Understanding Authentication

Authentication is the process of determining whether someone or something is, in fact, who or what it says it is. It is the mechanism of associating an incoming request with a set of identifying credentials.

The API server can be configured with one or more authentication plugins. It goes through the list of authentication plugins, so they can each examine the request and try to determine who's sending the request.

First, the request is authenticated via an authentication plugin, then authorized by an authorization plugin, after which is intercepted by an admission controller before it's persisted in the etcd which is the stateful component that stores the cluster's state and configuration.

Admission controllers are Kubernetes-native features that help you define and manage what is allowed to run on the cluster. For example, enforcing security policies, managing deployments resources and even blocking vulnerable images from being deployed.

Authentication Plugins

Several authentication plugins are available. They obtain the identity of the client using the following methods:

- X509 Client Certs
- Bearer token passed in an HTTP header
- OpenID Connect Tokens
- Others

When the API server receives a request, plugins attempt to associate and extract the following attributes from the request:

- `Username`: a string which identifies the end user. Common values might be kube-admin or jane@example.com.

- `UID`: a string which identifies the end user and attempts to be more consistent and unique than username.

- `Groups`: a set of strings, each of which indicates the user's membership in a named logical collection of users. Common values might be system:masters or devops-team.

Users

All Kubernetes clusters have two categories of users: service accounts managed by Kubernetes, and normal users (actual humans).

Service accounts are the mechanism by which the Pods (more specifically, applications running inside them) authenticate to the API server which are created and stored in the cluster as ServiceAccount resources.

Normal users are meant to be managed by an external system, such as a Single Sign On (SSO) system which means you can't create, update, or delete users through the API server.

Groups

Both human users and ServiceAccounts can belong to one or more groups. Authentication plugins return groups along with the username and user ID.

Groups are used to grant permissions to several users at once, instead of having to grant them to individual users.

Common built-in groups include:

- The `system:unauthenticated`: used for requests where none of the authentication plugins could authenticate the client.
- The `system:authenticated`: automatically assigned to a user who was authenticated successfully.
- The `system:serviceaccounts`: encompasses all ServiceAccounts in the system.
- The `system:serviceaccounts:<namespace>`: includes all ServiceAccounts in a specific namespace.

Understanding RBAC Authorization

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within your organization.

- The Kubernetes API server can be configured to use an authorization plugin to check whether an **action** is allowed to be performed by the user requesting the action.
- Because the API server exposes a REST interface, users perform **actions** by sending HTTP requests to the server.
- Users authenticate themselves by including credentials in the request (an authentication token, username and password, or a client certificate).

Actions

REST clients send GET, POST, PUT, DELETE, and other types of HTTP requests to specific URL paths, which represent specific REST resources.

In Kubernetes, those resources are Pods, Services, Secrets, and so on. Here are a few examples of actions in Kubernetes:

- Get Pods
- Create Services
- Update Secrets
- And so on

The verbs in those examples (get, create, update) map to HTTP methods (GET, POST, PUT) performed by the client. The nouns (Pods, Service, Secrets) map to Kubernetes resources.

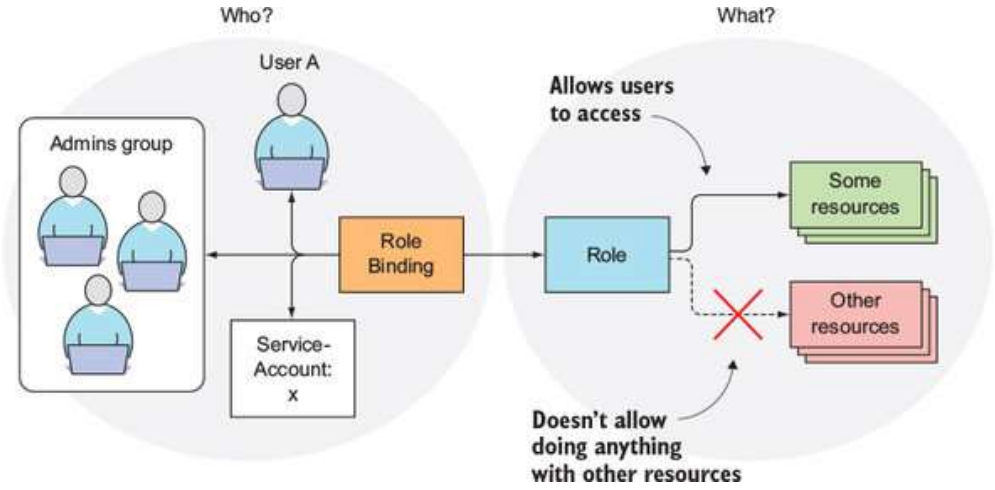
Verb	HTTP method	Description
`create`	`POST`	Create a new resource.
`delete`	`DELETE`	Delete an existing resource.
`get`	`GET`	Get a resource.
`list`	`GET`	List a collection of resources.
`patch`	`PATCH`	Modify an existing resource via a partial change.
`update`	`PUT`	Modify an existing resource via a complete object.

