Kubernetes Security topics

whoami

- DevOps Engineer @ KloudOps
- Father of two big cats and soon to be husband.
- Minimalism, automation, opensource, to break things.
- Music, mountains, vie ferrate, bodybuilding.

- Github: github.com/maxgio92
- Twitter: twitter.com/maxgio92
- Medium: medium.com/@maxgio92

Table Of Contents

Who can talk with me?

Authentication

Authentication

Verify and **confirm** that the presented **identity** is actually the real one.

Is the main **prevention** against **spoofing** which is:

- **pretending** to be an **identity** who **you are not**.

In Kubernetes

Identities in Kubernetes

User

- **Cluster**-wide
- Not for human users
- Used by the K8s components (e.g.: the kubelet)

Notes on human users:

- Kubernetes does not have objects which represent human users
- Human users are assumed to be managed externally (use delegated authentication)

Service account

- Bound to **namespaces**
- Users managed by the Kubernetes API
- Created automatically by the API server or manually through API calls
- Tied to a set of credentials stored as Secrets
  - which are mounted into pods allowing in-cluster processes to talk to the Kubernetes API

Identities make requests to API server

A **request** can be sent from **inside** or **outside** the cluster:

- Human user typing kubectl
- Kubelets on nodes
- Members of the control plane

Authenticated API **requests** are tied to:

- User, or
- Service account, or
- Nothing (anonymous)

How the API server authenticates?

Authentication plugins

To authenticate API request Kubernetes uses authentication plugins

Authentication plugins attempt to **associate** the following **attributes** with the **request**:

- **username**: a string which **identifies** the end user
- **UID**: a string which **identifies uniquely** and consistently the end user
- **groups**: a set of strings which **associate users** with a **set of grouped user**
- **extra fields**: a map of **additional information** useful for authorizers

All **values** only hold **significance** when interpreted by an **authorizer.**
All **values** are **opaque** to the **authenticator**.

x509 certificates

x509 certs

Use case:

among Kubernetes components

(and not only)

x509 provides a **standard** for certificates format to claim an identity, for which the Certicate Authority acts as **guarantor** in a **Public Key Infrastructure** (PKI).

Kubernetes requires **PKI** as so:

From	To	Client cert	Kubeconfig	Server cert
kubelet	kube-apiserver	YES	YES	/
*	kube-apiserver	/	/	YES
administrator	kube-apiserver	YES	YES	/
kube-apiserver	kubelet	YES	/	/
kube-apiserver	etcd	YES	/	/
kube-controller-manager	kube-apiserver	YES	YES	/
kube-scheduler	kube-apiserver	YES	YES	/
etcd	etcd	YES	/	YES

The **CommonName** of the certificate is used then as the **username**.

CA protection

You can **manage** the **root CA externally**, if you don’t want to inject the root CA, by creating and **injecting only the intermediate CAs** for:

- kubernetes (general)
- etcd
- front-proxy

under `/etc/kubernetes/pki`, and the key pair for Service Account signing.

CA offloading with Vault

You could leverage Hashicorp **Vault** with the **PKI Secrets Engine** to **inject**:
- **intermediate CA** certificates with shorter TTL (< 1 year),
  - which in case of certificates for admin users could be too much.

```
$ vault secrets enable -path=kubernetes-pki -description="Kubernetes certificate chain" pki

$ vault write kubernetes-pki/roles/admin-role \
    allowed_domains="system:kube-scheduler" \
    organization="system:kube-scheduler" \
    enforce_hostnames=false \
    allow_bare_domains=true \
    server_flag=false \
    client_flag=true \
    ttl=1h \
    max_ttl=2h

$ vault write kubernetes-pki/issue/admin-role common_name=system:kube-scheduler
```

Bearer tokens

Service Account token

Use case:

from workload to API server

It's an authenticator plugin **enabled by default** that uses **signed bearer tokens** to verify requests.

The **signer private key** is the which one specified to the API server argument:

```
$ kube-apiserver --service-account-key-file=<sa.key>
```

If not specified, the API server signs the SA tokens with its **private key**.

Service accounts are **automatically created** and **associated** to Pods via the `ServiceAccount` **admission controller**, via the specified `serviceAccountName` field in the **pod** `spec`, otherwise the `default` is used.

Service Account

A `default` Service account is automatically created and associated to Pods via the `ServiceAccount` admission controller.

Custom `ServiceAccount` can be request to be associated via the specified `serviceAccountName` field in the Pod `spec`; otherwise the `default` is used.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
```

Token mount

The token is mounted by default from the **Secret** `tmpfs` volume associated with the **ServiceAccount** object into pods.

The **mout point** is at well-known location:
`/var/run/secrets/kubernetes.io/serviceaccount`

The **authenticator** associates:

- Username:
  `system:serviceaccount:<namespace>:<serviceaccount>`
- Groups:
  - `system:serviceaccounts`
  - `system:serviceaccounts:<namespace>`

Automatically revoke deleted tokens

```
$ kube-apiserver --service-account-lookup
```

Deny authN by default

Disable explicitely token auto-mount

That's ok to enforce policies at the **authorization layer**, but the **authentication layer** remains **opened** by default.

Not all services needs to authenticate to API server.

Per Pod:

`spec.automountServiceAccountToken: false`

Per ServiceAccount:

`automountServiceAccountToken: false`

Disable by default token auto-mount

`Bad request`

…or actually `Good request`?

FR #57601

Enable to disable by default token auto-mount

TL;DR

- "The more layers we have to prevent an attack the better"
- "If an attacker is able to somehow hack that layer of authz, he has full access"
- "/reopen"
- "Disabling by default is not backwards compatible, so is not a realistic option until (if) a v2 Pod API is made"
- [..] workaround with Initializers [...]
- "Creating a new namespace auto-creates a new service token that is set to automount so it's not a viable solution."
- "/close"

Non-official prevention:

Karydia

github.com/karydia/karydia

Kubernetes add-on that **inverts** via mutating admission webhook the following **insecure default** settings:

- disable by default service account **token auto-mount**
- restrict **system calls** by adding a seccomp profile
- disallow **privilege escalation:**
  - `securityContext allowPrivilegeEscalation=false`
- restrict **network** communication:
  - provides three levels of custom network policy

Limits:
- beta
- K8s >= 1.15 (yes EKS, you can...)

On top of Kubernetes: workload authentication

Between workload services

To establish **trust** between the **parties**, below the application layer a **mTLS** with **PKI** is there for you.

With mTLS and a proper PKI in place you establish **trust** and **mutual authentication** between your **services**, especially for zero trust networks.

As you don't have kubeadm for application components you need a system to:

- **provide** the certificates
- **update** the certificates
- **revoke** the certificates
- update the **pods** accordingly
  - client keypair
  - trusted CA certs

mTLS with cert-manager

CSI

The Container Storage Interace is a **standard** for **exposing** arbitrary **block** and **file storage systems** to **containerized workloads** on container orchestrators.

Third-party storage **providers** can write and deploy **plugins** exposing **new storage systems** in Kubernetes without ever having to touch the core Kubernetes code.

A CSI **driver** is a **storage plugin** that can **honor volume requests** specified on Pods (like Secret or ConfigMap volume drivers).

cert-manager CSI driver

An experimental **CSI driver** to manage x509 **ephemeral** keypairs **unique** to each pod.

The cert-manager CSI driver leverages the ephemeral **inline volumes** with **CSI** driver as the **lifecycle** of the certificate key pair must match that of the Pod.

Features:

- All **private keys** are stored locally on the node and **never leave** the **node**
- **Unique keypair** per application **replica**
- Certificate **request** spec in-line of the **Pod template**
- Automatic **renewal** of certificates
- Automatic **destroy** of keypair on Pod’s termination

cert-manager CSI driver

How does it work?

The **driver** is deployed as a privileged **DaemonSet**.

The **Kubelet** will send a **NodePublishVolume** call (with Pods details from the in-line volume) to the node's **driver**.

The **driver** will generate a **private key** and a **CSR**.

The **driver** will create a **CertificateRequest** resource and **cert-manager** will returns a **signed certificate**.

The **driver** will write the resulting **keypair** to that node's **file system** to be **mounted** to the Pod's FS.

On **renewal** the driver simply **overwrites** the existing certificate in path.

When the **Pod** is marked for **termination**:

- the **Kubelet** will send a **NodeUnpublishVolume** call to the **driver**
- the **driver** will **destroy** the **keypair** from the node's **filesystem**

```
kind: Deployment
spec:
  template:
    spec:
      containers:
        - name: my-frontend
          image: busybox
          volumeMounts:
          - mountPath: "/tls"
            name: tls
          command: [ "sleep", "1000000" ]
      volumes:
        - name: tls
          csi:
            driver: csi.cert-manager.io
            volumeAttributes:
                  csi.cert-manager.io/issuer-name: ca-issuer
                  csi.cert-manager.io/dns-names: my-
service.sandbox.svc.cluster.local
```

This CSI driver plugin makes use of the **CSI Inline Volume** feature (beta in v1.16). Kubernetes version < 1.16 requires the following feature gate set:

```
$ kube-apiserver --feature-gates=CSIInlineVolume=true
```

But…

what if I also offload the handshake from application?

Service mesh

mTLS with service mesh

The term **service mesh** is used to describe the **network of microservices** that make up such applications and the **interactions** between them.

As it grows in **size** and **complexity**, it becomes **harder** to **manage** and also the **requirements grow**:
- end-to-end authentication
- access control
- observability
- service discovery
- load balancing
- canary rollouts / A/B testing
- etc.

Most **service mesh frameworks** are here to help satisfy these **requirements**.

AUTHENTICATION

mTLS with Istio

How does it work

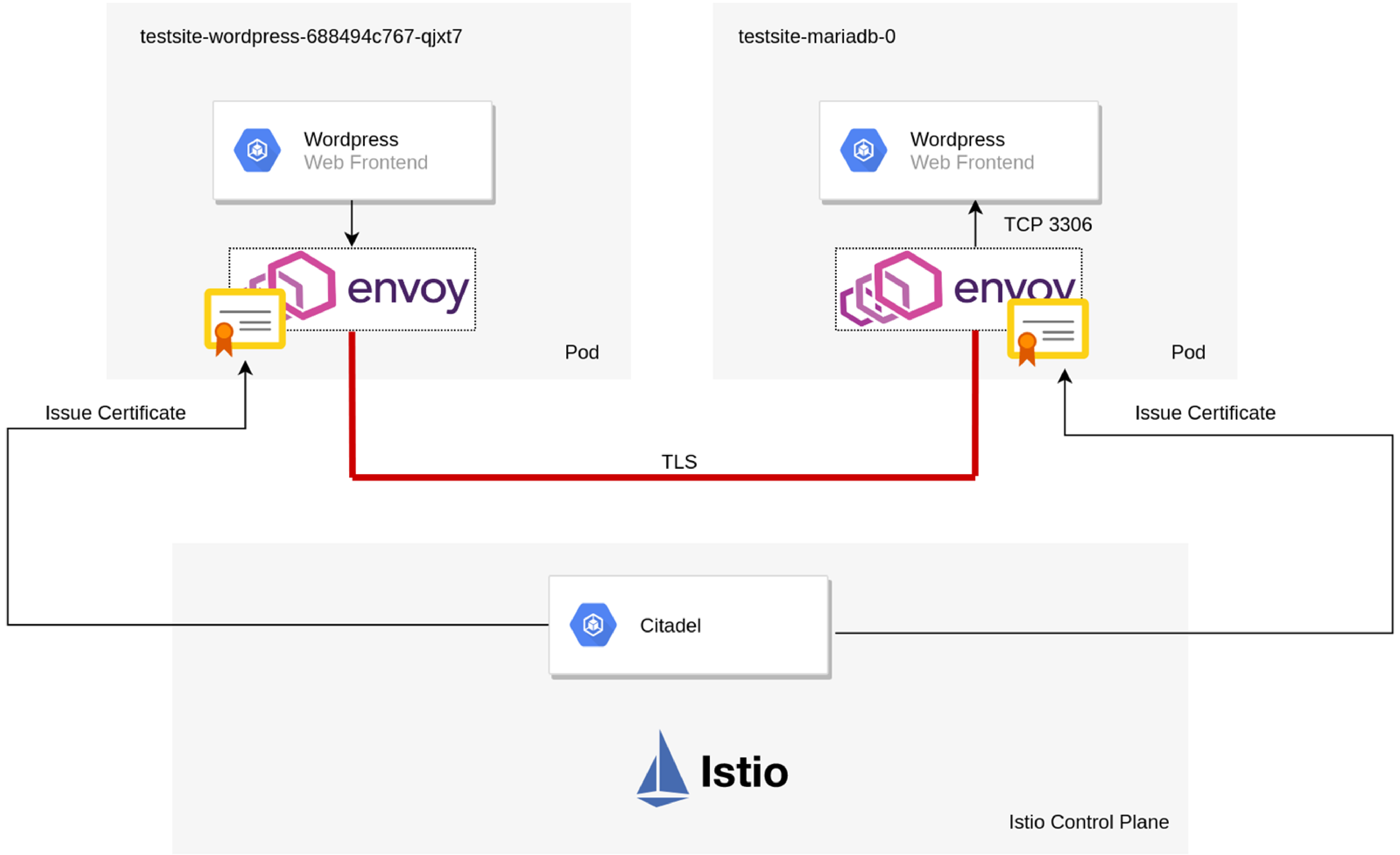
The Istio **traffic management model** relies on **Envoy** proxy, deployed as a **sidecar**.

All the data plane **traffic** is **proxied** through it.

The **traffic** can be **authenticated** and **encrypted trasparently**, other than directed and controlled.

Istio other than JWT-based request-level authentication offers **peer authentication**, used for **service-to-service authentication** via mTLS.

When **enabled** between two **services**, they **start** and **receive** the requests in **plain text** because **Envoy** manages the **TLS handshake**.



To start **enforcing** mTLS incrementally, you need:

- server-side: `Peer Authentication Policy`
- client-side: `Destination Rule`

Server side: Peer Authentication Policy

Peer Authentication Policies apply to **requests** that a service **receives**.

You decide the **mode** Istio enforces mTLS:

- **STRICT**: require mTLS
- **PERMISSIVE**: accept both mTLS and plain from out-of-mesh services
- **DISABLED**: never require mTLS

You can enforce peer authentication **globally** or **incrementally** with policies, by selecting the scope:

- mesh-wide (globally)
- namespace-wide
- workload-specific

```
# All workloads with the “app=reviews” label
# in namespace “foo” must use mutual TLS.

apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: example-peer-policy
  namespace: foo # defines the scope
spec:
  selector: # defines the scope
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT # defines the mode
```

Is applied the **narrowest** policy + this priority to each service:

- service-specific
- namespace-wide
- mesh-wide

Client side: Destination Rules

Destination rules and **virtual services** are a **key** part of Istio's **traffic management**.

- you use **virtual services** to decide **how you route** your traffic to a destination

- you use **destination rules** to decide **what happens to traffic** for that destination

Destination rules are applied **after** virtual service routing, so they apply to the traffic's **"real" destination**.

They let you configure Envoy's **traffic policies** such as **TLS mode**:

- **DISABLE**: plain text request
- **SIMPLE**: TLS connection with client-side only authn
- **MUTUAL**: mTLS with your own certs in place
- **ISTIO_MUTUAL**: mTLS with Istio generated certificates

```yaml
# It configures a client to use Istio
# mutual TLS when talking to rating services

apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: ratings-istio-mtls
spec:
  host: ratings.prod.svc.cluster.local
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL # Istio manages the PKI
```

Ok, now that I can trust him, what can he do?

Access control

Access control

The **selective restriction** of **access** to a **resource**.

Is used to prevent **information disclosure** and **tampering**.

In relation, authorization is the act of **specifying access privileges** to **resources**.

In Kubernetes I want to focus on access to critical data, starting from the main interface endpoint of Kubernetes.

API server

API server

The **permissions** to access to API **objects** itself are managed via **authorization methods** in Kubernetes.

But we can implement **access restriction** to the API server **from L2**:

- we should restrict the **availability** of the API Server endpoint to a private network
  - enable tunneling (e.g.: with VPN)
- we should enforce **access restriction** also with **firewall rules**
  - up to L7

Secret API

Secret API

You should apply **least privilege** and **deny by default** with RBAC authorization policies, as:

- Secrets can be used to **gain extra privileges**, as for **Service Account token Secrets**:
  - **pods** that can access Service Account token Secrets could run with **elevated permissions** if those Service Accounts are granted access to **permissive PSPs**
  - **allow** for `list` (verb) on **Secrets** allows to **inspect** the **values**
  - a **user** who **can't read** a **Secret** but **can create pods** could expose that Secret by via those pods
- `list` and `watch` all Secrets in a cluster should be reserved only to **trusted** system **components**
- `list` and `watch` all Secrets in a namespace should be avoided
- only `get` requests on the needed Secrets should be allowed to let applications access them.

deltatre

Secret API and the Kubelet

The **Kubelet** can **read any Secret** from the API server, so:

- being able to **impersonate** the Kubelet **enables you** to read all the Secrets

So enforce **restriction** with `Node Authorization`, which is an **authorization mode** that specifically **authorizes only** the API **requests** that the **kubelet** needs.

In order to be authorized by the Node authorizer, kubelets must use a credential that identifies them as:

- being in the `system:nodes` group
- with a username of `system:node:<nodeName>`

The value of `<nodeName>` must **match** the name of the **node** as **registered** by the kubelet

- from hostname
- or cloud provider metadata API when `--cloud-provider` is specified

To **enable** the Node authorizer, start the apiserver with:

```
$ kube-apiserver --authorization-mode=Node
```

You can further **review** kubelet’s **permissions** with the `NodeRestriction` admission controller:

```
$ kube-apiserver --enable-admission-plugins=...,NodeRestriction
```

Secret Volume

Secret Volume

A Secret volume is used to pass sensitive information to Pods.

- A Secret is **sent** to a node and **stored** on tmpfs only if a Pod on that **node requires** it
- once the **pod** that **depends** on the **Secret** is **deleted**, the **kubelet** will **delete** its **local copy** of the secret data as well
- only the **Secrets** that a **pod** requests are potentially visible within its **containers**
- only **containers** that request the **Secret volume** to be **mounted** can **access** it.

The **prevention** can be made **by design**:

- apply **single responsibility** + **least privilege** principles
- do not **expose** service that have **access** to **Secrets** as much as possible

Secret Volume

Enforce ACL with external providers

You can integrate external providers like **Vault**.

Hashicorp provides official **integration** with Vault-k8s that **mimics** the native Kubernetes Secret volume behaviour, with:

- **injector** as init container to **inject** Vault secrets
- **Vault Agent as sidecar** to **keep in sync** Vault secrets, mounted on a shared **tmpfs** volume (memory `EmptyDIr`) on your pods

so that the application access to Vault secrets trasparenlty, by **reading** from filesystem, without being aware of Vault.

External providers AND native integration: CSI drivers

Official

FR #7365: github.com/hashicorp/vault/issues/7365

But the hard part is the authentication.

Unofficial

A solution is KubeVault by AppsCode, which provides:

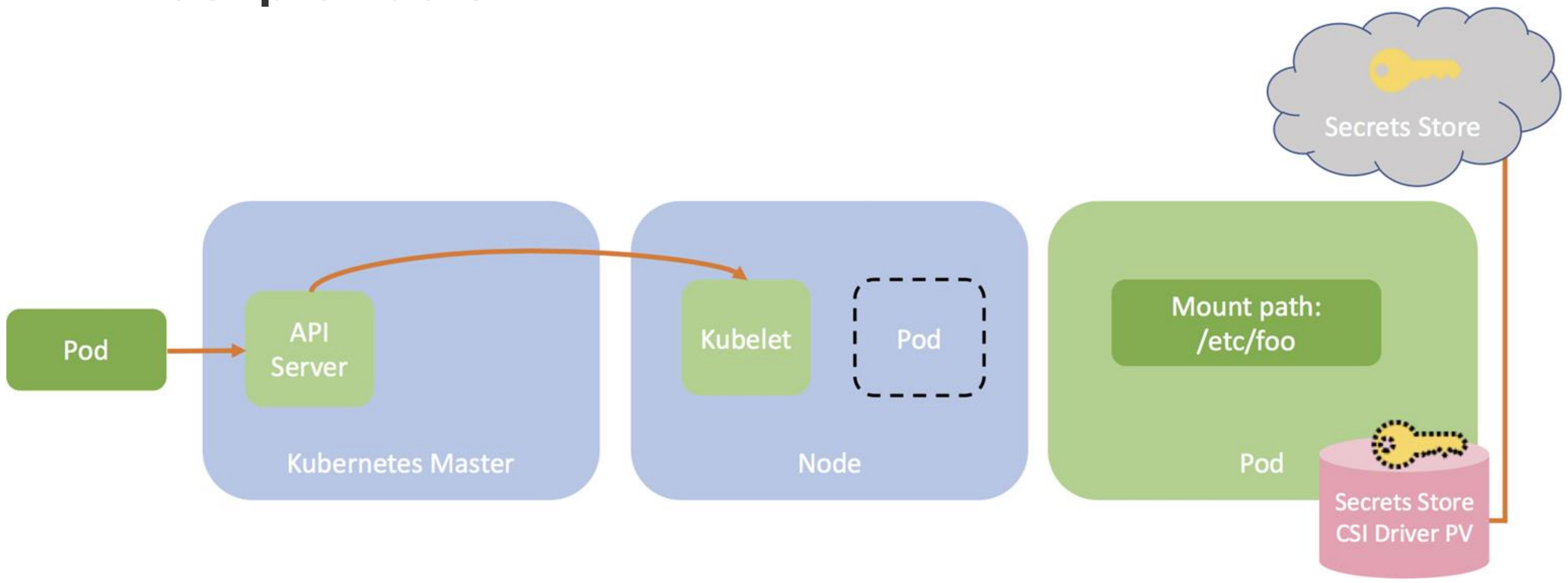
- a Vault Operator
- a Container Storage Interface (CSI) driver

Secret Volume

External provider AND more native integration with the Secrets Store CSI driver

You can better integrate using also **inline CSI volumes** (ephemeral) with the Sig’s **Secrets Store CSI driver** which:

- integrates **external secrets stores** with native **CSI volumes**
- in combination with third-party secrets stores as **CSI providers**.



It **mounts** secrets/keys/certs to pod using a **CSI volume**

It **supports**:

- **CSI Inline volume**
- mounting **multiple** secrets store **objects** as a **single volume**
- **pod identity** to restrict access with **specific identities** (Azure provider only)
- **multiple** secrets stores **simultaneously** as **providers**
- **pod portability** with the `SecretProviderClass` CRD
- **sync** with Kubernetes **Secrets**

Cloud metadata API

Cloud Metadata API

Consider that:

- **cloud providers** often **expose** metadata services to **nodes**.
- by **default** these APIs are **accessible** by **pods** running on an instance and can contain:
  - **cloud credentials** for that **node**
  - or **provisioning data** (such as **kubelet credentials**)
- these **credentials** could be used to **escalate** inside:
  - the **cluster**
  - or the **cloud account**

How can I prevent?

- **restrict access** to **cloud metadata API** with **Network Policies**
- apply **least privilege** to **cloud instance roles** (e.g.: to IAM instance roles in AWS)
- apply **single responsibility** to **cloud application roles**
  - letting different service identities in Kubernetes different roles in the cloud in order to interact with cloud resources (e.g.: use IAM Role for Service Account with EKS in AWS)

Network Policy

Restrict access to cloud metadata API

A **network policy** is a **specification** of:

- **how pods** are **allowed** to:
- **communicate** with:
  - each other
  - **network endpoints.**

These policies are **implemented** by the **network plugin**.

NetworkPolicy resources:

- use **labels** to **select pods**
- define **rules** which specify:
  - what **traffic** (in and out) is **allowed** to the selected endpoints/pods.

Notes:

- by **default**, pods **accept** traffic from **any source**
- instead Pods selected by **Network Policies** start to **reject** traffic that is **not explicitely allowed** by the union of these policies
  - the policies are **additive**
  - the **order** of evaluation **does not affect** the policy **result**.

```yaml
kind: NetworkPolicy
metadata:
  name: example-network-policy
  namespace: example
spec:
  podSelector:
    matchLabels:
      role: database
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    #- ipBlock:
    #  cidr: 169.254.169.254/32 # AWS metadata API endpoint
/latest/meta-data/
    ports:
    - protocol: TCP
      port: 5978
    #- procotol: TCP
    #  port: 80
```

Operating System

Operating system

Other than **Kubernetes resources** and **cloud resources**, also access to **operating system resources** must be controlled.

Two **builtin** features helps us:

- **Security Context**
- **Pod Security Policies**

The Security Context **defines** privilege and access control **settings** for a pod or container.

Other than **request restricted** access to workload, you also want to **prevent** to **request** unrestriced access, and you this with Pod Security Policies.

The Pod Security Policy **controls** security **aspects** of the **pod** specification on **creation** and **updates**.

Security Context

Restrict access to OS resources

The settings that a Security Context defines include:

- **UID** and **GID** of the containers;
- the allocation of an **FSGroup** that owns the pod's **volumes** (**gid** and **setgid** bit);
- **SELinux context**;
- **AppArmor profiles**;
- **Privileged** / unprivileged run;
- Linux **capabilities**;
- **Seccomp profiles**;
- Control on **privilege escalation**, which is allowed when the container is:
  - run as Privileged
  - has CAP_SYS_ADMIN capability
- Read-only container **root filesystem mount**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod-with-security-context
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
    AllowPrivilegeEscalation: false
    readOnlyRootFilesystem: true
  volumes:
  - name: example-volume
    emptyDir: {}
  containers:
  - name: example-container-with-security-
context
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: example-volume
      mountPath: /mnt/data/my-data
```

Pod Security Policy

Restrict access to OS resources

Pod Security Policies is a **cluster-level** resource that **controls** security **aspects** of the pod specification on **creation** and **updates**.

The policy objects define a **set of conditions** that a pod **must run** with, in order to be **accepted** into the system (*validation*), as well as defaults for the related fields (*mutation*).

