Red Hat OpenShift Service on AWS 4

Networking

Configuring Red Hat OpenShift Service on AWS networking

Last Updated: 2023-05-26

Red Hat OpenShift Service on AWS 4 Networking

Configuring Red Hat OpenShift Service on AWS networking

Legal Notice

Abstract

This document provides information about networking for Red Hat OpenShift Service on AWS (ROSA) clusters.

Table of Contents

CHAPTER 1. INGRESS OPERATOR IN RED HAT OPENSHIFT SERVICE ON AWS

1.1. RED HAT OPENSHIFT SERVICE ON AWS INGRESS OPERATOR

When you create your Red Hat OpenShift Service on AWS cluster, pods and services running on the cluster are each allocated their own IP addresses. The IP addresses are accessible to other pods and services running nearby but are not accessible to outside clients. The Ingress Operator implements the **IngressController** API and is the component responsible for enabling external access to Red Hat OpenShift Service on AWS cluster services.

The Ingress Operator makes it possible for external clients to access your service by deploying and managing one or more HAProxy-based Ingress Controllers to handle routing. Red Hat Site Reliability Engineers (SRE) manage the Ingress Operator for Red Hat OpenShift Service on AWS clusters. While you cannot alter the settings for the Ingress Operator, you may view the default Ingress Controller configurations, status, and logs as well as the Ingress Operator status.

1.2. VIEW THE DEFAULT INGRESS CONTROLLER

The Ingress Operator is a core feature of Red Hat OpenShift Service on AWS and is enabled out of the box.

Every new Red Hat OpenShift Service on AWS installation has an **ingresscontroller** named default. It can be supplemented with additional Ingress Controllers. If the default **ingresscontroller** is deleted, the Ingress Operator will automatically recreate it within a minute.

Procedure

- View the default Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/default
```

1.3. VIEW INGRESS OPERATOR STATUS

You can view and inspect the status of your Ingress Operator.

Procedure

- View your Ingress Operator status:

```
$ oc describe clusteroperators/ingress
```

1.4. VIEW INGRESS CONTROLLER LOGS

You can view your Ingress Controller logs.

Procedure

- View your Ingress Controller logs:

```
$ oc logs --namespace=openshift-ingress-operator deployments/ingress-operator -c
<container_name>
```

1.5. VIEW INGRESS CONTROLLER STATUS

Your can view the status of a particular Ingress Controller.

Procedure

- View the status of an Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/<name>
```

1.6. RED HAT OPENSHIFT SERVICE ON AWS INGRESS OPERATOR CONFIGURATIONS

The following table details the components of the Ingress Operator and if Red Hat Site Reliability Engineers (SRE) maintains this component on Red Hat OpenShift Service on AWS clusters.

Table 1.1. Ingress Operator Responsibility Chart

Ingress component	Managed by	Default configuration?
Scaling Ingress Controller	SRE	Yes
Ingress Operator thread count	SRE	Yes
Ingress Controller access logging	SRE	Yes
Ingress Controller sharding	SRE	Yes
Ingress Controller route admission policy	SRE	Yes
Ingress Controller wildcard routes	SRE	Yes
Ingress Controller X-Forwarded headers	SRE	Yes
Ingress Controller route compression	SRE	Yes

CHAPTER 2. OPENSHIFT SDN DEFAULT CNI NETWORK PROVIDER

2.1. ENABLING MULTICAST FOR A PROJECT

2.1.1. About multicast

With IP multicast, data is broadcast to many IP addresses simultaneously.

IMPORTANT

At this time, multicast is best used for low-bandwidth coordination or service discovery and not a high-bandwidth solution.

Multicast traffic between Red Hat OpenShift Service on AWS pods is disabled by default. If you are using the OpenShift SDN network plugin, you can enable multicast on a per-project basis.

When using the OpenShift SDN network plugin in **networkpolicy** isolation mode:

- Multicast packets sent by a pod will be delivered to all other pods in the project, regardless of **NetworkPolicy** objects. Pods might be able to communicate over multicast even when they cannot communicate over unicast.
- Multicast packets sent by a pod in one project will never be delivered to pods in any other project, even if there are **NetworkPolicy** objects that allow communication between the projects.

When using the OpenShift SDN network plugin in **multitenant** isolation mode:

- Multicast packets sent by a pod will be delivered to all other pods in the project.
- Multicast packets sent by a pod in one project will be delivered to pods in other projects only if each project is joined together and multicast is enabled in each joined project.

2.1.2. Enabling multicast between pods

You can enable multicast between pods for your project.

Prerequisites

- Install the OpenShift CLI (**oc**).
- You must log in to the cluster with a user that has the **cluster-admin** or the **dedicated-admin** role.

Procedure

- Run the following command to enable multicast for a project. Replace **<namespace>** with the namespace for the project you want to enable multicast for.

```
$ oc annotate netnamespace <namespace> \
    netnamespace.network.openshift.io/multicast-enabled=true
```

Verification

To verify that multicast is enabled for a project, complete the following procedure:

1. Change your current project to the project that you enabled multicast for. Replace **<project>** with the project name.

```
$ oc project <project>
```

2. Create a pod to act as a multicast receiver:

```
$ cat <<EOF| oc create -f -
apiVersion: v1
kind: Pod
metadata:
  name: mlistener
  labels:
    app: multicast-verify
spec:
  containers:
    - name: mlistener
      image: registry.access.redhat.com/ubi9
      command: ["/bin/sh", "-c"]
      args:
        ["dnf -y install socat hostname && sleep inf"]
      ports:
        - containerPort: 30102
          name: mlistener
          protocol: UDP
EOF
```

3. Create a pod to act as a multicast sender:

```
$ cat <<EOF| oc create -f -
apiVersion: v1
kind: Pod
metadata:
  name: msender
  labels:
    app: multicast-verify
spec:
  containers:
    - name: msender
      image: registry.access.redhat.com/ubi9
      command: ["/bin/sh", "-c"]
      args:
        ["dnf -y install socat && sleep inf"]
EOF
```

4. In a new terminal window or tab, start the multicast listener.

    a. Get the IP address for the Pod:

```
$ POD_IP=$(oc get pods mlistener -o jsonpath='{.status.podIP}')
```

b. Start the multicast listener by entering the following command:

```
$ oc exec mlistener -i -t -- \
    socat UDP4-RECVFROM:30102,ip-add-membership=224.1.0.1:$POD_IP,fork
EXEC:hostname
```

5. Start the multicast transmitter.

a. Get the pod network IP address range:

```
$ CIDR=$(oc get Network.config.openshift.io cluster \
    -o jsonpath='{.status.clusterNetwork[0].cidr}')
```

b. To send a multicast message, enter the following command:

```
$ oc exec msender -i -t -- \
    /bin/bash -c "echo | socat STDIO UDP4-
DATAGRAM:224.1.0.1:30102,range=$CIDR,ip-multicast-ttl=64"
```

If multicast is working, the previous command returns the following output:

```
mlistener
```

CHAPTER 3. NETWORK VERIFICATION FOR ROSA CLUSTERS

Network verification checks run automatically when you deploy a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC) or create an additional machine pool with a subnet that is new to your cluster. The checks validate your network configuration and highlight errors, enabling you to resolve configuration issues prior to deployment.

You can also run the network verification checks manually to validate the configuration for an existing cluster.

3.1. UNDERSTANDING NETWORK VERIFICATION FOR ROSA CLUSTERS

When you deploy a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC) or create an additional machine pool with a subnet that is new to your cluster, network verification runs automatically. This helps you identify and resolve configuration issues prior to deployment.

When you prepare to install your cluster by using Red Hat OpenShift Cluster Manager, the automatic checks run after you input a subnet into a subnet ID field on the **Virtual Private Cloud (VPC) subnet settings** page. If you create your cluster by using the ROSA CLI (**rosa**) with the interactive mode, the checks run after you provide the required VPC network information. If you use the CLI without the interactive mode, the checks begin immediately prior to the cluster creation.

When you add a machine pool with a subnet that is new to your cluster, the automatic network verification checks the subnet to ensure that network connectivity is available before the machine pool is provisioned.

After automatic network verification completes, a record is sent to the service log. The record provides the results of the verification check, including any network configuration errors. You can resolve the identified issues before a deployment and the deployment has a greater chance of success.

You can also run the network verification manually for an existing cluster. This enables you to verify the network configuration for your cluster after making configuration changes. For steps to run the network verification checks manually, see *Running the network verification manually* .

3.2. SCOPE OF THE NETWORK VERIFICATION CHECKS

The network verification includes checks for each of the following requirements:

- The parent Virtual Private Cloud (VPC) exists.
- All specified subnets belong to the VPC.
- The VPC has **enableDnsSupport** enabled.
- The VPC has **enableDnsHostnames** enabled.
- Egress is available to the required domain and port combinations that are specified in the AWS firewall prerequisites section.

3.3. AUTOMATIC NETWORK VERIFICATION BYPASSING

You can bypass the automatic network verification if you want to deploy a Red Hat OpenShift Service on AWS (ROSA) cluster with known network configuration issues into an existing Virtual Private Cloud (VPC).

If you bypass the network verification when you create a cluster, the cluster has a limited support status. After installation, you can resolve the issues and then manually run the network verification. The limited support status is removed after the verification succeeds.

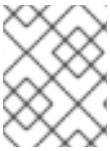
Bypassing automatic network verification by using OpenShift Cluster Manager

When you install a cluster into an existing VPC by using Red Hat OpenShift Cluster Manager, you can bypass the automatic verification by selecting **Bypass network verification** on the **Virtual Private Cloud (VPC) subnet settings** page.

Bypassing automatic network verification by using the ROSA CLI

When you install a cluster into an existing VPC by using the **rosa create cluster** command, you can bypass the automatic verification by including the **--bypass-network-verify --force** arguments. The following example bypasses the network verification before creating a cluster:

```
$ rosa create cluster --cluster-name mycluster \
                      --subnet-ids subnet-03146b9b52b6024cb,subnet-03146b9b52b2034cc \
                      --bypass-network-verify --force
```



NOTE

Alternatively, you can specify the **--interactive** argument and select the option in the interactive prompts to bypass the network verification checks.

3.4. RUNNING THE NETWORK VERIFICATION MANUALLY

After installing a Red Hat OpenShift Service on AWS (ROSA) cluster, you can run the network verification checks manually by using Red Hat OpenShift Cluster Manager or the ROSA CLI (**rosa**).

