Kubernetes Security - en

Agenda

1. Starting
   - The truth about security
   - The architecture of Kubernetes
   - Architecture DeepDive
   - Layers to protect (Security)
   - AttackVectors
   - The route from development to production to secure
   - Kill Chain
2. Checklist
   - Security Checklists
3. Getting hacked
   - Why is a cluster so rewarding to hack
   - Starting with Tesla
4. Category 1 by Layer: OS / Kernel
   - Securing the OS and the Kernel
   - Kernel Hardening Checker
5. Category 2 by Layer: Cluster
   - Securing the components
   - Securing kubelet
   - Least Privileges with RBAC
   - Admission Controller
6. Category 3 by Layer: Pods Container
   - The runAs Options in SecurityContext
   - sysctls in pods/containers
   - Overview capabilities
   - Start pod without capabilities & how can we see this
   - Hacking and exploration session HostPID
   - Disable ServiceLinksEnable false)
   - Great but still alpha User Namespaces
7. Reaction
   - The Audit Logs
8. RBAC
   - How does RBAC work ?
   - Where does RBAC play a role ?
   - kubeconfig decode certificate
   - kubectl check your permission - can-i
   - use kubectl in pod - default service account
   - create user for kubeconfig with using certificate
   - Components / moving parts of RBAC
   - practical exercise rbac
9. Obey Security Policies (AdmissionControllers)
   - Admission Controller
   - PSA (PodSecurity Admission)
   - Exercise with PSA
   - OPA Gatekeeper 01-Overview
   - OPA Gatekeeper 02-Install with Helm
   - OPA Gatekeeper 03-Simple Exercise
   - OPA Gatekeeper 04-Example-Job-Debug
   - Connaisseur: Verifying images before Deployment
10. Pod Security
    - Automount ServiceAccounts or not ?
    - Does every pod need to access the kubenernetes api server?
11. Unprivilegierte Pods/Container
    - Which images to use ?
    - How can i debug non-root - Container/Pods?
12. The SecurityContext
    - seccomp
    - privileged/unprivileged
    - appArmor example

- SELinux

13. Network Policies
    - Understand NetworkPolicies
    - Exercise NetworkPolicies
    - CNI Benchmarks
14. ServiceMesh
    - Why a ServiceMesh ?
    - How does a ServiceMeshs work? (example istio
    - istio security features
    - istio-service mesh - ambient mode
    - Performance comparison - baseline,sidecar,ambient
15. Image Security
    - When to scan ?
    - Example Image Security Scanning - using gitlab and trivy
16. Hacking Sessions
    - Hacking with HostPID
17. Extras
    - Canary deployment with basic kubernetes mechanisms
18. Documentation
    - Great video about attacking kubernetes - older, but some stuff is still applicable
    - Straight forward hacking session of kubernetes
    - github with manifests for creating bad pods

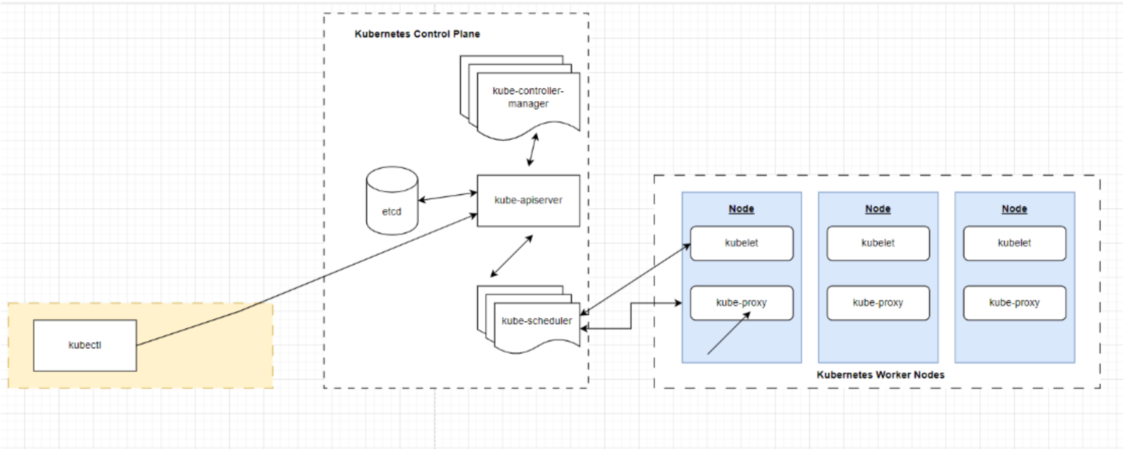
Starting

The truth about security

- It is an ongoing process
- Kubernetes is not safe by default

The architecture of Kubernetes

Overview



Components

Master (Control Plane)

Jobs

- The master coordinates the cluster
- The master coordinates the activities in the cluster
  - scheduling of applications
  - to take charge of the desired state of application
  - scaling of applications
  - rollout of new updates

Components of the Master

ETCD

- Persistent Storage (like a database), stores configuration and status of the cluster

KUBE-CONTROLLER-MANAGER

- In charge of making sure the desired state is achieved (done trough endless loops)
- Communicates with the cluster through the kubernetes-api (kube-api-server)

KUBE-API-SERVER

- provides api-frontend for administration (no gui)
- Exposes an HTTP API (users, parts of the cluster and external components communicate with it)
- REST API

KUBE-SCHEDULER

- assigns Pods to Nodes.
- scheduler determines which Nodes are valid placements for each Pod in the scheduling queue (according to constraints and available resources)
- The scheduler then ranks each valid Node and binds the Pod to a suitable Node.
- Reference implementation (other schedulers can be used)

Nodes

- Nodes (Knoten) sind die Arbeiter (Maschinen), die Anwendungen ausführen
- Ref: https://kubernetes.io/de/docs/concepts/architecture/nodes/

Pod/Pods

- pods are the smallest unit you can roll out on the clusteber
- a pod (basically another word for group) is a group of 1 or more containers
  - mutually used storage and network resources (all containers in the same pod can be reached with localhost)
  - They are always on the same (virtual server)

Control Plane Node (former: master) - components

Node (Minion) - components

General

- On the nodes we will rollout the applications

kubelet

```
Node Agent that runs on every node (worker)
its job is to download images and start containers
```

kube-proxy

- Runs on all of the nodes (DaemonSet)
- Is in charge of setting up the network rules in iptables for the network services
- Kube-proxy is in charge of the network communication inside of the cluster and to the outside

ref:

- https://www.redhat.com/en/topics/containers/kubernetes-architecture

Architecture DeepDive

- https://github.com/jmetzger/training-kubernetes-advanced/assets/1933318/1ca0d174-f354-43b2-81cc-67af8498b56c

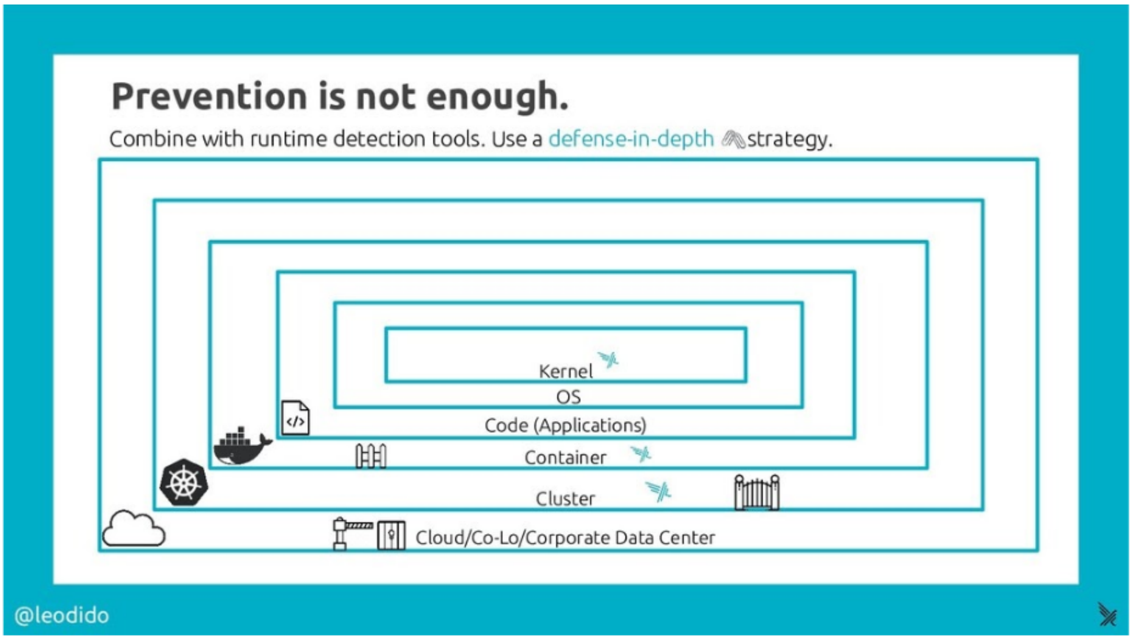
Layers to protect (Security)

Based on the 4-C - Model

- Cloud
- Cluster
- Container
- Code

But let us put it a bit further:

- OS
- Kernel



Credits: @Leodido

AttackVectors

What 3 types of attack vectors are there ?

