O'REILLY®

Certified Kubernetes Security Specialist (CKS) Study Guide

In-Depth Guidance and Practice

Free Chapter

Benjamin Muschko

Certified Kubernetes Security Specialist (CKS) Study Guide

In-Depth Guidance and Practice

This excerpt contains Chapter 2. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Benjamin Muschko

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Certified Kubernetes Security Specialist (CKS) Study Guide
by Benjamin Muschko

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 (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Liz Wheeler
Proofreader: Amnet Systems, LLC

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the First Edition
2023-06-08: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098132972* for release details.

978-1-098-13297-2

[LSI]

Table of Contents

CHAPTER 2

Cluster Setup

The first domain of the exam deals with concerns related to Kubernetes cluster setup and configuration. In this chapter, we'll only drill into the security-specific aspects and not the standard responsibilities of a Kubernetes administrator.

At a high level, this chapter covers the following concepts:

- Using network policies to restrict Pod-to-Pod communication
- Running CIS benchmark tooling to identify security risks for cluster components
- Setting up an Ingress object with TLS support
- Protecting node ports, API endpoints, and GUI access
- Verifying platform binaries against their checksums

Using Network Policies to Restrict Pod-to-Pod Communication

For a microservice architecture to function in Kubernetes, a Pod needs to be able to reach another Pod running on the same or on a different node without Network Address Translation (NAT). Kubernetes assigns a unique IP address to every Pod upon creation from the Pod CIDR range of its node. The IP address is ephemeral and therefore cannot be considered stable over time. Every restart of a Pod leases a new IP address. It's recommended to use Pod-to-Service communication over Pod-to-Pod communication so that you can rely on a consistent network interface.

The IP address assigned to a Pod is unique across all nodes and namespaces. This is achieved by assigning a dedicated subnet to each node when registering it. When creating a new Pod on a node, the IP address is leased from the assigned subnet. This

is handled by the Container Network Interface (CNI) plugin. As a result, Pods on a node can communicate with all other Pods running on any other node of the cluster.

Network policies act similarly to firewall rules, but for Pod-to-Pod communication. Rules can include the direction of network traffic (ingress and/or egress) for one or many Pods within a namespace or across different namespaces, as well as their targeted ports. For a deep-dive coverage on the basics of network policies, refer to the book *Certified Kubernetes Application Developer (CKAD) Study Guide* (O'Reilly) or the Kubernetes documentation. The CKS exam primarily focuses on restricting cluster-level access with network policies.

Defining the rules of network policies correctly can be challenging. The page networkpolicy.io provides a visual editor for network policies that renders a graphical representation in the browser.

Scenario: Attacker Gains Access to a Pod

Say you are working for a company that operates a Kubernetes cluster with three worker nodes. Worker node 1 currently runs two Pods as part of a microservices architecture. Given Kubernetes default behavior for Pod-to-Pod network communication, Pod 1 can talk to Pod 2 unrestrictedly and vice versa.

As you can see in Figure 2-1, an attacker gained access to Pod 1. Without defining network policies, the attacker can simply talk to Pod 2 and cause additional damage. This vulnerability isn't restricted to a single namespace. Pods 3 and 4 can be reached and compromised as well.

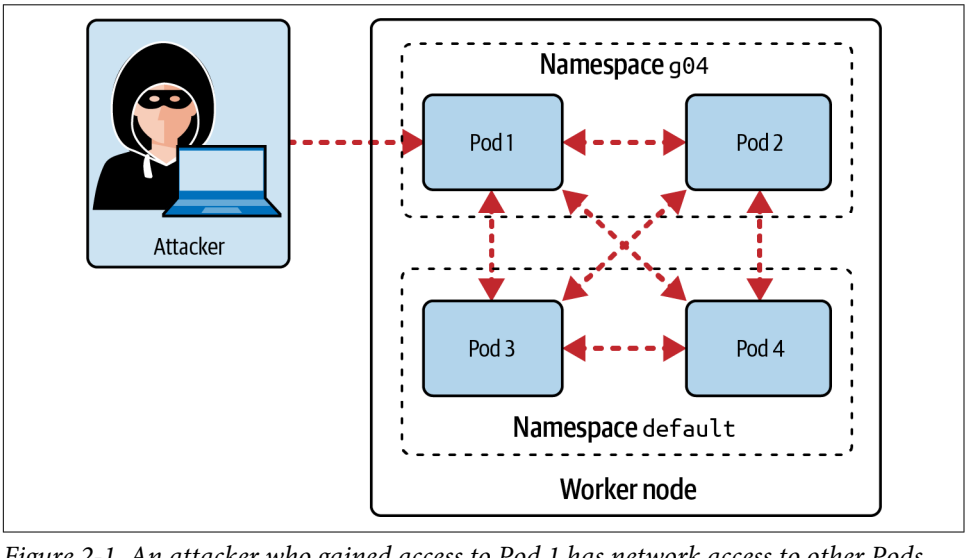

Figure 2-1. An attacker who gained access to Pod 1 has network access to other Pods

Observing the Default Behavior

We'll set up three Pods to demonstrate the unrestricted Pod-to-Pod network communication in practice. As you can see in Example 2-1, the YAML manifest defines the Pods named `backend` and `frontend` in the namespace `g04`. The `other` Pod lives in the `default` namespace. Observe the label assignment for the namespace and Pods. We will reference them a little bit later in this chapter when defining network policies.

Example 2-1. YAML manifest for three Pods in different namespaces

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    app: orion
  name: g04
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    tier: backend
  name: backend
  namespace: g04
spec:
  containers:
  - image: bmuschko/nodejs-hello-world:1.0.0
    name: hello
    ports:
    - containerPort: 3000
  restartPolicy: Never
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    tier: frontend
  name: frontend
  namespace: g04
spec:
  containers:
  - image: alpine
    name: frontend
    args:
    - /bin/sh
    - -c
    - while true; do sleep 5; done;
  restartPolicy: Never
---
apiVersion: v1
```

```
kind: Pod
metadata:
  labels:
    tier: outside
  name: other
spec:
  containers:
  - image: alpine
    name: other
    args:
    - /bin/sh
    - -c
    - while true; do sleep 5; done;
  restartPolicy: Never
```

Start by creating the objects from the existing YAML manifest using the declarative `kubectl apply` command:

```
$ kubectl apply -f setup.yaml
namespace/g04 created
pod/backend created
pod/frontend created
pod/other created
```

Let's verify that the namespace `g04` runs the correct Pods. Use the `-o wide` CLI option to determine the virtual IP addresses assigned to the Pods. The `backend` Pod uses the IP address 10.0.0.43, and the `frontend` Pod uses the IP address 10.0.0.193:

