O'REILLY®

Kubernetes Security

Operating Kubernetes Clusters and Applications Safely

Liz Rice & Michael Hausenblas

aqua

Full Lifecycle Security

For Containers and Cloud Native Applications

Building and managing secure Kubernetes clusters is a complex task. Aqua Security provides a complete solution that leverages native Kubernetes capabilities, makes it easy to establish policy-driven monitoring and enforcement, and further secures Kubernetes deployments with runtime protection and compliance controls at the cluster, namespace, node, pod and container levels.

- **Enhances** Native Kubernetes Security Controls
- **Protects** Applications in Runtime
- **Secures** The Build Pipeline
- **Provides Visibility** For Compliance

Aqua Security is the company behind open-source tools that enable you to improve the security of your Kubernetes cluster:

kube-bench

Check your cluster against 100+ tests of the CIS Kubernetes Benchmark so you can harden it according to best practices.
github.com/aquasecurity/kube-bench

kube-hunter

Penetration testing tool that “attacks” your cluster and nodes, looking for configuration issues.
github.com/aquasecurity/kube-hunter

Learn more

www.aquasec.com

Kubernetes Security

Operating Kubernetes Clusters and Applications Safely

Liz Rice and Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes Security
by Liz Rice and Michael Hausenblas

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Development Editor: Virginia Wilson
Production Editor: Justin Billing
Copyeditor: Sharon Wilkey
Proofreader: Chris Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2018: First Edition

Revision History for the First Edition
2018-09-28: First Release

This work is part of a collaboration between O'Reilly and Aqua Security Software. See our *statement of editorial independence*.

978-1-492-04600-4

[LSI]

Table of Contents

Introduction

This book will teach you practices to make your Kubernetes deployments more secure. It will introduce you to security features in Kubernetes and tell you about other things you should be aware of in the context of containerized applications running on Kubernetes; for example, container image best practices from a security point of view.

We describe practical techniques and provide an accompanying website with references and recipes, so if you want to follow along, check it out!

Why We Wrote This Book

Kubernetes has rapidly become a popular choice for deploying code "in the cloud" and is now used by enterprises of all sizes to deploy mission-critical applications. However, information about securing Kubernetes is distributed across the internet and in the code itself. We want to make it easier for anyone who is using Kubernetes to think about and address the security of their deployments by gathering information into one resource.

Who Is This Book For?

This book is written for developers, operation folks, and security professionals who are using Kubernetes. Please note that we assume familiarity with basic Kubernetes concepts. If you don't have that familiarity yet, a great book to get started is *Kubernetes: Up and Running* by Kelsey Hightower et al. (O'Reilly). In addition, *Kubernetes Cookbook* by Michael Hausenblas (one of the authors of this book)

and Sébastien Goasguen (O'Reilly) provides recipes for common tasks.

In this book, we tackle the technical aspects of Kubernetes security, but sidestep cultural and organizational issues, such as who should be responsible for implementing and ensuring the advice we offer. We do suggest that this is something you pay attention to, as no amount of technology will fix a broken culture.

Which Version of Kubernetes?

Kubernetes is an evolving project with improvements being made all the time. At the time of writing, the latest release of Kubernetes is v1.11. Several security-related features have been added and stabilized over the last few releases, with the general availability of role-based access control (RBAC) in v1.8 particularly worthy of note. With that in mind, we strongly recommend upgrading to v1.8 or newer if you haven't already.

We expect the advice in this book to be generally applicable to whatever version you are running from v1.8 onward. We point out when a particular version newer than 1.8 is required in order for a recommendation to work.

Via the accompanying website *kubernetes-security.info*, we plan to keep you up-to-date as new tooling and best practices become available and as Kubernetes evolves, so keep an eye on this site!

A Note on Federation

Federation is the concept of operating multiple Kubernetes clusters together, with the ability to synchronize and discover resources across them. At the time of writing, the Kubernetes Federation API has no clear path to general availability, so we have left the security of federated clusters out of the scope of this book.

Acknowledgments

A big thank you to the O'Reilly team, especially Virginia Wilson, for shepherding us through the process of writing this book.

We're super grateful to our technical reviewers Alban Crequy, Amir Jerbi, Andrew Martin, Ian Lewis, Jordan Liggitt, Michael Kehoe,

Seth Vargo, and Tim Mackey, who provided valuable, actionable feedback and advice.

CHAPTER 1
Approaching Kubernetes Security

Security is a funny, elusive thing. You will rarely hear a security professional describe something as "secure." You'll hear that something may be more or less secure than an alternative, but security is dependent on context.

In this book, we will show you ways to make your Kubernetes cluster more secure. Whether you need to apply a particular measure to make your deployment secure enough for your particular use case is something for you to assess, depending on the risks you are running. We hope that if your Kubernetes cluster holds our bank account details or our medical records, you will take all the precautions described herein, at the very least!

We will cover ways that you can configure your Kubernetes cluster to improve security. Your cluster runs containerized workloads, and we will discuss ways to make it more likely that you are running the workloads you expect (and nothing more). We present precautions you can take to limit the likelihood of a breach by an attacker, and to limit the likelihood of that breach resulting in data loss.

In addition, you can use plenty of non-Kubernetes-specific security tools and approaches that are outside the scope of this book. You can layer traditional network firewalls and intrusion-detection systems, in addition to everything that is described here. You may have an air-gapped deployment. And wherever humans interact with your system, they may constitute a risk to security, either maliciously or just due to human error. We don't pretend to address those issues in this book. As shown in Figure 1-1, there are various

ways that an attacker could attempt to compromise your Kubernetes cluster and the applications running on it.

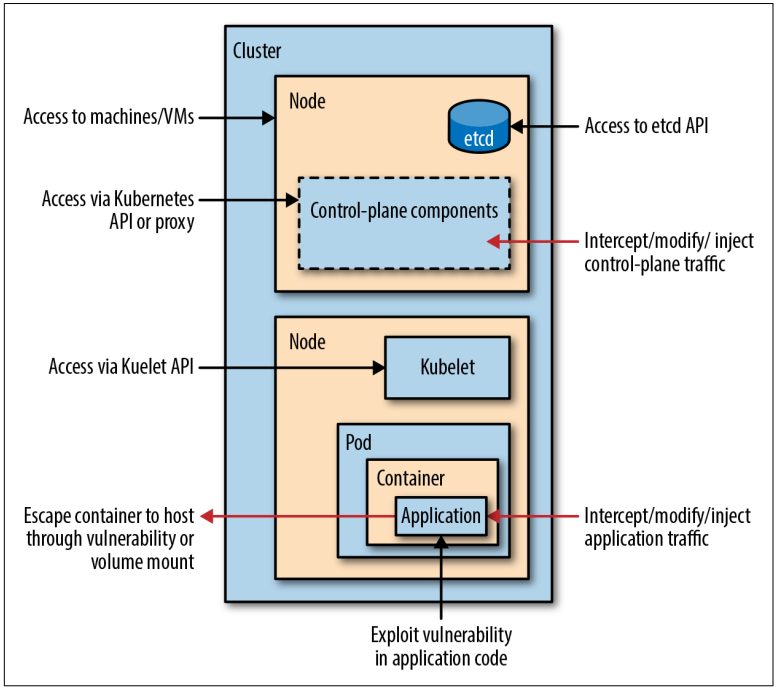

Figure 1-1. Kubernetes attack vectors

In this book, we explain controls, configurations, and best practices that you can apply to mitigate all these potential modes of attack. We present several aspects of Kubernetes security:

Configuring Kubernetes for security
: Chapter 2 considers the configuration of Kubernetes components, and Chapter 3 and Chapter 4 discuss how to limit access to Kubernetes resources so that they are accessible to only the people and applications that need them.

Preventing your application workloads from being exploited
: Chapter 5 explains approaches that ensure you are not running code with known vulnerabilities on your Kubernetes cluster. Chapter 6 presents additional ways you can limit the behavior of containers at runtime, making it harder for an attacker to abuse those containers.

Protecting credentials
: Chapter 7 discusses how to store credentials and pass them safely into applications.

We finish in Chapter 8 with some advanced ideas for securing your Kubernetes cluster.

But before we get started on Kubernetes-specific information, let's introduce a few important general security concepts that we'll use in the rest of the book.

Security Principles

In this section, we'll discuss three important principles that can be used to increase security: defense in depth, least privilege, and limiting the attack surface.

Defense in Depth

Picture a medieval castle under siege. It has strong, high walls to keep undesirables out. The wall is surrounded by a moat, with access via a drawbridge that is lowered only occasionally to let people in and out. The castle has thick doors, and bars across any windows. Archers patrol the castle walls, ready to fire at any attacker.

The castle has several layers of defense. Attackers who can swim might be prepared to cross the moat, but then they have the walls to scale, and the likelihood of being picked off by an archer. It might be possible to compromise any given layer in the defensive structure, but by having several layers, it's hard for an attacker to successfully enter the castle.

In the same way, it's preferable to have several layers of defense against attacks on your Kubernetes cluster. If you're relying on a single defensive measure, attackers might find their way around it.

Least Privilege

The *principle of least privilege* tells us to restrict access so that different components can access only the information and resources they need to operate correctly. In the event of a component being compromised, an attacker can reach only the subset of information and resources available to that component. This limits the "blast radius" of the attack.

Consider an example of an e-commerce store. Let's assume it is built using a "microservice" architecture with functionality broken into small, discrete components. Even if product and user information is held in the same database, different microservices might each be granted access to only the appropriate parts of that database. A product-search microservice needs read-only access to the product tables, but nothing more. If this microservice somehow gets compromised, or simply has a bug, the broken service can't overwrite product information (because it has only read access) or extract user information (because it has no access to that data at all). Applying the principle of least privilege means that we make it more difficult for an attacker to cause damage.

NOTE

Microservices, as defined by Martin Fowler, are a particular design of software apps, essentially a collection of independently deployable services.

The same principle can apply to humans too. In some organizations, sharing production credentials with all staff may make sense. In others, it's critical that only a small set of people have access, especially if that access is to sensitive information such as medical or financial records.

Limiting the Attack Surface

The *attack surface* is the set of all possible ways a system can be attacked. The more complex the system, the bigger the attack surface, and therefore the more likely it is that an attacker will find a way in.

Consider our castle metaphor again: the longer the length of the castle walls, the more archers we would need to patrol them effectively. A circular castle will be most efficient from this point of view; a complicated shape with lots of nooks and crannies would need more archers for the same interior volume.

In software systems, the fundamental way to reduce the attack surface is to minimize the amount of code. The more code that's present in the system, the more likely it is that it has vulnerabilities. The greater the complexity, the more likely that latent vulnerabilities exist, even in well-tested code.

Now that we have established some security concepts, let's see how we apply them to the configuration of a Kubernetes cluster.

CHAPTER 2

Securing the Cluster

Perhaps it goes without saying, but you don't want to allow unauthorized folks (or machines!) to have the ability to control what's happening in your Kubernetes cluster. Anyone who can run software on your deployment can, at the very least, use your compute resources (as in the well-publicized case of "cryptojacking" at Tesla); they could choose to play havoc with your existing services and even get access to your data.

Unfortunately, in the early days of Kubernetes, the default settings left the control plane insecure in important ways. The situation is further complicated by the fact that different installation tools may configure your deployment in different ways. The default settings have been improving from a security point of view, but it is well worth checking the configuration you're using.

In this chapter, we cover the configuration settings that are important to get right for the Kubernetes control-plane components, concluding with some advice on tools that can be used to verify the deployed configuration.

API Server

As its name suggests, the main function of the Kubernetes API server is to offer a REST API for controlling Kubernetes. This is powerful—a user who has full permissions on this API has the equivalent of root access on every machine in the cluster.

The command-line tool `kubectl` is a client for this API, making requests of the API server to manage resources and workloads. Anyone who has write access to this Kubernetes API can control the cluster in the same way.

By default, the API server will listen on what is rightfully called the *insecure port*, port 8080. Any requests to this port *bypass authentication and authorization checks*. If you leave this port open, *anyone who gains access to the host your master is running on has full control over your entire cluster*.

Close the insecure port by setting the API server's `--insecure-port` flag to 0, and ensuring that the `--insecure-bind-address` is not set.

NOTE

The `--insecure-port` flag was deprecated in Kubernetes v1.10 and is a target for removal altogether in the future.