You define **conditions** on:
- **Privileged** / unprivileged run
- Usage of host **namespaces**
- Usage of host **networking** and ports
- Usage of **volume** types
- Usage of the host **filesystem**
- A white list of **Flexvolume drivers**
- The allocation of an **FSGroup** that owns the pod’s volumes
- Requirements for use of a read only **root file system mount**
- **UID** and **GID** of the containers
- **Escalations** of root **privileges**
- Linux **capabilities**
- **SELinux context**
- **AppArmor profiles**
- **Seccomp profiles**
- **sysctl profile**

These **conditions** are **enforced** by an **admission controller**, which is not enabled by default.

The admission controller behaves with **deny-by-default** posture, so without policies it **prevents from creating all pods**.

Since the PSP API is **enabled independently** of the admission controller, you can create the policies **before enabling** the admission controller.

When the policies are **created**, the requesting user or service account must be **authorized** to `use` the **policy**, otherwise the admission controller (if enabled) **will prevents from creating all pods**.

So:
- create the policies
    - and authorize to use the policies
- (if not already) enable the admission controller

Pod Security Policy

Create the policy

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  runAsUser:
    rule: MustRunAsNonRoot
  supplementalGroups:
    rule: MustRunAs
    ranges:
      - min: 1
        max: 65535
  fsGroup:
    rule: MustRunAs
    ranges:
      - min: 1
        max: 65535
  seLinux:
    rule: RunAsAny
  volumes:
    - '*'
```

Authorize to use the policy

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: example-psp
rules:
- apiGroups:
    - policy
  resources:
    - podsecuritypolicies
  verbs:
    - use
  resourceNames:
  - example-psp
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: example-psp
roleRef:
  kind: ClusterRole
  name: example-psp
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: replicaset-controller
  namespace: kube-system
```

Other than **deny** you can also **allow** specific workload to run (for instance) privileged by authorizing specific Service Accounts in specific namespaces to use privileged PSP, instead of the replicaset-controller SA.

Enable the admission controller

```
$ kube-apiserver --enable-admission-plugins=...,PodSecurityPolicy
```

Open Policy Agent

The Open Policy Agent is an OSS **general-purpose policy engine** which provides:

- a **toolset**
- a **framework**

that enables:

- **unified**
- **context-aware**

policy enforcement on the **entire stack**.

In details it provides:

- a **declarative language** (Rego) to develop **policy as code**
- a simple **API**.

It **decouples policy decision-making** from **policy enforcement**.
Decoupling policy helps you build software services **at scale**:

- **makes** them **adaptable** to **changing** business requirements
- **improves** the ability to **discover violations** and **conflicts**
- **increases** the **consistency** of **policy compliance**
- **mitigates** the risk of **human error**.

Open Policy Agent

Policy decisions

It generates **policy decisions** by evaluating the **query input** against **policies** and **data**
The **input** can be a **representation of your system**.

Policy decisions **output** is not **not limited** to **allow/deny** answers, but they **can generate** arbitrary **structured data**.

OPA and **Rego** are **domain-agnostic** so you can describe **almost any kind of invariant** in your policies.

Rego lets you **encapsulate** and **re-use** logic with **rules** that are just if-then **logic statements**.

For example, this **policy decision** with provided **input** against **policy rule** allow will be `true`

Input

```
{
    "servers": [
        {"id": "app", "protocols": ["https", "ssh"]},
        {"id": "db", "protocols": ["mysql"], "ports": ["p3"]}
    ]
}
```

Rego policy

```
package example
default allow = false # unless otherwise defined, deny

# Rule

allow = true { # allow is true if...
    count(violation) == 0 # there are zero violations.
}

# Rule

violation[server.id] { # a server is in the violation set if..
    server := input.servers[_]
    server.protocols[_] == "http"# it accepts "http" protocol.
}
```

OPA Gatekeeper

Kubernetes integration

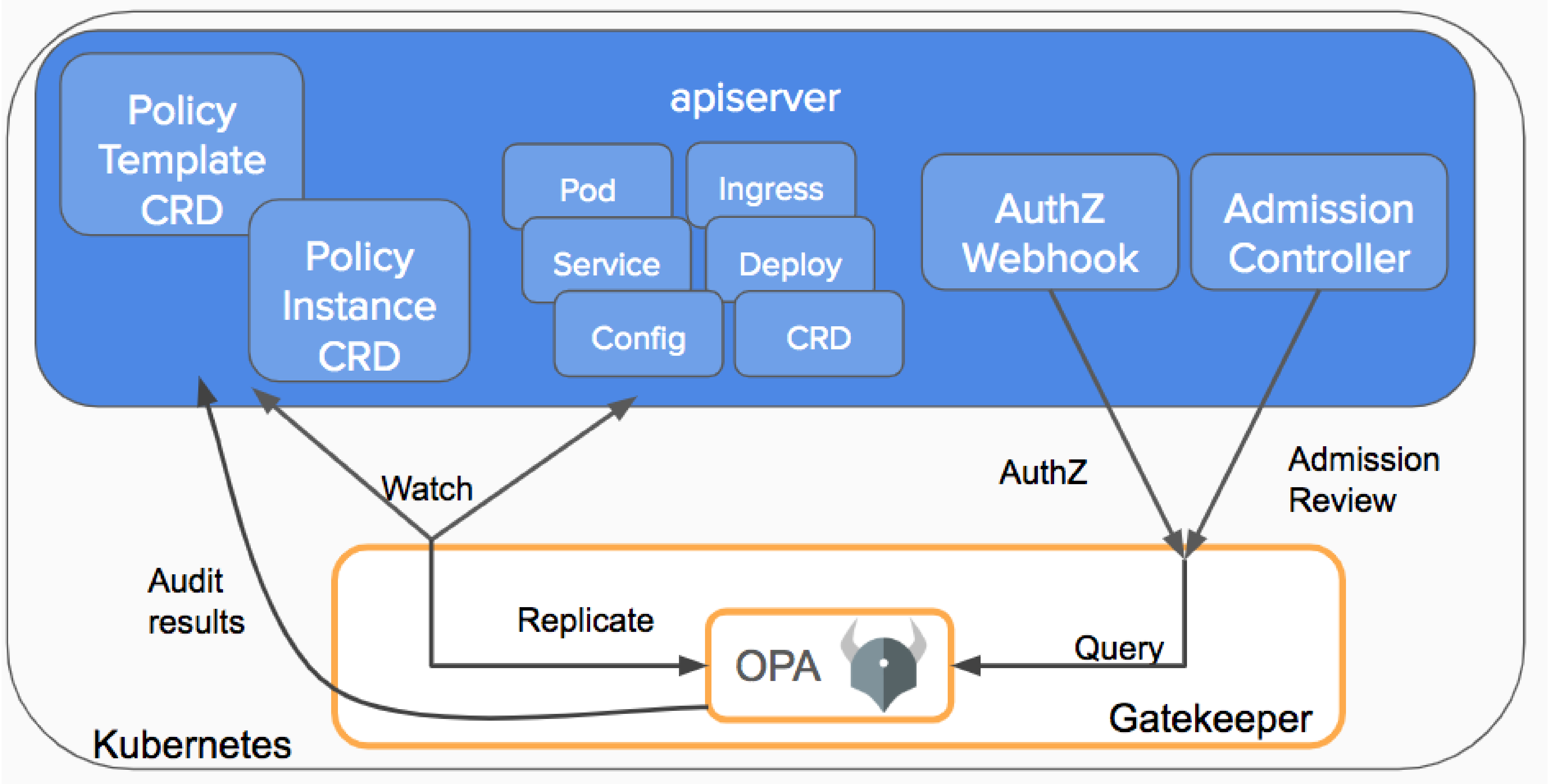
Gatekeeper is a project that provides **integration** between **OPA** and **Kubernetes**.

OPA Gatekeeper v3.0 adds the following on top of plain OPA:

- An **extensible**, **parameterized policy library**
  - **CRDs** for **instantiating** the policy library (aka **Constraints**)
  - **CRDs** for **extending** the policy library (aka **Constraint Templates**)
- **Audit** functionality.

It is deployed as a **validating admission webhook** that **enforces policies** executed by OPA:

- on create / update / delete **requests** the **API Server** sends the entire **Kubernetes object** in an **Admission Review** to the **Gatekeeper admission webhook**
- OPA evaluates the policies it has loaded using the Admission Review as **input**
- during the validation, Gatekeeper acts as a **bridge** between the **API server** and **OPA**
- the **policies** you give to OPA ultimately **generate** an **admission review response** that is sent back to the **API Server**
- the **API server** will **enforce** the **policies**.



OPA Gatekeeper

The Constraint framework

A **Constraint** (aka Policy Instance) is a **declaration** that its **author** wants a **system** to **meet** a given set of **requirements**, evaluated as a logical AND.

