**Understanding Security in Kubernetes**

Authentication and Authorization.

Diagram, schematic

Description automatically generated

**Users in Kubernetes**

* Users are managed by external systems.
* No User API Object.
* Authentication is pluggable. Authentication plugin implements authentication
* Usernames used for access control and logging.
* Users can be aggregated into groups.

**Service Accounts:**

* Authenticate Pods to the API Server
* Apply permissions for authorization.
* Default ServiceAccount per Namespace
* All Pods must have a ServiceAccount defined. By default, it is **Default** ServiceAccount.

**To get detailed information about the cluster:**

kubectl cluster-info

Kubernetes should report that both Kubernetes master and KubeDNS are running on localhost:6443

**Kubectl performs all its operations against the current context:**

kubectl config get-contexts

Note: Following files contains all Clusters and Contexts information

C:\Users\<user-name>\.kube\**config**

/home/<username>/.kube/config

To set the current context:

kubectl config **use-context**  <context-name>

**View the cluster and context configuration**

kubectl config view

kubectl config view --raw

Notice:

* We're using certificates to authenticate to our cluster
* Our certificate information is stored in the .kube/config
* kubectl reads the credentials and sends the API request on to the API Server

**Investigating Certificate based authentication used by kubectl from a client machine.**

#Let's read the certificate information out of our kubeconfig file

Following Command will work for Docker Desktop on Mac

kubectl config view --raw -o jsonpath='{ .users[\*].user.client-certificate-data }' | base64 --decode > **admin.crt**

openssl x509 -in **admin.crt** -text -nooutkub

OR

Following Command will work for Minikube on Mac/Windows/Linux

openssl x509 -in /home/dssadmin/.minikube/profiles/minikube/client.crt -text -noout

OR

Following Command will work on Docker Desktop on Windows

Use o/p of

kubectl config view --raw -o jsonpath='{ .users[\*].user.client-certificate-data }'

In <https://www.base64encode.com/> and <https://www.sslchecker.com/certdecoder>

#Look for Subject: CN= is the username which is kubernetes-admin, it's also in the group (O=) system:masters

Using Service Account with Secret to Pull a Private Image from the Docker Hub Registry

**Create Secret with Docker Login Credentials**

kubectl create secret docker-registry mydockersecret --docker-username="sandeepsoni" --docker-password "\*\*\*\*\*\*\*" --docker-server=docker.io

**Create a Service Account with ImagePullSecret**

myserviceaccount.yaml

apiVersion: v1

kind: ServiceAccount

metadata:

  name: mysvcaccount1

  namespace: default

imagePullSecrets:

- name: mydockersecret

**Configuring a Service Account in a Pod Spec**

**nginx-pod.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: mynginx

  labels:

    environment: production

    tier: frontend

spec:

**serviceAccountName: mysvcaccount1**

  containers:

  - name: nginx

    image: sandeepsoni/nginx:latest #Private Image

    ports:

    - containerPort: 80

~~imagePullSecrets:~~

~~- name: mydockersecret~~

**Commands:**

kubectl apply -f myserviceaccount.yaml

kubectl apply -f nginx-pod.yaml

kubectl get pod mynginx -o yaml #Note the property serviceAccountName: mysvcaccount1

# Also, note that the Private Image was pulled from registry using the secret associated with the service account.

**Role Based Access Control**

Authorization is based on RESTful API semantics: Concept of Verb on Noun

Default deny, **rules** are written to allow actions on the resource.

RBAC rules are applied when a request is made to the API server.

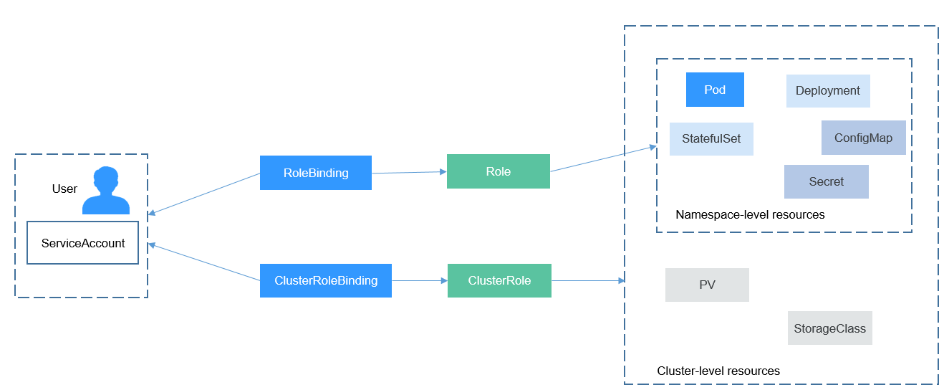
**Subjects** - Users, Groups or ServiceAccounts

Diagram

Description automatically generated

**API Objects for Implementing RBAC Rules**

1. Role
2. RoleBinding
3. ClusterRole
4. ClusterRoleBinding



**Defining Roles and ClusterRoles**

* apiGroups – An empty string designates the Core API group.
* Resources – Pods, Services, Deployments, Nodes and more.
* Verbs – get, list, create, update, patch, delete

Note: Roles and ClusterRoles can have several Rules defined together.

