CKS Summary

Cluster Setup - 10%

- Use Network security policies to restrict cluster level access :

Network Policy :

Creation: Implementing Network Policies for both ingress and egress traffic.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
spec:
  podSelector:
    matchLabels:
      role: my-role
  ingress: []
  egress: []
```

Default Deny: Utilizing the default deny principle for enhanced security.

```
spec:
  ingress: []
```

Access Control: Defining default access rules for network policies.

```
spec:
  egress: []
```

Networkpolicy exception :

```
spec:
  egress:
  - to:
      ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
```

test conection to url / port : **nc -v 1.1.1.1 53**

- Properly set up Ingress objects with security control :

Ingress :

Creation: Establishing ingress rules and understanding how they function.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /path1
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 80
```

IngressClass Name: Ensuring the correct IngressClass name is specified.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    kubernetes.io/ingress.class: <class_name>
```

Multiple Paths: Managing multiple paths for Ingress.

Testing with Curl: Employing Curl with options like -k for certificate skipping and -v for verbose output.

```
curl -k -v http://example.com
```

Securing Ingress :

- Creating TLS certificates using OpenSSL. :

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt
```

- Generating secrets with TLS certificates using kubectl.

```
kubectl create secret tls my-tls-secret --key=tls.key --cert=tls.crt
```

- Incorporating the TLS part in the Ingress manifest.

```yaml
tls:
- hosts:
  - example.com
  secretName: my-tls-secret
```

- Ensuring secrets are created in the same Ingress namespace.

- Use CIS benchmark to review the security configuration of Kubernetes components (etcd, kubelet, kubedns, kubeapi)

CIS benchmark : center of internet security that provides k8S default rules

```
# Install kube-bench
go get -u github.com/aquasecurity/kube-bench

# Run kube-bench for all components
kube-bench run --targets master,node

# Run kube-bench for a specific component (e.g., kube-apiserver)
kube-bench run --targets kube-apiserver
```

```
# Run kube-bench with JSON output
kube-bench run --targets master --json

# Run kube-bench with JUnit XML output
kube-bench run --targets node --junit
```

- Protect node metadata and endpoints :

Solution 1 :

```yaml
# all pods in namespace cannot access metadata endpoint
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: cloud-metadata-deny
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
        - 169.254.169.254/32
```

Solution 2 :

```yaml
# only pods with label are allowed to access metadata endpoint
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: cloud-metadata-allow
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: metadata-accessor
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 169.254.169.254/32
```

- Minimize use of, and access to, GUI elements

  1. Minimize GUI Access:

```
# Disable Kubernetes Dashboard
kubectl delete -f kubernetes-dashboard.yaml
```

2. RBAC Implementation:

```yaml
# Define RBAC role for limited GUI access
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: gui-access-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

3. Enable Audit Logging:

```yaml
# Enable audit logging in kube-apiserver
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
apiServer:
  auditLogPath: "/var/log/audit.log"
```

4. Disable GUI Access (Optional):

```
# Disable Kubernetes Dashboard Service
kubectl delete svc kubernetes-dashboard -n kube-system
```

- Verify platform binaries before deploying

```
# Example of verifying a binary using SHA256 checksum
sha256sum -c checksums.txt
```

Best Practices :

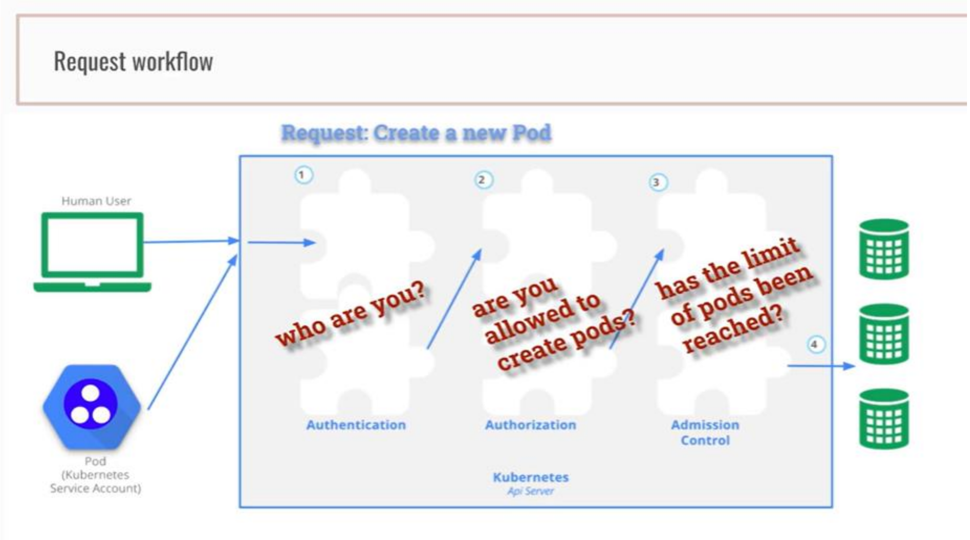
- **Checksums and Signatures:** Utilize checksums (e.g., SHA256) for quick integrity checks and digital signatures for both integrity and authenticity verification. These practices provide a robust mechanism for binary verification.

- **Source Verification:** Obtain binaries exclusively from official and reputable sources. Avoid downloading software from unauthorized or unverified locations to minimize the risk of deploying compromised versions.

- **Automation:** Implement automated tools and processes for verifying binaries. Automation ensures consistency and reduces the potential for human error in the verification process, contributing to a more secure deployment pipeline.

Cluster Hardening - 15%

- Utilize Role Based Access Controls (RBAC) to limit access to the Kubernetes API.



Restrictions

1. **Don't allow anonymous access** kube-apiserver --anonymous-auth=true|false
2. **Close insecure port** Since K8s 1.20 the insecure access is not longer possible kube-apiserver –insecure-port=8080
3. **Don't expose ApiServer to the outside**
4. **Restrict access from Nodes to API (NodeRestriction)** kube-apiserver --enable-admission-plugins=NodeRestriction
5. Prevent unauthorized access (RBAC)
6. Prevent pods from accessing API
7. Apiserver port behind firewall / allowed ip ranges (cloud provider)

 # inspect apiserver cert
cd /etc/kubernetes/pki
openssl x509 -in apiserver.crt -text

- Use Role Based Access Controls (RBAC) to minimize exposure

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

- Exercise caution in using service accounts e.g. disable defaults, minimize permissions on newly created ones , : so this is example of a sa and role then we just shall bind this by creating a rolebinding

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: minimal-permissions
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
```

In K8s there is no **User** as element we manage **serviceAccounts**

To test user ability to delete a deployment : **kubectl auth can-i delete deployments as ala**

A user in kubernetes is someone with a cert and key !!

 # create CertificateSigningRequest with base64 jane.csr

openssl genrsa -out jane.key 2048

openssl req -new -key jane.key -out jane.csr # only set Common Name = jane

cat jane.csr | base64 -w 0

Copy it into the YAML

Then add it to CSR yaml file

 # create Token to use in SA

Kubectl create token tokenName

 # from inside a Pod we can do:

cat /run/secrets/kubernetes.io/serviceaccount/token

curl https://kubernetes.default -k -H "Authorization: Bearer SA_TOKEN"

