Lesson 08 Demo 04

Enforcing Kubernetes Policies Using Gatekeeper

Objective: To enforce Kubernetes policies for pod resource limits and namespace creation using the open policy agent (OPA) Gatekeeper

Tools required: kubeadm, kubectl, kubelet, containerd, and OPA Gatekeeper

Prerequisites: A Kubernetes cluster (refer to Demo 01 from Lesson 01 for setting up a cluster)

Steps to be followed:

1. Install OPA Gatekeeper
2. Create a ConstraintTemplate and constraint for the resource limit policy
3. Test the resource limit policy
4. Create a ConstraintTemplate and constraint for the namespace policy
5. Test the namespace policy

Step 1: Install OPA Gatekeeper

1.1 After setting up the kubectl, run the following command to install OPA Gatekeeper:
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/v3.17.1/deploy/gatekeeper.yaml

```
labsuser@master:~$ kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/v3.17.1/deploy/gatekeeper.yaml
namespace/gatekeeper-system created
resourcequota/gatekeeper-critical-pods created
customresourcedefinition.apiextensions.k8s.io/assign.mutations.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/assignimage.mutations.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/assignmetadata.mutations.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/configs.config.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/constraintpodstatuses.status.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/constrainttemplatepodstatuses.status.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/constrainttemplates.templates.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/expansiontemplate.expansion.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/expansiontemplatepodstatuses.status.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/modifyset.mutations.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/mutatorpodstatuses.status.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/providers.externaldata.gatekeeper.sh created
customresourcedefinition.apiextensions.k8s.io/syncsets.syncset.gatekeeper.sh created
serviceaccount/gatekeeper-admin created
role.rbac.authorization.k8s.io/gatekeeper-manager-role created
clusterrole.rbac.authorization.k8s.io/gatekeeper-manager-role created
rolebinding.rbac.authorization.k8s.io/gatekeeper-manager-rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/gatekeeper-manager-rolebinding created
secret/gatekeeper-webhook-server-cert created
service/gatekeeper-webhook-service created
deployment.apps/gatekeeper-audit created
deployment.apps/gatekeeper-controller-manager created
poddisruptionbudget.policy/gatekeeper-controller-manager created
mutatingwebhookconfiguration.admissionregistration.k8s.io/gatekeeper-mutating-webhook-configuration created
validatingwebhookconfiguration.admissionregistration.k8s.io/gatekeeper-validating-webhook-configuration created
```

Note: For further information on the installation of OPA Gatekeeper, refer to the following official documentation:
https://open-policy-agent.github.io/gatekeeper/website/docs/install/#installation

1.2 Run the following command to list the running namespaces:
kubectl get ns

```
labsuser@master:~$ kubectl get ns
NAME                STATUS   AGE
default             Active   12h
gatekeeper-system   Active   9m12s
kube-node-lease     Active   12h
kube-public         Active   12h
kube-system         Active   12h
react-app           Active   12h
trivy-temp          Active   9h
labsuser@master:~$
```

1.3 Run the following command to list the pods running within the **gatekeeper-system** namespace:
kubectl get pods -n gatekeeper-system

```
labsuser@master:~$ kubectl get pods -n gatekeeper-system
NAME                                             READY   STATUS    RESTARTS      AGE
gatekeeper-audit-c794d5f69-v7cxf                 1/1     Running   2 (11m ago)   11m
gatekeeper-controller-manager-865cc64485-bjpjv   1/1     Running   0             11m
gatekeeper-controller-manager-865cc64485-mqbnc   1/1     Running   0             11m
gatekeeper-controller-manager-865cc64485-vx9hp   1/1     Running   0             11m
labsuser@master:~$
```

This verifies the successful installation of the OPA Gatekeeper.

Step 2: Create a ConstraintTemplate and constraint for the resource limit policy

2.1 Create a file named **resourcequota-template.yaml** using the following command:
vi resourcequota-template.yaml

```
labsuser@master:~$ vi resourcequota-template.yaml
labsuser@master:~$
```

2.2 Add the following YAML configuration in the **resourcequota-template.yaml** file that defines a policy to check both CPU and memory limits for pod creation:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: resourcelimits
spec:
  crd:
    spec:
      names:
        kind: ResourceLimits
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package resourcelimits

        violation[{"msg": msg}] {
          input.review.object.kind == "Pod"
          limits := input.review.object.spec.containers[_].resources.limits

          # Check CPU limit
          limits.cpu > "500m"
          msg := sprintf("CPU limit exceeds the allowed maximum: %v", [limits.cpu])
        }

        violation[{"msg": msg}] {
          input.review.object.kind == "Pod"
          limits := input.review.object.spec.containers[_].resources.limits

          # Check memory limit
          limits.memory > "256Mi"
          msg := sprintf("Memory limit exceeds the allowed maximum: %v",
[limits.memory])
        }
```

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: resourcelimits
spec:
  crd:
    spec:
      names:
        kind: ResourceLimits
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package resourcelimits

        violation[{"msg": msg}] {
          input.review.object.kind == "Pod"
          limits := input.review.object.spec.containers[_].resources.limits

          # Check CPU limit
          limits.cpu > "500m"
          msg := sprintf("CPU limit exceeds the allowed maximum: %v", [limits.cpu])
        }
:wq
```

This ConstraintTemplate enforces a maximum CPU limit of 500m and a memory limit of 256Mi for any pod created in the cluster.

