Configuring these fields for every Pod or container causes them to be prone to human errors. Unfortunately, setting them is usually the responsibility of the workload authors who are not typically the security subject-matter experts in the organization. That is why there are also cluster-level, policy-driven means defined by cluster administrators for ensuring all Pods in a namespace meet the minimum security standards. Let's briefly review that next.

Enforcing Security Policies

So far, we've explored setting security parameters of the container runtime using the `securityContext` definition as part of the Pod and container specifications. These specifications are created individually per Pod and usually indirectly through higher abstractions such as Deployments, Jobs, and CronJobs. But how can a cluster administrator or a security expert ensure that a collection of Pods follows certain security standards? The answer is in the Kubernetes Pod Security Standards (PSS) and Pod Security Admission (PSA) controller. PSS defines a common understanding and consistent language around security policies, and PSA helps enforce them. This way, the policies are independent of the underlying enforcement mechanism and can be applied through PSS or other third-party tools. These policies are grouped in three security profiles that are cumulative, from highly permissive to highly restrictive, as follows:

Privileged
: This is an unrestricted profile with the widest possible level of permissions. It is purposely left open and offers allow-by-default mechanisms for trusted users and infrastructure workloads.

Baseline
: This profile is for common noncritical application workloads. It has a minimally restrictive policy and provides a balance between ease of adoption and prevention from known privilege escalations. For example, it won't allow privileged containers, certain security capabilities, and even other configurations outside of the `securityContext` field.

Restricted
: This is the most restrictive profile that follows the latest security-hardening best practices at the expense of adoption. It is meant for security-critical applications, as well as lower-trust users. On top of the Baseline profile, it puts restrictions on the fields we reviewed earlier, such as `allowPrivilegeEscalation`, `runAsNonRoot`, `runAsUser`, and other container configurations.

PodSecurityPolicy was the legacy security-policy-enforcement mechanism that was replaced with PSA in Kubernetes v1.25. Going forward, you can use a third-party admission plugin or the built-in PSA controller to enforce the security standards for each namespace. The security standards are applied to a Kubernetes namespace using labels that define the standard level as described earlier and one or more actions to take when a potential violation is detected. Following are the actions you can take:

Warn
: The policy violations are allowed with a user-facing warning.

Audit
: The policy violations are allowed with an auditing log entry recorded.

Enforce
: Any policy violations will cause the Pod to be rejected.

With these options defined, Example 23-3 creates a namespace that rejects any Pods that don't satisfy the *baseline* standard, and also generates a warning for Pods that don't meet the *restricted* standards requirements.

Example 23-3. Set security standards for a namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: baseline              ❶
    pod-security.kubernetes.io/enforce-version: v1.25         ❷
    pod-security.kubernetes.io/warn: restricted               ❸
    pod-security.kubernetes.io/warn-version: v1.25
```

❶ Label hinting to the PSA controller to reject Pods that violate the *baseline* standard.

❷ Version of the security-standard requirements to use (optional).

❸ Label hinting to the PSA controller to warn about Pods that violate the *restricted* standard.

This example creates a new namespace and configures the security standards to apply to all Pods that will be created in this namespace. It is also possible to update the configuration of a namespace or apply the policy to one or all existing namespaces. For details on how to do this in the least distributive way, check out "More Information" on page 219.

Discussion

One of the common security challenges with Kubernetes is running legacy applications that are not implemented or containerized with Kubernetes security controls in mind. Running a privileged container can be a challenge on Kubernetes distributions or environments with strict security policies. Understanding how Kubernetes does process containment at runtime and configures security boundaries, as shown in Figure 23-1, will help you create applications that run on Kubernetes more securely. It is important to realize that a container is not only a packaging format and not only a resource isolation mechanism, but when configured properly, it is also a security fence.

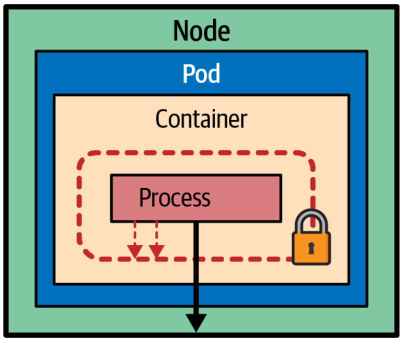

Figure 23-1. Process Containment pattern

The tendency of shifting left the security considerations and testing practices, including deploying into Kubernetes with the production security standards, is getting more popular. Such practices help identify and tackle security issues earlier in the development cycle and prevent last-minute surprises.

Shifting left is all about doing things earlier rather than later. It's about going leftward on the time ray that describes a development and deployment process. In our context, shift left implies that the developer already thinks about operational security when developing the application. See more details about the Shift Left model on Devopedia.

In this chapter, we hope that we have given you enough food for thought when creating secure cloud native applications. The guidelines in this chapter will help you design and implement applications that don't write to the local filesystem or require root privileges (for example, when containerizing applications, to ensure the container has a designated non-root user) and configure the security context. We hope that you understand exactly what your application needs and give it only the minimum permissions. We also aimed to help you build boundaries between the workloads and the host, to reduce container privileges and configuring the runtime environment to limit resource utilization in the event of a breach. In this endeavor, the *Process Containment* pattern ensures "what happens in a container stays in a container," including any security breaches.

More Information

- Process Containment Example
- Configure a Security Context for a Pod or Container
- Pod Security Admission
- Pod Security Standards
- Enforce Pod Security Standards with Namespace Labels
- Admission Controllers Reference: PodSecurity
- Linux Capabilities
- Introduction to Security Contexts and SCCs
- 10 Kubernetes Security Context Settings You Should Understand
- Security Risk Analysis Tool for Kubernetes Resources

CHAPTER 24

Network Segmentation

Kubernetes is a great platform for running distributed applications that communicate with one another over the network. By default, the network space within Kubernetes is flat, which means that every Pod can connect to every other Pod in the cluster. In this chapter, we will explore how to structure this network space for improved security and a lightweight multitenancy model.

Problem

Namespaces are a crucial part of Kubernetes, allowing you to group your workloads together. However, they only provide a grouping concept, imposing isolation constraints on the containers associated with specific namespaces. In Kubernetes, every Pod can talk to every other Pod, regardless of their namespace. This default behavior has security implications, particularly when multiple independent applications operated by different teams run in the same cluster.

Restricting network access to and from Pods is essential for enhancing the security of your application because not everyone may be allowed to access your application via an ingress. Outgoing egress network traffic for Pods should also be limited to what is necessary to minimize the blast radius of a security breach.

Network segmentation plays a vital role in multitenancy setups where multiple parties share the same cluster. For example, the following sidebar addresses some of the challenges of multitenancy on Kubernetes, such as creating network boundaries for applications.

Multitenancy with Kubernetes

Multitenancy refers to platform's ability to support multiple isolated user groups, also known as *tenants*. Kubernetes does not provide extensive support for multitenancy out of the box, and the concept itself can be complex and difficult to define. The Kubernetes documentation on Multitenancy covers various aspects and the support within the platform, including namespaces and access control (Chapter 26), quotas to prevent noisy neighbor issues, storage and network isolation, and handling of shared resources like cluster-wide DNS or CustomResourceDefinitions. In this chapter, we will focus on the network isolation aspects, which offer a softer approach to multitenancy. Stricter isolation requirements may require a more encapsulated approach, such as a virtual control plane per tenant, as provided by vcluster.

In the past, shaping the network topology was primarily the responsibility of administrators who managed firewalls and iptable rules. The challenge with this model is that administrators need to understand the networking requirements of the applications. In addition, the network graph can get very complex in a microservices world with many dependencies, requiring deep domain knowledge about the application. In this sense, the developer must communicate and sync information about dependencies with administrators. A DevOps setup can help, but the definition of network topologies is still far away from the application itself and can change dynamically over time.

So, what does defining and establishing a network segmentation look like in a Kubernetes world?

Solution

The good news is that Kubernetes shifts left these networking tasks so that developers using Kubernetes fully define their applications' networking topology. You have already seen this process model described briefly in Chapter 23, when we discussed the *Process Containment* pattern.

The essence of this *Network Segmentation* pattern is how we, as developers, can define the network segmentation for our applications by creating "application firewalls."

There are two ways to implement this feature that are complementary and can be applied together. The first is through the use of core Kubernetes features that operate on the L3/L4 networking layers.[1] By defining resources of the type NetworkPolicy, developers can create ingress and egress firewall rules for workload Pods.

The other method involves the use of a service mesh and targets the L7 protocol layer, specifically HTTP-based communication. This allows for filtering based on HTTP verbs and other L7 protocol parameters. We will explore Istio's AuthorizationPolicy later in this chapter.

To start, let's focus on how to use NetworkPolicies to define the network boundaries for your application.

Network Policies

NetworkPolicy is a Kubernetes resource type that allows users to define rules for inbound and outbound network connections for Pods. These rules act like a custom firewall and determine which Pods can be accessed and which destinations they can connect to. The user-defined rules are picked up by the Container Network Interface (CNI) add-on used by Kubernetes for its internal networking. However, not all CNI plugins support NetworkPolicies; for example, the popular Flannel CNI plugin does not support it, but many others, like Calico, do. All hosted Kubernetes cloud offerings support NetworkPolicy (either directly or by configuring an add-on) as well as other distributions like Minikube.

The NetworkPolicy definition consists of a selector for Pods and lists of inbound (ingress) or outbound (egress) rules.

The *Pod selector* is used to match the Pods to which the NetworkPolicy should be applied. This selection is done by using labels, which are metadata attached to Pods. The labels allow for a flexible and dynamic grouping of Pods, meaning that the same NetworkPolicy can be applied to multiple Pods that share the same labels and are running in the same namespace as the NetworkPolicy. Pod selectors are described in detail in "Labels" on page 8.

The list of *ingress* and *egress* rules defines which inbound and outbound connections are allowed for the Pods matched by the Pod selector. These rules specify which sources and destinations are allowed to connect to and from the Pods. For example, a rule could allow connections from a specific IP address or range of addresses, or it could block connections to a specific destination.

1 Level 3 and Level 4 of the OSI Network stack are mostly about IP and TCP/UDP, respectively.

Let's start with the simple example in Example 24-1 that allows access to all database Pods only from backend Pods and nothing else.

Example 24-1. Simple NetworkPolicy allowing ingress traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-database
spec:
  podSelector:          ❶
    matchLabels:
      app: chili-shop
      id: database
  ingress:              ❷
  - from:
    - podSelector:      ❸
        matchLabels:
          app: chili-shop
          id: backend
```

❶ Selector matching all Pods with the label `id: database` and `app: chili-shop`. All those Pods are affected by this NetworkPolicy.

❷ List of sources that are allowed for incoming traffic.

❸ Pod selector that will allow all Pods of the type `backend` to access the selected database Pods.

Figure 24-1 shows how the backend Pods can access the database Pods but frontend Pods can't.

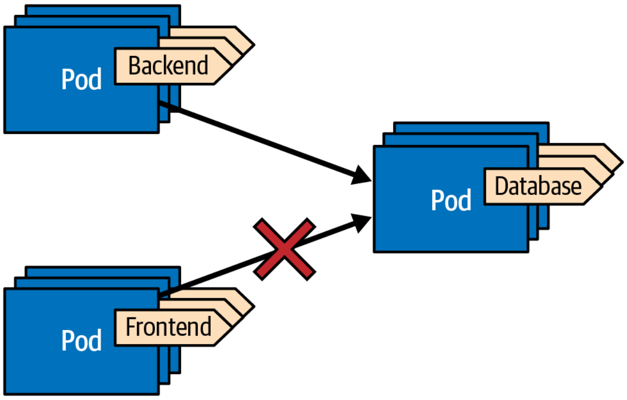

Figure 24-1. NetworkPolicy for ingress traffic

NetworkPolicy objects are namespace-scoped and match only Pods from within the NetworkPolicy's namespace. Unfortunately, there is no way to define cluster-wide defaults for all namespaces. However, some CNI plugins like Calico support customer extensions for defining cluster-wide behavior.

Network segment definition with labels

In Example 24-1, we can see how label selectors are used to dynamically define groups of Pods. This is a powerful concept in Kubernetes that allows users to easily create distinct networking segments.

Developers are typically the best ones to know which Pods belong to a specific application and how they communicate with one another. By carefully labeling the Pods, users can directly translate the dependency graphs of distributed applications into NetworkPolicies. These policies can then be used to define the network boundaries for an application, with well-defined entry and exit points.

To create network segmentation using labels, it's common to label all Pods in the application with a unique `app` label. The `app` label can be used in the selector of the NetworkPolicy to ensure that all Pods belonging to the application are covered by the policy. For example, in Example 24-1, the network segment is defined using an `app` label with the value `chili-shop`.

There are two common ways to consistently label workloads:

- Using workload-unique labels, you can directly model the dependency graph between application components such as other microservices or a database. These workloads can consist of multiple Pods, for example, when deployed in high availability. This technique is used to model the permission graph in Example 24-1, where we use a label `type` to identify the application component. Only one type of workload (e.g., Deployment or StatefulSet) is expected to carry the label `type: database`.
- In a more loosely coupled approach, you can define specific `role` or `permissions` labels that need to be attached to every workload that plays a certain role. Example 24-2 shows an example of this setup. This approach is more flexible and allows for new workloads to be added without updating the NetworkPolicy. However, the more straightforward approach of directly connecting workloads is often easier to understand by simply looking at the NetworkPolicy without having to look up all workloads that apply to a role.

Example 24-2. Role-based network segment definition

```
kind: Pod
metadata:
  label:
    app: chili-shop
    id: backend
    role-database-client: 'true'        ❶
    role-cache-client: 'true'
....
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-database-client
spec:
  podSelector:
    matchLabels:
      app: chili-shop
      id: database                      ❷
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: chili-shop
          role-database-client: 'true' ❸
```

❶ Add all roles that enable this backend Pod to access the requested services.

❷ Selector matching the database Pods—i.e., Pods with the label `id: database`.

❸ Every Pod that is a database client (`role-database-client: 'true'`) is allowed to send traffic to the backend Pod.

Deny-all as default policy

In Examples 24-1 and 24-2, we have seen how to individually configure the allowed incoming connections for a selected set of Pods. This setup works fine as long as you don't forget to configure one Pod, since the default mode, when NetworkPolicy is not configured in the namespace, does not restrict incoming and outgoing traffic (allow-all). Also, for Pods that we might create in the future, it is problematic that it might be necessary to remember to add the respective NetworkPolicy.

Therefore, it is highly recommended to start with a deny-all policy, as shown in Example 24-3.

Example 24-3. Deny-all policy for incoming traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
spec:
  podSelector: {}      ❶
  ingress: []          ❷
```

❶ An empty selector matches every Pod.

❷ An empty list of ingress rules implies that all incoming traffic gets dropped.

The list of allowed ingresses is set to an empty list (`[]`), which implies there is no ingress rule that allows incoming traffic. Note that an empty list `[]` is different from a list with a single empty element `[ {} ]`, which achieves the exact opposite since the single empty rule matches everything.

Ingress

Example 24-1 covers the primary use case of a policy that covers ingress traffic. We have already explained the `podSelector` field and given an example of an `ingress` list that matches Pods that are allowed to send traffic to the Pod under configuration. The selected Pod can receive traffic if any of the configured ingress rules in the list are matched.

Besides selecting Pods, you have additional options to configure the ingress rules. We already saw the `from` field for an ingress rule that can contain a `podSelector` for selecting all Pods that pass this rule. In addition, a `namespaceSelector` can be given to choose the namespaces in which the `podSelector` should be applied to identify the Pods that can send traffic.

Table 24-1 shows the effect of the various combinations of `podSelector` and `name spaceSelector`. Combining both fields allows for very flexible setups.

Table 24-1. Combinations of setting `podSelector` and `namespaceSelector` (`{}`: empty, `{...}`: non-empty, `---`: unset)

podSelector	namespaceSelector	Behavior
{}	{}	Every Pod in every namespace
{}	{ ... }	Every Pod in the matched namespaces
{ ... }	{}	Every matching Pod in all namespaces
{ ... }	{ ... }	Every matching Pod in the matching namespaces
---	{ ... } / {}	Every Pod in the matching namespace/all namespaces
{ ... } / {}	---	Matching Pods/every Pod in the NetworkPolicy's namespace

As an alternative for selecting Pods from the cluster, a range of IP addresses can be specified with a field `ipBlock`. We show IP ranges in Example 24-5.

Another option is to restrict the traffic to specific ports to the selected Pod. We can specify this list with a `ports` field that contains all allowed ports.

Egress

Not only can incoming traffic be regulated, but so can any request that a Pod sends in the outgoing direction. Egress rules are configured precisely with the same options as ingress rules. And as with ingress rules, starting with a very restrictive policy is recommended. However, denying all outgoing traffic is not practical. Every Pod needs interaction with Pods from the system namespace for DNS lookups. Also, if we use ingress rules to restrict incoming traffic, we would have to add mirrored egress rules for the source Pods. So let's be pragmatic and allow all egress within the cluster, forbid everything outside the cluster, and let ingress rules define the network boundaries.

Example 24-4 shows the definition of such a rule.

Example 24-4. Allow all internal egress traffic

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: egress-allow-internal-only
spec:
  policyTypes:               ❶
  - Egress
  podSelector: {}            ❷
  egress:
  - to:
    - namespaceSelector: {} ❸
```

❶ Add only `Egress` as policy type; otherwise, Kubernetes assumes that you want to specify ingress and egress.

❷ Apply NetworkPolicy to all Pods in the NetworkPolicy's namespace.

❸ Allow egress to every Pod in every other namespace.

Figure 24-2 illustrates the effect of this NetworkPolicy and how it prevents Pods from connecting to external services.

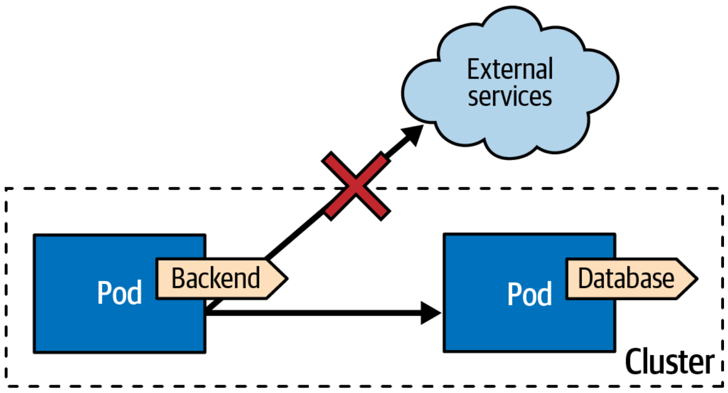

Figure 24-2. NetworkPolicy that allows only internal egress traffic

The `policyTypes` field in a NetworkPolicy determines the type of traffic the policy affects. It is a list that can contain the elements `Egress` and/or `Ingress`, and it specifies which rules are included in the policy. If the field is omitted, the default value is determined based on the presence of the `ingress` and `egress` rule sections:

- If an `ingress` section is present, the default value of `policyTypes` is `[Ingress]`.
- If an `egress` section is provided, the default value of `policyTypes` is `[Ingress, Egress]` regardless of whether ingress rules are provided.

This default behavior implies that to define an egress-only policy, you must explicitly set `policyTypes` to `[Egress]`, as in Example 24-4. Failing to do so would imply an empty `ingress` rules set, effectively forbidding all incoming traffic.

With this restriction for cluster-internal egress traffic in place, we can selectively activate access to external IP addresses for certain Pods that might require cluster-external network access. In Example 24-5, such an IP range block for allowing external egress access is defined.

Example 24-5. NetworkPolicy that allows access to all IP addresses, with some exceptions

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-external-ips
spec:
  podSelector: {}
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0   ❶
        except:
        - 192.168.0.0/16  ❷
        - 172.23.42.0/24
```

❶ Allow access to all IP addresses…

❷ …except IP addresses that belong to these subnets.

Some care must be taken if you decide to choose more strict egress rules and also want to restrict the cluster's internal egress traffic. First, it is essential to always allow access to the DNS server in the kube-system namespace. This configuration is best done by allowing access to port 53 for UDP and TCP to all ports in the system namespace.

For operators and controllers, the Kubernetes API server needs to be accessible. Unfortunately, no unique label would select the API server in the kube-system namespace, so the filtering should happen on the API server's IP address. The IP address can best be fetched from the `kubernetes` endpoints in the default namespace with `kubectl get endpoints -n default kubernetes`.

Tooling

Setting up the network topology with NetworkPolicies gets complex quickly since it involves creating many NetworkPolicy resources. It is best to start with some simple use cases that you can adapt to your specific needs. Kubernetes Network Policy Recipes is a good starting point.

Commonly, NetworkPolicies are defined along with the application's architecture. However, sometimes you must retrofit the policy schemas to an existing solution. In this case, policy advisor tools can be beneficial. They work by recording the network activity when playing through typical use cases. A comprehensive integration test suite with good test coverage pays off to catch all corner cases involving network connections. As of 2023, several tools can help you audit network traffic to create network policies.

Inspektor Gadget is a great tool suite for debugging and inspecting Kubernetes resources. It is entirely based on eBPF programs that enable kernel-level observability and provides a bridge from kernel features to high-level Kubernetes resources. One of Inspektor Gadget's features is to monitor network activity and record all UDP and TCP traffic for generating Kubernetes network policies. This technique works well but depends on the quality and depth of covered use cases.

What Is eBPF?

eBPF is a Linux technology that can run sandboxed programs in kernel space.[2] This technique extends the kernel's capabilities safely and allows for much faster innovation on top of this interface.

To some degree, eBPF is the next-generation plugin architecture for the Linux kernel. The flexibility of this API has fostered the evolution of many eBPF projects that cover a wide area of use cases, including observability and security.

Another great eBPF-based platform is Cilium, which has a dedicated audit mode that tracks all network traffic and matches it against a given network policy. By starting with a deny-all policy and audit mode enabled, Cilium will record all policy violations but will not block the traffic otherwise. The audit report helps create the proper NetworkPolicy to fit the traffic patterns exercised.

These are only two examples of the rich and growing landscape of tools for policy recommendation, simulations, and auditing.

Now that you have seen how we can model the network boundaries for our application on the TCP/UDP and IP levels, let's move up some levels in the OSI stack.

Authorization Policies

Until now, we looked at how we can control the network traffic between Pods on the TCP/IP level. However, it is sometimes beneficial to base the network restrictions on filtering on higher-level protocol parameters. This advanced network control requires knowledge of higher-level protocols like HTTP and the ability to inspect incoming and outgoing traffic. Kubernetes does not support this out of the box. Luckily, a whole family of add-ons extends Kubernetes to provide this functionality: service meshes.

2 eBPF was originally an acronym for "extended Berkeley Packet Filter" but is nowadays used as an independent term on its own.

Service Mesh

Some operational requirements like security, observability, or reliability affect all your applications. A service mesh takes care of these aspects in a generic way so that applications can focus on their business logic. Service meshes usually work by injecting sidecar containers into the workload Pods that act as the ambassador from Chapter 18 and adapter from Chapter 17 to intercept L7 incoming and outgoing traffic. Newer techniques to intercept the network traffic include node-wide proxies and a mesh data plane.

Prominent examples of service meshes are Istio, Gloo Mesh, and Linkerd. Still, many more are listed in the CNCF Cloud Native Interactive Landscape.

We chose Istio as our example service mesh, but you will find similar functionalities in other service meshes. We won't go into much detail about service meshes or Istio. Instead, we'll focus on a particular custom resource of Istio that helps us shape the networking segments on the HTTP protocol level.

Istio has a rich feature set for enabling authentication, transport security via mTLS, identity management with CERT rotations, and authorization.

As with other Kubernetes extensions, Istio leverages the Kubernetes API machinery by introducing its own CustomResourceDefinitions (CRDs) that are explained in detail in Chapter 28, "Operator". Authorization in Istio is configured with the AuthorizationPolicy resource. While AuthorizationPolicy is only one component in Istio's security model, it can be used alone and allows for partitioning the network space based on HTTP.

The schema of AuthorizationPolicy is very similar to NetworkPolicy but is more flexible and includes HTTP-specific filters. NetworkPolicy and AuthorizationPolicy should be used together. This can lead to a tricky debugging setup when two configurations must be checked and verified in parallel. Traffic will pass through to a Pod only if the two user-defined firewalls spanned by NetworkPolicy and Authorization-Policy definition will allow it.

An AuthorizationPolicy is a namespaced resource and contains a set of rules that control whether or not traffic is allowed or denied to a particular set of Pods in a Kubernetes cluster. The policy consists of the following three parts:

Selector
: Specifies which Pods the policy applies to. If no selector is specified, the policy applies to all Pods in the same namespace as the policy. If the policy is created in Istio's root namespace (`istio-system`), it applies to all matching Pods in all namespaces.

Action
: Defines what should be done with the traffic that matches the rules. The possible actions are `ALLOW`, `DENY`, `AUDIT` (for logging only), and `CUSTOM` (for user-defined actions).

List of rules
: These are evaluated for incoming traffic. All of the rules must be satisfied for the action to be taken. Each rule has three components: a `from` field that specifies the source of the request, a `to` field that specifies the HTTP operation that the request must match, and an optional `when` field for additional conditions (e.g., the identity associated with the request must match a particular value).

Example 24-6 shows a typical example that allows the monitoring operator access to application endpoints for collecting metric data.

Example 24-6. Authorization for a Prometheus setup

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: prometheus-scraper
  namespace: istio-system      ❶
spec:
  selector:                    ❷
    matchLabels:
      has-metrics: "true"
  action: ALLOW                ❸
  rules:
  - from:                      ❹
    - source:
        namespaces: ["prometheus"]
    to:
    - operation:               ❺
        methods: [ "GET" ]
        paths: ["/metrics/*"]
```

❶ When created in the namespace `istio-system`, the policy applies to all matching Pods in all namespaces.

❷ The policy is applied to all Pods with a `has-metrics` label set to `true`.

❸ The action should allow the request to pass if the rules match.

❹ Every request coming from a Pod from the `prometheus` namespace…

❺ …can perform a GET request on the `/metrics` endpoint.

In Example 24-6, every Pod that carries the label `has-metrics: "true"` allows traffic to its `/metrics` endpoint from each Pod of the `prometheus` namespace.

This policy has an effect only if, by default, all requests are denied. As for NetworkPolicy, the best starting point is to define a deny-all policy, as shown in Example 24-7, and then selectively build up the network topology by allowing dedicated routes.

Example 24-7. Deny-all policy as the default

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: deny-all
 namespace: istio-system ❶
spec: {}                ❷
```

❶ The policy applies to all namespaces since it is created in `istio-system`.

❷ Policies with an empty `spec` section deny all requests.

With the help of the proper labeling schema, AuthorizationPolicy helps define the application's network segments that are independent and isolated from one another. All that we said in "Network segment definition with labels" on page 225 also applies here.

However, AuthorizationPolicy can also be used for application-level authorization when we add an identity check to the rules. One crucial difference to the authorization that we describe in Chapter 26, "Access Control", is that AuthorizationPolicy is about *application authorization*, while the Kubernetes RBAC model is about securing the access to the Kubernetes API server. Access control is primarily helpful for operators monitoring their custom resources.

Discussion

In the early days of computing, network topologies were defined by physical wiring and devices like switches. This approach is secure but not very flexible. With the advent of virtualization, these devices were replaced by software-backed constructs to provide network security. *Software-defined networking* (SDN) is a type of computer networking architecture that allows network administrators to manage network services through abstraction of lower-level functionality. This abstraction is typically achieved by separating the control plane, which makes decisions about how data should be transmitted, from the data plane, which actually sends the data. Even with the use of SDN, administrators are still needed to set up and rearrange networking boundaries to effectively manage the network.

Kubernetes has the ability to overlay its flat cluster-internal network with network segments defined by users through the Kubernetes API. This is the next step in the evolution of network user interfaces. It shifts the responsibility to developers who understand the security requirements of their applications. This shift-left approach is beneficial in a world of microservices with many distributed dependencies and a complex network of connections. NetworkPolicies for L3/L4 network segmentation and AuthorizationPolicies for more granular control of network boundaries are essential for implementing this *Network Segmentation* pattern.

With the advent of eBPF-based platforms on top of Kubernetes, there is additional support for finding suitable network models. Cilium is an example of a platform that combines L3/L4 and L7 firewalling into a single API, making it easier to implement the pattern described in this chapter in future versions of Kubernetes.

More Information

- Network Segmentation Example
- Network Policies
- The Kubernetes Network Model
- Kubernetes Network Policy Recipes
- Using Network Policies
- Why You Should Test Your Kubernetes Network Policies
- Using the eBPF Superpowers to Generate Kubernetes Security Policies
- Using Advise Network-Policy with Inspektor Gadget
- You and Your Security Profiles; Generating Security Policies with the Help of eBPF
- kube-iptables-tailer
- Creating Policies from Verdicts
- Istio: Authorization Policy
- SIG Multitenancy Working Group

CHAPTER 25

Secure Configuration

No real-world application lives in isolation. Instead, each connects to external systems in one way or the other. Such external systems could include value-add services provided by the big cloud providers, other microservices that your service connects to, or a database. Regardless of which remote services your application connects to, you will likely need to go through authentication, which involves sending over credentials such as username and password or some other security token. This confidential information must be stored somewhere close to your application securely and safely. This chapter's *Secure Configuration* pattern is about the best ways to keep your credentials as secure as possible when running on Kubernetes.

Problem

As you learned in Chapter 20, "Configuration Resource", despite what its name implies, Secret resources are not encrypted but are only Base64 encoded. Nevertheless, Kubernetes does its best to restrict access to a Secret's content with the techniques described in "How Secure Are Secrets?" on page 190.

However, as soon as Secret resources are stored outside the cluster, they are naked and vulnerable. With the advent of GitOps as a prevalent paradigm for deploying and maintaining server-side applications, this security challenge is even more pressing. Should Secrets be stored on remote Git repositories? If so, then they must not be stored unencrypted. However, when those are committed encrypted in a source code management system like Git, where do they get decrypted on their way into a Kubernetes cluster?

Even when credentials are stored encrypted within the cluster, it is not guaranteed that nobody else can access that confidential information. While you can granularly regulate access to Kubernetes resources with RBAC rules,[1] at least one person has access to all data stored in the cluster: your cluster administrator. You might or might not be able to trust the cluster administrator. It all depends on the context in which your application operates. Are you running a Kubernetes cluster in the cloud operated by somebody else? Or is your application deployed on a big company-wide Kubernetes platform, and you need to know who is running this cluster? Different solutions are required depending on these trust boundaries and confidentiality requirements.

Secrets are the Kubernetes answer for confidential configuration in-cluster storage. We talked in depth about Secrets in Chapter 20, "Configuration Resource", so let's now have a look at how we can improve various security aspects of Secrets with additional techniques.

Solution

The most straightforward solution for secure configuration is decoding encrypted information within the application itself. This approach always works, and not just when running on Kubernetes. But it takes considerable work to implement this within your code, and it couples your business logic with this aspect of securing your configuration. There are better, more transparent ways to do this on Kubernetes.

The support for secure configuration on Kubernetes falls roughly into two categories:

Out-of-cluster encryption
: This stores encrypted configuration information outside of Kubernetes, which nonauthorized persons can also read. The transformation into Kubernetes Secrets happens just before entering the cluster (e.g., when applying a resource via the API server) or inside the cluster by a permanently running operator process.

Centralized secret management
: This uses specialized services that are either already offered by cloud providers (e.g., AWS Secrets Manager or Azure Key Vault) or are part of an in-house vault service (e.g., HashiCorp Vault) for storing confidential configuration data.

While out-of-cluster encryption techniques always eventually create a Secret within the cluster that your application can use, the support for external secret management systems (SMSs) provided by Kubernetes add-ons uses various other techniques to bring the confidential information to the deployed workloads.

1 RBAC rules are explained in detail in Chapter 26, "Access Control".

Out-of-Cluster Encryption

The gist of the out-of-cluster technique is simple: pick up secret and confidential data from outside the cluster and transform it into a Kubernetes Secret. A lot of projects have been grown that implement this technique. This chapter looks at the three most prominent ones (as of 2023): Sealed Secrets, External Secrets, and sops.

Sealed Secrets

One of the oldest Kubernetes add-ons for helping with encrypted secrets is *Sealed Secrets*, introduced by Bitnami in 2017. The idea is to store the encrypted data for a Secret in a CustomResourceDefinition (CRD) SealedSecret. In the background, an operator monitors such resources and creates one Kubernetes Secret for each SealedSecret with the decrypted content. To learn more about CRDs and operators in general, check out Chapter 28, "Operator", which explains this pattern in detail. While the decryption happens within the cluster, the *encryption* happens outside by a CLI tool called `kubeseal`, which takes a Secret and translates it to a SealedSecret that can be stored safely in a source code management system like Git.

Figure 25-1 shows the setup for Sealed Secrets.

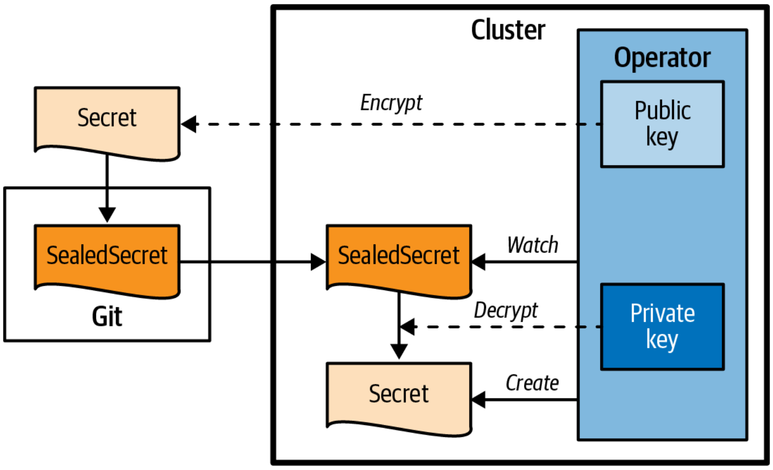

Figure 25-1. Sealed Secrets

Secrets are encrypted with AES-256-GCM symmetrically as a session key, and the session key is encrypted asymmetrically with RSA-OAEP, the same setup TLS uses.

The secret private key is stored within the cluster and is automatically created by the SealedSecret Operator. It is up to the administrator to back up this key and rotate it if needed. The public key used by `kubeseal` can be fetched directly from the cluster or accessed directly from a file. You also can safely store the public key in Git along with your SealedSecret.

SealedSecrets support three scopes that you can select when creating a SealedSecret from a Secret:

Strict
: This freezes the namespace and name of the SealedSecret. This mode means you can create the SealedSecret only in the same namespace and with the same name as the original Secret in any target cluster. This mode is the default behavior.

Namespace-wide
: This allows you to apply the SealedSecret to a different name than the initial Secret but still pins it to the same namespace.

Cluster-wide
: This allows you to apply the SealedSecret to different namespaces, as it was initially created to do, and the name can be changed too.

These scopes can be selected when creating the SealedSecret with `kubeseal`. Still, you can also add the nonstrict scopes with the annotations listed in Table 25-1 on the original Secret before encryption or on the SealedSecret directly.

Table 25-1. Annotation

Annotation	Value	Description
sealedsecrets.bitnami.com/namespace-wide	`"true"`	Enable namespace-wide scope when set to `true`—i.e., different name but same namespace
sealedsecrets.bitnami.com/cluster-wide	`"true"`	Enable cluster-wide scope when set to `true`—i.e., name and namespace can be changed on the SealedSecret after encryption

Example 25-1 shows a SealedSecret created by `kubeseal` that can be directly stored in Git.

Example 25-1. SealedSecret created with `kubeseal`

```
# Command to create this sealed secret:
# kubeseal --scope cluster-wide -f mysecret.yaml      ❶
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  annotations:
    sealedsecrets.bitnami.com/cluster-wide: "true"    ❷
  name: DB-credentials
spec:
  encryptedData:
    password: AgCrKIIF2gA7tSR/gqw+FH6cEV..wPWWkHJbo=  ❸
    user: AgAmvgFQBBNPlt9Gmx..0DNHJpDIMUGgwaQroXT+o=
```

❶ Command to create a SealedSecret from the secret stored in *mysecret.yaml*.

❷ Annotation that indicates that this SealedSecret can have any name and be applied to any namespace.

❸ The secret values are encrypted individually (and shortened here for the sake of demonstration).

A Sealed Secret is a tool that allows you to store encrypted secrets in a publicly available location, such as a GitHub repository. It is important to properly back up the secret key, as without it, it will not be possible to decrypt the secrets if the operator is uninstalled. One potential drawback of Sealed Secrets is that they require a server-side operator to be continuously running in the cluster in order to perform the decryption.

External Secrets

The External Secrets Operator is a Kubernetes operator that integrates a growing list of external SMSs. The main difference between External Secrets and Sealed Secrets is that you do not manage the encrypted data storage yourself but rely on an external SMS to do the hard work, including encryption, decryption, and secure persistence. That way, you benefit from all the features of your cloud's SMS, like key rotation and a dedicated user interface. SMS also provides an excellent way of separating concerns so that different roles can manage the application deployments and the secrets separately.

Figure 25-2 shows the External Secrets architecture.

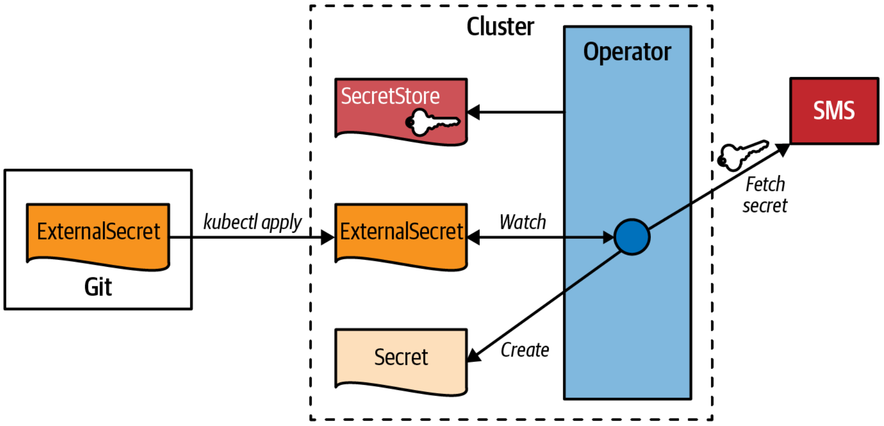

Figure 25-2. External Secrets

A central operator reconciles two custom resources:

- SecretStore is the resource that holds the type and configuration of the external SMS to access. Example 25-2 gives an example of a store that connects to AWS Secret Manager.
- ExternalSecret references a SecretStore, and the operator will create a corresponding Kubernetes Secret filled with the data fetched from the external SMS. For example, Example 25-3 references a secret in the AWS Secret Manager and exposes the value within the specified target Secret.

Example 25-2. SecretStore for connecting to AWS Secret Manager

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: secret-store-aws
spec:
  provider:
    aws:                              ❶
      service: SecretsManager
      region: us-east-1
      auth:
        secretRef:
          accessKeyIDSecretRef:       ❷
            name: awssm-secret
            key: access-key
          secretAccessKeySecretRef:
            name: awssm-secret
            key: secret-access-key
```

❶ Provider `aws` configures the usage of the AWS Secret Manager.

❷ Reference to a Secret that holds the access keys for talking with the AWS Secret Manager. A Secret with the name `awssm-secret` contains the keys `access-key` and `secret-access-key` used to authenticate against the AWS Secret Manager.

Example 25-3. ExternalSecret that will be transformed into a Secret

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: db-credentials
spec:
  refreshInterval: 1h
  secretStoreRef:                      ❶
    name: secret-store-aws
    kind: SecretStore
  target:
    name: db-credentials-secrets ❷
    creationPolicy: Owner
  data:
    - key: cluster/db-username   ❸
      name: username
    - key: cluster/db-password
      name: password
```

❶ Reference to the SecretStore object that holds the connection parameters for AWS Secret Manager.

❷ Name of the Secret to create.

❸ The username that will be looked up under `cluster/DB-username` in AWS Secret Manager and put under the key `username` in the resulting Secret.

You have a lot of flexibility in defining the mapping of the external secret data to the content of the mirrored Secret—for example, using a template to create a configuration with a particular structure. See the External Secrets documentation for more information. One significant advantage of this solution over a client-side solution is that only the server-side operator knows the credentials to authenticate against the external SMS.

The External Secrets Operator project merges several other Secret-syncing projects. In 2023, it is already the dominant solution for this specific use case of mapping and syncing an externally defined secret to a Kubernetes Secret. However, it has the same cost as a server-side component that runs all the time.

Sops

Do we need a server-side component to work with Secrets in a GitOps world where all resources are stored in a Git repository? Luckily, solutions exist that work entirely outside of a Kubernetes cluster. A pure client-side solution is sops ("Secret OPerationS") by Mozilla. Sops is not specific to Kubernetes but allows you to encrypt and decrypt any YAML or JSON file to safely store those in a source code repository. It does this by encrypting all values of such a document but leaving the keys untouched.

We can use various methods for encryption with sops:

- Asymmetric local encryption via age with the keys stored locally.
- Storing the secret encryption key in a centralized key management system (KMS). Supported platforms are AWS KMS, Google KMS, and Azure Key Vault as external cloud providers and HashiCorp Vault as an SMS you can host on your own. The identity management of those platforms allows for fine-granular access control to the encryption key.

SMS Versus KMS

In the previous sections, we talked about *secret management systems* (SMSs), cloud services that do secret management for you. They provide an API for storing and accessing the secrets with granular and configurable access control. Those secrets are encrypted transparently for the user, and you don't have to worry about this. *Key management systems* (KMSs) are cloud services you can access with an API. However, in contrast to SMSs, KMSs are not databases for secure data but care about the discovery and storage of encryption keys, which you can use to encrypt data outside of a KMS. The GnuPG keyservers are good examples of a KMS. Each leading cloud provider offers both SMSs and KMSs. If you are sold to one of the big clouds, you also get good integration with its identity management for defining and assigning the access rules to SMS- and KMS-managed data.

Sops is a CLI tool you can run locally on your machine or within a cluster (e.g., as part of a CI pipeline). Especially for the latter use case and if you are running in one of the big clouds, leveraging one of their KMSs provides a smooth integration.

Figure 25-3 illustrates how sops handles encryption and decryption on the client side.

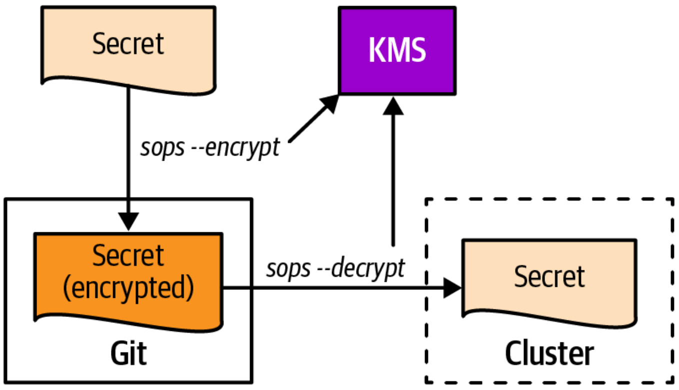

Figure 25-3. Sops for decrypting and encrypting resource files

Example 25-4 shows how to use sops to create an encrypted version of a ConfigMap.[2] This example uses `age` and a freshly generated keypair for the encryption, which should be stored safely.

2 In the real world, you should use a Secret for this kind of confidential information, but here we use a ConfigMap to demonstrate that you can use *any* resource file with sops.

Example 25-4. Sops for creating encrypted secrets

```
$ age-keygen -o keys.txt        ❶
Public key: age1j49ugcg2rzyye07ksyvj5688m6hmv

$ cat configmap.yaml            ❷
apiVersion: v1
kind: ConfigMap
metadata:
  name_unencrypted: db-auth     ❸
data:
  # User and Password
  USER: "batman"
  PASSWORD: "r0b1n"

$ sops --encrypt \              ❹
    --age age1j49ugcg2rzyye07ksyvj5688m6hmv \
    configmap.yaml > configmap_encrypted.yaml

$ cat configmap_encrypted.yaml
apiVersion: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str] ❺
kind: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
metadata:
    name_unencrypted: db-auth   ❻
data:
    #ENC[AES256_GCM,data:...,iv:...,tag:...,type:comment]
    USER: ENC[AES256_GCM,data:...,iv:...,tag:...=,type:str]
    PASSWORD: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
sops:                           ❼
    age:
        - recipient: age1j49ugcg2rzyye07ksyvj5688m6hmv
          enc: |                ❽
            -----BEGIN AGE ENCRYPTED FILE-----
            YWdlLWVuY3J5cHRpb24ub3JnL3YxCi0+IFgyNTUxOSBqems3QkU4aXRyQWxaNER1
            TTdqcUZTeXFXNWhSY0E1T05XMUhVUzFjR1FnCmdMZmhlSlZCRHlqTzlNM0E1Z280
            Y0tqQ2VKYXdxdDZIZHpDbmxTYzhQSTgKLS0tIHlBYmloL2laZlA4Q05DTmRwQ0ls
            bURoU2xITHNzSXp5US9mUUV0Z0RackkKFtH+uNNe3A13pzSvHjT6n3q9av0pN7Nb
            i3AULtKvAGs6oAnH8qYbnwoj3qt/LFfnbqfeFk1zC2uqNONWkKxa2Q==
            -----END AGE ENCRYPTED FILE-----
    last modified: "2022-09-20T09:56:49Z"
    mac: ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
    unencrypted_suffix: _unencrypted
```

❶ Create a secret key with `age` and store it in *keys.txt*.

❷ The ConfigMap to encrypt.

❸ The `name` field is changed to `name_unencrypted` to prevent it from getting encrypted.