 # Disable ServiceAccount mounting in a pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  automountServiceAccountToken: false
  containers:
  - name: my-container
    image: nginx:latest
```

 # add new KUBECONFIG

k config set-credentials jane --client-key=jane.key --client-certificate=jane.crt

k config set-context jane --cluster=kubernetes --user=jane

k config view

k config get-contexts

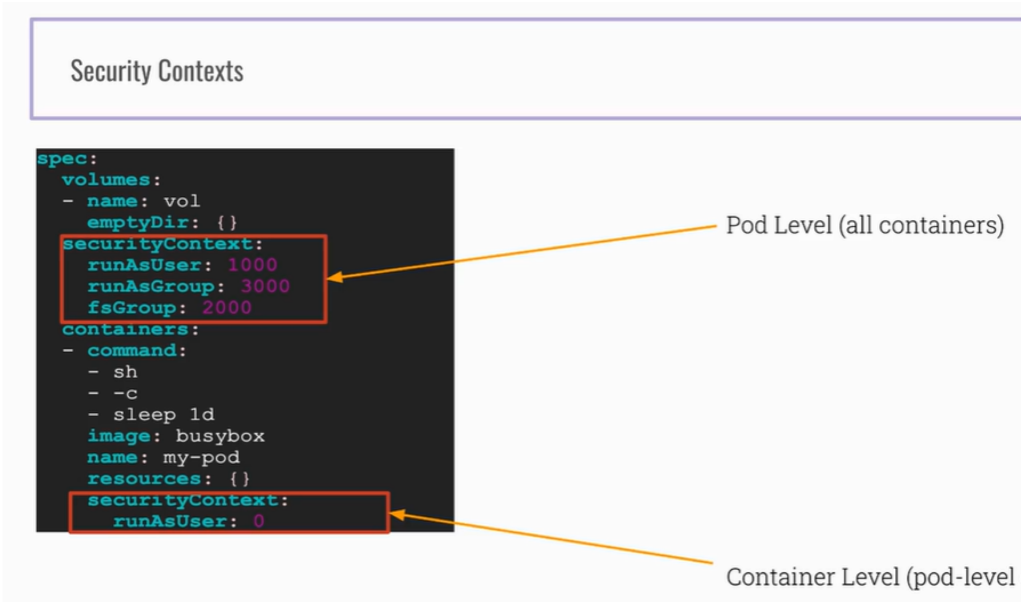
k config use-context jane

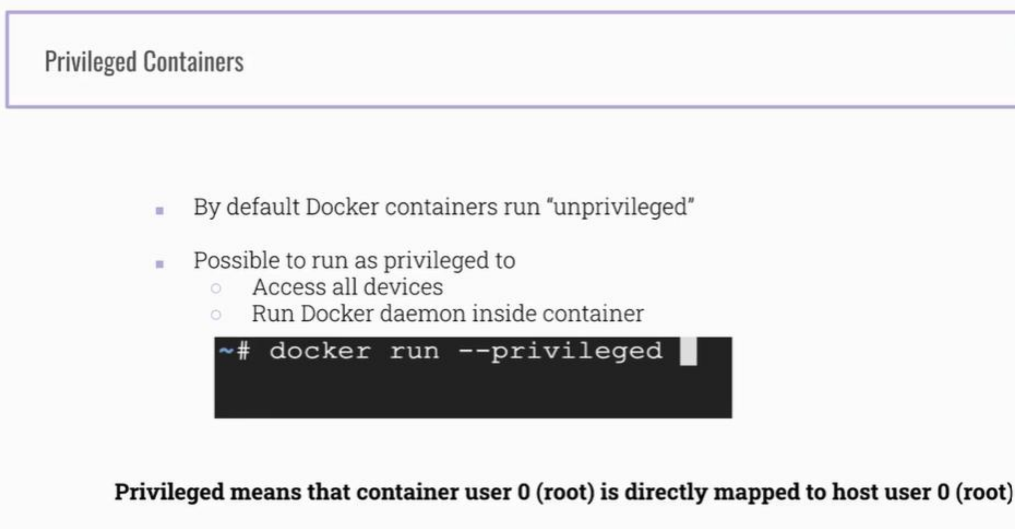
- Update Kubernetes frequently

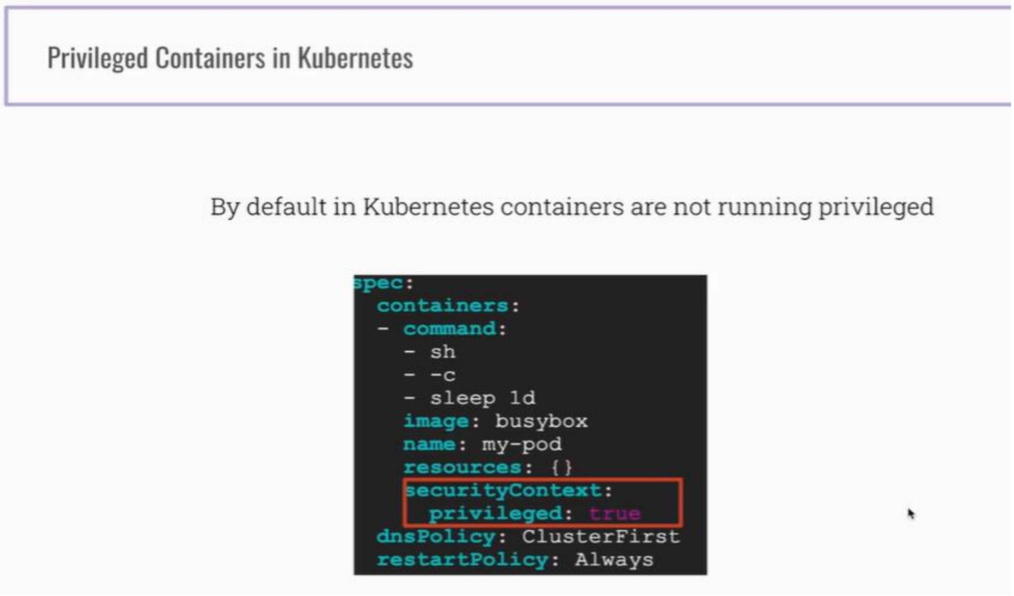
```
kubeadm upgrade plan
kubeadm upgrade apply <desired-version>
```

- ## Minimize Microservice Vulnerabilities - 20%

    - Setup appropriate OS level security domains e.g. using PSP, OPA, security contexts



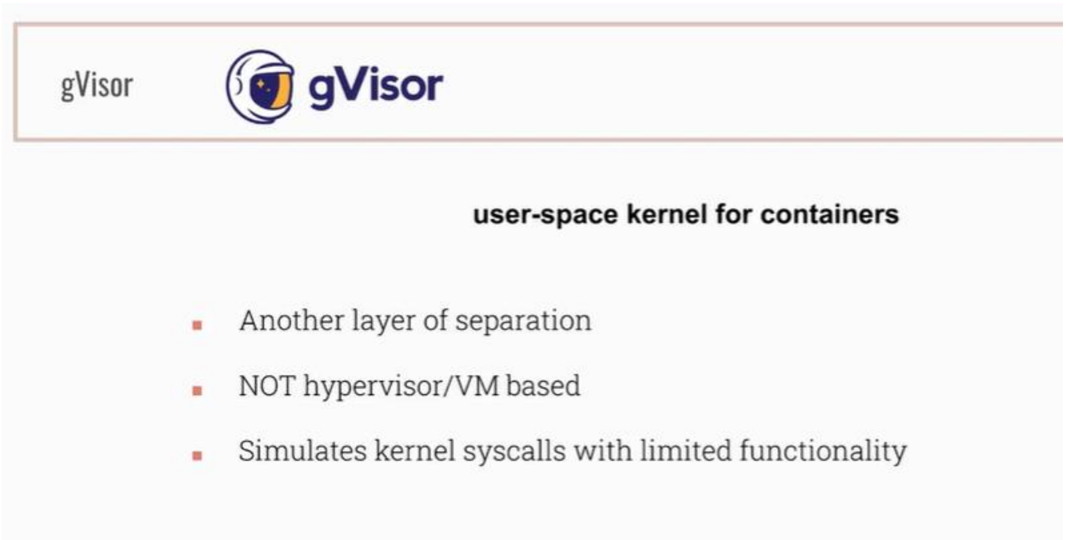


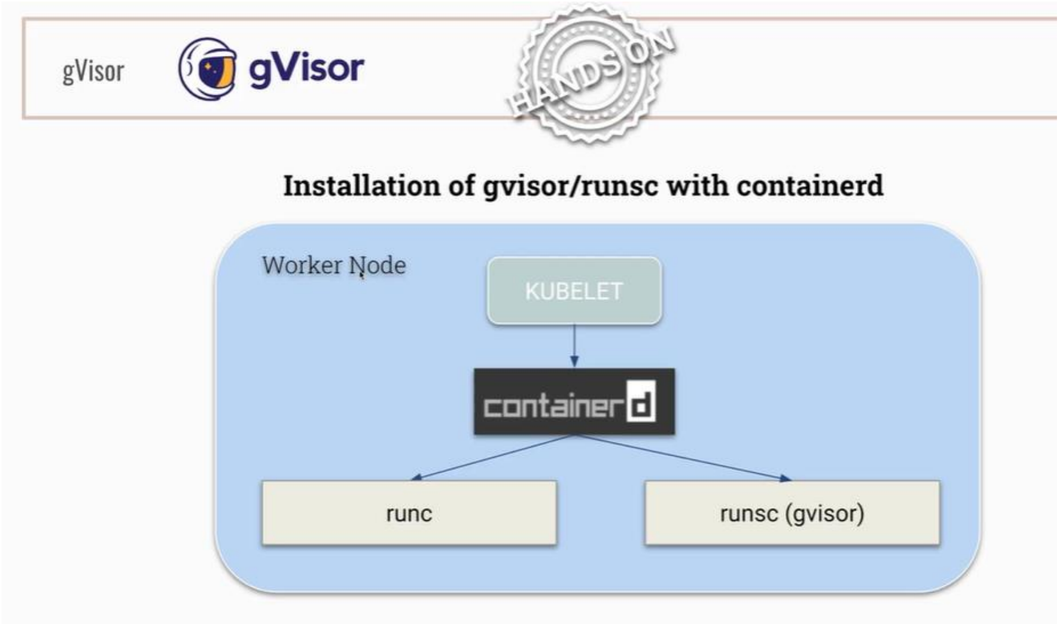


- Manage Kubernetes secrets Use container runtime **sandboxes** in multi-tenant environments (e.g. gvisor, kata containers)

➔ Sandbox is an additional security layer that reduce attack surface

Example : gvisor by google





- To use gvisor sandbox we need first to install it in our cluster then we need to create the runtimeClass and configure it in our pod manifest

First we create the *RuntimeClass*

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

And the *Pod* that uses it

