KPLABS Course

Certified Kubernetes Security Specialist

Minimize Microservice Vulnerabilities

ISSUED BY

Zeal

REPRESENTATIVE

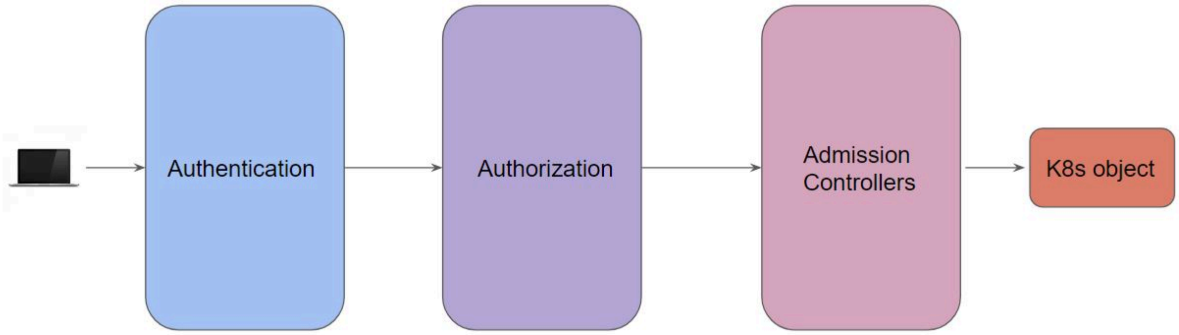
instructors@kplabs.in

Module 1: Admission Controllers

1.1 Overview of Admission Controllers

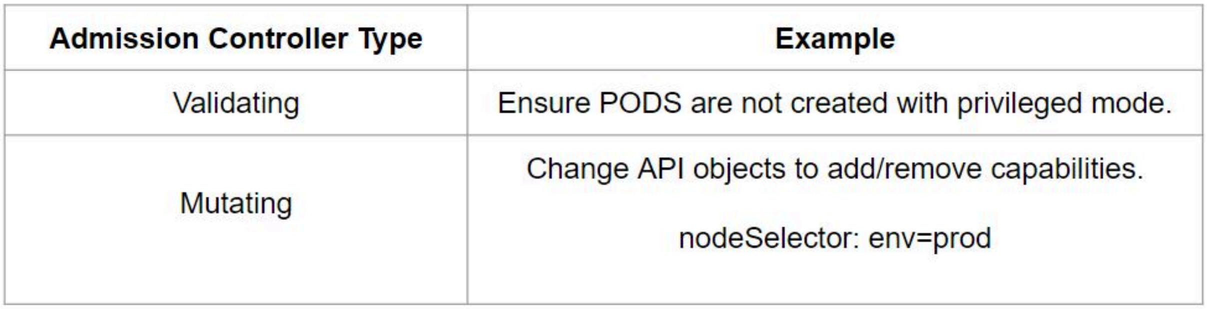
An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized.

Controllers that can intercept Kubernetes API requests, and modify or reject them based on custom logic.



1.2 Types of Admission Controllers

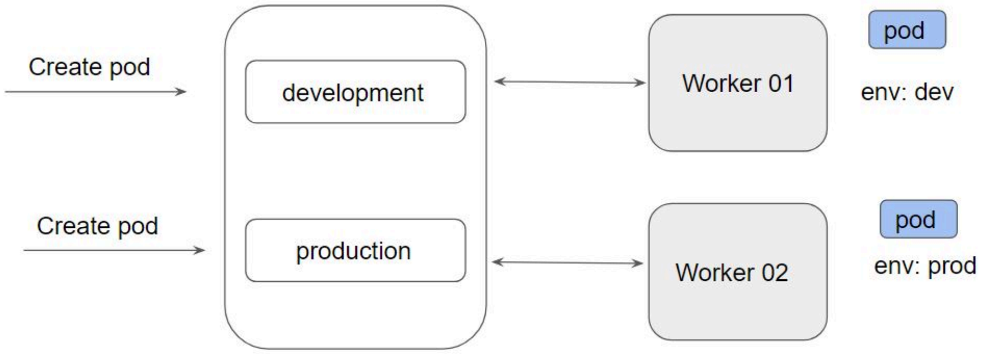
There are two primary types of Admission Controllers



Admission Controller Type	Example
Validating	Ensure PODS are not created with privileged mode.
Mutating	Change API objects to add/remove capabilities. nodeSelector: env=prod

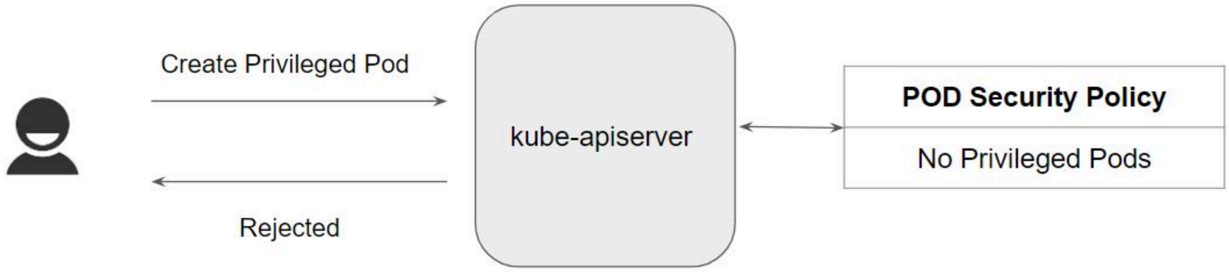
1.3 Use Case 1 - PodNodeSelector

This admission controller defaults and limits what node selectors may be used within a namespace by reading a namespace annotation and a global configuration.



1.4 Use Case 2 - PodSecurityPolicy

The PodSecurityPolicy defines a set of conditions that Pods must meet to be accepted by the cluster.



1.5 Multiple Admission Controllers

There are multiple admission controllers available, each with a specific set of capabilities.