Running the network verification manually using OpenShift Cluster Manager

You can manually run the network verification checks for an existing Red Hat OpenShift Service on AWS (ROSA) cluster by using Red Hat OpenShift Cluster Manager.

Prerequisites

- You have an existing ROSA cluster.
- You are the cluster owner or you have the cluster editor role.

Procedure

1. Navigate to OpenShift Cluster Manager Hybrid Cloud Console and select your cluster.
2. Select **Verify networking** from the **Actions** drop-down menu.

Running the network verification manually using the CLI

You can manually run the network verification checks for an existing Red Hat OpenShift Service on AWS (ROSA) cluster by using the ROSA CLI (**rosa**).

When you run the network verification, you can specify a set of VPC subnet IDs or a cluster name. If you are using a proxy service, you can specify a proxy URL.

Prerequisites

- You have installed and configured the latest ROSA CLI (**rosa**) on your installation host.
- You have an existing ROSA cluster.
- You are the cluster owner or you have the cluster editor role.

Procedure

- Verify the network configuration by using one of the following methods:
  - Verify the network configuration by specifying the cluster name. The subnet IDs are automatically detected:

```
$ rosa verify network -c <cluster_name> 1
```

1 Replace **<cluster_name>** with the name of your cluster.

Example output

```
I: ✓ Network verification successful
```

TIP

To output the full list of verification tests, you can include the **--debug** argument when you run **rosa verify network**.

  - Verify the network configuration by specifying the VPC subnets IDs:

```
$ rosa verify network --subnet-ids subnet-03146b9b52b6024cb,subnet-
03146b9b52b2034cc
```

Example output

```
E: Validating Subnet subnet-03146b9b52b6024cb egress
E: X Egress failed to https://events.pagerduty.com
```

  - Verify the network configuration by specifying the VPC subnets IDs and a proxy URL:

```
$ rosa verify network --subnet-ids subnet-03146b9b52b6024cb,subnet-
03146b9b52b2034cc \
                --additional-trust-bundle-file /path/to/ca.cert \
                --https-proxy <proxy_url> 1
```

1 Replace **<proxy_url>** with the URL of your proxy service, for example **https://10.10.0.1**.

Example output

```
I: Using proxy configuration
I: Subnet IDs detected:  subnet-03146b9b52b6024cb,subnet-03146b9b52b2034cc

E: Validating Subnet subnet-03146b9b52b6024cb egress
E: X Egress failed to https://events.pagerduty.com
```

CHAPTER 4. CONFIGURING A CLUSTER-WIDE PROXY

If you are using an existing Virtual Private Cloud (VPC), you can configure a cluster-wide proxy during a Red Hat OpenShift Service on AWS (ROSA) cluster installation or after the cluster is installed. When you enable a proxy, the core cluster components are denied direct access to the internet, but the proxy does not affect user workloads.

NOTE

Only cluster system egress traffic is proxied, including calls to the cloud provider API.

If you use a cluster-wide proxy, you are responsible for maintaining the availability of the proxy to the cluster. If the proxy becomes unavailable, then it might impact the health and supportability of the cluster.

4.1. PREREQUISITES FOR CONFIGURING A CLUSTER-WIDE PROXY

To configure a cluster-wide proxy, you must meet the following requirements. These requirements are valid when you configure a proxy during installation or post-installation.

General requirements

- You are the cluster owner.
- Your account has sufficient privileges.
- You have an existing Virtual Private Cloud (VPC) for your cluster.
- The proxy can access the VPC for the cluster and the private subnets of the VPC. The proxy is also accessible from the VPC for the cluster and from the private subnets of the VPC.
- You have added the **ec2.<aws_region>.amazonaws.com**, **elasticloadbalancing.<aws_region>.amazonaws.com**, and **s3.<aws_region>.amazonaws.com** endpoints to your VPC endpoint. These endpoints are required to complete requests from the nodes to the AWS EC2 API. Because the proxy works at the container level and not at the node level, you must route these requests to the AWS EC2 API through the AWS private network. Adding the public IP address of the EC2 API to your allowlist in your proxy server is not enough.

Network requirements

- If your proxy re-encyrpts egress traffic, you must create exclusions to the domain and port combinations. The following table offers guidance into these exceptions.
    - Add the following OpenShift URLs to your allowlist for re-encryption.

Address	Protocol/Port	Function
observatorium-mst.api.openshift.com	https/443	Required. Used for Managed OpenShift-specific telemetry.

Address	Protocol/Port	Function
sso.redhat.com	https/443	The https://cloud.redhat.com/openshift site uses authentication from sso.redhat.com to download the cluster pull secret and use Red Hat SaaS solutions to facilitate monitoring of your subscriptions, cluster inventory, and chargeback reporting.

- Add the following site reliability engineering (SRE) and management URLs to your allowlist for re-encryption.

Address	Protocol/Port	Function
***.osdsecuritylogs.splunkcloud.com** OR **inputs1.osdsecuritylogs.splunkcloud.cominputs2.osdsecuritylogs.splunkcloud.cominputs4.osdsecuritylogs.splunkcloud.cominputs5.osdsecuritylogs.splunkcloud.cominputs6.osdsecuritylogs.splunkcloud.cominputs7.osdsecuritylogs.splunkcloud.cominputs8.osdsecuritylogs.splunkcloud.cominputs9.osdsecuritylogs.splunkcloud.cominputs10.osdsecuritylogs.splunkcloud.cominputs11.osdsecuritylogs.splunkcloud.cominputs12.osdsecuritylogs.splunkcloud.cominputs13.osdsecuritylogs.splunkcloud.cominputs14.osdsecuritylogs.splunkcloud.cominputs15.osdsecuritylogs.splunkcloud.com**	tcp/9997	Used by the splunk-forwarder-operator as a log forwarding endpoint to be used by Red Hat SRE for log-based alerting.
http-inputs-osdsecuritylogs.splunkcloud.com	https/443	Used by the splunk-forwarder-operator as a log forwarding endpoint to be used by Red Hat SRE for log-based alerting.

IMPORTANT

The use of a proxy server to perform TLS re-encryption is currently not supported if the server is acting as a transparent forward proxy where it is not configured on-cluster via the **--http-proxy** or **--https-proxy** arguments.

A transparent forward proxy intercepts the cluster traffic, but it is not actually configured on the cluster itself.

Additional Resources

- For the installation prerequisites for ROSA clusters that use the AWS Security Token Service (STS), see AWS prerequisites for ROSA with STS.
- For the installation prerequisites for ROSA clusters that do not use STS, see AWS prerequisites for ROSA.

4.2. RESPONSIBILITIES FOR ADDITIONAL TRUST BUNDLES

If you supply an additional trust bundle, you are responsible for the following requirements:

- Ensuring that the contents of the additional trust bundle are valid
- Ensuring that the certificates, including intermediary certificates, contained in the additional trust bundle have not expired
- Tracking the expiry and performing any necessary renewals for certificates contained in the additional trust bundle
- Updating the cluster configuration with the updated additional trust bundle

4.3. CONFIGURING A PROXY DURING INSTALLATION

You can configure an HTTP or HTTPS proxy when you install a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC). You can configure the proxy during installation by using Red Hat OpenShift Cluster Manager or the ROSA CLI (**rosa**).

4.3.1. Configuring a proxy during installation using OpenShift Cluster Manager

If you are installing a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC), you can use Red Hat OpenShift Cluster Manager to enable a cluster-wide HTTP or HTTPS proxy during installation.

Prior to the installation, you must verify that the proxy is accessible from the VPC that the cluster is being installed into. The proxy must also be accessible from the private subnets of the VPC.

For detailed steps to configure a cluster-wide proxy during installation by using OpenShift Cluster Manager, see *Creating a cluster with customizations by using OpenShift Cluster Manager* .

4.3.2. Configuring a proxy during installation using the CLI

If you are installing a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC), you can use the ROSA CLI (**rosa**) to enable a cluster-wide HTTP or HTTPS proxy during installation.

The following procedure provides details about the ROSA CLI (**rosa**) arguments that are used to configure a cluster-wide proxy during installation. For general installation steps using the ROSA CLI, see *Creating a cluster with customizations using the CLI* .

Prerequisites

- You have verified that the proxy is accessible from the VPC that the cluster is being installed into. The proxy must also be accessible from the private subnets of the VPC.

Procedure

- Specify a proxy configuration when you create your cluster:

```
$ rosa create cluster \
 <other_arguments_here> \
 --additional-trust-bundle-file <path_to_ca_bundle_file> \ 1 2 3
 --http-proxy http://<username>:<password>@<ip>:<port> \ 4 5
 --https-proxy https://<username>:<password>@<ip>:<port> \ 6 7
 --no-proxy example.com 8
```

1 4 6 The **additional-trust-bundle-file**, **http-proxy**, and **https-proxy** arguments are all optional.

2 If you use the **additional-trust-bundle-file** argument without an **http-proxy** or **https-proxy** argument, the trust bundle is added to the trust store and used to verify cluster system egress traffic. In that scenario, the bundle is not configured to be used with a proxy.

3 The **additional-trust-bundle-file** argument is a file path pointing to a bundle of PEM-encoded X.509 certificates, which are all concatenated together. The **additionalTrustBundle** parameter is required unless the identity certificate of the proxy is signed by an authority from the RHCOS trust bundle. If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional CAs, you must provide the MITM CA certificate.

5 7 The **http-proxy** and **https-proxy** arguments must point to a valid URL.

8 A comma-separated list of destination domain names, IP addresses, or network CIDRs to exclude proxying.

Preface a domain with **.** to match subdomains only. For example, **.y.com** matches **x.y.com**, but not **y.com**. Use ***** to bypass proxy for all destinations. If you scale up workers that are not included in the network defined by the **networking.machineNetwork[].cidr** field from the installation configuration, you must add them to this list to prevent connection issues.

This field is ignored if neither the **httpProxy** or **httpsProxy** fields are set.

Additional Resources

- Creating a cluster with customizations by using OpenShift Cluster Manager
- Creating a cluster with customizations using the CLI

4.4. CONFIGURING A PROXY AFTER INSTALLATION

You can configure an HTTP or HTTPS proxy after you install a Red Hat OpenShift Service on AWS (ROSA) cluster into an existing Virtual Private Cloud (VPC). You can configure the proxy after installation by using Red Hat OpenShift Cluster Manager or the ROSA CLI (**rosa**).

4.4.1. Configuring a proxy after installation using OpenShift Cluster Manager

You can use Red Hat OpenShift Cluster Manager to add a cluster-wide proxy configuration to an existing Red Hat OpenShift Service on AWS cluster in a Virtual Private Cloud (VPC).

