Kubernetes Role Based Access Control (RBAC)

When a request is sent to the API Server, it first needs to be authenticated (to make sure the requestor is known by the system) before it's authorized (to make sure the requestor is allowed to perform the action requested).

RBAC is a way to define which users can do what within a Kubernetes cluster.

If you are working on Kubernetes for some time you may have faced a scenario where you have to give some users limited access to your Kubernetes cluster. For example you may want a user, say Michale from development, to have access only to some resources that are in the development namespace and nothing else. To achieve this type of role based access, we use the concept of Authentication and Authorization in Kubernetes.

Broadly, there are three kinds of users that need access to a Kubernetes cluster:

1. Developers/Admins:

Users that are responsible to do administrative or developmental tasks on the cluster. This includes operations like upgrading the cluster or creating the resources/workloads on the cluster.

2. End Users:

Users that access the applications deployed on our Kubernetes cluster. Access restrictions for these users are managed by the applications themselves. For example, a web application running on Kubernetes cluster, will have its own security mechanism in place, to prevent unauthorized access.

3. Applications/Bots:

There is a possibility that other applications need access to Kubernetes cluster, typically to talk to resources or workloads inside the cluster. Kubernetes facilitates this by using Service Accounts..

RBAC in Kubernetes is based on three key concepts:

1. Verbs: This is a set of operations that can be executed on resources. There are many verbs, but they're all Create, Read, Update, or Delete (also known as CRUD).
2. API Resources: These are the objects available on the clusters. They are the pods, services, nodes, Persistent Volumes and other things that make up Kubernetes.
3. Subjects: These are the objects (Users, Groups, Processes(Service Account)) allowed access to the API, based on Verbs and Resources.

```
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1beta1
metadata:
   name: pod-read-create
   namespace: test
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "create"]
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-pods
  namespace: test
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: ns-admin
  apiGroup: rbac.authorization.k8s.io
```

RBAC Role

A Role example named example-role which allows access to
the mynamespace with get, watch, and list operations:

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

In the rules above we:

1. apiGroups: [""] – set core API group
2. resources: ["pods"] – which resources are allowed for access
3. ["get", "watch", "list"] – which actions are allowed over the resources above

RBAC RoleBingding

To “map” those permissions to users we are using Kubernetes RoleBingding, which sets example-role in the mynamespace for the example-user user:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: example-rolebinding
  namespace: mynamespace
subjects:
- kind: User
  name: example-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: example-role
  apiGroup: rbac.authorization.k8s.io
```

Here we set:

- subjects:
  - kind: User – an object type which will have access, in our case this is a regular user
  - name: example-user – a user's name to set the permissions
- roleRef:
  - kind: Role – what exactly will be attached to the user, in this case, it is the Role object type
  - name: example-role – and the role name as it was set in the name: example-role in the example above

Role vs ClusterRole

Alongside with the Role and ClusterRole which are set of rules to describe permissions – Kubernetes also has RoleBinding and ClusterRoleBinding objects.

The difference is that Role is used inside of a namespace, while ClusterRole is cluster-wide permission without a namespace boundaries, for example:

- allow access to a cluster nodes
- resources in all namespaces
- allow access to endpoints like /healthz
-

A ClusterRole looks similar to a Role with the only difference that we have to set its kind as ClusterRole:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: example-clusterrole
rules:
- apiGroups: [""]
```

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

And a ClusterRoleBinding example:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: example-clusterrolebinding
subjects:
- kind: User
  name: example-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: example-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

Keep in mind that once you'll create a Binding you'll not be able to edit its roleRef value – instead, you'll have to delete a Binding and recreate and again

- Kubernetes uses RBAC to control different access levels to its resources depending on the rules set in Roles or ClusterRoles.
- Roles and ClusterRoles use API namespaces, verbs and resources to secure access.
- Roles and ClusterRoles are ineffective unless they are linked to a subject (User, serviceAccount...etc) through RoleBinding or ClusterRoleBinding.
- Roles work within the constraints of a namespace. It would default to the "default" namespace if none was specified.
- ClusterRoles are not bound to a specific namespace as they apply to the cluster as a whole.

Goal

- Create a user "Michale"
- He cannot be trusted, so he should have only limited access:
  - Play around with pods and configmaps
  - Only in the "default" namespace
- We will create a kubeconf for him, so that he can use kubectl right away

Kubernetes supports several user authentication methods. In this example, we'll use only one authentication method, the X509 certificate to create a user account called Michale.

So we will conduct the following steps:

- Create a key, CSR and eventually a certificate that is signed by the CA that was created by kubeadm during the installation.
- Create a kubeconf file for our user
- Setup a Role and a Rolebinding

Creating the Certificate

We can find the CA files in /etc/kubernetes/pki. Those files were created by kubeadm during the installation process and are protected. So we will run the following commands by root.

```
# ls -l /etc/kubernetes/pki/ca*
```

If Kubernetes Cluster is Created Using KOPS. Kops stores the CA key and certificate in its S3 bucket, so you can download said files like so:

Download the kops-generated CA certificate and signing key from S3:

Get net name of CA file

aws s3 ls s3://<s3BucketName>/<clusterName>/pki/private/ca

Ex:

aws se ls s3://balajimtaprilbatch.k8s.local/balajimtaprilbatch.k8s.local/pki/private/ca/

Download CA File Replace <File> with Name of the file which ends with .key which listed by above command.

aws s3 cp s3://<s3BucketName>/<clusterName>/pki/private/ca
/<File>.key ca.key

ex:

aws s3 cp
s3://balajimtaprilbatch.k8s.local/balajimtaprilbatch.k8s.local/pki/private/ca/6843968403600158568772877954.key ca.key

Find .crt file name

aws s3 ls s3://<s3BucketName>/<clusterName>/pki/issued/ca

aws s3 cp s3://<s3BucketName>/<clusterName>/pki/issued/ca/<FileName>.crt ca.crt

Creating a Kubernetes User Account Using X509 Client Certificate

openssl genrsa -out Michale.key 2048

openssl rand -out .rnd -hex 256

Then create a CSR using the key we created in step one:

openssl req -new -key Michale.key -out Michale.csr -subj "/CN=Michale/O=Developer"

The CN defines the username. The O refers to the group he is a member of. If you want to assign Hans to more than one group use: "/CN= Michale/O= Developer/O=vmware"
We now have two files in our directory: Michale.csr and Michale.key

openssl x509 -req -in Michale.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out Michale.crt -days 365

You can omit the "-CAcreateserial" if you have created other certificates before and/or have a "ca.srl" file

Test Certificate:

openssl x509 -text -noout -in Michale.crt | head -20

Setup the kubeconf File

Get API Server Details by executing

kubectl cluster-info

And Repliace <SERVER>

kubectl config --kubeconfig=MichaleConfig --embed-certs=true \
set-cluster kubernetes \
--server=<SERVER> \
--certificate-authority=ca.crt

Ex:

kubectl config --kubeconfig=MichaleConfig --embed-certs=true \

```
set-cluster kubernetes \
--server=https://api-balajimtaprilbatch-k8-g1jcko-2103539152.ap-south-
1.elb.amazonaws.com \
--certificate-authority=ca.crt
```

```
kubectl config set-credentials Michale  --kubeconfig=MichaleConfig  --client-certificate=
Michale.crt --client-key= Michale.key
```

```
kubectl config --kubeconfig=MichaleConfig \
set-context standard \
 --cluster=kubernetes --namespace=default --user=Michale

kubectl config --kubeconfig=MichaleConfig \
use-context standard
```

Finally We have MichaleConfig file

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: devrole
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "update", "list"]
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dev-role-binding-user
  namespace: default
subjects:
- kind: User
  name: Michale
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: devrole
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl --kubeconfig MichaleConfig get pods -n kube-system
```