A **Constraint Template** (aka Policy Template) describes:

- the Rego **logic** that enforces the **Constraint**
- the **schema** for the **Constraint**, which includes:
  - the **schema** of the **CRD**
  - the **parameters** that can be passed into a Constraint.

```
#  Prevent from running privileged containers

apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate # aka policy library template
metadata:
  name: k8spspprivilegedcontainer
spec:
  crd:
    spec:
      names:
        kind: K8sPSPPrivilegedContainer
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8spspprivileged
        violation[{"msg": msg, "details": {}}] {
            c := input_containers[_]
            c.securityContext.privileged
            msg := sprintf("NOT ALLOWED: %v, securityContext: %v", [c.name,
c.securityContext])
        }
        input_containers[c] {
            c := input.review.object.spec.containers[_]
        }
        input_containers[c] {
            c := input.review.object.spec.initContainers[_]
        }
---
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sPSPPrivilegedContainer # aka policy library instance
metadata:
  name: psp-privileged-container
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  # parameters:
```

deltatre

On top of Kubernetes resources aka your workload

ACCESS CONTROL

Network policy

```yaml
# allow only backend services to
# communicate with Redis service

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-redis
spec:
  podSelector:
    matchLabels:
      app: redis
  ingress:
  - from:
    - podSelector:
        matchLabels:
          area: "backend"
```

ACCESS CONTROL

Service mesh authorization with Istio

Istio's **authorization features** provide **access control** for your **workloads** in the **mesh** at:
- mesh-wide
- namespace-wide
- workload-wide

Its **authorization system** provides these benefits:
- **Workload-to-workload** and **end-user-to-workload** authorization
- A **Simple API** (a single `AuthorizationPolicy` **CRD**)
- Flexible **semantics** that lets you
  - define custom conditions on Istio attributes
  - use DENY and ALLOW actions
- High **performance**
  - Envoy **natively enforces** it
- High **compatibility**
  - gRPC
  - HTTP
  - HTTPS
  - HTTP2
  - plain TCP

How it works?

Each **Envoy proxy** runs an **authorization engine**:
- a **request** comes to the **proxy**
- the **authorization engine** evaluates the **request context** against the current **authorization policies**
- it returns the **authorization result** (`ALLOW` or `DENY`)

Without an authorization policy applied to a workload: **allow by default**;
With an authorization policy applied to that workload: **deny by default**.
Furthermore, **deny policies** take **precedence** over **allow policies**.

Istio

Authorization policy

An **authorization policy** object includes:

- **selector** field that specifies the **target** of the policy
  - along with the resource **namespace**
- **action** field that specifies ALLOW or DENY
- **list of rules** that specify **when** to trigger the action
  - **from** field in the rules specifies the **sources** of the request
  - **to** field in the rules specifies the **operations** of the request
  - **when** field specifies the **conditions** needed to apply the rule

This **policy** allows:

- the `cluster.local/ns/foo/sa/sleep` **service account** (the **source**)

to **access** with `GET` HTTP requests (the **operations**)

the **workloads**:

- with the `app: httpbin` label
- in the `namespace: foo`

when requests sent (the **conditions**):

- have a Google-issued JWT token (`request.auth.claims[iss]`)

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin
 namespace: foo
spec:
 selector:
   matchLabels:
     app: httpbin
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/foo/sa/sleep"]
   to:
   - operation:
       methods: ["GET"]
   when:
   - key: request.auth.claims[iss]
     values: ["https://accounts.google.com"]
```

If he could know my secrets,
no one should be able to
listen our conversation

Confidentiality

Confidentiality

Confidentiality aims to **protect** data sent from one **sender** to a **recipient** of that data against **third parties**.

It can be enforced to prevent **information disclosure** with **encryption**.

We should consider:

- data **at rest**
- data **in transit**

Data at rest

CONFIDENTIALITY

Secret API objects

Let’s start with **critical data**, which in Kubernetes is mostly represented by **Secret objects**.

Secrets objects are stored in **etcd**, but they are encrypted?

No, they don’t, but Kubernetes provides a feature…

Kubernetes encryption at rest

So that the Secrets are not stored in plaintext into etcd, and even if an attacker can gain access to it, he can’t read it.

Which are the requirements?

- Kubernetes 1.13
- etcd 3.0

How to configure it?

```
$ kube-apiserver \
--encryption-provider-config <encryption-configuration>
```

deltatre

Kubernetes encryption at rest

EncryptionConfiguration

The configuration is represented by the `EncryptionConfiguration` API object, part of the `apiserver.config.k8s.io` API group.

The `resources.resources` field is an array of resource names that should be encrypted.
The `providers` array is an ordered list of the supported encryption providers:

- `identity` (default; no encryption)
- `aescbc`
- `secretbox`
- `aesgcm`
- `kms`

On **write** request for a **Secret**:

- the **first provider** in the list is used to **encrypt** this resource (identity by default)

On **read** request for a Secret:

- **each key** for **each provider** that **matches** the stored **data** attempts to **decrypt** the data in order.

Warning: if you put the identity provider as the first one you actually disable encryption.

Since the config file can contain encoded data encryption keys:
the encryption could **fails to protect** against a **host compromise**.

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - aescbc:
        keys:
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
    - identity: {}
```

CONFIDENTIALITY

Envelope encryption FTW

With **envelope encryption** the **data encryption key** that encrypt data is **further encrypted** with a **key encryption key**.

Also the **key encryption key** can be **encrypted**; BTW eventually a **top-level** key in **plaintext** must exists and this is the **master encryption key**.

Additional features:

- the **DEK** can be **generated for each encryption**;
- the **DEK** can be **stored besides data**;
- the **MEK** can be **easily rotated**;
- the **MEK** can be **asymmetrical**.

Additional benefits:

- **performance**: as the data can be large the encryption can be time-consuming:
  - you don’t have to re-encrypt multiple times the data with DEKs
  - you can re-encrypt only the DEKs
    - only a key would transit instead of data

deltatre

Kubernetes and envelope encryption at rest

EncryptionConfiguration with the KMS provider

The **KMS encryption provider** uses **envelope encryption** and lets you **offload** key/master encryption key to **external KMS**.

The **DEK** is **generated** on **each encryption**, and the **KEK rotation controlled by you**.

The **KMS provider** uses **gRPC** to communicate with a specific **KMS plugin.**
The **KMS plugin**, which is implemented as a **gRPC server**, communicates with the **remote KMS**.

Note that if you are using EKS they introduced on March 2020 the support for envelope encryption of Secrets with AWS KMS a little time ago; here the announcement.

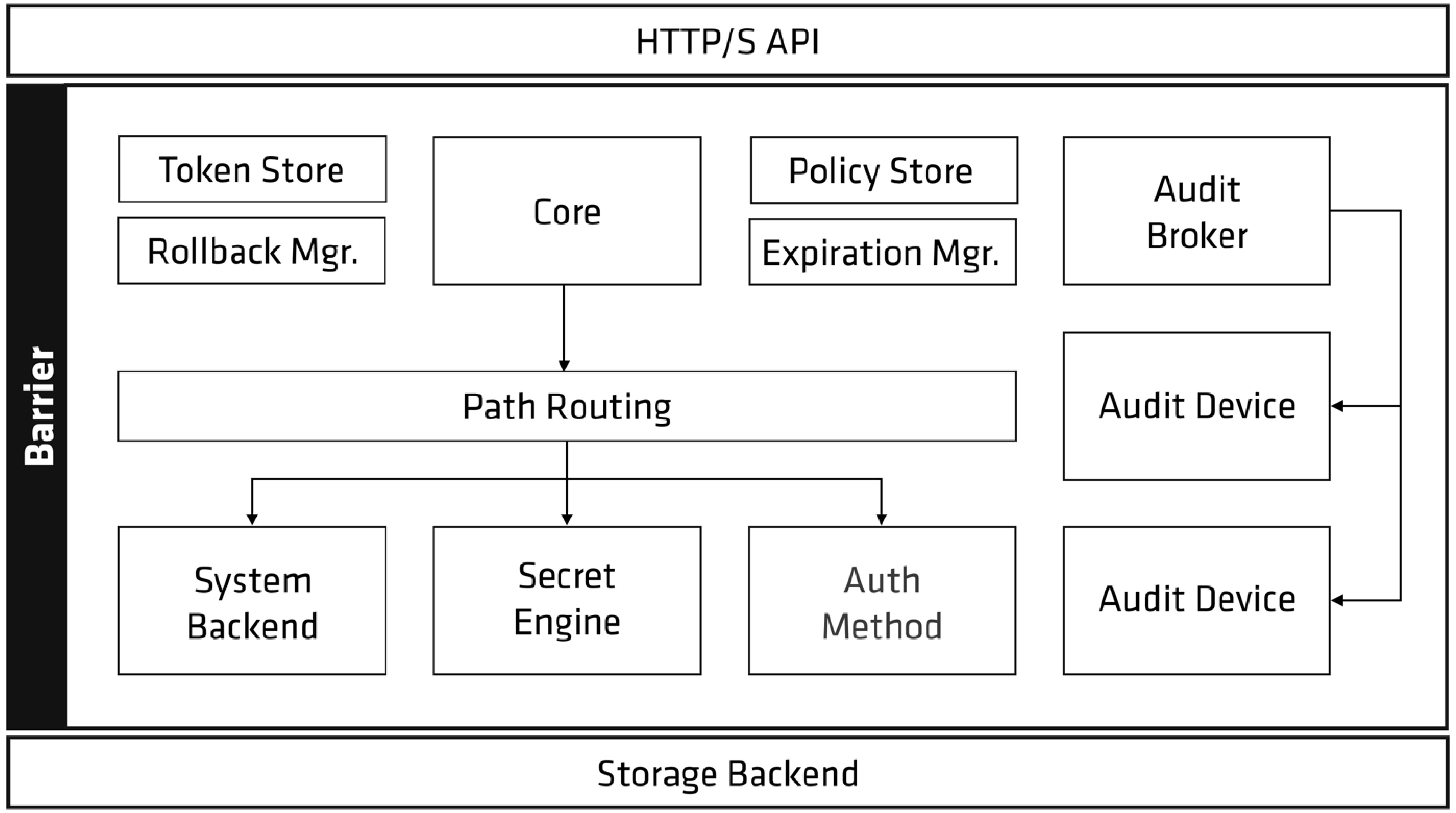
```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - kms:
        name: myKmsPlugin
        endpoint: unix:///tmp/socketfile.sock
        cachesize: 100
        timeout: 3s
    - identity: {}
```

