Auditing

Auditing in Kubernetes refers to the process of recording and analyzing activities that occur on the cluster. This can include actions taken by users, API requests, and changes to objects in the cluster. Auditing provides visibility into the behavior of the cluster and can be used for security, compliance, and troubleshooting purposes

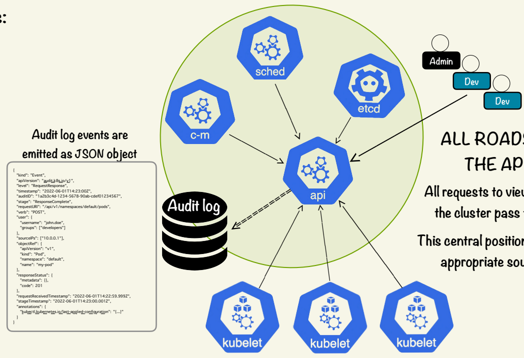
Audit levels in Kubernetes define the verbosity of the recorded events. There are four audit levels:

- **None**: Do not log any events.
- **Metadata**: Log request metadata only (e.g., who, what, where, when).
- **Request**: Log event metadata and request content (excluding the response).
- **RequestResponse**: Log event metadata, request content, and response content.

Memory consumption depends on the audit logging policy

Audit logging increases the memory consumption of the API Server

Audit log events are emitted as JSON object



ALL ROADS LEAD TO ... THE APISERVER

All requests to view or modify the state of the cluster pass through the apiserver

This central position makes the apiserver the appropriate source for auditing data

To enable audit logging in the Kubernetes API server, you need to configure the API server to use a specific audit policy and write audit logs to a file or other destination

1. Edit the API server configuration file, Add the following flags to the spec.containers.command section

```
- --audit-policy-file=/etc/kubernetes/audit/policy.yaml
- --audit-log-path= /var/log/kubernetes/audit/audit.log
- --audit-log-format=json
- --audit-log-maxsize=500
- --audit-log-maxbackup=3
```

The apiserver has some audit logging options:

audit-policy-file: sets the policy file to use

audit-log-*: setting configure log files

audit-webhook-*: settings configure log network endpoints

2. Add a volume and volumeMount to the spec section

```
volumes:
- name: audit-config
  hostPath:
    path: /etc/kubernetes/audit/policy.yaml
    type: File
- name: audit-logs
  hostPath:
    path: /var/log/kubernetes/audit
    type: DirectoryOrCreate
```

3. Add the corresponding volume mounts to the spec.containers.volumeMounts section

```
- name: audit-config
  mountPath: /etc/kubernetes/audit/policy.yaml
  readOnly: true
  subPath: audit-policy.yaml
- name: audit-logs
  mountPath: /var/log/kubernetes/audit
```

4. After the API server restarts and applies the policy.yaml file, you can tail the logs to see the events being recorded

```
tail -f /var/log/kubernetes/audit/audit.log | jq
```

Audit policy is a configuration that defines the rules for what events should be recorded and at what level

/etc/kubernetes/audit/policy.yaml

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
  resources:
  - group: ""
    resources: ["pods", "services"]
- level: Request
  users: ["system:serviceaccount:my-namespace:my-serviceaccount"]
  resources:
  - group: ""
    resources: ["configmaps"]
```

This policy logs metadata for all pod and service operations and logs request content for configmap operations performed by the specified service account

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  - level: Request
    resources:
    - group: ""
      resources: ["pods", "services"]
    verbs: ["create", "update", "delete"]
  - level: Metadata
    resources:
    - group: ""
      resources: ["namespaces", "configmap", "secrets"]
    verbs: ["create", "update", "delete"]
  - level: None
    resources:
    - group: ""
      resources: ["persistentvolumes", "persistentvolumeclaims"]
```

This policy will log request-level for pod and service creation, update, and deletion, also will log metadata-level events for namespace, configmap, and secret creation, update, and deletion. It will not log events related to persistent volume and persistent volume claim resources.

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:

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

- level: None
  resources:
  - group: "" # core
    resources: ["events"]

- level: None
  users:
  - "system:kube-scheduler"
  - "system:kube-proxy"
  - "system:apiserver"
  - "system:kube-controller-manager"
  - "system:serviceaccount:gatekeeper-system:gatekeeper-admin"

- level: None
  userGroups: ["system:nodes"]

- level: RequestResponse
```

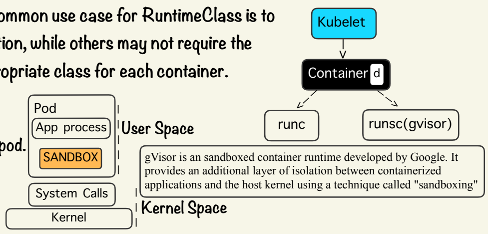
This policy has five rules, each specifying a different level of audit logging:

- specifies that all "get", "watch", and "list" operations should not be audited at all.
- specifies that events should not be audited at all.
- specifies these certain system users should not be audited
- specifies that all users belonging to the "system:nodes" group should not be audited.
- specifies that all other operations, including requests and responses, should be audited at the RequestResponse levels

RuntimeClass

RuntimeClass is a Kubernetes feature that allows users to specify different runtime configurations for their containers. One common use case for RuntimeClass is to run containers with different levels of isolation. For example, a user may want to run some containers with a higher level of isolation, while others may not require the same level of security. By defining multiple RuntimeClasses with different runtime configurations, the user can choose the appropriate class for each container.

gVisor is a user-space kernel that provides isolation for containers by intercepting and handling system calls. It can be used with k8s to provide an extra layer of security for your pods. To restrict syscalls for a pod running in k8s, you can use gVisor as the runtime for that pod.



How to use gVisor: Install gVisor → Configure containerd → Create a RuntimeClass resource → Use the gVisor RuntimeClass in your pod

First, you need to install gVisor on your Kubernetes nodes. You can do this using the runsc binary, which is the gVisor runtime. Download and install the runsc binary on each node:

```
wget https://storage.googleapis.com/gvisor/releases/nightly/latest/runsc
chmod +x runsc
sudo mv runsc /usr/local/bin
```

To use gVisor with Kubernetes, you need to configure the container runtime (e.g., containerd) to use gVisor. Create a configuration file for containerd:

```
sudo mkdir -p /etc/containerd
sudo nano /etc/containerd/config.toml
```

Add this configuration to the `config.toml` file:

```toml
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]
  runtime_type = "io.containerd.runsc.v1"
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc.options]
    BinaryName = "/usr/local/bin/runsc"
    Root = ""
    LogLevel = "info"
    Debug = false
    DebugLogFile = ""
    NoSandbox = false
```

Restart containerd to apply the new configuration:

```
sudo systemctl restart containerd
```

create a `RuntimeClass` resource in your Kubernetes cluster that specifies gVisor as the runtime. Save the following YAML file as `gvisor-runtime-class.yaml`:

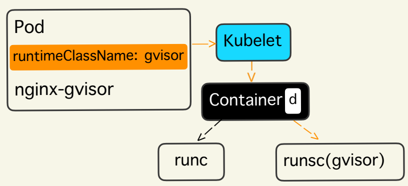
```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

```
kubectl apply pply -f gvisor-runtime-class.yaml
```

To use gVisor for a specific pod, set the `runtimeClassName` field to `gvisor` in the pod spec. Here's an example of a simple Nginx pod that uses gVisor:

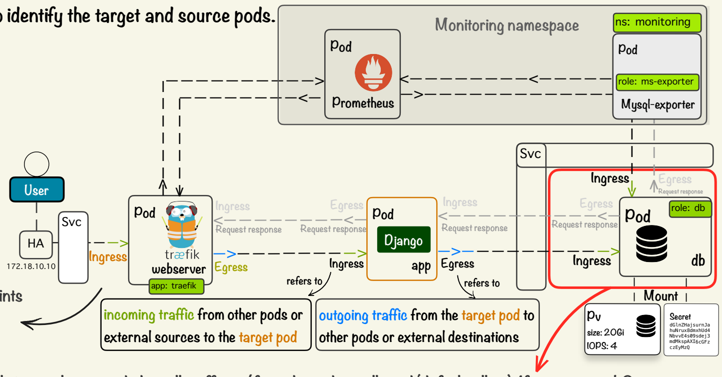
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-gvisor
spec:
  runtimeClassName: gvisor
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

```
kubectl apply pply -f nginx-gvisor-pod.yaml
```



Network policy

Network Policy is a Kubernetes feature that allows you to define rules for ingress and egress traffic between pods inside a cluster. It's a way to implement security and access control at the network level by specifying which pods can communicate with each other, using labels to identify the target and source pods.



features

- Policies are namespace scoped
- Policies are applied to pods using label selectors
- Policy rules can specify the traffic that is allowed to/from pods, namespaces, or CIDRs
- Policy rules can specify protocols (TCP, UDP, SCTP), named ports or port numbers

Network policies are applied to pods rather than services because pods are the network endpoints that actually receive the traffic. Services are not network endpoints and do not receive traffic directly. Instead, they route traffic to the appropriate pods based on their labels.

If no Kubernetes network policies apply to a pod, then all traffic to/from the pod are allowed (default-allow). If one or more k8s network policies apply to a pod, then only the traffic specifically defined in that network policy are allowed (default-deny)

Network policies are like firewall rules for your Kubernetes pods. By default, pods are non-isolated and can accept traffic from any source. When you apply a NetworkPolicy to a pod, that pod becomes isolated and only allows traffic that is permitted by the policy. There are several types of Network Policy rules that can be defined in Kubernetes:

PodSelector: This rule selects a specific set of pods to apply the policy to based on their labels.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: django
    ports:
    - protocol: TCP
      port: 3306
```

Allow ingress traffic from pods in the same namespace

This selects the pods to which the NetworkPolicy applies. In this case, it matches all pods with the label role: db

specifies that the policy only applies to ingress traffic

The "ingress" section specifies the traffic rules that govern inbound traffic to the selected pods. Specifically, it permits traffic from pods labeled with "role: django" to access the selected pods on TCP port 3306.

Ports filed allows you to specify the ports and protocols that are allowed for incoming or outgoing traffic.

NamespaceSelector: This rule selects all the pods in a specific namespace to apply the policy to.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy-namespace
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: ms-exporter
      namespaceSelector:
        matchLabels:
          ns: monitoring
  ports:
  - port: 3306
```

Allow ingress traffic from pods in a different namespace

This rule only allows inbound traffic from pods labeled "role: ms-exporter" in the "ns: monitoring" namespace. Incoming traffic is limited to port number 3306

ExternalEntities: This rule allows you to define specific IP addresses or IP ranges that are allowed to communicate with the selected pods

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: traefik-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: traefik
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.18.0.0/24
  egress:
  - to:
    - podSelector:
        matchLabels:
          role: django
```

specifies that the policy applies to both Ingress and Egress traffic

specifies that pods with the label "app: traefik" can only receive traffic from the IP block "172.18.0.0/24" and can only send traffic to pods with the label "role: django".

Please note that in order to use Network Policies, you must have a CNI (Container Network Interface) that supports them, such as Calico or Weave Net.

Security Context

SecurityContext is a configuration object that defines the security settings for a Pod or a specific container within a Pod. It allows you to set the access control and security-related properties for the containers, including their file system, users, and groups, as well as the capabilities and privileges of the processes running inside the containers.

SecurityContext object can be defined at the Pod level or at the container level, using the securityContext field in the Pod or container specification

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      runAsUser: 1000
      runAsGroup: 2000
      fsGroup: 3000
      readOnlyRootFilesystem: true
```

container will run as the user ID 1000, the group ID 2000, and have its filesystem owned by the group ID 3000. Additionally, the container's root filesystem will be read-only, which can help to improve security by preventing changes to critical system files.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      privileged: true
```

the privileged field is set to true, which means that the container will run in privileged mode

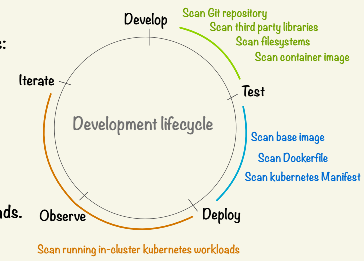
```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        drop:
        - CHOWN
      allowPrivilegeEscalation: false
```

the capabilities field is used to specify the Linux capabilities that the container is allowed to use. Here, the container is allowed to use the NET_ADMIN capability, but is not allowed to use the CHOWN capability.

Additionally, the allowPrivilegeEscalation field is set to false, which means that the container is not allowed to escalate privileges beyond what is specified in the SecurityContext object

Image security

Trivy is a simple and comprehensive vulnerability scanner for containers. It's used to identify vulnerabilities in operating system packages (Alpine, Red Hat Universal Base Image, CentOS, etc.) & application dependencies (Bundler, Composer, npm, yarn, etc.). It's especially useful in the Kubernetes (k8s) environment for scanning container images and ensuring your workloads are secure. Here's how Trivy can be integrated into different stages of Kubernetes deployment:



Pre-deployment Scanning:

Before deploying your workloads, you can use Trivy to scan various resources for vulnerabilities and misconfigurations. Here are some common use cases:

- **Third-party Libraries**: Scan your application's dependencies and libraries for known vulnerabilities.
- **Container Images**: Scan container images for vulnerabilities in the underlying operating system packages and application dependencies
- **Git Repositories**: Analyze your code repositories for secrets, sensitive information, or other security issues.

You can use the Trivy CLI on your local machine or integrate Trivy into your CI/CD pipeline to perform these pre-deployment scans. Trivy will provide you with a list of vulnerabilities and misconfigurations to address before deploying your workloads

Continuous Scanning of Running Workloads:

After deploying your workloads to Kubernetes, it's essential to set up automated and continuous scanning to detect vulnerabilities in your running workloads. Here are the recommended features for this stage:

- **Trivy K8s Command**: Use the trivy kubernetes command to scan Kubernetes Deployments or Namespaces. Trivy will scan the container images used by the running Pods and provide vulnerability reports
- **Trivy Operator**: Deploy the Trivy Operator in your Kubernetes cluster. The Trivy Operator automates the scanning of running workloads by continuously monitoring and scanning container images within the cluster

```
trivy k8s --namespace=kube-system --report=summary deploy
Summary Report for minikube

Workload Assessment
```

Namespace	Resource	Vulnerabilities C	H	M	L	U	Misconfigurations C	H	M	L	U	Secrets C	H	M	L	U
kube-system	Deployment/metrics-server								2	8						
kube-system	Deployment/coredns		1					1	3	5						
kube-system	Deployment/logviewer	2							4	11						

Severities: C=CRITICAL H=HIGH M=MEDIUM L=LOW U=UNKNOWN