You can check whether the insecure port is open on the default port with a simple `curl` command like the following, where `<IP address>` is the host where the API server is running (or `localhost` if you can SSH directly to that machine):

```
$ curl <IP address>:8080
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
...
```

If the response lists API endpoints, as in the preceding example, then the insecure port is open. However, if you see an error message of *Connection refused*, it's good news, as the port is not open.

With the insecure port closed, the API can be accessed only over a secure, encrypted TLS connection via the *secure port*. You may want to further restrict API access to known, authenticated users by setting `--anonymous-auth=false` for the API server. However, it is not reckless to allow anonymous access to the API so long as you are using RBAC, which we strongly recommend. We discuss this in more detail in "Access Control with RBAC" on page 27.

The default RBAC settings permit only limited API access for anonymous users. This allows for health and discovery checks to be made, for example, by components like load balancers.

One thing to be aware of, however, is that enabling anonymous access to discovery endpoints could also increase the likelihood of leaking information about the software that's running on the system to an attacker. This read-only information is unlikely to compromise anything important by itself, but it can signpost an attacker toward other weaknesses. For example, if attackers can use health-check information to learn that a particular database is in use, they could use that information to choose which types of attack are more likely to work against that database.

For this reason, you may want to protect network access to the API server by using other mechanisms—perhaps a traditional firewall or a virtual private network (VPN).

Although we cover RBAC in more detail later, for now let's cover how to enable it in the control plane:

- Set `--authorization-mode` on the API server to enable the `RBAC` authorization module.
- Include the `Node` authorizer in the `--authorization-mode` list, which (in conjunction with the `NodeRestriction` admission controller described in the next section) enables RBAC for kubelets.

Kubelet

The kubelet is the agent on each node that is responsible for interacting with the container runtime to launch pods, and report node and pod status and metrics. Each kubelet in the cluster also operates an API, through which other components ask it to do things like starting and stopping pods. If unauthorized users can access this API (on any node) to execute code on the cluster, it's possible to gain control of the entire cluster.

Fortunately, layers of defense are now available in Kubernetes that make it easy to prevent this kind of attack:

- You can limit the API access to authenticated requests; that is, anonymous requests are ignored.
- You can leverage access control to stop unauthorized actions from being performed (see "Access Control with RBAC" on page 27).

More specifically, here are some configuration options to lock down the kubelets and hence help minimize the attack surface:

- *Disable anonymous access* with `--anonymous-auth=false`, so that unauthenticated requests will receive *Unauthorized Access* error responses. This requires the API server to identify itself to the kubelet, which you can set up with the `--kubelet-client-certificate` and `--kubelet-client-key` flags.
- *Ensure that requests are authorized* by setting `--authorization-mode` to something other than `AlwaysAllow`. The `kubeadm` installation tool defaults this setting to `Webhook` so that the kubelet calls `SubjectAccessReview` on the API server for authorization.
- *Limit the permissions of kubelets* by including `NodeRestriction` in the `--admission-control` settings on the API server. This restricts a kubelet so that it can modify only pods that are bound to it and its own `Node` object.
- Set `--read-only-port=0` to *turn off the read-only port*. This port allows an anonymous user to access information about running workloads. While access to this port doesn't allow a hacker to control the cluster, exposing information about what's running could make it easier to attack.
- Older Kubernetes deployments used cAdvisor to provide metrics, but this has largely been superseded by stats on the Kubelet API. Unless you know you are using the kubelet cAdvisor port, you should turn it off to stop it from exposing information about your running workloads, by setting `--cadvisor-port=0`. This is the default setting in Kubernetes v1.11, and it is expected that the flag will be removed altogether in the future. If you want to run cAdvisor on your cluster, it is now recommended that you do this with a `DaemonSet`.

You can check what access is available on a kubelet by attempting an API request to the node as follows:

```
$ curl -sk https://<IP address>:10250/pods/
```

- If `--anonymous-auth` is `false`, you will see a `401 Unauthorized` response.
- If `--anonymous-auth` is `true` and `--authorization-mode` is `Webhook`, you'll see a `403 Forbidden` response with the message `Forbidden (user=system:anonymous, verb=get, resource=nodes, subresource=proxy)`.
- If `--anonymous-auth` is `true` and `--authorization-mode` is `AlwaysAllow`, you'll see a list of pods.

Kubelet Certificate Rotation

Each kubelet needs a client certificate so that it can communicate with the API server. From 1.8 onward, the kubelet supports rotating these certificates automatically with the `--rotate-certificates` flag, so that a new certificate will be requested and issued automatically as the expiry deadline approaches. Unless you have a good reason not to do so, we recommend enabling this feature.

Running etcd Safely

Kubernetes stores configuration and state information in a distributed key-value store called etcd. Anyone who can write to etcd can effectively control your Kubernetes cluster. Even just reading the contents of etcd could easily provide helpful hints to a would-be attacker. Therefore, you need to ensure that only authenticated access is permitted:

- Set `--cert-file` and `--key-file` to enable HTTPS connections to etcd.
- Set `--client-cert-auth=true` to ensure that access to etcd requires authentication. Set `--trusted-ca-file` to specify the certificate authority that has signed the client certificates.
- Set `--auto-tls=false` to disallow the generation and use of self-signed certificates.
- Require etcd nodes to communicate with each other securely by using `--peer-client-cert-auth=true`. Also set `--peer-auto-tls=false` and specify `--peer-cert-file`, `--peer-key-file`

and `--peer-trusted-ca-file`. You will need corresponding configuration on the Kubernetes API server so that it can communicate with etcd.

- Set `--etcd-cafile` on the API server to the certificate authority that signed etcd's certificate.
- Specify `--etcd-certfile` and `--etcd-keyfile` so that the API server can identify itself to etcd.

See the etcd documentation for more information.

You should take additional measures to encrypt etcd's data stored on disk. This is especially important if you are storing Kubernetes secrets in etcd rather than an external secrets store. See Chapter 7 for more details on this topic.

Because only the Kubernetes control-plane components have any business communicating with etcd, you can additionally use network firewalling to prevent traffic from other sources from reaching the etcd cluster.

Kubernetes Dashboard

The *Dashboard* has historically been used by attackers to gain control of Kubernetes clusters. It's a powerful tool, and in older versions of Kubernetes, the default settings made it easy to abuse; for example, prior to 1.7, the Dashboard had full admin privileges by default.

You might want to take several steps to ensure that your Kubernetes Dashboard is not an easy entry point for attackers, including but not limited to the following:

Allow only authenticated access
: Only known users should be able to access the Dashboard.

Use RBAC
: Limit the privileges that users have so they can administer only the resources they need to.

Make sure the Dashboard service account has limited access
: After reaching the Dashboard login screen, users have the option to Skip. Taking this path means that rather than authenticating as their own user identity (as discussed in "Identity" on page 15), they access the Dashboard with the service account

associated with the Dashboard application itself. This service account should have minimal permissions.

Don't expose your Dashboard to the public internet
: Unless you really know what you're doing.

We recommend checking the latest Kubernetes Dashboard installation recommendations.

You can use `kubectl proxy` to access the Dashboard securely from a local machine. If you want to give users access directly via their browser, the Heptio blog has a good discussion of the options.

Applying different security measures to the Dashboard gives you defense in depth to mitigate potential attacks. For example, suppose you use `NodePort` as the type for the `kubernetes-dashboard` service so that it is available only from cluster nodes. A compromised pod running within the cluster can still access the Dashboard service, but well-crafted RBAC rules will limit the damage that it could do through that service.

Validating the Configuration

Once you have set up your Kubernetes cluster, there are two main options for validating whether it is configured safely. These options are configuration testing, where tests validate the deployment against a recommended set of settings, and penetration testing, where tests explore the cluster from the perspective of an attacker.

CIS Security Benchmark

The Center for Internet Security (CIS) publishes a Benchmark for Kubernetes giving best practices for configuring a deployment to use secure settings. If you're using Docker as your underlying runtime, you may also want to follow the CIS Benchmark for Docker.

It's a good idea to check your deployment against this benchmark. You might decide that not all the recommendations apply for you, but checking against the benchmark may alert you to insecure settings that you were unaware of. As a simple example, the Kubernetes tests will let you know whether your cluster is configured to allow anonymous access to the Kubernetes API.

Running the benchmark tests on all your nodes on a regular basis will help you spot any configuration drift that might affect your security posture.

Manually running the benchmark tests would be time-consuming. Fortunately, tools exist to automate the process, such as the Kubernetes Benchmark tool (for which Liz is a maintainer).

Penetration Testing

Enterprises commonly recruit the services of a "pen-tester," or penetration testing company, to probe their deployed software, searching for ways that an attacker could exploit the software or the platform on which it runs. A penetration-testing specialist will use creative approaches to find weak points in your cluster configuration and in the software running on it.

Additionally, you may like to consider testing with kube-hunter. This project (also one that Liz maintains) is an open source penetration testing tool specifically for Kubernetes.

To learn more about how to secure the Kubernetes control plane, check out the resources on the accompanying website, in the "Securing the Cluster" section.

Now that we have covered configuring the Kubernetes control-plane components, let's move on to discussing how to enable access to the cluster by known users and software entities.

CHAPTER 3

Authentication

If you've been using public cloud offerings such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform, you might have come across the term *identity and access management* (IAM), which allows you to define access to resources for users and services. In this chapter and in Chapter 4, we discuss how this is realized in Kubernetes.

All components, such as a kubelet running on a node, as well as users issuing `kubectl` commands, need to communicate with the API server. To process the request, the API server first has to verify who (or what, in the case of machines) is issuing the request; the server has to establish the identity of the caller, or in other words, to authenticate the caller. This chapter covers how authentication in Kubernetes works and the options you have at hand as a cluster operator.

Identity

For the API server to authenticate a request, the request issuer needs to possess an identity. At the time of writing, Kubernetes doesn't have a first-class notion of a human user, but rather assumes that users are managed outside Kubernetes via a directory service such as Lightweight Directory Access Protocol (LDAP) or single sign-on (SSO) login standards like Security Assertion Markup Language (SAML) or Kerberos. This is the standard approach in production, but if you're not using such a system, other authentication strategies are available.

User accounts are considered cluster-wide, so make sure that the usernames are unique across namespaces.

NOTE

A *namespace* in Kubernetes is a way to logically divide the cluster into smaller units of management. You can have any number of namespaces; for example, you might have one per application, or one per client, or one per project. Resources in Kubernetes are either namespaced (services, deployments, etc.) or cluster-wide (nodes, persistent volumes, etc.) and you can consider a namespace as one of the built-in security boundaries. "Security Boundaries" on page 45 provides more information on this topic.

It's not just humans who interact with Kubernetes. We often want a programmatic way for applications to communicate with the Kubernetes API; for example, to query, create, or update resources such as pods, services, or deployments. To that end, Kubernetes has a top-level resource to represent the identity of an application: the service account. A *service account* is a namespaced resource that you can use if your application needs to communicate with the API server. Many business applications don't need to manipulate Kubernetes resources in this way, so (following the principle of least privilege) they can have service accounts with limited permissions.

By default, Kubernetes makes the credentials of the service account available via a secret that is mounted into the pod (note that all files shown here are owned by `root`):

```
$ kubectl run -it --rm jumpod \
              --restart=Never \
              --image=alpine -- sh
~ $ ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt          namespace       service-ca.crt  token
```

Most important here is the `token` file provided, which is a JSON Web Token as per RFC7519:

```
~ $ cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3Nl...
```

You can use the debugger provided by *jwt.io* to see what exactly the payload of that token is; so, for example, copying content from the preceding `token` file gives the output shown in Figure 3-1.

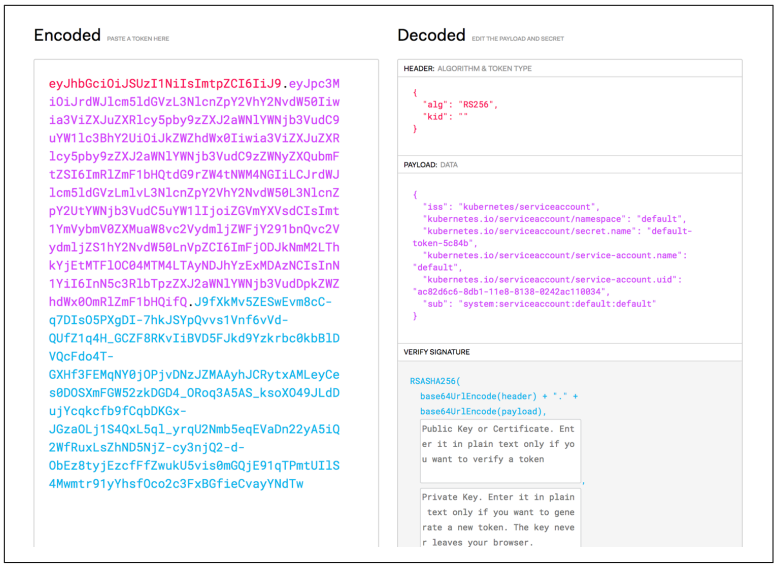

Figure 3-1. A JSON Web Token provided by a service account

If you don't explicitly specify a service account in the pod spec, the default service account for the namespace is used.