RBAC Resources

The RBAC authorization rules are configured through four resources, which can be grouped into two groups:

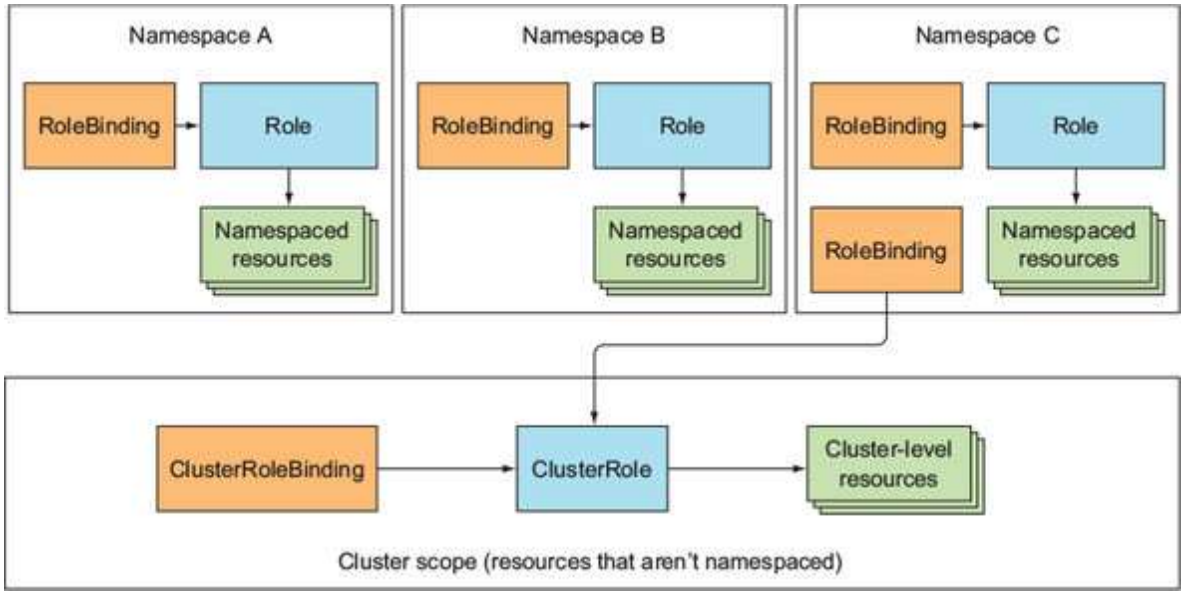
- **Roles** and **ClusterRoles**, which specify which verbs can be performed on which resources.
- **RoleBindings** and **ClusterRoleBindings**, which bind the above roles to specific users, groups, or ServiceAccounts.

Roles define what can be done, while bindings define who can do it



RBAC Resources

The distinction between a **Role** and a **ClusterRole**, or between a **RoleBinding** and a **ClusterRoleBinding**, is that the Role and RoleBinding are namespaced resources, whereas the ClusterRole and ClusterRoleBinding are cluster-level resources (not namespaced).



The best way to learn about these four resources and what their effects are is by trying them out, which we will do in the hands-on section of the lab.

Runtime Security in Kubernetes

Kubernetes runtime security protects containers (or pods) against active threats once the containers are running.

Runtime security helps protect workloads against a variety of threats that could emerge after containers have been deployed, such as:

- The activation of malware that is hidden inside a container image.
- Privilege escalation attacks in which a container exploits security bugs in the container runtime, Kubernetes, or the host OS.
- The deployment of unauthorized containers by an attacker who exploits a gap in an access control policy or a bug in Kubernetes.
- Unauthorized access to secrets or other sensitive information that a running container should not be able to read

Runtime Security Tools

Tools for Kubernetes runtime security fall into two main categories:

- **Enforcement** tools: allow you to define policies that restrict the access rights and permissions of resources within a runtime environment.
- **Auditing** tools: allow you to detect and react to threats based on data collected from your cluster.