```
$ kubectl get pods -n g04 -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP           NODE     \
  NOMINATED NODE   READINESS GATES
backend    1/1     Running   0          15s   10.0.0.43    minikube \
  <none>           <none>
frontend   1/1     Running   0          15s   10.0.0.193   minikube \
  <none>           <none>
```

The `default` namespace handles a single Pod:

```
$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
other   1/1     Running   0          4h45m
```

The `frontend` Pod can talk to the `backend` Pod as no communication restrictions have been put in place:

```
$ kubectl exec frontend -it -n g04 -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
remote file exists
/ # exit
```

The `other` Pod residing in the `default` namespace can communicate with the `backend` Pod without problems:

```
$ kubectl exec other -it -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
remote file exists
/ # exit
```

In the next section, we'll talk about restricting Pod-to-Pod network communication to a maximum level with the help of deny-all network policy rules. We'll then open up ingress and/or egress communication only for the kind of network communication required for the microservices architecture to function properly.

Denying Directional Network Traffic

The best way to restrict Pod-to-Pod network traffic is with the principle of least privilege. Least privilege means that Pods should communicate with the lowest privilege for network communication. You'd usually start by disallowing traffic in any direction and then opening up the traffic needed by the application architecture.

The Kubernetes documentation provides a couple of helpful YAML manifest examples. Example 2-2 shows a network policy that denies ingress traffic to all Pods in the namespace `g04`.

Example 2-2. A default deny-all ingress network policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: g04
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Selecting all Pods is denoted by the value `{}` assigned to the `spec.podSelector` attribute. The value attribute `spec.policyTypes` defines the denied direction of traffic. For incoming traffic, you can add `Ingress` to the array. Outgoing traffic can be specified by the value `Egress`. In this particular example, we disallow all ingress traffic. Egress traffic is still permitted.

The contents of the "deny-all" network policy have been saved in the file `deny-all-ingress-network-policy.yaml`. The following command creates the object from the file:

```
$ kubectl apply -f deny-all-ingress-network-policy.yaml
networkpolicy.networking.k8s.io/default-deny-ingress created
```

Let's see how this changed the runtime behavior for Pod-to-Pod network communication. The `frontend` Pod cannot talk to the `backend` Pod anymore, as observed by running the same `wget` command we used earlier. The network call times out after one second, as defined by the CLI option `--timeout`:

```
$ kubectl exec frontend -it -n g04 -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
wget: download timed out
/ # exit
```

Furthermore, Pods running in a different namespace cannot connect to the `backend` Pod anymore either. The following `wget` command makes a call from the `other` Pod running in the `default` namespace to the IP address of the `backend` Pod:

```
$ kubectl exec other -it -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
wget: download timed out
```

This call times out as well.

Allowing Fine-Grained Incoming Traffic

Network policies are additive. To grant more permissions for network communication, simply create another network policy with more fine-grained rules. Say we wanted to allow ingress traffic to the `backend` Pod only from the `frontend` Pod that lives in the same namespace. Ingress traffic from all other Pods should be denied independently of the namespace they are running in.

Network policies heavily work with label selection to define rules. Identify the labels of the `g04` namespace and the Pod objects running in the same namespace so we can use them in the network policy:

```
$ kubectl get ns g04 --show-labels
NAME   STATUS   AGE   LABELS
g04    Active   12m   app=orion,kubernetes.io/metadata.name=g04
$ kubectl get pods -n g04 --show-labels
NAME       READY   STATUS    RESTARTS   AGE     LABELS
backend    1/1     Running   0          9m46s   tier=backend
frontend   1/1     Running   0          9m46s   tier=frontend
```

The label assignment for the namespace `g04` includes the key-value pair `app=orion`. The Pod `backend` label set includes the key-value pair `tier=backend`, and the `front end` Pod the key-value pair `tier=frontend`.

Create a new network policy that allows the `frontend` Pod to talk to the `backend` Pod only on port 3000. No other communication should be allowed. The YAML manifest representation in Example 2-3 shows the full network policy definition.

Example 2-3. Network policy that allows ingress traffic

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-ingress
  namespace: g04
spec:
  podSelector:
    matchLabels:
      tier: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          app: orion
      podSelector:
        matchLabels:
          tier: frontend
    ports:
    - protocol: TCP
      port: 3000
```

The definition of the network policy has been stored in the file `backend-ingress-network-policy.yaml`. Create the object from the file:

```
$ kubectl apply -f backend-ingress-network-policy.yaml
networkpolicy.networking.k8s.io/backend-ingress created
```

The `frontend` Pod can now talk to the `backend` Pod:

```
$ kubectl exec frontend -it -n g04 -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
remote file exists
/ # exit
```

Pods running outside of the `g04` namespace still can't connect to the `backend` Pod. The `wget` command times out:

```
$ kubectl exec other -it -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.43:3000
Connecting to 10.0.0.43:3000 (10.0.0.43:3000)
wget: download timed out
```

Applying Kubernetes Component Security Best Practices

Managing an on-premises Kubernetes cluster gives you full control over the configuration options applied to cluster components, such as the API server, etcd, the kubelet, and others. It's not uncommon to simply go with the default configuration settings used by `kubeadm` when creating the cluster nodes. Some of those default settings may expose cluster components to unnecessary attack opportunities.

Hardening the security measures of a cluster is a crucial activity for any Kubernetes administrator seeking to minimize attack vectors. You can either perform this activity manually if you are aware of the best practices, or use an automated process.

The Center for Internet Security (CIS) is a not-for-profit organization that publishes cybersecurity best practices. Part of their best practices portfolio is the Kubernetes CIS Benchmark, a catalog of best practices for Kubernetes environments. You will find a detailed list of recommended security settings for cluster components on their web page.

CIS benchmarking for cloud provider Kubernetes environments

The Kubernetes CIS Benchmark is geared toward a self-managed installation of Kubernetes. Cloud provider Kubernetes environments, such as Amazon Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE), provide a managed control plane accompanied by their own command line tools. Therefore, the security recommendations made by the Kubernetes CIS Benchmark may be less fitting. Some tools, like kube-bench, discussed next, provide verification checks specifically for cloud providers.

Using kube-bench

You can use the tool kube-bench to check Kubernetes cluster components against the CIS Benchmark best practices in an automated fashion. Kube-bench can be executed in a variety of ways. For example, you can install it as a platform-specific binary in the form of an RPM or Debian file. The most convenient and direct way to run the verification process is by running kube-bench in a Pod directly on the Kubernetes cluster. For that purpose, create a Job object with the help of a YAML manifest checked into the GitHub repository of the tool.