2.3 Execute the following command to apply the ConstraintTemplate:
kubectl apply -f resourcequota-template.yaml

```
labsuser@master:~$ kubectl apply -f resourcequota-template.yaml
constrainttemplate.templates.gatekeeper.sh/resourcelimits created
labsuser@master:~$
```

2.4 Create a file named **resourcequota-constraint.yaml** using the following command:
vi resourcequota-constraint.yaml

```
labsuser@master:~$ vi resourcequota-constraint.yaml
labsuser@master:~$
```

2.5 Add the following YAML configuration in the **resourcequota-constraint.yaml** file to enforce resource limits policy on pods:

apiVersion: constraints.gatekeeper.sh/v1beta1
kind: ResourceLimits
metadata:
name: pod-resource-limits
spec:
match:
kinds:

- apiGroups: [""]
kinds: ["Pod"]

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: ResourceLimits
metadata:
  name: pod-resource-limits
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
~
~
~
~
~
~
~
~
```

This constraint ensures that the rules defined in the ConstraintTemplate are applied to all pod objects in the cluster.

2.6 Run the following command to apply the constraint:
kubectl apply -f resourcequota-constraint.yaml

```
labsuser@master:~$ kubectl apply -f resourcequota-constraint.yaml
resourcelimits.constraints.gatekeeper.sh/pod-resource-limits created
labsuser@master:~$
```

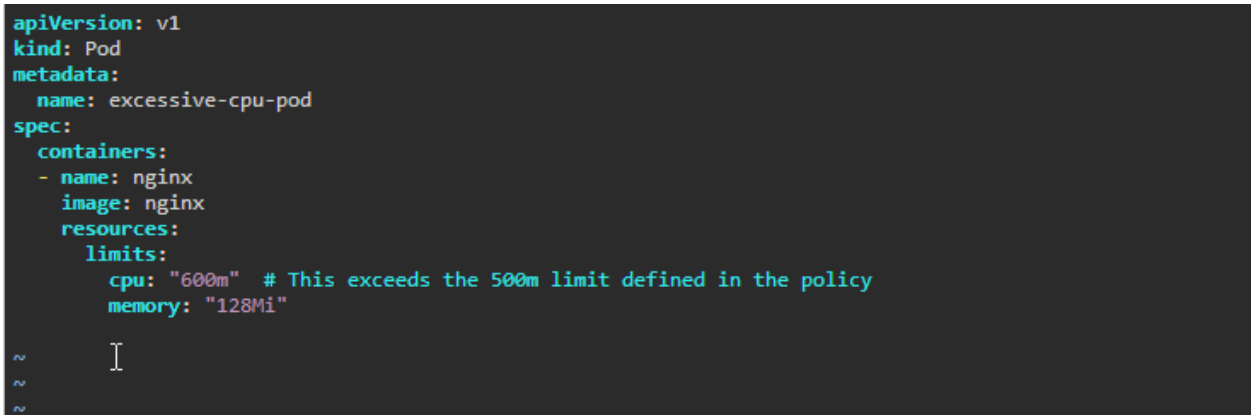
Step 3: Test the resource limit policy

3.1 Create a YAML file named **excessive-pod.yaml** using the following command:
vi excessive-pod.yaml

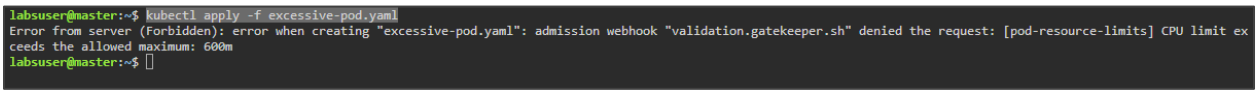
```
labsuser@master:~$ vi excessive-pod.yaml
```

3.2 Add the following YAML configuration in the **excessive-pod.yaml** file to define the pod:

```
apiVersion: v1
kind: Pod
metadata:
 name: excessive-cpu-pod
spec:
 containers:
 - name: nginx
  image: nginx
  resources:
   limits:
    cpu: "600m"  # This exceeds the 500m limit defined in the policy
    memory: "128Mi"
```



3.3 Try to apply the pod configuration above using the following command:
kubectl apply -f excessive-pod.yaml



You will get an error indicating that the pod creation is denied.

3.4 Run the **vi valid-pod.yaml** command to create a pod configuration file and add the following YAML configurations to the file:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: valid-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        cpu: "400m"  # This is within the allowed limit
        memory: "128Mi"  # This is within the allowed limit
```

```
labsuser@master:~$ vi valid-pod.yaml
labsuser@master:~$
```

```
apiVersion: v1
kind: Pod
metadata:
  name: valid-pod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        cpu: "400m"  # This is within the allowed limit
        memory: "128Mi"  # This is within the allowed limit

~
```

3.5 Execute the following command to apply the pod configuration:

kubectl apply -f valid-pod.yaml

```
labsuser@master:~$ kubectl apply -f valid-pod.yaml
pod/valid-pod created
labsuser@master:~$
```

This pod is created successfully.