1. External Attackers
2. Malicious containers
3. Compromised or malicious users

External Attackers

- You can have threat actors who have no access to the cluster but are able to reach the application running on it.

Malicious containers

- If a threat actor manages to breach a single container, they will attempt to increase their access and take over the entire cluster.

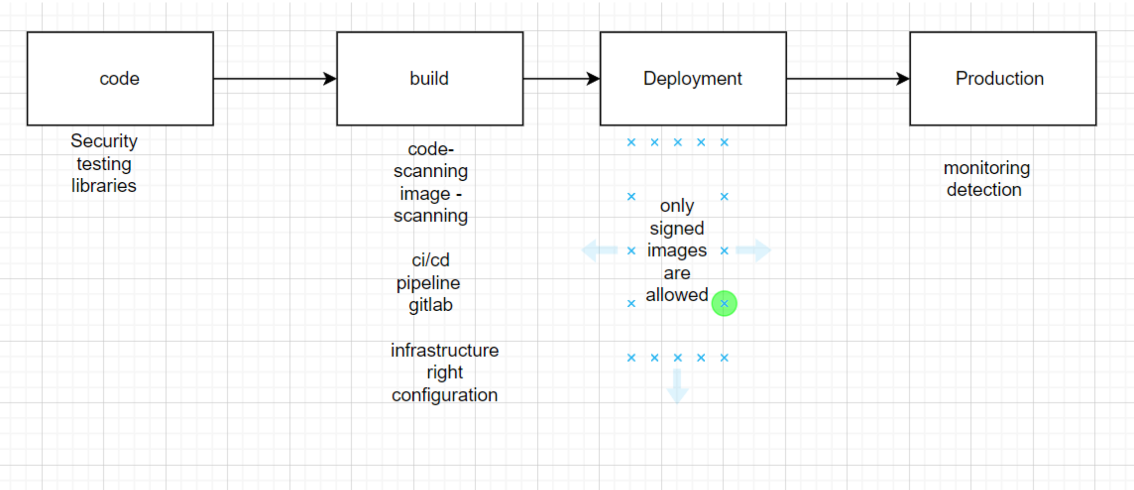
Compromised or malicious users

- When you're dealing with compromised accounts or malicious users, an attacker with stolen yet valid credentials will execute commands against network access and the Kubernetes API.
- Mitigation: Least Principle Policy

Reference:

- https://www.cncf.io/blog/2021/11/08/kubernetes-main-attack-vectors-tree-an-explainer-guide/

The route from development to production to secure

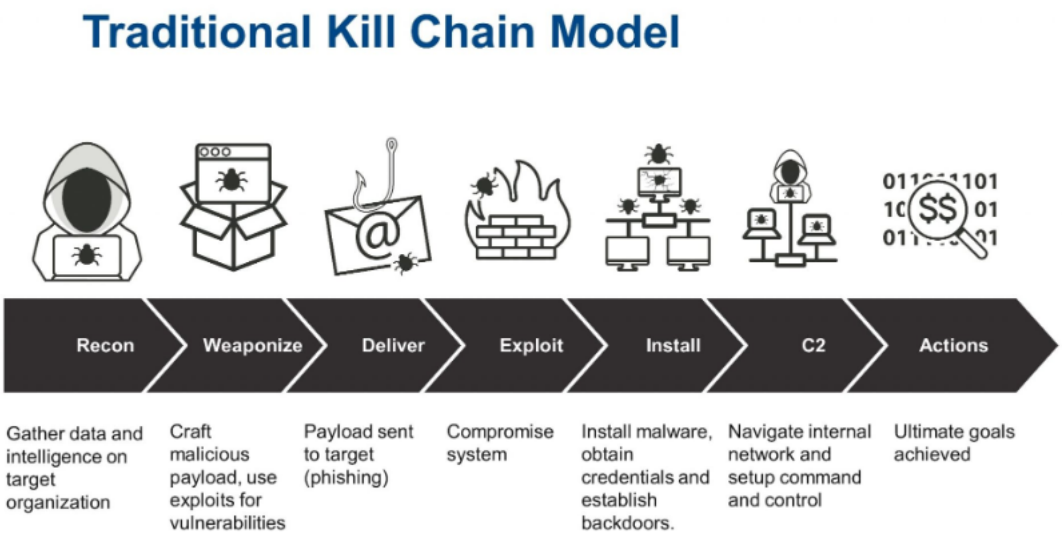


Kill Chain

Steps

1. Reconnaissance
2. Weaponization (Trojaner)
3. Delivery (wie liefern wie ihn aus ?)
4. Exploit (Sicherheitslücke ausnutzen)
5. Installation (phpshell)
6. Command & Control
7. Action/Objectives (mein Ziel)

Or better: graphical



(Source: techtag.de)

Checklist

Security Checklists

For containers/pods

```
Containers should not be running as root.
Containers are missing securityContext.
RBAC Protect cluster-admin ClusterRoleBindings.
Prohibit RBAC Wildcards for Verbs.
Services should not be using NodePort.
```

```
Containers should mount the root filesystem as read-only.
Containers should not share hostIPC.
Containers should not be using hostPort.
Containers should not be mounting the Docker sockets.
```

In general

- https://github.com/krol3/kubernetes-security-checklist/

Getting hacked

Why is a cluster so rewarding to hack

- You have lots and lots of computational power, just be hacking one cluster
- This means, you can even earn money: cryptominer can be installed

Starting with Tesla

- https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/

Category 1 by Layer: OS / Kernel

Securing the OS and the Kernel

Kernel

- Always patch to the newest kernel
- Be sure to restart the server (in most cases new kernel will start to get used after reboot)

Good tool for detecting great hardening kernel parameter

- Kernel hardening checker

Modules

```
## sysctl -w kernel.modules_disabled=1
kernel.modules_disabled = 1
```

```
* If possible harden your kernel, e.g.
* But of course, it is then not allowed to load modules after that isset
```

OS

- Only install the really needed software in os
  - Eventuall start from a minimal image
- Close unneeded ports

OS-Patching

- Patch frequently. Eventually using unattended-upgrades

Hardening Guide

- A bit older, but has really good hints

Telekom Hardening Guide

Kernel Hardening Checker

Checker

```
https://github.com/a13xp0p0v/kernel-hardening-checker?tab=readme-ov-file

## Installation
cd /usr/src
git clone https://github.com/a13xp0p0v/kernel-hardening-checker?tab=readme-ov-file
cd kernel-hardening-checker
```

```
sudo sysctl -a > sysctl.file && ./bin/kernel-hardening-checker -c /boot/config-6.8.0-44-generic -l /proc/cmdline -s sysctl.file
```

Kernel Defence Map

- https://github.com/a13xp0p0v/linux-kernel-defence-map

Guidelines

- https://gist.github.com/dante-robinson/3a2178e43009c8267ac02387633ff8ca

Category 2 by Layer: Cluster

Securing the components

Securing kubelet

Breach 1: bypass admission controller

```
If a static Pod fails admission control, the kubelet won't register the
Pod with the API server. However, the Pod still runs on the node.
```

- https://kubernetes.io/docs/concepts/security/api-server-bypass-risks/

Mitigate breach: Disable the directory for static pods on worker-nodes

- https://kubernetes.io/docs/concepts/security/api-server-bypass-risks/#static-pods-mitigations

```
## change the setting kubelef-config
staticPodPath: /etc/kubernetes/manifests
## -> to
staticPodPath:
```

```
after that:
```

- Log in to a kubeadm node
- Run `kubeadm upgrade node phase kubelet-config` to download the latest `kubelet-config` ConfigMap contents into the local file `/var/lib/kubelet/config.yaml`
- Edit the file `/var/lib/kubelet/kubeadm-flags.env` to apply additional configuration with flags
- Restart the kubelet service with `systemctl restart kubelet`

https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-reconfigure/

- Only Enable the behaviour really needed
- https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/#static-pod-creation

Breach 2: Allowing anonymous access to kubelet

Disabling it in anycase

- in the newer of kubeadm it is already the case
- Check with provider / installer

```
authentication:
     anonymous:
       enabled: false
```

- Log in to a kubeadm node
- Run `kubeadm upgrade node phase kubelet-config` to download the latest `kubelet-config` ConfigMap contents into the local file `/var/lib/kubelet/config.yaml`
- Edit the file `/var/lib/kubelet/kubeadm-flags.env` to apply additional configuration with flags
- Restart the kubelet service with `systemctl restart kubelet`

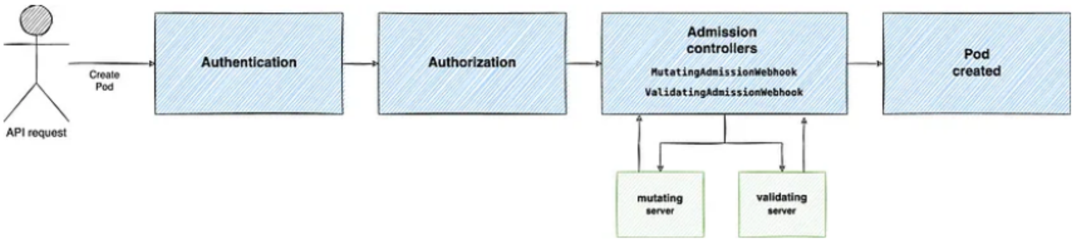
Least Privileges with RBAC

The least privileges principles

- Always design your pods, user and components, that they really only have the minimal principles they need
- RBAC Resources help you to do that (Service Accounts, Roles, ClusterRoles, Rolebinding, Clusterrolebindings, Groups)

Admission Controller

What does it do ? (The picture)



How do the docs describe it ?

```
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.
```

- intercepts request = gets all the requests and validates or changes (=mutates them)
- Reference: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

What kind of admissionControllers do we have ?