Some of these include:

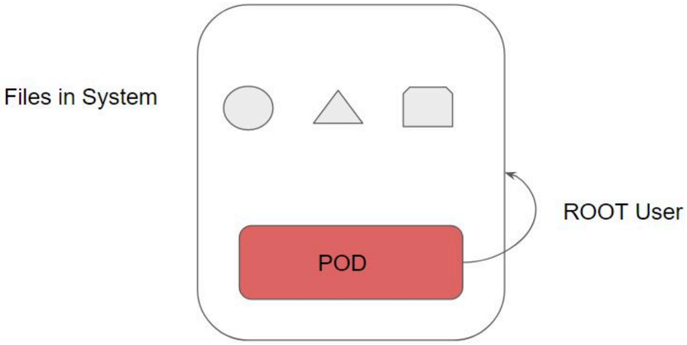
- AlwaysPullImages
- PodNodeSelector
- PodSecurityPolicy
- LimitRanger
- NamespaceExists
- EventRateLimit

Module 2: Security Context

2.1 Understanding the Challenge

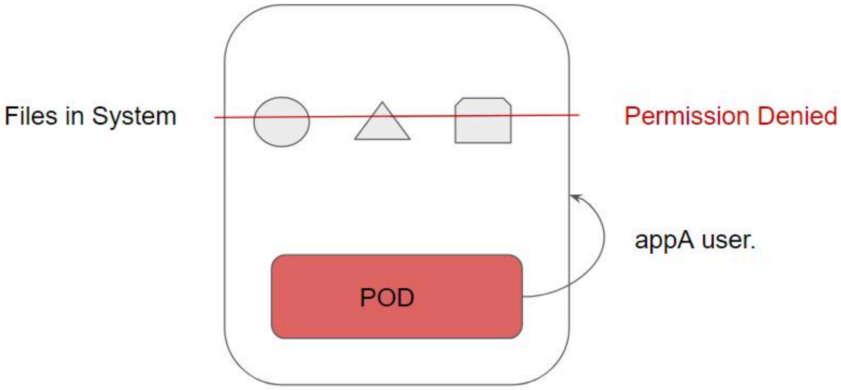
When you run a container, it runs with the UID 0 (Administrative Privilege)

In-case of container breakouts, an attacker can get root privileges to your entire system.

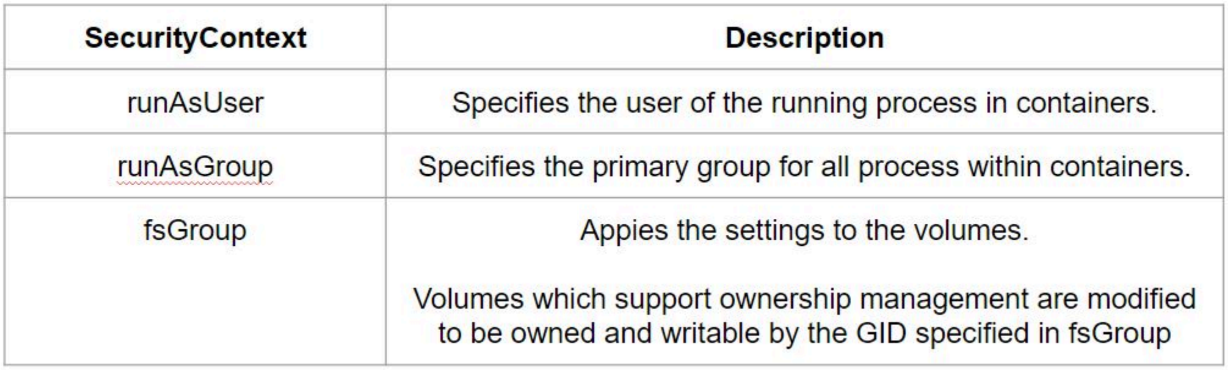


2.2 Running with Least Privilege User

We can run POD and container with limited privilege user instead of the ROOT user.



2.3 Three Important Configuration Parameters



SecurityContext	Description
runAsUser	Specifies the user of the running process in containers.
runAsGroup	Specifies the primary group for all process within containers.
fsGroup	Appies the settings to the volumes. Volumes which support ownership management are modified to be owned and writable by the GID specified in fsGroup

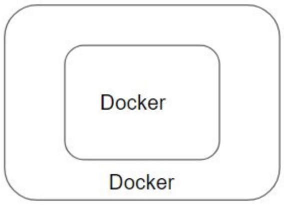
Module 3: Privileged container

3.1 Understanding the Challenge

By default, the docker container does not have many capabilities assigned to it.

Docker containers are also not allowed to access any devices.

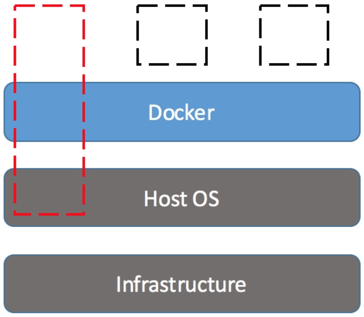
Hence, by default, docker container cannot perform various use-cases like run docker container inside a docker container



3.2 Privileged Containers

Privileged Container can access all the devices on the host as well as have a configuration in AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host.

- Limits that you set for privileged containers will not be followed.
- Running a privileged flag gives all the capabilities to the container



Module 4: Hack Misconfigured Cluster

4.1 Understanding the Use-Case

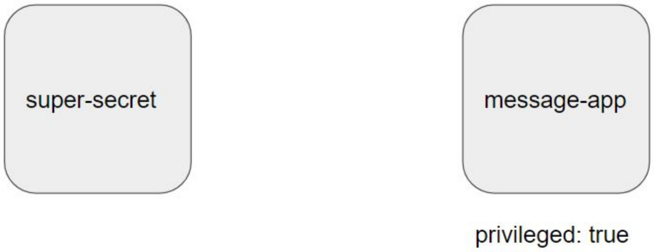
Let us assume that a developer named John is given a create and manage access on the POD resource in the namespace of development.