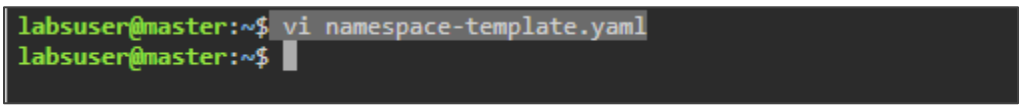
Step 4: Create a ConstraintTemplate and constraint for the namespace policy

4.1 Run the **vi namespace-template.yaml** command to create a ConstraintTemplate and add the following YAML configurations to the file to define the policy logic:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: musthaveprefix
spec:
  crd:
    spec:
      names:
        kind: MustHavePrefix
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package musthaveprefix

        violation[{"msg": msg}] {
          input.review.object.kind == "Namespace"
          name := input.review.object.metadata.name

          # Check if the namespace starts with "prod-"
          not startswith(name, "prod-")
          msg := sprintf("Namespace name %v must start with 'prod-'", [name])
        }
```



```yaml
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: musthaveprefix
spec:
  crd:
    spec:
      names:
        kind: MustHavePrefix
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package musthaveprefix

        violation[{"msg": msg}] {
          input.review.object.kind == "Namespace"
          name := input.review.object.metadata.name

          # Check if the namespace starts with "prod-"
          not startswith(name, "prod-")
          msg := sprintf("Namespace name %v must start with 'prod-'", [name])
        }
```

The policy checks the name of the namespace being created. If the name does not start with the prefix **prod-**, the request to create the namespace is denied and an appropriate error message is returned.

4.2 Run the following command to apply the ConstraintTemplate:
kubectl apply -f namespace-template.yaml

```
labsuser@master:~$ kubectl apply -f namespace-template.yaml
constrainttemplate.templates.gatekeeper.sh/musthaveprefix created
labsuser@master:~$
```

4.3 Run the **vi namespace-constraint.yaml** command to create the constraint and add the following YAML configurations to the file to enforce the policy:

apiVersion: constraints.gatekeeper.sh/v1beta1
kind: MustHavePrefix
metadata:
name: require-prod-prefix
spec:
match:
kinds:
- apiGroups: [""]
kinds: ["Namespace"]

```
labsuser@master:~$ vi namespace-constraint.yaml
labsuser@master:~$
```

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: MustHavePrefix
metadata:
  name: require-prod-prefix
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
~
```

4.4 Run the following command to apply the constraint:

kubectl apply -f namespace-constraint.yaml

```
labsuser@master:~$ kubectl apply -f namespace-constraint.yaml
musthaveprefix.constraints.gatekeeper.sh/require-prod-prefix created
labsuser@master:~$
```

Step 5: Test the namespace policy

5.1 Create a file called **invalid-namespace.yaml** and add the following YAML configurations:

apiVersion: v1

kind: Namespace

metadata:

name: test-environment # Does not start with "prod-", so it should be rejected

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-environment  # Does not start with "prod-", so it should be rejected
~
~
~
~
~
```

5.2 Apply the namespace using the following command:

kubectl apply -f invalid-namespace.yaml

```
labsuser@master:~$ kubectl apply -f invalid-namespace.yaml
Error from server (Forbidden): error when creating "invalid-namespace.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [require-prod-prefix] Namespac
e name test-environment must start with 'prod-'
labsuser@master:~$
```

You will see an error message, indicating that the namespace was rejected because it does not meet the required naming convention.

5.3 Create a file called **valid-namespace.yaml** and add the following YAML configurations:

apiVersion: v1
kind: Namespace
metadata:
name: prod-environment # This starts with "prod-", so it should be accepted

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod-environment  # This starts with "prod-", so it should be accepted

~
~
~
~
~
~
```

5.4 Apply the namespace using the following command:

kubectl apply -f valid-namespace.yaml

```
labsuser@master:~$ kubectl apply -f valid-namespace.yaml
namespace/prod-environment created
labsuser@master:~$
```

This namespace is successfully created.

By following these steps, you have successfully created and enforced Kubernetes policies for pod resource limits and namespace creation using the open policy agent (OPA) Gatekeeper tool to enhance cluster governance and compliance.