Kubernetes Access Control: Authentication and Authorization

In this lab you are going to,

- Create users and groups and setup certs based authentication
- Create service accounts for applications
- Create Roles and ClusterRoles to define authorizations
- Map Roles and ClusterRoles to subjects i.e. users, groups and service accounts using RoleBingings and ClusterRoleBindings.

How one can access the Kubernetes API?

The Kubernetes API can be accessed by three ways.

- Kubectl - A command line utility of Kubernetes
- Client libraries - Go, Python, etc.,
- REST requests

Who can access the Kubernetes API?

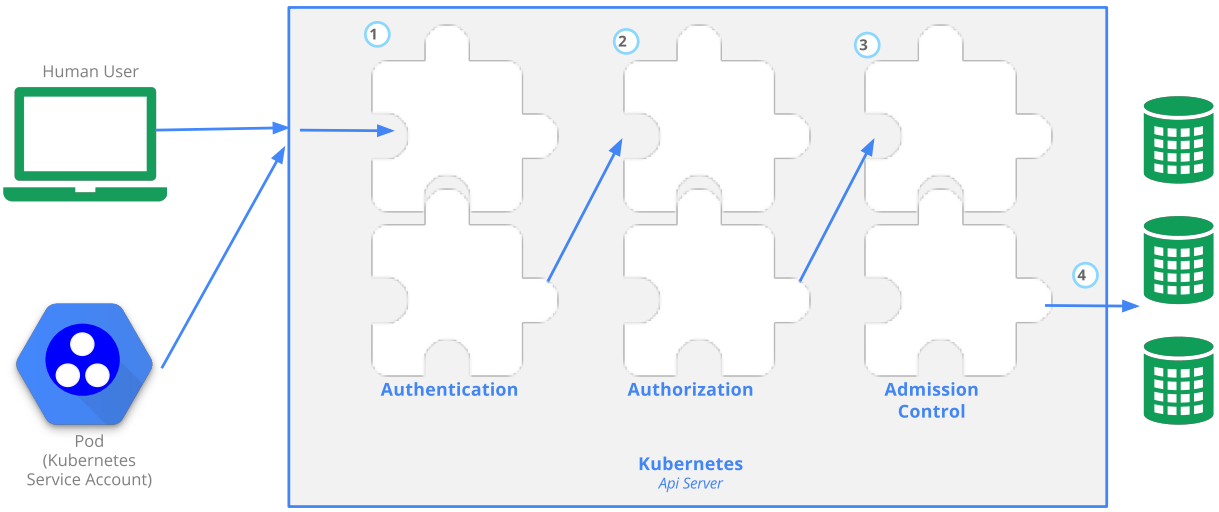
Kubernetes API can be accessed by,

- Human Users
- Service Accounts

Each of these topics will be discussed in detail in the later part of this chapter.

Stages of a Request

When a request tries to contact the API , it goes through various stages as illustrated in the image given below.



source: official kubernetes site

api groups and resources

apiGroup	Resources
apps	daemonsets, deployments, deployments/rollback, deployments/scale, replicasets, replicasets/scale, statefulsets, statefulsets/scale
core	configmaps, endpoints, persistentvolumeclaims, replicationcontrollers, replicationcontrollers/scale, secrets, serviceaccounts, services,services/proxy
autoscaling	horizontalpodautoscalers
batch	cronjobs, jobs
policy	poddisruptionbudgets
networking.k8s.io	networkpolicies
authorization.k8s.io	localsubjectaccessreviews
rbac.authorization.k8s.io	rolebindings,roles
extensions	deprecated (read notes)

Notes

In addition to the above apiGroups, you may see **extensions** being used in some example code snippets. Please note that **extensions** was initially created as a experiement and is been deprecated, by moving most of the matured apis to one of the groups mentioned above. You could read this comment and the thread to get clarity on this.

Role Based Access Control (RBAC)

Group	User	Namespaces	Resources	Access Type (verbs)
ops	maya	all	all	get, list, watch, update, patch, create, delete, deletecollection
dev	kim	instavote	deployments, statefulsets, services, pods, configmaps, secrets, replicasets, ingresses, endpoints, cronjobs, jobs, persistentvolumeclaims	get, list , watch, update, patch, create
interns	yono	instavote	readonly	get, list, watch

Service Accounts	Namespace	Resources	Access Type (verbs)
monitoring	all	all	readonly

Creating Kubernetes Users and Groups

Generate the user's private key

```
mkdir -p  ~/.kube/users
cd ~/.kube/users

openssl genrsa -out maya.key 2048
openssl genrsa -out kim.key 2048
openssl genrsa -out yono.key 2048
```

[sample Output]

```
openssl genrsa -out maya.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
.....................+++
e is 65537 (0x10001)
```

Lets now create a **Certification Signing Request (CSR)** for each of the users. When you generate the csr make sure you also provide

- CN: This will be set as username
- O: Org name. This is actually used as a **group** by kubernetes while authenticating/authorizing users. You could add as many as you need

e.g.

```
openssl req -new -key maya.key -out maya.csr -subj "/CN=maya/O=ops/O=example.org"
openssl req -new -key kim.key -out kim.csr -subj "/CN=kim/O=dev/O=example.org"
openssl req -new -key yono.key -out yono.csr -subj "/CN=yono/O=interns/O=example.org"
```

In order to be deemed authentic, these CSRs need to be signed by the **Certification Authority (CA)**

which in this case is Kubernetes Master. You need access to the following files on kubernetes master,

- Certificate : ca.crt (kubeadm) or ca.key (kubespray)
- Pricate Key : ca.key (kubeadm) or ca-key.pem (kubespray)

You would typically find it at one of the following paths

- /etc/kubernetes/pki (kubeadm)
- /etc/kubernetes/ssl (kubespray)

To verify which one is your cert and which one is key, use the following command,

```
$ file ca.pem
ca.pem: PEM certificate


$ file ca-key.pem
ca-key.pem: PEM RSA private key
```

Once signed, .csr files with added signatures become the certificates that could be used to authenticate.

You could either

- move the crt files to k8s master, sign and download
- copy over the CA certs and keys to your management node and use it to sign. Make sure to keep your CA related files secure.

In the example here, I have already downloaded **ca.pem** and **ca-key.pem** to my management workstation, which are used to sign the CSRs.

Assuming all the files are in the same directory, sign the CSR as,

```
openssl x509 -req -CA ca.pem -CAkey ca-key.pem -CAcreateserial -days 730 -in maya.csr -out maya

openssl x509 -req -CA ca.pem -CAkey ca-key.pem -CAcreateserial -days 730 -in kim.csr -out kim.c

openssl x509 -req -CA ca.pem -CAkey ca-key.pem -CAcreateserial -days 730 -in yono.csr -out yono
```

Setting up User configs with kubectl

In order to configure the users that you created above, following steps need to be performed with kubectl

- Add credentials in the configurations
- Set context to login as a user to a cluster
- Switch context in order to assume the user's identity while working with the cluster

to add credentials,

```
kubectl config set-credentials maya --client-certificate=/absolute/path/to/maya.crt --client-key=/absolute/path/to/maya.key

kubectl config set-credentials kim --client-certificate=/absolute/path/to/kim.crt --client-key=

kubectl config set-credentials yono --client-certificate=/absolute/path/to/yono.crt --client-ke
```

where,

- Replace /absolute/path/to/ with the path to these files.
  - `invalid` : ~/.kube/users/yono.crt
  - `valid` : /home/xyz/.kube/users/yono.crt

And proceed to set/create contexts (user@cluster). If you are not sure whats the cluster name, use the following command to find,

```
kubectl config get-contexts
```

[sample output]

```
CURRENT   NAME                          CLUSTER         AUTHINFO             NAMESPACE
          admin-prod           prod   admin-cluster.local   instavote
          admin-cluster4                cluster4        admin-cluster4       instavote
*         kubernetes-admin@kubernetes   kubernetes      kubernetes-admin     instavote
```

where, **prod**, **cluster4** and **kubernetes** are cluster names.

To set context for **prod** cluster,

```
kubectl config set-context maya-prod --cluster=prod  --user=maya --namespace=instavote

kubectl config set-context kim-prod --cluster=prod  --user=kim --namespace=instavote

kubectl config set-context yono-prod --cluster=prod  --user=yono --namespace=instavote
```

Where,

- maya-prod : name of the context
- prod : name of the kubernetes cluster you set while creating it
- maya : user you created and configured above to connect to the cluster

You could verify the configs with

```
kubectl config get-contexts

CURRENT   NAME         CLUSTER   AUTHINFO     NAMESPACE
*         admin-prod   prod      admin-prod
          kim-prod     prod      kim
          maya-prod    prod      maya
          yono-prod    prod      yono
```

and

```
kubectl config view

apiVersion: v1
clusters:
- cluster:
   certificate-authority-data: REDACTED
   server: https://128.199.248.240:6443
 name: prod
contexts:
- context:
   cluster: prod
   user: admin-prod
 name: admin-prod
- context:
   cluster: prod
   user: kim
 name: kim-prod
- context:
   cluster: prod
   user: maya
 name: maya-prod
- context:
   cluster: prod
   user: yono
 name: yono-prod
current-context: admin-prod
kind: Config
preferences: {}
users:
- name: admin-prod
 user:
   client-certificate-data: REDACTED
   client-key-data: REDACTED
- name: maya
 user:
   client-certificate: users/~/.kube/users/maya.crt
   client-key: users/~/.kube/users/maya.key
```

You could assume the identity of user **yono** and connect to the **prod** cluster as,

```
kubectl config use-context yono-prod

kubectl config get-contexts

CURRENT   NAME          CLUSTER   AUTHINFO     NAMESPACE
          admin-prod    prod      admin-prod
          kim-prod      prod      kim
          maya-prod     prod      maya
*         yono-prod     prod      yono
```

And then try running any command as,

```
kubectl get pods
```

Alternately, if you are a admin user, you could impersonate a user and run a command with that literally using --as option

```
kubectl config use-context admin-prod
kubectl get pods --as yono
```

[Sample Output]

```
No resources found.
Error from server (Forbidden): pods is forbidden: User "yono" cannot list pods in the namespace
```

Either ways, since there are authorization rules set, the user can not make any api calls. Thats when you would create some roles and bind it to the users in the next section.

Define authorisation rules with Roles and ClusterRoles

Whats the difference between Roles and ClusterRoles ??

- Role is limited to a namespace (Projects/Orgs/Env)
- ClusterRole is Global

Lets say you want to provide read only access to **instavote**, a project specific namespace to all users in the **example.org**

`file: interns-role.yaml`

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  namespace: instavote
  name: interns
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
```

In order to map it to all users in **example.org**, create a RoleBinding as

`interns-rolebinding.yml`

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: interns
  namespace: instavote
subjects:
- kind: Group
  name: interns
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: interns
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl create -f interns-role.yml

kubectl create -f interns-rolebinding.yml
```

To gt information about the objects created above,

```
kubectl get roles,rolebindings -n instavote

kubectl describe role interns
kubectl describe rolebinding interns
```

To validate the access,

```
kubectl config use-context yono-prod
kubectl get pods
```

To switch back to admin,

```
kubectl config use-context admin-prod
```

Exercise

Create a Role and Rolebinding for **dev** group with the authorizations defined in the table above. Once applied, test it