The general form of a service account is as follows:

```
system:serviceaccount:$NAMESPACE:$NAME
```

In Figure 3-2, you can see a more complex example setup.

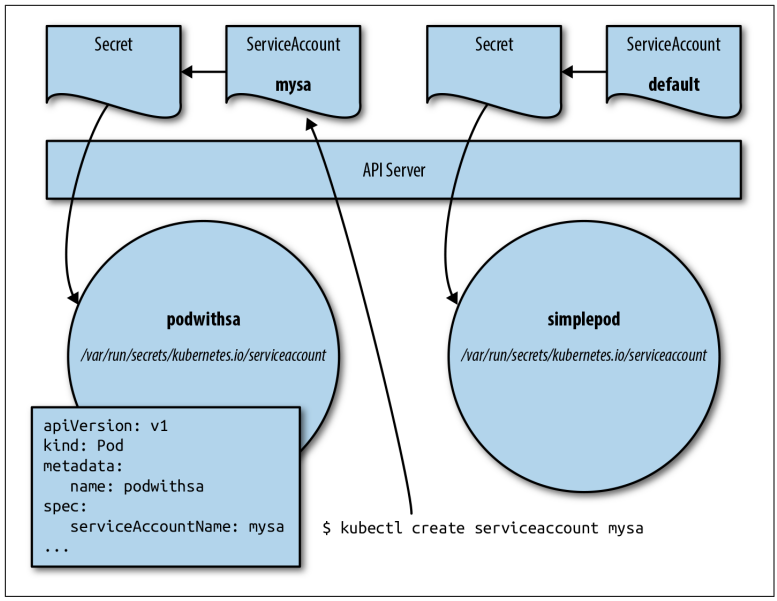

Figure 3-2. Service accounts

Here we have two pods, `simplepod` and `podwithsa`. The former doesn't specify the service account and hence ends up using the default service account of the namespace. On the other hand, `podwithsa` uses a dedicated service account called `mysa` that you can create, for example, using the following command:

```
$ kubectl create serviceaccount mysa
serviceaccount "mysa" created

$ kubectl describe serviceaccount mysa
Name:                mysa
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   mysa-token-prb4r
Tokens:              mysa-token-prb4r
Events:              <none>

$ kubectl get secrets
NAME                  TYPE                                  DATA AGE
default-token-dbcfn kubernetes.io/service-account-token 3    26m
mysa-token-prb4r    kubernetes.io/service-account-token 3    9m
```

What you can learn from the preceding output (also shown in Figure 3-2) is that the creation of a service account triggers the creation of a secret, attached to and managed by the service account. This secret contains the JSON Web Token discussed earlier.

Now that we have created the service account, we want to use it in a pod. How can you do that? Simply by using the `serviceAccountName` field in the pod spec to select the service account, in our case, `mysa`. Let's store a pod spec in a file called *podwithsa.yaml* with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: podwithsa
spec:
  serviceAccountName: mysa
  containers:
  - name: shell
    image: alpine:3.7
    command:
      - "sh"
      - "-c"
      - "sleep 10000"
```

You can launch the pod and inspect its properties as follows (the output has been edited for better readability):

```
$ kubectl apply -f podwithsa.yaml
pod "podwithsa" created

$ kubectl describe po/podwithsa
Name:         podwithsa
Namespace:    default
...
Volumes:
  mysa-token-prb4r:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  mysa-token-prb4r
    Optional:    false
...
```

And indeed, here you see that our `podwithsa` pod uses its own service account with the token `mysa-token-prb4r` (allowing it to communicate with the API server) available at the usual file location */var/run/secrets/kubernetes.io/serviceaccount/token* mounted into the pod.

At this point, you might be wondering why you would bother at all messing around with service accounts and not always use the default service account. This will make more sense when you learn how service accounts are used with RBAC to define permissions for users and applications in Chapter 4. For now, just remember that service accounts allow applications to communicate with the API servers (if they have to at all).

Now that we've covered the basics of identity in Kubernetes, let's move on to how authentication works.

Authentication Concepts

In Figure 3-3, you can see how the API server conceptually performs authentication by using one of the available strategies represented by the authentication plug-ins (learn more about the supported strategies in the next section).

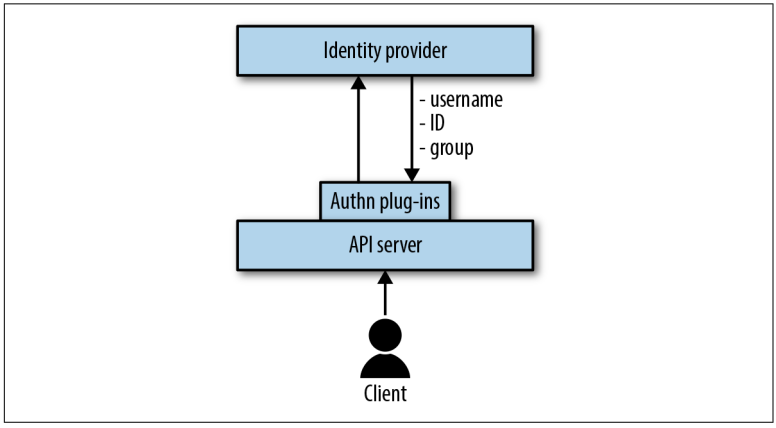

Figure 3-3. Authentication concepts

The flow Kubernetes uses to authenticate a client's request is as follows:

1. The client presents its credentials to the API server.
2. The API server uses one of the configured authentication plug-ins (you can enable multiple) to establish the identity with an identity provider.
3. The identity provider verifies the request information, including username and group membership.

4. If the credentials are in order, the API server moves on to check permissions as described in Chapter 4. Otherwise, it returns an HTTP `401 Unauthorized` client error status response code, and with that the request fails.

> **NOTE**
>
> The identity provider and its behavior depend on the authentication plug-in used. For example, it could simply be a file with usernames and passwords that you provide to the API server or an external system like Active Directory. Kubernetes is not opinionated concerning how you verify the credentials; it just provides the interface and enforces a certain flow to make sure requests come from well-known clients.

Kubernetes also supports user impersonation; that is, a user can act as another user. For example, as a cluster admin, you could use impersonation to debug any authorization issues.

Authentication Strategies

A couple of authentication strategies are available in Kubernetes, represented by authentication plug-ins. Depending on the size of the deployment, the target users (human versus processes), and organizational policies, you as a cluster admin can choose one or more of the following:

Static password or token file
: This strategy uses the Basic HTTP authentication scheme as per RFC7617. Essentially, the API server requires the client to provide the identify via an HTTP header named `Authorization` and the value of `Basic base64($USER:$PASSWORD)` in case of a static password file or `Bearer $TOKEN` in case of a static token file. Since it's inflexible to maintain a static file with the users and their passwords and requires direct access to the API server, this method is not recommended in production.

X.509 certificates
: With this strategy, every user has their own X.509 client certificate. The API server then validates the client certificate via a configured certificate authority (CA). If the client certificate is verified successfully, the common name of the subject is used as

the username for the request, and any organizations defined for the subject are used as groups. As an admin, you need to manage access to the CA as well as issue the client certificates, and reissue them as they approach expiry. Kubernetes does not, at the time of writing, support certificate revocation, and this is considered a good reason to use an SSO approach where possible.

OpenID Connect (OIDC)
: OIDC is an identity layer on top of the OAuth 2.0. With this strategy, you use OIDC to provide the API server with an `id-token` in the form of a JSON Web Token after using your provider's login page, such as Google or Azure Active Directory.

Bootstrap tokens
: These are an experimental feature targeting the cluster setup phase and can be used with installers such as `kubeadm`.

If you want to integrate with other authentication protocols such as LDAP, SAML, and Kerberos, you can use one of the following methods:

Authenticating proxy
: The API server can be configured to identify users from request header values, such as `X-Remote-User`. You need to take care of setting up and running the proxy; see, for example, Haoran Wang's post of an authentication example.

Webhook token authentication
: Essentially, a hook for verifying bearer tokens.

With that, we move on to some good practices and tooling around authentication.

Tooling and Good Practices

The majority of the effort in the context of authentication is with the Kubernetes cluster administrator. You would start off with existing infrastructure that you need to integrate with, such as an LDAP server your organization already uses to capture team members and group-related information. You also want to take into account the environment the cluster is running in, like a public cloud provider, a managed service (Amazon Elastic Container Service for Kubernetes, Azure Kubernetes Service, Google Kubernetes Engine, OpenShift

Online, etc.), or an on-premises deployment. The latter is important, as you may have different options depending on the environment and may end up having more or less work with the authentication bits, based on what authentication strategy you go for.

Several tools are available to help with this (you may wish to check the latest list on the website accompanying this book):

Keycloak
: An open source IAM solution with built-in support to connect to existing LDAP servers. Keycloak can authenticate users with existing OIDC or SAML 2.0 identity providers. A Helm chart is also available to deploy it in Kubernetes.

Dex
: An identity service that uses OIDC to drive authentication for other applications. Dex acts as a portal to other identity providers, allowing you to defer authentication to LDAP servers, SAML providers, or established identity providers like GitHub, Google, and Active Directory.

AWS IAM Authenticator for Kubernetes
: A tool to use AWS IAM credentials to authenticate to a Kubernetes cluster maintained by Heptio and Amazon.

Guard
: A Kubernetes webhook authentication server by AppsCode, allowing you to log into your Kubernetes cluster by using various identity providers, from GitHub to Google to LDAP.

In the last section of this chapter, we look at good practices in the context of authentication. Note that because a new Kubernetes release comes out every couple of months, some tips might be more relevant than others (as defaults change or new features are introduced):

Use third-party providers
: Unless you have to roll your own thing, integrate Kubernetes with third-party identity providers such as Azure, Google, or GitHub.

Don't use static files
: If you can't use third-party providers, prefer X.509 certificates over static password or token files.

Life cycle
: Ensure that when people leave the organization, their credentials are invalidated. With third-party providers, this task is typically easier compared to when you roll your own solution. In any case, regular audits help here as well to uncover holes.

To learn more about authentication options and gotchas, check out the resources on the accompanying website, in the "Authentication" section.

With this, we have reached the end of the discussion of authentication in Kubernetes, and you are ready to learn where and how the authentication information eventually is used: giving users and applications permissions and enforcing those, through a process known as *authorization*.

CHAPTER 4
Authorization

In this chapter, we focus on authorization in Kubernetes—assigning permissions to users and applications and in turn enforcing those. Authorization in Kubernetes verifies whether a certain action (such as "list pods" or "create a secret") is allowed by a certain user or application, and if it is allowed, performs that action or otherwise rejects it and potentially logs the attempt. We're building on the concepts and flows presented in Chapter 3, so if you haven't read that chapter yet, now is a good time.

Authorization Concepts

Kubernetes authorizes API requests by using the API server, evaluating the request attributes against the policies and subsequently allowing or denying the request. By default, permissions are denied, unless explicitly allowed by a policy. Conceptually, authorization in Kubernetes works as depicted in Figure 4-1.

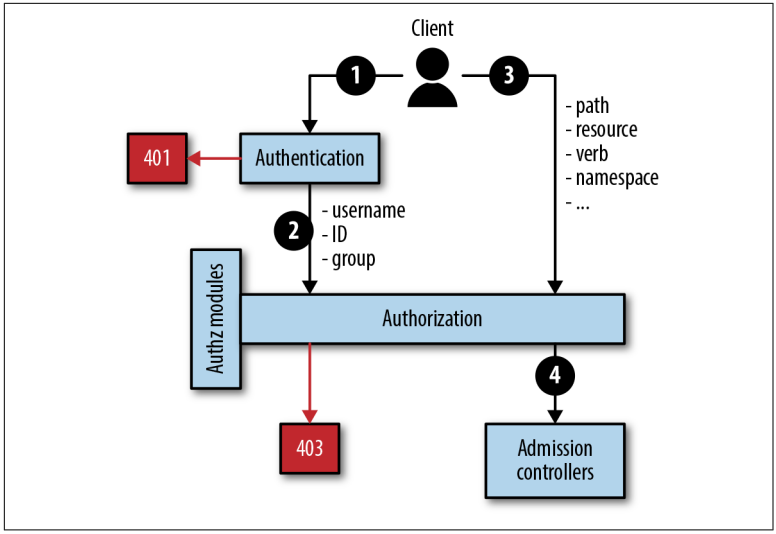

Figure 4-1. Authorization concepts

The authorization flow is as follows:

1. The client's request is authenticated. See "Authentication Concepts" on page 20 for details on this step.
2. If the authentication was successful, the credentials are taken as one input of the authorization module.
3. The second input to the authorization module is a vector containing the request path, resource, verb, and namespace (and other secondary attributes).
4. If the user or application is permitted to execute a certain action on a certain resource, the request is passed on further to the next component in the chain, the admission controller. If not, the authorization module returns an HTTP `403 Forbidden` client error status response code, and with that the request fails.

Now that you know how authorization works in principle in Kubernetes, let's look at the ways permissions can be enforced.

Authorization Modes

Kubernetes offers multiple ways to enforce permissions, represented by various authorization modes and modules:

Node authorization
: A special-purpose authorizer that grants permissions to kubelets based on the pods they are scheduled to run.

Attribute-based access control (ABAC)
: An authorizer through which access rights are granted to users through policies combining attributes (user attributes, resource attributes, objects, etc.).

Webhook
: A webhook is an HTTP callback—an HTTP `POST` that occurs when something happens. This mode allows for integration with Kubernetes-external authorizers.

Role-based access control (RBAC)
: This is explained in detail in the following section.

Since RBAC is the most important authorization method for both developers and admins in Kubernetes, let's look at it in greater detail.

Access Control with RBAC

Developed originally at Red Hat in the context of OpenShift, role-based access control (RBAC) was upstreamed to Kubernetes and is stable as of version 1.8 access. You should use RBAC for access control and not use ABAC or, even worse, use none.

As you can see in Figure 4-2, you have a few moving parts when dealing with RBAC:

Entity
: A group, user, or service account (representing an app—that wants to carry out a certain operation and requires permissions in order to do so).

Resource
: A pod, service, or secret that the entity wants to access.

Role
: Used to define rules for actions on resources.

Role binding
: This attaches (or binds) a role to an entity, stating that a set of actions is permitted for a certain entity on the specified resources.

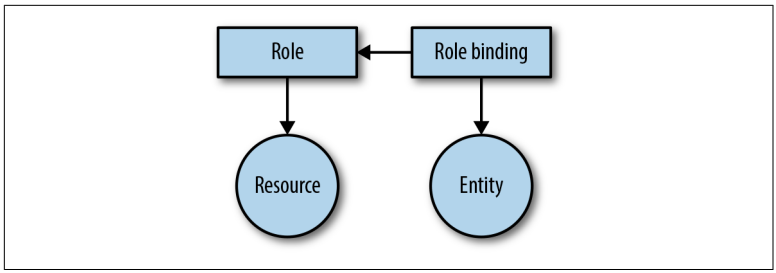

Figure 4-2. The RBAC concept

The actions on a resource that a role uses in its rules are the so-called verbs, such as the following:

- `get`, `list` (read-only)
- `create`, `update`, `patch`, `delete`, `deletecollection` (read-write)

Concerning the roles, we differentiate between two types:

Cluster-wide
: Cluster roles and their respective cluster role bindings

Namespace-wide
: Roles and role bindings

Sometimes it's not obvious whether you should use a role or a cluster role and/or role binding, so here are a few rules of thumb you might find useful:

- If you want to grant access to a namespaced resource (like a service or a pod) in a particular namespace, use a role and a role binding.
- If you want to reuse a role in a couple of namespaces, define a cluster role and use a role binding to bind it to a "subject" (an entity such as a user or service account).
- If you want to grant access to cluster-wide resources such as nodes or to namespaced resources across all namespaces, use a cluster role with a cluster role binding.

NOTE

Kubernetes prevents users from escalating privileges by editing roles or role bindings. Users can create or update a role only if they already have all the permissions contained in the role. For example, if user `alice` does not have the ability to list secrets cluster-wide, that user cannot create a cluster role containing that permission.

Kubernetes defines default roles that you should consider using before you start defining your own roles:

User-facing roles
: `cluster-admin`, `admin` (for namespaces), `edit`, and `view` that you can use out of the box for your end users.

Core components
: The Kubernetes control-plane components as well as nodes have predefined roles, such as `system:kube-controller-manager` or `system:node`, defining exactly the permissions the respective component needs in order to work properly.

Other components
: Kubernetes defines roles for noncore components that are almost always used alongside the core bits. For example, there's a role called `system:persistent-volume-provisioner` for enabling dynamic volume provisioning.

Other kinds of predefined roles also exist—for example, discovery roles (such as `system:basic-user`) or controller roles (such as `system:controller:deployment-controller`). These are internal to Kubernetes, and unless you're an admin debugging an installation or upgrade, they are typically not very relevant to your daily routine. If you want to know which roles are predefined and available in your environment, use the following (which in our case listed more than 50 roles, output omitted here):

