OPA Gatekeeper on Kubernetes

Use Case: Enforcing Trusted Container Image Registries

Overview

This Proof of Concept (POC) demonstrates how to use **OPA Gatekeeper** to enforce **policy-as-code** in Kubernetes by restricting Pods to pull container images only from approved registries.

The policy is enforced at **admission time**, meaning insecure workloads are blocked before they are persisted to **`etcd`**.

Objective

- Prevent Kubernetes Pods from using images from untrusted container registries**.**

- Enable platform and DevOps teams to automatically control which container images are allowed in Kubernetes, without relying on manual reviews or developer discipline**.**

From a Developer's Perspective

- Give developers clear and immediate feedback when they try to deploy an unapproved image.
- Reduce confusion by providing human-readable policy error messages.
- Allow developers to focus on coding while security rules are enforced automatically.

From a Security Team's Perspective

- Ensure only verified and approved container images enter the cluster
- Reduce the attack surface caused by public or unknown images
- Standardize image usage across all teams and environments

From a DevOps / Platform Engineer's Perspective

- Enforce security policies **centrally**, without modifying application code
- Avoid repetitive manual checks during deployments
- Maintain **consistent security enforcement** across all namespaces and teams

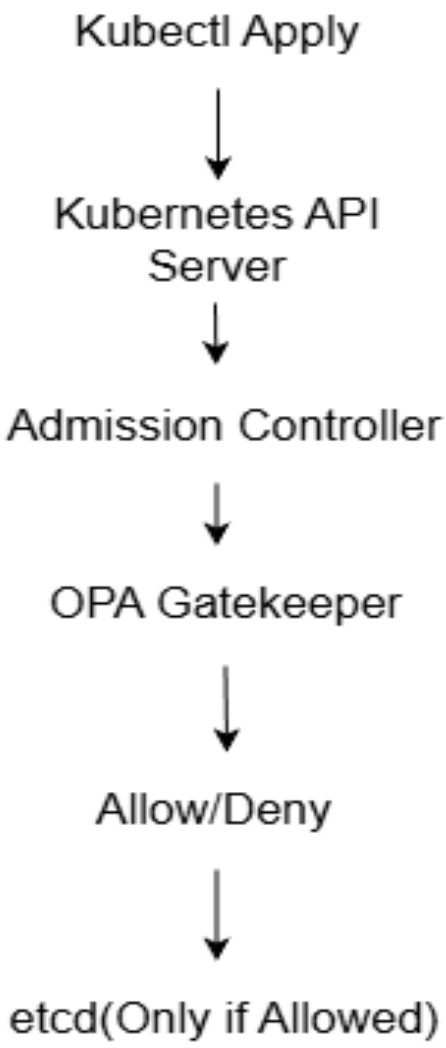
From an Organization's Perspective

- Align Kubernetes workloads with **organizational compliance policies**
- Reduce risks related to supply-chain attacks
- Enable scalable governance as the number of teams and services grows

Why this is important:

- Prevents deployment of insecure or unknown images
- Enforces organizational security standards automatically
- Shifts security left (before runtime)
- Demonstrates real-world DevSecOps practices

Architecture & Flow



Gatekeeper receives an **AdmissionReview** request from the API Server and evaluates it using Rego policies.

Step by Step Implementation

STEP 1 : Start Kubernetes Cluster

minikube start

Why:
OPA Gatekeeper works as a Kubernetes admission controller and requires a running cluster to intercept API requests.

Verify:

kubectl get nodes

STEP 2 : Install OPA Gatekeeper

kubectl apply -f
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml

Why:
Gatekeeper integrates OPA with Kubernetes by registering a validating admission webhook.

Verify:

kubectl get pods -n gatekeeper-system
kubectl get validatingwebhookconfigurations

STEP 3: Create ConstraintTemplate (Policy Logic)

`trusted-registry-template.yaml`

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8strustedregistry
spec:
  crd:
    spec:
      names:
        kind: K8sTrustedRegistry
      validation:
        openAPIV3Schema:
          type: object
          properties:
            registry:
              type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8strustedregistry

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not startswith(container.image, input.parameters.registry)
          msg := sprintf("Untrusted image used: %s", [container.image])
        }
```

Apply:

kubectl apply -f trusted-registry-template.yaml

Verify:

kubectl get constrainttemplates

Why:

- Defines the Rego policy logic
- Creates a custom CRD dynamically
- OpenAPI schema is mandatory for strict parameter validation

STEP 4: Create Constraint (Policy Enforcement)

`Trusted-registry-constraint.yaml`

```
apiVersion:
constraints.gatekeeper.sh/v1beta1
kind: K8sTrustedRegistry
metadata:
  name: allow-only-approved-registry
spec:
  parameters:
    registry: "gcr.io/approved/"
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

Apply:

kubectl apply -f trusted-registry-constraint.yaml

Verify:

kubectl get constraints

Why:

- Instantiates the template
- Defines scope (Pods)
- Supplies dynamic parameters

STEP 5: Test DENY Case

`invalid-pod.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-invalid
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

Apply:

kubectl apply -f invalid-pod.yaml

Expected:

admission webhook "validation.gatekeeper.sh" denied the request:
Untrusted image used: nginx:latest

Why:
Confirms that untrusted images are blocked at admission time.

STEP 6: Test ALLOW Case

`valid-pod.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-valid
spec:
  containers:
  - name: nginx
    image: gcr.io/approved/nginx:1.25
```

Apply:

kubectl apply -f valid-pod.yaml

Expected:

pod/nginx-valid created

Why:
Ensures valid workloads are not blocked by the policy.