kubernetes

Documentation    Kubernetes Blog    Training    Partners    Community    Case Studies    Versions ⌄    English ⌄

## KubeCon + CloudNativeCon 2024

Join us for three days of incredible opportunities to collaborate, learn and share with the cloud native community.
Buy your ticket now! 12 - 15 November | Salt Lake City

KubeCon | CloudNativeCon
—— North America 2024 ——

Kubernetes Documentation / Reference / API Access Control / Using RBAC Authorization

# Using RBAC Authorization

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

RBAC authorization uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing you to dynamically configure policies through the Kubernetes API.

To enable RBAC, start the API server with the `--authorization-mode` flag set to a comma-separated list that includes `RBAC`; for example:

```
kube-apiserver --authorization-mode=Example,RBAC --other-options -
```

## API objects

The RBAC API declares four kinds of Kubernetes object: *Role*, *ClusterRole*, *RoleBinding* and *ClusterRoleBinding*. You can describe or amend the RBAC

objects using tools such as `kubectl`, just like any other Kubernetes object.

> **Caution:**
>
> These objects, by design, impose access restrictions. If you are making changes to a cluster as you learn, see privilege escalation prevention and bootstrapping to understand how those restrictions can prevent you making some changes.

## Role and ClusterRole

An RBAC *Role* or *ClusterRole* contains rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules).

A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs in.

ClusterRole, by contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can't be both.

ClusterRoles have several uses. You can use a ClusterRole to:

1. define permissions on namespaced resources and be granted access within individual namespace(s)
2. define permissions on namespaced resources and be granted access across all namespaces
3. define permissions on cluster-scoped resources

If you want to define a role within a namespace, use a Role; if you want to define a role cluster-wide, use a ClusterRole.

## Role example

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

## ClusterRole example

A ClusterRole can be used to grant the same permissions as a Role. Because ClusterRoles are cluster-scoped, you can also use them to grant access to:

- cluster-scoped resources (like nodes)

- non-resource endpoints (like `/healthz`)

- namespaced resources (like Pods), across all namespaces

  For example: you can use a ClusterRole to allow a particular user to run `kubectl get pods --all-namespaces`

Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
```

```
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secr
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

The name of a Role or a ClusterRole object must be a valid path segment name.

## RoleBinding and ClusterRoleBinding

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of *subjects* (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.

The name of a RoleBinding or ClusterRoleBinding object must be a valid path segment name.

## RoleBinding examples

Here is an example of a RoleBinding that grants the "pod-reader" Role to the user "jane" within the "default" namespace. This allows "jane" to read pods in the "default" namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" na
# You need to already have a Role named "pod-reader" in that names
```

```yaml
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or Clust
  apiGroup: rbac.authorization.k8s.io
```

A RoleBinding can also reference a ClusterRole to grant the permissions defined in that ClusterRole to resources inside the RoleBinding's namespace. This kind of reference lets you define a set of common roles across your cluster, then reuse them within multiple namespaces.

For instance, even though the following RoleBinding refers to a ClusterRole, "dave" (the subject, case sensitive) will only be able to read Secrets in the "development" namespace, because the RoleBinding's namespace (in its metadata) is "development".

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "dave" to read secrets in the "developm
# You need to already have a ClusterRole named "secret-reader".
kind: RoleBinding
metadata:
  name: read-secrets
  #
  # The namespace of the RoleBinding determines where the permissi
  # This only grants permissions within the "development" namespac
  namespace: development
subjects:
- kind: User
  name: dave # Name is case sensitive
```

```
      apiGroup: rbac.authorization.k8s.io
  roleRef:
    kind: ClusterRole
    name: secret-reader
    apiGroup: rbac.authorization.k8s.io
```

## ClusterRoleBinding example

To grant permissions across a whole cluster, you can use a ClusterRoleBinding. The following ClusterRoleBinding allows any user in the group "manager" to read secrets in any namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group t
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

After you create a binding, you cannot change the Role or ClusterRole that it refers to. If you try to change a binding's `roleRef`, you get a validation error. If you do want to change the `roleRef` for a binding, you need to remove the binding object and create a replacement.

There are two reasons for this restriction:

1. Making `roleRef` immutable allows granting someone `update` permission on an existing binding object, so that they can manage the list of subjects, without being able to change the role that is granted to those subjects.

