Red Hat Advanced Cluster Management for Kubernetes 2.5

Governance

Read more to learn about the governance policy framework, which helps harden cluster security by using policies.

Last Updated: 2023-07-13

Red Hat Advanced Cluster Management for Kubernetes 2.5 Governance

Read more to learn about the governance policy framework, which helps harden cluster security by using policies.

Legal Notice

Abstract

Read more to learn about the governance policy framework, which helps harden cluster security by using policies.

Table of Contents

CHAPTER 1. RISK AND COMPLIANCE

Manage your security of Red Hat Advanced Cluster Management for Kubernetes components. Govern your cluster with defined policies and processes to identify and minimize risks. Use policies to define rules and set controls.

Prerequisite: You must configure authentication service requirements for Red Hat Advanced Cluster Management for Kubernetes. See Access control for more information.

Review the following topics to learn more about securing your cluster:

- Role-based access control
- Managing credentials overview
- Certificates
- Governance
  - Support for templates in configuration policies
  - Integrity shield protection (Technology Preview)

1.1. CERTIFICATES

Various certificates are created and used throughout Red Hat Advanced Cluster Management for Kubernetes.

You can bring your own certificates. You must create a Kubernetes TLS Secret for your certificate. After you create your certificates, you can replace certain certificates that are created by the Red Hat Advanced Cluster Management installer.

Required access: Cluster administrator

All certificates required by services that run on Red Hat Advanced Cluster Management are created during the installation of Red Hat Advanced Cluster Management. Certificates are created and managed by the following components:

- OpenShift Service Serving Certificates
- Red Hat Advanced Cluster Management webhook controllers
- Kubernetes Certificates API
- OpenShift default ingress

Continue reading to learn more about certificate management:

Red Hat Advanced Cluster Management hub cluster certificates

- Replacing the management ingress certificates
- Replacing the OpenShift default ingress certificate
- Observability certificates
  - Bring Your Own (BYO) observability certificate authority (CA) certificates

  - OpenSSL commands to generate CA certificate
  - Create the secrets associated with the BYO observability CA certificates
  - Replacing certificates for alertmanager route

Red Hat Advanced Cluster Management component certificates

- List hub cluster managed certificates
- Refresh hub cluster managed certificates
- Refresh a Red Hat Advanced Cluster Management webhook certificate

Red Hat Advanced Cluster Management managed certificates

- Channel certificates
- Managed cluster certificates

Third-party certificates

- Rotating the gatekeeper webhook certificate
- Rotating the integrity shield webhook certificate (Technology Preview)

Note: Users are responsible for certificate rotations and updates.

1.1.1. Red Hat Advanced Cluster Management hub cluster certificates

1.1.1.1. Observability certificates

After Red Hat Advanced Cluster Management is installed, observability certificates are created and used by the observability components, to provide mutual TLS on the traffic between the hub cluster and managed cluster. The Kubernetes secrets that are associated with the observability certificates.

The **open-cluster-management-observability** namespace contain the following certificates:

- **observability-server-ca-certs**: Has the CA certificate to sign server-side certificates
- **observability-client-ca-certs**: Has the CA certificate to sign client-side certificates
- **observability-server-certs**: Has the server certificate used by the **observability-observatorium-api** deployment
- **observability-grafana-certs**: Has the client certificate used by the **observability-rbac-query-proxy** deployment

The **open-cluster-management-addon-observability** namespace contain the following certificates on managed clusters:

- **observability-managed-cluster-certs**: Has the same server CA certificate as **observability-server-ca-certs** in the hub server
- **observability-controller-open-cluster-management.io-observability-signer-client-cert**: Has the client certificate used by the **metrics-collector-deployment**

The CA certificates are valid for five years and other certificates are valid for one year. All observability certificates are automatically refreshed upon expiration.

View the following list to understand the effects when certificates are automatically renewed:

- Non-CA certificates are renewed automatically when the remaining valid time is no more than 73 days. After the certificate is renewed, the pods in the related deployments restart automatically to use the renewed certificates.

- CA certificates are renewed automatically when the remaining valid time is no more than one year. After the certificate is renewed, the old CA is not deleted but co-exist with the renewed ones. Both old and renewed certificates are used by related deployments, and continue to work. The old CA certificates are deleted when they expire.

- When a certificate is renewed, the traffic between the hub cluster and managed cluster is not interrupted.

1.1.1.2. Bring Your Own (BYO) observability certificate authority (CA) certificates

If you do not want to use the default observability CA certificates generated by Red Hat Advanced Cluster Management, you can choose to use the BYO observability CA certificates before you enable observability.

1.1.1.2.1. OpenSSL commands to generate CA certificate

Observability requires two CA certificates; one is for the server-side and the other is for the client-side.

- Generate your CA RSA private keys with the following commands:

  ```
  openssl genrsa -out serverCAKey.pem 2048

  openssl genrsa -out clientCAKey.pem 2048
  ```

- Generate the self-signed CA certificates using the private keys. Run the following commands:

  ```
  openssl req -x509 -sha256 -new -nodes -key serverCAKey.pem -days 1825 -out
  serverCACert.pem

  openssl req -x509 -sha256 -new -nodes -key clientCAKey.pem -days 1825 -out
  clientCACert.pem
  ```

1.1.1.2.2. Create the secrets associated with the BYO observability CA certificates

Complete the following steps to create the secrets:

1. Create the **observability-server-ca-certs** secret by using your certificate and private key. Run the following command:

   ```
   oc -n open-cluster-management-observability create secret tls observability-server-ca-certs -
   -cert ./serverCACert.pem --key ./serverCAKey.pem
   ```

2. Create the **observability-client-ca-certs** secret by using your certificate and private key. Run the following command:

```
oc -n open-cluster-management-observability create secret tls observability-client-ca-certs --
cert ./clientCACert.pem --key ./clientCAKey.pem
```

1.1.1.2.3. Replacing certificates for alertmanager route

You can replace alertmanager certificates by updating the alertmanager route, if you do not want to use the OpenShift default ingress certificate. Complete the following steps:

1. Examine the observability certificate with the following command:

   ```
   openssl x509  -noout -text -in ./observability.crt
   ```

2. Change the common name (**CN**) on the certificate to **alertmanager**.

3. Change the SAN in the **csr.cnf** configuration file with the hostname for your alertmanager route.

4. Create the two following secrets in the **open-cluster-management-observability** namespace. Run the following command:

   ```
   oc -n open-cluster-management-observability create secret tls alertmanager-byo-ca --cert
   ./ca.crt --key ./ca.key

   oc -n open-cluster-management-observability create secret tls alertmanager-byo-cert --cert
   ./ingress.crt --key ./ingress.key
   ```

For more information, see OpenSSL commands for generating a certificate . If you want to restore the default self-signed certificate for alertmanager route, see Restore the default self-signed certificate for management ingress to delete the two secrets in the **open-cluster-management-observability** namespace.

1.1.2. Red Hat Advanced Cluster Management component certificates

1.1.2.1. List hub cluster managed certificates

You can view a list of hub cluster managed certificates that use OpenShift Service Serving Certificates service internally. Run the following command to list the certificates:

```
for ns in multicluster-engine open-cluster-management ; do echo "$ns:" ; oc get secret -n $ns -o
custom-
columns=Name:.metadata.name,Expiration:.metadata.annotations.service\\.beta\\.openshift\\.io/expiry
| grep -v '<none>' ; echo ""; done
```

Note: If observability is enabled, there are additional namespaces where certificates are created.

1.1.2.2. Refresh hub cluster managed certificates

You can refresh a hub cluster managed certificate by running the **delete secret** command in the List hub cluster managed certificates section. When you identify the certificate that you need to refresh, delete the secret that is associated with the certificate. For example, you can delete a secret by running the following command:

```
oc delete secret grc-0c925-grc-secrets -n open-cluster-management
```

–

Note: After you delete the secret, a new one is created. However, you must restart pods that use the secret manually so they can begin to use the new certificate.

1.1.2.3. Refresh a Red Hat Advanced Cluster Management webhook certificate

You can refresh OpenShift Container Platform managed certificates, which are certificates that are used by Red Hat Advanced Cluster Management webhooks.

Complete the following steps to refresh Red Hat Advanced Cluster Management webhook certificate:

1. Delete the secret that is associated with the OpenShift Container Platform managed certificate by running the following command:

   ```
   oc delete secret -n open-cluster-management ocm-webhook-secret
   ```

   Note: Some services might not have a secret that needs to be deleted.

2. Restart the services that are associated with the OpenShift Container Platform managed certificate(s) by running the following command:

   ```
   oc delete po -n open-cluster-management ocm-webhook-679444669c-5cg76
   ```

   Important: There are replicas of many services; each service must be restarted.

View the following table for a summarized list of the pods that contain certificates and whether a secret needs to be deleted prior to restarting the pod:

Table 1.1. Pods that contain OpenShift Container Platform managed certificates

Service name	Namespace	Sample pod name	Secret name (if applicable)
channels-apps-open-cluster-management-webhook-svc	open-cluster-management	multicluster-operators-application-8c446664c-5lbfk	channels-apps-open-cluster-management-webhook-svc-ca
multicluster-operators-application-svc	open-cluster-management	multicluster-operators-application-8c446664c-5lbfk	multicluster-operators-application-svc-ca
cluster-manager-registration-webhook	open-cluster-management-hub	cluster-manager-registration-webhook-fb7b99c-d8wfc	registration-webhook-serving-cert
cluster-manager-work-webhook	open-cluster-management-hub	cluster-manager-work-webhook-89b8d7fc-f4pv8	work-webhook-serving-cert

1.1.3. Red Hat Advanced Cluster Management managed certificates

1.1.3.1. Channel certificates

CA certificates can be associated with Git channel that are a part of the Red Hat Advanced Cluster Management application management. See Using custom CA certificates for a secure HTTPS connection for more details.

Helm channels allow you to disable certificate validation. Helm channels where certificate validation is disabled, must be configured in development environments. Disabling certificate validation introduces security risks.

1.1.3.2. Managed cluster certificates

Certificates are used to authenticate managed clusters with the hub. Therefore, it is important to be aware of troubleshooting scenarios associated with these certificates. View Troubleshooting imported clusters offline after certificate change for more details.

The managed cluster certificates are refreshed automatically.

1.1.4. Third-party certificates

1.1.4.1. Rotating the gatekeeper webhook certificate

Complete the following steps to rotate the gatekeeper webhook certificate:

1. Edit the secret that contains the certificate with the following command:

   ```
   oc edit secret -n openshift-gatekeeper-system gatekeeper-webhook-server-cert
   ```

2. Delete the following content in the **data** section: **ca.crt**, **ca.key**, tls.crt`, and **tls.key**.

3. Restart the gatekeeper webhook service by deleting the **gatekeeper-controller-manager** pods with the following command:

   ```
   oc delete po -n openshift-gatekeeper-system -l control-plane=controller-manager
   ```

The gatekeeper webhook certificate is rotated.

1.1.4.2. Rotating the integrity shield webhook certificate (Technology Preview)

Complete the following steps to rotate the integrity shield webhook certificate:

1. Edit the IntegrityShield custom resource and add the **integrity-shield-operator-system** namespace to the excluded list of namespaces in the **inScopeNamespaceSelector** setting. Run the following command to edit the resource:

   ```
   oc edit integrityshield integrity-shield-server -n integrity-shield-operator-system
   ```

2. Delete the secret that contains the integrity shield certificate by running the following command:

   ```
   oc delete secret -n integrity-shield-operator-system ishield-server-tls
   ```

3. Delete the operator so that the secret is recreated. Be sure that the operator pod name matches the pod name on your system. Run the following command:

```
oc delete po -n integrity-shield-operator-system integrity-shield-operator-controller-manager-
64549569f8-v4pz6
```

4. Delete the integrity shield server pod to begin using the new certificate with the following command:

   ```
   oc delete po -n integrity-shield-operator-system integrity-shield-server-5fbdfbbbd4-bbfbz
   ```

Use the certificate policy controller to create and manage certificate policies on managed clusters. See Policy controllers to learn more about controllers. Return to the Risk and compliance page for more information.

1.2. REPLACING THE MANAGEMENT INGRESS CERTIFICATES

You can replace management ingress certificates by updating the Red Hat Advanced Cluster Management for Kubernetes route if you do not want to use the OpenShift default ingress certificate.

- Prerequisites to replace management ingress certificate
- Replace the Bring Your Own (BYO) ingress certificate
- Restore the default self-signed certificate for management ingress

1.2.1. Prerequisites to replace management ingress certificate

Prepare and have your **management-ingress** certificates and private keys ready. If needed, you can generate a TLS certificate by using OpenSSL. Set the common name parameter (**CN**) on the certificate to **manangement-ingress**. If you are generating the certificate, include the following settings:

- Include the route name for Red Hat Advanced Cluster Management for Kubernetes as the domain name in your certificate Subject Alternative Name (SAN) list.
  Receive the route name by running the following command:

  ```
  oc get route -n open-cluster-management
  ```

  You might receive the following response:

  ```
  multicloud-console.apps.grchub2.dev08.red-chesterfield.com
  ```

1.2.1.1. Example configuration file for generating a certificate

The following example configuration file and OpenSSL commands provide an example for how to generate a TLS certificate by using OpenSSL. View the following **csr.cnf** configuration file, which defines the configuration settings for generating certificates with OpenSSL.

```
[ req ]              # Main settings
default_bits = 2048       # Default key size in bits.
prompt = no              # Disables prompting for certificate values so the configuration file values are
used.
default_md = sha256       # Specifies the digest algorithm.
req_extensions = req_ext  # Specifies the configuration file section that includes any extensions.
distinguished_name = dn   # Specifies the section that includes the distinguished name information.
```

```
[ dn ]              # Distinguished name settings
C = US                   # Country
ST = North Carolina            # State or province
L = Raleigh                # Locality
O = Red Hat Open Shift     # Organization
OU = Red Hat Advanced Container Management        # Organizational unit
CN = management-ingress  # Common name.

[ req_ext ]         # Extensions
subjectAltName = @alt_names # Subject alternative names

[ alt_names ]       # Subject alternative names
DNS.1 = multicloud-console.apps.grchub2.dev08.red-chesterfield.com

[ v3_ext ]          # x509v3 extensions
authorityKeyIdentifier=keyid,issuer:always  # Specifies the public key that corresponds to the private key that is used to sign a certificate.
basicConstraints=CA:FALSE                  # Indicates whether the certificate is a CA certificate during the certificate chain verification process.
#keyUsage=keyEncipherment,dataEncipherment   # Defines the purpose of the key that is contained in the certificate.
extendedKeyUsage=serverAuth                # Defines the purposes for which the public key can be used.
subjectAltName=@alt_names                  # Identifies the subject alternative names for the identify that is bound to the public key by the CA.
```

Note: Be sure to update the SAN labeled, **DNS.1** with the correct hostname for your management ingress.

1.2.1.2. OpenSSL commands for generating a certificate

The following OpenSSL commands are used with the preceding configuration file to generate the required TLS certificate.

1. Generate your certificate authority (CA) RSA private key:

   ```
   openssl genrsa -out ca.key 4096
   ```

2. Generate a self-signed CA certificate by using your CA key:

   ```
   openssl req -x509 -new -nodes -key ca.key -subj "/C=US/ST=North
   Carolina/L=Raleigh/O=Red Hat OpenShift" -days 400 -out ca.crt
   ```

3. Generate the RSA private key for your certificate:

   ```
   openssl genrsa -out ingress.key 4096
   ```

4. Generate the Certificate Signing request (CSR) by using the private key:

   ```
   openssl req -new -key ingress.key -out ingress.csr -config csr.cnf
   ```

5. Generate a signed certificate by using your CA certificate and key and CSR:

```
openssl x509 -req -in ingress.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out ingress.crt -sha256 -days 300 -extensions v3_ext -extfile csr.cnf
```

6. Examine the certificate contents:

```
openssl x509  -noout -text -in ./ingress.crt
```

1.2.2. Replace the Bring Your Own (BYO) ingress certificate

Complete the following steps to replace your BYO ingress certificate:

1. Create the **byo-ingress-tls** secret by using your certificate and private key. Run the following command:

```
oc -n open-cluster-management create secret tls byo-ingress-tls-secret --cert ./ingress.crt --key ./ingress.key
```

2. Verify that the secret is created in the correct namespace with the following command:

```
oc get secret -n open-cluster-management | grep -e byo-ingress-tls-secret -e byo-ca-cert
```

3. Optional: Create a secret containing the CA certificate by running the following command:

```
oc -n open-cluster-management create secret tls byo-ca-cert --cert ./ca.crt --key ./ca.key
```

4. Delete the **management-ingress** subscription in order to redeploy the subscription. The secrets created in the previous steps are used automatically. Run the following command:

```
oc delete subscription management-ingress-sub -n open-cluster-management
```

5. Verify that the current certificate is your certificate, and that all console access and login functionality remain the same.

1.2.3. Restore the default self-signed certificate for management ingress

1. Delete the bring your own certificate secrets with the following command:

```
oc delete secret byo-ca-cert byo-ingress-tls-secret -n open-cluster-management
```

2. Delete the **management-ingress** subscription in order to redeploy the subscription. The secrets created in the previous steps are used automatically. Run the following command:

```
oc delete subscription management-ingress-sub -n open-cluster-management
```

3. Verify that the current certificate is your certificate, and that all console access and login functionality remain the same.

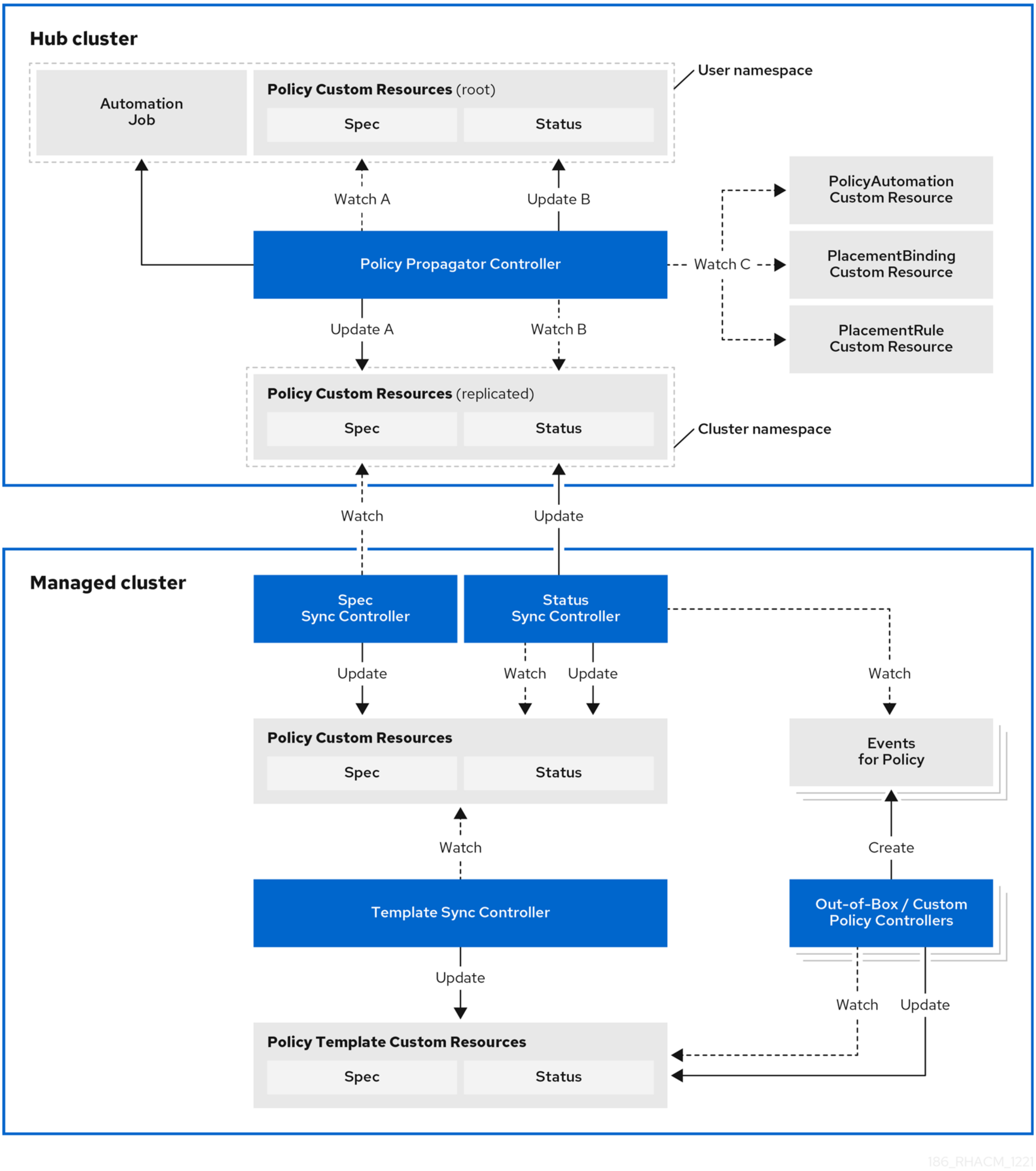
See Certificates for more information about certificates that are created and managed by Red Hat Advanced Cluster Management. Return to the Risk and compliance page for more information on securing your cluster.