Start by creating the Job from the file `job-master.yaml`, or `job-node.yaml` depending on whether you want to inspect a control plane node or a worker node. The following command runs the verification checks against the control plane node:

```
$ kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-bench/\
main/job-master.yaml
job.batch/kube-bench-master created
```

Upon Job execution, the corresponding Pod running the verification process can be identified by its name in the `default` namespace. The Pod's name starts with the prefix `kube-bench`, then appended with the type of the node plus a hash at the end. The following output uses the Pod named `kube-bench-master-8f6qh`:

```
$ kubectl get pods
NAME                      READY   STATUS      RESTARTS   AGE
kube-bench-master-8f6qh   0/1     Completed   0          45s
```

Wait until the Pod transitions into the "Completed" status to ensure that all verification checks have finished. You can have a look at the benchmark result by dumping the logs of the Pod:

```
$ kubectl logs kube-bench-master-8f6qh
```

Sometimes, it may be more convenient to write the verification results to a file. You can redirect the output of the `kubectl logs` command to a file, e.g., with the command `kubectl logs kube-bench-master-8f6qh > control-plane-kube-bench-results.txt`.

The kube-bench Verification Result

The produced verification result can be lengthy and detailed, but it consists of these key elements: the type of the inspected node, the inspected components, a list of passed checks, a list of failed checks, a list of warnings, and a high-level summary:

```
[INFO] 1 Control Plane Security Configuration ❶
[INFO] 1.1 Control Plane Node Configuration Files
[PASS] 1.1.1 Ensure that the API server pod specification file permissions are \
set to 644 or more restrictive (Automated) ❷
...
[INFO] 1.2 API Server
[WARN] 1.2.1 Ensure that the --anonymous-auth argument is set to false \
(Manual) ❸
...
[FAIL] 1.2.6 Ensure that the --kubelet-certificate-authority argument is set \
as appropriate (Automated) ❹

== Remediations master ==
...
1.2.1 Edit the API server pod specification file /etc/kubernetes/manifests/ \
kube-apiserver.yaml on the control plane node and set the below parameter.
--anonymous-auth=false
```

```
...
1.2.6 Follow the Kubernetes documentation and setup the TLS connection between ❺
the apiserver and kubelets. Then, edit the API server pod specification file ❺
/etc/kubernetes/manifests/kube-apiserver.yaml on the control plane node and \ ❺
set the --kubelet-certificate-authority parameter to the path to the cert \ ❺
file for the certificate authority. ❺
--kubelet-certificate-authority=<ca-string> ❺

...
== Summary total == ❻
42 checks PASS
9 checks FAIL
11 checks WARN
0 checks INFO
```

❶ The inspected node, in this case the control plane node.

❷ A passed check. Here, the file permissions of the API server configuration file.

❸ A warning message that prompts you to manually check the value of an argument provided to the API server executable.

❹ A failed check. For example, the flag `--kubelet-certificate-authority` should be set for the API server executable.

❺ The remediation action to take to fix a problem. The number, e.g., 1.2.1, of the failure or warning corresponds to the number assigned to the remediation action.

❻ The summary of all passed and failed checks plus warning and informational messages.

Fixing Detected Security Issues

The list of reported warnings and failures can be a bit overwhelming at first. Keep in mind that you do not have to fix them all at once. Some checks are merely guidelines or prompts to verify an assigned value for a configuration. The following steps walk you through the process of eliminating a warning message.

The configuration files of the control plane components can be found in the directory `/etc/kubernetes/manifests` on the host system of the control plane node. Say you wanted to fix the warning 1.2.12 reported by kube-bench:

```
[INFO] 1.2 API Server
...
[WARN] 1.2.12 Ensure that the admission control plugin AlwaysPullImages is \
set (Manual)
```

```
== Remediations master ==
...
1.2.12 Edit the API server pod specification file /etc/kubernetes/manifests/ \
kube-apiserver.yaml
on the control plane node and set the --enable-admission-plugins parameter \
to include AlwaysPullImages.
--enable-admission-plugins=...,AlwaysPullImages,...
```

As proposed by the remediation action, you are supposed to edit the configuration file for the API server and add the value `AlwaysPullImages` to the list of admission plugins. Go ahead and edit the file `kube-apiserver.yaml`:

```
$ sudo vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

After appending the value `AlwaysPullImages` to the argument `--enable-admission-plugins`, the result could look as follows:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: \
    192.168.56.10:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.56.10
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction,AlwaysPullImages
...
```

Save the changes to the file. The Pod running the API server in the `kube-system` namespace will be restarted automatically. The startup process can take a couple of seconds. Therefore, executing the following command may take a while to succeed:

```
$ kubectl get pods -n kube-system
NAME                           READY   STATUS    RESTARTS   AGE
...
kube-apiserver-control-plane   1/1     Running   0          71m
...
```

You will need to delete the existing Job object before you can verify the changed result:

```
$ kubectl delete job kube-bench-master
job.batch "kube-bench-master" deleted
```

The verification check 1.2.12 now reports a passed result:

```
$ kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-bench/\
main/job-master.yaml
job.batch/kube-bench-master created
$ kubectl get pods
NAME                      READY   STATUS      RESTARTS   AGE
kube-bench-master-5gjdn   0/1     Completed   0          10s
$ kubectl logs kube-bench-master-5gjdn | grep 1.2.12
[PASS] 1.2.12 Ensure that the admission control plugin AlwaysPullImages is \
set (Manual)
```

Creating an Ingress with TLS Termination

An Ingress routes HTTP and/or HTTPS traffic from outside of the cluster to one or many Services based on a matching URL context path. You can see its functionality in action in Figure 2-2.

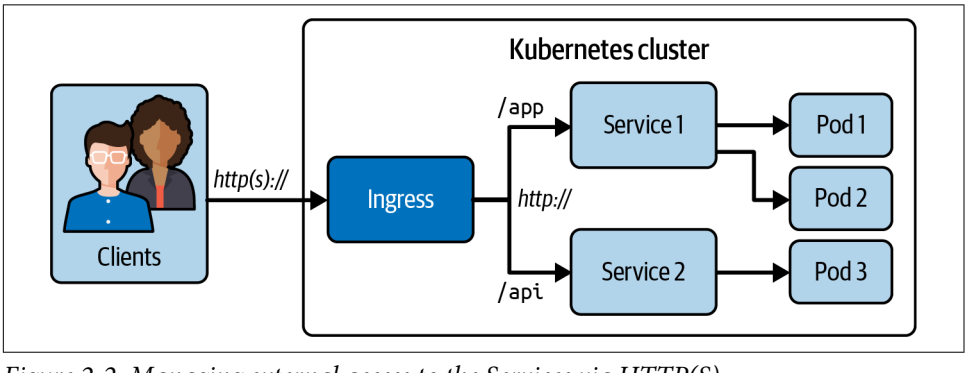

Figure 2-2. Managing external access to the Services via HTTP(S)

The Ingress has been configured to accept HTTP and HTTPS traffic from outside of the cluster. If the caller provides the context path `/app`, then the traffic is routed to Service 1. If the caller provides the context path `/api`, then the traffic is routed to Service 2. It's important to point out that the communication typically uses unencrypted HTTP network communication as soon as it passes the Ingress.