Note: Namespace is applicable for only ServiceAccount. User and Group are cluster-wide resources.

**About Roles**

* Roles are what can be done to Resources.
* Roles are made up of one or many Rules.
* Roles are **namespaced** – They are only available in the namespace they are created.
* Verbs on resources.
  + Get Pods, Create Deployment
* Default deny, add permissions to Resources.
* There is no deny permission

kubectl create role demorole -n ns1 --resource=pods --verb=get,list,create

OR

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

  name: demorole

  namespace: ns1

rules:

- apiGroups: [""]

  resources: ["pods"]

  verbs: ["get", "list", "create"]

- apiGroups: ["apps"]

  resources: ["deployments"]

  verbs: ["get", "list"]

Note: **apiGroups** – the namespace of the resource, which you use in apiVersion in object metadata; Pods are just v1 so the apiGroup is blank, Deployments are apps/v1 so the apiGroup would be apps.

**About RoleBinding**

* Defines the Subjects and refers to a Role/ClusterRole.
* Who can do what defined in a Role/ClusterRole.
* Role and RoleBinding are used in namespaced scoped security. They must be defined in the same namespace.
* ClusterRole and RoleBinding are used provide access to more than one namespace or the whole cluster

apiVersion: v1

kind: ServiceAccount

metadata:

  name: pod-manager-dep-reader

  namespace: ns1

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: demorolebinding

  namespace: ns1

roleRef:

  apiGroup: rbac.authorization.k8s.io

  kind: Role

  name: demorole

subjects:

- kind: ServiceAccount

  name: pod-manager-dep-reader

  namespace: ns1

- apiGroup: rbac.authorization.k8s.io

  kind: User

  name: [demouser@somedomain.com](mailto:demouser@somedomain.com)

**Now Try the following Commands:**

1. kubectl get pods -n ns1 --as=system:serviceaccount:ns1:pod-manager-dep-reader
2. kubectl get deployments -n ns1 --as=system:serviceaccount:ns1:pod-manager-dep-reader
3. kubectl get services -n ns1 --as=system:serviceaccount:ns1:pod-manager-dep-reader
4. kubectl get pods -n ns1 --as=demouser@somedomain.com

**About ClusterRoles**

* To avoid the problem of "Defining Roles in each namespace can increase administrative overhead and can be error prone".
* Similar to a Role, enables access to Resources.
* Cluster scoped resources
  + Nodes, PersistentVolumes
* Give access across more than one namespace or all namespaces.

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

  name: democlusterrole

rules:

- apiGroups: [""]

  resources: ["pods"]

  verbs: ["get", "list", "create"]

- apiGroups: ["apps"]

  resources: ["deployments"]

  verbs: ["get", "list"]

- apiGroups: ["metrics.k8s.io"]

  resources: ["nodes"]

  verbs: ["get", "list", "watch"]

**About ClusterRoleBinding**

* ClusterRoleBinding grants access cluster-wide resources.
* Combing a ClusterRole with a ClusterRoleBinding
* Will scope security independent of namespace
  + Non-namespaced
  + Cluster-scoped resources

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

  name: democlusterrolebinding

roleRef:

  apiGroup: rbac.authorization.k8s.io

  kind: ClusterRole

  name: **democlusterrole**

subjects:

- kind: ServiceAccount

  name: **pod-manager-dep-reader**

  namespace: ns1

Now the SA can be used for getting list of pods, deployments and nodes of **ANY NAMESPACE**

kubectl get pods **-A** --as=system:serviceaccount:ns1:pod-manager-dep-reader

**What to use when?**

* Use **Role** and a **RoleBinding** to scope security to a **single** namespace. Example: Giving access to developer for a given namespace to work with entire project in that namespace.
* Use **ClusterRole** and **ClusterRoleBinding** to scope security to **all** namespaces OR **cluster-scoped** resources. Example: Cluster Admin responsible for all resources including persistent volumes.
* Use **ClusterRole** and **RoleBinding** to scope security to **several** namespaces. Example: Giving access to some architect who would work with multiple projects in cluster but not entire cluster.

Default ClusterRoles

**cluster-admin**

* Cluster-wide super user. Full Control.
* Kubernetes-admin user has this role.
* When used with RoleBinding, this will allow namespace wide Admin access including the namespace itself.
* Can Edit Roles, RoleBindings and Resource Quotas

**admin**

* Full access within a namespace
* When used with RoleBinding, this will allow namespace wide Admin access but **not** the namespace itself.
* Subjects can Edit Roles and RoleBindings within that namespace.

