



PUBLISHED · OCTOBER 2, 2026

# Kubernetes RBAC: ServiceAccounts, Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings

Build a practical Kubernetes RBAC mental model, give workloads distinct ServiceAccount identities, grant least-privilege access, and verify authorization with kubectl.



This is Learn post 22 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier lessons followed requests through the API server and created workloads, configuration, and clusters. Now we control which authenticated identities may perform which API actions.

Role-based access control (RBAC) is Kubernetes' built-in authorization model.

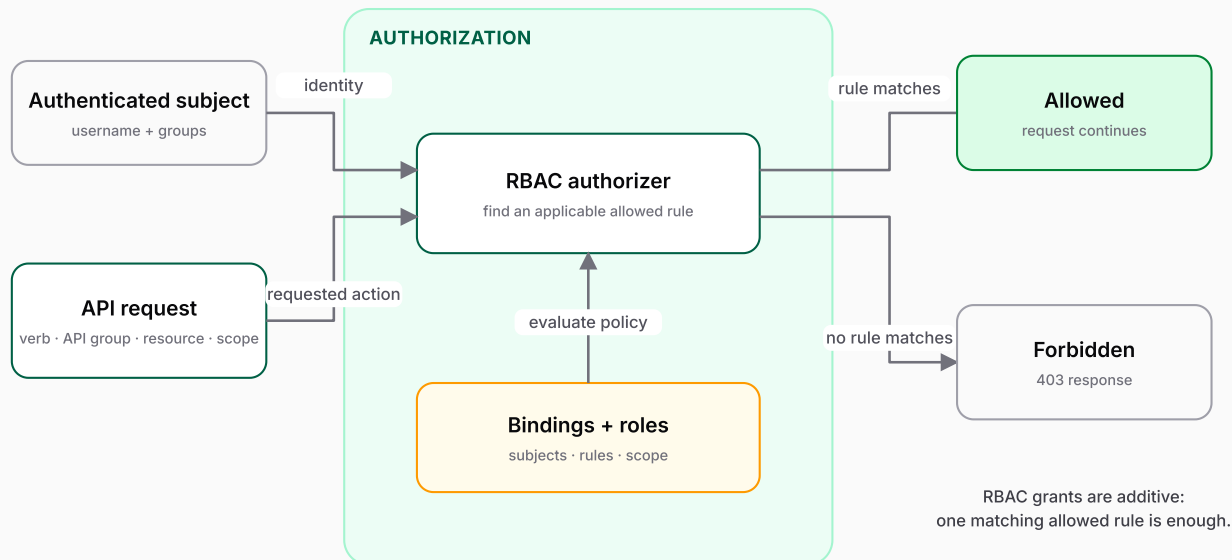
Authentication answers who sent a request; RBAC authorization decides whether that identity may carry out the requested verb on the requested resource. A clear identity with no matching grant is denied.

## What you'll learn

- Read an authorization decision as subject, action, resource, and scope.
- Give a Pod a distinct ServiceAccount identity and understand its projected credentials.
- Choose correctly among Role, ClusterRole, RoleBinding, and ClusterRoleBinding.
- Write focused RBAC rules for API groups, resources, subresources, and verbs.
- Generate, apply, inspect, and verify permissions with kubectl.

## The mental model: identity, permission, attachment

A ServiceAccount supplies a workload identity. A Role or ClusterRole describes allowed actions. A RoleBinding or ClusterRoleBinding attaches those permissions to identities. The binding's scope determines where the grant applies.



### How RBAC turns an authenticated API request into an authorization decision

Identity and requested action are evaluated together. RBAC permits the request when any applicable binding leads to a matching allowed rule.

RBAC rules are additive. A request is allowed when at least one applicable rule matches; Kubernetes RBAC has no deny rule that cancels another grant. Because access accumulates across bindings, troubleshoot the subject's complete set of grants rather than inspecting only one Role.

## ServiceAccounts give workloads an API identity

Kubernetes represents workload identities with namespaced ServiceAccount objects. A ServiceAccount named report-reader in namespace cka-rbac authenticates as system:serviceaccount:cka-rbac:report-reader and belongs to groups that include system:serviceaccounts and system:serviceaccounts:cka-rbac.