John

4.2 Resource Created by John

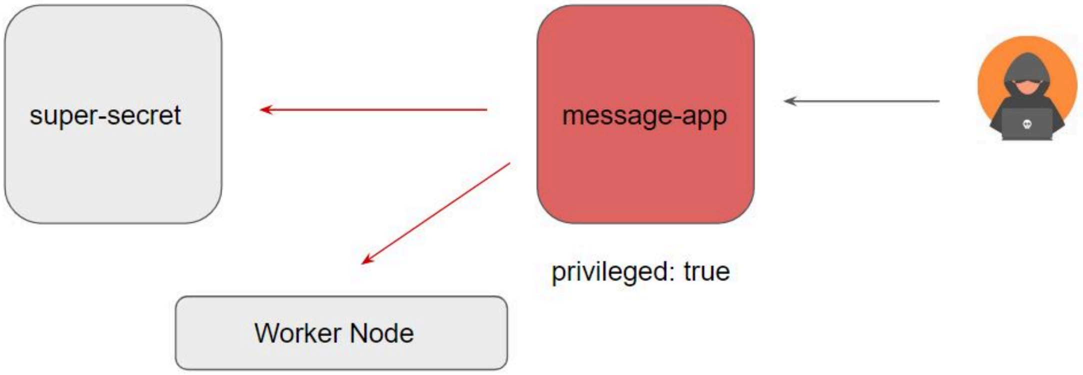
John has created two PODS according to following configuration.

Super-secret app is storing sensitive data temporarily in its file system.



4.3 Possible Implications

Mr X is a hacker and he managed to get access to the messaging-app due to some application vulnerability.

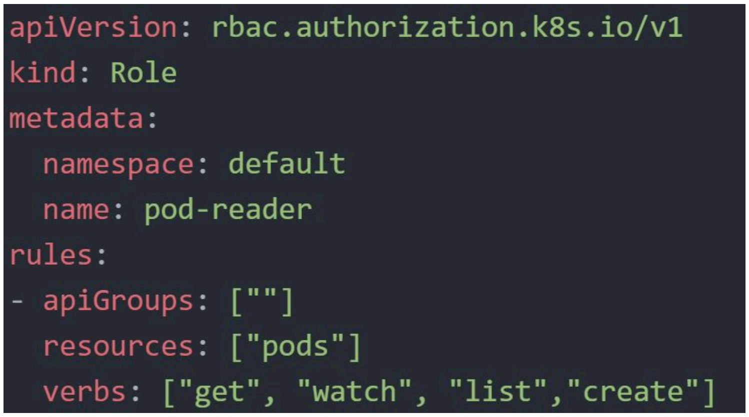


Module 5: POD Security Policies

5.1 Understanding the Challenge

If you have access to create a POD, you can create a POD with a wide variety of configurations like privileged containers, HostPID, and others.

A simple permission of allowing user to create a POD can lead to a breach of your environment.

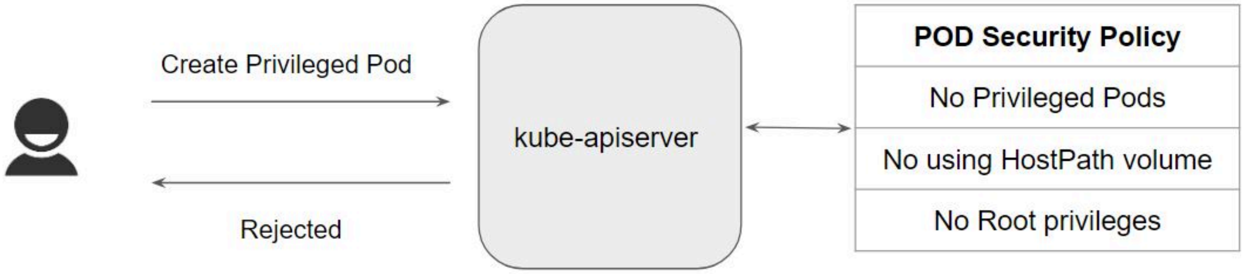


```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list","create"]
```

5.2 Understanding POD Security Policy

The PodSecurityPolicy defines a set of conditions that Pods must meet to be accepted by the cluster.

When a request to create or update a Pod does not meet the conditions in the PodSecurityPolicy, that request is rejected and an error is returned.



POD Security Policy
No Privileged Pods
No using HostPath volume
No Root privileges

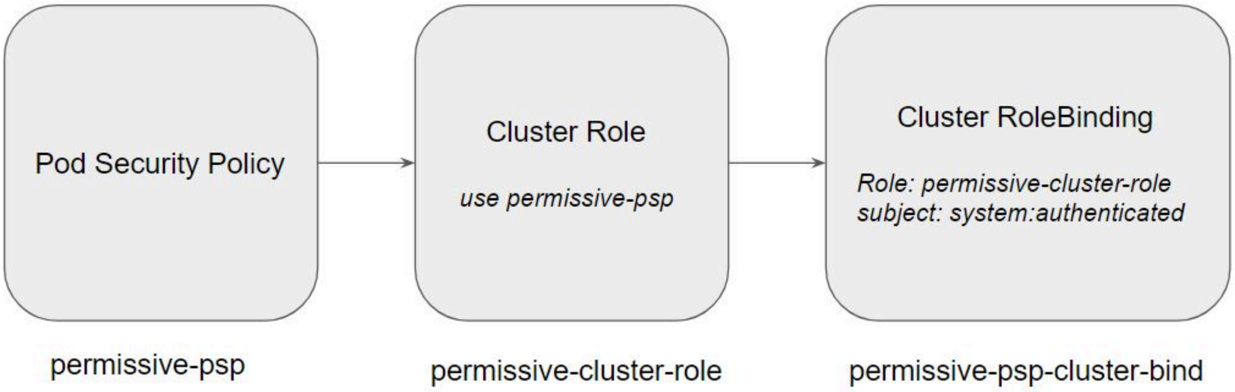
5.3 PSP Control

POD Security Policies allows an administrator to control various areas, some includes:

- Running of privileged containers
- Usage of host namespaces
- Usage of volume types
- Allocating an FSGroup that owns the pod's volumes
- Linux capabilities