- Mutating and Validating
- There are 2 phases of the AdmissionControlProcess: First mutating, then validating

The static admissionPlugins

- There are static admissionPlugins which are activated by config
- You can see the activated like so

```
## in our system like so.
kubectl -n kube-system describe pods kube-apiserver-controlplane  | grep enable-adm
```

```
--enable-admission-plugins=NodeRestriction
```

- There are some that are activated by default:

```
## in Kubernetes 1.31
CertificateApproval,
CertificateSigning,
CertificateSubjectRestriction,
DefaultIngressClass,
DefaultStorageClass,
DefaultTolerationSeconds,
LimitRanger,
MutatingAdmissionWebhook,
NamespaceLifecycle,
PersistentVolumeClaimResize,
PodSecurity,
Priority,
ResourceQuota,
RuntimeClass,
ServiceAccount,
StorageObjectInUseProtection,
TaintNodesByCondition,
ValidatingAdmissionPolicy,
ValidatingAdmissionWebhook
```

- Reference: What does which AdmissionController (= AdmissionPlugin) do ? https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do

Category 3 by Layer: Pods Container

The runAs Options in SecurityContext

- Best practices. Set on level of the pod
- This also reflects containers that are started with kubectl debug (ephemerla containers)

runAsUser

- Important: UID does not need to exist in container
- Really run as specific user.
- If not set the user from Dockerfile is taken
- Recommended to set it, that will be deep defense line
    - If image has a root user or can not run as root this will fail

Exercise 1

```
cd
mkdir -p manifests
cd manifests
mkdir run1
cd run1
```

```
nano 01-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: pod
  name: nginxrun
spec:
  securityContext:
    runAsUser: 10001
  containers:
  - image: nginx:1.23
    name: pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

```
kubectl apply -f 01-pod.yaml
kubectl describe pod nginxrun
kubectl logs nginxrun
```

Exercise 2: (works)

```
cd
mkdir -p manifests
cd manifests
mkdir run2
cd run2
```

```
nano 01-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: alpinerun
spec:
  securityContext:
    runAsUser: 10001
  containers:
  - image: alpine
    command:
      - sleep
      - infinity
    name: pod
```

```
kubectl apply -f .
kubectl describe pod alpinerun
kubectl exec -it alpinerun -- sh
```

```
id
cd /proc/1/ns
ls -la
```

runAsGroup

- Recommended to set this as well

runAsNonRoot

- Indicates that use must run as none root
- If this is not configure in the image, the start fails

sysctls in pods/containers

What ? Set kernel parameters

- set kernel params in container

- This will only be set for the namespace

Two groups

- sysctl's that are considered safe
- sysctl's that are considered unsecure (these are not enabled by default in Kubernetes)

Safe Settings

```
kernel.shm_rmid_forced;
net.ipv4.ip_local_port_range;
net.ipv4.tcp_syncookies;
net.ipv4.ping_group_range (since Kubernetes 1.18);
net.ipv4.ip_unprivileged_port_start (since Kubernetes 1.22);
net.ipv4.ip_local_reserved_ports (since Kubernetes 1.27, needs kernel 3.16+);
net.ipv4.tcp_keepalive_time (since Kubernetes 1.29, needs kernel 4.5+);
net.ipv4.tcp_fin_timeout (since Kubernetes 1.29, needs kernel 4.6+);
net.ipv4.tcp_keepalive_intvl (since Kubernetes 1.29, needs kernel 4.5+);
net.ipv4.tcp_keepalive_probes (since Kubernetes 1.29, needs kernel
```

Exceptions form Safe Settings

```
The net.* sysctls are not allowed with host networking enabled.
The net.ipv4.tcp_syncookies sysctl is not namespaced on Linux kernel version 4.5 or lower.
```

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
```

Overview capabilities

What are these ?

- Capabilities allow us to execute stuff, that normally only the root user can do.

Best practice for security and container/pods

- Tear capabilities down to Drop: all and set up those, that you need

As little capabilities as possible.

- Use as little capabilities as possible in your pod/cont

List of capabilities

- cap_chown
- cap_dac_override
- cap_fowner
- cap_fsetid
- cap_kill
- cap_setgid
- cap_setuid
- cap_setpcap
- cap_net_bind_service
- cap_net_raw
- cap_sys_chroot
- cap_mknod
- cap_audit_write
- cap_setfcap

cap_chown

- User can chown (Change Owner without being root)

cap_dac_override

- Bypasses permission checks

cap_fowner

```
Bypass permission checks on operations that normally
require the filesystem UID of the process to match the
UID of the file (e.g., chmod(2), utime(2)),
```

cap_fsetid

- Safe to disable
- Not needed as we should not use setgid and setuid bits anyway

```
when creating a new folder in folder with setgit, new folder
will have this permission as well
```

cap_kill

- Please not have this enabled.
- User is allowed to kill processes within the container

```
Bypass permission checks for sending signals
## should not be the job of the container
```

cap_setgid

- Do not enable !

```
Allow to change GID of a process
```

cap_setuid

- makes it possible to privilege escalations

```
make arbitrary manipulations of process UID
(setuid(2), setreuid(2), setresuid(2), setfsuid(2));
```

cap_setpcap

```
Allow user to set other process capabilities
```

cap_net_bind_service

- If you want to bind a privilege port, you will need this
- Like starting httpd on port 80

```
Security Question/Hint:
Is there probably a better way to do this:
e.g. Open Port 8080 -> and having the service on port 80
```

cap_net_raw

- Needed to open a raw socket
- Needed to perform ping
- There have been many vulnerabilites concerning this capability in the past e.g. [CVE-2020-14385]
- Warning: Do not activate it, if you really, really need this
  - You can ping with a debug container if you need to

cap_sys_chroot

```
CAP_SYS_CHROOT permits the use of the chroot(2) system call. This may allow escaping of any chroot(2) environment, using known
weaknesses and
```

cap_mknod

- No need to have this
- Allow to create special files under /dev

cap_audit_write

- Allow to write to kernel audit log

cap_setfcap

- Allow user to set other file capabilities

Documentation

- https://man7.org/linux/man-pages/man7/capabilities.7.html

Start pod without capabilities & how can we see this

Exercise 1:

```
cd
mkdir -p manifests
cd manifests
```

```
mkdir -p nocap
cd nocap
```

```
nano 01-pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nocap-nginx
spec:
  containers:
    - name: web
      image: bitnami/nginx
      securityContext:
        capabilities:
          drop:
          - all
```

```
kubectl apply -f .
kubectl get pods
kubectl logs nocap-nginx
```

Exercise 2

```
nano 02-alpine.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nocap-alpine
spec:
  containers:
    - name: web
      command:
        - sleep
        - infinity
      image: alpine
      securityContext:
        capabilities:
          drop:
          - all
```

```
kubectl apply -f .
kubectl get pods
kubectl logs nocap-alpine
kubectl exec -it nocap-alpine -- sh
```

```
ping www.google.de
wget -O - http://www.google.de
```

Hacking and exploration session HostPID

Disable ServiceLinksEnable false

- https://github.com/jmetzger/training-kubernetes-security-en/blob/main/security/by.layer/pods-container/enableServiceLinks/disable-howto-and-why.m

Great but still alpha User Namespaces

Reaction

The Audit Logs

Hints (this is on a system, where kubernetes-api-server runs as static pod

- When the config unter /etc/kubernetes/manifests/kupe-apiserver.yaml changes
  - kubelet automatically detects this and restarts the server
  - if there is a misconfig the pod will vanish

```
## There are 4 stages, that can be monitored:
```

- `RequestReceived` - The stage for events generated as soon as the audit handler receives the request, and before it is delegated down the handler chain.
- `ResponseStarted` - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- `ResponseComplete` - The response body has been completed and no more bytes will be sent.
- `Panic` - Events generated when a panic occurred.