```
apiVersion: v1
kind: Pod
metadata:
  name: sec
spec:
  runtimeClassName: gvisor
  containers:
    - image: nginx:1.21.5-alpine
      name: sec
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

➔ If ETCD secrets aren't encrypted, it's easy to access them. To boost security, we should encrypt secrets in ETCD.

 # accessing secret in etcd

cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep etcd

ETCDCTL_API=3 etcdctl --cert /etc/kubernetes/pki/apiserver-etcd-client.crt --key /etc/kubernetes/pki/apiserver-etcd-client.key --cacert /etc/kubernetes/pki/etcd/ca.crt endpoint health

 # --endpoints "https://127.0.0.1:2379" not necessary because we're on same node

ETCDCTL_API=3 etcdctl --cert /etc/kubernetes/pki/apiserver-etcd-client.crt --key /etc/kubernetes/pki/apiserver-etcd-client.key --cacert /etc/kubernetes/pki/etcd/ca.crt get /registry/secrets/default/secret1

Thats why we need to encrypt our secrets in ETCD :

Encrypt (all Secrets) in etcd

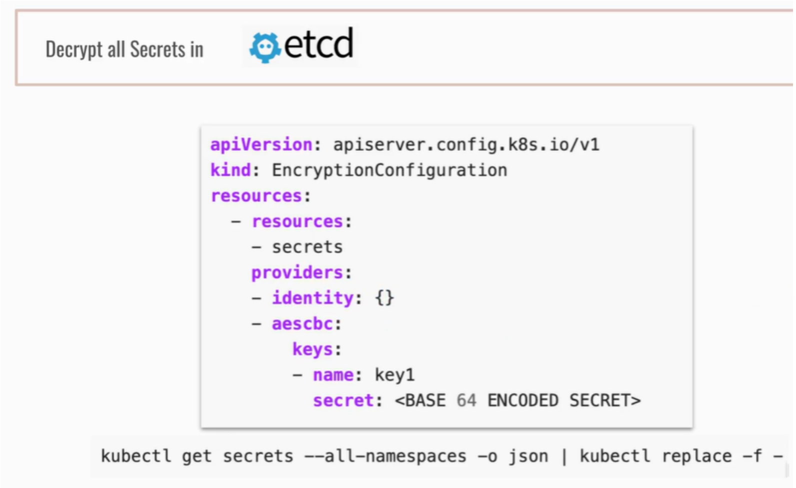
```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - identity: {}
```

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

And in kube-apserver.yaml we need to add these configurations :

```yaml
spec:
  containers:
  - command:
    - kube-apiserver
...
    - --encryption-provider-config=/etc/kul
...
    volumeMounts:
    - mountPath: /etc/kubernetes/etcd
      name: etcd
      readOnly: true
...
  hostNetwork: true
  priorityClassName: system-cluster-critic
  volumes:
  - hostPath:
      path: /etc/kubernetes/etcd
      type: DirectoryOrCreate
    name: etcd
...
```

To decrypt secrets in ETCD :



 # encrypt etcd docs page

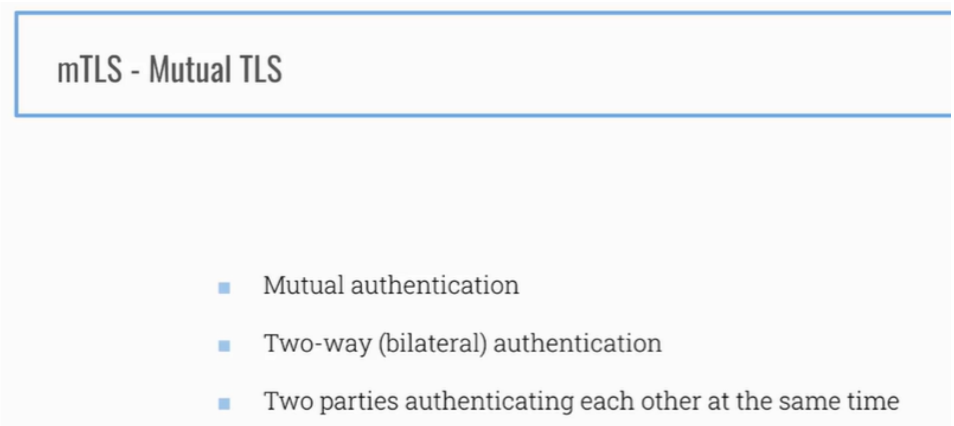
https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data

 # read secret from etcd

ETCDCTL_API=3 etcdctl --cert /etc/kubernetes/pki/apiserver-etcd-client.crt --key /etc/kubernetes/pki/apiserver-etcd-client.key --cacert /etc/kubernetes/pki/etcd/ca.crt get /registry/secrets/default/very-secure

- Implement pod to pod encryption by use of Mtls / Sidecar proxy injection (Istio injection)

➔ **mTLS :**



Sidecar injection

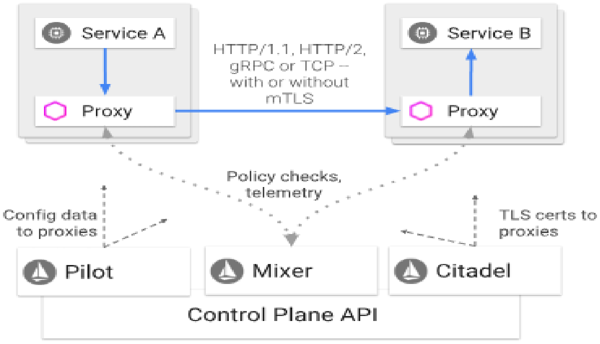
In simple terms, sidecar injection is adding the configuration of additional containers to the pod template. The added containers needed for the Istio service mesh are:

istio-init This init container is used to setup the iptables rules so that inbound/outbound traffic will go through the sidecar proxy. An init container is different than an app container in following ways:

- It runs before an app container is started and it always runs to completion.
- If there are many init containers, each should complete with success before the next container is started.

