**ROLE BASED ACCESS CONTROL IN AWS EKS CLUSTER**

**Prerequisites:**

Workstation with kubectl and aws-cli installed

EKS cluster up and running in AWS

**Install AWS CLI :**

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

unzip awscliv2.zip

sudo ./aws/install

aws –version

**Install eksctl :**

curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/ download/eksctl\_$(uname -s)\_amd64.tar.gz" | tar xz -C /tmp

sudo mv /tmp/eksctl /usr/local/bin

eksctl version

**Install kubectl :**

curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin

kubectl version --short –client

**creating cluster using eksctl :** vpc – container vpc ->vpc-05e69865fd7621958 subnet ->DMZ-1,DMZ-2

eksctl create cluster

--name eks-demo30

--version 1.17

--region ap-southeast-1

--nodegroup-name eks-demo30-wns

--node-type t3.medium

--nodes 1

--nodes-min 1

--nodes-max 2

--ssh-access

--ssh-public-key lahari-eks

--managed

--vpc-public-subnets subnet-0fb67b0aecf8222a4,subnet-069743f33a9835f0b

**To delete a resource :**

eksctl delete cluster --region=ap-southeast-1 --name=eks-demo21

**Available verbs :**

**Steps to create user:**

1. **CREATE AN IAM USER**

We need to create an user using console in aws .

Let us create an user pavan and download the .csv credentials files respectively.

1. **AWS CLI**

We need to configure the aws configuration for the user that we are going to work with.

$ aws configure --profile pavan

AWS Access Key ID [None]: XXXXXXXXXXXXXXX

AWS Secret Access Key [None]: XXXXXXXXXXXXXX

Default region name [None]: us-east-1

Default output format [None]: text

Here, the default region name and output format are optional.

We, can check whether the configuration is proper or not using

$ aws sts get-caller-identity --profile pavan

Output:

{

"UserId": "AIDAX7JPBEM4A6FTJRTMB",

"Account": "123456789012",

"Arn": "arn:aws:iam::123456789012:user/pavan"

}

1. **CREATING A ROLE**

**Role** is the presence of a namespace, because the permissions defined here are namespace-scoped.

A namespace is a way to group services for an application.

**CREATE NAMESPACE**

kubectl create namespace develop

Create a role.yaml file

kind: Role

metadata:

name: pavan-role

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["list"]

Here, to the role, list permission to the pods resource in your cluster is given.

The ***resources*** can be configmaps, secrets, deployments, events like wise.

The ***kind*** refers to whether role or cluster role.

The ***verbs*** are the actions or operations to be performed on resources like list, watch, update, delete, create, get.

The ***apiGroups*** key defines the location in the API where the resources are found. If you provide an empty value in this list, it means the core API group.

$ kubectl apply –f role.yaml

**ClusterRole** is the absence of a namespace, because the permissions defined here are cluster-scoped.

Create a clusterrole.yaml

kind: ClusterRole  
apiVersion: rbac.authorization.k8s.io/v1  
metadata:  
name: : pavan-clusterrole  
rules:  
- apiGroups: [""]  
resources: ["secrets"]  
verbs: ["get",”list”,”watch”]

Here, to the cluster role, list , get ,watch permission to the secrets resource in your cluster is given. You can also have create , delete, deletecollection ,patch ,update actions.

The ***resources*** can be configmaps, secrets, deployments, events like wise.

The ***kind*** refers to whether role or cluster role.

The ***verbs*** are the actions or operations to be performed on resources like list, watch, update, delete, create, get.

The ***apiGroups*** key defines the location in the API where the resources are found. If you provide an empty value in this list, it means the core API group.

$ kubectl apply –f clusterrole.yaml

1. **ROLE BINDING TO THE USER**

With the role configured we need to create a corresponding ***RoleBinding***. A ***RoleBinding*** allows you to associate a Role with a user or list of users. This grants the Role permissions to the users. The user(s) are defined under subjects, and the Role association under role references (roleRef).