CONFIDENTIALITY

Secret without Secret API objects with Vault

You can also manage secrets without the related Kubernetes API objects and offload secrets management to external systems like Hashicorp Vault.

Vault is a **secrets management system** and is like a secrets fortress with a powerful architecture.



Let's focus on the encryption at rest features!

deltatre

Vault encryption fortress

The barrier

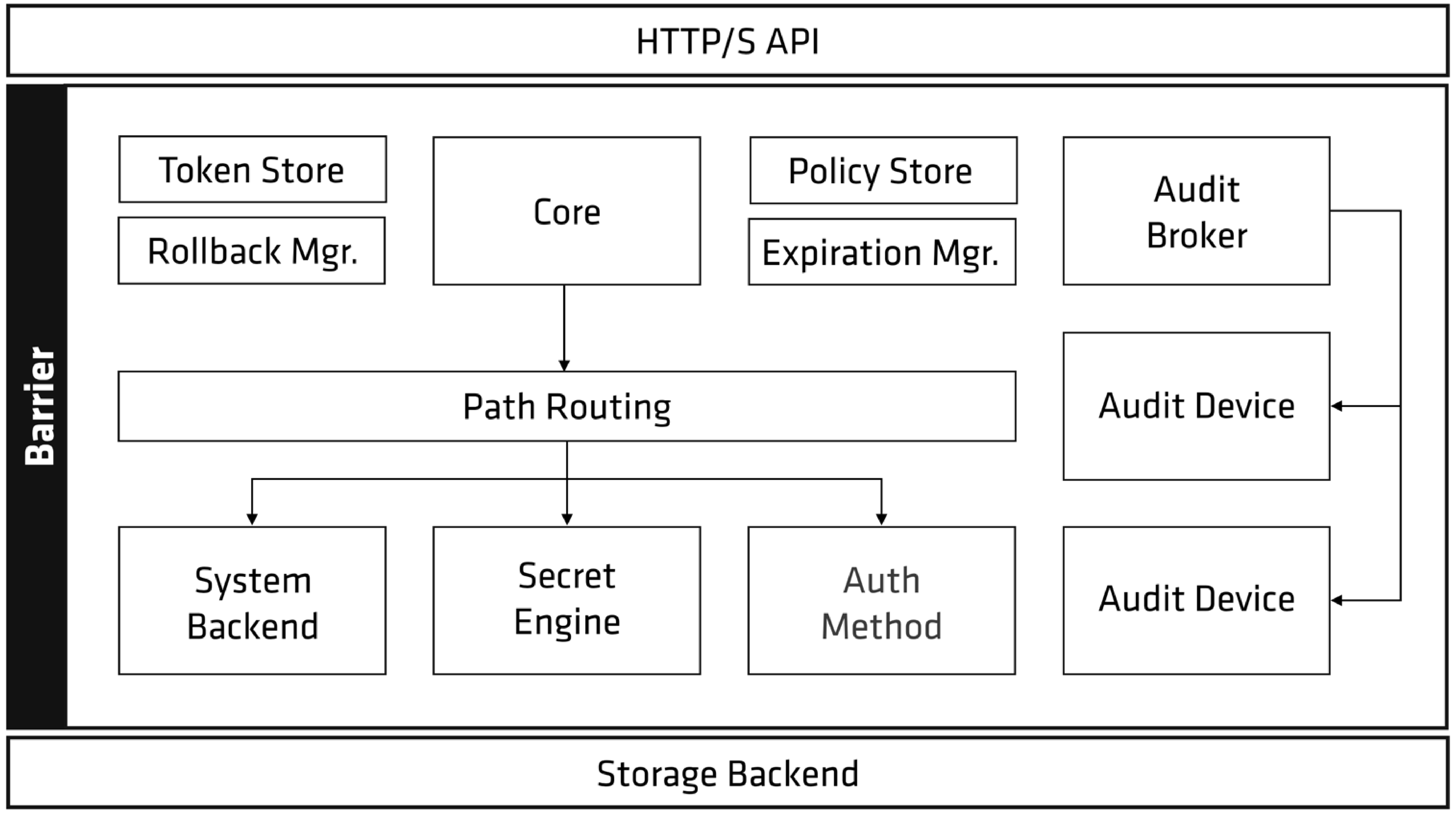
It’s a **cryptographic steel-and-concrete** between the **storage** and **Vault**.

It **ensures** that:
- the **data** is **encrypted** when **written** out;
- that **data** is **verified** and **decrypted** when **reading**;

There is a **clear separation** between **inside** or **outside** of the barrier.

Only two **components** are **outside** the barrier:
- storage backend
- HTTP API

Storage backend is:
- durable storage of encrypted data
- not trusted by Vault.



CONFIDENTIALITY

Vault encryption fortress

The seal

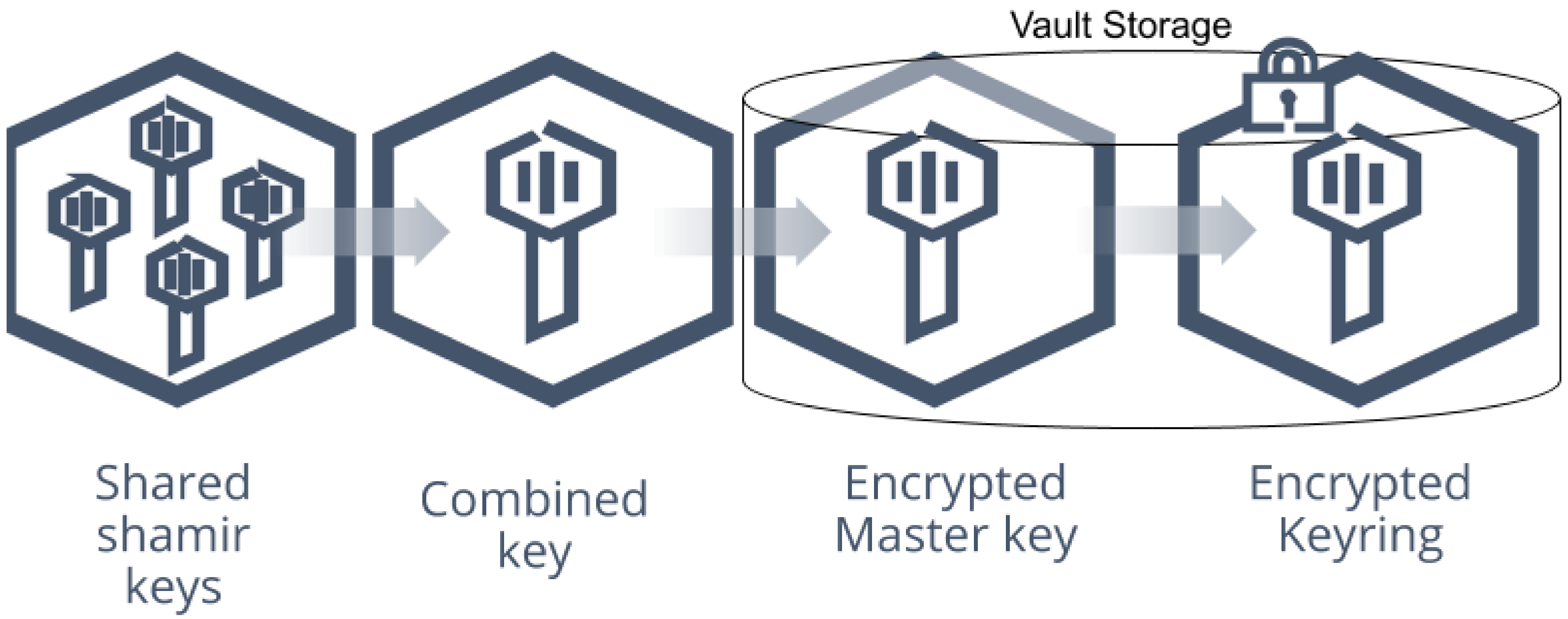
Vault implements **envelope encryption** once started is in a **sealed state**.