Given that the Ingress API resource is a part of the CKAD and CKA exam, we are not going to discuss the basics anymore here. For a detailed discussion, refer to the information in the *Certified Kubernetes Administrator (CKA) Study Guide* or the Kubernetes documentation.

The role of an Ingress controller

Remember that an Ingress cannot work without an Ingress controller. The Ingress controller evaluates the collection of rules defined by an Ingress that determine traffic routing. One example of a production-grade Ingress controller is the F5 NGINX Ingress Controller or AKS Application Gateway Ingress Controller. You can find other options listed in the Kubernetes documentation. If you are using minikube, make sure to enable the Ingress add-on.

The primary focus of the CKS lies on setting up Ingress objects with TLS termination. Configuring the Ingress for HTTPS communication relieves you from having to deal with securing the network communication on the Service level. In this section of the book, you will learn how to create a TLS certificate and key, how to feed the certificate and key to a TLS-typed Secret object, and how to configure an Ingress object so that it supports HTTPS communication.

Setting Up the Ingress Backend

In the context of an Ingress, a *backend* is the combination of Service name and port. Before creating the Ingress, we'll take care of the Service, a Deployment, and the Pods running nginx so we can later on demonstrate the routing of HTTPS traffic to an actual application. All of those objects are supposed to exist in the namespace `t75`. Example 2-4 defines all of those resources in a single YAML manifest file `setup.yaml` as a means to quickly create the Ingress backend.

Example 2-4. YAML manifest for exposing nginx through a Service

```
apiVersion: v1
kind: Namespace
metadata:
  name: t75
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: t75
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
```

```
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: accounting-service
  namespace: t75
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Create the objects from the YAML file with the following command:

```
$ kubectl apply -f setup.yaml
namespace/t75 created
deployment.apps/nginx-deployment created
service/accounting-service created
```

Let's quickly verify that the objects have been created properly, and the Pods have transitioned into the "Running" status. Upon executing the `get all` command, you should see a Deployment named `nginx-deployment` that controls three replicas, and a Service named `accounting-service` of type `ClusterIP`:

```
$ kubectl get all -n t75
NAME                                    READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-6595874d85-5rdrh   1/1     Running   0          108s
pod/nginx-deployment-6595874d85-jmhvh   1/1     Running   0          108s
pod/nginx-deployment-6595874d85-vtwxp   1/1     Running   0          108s

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S) \
  AGE
service/accounting-service   ClusterIP   10.97.101.228   <none>        80/TCP \
  108s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment    3/3     3            3           108s
```

Calling the Service endpoint from another Pod running on the same node should result in a successful response from the nginx Pod. Here, we are using the `wget` command to verify the behavior:

```
$ kubectl run tmp --image=busybox --restart=Never -it --rm \
  -- wget 10.97.101.228:80
Connecting to 10.97.101.228:80 (10.97.101.228:80)
saving to 'index.html'
index.html           100% |**|   612  0:00:00 ETA
'index.html' saved
pod "tmp" deleted
```

With those objects in place and functioning as expected, we can now concentrate on creating an Ingress with TLS termination.

Creating the TLS Certificate and Key

We will need to generate a TLS certificate and key before we can create a TLS Secret. To do this, we will use the OpenSSL command. The resulting files are named `accounting.crt` and `accounting.key`:

```
$ openssl req -nodes -new -x509 -keyout accounting.key -out accounting.crt \
  -subj "/CN=accounting.tls"
Generating a 2048 bit RSA private key
.........................+
.........................+
writing new private key to 'accounting.key'
-----
$ ls
accounting.crt accounting.key
```

For use in production environments, you'd generate a key file and use it to obtain a TLS certificate from a certificate authority (CA). For more information on creating a TLS certification and key, see the OpenSSL documentation.

Creating the TLS-Typed Secret

The easiest way to create a Secret is with the help of an imperative command. This method of creation doesn't require you to manually base64-encode the certificate and key values. The encoding happens automatically upon object creation. The following command uses the Secret option `tls` and assigns the certificate and key file name with the options `--cert` and `--key`:

```
$ kubectl create secret tls accounting-secret --cert=accounting.crt \
  --key=accounting.key -n t75
secret/accounting-secret created
```

Example 2-5 shows the YAML representation of a TLS Secret if you want to create the object declaratively.

Example 2-5. A Secret using the type `kubernetes.io/tls`

```
apiVersion: v1
kind: Secret
metadata:
  name: accounting-secret
  namespace: t75
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk...
  tls.key: LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk...
```

Make sure to assign the values for the attributes `tls.crt` and `tls.key` as single-line, base64-encoded values. To produce the base64-encoded value, simply point the `base64` command to the file name you want to convert the contents for. The following example base64-encoded the contents of the file `accounting.crt`:

```
$ base64 accounting.crt
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNyakNDQ...
```

Creating the Ingress

You can use the imperative method to create the Ingress with the help of a one-liner command shown in the following snippet. Crafting the value of the `--rule` argument is hard to get right. You will likely have to refer to the `--help` option for the `create ingress` command as it requires a specific expression. The information relevant to creating the connection between Ingress object and the TLS Secret is the appended argument `tls=accounting-secret`:

```
$ kubectl create ingress accounting-ingress \
  --rule="accounting.internal.acme.com/*=accounting-service:80, \
  tls=accounting-secret" -n t75
ingress.networking.k8s.io/accounting-ingress created
```

Example 2-6 shows a YAML representation of an Ingress. The attribute for defining the TLS information is `spec.tls[]`.

Example 2-6. A YAML manifest for defning a TLS-terminated Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: accounting-ingress
  namespace: t75
spec:
  tls:
  - hosts:
    - accounting.internal.acme.com
    secretName: accounting-secret
```

```
rules:
- host: accounting.internal.acme.com
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: accounting-service
          port:
            number: 80
```

After creating the Ingress object with the imperative or declarative approach, you should be able to find it in the namespace `t75`. As you can see in the following output, the port 443 is listed in the "PORT" column, indicating that TLS termination has been enabled:

```
$ kubectl get ingress -n t75
NAME                CLASS   HOSTS                          ADDRESS       \
  PORTS     AGE
accounting-ingress   nginx   accounting.internal.acme.com   192.168.64.91 \
  80, 443   55s
```