2. A binding to a different role is a fundamentally different binding. Requiring a binding to be deleted/recreated in order to change the `roleRef` ensures the full list of subjects in the binding is intended to be granted the new role (as opposed to enabling or accidentally modifying only the roleRef without verifying all of the existing subjects should be given the new role's permissions).

The `kubectl auth reconcile` command-line utility creates or updates a manifest file containing RBAC objects, and handles deleting and recreating binding objects if required to change the role they refer to. See command usage and examples for more information.

## Referring to resources

In the Kubernetes API, most resources are represented and accessed using a string representation of their object name, such as `pods` for a Pod. RBAC refers to resources using exactly the same name that appears in the URL for the relevant API endpoint. Some Kubernetes APIs involve a *subresource*, such as the logs for a Pod. A request for a Pod's logs looks like:

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

In this case, `pods` is the namespaced resource for Pod resources, and `log` is a subresource of `pods`. To represent this in an RBAC role, use a slash ( `/` ) to delimit the resource and subresource. To allow a subject to read `pods` and also access the `log` subresource for each of those Pods, you write:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-and-pod-logs-reader
rules:
```

```yaml
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
```

You can also refer to resources by name for certain requests through the `resourceNames` list. When specified, requests can be restricted to individual instances of a resource. Here is an example that restricts its subject to only `get` or `update` a ConfigMap named `my-configmap`:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: configmap-updater
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Conf
  # objects is "configmaps"
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

> **Note:**
> You cannot restrict `create` or `deletecollection` requests by their resource name. For `create`, this limitation is because the name of the new object may not be known at authorization time. If you restrict `list` or `watch` by resourceName, clients must include a `metadata.name` field selector in their `list` or `watch` request that matches the specified resourceName in order to be authorized. For example, `kubectl get configmaps --field-selector=metadata.name=my-configmap`

Rather than referring to individual `resources`, `apiGroups`, and `verbs`, you can use the wildcard `*` symbol to refer to all such objects. For `nonResourceURLs`, you can use the wildcard `*` as a suffix glob match. For `resourceNames`, an empty set means that everything is allowed. Here is an example that allows access to perform any current and future action on all current and future resources in the `example.com` API group. This is similar to the built-in `cluster-admin` role.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: example.com-superuser # DO NOT USE THIS ROLE, IT IS JUST A
rules:
- apiGroups: ["example.com"]
  resources: ["*"]
  verbs: ["*"]
```

> **Caution:**
>
> Using wildcards in resource and verb entries could result in overly permissive access being granted to sensitive resources. For instance, if a new resource type is added, or a new subresource is added, or a new custom verb is checked, the wildcard entry automatically grants access, which may be undesirable. The principle of least privilege should be employed, using specific resources and verbs to ensure only the permissions required for the workload to function correctly are applied.

## Aggregated ClusterRoles

You can *aggregate* several ClusterRoles into one combined ClusterRole. A controller, running as part of the cluster control plane, watches for ClusterRole objects with an `aggregationRule` set. The `aggregationRule` defines a label selector that the controller uses to match other

ClusterRole objects that should be combined into the `rules` field of this one.

> **Caution:**
>
> The control plane overwrites any values that you manually specify in the `rules` field of an aggregate ClusterRole. If you want to change or add rules, do so in the `ClusterRole` objects that are selected by the `aggregationRule`.

Here is an example aggregated ClusterRole:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # The control plane automatically fills in the rules
```

If you create a new ClusterRole that matches the label selector of an existing aggregated ClusterRole, that change triggers adding the new rules into the aggregated ClusterRole. Here is an example that adds rules to the "monitoring" ClusterRole, by creating another ClusterRole labeled `rbac.example.com/aggregate-to-monitoring: true`.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# When you create the "monitoring-endpoints" ClusterRole,
```

```yaml
# the rules below will be added to the "monitoring" ClusterRole.
rules:
- apiGroups: [""]
  resources: ["services", "endpointslices", "pods"]
  verbs: ["get", "list", "watch"]
```

The default user-facing roles use ClusterRole aggregation. This lets you, as a cluster administrator, include rules for custom resources, such as those served by CustomResourceDefinitions or aggregated API servers, to extend the default roles.