```
$ kubectl get clusterroles
```

Now, this may sound intimidating and complex, so let's look at a concrete example. Say you have an application that needs to have access to pod information. You could use the `view` default cluster role for it:

```
$ kubectl describe clusterrole view
Name:         view
```

```
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources                                  ...  Verbs
  ---------                                  ...  -----
  bindings                                   ...  [get list watch]
  configmaps                                 ...  [get list watch]
  endpoints                                  ...  [get list watch]
  events                                     ...  [get list watch]
  limitranges                                ...  [get list watch]
  namespaces                                 ...  [get list watch]
  namespaces/status                          ...  [get list watch]
  persistentvolumeclaims                     ...  [get list watch]
  pods                                       ...  [get list watch]
  pods/log                                   ...  [get list watch]
  pods/status                                ...  [get list watch]
  replicationcontrollers                     ...  [get list watch]
  replicationcontrollers/scale               ...  [get list watch]
  replicationcontrollers/status              ...  [get list watch]
  resourcequotas                             ...  [get list watch]
  resourcequotas/status                      ...  [get list watch]
  serviceaccounts                            ...  [get list watch]
  services                                   ...  [get list watch]
  daemonsets.apps                            ...  [get list watch]
  deployments.apps                           ...  [get list watch]
  deployments.apps/scale                     ...  [get list watch]
  replicasets.apps                           ...  [get list watch]
  replicasets.apps/scale                     ...  [get list watch]
  statefulsets.apps                          ...  [get list watch]
  horizontalpodautoscalers.autoscaling       ...  [get list watch]
  cronjobs.batch                             ...  [get list watch]
  jobs.batch                                 ...  [get list watch]
  daemonsets.extensions                      ...  [get list watch]
  deployments.extensions                     ...  [get list watch]
  deployments.extensions/scale               ...  [get list watch]
  ingresses.extensions                       ...  [get list watch]
  networkpolicies.extensions                 ...  [get list watch]
  replicasets.extensions                     ...  [get list watch]
  replicasets.extensions/scale               ...  [get list watch]
  replicationcontrollers.extensions/scale ...  [get list watch]
  networkpolicies.networking.k8s.io          ...  [get list watch]
  poddisruptionbudgets.policy                ...  [get list watch]
```

As you can see, the `view` default role would work, but it additionally allows your application access to many other resources such as deployments and services. This is a potential security risk and goes against the principle of least privilege, so let's create a dedicated role for it. A role that allows you to retrieve only info about pods.

Since we want to set permissions for an application rather than a user whose identity is managed outside Kubernetes, we first have to create a dedicated service account representing the application's identity toward the API server. Also, it's a good practice to not use the default namespace, so let's start by creating a namespace `coolapp` that our application will live in and then a service account `myappid` in this namespace:

```
$ kubectl create namespace coolapp
namespace "coolapp" created
$ kubectl --namespace=coolapp create serviceaccount myappid
serviceaccount "myappid" created
```

Now that we have established an identity for our application, we can define a role `podview` that allows only viewing and listing pods in its namespace:

```
$ kubectl --namespace=coolapp create role podview \
          --verb=get --verb=list \
          --resource=pods

$ kubectl --namespace=coolapp describe role/podview
Name:         podview
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [get list]
```

That looks more like it! The role `podview` allows only for viewing pods. Next, we need to attach the role `podview` to our application, represented by the service account `myappid`. We do this by creating a role binding (which binds a role to a human or machine user) called `mypodviewer`, like so:

```
$ kubectl --namespace=coolapp create rolebinding mypodviewer \
          --role=podreader \
          --serviceaccount=coolapp:myappid
rolebinding.rbac.authorization.k8s.io "mypodviewer" created

$ kubectl --namespace=coolapp describe rolebinding/mypodviewer
Name:         mypodviewer
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  podreader
Subjects:
```

```
Kind            Name     Namespace
----            ----     ---------
ServiceAccount  myappid  coolapp
```

Note that for the service account parameter, we had to use the fully qualified name (`$NAMESPACE:$SERVICEACCOUNT`). And with this last command, the service account `myappid` representing our application is bound to the `podreader` role and all of that in the namespace `coolapp`.

But how can you be sure that only the required permissions have been granted? You can check it like so:

```
$ kubectl --namespace=coolapp auth can-i  \
          --as=system:serviceaccount:coolapp:myappid
          list pods
yes

$ kubectl --namespace=coolapp auth can-i  \
          --as=system:serviceaccount:coolapp:myappid
          list services
no
```

The last step, not shown here, is simply to use `serviceAccountName` in the pod spec of your app, as you saw in the example at the end of "Identity" on page 15.

Tooling and Good Practices

Several tools focus on authorization with RBAC (see also the up-to-date list on our website):

audit2rbac
: A tool that allows you to automatically determine what permissions are necessary for a certain application and generate RBAC roles and bindings for you.

rbac-manager
: A Kubernetes operator that simplifies the management of role bindings and service accounts.

kube2iam
: A tool that provides AWS IAM credentials to containers based on annotations.

In the last section of this chapter, we look at good practices in the context of authorization:

Use RBAC
: This should be the standard now—if not, please do upgrade Kubernetes to a version equal to or greater than 1.8. Pass the `--authorization-mode=RBAC` parameter to the API server to enable this.

Disable automounting of the default service account token
: Most applications don't need to talk to the API server, so they don't need an access token. This is especially important if you're not using RBAC. You can do this by specifying `automountServiceAccountToken: false` in the PodSpec for your applications, or you can patch the default service account so that its credentials are not automatically mounted into pods:

```
$ kubectl patch serviceaccount default \
    -p $'automountServiceAccountToken: false'
serviceaccount "default" patched
```

Use dedicated service accounts
: If your application needs access to the API server, either because it's a system-level thing or has been written with Kubernetes in mind, it is good practice to create a dedicated service account per application and configure RBAC to be specifically limited to the needs of that application. Bear in mind that if a pod is compromised in some way, the attacker will have access to the service account associated with that pod, and its corresponding permissions. See also "Identity" on page 15 for more details.

To learn more about RBAC and how to use it, check out the resources on the accompanying website, in the "Authorization" section.

Now that you know the basics of performing authentication and authorization in Kubernetes, let's discuss how to make your applications more secure, starting with container images in the next chapter.

CHAPTER 5

Securing Your Container Images

Until now, we've been discussing things mainly from the point of view of a Kubernetes cluster administrator. Going forward, we'll switch gears and focus more on developers, operators, or even DevOps teams who want to deploy code to run on the cluster.

The software that you run in your Kubernetes cluster gets there in the form of container images. In this chapter, we'll discuss how to check that your images:

- Don't include known critical vulnerabilities
- Are the images you intended to use, and haven't been manipulated or replaced by a third party
- Meet other image policy requirements your organization might have in place

Vulnerabilities

In this context, a *vulnerability* is a flaw in a piece of code that an attacker can exploit to cause undesirable consequences, and that has been publicly disclosed (typically, through the National Vulnerability Database). For example, the renowned Heartbleed vulnerability was a flaw in the OpenSSL library that allowed attackers to access system memory, and hence steal encrypted information.

Scanning Container Images

To detect vulnerabilities, you need to use a container image scanner. The basic function of a container image scanner is to inspect the packages included in an image, and report on any known vulnerabilities included in those packages. At a minimum, this looks at the packages installed through a package manager (like `yum` or `apt`, depending on the OS distribution). Some scanners may also examine files installed at image build time; for example, through ADD, COPY, or RUN operations in a Dockerfile. Some scanners also report on known malware (e.g., viruses) or the presence of sensitive data (like passwords and tokens).

To ensure that you're not running vulnerable code in your deployment, you should scan any third-party container images as well as the ones built by your own organization.

New vulnerabilities continue to be found in existing software, so it's important to rescan your images on a regular basis. In our experience, it's typical for enterprise customers to rescan the images in use on their production systems every 24 hours, but you should consider your own risk profile. Depending on the scanning tool you use, this may be a simple configuration setting, or you may need to write automation scripting to put this in place.

Several commercial image-scanning tools are available as well as some open source and/or free-to-use options.

Some registries provide metrics on the health of the container images they store. For example, the Red Hat Container Catalog grades images from A–F, and the Google Container Registry and Docker Trusted Registry also include image scan results.

Patching Container Images

Once you have identified that you have a container image that includes a package with a vulnerability, you need to update the container to use a fixed version of the package. Please don't be tempted to SSH into your running containers and run something like `yum update` or `apt-get update`, as this is an antipattern for containers! It quickly becomes unfeasible to manually patch like this when running hundreds or thousands of instances across a cluster. Factor in the self-healing nature of Kubernetes, which ensures that a failed

container will be replaced with a new one, and autoscaling, which can create and destroy containers automatically, and it becomes clear that it's really not possible to keep up with the patching process manually.

The key to "patching" in a container deployment is to rebuild a new container image, and then redeploy the containers based on that new image. The build part is typically automated through a continuous integration (CI) pipeline, and this may be extended to cover continuous deployment (CD) as well. While CI/CD and its bright new cousin, GitOps, are out of scope for this book, it is worth examining how security tooling fits into the CI/CD pipeline.

CI/CD Best Practices

Image scanning can be integrated into the CI/CD pipeline to automate the process of rejecting images, as shown in Figure 5-1. Many scanners can report a pass or fail for each image, either on basic criteria ("fail all images with high-severity vulnerabilities") or more-complex, custom policies ("fail if the image has any high-severity vulnerabilities, ignoring this set of whitelisted vulnerabilities, and also fail if the image has this particular blacklisted medium-severity vulnerability, or includes sensitive data").

You can use this pass/fail in several places in your CI/CD pipeline:

- A failed scan can result in a failed build.
- A failed scan before deployment can prevent the image from being deployed.
- A failed scan on an image that's already in production can result in an alert so that operators can take remedial action.

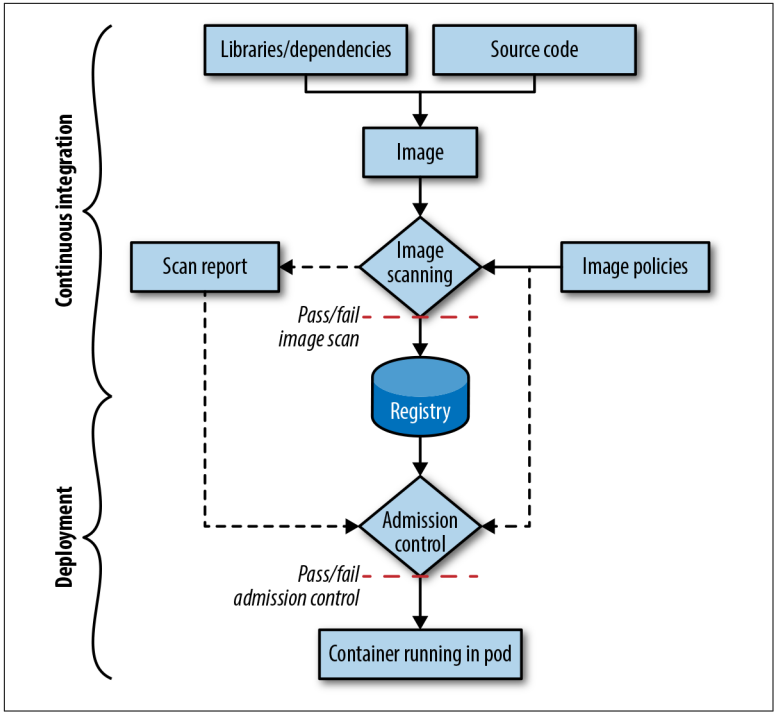

Figure 5-1. The CI/CD pipeline

In Figure 5-1, we also see an admission control step. Advanced solutions may also use some form of dynamic admission control (see "Dynamic Admission Control" on page 72) to ensure that images are deployed only if they have been scanned, and the scan was successful. This step can also automatically check whether the image can be trusted, as we'll come to in "Image Trust and Supply Chain" on page 40.

A good best practice is to use automation to scan all images before they are stored in a container registry, rejecting any images that fail the scan. The next question to consider, then, is the use of a secure container registry.

Image Storage

Container images can be stored in public or private registries. Many security-conscious organizations use one or more private registries and require that only images from these registries can be deployed.

Running a private registry means that you have greater control over who has permissions to read and write images. You can also deploy the registry with limited network access, perhaps using a firewall so that only known IP addresses can access it.

Several offerings are available for running your own registry, including Docker's own implementation, GitLab's Container Registry, and Quay from Red Hat.

The major hosted Kubernetes solutions all offer a container registry solution, which can have the advantage of tight integrations with the cloud platform that you are already familiar with. For example, if you are using AWS, the Elastic Container Registry uses IAM for access control.

Whichever registry solution you are using, unless you are pulling public images, you will need to grant access to your Kubernetes cluster so that it can pull images from the registry. It's a good idea to use *read-only accounts* for this purpose; with the exception of, say, a CI/CD system deployed on Kubernetes, it's highly unusual that your Kubernetes nodes would need to push images into the registry. By using read-only credentials, you mitigate the possibility that an attacker who gains access to the cluster can push modified images into your registry, which then get pulled and run.

Correct Image Versions

When we define the containers that will run in pods, the PodSpec refers to the container image by using a fully qualified image name that includes the registry, the owner, the repository, and a reference to a particular image version—for example, *gcr.io/myname/myimage:1.0*.

Typically, the version reference is in the form of a tag (`1.0` in this example). However, tags are mutable (the same tag can be moved to refer to a different image), and an image can have multiple tags, so you need to handle your tags with care.

The Container Solutions blog provides a good demonstration of the confusion that can be created with image tags.

To be certain that you are deploying a particular version of an image, it's possible to refer to it by its unique digest instead of the

tag. Here's an example of YAML specifying a container in this way (digest truncated for clarity):