CHAPTER 2. GOVERNANCE

Enterprises must meet internal standards for software engineering, secure engineering, resiliency, security, and regulatory compliance for workloads hosted on private, multi and hybrid clouds. Red Hat Advanced Cluster Management for Kubernetes governance provides an extensible policy framework for enterprises to introduce their own security policies.

2.1. GOVERNANCE ARCHITECTURE

Enhance the security for your cluster with the Red Hat Advanced Cluster Management for Kubernetes governance lifecycle. The product governance lifecycle is based on defined policies, processes, and procedures to manage security and compliance from a central interface page. View the following diagram of the governance architecture:



The governance architecture is composed of the following components:

- **Governance dashboard**: Provides a summary of your cloud governance and risk details, which include policy and cluster violations.
  Notes:
  - When a policy is propagated to a managed cluster, the replicated policy is named **namespaceName.policyName**. When you create a policy, make sure that the length of the **namespaceName.policyName** must not exceed 63 characters due to the Kubernetes limit for object names.

  - When you search for a policy in the hub cluster, you might also receive the name of the replicated policy on your managed cluster. For example, if you search for **policy-dhaz-cert**, the following policy name from the hub cluster might appear: **default.policy-dhaz-cert**.

- **Policy-based governance framework:** Supports policy creation and deployment to various managed clusters based on attributes associated with clusters, such as a geographical region. See the **policy-collection** repository to view examples of the predefined policies, and instructions on deploying policies to your cluster. You can also contribute custom policy controllers and policies. When policies are violated, automations can be configured to run and take any action that the user chooses. See Configuring Ansible Tower for governance for more information.
  Use the **policy_governance_info** metric to view trends and analyze any policy failures. See Governance metric for more details.

- **Policy controller**: Evaluates one or more policies on the managed cluster against your specified control and generates Kubernetes events for violations. Violations are propagated to the hub cluster. Policy controllers that are included in your installation are the following: Kubernetes configuration, Certificate, and IAM. You can also create a custom policy controller.

- **Open source community**: Supports community contributions with a foundation of the Red Hat Advanced Cluster Management policy framework. Policy controllers and third-party policies are also a part of the **stolostron/policy-collection** repository. Learn how to contribute and deploy policies using GitOps. For more information, see Deploy policies using GitOps . Learn how to integrate third-party policies with Red Hat Advanced Cluster Management for Kubernetes. For more information, see Integrate third-party policy controllers .

Learn about the structure of an Red Hat Advanced Cluster Management for Kubernetes policy framework, and how to use the Red Hat Advanced Cluster Management for Kubernetes *Governance* dashboard.

- Policy overview
- Policy controllers
- Supported policies
- Manage security policies
- Secure the hub cluster

2.2. POLICY OVERVIEW

Use the Red Hat Advanced Cluster Management for Kubernetes security policy framework to create custom policy controllers and other policies. Kubernetes custom resource definition (CRD) instance are used to create policies. For more information about CRDs, see Extend the Kubernetes API with CustomResourceDefinitions.

Each Red Hat Advanced Cluster Management for Kubernetes policy can have at least one or more templates. For more details about the policy elements, view the following *Policy YAML table* section on this page.

The policy requires a *PlacementRule* or *Placement* that defines the clusters that the policy document is applied to, and a *PlacementBinding* that binds the Red Hat Advanced Cluster Management for Kubernetes policy to the placement rule. For more on how to define a **PlacementRule**, see Placement rules in the Application lifecycle documentation. For more on how to define a **Placement** see Placement overview in the Cluster lifecycle documentation.

Important:

- You must create the **PlacementBinding** and associate it with either the **PlacementRule** or a **Placement**.
  Best practice: Use the command line interface (CLI) to make updates to the policies when you use the **Placement** resource.

- You can create a policy in any namespace on the hub cluster except the cluster namespace. If you create a policy in the cluster namespace, it is deleted by Red Hat Advanced Cluster Management for Kubernetes.

- Each client and provider is responsible for ensuring that their managed cloud environment meets internal enterprise security standards for software engineering, secure engineering, resiliency, security, and regulatory compliance for workloads hosted on Kubernetes clusters. Use the governance and security capability to gain visibility and remediate configurations to meet standards.

Learn more details about the policy components in the following sections:

- Policy YAML structure
- Policy YAML table
- Policy sample file
- Placement YAML sample file

2.2.1. Policy YAML structure

When you create a policy, you must include required parameter fields and values. Depending on your policy controller, you might need to include other optional fields and values. View the following YAML structure for the explained parameter fields:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name:
  annotations:
    policy.open-cluster-management.io/standards:
    policy.open-cluster-management.io/categories:
    policy.open-cluster-management.io/controls:
spec:
  policy-templates:
    - objectDefinition:
        apiVersion:
        kind:
        metadata:
          name:
        spec:
  remediationAction:
  disabled:

---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name:
placementRef:
  name:
  kind:
  apiGroup:
subjects:
- name:
  kind:
  apiGroup:
---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name:
spec:
  clusterConditions:
  - type:
  clusterLabels:
    matchLabels:
      cloud:
```

2.2.2. Policy YAML table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.annotations	Optional. Used to specify a set of security details that describes the set of standards the policy is trying to validate. All annotations documented here are represented as a string that contains a comma-separated list. **Note**: You can view policy violations based on the standards and categories that you define for your policy on the *Policies* page, from the console.
annotations.policy.open-cluster-management.io/standards	The name or names of security standards the policy is related to. For example, National Institute of Standards and Technology (NIST) and Payment Card Industry (PCI).

Field	Description
annotations.policy.open-cluster-management.io/categories	A security control category represent specific requirements for one or more standards. For example, a System and Information Integrity category might indicate that your policy contains a data transfer protocol to protect personal information, as required by the HIPAA and PCI standards.
annotations.policy.open-cluster-management.io/controls	The name of the security control that is being checked. For example, the certificate policy controller.
spec.policy-templates	Required. Used to create one or more policies to apply to a managed cluster.
spec.disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**. If specified, the **spec.remediationAction** value that is defined overrides the **remediationAction** parameter defined in the child policy, from the **policy-templates** section. For example, if **spec.remediationAction** value section is set to **enforce**, then the **remediationAction** in the **policy-templates** section is set to **enforce** during runtime. **Important**: Some policies might not support the enforce feature.

2.2.3. Policy sample file

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-role
  annotations:
    policy.open-cluster-management.io/standards: NIST SP 800-53
    policy.open-cluster-management.io/categories: AC Access Control
    policy.open-cluster-management.io/controls: AC-3 Access Enforcement
spec:
  remediationAction: inform
  disabled: false
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          name: policy-role-example
        spec:
          remediationAction: inform # the policy-template spec.remediationAction is overridden by the preceding parameter value for spec.remediationAction.
          severity: high
          namespaceSelector:
            include: ["default"]
          object-templates:
            - complianceType: mustonlyhave # role definition should exact match
              objectDefinition:
                apiVersion: rbac.authorization.k8s.io/v1
                kind: Role
                metadata:
                  name: sample-role
                rules:
                  - apiGroups: ["extensions", "apps"]
                    resources: ["deployments"]
                    verbs: ["get", "list", "watch", "delete","patch"]
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-policy-role
placementRef:
  name: placement-policy-role
  kind: PlacementRule
  apiGroup: apps.open-cluster-management.io
subjects:
- name: policy-role
  kind: Policy
  apiGroup: policy.open-cluster-management.io
---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-policy-role
spec:
  clusterConditions:
  - status: "True"
    type: ManagedClusterConditionAvailable
  clusterSelector:
    matchExpressions:
      - {key: environment, operator: In, values: ["dev"]}
```

2.2.4. Placement YAML sample file

The **PlacementBinding** and **Placement** resources can be combined with the previous policy example to deploy the policy using the cluster **Placement** API instead of the **PlacementRule** API.

```
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-policy-role
```

```
placementRef:
  name: placement-policy-role
  kind: Placement
  apiGroup: cluster.open-cluster-management.io
subjects:
- name: policy-role
  kind: Policy
  apiGroup: policy.open-cluster-management.io
---
//Depends on if governance would like to use v1beta1
apiVersion: cluster.open-cluster-management.io/v1beta1
kind: Placement
metadata:
  name: placement-policy-role
spec:
  predicates:
  - requiredClusterSelector:
      labelSelector:
        matchExpressions:
          - {key: environment, operator: In, values: ["dev"]}
```

See Managing security policies to create and update a policy. You can also enable and updateRed Hat Advanced Cluster Management policy controllers to validate the compliance of your policies. Refer to Policy controllers.

To learn more policy topics, see Governance.

2.3. POLICY CONTROLLERS

Policy controllers monitor and report whether your cluster is compliant with a policy. Use the Red Hat Advanced Cluster Management for Kubernetes policy framework by using the out-of-the-box policy templates to apply predefined policy controllers and policies. The policy controllers are Kubernetes custom resource definition (CRD) instances.

For more information about CRDs, see Extend the Kubernetes API with CustomResourceDefinitions. Policy controllers remediate policy violations to make the cluster status compliant.

You can create custom policies and policy controllers with the product policy framework. See Creating a custom policy controller (deprecated) for more information.

View the following topics to learn more about the following Red Hat Advanced Cluster Management for Kubernetes policy controllers:

- Kubernetes configuration policy controller
- Certificate policy controller
- IAM policy controller
- Policy set controller

Important: Only the configuration policy controller policies support the **enforce** feature. You must manually remediate policies, where the policy controller does not support the **enforce** feature.

Refer to Governance for more topics about managing your policies.

2.3.1. Kubernetes configuration policy controller

Configuration policy controller can be used to configure any Kubernetes resource and apply security policies across your clusters.

The configuration policy controller communicates with the local Kubernetes API server to get the list of your configurations that are in your cluster. For more information about CRDs, see Extend the Kubernetes API with CustomResourceDefinitions.

The configuration policy controller is created on the hub cluster during installation. Configuration policy controller supports the **enforce** feature and monitors the compliance of the following policies:

- Memory usage policy
- Namespace policy
- Image vulnerability policy
- Pod policy
- Pod security policy
- Role policy
- Role binding policy
- Security content constraints (SCC) policy
- ETCD encryption policy
- Compliance operator policy
- Integrating gatekeeper constraints and constraint templates

When the **remediationAction** for the configuration policy is set to **enforce**, the controller creates a replicate policy on the target managed clusters. You can also use templates in configuration policies. For more information, see Support for templates in configuration policies .

Continue reading to learn more about the configuration policy controller:

- Configuration policy controller YAML structure
- Configuration policy sample
- Configuration policy YAML table

2.3.1.1. Configuration policy controller YAML structure

```
Name:         configuration-policy-example
Namespace:
Labels:
APIVersion:   policy.open-cluster-management.io/v1
Kind:         ConfigurationPolicy
Metadata:
  Finalizers:
    finalizer.policy.open-cluster-management.io
Spec:
  Conditions:
    Ownership:
    NamespaceSelector:
      Exclude:
      Include:
    RemediationAction:
 Status:
   CompliancyDetails:
     Configuration-Policy-Example:
       Default:
       Kube - Public:
   Compliant:          Compliant
 Events:
```

2.3.1.2. Configuration policy sample

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: policy-config
spec:
  namespaceSelector:
    include: ["default"]
    exclude: []
  remediationAction: inform
    severity: low
    object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: v1
        kind: Pod
        metadata:
          name: pod
        spec:
          containers:
          - image: 'pod-image'
            name:
            ports:
           - containerPort: 80
  evaluationInterval:
    compliant:
    noncompliant:
```

2.3.1.3. Configuration policy YAML table

Table 2.1. Parameter table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.

Field	Description
kind	Required. Set the value to **ConfigurationPolicy** to indicate the type of policy.
metadata.name	Required. The name of the policy.
spec	Required. Specifications of which configuration policy to monitor and how to remediate them.
spec.namespace	Required for namespaced objects or resources. The namespaces in the hub cluster that the policy is applied to. Enter at least one namespace for the **include** parameter, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to.
spec.remediationAction	Required. Specifies the remediation of your policy. Enter **inform**
spec.remediationAction.severity	Required. Specifies the severity when the policy is non-compliant. Use the following parameter values: **low**, **medium**, or **high**.
spec.remediationAction.complianceType	Required. Used to list expected behavior for roles and other Kubernetes object that must be evaluated or applied to the managed clusters. You must use the following verbs as parameter values: **mustonlyhave**: Indicates that an object must exist with the exact name and relevant fields. **musthave**: Indicates an object must exist with the same name as specified object-template. The other fields in the template are a subset of what exists in the object. **mustnothave**: Indicated that an object with the same name or labels cannot exist and need to be deleted, regardless of the specification or rules.
spec.evaluationInterval.compliant	Optional. Used to define how often the policy is evaluated when it is in the compliant state. The values must be in the format of a duration which is a sequence of numbers with time unit suffixes. For example, **12h30m5s** represents 12 hours, 30 minutes, and 5 seconds. It can also be set to **never** so that the policy is not reevaluated on the compliant cluster, unless the policy **spec** is updated.

Field	Description
spec.evaluationInterval.noncompliant	Optional. Used to define how often the policy is evaluated when it is in the non-compliant state. Similar to the **evaluationInterval.compliant** parameter, the values must be in the format of a duration which is a sequence of numbers with time unit suffixes. It can also be set to **never** so that the policy is not reevaluated on the noncompliant cluster, unless the policy **spec** is updated.

See the policy samples that use NIST Special Publication 800-53 (Rev. 4) , and are supported by Red Hat Advanced Cluster Management from the **CM-Configuration-Management** folder. Learn about how policies are applied on your hub cluster, see Supported policies for more details.

Learn how to create and customize policies, see Manage security policies . Refer to Policy controllers for more details about controllers.

2.3.2. Certificate policy controller

Certificate policy controller can be used to detect certificates that are close to expiring, and detect time durations (hours) that are too long or contain DNS names that fail to match specified patterns.

Configure and customize the certificate policy controller by updating the following parameters in your controller policy:

- **minimumDuration**
- **minimumCADuration**
- **maximumDuration**
- **maximumCADuration**
- **allowedSANPattern**
- **disallowedSANPattern**

Your policy might become non-compliant due to either of the following scenarios:

- When a certificate expires in less than the minimum duration of time or exceeds the maximum time.
- When DNS names fail to match the specified pattern.

The certificate policy controller is created on your managed cluster. The controller communicates with the local Kubernetes API server to get the list of secrets that contain certificates and determine all non-compliant certificates. For more information about CRDs, see Extend the Kubernetes API with CustomResourceDefinitions.

Certificate policy controller does not support the **enforce** feature.

2.3.2.1. Certificate policy controller YAML structure

View the following example of a certificate policy and review the element in the YAML table:

```
apiVersion: policy.open-cluster-management.io/v1
kind: CertificatePolicy
metadata:
  name: certificate-policy-example
  namespace:
  labels: category=system-and-information-integrity
spec:
  namespaceSelector:
    include: ["default"]
  remediationAction:
  severity:
  minimumDuration:
  minimumCADuration:
  maximumDuration:
  maximumCADuration:
  allowedSANPattern:
  disallowedSANPattern:
```

2.3.2.1.1. Certificate policy controller YAML table

Table 2.2. Parameter table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **CertificatePolicy** to indicate the type of policy.
metadata.name	Required. The name to identify the policy.
metadata.namespace	Required. The namespaces within the managed cluster where the policy is created.
metadata.labels	Optional. In a certificate policy, the **category=system-and-information-integrity** label categorizes the policy and facilitates querying the certificate policies. If there is a different value for the **category** key in your certificate policy, the value is overridden by the certificate controller.
spec	Required. Specifications of which certificates to monitor and refresh.

Field	Description
spec.namespaceSelector	Required. Managed cluster namespace to which you want to apply the policy. Enter parameter values for **Include** and **Exclude**. **Notes**: • When you create multiple certificate policies and apply them to the same managed cluster, each policy **namespaceSelector** must be assigned a different value. • If the **namespaceSelector** for the certificate policy controller does not match any namespace, the policy is considered compliant.
spec.remediationAction	Required. Specifies the remediation of your policy. Set the parameter value to **inform**. Certificate policy controller only supports **inform** feature.
spec.severity	Optional. Informs the user of the severity when the policy is non-compliant. Use the following parameter values: **low**, **medium**, or **high**.
spec.minimumDuration	Required. When a value is not specified, the default value is **100h**. This parameter specifies the smallest duration (in hours) before a certificate is considered non-compliant. The parameter value uses the time duration format from Golang. See Golang Parse Duration for more information.
spec.minimumCADuration	Optional. Set a value to identify signing certificates that might expire soon with a different value from other certificates. If the parameter value is not specified, the CA certificate expiration is the value used for the **minimumDuration**. See Golang Parse Duration for more information.
spec.maximumDuration	Optional. Set a value to identify certificates that have been created with a duration that exceeds your desired limit. The parameter uses the time duration format from Golang. See Golang Parse Duration for more information.
spec.maximumCADuration	Optional. Set a value to identify signing certificates that have been created with a duration that exceeds your defined limit. The parameter uses the time duration format from Golang. See Golang Parse Duration for more information.

Field	Description
spec.allowedSANPattern	Optional. A regular expression that must match every SAN entry that you have defined in your certificates. This parameter checks DNS names against patterns. See the Golang Regular Expression syntax for more information.
spec.disallowedSANPattern	Optional. A regular expression that must not match any SAN entries you have defined in your certificates. This parameter checks DNS names against patterns. **Note**: To detect wild-card certificate, use the following SAN pattern: **disallowedSANPattern: "[\\*]"** See the Golang Regular Expression syntax for more information.

2.3.2.2. Certificate policy sample

When your certificate policy controller is created on your hub cluster, a replicated policy is created on your managed cluster. See **policy-certificate.yaml** to view the certificate policy sample.

Learn how to manage a certificate policy, see Managing security policies for more details. Refer to Policy controllers for more topics.

2.3.3. IAM policy controller

The Identity and Access Management (IAM) policy controller can be used to receive notifications about IAM policies that are non-compliant. The compliance check is based on the parameters that you configure in the IAM policy.

The IAM policy controller monitors for the desired maximum number of users with a particular cluster role (i.e. **ClusterRole**) in your cluster. The default cluster role to monitor is **cluster-admin**. The IAM policy controller communicates with the local Kubernetes API server. For more information, see Extend the Kubernetes API with CustomResourceDefinitions.

The IAM policy controller runs on your managed cluster. View the following sections to learn more:

- IAM policy YAML structure
- IAM policy YAML table
- IAM policy sample

2.3.3.1. IAM policy YAML structure

View the following example of an IAM policy and review the parameters in the YAML table:

```
apiVersion: policy.open-cluster-management.io/v1
kind: IamPolicy
metadata:
    name:
  spec:
    clusterRole:
    severity:
    remediationAction:
    maxClusterRoleBindingUsers:
    ignoreClusterRoleBindings:
```

2.3.3.2. IAM policy YAML table

View the following parameter table for descriptions:

Table 2.3. Parameter table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
spec	Required. Add configuration details for your policy.
spec.clusterRole	Optional. The cluster role (i.e. **ClusterRole**) to monitor. This defaults to **cluster-admin** if not specified.
spec.severity	Optional. Informs the user of the severity when the policy is non-compliant. Use the following parameter values: **low**, **medium**, or **high**.
spec.remediationAction	Optional. Specifies the remediation of your policy. Enter **inform**.
spec.ignoreClusterRoleBindings	Optional. A list of regular expression (regex) values that indicate which cluster role binding names to ignore. These regular expression values must follow Go regexp syntax. By default, all cluster role bindings that have a name that starts with **system:** are ignored. It is recommended to set this to a stricter value. To not ignore any cluster role binding names, set the list to a single value of **.^** or some other regular expression that never matches.
spec.maxClusterRoleBindingUsers	Required. Maximum number of IAM role bindings that are available before a policy is considered non-compliant.

2.3.3.3. IAM policy sample

See **policy-limitclusteradmin.yaml** to view the IAM policy sample. See Managing security policies for more information.

Refer to Policy controllers for more topics.

2.3.4. Policy set controller

The policy set controller aggregates the policy status scoped to policies that are defined in the same namespace. Create a policy set (**PolicySet**) to group policies that in the same namespace. All policies in the **PolicySet** are placed together in a selected cluster by creating a **PlacementBinding** to bind the **PolicySet** and **Placement**. The policy set is deployed to the hub cluster.

Additionally, when a policy is a part of multiple policy sets, existing and new **Placement** resources remain in the policy. When a user removes a policy from the policy set, the policy is not applied to the cluster that is selected in the policy set, but the placements remain. The policy set controller only checks for violations in clusters that include the policy set placement.

Note: The Red Hat Advanced Cluster Management hardening sample policy set uses cluster placement. If you use cluster placement, bind the namespace containing the policy to the managed cluster set. See Deploying policies to your cluster for more details on using cluster placement.

Learn more details about the policy set structure in the following sections:

- Policy set controller YAML structure
- Policy set controller YAML table
- Policy set sample

2.3.4.1. Policy set YAML structure

Your policy set might resemble the following YAML file:

```
apiVersion: policy.open-cluster-management.io/v1beta1
kind: PolicySet
metadata:
  name: demo-policyset
spec:
  policies:
  - policy-demo

---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: demo-policyset-pb
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: demo-policyset-pr
subjects:
- apiGroup: policy.open-cluster-management.io
  kind: PolicySet
  name: demo-policyset
```

```
---
apiVersion: apps.open-cluster-management.io
kind: PlacementRule
metadata:
  name: demo-policyset-pr
spec:
  clusterConditions:pagewidth:
  - status: "True"
    type: ManagedCLusterConditionAvailable
  clusterSelectors:
    matchExpressions:
      - key: name
        operator: In
        values:
          - local-cluster
```

2.3.4.2. Policy set table

View the following parameter table for descriptions:

Table 2.4. Parameter table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1beta1**.
kind	Required. Set the value to **PolicySet** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
spec	Required. Add configuration details for your policy.
spec.policies	Optional. The list of policies that you want to group together in the policy set.

2.3.4.3. Policy set sample

```
apiVersion: policy.open-cluster-management.io/v1beta1
kind: PolicySet
metadata:
  name: pci
  namespace: default
spec:
  description: Policies for PCI compliance
  policies:
  - policy-pod
  - policy-namespace
status:
  compliant: NonCompliant
```

```
placement:
- placementBinding: binding1
  placementRule: placement1
  policySet: policyset-ps
```

See the *Creating policy sets* section in the Managing security policies topic. Also view the stable **PolicySets**, which require the policy generator for deployment, PolicySets-- Stable. See the Policy generator documentation.

2.3.5. Creating a custom policy controller (deprecated)

Learn to write, apply, view, and update your custom policy controllers. You can create a YAML file for your policy controller to deploy onto your cluster. View the following sections to create a policy controller:

2.3.5.1. Writing a policy controller

Use the policy controller framework that is in the **governance-policy-framework** repository. Complete the following steps to create a policy controller:

1. Clone the **governance-policy-framework** repository by running the following command:

   ```
   git clone git@github.com:stolostron/governance-policy-framework.git
   ```

2. Customize the controller policy by updating the policy schema definition. Your policy might resemble the following content:

   ```
   metadata:
     name: samplepolicies.policies.open-cluster-management.io
   spec:
     group: policy.open-cluster-management.io
     names:
       kind: SamplePolicy
       listKind: SamplePolicyList
       plural: samplepolicies
       singular: samplepolicy
   ```

3. Update the policy controller to watch for the **SamplePolicy** kind. Run the following command:

   ```
   for file in $(find . -name "*.go" -type f); do  sed -i "" "s/SamplePolicy/g" $file; done
   for file in $(find . -name "*.go" -type f); do  sed -i "" "s/samplepolicy-controller/samplepolicy-
   controller/g" $file; done
   ```

4. Recompile and run the policy controller by completing the following steps:

   a. Log in to your cluster.

   b. Select the user icon, then click **Configure client**.

   c. Copy and paste the configuration information into your command line, and press **Enter**.

   d. Run the following commands to apply your policy CRD and start the controller:

      ```
      export GO111MODULE=on
      ```

```
kubectl apply -f deploy/crds/policy.open-cluster-management.io_samplepolicies_crd.yaml

export WATCH_NAMESPACE=<cluster_namespace_on_hub>

go run cmd/manager/main.go
```

You might receive the following output that indicates that your controller runs:

```
{“level”:”info”,”ts”:1578503280.511274,”logger”:”controller-
runtime.manager”,”msg”:”starting metrics server”,”path”:”/metrics”}
{“level”:”info”,”ts”:1578503281.215883,”logger”:”controller-
runtime.controller”,”msg”:”Starting Controller”,”controller”:”samplepolicy-controller”}
{“level”:”info”,”ts”:1578503281.3203468,”logger”:”controller-
runtime.controller”,”msg”:”Starting workers”,”controller”:”samplepolicy-controller”,”worker
count”:1}
Waiting for policies to be available for processing…
```

e. Create a policy and verify that the controller retrieves it and applies the policy onto your cluster. Run the following command:

```
kubectl apply -f deploy/crds/policy.open-cluster-management.io_samplepolicies_crd.yaml
```

When the policy is applied, a message appears to indicate that policy is monitored and detected by your custom controller. The message might resemble the following contents:

```
{"level":"info","ts":1578503685.643426,"logger":"controller_samplepolicy","msg":"Reconciling
SamplePolicy","Request.Namespace":"default","Request.Name":"example-samplepolicy"}
{"level":"info","ts":1578503685.855259,"logger":"controller_samplepolicy","msg":"Reconciling
SamplePolicy","Request.Namespace":"default","Request.Name":"example-samplepolicy"}
Available policies in namespaces:
namespace = kube-public; policy = example-samplepolicy
namespace = default; policy = example-samplepolicy
namespace = kube-node-lease; policy = example-samplepolicy
```

5. Check the **status** field for compliance details by running the following command:

```
kubectl describe SamplePolicy example-samplepolicy -n default
```

Your output might resemble the following contents:

```
status:
  compliancyDetails:
    example-samplepolicy:
      cluster-wide:
      - 5 violations detected in namespace `cluster-wide`, there are 0 users violations
        and 5 groups violations
      default:
      - 0 violations detected in namespace `default`, there are 0 users violations
        and 0 groups violations
      kube-node-lease:
      - 0 violations detected in namespace `kube-node-lease`, there are 0 users violations
        and 0 groups violations
      kube-public:
```

```
    - 1 violations detected in namespace `kube-public`, there are 0 users violations
      and 1 groups violations
  compliant: NonCompliant
```

6. Change the policy rules and policy logic to introduce new rules for your policy controller. Complete the following steps:

   a. Add new fields in your YAML file by updating the **SamplePolicySpec**. Your specification might resemble the following content:

   ```
   spec:
     description: SamplePolicySpec defines the desired state of SamplePolicy
     properties:
       labelSelector:
         additionalProperties:
           type: string
         type: object
       maxClusterRoleBindingGroups:
         type: integer
       maxClusterRoleBindingUsers:
         type: integer
       maxRoleBindingGroupsPerNamespace:
         type: integer
       maxRoleBindingUsersPerNamespace:
         type: integer
   ```

   b. Update the **SamplePolicySpec** structure in the samplepolicy_controller.go with new fields.

   c. Update the **PeriodicallyExecSamplePolicies** function in the **samplepolicy_controller.go** file with new logic to run the policy controller. View an example of the **PeriodicallyExecSamplePolicies** field, see stolostron/multicloud-operators-policy-controller.

   d. Recompile and run the policy controller. See Writing a policy controller

Your policy controller is functional.

2.3.5.2. Deploying your controller to the cluster

Deploy your custom policy controller to your cluster and integrate the policy controller with the *Governance* dashboard. Complete the following steps:

1. Build the policy controller image by running the following command:

   ```
   make build
   docker build . -f build/Dockerfile -t <username>/multicloud-operators-policy-controller:latest
   ```

2. Run the following command to push the image to a repository of your choice. For example, run the following commands to push the image to Docker Hub:

   ```
   docker login

   docker push <username>/multicloud-operators-policy-controller
   ```

3. Configure **kubectl** to point to a cluster managed by Red Hat Advanced Cluster Management for Kubernetes.

4. Replace the operator manifest to use the built-in image name and update the namespace to watch for policies. The namespace must be the cluster namespace. Your manifest might resemble the following contents:

```
sed -i "" 's|stolostron/multicloud-operators-policy-controller|ycao/multicloud-operators-policy-controller|g' deploy/operator.yaml
sed -i "" 's|value: default|value: <namespace>|g' deploy/operator.yaml
```

5. Update the RBAC role by running the following commands:

```
sed -i "" 's|samplepolicies|testpolicies|g' deploy/cluster_role.yaml
sed -i "" 's|namespace: default|namespace: <namespace>|g'
deploy/cluster_role_binding.yaml
```

6. Deploy your policy controller to your cluster:

   a. Set up a service account for cluster by running the following command:

   ```
   kubectl apply -f deploy/service_account.yaml -n <namespace>
   ```

   b. Set up RBAC for the operator by running the following commands:

   ```
   kubectl apply -f deploy/role.yaml -n <namespace>

   kubectl apply -f deploy/role_binding.yaml -n <namespace>
   ```

   c. Set up RBAC for your policy controller. Run the following commands:

   ```
   kubectl apply -f deploy/cluster_role.yaml
   kubectl apply -f deploy/cluster_role_binding.yaml
   ```

   d. Set up a custom resource definition (CRD) by running the following command:

   ```
   kubectl apply -f deploy/crds/policies.open-cluster-management.io_samplepolicies_crd.yaml
   ```

   e. Deploy the **multicloud-operator-policy-controller** by running the following command:

   ```
   kubectl apply -f deploy/operator.yaml -n <namespace>
   ```

   f. Verify that the controller is functional by running the following command:

   ```
   kubectl get pod -n <namespace>
   ```

7. You must integrate your policy controller by creating a **policy-template** for the controller to monitor. For more information, see Creating a cluster security policy from the console .

2.3.5.2.1. Scaling your controller deployment

Policy controller deployments do not support deletion or removal. You can scale your deployment to

Policy controller deployments do not support deletion or removal. You can scale your deployment to update which pods the deployment is applied to. Complete the following steps:

1. Log in to your managed cluster.
2. Navigate to the deployment for your custom policy controller.
3. Scale the deployment. When you scale your deployment to zero pods, the policy controler deployment is disabled.

For more information on deployments, see OpenShift Container Platform Deployments.

Your policy controller is deployed and integrated on your cluster. View the product policy controllers, see Policy controllers for more information.

2.4. INTEGRATE THIRD-PARTY POLICY CONTROLLERS

Integrate third-party policies to create custom annotations within the policy templates to specify one or more compliance standards, control categories, and controls.

You can also use the third-party party policies from the policy-collection/community.

Learn to integrate the following third-party policies:

- Integrating gatekeeper constraints and constraint templates
- Policy generator

2.4.1. Integrating gatekeeper constraints and constraint templates

Gatekeeper is a validating webhook that enforces custom resource definition (CRD) based policies that are run with the Open Policy Agent (OPA). You can install gatekeeper on your cluster by using the gatekeeper operator policy. Gatekeeper policy can be used to evaluate Kubernetes resource compliance. You can leverage a OPA as the policy engine, and use Rego as the policy language.

The gatekeeper policy is created as a Kubernetes configuration policy in Red Hat Advanced Cluster Management. Gatekeeper policies include constraint templates (**ConstraintTemplates**) and **Constraints**, audit templates, and admission templates. For more information, see the Gatekeeper upstream repository.

Red Hat Advanced Cluster Management supports version 3.3.0 for Gatekeeper and applies the following constraint templates in your Red Hat Advanced Cluster Management gatekeeper policy:

- **ConstraintTemplates** and constraints: Use the **policy-gatekeeper-k8srequiredlabels** policy to create a gatekeeper constraint template on the managed cluster.

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: policy-gatekeeper-k8srequiredlabels
spec:
  remediationAction: enforce # will be overridden by remediationAction in parent policy
  severity: low
  object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: templates.gatekeeper.sh/v1beta1
        kind: ConstraintTemplate
        metadata:
          name: k8srequiredlabels
        spec:
          crd:
            spec:
              names:
                kind: K8sRequiredLabels
              validation:
                # Schema for the `parameters` field
                openAPIV3Schema:
                  properties:
                    labels:
                      type: array
                      items: string
          targets:
            - target: admission.k8s.gatekeeper.sh
              rego: |
                package k8srequiredlabels
                violation[{"msg": msg, "details": {"missing_labels": missing}}] {
                  provided := {label | input.review.object.metadata.labels[label]}
                  required := {label | label := input.parameters.labels[_]}
                  missing := required - provided
                  count(missing) > 0
                  msg := sprintf("you must provide labels: %v", [missing])
                }
    - complianceType: musthave
      objectDefinition:
        apiVersion: constraints.gatekeeper.sh/v1beta1
        kind: K8sRequiredLabels
        metadata:
          name: ns-must-have-gk
        spec:
          match:
            kinds:
              - apiGroups: [""]
                kinds: ["Namespace"]
            namespaces:
              - e2etestsuccess
              - e2etestfail
          parameters:
            labels: ["gatekeeper"]
```

- audit template: Use the **policy-gatekeeper-audit** to periodically check and evaluate existing resources against the gatekeeper policies that are enforced to detect existing misconfigurations.

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: policy-gatekeeper-audit
spec:
  remediationAction: inform # will be overridden by remediationAction in parent policy
  severity: low
  object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: constraints.gatekeeper.sh/v1beta1
        kind: K8sRequiredLabels
        metadata:
          name: ns-must-have-gk
        status:
          totalViolations: 0
```

- admission template: Use the **policy-gatekeeper-admission** to check for misconfigurations that are created by the gatekeeper admission webhook:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: policy-gatekeeper-admission
spec:
  remediationAction: inform # will be overridden by remediationAction in parent policy
  severity: low
  object-templates:
    - complianceType: mustnothave
      objectDefinition:
        apiVersion: v1
        kind: Event
        metadata:
          namespace: openshift-gatekeeper-system # set it to the actual namespace where
gatekeeper is running if different
          annotations:
            constraint_action: deny
            constraint_kind: K8sRequiredLabels
            constraint_name: ns-must-have-gk
            event_type: violation
```

See **policy-gatekeeper-sample.yaml** for more details.

See Managing configuration policies for more information about managing other policies. Refer to Governance for more topics on the security framework.

2.4.2. Policy generator

The policy generator is a part of the Red Hat Advanced Cluster Management for Kubernetes application lifecycle subscription GitOps workflow that generates Red Hat Advanced Cluster Management for Kubernetes policies using Kustomize. The policy generator builds Red Hat Advanced Cluster Management for Kubernetes policies from Kubernetes manifest YAML files, which are provided through a **PolicyGenerator** manifest YAML file that is used to configure it. The policy generator is implemented as a Kustomize generator plugin. For more information on Kustomize, see the Kustomize documentation.

The policy generator version bundled in this version of Red Hat Advanced Cluster Management is v1.8.0.

2.4.2.1. Policy generator capabilities

The policy generator and its integration with the Red Hat Advanced Cluster Management application lifecycle subscription GitOps workflow simplifies the distribution of Kubernetes resource objects to managed OpenShift clusters, and Kubernetes clusters through Red Hat Advanced Cluster Management policies. In particular, use the policy generator to complete the following actions:

- Convert any Kubernetes manifest files to Red Hat Advanced Cluster Management configuration policies.
- Patch the input Kubernetes manifests before they are inserted into a generated Red Hat Advanced Cluster Management policy.
- Generate additional configuration policies to be able to report on Gatekeeper and Kyverno policy violations through Red Hat Advanced Cluster Management for Kubernetes.
- Generate policy sets on the hub cluster. See Policy set controller for more details.

View the following topics to for more information:

- Policy generator configuration structure
- Generating a policy to install an Operator
  - A policy to install OpenShift GitOps
  - A policy to install the Compliance Operator
- Install the policy generator on OpenShift GitOps (ArgoCD)
- Policy generator configuration reference table

2.4.2.2. Policy generator configuration structure

The policy generator is a Kustomize generator plugin that is configured with a manifest of the **PolicyGenerator** kind and **policy.open-cluster-management.io/v1** API version.

To use the plugin, start by adding a **generators** section in a **kustomization.yaml** file. View the following example:

```
generators:
  - policy-generator-config.yaml
```

The **policy-generator-config.yaml** file referenced in the previous example is a YAML file with the instructions of the policies to generate. A simple policy generator configuration file might resemble the following example:

```
apiVersion: policy.open-cluster-management.io/v1
kind: PolicyGenerator
metadata:
  name: config-data-policies
policyDefaults:
  namespace: policies
  policySets: []
policies:
  - name: config-data
    manifests:
      - path: configmap.yaml
```

The **configmap.yaml** represents a Kubernetes manifest YAML file to be included in the policy. View the following example:

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: my-config
  namespace: default
data:
  key1: value1
  key2: value2
```

The generated **Policy**, along with the generated **PlacementRule** and **PlacementBinding** might resemble the following example:

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-config-data
  namespace: policies
spec:
  clusterConditions:
  - status: "True"
    type: ManagedClusterConditionAvailable
  clusterSelector:
    matchExpressions: []
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-config-data
  namespace: policies
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: placement-config-data
subjects:
- apiGroup: policy.open-cluster-management.io
  kind: Policy
  name: config-data
---
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  annotations:
    policy.open-cluster-management.io/categories: CM Configuration Management
    policy.open-cluster-management.io/controls: CM-2 Baseline Configuration
    policy.open-cluster-management.io/standards: NIST SP 800-53
  name: config-data
  namespace: policies
spec:
  disabled: false
  policy-templates:
  - objectDefinition:
      apiVersion: policy.open-cluster-management.io/v1
      kind: ConfigurationPolicy
      metadata:
        name: config-data
      spec:
        object-templates:
```

```
        - complianceType: musthave
          objectDefinition:
            apiVersion: v1
            data:
              key1: value1
              key2: value2
            kind: ConfigMap
            metadata:
              name: my-config
              namespace: default
        remediationAction: inform
        severity: low
```

See the **policy-generator-plugin** repository for more details.

2.4.2.3. Generating a policy to install an Operator

A common use of Red Hat Advanced Cluster Management policies is to install an Operator on one or more managed OpenShift clusters. View the following examples of the different installation modes and the required resources.

2.4.2.3.1. A policy to install OpenShift GitOps

This example shows how to generate a policy that installs OpenShift GitOps using the policy generator. The OpenShift GitOps operator offers the *all namespaces* installation mode. First, a **Subscription** manifest file called **openshift-gitops-subscription.yaml** needs to be created like the following example.