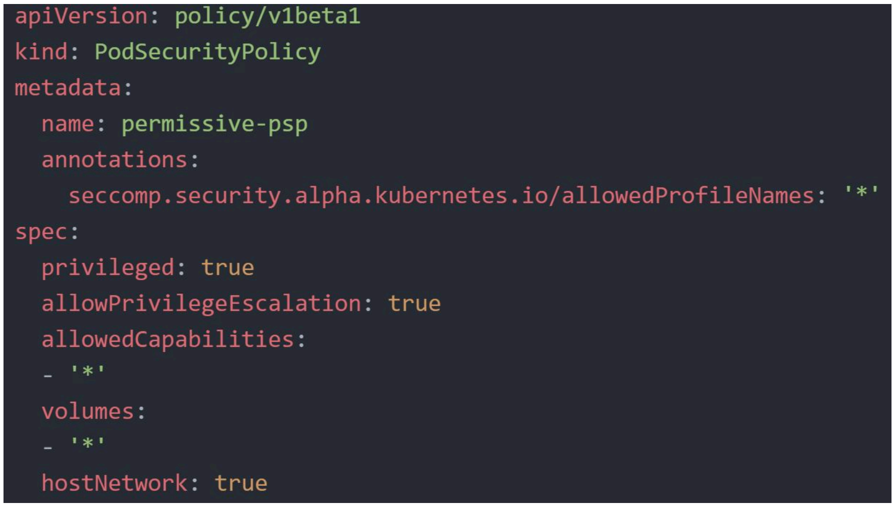
Module 6: PSP Workflow

The following diagram depicts the workflow



Step 1 - Create a Pod Security Policy

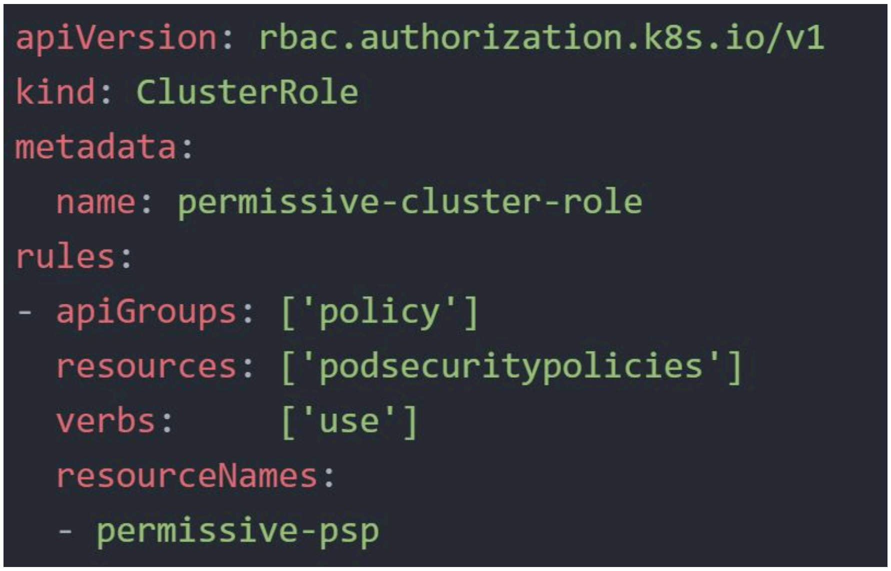
In this step, we create a PSP according to our requirements.



```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive-psp
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
```

Step 2- Create a Cluster Role

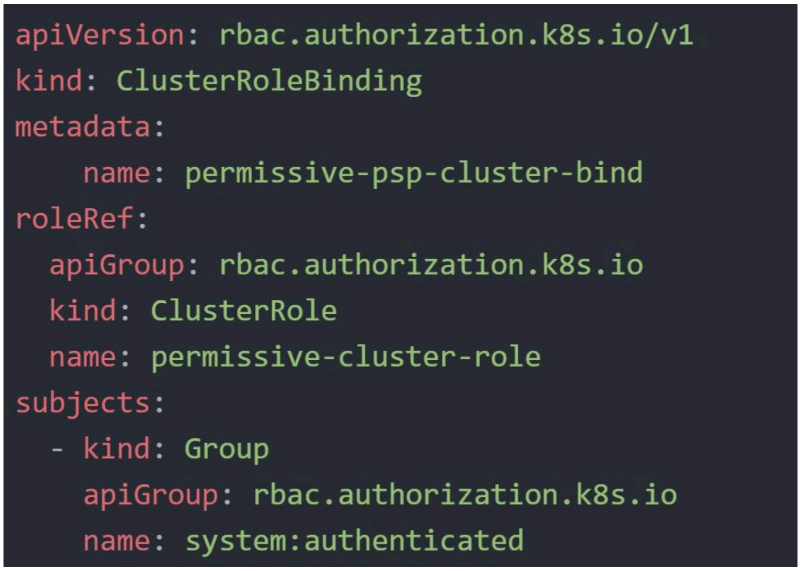
Create a Role that references to the created pod security policy.



```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: permissive-cluster-role
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames:
  - permissive-psp
```

Step 3 - Create Cluster Role Binding

Reference the Cluster Role to the appropriate Subjects.



```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
    name: permissive-psp-cluster-bind
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: permissive-cluster-role
subjects:
  - kind: Group
    apiGroup: rbac.authorization.k8s.io
    name: system:authenticated
```

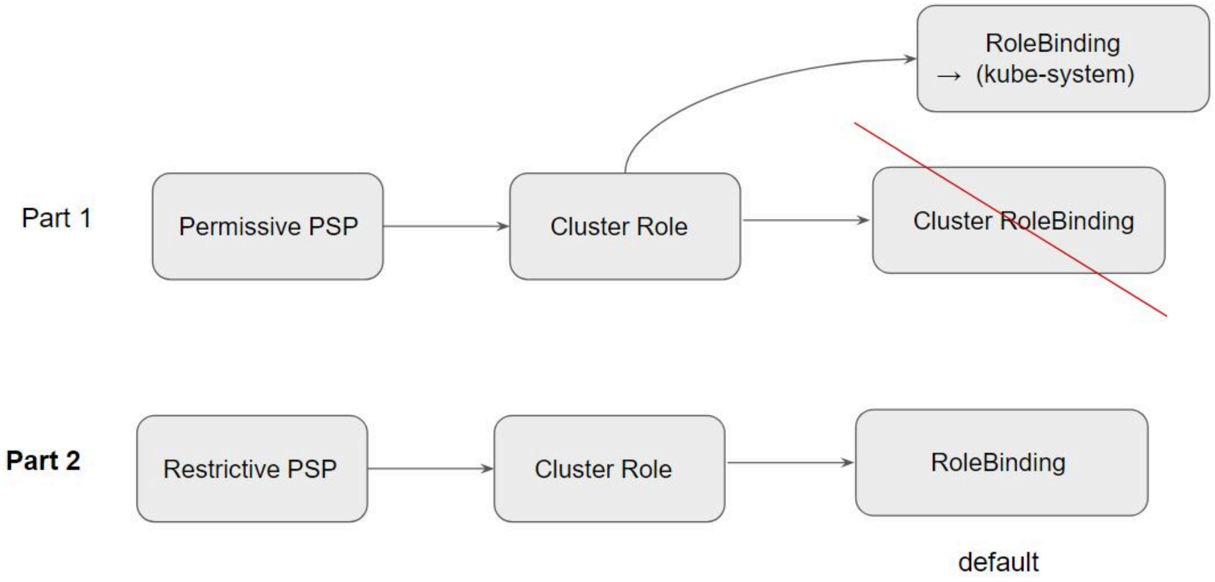
Important Note

PodSecurityPolicies are enforced by enabling the admission controller, but doing so without authorizing any policies will prevent any pods from being created in the cluster.