For example: the following ClusterRoles let the "admin" and "edit" default roles manage the custom resource named CronTab, whereas the "view" role can perform only read actions on CronTab resources. You can assume that CronTab objects are named `"crontabs"` in URLs as seen by the API server.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: aggregate-cron-tabs-edit
  labels:
    # Add these permissions to the "admin" and "edit" default role
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
rules:
- apiGroups: ["stable.example.com"]
  resources: ["crontabs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "de
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true"
rules:
```

```yaml
- apiGroups: ["stable.example.com"]
  resources: ["crontabs"]
  verbs: ["get", "list", "watch"]
```

## Role examples

The following examples are excerpts from Role or ClusterRole objects, showing only the `rules` section.

Allow reading `"pods"` resources in the core API Group:

```yaml
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Pod
  # objects is "pods"
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

Allow reading/writing Deployments (at the HTTP level: objects with `"deployments"` in the resource part of their URL) in the `"apps"` API groups:

```yaml
rules:
- apiGroups: ["apps"]
  #
  # at the HTTP level, the name of the resource for accessing Depl
  # objects is "deployments"
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "de
```

Allow reading Pods in the core API group, as well as reading or writing Job resources in the `"batch"` API group:

```yaml
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Pod
  # objects is "pods"
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["batch"]
  #
  # at the HTTP level, the name of the resource for accessing Job
  # objects is "jobs"
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "de
```

Allow reading a ConfigMap named "my-config" (must be bound with a RoleBinding to limit to a single ConfigMap in a single namespace):

```yaml
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Conf
  # objects is "configmaps"
  resources: ["configmaps"]
  resourceNames: ["my-config"]
  verbs: ["get"]
```

Allow reading the resource `"nodes"` in the core group (because a Node is cluster-scoped, this must be in a ClusterRole bound with a ClusterRoleBinding to be effective):

```yaml
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Node
  # objects is "nodes"
```

```
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
```

Allow GET and POST requests to the non-resource endpoint `/healthz` and all subpaths (must be in a ClusterRole bound with a ClusterRoleBinding to be effective):

```
rules:
- nonResourceURLs: ["/healthz", "/healthz/*"] # '*' in a nonResour
    verbs: ["get", "post"]
```

## Referring to subjects

A RoleBinding or ClusterRoleBinding binds a role to subjects. Subjects can be groups, users or ServiceAccounts.

Kubernetes represents usernames as strings. These can be: plain names, such as "alice"; email-style names, like "bob@example.com"; or numeric user IDs represented as a string. It is up to you as a cluster administrator to configure the authentication modules so that authentication produces usernames in the format you want.

> **Caution:**
>
> The prefix `system:` is reserved for Kubernetes system use, so you should ensure that you don't have users or groups with names that start with `system:` by accident. Other than this special prefix, the RBAC authorization system does not require any format for usernames.

In Kubernetes, Authenticator modules provide group information. Groups, like users, are represented as strings, and that string has no format requirements, other than that the prefix `system:` is reserved.

ServiceAccounts have names prefixed with `system:serviceaccount:` , and belong to groups that have names prefixed with `system:serviceaccounts:` .

> **Note:**
> - `system:serviceaccount:` (singular) is the prefix for service account usernames.
> - `system:serviceaccounts:` (plural) is the prefix for service account groups.

## RoleBinding examples

The following examples are `RoleBinding` excerpts that only show the `subjects` section.

For a user named `alice@example.com` :

```
subjects:
- kind: User
  name: "alice@example.com"
  apiGroup: rbac.authorization.k8s.io
```

For a group named `frontend-admins` :

```
subjects:
- kind: Group
  name: "frontend-admins"
  apiGroup: rbac.authorization.k8s.io
```

For the default service account in the "kube-system" namespace:

```yaml
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

For all service accounts in the "qa" namespace:

```yaml
subjects:
- kind: Group
  name: system:serviceaccounts:qa
  apiGroup: rbac.authorization.k8s.io
```

For all service accounts in any namespace:

```yaml
subjects:
- kind: Group
  name: system:serviceaccounts
  apiGroup: rbac.authorization.k8s.io
```

For all authenticated users:

```yaml
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
```

For all unauthenticated users:

```yaml
subjects:
- kind: Group
```

```
    name: system:unauthenticated
    apiGroup: rbac.authorization.k8s.io
```

For all users:

```
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: system:unauthenticated
  apiGroup: rbac.authorization.k8s.io
```

# Default roles and role bindings

API servers create a set of default ClusterRole and ClusterRoleBinding objects. Many of these are `system:` prefixed, which indicates that the resource is directly managed by the cluster control plane. All of the default ClusterRoles and ClusterRoleBindings are labeled with `kubernetes.io/bootstrapping=rbac-defaults`.

> **Caution:**
>
> Take care when modifying ClusterRoles and ClusterRoleBindings with names that have a `system:` prefix. Modifications to these resources can result in non-functional clusters.

## Auto-reconciliation

At each start-up, the API server updates default cluster roles with any missing permissions, and updates default cluster role bindings with any missing subjects. This allows the cluster to repair accidental modifications, and helps to keep roles and role bindings up-to-date as permissions and subjects change in new Kubernetes releases.

To opt out of this reconciliation, set the `rbac.authorization.kubernetes.io/autoupdate` annotation on a default cluster role or default cluster RoleBinding to `false`. Be aware that missing default permissions and subjects can result in non-functional clusters.

Auto-reconciliation is enabled by default if the RBAC authorizer is active.

## API discovery roles

Default cluster role bindings authorize unauthenticated and authenticated users to read API information that is deemed safe to be publicly accessible (including CustomResourceDefinitions). To disable anonymous unauthenticated access, add `--anonymous-auth=false` flag to the API server configuration.

To view the configuration of these roles via `kubectl` run:

```
kubectl get clusterroles system:discovery -o yaml
```

> **Note:**
> If you edit that ClusterRole, your changes will be overwritten on API server restart via auto-reconciliation. To avoid that overwriting, either do not manually edit the role, or disable auto-reconciliation.

| Default ClusterRole | Default ClusterRoleBinding | Description |
| --- | --- | --- |
| **system:basic-user** | **system:authenticated** group | Allows a user read-o access to basic information about themselves. Prior to v1.14, this role was a bound to |

| | | system:unauthenti by default. |
|---|---|---|
| **system:discovery** | **system:authenticated** group | Allows read-only acc API discovery endpc needed to discover negotiate an API lev Prior to v1.14, this r was also bound to `system:unauthenti` by default. |
| **system:public-info-viewer** | **system:authenticated** and **system:unauthenticated** groups | Allows read-only acc non-sensitive inform about the cluster. Introduced in Kuber v1.14. |

Kubernetes RBAC API discovery roles

## User-facing roles

Some of the default ClusterRoles are not `system:` prefixed. These are intended to be user-facing roles. They include super-user roles ( `cluster-admin` ), roles intended to be granted cluster-wide using ClusterRoleBindings, and roles intended to be granted within particular namespaces using RoleBindings ( `admin` , `edit` , `view` ).

User-facing ClusterRoles use [ClusterRole aggregation](#) to allow admins to include rules for custom resources on these ClusterRoles. To add rules to the `admin` , `edit` , or `view` roles, create a ClusterRole with one or more of the following labels:

```
metadata:
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
```

```
rbac.authorization.k8s.io/aggregate-to-edit: "true"
rbac.authorization.k8s.io/aggregate-to-view: "true"
```

| Default ClusterRole | Default ClusterRoleBinding | Description |
|---|---|---|
| **cluster-admin** | **system:masters** group | Allows super-user access to perform any action on any resource. When used in a **ClusterRoleBinding**, it gives full control over every resource in the cluster and in all namespaces. When used in a **RoleBinding**, it gives full control over every resource in the role binding's namespace, including the namespace itself. |
| **admin** | None | Allows admin access, intended to be granted within a namespace using a **RoleBinding**. If used in a **RoleBinding**, allows read/write access to most resources in a namespace, including the ability to create roles and role bindings within the namespace. This role does not allow write access to resource quota or to the namespace itself. This role also does not allow write access to EndpointSlices |

(or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the "Write Access for EndpointSlices and Endpoints" section.

| | | |
|---|---|---|
| **edit** | None | Allows read/write access to most objects in a namespace.<br>This role does not allow viewing or modifying roles or role bindings. However, this role allows accessing Secrets and running Pods as any ServiceAccount in the namespace, so it can be used to gain the API access levels of any ServiceAccount in the namespace. This role also does not allow write access to EndpointSlices (or Endpoints) in clusters created using Kubernetes v1.22+. More information is available in the "Write Access for EndpointSlices and Endpoints" section. |
| **view** | None | Allows read-only access to see most objects in a namespace. It does not allow viewing roles or role bindings. |

This role does not allow viewing Secrets, since reading the contents of Secrets enables access to ServiceAccount credentials in the namespace, which would allow API access as any ServiceAccount in the namespace (a form of privilege escalation).

## Core component roles

| Default ClusterRole | Default ClusterRoleBinding | Description |
| --- | --- | --- |
| **system:kube-scheduler** | **system:kube-scheduler** user | Allows access to the resources required by the scheduler component. |
| **system:volume-scheduler** | **system:kube-scheduler** user | Allows access to the volume resources required by the kube-scheduler component. |
| **system:kube-controller-manager** | **system:kube-controller-manager** user | Allows access to the resources required by the controller manager component. The permissions required by individual controllers are detailed in the controller roles. |

| | | |
|---|---|---|
| **system:node** | None | Allows access to resources required by the kubelet, **including read access to all secrets, and write access to all pod status objects**. You should use the [Node authorizer] and [NodeRestriction admission plugin] instead of the `system:node` role, and allow granting API access to kubelets based on the Pods scheduled to run on them.<br><br>The `system:node` role only exists for compatibility with Kubernetes clusters upgraded from versions prior to v1.8. |
| **system:node-proxier** | **system:kube-proxy** user | Allows access to the resources required by the kube-proxy component. |

## Other component roles

| Default ClusterRole | Default ClusterRoleBinding | Description |
|---|---|---|

| system:auth-delegator | None | Allows delegated authentication and authorization checks. This is commonly used by add-on API servers for unified authentication and authorization. |
| --- | --- | --- |
| system:heapster | None | Role for the Heapster component (deprecated). |
| system:kube-aggregator | None | Role for the kube-aggregator component. |
| system:kube-dns | **kube-dns** service account in the **kube-system** namespace | Role for the kube-dns component. |
| system:kubelet-api-admin | None | Allows full access to the kubelet API. |
| system:node-bootstrapper | None | Allows access to the resources required to perform kubelet TLS bootstrapping. |
| system:node-problem-detector | None | Role for the node-problem-detector component. |

| | | |
|---|---|---|
| **system:persistent-volume-provisioner** | None | Allows access to the resources required by most dynamic voiume provisioners. |
| **system:monitoring** | **system:monitoring** group | Allows read access to control-plane monitoring endpoints (i.e. kube-apiserver liveness and readiness endpoints (`/healthz`, `/livez`, `/readyz`), the individual health-check endpoints (`/healthz/*`, `/livez/*`, `/readyz/*`), and `/metrics`). Note that individual health check endpoints and the metric endpoint may expose sensitive information. |

## Roles for built-in controllers

The Kubernetes controller manager runs controllers that are built in to the Kubernetes control plane. When invoked with `--use-service-account-credentials`, kube-controller-manager starts each controller using a separate service account. Corresponding roles exist for each built-in controller, prefixed with `system:controller:`. If the controller manager is not started with `--use-service-account-credentials`, it runs all control loops using its own credential, which must be granted all the relevant roles. These roles include:

- `system:controller:attachdetach-controller`
- `system:controller:certificate-controller`
- `system:controller:clusterrole-aggregation-controller`
- `system:controller:cronjob-controller`
- `system:controller:daemon-set-controller`
- `system:controller:deployment-controller`
- `system:controller:disruption-controller`
- `system:controller:endpoint-controller`
- `system:controller:expand-controller`
- `system:controller:generic-garbage-collector`
- `system:controller:horizontal-pod-autoscaler`
- `system:controller:job-controller`
- `system:controller:namespace-controller`
- `system:controller:node-controller`
- `system:controller:persistent-volume-binder`
- `system:controller:pod-garbage-collector`
- `system:controller:pv-protection-controller`
- `system:controller:pvc-protection-controller`
- `system:controller:replicaset-controller`
- `system:controller:replication-controller`
- `system:controller:resourcequota-controller`
- `system:controller:root-ca-cert-publisher`
- `system:controller:route-controller`
- `system:controller:service-account-controller`
- `system:controller:service-controller`
- `system:controller:statefulset-controller`
- `system:controller:ttl-controller`

# Privilege escalation prevention and bootstrapping

The RBAC API prevents users from escalating privileges by editing roles or role bindings. Because this is enforced at the API level, it applies even when the RBAC authorizer is not in use.

## Restrictions on role creation or update

You can only create/update a role if at least one of the following things is true:

1. You already have all the permissions contained in the role, at the same scope as the object being modified (cluster-wide for a ClusterRole, within the same namespace or cluster-wide for a Role).
2. You are granted explicit permission to perform the `escalate` verb on the `roles` or `clusterroles` resource in the `rbac.authorization.k8s.io` API group.

For example, if `user-1` does not have the ability to list Secrets cluster-wide, they cannot create a ClusterRole containing that permission. To allow a user to create/update roles:

1. Grant them a role that allows them to create/update Role or ClusterRole objects, as desired.
2. Grant them permission to include specific permissions in the roles they create/update:
   - implicitly, by giving them those permissions (if they attempt to create or modify a Role or ClusterRole with permissions they themselves have not been granted, the API request will be forbidden)
   - or explicitly allow specifying any permission in a `Role` or `ClusterRole` by giving them permission to perform the `escalate` verb on `roles` or `clusterroles` resources in the `rbac.authorization.k8s.io` API group

## Restrictions on role binding creation or update

You can only create/update a role binding if you already have all the permissions contained in the referenced role (at the same scope as the role binding) *or* if you have been authorized to perform the `bind` verb on the referenced role. For example, if `user-1` does not have the ability to list Secrets cluster-wide, they cannot create a ClusterRoleBinding to a role that grants that permission. To allow a user to create/update role bindings:

1. Grant them a role that allows them to create/update RoleBinding or ClusterRoleBinding objects, as desired.
2. Grant them permissions needed to bind a particular role:
   - implicitly, by giving them the permissions contained in the role.
   - explicitly, by giving them permission to perform the `bind` verb on the particular Role (or ClusterRole).

For example, this ClusterRole and RoleBinding would allow `user-1` to grant other users the `admin`, `edit`, and `view` roles in the namespace `user-1-namespace`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: role-grantor
rules:
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["rolebindings"]
  verbs: ["create"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["clusterroles"]
  verbs: ["bind"]
  # omit resourceNames to allow binding any ClusterRole
  resourceNames: ["admin","edit","view"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-grantor-binding
```

```yaml
  namespace: user-1-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: role-grantor
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user-1
```