Every namespace receives a ServiceAccount named default. A Pod that omits spec.serviceAccountName uses it, but the default account does not receive useful application permissions automatically. Giving broad access to default silently gives that

access to every Pod in the namespace that has not selected another account, so application-specific accounts are the safer pattern.

```
bash
```

```
kubectl create namespace cka-rbac
```

```
kubectl create serviceaccount report-reader \
--namespace=cka-rbac \
--dry-run=client -o yaml
```

The dry run shows the small declarative object without changing the cluster. Save the output when it belongs in version control, or remove the dry-run flags to create it directly.

```
serviceaccount-and-pod.yaml · yaml
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: report-reader
  namespace: cka-rbac
---
apiVersion: v1
kind: Pod
metadata:
  name: report-client
  namespace: cka-rbac
spec:
  serviceAccountName: report-reader
  containers:
    - name: client
      image: busybox:1.36.1
      command: ["sh", "-c"]
      args: ["sleep 3600"]
```

serviceAccountName is fixed when the Pod is created. When token automounting is enabled, the control plane issues a short-lived, audience-bound token and kubelet projects it into the Pod with the namespace and cluster CA certificate. The token proves identity; it does not itself define permissions.

bash

```
kubectl apply -f serviceaccount-and-pod.yaml
kubectl wait -n cka-rbac --for=condition=Ready pod/report-client --timeout=90s

kubectl get pod report-client -n cka-rbac \
  -o jsonpath='{.spec.serviceAccountName}'

kubectl exec -n cka-rbac report-client -- \
  ls /var/run/secrets/kubernetes.io/serviceaccount
```

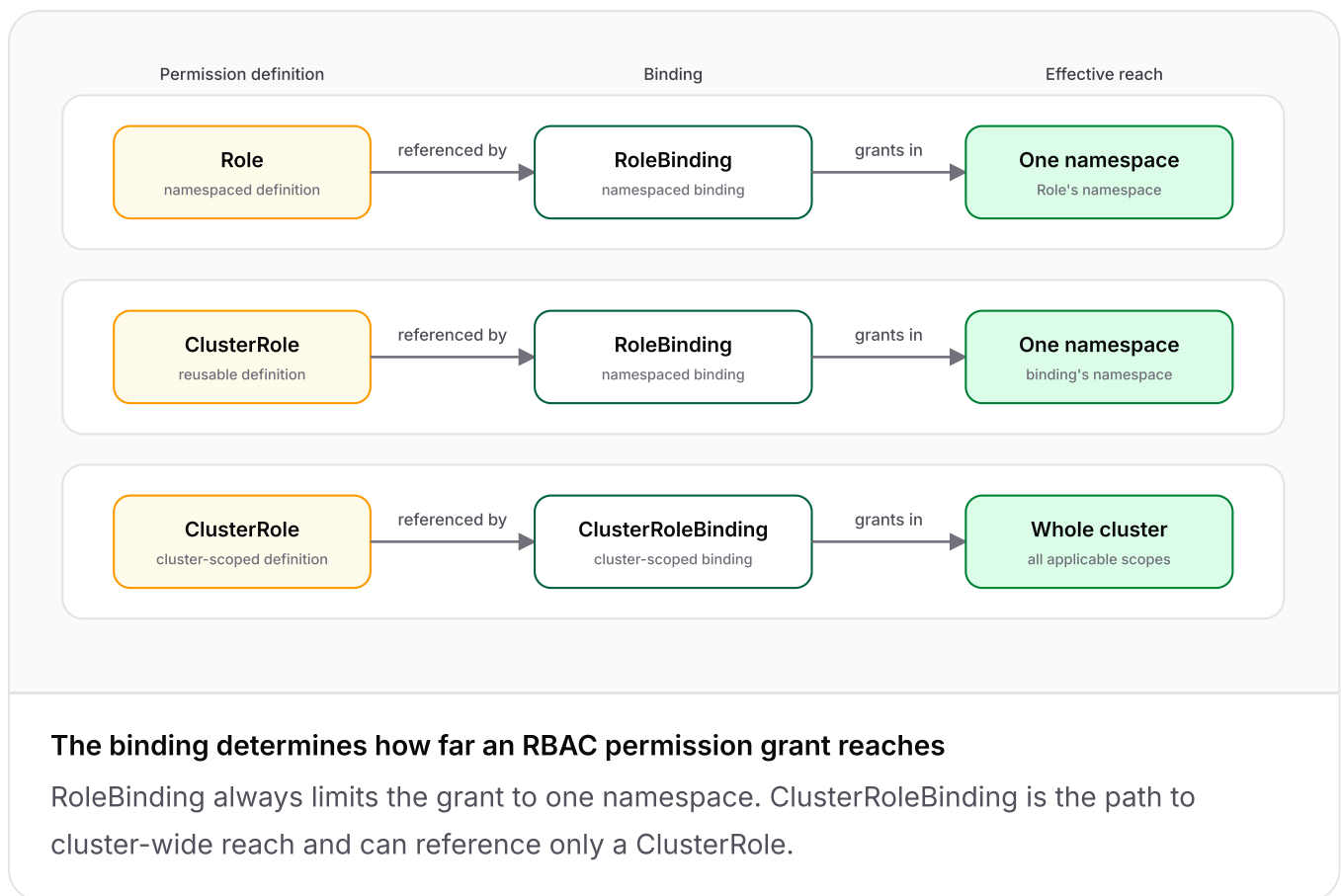
plaintext

```
report-reader
ca.crt
namespace
token
```