You can also use OpenShift Cluster Manager to update an existing cluster-wide proxy configuration. For example, you might need to update the network address for the proxy or replace the additional trust bundle if any of the certificate authorities for the proxy expire.

IMPORTANT

The cluster applies the proxy configuration to the control plane and compute nodes. While applying the configuration, each cluster node is temporarily placed in an unschedulable state and drained of its workloads. Each node is restarted as part of the process.

Prerequisites

- You have an Red Hat OpenShift Service on AWS cluster .
- Your cluster is deployed in a VPC.

Procedure

1. Navigate to OpenShift Cluster Manager Hybrid Cloud Console and select your cluster.
2. Under the **Virtual Private Cloud (VPC)**section on the **Networking** page, click **Edit cluster-wide proxy**.
3. On the **Edit cluster-wide proxy**page, provide your proxy configuration details:
   a. Enter a value in at least one of the following fields:
      - Specify a valid **HTTP proxy URL**
      - Specify a valid **HTTPS proxy URL**
      - In the **Additional trust bundle** field, provide a PEM encoded X.509 certificate bundle. If you are replacing an existing trust bundle file, select **Replace file** to view the field. The bundle is added to the trusted certificate store for the cluster nodes. An additional trust bundle file is required unless the identity certificate for the proxy is signed by an authority from the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle.
        If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional certificate authorities (CAs), you must provide the MITM CA certificate.

NOTE

If you upload an additional trust bundle file without specifying an HTTP or HTTPS proxy URL, the bundle is set on the cluster but is not configured to be used with the proxy.

b. Click **Confirm**.

Verification

- Under the **Virtual Private Cloud (VPC)**section on the **Networking** page, verify that the proxy configuration for your cluster is as expected.

4.4.2. Configuring a proxy after installation using the CLI

You can use the Red Hat OpenShift Service on AWS (ROSA) CLI (**rosa**) to add a cluster-wide proxy configuration to an existing ROSA cluster in a Virtual Private Cloud (VPC).

You can also use **rosa** to update an existing cluster-wide proxy configuration. For example, you might need to update the network address for the proxy or replace the additional trust bundle if any of the certificate authorities for the proxy expire.

IMPORTANT

The cluster applies the proxy configuration to the control plane and compute nodes. While applying the configuration, each cluster node is temporarily placed in an unschedulable state and drained of its workloads. Each node is restarted as part of the process.

Prerequisites

- You have installed and configured the latest ROSA (**rosa**) and OpenShift (**oc**) CLIs on your installation host.
- You have a ROSA cluster that is deployed in a VPC.

Procedure

- Edit the cluster configuration to add or update the cluster-wide proxy details:

```
$ rosa edit cluster \
 --cluster $CLUSTER_NAME \
 --additional-trust-bundle-file <path_to_ca_bundle_file> \ 1 2 3
 --http-proxy http://<username>:<password>@<ip>:<port> \ 4 5
 --https-proxy https://<username>:<password>@<ip>:<port> \ 6 7
  --no-proxy example.com 8
```

1 4 6 The **additional-trust-bundle-file**, **http-proxy**, and **https-proxy** arguments are all optional.

2 If you use the **additional-trust-bundle-file** argument without an **http-proxy** or **https-proxy** argument, the trust bundle is added to the trust store and used to verify cluster system egress traffic. In that scenario, the bundle is not configured to be used with a proxy.

3 The **additional-trust-bundle-file** argument is a file path pointing to a bundle of PEM-encoded X.509 certificates, which are all concatenated together. The **additionalTrustBundle** parameter is required unless the identity certificate of the proxy is signed by an authority from the RHCOS trust bundle. If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional CAs, you must provide the MITM CA certificate.

NOTE

You should not attempt to change the proxy or additional trust bundle configuration on the cluster directly. These changes must be applied by using the ROSA CLI (**rosa**) or Red Hat OpenShift Cluster Manager. Any changes that are made directly to the cluster will be reverted automatically.

5 7 The **http-proxy** and **https-proxy** arguments must point to a valid URL.

8 A comma-separated list of destination domain names, IP addresses, or network CIDRs to exclude proxying.

Preface a domain with **.** to match subdomains only. For example, **.y.com** matches **x.y.com**, but not **y.com**. Use ***** to bypass proxy for all destinations. If you scale up workers that are not included in the network defined by the **networking.machineNetwork[].cidr** field from the installation configuration, you must add them to this list to prevent connection issues.

This field is ignored if neither the **httpProxy** or **httpsProxy** fields are set.

Verification

1. List the status of the machine config pools and verify that they are updated:

```
$ oc get machineconfigpools
```

Example output

```
NAME     CONFIG                                             UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master   rendered-master-d9a03f612a432095dcde6dcf44597d90   True      False      False
3            3                 3                   0                    31h
worker   rendered-worker-f6827a4efe21e155c25c21b43c46f65e   True      False      False
6            6                 6                   0                    31h
```

2. Display the proxy configuration for your cluster and verify that the details are as expected:

```
$ oc get proxy cluster -o yaml
```

Example output

```
apiVersion: config.openshift.io/v1
kind: Proxy
spec:
  httpProxy: http://proxy.host.domain:<port>
  httpsProxy: https://proxy.host.domain:<port>
  <...more...>
status:
  httpProxy: http://proxy.host.domain:<port>
  httpsProxy: https://proxy.host.domain:<port>
  <...more...>
```

4.5. REMOVING A CLUSTER-WIDE PROXY

You can remove your cluster-wide proxy by using the **rosa** CLI tool. After removing the cluster, you should also remove any trust bundles that are added to the cluster.

4.5.1. Removing the cluster-wide proxy using CLI

You must use the **rosa** CLI to remove the proxy's address from your cluster.

Prerequisites

- You must have cluster administrator privileges.
- You have installed the Red Hat OpenShift Service on AWS **rosa** CLI tool.

Procedure

- Use the **rosa edit** command to modify the proxy. You must pass empty strings to the **--http-proxy** and **--https-proxy** arguments to clear the proxy from the cluster:

```
$ rosa edit cluster -c <cluster_name> --http-proxy "" --https-proxy ""
```

NOTE

While your proxy might only use one of the proxy arguments, the empty fields are ignored, so passing empty strings to both the **--http-proxy** and **--https-proxy** arguments do not cause any issues.

Example Output

```
I: Updated cluster <cluster_name>
```

Verification

- You can verify that the proxy has been removed from the cluster by using the **rosa describe** command:

```
$ rosa describe cluster -c <cluster_name>
```

Before removal, the proxy IP displays in a proxy section:

```
Name:                  <cluster_name>
ID:                    <cluster_internal_id>
External ID:           <cluster_external_id>
OpenShift Version:     4.13.0
Channel Group:         stable
DNS:                   <dns>
AWS Account:           <aws_account_id>
API URL:               <api_url>
Console URL:           <console_url>
Region:                us-east-1
Multi-AZ:              false
```

```
Nodes:
 - Control plane:        3
 - Infra:              2
 - Compute:            2
Network:
 - Type:               OVNKubernetes
 - Service CIDR:       <service_cidr>
 - Machine CIDR:       <machine_cidr>
 - Pod CIDR:           <pod_cidr>
 - Host Prefix:        <host_prefix>
Proxy:
 - HTTPProxy:          <proxy_url>
Additional trust bundle:    REDACTED
```

After removing the proxy, the proxy section is removed:

```
Name:                  <cluster_name>
ID:                    <cluster_internal_id>
External ID:           <cluster_external_id>
OpenShift Version:     4.13.0
Channel Group:         stable
DNS:                   <dns>
AWS Account:           <aws_account_id>
API URL:               <api_url>
Console URL:           <console_url>
Region:                us-east-1
Multi-AZ:              false
Nodes:
 - Control plane:        3
 - Infra:              2
 - Compute:            2
Network:
 - Type:               OVNKubernetes
 - Service CIDR:       <service_cidr>
 - Machine CIDR:       <machine_cidr>
 - Pod CIDR:           <pod_cidr>
 - Host Prefix:        <host_prefix>
Additional trust bundle:    REDACTED
```

4.5.2. Removing certificate authorities on a Red Hat OpenShift Service on AWS cluster

You can remove certificate authorities (CA) from your cluster with the **rosa** CLI tool.

Prerequisites

- You must have cluster administrator privileges.
- You have installed the **rosa** CLI tool.
- Your cluster has certificate authorities added.

Procedure

- Use the **rosa edit** command to modify the CA trust bundle. You must pass empty strings to the **--additional-trust-bundle-file** argument to clear the trust bundle from the cluster:

```
$ rosa edit cluster -c <cluster_name> --additional-trust-bundle-file ""
```

Example Output

```
I: Updated cluster <cluster_name>
```

Verification

- You can verify that the trust bundle has been removed from the cluster by using the **rosa describe** command:

```
$ rosa describe cluster -c <cluster_name>
```

Before removal, the Additional trust bundle section appears, redacting its value for security purposes:

```
Name:                  <cluster_name>
ID:                    <cluster_internal_id>
External ID:           <cluster_external_id>
OpenShift Version:     4.13.0
Channel Group:         stable
DNS:                   <dns>
AWS Account:           <aws_account_id>
API URL:               <api_url>
Console URL:           <console_url>
Region:                us-east-1
Multi-AZ:              false
Nodes:
 - Control plane:      3
 - Infra:              2
 - Compute:            2
Network:
 - Type:               OVNKubernetes
 - Service CIDR:       <service_cidr>
 - Machine CIDR:       <machine_cidr>
 - Pod CIDR:           <pod_cidr>
 - Host Prefix:        <host_prefix>
Proxy:
 - HTTPProxy:          <proxy_url>
Additional trust bundle:  REDACTED
```

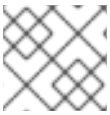
After removing the proxy, the Additional trust bundle section is removed:

```
Name:                  <cluster_name>
ID:                    <cluster_internal_id>
External ID:           <cluster_external_id>
OpenShift Version:     4.13.0
Channel Group:         stable
DNS:                   <dns>
AWS Account:           <aws_account_id>
API URL:               <api_url>
```

```
Console URL:            <console_url>
Region:                 us-east-1
Multi-AZ:               false
Nodes:
 - Control plane:       3
 - Infra:               2
 - Compute:             2
Network:
 - Type:                OVNKubernetes
 - Service CIDR:        <service_cidr>
 - Machine CIDR:        <machine_cidr>
 - Pod CIDR:            <pod_cidr>
 - Host Prefix:         <host_prefix>
Proxy:
 - HTTPProxy:           <proxy_url>
```

CHAPTER 5. CIDR RANGE DEFINITIONS

You must specify non-overlapping ranges for the following CIDR ranges.