When bootstrapping the first roles and role bindings, it is necessary for the initial user to grant permissions they do not yet have. To bootstrap initial roles and role bindings:

- Use a credential with the "system:masters" group, which is bound to the "cluster-admin" super-user role by the default bindings.

# Command-line utilities

## kubectl create role

Creates a Role object defining permissions within a single namespace. Examples:

- Create a Role named "pod-reader" that allows users to perform `get` , `watch` and `list` on pods:

  ```
  kubectl create role pod-reader --verb=get --verb=list --verb=
  ```

- Create a Role named "pod-reader" with resourceNames specified:

  ```
  kubectl create role pod-reader --verb=get --resource=pods --r
  ```

- Create a Role named "foo" with apiGroups specified:

  ```
  kubectl create role foo --verb=get,list,watch --resource=repl
  ```

- Create a Role named "foo" with subresource permissions:

  ```
  kubectl create role foo --verb=get,list,watch --resource=pods
  ```

- Create a Role named "my-component-lease-holder" with permissions to get/update a resource with a specific name:

  ```
  kubectl create role my-component-lease-holder --verb=get,list
  ```

## kubectl create clusterrole

Creates a ClusterRole. Examples:

- Create a ClusterRole named "pod-reader" that allows user to perform `get` , `watch` and `list` on pods:

  ```
  kubectl create clusterrole pod-reader --verb=get,list,watch -
  ```

- Create a ClusterRole named "pod-reader" with resourceNames specified:

  ```
  kubectl create clusterrole pod-reader --verb=get --resource=p
  ```

- Create a ClusterRole named "foo" with apiGroups specified:

```
kubectl create clusterrole foo --verb=get,list,watch --resour
```

- Create a ClusterRole named "foo" with subresource permissions:

```
kubectl create clusterrole foo --verb=get,list,watch --resour
```

- Create a ClusterRole named "foo" with nonResourceURL specified:

```
kubectl create clusterrole "foo" --verb=get --non-resource-ur
```

- Create a ClusterRole named "monitoring" with an aggregationRule specified:

```
kubectl create clusterrole monitoring --aggregation-rule="rba
```

## kubectl create rolebinding

Grants a Role or ClusterRole within a specific namespace. Examples:

- Within the namespace "acme", grant the permissions in the "admin" ClusterRole to a user named "bob":

```
kubectl create rolebinding bob-admin-binding --clusterrole=ad
```

