



Role-Based Access Control (RBAC)

Kubernetes - Beginners | Intermediate | Advanced

[View on GitHub](#) [Join Slack](#)

Role-Based Access Control (RBAC)

RBAC is a security design that restricts access to valuable resources based on the role the user holds, hence the name role-based. To understand the importance and the need of having RBAC policies in place, let's consider a system that doesn't use it. Let's say that you have an HR management solution, but the only security access measure used is that users must authenticate themselves through a username and a password. Having provided their credentials, users gain full access to every module in the system (recruitment, training, staff performance, salaries, etc.). A slightly more secure system will differentiate between regular user access and "admin" access, with the latter providing potentially destructive privileges. For example, ordinary users cannot delete a module from the system, whereas an administrator can. But still, users without admin access can read and modify the module's data regardless of whether their current job entails doing this.

If you worked as a Linux administrator for any length of time, you appreciate the importance of having a security system that implements a security matrix of access and authority. In the old days of Linux and UNIX, you could either be a "normal" user with minimal access to the system resources, or you can have "root" access. Root access virtually gives you full control over the machine that you can accidentally bring the whole

system down. Needless to say that if an intruder could gain access to this root account, your entire system is at high risk. Accordingly, RBAC systems were introduced.

In a system that uses RBAC, there is minimal mention of the “superuser” or the administrator who has access to everything. Instead, there’s more reference to the access level, the role, and the privilege. Even administrators can be categorized based on their job requirements. So, backup administrators should have full access to the tools that they use to do backups. But they shouldn’t be able to stop the webserver or change the system’s date and time.

Creating a Kubernetes User Account Using X509 Client Certificate

Pre-requisite

- Open <https://labs.play-with-k8s.com/>
- Follow <https://collabnix.github.io/kubelabs/kube101.html> to create 3 Node K8s Cluster

Creating Client Certificates

To create a client certificate in PWK you need to have openssl tool installed. To do that run the following command

```
[node1 ~]$ yum install openssl
```

Let’s create a user account for our labs. Kubernetes supports several user authentication methods. It also supports combining more than one to authenticate a user. If one of the chained methods fail, the user is not verified. In this example, we’ll use only one authentication method, the X509 certificate to create a user account called div.

First, we need to create the client key:

```
[node1 ~]$ openssl genrsa -out div.key 2048
```

```
Generating RSA private key, 2048 bit long modulus
.....+++
.....
e is 65537 (0x10001)
```

Then, we need to create a certificate signing request:

```
[node1 ~]$ openssl req -new -key div.key -out div.csr -subj "/CN=div"
```

Next we need to copy the certificate and key that exists in kubernetes (you can use your own CA, but for this lab we are restricting ourselves to pre-existing certificate and key for kubernetes cluster). Once you copy the certificate and the key to the root folder you should have displayed files in your root folder

```
[node1 ~]$ cd /etc/kubernetes/pki/
[node1 pki]$ cp ca.crt ca.key /root/
[node1 pki]$ cd $home
[node1 ~]$ ls
anaconda-ks.cfg  ca.crt  ca.key  div.csr  div.key
```

Now lets sign the user key and signing request with cluster certificate and key.

```
[node1 ~]$ openssl x509 -req -in div.csr -CA ca.crt -CAkey ca.key -CAcreates
Signature ok
subject=/CN=div
Getting CA Private Key
```

Adding user's credentials to our kubeconfig file

```
[node1 ~]$ kubectl config set-credentials div --client-certificate=div.crt --
User "div" set.
```

Testing permissions for the user

```
[node1 ~]$ kubectl run apache --image=apache
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed
[node1 ~]$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
apache-64df49d484-2lfbb            0/1     Pending   0           6s
```

```
[node1 ~]$ kubectl --user=div get pods
Error from server (Forbidden): pods is forbidden: User "div" cannot list res
```

Creating Role and Role Binding

A Role in Kubernetes is as a Group in other RBAC implementations. Instead of defining different authorization rules for each user, you attach those rules to a group and add users to it. When users resign, for example, you only need to remove them from one place. Similarly, when a new user joins the company or gets transferred to another department, you need to change the roles they're associated with.