So, you can see how this type of container is perfect for a set-up or initialization job which does not need to be a part of the actual application container. In this case, istio-init does just that and sets up the iptables rules.

istio-proxy This is the actual sidecar proxy (based on Envoy).



Istio Architecture

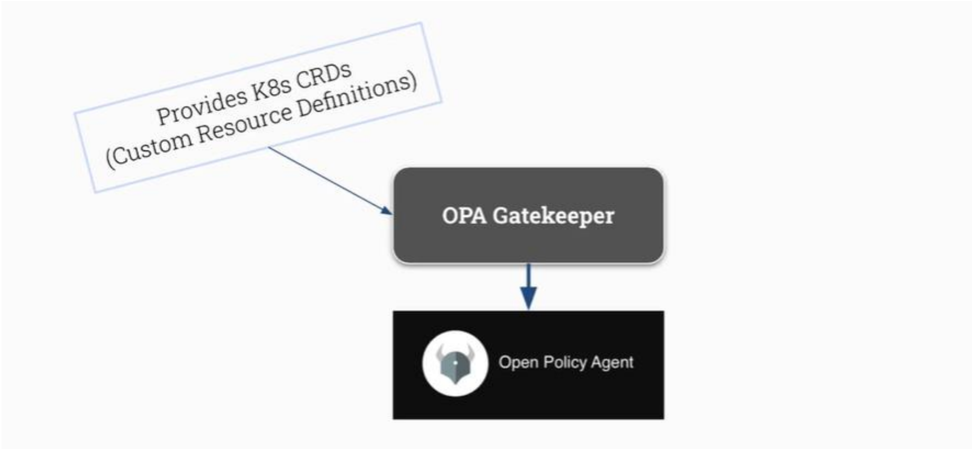
OPA : Open Agent Policy :

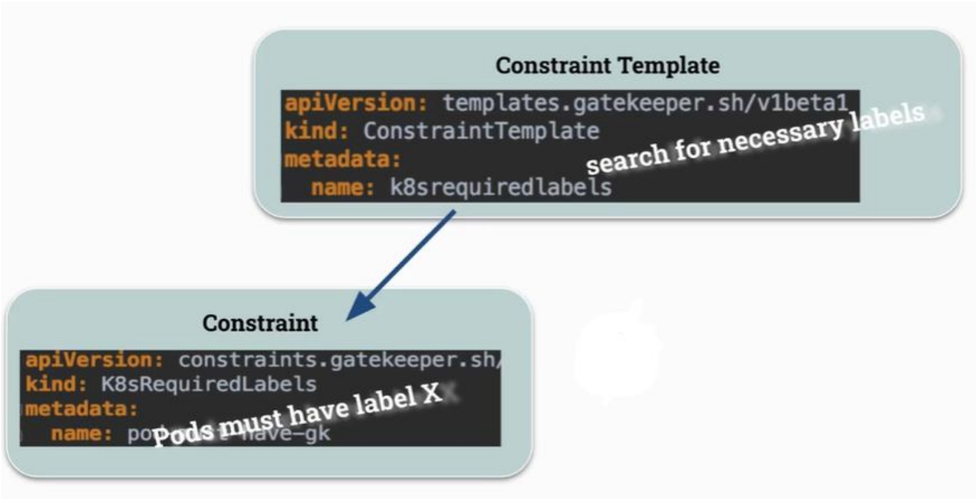
"The Open Policy Agent (OPA) is an open source, general-purpose policy engine that enables unified, context-aware policy enforcement across the entire stack."

- Not Kubernetes specific
- Easy implementation of policies (Rego language)
- Works with JSON/YAML
- In K8s it uses Admission Controllers
- Does not know concepts like pods or deployments

OPA Gatekeeper Overview :

OPA Gatekeeper is a validating admission controller which will be called through Kubernetes webhook. Validating admission controllers are used to validate whether a Kubernetes resource is permitted with a set of rules or policies before it is created or updated in the actual system.





- **Supply Chain Security 20%**
  - Minimize base image footprint Secure your supply chain: whitelist allowed registries, sign and validate images

AND HERE A GOOD EXAMPLE OF DOCKER IMAGE of a Golang APP :

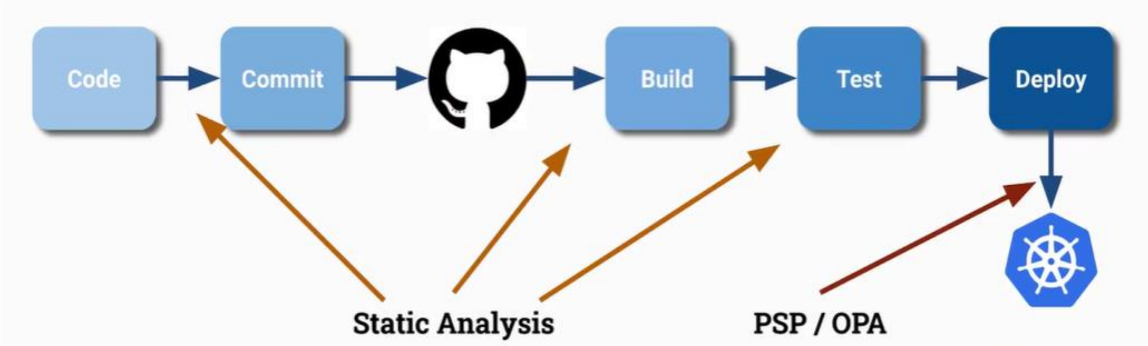
```
# build container stage 1
FROM ubuntu:20.04
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y golang-go=2:1.13~1ubuntu2
COPY app.go .
RUN pwd
RUN CGO_ENABLED=0 go build app.go

# app container stage 2
FROM alpine:3.12.0
RUN addgroup -S appgroup && adduser -S appuser -G appgroup -h /home/appuser
RUN rm -rf /bin/*
COPY --from=0 /app /home/appuser/
USER appuser
CMD ["/home/appuser/app"]
```

- Use static analysis of user workloads (e.g.Kubernetes resources, Docker files)

  This involves employing **static analysis tools** to check Kubernetes workloads and adhere to defined standards. Tools such as Chekov, KubeSec, and KubeLinter can be used for this purpose. These tools help identify and correct common best practices and potential security issues in Kubernetes workloads, including Dockerfiles and Kubernetes YAML files

  Static Analysis in CI/CD :



Kubesec :

- Security risk analysis for Kubernetes resources
- Opensource
- Opinionated ! Fixed set of rules (Security Best Practices)
- Run as:
  - Binary
  - Docker container
  - Kubectl plugin
  - Admission Controller (kubesec-webhook)

 # docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < pod.yaml

ConfTest - OPA :

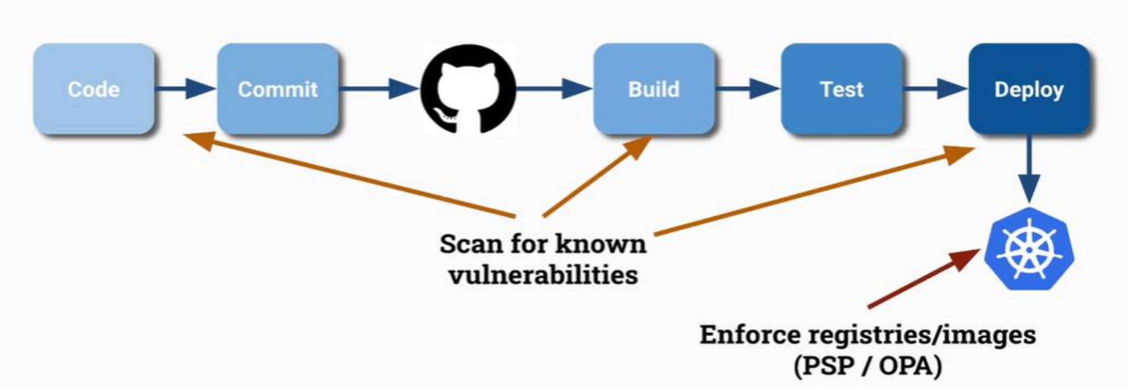
- OPA = Open Policy Agent
- Unit test framework for Kubernetes configurations
- Uses Rego language

```
package main

deny[msg] {
  input.kind = "Deployment"
  not input.spec.template.spec.securityContext.runAsNonRoot = true
  msg = "Containers must not run as root"
}
```