It is recommended that policies are added and authorized before enabling the admission controller.

Module 7: Implementing Restrictive PSP

The following diagram represents the planned implementation of PSP that we take up in the video.

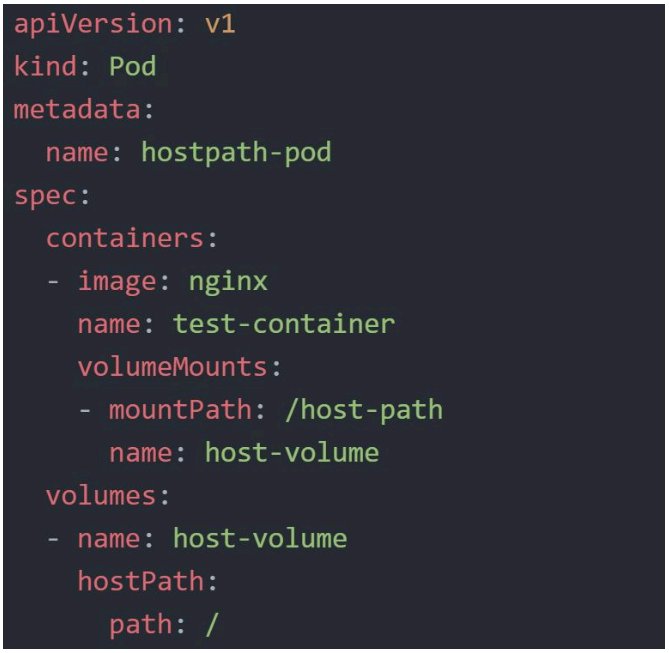


.

Module 8: PSP Security Consideration - Volumes

8.1 Overview of HostPath Volume

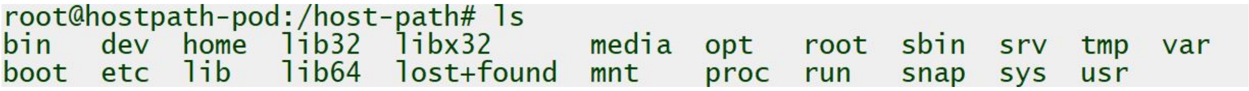
A hostPath volume mounts a file or directory from the host node's filesystem into your Pod



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  containers:
  - image: nginx
    name: test-container
    volumeMounts:
    - mountPath: /host-path
      name: host-volume
  volumes:
  - name: host-volume
    hostPath:
      path: /
```

8.2 Security Consideration

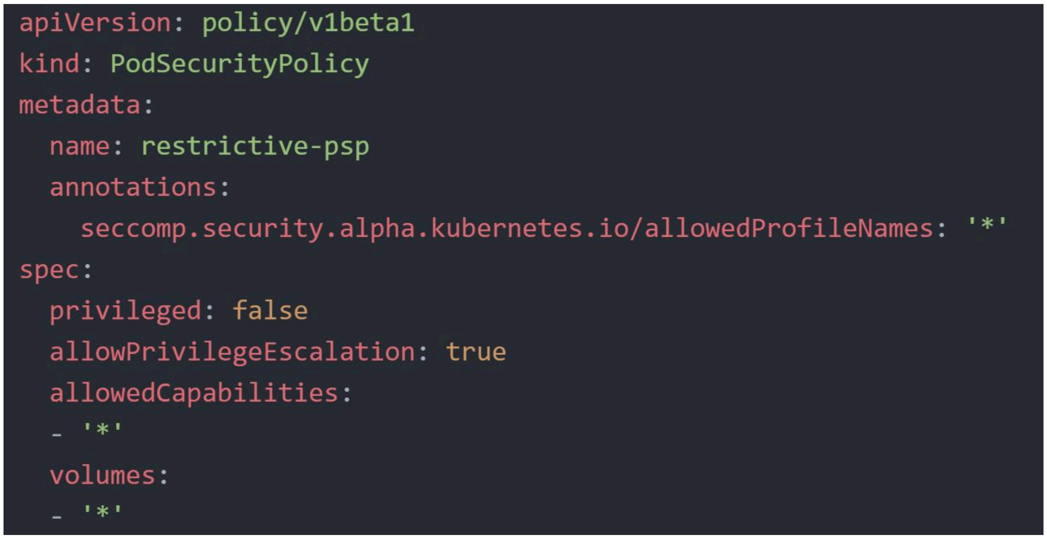
With HostPath based volume, we can mount the entire ROOT (/) inside a container and can access all the files of the host.



```
root@hostpath-pod:/host-path# ls
bin   dev   home  lib32  libx32      media  opt   root  sbin  srv  tmp  var
boot  etc   lib   lib64  lost+found  mnt    proc  run   snap  sys  usr
```

8.3 Our Current Restrictive PSP

Our current PSP is allowing all of the volume types with *