NOTE

Machine CIDR ranges cannot be changed after creating your cluster.

When specifying subnet CIDR ranges, ensure that the subnet CIDR range is within the defined Machine CIDR. You must verify that the subnet CIDR ranges allow for enough IP addresses for all intended workloads, including at least eight IP addresses for possible AWS Load Balancers.

IMPORTANT

OVN-Kubernetes, the default network provider in Red Hat OpenShift Service on AWS 4.11 and later, uses the **100.64.0.0/16** IP address range internally. If your cluster uses OVN-Kubernetes, do not include the **100.64.0.0/16** IP address range in any other CIDR definitions in your cluster.

5.1. MACHINE CIDR

In the Machine CIDR field, you must specify the IP address range for machines or cluster nodes. This range must encompass all CIDR address ranges for your virtual private cloud (VPC) subnets. Subnets must be contiguous. A minimum IP address range of 128 addresses, using the subnet prefix **/25**, is supported for single availability zone deployments. A minimum address range of 256 addresses, using the subnet prefix **/24**, is supported for deployments that use multiple availability zones. The default is **10.0.0.0/16**. This range must not conflict with any connected networks.

5.2. SERVICE CIDR

In the Service CIDR field, you must specify the IP address range for services. The range must be large enough to accommodate your workload. The address block must not overlap with any external service accessed from within the cluster. The default is **172.30.0.0/16**. This address block needs to be the same between clusters.

5.3. POD CIDR

In the pod CIDR field, you must specify the IP address range for pods. The range must be large enough to accommodate your workload. The address block must not overlap with any external service accessed from within the cluster. The default is **10.128.0.0/14**. This address block needs to be the same between clusters.

5.4. HOST PREFIX

In the Host Prefix field, you must Specify the subnet prefix length assigned to pods scheduled to individual machines. The host prefix determines the pod IP address pool for each machine. For example, if the host prefix is set to **/23**, each machine is assigned a **/23** subnet from the pod CIDR address range. The default is **/23**, allowing 512 cluster nodes, and 512 pods per node (both of which are beyond our maximum supported).

CHAPTER 6. NETWORK POLICY

6.1. ABOUT NETWORK POLICY

As a cluster administrator, you can define network policies that restrict traffic to pods in your cluster.

6.1.1. About network policy

In a cluster using a network plugin that supports Kubernetes network policy, network isolation is controlled entirely by **NetworkPolicy** objects. In Red Hat OpenShift Service on AWS 4, OpenShift SDN supports using network policy in its default network isolation mode.

WARNING

Network policy does not apply to the host network namespace. Pods with host networking enabled are unaffected by network policy rules.

By default, all pods in a project are accessible from other pods and network endpoints. To isolate one or more pods in a project, you can create **NetworkPolicy** objects in that project to indicate the allowed incoming connections. Project administrators can create and delete **NetworkPolicy** objects within their own project.

If a pod is matched by selectors in one or more **NetworkPolicy** objects, then the pod will accept only connections that are allowed by at least one of those **NetworkPolicy** objects. A pod that is not selected by any **NetworkPolicy** objects is fully accessible.

The following example **NetworkPolicy** objects demonstrate supporting different scenarios:

- Deny all traffic:
  To make a project deny by default, add a **NetworkPolicy** object that matches all pods but accepts no traffic:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
spec:
  podSelector: {}
  ingress: []
```

- Only allow connections from the Red Hat OpenShift Service on AWS Ingress Controller:
  To make a project allow only connections from the Red Hat OpenShift Service on AWS Ingress Controller, add the following **NetworkPolicy** object.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
```

```
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
```

- Only accept connections from pods within a project:
  To make pods accept connections from other pods in the same project, but reject all other connections from pods in other projects, add the following **NetworkPolicy** object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector: {}
  ingress:
  - from:
    - podSelector: {}
```

- Only allow HTTP and HTTPS traffic based on pod labels:
  To enable only HTTP and HTTPS access to the pods with a specific label (**role=frontend** in following example), add a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-http-and-https
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
  - ports:
    - protocol: TCP
      port: 80
    - protocol: TCP
      port: 443
```

- Accept connections by using both namespace and pod selectors:
  To match network traffic by combining namespace and pod selectors, you can use a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-pod-and-namespace-both
spec:
  podSelector:
    matchLabels:
      name: test-pods
```

```
ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: project_name
      podSelector:
        matchLabels:
          name: test-pods
```

NetworkPolicy objects are additive, which means you can combine multiple **NetworkPolicy** objects together to satisfy complex network requirements.

For example, for the **NetworkPolicy** objects defined in previous samples, you can define both **allow-same-namespace** and **allow-http-and-https** policies within the same project. Thus allowing the pods with the label **role=frontend**, to accept any connection allowed by each policy. That is, connections on any port from pods in the same namespace, and connections on ports **80** and **443** from pods in any namespace.

6.1.1.1. Using the allow-from-router network policy

Use the following **NetworkPolicy** to allow external traffic regardless of the router configuration:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-router
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          policy-group.network.openshift.io/ingress:"" (1)
  podSelector: {}
  policyTypes:
  - Ingress
```

(1) **policy-group.network.openshift.io/ingress:""** label supports both Openshift-SDN and OVN-Kubernetes.

6.1.1.2. Using the allow-from-hostnetwork network policy

Add the following **allow-from-hostnetwork NetworkPolicy** object to direct traffic from the host network pods:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-hostnetwork
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          policy-group.network.openshift.io/host-network:""
```

```
podSelector: {}
policyTypes:
- Ingress
```

6.1.2. Optimizations for network policy with OpenShift SDN

Use a network policy to isolate pods that are differentiated from one another by labels within a namespace.

It is inefficient to apply **NetworkPolicy** objects to large numbers of individual pods in a single namespace. Pod labels do not exist at the IP address level, so a network policy generates a separate Open vSwitch (OVS) flow rule for every possible link between every pod selected with a **podSelector**.

For example, if the spec **podSelector** and the ingress **podSelector** within a **NetworkPolicy** object each match 200 pods, then 40,000 (200*200) OVS flow rules are generated. This might slow down a node.

When designing your network policy, refer to the following guidelines:

- Reduce the number of OVS flow rules by using namespaces to contain groups of pods that need to be isolated.
  NetworkPolicy objects that select a whole namespace, by using the **namespaceSelector** or an empty **podSelector**, generate only a single OVS flow rule that matches the VXLAN virtual network ID (VNID) of the namespace.

- Keep the pods that do not need to be isolated in their original namespace, and move the pods that require isolation into one or more different namespaces.

- Create additional targeted cross-namespace network policies to allow the specific traffic that you do want to allow from the isolated pods.

6.1.3. Optimizations for network policy with OpenShift OVN

When designing your network policy, refer to the following guidelines:

- For network policies with the same **spec.podSelector** spec, it is more efficient to use one network policy with multiple **ingress** or **egress** rules, than multiple network policies with subsets of **ingress** or **egress** rules.

- Every **ingress** or **egress** rule based on the **podSelector** or **namespaceSelector** spec generates the number of OVS flows proportional to **number of pods selected by network policy + number of pods selected by ingress or egress rule**. Therefore, it is preferable to use the **podSelector** or **namespaceSelector** spec that can select as many pods as you need in one rule, instead of creating individual rules for every pod.
  For example, the following policy contains two rules:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: test-network-policy
spec:
  podSelector: {}
  ingress:
    - from:
        - podSelector:
            matchLabels:
```

```
          role: frontend
    - from:
      - podSelector:
          matchLabels:
            role: backend
```

The following policy expresses those same two rules as one:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: test-network-policy
spec:
  podSelector: {}
  ingress:
    - from:
      - podSelector:
          matchExpressions:
            - {key: role, operator: In, values: [frontend, backend]}
```

The same guideline applies to the **spec.podSelector** spec. If you have the same **ingress** or **egress** rules for different network policies, it might be more efficient to create one network policy with a common **spec.podSelector** spec. For example, the following two policies have different rules:

```
metadata:
 name: policy1
spec:
 podSelector:
   matchLabels:
     role: db
  ingress:
    - from:
      - podSelector:
          matchLabels:
            role: frontend

metadata:
 name: policy2
spec:
 podSelector:
   matchLabels:
     role: client
  ingress:
    - from:
      - podSelector:
          matchLabels:
            role: frontend
```

The following network policy expresses those same two rules as one:

```
metadata:
 name: policy3
spec:
 podSelector:
```

```
    matchExpressions:
      - {key: role, operator: In, values: [db, client]}
  ingress:
    - from:
      - podSelector:
          matchLabels:
            role: frontend
```

You can apply this optimization when only multiple selectors are expressed as one. In cases where selectors are based on different labels, it may not be possible to apply this optimization. In those cases, consider applying some new labels for network policy optimization specifically.

6.1.4. Next steps

- Creating a network policy

6.2. CREATING A NETWORK POLICY

As a user with the **admin** role, you can create a network policy for a namespace.

6.2.1. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 (1)
spec:
  podSelector: (2)
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector: (3)
        matchLabels:
          app: app
    ports: (4)
    - protocol: TCP
      port: 27017
```

(1) The name of the NetworkPolicy object.

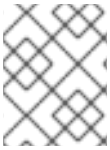
(2) A selector that describes the pods to which the policy applies. The policy object can only select pods in the project that defines the NetworkPolicy object.

(3) A selector that matches the pods from which the policy object allows ingress traffic. The selector matches pods in the same namespace as the NetworkPolicy.

(4) A list of one or more destination ports on which to accept traffic.

6.2.2. Creating a network policy using the CLI

To define granular rules describing ingress or egress network traffic allowed for namespaces in your cluster, you can create a network policy.



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy rule:
   a. Create a **<policy_name>.yaml** file:

```
$ touch <policy_name>.yaml
```

   where:

   <policy_name>
   Specifies the network policy file name.

   b. Define a network policy in the file that you just created, such as in the following examples:

   Deny ingress from all pods in all namespaces

   This is a fundamental policy, blocking all cross-pod networking other than cross-pod traffic allowed by the configuration of other Network Policies.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
spec:
  podSelector:
  ingress: []
```

   Allow ingress from all pods in the same namespace

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
```

```
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

Allow ingress traffic to one pod from a particular namespace

This policy allows traffic to pods labelled **pod-a** from pods running in **namespace-y**.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-traffic-pod
spec:
  podSelector:
    matchLabels:
      pod: pod-a
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: namespace-y
```

2. To create the network policy object, enter the following command:

```
$ oc apply -f <policy_name>.yaml -n <namespace>
```

where:

<policy_name>

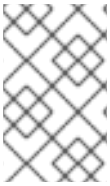
Specifies the network policy file name.

