❹ Call sops with the public part of the age key, and store the result in *configmap_encrypted.yml*.

❺ Each value is replaced with an encrypted version in ENC[...] (output shortened for readability).

❻ The name of the ConfigMap is left untouched.

❼ An extra section, sops is appended to contain metadata that is needed for decryption.

❽ Encrypted session key that is used for symmetrical decryption. This key itself is encrypted asymmetrically by age.

As you can see, every value of the ConfigMap resource gets encrypted, even those that are not confidential, like resource types or the name of the resource. You can skip the encryption for specific values by appending an _unencrypted suffix to the key (which gets stripped off later when doing the decryption).

The generated *configmap_encrypted.yml* can safely be stored in Git or any other source control management. As shown in Example 25-5, you need the private key to decrypt the ciphered ConfigMap to apply it to the cluster.

Example 25-5. Decrypt sops-encoded resource and apply it to Kubernetes

```
$ export SOPS_AGE_KEY_FILE=keys.txt   ❶
$ sops --decrypt configmap_encrypted.yaml | kubectl apply -f - ❷
configmap/db-auth created
```

❶ Point sops to the private key to decrypt the session key.

❷ Decrypt and apply to Kubernetes. Note that every _unencrypted suffix on the resource keys is removed during sops decryption.

Sops is an excellent solution for easy GitOps-style integration of Secrets without worrying about installing and maintaining Kubernetes add-ons. However, while your configuration can now be stored securely in Git, it is essential to understand that as soon as those configurations have been handed over to the cluster, anybody with elevated access rights can read that data directly via the Kubernetes API.

If this is not something you can tolerate, we need to dig deeper into the toolbox and look again at centralized SMSs.

Centralized Secret Management

As explained in "How Secure Are Secrets?" on page 190, Secrets are as secure as possible. Still, any administrator with cluster-wide read access can read every Secret stored unencrypted. Depending on your trust relationship with your cluster operators and security requirements, this might or might not be a problem.

Besides baking individual secret handling into your application code, an alternative is to keep the secure information outside the cluster in the external SMS and request the confidential information on demand over secure channels.

There is a growing number of such SMSs out there, and every cloud provider offers its variant. We won't go into many details here for those individual offerings but focus on the mechanism of how such systems integrate into Kubernetes. You will find a list of relevant products as of 2023 in "More Information" on page 252.

Secrets Store CSI Driver

The Container Storage Interface (CSI) is a Kubernetes API for exposing storage systems to containerized applications. CSI shows the path for third-party storage providers to plug in new types of storage that can be mounted as volumes in Kubernetes. Of particular interest in the context of this pattern is the Secrets Store CSI Driver. This driver, developed and maintained by the Kubernetes community, allows access to various centralized SMSs and mounts them as regular Kubernetes volumes. The difference from a mounted Secret volume as described in Chapter 20, "Configuration Resource", is that nothing is stored in the Kubernetes etcd database but securely outside the cluster.

The Secrets Store CSI Driver supports the SMS from major cloud vendors (AWS, Azure, and GCP) and HashiCorp Vault.

The Kubernetes setup for connecting a secret manager via the CSI driver involves performing these two administrative tasks:

- Installing the Secrets Store CSI Driver and configuration for accessing a specific SMS. Cluster-admin permissions are required for the installation process.
- Configuring access rules and policies. Several provider-specific steps need to be completed, but the result is that a Kubernetes service account is mapped to a secret manager-specific role that allows access to the secrets.

Figure 25-4 shows the overall setup needed for enabling the Secrets Store CSI Driver with a HashiCorp Vault backend.

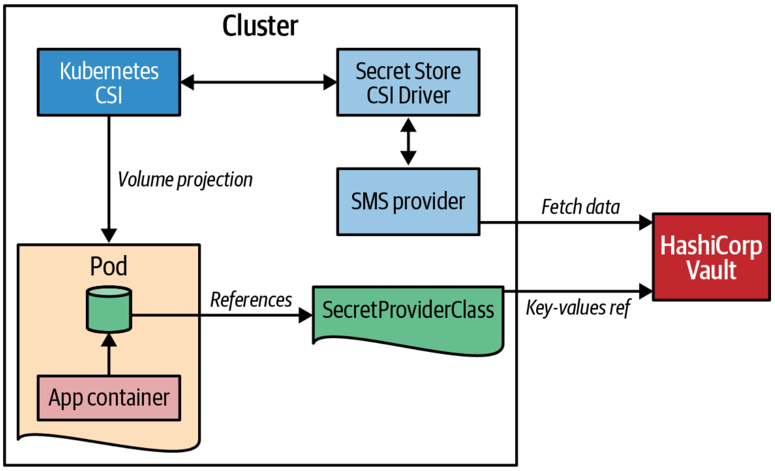

Figure 25-4. Secrets Store CSI Driver

After the setup is done, the usage of secret volumes is straightforward. First, you must define a SecretProviderClass, as demonstrated in Example 25-6. In this resource, you select the backend provider for the secret manager. For our example, we selected HashiCorp's Vault. In the `parameters` section, the provider-specific configuration is added, which contains the connection parameter to the vault, the role to impersonate, and a pointer to the secret information that Kubernetes will mount into a Pod.

Example 25-6. Configuration of how to access a secret manager

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: vault-database
spec:
  provider: vault                                ❶
  parameters:
    vaultAddress: "http://vault.default:8200"    ❷
    roleName: "database"                         ❸
    objects: |
      - objectName: "database-password"          ❹
        secretPath: "secret/data/database-creds" ❺
        secretKey: "password"                    ❻
```

❶ Type of provider to use (`azure`, `gcp`, `aws`, or `vault` as of 2023).

❷ Connection URL to the Vault service instance.

❸ Vault-specific authentication role contains the Kubernetes service account allowed to connect.

❹ Name of the file that should be mapped into the mounted volume.

❺ Path to the stored secret in the vault.

❻ Key to pick from the Vault secret.

This secret manager configuration can then be referenced by its name when used as a Pod volume. Example 25-7 shows a Pod that mounts the secrets configured in Example 25-6. One key aspect is the service account `vault-access-sa` with which this Pod runs. This service account must be configured on the Vault side to be part of the role `database` referenced in the SecretProviderClass.

You can find this Vault configuration in our complete working and self-contained example, along with setup instructions.

Example 25-7. Pod mounting a CSI volume from Vault

```
kind: Pod
apiVersion: v1
metadata:
  name: shell-pod
spec:
  serviceAccountName: vault-access-sa   ❶
  containers:
  - image: k8spatterns/random
    volumeMounts:
    - name: secrets-store
      mountPath: "/secrets-store"       ❷
  volumes:
    - name: secrets-store
      csi:                              ❸
        driver: secrets-store.csi.k8s.io
        readOnly: true
        volumeAttributes:
          secretProviderClass: "vault-database" ❹
```

❶ Service account that is used to authenticate against Vault.

❷ Directory in which to mount the secrets.

❸ Declaration of a CSI Driver, which points to the Secret Store CSI driver.

❹ Reference to the SecretProviderClass that provides the connection to the Vault service.

While the setup for a CSI Secret Storage drive is quite complex, the usage is straightforward, and you can avoid storing confidential data within Kubernetes. However, there are more moving parts than with Secrets alone, so more things can go wrong, and it's harder to troubleshoot.

Let's look at a final alternative for offering secrets to applications via well-known Kubernetes abstractions.

Pod injection

As mentioned, an application can always access external SMSs via proprietary client libraries. This approach's disadvantage is that you still have to store the credentials to access the SMS along your application and add a hard dependency within your code to a particular SMS. The CSI abstraction for projecting secret information into volumes visible as files for the deployed application is much more decoupled.

Alternative solutions leverage other well-known patterns described in this book:

- An *Init Container* (see Chapter 15) fetches the confidential data from an SMS and then copies it to a shared local volume that is mounted by the application container. The secret data is fetched only once before the main container starts.
- A *Sidecar* (see Chapter 16) syncs the secret data from the SMS to a local ephemeral volume that is also accessed by the application. The benefit of the sidecar approach is that it can update the secrets locally in case the SMS starts to rotate the secrets.

You can leverage these patterns on your own for your applications, but this is tedious. It is much better to let an external controller inject the init container or sidecar into your application.

An excellent example of such an injector is the HashiCorp Vault Sidecar Agent Injector. This injector is implemented as a so-called *mutating webhook*, a variant of a controller (see Chapter 27, "Controller"), that allows modification of any resource when it is created. When a Pod specification contains a particular, vault-specific annotation, the vault controller will modify this specification to add a container for syncing with Vault and to mount a volume for the secret data.

Figure 25-5 visualizes this technique, which is entirely transparent to the user.

While you still need to install the Vault Injector controller, it has fewer moving parts than hooking up a CSI secret storage volume with the provider deployment for a particular SMS product. Still, you can access all the secrets by just reading a file without using a proprietary client library.

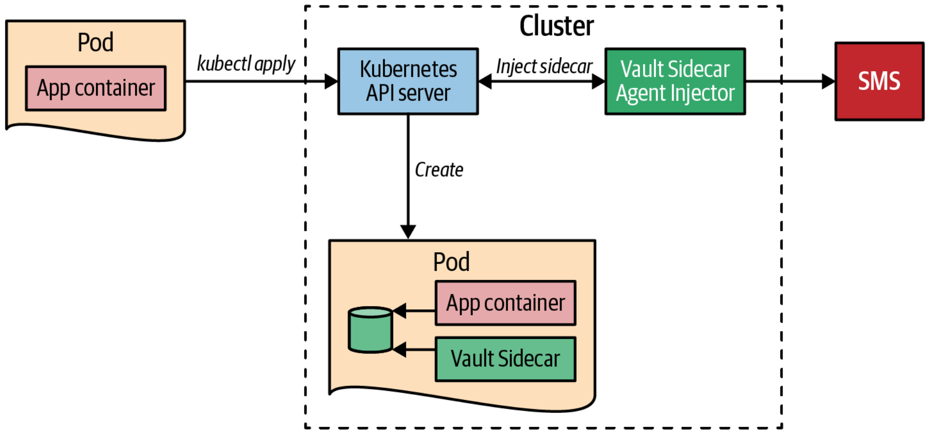

Figure 25-5. Vault Injector

Discussion

Now that we have seen the many ways you can make access to your confidential information more secure, the question is, which one is the best?

As usual, it depends:

- If your main goal is a simple way to encrypt Secrets stored in public-readable places like a remote Git repository, the pure *client-side* encryption that *Sops* offers is perfect.
- The secret synchronization that the *External Secrets Operator* implements is a good choice when separating the concerns of retrieving credentials in a remote SMS and using them is essential.
- The ephemeral volume projection of secret information provided by *Secret Storage CSI Providers* is the right choice for you when you want to ensure that no confidential information is stored permanently in the cluster except the access tokens for accessing external vaults.
- Sidebar injections like the *Vault Sidecar Agent Injector* have the benefit of shielding from a direct access to an SMS. They are easily approachable at the cost of blurring the boundary between developers and administrator because of security annotations leaking into application deployment.

Note that the listed projects are the most prominent as of this writing in 2023. The landscape is constantly evolving, so by the time you read this book, there might be new contenders (or some existing projects might have stopped). However, the techniques used (client-side encryption, Secret synchronization, volume projections, and sidecar injections) are universal and will be part of future solutions.

But a clear warning at the end: regardless of how securely and safely you can access your secret configuration, if somebody with evil intentions has full root access to your cluster and containers, a means to get to that data will always exist. This pattern makes these kinds of exploits as difficult as possible by adding an extra layer on the Kubernetes Secret abstraction.

More Information

- Secure Configuration Example
- Alex Soto Bueno and Andrew Block's *Kubernetes Secrets Management* (Manning, 2022)
- Kubernetes: Sealed Secrets
- Sealed Secrets
- External Secrets Operator
- Kubernetes External Secrets
- Sops
- Kubernetes Secrets Store CSI Driver
- Retrieve HashiCorp Vault Secrets with Kubernetes CSI
- HashiCorp Vault
- Secret Management Systems:
  - — Azure Key Vault
  - — AWS Secrets Manager
  - — AWS Systems Manager Parameter Store
  - — GCP Secret Manager

CHAPTER 26

Access Control

As the world becomes increasingly reliant on cloud infrastructure and containerization, the importance of security can never be understated. In 2022, security researchers made a troubling discovery: nearly one million Kubernetes instances were left exposed on the internet due to misconfigurations.[1] Using specialized security scanners, researchers were able to easily access these vulnerable nodes, highlighting the need for stringent access-control measures to protect the Kubernetes control plane. But while developers often focus on application-level authorization, they sometimes also need to extend Kubernetes capabilities using the *Operator* pattern from Chapter 28. In these cases, access control on the Kubernetes platform becomes critical. In this chapter, we delve into the *Access Control* pattern and explore the concepts of Kubernetes authorization. With the potential risks and consequences at stake, it's never been more important to ensure the security of your Kubernetes deployment.

Problem

Security is a crucial concern when it comes to operating applications. At the core of security are two essential concepts: authentication and authorization.

Authentication focuses on identifying the subject, or *who*, of an operation and preventing access by unauthorized actors. *Authorization*, on the other hand, involves determining the permissions for *what* actions are allowed on resources.

In this chapter, we will discuss authentication briefly, as it is primarily an administrative concern that involves integrating various identity-management techniques with Kubernetes. On the other hand, developers are typically more concerned with

1 See the blog post "Exposed Kubernetes Clusters".

authorization, such as who can perform which operations in the cluster and access specific parts of an application.

To secure access to their applications running on top of Kubernetes, developers must consider a range of security strategies, from simple web-based authentication to sophisticated single-sign-on scenarios involving external providers for identity and access management. At the same time, access control to the Kubernetes API server is also an essential concern for applications running on Kubernetes.

Misconfigured access can lead to privilege escalation and deployment failures. High-privilege deployments can access or modify configuration and resources for other deployments, increasing the risk of a cluster compromise.[2] It is important for developers to understand the authorization rules set up by administrators and consider security when making configuration changes and deploying new workloads to meet the organization-wide policies in the Kubernetes cluster.

Furthermore, as more and more Kubernetes-native applications extend the Kubernetes API and offer their services via CustomResourceDefinitions (CRDs) to users, as described in "Controller and Operator Classification" on page 297, access control becomes even more critical. Kubernetes patterns like Chapter 27, "Controller", and Chapter 28, "Operator", require high privileges to observe the state of cluster-wide resources, making it crucial to have fine-grained access management and restrictions in place to limit the impact of any potential security breaches.

Solution

Every request to the Kubernetes API server has to pass through three stages—Authentication, Authorization, and Admission Control, as shown in Figure 26-1.

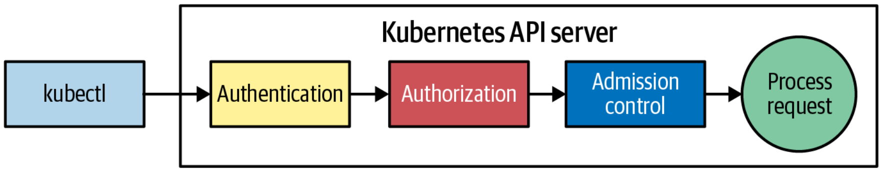

Figure 26-1. A request to the Kubernetes API server must pass through these stages

Once a request passes the Authentication and Authorization stages described in the following sections, a final check is done by Admission controllers before the request is eventually processed. Let's look at these stages separately.

2 An attacker with escalated privileges on a node can compromise a full Kubernetes cluster.

Authentication

As mentioned, we won't go into too much detail about authentication because it is mainly an administration concern. But it's good to know which options are available, so let's have a look at the pluggable authentication strategies Kubernetes has to offer that an administrator can configure:

Bearer Tokens (OpenID Connect) with OIDC Authenticators
: OpenID Connect (OIDC) Bearer Tokens can authenticate clients and grant access to the API Server. OIDC is a standard protocol that allows clients to authenticate with an OAuth2 provider that supports OIDC. The client sends the OIDC token in the Authorization header of their request, and the API Server validates the token to allow access. For the entire flow, see the Kubernetes documentation at OpenID Connect Tokens.

Client certificates (X.509)
: By using client certificates, the client presents a TLS certificate to the API Server, which is then validated and used to grant access.

Authenticating Proxy
: This configuration option refers to using a custom authenticating proxy to verify the client's identity before granting access to the API Server. The proxy acts as an intermediary between the client and the API Server and performs authentication and authorization checks before allowing access.

Static Token files
: Tokens can also be stored in standard files and used for authentication. In this approach, the client presents a token to the API Server, which is then used to look up the token file and search for a match.

Webhook Token Authentication
: A webhook can authenticate clients and grant access to the API Server. In this approach, the client sends a token in the Authorization header of their request, and the API Server forwards the token to a configured webhook for validation. The client is granted access to the API Server if the webhook returns a valid response. This technique is similar to the Bearer Token option, except that you can use an external custom service for performing the token validation.

Kubernetes allows you to use multiple authentication plugins simultaneously, such as Bearer Tokens and Client certificates. If the Bearer Token strategy authenticates a request, Kubernetes won't check the Client certificates, and vice versa. Unfortunately, the order in which these strategies are evaluated is not fixed, so it's impossible to know which one will be checked first. When evaluating the strategies, the process will stop after one is successful, and Kubernetes will forward the request to the next stage.

After authentication, the authorization process will begin.

Authorization

Kubernetes provides RBAC as a standard way to manage access to the system. RBAC allows developers to control and execute actions in a fine-grained manner. The authorization plugin in Kubernetes also provides easy pluggability, allowing users to switch between the default RBAC and other models, such as attribute-based access control (ABAC), webhooks, or delegation to a custom authority.

The ABAC-based approach requires a file containing policies in a JSON per-line format. However, this approach requires the server to be reloaded for any changes, which can be a disadvantage. This static nature is one of the reasons ABAC-based authorization is used only in some cases.

Instead, nearly every Kubernetes cluster uses the default RBAC-based access control, which we describe in great detail in "Role-Based Access Control" on page 263.

Before we focus on authorization in the rest of this chapter, let's quickly look at the last stage performed by admission controllers.

Admission Controllers

Admission controllers are a feature of the Kubernetes API server that allows you to intercept requests to the API server and take additional actions based on those requests. For example, you can use them to enforce policies, perform validations, and modify incoming resources.

Kubernetes uses Admission controller plugins for implementing various functions. The functionality ranges from setting default values on specific resources (like the default storage class on persistent volumes), to validations (like the allowed resource limits for Pods), by calling external web hooks.

These external webhooks can be configured with dedicated resources and are used for validation (ValidatingWebhookConfiguration) and updating (MutatingWebhookConfiguration) API resources. The details of configuring such webhooks are explained in detail in the Kubernetes documentation "Dynamic Admission Control".

We won't go into more detail here as Admission controllers are mostly an administrative concept, and many other good resources describe Admission controllers in particular (see "More Information" on page 275 for some references).

Instead, for the remainder of the chapter, we will focus on the authorization aspect and how we can configure a fine-grained permission model for securing access to the Kubernetes API server.

As mentioned, authentication has two fundamental parts and authorization: the *who*, represented by a subject that can be either a human person or a workload identity, and the *what*, representing the actions those subjects can trigger at the Kubernetes

API server. In the next section, we discuss the who before diving into the details of the *what*.

Subject

A *subject* is all about the *who*, the identity associated with a request to the Kubernetes API server. In Kubernetes, there are two kinds of subjects, as shown in Figure 26-2: human *users* and *service accounts* that represent the workload identity of Pods.

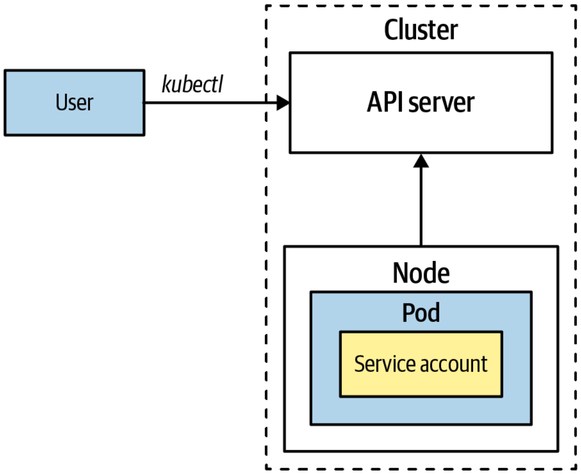

Figure 26-2. Subject (user or service account) requests to API Server

Human users and ServiceAccounts can be separately grouped in *user groups* and *service account groups*, respectively. Those groups can act as a single subject in which all members of the group share the same permission model. We will talk about groups later in this chapter, but first, let's look closely at how human users are represented in the Kubernetes API.

Users

Unlike many other entities in Kubernetes, human users are not defined as explicit resources in the Kubernetes API. This design decision implies that you can't manage users via an API call. The authentication and mapping to a user subject happens outside the usual Kubernetes API machinery by external user management.

As we have seen, Kubernetes supports many ways of authenticating an external user. Each component knows how to extract the subject information after successful authentication. Although this mechanism is different for each authentication component, they will eventually create the same user representation and add it to the actual API request to verify by later stages, as shown in Example 26-1.

Example 26-1. Representation of an external user after successful authentication

```
alice,4bc01e30-406b-4514,"system:authenticated,developers","scopes:openid"
```

This comma-separated list is a representation of the user and contains the following parts:

- The username (`alice`)
- A unique user id (UID) (`4bc01e30-406b-4514`)
- A list of groups that this user belongs to (`system:authenticated,developers`)
- Additional information as comma-separated key-value pairs (`scopes:openid`)

This information is evaluated by the Authorization plugin against the authorization rules associated with the user or via its membership to a user group. In Example 26-1, a user with the username `alice` has the default access associated with the group `system:authenticated` and the group `developers`. The extra information `scope:openid` indicates OIDC is being used to verify the user's identity.

Certain usernames are reserved for internal Kubernetes use and are distinguished by the special prefix `system:`. For example, the username `system:anonymous` represents anonymous requests to the Kubernetes API server. It is recommended to avoid creating your own users or groups with the `system:` prefix to avoid conflicts. Table 26-1 lists the default usernames in Kubernetes that are used when internal Kubernetes components communicate to one another.

Table 26-1. Default usernames in Kubernetes

Username	Purpose
`system:anonymous`	Represents anonymous requests to the Kubernetes API server
`system:apiserver`	Represents the API server itself
`system:kube-proxy`	Represents process identity of the kube-proxy service
`system:kube-controller-manager`	Represents the user agent of the controller manager
`system:kube-scheduler`	Represents the user of the scheduler

While the management and authentication of external users can vary depending on the specific setup of a Kubernetes cluster, the management of workload identities for Pods is a standardized part of the Kubernetes API and is consistent across all clusters.

Service accounts

Service accounts in Kubernetes represent nonhuman actors within the cluster and are used as workload identities. They are associated with Pods and allow running processes inside a Pod to communicate with the Kubernetes API Server. In contrast to the many ways that Kubernetes can authenticate human users, service accounts always use an OpenID Connect handshake and JSON Web Tokens to prove their identity.

Service accounts in Kubernetes are authenticated by the API server using a username in the following format: `system:serviceaccount:<namespace>:<name>`. For example, if you have a service account, `random-sa`, in the `default` namespace, the service account's username would be `system:serviceaccount:default:random-sa`.

JSON Web Tokens in Kubernetes

JSON Web Tokens (JWTs) are digitally signed tokens that carry a payload. They consist of a header, payload, and signature and are represented as a sequence of Base64 URL-encoded parts separated by periods. Tools like jwt.io can decode, validate, and inspect JWTs.

In the context of Kubernetes, JWTs are used as Bearer Tokens in the `Authorization` HTTP header of API requests to specify the identity of the workload making the request and additional information, such as the expiration time or issuer. The Kubernetes API server verifies the signature of the JWT by comparing it with a public key published in a JSON Web Key Set (JWKS). This process is governed by the JSON Web Key (JWK) specification, which defines the cryptographic algorithms used in the verification process in RFC 7517.

The tokens issued by Kubernetes contain helpful information in the payload of the JWT, such as the issuer of the token, its expiration time, all the user information described in Example 26-1, and the associated service accounts (if any).

A ServiceAccount is a standard Kubernetes resource, as shown in Example 26-2.

Example 26-2. ServiceAccount definition

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: random-sa                  ❶
  namespace: default
automountServiceAccountToken: false ❷
...
```

❶ Name of the service account.

❷ Flag indicating whether the service account token should be mounted by default into a Pod. The default is set to `true`.

A ServiceAccount has a simple structure and serves all identity-related information needed for a Pod when talking with the Kubernetes API server. Every namespace has a default ServiceAccount with the name `default` used to identify any Pod that does not define an associated ServiceAccount.

Each ServiceAccount has a JWT associated with it that is fully managed by the Kubernetes backend. A Pod's associated ServiceAccount's token is automatically mounted into the filesystem of each Pod. Example 26-3 shows the relevant part of a Pod specification that Kubernetes has automatically added for every Pod created.

Example 26-3. ServiceAccount token mounted as a file for a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: random
spec:
  serviceAccountName: default          ❶
  containers:
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount ❷
      name: kube-api-access-vzfp7      ❸
      readOnly: true
  ...
  volumes:
  - name: kube-api-access-vzfp7
    projected:                         ❹
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3600      ❺
          path: token                  ❻
    ...
```

❶ `serviceAccountName` to set the name of the service account (`serviceAccount` is a deprecated alias for `serviceAccountName`).

❷ */var/run/secrets/kubernetes.io/serviceaccount* is the directory under which the service account token is mounted.

❸ Kubernetes assigns a random Pod-unique name to the auto-generated volume.

❹ A projected volume injects the ServiceAccount token directly into the filesystem.

❺ Expiration time of the token in seconds. After this time, the token expires, and the mounted token file is updated with a new token.

❻ The name of the file that will contain the token.

To view the mounted token, we can execute a `cat` on the mounted file in the running Pod, as shown in Example 26-4.

Example 26-4. Print out the service account JWT (output is shortened)

```
$ kubectl exec random -- \
     cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6InVHYV9NZEVYOEZteUNUZFl...
```

In Example 26-3, the token is mounted into the Pod as a projected volume. Projected volumes allow you to merge multiple volume sources, such as Secret and ConfigMap volumes (described in Chapter 20, "Configuration Resource"), into a single directory. With this volume type, the ServiceAccount token can also be directly mapped into the Pod's filesystem using a `serviceAccountToken` subtype. This method has several benefits, including reducing the attack surface by eliminating the need for an intermediate representation of the token and by providing the ability to set an expiration time for the token, which the Kubernetes token controller will rotate after it expires. Furthermore, the token injected into the Pod will be valid only for the duration of the Pod's existence, further reducing the risk of unauthorized inspection of the service account's token.

Before Kubernetes 1.24, Secrets were used to represent these tokens and were mounted directly with a `secret` volume type, which had the disadvantage of long lifetimes and lack of rotation. Thanks to the new projected volume type, the token is available only to the Pod and is not exposed as an additional resource, which reduces the attack surface. You can still create a Secret manually to contain a ServiceAccount's token, as demonstrated in Example 26-5.

Example 26-5. Create a Secret for ServiceAccount `random-sa`

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token                ❶
metadata:
  name: random-sa
  annotations:
    kubernetes.io/service-account.name: "random-sa" ❷
```

❶ Special type to indicate that this Secret is about holding a ServiceAccount.

❷ Reference to ServiceAccount, whose token should be added.

Kubernetes will fill in the token and the public key for validation into the secret. Also, the lifecycle of this Secret is now bound to the ServiceAccount itself. If you delete the ServiceAccount, Kubernetes will also delete this secret.

The ServiceAccount resource has two additional fields for specifying credentials for pulling container images and defining the secrets allowed to be mounted:

Image pull secrets
: Image pull secrets allow a workload to authenticate with a private registry when pulling images. Typically, you would need to manually specify the pull secrets as part of the Pod specification in the fields `.spec.imagePullSecrets`. However, Kubernetes provides a shortcut by allowing you to attach a pull secret directly to a ServiceAccount in the top-level field `imagePullSecrets`. Every Pod associated with the ServiceAccount will automatically have the pull secrets injected into its specification when it is created. This automation eliminates the need to manually include the image pull secrets in the Pod specification every time a new Pod is created in the namespace, reducing the manual effort required.

Mountable secrets
: The `secrets` field in the ServiceAccount resource allows you to specify which secrets a Pod associated with the ServiceAccount can mount. You can enable this restriction by adding the `kubernetes.io/enforce-mountable-secrets` annotation to the ServiceAccount. If this annotation is set to `true`, only the Secrets listed will be allowed to be mounted by Pods associated with the ServiceAccount.

Groups

Both user and service accounts in Kubernetes can belong to one or more groups. Groups are attached to requests by the authentication system and are used to grant permissions to all group members. As seen in Example 26-1, group names are plain strings that represent the group name.

As mentioned earlier, groups can be freely defined and managed by the identity provider to create groups of subjects with the same permission model. A set of predefined groups in Kubernetes are also implicitly defined and have a `system:` prefix in their name. These predefined groups are listed in Table 26-2.

We will see how group names can be used in a RoleBinding to grant permissions to all group members in "RoleBinding" on page 267.

Table 26-2. System groups in Kubernetes

Group	Purpose
`system:unauthenticated`	Group assigned to every unauthenticated request
`system:authenticated`	Group assigned to an authenticated user
`system:masters`	Group whose members have unrestricted access to the Kubernetes API server
`system:serviceaccounts`	Group with all ServiceAccounts of the cluster
`system:serviceaccounts:<namespace>`	Group with all ServiceAccounts of this namespace

Now that you have a clear understanding of users, ServiceAccounts, and groups, let's examine how these subjects can be associated with Roles that define the actions they are allowed to perform against the Kubernetes API server.

Role-Based Access Control

In Kubernetes, Roles define the specific actions that a subject can perform on particular resources. You can then assign these Roles to subjects, such as users or service accounts, as described in "Subject" on page 257, through the use of RoleBindings. Roles and RoleBindings are Kubernetes resources that can be created and managed like any other resource. They are tied to a specific namespace and apply to its resources.

Figure 26-3 illustrates the relationship between subjects, Roles, and RoleBindings.

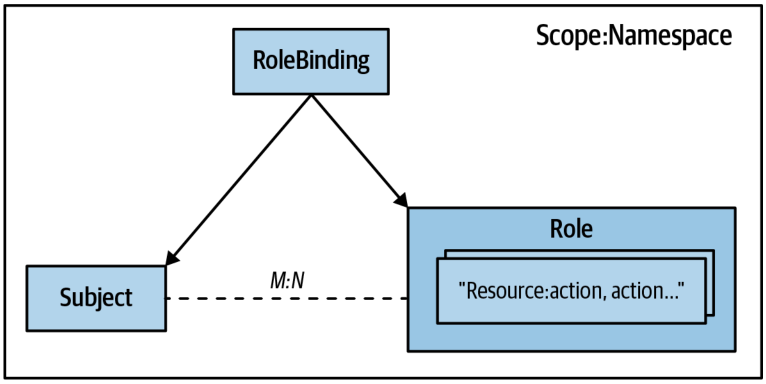

Figure 26-3. Relationship between Role, RoleBinding, and subjects

In Kubernetes RBAC, it is important to understand that there is a many-to-many relationship between subjects and Roles. This means that a single subject can have multiple Roles, and a single Role can be applied to multiple subjects. The relationship between a subject and a Role is established using a RoleBinding, which contains references to a list of subjects and a specific Role.

The RBAC concepts are best explained with a concrete example. Example 26-6 shows the definition of a Role in Kubernetes.

Example 26-6. Role for allowing access to core resources

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer-ro ❶
  namespace: default ❷
rules:
- apiGroups:
  - ""                ❸
  resources:          ❹
  - pods
  - services
  verbs:              ❺
  - get
  - list
  - watch
```

❶ The name of the Role, which is used to reference it.

❷ Namespace to which this Role applies. Roles are always connected to a namespace.

❸ An empty string indicates the core API group.

❹ List of Kubernetes core resources to which the rule applies.

❺ API actions are represented by verbs allowed by subjects associated with this Role.

The Role defined in Example 26-6 specifies that any user or service account associated with this Role can perform read-only operations on Pods and Services.

This Role can then be referenced in the RoleBinding shown in Example 26-7 to grant access to both the user, `alice`, and the ServiceAccount, `contractor`.

Example 26-7. RoleBinding specification

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-rolebinding
subjects:               ❶
- kind: User            ❷
  name: alice
  apiGroup: "rbac.authorization.k8s.io"
- kind: ServiceAccount    ❸
  name: contractor
  apiGroup: ""
roleRef:
  kind: Role              ❹
  name: developer-ro
  apiGroup: rbac.authorization.k8s.io
```

❶ List of subjects to connect to a Role.