```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restrictive-psp
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: false
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
```

8.4 Recommended Minimals

The recommended minimum set of allowed volumes for new PSPs are:

- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- secret
- projected

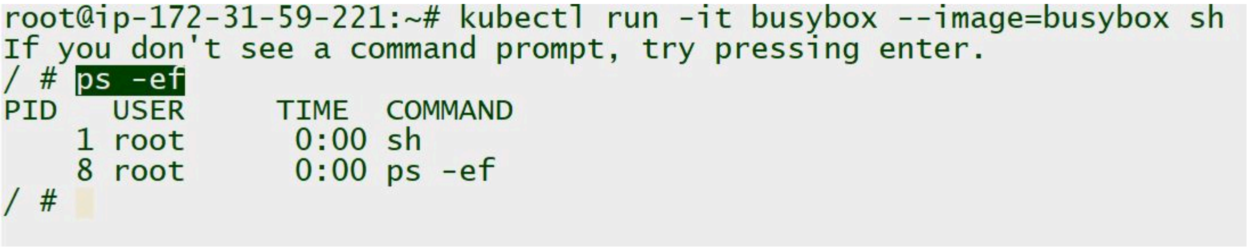
.

Module 9: PSP Security Consideration - Host PID

9.1 Importance of PID Namespace

The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces

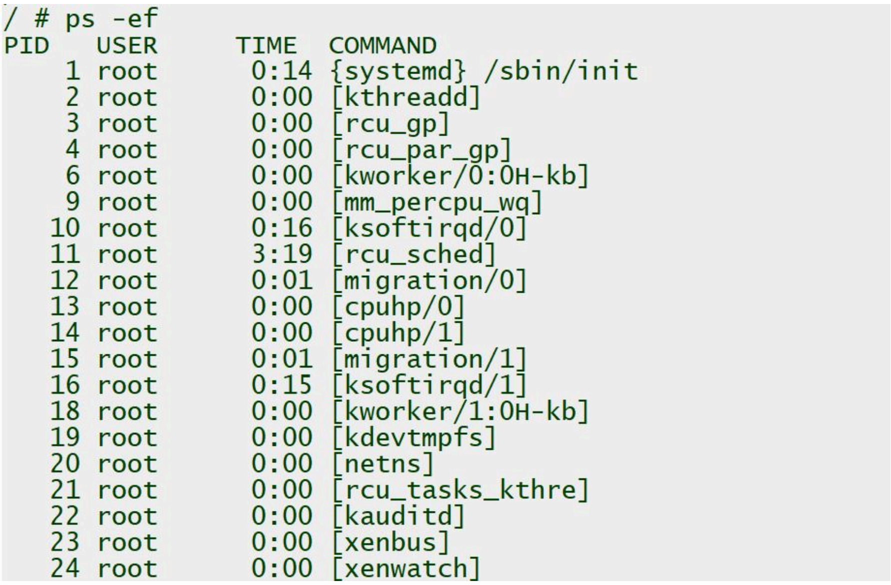
With PID namespace isolation, processes in the child namespace have no way of knowing of the parent process's existence.



```
root@ip-172-31-59-221:~# kubectl run -it busybox --image=busybox sh
If you don't see a command prompt, try pressing enter.
/ # ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    8 root      0:00 ps -ef
/ #
```

9.2 What if Host PID is used?

Allows using of the hosts PID namespace

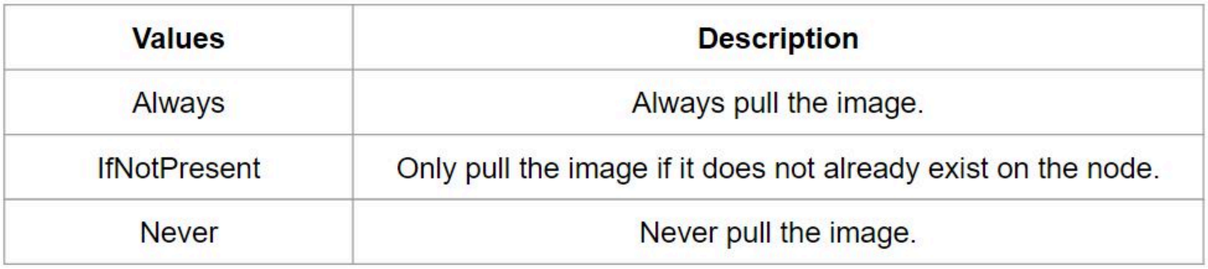


