizer-properties zookeeper.connect=<Zk host:port> \
 --add --allow-principal User:<Sink Connector Principal> \
 --consumer --topic logs --group connect-hdfs-logs
```

Role-Based Access Control

If your organization has enabledRole-Based Access Control (RBAC), you need to review your user principal, RBAC role, and RBAC role permissions before performing any Kafka Connect or Apache Kafka® cluster operations. Refer to Kafka Connect and RBACto learn more about how RBAC is configured for Kafka Connect to protect your Kafka cluster.

Externalizing Secrets

You can use a `ConfigProvider` implementation to prevent secrets from appearing in cleartext for Connector configurations on the filesystem (standalone mode) or in internal topics (distributed mode). You can specify variables in the configuration that are replaced at runtime with secrets from an external source. A reference implementation of `ConfigProvider` is provided with Kafka 2.0 called `FileConfigProvider` that

allows variable references to be replaced with values from a properties file. However, the reference `FileConfigProvider` implementation still shows secrets in cleartext in the properties file that is managed by `FileConfigProvider`.

Here is an example of how to use `FileConfigProvider` in the worker configuration:

```
# Additional properties for the worker configuration
config.providers=file    # multiple comma-separated provider types can be specified here
config.providers.file.class=org.apache.kafka.common.config.provider.FileConfigProvider
# Other ConfigProvider implementations might require parameters passed in to configure() as follows:
#config.providers.other-provider.param.foo=value1
#config.providers.other-provider.param.bar=value2
```

Then you can reference the configuration variables in the connector configuration as follows:

```
# Additional properties for the connector configuration
# Variable references are of the form ${provider:[path:]key} where the path is optional,
# depending on the ConfigProvider implementation.
my.secret=${file:/var/run/secret.properties:secret-key}
```

Note that the variable reference is of the form `${provider:[path:]key}`, where the `path:` is optional, depending on the `ConfigProvider` implementation.

All `ConfigProvider` implementations are discovered using the standard Java `ServiceLoader` mechanism. To use a custom implementation of `ConfigProvider`, package a JAR file containing the fully qualified name of a class that implements the `ConfigProvider` interface. Name the JAR file `META-INF/services/org.apache.kafka.common.config.ConfigProvider`. The JAR file should also include all runtime dependencies except those provided by the Connect framework.

To install the custom `ConfigProvider` implementation, add a new subdirectory containing the JAR file to the directory that is on Connect's `plugin.path`, and (re)start the Connect workers. When the Connect worker starts up it instantiates al `ConfigProvider` implementations specified in the worker configuration. All properties prefixed with `config.providers.[provider].param.` are passed to the `configure()` method of the `ConfigProvider`. When the Connect worker shuts down, it calls the `close()` method of the `ConfigProvider`.

Configuring the Connect REST API for HTTP or HTTPS

By default you can make REST API calls over HTTP with Kafka Connect. You can also configure Connect to allow either HTTP or HTTPS, or both.

The listeners configuration parameter determines the protocol used by Kafka Connect. This configuration should contain a list of listeners in this format: `protocol://host:port,protocol2://host2:port2`. For example:

```
listeners=http://localhost:8080,https://localhost:8443
```

By default, if no listeners are specified, the REST server runs on port 8083 using the HTTP protocol. When using HTTPS, the configuration must include the SSL configuration. By default, it will use the `ssl.*` settings. You can use a different configuration for the REST API than for the Kafka brokers, by using the `listeners.https` prefix. If you use the `listeners.https` prefix, the `ssl.*` options are ignored.

You can use the following fields to configure HTTPS for the REST API:

- `ssl.keystore.location`
- `ssl.keystore.password`
- `ssl.keystore.type`
- `ssl.key.password`
- `ssl.truststore.location`
- `ssl.truststore.password`
- `ssl.truststore.type`
- `ssl.enabled.protocols`

- `ssl.provider`
- `ssl.protocol`
- `ssl.cipher.suites`
- `ssl.keymanager.algorithm`
- `ssl.secure.random.implementation`
- `ssl.trustmanager.algorithm`
- `ssl.endpoint.identification.algorithm`
- `ssl.client.auth`

For more information, see Distributed Worker Configuration.

The REST API is used to monitor and manage Kafka Connect and for Kafka Connect cross-cluster communication. Requests that are received on the follower nodes REST API are forwarded on to the leader node REST API. If the URI host is different from the URI that it listens on, you can change the URI with the `rest.advertised.host.name`, `rest.advertised.port` and `rest.advertised.listener` configuration options. This URI will be used by the follower nodes to connect with the leader.

When using both HTTP and HTTPS listeners, you can use the `rest.advertised.listener` option to define which listener is used for the cross-cluster communication. When using HTTPS for communication between nodes, the same `ssl.*` or `listeners.https` options are used to configure the HTTPS client.

These are the currently supported REST API endpoints:

- `GET /connectors` - Return a list of active connectors.
- `POST /connectors` - Create a new connector; the request body should be a JSON object containing a string name field and an object config field with the connector configuration parameters.
- `GET /connectors/{name}` - Get information about a specific connector.
- `GET /connectors/{name}/config` - Get the configuration parameters for a specific connector.
- `PUT /connectors/{name}/config` - Update the configuration parameters for a specific connector.
- `GET /connectors/{name}/status` - Get the current status of the connector, including whether it is running, failed, or paused; which worker it is assigned to, error information if it has failed, and the state of all its tasks.
- `GET /connectors/{name}/tasks` - Get a list of tasks currently running for a connector.
- `GET /connectors/{name}/tasks/{taskid}/status` - Get current status of the task, including if it is running, failed, or paused; which worker it is assigned to, and error information if it has failed.
- `PUT /connectors/{name}/pause` - Pause the connector and its tasks, which stops message processing until the connector is resumed.
- `PUT /connectors/{name}/resume` - Resumes a paused connector or does nothing if the connector is not paused.
- `POST /connectors/{name}/restart` - Restart a connector. This is typically used because it has failed.
- `POST /connectors/{name}/tasks/{taskId}/restart` - Restart an individual task. This is typically used because it has failed.
- `DELETE /connectors/{name}` - Delete a connector, halting all tasks and deleting its configuration.

You can also use Kafka Connect REST API to get information about connector plugins:

- `GET /connector-plugins` - Returns a list of connector plugins installed in the Kafka Connect cluster. The API only checks for connectors on the worker that handles the request. This means you might see inconsistent results, especially during a rolling upgrade if you add new connector JARs.
- `PUT /connector-plugins/{connector-type}/config/validate` - Validate the provided configuration values against the configuration definition. This API performs per config validation, returns suggested values and error messages during validation.

For more information, see Kafka Connect REST Interface

For demo of Kafka Connect configured with an HTTPS endpoint, and Confluent Control Center connecting to it, check out Confluent Platform demo.

Please report any inaccuracies on this page or suggest an edit.

6 Votes

Last updated on Oct 17, 2019.