- Scan images for known vulnerabilities

This part focuses on scanning container images and containers for known vulnerabilities using tools like **Clair**, **Trivy** , and **Grype** ... These tools allow users to scan for known CVE vulnerabilities and ensure configuration compliance. They also enable the remediation of container images to specified policies



Trivy :

 # docker run ghcr.io/aquasecurity/trivy:latest image nginx:latest

 # docker run ghcr.io/aquasecurity/trivy:latest image nginx:latest | grep CRITICAL

Best Practices for securing Supply chain :

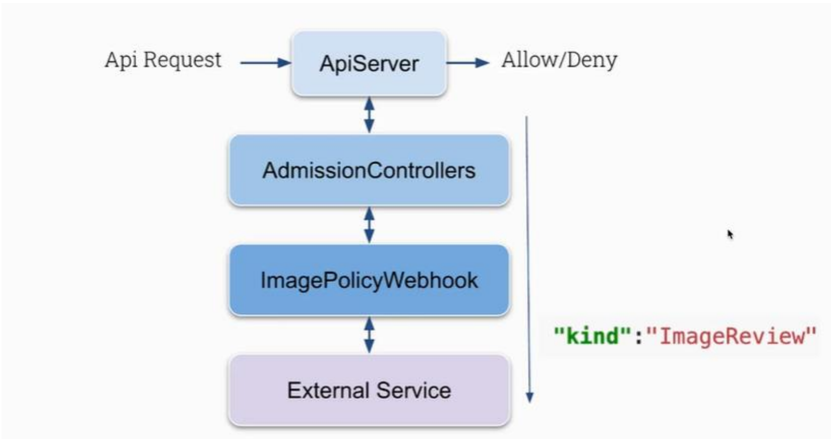
- In kubeapi-server we use the image digest and not image version

- OPA policies to use specific registries :

to check all existing images in our cluster ➔ **# kubectl get po -A -o yaml | grep -v "f :"**

- **Image policy webhook**

The ImagePolicyWebhook is an admission controller that evaluates only images.



And here and example of an ImagePolicyWebhook :

The `/etc/kubernetes/policywebhook/admission_config.json` should look like this:

```
{
   "apiVersion": "apiserver.config.k8s.io/v1",
   "kind": "AdmissionConfiguration",
   "plugins": [
      {
         "name": "ImagePolicyWebhook",
         "configuration": {
            "imagePolicy": {
               "kubeConfigFile": "/etc/kubernetes/policywebhook/kubeconf",
               "allowTTL": 100,
               "denyTTL": 50,
               "retryBackoff": 500,
               "defaultAllow": false
            }
         }
      }
   ]
}
```

The `/etc/kubernetes/policywebhook/kubeconf` should contain the correct server:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/policywebhook/external-cert.pem
    server: https://localhost:1234
  name: image-checker
...
```

The apiserver needs to be configured with the `ImagePolicyWebhook` admission plugin:

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --enable-admission-plugins=NodeRestriction,ImagePolicyWebhook
    - --admission-control-config-file=/etc/kubernetes/policywebhook/admission_config.json
```

 # to debug the apiserver we check logs in:

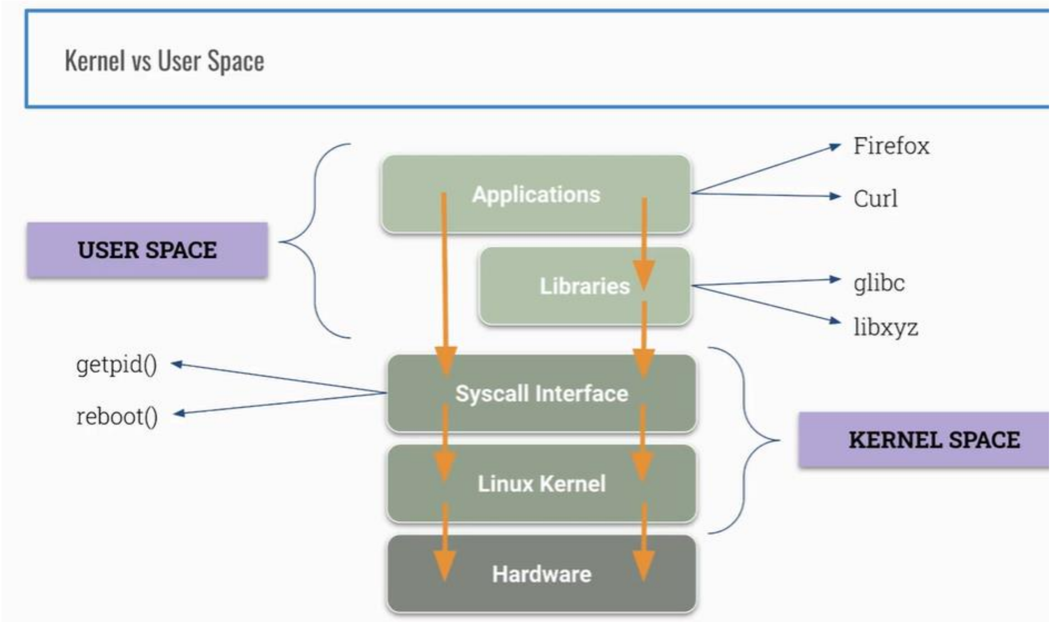
/var/log/pods/kube-system_kube-apiserver*

 # example of an external service which can be used

https://github.com/flavio/kube-image-bouncer

Monitoring, Logging and Runtime Security - 20%

- Perform behavioral analytics of syscall process and file activities at the host and container level to detect malicious activities



Strace :

- strace trace system calls and signals
example: # **strace -cw ls /**

- Binaries are a process of a set of syscalls like write read nmap etc..

- with strace k8S etcd we should :

list sysacalls : # **strace -p 2113 -cw -f**
to find open files : **# cd proc/2113**
 # ls

 # cd fd
 # ls -lh
 # cat 7 | strings | grep secret -A20 -B20
read secret value : **# strace -p 2113 -cw -f**

pstree - display a tree of processes

 # pstree -p

we can find the values of env part in the running pod by searching the process of the container the environ file:

 # cd proc/2544
 # ls -lh
 # cat environ

so secret on variables env can be read on the proc directory on the host !!!!

Falco :

Falco detect security threats in real time ,Falco is a cloud-native security tool designed for Linux systems. It employs custom rules on kernel events, which are enriched with container and Kubernetes metadata, to provide real-time alerts.

Install it as a daemonset

 # install falco

curl -s https://falco.org/repo/falcosecurity-packages.asc | apt-key add -
echo "deb https://download.falco.org/packages/deb stable main" | tee -a /etc/apt/sources.list.d/falcosecurity.list

apt-get update -y

apt-get install -y linux-headers-$(uname -r)