Step 1: 1st -> session (on control plane): Watch kube-apiserver - pod on controlplane

```
## we want to
watch crictl pods | grep api
```

Step 2: 2nd -> session (on control plane): create a policy

```
cd /etc/kubernetes
```

```
nano audit-policy.yaml
```

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
## Don't generate audit events for all requests in RequestReceived stage.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
      # Resource "pods" doesn't match requests to any subresource of pods,
      # which is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
      resourceNames: ["controller-leader"]

  # Don't log watch requests by the "system:kube-proxy" on endpoints or services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs:
    - "/api*" # Wildcard matching.
    - "/version"

  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
```

```
  - group: "" # core API group
    resources: ["configmaps"]
  # This rule only applies to resources in the "kube-system" namespace.
  # The empty string "" can be used to select non-namespaced resources.
  namespaces: ["kube-system"]

# Log configmap and secret changes in all other namespaces at the Metadata level.
- level: Metadata
  resources:
  - group: "" # core API group
    resources: ["secrets", "configmaps"]

# Log all other resources in core and extensions at the Request level.
- level: Request
  resources:
  - group: "" # core API group
  - group: "extensions" # Version of group should NOT be included.

# A catch-all rule to log all other requests at the Metadata level.
- level: Metadata
  # Long-running requests like watches that fall under this rule will not
  # generate an audit event in RequestReceived.
  omitStages:
    - "RequestReceived"
```

```
## Important: You do not need to apply/create it.
```

Step 3: 2nd -> session: Change settings in /etc/kubernetes/manifests/kube-apiserver.yaml

```
## security copy
cp /etc/kubernetes/manifests/kube-apiserver.yaml /root
```

```
## Add lines in /etc/kubernetes/manifests/kube.apiserver.yaml
- --audit-log-path=/var/log/kubernetes/apiserver/audit/audit.log
- --audit-policy-file=/etc/kubernetes/audit-policies.yaml
```

```
## Add lines under volumeMounts
- mountPath: /etc/kubernetes/audit-policy.yaml
  name: audit
  readOnly: true
- mountPath: /var/log/kubernetes/apiserver/audit/
  name: audit-log
  readOnly: false
```

```
## Add volumes lines under volumes
- name: audit
  hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File

- name: audit-log
  hostPath:
    path: /var/log/kubernetes/audit/
    type: DirectoryOrCreate
```

Step 4: 1st -> session

```
POD ID               CREATED              STATE               NAME                            NAMESPACE          A
TTEMPT              RUNTIME
27bfc9f8d1552        7 seconds ago        Ready               kube-apiserver-controlplane     kube-system        0
                    (default)
43e9ba82707c0        2 hours ago          Ready               csi-node-driver-28l2j           calico-system      1
```

c

Step 5: 2nd -> session

```
## There should be enought noise already
cat /var/log/kubernetes/audit/audit.log
```

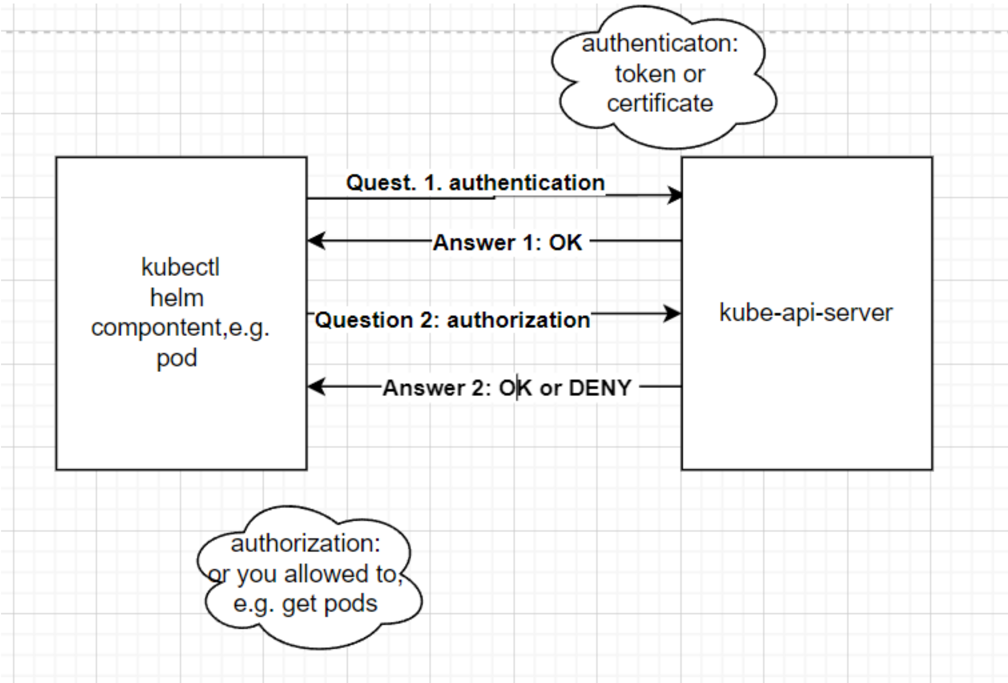
Reference

- https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/

RBAC

How does RBAC work ?

- Let us see in a picture



Where does RBAC play a role ?

Users -> kube-api-server

- User how want to access the kube api server

Components -> kube-api-server

- e.g. kubelet -> kube-api-server

Pods / System Pods -> kube-api-server

- Pods and System Pods (e.g. kube-proxy a.ka. CoreDNS) how want to access the kube-api-server

kubeconfig decode certificate

```
cd
cd .kube
cat config
## copy client-certificate-data
```

```
echo
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURLVENDQWhHZ0F3SUJBZ0lJVGpPWWowbXpWMFl3RFFZSktvWklodmNOQVFFTEJRQXdGVEVUTUJFR0ExVUUKQXhNS2Ez
 base64 -d > out.crt
openssl x509 -in out.crt -text -noout
```

kubectl check your permission - can-i

A specific command

```
kubectl auth can-i get pods
```

List all

```
kubectl auth can-i --list
```

use kubectl in pod - default service account

Walkthrough

```
kubectl run -it --rm kubectltester --image=alpine -- sh
```

```
## in shell
apk add kubectl
## it uses in in-cluster configuration in folder
```

```
## /var/run/secrets/kubernetes.io/serviceaccount
kubectl auth can-i --list
```

create user for kubeconfig with using certificate

Step 0: create an new rolebinding for the group (we want to use)

```
kubectl create rolebinding developers --clusterrole=view --group=developers
```

Step 1: on your client: create private certificate

```
cd
mkdir -p certs
## create your private key
openssl genrsa -out ~/certs/jochen.key 4096
```

Step 2: on your client: create csr (certificate signing request)

```
nano ~/certs/jochen.csr.cnf
```

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn
[ dn ]
CN = jochen
O = developers
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
```

```
## Create Certificate Signing Request
openssl req -config ~/certs/jochen.csr.cnf -new -key ~/certs/jochen.key -nodes -out ~/certs/jochen.csr
openssl req -in certs/jochen.csr --noout -text
```

Step 3: Send approval request to server

```
## get csr (base64 decoded)
cat ~/certs/jochen.csr | base64 | tr -d '\n'
```

```
cd certs
nano jochen-csr.yaml
```

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: jochen-authentication
spec:
  signerName: kubernetes.io/kube-apiserver-client
  groups:
    - system:authenticated
  request:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJRWF6Q0NBbE1DQVFBd0pqRVBNQTBHQTFVRUF3d0dhbTlqYUdWdU1STXdFUVlEVlFRS0RBcGtaWFpsYkc5

  usages:
  - client auth
```

```
kubectl apply -f jochen-csr.yaml
kubectl get -f jochen-csr.yaml
## show me the current state -> pending
kubectl describe -f jochen-csr.yaml
```

Step 4: approve signing request

```
kubectl certificate approve jochen-authentication
## or:
kubectl certificate approve -f jochen-csr.yaml

## see, that it is approved
kubectl describe -f jochen-csr.yaml
```

Step 5: get the approved certificate to be used

```
kubectl get csr jochen-authentication -o jsonpath='{.status.certificate}' | base64 --decode > ~/certs/jochen.crt
```

Step 6: construct kubeconfig for new user

```
cd
cd certs
```

```
## create new user
kubectl config set-credentials jochen --client-certificate=jochen.crt --client-key=jochen.key
```

```
## add a new context
kubectl config set-context jochen --user=jochen --cluster=kubernetes
```

Step 7: Use and test the new context

```
kubectl config use-context jochen
kubectl get pods
```

Ref:

- https://kb.leaseweb.com/kb/users-roles-and-permissions-on-kubernetes-rbac/kubernetes-users-roles-and-permissions-on-kubernetes-rbac-create-a-certificate-based-kubeconfig/

practical exercise rbac

Schritt 1: Create a service account and a secret

```
cd
mkdir -p manifests/rbac
cd manifests/rbac
```

Mini-Step 1: definition of the user

```
## vi service-account.yml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: training
  namespace: default
```

```
kubectl apply -f service-account.yml
```

Mini-Step 1.5: create Secret

- From Kubernetes 1.25 tokens are not created automatically when creating a service account (sa)
- You have to create them manually with annotation attached
- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/#create-token