Let's create a role that enables our user to execute the get pods command:

```
[node1 ~]$ cat > role.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: get-pods
rules:
- apiGroups: ["*"]
  resources: ["pods"]
  verbs: ["list"]
^C
[node1 ~]$ kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/get-pods created
```

Now, we have a role that enables its users to list the pods on the default namespace. But, in order for the div user to be able to execute the get pods, it needs to get bound to this role.

Kubernetes offers the RoleBinding resource to link roles with their objects (for example, users). Let's add role-binding.yaml file to look as follows:

```
[node1 ~]$ cat > role-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: div-get-pods
subjects:
- kind: User
  name: div
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: get-pods
  apiGroup: rbac.authorization.k8s.io
^C
[node1 ~]$ kubectl apply -f role-binding.yaml
rolebinding.rbac.authorization.k8s.io/div-get-pods created
```

Now let's see if div can list pods on the cluster

```
[node1 ~]$ kubectl --user=div get pods
NAME                                READY   STATUS    RESTARTS   AGE
my-nginx-6dd86d77d-4t879           1/1     Running   0           28m
my-nginx-6dd86d77d-l6dzk           1/1     Running   0           28m
my-nginx-6dd86d77d-l8ct6           1/1     Running   0           28m
tomcat-5bf5db7bbd-xvwlr            1/1     Running   0           3m19s
```

Now let's try to delete the tomcat pod using the user div.

```
[node1 ~]$ kubectl --user=div delete pods tomcat-5bf5db7bbd-xvwlr
Error from server (Forbidden): pods "tomcat-5bf5db7bbd-xvwlr" is forbidden:
```

As you can see, the user is not able to delete the pods, yet it was able to list them. To understand why this behaviour happened, let's have a look at the get-pods Role rules:

- The `apiGroups` is an array that contains the different API namespaces that

- The `resources` is an array that defines which resources this rule applies to
- The `verbs` in an array that contains the allowed verbs. The verb in Kubernetes

Let's assume that we need `div` to have read-only access to the pods, both as a collection and as a single resource (get and list verbs). But we don't want it to delete Pods directly. Instead, we grant it access to the Deployment resource and, through Deployments, it can delete and recreate pods (like though rolling updates). A policy to achieve this may look as follows:

```
[node1 ~]$ rm -f role.yaml
[node1 ~]$ cat > role.yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: get-pods
rules:
- apiGroups: ["*"]
  resources: ["pods"]
  verbs: ["list","get","watch"]
- apiGroups: ["extensions","apps"]
  resources: ["deployments"]
  verbs: ["get","list","watch","create","update","patch","delete"]
^C
[node1 ~]$ kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/get-pods configured
```

We made two changes here:

- Added the `get` and `watch` to the allowed verbs against Pods.
- Created a new rule that targets Deployments and specified the necessary verbs

```
[node1 ~]$ cat > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
```

```

  app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

```

^C

```
[node1 ~]$ kubectl --user=div get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------|-------|---------|----------|-----|
| tomcat-5bf5db7bbd-xvwlr | 1/1 | Running | 0 | 18m |

```
[node1 ~]$ kubectl --user=div apply -f deployment.yaml
```

```
deployment.apps/nginx-deployment created
```

```
[node1 ~]$ kubectl --user=div get po
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------------------|-------|-------------------|----------|-----|
| nginx-deployment-6dd86d77d-bccwq | 0/1 | Pending | 0 | 6s |
| nginx-deployment-6dd86d77d-bhjmn | 0/1 | ContainerCreating | 0 | 6s |
| nginx-deployment-6dd86d77d-hfhhm | 0/1 | ContainerCreating | 0 | 6s |
| tomcat-5bf5db7bbd-xvwlr | 1/1 | Running | 0 | 18m |

Get a single Pod

```
[node1 ~]$ kubectl --user=div get po nginx-deployment-6dd86d77d-hfhhm
```

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------------------|-------|---------|----------|-----|
| nginx-deployment-6dd86d77d-hfhhm | 1/1 | Running | 0 | 95s |

What about deleting this Pod

```
[node1 ~]$ kubectl --user=div delete po nginx-deployment-6dd86d77d-hfhhm
```

```
Error from server (Forbidden): pods "nginx-deployment-6dd86d77d-hfhhm" is forbidden
```

OK so we cannot directly delete Pods. But we should be able to delete them by deleting the deployment.

```
[node1 ~]$ kubectl --user=div delete -f deployment.yaml
deployment.apps "nginx-deployment" deleted
[node1 ~]$ kubectl --user=div get po
NAME                                READY   STATUS    RESTARTS   AGE
tomcat-5bf5db7bbd-xvwlr           1/1     Running   0           27m
```

Notice that in all the preceding examples, we didn't specify a namespace, so our Role is applied to the default namespace. A Role is bound to the namespace defined in its configuration. So, if we changed the metadata of our Role to look like this:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: get-pods
  namespace: web
```

The div user wouldn't have access to the pods or the deployments unless working in the web namespace.

But, sometimes you need to specify Roles that are not bound to a specific namespace but rather to the cluster as a whole. That's when the ClusterRole comes into play.

Cluster-Wide Authorization Using ClusterRoles

ClusterRoles work the same as Roles, but they are applied to the cluster as a whole. They are typically used with service accounts (accounts used and managed internally by the cluster). For example, the Kubernetes External DNS Incubator (<https://github.com/kubernetes-incubator/external-dns>) project uses a ClusterRole to gain the necessary permissions it needs to work. The External DNS Incubator can be used to utilize external DNS servers for Kubernetes service discovery. The application needs read-only access to Services and Ingresses on all namespaces, but it shouldn't be granted any further privileges (like modifying or deleting resources). The ClusterRole for such an account should look as follows:


```
[node1 ~]$ cat > cluster-role-binding.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: external-dns
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: external-dns
rules:
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["services"]
  verbs: ["get","watch","list"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["ingresses"]
  verbs: ["get","watch","list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: external-dns-viewer
subjects:
- kind: ServiceAccount
  name: external-dns
  namespace: default
roleRef:
  kind: ClusterRole
  name: external-dns
  apiGroup: rbac.authorization.k8s.io
^C
[node1 ~]$ kubectl apply -f cluster-role-binding.yaml
serviceaccount/external-dns created
clusterrole.rbac.authorization.k8s.io/external-dns created
clusterrolebinding.rbac.authorization.k8s.io/external-dns-viewer created
```

The above definition contains three definitions:

- A service account to use with the container running the application.

- A `ClusterRole` that grants the read-only verbs to the Service and Ingress resources
- A `ClusterRoleBinding` which works that same as a `RoleBinding` but with `ClusterRole`

Another everyday use case with `ClusterRoles` is granting cluster administrators different privileges depending on their roles. For example, a junior cluster operator should have read-only access to resources to get acquainted; then more access can be granted later on.

Note


- Kubernetes uses RBAC to control different access levels to its resources
- Roles and ClusterRoles use API namespaces, verbs and resources to secure access
- Roles and ClusterRoles are ineffective unless they are linked to a subject
- Roles work within the constraints of a namespace. It would default to the namespace of the role
- ClusterRoles are not bound to a specific namespace as they apply to the cluster

[Join KubeDaily](#)

7 Members Online

Support


MEMBERS ONLINE


 h3ll_boy


 MEE6


 Nitinkashyap

Apex Legends

 ojaswa

 Parmeshwar

 prasad

 vikas027

Free voice chat from Discord

Connect

[Tweets by collabnix](#)

kubelabs is maintained by [collabnix](#).

This page was generated by [GitHub Pages](#).