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-gitops-operator
  namespace: openshift-operators
spec:
  channel: stable
  name: openshift-gitops-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

To pin to a specific version of the operator, you can add the following parameter and value: **spec.startingCSV: openshift-gitops-operator.v<version>**. Replace **<version>** with your preferred version.

Next, a policy generator configuration file called **policy-generator-config.yaml** is required. The following example shows a single policy that installs OpenShift GitOps on all OpenShift managed clusters:

```
apiVersion: policy.open-cluster-management.io/v1
kind: PolicyGenerator
metadata:
  name: install-openshift-gitops
policyDefaults:
  namespace: policies
  placement:
    clusterSelectors:
      vendor: "OpenShift"
  remediationAction: enforce
policies:
  - name: install-openshift-gitops
    manifests:
      - path: openshift-gitops-subscription.yaml
```

The last file that is required is the **kustomization.yaml** file. The **kustomization.yaml** file requires the following configuration:

```
generators:
  - policy-generator-config.yaml
```

The generated policy might resemble the following file:

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-install-openshift-gitops
  namespace: policies
spec:
  clusterConditions:
    - status: "True"
      type: ManagedClusterConditionAvailable
  clusterSelector:
    matchExpressions:
      - key: vendor
        operator: In
        values:
          - OpenShift
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-install-openshift-gitops
  namespace: policies
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: placement-install-openshift-gitops
subjects:
  - apiGroup: policy.open-cluster-management.io
    kind: Policy
    name: install-openshift-gitops
---
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  annotations:
    policy.open-cluster-management.io/categories: CM Configuration Management
    policy.open-cluster-management.io/controls: CM-2 Baseline Configuration
    policy.open-cluster-management.io/standards: NIST SP 800-53
  name: install-openshift-gitops
  namespace: policies
spec:
```

```
disabled: false
policy-templates:
 - objectDefinition:
   apiVersion: policy.open-cluster-management.io/v1
   kind: ConfigurationPolicy
   metadata:
    name: install-openshift-gitops
   spec:
    object-templates:
     - complianceType: musthave
      objectDefinition:
       apiVersion: operators.coreos.com/v1alpha1
       kind: Subscription
       metadata:
        name: openshift-gitops-operator
        namespace: openshift-operators
       spec:
        channel: stable
        name: openshift-gitops-operator
        source: redhat-operators
        sourceNamespace: openshift-marketplace
    remediationAction: enforce
    severity: low
```

All policies where the input is from the OpenShift Container Platform documentation and are generated by the policy generator are fully supported. View the following examples of YAML input that is supported in the OpenShift Container Platform documentation:

- Post-installation cluster tasks
- Configuring the audit log policy
- About forwarding logs to third-party systems

See Understanding OpenShift GitOps and the Operator documentation for more details.

2.4.2.3.2. A policy to install the Compliance Operator

For an operator that uses the *namespaced* installation mode, such as the Compliance Operator, an **OperatorGroup** manifest is also required. This example shows a generated policy to install the Compliance Operator.

First, a YAML file with a **Namespace**, a **Subscription**, and an **OperatorGroup** manifest called **compliance-operator.yaml** must be created. The following example installs these manifests in the **compliance-operator** namespace:

```
apiVersion: v1
kind: Namespace
metadata:
 name: openshift-compliance
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: compliance-operator
 namespace: openshift-compliance
```

```
spec:
  channel: release-0.1
  name: compliance-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: compliance-operator
  namespace: openshift-compliance
spec:
  targetNamespaces:
    - compliance-operator
```

Next, a policy generator configuration file called **policy-generator-config.yaml** is required. The following example shows a single policy that installs the Compliance Operator on all OpenShift managed clusters:

```
apiVersion: policy.open-cluster-management.io/v1
kind: PolicyGenerator
metadata:
  name: install-compliance-operator
policyDefaults:
  namespace: policies
  placement:
    clusterSelectors:
      vendor: "OpenShift"
  remediationAction: enforce
policies:
  - name: install-compliance-operator
    manifests:
      - path: compliance-operator.yaml
```

The last file that is required is the **kustomization.yaml** file. The following configuration is required in the **kustomization.yaml** file:

```
generators:
  - policy-generator-config.yaml
```

As a result, the generated policy should resemble the following file:

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-install-compliance-operator
  namespace: policies
spec:
  clusterConditions:
    - status: "True"
      type: ManagedClusterConditionAvailable
  clusterSelector:
    matchExpressions:
      - key: vendor
        operator: In
        values:
          - OpenShift
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-install-compliance-operator
  namespace: policies
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: placement-install-compliance-operator
subjects:
  - apiGroup: policy.open-cluster-management.io
    kind: Policy
    name: install-compliance-operator
---
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  annotations:
    policy.open-cluster-management.io/categories: CM Configuration Management
    policy.open-cluster-management.io/controls: CM-2 Baseline Configuration
    policy.open-cluster-management.io/standards: NIST SP 800-53
  name: install-compliance-operator
  namespace: policies
spec:
  disabled: false
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          name: install-compliance-operator
        spec:
          object-templates:
            - complianceType: musthave
              objectDefinition:
                apiVersion: v1
                kind: Namespace
                metadata:
                  name: openshift-compliance
            - complianceType: musthave
              objectDefinition:
                apiVersion: operators.coreos.com/v1alpha1
                kind: Subscription
                metadata:
                  name: compliance-operator
                  namespace: openshift-compliance
                spec:
                  channel: release-0.1
                  name: compliance-operator
                  source: redhat-operators
                  sourceNamespace: openshift-marketplace
            - complianceType: musthave
              objectDefinition:
                apiVersion: operators.coreos.com/v1
                kind: OperatorGroup
                metadata:
                  name: compliance-operator
                  namespace: openshift-compliance
                spec:
                  targetNamespaces:
                    - compliance-operator
          remediationAction: enforce
          severity: low
```

See the Compliance Operator documentation for more details.

2.4.2.4. Install the policy generator on OpenShift GitOps (ArgoCD)

OpenShift GitOps, based on ArgoCD, can also be used to generate policies using the policy generator through GitOps. Since the policy generator does not come preinstalled in the OpenShift GitOps container image, some customization must take place. In order to follow along, it is expected that you have the OpenShift GitOps Operator installed on the Red Hat Advanced Cluster Management hub cluster and be sure to log into the hub cluster.

In order for OpenShift GitOps to have access to the policy generator when you run Kustomize, an Init Container is required to copy the policy generator binary from the Red Hat Advanced Cluster Management Application Subscription container image to the OpenShift GitOps container, that runs Kustomize. For more details, see Using Init Containers to perform tasks before a pod is deployed . Additionally, OpenShift GitOps must be configured to provide the **--enable-alpha-plugins** flag when you run Kustomize. Start editing the OpenShift GitOps **argocd** object with the following command:

```
oc -n openshift-gitops edit argocd openshift-gitops
```

Then modify the OpenShift GitOps **argocd** object to contain the following additional YAML content. When a new major version of Red Hat Advanced Cluster Management is released and you want to update the policy generator to a newer version, you need to update the **registry.redhat.io/rhacm2/multicluster-operators-subscription-rhel8** image used by the Init Container to a newer tag. View the following example, and replace **<version>** with **2.5** or your desired Red Hat Advanced Cluster Management version:

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: openshift-gitops
  namespace: openshift-gitops
spec:
  kustomizeBuildOptions: --enable-alpha-plugins
  repo:
    env:
    - name: KUSTOMIZE_PLUGIN_HOME
      value: /etc/kustomize/plugin
    initContainers:
    - args:
      - -c
      - cp /etc/kustomize/plugin/policy.open-cluster-management.io/v1/policygenerator/PolicyGenerator
        /policy-generator/PolicyGenerator
      command:
      - /bin/bash
      image: registry.redhat.io/rhacm2/multicluster-operators-subscription-rhel8:v<version>
      name: policy-generator-install
      volumeMounts:
      - mountPath: /policy-generator
        name: policy-generator
    volumeMounts:
    - mountPath: /etc/kustomize/plugin/policy.open-cluster-management.io/v1/policygenerator
      name: policy-generator
    volumes:
    - emptyDir: {}
      name: policy-generator
```

Now that OpenShift GitOps can use the policy generator, OpenShift GitOps must be granted access to create policies on the Red Hat Advanced Cluster Management hub cluster. Create the following **ClusterRole** resource called **openshift-gitops-policy-admin**, with access to create, read, update, and delete policies and placements. Your **ClusterRole** might resemble the following example:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: openshift-gitops-policy-admin
rules:
  - verbs:
      - get
      - list
      - watch
      - create
      - update
      - patch
      - delete
    apiGroups:
      - policy.open-cluster-management.io
    resources:
      - policies
      - placementbindings
  - verbs:
      - get
      - list
      - watch
      - create
      - update
      - patch
      - delete
    apiGroups:
      - apps.open-cluster-management.io
    resources:
      - placementrules
  - verbs:
      - get
      - list
      - watch
      - create
      - update
      - patch
      - delete
    apiGroups:
      - cluster.open-cluster-management.io
    resources:
      - placements
      - placements/status
      - placementdecisions
      - placementdecisions/status
```

Additionally, create a **ClusterRoleBinding** object to grant the OpenShift GitOps service account access to the **openshift-gitops-policy-admin ClusterRole**. Your **ClusterRoleBinding** might resemble the following resource:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: openshift-gitops-policy-admin
subjects:
  - kind: ServiceAccount
    name: openshift-gitops-argocd-application-controller
    namespace: openshift-gitops
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: openshift-gitops-policy-admin
```

2.4.2.5. Policy generator configuration reference table

Note that all the fields in the **policyDefaults** section except for **namespace** can be overridden per policy.

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
complianceType	Optional. Determines the policy controller behavior when comparing the manifest to objects on the cluster. The parameter values are **musthave**, **mustonlyhave**, or **mustnothave**. The default value is **musthave**.
kind	Required. Set the value to **PolicyGenerator** to indicate the type of policy.
metadata	Required. Used to uniquely identify the configuration file.
metadata.name	Required. The name for identifying the policy resource.

Field	Description
placementBindingDefaults	Required. Used to consolidate multiple policies in a **PlacementBinding**, so that the generator can create unique **PlacementBinding** names using the name that is defined.
placementBindingDefaults.name	Optional. It is best practice to set an explicit placement binding name to use rather than use the default value.
policyDefaults	Required. Any default value listed here is overridden for an entry in the policies array except for **namespace**.
policyDefaults.categories	Optional. Array of categories to be used in the **policy.open-cluster-management.io/categories** annotation. The default value is **CM Configuration Management**.
policyDefaults.controls	Optional. Array of controls to be used in the **policy.open-cluster-management.io/controls** annotation. The default value is **CM-2 Baseline Configuration**.
policyDefaults.consolidateManifests	Optional. This determines if a single configuration policy should be generated for all the manifests being wrapped in the policy. If set to **false**, a configuration policy per manifest is generated. The default value is **true**.
policyDefaults.informGatekeeperPolicies	Optional. When the policy references a violated gatekeeper policy manifest, this determines if an additional configuration policy should be generated in order to receive policy violations in Red Hat Advanced Cluster Management. The default value is **true**.
policyDefaults.informKyvernoPolicies	Optional. When the policy references a Kyverno policy manifest, this determines if an additional configuration policy should be generated to receive policy violations in Red Hat Advanced Cluster Management, when the Kyverno policy has been violated. The default value is **true**.
policyDefaults.namespace	Required. The namespace of all the policies.
policyDefaults.placement	Optional. The placement configuration for the policies. This defaults to a placement configuration that matches all clusters.

Field	Description
placement.clusterSelectors	Optional. Specify a placement by defining a cluster selector in the following format, **key:value**. See **placementRulePath** to specify an existing file.
placement.name	Optional. Specify a name to consolidate placement rules that contain the same cluster selectors.
placement.placementRulePath	Optional. To reuse an existing placement rule, specify the path here relative to the **kustomization.yaml** file. If provided, this placement rule is used by all policies by default. See **clusterSelectors** to generate a new **Placement**.
policyDefaults.remediationAction	Optional. The remediation mechanism of your policy. The parameter values are **enforce** and **inform**. The default value is **inform**.
policyDefaults.severity	Optional. The severity of the policy violation. The default value is **low**.
policyDefaults.standards	Optional. An array of standards to be used in the **policy.open-cluster-management.io/standards** annotation. The default value is **NIST SP 800-53**.
policies	Required. The list of policies to create along with overrides to either the default values, or the values that are set in **policyDefaults.**
policies[ ].manifests	Required. The list of Kubernetes object manifests to include in the policy.
policies[ ].name	Required. The name of the policy to create.
policies[ ].manifests[ ].complianceType	Optional. Determines the policy controller behavior when comparing the manifest to objects on the cluster. The parameter values are **musthave**, **mustonlyhave**, or **mustnothave**. The default value is **musthave**.
policies[ ].manifests[ ].path	Required. Path to a single file or a flat directory of files relative to the **kustomization.yaml** file.

Field	Description
policies[ ].manifests[ ].patches	Optional. A Kustomize patch to apply to the manifest at the path. If there are multiple manifests, the patch requires the **apiVersion**, **kind**, **metadata.name**, and **metadata.namespace** (if applicable) fields to be set so Kustomize can identify the manifest that the patch applies to. If there is a single manifest, the **metadata.name** and **metadata.namespace** fields can be patched.

2.5. SUPPORTED POLICIES

View the supported policies to learn how to define rules, processes, and controls on the hub cluster when you create and manage policies in Red Hat Advanced Cluster Management for Kubernetes.

Note: You can copy and paste an existing policy in to the *Policy YAML*. The values for the parameter fields are automatically entered when you paste your existing policy. You can also search the contents in your policy YAML file with the search feature.

2.5.1. Support matrix for out-of-box policies

Policy	Red Hat OpenShift Container Platform 3.11	Red Hat OpenShift Container Platform 4
Memory usage policy	x	x
Namespace policy	x	x
Image vulnerability policy	x	x
Pod policy	x	x
Pod security policy (deprecated)		
Role policy	x	x
Role binding policy	x	x
Security Context Constraints policy (SCC)	x	x
ETCD encryption policy		x
Gatekeeper policy		x
Compliance operator policy		x

Policy	Red Hat OpenShift Container Platform 3.11	Red Hat OpenShift Container Platform 4
E8 scan policy		x
OpenShift CIS scan policy		x
Policy set		x
Kyverno add network		x
Kyverno add quota		x
Kyverno sync secrets		x

View the following policy samples to view how specific policies are applied:

- Image vulnerability policy
- Memory usage policy
- Namespace policy
- Pod policy
- Pod security policy
- Role policy
- Role binding policy
- Security context constraints policy
- ETCD encryption policy
- Compliance operator policy
- E8 scan policy
- OpenShift CIS scan policy
- Policy set controller
- Kyverno add network policy
- Kyverno add quota policy
- Kyverno sync secrets policy

Refer to Governance for more topics.

2.5.2. Memory usage policy

Kubernetes configuration policy controller monitors the status of the memory usage policy. Use the memory usage policy to limit or restrict your memory and compute usage. For more information, see *Limit Ranges* in the Kubernetes documentation.

Learn more details about the memory usage policy structure in the following sections:

- Memory usage policy YAML structure
- Memory usage policy table
- Memory usage policy sample

2.5.2.1. Memory usage policy YAML structure

Your memory usage policy might resemble the following YAML file:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-limitrange
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        apiVersion:
        kind:
        metadata:
          name:
        spec:
          limits:
          - default:
              memory:
            defaultRequest:
              memory:
            type:
        ...
```

2.5.2.2. Memory usage policy table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.

Field	Description
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.2.3. Memory usage policy sample

See the **policy-limitmemory.yaml** to view a sample of the policy. See Managing security policies for more details. Refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the controller.

2.5.3. Namespace policy

Kubernetes configuration policy controller monitors the status of your namespace policy. Apply the namespace policy to define specific rules for your namespace.

Learn more details about the namespace policy structure in the following sections:

- Namespace policy YAML structure
- Namespace policy YAML table
- Namespace policy sample

2.5.3.1. Namespace policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-namespace-1
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        kind:
        apiVersion:
        metadata:
          name:
    ...
```

2.5.3.2. Namespace policy YAML table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.

Field	Description
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.3.3. Namespace policy sample

See **policy-namespace.yaml** to view the policy sample.

See Managing security policies for more details. Refer to Kubernetes configuration policy controller to learn about other configuration policies.

2.5.4. Image vulnerability policy

Apply the image vulnerability policy to detect if container images have vulnerabilities by leveraging the Container Security Operator. The policy installs the Container Security Operator on your managed cluster if it is not installed.

The image vulnerability policy is checked by the Kubernetes configuration policy controller. For more information about the Security Operator, see the *Container Security Operator* from the Quay repository.

Notes:

- Image vulnerability policy is not functional during a disconnected installation.
- The Image vulnerability policy is not supported on the IBM Power and IBM Z architectures. It relies on the Quay Container Security Operator. There are no **ppc64le** or **s390x** images in the container-security-operator registry.

View the following sections to learn more:

- Image vulnerability policy YAML structure
- Image vulnerability policy YAML table
- Image vulnerability policy sample

2.5.4.1. Image vulnerability policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-imagemanifestvulnpolicy
  namespace: default
  annotations:
    policy.open-cluster-management.io/standards: NIST-CSF
    policy.open-cluster-management.io/categories: DE.CM Security Continuous Monitoring
    policy.open-cluster-management.io/controls: DE.CM-8 Vulnerability Scans
spec:
  remediationAction:
```

```
  disabled:
  policy-templates:
  - objectDefinition:
      apiVersion: policy.open-cluster-management.io/v1
      kind: ConfigurationPolicy
      metadata:
        name:
      spec:
        remediationAction:
        severity: high
        object-templates:
          - complianceType:
            objectDefinition:
              apiVersion: operators.coreos.com/v1alpha1
              kind: Subscription
              metadata:
                name: container-security-operator
                namespace:
              spec:
                channel:
                installPlanApproval:
                name:
                source:
                sourceNamespace:
  - objectDefinition:
      apiVersion: policy.open-cluster-management.io/v1
      kind: ConfigurationPolicy
      metadata:
        name:
      spec:
        remediationAction:
        severity:
        namespaceSelector:
          exclude:
          include:
        object-templates:
          - complianceType:
            objectDefinition:
              apiVersion: secscan.quay.redhat.com/v1alpha1
              kind: ImageManifestVuln # checking for a kind
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-policy-imagemanifestvulnpolicy
  namespace: default
placementRef:
  name:
  kind:
  apiGroup:
subjects:
- name:
  kind:
  apiGroup:
---
apiVersion: apps.open-cluster-management.io/v1
```

```
kind: PlacementRule
metadata:
  name: placement-policy-imagemanifestvulnpolicy
  namespace: default
spec:
  clusterConditions:
  - status:
    type:
  clusterSelector:
    matchExpressions:
      [] # selects all clusters if not specified
```

2.5.4.2. Image vulnerability policy YAML table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.4.3. Image vulnerability policy sample

See **policy-imagemanifestvuln.yaml**. See Managing security policies for more information.

Refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the configuration controller.

2.5.5. Pod policy

Kubernetes configuration policy controller monitors the status of your pod policies. Apply the pod policy to define the container rules for your pods. A pod must exist in your cluster to use this information.

Learn more details about the pod policy structure in the following sections:

- Pod policy YAML structure
- Pod policy table
- Pod policy sample

2.5.5.1. Pod policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-pod
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        apiVersion:
        kind: Pod # pod must exist
        metadata:
          name:
        spec:
          containers:
          - image:
            name:
            ports:
            - containerPort:
    ...
```

2.5.5.2. Pod policy table

Field	Description

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.5.3. Pod policy sample

See **policy-pod.yaml** to view the policy sample.

Refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the configuration controller. See Managing configuration policies to manage other policies.

2.5.6. Pod security policy

Kubernetes configuration policy controller monitors the status of the pod security policy. Apply a pod security policy to secure pods and containers. For more information, see *Pod Security Policies* in the Kubernetes documentation.

Learn more details about the pod security policy structure in the following sections:

- Pod security policy YAML structure
- Pod security policy table
- Pod security policy sample

2.5.6.1. Pod security policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-podsecuritypolicy
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        apiVersion:
        kind: PodSecurityPolicy # no privileged pods
        metadata:
          name:
          annotations:
        spec:
          privileged:
          allowPrivilegeEscalation:
          allowedCapabilities:
          volumes:
          hostNetwork:
          hostPorts:
          hostIPC:
          hostPID:
          runAsUser:
            rule:
          seLinux:
            rule:
          supplementalGroups:
            rule:
          fsGroup:
            rule:
     ...
```

2.5.6.2. Pod security policy table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.

Field	Description
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.6.3. Pod security policy sample

See **policy-psp.yaml** to view the sample policy. View Managing configuration policies for more information.

Refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the controller.

2.5.7. Role policy

Kubernetes configuration policy controller monitors the status of role policies. Define roles in the **object-template** to set rules and permissions for specific roles in your cluster.

Learn more details about the role policy structure in the following sections:

- Role policy YAML structure

- Role policy table
- Role policy sample

2.5.7.1. Role policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-role
  namespace:
  annotations:
    policy.open-cluster-management.io/standards: NIST-CSF
    policy.open-cluster-management.io/categories: PR.AC Identity Management Authentication and Access Control
    policy.open-cluster-management.io/controls: PR.AC-4 Access Control
spec:
  remediationAction: inform
  disabled: false
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          name: policy-role-example
        spec:
          remediationAction: inform # will be overridden by remediationAction in parent policy
          severity: high
          namespaceSelector:
            include: ["default"]
          object-templates:
            - complianceType: mustonlyhave # role definition should exact match
              objectDefinition:
                apiVersion: rbac.authorization.k8s.io/v1
                kind: Role
                metadata:
                  name: sample-role
                rules:
                  - apiGroups: ["extensions", "apps"]
                    resources: ["deployments"]
                    verbs: ["get", "list", "watch", "delete","patch"]
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-policy-role
  namespace:
placementRef:
  name: placement-policy-role
  kind: PlacementRule
  apiGroup: apps.open-cluster-management.io
subjects:
- name: policy-role
  kind: Policy
  apiGroup: policy.open-cluster-management.io
---
```

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-policy-role
  namespace:
spec:
  clusterConditions:
    - type: ManagedClusterConditionAvailable
      status: "True"
  clusterSelector:
    matchExpressions:
      []

        ...
```

2.5.7.2. Role policy table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**

Field	Description
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.7.3. Role policy sample

Apply a role policy to set rules and permissions for specific roles in your cluster. For more information on roles, see Role-based access control. View a sample of a role policy, see **policy-role.yaml**.

To learn how to manage role policies, refer to Managing configuration policies for more information. See the Kubernetes configuration policy controller to view other configuration policies that are monitored the controller.

2.5.8. Role binding policy

Kubernetes configuration policy controller monitors the status of your role binding policy. Apply a role binding policy to bind a policy to a namespace in your managed cluster.

Learn more details about the namespace policy structure in the following sections:

- Role binding policy YAML structure
- Role binding policy table
- Role binding policy sample

2.5.8.1. Role binding policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name:
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        kind: RoleBinding # role binding must exist
        apiVersion: rbac.authorization.k8s.io/v1
        metadata:
          name: operate-pods-rolebinding
        subjects:
        - kind: User
          name: admin # Name is case sensitive
          apiGroup:
        roleRef:
```

```
      kind: Role #this must be Role or ClusterRole
      name: operator # this must match the name of the Role or ClusterRole you wish to bind to
      apiGroup: rbac.authorization.k8s.io
  ...
```

2.5.8.2. Role binding policy table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name to identify the policy resource.
metadata.namespaces	Required. The namespace within the managed cluster where the policy is created.
spec	Required. Specifications of how compliance violations are identified and fixed.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.complianceType	Required. Set the value to **"musthave"**
spec.namespace	Required. Managed cluster namespace to which you want to apply the policy. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
spec.remediationAction	Required. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**.
spec.object-template	Required. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

2.5.8.3. Role binding policy sample

See **policy-rolebinding.yaml** to view the policy sample. See Managing configuration policies for more information about managing other policies.

Refer to Kubernetes configuration policy controller to learn about other configuration policies.

2.5.9. Security Context Constraints policy

Kubernetes configuration policy controller monitors the status of your Security Context Constraints (SCC) policy. Apply an Security Context Constraints (SCC) policy to control permissions for pods by defining conditions in the policy.

Learn more details about SCC policies in the following sections:

- SCC policy YAML structure
- SCC policy table
- SCC policy sample

2.5.9.1. SCC policy YAML structure

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-scc
  namespace: open-cluster-management-policies
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        apiVersion:
        kind: SecurityContextConstraints # restricted scc
        metadata:
          annotations:
            kubernetes.io/description:
          name: sample-restricted-scc
        allowHostDirVolumePlugin:
        allowHostIPC:
        allowHostNetwork:
        allowHostPID:
        allowHostPorts:
        allowPrivilegeEscalation:
        allowPrivilegedContainer:
        allowedCapabilities:
        defaultAddCapabilities:
        fsGroup:
         type:
        groups:
        - system:
        priority:
        readOnlyRootFilesystem:
```

```
requiredDropCapabilities:
runAsUser:
  type:
seLinuxContext:
  type:
supplementalGroups:
  type:
users:
volumes:
```

2.5.9.2. SCC policy table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy.
metadata.name	Required. The name to identify the policy resource.
metadata.namespace	Required. The namespace within the managed cluster where the policy is created.
spec.complianceType	Required. Set the value to **"musthave"**
spec.remediationAction	Required. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**. **Important**: Some policies might not support the enforce feature.
spec.namespace	Required. Managed cluster namespace to which you want to apply the policy. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
spec.object-template	Required. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters.

For explanations on the contents of a SCC policy, see Managing Security Context Constraints from the OpenShift Container Platform documentation.

2.5.9.3. SCC policy sample

Apply a Security context constraints (SCC) policy to control permissions for pods by defining conditions in the policy. For more information see, Managing Security Context Constraints (SCC) .

See **policy-scc.yaml** to view the policy sample. See Managing configuration policies for more information about managing other policies.

Refer to Kubernetes configuration policy controller to learn about other configuration policies.

2.5.10. ETCD encryption policy

Apply the **etcd-encryption** policy to detect, or enable encryption of sensitive data in the ETCD data-store. Kubernetes configuration policy controller monitors the status of the **etcd-encryption** policy. For more information, see Encrypting etcd data in the OpenShift Container Platform documentation. **Note**: The ETCD encryption policy only supports Red Hat OpenShift Container Platform 4 and later.

Learn more details about the **etcd-encryption** policy structure in the following sections:

- ETCD encryption policy YAML structure
- ETCD encryption policy table
- Etcd encryption policy sample

2.5.10.1. ETCD encryption policy YAML structure

Your **etcd-encryption** policy might resemble the following YAML file:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: policy-etcdencryption
  namespace:
spec:
  complianceType:
  remediationAction:
  namespaces:
    exclude:
    include:
  object-templates:
    - complianceType:
      objectDefinition:
        apiVersion: config.openshift.io/v1
        kind: APIServer
        metadata:
          name: cluster
        spec:
          encryption:
            type:
        ...
```

2.5.10.2. ETCD encryption policy table

Table 2.5. Parameter table

Field	Description
apiVersion	Required. Set the value to **policy.open-cluster-management.io/v1**.
kind	Required. Set the value to **Policy** to indicate the type of policy, for example, **ConfigurationPolicy**.
metadata.name	Required. The name for identifying the policy resource.
metadata.namespaces	Optional.
spec.namespace	Required. The namespaces within the hub cluster that the policy is applied to. Enter parameter values for **include**, which are the namespaces you want to apply to the policy to. The **exclude** parameter specifies the namespaces you explicitly do not want to apply the policy to. **Note**: A namespace that is specified in the object template of a policy controller overrides the namespace in the corresponding parent policy.
remediationAction	Optional. Specifies the remediation of your policy. The parameter values are **enforce** and **inform**. **Important**: Some policies might not support the enforce feature.
disabled	Required. Set the value to **true** or **false**. The **disabled** parameter provides the ability to enable and disable your policies.
spec.complianceType	Required. Set the value to **"musthave"**
spec.object-template	Optional. Used to list any other Kubernetes object that must be evaluated or applied to the managed clusters. See Encrypting etcd data in the OpenShift Container Platform documentation.

2.5.10.3. Etcd encryption policy sample

See **policy-etcdencryption.yaml** for the policy sample. See Managing security policies for more information.

Refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the controller.

2.5.11. Compliance operator policy

Compliance operator is an operator that runs OpenSCAP and allows you to keep your Red Hat OpenShift Container Platform cluster compliant with the security benchmark that you need. You can install the compliance operator on your managed cluster by using the compliance operator policy.

The compliance operator policy is created as a Kubernetes configuration policy in Red Hat Advanced Cluster Management. OpenShift Container Platform 4.7 and 4.6, support the compliance operator policy. For more information, see Understanding the Compliance Operator in the OpenShift Container Platform documentation for more details.

Note: The Compliance operator policy relies on the OpenShift Container Platform Compliance Operator, which is not supported on the IBM Power or IBM Z architectures. See Understanding the Compliance Operator in the OpenShift Container Platform documentation for more information about the Compliance Operator.

2.5.11.1. Compliance operator resources

When you create a compliance operator policy, the following resources are created:

- A compliance operator namespace (**openshift-compliance**) for the operator installation:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: comp-operator-ns
spec:
  remediationAction: inform # will be overridden by remediationAction in parent policy
  severity: high
  object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: v1
        kind: Namespace
        metadata:
          name: openshift-compliance
```

- An operator group (**compliance-operator**) to specify the target namespace:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: comp-operator-operator-group
spec:
  remediationAction: inform # will be overridden by remediationAction in parent policy
  severity: high
  object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: operators.coreos.com/v1
        kind: OperatorGroup
        metadata:
          name: compliance-operator
          namespace: openshift-compliance
        spec:
          targetNamespaces:
            - openshift-compliance
```

- A subscription (**comp-operator-subscription**) to reference the name and channel. The subscription pulls the profile, as a container, that it supports:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: comp-operator-subscription
spec:
  remediationAction: inform  # will be overridden by remediationAction in parent policy
  severity: high
  object-templates:
    - complianceType: musthave
      objectDefinition:
        apiVersion: operators.coreos.com/v1alpha1
        kind: Subscription
        metadata:
          name: compliance-operator
          namespace: openshift-compliance
        spec:
          channel: "4.7"
          installPlanApproval: Automatic
          name: compliance-operator
          source: redhat-operators
          sourceNamespace: openshift-marketplace
```

After you install the compliance operator policy, the following pods are created: **compliance-operator**, **ocp4**, and **rhcos4**. See a sample of the **policy-compliance-operator-install.yaml**.

You can also create and apply the E8 scan policy and OpenShift CIS scan policy, after you have installed the compliance operator. For more information, see E8 scan policy and OpenShift CIS scan policy.

To learn about managing compliance operator policies, see Managing security policies for more details. Refer to Kubernetes configuration policy controller for more topics about configuration policies.

2.5.12. E8 scan policy

An Essential 8 (E8) scan policy deploys a scan that checks the master and worker nodes for compliance with the E8 security profiles. You must install the compliance operator to apply the E8 scan policy.

The E8 scan policy is created as a Kubernetes configuration policy in Red Hat Advanced Cluster Management. OpenShift Container Platform 4.7 and 4.6, support the E8 scan policy. For more information, see *Understanding the Compliance Operator* in the OpenShift Container Platform documentation for more details.

2.5.12.1. E8 scan policy resources

When you create an E8 scan policy the following resources are created:

- A **ScanSettingBinding** resource (**e8**) to identify which profiles to scan:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-suite-e8
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: musthave # this template checks if scan has completed by checking the status field
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ScanSettingBinding
        metadata:
          name: e8
          namespace: openshift-compliance
        profiles:
        - apiGroup: compliance.openshift.io/v1alpha1
          kind: Profile
          name: ocp4-e8
        - apiGroup: compliance.openshift.io/v1alpha1
          kind: Profile
          name: rhcos4-e8
        settingsRef:
          apiGroup: compliance.openshift.io/v1alpha1
          kind: ScanSetting
          name: default
```

- A **ComplianceSuite** resource (**compliance-suite-e8**) to verify if the scan is complete by checking the **status** field:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-suite-e8
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: musthave # this template checks if scan has completed by checking the status field
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ComplianceSuite
        metadata:
          name: e8
          namespace: openshift-compliance
        status:
          phase: DONE
```

- A **ComplianceCheckResult** resource (**compliance-suite-e8-results**) which reports the results of the scan suite by checking the **ComplianceCheckResult** custom resources (CR):

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-suite-e8-results
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: mustnothave # this template reports the results for scan suite: e8 by
looking at ComplianceCheckResult CRs
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ComplianceCheckResult
        metadata:
          namespace: openshift-compliance
          labels:
            compliance.openshift.io/check-status: FAIL
            compliance.openshift.io/suite: e8
```

Note: Automatic remediation is supported. Set the remediation action to **enforce** to create **ScanSettingBinding** resource.

See a sample of the **policy-compliance-operator-e8-scan.yaml**. See Managing security policies for more information. **Note**: After your E8 policy is deleted, it is removed from your target cluster or clusters.

2.5.13. OpenShift CIS scan policy

An OpenShift CIS scan policy deploys a scan that checks the master and worker nodes for compliance with the OpenShift CIS security benchmark. You must install the compliance operator to apply the OpenShift CIS policy.

The OpenShift CIS scan policy is created as a Kubernetes configuration policy in Red Hat Advanced Cluster Management. OpenShift Container Platform 4.9, 4.7, and 4.6, support the OpenShift CIS scan policy. For more information, see Understanding the Compliance Operator in the OpenShift Container Platform documentation for more details.

2.5.13.1. OpenShift CIS resources

When you create an OpenShift CIS scan policy the following resources are created:

- A **ScanSettingBinding** resource (**cis**) to identify which profiles to scan:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-cis-scan
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: musthave # this template creates ScanSettingBinding:cis
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ScanSettingBinding
        metadata:
          name: cis
          namespace: openshift-compliance
        profiles:
        - apiGroup: compliance.openshift.io/v1alpha1
          kind: Profile
          name: ocp4-cis
        - apiGroup: compliance.openshift.io/v1alpha1
          kind: Profile
          name: ocp4-cis-node
        settingsRef:
          apiGroup: compliance.openshift.io/v1alpha1
          kind: ScanSetting
          name: default
```

- A **ComplianceSuite** resource (**compliance-suite-cis**) to verify if the scan is complete by checking the **status** field:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-suite-cis
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: musthave # this template checks if scan has completed by checking the status field
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ComplianceSuite
        metadata:
          name: cis
          namespace: openshift-compliance
        status:
          phase: DONE
```

- A **ComplianceCheckResult** resource (**compliance-suite-cis-results**) which reports the results of the scan suite by checking the **ComplianceCheckResult** custom resources (CR):

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: compliance-suite-cis-results
spec:
  remediationAction: inform
  severity: high
  object-templates:
    - complianceType: mustnothave # this template reports the results for scan suite: cis by looking at ComplianceCheckResult CRs
      objectDefinition:
        apiVersion: compliance.openshift.io/v1alpha1
        kind: ComplianceCheckResult
        metadata:
          namespace: openshift-compliance
          labels:
            compliance.openshift.io/check-status: FAIL
            compliance.openshift.io/suite: cis
```

See a sample of the **policy-compliance-operator-cis-scan.yaml** file. For more information on creating policies, see Managing security policies.

2.5.14. Kyverno add network policy

The Kyverno add network policy configures a new **NetworkPolicy** resource called **default-deny** which denies all traffic when you create a new namespace. You must install Kyverno controller to use the Kyverno policy. See **policy-install-kyverno.yaml** for the install policy.

Learn more details about the **policy-kyverno-add-network-policy** policy structure in the following sections:

- Kyverno add network policy YAML structure
- Kyverno add network policy sample

2.5.14.1. Kyverno add network policy YAML structure

Your **policy-kyverno-add-network-policy** policy might resemble the following YAML file:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: add-networkpolicy
  annotations:
    policies.kyverno.io/title: Add Network Policy
    policies.kyverno.io/category: Multi-Tenancy
    policies.kyverno.io/subject: NetworkPolicy
    policies.kyverno.io/description: >-
      By default, Kubernetes allows communications across all Pods within a cluster.
      The NetworkPolicy resource and a CNI plug-in that supports NetworkPolicy must be used to
restrict
      communications. A default NetworkPolicy should be configured for each Namespace to
      default deny all ingress and egress traffic to the Pods in the Namespace. Application
      teams can then configure additional NetworkPolicy resources to allow desired traffic
      to application Pods from select sources. This policy will create a new NetworkPolicy resource
      named `default-deny` which will deny all traffic anytime a new Namespace is created.
spec:
  rules:
  - name: default-deny
    match:
      resources:
        kinds:
        - Namespace
    generate:
      apiVersion: networking.k8s.io/v1
      kind: NetworkPolicy
      name: default-deny
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
      data:
        spec:
          # select all pods in the namespace
          podSelector: {}
          # deny all traffic
          policyTypes:
          - Ingress
          - Egress
```

2.5.14.2. Kyverno add network policy sample

See **policy-kyverno-add-network-policy.yaml** for the policy sample. See the Policy overview documentation and the Kubernetes configuration policy controller to view additional details on policy and configuration policy fields.

2.5.15. Kyverno add quota policy

The Kyverno add quota policy configures new **ResourceQuota** and **LimitRange** resources when you create a new namespace. You must install Kyverno controller to use the Kyverno policy. See **policy-install-kyverno.yaml** for the install policy.

Learn more details about the **policy-kyverno-add-quota** policy structure in the following sections:

- Kyverno add quota policy YAML structure
- Kyverno add quota policy sample

2.5.15.1. Kyverno add quota policy YAML structure

Your **policy-kyverno-add-quota** policy might resemble the following YAML file:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: add-ns-quota
  annotations:
    policies.kyverno.io/title: Add Quota
    policies.kyverno.io/category: Multi-Tenancy
    policies.kyverno.io/subject: ResourceQuota, LimitRange
    policies.kyverno.io/description: >-
      To better control the number of resources that can be created in a given
      Namespace and provide default resource consumption limits for Pods,
      ResourceQuota and LimitRange resources are recommended.
      This policy will generate ResourceQuota and LimitRange resources when
      a new Namespace is created.
spec:
  rules:
  - name: generate-resourcequota
    match:
      resources:
        kinds:
        - Namespace
    generate:
      apiVersion: v1
      kind: ResourceQuota
      name: default-resourcequota
      synchronize: true
      namespace: "{{request.object.metadata.name}}"
      data:
        spec:
          hard:
            requests.cpu: '4'
            requests.memory: '16Gi'
            limits.cpu: '4'
            limits.memory: '16Gi'
  - name: generate-limitrange
    match:
      resources:
        kinds:
        - Namespace
    generate:
      apiVersion: v1
      kind: LimitRange
      name: default-limitrange
      synchronize: true
      namespace: "{{request.object.metadata.name}}"
      data:
        spec:
          limits:
          - default:
              cpu: 500m
              memory: 1Gi
            defaultRequest:
              cpu: 200m
              memory: 256Mi
            type: Container
```

2.5.15.2. Kyverno add quota policy policy sample

See **policy-kyverno-add-quota.yaml** for the policy sample. See the Policy overview documentation and the Kubernetes configuration policy controller to view additional details on policy and configuration policy fields.

2.5.16. Kyverno sync secrets policy

The Kyverno sync secrets policy copies a secret called **regcred** which exists in the **default** namespace to new namespaces you create and updates the secret when changes are detected. You must install Kyverno controller to use the Kyverno policy. See **policy-install-kyverno.yaml** for the install policy.