❷ Human user reference for a user named `alice`.

❸ Service account with name `contractor`.

❹ Reference to the Role with the name `developer-ro` that has been defined in Example 26-6.

Now that you have a basic understanding of the relationship between subjects, Roles, and RoleBindings, let's delve deeper into the specifics of Roles and RoleBindings.

Role

Roles in Kubernetes allow you to define a set of permitted actions for a group of Kubernetes resources or subresources. Typical activities on Kubernetes resources include the following:

- Getting Pods
- Deleting Secrets
- Updating ConfigMaps
- Creating ServiceAccounts

You have already seen a Role in Example 26-6. Besides metadata, such as names and namespaces, a Role definition consists of a list of rules that describe which resources can be accessed.

Only one rule must match a request to grant access to this Role. Three fields describe each `rule`:

apiGroups
: This list is used rather than a single value because wildcards can specify all resources of multiple API groups. For example, an empty string (`""`) is used for the core API group, which contains primary Kubernetes resources such as Pods and Services. A wildcard character (*) can match all available API groups the cluster is aware of.

resources
: This list specifies the resources that Kubernetes should grant access to. Each entry should belong to at least one of the configured `apiGroups`. A single `*` wildcard entry means all resources from all configured `apiGroups` are allowed.

verbs
: Allowed actions in a system are defined using verbs that are similar to HTTP methods. These verbs include CRUD operations on resources (CRUD stands for *Create-Read-Update-Delete* and describes the usual read-write operations that you can perform on persistent entities), and separate actions for operations on collections, such as `list` and `deletecollection`. Additionally, a `watch` verb allows access to resource change events and is separate from directly reading the resource with `get`. This `watch` verb is crucial for operators to receive notifications about the current status of resources they are managing. Chapter 27, "Controller", and Chapter 28, "Operator", has more on this topic. Table 26-3 lists the most common verbs. Using the `*` wildcard character is also possible to allow all operations on the configured resources for a given rule.

Table 26-3. Kubernetes verb mapping to HTTP request methods for CRUD operations

Verbs	HTTP request methods
get, watch, list	GET
create	POST
patch	PATCH
update	PUT
delete, delete collection	DELETE

Wildcard permissions make it easier to define all operations without listing each option individually. All of the properties of a Role's `rule` element allow for an `*` wildcard, which matches everything. Example 26-8 allows for all operations on all resources in the core and `networking.k8s.io` API group. If a wildcard is used, this list should have only this wildcard as its single entry.

Example 26-8. Wildcard permission for resources and permitted operations

```
rules:
- apiGroups:
  - ""
  - "networking.k8s.io"
  resources:
  - "*"  ❶
  verbs:
  - "*"  ❷
```

❶ All Resources in the listed API groups, core, and `networking.k8s.io`.

❷ All actions are allowed on those resources.

Wildcards help developers to configure rules quickly. But they come with the security risk of privilege escalation. Such broader privileges can cause security gaps and allow users to perform any operations that can compromise the Kubernetes cluster or cause unwanted changes.

Now that we have looked into the *what* (Roles) and *who* (subjects) of the Kubernetes RBAC model, let's have a closer look at how we can combine both concepts with RoleBindings.

RoleBinding

In Example 26-7, we saw how RoleBindings link one or more subjects to a given Role.

Each RoleBinding can connect a list of subjects to a Role. The `subjects` list field takes resource references as elements. Those resource references have a `name` field plus `kind` and `apiGroup` fields for defining the resource type to reference.

A subject in a RoleBinding can be one of the following types:

User
: A user is a human or system authenticated by the API server, as described in "Users" on page 257. User entries have a fixed `apiGroup` value of `rbac.authorization.k8s.io`.

Group
: A group is a collection of users, as explained in "Groups" on page 262. As for users, the group entries carry a `rbac.authorization.k8s.io` as `apiGroup`.

ServiceAccount
: We discussed ServiceAccount in depth in "Service accounts" on page 258. ServiceAccounts belong to the core API Group that is represented by an empty string (`""`). One unique aspect of ServiceAccounts is that it is the only subject type that can also carry a `namespace` field. This allows you to grant access to Pods from other namespaces.

Table 26-4 summarizes the possible field values for entries in a RoleBinding's `subject` list.

Table 26-4. Possible types for an element `subjects` list in a RoleBinding

Kind	API Group	Namespace	Description
User	rbac.authorization.k8s.io	N/A	`name` is a reference to a user.
Group	rbac.authorization.k8s.io	N/A	`name` is a reference to a group of users.
ServiceAccount	""	Optional	`name` is a reference to a ServiceAccount resource in the configured namespace.

The other end of a RoleBinding points to a single Role. This Role can either be a Role resource within the same namespace as the RoleBinding or a ClusterRole resource shared across multiple bindings in the cluster. ClusterRoles are described in detail in "ClusterRole" on page 269.

Similar to the subjects list, Role references are specified by `name`, `kind`, and `apiGroup`. Table 26-5 shows the possible values for the `roleRef` field.

Table 26-5. Possible types for a `roleRef` field in a RoleBinding

Kind	API Group	Description
Role	rbac.authorization.k8s.io	`name` is a reference to a Role in the same namespace.
ClusterRole	rbac.authorization.k8s.io	`name` is a reference to cluster-wide ClusterRole.

Privilege-Escalation Prevention

The RBAC subsystem is responsible for managing Roles and RoleBindings (as well as ClusterRoles and ClusterRoleBindings). To prevent privilege escalation, in which users with permissions to control the RBAC resource elevate their permissions, the following restrictions apply:

- Users can update a Role only if they already have all the permissions in that Role or if they have permission to use the `escalate` verb on all resources in the `rbac.authorization.k8s` API group.
- For RoleBindings, a similar restriction applies: users must have all the permissions granted in the referenced Role, or they must have the `bind` verb allowance on the RBAC resources.

More information about these restrictions and how they help prevent privilege escalation can be found in the Kubernetes documentation "Privilege Escalation Prevention and Bootstrapping".