apt-get install -y falco=0.32.1

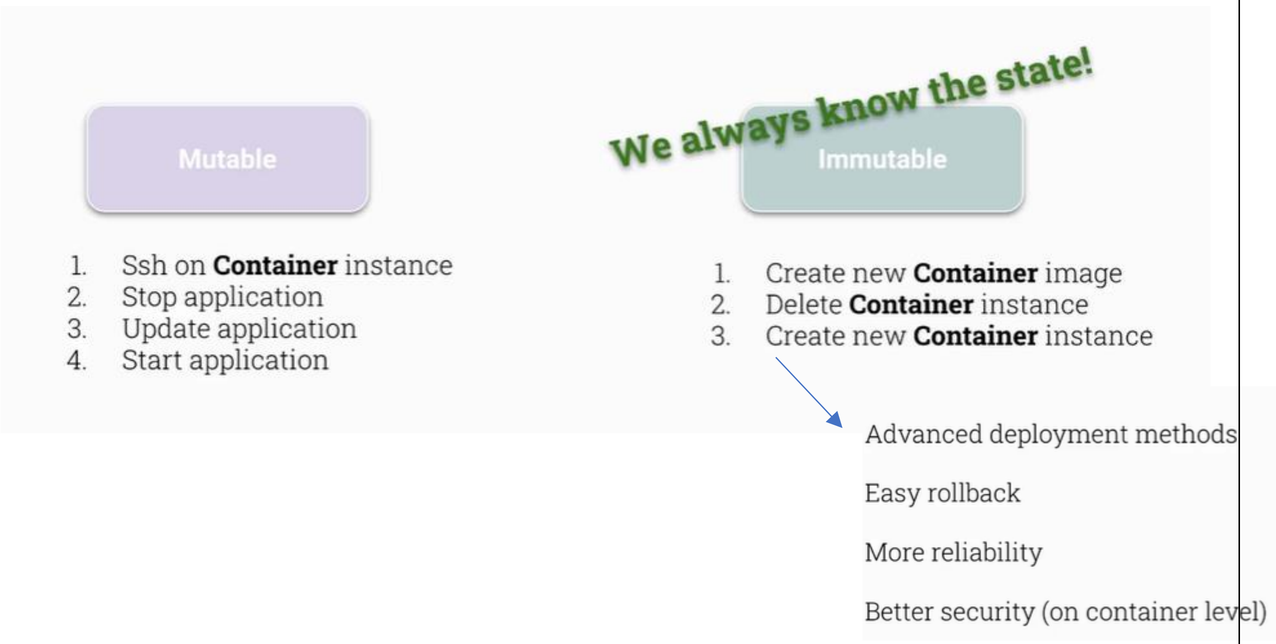
#invesigate with falco

tail -f var/log/syslog | grep falco

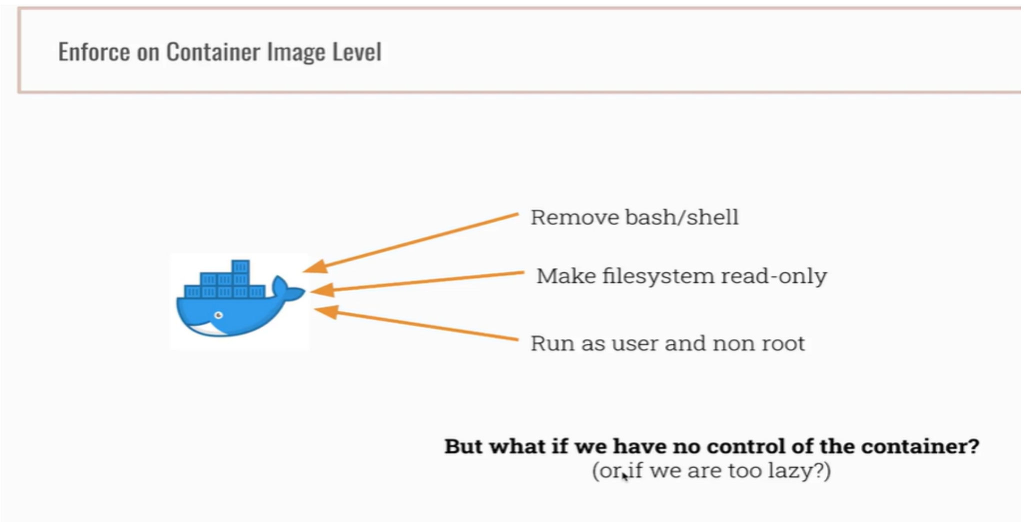
To change rule in falco we should copy the rule from falco.rule.yaml ➔ falco.rule.local.yaml to override the rule localy.

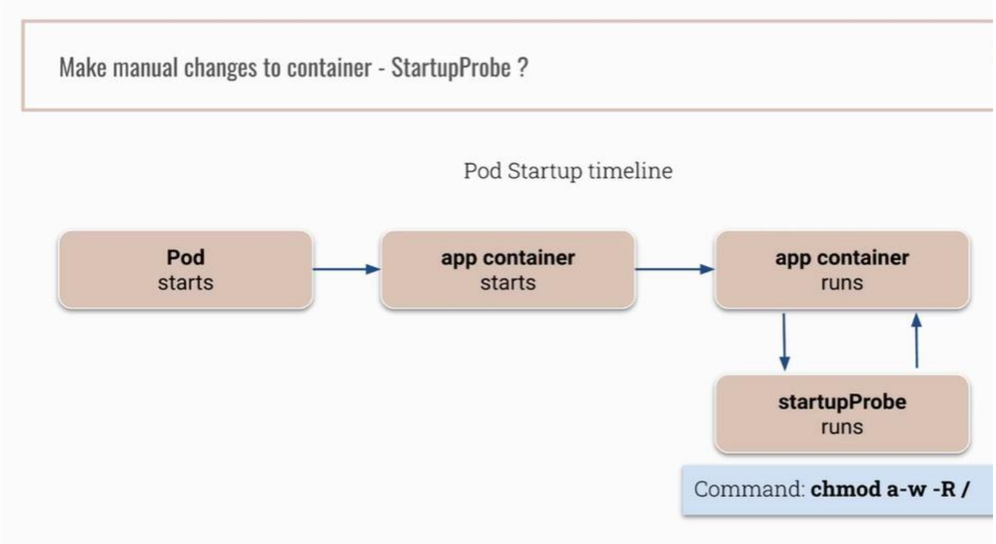
- Detect threats within physical infrastructure, apps, networks, data, users and workloads
- Detect all phases of attack regardless where it occurs and how it spreads
- Ensure immutability of containers at runtime

Difference between mutable and immutable in containers :



There is different ways to enforce immutability :





Create Pod SecurityContext to make filesystem Read-Only

Ensure some directories are still writeable using emptyDir volume

```
docker run --read-only --tmpfs /run my-container
```

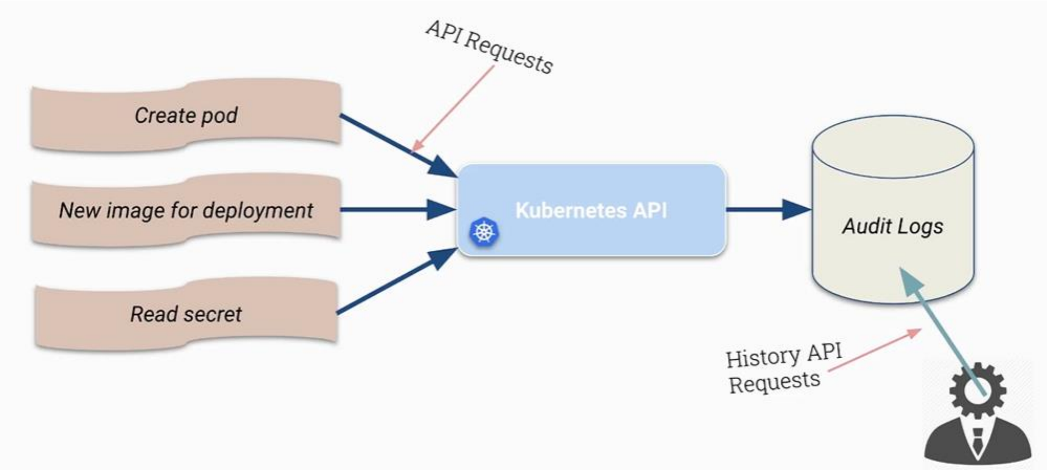
And here an example of the use of security context in pod level :