Learn more details about the **policy-kyverno-sync-secrets** policy structure in the following sections:

- Kyverno sync secrets policy YAML structure
- Kyverno sync secrets policy sample

2.5.16.1. Kyverno sync secrets policy YAML structure

Your **policy-kyverno-sync-secrets** policy might resemble the following YAML file:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: sync-secrets
  annotations:
    policies.kyverno.io/title: Sync Secrets
    policies.kyverno.io/category: Sample
    policies.kyverno.io/subject: Secret
    policies.kyverno.io/description: >-
      Secrets like registry credentials often need to exist in multiple
      Namespaces so Pods there have access. Manually duplicating those Secrets
      is time consuming and error prone. This policy will copy a
      Secret called `regcred` which exists in the `default` Namespace to
      new Namespaces when they are created. It will also push updates to
      the copied Secrets should the source Secret be changed.
spec:
  rules:
  - name: sync-image-pull-secret
    match:
      resources:
        kinds:
        - Namespace
    generate:
      apiVersion: v1
      kind: Secret
      name: regcred
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
      clone:
        namespace: default
        name: regcred
```

2.5.16.2. Kyverno sync secrets policy policy sample

See **policy-kyverno-sync-secrets.yaml** for the policy sample. See the Policy overview documentation and the Kubernetes configuration policy controller to view additional details on policy and configuration policy fields.

2.6. MANAGE SECURITY POLICIES

Use the *Governance* dashboard to create, view, and manage your security policies and policy violations. You can create YAML files for your policies from the CLI and console.

2.6.1. Governance page

The following tabs are displayed on the *Governance* page:

- **Overview**
  View the following summary cards from the *Overview* tab: *Policy set violations*, *Policy violations*, *Clusters*, *Categories*, *Controls*, and *Standards*.

- **Policy sets**
  Create and manage hub cluster policy sets.

- **Policies**
  Create and manage security policies. The table of policies lists the following details of a policy: *Name*, *Namespace*, *Status*, *Remediation*, *Policy set*, *Cluster violations*, *Source*, *Automation* and *Created*.

  You can edit, enable or disable, set remediation to inform or enforce, or remove a policy by selecting the **Actions** icon. You can view the categories and standards of a specific policy by selecting the drop-down arrow to expand the row.

  Complete bulk actions by selecting multiple policies and clicking the **Actions** button. You can also customize your policy table by clicking the **Filter** button.

If there is a configured automation for a specific policy, you can select the automation to view more details. View the following descriptions of the schedule frequency options for your automation:

- *Manual run* mode: Manually set this automation to run once. After the automation runs, it is set to **disabled**. Select the *Manual run* checkbox for the automation.

- *Once mode*: When a policy is violated, the automation runs one time. After the automation runs, it is set to **disabled**. After the automation is set to **disabled**, you must continue to run the automation manually. When you run *once mode*, the extra variable of **target_clusters** is automatically supplied with the list of clusters that violated the policy. The Ansible Tower Job Template must have **PROMPT ON LAUNCH** enabled for the **EXTRA VARIABLES** section.

- *Disabled*: When the scheduled automation is set to **disabled**, the automation does not run until the setting is updated.

When you select a policy in the table list, the following tabs of information are displayed from the console:

- *Details*: Select the *Details* tab to view policy details and placement details. In the *Placement* table, the *Compliance* column provides links to view the compliance of the clusters that are displayed.

- *Results*: Select the *Results* tab to view a table list of all clusters that are associated to the policy. From the *Message* column, click the **View details** link to view the template details, template YAML, and related resources. You can also view related resources. Click the **View history** link to view the violation message and a time of the last report.

Review the following topics to learn more about creating and updating your security policies:

- Managing security policies

- Managing configuration policies

- Managing gatekeeper policies

- Configuring Ansible Tower for governance

Refer to Governance for more topics.

2.6.2. Configuring Ansible Tower for governance

Red Hat Advanced Cluster Management for Kubernetes governance can be integrated with Ansible Tower automation to create policy violation automations. You can configure the automation from the Red Hat Advanced Cluster Management console.

- Prerequisites

- Create a policy violation automation from the console

- Create a policy violation automation from the CLI

2.6.2.1. Prerequisites

- Red Hat OpenShift Container Platform 4.5 or later

- You must have Ansible Tower version 3.7.3 or a later version installed. It is best practice to install the latest supported version of Ansible Tower. See Red Hat Ansible Tower documentation for more details.

- Install the Ansible Automation Platform Resource Operator on to your hub cluster to connect Ansible jobs to the governance framework. For best results when using the AnsibleJob to launch Ansible Tower jobs, the Ansible Tower job template should be idempotent when it is run. If you do not have Ansible Automation Platform Resource Operator, you can find it from the Red Hat OpenShift Container Platform *OperatorHub* page.

For more information about installing and configuring Ansible Tower automation, see Setting up Ansible tasks

2.6.2.2. Create a policy violation automation from the console

After you log into your Red Hat Advanced Cluster Management hub cluster, select **Governance** from the navigation menu, and then click on the *Policies* tab to view the policy tables.

Configure an automation for a specific policy by clicking **Configure** in the *Automation* column. You can create automation when the policy automation panel appears. From the *Ansible credential* section, click the drop-down menu to select an Ansible credential. If you need to add a credential, see Managing credentials overview.

Note: This credential is copied to the same namespace as the policy. The credential is used by the **AnsibleJob** resource that is created to initiate the automation. Changes to the Ansible credential in the *Credentials* section of the console is automatically updated.

After a credential is selected, click the Ansible job drop-down list to select a job template. In the *Extra variables* section, add the parameter values from the **extra_vars** section of the **PolicyAutomation**. Select the frequency of the automation. You can select *Run once mode* , or *Disable automation*.

- *Manual run*: Manually set this automation to run once. After the automation runs, it is set to **disabled**. **Note**: You can only select *Manual run* mode when the schedule frequency is disabled.

- *Run once mode* : When a policy is violated, the automation runs one time. After the automation runs, it is set to **disabled**. After the automation is set to **disabled**, you must continue to run the automation manually. When you run *once mode* , the extra variable of **target_clusters** is automatically supplied with the list of clusters that violated the policy. The Ansible Tower Job template must have **PROMPT ON LAUNCH** enabled for the **EXTRA VARIABLES** section.

- *Disable automation*: When the scheduled automation is set to **disabled**, the automation does not run until the setting is updated.

Save your policy violation automation by selecting **Submit**. When you select the *View Job* link from the Ansible job details side panel, the link directs you to the job template on the *Search* page. After you successfully create the automation, it is displayed in the *Automation* column.

Note: When you delete a policy that has an associated policy automation, the policy automation is automatically deleted as part of clean up.

Your policy violation automation is created from the console.

2.6.2.3. Create a policy violation automation from the CLI

Complete the following steps to configure a policy violation automation from the CLI:

1. From your terminal, log in to your Red Hat Advanced Cluster Management hub cluster using the **oc login** command.

2. Find or create a policy that you want to add an automation to. Note the policy name and namespace.

3. Create a **PolicyAutomation** resource using the following sample as a guide:

   ```
   apiVersion: policy.open-cluster-management.io/v1beta1
   kind: PolicyAutomation
   metadata:
     name: policyname-policy-automation
   spec:
     automationDef:
       extra_vars:
         your_var: your_value
       name: Policy Compliance Template
       secret: ansible-tower
       type: AnsibleJob
     mode: disabled
     policyRef: policyname
   ```

4. The Ansible job template name in the previous sample is **Policy Compliance Template**. Change that value to match your job template name.

5. In the **extra_vars** section, add any parameters you need to pass to the Ansible job template.

6. Set the mode to either **once** or **disabled**. The **once** mode runs the job one time and then the mode is set to **disabled**.

   - *once mode*: When a policy is violated, the automation runs one time. After the automation runs, it is set to **disabled**. After the automation is set to **disabled**, you must continue to run the automation manually. When you run *once mode*, the extra variable of **target_clusters** is automatically supplied with the list of clusters that violated the policy. The Ansible Tower Job template must have **PROMPT ON LAUNCH** enabled for the **EXTRA VARIABLES** section.

   - *Disable automation*: When the scheduled automation is set to **disabled**, the automation does not run until the setting is updated.

7. Set the **policyRef** to the name of your policy.

8. Create a secret in the same namespace as this **PolicyAutomation** resource that contains the Ansible Tower credential. In the previous example, the secret name is **ansible-tower**. Use the sample from application lifecycle to see how to create the secret.

9. Create the **PolicyAutomation** resource.
   Notes:

   - An immediate run of the policy automation can be initiated by adding the following annotation to the **PolicyAutomation** resource:

     ```
     metadata:
       annotations:
         policy.open-cluster-management.io/rerun: "true"
     ```

- When the policy is in **once** mode, the automation runs when the policy is non-compliant. The **extra_vars** variable, named **target_clusters** is added and the value is an array of each managed cluster name where the policy is non-compliant.

2.6.3. Deploy policies using GitOps

You can deploy a set of policies across a fleet of managed clusters with the governance framework. You can add to the open source community, **policy-collection** by contributing to and using the policies in the repository. For more information, see Contributing a custom policy. Policies in each of the **stable** and **community** folders from the open source community are further organized according to NIST Special Publication 800-53.

Continue reading to learn best practices to use GitOps to automate and track policy updates and creation through a Git repository.

Prerequisite: Before you begin, be sure to fork the **policy-collection** repository.

- Customizing your local repository
- Committing to your local repository
- Deploying policies to your cluster
- Verifying GitOps policy deployments from the console
- Verifying GitOps policy deployments from the CLI

2.6.3.1. Customizing your local repository

Customize your local repository by consolidating the **stable** and **community** policies into a single folder. Remove the policies you do not want to use. Complete the following steps to customize your local repository:

1. Create a new directory in the repository to hold the policies that you want to deploy. Be sure that you are in your local **policy-collection** repository on your main default branch for GitOps. Run the following command:

   ```
   mkdir my-policies
   ```

2. Copy all of the **stable** and **community** policies into your **my-policies** directory. Start with the **community** policies first, in case the **stable** folder contains duplicates of what is available in the community. Run the following commands:

   ```
   cp -R community/* my-policies/

   cp -R stable/* my-policies/
   ```

   Now that you have all of the policies in a single parent directory structure, you can edit the policies in your fork.

   Tips:

   - It is best practice to remove the policies you are not planning to use.
   - Learn about policies and the definition of the policies from the following list:

  - Purpose: Understand what the policy does.
  - Remediation Action: Does the policy only inform you of compliance, or enforce the policy and make changes? See the **spec.remediationAction** parameter. If changes are enforced, make sure you understand the functional expectation. Remember to check which policies support enforcement. For more information, view the *Validate* section. **Note**: The **spec.remediationAction** set for the policy overrides any remediation action that is set in the individual **spec.policy-templates**.
  - Placement: What clusters is the policy deployed to? By default, most policies target the clusters with the **environment: dev** label. Some policies may target OpenShift Container Platform clusters or another label. You can update or add additional labels to include other clusters. When there is no specific value, the policy is applied to all of your clusters. You can also create multiple copies of a policy and customize each one if you want to use a policy that is configured one way for one set of clusters and configured another way for another set of clusters.

2.6.3.2. Committing to your local repository

After you are satisfied with the changes you have made to your directory, commit and push your changes to Git so that they can be accessed by your cluster.

Note: This example is used to show the basics of how to use policies with GitOps, so you might have a different workflow to get changes to your branch.

Complete the following steps:

1. From your terminal, run **git status** to view your recent changes in your directory that you previously created. Add your new directory to the list of changes to be committed with the following command:

   ```
   git add my-policies/
   ```

2. Commit the changes and customize your message. Run the following command:

   ```
   git commit -m “Policies to deploy to the hub cluster”
   ```

3. Push the changes to the branch of your forked repository that is used for GitOps. Run the following command:

   ```
   git push origin <your_default_branch>master
   ```

Your changes are committed.

2.6.3.3. Deploying policies to your cluster

After you push your changes, you can deploy the policies to your Red Hat Advanced Cluster Management for Kubernetes installation. Post deployment, your hub cluster is connected to your Git repository. Any further changes to your chosen branch of the Git repository is reflected in your cluster.

Note: By default, policies deployed with GitOps use the **merge** reconcile option. If you want to use the **replace** reconcile option instead, add the **apps.open-cluster-management.io/reconcile-option: replace** annotation to the **Subscription** resource. See Application Lifecycle for more details.

The **deploy.sh** script creates **Channel** and **Subscription** resources in your hub cluster. The channel connects to the Git repository, and the subscription specifies the data to bring to the cluster through the channel. As a result, all policies defined in the specified subdirectory are created on your hub. After the policies are created by the subscription, Red Hat Advanced Cluster Management analyzes the policies and creates additional policy resources in the namespace associated with each managed cluster that the policy is applied to, based on the defined placement rule.

The policy is then copied to the managed cluster from its respective managed cluster namespace on the hub cluster. As a result, the policies in your Git repository are pushed to all managed clusters that have labels that match the **clusterSelector** that are defined in the placement rule of your policy.

Complete the following steps:

1. From the **policy-collection** folder, run the following command to change the directory:

   ```
   cd deploy
   ```

2. Make sure that your command line interface (CLI) is configured to create resources on the correct cluster with the following command:

   ```
   oc cluster-info
   ```

   The output of the command displays the API server details for the cluster, where Red Hat Advanced Cluster Management is installed. If the correct URL is not displayed, configure your CLI to point to the correct cluster. See Using the OpenShift CLI for more information.

3. Create a namespace where your policies are created to control access and to organize the policies. Run the following command:

   ```
   oc create namespace policy-namespace
   ```

4. Run the following command to deploy the policies to your cluster:

   ```
   ./deploy.sh -u https://github.com/<your-repository>/policy-collection -p my-policies -n policy-namespace
   ```

   Replace **your-repository** with your Git user name or repository name.

   Note: For reference, the full list of arguments for the **deploy.sh** script uses the following syntax:

   ```
   ./deploy.sh [-u <url>] [-b <branch>] [-p <path/to/dir>] [-n <namespace>] [-a|--name <resource-name>]
   ```

   View the following explanations for each argument:

   - URL: The URL to the repository that you forked from the main **policy-collection** repository. The default URL is **https://github.com/stolostron/policy-collection.git**.

   - Branch: Branch of the Git repository to point to. The default branch is **main**.

   - Subdirectory Path: The subdirectory path you created to contain the policies you want to use. In the previous sample, we used the **my-policies** subdirectory, but you can also specify which folder you want start with. For example, you can use **my-policies/AC-Access-Control**. The default folder is **stable**.

- Namespace: The namespace where the resources and policies are created on the hub cluster. These instructions use the **policy-namespace** namespace. The default namespace is **policies**.
- Name Prefix: Prefix for the **Channel** and **Subscription** resources. The default is **demo-stable-policies**.

After you run the **deploy.sh** script, any user with access to the repository can commit changes to the branch, which pushes changes to existing policies on your clusters.

Note: To deploy policies with subscriptions, complete the following steps:

1. Bind the **open-cluster-management:subscription-admin** ClusterRole to the user creating the subscription.
2. If you are using an allow list in the subscription, include the following API entries:

```
- apiVersion: policy.open-cluster-management.io/v1
  kinds:
    - "*"
- apiVersion: policy.open-cluster-management.io/v1beta1
  kinds:
    - "*"
- apiVersion: apps.open-cluster-management.io/v1
  kinds:
    - PlacementRule
- apiVersion: cluster.open-cluster-management.io/v1beta1
  kinds:
    - Placement
```

2.6.3.4. Verifying GitOps policy deployments from the console

Verify that your changes were applied to your policies from the console. You can also make more changes to your policy from the console, however the changes are reverted when the **Subscription** is reconciled with the Git repository. Complete the following steps:

1. Log in to your Red Hat Advanced Cluster Management cluster.
2. From the navigation menu, select **Governance**.
3. Locate the policies that you deployed in the table. Policies that are deployed using GitOps have a *Git* label in the *Source* column. Click the label to view the details for the Git repository.

2.6.3.4.1. Verifying GitOps policy deployments from the CLI

Complete the following steps:

1. Check for the following policy details:
   - Why is a specific policy compliant or non-compliant on the clusters that it was distributed to?
   - Are the policies applied to the correct clusters?
   - If this policy is not distributed to any clusters, why?

2. Identify the GitOps deployed policies that you created or modified. The GitOps deployed policies can be identified by the annotation that is applied automatically. Annotations for the GitOps deployed policies resemble the following paths:

   ```
   apps.open-cluster-management.io/hosting-deployable: policies/deploy-stable-policies-Policy-policy-role9

   apps.open-cluster-management.io/hosting-subscription: policies/demo-policies

   apps.open-cluster-management.io/sync-source: subgbk8s-policies/demo-policies
   ```

   GitOps annotations are valuable to see which subscription created the policy. You can also add your own labels to your policies so that you can write runtime queries that select policies based on labels.

   For example, you can add a label to a policy with the following command:

   ```
   oc label policy <policy-name> -n <policy-namespace> <key>=<value>
   ```

   Then, you can query policies that have labels with the following command:

   ```
   oc get policy -n <policy-namespace> -l <key>=<value>
   ```

Your policies are deployed using GitOps.

2.6.4. Support for templates in configuration policies

Configuration policies support the inclusion of Golang text templates in the object definitions. These templates are resolved at runtime either on the hub cluster or the target managed cluster using configurations related to that cluster. This gives you the ability to define configuration policies with dynamic content, and inform or enforce Kubernetes resources that are customized to the target cluster.

- Prerequisite
- Template functions
- Support for hub cluster templates in configuration policies
  - Template processing
  - Special annotation for reprocessing
  - Bypass template processing
  - Comparison of hub cluster and managed cluster templates

2.6.4.1. Prerequisite

- The template syntax must be conformed to the Golang template language specification, and the resource definition generated from the resolved template must be a valid YAML. See the Golang documentation about Package templates for more information. Any errors in template validation are recognized as policy violations. When you use a custom template function, the values are replaced at runtime.

2.6.4.2. Template functions

Template functions, such as resource-specific and generic **lookup** template functions, are available for referencing Kubernetes resources on the hub cluster (using the **{{hub ... hub}}** delimiters), or managed cluster (using the **{{ ... }}** delimiters). See Support for hub cluster templates in configuration policies for more details. The resource-specific functions are used for convenience and makes content of the resources more accessible. If you use the generic function, **lookup**, which is more advanced, it is best to be familiar with the YAML structure of the resource that is being looked up. In addition to these functions, utility functions like **base64encode**, **base64decode**, **indent**, **autoindent**, **toInt**, **toBool**, and more are also available.

To conform templates with YAML syntax, templates must be set in the policy resource as strings using quotes or a block character (**|** or **>**). This causes the resolved template value to also be a string. To override this, consider using **toInt** or **toBool** as the final function in the template to initiate further processing that forces the value to be interpreted as an integer or boolean respectively.

Continue reading to view descriptions and examples for some of the custom template functions that are supported:

- fromSecret function
- fromConfigmap function
- fromClusterClaim function
- lookup function
- base64enc function
- base64dec function
- indent function
- autoindent function
- toInt function
- toBool function
- protect function

2.6.4.2.1. fromSecret function

The **fromSecret** function returns the value of the given data key in the secret. View the following syntax for the function:

> func fromSecret (ns string, secretName string, datakey string) (dataValue string, err error)

When you use this function, enter the namespace, name, and data key of a Kubernetes **Secret** resource. You must use the same namespace that is used for the policy when using the function in a hub cluster template. See, Support for hub cluster templates in configuration policies for more details.

Note: When you use this function with hub cluster templates, the output is automatically encrypted using the the protect function.

You receive a policy violation if the Kubernetes **Secret** resource does not exist on the target cluster. If the data key does not exist on the target cluster, the value becomes an empty string. View the following configuration policy that enforces a **Secret** resource on the target cluster. The value for the **PASSWORD** data key is a template that references the secret on the target cluster:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromsecret
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
      apiVersion: v1
      data:
        USER_NAME: YWRtaW4=
        PASSWORD: '{{ fromSecret "default" "localsecret" "PASSWORD" }}'
      kind: Secret
      metadata:
        name: demosecret
        namespace: test
      type: Opaque
  remediationAction: enforce
  severity: low
```

2.6.4.2.2. fromConfigmap function

The **fromConfigmap** function returns the value of the given data key in the ConfigMap. View the following syntax for the function:

```
func fromConfigMap (ns string, configmapName string, datakey string) (dataValue string, err Error)
```

When you use this function, enter the namespace, name, and data key of a Kubernetes **ConfigMap** resource. You must use the same namespace that is used for the policy using the function in a hub cluster template. See, Support for hub cluster templates in configuration policies for more details. You receive a policy violation if the Kubernetes **ConfigMap** resource does not exist on the target cluster. If the data key does not exist on the target cluster, the value becomes an empty string. View the following configuration policy that enforces a Kubernetes resource on the target managed cluster. The value for the **log-file** data key is a template that retrieves the value of the **log-file** from the ConfigMap, **logs-config** from the **default** namespace, and the **log-level** is set to the data key **log-level**.