When the Vault is **initialized** it **generates** a **DEK** to encrypt **data**.
The **DEK** is stored **with the data** (in the **keyring**) and is encrypted by a **MEK**.
The **MEK** is stored **alongside all other data** and is further encrypted by a **seal key**.

Default seal key: Shamir
The Shamir's Secret Sharing algorithm **splits the key** into 5 **shards**.
A **threshold** of 3 **shards** is required to **reconstruct** the **unseal key**.

In the **sealed state** Vault is able to **access** the physical **storage**, but **not** to **decrypt** the data.

Unsealing is the process of **obtaining** the **plaintext MEK** to eventually **read** the **data**.
Once **unsealed**, Vault **loads** all of the **other configurations**.



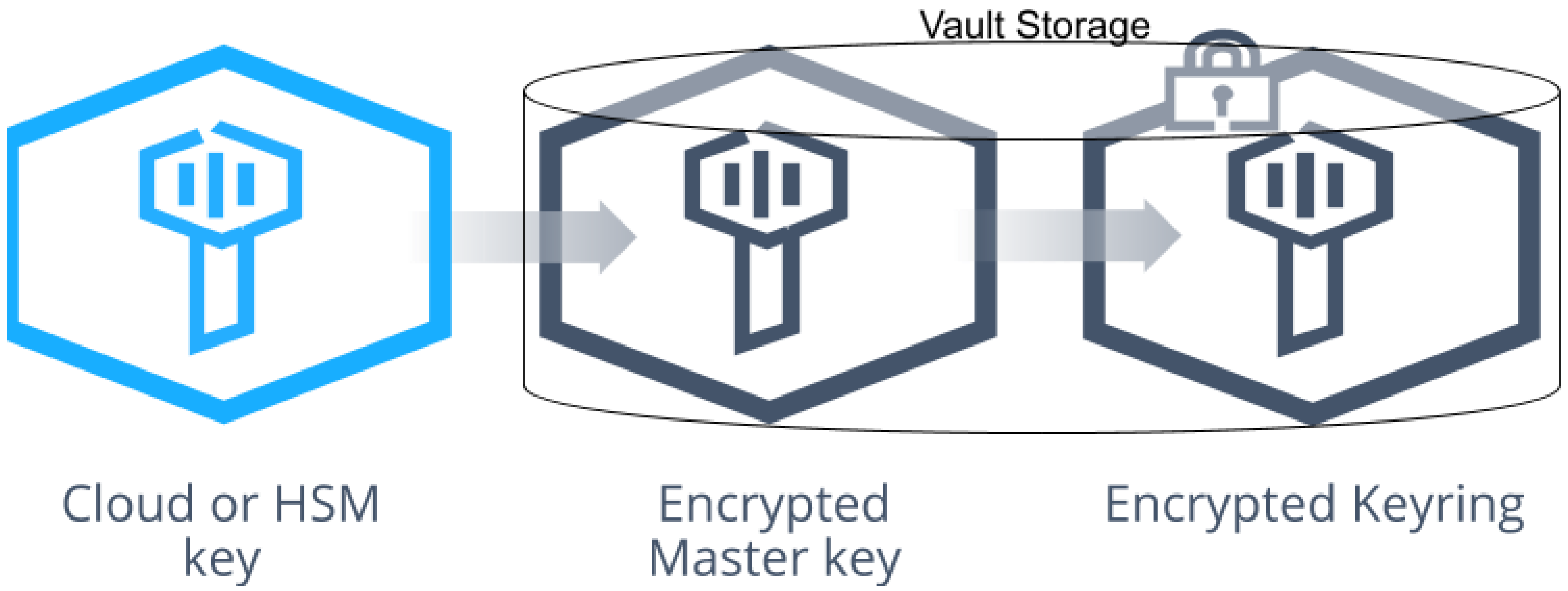
CONFIDENTIALITY

Vault encryption fortress

The auto unseal

Auto unseal lets you **delegate seal key** protection to **external KMS**.

At **startup** Vault will **connect** to the seal **device** or **service** asking to **decrypt** the **MEK** in order to **read** from **storage**.



Vault in Kubernetes

Recap

Client

Currently there are three ways to **injects** Vault **secrets** into Pods:

- **Hashicorp**'s official **Vault Injector** init container + **Vault Agent sidecar** which injects into **tmpfs** shared volume via mutating webhook sidecar injector (**mutable** solution)

- **Banzai** Cloud's **Vault Injector** init container which injects into your container **environment variables** (**immutable** solution)

- **Hashicorp**'s **Secrets Store CSI** driver **Vault provider** via `SecretProviderClass` Secrets Store CSI API objects

Server

Hashicorp provides an official **Helm chart** github.com/hashicorp/vault-helm.

Furthermore, the Hashicorp documentation is great.

CONFIDENTIALITY

Version control system

Obvious but worth say

base64 is an **encoding** scheme... That’s ok if you version Secrets objects, but **encrypt**.

For instance, Mozilla’s **SOPS** is an **editor** of **encrypted files** that supports these **formats**:

- YAML
- JSON
- ENV
- INI
- BINARY

and **encrypts** with:

- PGP
- AWS KMS
- GCP KMS
- Azure Key Vault

Decouple

Anyway, I prefer to **offload secrets management** and **decouple secrets lifecycle** from **code versioning**.

deltatre

Data in transit

Between workload services

A few levels above TLS

Encryption as a Service

In addition to the **encryption** of the **whole communication**, you can and should also encrypt specific **part** of the **L7 payload** by leveraging **secrets manager** like **Vault**.

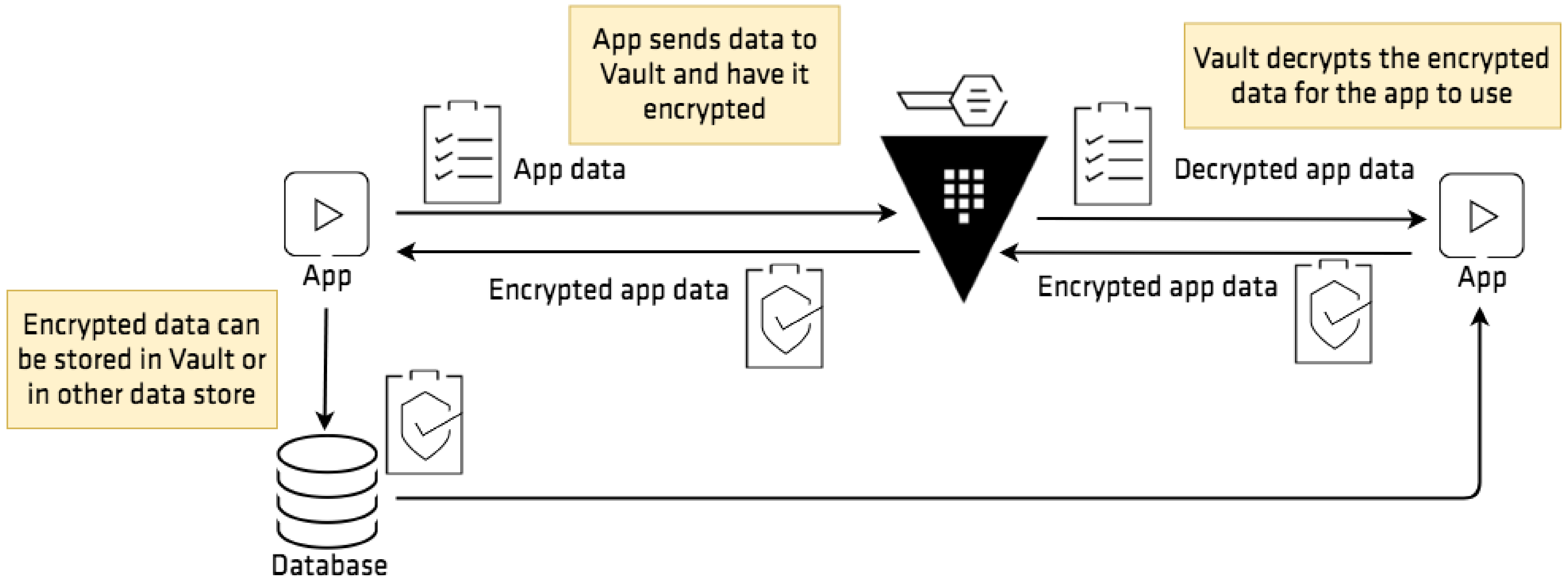
Vault provides **Encryption as a Service** thanks to its **Transit Secrets Engine**.