```
spec:
  containers:
  - name: myimage
    image: gcr.io/myname/myimage@sha256:4a5573037f358b6cdfa2...
```

While this ensures that you pick up a particular version of a container image, it means updating YAML whenever there is a new revision. In our experience, it's much more common to refer to images by using a semantic version tag.

If you supply neither a tag nor a digest, the image version tagged `latest` will be used. It's recommended to avoid using the `latest` version, at least in production, because it's hard to keep track of exactly what code is running, and worse, what version to use should you want to roll back to a previous version.

Running the Correct Version of Container Images

Make sure to always run the correct version of your container image by doing the following:

- Using semantic versioning when tagging your images. That way, it's easy to identify the version you expect to be running. An alternative approach is to always refer to an image by its unique SHA digest.
- Using the `AlwaysPullImages` admission controller to ensure that the most recent version that matches the specified tag is obtained. Without this, a node may run a stale version of the image that it pulled some time in the past. You don't need this if you are confident that all your images have immutable tags, or your YAML refers to all images by SHA. Using `AlwaysPullImages` also ensures that the pod doesn't bypass the credentials check that it is entitled to access that image, by using a locally cached version.

Image Trust and Supply Chain

We have discussed how to specify the correct version of an image in your YAML files, but a potential problem still remains: ensuring that the version pulled from the image registry is the genuine, intended

code. Several projects aim to help with the problem of ensuring the provenance of the application software running in a deployment:

- The TUF project, and its implementation Notary, use signing to ensure the integrity of an image—that is, to make sure that the image retrieved from a registry at deployment time is the correct version as signed by a trusted authority. The Portieris admission controller can prevent images from being deployed if they don't have a valid Notary signature.
- Grafeas is another approach to storing and assuring image metadata, including signatures to validate image integrity.
- The in-toto project provides a framework to protect the integrity of the components installed into an image, and the tests they have passed.
- Commercial security solutions can also add validation that the image being deployed is a precisely approved version that matches your policies.

In a high-risk environment, you will want to explore tools like these for validating image provenance.

Minimizing Images to Reduce the Attack Surface

Following the principle of "Limiting the Attack Surface" on page 4, you can take it as a general rule that the smaller the image, the smaller the attack surface:

- By minimizing the amount of code you include in the image, you can reduce the likelihood of a vulnerability.
- There is rarely a good reason to include an SSH daemon, as explained by Jérôme Petazzoni.
- Along similar lines, other utilities in your images may not be required by the application code. Excluding them will make the running container less useful to an attacker who manages to compromise it. For example, suppose that a container has access to database credentials that it accesses by reading from a secrets file (see Chapter 7). If the container image doesn't include utilities like `cat` or `more`, it will be that much harder for attackers to read the credentials even if they gain access to the running con-

tainer. If the image doesn't even have a shell (like `sh` or `bash`) included in the image, this will make an attack even harder.

- Taking this idea even further, if your application code can be built as a static binary, you can build an image that contains nothing but that binary. This image will have no utilities that an attacker can take advantage of.

As a counterpoint, however, consider that by excluding core tooling such as `cat`, troubleshooting will also be hard for you, so you want to aim for a sensible trade-off here.

To learn more about reducing image sizes, see Abby Fuller's talk on reducing image sizes. For more information on building secure container images, check out the resources on the accompanying website, in the "Securing Your Container Images" section.

As you've seen in this chapter, a lot can be done at the image build stage to ensure that the application code is safe to deploy. Next, we turn our attention to Kubernetes security features that apply while code is running.

CHAPTER 6

Running Containers Securely

Now that you know how to build container images in a secure manner from Chapter 5, we move on to the topic of running those images as containers in Kubernetes. In order to run containers securely in Kubernetes, we aim to do the following:

- Use least privilege to carry out the task at hand.
- Do only the minimal host mounts necessary.
- Limit communication between applications, and to and from the outside world, to a defined and deterministic set of connections.

Before we discuss the security boundaries in Kubernetes and the features that you have at your disposal to enforce policies, let's have a quick look at two topics essential for you to appreciate the rest of the chapter: why you should not run containers as root (unless you have to) and how the API server deals with enforcing policies.

Say No to Root

As "Mr. SELinux" Dan Walsh pointed out in "Just Say No to Root (in Containers)," there's little need to run containers as root. Some exceptions are as follows:

- Your container needs to modify the host system; for example, modifying the kernel's configuration.

- The container needs to bind to privileged ports on the node (below 1024—for example, *nginx* serving on port 80). In practice, this can be by-and-large avoided through port mappings and the service abstraction in Kubernetes.
- Installing software into a container at runtime: traditional package management systems might require root to function or store files in a certain location with a different user ID than the user executing the program. This approach is generally considered bad practice since any code installed at runtime has not been scanned for vulnerabilities or other policy requirements (see Chapter 5).

If your container does not fall into one of the preceding categories, then according to the principle of least privilege, it would make sense to run it as a nonroot user. You can do this by including a `USER` command in the Dockerfile, defining a user identity that the code should run under.

The advocacy site *canihaznonprivilegedcontainers.info* has more background resources on this topic, and Liz explored it in her "Running with Scissors" keynote at KubeCon Copenhagen in 2018. However, general awareness around this topic is sadly still low, and most images on Docker Hub are built to run as the root user by default (having no `USER` command).

Let's move on and see how the API server enforces policies.

Admission Control

When a client submits a request to the API server and that request has been authenticated (Chapter 3) and the client is authorized (Chapter 4) to carry out the operation, there is one more step the API server performs before persisting the resource in etcd: admission control. A whole slew of admission controllers are included in the API server, that you, as a cluster admin, can configure. The official docs list explains the more than 30 controllers in great detail; some relevant ones in the context of running containers securely are as follows:

`AlwaysPullImages`
: Modifies every new pod to force the image pull policy to `Always`, overwriting the default specification. This can be

important (especially in a multitenant environment) since the default behavior is that when an image is pulled to a node, it is stored locally and can be accessed by other pods on the node without them needing to pull it again. These other pods would therefore bypass the registry credentials check when pulling an image to ensure that they are entitled to access that image.

`DenyEscalatingExec`
: Denies `exec` and `attach` commands to pods that run with escalated privileges, allowing host access. Prevents attackers from launching interactive shells into privileged containers, so this is recommended.

`PodSecurityPolicy`
: Acts on creation and modification of the pod and determines whether it should be admitted based on the requested security context and the available policies. See also "Policies" on page 47.

`LimitRange` *and* `ResourceQuota`
: Observes the incoming request and ensures that it does not violate any of the constraints enumerated in the `LimitRange` and `ResourceQuota` object in each namespace, respectively, which helps to combat denial-of-service attacks.

`NodeRestriction`
: Limits the permissions of each kubelet, as discussed in Chapter 2.

Now that you're equipped with the basics of policy enforcement in Kubernetes, let's focus on the main topic of this chapter: security boundaries and how to realize them.

Security Boundaries

We introduced some security principles in Chapter 1, and one of those principles is defense in depth. Kubernetes gives you a set of first-class mechanisms to realize defense in depth. To better understand what that means, have a look at Figure 6-1, which shows the security boundaries present by default.

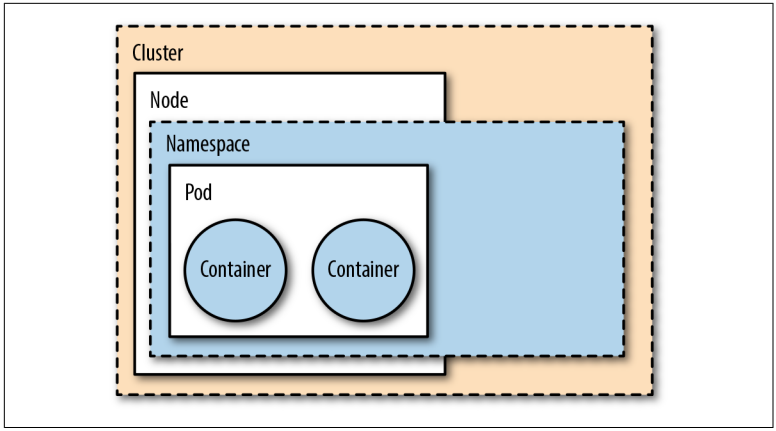

Figure 6-1. Security boundaries

By *security boundary*, we mean a set of controls to prevent a process from affecting other processes and/or accessing data from other users. From the most outer to the most inner layers of isolation, these boundaries are as follows:

Cluster
: Comprises all nodes as well as control-plane components, providing network isolation, and forms the top-level unit (modulo federation, not in scope here). You might prefer different clusters for each team and/or stage (for example, development, staging, production) to implement multitenancy over, say, namespace-level or node-level isolation.

Node
: A virtual or bare-metal machine in the cluster hosting multiple pods and system components such as the kubelet or kube-proxy, typically labeled with system properties. Those nodes are restricted to exactly access the resources necessary to carry out their tasks; for example, when a pod is scheduled on that node. You can separate sensitive workloads by assigning pods to certain nodes either using the `nodeSelector` or, even better, using node or pod affinity. The node authorizer discussed in "Authorization Modes" on page 26 allows you to minimize the blast radius, effectively helping to confine an attack to a single node.

Namespace
: A sort of virtual cluster, containing multiple resources such as services and pods. Namespaces are the basic unit for authoriza-

tion (see "Access Control with RBAC" on page 27). With certain admission controllers, as shown in "Admission Control" on page 44, you can restrict resource depletion and with that help combat; for example, denial-of-service attacks.

Pod
: A management unit Kubernetes uses to group containers with the guarantee that all containers in the pod are scheduled on the same node. It offers a certain level of isolation: you can define a security context and enforce it (discussed in "Security Context and Policies" on page 48) as well as specify network-level isolation (see "Network Policies" on page 52).

Container
: A container is essentially a combination of cgroups, namespaces, and copy-on-write filesystems that manages the application-level dependencies. By configuring the Quality of Service of your pods, you can influence the runtime behavior, but unless you're using advanced runtime sandboxing techniques as discussed in "Sandboxing and Runtime Protection" on page 69, containers typically do not provide strong isolation guarantees beyond the kernel-level security ones.

Remember that maximizing the defense here requires a joint effort by developers and cluster or namespace admins since some of the responsibilities (such as creating a container image) fall into the realm of the former, and others (like managing nodes or namespaces) fall into the realm of the latter.

The layout and composition of the security boundaries shown here were based on and motivated by an excellent blog post by the Google Cloud Platform team. You can read more here: "Exploring Container Security: Isolation at Different Layers of the Kubernetes Stack."

Now that you know about the security boundaries, let's see what mechanisms you have available to establish and enforce them.

Policies

Kubernetes offers two pod-level security-policy mechanisms allowing you to restrict what processes can do within a pod (as described

in the next section) and how pods are allowed to communicate (as laid out in "Network Policies" on page 52).

Security Context and Policies

A security context defines privilege and access control settings on either the pod or container level. The supported settings are as follows:

Implement discretionary access control
: Set permissions to access operating system objects, such as files, based on user or group ID as well as running as an (un)privileged process.

Capabilities
: Rather than giving someone root access, you can use capabilities to split the (unrestricted) root access into a set of separate permissions such as `CHOWN` or `NET_RAW`.

Apply profiles
: Configure seccomp by filtering system calls for processes or configure AppArmor to restrict the capabilities of processes.

Implementing mandatory access control
: Through configuring SELinux, by assigning security labels to operating system objects.

Using a security context is straightforward: use the `securityContext` field on either the pod level or on the level of a particular container. For example, imagine you want to define the following:

- All containers in the pod must run under user `1001`, through the `runAsUser` setting.
- In the `webserver` container, prevent `setuid` binaries from changing the effective user ID as well as prevent files from enabling extra capabilities by setting `allowPrivilegeEscalation` to `false`.

This translates into the following pod specification we store under *secconpod.yaml*:

```
apiVersion: v1
kind: Pod
metadata:
  name: securepod
```

```
spec:
  securityContext:
    runAsUser: 1001
  containers:
  - name: webserver
    image: quay.io/mhausenblas/pingsvc:2
    securityContext:
      allowPrivilegeEscalation: false
  - name: shell
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

Now you can launch the pod and check the user that the respective container is running under:

```
$ kubectl apply -f secconpod.yaml
pod "securepod" created

$ kubectl exec -it securepod --container=webserver -- id
uid=1001 gid=0(root) groups=0(root)

$ kubectl exec -it securepod --container=shell -- id
uid=1001 gid=0(root) groups=0(root)
```

So that works great; we can make sure that, for example, a container in a pod doesn't run as root. But how can you make sure, as a cluster or namespace admin, that your developers use appropriate security contexts?

Enter *pod security policies*. A pod security policy is a cluster-wide resource that allows you to enforce the usage of security contexts. The API server automatically enforces those policies for you, using admission controllers, as described in "Admission Control" on page 44. If a pod's specification doesn't meet the requirements of the pod security policy, it won't be run (although note that for pod security policies to take effect, the `PodSecurityPolicy` admission plugin must be enabled, and permission must be granted to use that policy for the appropriate users).

`PodSecurityPolicy` allows us to define `securityContext` context settings, along with other security-related settings such as the seccomp and AppArmor profiles.

NOTE

A word on seccomp and AppArmor profiles: by default, containers running under Docker use a seccomp and AppArmor profile that prevent some system calls that most containerized applications have no business trying to run. Docker is used to provide the runtime layer for many Kubernetes installations, and it would be easy—but sadly incorrect, at least at the time of writing—to assume that the same profiles would be used by default in Kubernetes as well.

To enable the default Docker seccomp profile, include the following annotations set in your pod security policies:

```
annotations:
  seccomp.security.alpha.kubernetes.io/allowedProfileNames: \
    'docker/default'
  seccomp.security.alpha.kubernetes.io/defaultProfileName: \
    'docker/default'
```

Let's see how we can enforce the "must run as nonroot user" scenario, along with Docker's default seccomp and AppArmor profiles, using the following policy:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: nonroot
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: \
      'docker/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: \
      'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName: \
      'docker/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName: \
      'runtime/default'
spec:
  privileged: false
  allowPrivilegeEscalation: false
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: MustRunAs
    ranges:
      - min: 1000
        max: 1500
  fsGroup:
```

```
    rule: MustRunAs
    ranges:
      - min: 1000
        max: 1500
```

Least privilege security settings

We have already discussed why it's preferable to limit the container to running as nonroot, and that it's a good idea to use seccomp and AppArmor profiles. Other settings in `securityContext` and `PodSecurityPolicy` are worth considering to restrict a pod to its least privileges:

Use a read-only root filesystem
: A common attack pattern in a compromised container is for attackers to write an executable file that they will then run. If your application code doesn't need to be able to write into the filesystem inside the container, the `readOnlyRootFilesystem` setting prevents that approach.

Limiting host volume mounts
: As discussed in "Running with Scissors", certain sensitive directories should not be mounted from the host into a container without very good reason, as a compromised container (or just buggy code) could lead to undesired changes on the host. The `allowedHostPaths` parameter in `PodSecurityPolicy` allows you to limit what can be mounted and therefore made accessible to the container.

Disallow privileged access
: Unless your container has a particular need for privileged Linux capabilities, `privileged` and `allowPrivilegeEscalation` should be `false`. Privileged access within a container is effectively the same as root access on the host.

The reference documentation on `PodSecurityPolicy` is worth examining in detail if you want to carefully restrict the permissions granted to container code.

With that, we've covered the basics of pod-level policies and move on to communication between pods.

Network Policies

Limiting the traffic that can flow between pods adds a good layer of security:

- Even if an external attacker is able to reach the cluster network, network policy can stop that attacker from sending traffic that reaches application code running inside pods.
- If a container somehow becomes compromised, an attacker will typically try to explore the network to move laterally to other containers or hosts. By restricting the addresses, ports, and pods that can be contacted, the attacker's ability to reach other parts of the deployment is curtailed.

Applications running in Kubernetes can potentially communicate with outside clients (north-south traffic) as well as with other applications running within the Kubernetes cluster (east-west traffic).

By default, all kinds of ingress (incoming) and egress (outgoing) traffic are allowed, but you can control how pods are allowed to communicate by using a Kubernetes feature called network policies. From version 1.7 onward, this feature is considered stable and hence ready for use in production.

NOTE

Not all Kubernetes networking solutions support network policy! If your networking add-on doesn't implement a controller for `NetworkPolicy` resources, defining them will have no effect.

Different Kubernetes distributions support network policies to different degrees. Popular network policy providers include Calico, Weave Net, OpenShift SDN, and Cilium. The latter is a BPF-based (Berkeley Packet Filter) implementation with a promising future.

Example Network Policy

Let's look at an example network policy preventing all ingress traffic to and egress traffic from all pods in the namespace `lockeddown`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: nonetworkio
  namespace: lockeddown
```

```
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

A network policy applies to the set of pods that match the `podSelector` defined in the `spec`. Typically, a label selector is used to match a subset of pods; an empty `podSelector` as in the preceding example matches all pods.

If a pod is not matched by any network policies, all traffic is allowed to and from that pod.

The Kubernetes documentation includes example network policies that you might use to limit traffic to and from all pods by default.

Effective Network Policies

For network policies to be at their most effective, we want to ensure that traffic can flow only where it is needed, and nowhere else. To achieve this, you will typically start with a `DenyAll` default policy that matches all pods with an empty `podSelector`, just as in the preceding `lockeddown` example. Then take a structured approach to adding network policies that allows traffic between application pods as necessary.

Suppose we have an application called `my-app` that stores data in a Postgres database. The following example defines a policy that allows traffic from `my-app` to `my-postgres` on the default port for Postgres:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-myapp-mypostgres
  namespace: lockeddown
spec:
  podSelector:
    matchLabels:
      app: my-postgres
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: my-app
      ports:
```

```
    - protocol: TCP
      port: 5432
```

We then likely want to allow traffic from the internet to access `my-app`, which we can achieve with another network policy like this:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-external
  namespace: lockeddown
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
  - from: []
```

The combination of these three network policies allows the application traffic to flow as desired from external users to the application pods, and from the application pods to the database, but traffic is not permitted anywhere else in this `lockeddown` namespace.

Ahmet Alp Balkan has put together a set of useful network policy recipes as well as a good post on the topic of "Securing Kubernetes Cluster Networking". Another helpful backgrounder is "Kubernetes Security Context, Security Policy, and Network Policy" by Mateo Burillo.

Going forward, service meshes will also play a role in this area. See "Service Meshes" on page 72 for more on this topic.

Metadata API in cloud platforms

Platforms such as AWS, Microsoft Azure, and Google Cloud Platform pass configuration information to nodes through a Metadata API. This can be the source of serious escalations; for example, as disclosed in a bug bounty at Shopify. This can include critical information including the node's kubelet credentials, so it is important to restrict access to these APIs.

This can be achieved through network policies that block traffic to the Metadata API for all pods that don't explicitly need access. Azure and AWS both use the IP address 169.254.169.254, and Google uses the domain name *metadata.google.internal*.

Resource quotas and networking

Resource quotas are used to limit the resources available to a namespace. We discuss this more generally in "Multitenancy" on page 70, but they are worth also considering here while we are thinking about restricting network access.

Resource quotas can be used to limit the number of service resources—and more specifically, the number of `NodePort`- or `LoadBalancer`-type services. Setting a limit of 0 can prevent a user from creating an application that is accessible from outside the cluster.

Reaching the end of this chapter, you now know how to create images in a secure way and subsequently use those images to run containers securely in Kubernetes by using the built-in features such as namespaces, RBAC, and policies on the pod and network level. With that as a solid foundation, we next look at how to manage sensitive data in Kubernetes.

CHAPTER 7

Secrets Management

Your application code often needs access to secret information, like credentials, in order to do its job. For example, if you run an e-commerce site, some components will need access to a product database, and other components likely will need to be able to manage user or payment information. These components will need the right access tokens or username/password combinations so they can access the data they need to view or manipulate.

In this chapter, we consider the options for passing secret information into your code running under Kubernetes. Kubernetes provides a "secret" resource for this purpose, but there are different ways of using these secrets that you will want to weigh. In addition, you also need to be aware of some aspects of the Kubernetes secrets implementation from a security perspective.

Applying the Principle of Least Privilege

The principle of least privilege has two consequences on secrets management in Kubernetes:

- We want to ensure that containerized code can read only the secrets that it needs.
- It's a good idea to have a different set of secrets for different environments (like production, development, and testing). The development and test credentials can be shared with a wider set

of team members without necessarily giving them full access to the production credentials.

Secret Encryption

Since secret values protect sensitive data, we want them to be hard to access. Ideally, they should by protected at rest and in transit:

At rest
: Secrets should always be stored on disk in encrypted form, so that an attacker with access to the filesystem cannot simply read them from the file. In this chapter, you will see how secrets are stored in Kubernetes, and our options for encrypting at rest.

In transit
: Secrets should be encrypted whenever they are sent as network traffic, so that an attacker snooping on the network cannot read them as they pass.

Encryption in transit is achieved by encrypting the traffic between the Kubernetes control-plane components using TLS (see Chapter 2). In order to consider how we can store our secrets safely in encrypted form, let's look at how Kubernetes manages secrets, and the options for where they can be stored.

Kubernetes Secret Storage

The Kubernetes secret resource type is a mechanism for passing secrets to your code without them appearing in plain text in the pod's YAML. Instead, the pod specification refers to a secret by name, and the actual value of the secret is configured separately.

Take care with the storage and control of any YAML or JSON manifest files used to define secrets. If you check these manifests into source code control, the secret value is accessible to anyone with access to that repository. Note also that secret values encoded in base64 in secret manifest files are not encrypted!

The default storage for secret values is etcd, or you can use third-party secret storage solutions.

Storing Secrets in etcd

By default, secret values are stored alongside other configuration information in the etcd database; they are simply base64 encoded.

Although base64 encoding makes content unreadable to the human eye, it does not encrypt it. If you come across base64-encoded information, you simply need to pass it through base64 decoding to retrieve the original information; no key is required. In other words, base64 is effectively plain text to an attacker.

Anyone who gains access to your etcd database will be able to read base64-encoded secrets from it. You can control access by configuring secure access only (see Chapter 2), but even then there is a risk: your data is written to disk unencrypted, and if an attacker gains access to the filesystem, your data may be compromised.

You can avoid this risk by ensuring that your etcd cluster is also encrypted on disk. (The API server has access to the encrypted data in etcd, so you will also want to limit API access as described in Chapter 2 and Chapter 4.)

The *EncryptionConfig* YAML file for etcd encryption includes a secret for unlocking the encrypted data. You should make sure that this file can be read only by the user account that runs the Kubernetes API server, which often is `root`.

It's good practice to rotate the encryption secret from time to time. If you have multiple etcd nodes, you should also encrypt the communication between them, to prevent the secret values from being passed in the clear.

Storing Secrets in Third-Party Stores

Some third-party systems are specifically designed to store secret and sensitive values. Using these in conjunction with Kubernetes is considered by many to be a more secure option than storing the secrets alongside less-sensitive information in etcd.

The major cloud providers have key management systems that can be used in this way. Other third-party solutions include HashiCorp Vault and CyberArk Conjur.

Commercial container security tools offer integration with multiple backend secret stores, and can also control access so that only specific containers have access to particular secrets.

Passing Secrets into Containerized Code

There are three ways to get secrets (or any other kind of information) into a container so that they are accessible to the application:

- Building them into the image itself
- Passing them into the container as environment variables
- Mounting a volume into a container so that code can read the information out of a file on that volume

You might be thinking that the container application could query secrets through some kind of network activity, but then the question arises: how do you stop that information from being available to bad actor entities without requiring credentials of some kind? Then those credentials are a secret that needs to be passed into the container first, and we are back to the set of three options.

Kubernetes secrets support the last two of these approaches, although for reasons we will cover shortly, the third option of mounting a volume is generally considered the safest, so if you are short of time, skip ahead to "Passing Secrets in Files" on page 63. Before we come to that, let's consider why building secrets into container images is really not a great idea.

Don't Build Secrets into Images

The first of these options is a bad idea for secrets, and here are a few reasons:

- Anyone who has access to the image can obtain the secrets it holds. Bear in mind that the set of people who can access the image may not be the same set of people who need your production credentials.

- If you want to change a secret value, you need to rebuild the image. This can imply downtime: for example, if you change database credentials, your application code can't access the database until it is rebuilt and redeployed.
- Anything that is built into the image is likely under source code control, and unfortunately it's all too common to see secret information made publicly available through GitHub and similar tools. Even if your repos are private, consider the possibility that an authorized user forks your repo and makes it public.

Passing Secrets as Environment Variables

The Twelve-Factor App manifesto taught us to pass configuration information into applications as environment variables. This allows us to separate configuration from code, which is helpful when you need to run the same code in different scenarios (production, test, your own laptop…).

You can pass environment variables into containers at runtime. This means you can take the container image (code) and configure it according to the scenario it is running in. In Kubernetes, "ordinary" environment variables can be specified directly in the pod YAML or via a ConfigMap. But be careful: including secret values in YAML files that you check into code control means those secrets are accessible to the same people who can see the source code.

To mitigate this, Kubernetes also supports the secret resource, which can be passed into a pod as an environment variable. Your pod spec or ConfigMap YAML refers to the secret resource by name rather than directly to the secret value, so that it's safe to put that YAML under code control.

However, it's easy to "leak" information from environment variables to places you might not have considered. Let's have a look at three cases:

Case 1

It's common for a process to log out its entire environment in the event of a crash. This may be written to file, or in many deployments it will make its way to a centralized log aggregation system. Should the people who have access to this system also have access to your production database credentials?

Case 2

Take a look at the results from `kubectl describe pod <example>` and you will find the environment variables are available in plain text, such as shown in the following:

```
$ kubectl describe pod nginx-env-6c5b7b8ddd-dwpvk
Name:           nginx-env-6c5b7b8ddd-dwpvk
...
Containers:
  nginx-env:
  ...
    Environment:
      NOT_SO_SECRET:  some_value
  ...
```

You may have people who are allowed to inspect running pods in your cluster who again don't need access to the most privileged of credentials in your system.

Case 3

If you are using Docker as your container runtime, the environment is accessible using `docker inspect <container>`, running on the host as shown here:

```
$ sudo docker inspect b5ad78e251a3
[
  {
   "Id": "b5ad78e251a3f94c10b9336ccfe88e576548b4f387f5c7040...",
   ...
   "Config": {
      "Hostname": "nginx-env-6c5b7b8ddd-dwpvk",
     ...
     "Env": [
      "NOT_SO_SECRET=some_value",
      ...
    ]
   }
  }
]
```

The fact that environment variables are accessible via logs or via the command line to a broader set of people than might need access to secret credentials can be considered a security risk. You should contemplate whether this is an issue in your application and in your organization.

Some commercial solutions inject secrets into the environment by using a proprietary technique that means that the secret is not avail-

able through the command line via `kubectl describe` or `docker inspect`. However, this approach still doesn't protect against leaks through dumping the environment to a file or to a log.

You can restrict access via `kubectl` by using RBAC (see Chapter 4) so that only a subset of users can access specific pods.

Passing Secrets in Files

The last mechanism for passing information into a container is via a volume mounted into the container, where secret values are written into files on that volume. Kubernetes supports passing secrets into pods through volume mounts. The containerized code needs to read values out of these files.

If the mounted volume is a temporary filesystem, so much the better. This means the files are not written to disk, but held in memory, thus helping our aim of never storing secrets in plain text at rest.

The values held in the file are not accessible via `docker inspect` or `kubectl describe`. It's possible that the application code might write these secrets to somewhere undesirable, but this is much less likely than the inadvertent inclusion of secrets as part of a dump of environment variable values.

Secret Rotation and Revocation

For humans, it is no longer considered advisable to require frequent password changes, but this advice doesn't apply to the secrets used by machines. The longer a given secret remains valid, the more likely that it has been compromised. By regularly changing, or "rotating," secret values, we can ensure that a secret stops being of any use to an attacker.

For both the human- and machine-readable credentials related to your applications, you should have a mechanism in place that allows you to revoke a value if you discover it has been compromised. If you have a regular secret rotation process in place, you can have confidence that you can revoke or change a secret in the event of an emergency without harmful effects on the system.

Depending on how your application code is written, you may need to restart a pod in order for a new secret value to take effect. For example, an application might read a database password (from a file

or from an environment variable) just once as part of its initialization; it would need to be restarted in order to read the new value when it changes.

A different approach in the application might be to reread the secret value, perhaps on a regular basis, or in response to a failure using the currently held value. If the application code can cope with a secret being updated, this leads us to a benefit of the file-based approach to passing secrets: Kubernetes can update the secret value written to file without having to restart the pod. If you're using the environment variable mechanism in Kubernetes to pass secrets, they can't be updated live without a pod restart (although commercial solutions provide this capability).

Secret Access from Within the Container

If attackers gain execution access to a container, there is a high likelihood they will have access to the secrets within that container. This includes access via `kubectl exec` and `docker exec`.

In this situation, runtime protection can help (see "Sandboxing and Runtime Protection" on page 69). The fewer tools that the attacker can run inside the container, the better. For example, if the secrets are held in a file, preventing the attacker from being able to run commands like `cat`, `more`, or `less` make it harder to reach the secrets.

You can also limit the attacker's ability to access secrets by not building those commands into the container image in the first place. See Chapter 5 for more on reducing the attack surface in an image.

Secret Access from a Kubelet

Prior to Kubernetes 1.7, any kubelet could access any secret, but nowadays node authorization ensures that a kubelet can access only the secrets related to those pods that are scheduled to its node. If a node is compromised, this limits the effect of secrets access. You can ensure that this is in use by confirming that `--enable-admission-plugins` includes `NodeRestriction`.

To learn more about secrets and how to use them, check out the resources on the accompanying website, in the "Secrets" section.

So far, we have discussed how to set up your Kubernetes cluster with security in mind, and approaches for building and running application code safely. We now move on to discuss advanced security features that might apply, depending on your particular needs, and some new projects and capabilities that are under development at the time of writing.

CHAPTER 8

Advanced Topics

This chapter covers a collection of crosscutting topics related to making your Kubernetes cluster and its applications more secure. We'll build on the topics discussed in the previous chapters and sometimes go beyond Kubernetes proper (for example, with monitoring or service meshes).

Many of the ideas in this chapter are evolving and under discussion within the Kubernetes community. We welcome involvement from end users as well as those contributing to the development of cloud native projects themselves. If you're not already involved, there is a list of different ways to get involved; the Community section of the Kubernetes website provides a list of ways to get involved, from mailing lists and Slack channels to in-person events.

Monitoring, Alerting, and Auditing

The community seems to be standardizing on Prometheus for monitoring Kubernetes clusters, so a good start is to familiarize yourself with it. Since there are so many moving parts (from nodes to pods to services), alerting on each event is not practical. What you can do, however, is think about who needs to be informed about what kind of event. For example, a policy could be that node-related or namespace-related events are handled by admins, and developers are paged for pod-level events. The same applies more or less for

logs, but here you also should be aware of where and when your sensitive data lands on disk; see Chapter 7 for details.

Another useful feature Kubernetes offers via the API server is auditing, effectively recording the sequence of activities affecting the cluster. Different strategies are available in the auditing policy (from no logging to logging event metadata, request and response bodies), and you can choose between a simple log backend as well as using a webhook for integrating with third-party systems.

Host Security

Much discussion of Kubernetes security focuses on the setup of Kubernetes itself, or on securing the containerized applications that run within it. There is another layer to be secured: the host machines on which Kubernetes runs.

Whether they are bare-metal or VMs, there are many steps you can take to restrict access to your hosts that are essentially the same as you would take in a traditional cluster. For example, you should, of course, restrict access to the machines, and you might well configure them to use a dedicated VPN. These general machine security measures are outside the scope of this book, but there are some specific things you would do well to consider when you are running Kubernetes (or any container orchestrator). Resources such as OpenSCAP and OVAL can help with a broader security assessment.

Host Operating System

The host machines need to be able to run only the Kubernetes code, its dependencies (a runtime system like Docker), and supporting features like logging or security tools. Following the principle of reducing the attack surface, a best-practice setup would have *only* the necessary code and no superfluous libraries or binaries. This is sometimes referred to as a *thin OS*.

Container-specific distributions like Container Linux from CoreOS (now part of Red Hat), RancherOS, or Red Hat's own Atomic are one approach to minimizing the amount of code installed on your host machines. These can also include other security features like a read-only root filesystem. A general-purpose Linux distribution is also fine, but it's worth checking that you're not using a machine image with extra libraries and tools installed (particularly if, out of

habit, you are using the same machine image you have used in a traditional deployment).

Node Recycling

In a cloud-native deployment, we treat nodes as "cattle not pets," and it should be trivially easy to create a new node or replace a faulty one, as it should be automated through an infrastructure as code approach. This enables node recycling, where you tear down and replace your nodes on a regular (or random) schedule.

When you recycle a node, you know that it has been returned to the desired state, as determined by your infrastructure as code. If there has been any "drift"—for example, because an undetected attacker got a foothold into the system—that drift is removed.

Another benefit of recycling your nodes (especially for small or new deployments) is that it's effectively a fire drill. If you are frequently replacing nodes as a matter of course, you can have more confidence in the system's ability to cope through node failure. It's a baby step toward chaos engineering!

Sandboxing and Runtime Protection

Sandboxing is the ability to isolate containers from each other and from the underlying host, so that code running within one container can't effect change outside that container.

Runtime protection is the concept of limiting the set of code that can be executed within the container itself. If attackers can access a container, but can't execute their own code within it, the potential damage is limited.

While the end goal of these two concepts is different, some overlap exists in the mechanisms used to achieve them. For example, seccomp or AppArmor profiles can limit a container to have access to a limited set of system calls. This restricts what the container can do (runtime protection) and increases its isolation by giving it less access to the kernel (sandboxing).

Containers share the host's kernel, so vulnerabilities in the kernel could conceivably allow an exploit to escape one container and move to another or to the host.

At time of writing, several projects and vendors are aimed at improving sandboxing and/or runtime protection, all with their own strengths and characteristics:

- seccomp is a kernel mechanism for limiting the system calls that application code can make. A `securityContext` or `PodSecurity Policy` can specify the seccomp profile.
- AppArmor and SELinux are kernel security modules, also configurable through profiles attached to `securityContext` or `Pod SecurityPolicy` in Kubernetes.
- Kata Containers run each application inside a "lightweight" VM (so each has its own kernel).
- Google's gVisor project runs container code in a sandbox through a kernel API implemented in user space (if that's not a contradiction in terms).
- Nabla containers use unikernel technology to provide isolation and limit access to the shared kernel.
- Enterprise container security solutions such as those from Aqua Security and Twistlock use regular containers, with proprietary runtime protection technology that includes whitelisting/blacklisting the set of executables that can run in a given container.

Multitenancy

In some deployments, multiple "tenants" using the same cluster don't fully trust each other or might not be trusted by the cluster operator. Tenants could be users, groups of users, or applications. The level of trust between users depends on the environment; for example, in a platform-as-a-service (PaaS) environment where users can upload and run their own applications, they should be treated as entirely untrusted, whereas in an enterprise, the tenants might map to different organizational teams, who do cooperate and trust each other to some extent, but who want to limit the risk of someone outside one team maliciously or inadvertently affecting another team's application.

Multitenant isolation has two parts:

- The control plane, so that users can't, for example, run `kubectl` commands that impact other tenants' resources

- The runtime and networking environment, where container workloads should not be able to interfere with each other or steal resources

The Kubernetes namespace gives us the first building block for multitenancy in the control plane. Typically, there will be one namespace per tenant. Users are given RBAC permissions to create, update, and delete resources within the namespace that maps to their tenancy (see Chapter 3 and Chapter 4). Resource quotas allow each namespace (and thereby, each tenant) to be restricted to a limit of compute and storage resources, and of Kubernetes objects (for example, upper bounds on the number of services, secrets, and persistent volume claims allowed within the namespace).

In an enterprise environment, this may be sufficient. Namespace-based RBAC controls mean that one team can't update application code and associated Kubernetes resources that they are not responsible for. Quotas mean that one team's application can't use all available resources so that another is starved.

However, this is unlikely to be sufficient protection in a fully untrusted environment. Here, tenant workloads should be isolated from each other at a container level so that if there were to be an escape from one container (perhaps because a user deploys code with a serious vulnerability, perhaps even deliberately), they can't affect or inspect other tenants' applications or data. Container sandboxing, as described in "Sandboxing and Runtime Protection" on page 69, is largely designed to solve this problem (for example, the gVisor approach is based on the way Google isolates user workloads from each other in Google App Engine).

Another approach to workload isolation is to assign one or more nodes to each tenant, and then schedule pods so that workloads are only ever colocated with other workloads from the same tenant. This is straightforward (using taints and tolerations), but potentially wasteful unless each tenant needs a node's worth of compute resources.

In an untrusted multitenant environment, you would want strong network policies to isolate traffic so that it can't flow between namespaces. See "Network Policies" on page 52 for more information.

Dynamic Admission Control

From Kubernetes 1.9, dynamic admission controllers allow for flexible, extensible mechanisms for making checks before allowing a resource to be deployed to a cluster. As an example, check out Kelsey Hightower's Grafeas tutorial, which includes a validating webhook that ensures that only signed images are admitted.

Network Protection

In a typical traditional deployment, a significant proportion of the security measures is network based: firewalling and the use of VPNs come immediately to mind. In the cloud-native world, similar approaches are dedicated to restricting traffic so that only approved flows can take place.

Security solutions for containers have for some years talked about network micro- or nano-segmentation, and these approaches have been made fairly common practice in Kubernetes deployments through the use of network policies (as discussed in "Network Policies" on page 52).

At the time of writing, service meshes are a popular topic—although, in our opinion, currently at the stage of "early adopter" rather than "early majority" market penetration.

Service Meshes

The idea of a service mesh like Istio or Linkerd is to take on much of the burden of networking communication and control, so that application developers don't need to concern themselves with these nonfunctional capabilities. While they offer several features like load balancing and routing that are outside the scope of this book, they offer two features that are of particular interest in relation to security: mutually authenticated TLS connections and service mesh network policies.

Mutually authenticated TLS connections

The service mesh intercepts network traffic to and from a pod, and ensures that connections are all set up using TLS between authenticated components. This automatically ensures that all communica-

tions are encrypted. Even if attackers find their way into your cluster, they will struggle to intercept the network traffic within it.

Service mesh network policy

The service mesh can control which services can communicate with each other, adding another layer of protection that makes it harder for an attacker to move within the cluster.

As discussed in "Network Policies" on page 52, Kubernetes network policies define what traffic is allowed to and from a group of pods, so you may well be wondering how Kubernetes and service mesh network policies interact.

Kubernetes network policy acts at the networking level, based on IP addresses and ports as well as pods (identified by label), whereas service mesh policy acts at the service level. There is a good discussion of this in a series of blog posts from Project Calico.

Static Analysis of YAML

In many organizations, the YAML associated with an application could be written by a developer who may not have intimate knowledge of the security policies that the organization requires (or who may simply make a mistake). Static analysis tools such as `kubetest` and `kubesec` can be useful to look for issues in YAML configuration files (for example, checking for the use of the `privileged` flag for a pod), or to enforce a particular labeling policy. Just like image scanning (see "Scanning Container Images" on page 36), this is an example of "shift-left," where security practices are dealt with and enforced earlier in the development lifecycle.

Fork Bombs and Resource-Based Attacks

A *fork bomb* is a process that continually launches copies of itself, with the intention of using all the available resources, effectively creating a denial-of-service attack. Kubernetes addresses this by allowing you to configure a limit on the number of processes within a pod. At the time of writing, this is an Alpha feature that may be subject to significant changes in the future.

Other resource-based attacks might involve trying to consume excessive memory and CPU, denying those resources to legitimate

workloads. You can set resource limits to limit exposure to this kind of attack.

Cryptocurrency Mining

A famous attack on Tesla exploited control-plane insecurities to allow hackers to use the company's resources to mine cryptocurrency. Following the advice in Chapter 2 will go a long way to preventing this from happening in your cluster.

Additional research shows other approaches that would-be miners are attempting. Ensuring that only trusted images can run in your cluster would prevent, for example, a bad actor with approved access to the cluster from running an unexpected mining image. See Chapter 5 for advice on preventing untrusted or compromised images from running.

Runtime protection can add another layer of defense to ensure that even if an approved image has a vulnerability that allows code to be injected into a running container, that code can't be executed.

Monitoring for unusual activity, such as unexpected CPU usage and unexpected resources being scaled out, can help spot when your resources are being used by an attacker. See "Monitoring, Alerting, and Auditing" on page 67.

Kubernetes Security Updates

From time to time, security issues in Kubernetes itself are unearthed, and the project has a security process for dealing with these. The project documentation includes instructions for reporting a vulnerability in Kubernetes to the security team.

If you want to be alerted as soon as vulnerabilities in Kubernetes are announced, subscribe to the kubernetes-announce mailing list.

To learn more about the topics discussed in this chapter, check out the resources on the accompanying website, in the "Advanced Topics" section.

We've reached the conclusion of the book and want to thank you for sticking with us until the very end. Enjoy Kubernetes in production—and stay safe!

About the Authors

Liz Rice is the Technology Evangelist with container security specialist Aqua Security, where she also works on container-related open source projects including kube-bench and kube-hunter. She is cochair of the CNCF's KubeCon + CloudNativeCon events and has a wealth of software development, team, and product management experience from working on network protocols and distributed systems, and in digital technology sectors such as VOD, music, and VoIP. When not writing code or talking about it, Liz loves riding bikes in places with better weather than her native London, and competing in virtual races on Zwift.

Michael Hausenblas is a developer advocate for Kubernetes and OpenShift at Red Hat where he helps appops to build and operate apps. His background is in large-scale data processing and container orchestration and he's experienced in advocacy and standardization at W3C and IETF. Before Red Hat, Michael worked at Mesosphere, MapR, and in two research institutions in Ireland and Austria. He contributes to open source software including Kubernetes, speaks at conferences and user groups, and shares good practices around cloud native topics via blog posts and books.