Describing the Ingress object shows that the backend could be mapped to the path `/` and will route traffic to the Pod via the Service named `accounting-service`:

```
$ kubectl describe ingress accounting-ingress -n t75
Name:             accounting-ingress
Labels:           <none>
Namespace:        t75
Address:          192.168.64.91
Ingress Class:    nginx
Default backend:  <default>
TLS:
  accounting-secret terminates accounting.internal.acme.com
Rules:
  Host                          Path  Backends
  ----                          ----  --------
  accounting.internal.acme.com
                                /   accounting-service:80 \
                                (172.17.0.5:80,172.17.0.6:80,172.17.0.7:80)
Annotations:                    <none>
Events:
  Type    Reason  Age               From                      Message
  ----    ------  ----              ----                      -------
  Normal  Sync    1s (x2 over 31s)  nginx-ingress-controller  Scheduled for sync
```

Calling the Ingress

To test the behavior on a local Kubernetes cluster on your machine, you need to first find out the IP address of a node. The following command reveals the IP address in a minikube environment:

```
$ kubectl get nodes -o wide
NAME       STATUS   ROLES           AGE     VERSION   INTERNAL-IP   \
  EXTERNAL-IP   OS-IMAGE               KERNEL-VERSION   CONTAINER-RUNTIME
minikube   Ready    control-plane   3d19h   v1.24.1   192.168.64.91 \
  <none>        Buildroot 2021.02.12   5.10.57          docker://20.10.16
```

Next, you'll need to add the IP address to the hostname mapping to your `/etc/hosts` file:

```
$ sudo vim /etc/hosts
...
192.168.64.91   accounting.internal.acme.com
```

You can now send HTTPS requests to the Ingress using the assigned domain name and receive an HTTP response code 200 in return:

```
$ wget -O- https://accounting.internal.acme.com --no-check-certificate
--2022-07-28 15:32:43--  https://accounting.internal.acme.com/
Resolving accounting.internal.acme.com (accounting.internal.acme.com)... \
192.168.64.91
Connecting to accounting.internal.acme.com (accounting.internal.acme.com) \
|192.168.64.91|:443... connected.
WARNING: cannot verify accounting.internal.acme.com's certificate, issued \
by ‘CN=Kubernetes Ingress Controller Fake Certificate,O=Acme Co’:
  Self-signed certificate encountered.
WARNING: no certificate subject alternative name matches
        requested host name ‘accounting.internal.acme.com’.
HTTP request sent, awaiting response... 200 OK
```

Protecting Node Metadata and Endpoints

Kubernetes clusters expose ports used to communicate with cluster components. For example, the API server uses the port 6443 by default to enable clients like `kubectl` to talk to it when executing commands.

The Kubernetes documentation lists those ports in "Ports and Protocols". The following two tables show the default port assignments per node.

Table 2-1 shows the default inbound ports on the cluster node.

Table 2-1. Inbound control plane node ports

Port range	Purpose
6643	Kubernetes API server
2379–2380	etcd server client API
10250	Kubelet API
10259	kube-scheduler
10257	kube-controller-manager

Many of those ports are configurable. For example, you can modify the API server port by providing a different value with the flag `--secure-port` in the configuration file `/etc/kubernetes/manifests/kube-apiserver.yaml`, as documented for the cluster component. For all other cluster components, please refer to their corresponding documentation.

Table 2-2 lists the default inbound ports on a worker node.

Table 2-2. Inbound worker node ports

Port range	Purpose
10250	Kubelet API
30000–32767	NodePort Services

To secure the ports used by cluster components, set up firewall rules to minimize the attack surface area. For example, you could decide not to expose the API server to anyone outside of the intranet. Clients using `kubectl` would only be able to run commands against the Kubernetes cluster if logged into the VPN, making the cluster less vulnerable to attacks.

Cloud provider Kubernetes clusters (e.g., on AWS, Azure, or Google Cloud) expose so-called metadata services. Metadata services are APIs that can provide sensitive data like an authentication token for consumption from VMs or Pods without any additional authorization. For the CKS exam, you need to be aware of those node endpoints and cloud provider metadata services. Furthermore, you should have a high-level understanding of how to protect them from unauthorized access.

Scenario: A Compromised Pod Can Access the Metadata Server

Figure 2-3 shows an attacker who gained access to a Pod running on a node within a cloud provider Kubernetes cluster.

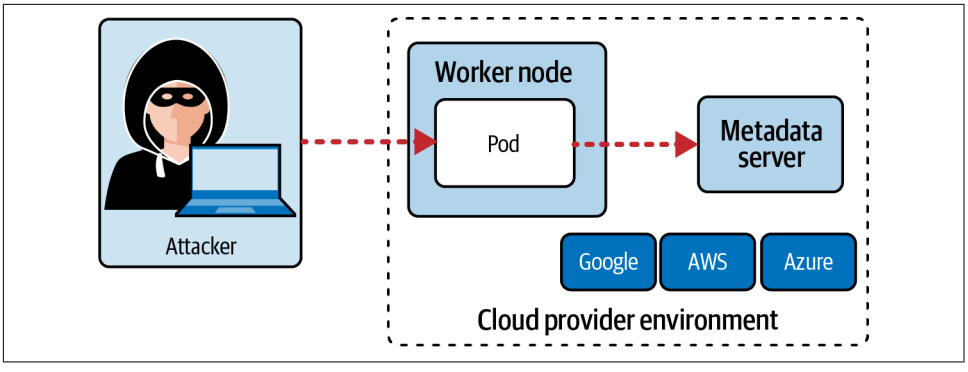

Figure 2-3. An attacker who gained access to the Pod has access to metadata server

Access to the metadata server has not been restricted in any form. The attacker can retrieve sensitive information, which could open other possibilities of intrusion.

Protecting Metadata Server Access with Network Policies

Let's pick one of the cloud providers that exposes a metadata endpoint. In AWS, the metadata server can be reached with the IP address 169.254.169.254, as described in the AWS documentation. The endpoints exposed can provide access to EC2 instance metadata. For example, you can retrieve the local IP address of an instance to manage a connection to an external application or to contact the instance with the help of a script. See the corresponding documentation page for calls to those endpoints made with the curl command line tool.

To prevent any Pod in a namespace from reaching the IP address of the metadata server, set up a network policy that allows egress traffic to all IP addresses except 169.254.169.254. Example 2-7 demonstrates a YAML manifest with such a rule set.

Example 2-7. A default deny-all egress to IP address 169.254.169.254 network policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress-metadata-server
  namespace: a12
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
```

```
    except:
    - 169.254.169.254/32
```

Once the network policy has been created, Pods in the namespace `a12` should not be able to reach the metadata endpoints anymore. For detailed examples that use the endpoints via `curl`, see the relevant AWS documentation.

Protecting GUI Elements

The `kubectl` tool isn't the only user interface (UI) for managing a cluster. While `kubectl` allows for fine-grained operations, most organizations prefer a more convenient graphical user interface (GUI) for managing the objects of a cluster. You can choose from a variety of options. The Kubernetes Dashboard is a free, web-based application. Other GUI dashboards for Kubernetes like Portainer go beyond the basic functionality by adding tracing of events or visualizations of hardware resource consumption. In this section, we'll focus on the Kubernetes Dashboard as it is easy to install and configure.

Scenario: An Attacker Gains Access to the Dashboard Functionality

The Kubernetes Dashboard runs as a Pod inside of the cluster. Installing the Dashboard also creates a Service of type `ClusterIP` that only allows access to the endpoint from within the cluster. To make the Dashboard accessible to end users, you'd have to expose the Service outside of the cluster. For example, you could switch to a `NodePort` Service type or stand up an Ingress. Figure 2-4 illustrates the high-level architecture of deploying and accessing the Dashboard.

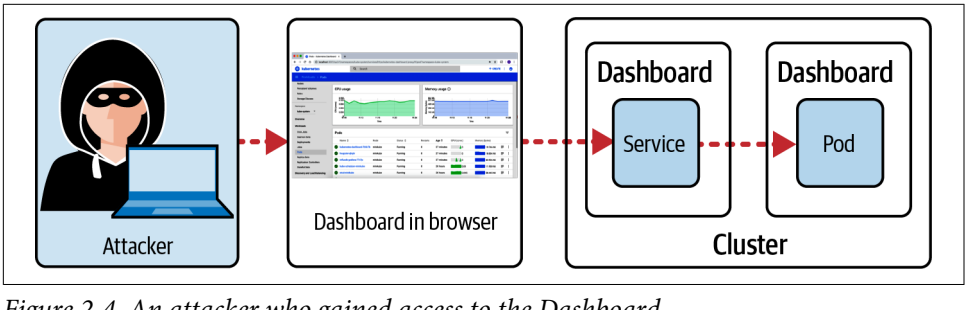

Figure 2-4. An attacker who gained access to the Dashboard

As soon as you expose the Dashboard to the outside world, attackers can potentially gain access to it. Without the right security settings, objects can be deleted, modified, or used for malicious purposes. The most prominent victim of such an attack was Tesla, which in 2018 fell prey to hackers who gained access to its unprotected Dashboard to mine cryptocurrencies. Since then, newer versions of the Dashboard changed default settings to make it more secure from the get-go.

Installing the Kubernetes Dashboard

Installing the Kubernetes Dashboard is straightforward. You can create the relevant objects with the help of the YAML manifest available in the project's GitHub repository. The following command installs all necessary objects:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/\
v2.6.0/aio/deploy/recommended.yaml
```

Rendering metrics in Dashboard

You may also want to install the metrics server if you are interested in inspecting resource consumption metrics as part of the Dashboard functionality.

You can find the objects created by the manifest in the `kubernetes-dashboard` namespace. Among them are Deployments, Pods, and Services. The following command lists all of them:

```
$ kubectl get deployments,pods,services -n kubernetes-dashboard
NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dashboard-metrics-scraper   1/1     1            1           11m
deployment.apps/kubernetes-dashboard        1/1     1            1           11m

NAME                                             READY   STATUS    RESTARTS   AGE
pod/dashboard-metrics-scraper-78dbd9dbf5-f8z4x   1/1     Running   0          11m
pod/kubernetes-dashboard-5fd5574d9f-ns7nl        1/1     Running   0          11m

NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP \
  PORT(S)    AGE
service/dashboard-metrics-scraper   ClusterIP   10.98.6.37       <none>      \
  8000/TCP   11m
service/kubernetes-dashboard        ClusterIP   10.102.234.158   <none>      \
  80/TCP     11m
```

Accessing the Kubernetes Dashboard

The `kubectl proxy` command can help with temporarily creating a proxy that allows you to open the Dashboard in a browser. This functionality is only meant for troubleshooting purposes and is not geared toward production environments. You can find information about the `proxy` command in the documentation:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Open the browser with the URL *http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy*. The Dashboard will ask you to provide an authentication method and credentials. The recommended way to configure the Dashboard is through bearer tokens.

Creating a User with Administration Privileges

Before you can authenticate in the login screen, you need to create a ServiceAccount and ClusterRoleBinding object that grant admin permissions. Start by creating the file `admin-user-serviceaccount.yaml` and populate it with the contents shown in Example 2-8.

Example 2-8. Service account for admin permissions

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

Next, store the contents of Example 2-9 in the file `admin-user-clusterrolebinding.yaml` to map the ClusterRole named `cluster-admin` to the ServiceAccount.

Example 2-9. ClusterRoleBinding for admin permissions

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

Create both objects with the following declarative command:

```
$ kubectl create -f admin-user-serviceaccount.yaml
serviceaccount/admin-user created
$ kubectl create -f admin-user-clusterrolebinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/admin-user created
```

You can now create the bearer token of the admin user with the following command. The command will generate a token for the provided ServiceAccount object and render it on the console:

```
$ kubectl create token admin-user -n kubernetes-dashboard
eyJhbGciOiJSUzI1NiIsImtpZCI6...
```

Expiration of a service account token

By default, this token will expire after 24 hours. That means that the token object will be deleted automatically once the "time to live" (TTL) has passed. You can change the TTL of a token by providing the command line option `--ttl`. For example, a value of `40h` will expire the token after 40 hours. A value of `0` indicates that the token should never expire.

Copy the output of the command and paste it into the "Enter token" field of the login screen, as shown in Figure 2-5.

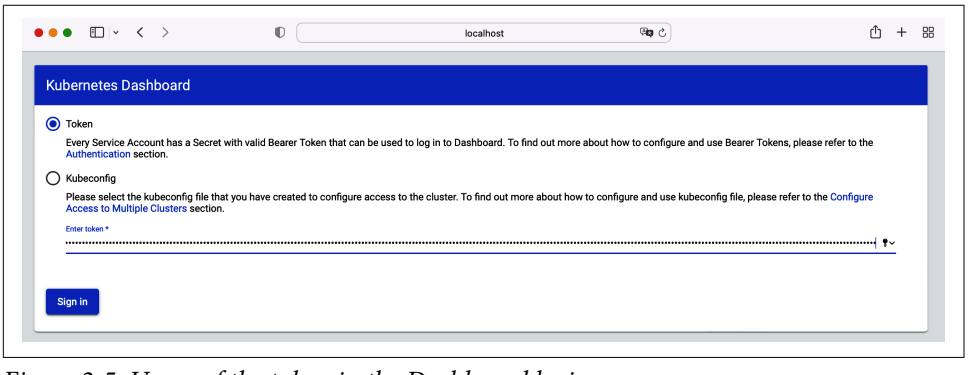

Figure 2-5. Usage of the token in the Dashboard login screen

Pressing the "Sign in" button will bring you to the Dashboard shown in Figure 2-6.