ClusterRole

ClusterRoles in Kubernetes are similar to regular Roles but are applied cluster-wide rather than to a specific namespace. They have two primary uses:

- Securing cluster-wide resources such as CustomResourceDefinitions or Storage-Classes. These resources are typically managed at the cluster-admin level and require additional access control. For example, developers may have read access to these resources but need help writing to them. ClusterRoleBindings are used to grant subjects access to cluster-wide resources.
- Defining typical Roles that are shared across namespaces. As we saw in "RoleBinding" on page 267, RoleBindings can refer only to Roles defined in the same namespace. ClusterRoles allow you to define general-access control Roles (e.g., "view" for read-only access to all resources) that can be used in multiple RoleBindings.

Example 26-9 shows a ClusterRole that can be reused in multiple RoleBindings. It has the same schema as a Role except that it ignores any `.meta.namespace` field.

Example 26-9. ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: view-pod ❶
rules:
- apiGroups:      ❷
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
```

❶ Name of the ClusterRole but no namespace declaration.

❷ Rule that allows reading operations on all Pods.

Figure 26-4 shows how a single ClusterRole can be shared across multiple RoleBindings in different namespaces. In this example, the ClusterRole allows the reading of Pods in the `dev-1` and `dev-2` namespaces by a ServiceAccount in the `test` namespace.

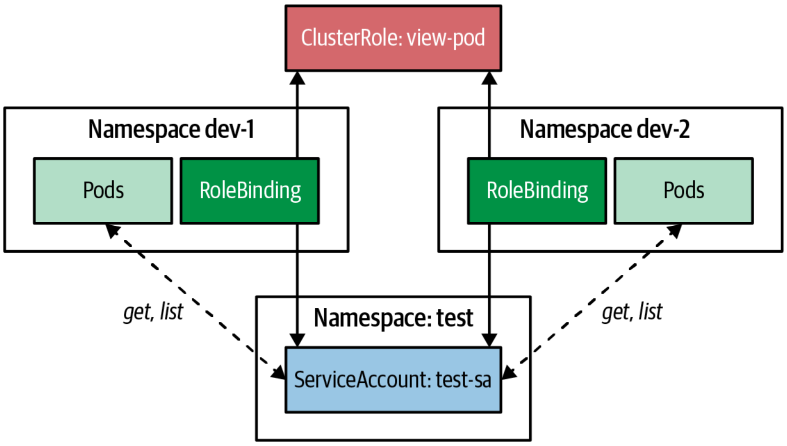

Figure 26-4. Sharing a ClusterRole in multiple namespaces

Using a single ClusterRole in multiple RoleBindings allows you to create typical access-control schemes that can be easily reused. For example, Table 26-6 includes a selection of useful user-facing ClusterRoles that Kubernetes provides out of the box. You can view the complete list of ClusterRoles available in a Kubernetes cluster using the `kubectl get clusterroles` command, or refer to the Kubernetes documentation for a list of default ClusterRoles.

Table 26-6. Standard user-facing ClusterRoles

ClusterRole	Purpose
`view`	Allows reading for most resources in a namespace, except Role, RoleBinding, and Secret
`edit`	Allows reading and modifying most resources in a namespace, except Role and RoleBinding
`admin`	Grants full control of all resources in a namespace, including Role and RoleBinding
`cluster-admin`	Grants full control of all namespace resources, including cluster-wide resources

Sometimes you may need to combine the permissions defined in two ClusterRoles. One way to do this is to create multiple RoleBindings that refer to both ClusterRoles. However, there is a more elegant way to achieve this using aggregation.

To use aggregation, you can define a ClusterRole with an empty `rules` field and a populated `aggregationRule` field containing a list of label selectors. Then, the rules defined by every other ClusterRole that has labels matching these selectors will be combined and used to populate the `rules` field of the aggregated ClusterRole.

When you set the `aggregationRule` field, you are handing ownership of the `rules` field over to Kubernetes, which will fully manage it. Therefore, any manual changes to the rules field will be constantly overwritten with the aggregated rules from the selected ClusterRoles in the `aggregationRule`.

This aggregation technique allows you to dynamically and elegantly build up large rule sets by combining smaller, more focused ClusterRoles.

Example 26-10 shows how the default `view` role uses aggregation to pick up more specific ClusterRoles labeled with `rbac.authorization.k8s.io/aggregate-to-view`. The `view` role itself also has the label `rbac.authorization.k8s.io/aggregate-to-edit`, which is used by the `edit` role to include the aggregated rules from the `view` ClusterRole.

Example 26-10. Aggregated ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: view
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"    ❶
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.authorization.k8s.io/aggregate-to-view: "true" ❷
rules: []  ❸
```

❶ This label exposes the ClusterRole as eligible for inclusion in the `edit` role.

❷ All ClusterRoles that match this selector will be picked up for the `view` ClusterRole. Note that this ClusterRole declaration does not need to be changed if you want to add additional permissions to the `view` ClusterRole—you can create a new ClusterRole with the appropriate label.

❸ The `rules` field will be managed by Kubernetes and populated with the aggregated rules.

This technique allows you to quickly compose more specialized ClusterRoles by aggregating a set of basic ClusterRoles. Example 26-10 also demonstrates how aggregation can be nested to build an inheritance chain of permission rule sets.

Since all of the user-facing default ClusterRoles use this aggregation technique, you can quickly hook into the permission model of custom resources (as described in

Chapter 28, "Operator") by simply adding the aggregation-triggering labels of the standard ClusterRoles (e.g., `view`, `edit`, and `admin`).

Now that we've covered the creation of a flexible and reusable permission model using ClusterRoles and RoleBindings, the final piece of the puzzle is establishing cluster-wide access rules with ClusterRoleBindings.

ClusterRoleBinding

The schema for a ClusterRoleBinding is similar to that of a RoleBinding, except that it ignores the `namespace` field. The rules defined in a ClusterRoleBinding apply to all namespaces in the cluster.

Example 26-11 shows a ClusterRoleBinding that connects a ServiceAccount `test-sa` with the ClusterRole `view-pod` defined in Example 26-9.

Example 26-11. ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: test-sa-crb
subjects:                ❶
- kind: ServiceAccount
  name: test-sa
  namespace: test
roleRef:                 ❷
  kind: ClusterRole
  name: view-pod
  apiGroup: rbac.authorization.k8s.io
```

❶ Connects ServiceAccount `test-sa` from the `test` namespace.

❷ Allows the rules from the ClusterRole `view-pod` for every namespace.

The rules defined in the ClusterRole `view-pod` apply to all namespaces in the cluster so that any Pod associated with the ServiceAccount `test-sa` can read all Pods in every namespace, which is illustrated in Figure 26-5. However, it is crucial to use ClusterRoleBindings with caution, as they grant wide-ranging permissions across the entire cluster. Therefore, it is recommended that you carefully consider whether using a ClusterRoleBinding is necessary.

Using a ClusterRoleBinding may be convenient as it automatically grants permissions to newly created namespaces. However, using individual RoleBindings per namespace is generally better for more granular control over permissions. This extra effort allows you to omit specific namespaces, such as `kube-system`, from unauthorized access.

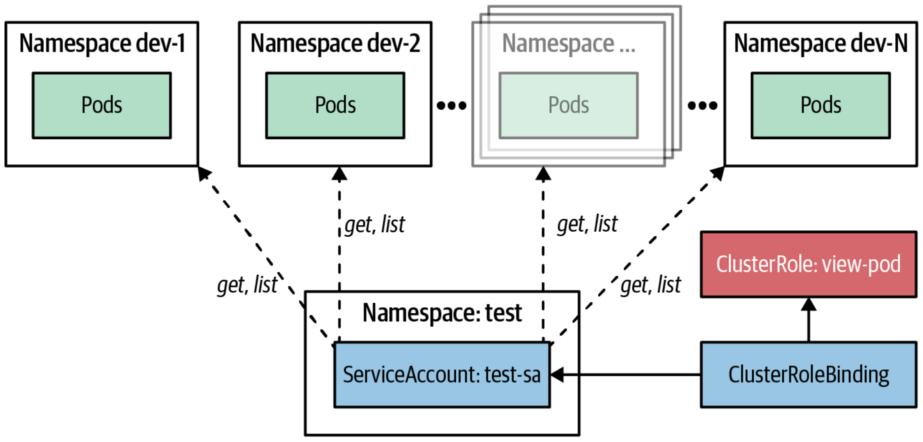

Figure 26-5. ClusterRoleBinding for reading all Pods

ClusterRoleBindings should be used only for administrative tasks, such as managing cluster-wide resources like Nodes, Namespaces, CustomResourceDefinitions, or even ClusterRoleBindings.

These final warnings conclude our tour through the world of Kubernetes RBAC. This machinery is mighty, but it's also complex to understand and sometimes even more complicated to debug. The following sidebar gives you some tips for better understanding a given RBAC setup.

Debugging RBAC Rules

In a Kubernetes cluster, many RBAC objects define the overall security model for accessing the API server. Understanding the authorization decisions made by the Kubernetes API server can be challenging, but the Access Review API can help by allowing you to query the authorization subsystem for permissions.

One way to use this API is through the `kubectl auth can-i` command. For example, you can use it to check whether a ServiceAccount named `test-sa` in the `test` namespace has permission to list all pods in the `dev-1` namespace. The command would look like Example 26-12. This command will return a simple "yes" or "no" indicating whether the ServiceAccount has the specified permission.

Example 26-12. Check access permissions with `kubectl`

```
kubectl auth can-i \
    list pods --namespace dev-1 --as system:serviceaccount:test:test-sa
```

Behind the scenes, a resource of the type SubjectAccessReview is created, and the Kubernetes authorization controller updates the `status` section of this resource with the result of the authorization check. You can read more about this API in the Kubernetes RBAC documentation.

While `kubectl auth can-i` helps check specific permissions, it can be tedious and does not provide a comprehensive overview of a subject's permissions across the cluster. To better understand what actions a subject can perform on all resources, tools like rakkess can be helpful. Rakkess is available as a `kubectl` plugin and can be run with the command `kubectl access-matrix`. It provides a matrix view of the actions a subject can perform on specific resources.

Another tool to help visualize and verify the application of fine-grained permissions is KubiScan, which allows you to scan a Kubernetes cluster for risky permissions in the RBAC configuration.

The final section will discuss some general tips for properly using Kubernetes RBAC.

Discussion

Kubernetes RBAC is a powerful tool for controlling access to API resources. However, it can be challenging to understand which definition objects to use and how to combine them to fit a particular security setup. Here are some guidelines to help you navigate these decisions:

- If you want to secure resources in a specific namespace, use a Role with a RoleBinding that connects to a user or ServiceAccount. The ServiceAccount does not have to be in the same namespace, allowing you to grant access to Pods from other namespaces.
- If you want to reuse the same access rules in multiple namespaces, use a Role-Binding with a ClusterRole that defines these shared-access rules.
- If you want to extend one or more existing predefined ClusterRoles, create a new ClusterRole with an `aggregationRule` field that refers to the ClusterRoles you wish to extend, and add your permissions to the `rules` field.
- If you want to grant a user or ServiceAccount access to all resources of a specific kind in all namespaces, use a ClusterRole and a ClusterRoleBinding.
- If you want to manage access to a cluster-wide resource like a CustomResource-Definition, use a ClusterRole and a ClusterRoleBinding.

We have seen how RBAC allows us to define fine-grained permissions and manage them. It can reduce risk by ensuring the applied permission does not leave gaps for the escalation path. On the other hand, defining any broad open permissions can lead to security escalations. Let's close this chapter with a summary of some general RBAC advice:

Avoid wildcard permissions

We recommend following the principle of least privilege when composing the fine-grained access control in the Kubernetes cluster. To avoid unintentional operations, avoid wildcard permissions when defining the Role and ClusterRoles. For rare occasions, it might make sense to use wildcards (i.e., to secure all resources of an API group), but it is a good practice to establish a general "no wildcard" policy that could be relaxed for well-reasoned exceptions.

Avoid `cluster-admin` *ClusterRole*

ServiceAccounts with high privileges can allow you to perform actions over any resources, like modifying permissions or viewing secrets in any namespace, which can lead to severe security implications. Therefore, never assign the `cluster-admin` ClusterRole to a Pod. Never.

Don't automount ServiceAccount tokens

By default, tokens of ServiceAccounts are mounted within a container's filesystem at */var/run/secrets/kubernetes.io/serviceaccount/token*. If such a Pod gets compromised, any attacker can talk with the API server with the permissions of the Pod's associated ServiceAccount. However, many applications don't need that token for business operations. For such a use case, avoid the token mount by setting the ServiceAccount's field `automountServiceAccountToken` to `false`.

Kubernetes RBAC is a flexible and powerful method for controlling access to the Kubernetes API. Therefore, even if your application is not directly interacting with the API Server to install your application and connect it to other Kubernetes servers, *Access Control* is a valuable pattern to secure the operation of your application.

More Information

- Access Control Example
- Escalation Paths
- Controlling Access to the Kubernetes API
- Auditing
- Admission Controllers Reference
- Dynamic Admission Control
- Kubernetes: Authentication Strategies
- RBAC Good Practices
- Workload Creation
- Bound Service Account Tokens
- BIG Change in K8s 1.24 About ServiceAccounts and Their Secrets