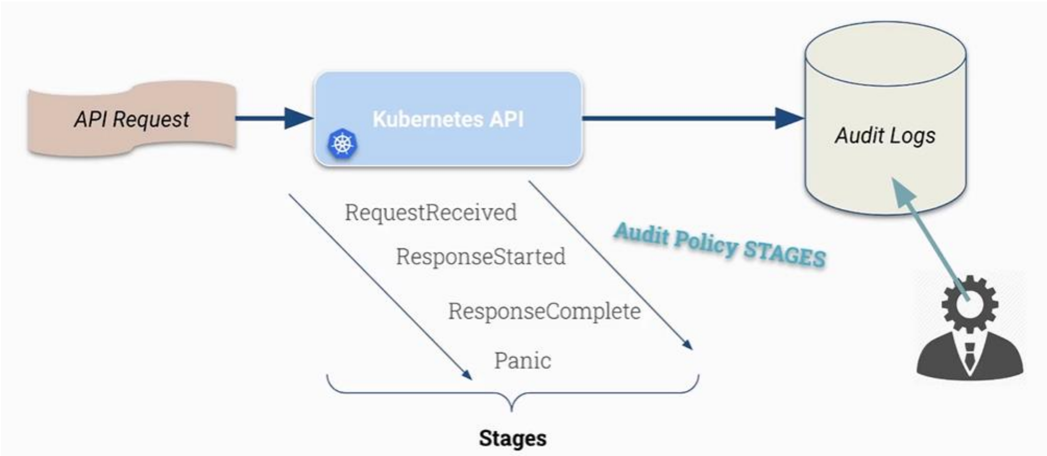
```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: immutable
  name: immutable
spec:
  containers:
  - image: httpd
    name: immutable
    resources: {}
    securityContext:
      readOnlyRootFilesystem: true
    volumeMounts:
    - mountPath: /usr/local/apache2/logs
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
~
~
```

- Use Audit Logs to monitor access

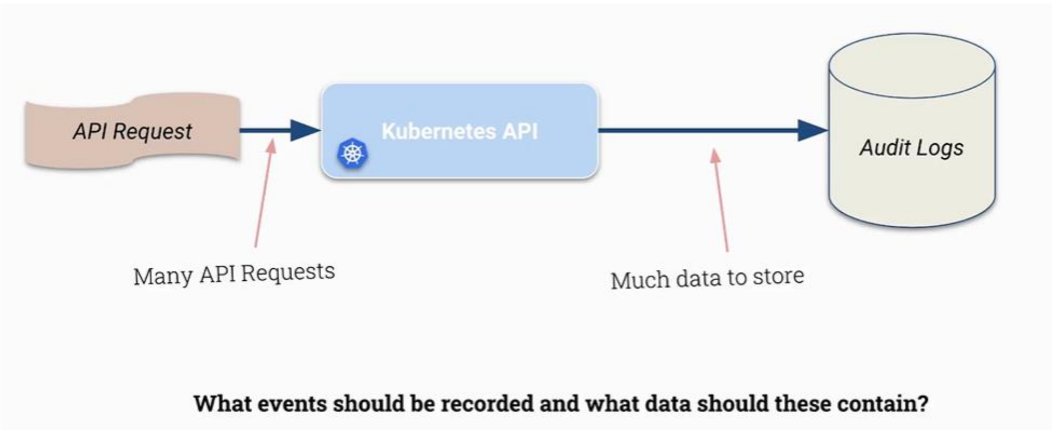
Audit Logs :

Audit logs record the occurrence of an event, the time at which it occurred, the responsible user or service, and the impacted entity. All of the components in your cluster emit logs that may be used for auditing purposes.





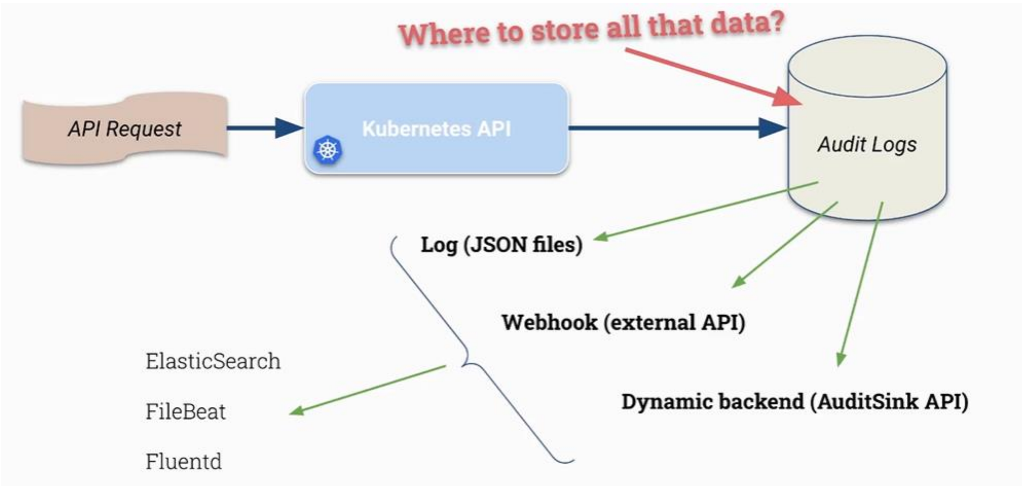
So what Data to store ?



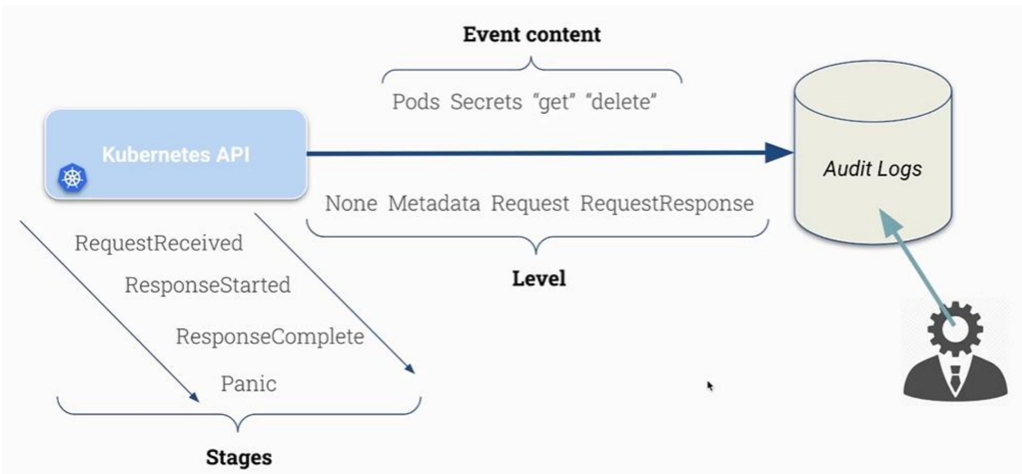
That why we should define Audit policy ..

Audit Policy Rule LEVELS

- `None` - don't log events that match this rule.
- `Metadata` - log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- `Request` - log event metadata and request body but not response body. This does not apply for non-resource requests.
- `RequestResponse` - log event metadata, request and response bodies. This does not apply for non-resource requests.



Audit Logs - Overview :



And here an Example of an Audit Policy:

We want to restrict logged data with an Audit Policy

- Nothing from stage RequestReceived
- Nothing from "get", "watch", "list"
- From Secrets only metadata level
- Everything else RequestResponse level

1. Change policy file
2. Disable audit logging in apiserver, wait till restart
3. Enable audit logging in apiserver, wait till restart
   a. If apiserver doesn't start, then check: /var/log/pods/kube-system_kube-apiserver*
4. Test your changes

```yaml
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages:
  - "RequestReceived"
rules:

- level: None
  verbs: ["get", "list", "watch"]

- level: Metadata
  resources:
  - group: ""
    resources: ["secrets"]

- level: RequestResponse
```

And we must add this configurations in our kube-apiserver.yaml !!

```yaml
  - command:
    - kube-apiserver
    - --audit-policy-file=/etc/kubernetes/audit/policy.yaml       # add
    - --audit-log-path=/etc/kubernetes/audit/logs/audit.log       # add
    - --audit-log-maxsize=500                                     # add
    - --audit-log-maxbackup=5                                     # add
```