```
/ # ps -ef
PID   USER     TIME  COMMAND
    1 root      0:14 {systemd} /sbin/init
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    6 root      0:00 [kworker/0:0H-kb]
    9 root      0:00 [mm_percpu_wq]
   10 root      0:16 [ksoftirqd/0]
   11 root      3:19 [rcu_sched]
   12 root      0:01 [migration/0]
   13 root      0:00 [cpuhp/0]
   14 root      0:00 [cpuhp/1]
   15 root      0:01 [migration/1]
   16 root      0:15 [ksoftirqd/1]
   18 root      0:00 [kworker/1:0H-kb]
   19 root      0:00 [kdevtmpfs]
   20 root      0:00 [netns]
   21 root      0:00 [rcu_tasks_kthre]
   22 root      0:00 [kauditd]
   23 root      0:00 [xenbus]
   24 root      0:00 [xenwatch]
```

Module 10: Image Pull Policy

When Kubernetes creates containers, it uses the container’s imagePullPolicy to determine if the image should be pulled prior to starting the container.

There are three primary image pull policy values



Values	Description
Always	Always pull the image.
IfNotPresent	Only pull the image if it does not already exist on the node.
Never	Never pull the image.

Important Note:

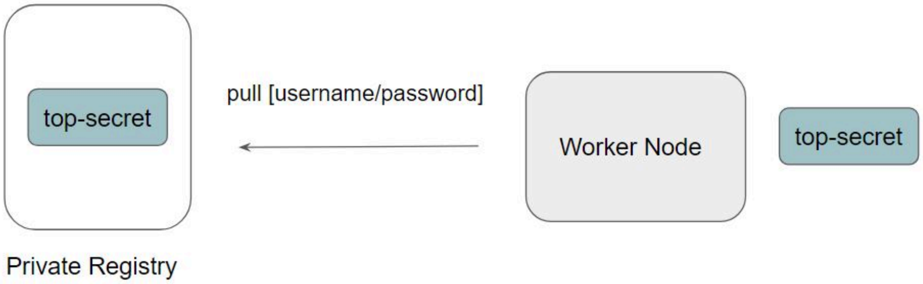
By default, the kubelet tries to pull each image from the specified registry.

However, if the imagePullPolicy property of the container is set to IfNotPresent or Never, then a local image is used (preferentially or exclusively, respectively).

Module 11: Admission Controller - AlwaysPullImages

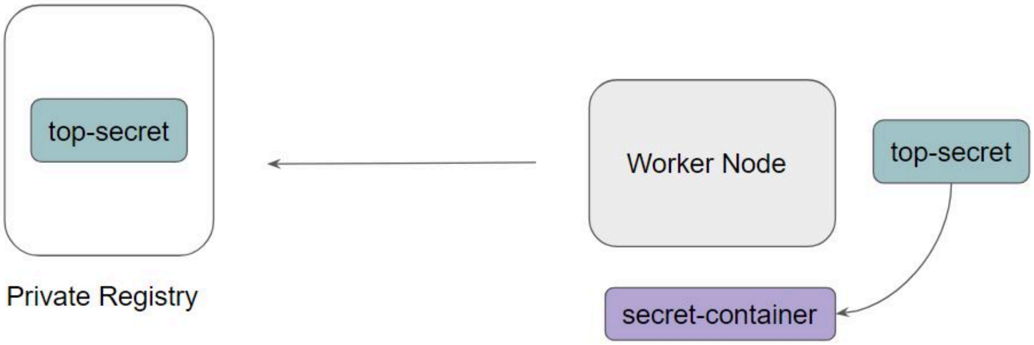
11.1 Understanding the Challenge

SysOps Team Member A has pulled a top-secret image from the private registry after supplying the required level of authentication.



A smart developer B decided to create a pod with the top-secret image.

He made use of ImagePullPolicy to Never



11.2 Overview of AlwaysPullImages

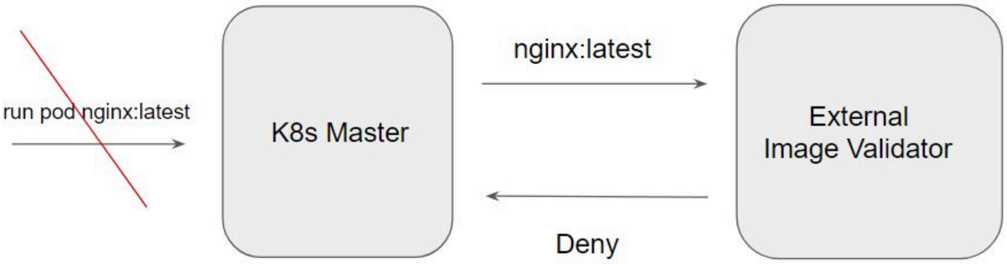
This admission controller modifies every new Pod to force the image pull policy to Always.

This is useful in a multitenant cluster so that users can be assured that their private images can only be used by those who have the credentials to pull them.

Without this admission controller, once an image has been pulled to a node, any pod from any user can use it simply by knowing the image's name without any authorization check against the image.

Module 12: ImagePolicyWebhook

The ImagePolicyWebhook admission controller allows a backend webhook to make admission decisions.



Following are the implementation steps:

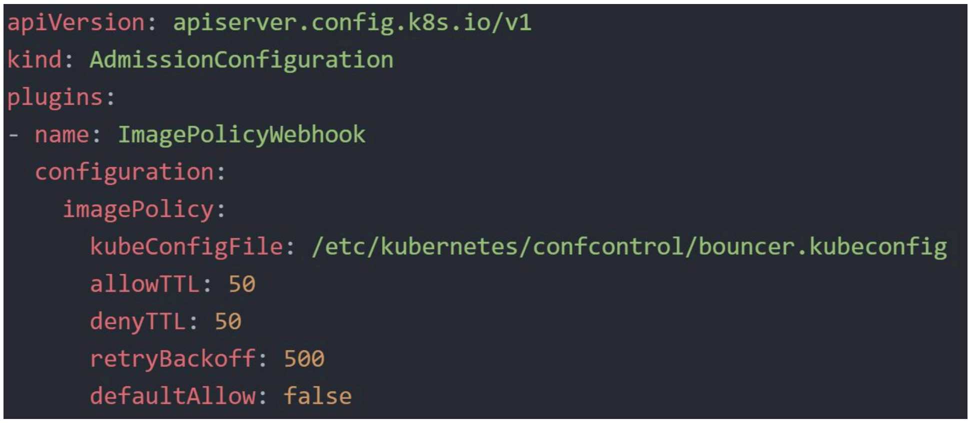
Step 1: Create Configuration File.
Step 2: Create Kubeconfig file.
Step 3: Mount the above files to kube-apiserver (hostpath)
Step 4: Add ImagePolicyWebook within kube-apiserver manifest.

Let us understand each step in more detail.

Step 1 - Create Configuration File

defaultAllow determines whether image would be allowed if external image validator is not reachable.

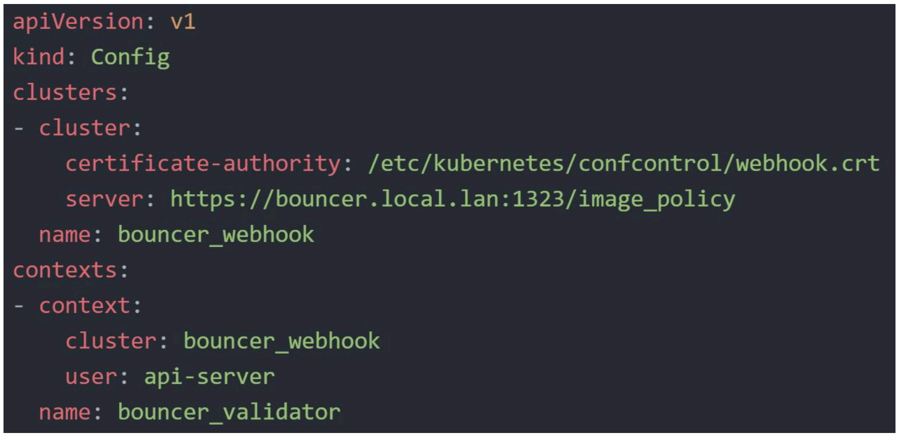
kubeConfigFile location is second important paramter.