```
## vi secret.yml
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: trainingtoken
  annotations:
    kubernetes.io/service-account.name: training
```

```
kubectl apply -f .
```

Mini-Schritt 2: ClusterRole creation - Valid for all namespaces but it has to get assigned to a clusterrolebinding or rolebinding

```
### Does not work, before there is no assignment
## vi pods-clusterrole.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pods-clusterrole
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list", "create"]
```

```
kubectl apply -f pods-clusterrole.yml
```

Mini-Schritt 3: Assigning the clusterrole to a specific service account

```
## vi rb-training-ns-default-pods.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rolebinding-ns-default-pods
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pods-clusterrole
subjects:
- kind: ServiceAccount
  name: training
  namespace: default
```

```
kubectl apply -f rb-training-ns-default-pods.yml
```

Mini-Step 4: Test it (does the access work)

```
kubectl auth can-i get pods -n default --as system:serviceaccount:default:training
kubectl auth can-i --list --as system:serviceaccount:default:training
```

Schritt 2: create Context / read Credentials and put them in kubeconfig (the Kubernetes-Version 1.25. way)

Mini-Step 1: kubeconfig setzen

```
kubectl config set-context training-ctx --cluster kubernetes --user training

## extract name of the token from here

TOKEN=`kubectl get secret trainingtoken -o jsonpath='{.data.token}' | base64 --decode`
echo $TOKEN
kubectl config set-credentials training --token=$TOKEN
kubectl config use-context training-ctx

## Hier reichen die Rechte nicht aus
kubectl get deploy
## Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:kube-system:training" cannot list # resource
"pods" in API group "" in the namespace "default"
```

Mini-Step 2:

```
kubectl config use-context training-ctx
kubectl get pods
```

Mini-Step 3: back to the old context

```
kubectl config get-contexts
```

```
kubectl config use-context cluster-admin@kubernetes
```

Refs:

- https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengaddingserviceaccttoken.htm
- https://microk8s.io/docs/multi-user
- https://faun.pub/kubernetes-rbac-use-one-role-in-multiple-namespaces-d1d08bb08286

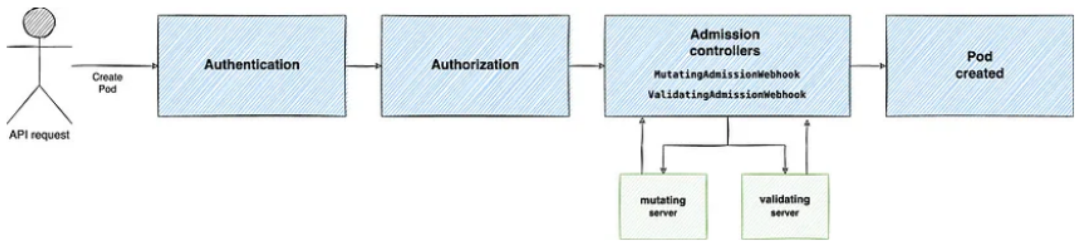
Ref: Create Service Account Token

- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/#create-token

Obey Security Policies (AdmissionControllers)

Admission Controller

What does it do ? (The picture)



How do the docs describe it ?

```
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.
```

- intercepts request = gets all the requests and validates or changes (=mutates them)
- Reference: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

What kind of admissionControllers do we have ?

- Mutating and Validating
- There are 2 phases of the AdmissionControlProcess: First mutating, then validating

The static admissionPlugins

- There are static admissionPlugins which are activated by config
- You can see the activated like so

```
## in our system like so.
kubectl -n kube-system describe pods kube-apiserver-controlplane  | grep enable-adm
```

```
--enable-admission-plugins=NodeRestriction
```

- There are some that are activated by default:

```
## in Kubernetes 1.31
CertificateApproval,
CertificateSigning,
CertificateSubjectRestriction,
DefaultIngressClass,
DefaultStorageClass,
DefaultTolerationSeconds,
LimitRanger,
MutatingAdmissionWebhook,
NamespaceLifecycle,
PersistentVolumeClaimResize,
PodSecurity,
Priority,
ResourceQuota,
RuntimeClass,
ServiceAccount,
StorageObjectInUseProtection,
TaintNodesByCondition,
ValidatingAdmissionPolicy,
ValidatingAdmissionWebhook
```

- Reference: What does which AdmissionController (= AdmissionPlugin) do ? https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do

Exercise with PSA

Seit: 1.2.22 Pod Security Admission

- 1.2.22 - Alpha Feature, was not activated by default. need to activate it as feature gate (Kind)
- 1.2.23 - Beta -> probably

Predefined settings

- privileges - no restrictions
- baseline - some restriction
- restricted - really restrictive
- Reference: https://kubernetes.io/docs/concepts/security/pod-security-standards/

Practical Example starting from Kubernetes 1.23

```
mkdir -p manifests
cd manifests
mkdir psa
cd psa
nano 01-ns.yml
```

```
## Step 1: create namespace


apiVersion: v1
kind: Namespace
metadata:
  name: test-ns1
  labels:
    # soft version - running but showing complaints
    # pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

```
kubectl apply -f 01-ns.yml
```

```
## Schritt 2: Testen mit nginx - pod
## vi 02-nginx.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns1
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
```

```
## a lot of warnings will come up
## because this image runs as root !! (by default)
kubectl apply -f 02-nginx.yml
```

```
## Schritt 3:
## Change SecurityContext in  Container

## vi 02-nginx.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns1
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
```

```
kubectl delete -f 02-nginx.yml
kubectl apply -f 02-nginx.yml
kubectl -n test-ns1 get pods
```

```
## Schritt 4:
## Weitere Anpassung runAsNotRoot
## vi 02-nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns<tln>
```

```
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        runAsNonRoot: true
```

```
## pod kann erstellt werden, wird aber nicht gestartet
kubectl delete -f 02-nginx.yml
kubectl apply -f 02-nginx.yml
kubectl -n test-ns1 get pods
kubectl -n test-ns1 describe pods nginx
```

```
## Schritt 4:
## Anpassen der Sicherheitseinstellung (Phase1) im Container
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-ns1
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        runAsNonRoot: true
        allowPrivilegeEscalation: false
        capabilities:
          drop: ["ALL"]
```

```
kubectl delete -f 02-nginx.yml
kubectl apply -f 02-nginx.yml
kubectl -n test-ns1 get pods
```

Praktisches Beispiel für Version ab 1.2.23 -Lösung - Container als NICHT-Root laufen lassen

- Wir müssen ein image, dass auch als NICHT-Root laufen kann
- .. oder selbst eines bauen (;o)) o bei nginx ist das bitnami/nginx

```
## vi 03-nginx-bitnami.yml
apiVersion: v1
kind: Pod
metadata:
  name: bitnami-nginx
  namespace: test-ns1
spec:
  containers:
    - image: bitnami/nginx
      name: bitnami-nginx
      ports:
        - containerPort: 80
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        runAsNonRoot: true
```

```
## und er läuft als nicht root
kubectl apply -f 03_pod-bitnami.yml
kubectl -n test-ns1 get pods
```

OPA Gatekeeper 01-Overview

How does it work ?

- It is called by the definition in mutationAdmissionWebhook and validatingAdmissionWebhook

What can the OPA Gatekeeper do ?

- It can validate

- It can mutate

How does it do this ?

- It uses a language which is called REGO
- It uses objects like

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate

apiVersion: templates.gatekeeper.sh/v1
kind: Constraint
```

- for the validation

OPA Gatekeeper 02-Install with Helm

Step 1: Installation (helm)

```
helm repo add gatekeeper https://open-policy-agent.github.io/gatekeeper/charts
helm upgrade gatekeeper gatekeeper/gatekeeper --install  --namespace gatekeeper-system --create-namespace
```

Step 2: Webhooks (lookaround)

- This create a mutation and a validationWebhook

```
kubectl get validatingwebhookconfigurations gatekeeper-validating-webhook-configuration
kubectl get mutatingwebhookconfigurations gatekeeper-mutating-webhook-configuration

```

- Let's look in the mutation more deeply

```
kubectl get mutatingwebhookconfiguration gatekeeper-mutating-webhook-configuration -o yaml
```

Step 3: The components

```
## controllers are the endpoint for the webhooking
## audit is done every 60 seconds in the audit-pod
kubectl -n gatekeeper-system get all
```

OPA Gatekeeper 03-Simple Exercise

Step 1: Create constraintTemplate

- I took this from the library: https://open-policy-agent.github.io/gatekeeper-library/website/
- https://open-policy-agent.github.io/gatekeeper-library/website/validation/block-nodeport-services/

```
cd
mkdir -p manifests
cd manifests
mkdir restrict-node-port
cd restrict-node-port
```

```
nano 01-constraint-template.yaml
```

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sblocknodeport
  annotations:
    metadata.gatekeeper.sh/title: "Block NodePort"
    metadata.gatekeeper.sh/version: 1.0.0
    description: >-
      Disallows all Services with type NodePort.

      https://kubernetes.io/docs/concepts/services-networking/service/#nodeport
spec:
  crd:
    spec:
      names:
        kind: K8sBlockNodePort
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sblocknodeport

        violation[{"msg": msg}] {
          input.review.kind.kind == "Service"
```

```
          input.review.object.spec.type == "NodePort"
          msg := "User is not allowed to create service of type NodePort"
        }
```

```
kubectl apply -f .
```