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to the service account in the namespace "acme" named "myapp":

```
kubectl create rolebinding myapp-view-binding --clusterrole=v
```

- Within the namespace "acme", grant the permissions in the "view" ClusterRole to a service account in the namespace "myappnamespace" named "myapp":

```
kubectl create rolebinding myappnamespace-myapp-view-binding
```

## kubectl create clusterrolebinding

Grants a ClusterRole across the entire cluster (all namespaces). Examples:

- Across the entire cluster, grant the permissions in the "cluster-admin" ClusterRole to a user named "root":

```
kubectl create clusterrolebinding root-cluster-admin-binding
```

- Across the entire cluster, grant the permissions in the "system:node-proxier" ClusterRole to a user named "system:kube-proxy":

```
kubectl create clusterrolebinding kube-proxy-binding --cluste
```

- Across the entire cluster, grant the permissions in the "view" ClusterRole to a service account named "myapp" in the namespace "acme":

```
kubectl create clusterrolebinding myapp-view-binding --cluste
```

## kubectl auth reconcile

Creates or updates `rbac.authorization.k8s.io/v1` API objects from a manifest file.

Missing objects are created, and the containing namespace is created for namespaced objects, if required.

Existing roles are updated to include the permissions in the input objects, and remove extra permissions if `--remove-extra-permissions` is specified.

Existing bindings are updated to include the subjects in the input objects, and remove extra subjects if `--remove-extra-subjects` is specified.

Examples:

- Test applying a manifest file of RBAC objects, displaying changes that would be made:

```
kubectl auth reconcile -f my-rbac-rules.yaml --dry-run=client
```

- Apply a manifest file of RBAC objects, preserving any extra permissions (in roles) and any extra subjects (in bindings):

```
kubectl auth reconcile -f my-rbac-rules.yaml
```

- Apply a manifest file of RBAC objects, removing any extra permissions (in roles) and any extra subjects (in bindings):

```
kubectl auth reconcile -f my-rbac-rules.yaml --remove-extra-s
```

# ServiceAccount permissions

Default RBAC policies grant scoped permissions to control-plane components, nodes, and controllers, but grant *no permissions* to service accounts outside the `kube-system` namespace (beyond the permissions given by API discovery roles).

This allows you to grant particular roles to particular ServiceAccounts as needed. Fine-grained role bindings provide greater security, but require more effort to administrate. Broader grants can give unnecessary (and potentially escalating) API access to ServiceAccounts, but are easier to administrate.

In order from most secure to least secure, the approaches are:

1. Grant a role to an application-specific service account (best practice)

   This requires the application to specify a `serviceAccountName` in its pod spec, and for the service account to be created (via the API, application manifest, `kubectl create serviceaccount`, etc.).

   For example, grant read-only permission within "my-namespace" to the "my-sa" service account:

   ```
   kubectl create rolebinding my-sa-view \
      --clusterrole=view \
      --serviceaccount=my-namespace:my-sa \
      --namespace=my-namespace
   ```

2. Grant a role to the "default" service account in a namespace

   If an application does not specify a `serviceAccountName`, it uses the "default" service account.

   > **Note:**
   > Permissions given to the "default" service account are available to any pod in the namespace that does not specify a `serviceAccountName`.

For example, grant read-only permission within "my-namespace" to the "default" service account:

```
kubectl create rolebinding default-view \
   --clusterrole=view \
   --serviceaccount=my-namespace:default \
   --namespace=my-namespace
```

Many add-ons run as the "default" service account in the `kube-system` namespace. To allow those add-ons to run with super-user access, grant cluster-admin permissions to the "default" service account in the `kube-system` namespace.

> **Caution:**
>
> Enabling this means the `kube-system` namespace contains Secrets that grant super-user access to your cluster's API.

```
kubectl create clusterrolebinding add-on-cluster-admin \
   --clusterrole=cluster-admin \
   --serviceaccount=kube-system:default
```

3. Grant a role to all service accounts in a namespace

   If you want all applications in a namespace to have a role, no matter what service account they use, you can grant a role to the service account group for that namespace.

   For example, grant read-only permission within "my-namespace" to all service accounts in that namespace:

```
kubectl create rolebinding serviceaccounts-view \
   --clusterrole=view \
   --group=system:serviceaccounts:my-namespace \
   --namespace=my-namespace
```

4. Grant a limited role to all service accounts cluster-wide (discouraged)

   If you don't want to manage permissions per-namespace, you can grant a cluster-wide role to all service accounts.