Enforcement Tools

While the tools can’t prevent a threat from materializing, they can minimize its impact by ensuring that, for example, malware that appears within one container can't access resources external to the container.

Key enforcement tools for Kubernetes include:

- Security Context: defines privilege and access control settings for a Pod or Container.
- Seccomp: a Linux kernel-level tool that you can use to force processes to run in a secure state.
- SELinux: a kernel module that lets you define a broad array of access controls that the kernel enforces.
- AppArmor: also a kernel module that enables the definition of a broad set of access control rules.

While SELinux and AppArmor share similar characteristics, they are both fundamentally different in favor of SELinux being more secure and complex.

For more information, you can read about the comparison in this link.

Auditing Tools

Kubernetes provides the tooling for generating audit logs, but it doesn't actually analyze the logs for you. For that, you'll want to use a tool like Falco, an open source threat detection engine.

Falco lets you define rules that will trigger alerts if the **Falco** engine detects the presence of certain conditions based on data like Kubernetes audit logs.

Simple rule example:

```
- rule: shell_in_container
desc: notice shell activity within a container
condition: container.id != host and proc.name = bash
Output (**do not copy the output window to the terminal line): shell in a
container (user=%user.name container_name=%container.name)
priority: warning
tags: [shell, container]
```

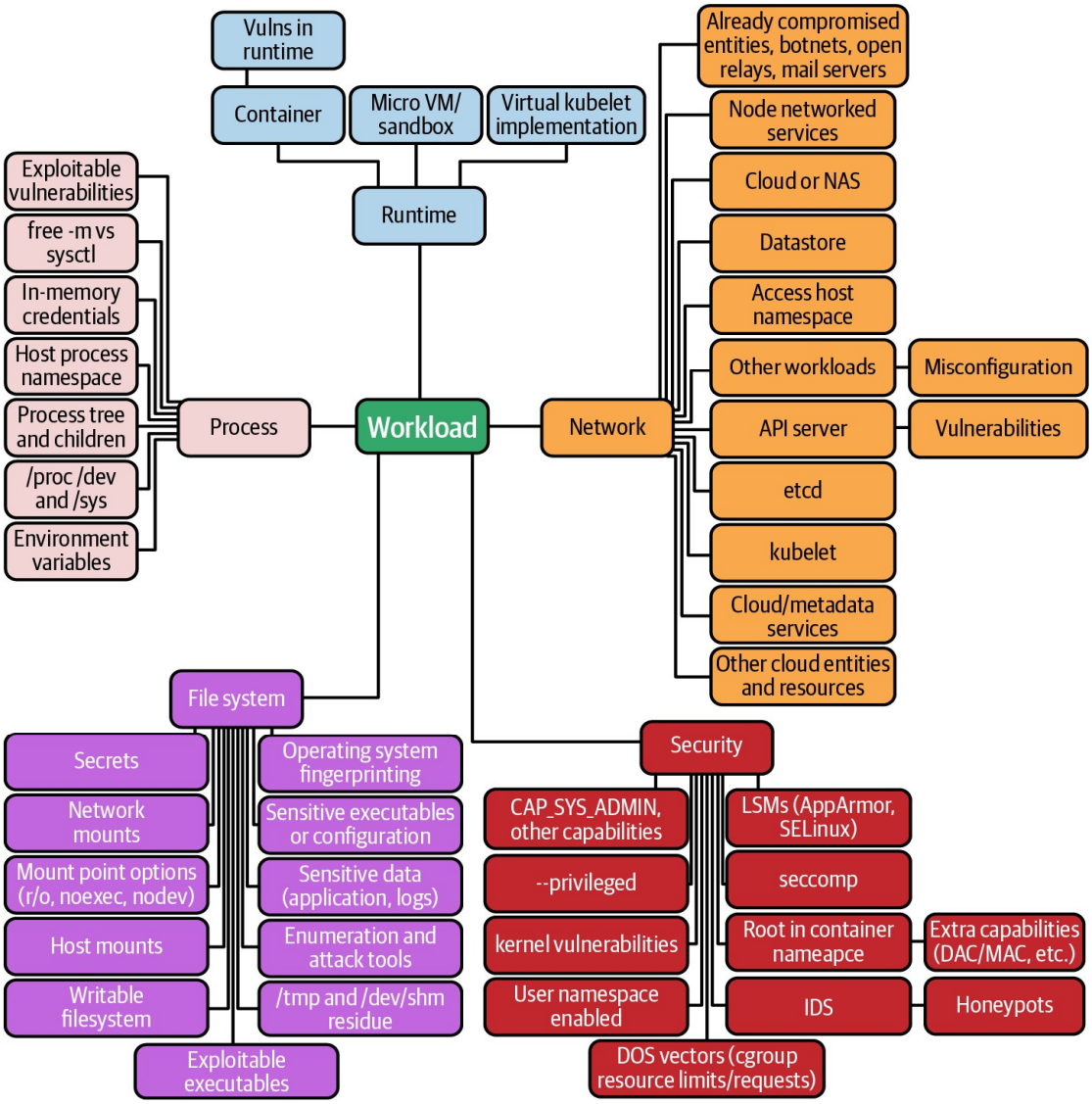
In this rule, any shell activity within a container that does not meet the following criteria will trigger an alert:

- The event occurred within the container
- The listening process is `bash`

With a rule like this, you can detect if a shell was spawned in a container and by whom to prevent unauthorized access to containers.

Pod Security Standards

The following map describes the different attack surfaces of a Pod:



The Kubernetes Pod Security Standards define different isolation levels for Pods to help reduce attack vectors.

Pod Security Standards

The Pod Security Standards define three different policies to broadly cover the security spectrum. These policies are cumulative and range from highly-permissive to highly-restrictive.

Profile	Description
Privileged	Unrestricted policy, providing the widest possible level of permissions. This policy allows for known privilege escalations.
Baseline	Minimally restrictive policy which prevents known privilege escalations. Allows the default (minimally specified) Pod configuration.
Restricted	Heavily restricted policy, following current Pod hardening best practices.

Kubernetes offers two ways to enforce these standards: **Pod Security Admission** and **Pod Security Policies**.

Pod Security Admission

Pod Security admission controller is a built-in **Beta** feature as of **v1.23** and a successor to **PodSecurityPolicies**.

Pod security restrictions are applied at the namespace level when pods are created.

Pod Security admission places requirements on a Pod's Security Context and other related fields according to the three levels defined by the Pod Security Standards: **privileged**, **baseline**, and **restricted**.

You can configure namespaces to define the admission control mode you want to use for pod security in each namespace.

Kubernetes defines a set of labels that you can set to define which of the predefined Pod Security Standard levels you want to use for a namespace.

The label you select defines what action the control plane takes if a potential violation is detected:

Mode	Description
enforce	Policy violations will cause the pod to be rejected.
audit	Policy violations will trigger the addition of an audit annotation to the event recorded in the audit log, but are otherwise allowed.
warn	Policy violations will trigger a user-facing warning, but are otherwise allowed.

Pod Security Admission

A namespace can configure any or all modes, or even set a different level for different modes.

For each mode, there are two labels that determine the policy used:

```
# The per-mode level label indicates which policy level to apply for the mode.
#
# MODE must be one of `enforce`, `audit`, or `warn`.
# LEVEL must be one of `privileged`, `baseline`, or `restricted`.
pod-security.kubernetes.io/<MODE>: <LEVEL>

# Optional: per-mode version label that can be used to pin the policy to the
# version that shipped with a given Kubernetes minor version (for example
v1.23).
#
# MODE must be one of `enforce`, `audit`, or `warn`.
# VERSION must be a valid Kubernetes minor version, or `latest`.
pod-security.kubernetes.io/<MODE>-version: <VERSION>
```

Pod Security Policies

A Pod Security Policy is a cluster-level resource that controls security sensitive aspects of the pod specification.

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

PodSecurityPolicy is deprecated as of Kubernetes **v1.21**, and will be removed in **v1.25**.

Therefore, it is recommended to use Pod Security Admission instead.