Step 2: Create constraint

- it is like an instance (in code = usage of classes, can be created multiple times)
- the match defines, when it triggers -> when it calls the constraintTemplate for validation

```
nano 02-constraint.yaml
```

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sBlockNodePort
metadata:
  name: block-node-port
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Service"]
```

```
kubectl apply -f .
```

Step 3: Test constraint with Service

```
nano 03-service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: my-service-disallowed
spec:
  type: NodePort
  ports:
    - port: 80
```

```
kubectl apply -f .
## should no appear
kubectl get svc my-service-disallowed
```

```
Error from server (Forbidden): error when creating "03-service.yaml":
admission webhook "validation.gatekeeper.sh" denied the request:
[block-node-port] User is not allowed to create service of type NodePort
```

OPA Gatekeeper 04-Example-Job-Debug

Step 1: Create constraintTemplate

```
cd
mkdir -p manifests
cd manifests
mkdir blockjob
cd blockjob
```

```
nano 01-constraint-template.yaml
```

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sblockjob
  annotations:
    metadata.gatekeeper.sh/title: "Block Job"
    metadata.gatekeeper.sh/version: 1.0.0
    description: >-
      Blocks certain jobs
spec:
  crd:
    spec:
      names:
        kind: K8sBlockJob
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
```

```
        package k8sblockjob

        violation[{"msg": msg}] {
          # input.review.kind.kind == "Job"
          msg1:= sprintf("Data: %v", [input.review.userInfo])
          msg2:= sprintf("JOBS not allowed .. REVIEW OBJECT: %v", [input.review])

          msg:= concat("",[msg1,msg2])
        }
```

```
kubectl apply -f .
## Was it sucessfully parsed and compiled ?
kubectl describe -f 01-constraint-template.yaml
```

Step 2: Create constraint

```
nano 02-constraint.yaml
```

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sBlockJob
metadata:
  name: block-job
spec:
  match:
    kinds:
 # Important batch not Batch, Batch will not work
      - apiGroups: ["batch"]
        kinds: ["Job"]
```

```
kubectl apply -f .
```

Step 3: Test with Job

```
nano 03-job.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

```
Should not work:
```

```
kubectl apply -f .
```

Step 4: Let's try from a pod

- Prepare user

```
kubectl create sa podjob
kubectl create rolebinding podjob-binding --clusterrole=cluster-admin --serviceaccount=default:podjob
kubectl run -it --rm jobmaker --image=alpine --overrides='{"spec": {"serviceAccount": "podjob"}}' -- sh
```

```
## in pod install kubectl and nano
apk add nano kubectl
## create pod manifests
cd
nano job.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
```

```
      - name: pi
        image: perl:5.34.0
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

```
## should not work
kubectl apply -f pod.yaml
```

Connaisseur: Verifying images before Deployment

Prerequisites

- You must have create a private/public key pair
- You must have signed some images in your registry on docker Hub.
- This was done here: [Signing an image with cosign]

Step 1: Install Connaisseur with helm

```
cd
mkdir -p manifests
cd manifests
mkdir connaisseur
cd connaisseur
```

```
nano values.yaml
```

```
## 1. We will add the public key of ours in validators: cosign->type:cosign
## 2. We will add a policy for the system to know, when to use it:
  - pattern: "docker.io/dockertrainereu/*:*"
      validator: cosign
## Unfortunately we muss add everything from the defaults values file
## concerning -> validators, policy
## I have tested this ....
```

```
application:
  features:
      namespacedValidation:
            mode: validate

## validator options: https://sse-secure-systems.github.io/connaisseur/latest/validators/
  validators:
    - name: allow
      type: static
      approve: true
    - name: deny
      type: static
      approve: false
    - name: cosign
      type: cosign
      trustRoots:
        - name: default
          key: |
            -----BEGIN PUBLIC KEY-----
            MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE2LmRkI8sNi6fSluqU5UEisptlwZl
            YaJBAJqTf96ccM4R3MstL8PfR5fhy877TG7bnpc4YnlfejT6F7XE71FWkA==
            -----END PUBLIC KEY-----
    - name: dockerhub
      type: notaryv1
      trustRoots:
        - name: default # root from dockerhub
          key: |
            -----BEGIN PUBLIC KEY-----
            MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEOXYta5TgdCwXTCnLU09W5T4M4r9f
            QQrqJuADP6U7g5r9ICgPSmZuRHP/1AYUfOQW3baveKsT969EfELKj1lfCA==
            -----END PUBLIC KEY-----
        - name: sse # root from sse
          key: |
            -----BEGIN PUBLIC KEY-----
            MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEsx28WV7BsQfnHF1kZmpdCTTLJaWe
            d0CA+JOi8H4REuBaWSZ5zPDe468WuOJ6f71E7WFg3CVEVYHuoZt2UYbN/Q==
            -----END PUBLIC KEY-----

  policy:
    - pattern: "*:*"
      validator: deny
    - pattern: "docker.io/dockertrainereu/*:*"
```

```
      validator: cosign
    - pattern: "docker.io/library/*:*"
      validator: dockerhub
    - pattern: "docker.io/securesystemsengineering/*:*"
      validator: dockerhub
      with:
        trustRoot: sse
    - pattern: "registry.k8s.io/*:*"
      validator: allow
```

```
## Add the repo
helm repo add connaisseur https://sse-secure-systems.github.io/connaisseur/charts
## Install the helm chart
helm upgrade connaisseur connaisseur/connaisseur --install --create-namespace --namespace connaisseur -f values.yaml
```

Step 2: Create a namespace and label it with connaisseur

- In our example we apply it only in a specific namespace
- But you can also use it for all namespaces
- According to: https://sse-secure-systems.github.io/connaisseur/latest/features/namespaced_validation/

```
## create namespace
kubectl create ns app1
kubectl label ns app1 securesystemsengineering.connaisseur/webhook=validate
```

Step 3: Try to run image in namespace

```
## image from docker -> works
kubectl -n app1 run nginxme --image=nginx:1.23

## signed image from dockertrainereu
kubectl -n app1 run pod1 --image=dockertrainereu/alpine-rootless:1.20

## unsigned image from dockertrainereu
kubectl -n app1 run pod1 --image=dockertrainereu/pinger
```

Pod Security

Automount ServiceAccounts or not ?

Why ?

- Every attacker tries to get as much information as possible
- Although there are not severe permissions in here, show as little information as possible
- For example, use will see, which namespace he is in ;o)

Disable ?

```
## enabled by default
kubectl explain pod.spec.automountServiceAccountToken
```

Unprivilegierte Pods/Container

Which images to use ?

docker hub

- bitnami images
- Also search for unprivileged -> e.g. https://hub.docker.com/search?q=unprivileged
  - BUT: be careful whom you trust

The SecurityContext

appArmor example

- https://kubernetes.io/docs/tutorials/security/apparmor/

Network Policies

Exercise NetworkPolicies

Step 1: Create Deployment and Service

```
SHORT=jm
kubectl create ns policy-demo-$SHORT
```

```
cd
mkdir -p manifests
```

```
cd manifests
mkdir -p np
cd np
```

```
nano 01-deployment.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.23
        ports:
        - containerPort: 80
```

```
kubectl -n policy-demo-$SHORT apply -f .
```

```
nano 02-service.yml
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: ClusterIP # Default Wert
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl -n policy-demo-$SHORT apply -f .
```

Step 2: Testing access without any rules

```
## Run a 2nd pod to access nginx
kubectl run --namespace=policy-demo-$SHORT access --rm -ti --image busybox
```

```
## Within the shell/after prompt
wget -q nginx -O -
```

```
## Optional: Show pod in second 2. ssh-session on jump-host
kubectl -n policy-demo-$SHORT get pods --show-labels
```

Step 3: Define policy: no access is allowed by default (in this namespace)

```
nano 03-default-deny.yaml
```

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: default-deny
spec:
  podSelector:
    matchLabels: {}
```

```
kubectl -n policy-demo-$SHORT apply -f .
```

Step 4: Test connection with deny all rules

```
kubectl run --namespace=policy-demo-$SHORT access --rm -ti --image busybox
```

```
## Within the shell
wget -q nginx -O -
```

Step 5: Allow access vof pods with the Label run=access (all pods startet with run with the name access, do have this label by default)

```
nano 04-access-nginx.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
    - from:
      - podSelector:
          matchLabels:
            run: access
```

```
kubectl -n policy-demo-$SHORT apply -f .
```

Schritt 5: Test it (access should work)

```
## Start a 2nd pod to access nginx
## becasue of run->access pod automtically has the label run:access
kubectl run --namespace=policy-demo-$SHORT access --rm -ti --image busybox
```

```
## innerhalb der shell
wget -q nginx -O -
```

Step 6: start Pod with label run=no-access - this should not work

```
kubectl run --namespace=policy-demo-$SHORT no-access --rm -ti --image busybox
```

```
## in der shell
wget -q nginx -O -
```

Step 7: Cleanup

```
kubectl delete ns policy-demo-$SHORT
```

Ref:

- https://projectcalico.docs.tigera.io/security/tutorials/kubernetes-policy-basic

CNI Benchmarks

- https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-40gbit-s-network-2024-156f085a5e4e

ServiceMesh

Why a ServiceMesh ?