**edit**

* Read/write within a Namespace
* Cannot View/Edit Roles, RoleBindings and Resource Quotas
* Access to Secrets

**view**

* Read-only within a Namespace
* Cannot View/Edit Roles, RoleBindings and Resource Quotas
* No Access to Secrets

**Docker desktop Kubernetes NOT enforcing RBAC rules**

<https://www.portainer.io/blog/docker-desktop-kubernetes-not-enforcing-rbac-rules>

**Lab #1 - Role/RoleBinding**

kubectl create namespace ns1

kubectl create deployment nginx --image=nginx **--namespace ns1**

#Create a Role, apiGroup is '' since a Pod is in core. Resources (pods) will need to be plural.

kubectl create **role** demorole --verb=get,list --resource=pods **--namespace ns1**

#Create a RoleBinding, defining which user can access the resources defined in the Role demorole

kubectl create **rolebinding** demorolebinding --role=**demorole** --user=**demouser** --namespace **ns1**

#Testing access to resources using can-i and using impersonation...this is a great way to test your rbac configuration

kubectl auth can-i list pods  #yes, runs as kubernetes-admin

kubectl auth can-i list pods --as=demouser   #no, runs as demouser, but wrong namespace

kubectl auth can-i list pods --as=demouser --namespace ns1 #yes, runs as demo user which has rights within the ns1 namespace

kubectl auth can-i list deployments --as=demouser --namespace ns1 #no, runs as demouser, but user cannot get/list deployments...just pods

#Get all the pods in our deployment AS our demouser

kubectl get pods --namespace ns1 **--as=demouser**

#Let's try to delete a pod using our user demouser that can only get/list pods in the ns1 namespace

PODNAME=$(kubectl get pods -l app=nginx --as=demouser --namespace ns1 -o jsonpath='{ .items[\*].metadata.name }')

echo $PODNAME

#The user demouser does not have the permissions to delete a pod, this will fail

kubectl **delete** pod $PODNAME --namespace ns1 --as=demouser

#Since this user has only get and list pods, if we try to access another resource with this service account it will fail

#Let's hold onto this user configuration a little longer and we'll adjust its rights to control that deployment

kubectl get deployments --namespace ns1 --as=demouser

**#2 - ClusterRole/ClusterRoleBinding**

#Create a ClusterRole to access cluster wide/non-namespaced resources

#Goal is to give demouser access to a cluster-wide resource, nodes.

kubectl auth can-i list nodes --as=demouser #no

kubectl get nodes --as=demouser

#To give this user access to the node information, we can use a clusterrole and clusterrolebinding

kubectl create **clusterrole** democlusterrole --verb=get,list --resource=nodes

#Create a ClusterRoleBinding, allowing the user to read Node information

kubectl create clusterrolebinding democlusterrolebinding --clusterrole=**democlusterrole** --user=**demouser**

kubectl auth can-i list nodes --as=demouser #yes

kubectl get nodes --as=demouser

kubectl create **clusterrole** democlusterroleforpod --verb=get,list --resource=**pods**

kubectl create **clusterrolebinding** democlusterrolebindingforpodanddemouser –clusterrole=democlusterroleforpod --user=demouser

kubectl auth can-i list pods #yes

kubectl auth can-i list pods -n ns1 #yes

kubectl auth can-i get pods -n kube-system #yes

**#3 - Using ClusterRole/RoleBinding - to give a user access to more than one namespace**

#Let's now create a new namespace and a deployment and try to access that deployment with our user...it will fail

kubectl create namespace ns2    #runs as kubernetes-admin

kubectl create deployment nginx2 --image=nginx --namespace ns2    #runs as kubernetes-admin

kubectl get deployment --as=demouser --namespace ns2 #no

#Rather than maintain the role in the demorole Role in each namespace, let's delete that RoleBinding and Role from the first demo

# and use a ClusterRole and RoleBinding for access to resources in  ns1 and ns2 namespaces for our demouser

kubectl delete rolebinding demorolebinding --namespace ns1

kubectl delete role demorole --namespace ns1

#Create a ClusterRole to be used on both namespaces enabling this user to get/list pods in both namespaces

~~kubectl create~~ **~~clusterrole~~** ~~democlusterrolepods --verb=get,list --resource=pods~~

#Create a RoleBinding in each namespace referring to the ClusterRole we just created

#The name can be the same since the rolebinding is in each namespace

#This gives our demouser access to get/list pods in each namespace

kubectl create **rolebinding** demorolebindingpods --**clusterrole**=democlusterrolepods  --**user**=demouser **--namespace ns1**

kubectl create **rolebinding** demorolebindingpods --**clusterrole**=democlusterrolepods  --**user**=demouser **--namespace ns2**

#Can we read from both namespaces with our demouser?

kubectl auth can-i list pods --as=demouser --namespace ns1 #Yes

kubectl auth can-i list pods --as=demouser --namespace ns2 #Yes

kubectl auth can-i list pods --as=demouser --namespace default #No

kubectl auth can-i list pods --as=demouser --namespace kube-system #No

kubectl get pods --as=demouser --namespace ns1

kubectl get pods --as=demouser --namespace ns2

#Clean up this demo

kubectl delete clusterrole democlusterrole

kubectl delete clusterrole democlusterrolepods

kubectl delete clusterrolebinding democlusterrolebinding

kubectl delete namespace ns1

kubectl delete namespace ns2

#Clean up the user we createing the module 3 demos (or not you can keep it around)

sudo userdel --remove demouser