<namespace>

Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
networkpolicy.networking.k8s.io/deny-by-default created
```

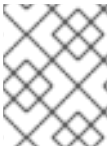


NOTE

If you log in to the web console with **cluster-admin** privileges, you have a choice of creating a network policy in any namespace in the cluster directly in YAML or from a form in the web console.

6.2.3. Creating a default deny all network policy

This is a fundamental policy, blocking all cross-pod networking other than network traffic allowed by the configuration of other deployed network policies. This procedure enforces a default **deny-by-default** policy.



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.

Procedure

1. Create the following YAML that defines a **deny-by-default** policy to deny ingress from all pods in all namespaces. Save the YAML in the **deny-by-default.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: default (1)
spec:
  podSelector: {} (2)
  ingress: [] (3)
```

(1) **namespace: default** deploys this policy to the **default** namespace.

(2) **podSelector:** is empty, this means it matches all the pods. Therefore, the policy applies to all pods in the default namespace.

(3) There are no **ingress** rules specified. This causes incoming traffic to be dropped to all pods.

2. Apply the policy by entering the following command:

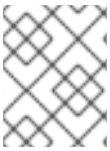
```
$ oc apply -f deny-by-default.yaml
```

Example output

```
networkpolicy.networking.k8s.io/deny-by-default created
```

6.2.4. Creating a network policy to allow traffic from external clients

With the **deny-by-default** policy in place you can proceed to configure a policy that allows traffic from external clients to a pod with the label **app=web**.



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Follow this procedure to configure a policy that allows external service from the public Internet directly or by using a Load Balancer to access the pod. Traffic is only allowed to a pod with the label **app=web**.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy that allows traffic from the public Internet directly or by using a load balancer to access the pod. Save the YAML in the **web-allow-external.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-external
  namespace: default
spec:
  policyTypes:
  - Ingress
  podSelector:
    matchLabels:
      app: web
  ingress:
    - {}
```

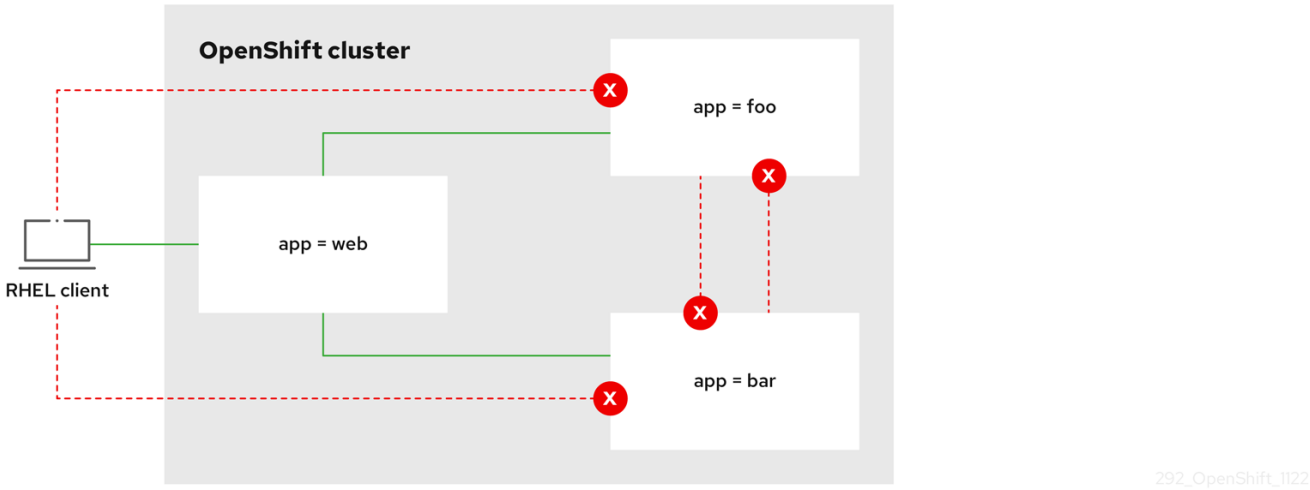
2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-external.yaml
```

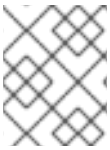
Example output

```
networkpolicy.networking.k8s.io/web-allow-external created
```

This policy allows traffic from all resources, including external traffic as illustrated in the following diagram:



6.2.5. Creating a network policy allowing traffic to an application from all namespaces



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Follow this procedure to configure a policy that allows traffic from all pods in all namespaces to a particular application.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.

Procedure

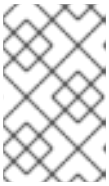
1. Create a policy that allows traffic from all pods in all namespaces to a particular application. Save the YAML in the **web-allow-all-namespaces.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-all-namespaces
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: web 1
  policyTypes:
```

```
  - Ingress
ingress:
- from:
  - namespaceSelector: {} 2
```

1 Applies the policy only to **app:web** pods in default namespace.

2 Selects all pods in all namespaces.



NOTE

By default, if you omit specifying a **namespaceSelector** it does not select any namespaces, which means the policy allows traffic only from the namespace the network policy is deployed to.

2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-all-namespaces.yaml
```

Example output

```
networkpolicy.networking.k8s.io/web-allow-all-namespaces created
```

Verification

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to deploy an **alpine** image in the **secondary** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=secondary --rm -i -t --image=alpine -- sh
```

3. Run the following command in the shell and observe that the request is allowed:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

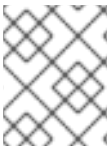
```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
```

```
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

6.2.6. Creating a network policy allowing traffic to an application from a namespace



NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Follow this procedure to configure a policy that allows traffic to a pod with the label **app=web** from a particular namespace. You might want to do this to:

- Restrict traffic to a production database only to namespaces where production workloads are deployed.
- Enable monitoring tools deployed to a particular namespace to scrape metrics from the current namespace.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy that allows traffic from all pods in a particular namespaces with a label **purpose=production**. Save the YAML in the **web-allow-prod.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-prod
  namespace: default
spec:
  podSelector:
```

```
    matchLabels:
      app: web 1
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production 2
```

1 Applies the policy only to **app:web** pods in the default namespace.

2 Restricts traffic to only pods in namespaces that have the label **purpose=production**.

2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-prod.yaml
```

Example output

```
networkpolicy.networking.k8s.io/web-allow-prod created
```

Verification

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to create the **prod** namespace:

```
$ oc create namespace prod
```

3. Run the following command to label the **prod** namespace:

```
$ oc label namespace/prod purpose=production
```

4. Run the following command to create the **dev** namespace:

```
$ oc create namespace dev
```

5. Run the following command to label the **dev** namespace:

```
$ oc label namespace/dev purpose=testing
```

6. Run the following command to deploy an **alpine** image in the **dev** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=dev --rm -i -t --image=alpine -- sh
```

7. Run the following command in the shell and observe that the request is blocked:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

```
wget: download timed out
```

8. Run the following command to deploy an **alpine** image in the **prod** namespace and start a shell:

```
$ oc run test-$RANDOM --namespace=prod --rm -i -t --image=alpine -- sh
```

9. Run the following command in the shell and observe that the request is allowed:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

6.2.7. Creating a network policy using OpenShift Cluster Manager

To define granular rules describing the ingress or egress network traffic allowed for namespaces in your cluster, you can create a network policy.

Prerequisites

- You logged in to OpenShift Cluster Manager Hybrid Cloud Console .
- You created an Red Hat OpenShift Service on AWS cluster.
- You configured an identity provider for your cluster.

- You added your user account to the configured identity provider.
- You created a project within your Red Hat OpenShift Service on AWS cluster.

Procedure

1. From OpenShift Cluster Manager Hybrid Cloud Console , click on the cluster you want to access.
2. Click **Open console** to navigate to the OpenShift web console.
3. Click on your identity provider and provide your credentials to log in to the cluster.
4. From the administrator perspective, under **Networking**, click **NetworkPolicies**.
5. Click **Create NetworkPolicy**.
6. Provide a name for the policy in the **Policy name** field.
7. Optional: You can provide the label and selector for a specific pod if this policy applies only to one or more specific pods. If you do not select a specific pod, then this policy will be applicable to all pods on the cluster.
8. Optional: You can block all ingress and egress traffic by using the **Deny all ingress traffic** or **Deny all egress traffic** checkboxes.
9. You can also add any combination of ingress and egress rules, allowing you to specify the port, namespace, or IP blocks you want to approve.
10. Add ingress rules to your policy:
    a. Select **Add ingress rule** to configure a new rule. This action creates a new **Ingress rule** row with an **Add allowed source** drop-down menu that enables you to specify how you want to limit inbound traffic. The drop-down menu offers three options to limit your ingress traffic:
        - **Allow pods from the same namespace** limits traffic to pods within the same namespace. You can specify the pods in a namespace, but leaving this option blank allows all of the traffic from pods in the namespace.
        - **Allow pods from inside the cluster** limits traffic to pods within the same cluster as the policy. You can specify namespaces and pods from which you want to allow inbound traffic. Leaving this option blank allows inbound traffic from all namespaces and pods within this cluster.
        - **Allow peers by IP block** limits traffic from a specified Classless Inter-Domain Routing (CIDR) IP block. You can block certain IPs with the exceptions option. Leaving the CIDR field blank allows all inbound traffic from all external sources.

    b. You can restrict all of your inbound traffic to a port. If you do not add any ports then all ports are accessible to traffic.
11. Add egress rules to your network policy:
    a. Select **Add egress rule** to configure a new rule. This action creates a new **Egress rule** row with an **Add allowed destination**"* drop-down menu that enables you to specify how you want to limit outbound traffic. The drop-down menu offers three options to limit your egress traffic:
        - **Allow pods from the same namespace** limits outbound traffic to pods within the same

namespace. You can specify the pods in a namespace, but leaving this option blank allows all of the traffic from pods in the namespace.

- **Allow pods from inside the cluster**limits traffic to pods within the same cluster as the policy. You can specify namespaces and pods from which you want to allow outbound traffic. Leaving this option blank allows outbound traffic from all namespaces and pods within this cluster.

- **Allow peers by IP block** limits traffic from a specified CIDR IP block. You can block certain IPs with the exceptions option. Leaving the CIDR field blank allows all outbound traffic from all external sources.

b. You can restrict all of your outbound traffic to a port. If you do not add any ports then all ports are accessible to traffic.

6.3. VIEWING A NETWORK POLICY

As a user with the **admin** role, you can view a network policy for a namespace.

6.3.1. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 1
spec:
  podSelector: 2
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector: 3
        matchLabels:
          app: app
    ports: 4
    - protocol: TCP
      port: 27017
```

1 The name of the NetworkPolicy object.

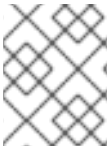
2 A selector that describes the pods to which the policy applies. The policy object can only select pods in the project that defines the NetworkPolicy object.