Do not create a long-lived token Secret merely because a Pod needs API access.

Modern Pods normally receive rotating projected tokens. If a workload never calls the Kubernetes API, set `automountServiceAccountToken: false` on the Pod or ServiceAccount to avoid supplying an unnecessary credential.

# Roles describe permissions; bindings grant them



## Role: permissions inside one namespace

A Role is namespaced, and its rules apply only to namespaced resources in that Role's namespace. Use it when the permission definition itself belongs to one namespace.

pod-reader-role.yaml · yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: cka-rbac
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get"]
```

The empty `apiGroups` value selects the core API group, which contains Pods. Resource names are lowercase plural API resource names, not Kind names. `get` reads one object, `list` reads a collection, and `watch` follows collection changes. Pod logs are exposed as the separate  `pods/log` subresource, so reading Pods does not automatically authorize reading their logs.

## ClusterRole: cluster-level or reusable permissions

A `ClusterRole` is not namespaced. Its rules can cover cluster-scoped resources such as Nodes, namespaced resources such as Pods, or non-resource URLs. For namespaced resources, the binding decides whether the `ClusterRole` applies in one namespace or across the cluster.

node-reader-clusterrole.yaml · yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "list", "watch"]
```

A `Role` cannot grant access to Nodes because Nodes are cluster-scoped. Confirm resource scope and API group instead of guessing:

bash

```
kubectl api-resources --namespaced=true
kubectl api-resources --namespaced=false
kubectl api-resources --api-group=apps
```

For example, Deployments belong to the `apps` API group, so a rule for them uses `apiGroups: ["apps"]` and `resources: ["deployments"]`. API discovery is the reliable bridge from the object you know to the exact RBAC rule fields.

## RoleBinding: grant access in one namespace

A RoleBinding names its subjects and one roleRef. It may reference a Role in the same namespace or a ClusterRole. Either way, the resulting permission is limited to the RoleBinding's namespace.

pod-reader-binding.yaml · yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: report-reader-can-read-pods
  namespace: cka-rbac
subjects:
- kind: ServiceAccount
  name: report-reader
  namespace: cka-rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-reader
```

The binding's namespace controls where the grant applies; the ServiceAccount subject's namespace controls which identity receives it. These namespaces can differ. That is how a ServiceAccount from one namespace can be granted access to resources in another namespace without moving either object.

## ClusterRoleBinding: grant access across the cluster

A ClusterRoleBinding is cluster-scoped, references a ClusterRole, and grants its permissions at cluster scope. If that ClusterRole includes Pods, the subject receives those Pod permissions in every namespace. Because the blast radius is larger, choose this binding only when the identity truly needs cluster-wide access.

node-reader-binding.yaml · yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-auditors-can-read-nodes
subjects:
- kind: Group
  name: cluster-auditors
  apiGroup: rbac.authorization.k8s.io
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: node-reader
```

Users and groups come from the cluster's authentication system; Kubernetes does not normally store them as User or Group API objects. ServiceAccounts are Kubernetes objects. RBAC bindings can name all three subject kinds because authorization operates on the authenticated username and groups attached to the request.