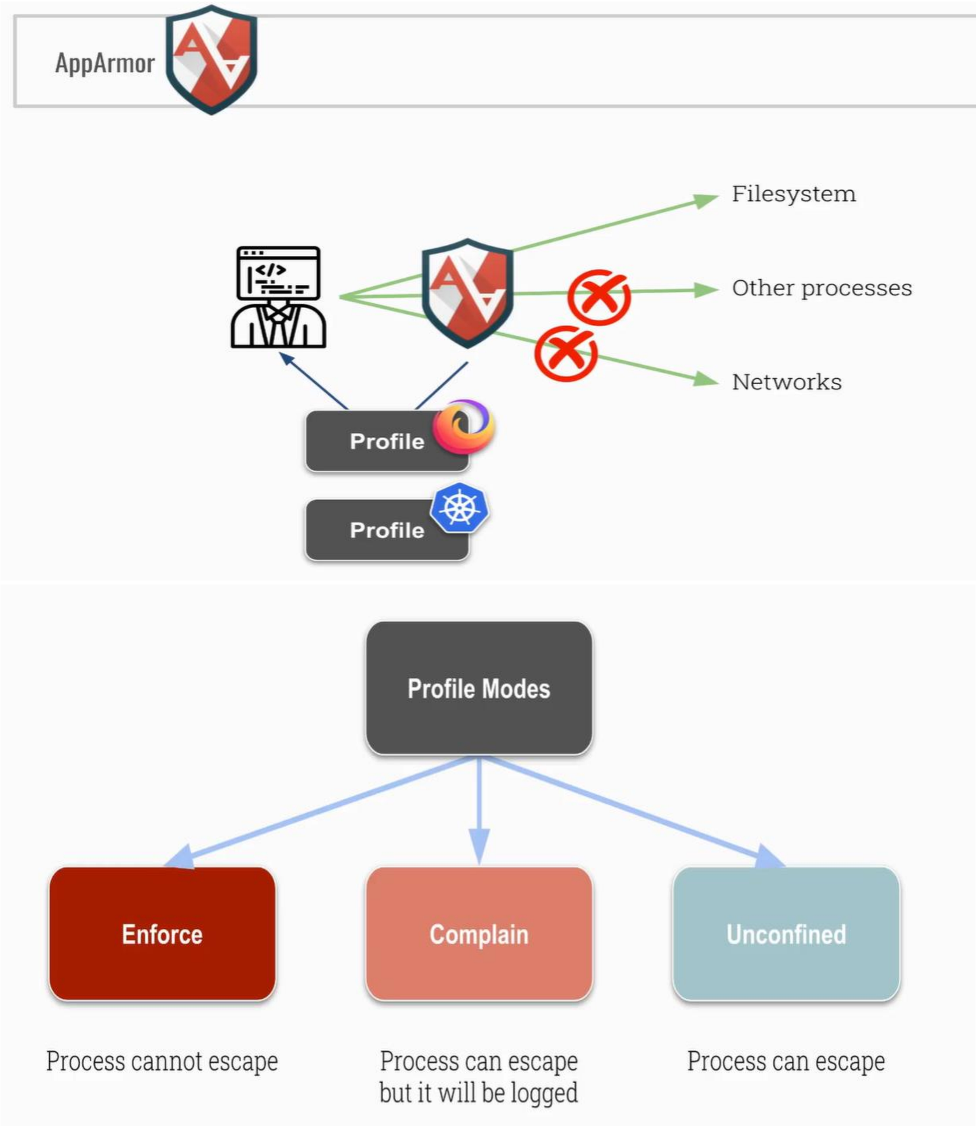
```yaml
    - mountPath: /etc/kubernetes/audit      # add
      name: audit                           # add
  hostNetwork: true
  priorityClassName: system-node-critical
  volumes:
  - hostPath:                               # add
      path: /etc/kubernetes/audit           # add
      type: DirectoryOrCreate               # add
    name: audit                             # add
```

System Hardening - 15%

- Appropriately use kernel hardening tools such as AppArmor, seccomp

AppArmor

AppArmor ("Application Armor") is a Linux kernel security module that allows the system administrator to restrict programs' capabilities with per-program profiles



➔ **Prerequisites : # apt install apparmor-utils**

Main Commands

\# show all profiles
aa-status

\# generate a new profile (smart wrapper around aa-logprof)
aa-genprof

\# put profile in complain mode
aa-complain

\# put profile in enforce mode
aa-enforce

\# update the profile if app produced some more usage logs (syslog)
aa-logprof

To install a profile file : **# apparmor_parser path/to/profile**

Using docker : **# docker run --security-opt apparmor=docker-nginx nginx**

AppArmor Kubernetes

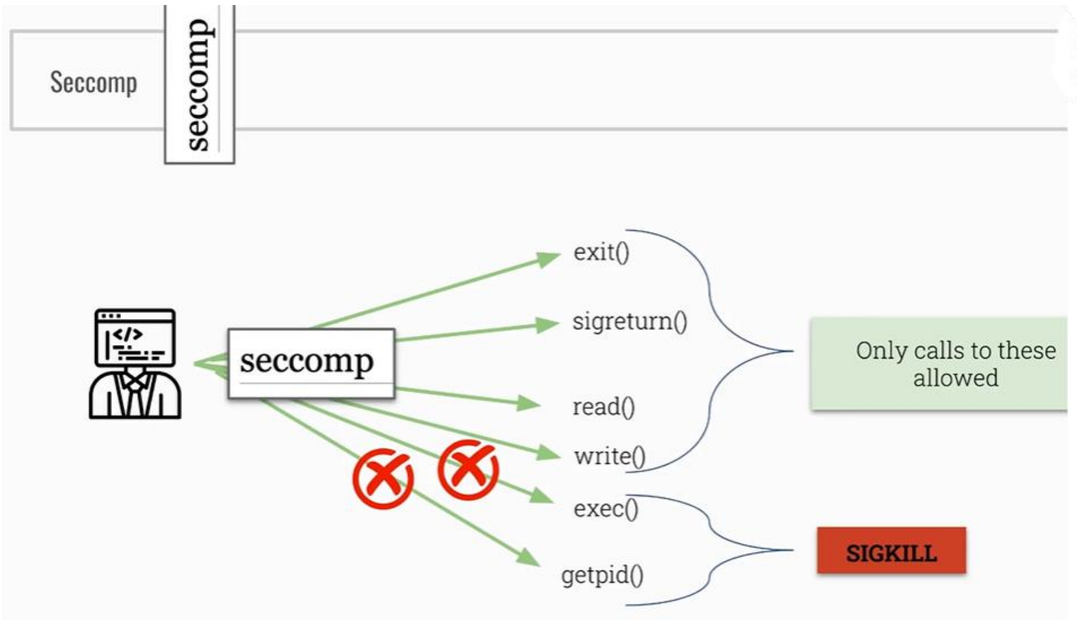
- Container runtime needs to support AppArmor
- AppArmor needs to be installed on every node
- AppArmor profiles need to be available on every node
- AppArmor profiles are **specified per container**
  - done using annotations

And here an Example :

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  annotations:
    container.apparmor.security.beta.kubernetes.io/secure: localhost/docker-nginx
  labels:
    run: secure
  name: secure
spec:
  containers:
  - image: nginx
    name: secure
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Seccomp

Seccomp stands for secure computing mode and has been a feature of the Linux kernel since version 2.6. 12. It can be used to sandbox the privileges of a process, restricting the calls it is able to make from userspace into the kernel



Using docker : **# docker run --security-opt seccomp=profile.json nginx**

Using K8s : first we need to add our profile to the kubelet (worker node)

 # mkdir /var/lib/kubelet/seccomp

 # cd /var/lib/kubelet/seccomp

 # mv profile.json /var/lib/kubelet/seccomp/profiles/

And here an example of pod using seccomp :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:1.0
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

- Minimize host OS footprint (reduce attack surface)

Some linux commands

 # systemctl status snapd

 # systemctl stop snapd

 # systemctl start snapd

 # systemctl kill snapd

 # systemctl disable snapd

 # systemctl list-units | grep snapd

 # ps aux | grep snapd

 # netstat -plnt | grep snapd

 # lsof -i :445

to kill process search for pid then **# kill pid**

 # rm use/bin/snapd

to login as ubuntu user **# su ubuntu**

to get actual used user **# whoami**

to login as root **# sudo -i**

to create user **# adduser test**

to remove package **# apt remove snapd**

to show package details **# apt show snapd**