```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: ImagePolicyWebhook
  configuration:
    imagePolicy:
      kubeConfigFile: /etc/kubernetes/confcontrol/bouncer.kubeconfig
      allowTTL: 50
      denyTTL: 50
      retryBackoff: 500
      defaultAllow: false
```

Step 2 - Create KubeConfig

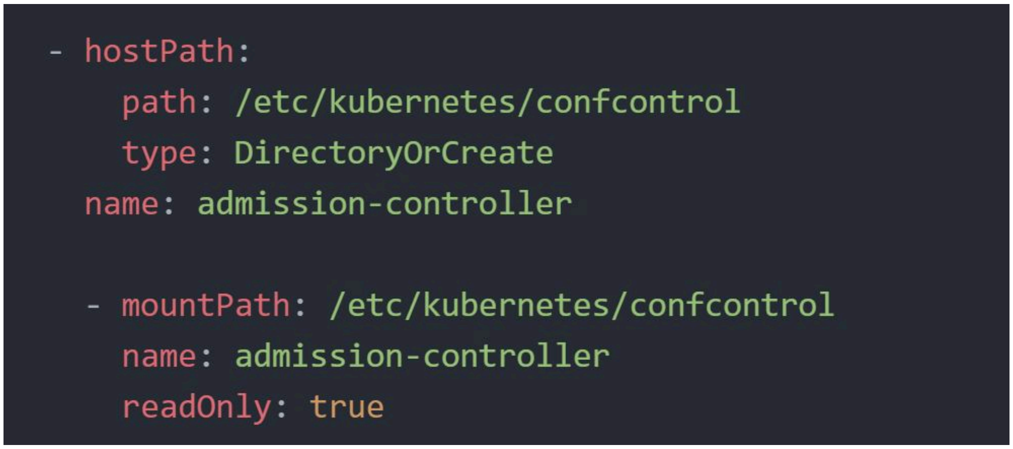
The kubeconfig file's cluster field must point to the remote service, and the user field must contain the returned authorizer.



```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/confcontrol/webhook.crt
    server: https://bouncer.local.lan:1323/image_policy
  name: bouncer_webhook
contexts:
- context:
    cluster: bouncer_webhook
    user: api-server
  name: bouncer_validator
```

Step 3 - Mount Volumes

We mount the directory that contains the configuration file, kubeconfig file, and other data.

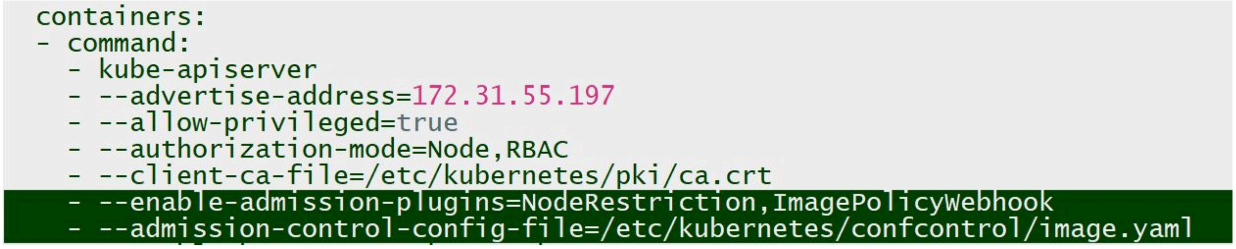


```
- hostPath:
    path: /etc/kubernetes/confcontrol
    type: DirectoryOrCreate
  name: admission-controller

  - mountPath: /etc/kubernetes/confcontrol
    name: admission-controller
    readOnly: true
```

Step 4 - Enable Admission Controller

Add the ImagePolicyWebhook plugin and specify the configuration file path.

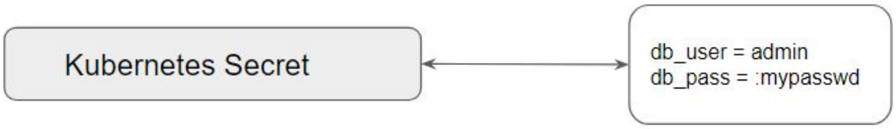


```
containers:
- command:
  - kube-apiserver
  - --advertise-address=172.31.55.197
  - --allow-privileged=true
  - --authorization-mode=Node,RBAC
  - --client-ca-file=/etc/kubernetes/pki/ca.crt
  - --enable-admission-plugins=NodeRestriction,ImagePolicyWebhook
  - --admission-control-config-file=/etc/kubernetes/confcontrol/image.yaml
```

Module 13: Overview of Kubernetes Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key.

Allows customers to store secrets centrally to reduce the risk of exposure.
Stored in the ETCD database.



Following is the syntax for creating a secret via CLI:

kubectl create secret [TYPE] [NAME] [DATA]

Elaborating the types:

i) Generic:

- File (--from-file)
- directory
- literal value

ii) Docker Registry
iii) TLS

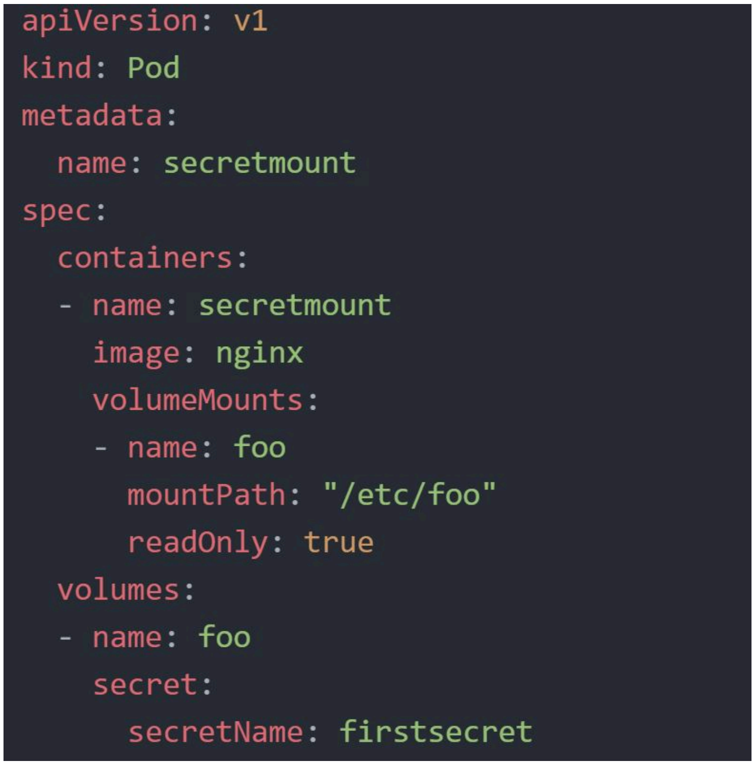
.

Module 14: Mounting Secrets in Containers

Once a secret is created, it is necessary to make it available to containers in a pod.

There are two approaches to achieve this:

- Volumes
- Environment Variables.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secretmount
spec:
  containers:
  - name: secretmount
    image: nginx
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: firstsecret
```