3 A selector that matches the pods from which the policy object allows ingress traffic. The selector matches pods in the same namespace as the NetworkPolicy.

4 A list of one or more destination ports on which to accept traffic.

6.3.2. Viewing network policies using the CLI

You can examine the network policies in a namespace.



NOTE

If you log in with a user with the **cluster-admin** role, then you can view any network policy in the cluster.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace where the network policy exists.

Procedure

- List network policies in a namespace:
  - To view network policy objects defined in a namespace, enter the following command:

```
$ oc get networkpolicy
```

  - Optional: To examine a specific network policy, enter the following command:

```
$ oc describe networkpolicy <policy_name> -n <namespace>
```

    where:

    <policy_name>

    Specifies the name of the network policy to inspect.

    <namespace>

    Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

    For example:

```
$ oc describe networkpolicy allow-same-namespace
```

    Output for oc describe command

```
Name:         allow-same-namespace
Namespace:    ns1
Created on:   2021-05-24 22:28:56 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: <none>
  Not affecting egress traffic
  Policy Types: Ingress
```

NOTE

If you log in to the web console with **cluster-admin** privileges, you have a choice of viewing a network policy in any namespace in the cluster directly in YAML or from a form in the web console.

6.3.3. Viewing network policies using OpenShift Cluster Manager

You can view the configuration details of your network policy in Red Hat OpenShift Cluster Manager.

Prerequisites

- You logged in to OpenShift Cluster Manager Hybrid Cloud Console .
- You created an Red Hat OpenShift Service on AWS cluster.
- You configured an identity provider for your cluster.
- You added your user account to the configured identity provider.
- You created a network policy.

Procedure

1. From the **Administrator** perspective in the OpenShift Cluster Manager web console, under **Networking**, click **NetworkPolicies**.
2. Select the desired network policy to view.
3. In the **Network Policy** details page, you can view all of the associated ingress and egress rules.
4. Select **YAML** on the network policy details to view the policy configuration in YAML format.

NOTE

You can only view the details of these policies. You cannot edit these policies.

6.4. DELETING A NETWORK POLICY

As a user with the **admin** role, you can delete a network policy from a namespace.

6.4.1. Deleting a network policy using the CLI

You can delete a network policy in a namespace.

NOTE

If you log in with a user with the **cluster-admin** role, then you can delete any network policy in the cluster.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace where the network policy exists.

Procedure

- To delete a network policy object, enter the following command:

```
$ oc delete networkpolicy <policy_name> -n <namespace>
```

where:

<policy_name>
Specifies the name of the network policy.

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
networkpolicy.networking.k8s.io/default-deny deleted
```

NOTE

If you log in to the web console with **cluster-admin** privileges, you have a choice of deleting a network policy in any namespace in the cluster directly in YAML or from the policy in the web console through the **Actions** menu.

6.4.2. Deleting a network policy using OpenShift Cluster Manager

You can delete a network policy in a namespace.

Prerequisites

- You logged in to OpenShift Cluster Manager Hybrid Cloud Console .
- You created an Red Hat OpenShift Service on AWS cluster.
- You configured an identity provider for your cluster.
- You added your user account to the configured identity provider.

Procedure

1. From the **Administrator** perspective in the OpenShift Cluster Manager web console, under **Networking**, click **NetworkPolicies**.

2. Use one of the following methods for deleting your network policy:

   - Delete the policy from the **Network Policies** table:

     a. From the **Network Policies** table, select the stack menu on the row of the network policy you want to delete and then, click **Delete NetworkPolicy**.

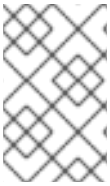
   - Delete the policy using the **Actions** drop-down menu from the individual network policy details:

     a. Click on **Actions** drop-down menu for your network policy.

     b. Select **Delete NetworkPolicy** from the menu.

6.5. CONFIGURING MULTITENANT ISOLATION WITH NETWORK POLICY

As a cluster administrator, you can configure your network policies to provide multitenant network isolation.



NOTE

If you are using the OpenShift SDN network plugin, configuring network policies as described in this section provides network isolation similar to multitenant mode but with network policy mode set.

6.5.1. Configuring multitenant isolation by using network policy

You can configure your project to isolate it from pods and services in other project namespaces.

Prerequisites

- Your cluster uses a network plugin that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.

Procedure

1. Create the following **NetworkPolicy** objects:

   a. A policy named **allow-from-openshift-ingress**.

```
$ cat << EOF| oc create -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
```

```
      matchLabels:
        policy-group.network.openshift.io/ingress: ""
  podSelector: {}
  policyTypes:
  - Ingress
EOF
```

NOTE

policy-group.network.openshift.io/ingress: "" is the preferred namespace selector label for OpenShift SDN. You can use the **network.openshift.io/policy-group: ingress** namespace selector label, but this is a legacy label.

b. A policy named **allow-from-openshift-monitoring**:

```
$ cat << EOF| oc create -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: monitoring
  podSelector: {}
  policyTypes:
  - Ingress
EOF
```

c. A policy named **allow-same-namespace**:

```
$ cat << EOF| oc create -f -
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
EOF
```

d. A policy named **allow-from-kube-apiserver-operator**:

```
$ cat << EOF| oc create -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-kube-apiserver-operator
spec:
```

```
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: openshift-kube-apiserver-operator
      podSelector:
        matchLabels:
          app: kube-apiserver-operator
  policyTypes:
  - Ingress
EOF
```

For more details, see New **kube-apiserver-operator** webhook controller validating health of webhook.

2. Optional: To confirm that the network policies exist in your current project, enter the following command:

```
$ oc describe networkpolicy
```

Example output

```
Name:         allow-from-openshift-ingress
Namespace:    example1
Created on:   2020-06-09 00:28:17 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: network.openshift.io/policy-group: ingress
  Not affecting egress traffic
  Policy Types: Ingress


Name:         allow-from-openshift-monitoring
Namespace:    example1
Created on:   2020-06-09 00:29:57 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: network.openshift.io/policy-group: monitoring
  Not affecting egress traffic
  Policy Types: Ingress
```

CHAPTER 7. CONFIGURING ROUTES

7.1. ROUTE CONFIGURATION

7.1.1. Creating an HTTP-based route

A route allows you to host your application at a public URL. It can either be secure or unsecured, depending on the network security configuration of your application. An HTTP-based route is an unsecured route that uses the basic HTTP routing protocol and exposes a service on an unsecured application port.

The following procedure describes how to create a simple HTTP-based route to a web application, using the **hello-openshift** application as an example.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in as an administrator.
- You have a web application that exposes a port and a TCP endpoint listening for traffic on the port.

Procedure

1. Create a project called **hello-openshift** by running the following command:

```
$ oc new-project hello-openshift
```

2. Create a pod in the project by running the following command:

```
$ oc create -f https://raw.githubusercontent.com/openshift/origin/master/examples/hello-openshift/hello-pod.json
```

3. Create a service called **hello-openshift** by running the following command:

```
$ oc expose pod/hello-openshift
```

4. Create an unsecured route to the **hello-openshift** application by running the following command:

```
$ oc expose svc hello-openshift
```

Verification

- To verify that the **route** resource that you created, run the following command:

```
$ oc get routes -o yaml <name of resource> 1
```

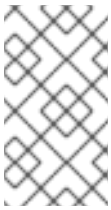
1 In this example, the route is named **hello-openshift**.

Sample YAML definition of the created unsecured route:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello-openshift
spec:
  host: hello-openshift-hello-openshift.<Ingress_Domain> 1
  port:
    targetPort: 8080 2
  to:
    kind: Service
    name: hello-openshift
```

1 **<Ingress_Domain>** is the default ingress domain name. The **ingresses.config/cluster** object is created during the installation and cannot be changed. If you want to specify a different domain, you can specify an alternative cluster domain using the **appsDomain** option.

2 **targetPort** is the target port on pods that is selected by the service that this route points to.



NOTE

To display your default ingress domain, run the following command:

```
$ oc get ingresses.config/cluster -o jsonpath={.spec.domain}
```

7.1.2. Configuring route timeouts

You can configure the default timeouts for an existing route when you have services in need of a low timeout, which is required for Service Level Availability (SLA) purposes, or a high timeout, for cases with a slow back end.

Prerequisites

- You need a deployed Ingress Controller on a running cluster.

Procedure

1. Using the **oc annotate** command, add the timeout to the route:

```
$ oc annotate route <route_name> \
    --overwrite haproxy.router.openshift.io/timeout=<timeout><time_unit> 1
```

1 Supported time units are microseconds (us), milliseconds (ms), seconds (s), minutes (m), hours (h), or days (d).

The following example sets a timeout of two seconds on a route named **myroute**:

```
$ oc annotate route myroute --overwrite haproxy.router.openshift.io/timeout=2s
```

7.1.3. HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) policy is a security enhancement, which signals to the browser client that only HTTPS traffic is allowed on the route host. HSTS also optimizes web traffic by signaling HTTPS transport is required, without using HTTP redirects. HSTS is useful for speeding up interactions with websites.

When HSTS policy is enforced, HSTS adds a Strict Transport Security header to HTTP and HTTPS responses from the site. You can use the **insecureEdgeTerminationPolicy** value in a route to redirect HTTP to HTTPS. When HSTS is enforced, the client changes all requests from the HTTP URL to HTTPS before the request is sent, eliminating the need for a redirect.

Cluster administrators can configure HSTS to do the following:

- Enable HSTS per-route
- Disable HSTS per-route
- Enforce HSTS per-domain, for a set of domains, or use namespace labels in combination with domains

IMPORTANT

HSTS works only with secure routes, either edge-terminated or re-encrypt. The configuration is ineffective on HTTP or passthrough routes.