A **Secrets Engine** is responsible for **managing secrets**.
Secrets Engines support **dynamic secrets** (generated every time they are required) and **static secrets** ones as simple K/V objects.

The **Transit Secrets Engine** handles **cryptographic functions** on **data in transit**.
Vault **does not store** the **data** sent to the **Secrets Engine**.

The Transit Secrets Engine can also:

- **sign** and **verify data**
- generate hashes and HMACs of data
- act as a source of random bytes.



CONFIDENTIALITY

Vault in Kubernetes

Encryption

```
$ curl --header "X-Vault-Token: $VAULT_TOKEN" \
       --request POST \
       --data '{"plaintext": "<PLAIN_BASE64_SECRET_DATA>"}' \
       <VAULT_ADDRESS>/v1/transit/encrypt/orders | jq

{
  "request_id": "<REQUEST_ID>",
  "lease_id": "",
  "renewable": false,
  "lease_duration": 0,
  "data": {
    "ciphertext":
"vault:v1:/9hdQutaWpZR72s3+VSCLK1JNhV1wKM49hYVjh7RjmuxIy/Ovsht
gV4L4uVB+aQ="
  }
}
```

Decryption

```
$ curl --header "X-Vault-Token: $VAULT_TOKEN" \
       --request POST \
       --data '{"ciphertext":
"vault:v1:/9hdQutaWpZR72s3+VSCLK1JNhV1wKM49hYVjh7RjmuxIy/OvshtgV4L4u
VB+aQ="}' \
       <VAULT_ADDRESS>/v1/transit/decrypt/orders | jq

{
   "request_id": "<REQUEST_ID>",
   "lease_id": "",
   "renewable": false,
   "lease_duration": 0,
   "data": {
     "plaintext": "Y3JlZGl0LWNhcmQtbnVtYmVyCg=="
   }
}
```

Anyway, I'm keeping an eye on him

Audit

deltatre

Audit

Audit we can **prevent** from **repudiation**, in order to **proove**:

- what
- when
- where
- why
- who
- how

on certain **actions**, by **recording** them.

We generally gain **visibility** and It also helps to **watch** for **evidence** of **tampering.**

In Kubernetes

Kubernetes API Audit

The **API server** performs **audit**

For **each stage** of **each request**:

- generates an **event**, which is:
  - **pre-processed** according to a **policy**
    - that determines what's **recorded**
  - written to a **backend**
    - persists the **records**, and can be:
      - **log**
      - **static webhook**
      - **dynamic webhook**

Kubernetes API Audit

Request stages

The available **stages** for each **request** to be **recorded** are:

- **request received**: as soon as the audit handler receives the request;
- **response started**: once the response headers are sent, but before the response body is sent; it’s only generated for long-running requests (e.g.: `watch` requests)
- **response complete**: the response body has been completed and no more bytes will be sent
- **panic**: events generated when a panic occurred.

Note: with log backend the memory consumption of the API server increases as context required is stored for each request.

Kubernetes Audit

Audit Policy

Audit policy **defines rules** about **which events** should be **recorded**.
When an **event** is **processed**, it's **compared** against the **list of rules** of the `AuditPolicy` in order.

The **audit levels** are:
- **None**, which logs:
  - no events that match this rule;
- **Metadata**, which logs:
  - request metadata
  - not request / response body
- **Request** (it does not apply for non-resource requests), which logs:
  - event metadata
  - request body
  - not response body. This does not apply for non-resource requests.
- **RequestResponse** (it does not apply for non-resource requests) which logs:
  - event metadata
  - request body
  - response body.

The **first** matching rule sets the **audit level** of the **event**.

```
apiVersion: audit.k8s.io/v1
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: ""
      resources: ["pods"]
  # don't log requests to configmaps
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
```

Enable the policy

```
$ kube-apiserver --audit-policy-file <policy-file>
```

Kubernetes Audit

Audit Backend

Audit backends **persist** audit **events** to an external **storage**.
BTW **audit event** is represented by the API resource in the audit.k8s.io group.

Kube-apiserver out of the box provides three backends:

- **Log**: it writes audit events to a file in JSON format
- **Webhook**: it sends audit events to a remote API, which is assumed to be the same API as kube-apiserver exposes
- **Dynamic**: it configures webhook backends through an `AuditSink` API object.

Events buffering

Both logging and webhook backends support **batching** for example to **buffer** events and **asynchronously process** them (enabled by default in webhook).

Kubernetes Audit

Dynamic Audit Backend

With **dynamic backend** you can provide **different** audit **use cases** dynamically.

You can define:

- **different backends**
- with **different policies**

and let them **evolve** at **runtime**, regardless of the **static policy**.

To enable dynamic auditing, `kube-apiserver`:

`--audit-dynamic-configuration=true`; # configure audit with dynamic backend

`--feature-gates=DynamicAuditing=true`; # enable the feature gate

`--runtime-config=auditregistration.k8s.io/v1alpha1=true` # enable the API

```
apiVersion:
auditregistration.k8s.io/v1alpha1
kind: AuditSink
metadata:
  name: mysink
spec:
# audit all events of requests when the
response is complete
  policy:
    level: Metadata
    stages:
    - ResponseComplete
  webhook:
    throttle:
      qps: 10
      burst: 15
    clientConfig:
      url: "https://audit.app"
```

Kubernetes components logs

With `systemd`

- `journald`

Without `systemd`

- `/var/log/kube-apiserver.log`
- `/var/log/kube-scheduler.log`
- `/var/log/kube-controller-manager.log`
- `/var/log/kubelet.log`
- `/var/log/kube-proxy.log`

On Kubernetes

AUDIT

Workload container logs

As **standard streams** are **communication channels** between a **program** and the **environment** on which **runs**:
- **decouple** logging **system** from the **application** with `stdout`/`stderr`

As a standard interface is likely accepted by **OCI** compliant and **CRI** compatible **runtimes**.

There’s a proposal about how CRI should handle container's stdout/stderr log streams:
- github.com/kubernetes/community/blob/master/contributors/design-proposals/node/kubelet-cri-logging.md

For example the Docker **json logging driver** catches `stdout`/`stderr` and **writes** them into **json-formatted files**.

At the same way **containerd** supports **log plugins**, even if his **CRI plugin** currently **does not support** it (issue #1342).

deltatre

Under Kubernetes

From Kubernetes to the Linux kernel with Falco

Falco is an OSS security project part of the CNCF at sandbox level, that can detect and alert on any anomalous behavior that involves making Linux system calls.

Falco features

Tracing

It **traces kernel events** via:
- kernel module
- eBPF probes
- ptrace

Parsing and assertion

It ships with **policies**, which are a **set of rules** that Falco will **assert against**.
They can **detect unusual behavior** in the **kernel**, such as:
- privilege escalation using privileged containers
- namespace changes using tools like setns
- spawned processes using execve
- executing shell binaries
- etc.

Alerting

It ships with **alerts support**, which are **configurable** downstream **actions** that can be:
- stdout/stderr;
- a file;
- syslog;
- a spawned program;
- HTTP/S endpoint.

Furthermore, you can **integrate** with its **API** for **incident response** and **mitigation**.

Falco components

Userspace program: this is the program a user interacts with, responsible for:
- **handling** signals
- **parsing** information
- **alerting**.

Kernel driver: this is a piece of software that can send a stream of system call information from the kernel.

Configuration: which defines:
- **how** Falco is **run**
- **what** rules to **assert**
- **how** to perform **alerts**.

The hearth of Falco

The rules engine

Because Falco is built on top of **Sysdig event processing libraries**.
Events are made from **syscalls** and include the **context** in which a **system call** was performed, including:

- the **process name** performing the system call
- the **process's parents**, grandparents, etc.
- the **remote IP** address to which the process is **communicating**
- the **directory** of the file being **read/written**
- the **current memory usage** of the process
- etc.

```
- rule: File Open by Privileged
Container
  desc: Any open by a privileged
container. Exceptions are made for known
trusted images.
  condition: (open_read or open_write)
and container and
container.privileged=true and not
trusted_containers
  output: File opened for read/write by
non-privileged container
(user=%user.name command=%proc.cmdline
%container.info file=%fd.name)
  priority: WARNING
```

The `condition` field is a **filter** applied to each **syscall**.
When an **event matches**, the `output` field is used to format a **notification** message.

It’s time to break something...