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromcm-lookup
  namespace: test-templates
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
```

```
    objectDefinition:
      kind: ConfigMap
      apiVersion: v1
      metadata:
        name: demo-app-config
        namespace: test
      data:
        app-name: sampleApp
        app-description: "this is a sample app"
        log-file: '{{ fromConfigMap "default" "logs-config" "log-file" }}'
        log-level: '{{ fromConfigMap "default" "logs-config" "log-level" }}'
  remediationAction: enforce
  severity: low
```

2.6.4.2.3. fromClusterClaim function

The **fromClusterClaim** function returns the value of the **Spec.Value** in the **ClusterClaim** resource. View the following syntax for the function:

```
func fromClusterClaim (clusterclaimName string) (value map[string]interface{}, err Error)
```

When you use this function, enter the name of a Kubernetes **ClusterClaim** resource. You receive a policy violation if the **ClusterClaim** resource does not exist. View the following example of the configuration policy that enforces a Kubernetes resource on the target managed cluster. The value for the **platform** data key is a template that retrieves the value of the **platform.open-cluster-management.io** cluster claim. Similarly, it retrieves values for **product** and **version** from the **ClusterClaim**:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-clusterclaims
  namespace: default
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
      kind: ConfigMap
      apiVersion: v1
      metadata:
        name: sample-app-config
        namespace: default
      data:
        # Configuration values can be set as key-value properties
        platform: '{{ fromClusterClaim "platform.open-cluster-management.io" }}'
        product: '{{ fromClusterClaim "product.open-cluster-management.io" }}'
        version: '{{ fromClusterClaim "version.openshift.io" }}'
  remediationAction: enforce
  severity: low
```

2.6.4.2.4. lookup function

The **lookup** function returns the Kubernetes resource as a JSON compatible map. If the requested resource does not exist, an empty map is returned. If the resource does not exist and the value is provided to another template function, you might get the following error: **invalid value; expected string**.

Note: Use the **default** template function, so the correct type is provided to later template functions. See the *Open source community functions* section.

View the following syntax for the function:

```
func lookup (apiversion string, kind string, namespace string, name string) (value string, err Error)
```

When you use this function, enter the API version, kind, namespace, and name of the Kubernetes resource. You must use the same namespace that is used for the policy within the hub cluster template. See, Support for hub cluster templates in configuration policies for more details. View the following example of the configuration policy that enforces a Kubernetes resource on the target managed cluster. The value for the **metrics-url** data key is a template that retrieves the **v1/Service** Kubernetes resource **metrics** from the **default** namespace, and is set to the value of the **Spec.ClusterIP** in the queried resource:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-lookup
  namespace: test-templates
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
      kind: ConfigMap
      apiVersion: v1
      metadata:
        name: demo-app-config
        namespace: test
      data:
        # Configuration values can be set as key-value properties
        app-name: sampleApp
        app-description: "this is a sample app"
        metrics-url: |
          http://{{ (lookup "v1" "Service" "default" "metrics").spec.clusterIP }}:8080
  remediationAction: enforce
  severity: low
```

2.6.4.2.5. base64enc function

The **base64enc** function returns a **base64** encoded value of the input **data string**. View the following syntax for the function:

```
func base64enc (data string) (enc-data string)
```

When you use this function, enter a string value. View the following example of the configuration policy that uses the **base64enc** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromsecret
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    data:
      USER_NAME: '{{ fromConfigMap "default" "myconfigmap" "admin-user" | base64enc }}'
```

2.6.4.2.6. base64dec function

The **base64dec** function returns a **base64** decoded value of the input **enc-data string**. View the following syntax for the function:

```
func base64dec (enc-data string) (data string)
```

When you use this function, enter a string value. View the following example of the configuration policy that uses the **base64dec** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromsecret
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    data:
      app-name: |
         "{{ ( lookup "v1"  "Secret" "testns" "mytestsecret") .data.appname ) | base64dec }}"
```

2.6.4.2.7. indent function

The **indent** function returns the padded **data string**. View the following syntax for the function:

```
func indent (spaces  int,  data string) (padded-data string)
```

When you use this function, enter a data string with the specific number of spaces. View the following example of the configuration policy that uses the **indent** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromsecret
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    data:
      Ca-cert:  |
        {{ ( index ( lookup "v1" "Secret" "default" "mycert-tls"  ).data  "ca.pem"  ) |  base64dec | indent 4
}}
```

2.6.4.2.8. autoindent function

The **autoindent** function acts like the **indent** function that automatically determines the number of leading spaces based on the number of spaces before the template. View the following example of the configuration policy that uses the **autoindent** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-fromsecret
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    data:
      Ca-cert:  |
        {{ ( index ( lookup "v1" "Secret" "default" "mycert-tls"  ).data  "ca.pem"  ) |  base64dec |
autoindent }}
```

2.6.4.2.9. toInt function

The **toInt** function casts and returns the integer value of the input value. Also, when this is the last function in the template, there is further processing of the source content. This is to ensure that the value is interpreted as an integer by the YAML. View the following syntax for the function:

```
func toInt (input interface{}) (output int)
```

When you use this function, enter the data that needs to be casted as an integer. View the following example of the configuration policy that uses the **toInt** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-template-function
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    spec:
      vlanid:  |
        {{ (fromConfigMap "site-config" "site1" "vlan")  | toInt }}
```

2.6.4.2.10. toBool function

The **toBool** function converts the input string into a boolean, and returns the boolean. Also, when this is the last function in the template, there is further processing of the source content. This is to ensure that the value is interpreted as a boolean by the YAML. View the following syntax for the function:

```
func toBool (input string) (output bool)
```

When you use this function, enter the string data that needs to be converted to a boolean. View the following example of the configuration policy that uses the **toBool** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-template-function
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    spec:
      enabled: |
        {{ (fromConfigMap "site-config" "site1" "enabled")  | toBool }}
```

2.6.4.2.11. protect function

The **protect** function enables you to encrypt a string in a hub cluster policy template. It is automatically decrypted on the managed cluster when the policy is evaluated. View the following example of the configuration policy that uses the **protect** function:

```
apiVersion: policy.open-cluster-management.io/v1
kind: ConfigurationPolicy
metadata:
  name: demo-template-function
  namespace: test
spec:
  namespaceSelector:
    exclude:
    - kube-*
    include:
    - default
  object-templates:
  - complianceType: musthave
    objectDefinition:
    ...
    spec:
      enabled: |
        {{hub "(lookup "v1" "Secret" "default" "my-hub-secret").data.message | protect hub}}
```

In the previous YAML example, there is an existing hub cluster policy template that is defined to use the **lookup** function. On the replicated policy in the managed cluster namespace, the value might resemble the following syntax: **$ocm_encrypted:okrrBqt72oI+3WT/0vxeI3vGa+wpLD7Z0ZxFMLvL204=**

Each encryption algorithm used is AES-CBC using 256-bit keys. Each encryption key is unique per managed cluster and is automatically rotated every 30 days.

This ensures that your decrypted value is to never be stored in the policy on the managed cluster.

To force an immediate rotation, delete the **policy.open-cluster-management.io/last-rotated** annotation on the **policy-encryption-key** Secret in the managed cluster namespace on the hub cluster. Policies are then reprocessed to use the new encryption key.

2.6.4.3. Support for hub cluster templates in configuration policies

In addition to managed cluster templates that are dynamically customized to the target cluster, Red Hat Advanced Cluster Management also supports hub cluster templates to define configuration policies using values from the hub cluster. This combination reduces the need to create separate policies for each target cluster or hardcode configuration values in the policy definitions.

Hub cluster templates are based on Golang text template specifications, and the **{{hub … hub}}** delimiter indicates a hub cluster template in a configuration policy.

For security, both resource-specific and the generic lookup functions in hub cluster templates are restricted to the namespace of the policy on the hub cluster. View the Comparison of hub and managed cluster templates for additional details.

Important: If you use hub cluster templates to propagate secrets or other sensitive data, the sensitive data exists in the managed cluster namespace on the hub cluster and on the managed clusters where that policy is distributed. The template content is expanded in the policy, and policies are not encrypted by the OpenShift Container Platform ETCD encryption support. To address this, use **fromSecret**, which automatically encrypts the values from the Secret, or **protect** to encrypt other values.

2.6.4.3.1. Template processing

A configuration policy definition can contain both hub cluster and managed cluster templates. Hub cluster templates are processed first on the hub cluster, then the policy definition with resolved hub cluster templates is propagated to the target clusters. On the managed cluster, the **ConfigurationPolicyController** processes any managed cluster templates in the policy definition and then enforces or verifies the fully resolved object definition.

2.6.4.3.2. Special annotation for reprocessing

Policies are processed on the hub cluster only upon creation or after an update. Therefore, hub cluster templates are only resolved to the data in the referenced resources upon policy creation or update. Any changes to the referenced resources are not automatically synced to the policies.

A special annotation, **policy.open-cluster-management.io/trigger-update** can be used to indicate changes to the data referenced by the templates. Any change to the special annotation value initiates template processing, and the latest contents of the referenced resource are read and updated in the policy definition that is the propagator for processing on managed clusters. A typical way to use this annotation is to increment the value by one each time.

2.6.4.3.3. Bypass template processing

You might create a policy that contains a template that is not intended to be processed by Red Hat Advanced Cluster Management. By default, Red Hat Advanced Cluster Management processes all templates.

To bypass template processing for your hub cluster, you must change **{{ template content }}** to **{{ `{{ template content }}` }}**.

Alternatively, you can add the following annotation in the **ConfigurationPolicy** section of your **Policy**: **policy.open-cluster-management.io/disable-templates: "true"**. When this annotation is included, the previous workaround is not necessary. Template processing is bypassed for the **ConfigurationPolicy**.

See the following table for a comparison of hub cluster and managed cluster templates:

2.6.4.3.4. Comparison of hub cluster and managed cluster templates

Table 2.6. Comparison table

Templates	Hub cluster	Managed cluster
Syntax	Golang text template specification	Golang text template specification
Delimiter	{{hub … hub}}	{{ … }}

Templates	Hub cluster	Managed cluster
Context	A **.ManagedClusterName** variable is available, which at runtime, resolves to the name of the target cluster where the policy is propagated.	No context variables
Access control	You can only reference namespaced Kubernetes objects that are in the same namespace as the **Policy** resource.	You can reference any resource on the cluster.
Functions	A set of template functions that support dynamic access to Kubernetes resources and string manipulation. See Template functions for more information. See the Access control row for lookup restrictions. The **fromSecret** template function on the hub cluster stores the resulting value as an encrypted string on the replicated policy, in the managed cluster namespace. The equivalent call might use the following syntax: **{{hub "(lookup "v1" "Secret" "default" "my-hub-secret").data.message | protect hub}}**	A set of template functions support dynamic access to Kubernetes resources and string manipulation. See Template functions for more information.

Templates	Hub cluster	Managed cluster
Function output storage	The output of template functions are stored in **Policy** resource objects in each applicable managed cluster namespace on the hub cluster, before it is synced to the managed cluster. This means that any sensitive results from template functions are readable by anyone with read access to the **Policy** resource objects on the hub cluster, and read access with **ConfigurationPolicy** resource objects on the managed clusters. Additionally, if etcd encryption is enabled, the **Policy** and **ConfigurationPolicy** resource objects are not encrypted. It is best to carefully consider this when using template functions that return sensitive output (e.g. from a secret).	The output of template functions are not stored in policy related resource objects.
Processing	Processing occurs at runtime on the hub cluster during propagation of replicated policies to clusters. Policies and the hub cluster templates within the policies are processed on the hub cluster only when templates are created or updated.	Processing occurs in the **ConfigurationPolicyController** on the managed cluster. Policies are processed periodically, which automatically updates the resolved object definition with data in the referenced resources.
Processing errors	Errors from the hub cluster templates are displayed as violations on the managed clusters the policy applies to.	Errors from the managed cluster templates are displayed as violations on the specific target cluster where the violation occurred.

2.6.5. Governance metric

The policy framework exposes metrics that show policy distribution and compliance. Use the **policy_governance_info** metric on the hub cluster to view trends and analyze any policy failures.

2.6.5.1. Metric overview

The **policy_governance_info** is collected by OpenShift Container Platform monitoring, and some aggregate data is collected by Red Hat Advanced Cluster Management observability, if it is enabled.

Note: If observability is enabled, you can enter a query for the metric from the Grafana *Explore* page.

When you create a policy, you are creating a *root* policy. The framework watches for root policies as well as **PlacementRules** and **PlacementBindings**, to determine where to create *propagated* policies in order to distribute the policy to managed clusters. For both root and propagated policies, a metric of **0** is recorded if the policy is compliant, and **1** if it is non-compliant.

The **policy_governance_info** metric uses the following labels:

- **type**: The label values are **root** or **propagated**.
- **policy**: The name of the associated root policy.
- **policy_namespace**: The namespace on the hub cluster where the root policy was defined.
- **cluster_namespace**: The namespace for the cluster where the policy is distributed.

These labels and values enable queries that can show us many things happening in the cluster that might be difficult to track.

Note: If the metrics are not needed, and there are any concerns about performance or security, this feature can be disabled. Set the **DISABLE_REPORT_METRICS** environment variable to **true** in the propagator deployment. You can also add **policy_governance_info** metric to the observability allowlist as a custom metric. See Adding custom metrics for more details.

2.6.6. Managing security policies

Create a security policy to report and validate your cluster compliance based on your specified security standards, categories, and controls. To create a policy for Red Hat Advanced Cluster Management for Kubernetes, you must create a YAML file on your managed clusters.

Note: You can copy and paste an existing policy in to the *Policy YAML*. The values for the parameter fields are automatically entered when you paste your existing policy. You can also search the contents in your policy YAML file with the search feature.

View the following sections:

- Creating a security policy
  - Creating a security policy from the command line interface
  - Viewing your security policy from the CLI
  - Creating a cluster security policy from the console
  - Viewing your security policy from the console
  - Creating policy sets from the CLI
  - Creating policy sets from the console
- Updating security policies
  - Disabling security policies
- Deleting a security policy
  - Deleting policy sets from the console

2.6.6.1. Creating a security policy

You can create a security policy from the command line interface (CLI) or from the console.

Required access: Cluster administrator

Important: You must define a placement rule and placement binding to apply your policy to a specific cluster. Enter a valid value for the *Cluster selector* field to define a **PlacementRule** and **PlacementBinding**. See Resources that support support set-based requirements in the Kubernetes documentation for a valid expression. View the definitions of the objects that are required for your Red Hat Advanced Cluster Management policy:

- *PlacementRule*: Defines a *cluster selector* where the policy must be deployed.
- *PlacementBinding*: Binds the placement to a placement rule.

View more descriptions of the policy YAML files in the Policy overview.

2.6.6.1.1. Creating a security policy from the command line interface

Complete the following steps to create a policy from the command line interface (CLI):

1. Create a policy by running the following command:

```
kubectl create -f policy.yaml -n <namespace>
```

2. Define the template that the policy uses. Edit your **.yaml** file by adding a **templates** field to define a template. Your policy might resemble the following YAML file:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy1
spec:
  remediationAction: "enforce" # or inform
  disabled: false # or true
  namespaces:
    include: ["default"]
    exclude: ["kube*"]
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          namespace: kube-system # will be inferred
          name: operator
        spec:
          remediationAction: "inform"
          object-templates:
          - complianceType: "musthave" # at this level, it means the role must exist and must
have the following rules
            apiVersion: rbac.authorization.k8s.io/v1
            kind: Role
            metadata:
              name: example
            objectDefinition:
              rules:
                - complianceType: "musthave" # at this level, it means if the role exists the rule is a musthave
                  apiGroups: ["extensions", "apps"]
                  resources: ["deployments"]
                  verbs: ["get", "list", "watch", "create", "delete","patch"]
```

3. Define a **PlacementRule**. Be sure to change the **PlacementRule** to specify the clusters where the policies need to be applied, either by **clusterNames**, or **clusterLabels**. View Placement rule samples overview
   Your **PlacementRule** might resemble the following content:

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement1
spec:
  clusterConditions:
    - type: ManagedClusterConditionAvailable
      status: "True"
  clusterNames:
  - "cluster1"
  - "cluster2"
  clusterLabels:
    matchLabels:
      cloud: IBM
```

4. Define a **PlacementBinding** to bind your policy and your **PlacementRule**. Your **PlacementBinding** might resemble the following YAML sample:

```
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding1
placementRef:
  name: placement1
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
subjects:
- name: policy1
  apiGroup: policy.open-cluster-management.io
  kind: Policy
```

2.6.6.1.1.1. Viewing your security policy from the CLI

Complete the following steps to view your security policy from the CLI:

1. View details for a specific security policy by running the following command:

```
kubectl get securitypolicy <policy-name> -n <namespace> -o yaml
```

2. View a description of your security policy by running the following command:

```
kubectl describe securitypolicy <name> -n <namespace>
```

2.6.6.1.2. Creating a cluster security policy from the console

After you log into your Red Hat Advanced Cluster Management, navigate to the *Governance* page and click **Create policy**.

As you create your new policy from the console, a YAML file is also created in the YAML editor. To view the YAML editor, select the toggle at the beginning of the *Create policy* form to enable it.

Complete the *Create policy* form, then select the **Submit** button.

Your YAML file might resemble the following policy:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-pod
  annotations:
    policy.open-cluster-management.io/categories:
'SystemAndCommunicationsProtections,SystemAndInformationIntegrity'
    policy.open-cluster-management.io/controls: 'control example'
    policy.open-cluster-management.io/standards: 'NIST,HIPAA'
spec:
  complianceType: musthave
  namespaces:
    exclude: ["kube*"]
    include: ["default"]
  object-templates:
  - complianceType: musthave
    objectDefinition:
      apiVersion: v1
      kind: Pod
      metadata:
        name: pod1
      spec:
        containers:
        - name: pod-name
          image: 'pod-image'
          ports:
          - containerPort: 80
  remediationAction: enforce
  disabled: false

---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: binding-pod
placementRef:
  name: placement-pod
  kind: PlacementRule
  apiGroup: apps.open-cluster-management.io
subjects:
- name: policy-pod
  kind: Policy
  apiGroup: policy.open-cluster-management.io
```

```
---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: placement-pod
spec:
  clusterConditions:
    - type: ManagedClusterConditionAvailable
      status: "True"
  clusterLabels:
    matchLabels:
      cloud: "IBM"
```

Click **Create Policy**. A security policy is created from the console.

2.6.6.1.2.1. Viewing your security policy from the console

View any security policy and its status from the console. Navigate to the *Governance* page to view a table list of your policies. **Note**: You can filter the table list of your policies by selecting the *Policies* tab or *Cluster violations* tab.

Select one of your policies to view more details. The *Details*, *Clusters*, and *Templates* tabs are displayed. When the cluster or policy status cannot be determined, the following message is displayed: **No status**.

2.6.6.1.3. Creating policy sets from the CLI

By default, the policy set is created with no policies or placements. You must create a placement for the policy set and have at least one policy that exists on your cluster. When you create a policy set, you can add numerous policies. Run the following command to create a policy set from the CLI:

```
kubectl apply -f <policyset-filename>
```

2.6.6.1.4. Creating policy sets from the console

From the navigation menu, select **Governance**. Then select the *Policy sets* tab. Select the **Create policy set** button and complete the form. After you add the details for your policy set, select the **Submit** button.