## Worked demonstration: least-privilege Pod observation

We will grant report-reader only the Pod read operations defined above. Checking before and after the binding makes the causal relationship visible: the identity stays the same; only the attachment changes.

### 1. Observe the initial denial

bash

```
kubectl auth can-i list pods \
--namespace=cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader

kubectl auth can-i delete pods \
--namespace=cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader
```

```
plaintext
```

```
no  
no
```

kubectl auth can-i asks the API server's authorization layer whether an action is allowed. The --as flag impersonates the exact ServiceAccount username; your current administrator identity must itself be allowed to impersonate that subject. The initial no confirms that merely creating a ServiceAccount grants no application access.

## 2. Apply the permission and its attachment

```
bash
```

```
kubectl apply -f pod-reader-role.yaml  
kubectl apply -f pod-reader-binding.yaml  
  
kubectl get role,rolebinding -n cka-rbac  
kubectl describe role pod-reader -n cka-rbac  
kubectl describe rolebinding report-reader-can-read-pods -n cka-rbac
```

Describe the Role to read its verbs, resources, and API groups. Describe the RoleBinding to follow the subject-to-roleRef connection. When access is surprising, inspect both sides of that connection.

### 3. Verify the exact boundary

bash

```
kubectl auth can-i list pods \
-n cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader

kubectl auth can-i get pods --subresource=log \
-n cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader

kubectl auth can-i delete pods \
-n cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader

kubectl auth can-i list pods \
-n default \
--as=system:serviceaccount:cka-rbac:report-reader
```

plaintext

```
yes
yes
no
no
```

The two yes answers match the Role. Deleting was never granted, and the RoleBinding does not cross the cka-rbac namespace boundary. Negative checks matter because a working operation alone does not prove that the grant is narrow.

To view the effective permissions that the API server reports for this subject in the namespace, use:

bash

```
kubectl auth can-i --list \
-n cka-rbac \
--as=system:serviceaccount:cka-rbac:report-reader
```

This is useful for orientation, but test the precise verb, resource, subresource, name, and namespace when validating a specific operation.

## Reuse a ClusterRole without granting cluster-wide access

A common pattern is to define a reusable ClusterRole once, then attach it with separate RoleBindings wherever it is needed. The definition is cluster-scoped, but each grant remains namespaced.

reusable-workload-view.yaml · yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: workload-viewer
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: report-reader-can-view-workloads
  namespace: cka-rbac
subjects:
  - kind: ServiceAccount
    name: report-reader
    namespace: cka-rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: workload-viewer
```

The same ClusterRole can be referenced by another RoleBinding in another namespace. Changing the ClusterRole changes permissions for every binding that references it, so reuse reduces duplication but increases the impact of edits.

# Generate RBAC YAML quickly, then read it

Imperative generators are fast and reduce typing errors. Emit YAML first when the object should be reviewed or stored declaratively.

bash

```
kubectl create role pod-reader \
  --verb=get,list,watch \
  --resource=pods \
  --namespace=cka-rbac \
  --dry-run=client -o yaml

kubectl create rolebinding report-reader-can-read-pods \
  --role=pod-reader \
  --serviceaccount=cka-rbac:report-reader \
  --namespace=cka-rbac \
  --dry-run=client -o yaml

kubectl create clusterrole node-reader \
  --verb=get,list,watch \
  --resource=nodes \
  --dry-run=client -o yaml
```

The generator produces one rule from the supplied flags. Edit declarative YAML when you need multiple rule groups, subresources, resourceNames, labels, or reviewable descriptions of intent.

## Rule details that prevent common mistakes

### Verbs are API operations, not kubectl command names

Typical resource verbs are get, list, watch, create, update, patch, and delete. kubectl get pod/name needs get; kubectl get pods needs list; kubectl delete needs delete. Watching normally begins by listing and then opening a watch, so controllers commonly need both list and watch.

## Subresources need explicit rules

Operations such as Pod logs and Deployment scale use subresources. Write them with a slash in rules, such as pods/log or deployments/scale, and verify them with `kubectl auth can-i` using `--subresource` where supported.

## resourceNames narrows named-object operations

yaml