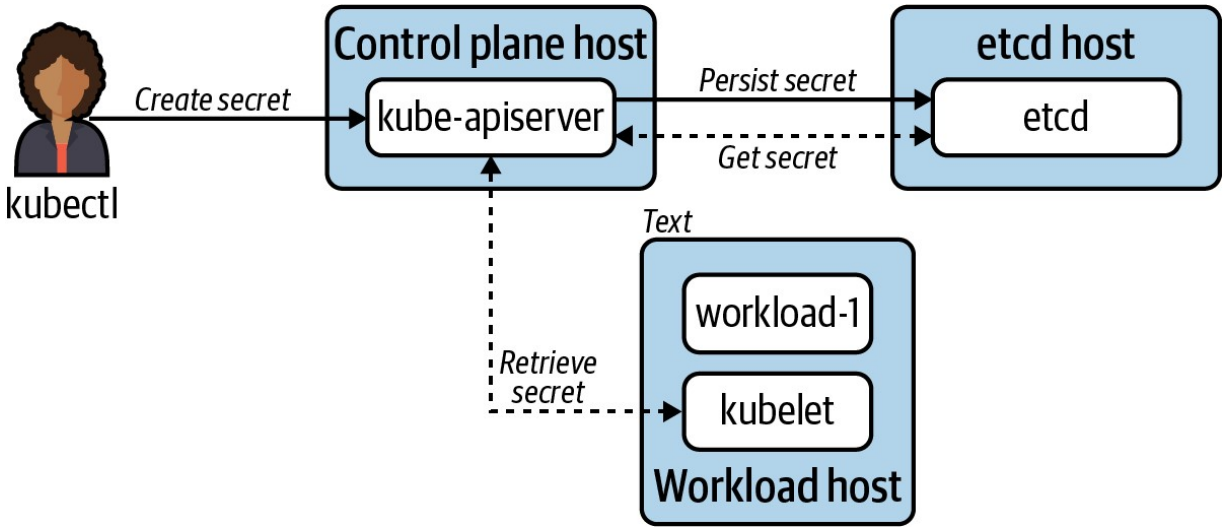
Encryption

Protection of secret data largely comes down to how secure we want our data to be.

Mainly, there are three layers of encryption:

- **Disk Encryption**: protects information at rest by converting it into unreadable code that cannot be deciphered easily by unauthorized people. It can be enabled in the filesystem and/or storage systems used by Kubernetes.
- **Transport Encryption**: protects information that is actively in flight by ensuring that communication between components is encrypted. This can be achieved by having all the internal components of Kubernetes and etcd interact over TLS. By default, all traffic between the Pods is not encrypted and will require an encrypted networking layer provided by `Service Meshes` over Mutual TLS (mTLS) such as Istio and Linkerd or Cilium.
- **Application Encryption**: protects information within our system components or workloads running in Kubernetes, for example, encrypting `secret` objects data.

The Kubernetes Secret API



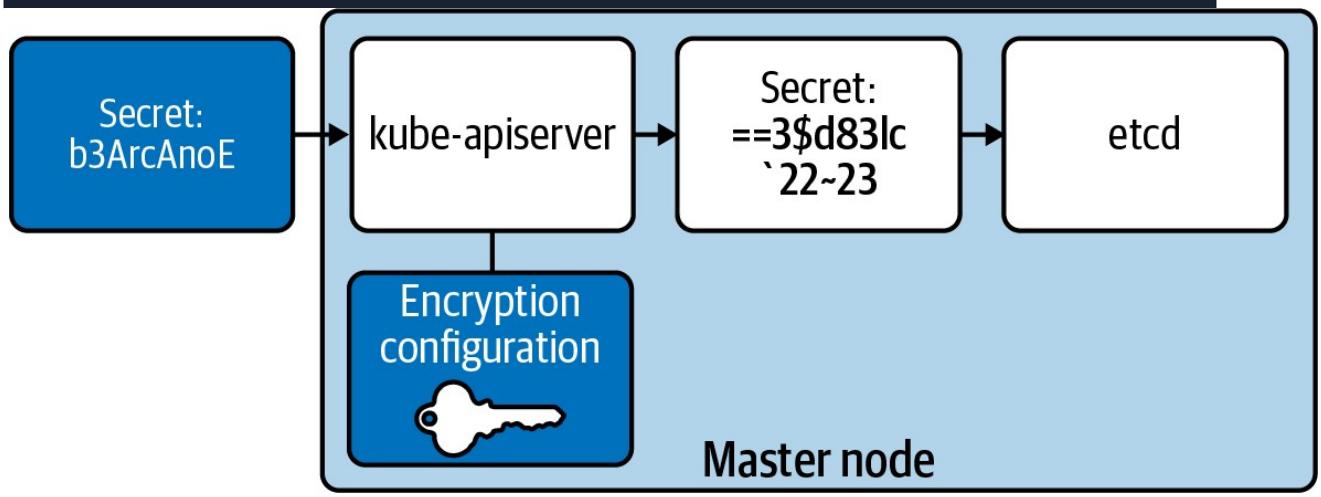
The Kubernetes Secret API is one of the most-used APIs in Kubernetes. While there are many ways for us to populate the Secret objects, the API provides a consistent means for workloads to interact with secret data.