View the stable **Policyets**, which require the policy generator for deployment, PolicySets-- Stable.

2.6.6.2. Updating security policies

Learn to update security policies by viewing the following section.

2.6.6.2.1. Adding a policy to a policy set from the CLI

Run the following command to edit your policy set: **kubectl edit policysets your-policyset-name**

Add the policy name to the list in the **policies** section of the policy set. Apply your added policy in the placement section of your policy set with the following command: **kubectl apply -f your-added-policy.yaml**. A **PlacementBinding** and **PlacementRule** are created. **Note**: If you delete the placement binding, the policy is still placed by the policy set.

2.6.6.2.2. Adding a policy to a policy set from the console

Add a policy to the policy set by selecting the ***Policy sets*** tab. Select the Actions icon and select **Edit**. The *Edit policy set* form appears.

Navigate to the *Policies* section of the form to select a policy to add to the policy set.

2.6.6.2.3. Disabling security policies

Your policy is enabled by default. Disable your policy from the console.

After you log into your Red Hat Advanced Cluster Management for Kubernetes console, navigate to the *Governance* page to view a table list of your policies.

Select the **Actions** icon > **Disable policy**. The *Disable Policy* dialog box appears.

Click **Disable policy**. Your policy is disabled.

2.6.6.3. Deleting a security policy

Delete a security policy from the CLI or the console.

- Delete a security policy from the CLI:

    a. Delete a security policy by running the following command:

    ```
    kubectl delete policy <securitypolicy-name> -n <open-cluster-management-namespace>
    ```

    After your policy is deleted, it is removed from your target cluster or clusters. Verify that your policy is removed by running the following command: **kubectl get policy <securitypolicy-name> -n <open-cluster-management-namespace>**

- Delete a security policy from the console:
  From the navigation menu, click **Governance** to view a table list of your policies. Click the **Actions** icon for the policy you want to delete in the policy violation table.

  Click **Remove**. From the *Remove policy* dialog box, click **Remove policy**

2.6.6.3.1. Deleting policy sets from the console

From the *Policy sets* tab, select the **Actions** icon for the policy set. When you click **Delete**, the *Permanently delete Policyset?* dialogue box appears.

Click the **Delete** button.

To manage other policies, see Managing security policies for more information. Refer to Governance for more topics about policies.

2.6.7. Managing configuration policies

Learn to create, apply, view, and update your configuration policies. As a reminder, the following resources are configuration policies:

- Memory usage policy

- Namespace policy

- Image vulnerability policy

- Pod policy
- Pod security policy
- Role policy
- Role binding policy
- Security content constraints (SCC) policy
- ETCD encryption policy
- Compliance operator policy

Required access: Administrator or cluster administrator

- Creating a configuration policy
  - Creating a configuration policy from the CLI
  - Viewing your configuration policy from the CLI
  - Creating a configuration policy from the console
  - Viewing your configuration policy from the console
- Updating configuration policies
  - Disabling configuration policies
- Deleting a configuration policy

2.6.7.1. Creating a configuration policy

You can create a YAML file for your configuration policy from the command line interface (CLI) or from the console. View the following sections to create a configuration policy:

2.6.7.1.1. Creating a configuration policy from the CLI

Complete the following steps to create a configuration policy from the (CLI):

1. Create a YAML file for your configuration policy. Run the following command:

```
kubectl create -f configpolicy-1.yaml
```

Your configuration policy might resemble the following policy:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-1
  namespace: kube-system
spec:
  namespaces:
    include: ["default", "kube-*"]
    exclude: ["kube-system"]
  remediationAction: inform
  disabled: false
  complianceType: musthave
  object-templates:
  ...
```

2. Apply the policy by running the following command:

```
kubectl apply -f <policy-file-name>  --namespace=<namespace>
```

3. Verify and list the policies by running the following command:

```
kubectl get policy --namespace=<namespace>
```

Your configuration policy is created.

2.6.7.1.2. Viewing your configuration policy from the CLI

Complete the following steps to view your configuration policy from the CLI:

1. View details for a specific configuration policy by running the following command:

```
kubectl get policy <policy-name> -n <namespace> -o yaml
```

2. View a description of your configuration policy by running the following command:

```
kubectl describe policy <name> -n <namespace>
```

2.6.7.1.3. Creating a configuration policy from the console

As you create a configuration policy from the console, a YAML file is also created in the YAML editor.

Log in to your cluster from the console, and select **Governance** from the navigation menu.

Click **Create policy**. Specify the policy you want to create by selecting one of the configuration policies for the specification parameter.

Continue with configuration policy creation by completing the policy form. Enter or select the appropriate values for the following fields:

- Name
- Specifications
- Cluster selector
- Remediation action
- Standards
- Categories
- Controls

Click **Create**. Your configuration policy is created.

2.6.7.1.4. Viewing your configuration policy from the console

View any configuration policy and its status from the console.

After you log into your cluster from the console, select **Governance** to view a table list of your policies. **Note**: You can filter the table list of your policies by selecting the *All policies* tab or *Cluster violations* tab.

Select one of your policies to view more details. The *Details*, *Clusters*, and *Templates* tabs are displayed.

2.6.7.2. Updating configuration policies

Learn to update configuration policies by viewing the following section.

2.6.7.2.1. Disabling configuration policies

Disable your configuration policy. Similar to the instructions mentioned earlier, log in and navigate to the *Governance* page.

Select the **Actions** icon for a configuration policy from the table list, then click **Disable**. The *Disable Policy* dialog box appears.

Click **Disable policy**.

Your configuration policy is disabled.

2.6.7.3. Deleting a configuration policy

Delete a configuration policy from the CLI or the console.

- Delete a configuration policy from the CLI:

  a. Delete a configuration policy by running the following command:

     ```
     kubectl delete policy <policy-name> -n <namespace>
     ```

     After your policy is deleted, it is removed from your target cluster or clusters.

  b. Verify that your policy is removed by running the following command:

  ```
  kubectl get policy <policy-name> -n <namespace>
  ```

- Delete a configuration policy from the console:
  From the navigation menu, click **Governance** to view a table list of your policies.

  Click the **Actions** icon for the policy you want to delete in the policy violation table. Then click **Remove**. From the *Remove policy* dialog box, click **Remove policy**.

Your policy is deleted.

See configuration policy samples that are supported by Red Hat Advanced Cluster Management from the CM-Configuration-Management folder.

Alternatively, you can refer to Kubernetes configuration policy controller to view other configuration policies that are monitored by the controller. For details to manage other policies, refer to Managing security policies.

2.6.8. Managing gatekeeper operator policies

Use the gatekeeper operator policy to install the gatekeeper operator and gatekeeper on a managed cluster. Learn to create, view, and update your gatekeeper operator policies in the following sections.

Required access: Cluster administrator

- Installing gatekeeper using a gatekeeper operator policy
- Creating a gatekeeper policy from the console
    - Gatekeeper operator CR
- Upgrading gatekeeper and the gatekeeper operator
- Updating gatekeeper operator policy
    - Viewing gatekeeper operator policy from the console
    - Disabling gatekeeper operator policy
- Deleting gatekeeper operator policy
- Uninstalling gatekeeper policy, gatekeeper, and gatekeeper operator policy

2.6.8.1. Installing gatekeeper using a gatekeeper operator policy

Use the governance framework to install the gatekeeper operator. Gatekeeper operator is available in the OpenShift Container Platform catalog. See *Adding Operators to a cluster* in the OpenShift Container Platform documentation for more information.

Use the configuration policy controller to install the gatekeeper operator policy. During the install, the operator group and subscription pull the gatekeeper operator to install it in your managed cluster. Then, the gatekeeper operator creates a gatekeeper CR to configure gatekeeper. View the Gatekeeper operator CR sample.

Gatekeeper operator policy is monitored by the Red Hat Advanced Cluster Management configuration policy controller, where **enforce** remediation action is supported. Gatekeeper operator policies are created automatically by the controller when set to **enforce**.

2.6.8.2. Creating a gatekeeper policy from the console

Install the gatekeeper policy by creating a gatekeeper policy from the console.

After you log into your cluster, navigate to the *Governance* page.

Select **Create policy**. As you complete the form, select **GatekeeperOperator** from the *Specifications* field. The parameter values for your policy are automatically populated and the policy is set to **inform** by default. Set your remediation action to **enforce** to install gatekeeper. See **policy-gatekeeper-operator.yaml** to view an the sample.

+ **Note:** Consider that default values can be generated by the operator. See Gatekeeper Helm Chart for an explanation of the optional parameters that can be used for the gatekeeper operator policy.

2.6.8.2.1. Gatekeeper operator CR

```
apiVersion: operator.gatekeeper.sh/v1alpha1
kind: Gatekeeper
metadata:
  name: gatekeeper
spec:
  audit:
    replicas: 1
    logLevel: DEBUG
    auditInterval: 10s
    constraintViolationLimit: 55
    auditFromCache: Enabled
    auditChunkSize: 66
    emitAuditEvents: Enabled
    resources:
      limits:
        cpu: 500m
        memory: 150Mi
      requests:
        cpu: 500m
        memory: 130Mi
  validatingWebhook: Enabled
  webhook:
    replicas: 2
    logLevel: ERROR
    emitAdmissionEvents: Enabled
    failurePolicy: Fail
    resources:
      limits:
        cpu: 480m
        memory: 140Mi
      requests:
        cpu: 400m
        memory: 120Mi
  nodeSelector:
    region: "EMEA"
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              auditKey: "auditValue"
          topologyKey: topology.kubernetes.io/zone
  tolerations:
    - key: "Example"
      operator: "Exists"
      effect: "NoSchedule"
  podAnnotations:
    some-annotation: "this is a test"
    other-annotation: "another test"
```

2.6.8.3. Upgrading gatekeeper and the gatekeeper operator

You can upgrade the versions for gatekeeper and the gatekeeper operator. Complete the following steps:

- When you install the gatekeeper operator with the gatekeeper operator policy, notice the value for **installPlanApproval**. The operator upgrades automatically when **installPlanApproval** is set to **Automatic**. You must approve the upgrade of the gatekeeper operator manually, for each cluster, when **installPlanApproval** is set to **Manual**.

2.6.8.4. Updating gatekeeper operator policy

Learn to update the gatekeeper operator policy by viewing the following section.

2.6.8.4.1. Viewing gatekeeper operator policy from the console

View your gatekeeper operator policy and its status from the console.

After you log into your cluster from the console, click **Governance** to view a table list of your policies. **Note**: You can filter the table list of your policies by selecting the *Policies* tab or *Cluster violations* tab.

Select the **policy-gatekeeper-operator** policy to view more details. View the policy violations by selecting the *Clusters* tab.

2.6.8.4.2. Disabling gatekeeper operator policy

Disable your gatekeeper operator policy.

After you log into your Red Hat Advanced Cluster Management for Kubernetes console, navigate to the *Governance* page to view a table list of your policies.

Select the **Actions** icon for the **policy-gatekeeper-operator** policy, then click **Disable**. The *Disable Policy* dialog box appears.

Click **Disable policy**. Your **policy-gatekeeper-operator** policy is disabled.

2.6.8.5. Deleting gatekeeper operator policy

Delete the gatekeeper operator policy from the CLI or the console.

- Delete gatekeeper operator policy from the CLI:

  a. Delete gatekeeper operator policy by running the following command:

  ```
  kubectl delete policy <policy-gatekeeper-operator-name> -n <namespace>
  ```

  After your policy is deleted, it is removed from your target cluster or clusters.

  b. Verify that your policy is removed by running the following command:

  ```
  kubectl get policy <policy-gatekeeper-operator-name> -n <namespace>
  ```

- Delete gatekeeper operator policy from the console:
  Navigate to the *Governance* page to view a table list of your policies.

  Similar to the previous console instructions, click the **Actions** icon for the **policy-gatekeeper-operator** policy. Click **Remove** to delete the policy. From the *Remove policy* dialog box, click **Remove policy**.

Your gatekeeper operator policy is deleted.

2.6.8.6. Uninstalling gatekeeper policy, gatekeeper, and gatekeeper operator policy

Complete the following steps to uninstall gatekeeper policy, gatekeeper, and gatekeeper operator policy:

1. Remove the gatekeeper **Constraint** and **ConstraintTemplate** that is applied on your managed cluster:
   a. Edit your gatekeeper operator policy. Locate the **ConfigurationPolicy** template that you used to create the gatekeeper **Constraint** and **ConstraintTemplate**.
   b. Change the value for **complianceType** of the **ConfigurationPolicy** template to **mustnothave**.
   c. Save and apply the policy.
2. Remove gatekeeper instance from your managed cluster:
   a. Edit your gatekeeper operator policy. Locate the **ConfigurationPolicy** template that you used to create the Gatekeeper custom resource (CR).
   b. Change the value for **complianceType** of the **ConfigurationPolicy** template to **mustnothave**.
3. Remove the gatekeeper operator that is on your managed cluster:
   a. Edit your gatekeeper operator policy. Locate the **ConfigurationPolicy** template that you used to create the Subscription CR.
   b. Change the value for **complianceType** of the **ConfigurationPolicy** template to **mustnothave**.

Gatekeeper policy, gatekeeper, and gatekeeper operator policy are uninstalled.

See Integrating gatekeeper constraints and constraint templates for details about gatekeeper. For a list of topics to integrate third-party policies with the product, see Integrate third-party policy controllers.

2.7. SECURE THE HUB CLUSTER

Secure your Red Hat Advanced Cluster Management for Kubernetes installation by hardening the hub cluster security. Complete the following steps:

1. Secure Red Hat OpenShift Container Platform. For more information, see OpenShift Container Platform security and compliance.
2. Setup role-based access control (RBAC). For more information, see Role-based access control.
3. Customize certificates, see Certificates.
4. Define your cluster credentials, see Managing credentials overview
5. Review the policies that are available to help you harden your cluster security. See Supported policies

2.8. INTEGRITY SHIELD PROTECTION (TECHNOLOGY PREVIEW)

Integrity shield is a tool that helps with integrity control for enforcing signature verification for any requests to create, or update resources. Integrity shield supports Open Policy Agent (OPA) and Gatekeeper, verifies if the requests have a signature, and blocks any unauthorized requests according to the defined constraint.

See the following integrity shield capabilities:

- Support the deployment of authorized Kubernetes manifests only.
- Support zero-drift in resource configuration unless the resource is added to the allowlist.
- Perform all integrity verification on the cluster such as enforcing the admission controller.
- Monitor resources continuously to report if unauthorized Kubernetes resources are deployed on the cluster.
- X509, GPG, and Sigstore signing are supported to sign Kubernetes manifest YAML files. Kubernetes integrity shield supports Sigstore signing by using the **k8s-manifest-sigstore**.

2.8.1. Integrity shield architecture

Integrity shield consists of two main components, API and Observer. Integrity shield operator supports the installation and management of the integrity shield components on your cluster. View the following description of the components:

- *Integrity shield API* receives a Kubernetes resource from the OPA or gatekeeper, validates the resource that is included in the admission request, and sends the verification result to the OPA or gatekeeper. The integrity shield API uses the **verify-resource** feature of the **k8s-manifest-sigstore** internally to verify the Kubernetes manifest YAML file. Integrity shield API validates resources according to **ManifestingIntegrityConstraint**, which is a custom resource based on the constraint framework of OPA or gatekeeper.
- *Integrity shield Observer* continuously verifies Kubernetes resources on clusters according to **ManifestingIntegrityConstraint** resources and exports the results to resources called, **ManifestIntegrityState**. Integrity shield Observer also uses **k8s-manifest-sigstore** to verify signatures.

2.8.2. Supported versions

The following product versions support integrity shield protection:

- Red Hat OpenShift Container Platform 4.7.1 and later
- Kubernetes v1.19.7 and later
- **gatekeeper-operator.v-.2.0**
- **gatekeeper v3.5**

See Enable integrity shield protection (Technology Preview) for more details.

2.8.3. Enable integrity shield protection (Technology Preview)

Enable integrity shield protection in an Red Hat Advanced Cluster Management for Kubernetes cluster to protect the integrity of Kubernetes resources.

2.8.3.1. Prerequisites

The following prerequisites are required to enable integrity shield protection on a Red Hat Advanced Cluster Management managed cluster:

- Install an Red Hat Advanced Cluster Management hub cluster that has one or more managed clusters, along with cluster administrator access to the cluster to use the **oc** or **kubectl** commands.
- Install integrity shield. Before you install the integrity shield, you must install an Open Policy Agent or gatekeeper on your cluster. Complete the following steps to install the integrity shield operator:
  a. Install the integrity shield operator in a namespace for integrity shield by running the following command:

  ```
  kubectl create -f https://raw.githubusercontent.com/open-cluster-management/integrity-shield/master/integrity-shield-operator/deploy/integrity-shield-operator-latest.yaml
  ```

  b. Install the integrity shield custom resource with the following command:

  ```
  kubectl create -f https://raw.githubusercontent.com/open-cluster-management/integrity-shield/master/integrity-shield-operator/config/samples/apis_v1_integrityshield.yaml -n integrity-shield-operator-system
  ```

  c. Integrity shield requires a pair of keys for signing and verifying signatures of resources that need to be protected in a cluster. Set up signing and verification key pair:
     - Generate a new GPG key with the following command:

     ```
     gpg --full-generate-key
     ```

     - Export your new GPG public key to a file with the following command:

     ```
     gpg --export signer@enterprise.com > /tmp/pubring.gpg
     ```

- Install **yq** to run the script for signing a Red Hat Advanced Cluster Management policy.
- Enabling integrity shield protection and signing Red Hat Advanced Cluster Management include retrieving and committing sources from the **integrity-shield** repository. You must install Git.

2.8.3.2. Enabling integrity shield protection

Enable the integrity shield on your Red Hat Advanced Cluster Management managed cluster by completing the following steps:

1. Create a namespace on your hub cluster for the integrity shield. Run the following command:

   ```
   oc create ns your-integrity-shield-ns
   ```

2. Deploy a verification key to a Red Hat Advanced Cluster Management managed cluster. As a reminder, you must create signing and verification keys. Run the **acm-verification-key-setup.sh** on your hub cluster to setup a verification key. Run the following command:

```
curl -s  https://raw.githubusercontent.com/stolostron/integrity-shield/master/scripts/ACM/acm-
verification-key-setup.sh | bash -s \
          --namespace integrity-shield-operator-system  \
          --secret keyring-secret  \
          --path /tmp/pubring.gpg \
          --label environment=dev  |  oc apply -f -
```

To remove the verification key, run the following command:

```
curl -s  https://raw.githubusercontent.com/stolostron/integrity-shield/master/scripts/ACM/acm-
verification-key-setup.sh | bash -s - \
          --namespace integrity-shield-operator-system  \
          --secret keyring-secret  \
          --path /tmp/pubring.gpg \
          --label environment=dev  |  oc delete -f -
```

3. Create a Red Hat Advanced Cluster Management policy named **policy-integrity-shield** on your hub cluster.

    a. Retrieve the **policy-integrity-shield** policy from the **policy-collection** repository. Be sure to fork the repository.

    b. Configure the namespace to deploy the integrity shield on a Red Hat Advanced Cluster Management managed cluster by updating the **remediationAction** parameter value, from **inform** to **enforce**.

    c. Configure a email for the signer and verification key by updating the **signerConfig** section.

    d. Configure the **PlacementRule** which determines what Red Hat Advanced Cluster Management managed clusters that integrity shield should be deployed to.

    e. Sign **policy-integrity-shield.yaml** by running the following command:

    ```
    curl -s  https://raw.githubusercontent.com/stolostron/integrity-shield/master/scripts/gpg-
    annotation-sign.sh | bash -s \
             signer@enterprise.com \
             policy-integrity-shield.yaml
    ```

    Note: You must create a new signature whenever you change the policy and apply to other clusters. Otherwise, the change is blocked and not applied.

See **policy-integrity-shield** policy for an example.