7.1.3.1. Enabling HTTP Strict Transport Security per-route

HTTP strict transport security (HSTS) is implemented in the HAProxy template and applied to edge and re-encrypt routes that have the **haproxy.router.openshift.io/hsts_header** annotation.

Prerequisites

- You are logged in to the cluster with a user with administrator privileges for the project.
- You installed the **oc** CLI.

Procedure

- To enable HSTS on a route, add the **haproxy.router.openshift.io/hsts_header** value to the edge-terminated or re-encrypt route. You can use the **oc annotate** tool to do this by running the following command:

```
$ oc annotate route <route_name> -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=31536000;\ 1
includeSubDomains;preload"
```

1 In this example, the maximum age is set to **31536000** ms, which is approximately eight and a half hours.

NOTE

In this example, the equal sign (**=**) is in quotes. This is required to properly execute the annotate command.

Example route configured with an annotation

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubDomains;preload
1 2 3
...
spec:
  host: def.abc.com
  tls:
    termination: "reencrypt"
    ...
  wildcardPolicy: "Subdomain"
```

1. Required. **max-age** measures the length of time, in seconds, that the HSTS policy is in effect. If set to **0**, it negates the policy.
2. Optional. When included, **includeSubDomains** tells the client that all subdomains of the host must have the same HSTS policy as the host.
3. Optional. When **max-age** is greater than 0, you can add **preload** in **haproxy.router.openshift.io/hsts_header** to allow external services to include this site in their HSTS preload lists. For example, sites such as Google can construct a list of sites that have **preload** set. Browsers can then use these lists to determine which sites they can communicate with over HTTPS, even before they have interacted with the site. Without **preload** set, browsers must have interacted with the site over HTTPS, at least once, to get the header.

7.1.3.2. Disabling HTTP Strict Transport Security per-route

To disable HTTP strict transport security (HSTS) per-route, you can set the **max-age** value in the route annotation to **0**.

Prerequisites

- You are logged in to the cluster with a user with administrator privileges for the project.
- You installed the **oc** CLI.

Procedure

- To disable HSTS, set the **max-age** value in the route annotation to **0**, by entering the following command:

```
$ oc annotate route <route_name> -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=0"
```

TIP

You can alternatively apply the following YAML to create the config map:

Example of disabling HSTS per-route

```
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=0
```

- To disable HSTS for every route in a namespace, enter the following command:

```
$ oc annotate route --all -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=0"
```

Verification

1. To query the annotation for all routes, enter the following command:

```
$ oc get route  --all-namespaces -o go-template='{{range .items}}{{if .metadata.annotations}}
{{$a := index .metadata.annotations "haproxy.router.openshift.io/hsts_header"}}{{$n :=
.metadata.name}}{{with $a}}Name: {{$n}} HSTS: {{$a}}{{"\n"}}{{else}}{{""}}{{end}}{{end}}
{{end}}'
```

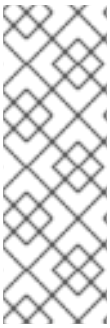
Example output

```
Name: routename HSTS: max-age=0
```

7.1.4. Using cookies to keep route statefulness

Red Hat OpenShift Service on AWS provides sticky sessions, which enables stateful application traffic by ensuring all traffic hits the same endpoint. However, if the endpoint pod terminates, whether through restart, scaling, or a change in configuration, this statefulness can disappear.

Red Hat OpenShift Service on AWS can use cookies to configure session persistence. The Ingress controller selects an endpoint to handle any user requests, and creates a cookie for the session. The cookie is passed back in the response to the request and the user sends the cookie back with the next request in the session. The cookie tells the Ingress Controller which endpoint is handling the session, ensuring that client requests use the cookie so that they are routed to the same pod.



NOTE

Cookies cannot be set on passthrough routes, because the HTTP traffic cannot be seen. Instead, a number is calculated based on the source IP address, which determines the backend.

If backends change, the traffic can be directed to the wrong server, making it less sticky. If you are using a load balancer, which hides source IP, the same number is set for all connections and traffic is sent to the same pod.

7.1.4.1. Annotating a route with a cookie

You can set a cookie name to overwrite the default, auto-generated one for the route. This allows the application receiving route traffic to know the cookie name. By deleting the cookie it can force the next request to re-choose an endpoint. So, if a server was overloaded it tries to remove the requests from the client and redistribute them.

Procedure

1. Annotate the route with the specified cookie name:

```
$ oc annotate route <route_name> router.openshift.io/cookie_name="<cookie_name>"
```

where:

<route_name>
Specifies the name of the route.

<cookie_name>
Specifies the name for the cookie.

For example, to annotate the route **my_route** with the cookie name **my_cookie**:

```
$ oc annotate route my_route router.openshift.io/cookie_name="my_cookie"
```

2. Capture the route hostname in a variable:

```
$ ROUTE_NAME=$(oc get route <route_name> -o jsonpath='{.spec.host}')
```

where:

<route_name>
Specifies the name of the route.

3. Save the cookie, and then access the route:

```
$ curl $ROUTE_NAME -k -c /tmp/cookie_jar
```

Use the cookie saved by the previous command when connecting to the route:

```
$ curl $ROUTE_NAME -k -b /tmp/cookie_jar
```

7.1.5. Path-based routes

Path-based routes specify a path component that can be compared against a URL, which requires that the traffic for the route be HTTP based. Thus, multiple routes can be served using the same hostname, each with a different path. Routers should match routes based on the most specific path to the least. However, this depends on the router implementation.

The following table shows example routes and their accessibility:

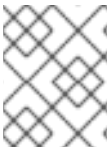
Table 7.1. Route availability

Route	When Compared to	Accessible
www.example.com/test	*www.example.com/test*	Yes
	www.example.com	No
www.example.com/test and *www.example.com*	*www.example.com/test*	Yes
	www.example.com	Yes
www.example.com	*www.example.com/text*	Yes (Matched by the host, not the route)
	www.example.com	Yes

An unsecured route with a path

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  path: "/test" 1
  to:
    kind: Service
    name: service-name
```

1 The path is the only added attribute for a path-based route.



NOTE

Path-based routing is not available when using passthrough TLS, as the router does not terminate TLS in that case and cannot read the contents of the request.

7.1.6. Route-specific annotations

The Ingress Controller can set the default options for all the routes it exposes. An individual route can override some of these defaults by providing specific configurations in its annotations. Red Hat does not support adding a route annotation to an operator-managed route.

IMPORTANT

To create a whitelist with multiple source IPs or subnets, use a space-delimited list. Any other delimiter type causes the list to be ignored without a warning or error message.

Table 7.2. Route annotations

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/balance	Sets the load-balancing algorithm. Available options are **random**, **source**, **roundrobin**, and **leastconn**. The default value is **random**.	**ROUTER_TCP_BALANCE_SCHEME** for passthrough routes. Otherwise, use **ROUTER_LOAD_BALANCE_ALGORITHM**.
haproxy.router.openshift.io/disable_cookies	Disables the use of cookies to track related connections. If set to **'true'** or **'TRUE'**, the balance algorithm is used to choose which back-end serves connections for each incoming HTTP request.	
router.openshift.io/cookie_name	Specifies an optional cookie to use for this route. The name must consist of any combination of upper and lower case letters, digits, "_", and "-". The default is the hashed internal key name for the route.	
haproxy.router.openshift.io/pod-concurrent-connections	Sets the maximum number of connections that are allowed to a backing pod from a router. Note: If there are multiple pods, each can have this many connections. If you have multiple routers, there is no coordination among them, each may connect this many times. If not set, or set to 0, there is no limit.	
haproxy.router.openshift.io/rate-limit-connections	Setting **'true'** or **'TRUE'** enables rate limiting functionality which is implemented through stick-tables on the specific backend per route. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp	Limits the number of concurrent TCP connections made through the same source IP address. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/rate-limit-connections.rate-http	Limits the rate at which a client with the same source IP address can make HTTP requests. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/rate-limit-connections.rate-tcp	Limits the rate at which a client with the same source IP address can make TCP connections. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/timeout	Sets a server-side timeout for the route. (TimeUnits)	**ROUTER_DEFAULT_SERVER_TIMEOUT**
haproxy.router.openshift.io/timeout-tunnel	This timeout applies to a tunnel connection, for example, WebSocket over cleartext, edge, reencrypt, or passthrough routes. With cleartext, edge, or reencrypt route types, this annotation is applied as a timeout tunnel with the existing timeout value. For the passthrough route types, the annotation takes precedence over any existing timeout value set.	**ROUTER_DEFAULT_TUNNEL_TIMEOUT**
ingresses.config/cluster ingress.operator.openshift.io/hard-stop-after	You can set either an IngressController or the ingress config . This annotation redeploys the router and configures the HA proxy to emit the haproxy **hard-stop-after** global option, which defines the maximum time allowed to perform a clean soft-stop.	**ROUTER_HARD_STOP_AFTER**
router.openshift.io/haproxy.health.check.interval	Sets the interval for the back-end health checks. (TimeUnits)	**ROUTER_BACKEND_CHECK_INTERVAL**

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/ip_whitelist	Sets a whitelist for the route. The whitelist is a space-separated list of IP addresses and CIDR ranges for the approved source addresses. Requests from IP addresses that are not in the whitelist are dropped. The maximum number of IP addresses and CIDR ranges allowed in a whitelist is 61.	
haproxy.router.openshift.io/hsts_header	Sets a Strict-Transport-Security header for the edge terminated or re-encrypt route.	
haproxy.router.openshift.io/log-send-hostname	Sets the **hostname** field in the Syslog header. Uses the hostname of the system. **log-send-hostname** is enabled by default if any Ingress API logging method, such as sidecar or Syslog facility, is enabled for the router.	
haproxy.router.openshift.io/rewrite-target	Sets the rewrite path of the request on the backend.	
router.openshift.io/cookie-same-site	Sets a value to restrict cookies. The values are: **Lax**: cookies are transferred between the visited site and third-party sites. **Strict**: cookies are restricted to the visited site. **None**: cookies are restricted to the visited site. This value is applicable to re-encrypt and edge routes only. For more information, see the SameSite cookies documentation.	

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/set-forwarded-headers	Sets the policy for handling the **Forwarded** and **X-Forwarded-For** HTTP headers per route. The values are: **append**: appends the header, preserving any existing header. This is the default value. **replace**: sets the header, removing any existing header. **never**: never sets the header, but preserves any existing header. **if-none**: sets the header if it is not already set.	**ROUTER_SET_FORWARDED_HEADERS**

NOTE

Environment variables cannot be edited.