By default, Secrets are encoded and not encrypted. This is the key mechanism for ensuring that Secrets aren't compromised is **RBAC**.

There are a number of different ways to encrypt Secrets data including: **Static-Key Encryption**, **Envelope Encryption**, **External Providers** and **CSI Integration**.

Static-Key Encryption

The Kubernetes API server supports encrypting secrets at rest. This is achieved by providing the Kubernetes API server with an encryption key, which it will use to encrypt all secret objects before persisting them to etcd.



The key, held within an **EncryptionConfiguration** object, is used to encrypt and decrypt Secret objects as they move through the API server.

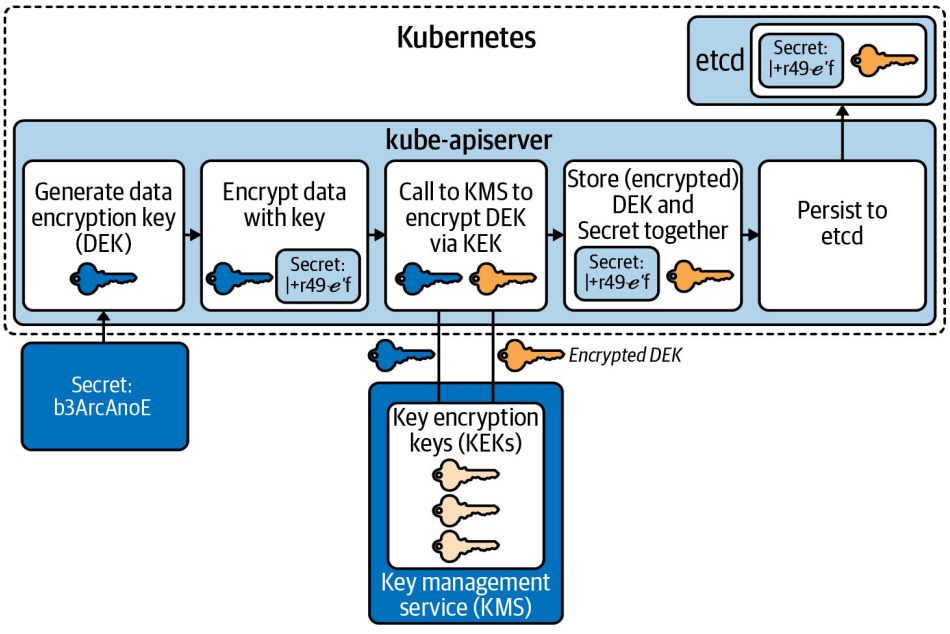
Should an attacker get access to etcd, they would see the encrypted data within, meaning the Secret data is not compromised.

Keys can be created using a variety of providers, including secretbox, aescbc, and aesgcm.

Envelope Encryption

Kubernetes supports integrating with a KMS (Key Management Service) to achieve envelope encryption.

- Envelope encryption involves two keys: the key encryption key (KEK) and the data encryption key (DEK).
- KEKs are stored externally in a KMS and aren't at risk unless the KMS provider is compromised.
- KEKs are used to encrypt DEKs, which are responsible for encrypting Secret objects.
- Each Secret object gets its own unique DEK to encrypt and decrypt the data.
- Since DEKs are encrypted by a KEK, they can be stored with the data itself, preventing the kube-apiserver from needing to be aware of many keys.
- Architecturally, the flow of envelope encryption would look like this:



There can be some variance in how this flow works, based on a KMS provider, but generally this demonstrates how envelope encryption functions

External Providers

Kubernetes is not generally considered an enterprise-grade secret store. While it does offer a Secret API that will be used for things like Service Accounts, for enterprise secret data it may fall short. Many clients demand more than what the Secret API can offer, especially those working in sectors such as financial services. These users need capabilities such as integration with a hardware security module (HSM) and have advanced key rotation policies.

By far, the largest and most popular open source secret management solution is Vault by HashiCorp.

CSI Integration

A newer approach to secret store integration is to leverage the secrets-store-csi-driver.

This approach enables integration with secret management systems at a lower level. Namely, it enables Pods to gain access to externally hosted secrets without running a sidecar or initContainer to inject secret data into the Pod.

The secrets-store-csi-driver runs a driver Pod (as a DaemonSet) on every host, similar to how you'd expect a CSI driver to work with a storage provider such as AWS Secrets and Configuration Provider or Azure Key Vault Provider.

First, let's create a new namespace to host our demo: