Replacing Users With Groups

In this lesson, we will amend the access to a cluster as a single user to a group of users.

WE'LL COVER THE FOLLOWING

The User-Specific Namespace

Defining a single user that can access the `jdoe` Namespace was probably the best approach. We expect that only John will want to access it. He is the owner of that Namespace. It's his private playground. Even if he chooses to add more users to it, he'll probably do it independently from our YAML definitions.

After all, what's the point of giving him god-like privileges if not to let him do things without asking for our permission or involvement? From our perspective, that Namespace has, and will continue having only one User.

Exploring the Prospective Roles

We cannot apply the same logic to the permissions in `default` and `dev` Namespaces. We might choose to give everyone in our organization the `view` role in the `default` Namespace. Similarly, developers in our company should be able to deploy, update, and delete resources from the `dev` Namespace.

All in all, we can expect that the number of users in the `view` and `dev` bindings will increase with time. Continually adding new users is repetitive, boring, and error-prone process you probably don't want to do. Instead of becoming a person who hates his tedious job, we can create a system that groups users based on their roles. We already did a step in that direction when we created John's certificate.

Let's take another look at the subject of the certificate we created earlier.

```
openssl req -in keys/jdoe.csr \
    -noout -subject
```

The **output** is as follows.

```
subject=/CN=jdoe/O=devs
```

We can see that the name is `jdoe` and that he belongs to the organization `devs`. We'll ignore the fact that he should probably belong to at least one more organization (`release-manager`).

If you paid close attention, you probably remember that we mentioned a few times that RBAC can be used with Users, Groups, and Service Accounts. Groups are the same as Users, except that they are validating whether the certificate attached to a request to the API belongs to a specified group (`O`), instead of a name (`CN`).

Amending the Permissions

Let's take a quick look at yet another YAML definition.

```
cat auth/groups.yml
```

The **output** is as follows.

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev

---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev
  namespace: dev
subjects:
- kind: Group
  name: devs
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io


---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: view
subjects:
- kind: Group
  name: devs
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
```

You'll notice that the Role Binding `dev` and the Cluster Role Binding `view` are almost the same as those we used before. The only difference is in the `subjects.kind` field. This time, we're using `Group` as the value. As a result, we'll grant permissions to all users that belong to the organization `devs`.

We'll need to switch the context back to `minikube` before we apply the changes.

```
kubectl config use-context minikube

kubectl apply -f auth/groups.yml \
    --record
```

The **output** is as follows.

```
namespace/dev configured
rolebinding.rbac.authorization.k8s.io/dev configured
clusterrolebinding.rbac.authorization.k8s.io/view configured
```

We can see that the new definition reconfigured a few resources.

Now that the new definition is applied, we can validate whether John can still create objects inside the `dev` Namespace.

```
kubectl --namespace dev auth \
    can-i create deployments --as jdoe
```

The output is `no`, indicating that `jdoe` cannot `create deployments`. Before you start wondering what's wrong, I should inform you that the response is expected and correct. The `--as` argument is impersonating John, but the certificate is still from `minikube`. Kubernetes has no way of knowing that `jdoe` belongs to the group `devs`. At least, not until John issues a request with his own certificate.

Instead of using the `--as` argument, we'll switch back to the `jdoe` context and try to create a Deployment.

```
kubectl config use-context jdoe

kubectl --namespace dev \
     create deployment new-db \
     --image mongo:3.3
```

This time the output is `deployment "new-db" created`, clearly indicating that the John as a member of the `devs` group can `create deployments`.

From now on, any user with a certificate that has `/O=devs` in the subject will have the same permissions as John within the `dev` Namespace as well as `view` permissions everywhere else. We just saved ourselves from constantly modifying YAML files and applying changes.

In the next lesson, you can test your understanding of securing a Kubernetes cluster with the help of a quick quiz.