Router timeout variables

TimeUnits are represented by a number followed by the unit: **us** *(microseconds), **ms** (milliseconds, default), **s** (seconds), **m** (minutes), **h** *(hours), **d** (days).

The regular expression is: [1-9][0-9]*(**us**\|**ms**\|**s**\|**m**\|**h**\|**d**).

Variable	Default	Description
ROUTER_BACKEND_CHECK_INTERVAL	**5000ms**	Length of time between subsequent liveness checks on back ends.
ROUTER_CLIENT_FIN_TIMEOUT	**1s**	Controls the TCP FIN timeout period for the client connecting to the route. If the FIN sent to close the connection does not answer within the given time, HAProxy closes the connection. This is harmless if set to a low value and uses fewer resources on the router.
ROUTER_DEFAULT_CLIENT_TIMEOUT	**30s**	Length of time that a client has to acknowledge or send data.
ROUTER_DEFAULT_CONNECT_TIMEOUT	**5s**	The maximum connection time.

Variable	Default	Description
ROUTER_DEFAULT_SERVER_FIN_TIMEOUT	**1s**	Controls the TCP FIN timeout from the router to the pod backing the route.
ROUTER_DEFAULT_SERVER_TIMEOUT	**30s**	Length of time that a server has to acknowledge or send data.
ROUTER_DEFAULT_TUNNEL_TIMEOUT	**1h**	Length of time for TCP or WebSocket connections to remain open. This timeout period resets whenever HAProxy reloads.
ROUTER_SLOWLORIS_HTTP_KEEPALIVE	**300s**	Set the maximum time to wait for a new HTTP request to appear. If this is set too low, it can cause problems with browsers and applications not expecting a small **keepalive** value. Some effective timeout values can be the sum of certain variables, rather than the specific expected timeout. For example, **ROUTER_SLOWLORIS_HTTP_KEEPALIVE** adjusts **timeout http-keep-alive**. It is set to **300s** by default, but HAProxy also waits on **tcp-request inspect-delay**, which is set to **5s**. In this case, the overall timeout would be **300s** plus **5s**.
ROUTER_SLOWLORIS_TIMEOUT	**10s**	Length of time the transmission of an HTTP request can take.
RELOAD_INTERVAL	**5s**	Allows the minimum frequency for the router to reload and accept new changes.
ROUTER_METRICS_HAPROXY_TIMEOUT	**5s**	Timeout for the gathering of HAProxy metrics.

A route setting custom timeout

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms (1)
...
```

(1) Specifies the new timeout with HAProxy supported units (**us**, **ms**, **s**, **m**, **h**, **d**). If the unit is not provided, **ms** is the default.

NOTE

Setting a server-side timeout value for passthrough routes too low can cause WebSocket connections to timeout frequently on that route.

A route that allows only one specific IP address

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10
```

A route that allows several IP addresses

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10 192.168.1.11 192.168.1.12
```

A route that allows an IP address CIDR network

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.0/24
```

A route that allows both IP an address and IP address CIDR networks

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 180.5.61.153 192.168.1.0/24 10.0.0.0/8
```

A route specifying a rewrite target

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/rewrite-target: / (1)
...
```

(1) Sets / as rewrite path of the request on the backend.

Setting the **haproxy.router.openshift.io/rewrite-target** annotation on a route specifies that the Ingress Controller should rewrite paths in HTTP requests using this route before forwarding the requests to the backend application. The part of the request path that matches the path specified in **spec.path** is replaced with the rewrite target specified in the annotation.

The following table provides examples of the path rewriting behavior for various combinations of **spec.path**, request path, and rewrite target.

Table 7.3. rewrite-target examples:

Route.spec.path	Request path	Rewrite target	Forwarded request path
/foo	/foo	/	/
/foo	/foo/	/	/
/foo	/foo/bar	/	/bar
/foo	/foo/bar/	/	/bar/
/foo	/foo	/bar	/bar
/foo	/foo/	/bar	/bar/
/foo	/foo/bar	/baz	/baz/bar
/foo	/foo/bar/	/baz	/baz/bar/
/foo/	/foo	/	N/A (request path does not match route path)
/foo/	/foo/	/	/
/foo/	/foo/bar	/	/bar

7.1.7. Creating a route using the default certificate through an Ingress object

If you create an Ingress object without specifying any TLS configuration, Red Hat OpenShift Service on AWS generates an insecure route. To create an Ingress object that generates a secure, edge-terminated route using the default ingress certificate, you can specify an empty TLS configuration as follows.

Prerequisites

- You have a service that you want to expose.
- You have access to the OpenShift CLI (**oc**).

Procedure

1. Create a YAML file for the Ingress object. In this example, the file is called **example-ingress.yaml**:

   YAML definition of an Ingress object

   ```
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
     name: frontend
   ```

```
  ...
spec:
  rules:
    ...
  tls:
  - {} (1)
```

(1) Use this exact syntax to specify TLS without specifying a custom certificate.

2. Create the Ingress object by running the following command:

```
$ oc create -f example-ingress.yaml
```

Verification

- Verify that Red Hat OpenShift Service on AWS has created the expected route for the Ingress object by running the following command:

```
$ oc get routes -o yaml
```

Example output

```
apiVersion: v1
items:
- apiVersion: route.openshift.io/v1
  kind: Route
  metadata:
    name: frontend-j9sdd (1)
    ...
  spec:
  ...
    tls: (2)
      insecureEdgeTerminationPolicy: Redirect
      termination: edge (3)
  ...
```

(1) The name of the route includes the name of the Ingress object followed by a random suffix.

(2) In order to use the default certificate, the route should not specify **spec.certificate**.

(3) The route should specify the **edge** termination policy.

7.1.8. Creating a route using the destination CA certificate in the Ingress annotation

The **route.openshift.io/destination-ca-certificate-secret** annotation can be used on an Ingress object to define a route with a custom destination CA certificate.

Prerequisites

- You may have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.

- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a separate destination CA certificate in a PEM-encoded file.
- You must have a service that you want to expose.

Procedure

1. Add the **route.openshift.io/destination-ca-certificate-secret** to the Ingress annotations:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend
  annotations:
    route.openshift.io/termination: "reencrypt"
    route.openshift.io/destination-ca-certificate-secret: secret-ca-cert (1)
...
```

(1) The annotation references a kubernetes secret.

2. The secret referenced in this annotation will be inserted into the generated route.

Example output

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend
  annotations:
    route.openshift.io/termination: reencrypt
    route.openshift.io/destination-ca-certificate-secret: secret-ca-cert
spec:
...
  tls:
    insecureEdgeTerminationPolicy: Redirect
    termination: reencrypt
    destinationCACertificate: |
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
...
```

Additional resources

- Specifying an alternative cluster domain using the appsDomain option

7.2. SECURED ROUTES

Secure routes provide the ability to use several types of TLS termination to serve certificates to the client. The following sections describe how to create re-encrypt, edge, and passthrough routes with custom certificates.

7.2.1. Creating a re-encrypt route with a custom certificate

You can configure a secure route using reencrypt TLS termination with a custom certificate by using the **oc create route** command.

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.
- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a separate destination CA certificate in a PEM-encoded file.
- You must have a service that you want to expose.

NOTE

Password protected key files are not supported. To remove a passphrase from a key file, use the following command:

```
$ openssl rsa -in password_protected_tls.key -out tls.key
```

Procedure

This procedure creates a **Route** resource with a custom certificate and reencrypt TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You must also specify a destination CA certificate to enable the Ingress Controller to trust the service's certificate. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, **cacert.crt**, and (optionally) **ca.crt**. Substitute the name of the **Service** resource that you want to expose for **frontend**. Substitute the appropriate hostname for **www.example.com**.

- Create a secure **Route** resource using reencrypt TLS termination and a custom certificate:

```
$ oc create route reencrypt --service=frontend --cert=tls.crt --key=tls.key --dest-ca-
cert=destca.crt --ca-cert=ca.crt --hostname=www.example.com
```

If you examine the resulting **Route** resource, it should look similar to the following:

YAML Definition of the Secure Route

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: reencrypt
    key: |-
```

```
    -----BEGIN PRIVATE KEY-----
    [...]
    -----END PRIVATE KEY-----
  certificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
  caCertificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
  destinationCACertificate: |-
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
```

See **oc create route reencrypt --help** for more options.

7.2.2. Creating an edge route with a custom certificate

You can configure a secure route using edge TLS termination with a custom certificate by using the **oc create route** command. With an edge route, the Ingress Controller terminates TLS encryption before forwarding traffic to the destination pod. The route specifies the TLS certificate and key that the Ingress Controller uses for the route.

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.
- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a service that you want to expose.

NOTE

Password protected key files are not supported. To remove a passphrase from a key file, use the following command:

```
$ openssl rsa -in password_protected_tls.key -out tls.key
```

Procedure

This procedure creates a **Route** resource with a custom certificate and edge TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, and (optionally) **ca.crt**. Substitute the name of the service that you want to expose for **frontend**. Substitute the appropriate hostname for **www.example.com**.

- Create a secure **Route** resource using edge TLS termination and a custom certificate.

```
$ oc create route edge --service=frontend --cert=tls.crt --key=tls.key --ca-cert=ca.crt --
hostname=www.example.com
```

If you examine the resulting **Route** resource, it should look similar to the following:

YAML Definition of the Secure Route

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

See **oc create route edge --help** for more options.

7.2.3. Creating a passthrough route

You can configure a secure route using passthrough termination by using the **oc create route** command. With passthrough termination, encrypted traffic is sent straight to the destination without the router providing TLS termination. Therefore no key or certificate is required on the route.

Prerequisites

- You must have a service that you want to expose.

Procedure

- Create a **Route** resource:

```
$ oc create route passthrough route-passthrough-secured --service=frontend --port=8080
```

If you examine the resulting **Route** resource, it should look similar to the following:

A Secured Route Using Passthrough Termination

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-passthrough-secured (1)
```

```
spec:
  host: www.example.com
  port:
    targetPort: 8080
  tls:
    termination: passthrough (2)
    insecureEdgeTerminationPolicy: None (3)
  to:
    kind: Service
    name: frontend
```

(1) The name of the object, which is limited to 63 characters.

(2) The **termination** field is set to **passthrough**. This is the only required **tls** field.

(3) Optional **insecureEdgeTerminationPolicy**. The only valid values are **None**, **Redirect**, or empty for disabled.

The destination pod is responsible for serving certificates for the traffic at the endpoint. This is currently the only method that can support requiring client certificates, also known as two-way authentication.