What is a service mesh ?

```
A service mesh is an infrastructure layer
that gives applications capabilities
like zero-trust security, observability,
and advanced traffic management, without code changes.
```

Advantages / Features

1. Observability & monitoring
2. Traffic management
3. Resilience & Reliability
4. Security
5. Service Discovery

Observability & monitoring

- Service mesh offers:
  - valuable insights into the communication between services
  - effective monitoring to help in troubleshooting application errors.

Traffic management

- Service mesh offers:
  - intelligent request distribution
  - load balancing,
  - support for canary deployments.
  - These capabilities enhance resource utilization and enable efficient traffic management

Resilience & Reliability

- By handling retries, timeouts, and failures,
  - service mesh contributes to the overall stability and resilience of services
  - reducing the impact of potential disruptions.

Security

- Service mesh enforces security policies, and handles authentication, authorization, and encryption
  - ensuring secure communication between services and eventually, strengthening the overall security posture of the application.

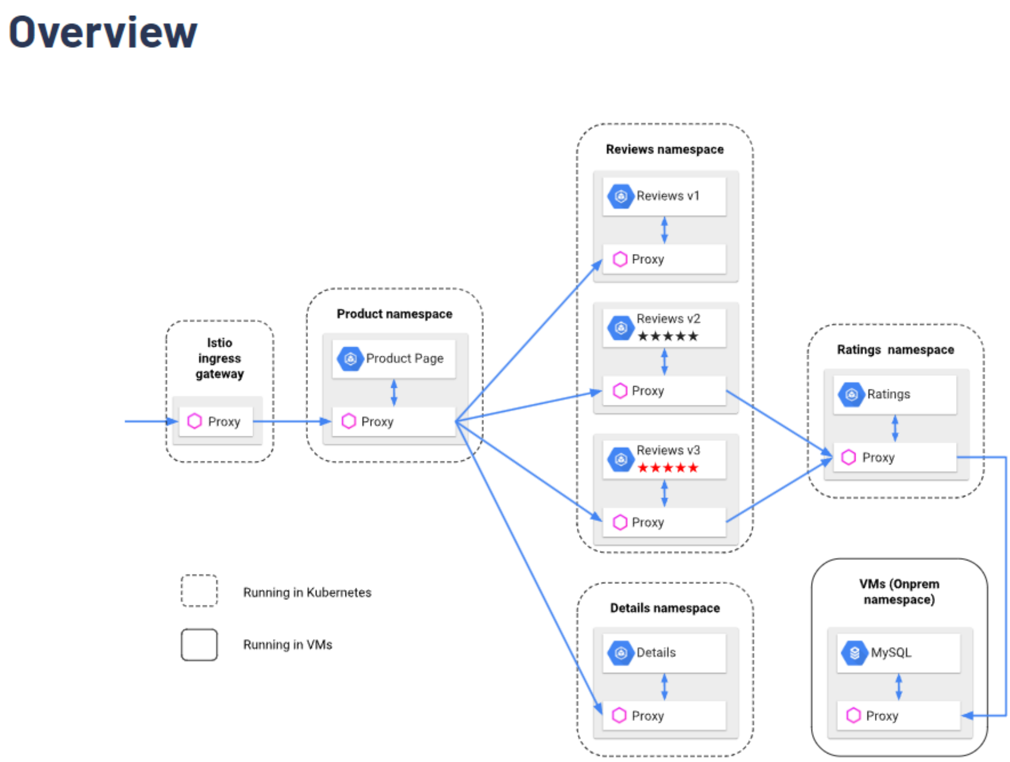
Service Discovery

- With service discovery features, service mesh can simplify the process of locating and routing services dynamically
- adapting to system changes seamlessly. This enables easier management and interaction between services.

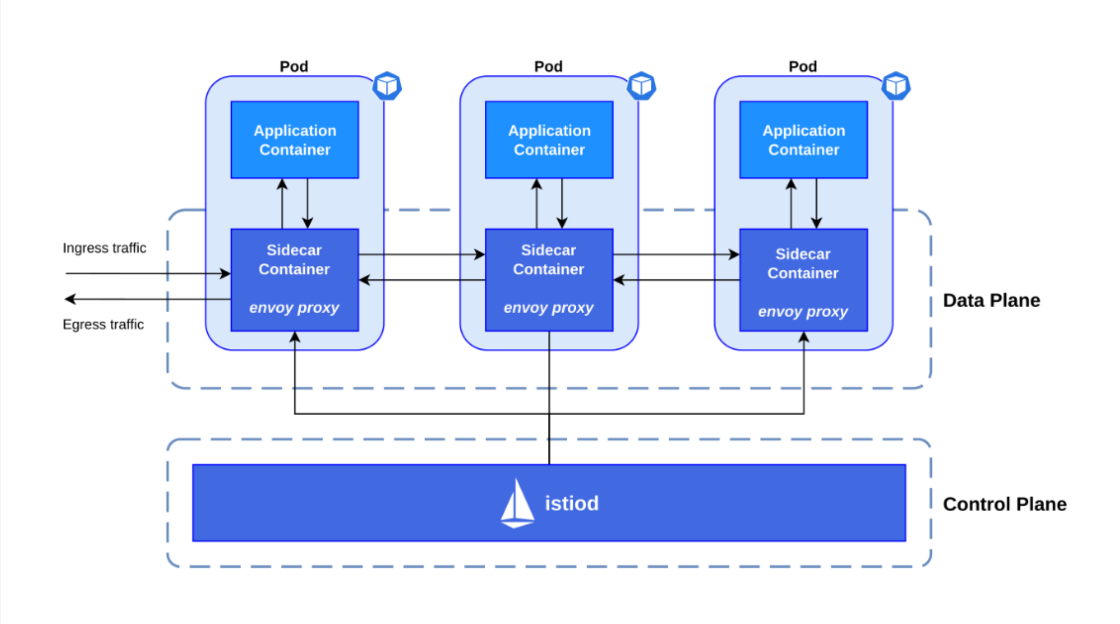
Overall benefits

```
Microservices communication:
Adopting a service mesh can simplify the implementation of a microservices architecture by abstracting away infrastructure
complexities.
It provides a standardized approach to manage and orchestrate communication within the microservices ecosystem.
```

How does a ServiceMeshs work? (example istio



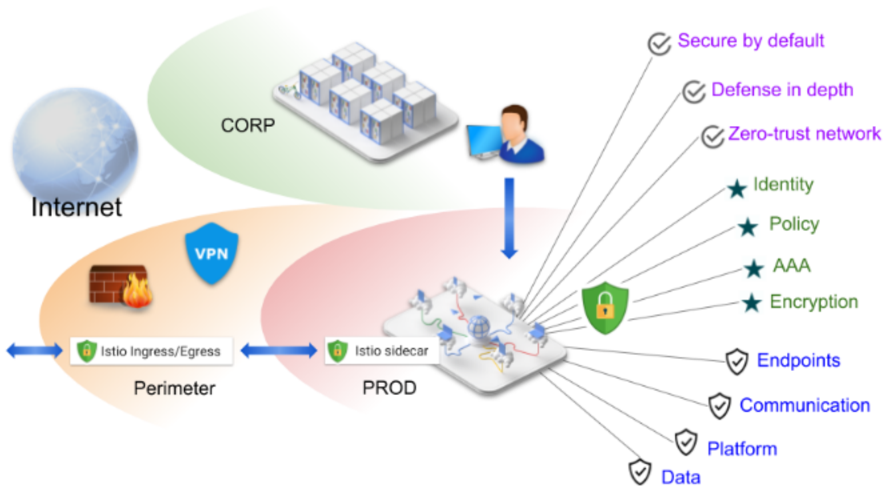
Istio control plane and data plane



- Source: kubebyexample.com

istio security features

Overview



Security overview

Security needs of microservices

- To defend against man-in-the-middle attacks, they need traffic encryption.
- To provide flexible service access control, they need mutual TLS and fine-grained access policies.
- To determine who did what at what time, they need auditing tools.

Implementation of security

The Istio security features provide strong identity, powerful policy, transparent TLS encryption, and authentication, authorization and audit (AAA) tools to protect your services and data. The goals of Istio security are:

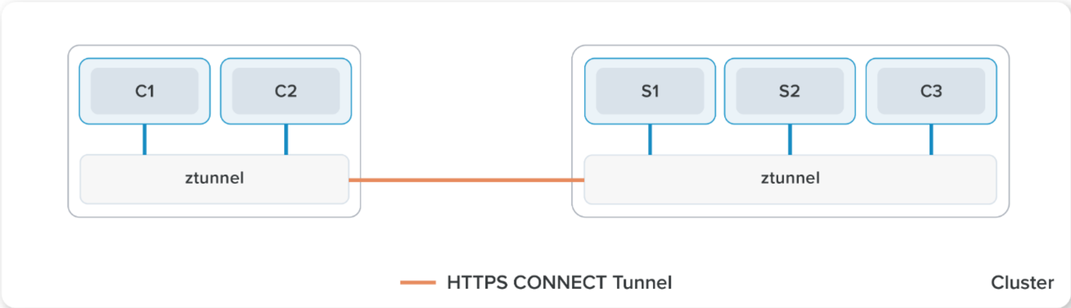
- Security by default: no changes needed to application code and infrastructure
- Defense in depth: integrate with existing security systems to provide multiple layers of defense
- Zero-trust network: build security solutions on distrusted networks

istio-service mesh - ambient mode

Light: Only Layer 4 per node (ztunnel)

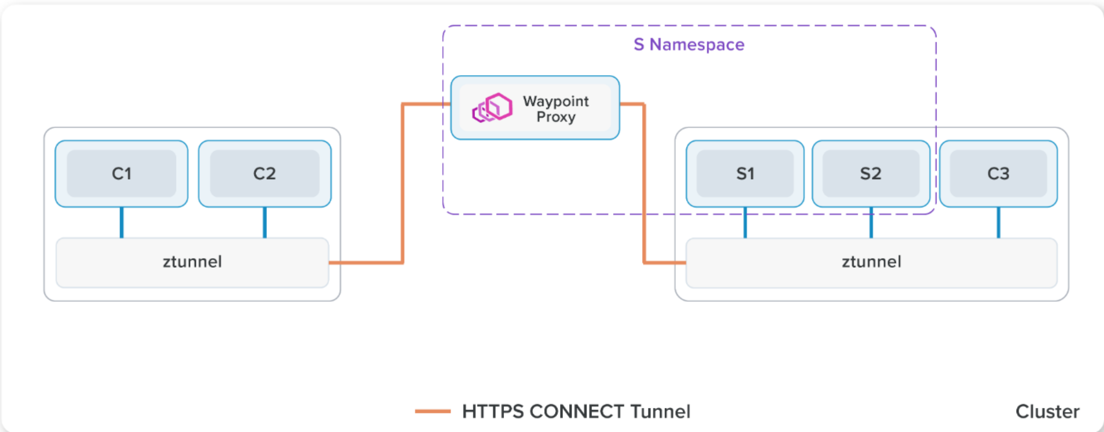
- No sidecar (envoy-proxy) per Pod, but one ztunnel agent per Node (Layer 4)
- Enables security features (mtls, traffic encryption)

Like so:



Full fledged: Layer 4 (ztunnel) per Node & Layer 7 per Namespace (

- One waypoint - proxy is rolled out per Namespace, which connects to the ztunnel agents



When additional features are needed, ambient mesh deploys waypoint proxies, which ztunnels connect through for policy enforcement

Features in "fully-fledged" - ambient - mode

Application deployment use case	Ambient mode configuration
Zero Trust networking via mutual-TLS, encrypted and tunneled data transport of client application traffic, L4 authorization, L4 telemetry	ztunnel only (default)
As above, plus advanced Istio traffic management features (including L7 authorization, telemetry and VirtualService routing)	ztunnel and waypoint proxies

Advantages:

- Less overhead
- One can start step-by-step moving towards a mesh (Layer 4 firstly and if wanted Layer 7 for specicfic namespaces)
- Old pattern: sidecar and new pattern: ambient can live side by side.

Performance comparison - baseline,sidecar,ambient

- https://livewyer.io/blog/2024/06/06/comparison-of-service-meshes-part-two/
- https://github.com/livewyer-ops/poc-servicemesh2024/blob/main/docs/test-report-istio.md

Image Security

When to scan ?

When to scan ?

- In Development
- When Building Software (before pushing to the registry)
- Before Deploying Software
- In Production
- Ongoing in the registry itself

Conceptional Ideas

1. Scan the image when built before being pushed
2. In Kubernetes Cluster only images from your corporate private registry and k8s.io (Option: Only allow signed images, where signing is verified)
3. (Policy) To always pull when starting a new Deployment,Pod,Statefulset (private repo)

Why ?

- We want to be sure, our system is not compromised
- One way of compromising is an malicious image, that we use
- We want to avoid this.

Which approach do we take here....

- For our own images, that we build, we want to be sure, they are all "clean" before being pushed to the registry

Example Image Security Scanning - using gitlab and trivy

Pre-Thoughts

- Gitlab offers a security scanner based on Trivy
- BUT: This scanner tests already uploading images
- If we think about ShiftLeft-approach (Security early) on, this might not be the best option

What needs to be done ?

1. We want to scan directly after building the image, but before pushing
2. If we have vulnerabilities with CRITICAL-Score (CVE), the pipeline will fail (and it stops)
   - Image is not uploaded

Trivy modes

- Trivy can be used standalone or as client/server
- in our case, we will use it standalone

Demonstration:

- https://gitlab.com/jmetzger/container-scanning-session

```
stages:
  - prebuild
  - build

prebuild:
  stage: prebuild
  image:
    name: docker.io/curlimages/curl:8.3.0
    entrypoint: [""]
  script:
```

```
    - curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b $CI_PROJECT_DIR v0.45.1
    - curl -L --output - https://github.com/google/go-containerregistry/releases/download/v0.16.1/go-
containerregistry_Linux_x86_64.tar.gz | tar -xz crane
  artifacts:
    paths:
      - crane
      - trivy

build image:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:v1.23.2-debug
    entrypoint: [""]
  variables:
    DOCKER_IMAGE: "${CI_REGISTRY_IMAGE}:${CI_COMMIT_SHORT_SHA}"
    TRIVY_INSECURE: "true"
    TRIVY_NO_PROGRESS: "true"
  script:
    - /kaniko/executor
      --context $CI_PROJECT_DIR
      --dockerfile $CI_PROJECT_DIR/Dockerfile
      --no-push
      --tar-path image.tar
    - ./trivy image
      --ignore-unfixed
      --exit-code 0
      --severity HIGH
      --input image.tar
    - ./trivy image
      --ignore-unfixed
      --exit-code 1
      --severity CRITICAL
      --input image.tar
    - ./crane auth login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - ./crane push image.tar $DOCKER_IMAGE
```

References:

- https://bluelight.co/blog/how-to-set-up-trivy-scanner-in-gitlab-ci-guide
- https://gitlab.com/bluelightco/blog-examples/trivy

Hacking Sessions

Hacking with HostPID

Extras

Canary deployment with basic kubernetes mechanisms

Phase 1: stable application (without canary)

```
cd
cd manifests
mkdir ab
cd ab
```

```
## vi 01-cm-version1.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-version-1
data:
  index.html: |
    <html>
    <h1>Welcome to Version 1</h1>
    </br>
    <h1>Hi! This is a configmap Index file Version 1 </h1>
    </html>
```

```
## vi 02-deployment-v1.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy-v1
spec:
  selector:
    matchLabels:
      version: v1
```

```
  replicas: 20
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
            - name: nginx-index-file
              mountPath: /usr/share/nginx/html/
      volumes:
      - name: nginx-index-file
        configMap:
          name: nginx-version-1
```

```
## vi 05-svc.yml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    svc: nginx
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: nginx
```

```
kubectl apply -f .
kubectl run -it --rm podtest --image=busybox
```

```
## wget -O - http://my-nginx.default.svc.cluster.local
while [ true ]; do wget -O - http://my-nginx.default.svc.cluster.local; done
```

Step 2: Additional Deployment on top (only 2 vs. 20)

```
nano 03-cm-version2.yml
```

```
## vi 03-cm-version2.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-version-2
data:
  index.html: |
    <html>
    <h1>Welcome to Version 2</h1>
    </br>
    <h1>Hi! This is a configmap Index file Version 2 </h1>
    </html>
```

```
nano 04-deployment-v2.yml
```

```
## vi 04-deployment-v2.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy-v2
spec:
  selector:
    matchLabels:
      version: v2
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
        version: v2
    spec:
```

```
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
            - name: nginx-index-file
              mountPath: /usr/share/nginx/html/
      volumes:
      - name: nginx-index-file
        configMap:
          name: nginx-version-2
```

```
kubectl apply -f .
kubectl run -it --rm podtest --image=busybox
```

```
## wget -O - http://my-nginx.default.svc.cluster.local
while [ true ]; do wget -O - http://my-nginx.default.svc.cluster.local; done
```

Documentation

Great video about attacking kubernetes - older, but some stuff is still applicable

- https://www.youtube.com/watch?v=HmoVSmTIOxM

Straight forward hacking session of kubernetes

- https://youtu.be/iD_klswHJQs?si=97rWNuAbGjLwCjpa

github with manifests for creating bad pods

- https://bishopfox.com/blog/kubernetes-pod-privilege-escalation#pod8