   For example, grant read-only permission across all namespaces to all service accounts in the cluster:

   ```
   kubectl create clusterrolebinding serviceaccounts-view \
     --clusterrole=view \
    --group=system:serviceaccounts
   ```

5. Grant super-user access to all service accounts cluster-wide (strongly discouraged)

   If you don't care about partitioning permissions at all, you can grant super-user access to all service accounts.

   > **Warning:**
   >
   > This allows any application full access to your cluster, and also grants any user with read access to Secrets (or the ability to create any pod) full access to your cluster.

   ```
   kubectl create clusterrolebinding serviceaccounts-cluster-adm
     --clusterrole=cluster-admin \
     --group=system:serviceaccounts
   ```

# Write access for EndpointSlices and Endpoints

Kubernetes clusters created before Kubernetes v1.22 include write access to EndpointSlices (and Endpoints) in the aggregated "edit" and "admin" roles. As a mitigation for CVE-2021-25740, this access is not part of the aggregated roles in clusters that you create using Kubernetes v1.22 or later.

Existing clusters that have been upgraded to Kubernetes v1.22 will not be subject to this change. The CVE announcement includes guidance for restricting this access in existing clusters.

If you want new clusters to retain this level of access in the aggregated roles, you can create the following ClusterRole:

access/endpoints-aggregated.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    kubernetes.io/description: |-
      Add endpoints write permissions to the edit and admin roles.
      removed by default in 1.22 because of CVE-2021-25740. See
      https://issue.k8s.io/103675. This can allow writers to direc
      or Ingress implementations to expose backend IPs that would
      be accessible, and can circumvent network policies or securi
      intended to prevent/isolate access to those backends.
      EndpointSlices were never included in the edit or admin role
      is nothing to restore for the EndpointSlice API.
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
  name: custom:aggregate-to-edit:endpoints # you can change this i
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["create", "delete", "deletecollection", "patch", "upda
```

# Upgrading from ABAC

Clusters that originally ran older Kubernetes versions often used permissive ABAC policies, including granting full API access to all service accounts.

Default RBAC policies grant scoped permissions to control-plane components, nodes, and controllers, but grant *no permissions* to service accounts outside the `kube-system` namespace (beyond the permissions given by API discovery roles).

While far more secure, this can be disruptive to existing workloads expecting to automatically receive API permissions. Here are two approaches for managing this transition:

## Parallel authorizers

Run both the RBAC and ABAC authorizers, and specify a policy file that contains the legacy ABAC policy:

```
--authorization-mode=...,RBAC,ABAC --authorization-policy-file=myp
```

To explain that first command line option in detail: if earlier authorizers, such as Node, deny a request, then the RBAC authorizer attempts to authorize the API request. If RBAC also denies that API request, the ABAC authorizer is then run. This means that any request allowed by *either* the RBAC or ABAC policies is allowed.

When the kube-apiserver is run with a log level of 5 or higher for the RBAC component ( `--vmodule=rbac*=5` or `--v=5` ), you can see RBAC denials in the API server log (prefixed with `RBAC` ). You can use that information to determine which roles need to be granted to which users, groups, or service accounts.

Once you have granted roles to service accounts and workloads are running with no RBAC denial messages in the server logs, you can

remove the ABAC authorizer.

## Permissive RBAC permissions

You can replicate a permissive ABAC policy using RBAC role bindings.

> **Warning:**
>
> The following policy allows **ALL** service accounts to act as cluster administrators. Any application running in a container receives service account credentials automatically, and could perform any action against the API, including viewing secrets and modifying permissions. This is not a recommended policy.
>
> ```
> kubectl create clusterrolebinding permissive-binding \
>    --clusterrole=cluster-admin \
>    --user=admin \
>    --user=kubelet \
>    --group=system:serviceaccounts
> ```

After you have transitioned to use RBAC, you should adjust the access controls for your cluster to ensure that these meet your information security needs.
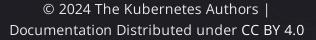
## Feedback

Was this page helpful?

Yes     No

kubernetes

Documentation

Last modified June 28, 2024 at 3:48 PM PST: Update
content/en/docs/reference/access-authn-authz/rbac.md (a7fe6cf8c5)