**ROLE BINDING**

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pavan-role-binding

namespace:develop

subjects:

- kind: User

name: pavan

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pavan-role

apiGroup: rbac.authorization.k8s.io

The ***apiGroup*** key defines the location in the API where the resources are found.

$ kubectl apply -f role-binding.yaml

**CLUSTER ROLE BINDING**

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: pavan-clusterrole-binding

namespace:develop

subjects:

- kind: User

name: pavan

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: pavan-clusterrole

apiGroup: rbac.authorization.k8s.io

Here, the permissions are granted in the ***develop*** namespace.

In case you want to grant permissions for your IAM role in all namespaces, you can simply use ClusterRoleBinding as follows:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: pavan-clusterrole-binding

subjects:

- kind: User

name: pavan

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: pavan-clusterrole

apiGroup: rbac.authorization.k8s.io

The ***apiGroup*** key defines the location in the API where the resources are found.

$ kubectl apply -f clusterrole-binding.yaml

1. **ADDING THE USER TO THE AWS-AUTH CONFIGMAP**

If you want to grant additional AWS users or roles the ability to interact with your EKS cluster, you must add the users/roles to the ***aws-auth ConfigMap*** within Kubernetes in the kube-system namespace.

You can do this by either editing it using the ***kubectl*** edit command:

$ kubectl edit configmap aws-auth -n kube-system

Or by importing the ***aws-auth ConfigMap*** and applying the changes:

$ kubectl get configmap aws-auth -n kube-system -o yaml > aws-auth.yaml

Add the user under the mapUsers as an item in the aws-auth ConfigMap:

data:

mapUsers: |

- userarn: arn:aws:iam::123456789012:user/pavan

username: pavan

After adding, the configmap file while look as follows:

apiVersion: v1  
kind: ConfigMap  
metadata:  
name: aws-auth  
namespace: kube-system

data:

mapUsers: |

- userarn: arn:aws:iam::123456789012:user/pavan

username: pavan

## CHECK THE CONFIGURING PERMISSIONS FOR THE USER

If the user is properly configured you should be able to list pods in the Cluster:

$ kubectl get pods --as pavan

You can test the same using the above mentioned --as USERNAME flag or set the pavan as the default profile for the aws cli.

$ export AWS\_PROFILE=pavan

Once configured you can test to see if the user is properly configured using the aws sts get-caller-identity command:

$ aws sts get-caller-identity

You should see a response like the following, indicating the user is properly configured with your aws cli utility:

{

"UserId": "AIDAX7JPBEM4A6FTJRTMB",

"Account": "123456789012",

"Arn": "arn:aws:iam::123456789012:user/pavan"

}

You can use auth can-i to check if you have permission to a resource. To see if you have the permission to get pods simply run:

$ kubectl auth can-i get pods

The answer will be a simple yes or no.

Wanna check if you have cluster-admin permissions?

We can try the following command. $ kubectl auth can-i "\*" "\*"

**Managed node groups :**

Managed Node Groups will automatically scale the EC2 instances using an Auto Scaling Group managed by EKS. This ASG also runs the latest Amazon EKS-optimized Amazon Linux 2 AMI. This is great on one hand — because updates will be applied automatically for you — but if you want control over this you will want to manage your own node groups. Finally, security groups, IAM roles, and connecting them together is handled for you.

The amiFamily field supports only AmazonLinux2

**Drawbacks :**

Greatly reduced user options for instance and node configuration, such as not allowing custom instance user data. In particular, the inability to modify instance user data has the side effect of leaving [customization of the cluster networking](https://docs.aws.amazon.com/eks/latest/userguide/cni-custom-network.html) unsupported for managed node groups and makes it very difficult to install monitoring agents directly on the nodes.

**Self-Managed Node groups :**

Supports Amazon Linux, Ubuntu, or even custom AMIs for node image.

**Drawbacks**

* Node group creation is not as automated as it is for managed node groups. Tools like [eksctl](https://eksctl.io/) can automate much of this work, however.
* Requires manually replacing nodes or migrating to new node groups during node version upgrades. eksctl or other automation can ease this workload.