```
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["report-settings"]
  verbs: ["get", "update", "patch"]
```

This rule addresses one existing ConfigMap by name. `resourceNames` cannot use patterns, and it cannot generally restrict create because a create authorization request does not yet identify an existing object by name. List and watch also need special request field selectors to benefit from `resourceNames`, so do not assume that a named get grant implies collection access.

## roleRef is intentionally immutable

You may update a binding's subjects, but not its `roleRef`. Replacing the referenced role could silently give every existing subject a different permission set. Delete and recreate the binding, or use `kubectl auth reconcile` for a managed RBAC manifest, when the `roleRef` must change.

## Built-in roles are useful building blocks

Kubernetes supplies ClusterRoles such as view, edit, admin, and cluster-admin. A RoleBinding to view grants its namespaced read permissions only in that binding's namespace. A ClusterRoleBinding to cluster-admin grants superuser access cluster-

wide. The same role definition can therefore have radically different reach depending on the binding.

```
bash
```

```
kubectl create rolebinding application-viewers \
  --clusterrole=view \
  --group=application-viewers \
  --namespace=cka-rbac \
  --dry-run=client -o yaml
```

```
kubectl get clusterrole view -o yaml
```

Inspect built-in roles rather than assuming what they include. In particular, `view` intentionally avoids reading Secrets, while `edit` can create workloads and access Secrets; these capabilities may enable acting as other ServiceAccounts in the namespace. Least privilege is about what an identity can ultimately cause, not only the friendly name of a role.

Avoid editing system:-prefixed ClusterRoles and bindings casually. The API server maintains many bootstrapping roles, and changes can damage cluster components or be repaired by automatic reconciliation.

## A repeatable administrator workflow

1. Identify the subject exactly: user, group, or namespace-qualified ServiceAccount.
2. Identify the API action: verb, API group, resource or subresource, optional resource name, and namespace.
3. Use `kubectl api-resources` to confirm whether the resource is namespaced and which API group owns it.
4. Choose Role for a local definition or ClusterRole for cluster-scoped and reusable definitions.
5. Choose RoleBinding for one namespace or ClusterRoleBinding for cluster-wide reach.
6. Apply the smallest rule, then verify allowed and intentionally disallowed operations with `kubectl auth can-i`.

When an application reports Forbidden, read the error for its authenticated username, verb, resource, and namespace. Compare those exact values with the binding subject and rule. A 401 Unauthorized points to missing or invalid authentication; a 403 Forbidden means the API server recognized the caller but authorization did not allow the request.

## Clean up the demonstration

```
bash
```

```
kubectl delete namespace cka-rbac  
kubectl delete clusterrole node-reader workload-viewer --ignore-not-found  
kubectl delete clusterrolebinding cluster-auditors-can-read-nodes --ignore-not-found
```

Deleting the namespace removes its ServiceAccount, Pod, Roles, and RoleBindings. ClusterRoles and ClusterRoleBindings are outside that namespace and must be removed separately if you created them.

## Distinctions worth remembering

- Authentication establishes identity; RBAC authorization evaluates what that identity may do.
- A ServiceAccount is an identity, not a permission set; its grants come from bindings.
- Role is namespaced; ClusterRole is cluster-scoped and can also serve as a reusable definition for namespaced access.
- RoleBinding grants within one namespace; ClusterRoleBinding grants across the cluster.
- A RoleBinding may reference a Role or ClusterRole; a ClusterRoleBinding references a ClusterRole only.
- RBAC grants accumulate and do not contain explicit deny rules.
- Verify both the operation that should work and a nearby operation that should remain forbidden.

# Where this fits in the series

This lesson completes the core authorization model needed to understand how administrators and workloads receive API access. Post 23 next uses Helm to install and manage Kubernetes components; some charts create the ServiceAccounts and RBAC objects you can now inspect instead of treating them as boilerplate. Guided diagnosis of broken RBAC returns in Practice post 47.

## Official documentation

[Using RBAC Authorization](#) defines the four RBAC objects, rule semantics, built-in roles, and privilege-escalation protections.

[Service Accounts](#) explains workload identities, default accounts, projected tokens, and token automounting.

[kubectl auth can-i](#) documents precise authorization checks, impersonation, namespace selection, and subresource queries.

[Kubernetes Authorization](#) places RBAC in the API request path and describes authorization review APIs.