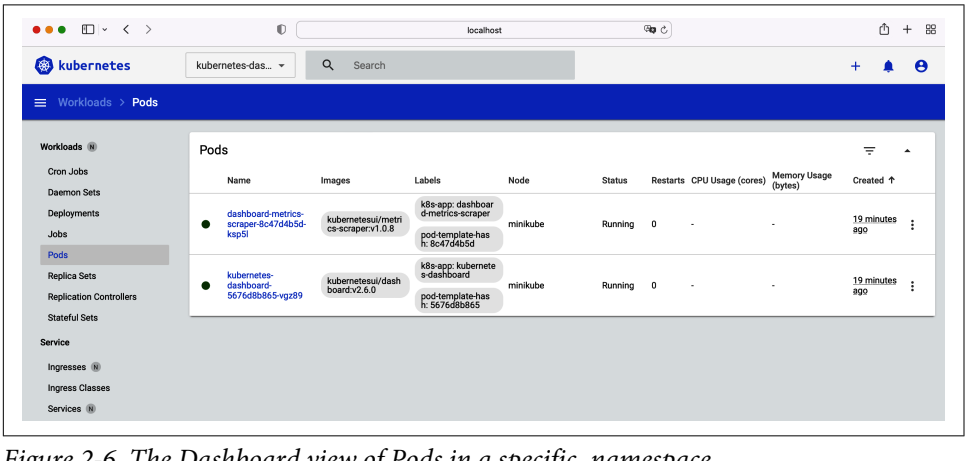

Figure 2-6. The Dashboard view of Pods in a specific namespace

You can now manage end user and cluster objects without any restrictions.

Creating a User with Restricted Privileges

In the previous section, you learned how to create a user with cluster-wide administrative permissions. Most users of the Dashboard only need a restricted set of permissions, though. For example, developers implementing and operating cloud-native applications will likely only need a subset of administrative permissions to perform their tasks on a Kubernetes cluster. Creating a user for the Dashboard with restricted privileges consists of a three-step approach:

1. Create a ServiceAccount object.
2. Create a ClusterRole object that defines the permissions.
3. Create a ClusterRoleBinding that maps the ClusterRole to the ServiceAccount.

As you can see, the process is very similar to the one we went through for the admin user. Step 2 is new, as we need to be specific about which permissions we want to grant. The YAML manifests that follow will model a user working as a developer that should only be allowed read-only permissions (e.g., getting, listing, and watching resources).

Start by creating the file `restricted-user-serviceaccount.yaml` and populate it with the contents shown in Example 2-10.

Example 2-10. Service account for restricted permissions

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: developer-user
  namespace: kubernetes-dashboard
```

The ClusterRole in Example 2-11 only allows getting, listing, and watching resources. All other operations are not permitted. Store the contents in the file `restricted-user-clusterrole.yaml`.

Example 2-11. ClusterRole for restricted permissions

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  name: cluster-developer
rules:
- apiGroups:
  - '*'
  resources:
```

```
  - '*'
  verbs:
  - get
  - list
  - watch
- nonResourceURLs:
  - '*'
  verbs:
  - get
  - list
  - watch
```

Last, map the ServiceAccount to the ClusterRole in the file `restricted-user-clusterrolebinding.yaml`, as shown in Example 2-12.

Example 2-12. ClusterRoleBinding for restricted permissions

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: developer-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-developer
subjects:
- kind: ServiceAccount
  name: developer-user
  namespace: kubernetes-dashboard
```

Create all objects with the following declarative command:

```
$ kubectl create -f restricted-user-serviceaccount.yaml
serviceaccount/restricted-user created
$ kubectl create -f restricted-user-clusterrole.yaml
clusterrole.rbac.authorization.k8s.io/cluster-developer created
$ kubectl create -f restricted-user-clusterrolebinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/developer-user created
```

Generate the bearer token of the restricted user with the following command:

```
$ kubectl create token developer-user -n kubernetes-dashboard
eyJhbGciOiJSUzI1NiIsImtpZCI6...
```

Operations that are not allowed for the logged-in user will not be rendered as disabled options in the GUI. You can still select the option; however, an error message is rendered. Figure 2-7 illustrates the behavior of the Dashboard if you try to delete a Pod via the user that doesn't have the permissions to perform the operation.

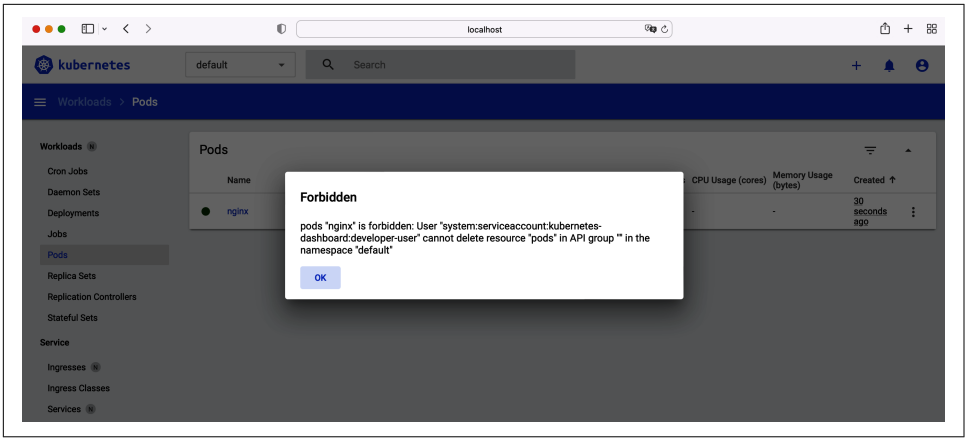

Figure 2-7. An error message rendered when trying to invoke a permitted operation

Avoiding Insecure Configuration Arguments

Securing the Dashboard in production environments involves the usage of execution arguments necessary for properly configuring authentication and authorization. By default, login functionality is enabled and the HTTPS endpoint will be exposed on port 8443. You can provide TLS certificates with the `--tls-cert-file` and `--tls-cert-key` command line options if you don't want them to be auto-generated.

Avoid setting the command line arguments `--insecure-port` to expose an HTTP endpoint and `--enable-insecure-login` to enable serving the login page over HTTP instead of HTTPS. Furthermore, make sure you *don't* use the option `--enable-skip-login` as it would allow circumventing an authentication method by simply clicking a Skip button in the login screen.

Verifying Kubernetes Platform Binaries

The Kubernetes project publishes client and server binaries with every release. The client binary refers to the executable `kubectl`. Server binaries include `kubeadm`, as well as the executable for the API server, the scheduler, and the kubelet. You can find those files under the "tags" sections of the Kubernetes GitHub repository or on the release page at *https://dl.k8s.io*.

Scenario: An Attacker Injected Malicious Code into Binary

The executables `kubectl` and `kubeadm` are essential for interacting with Kubernetes. `kubectl` lets you run commands against the API server, e.g., for managing objects. `kubeadm` is necessary for upgrading cluster nodes from one version to another. Say you are in the process of upgrading the cluster version from 1.23 to 1.24. As part

of the process, you will need to upgrade the `kubeadm` binary as well. The official upgrade documentation is very specific about what commands to use for upgrading the binary.

Say an attacker managed to modify the `kubeadm` executable for version 1.24 and coaxed you into thinking that you need to download that very binary from a location where the malicious binary was placed. As shown in Figure 2-8, you'd expose yourself to running malicious code every time you invoke the modified `kubeadm` executable. For example, you may be sending credentials to a server outside of your cluster, which would open new ways to infiltrate your Kubernetes environment.

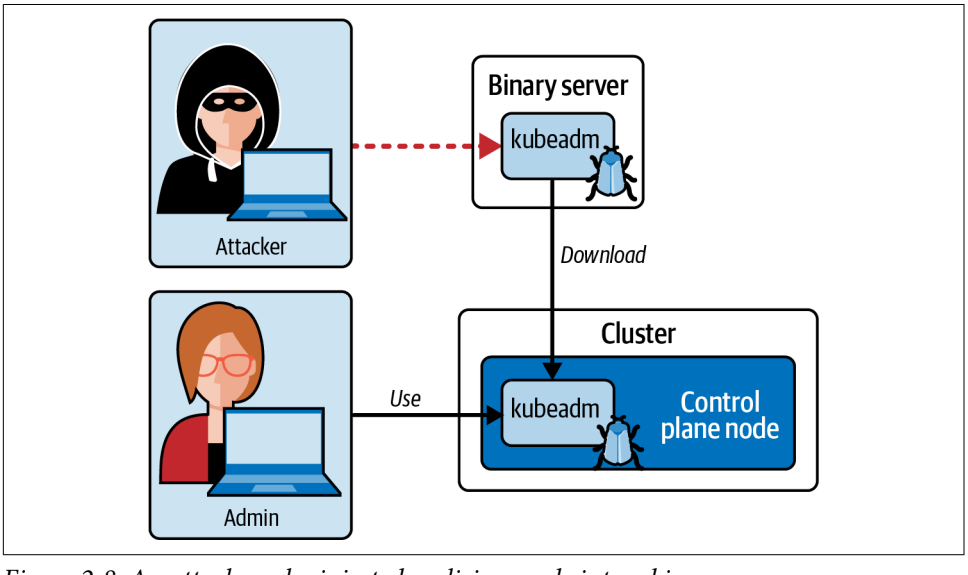

Figure 2-8. An attacker who injected malicious code into a binary

Verifying a Binary Against Hash

You can verify the validity of a binary with the help of a hash code like MD5 or SHA. Kubernetes publishes SHA256 hash codes for each binary. You should run through a hash validation for individual binaries before using them for the first time. Should the generated hash code not match with the one you downloaded, then there's something off with the binary. The binary may have been modified by a third party or you didn't use the hash code for the correct binary type or version.

You can download the corresponding hash code for a binary from *https://dl.k8s.io*. The full URL for a hash code reflects the version, operating system, and architecture of the binary. The following list shows example URLs for platform binaries compatible with Linux AMD64:

- `kubectl`: *https://dl.k8s.io/v1.26.1/bin/linux/amd64/kubectl.sha256*

- `kubeadm`: *https://dl.k8s.io/v1.26.1/bin/linux/amd64/kubeadm.sha256*
- `kubelet`: *https://dl.k8s.io/v1.26.1/bin/linux/amd64/kubelet.sha256*
- `kube-apiserver`: *https://dl.k8s.io/v1.26.1/bin/linux/amd64/kube-apiserver.sha256*

You'll have to use an operating system-specific hash code validation tool to check the validity of a binary. You may have to install the tool if you do not have it available on your machine yet. The following commands show the usage of the tool for different operating systems, as explained in the Kubernetes documentation:

- Linux: `echo "$(cat kubectl.sha256) kubectl" | sha256sum --check`
- MacOSX: `echo "$(cat kubectl.sha256) kubectl" | shasum -a 256 --check`
- Windows with Powershell: `$($(CertUtil -hashfile .\kubectl.exe SHA256)[1] -replace " ", "") -eq $(type .\kubectl.exe.sha256)`

The following commands demonstrate downloading the `kubeadm` binary for version 1.26.1 and its corresponding SHA256 hash file:

```
$ curl -LO "https://dl.k8s.io/v1.26.1/bin/linux/amd64/kubeadm"
$ curl -LO "https://dl.k8s.io/v1.26.1/bin/linux/amd64/kubeadm.sha256"
```

The validation tool `shasum` can verify if the checksum matches:

```
$ echo "$(cat kubeadm.sha256)  kubeadm" | shasum -a 256 --check
kubeadm: OK
```

The previous command returned with an "OK" message. The binary file wasn't tampered with. Any other message indicates a potential security risk when executing the binary.

Summary

The domain "cluster setup" dials in on security aspects relevant to setting up a Kubernetes cluster. Even though you might be creating a cluster from scratch with `kubeadm`, that doesn't mean you are necessarily following best practices. Using kube-bench to detect potential security risks is a good start. Fix the issues reported on by the tool one by one. You may also want to check client and server binaries against their checksums to ensure that they haven't been modified by an attacker. Some organizations use a Dashboard to manage the cluster and its objects. Ensure that authentication and authorization for the Dashboard restrict access to a small subset of stakeholders.

An important security aspect is network communication. Pod-to-Pod communication is unrestricted by default. Have a close look at your application architecture running inside of Kubernetes. Only allow directional network traffic from and to Pods to fulfill the requirements of your architecture. Deny all other network traffic. When exposing the application outside of the cluster, make sure that Ingress objects

have been configured with TLS termination. This will ensure that the data is encrypted both ways so that attackers cannot observe sensitive information like passwords sent between a client and the Kubernetes cluster.

Exam Essentials

Understand the purpose and effects of network policies
: By default, Pod-to-Pod communication is unrestricted. Instantiate a default deny rule to restrict Pod-to-Pod network traffic with the principle of least privilege. The attribute `spec.podSelector` of a network policy selects the target Pod the rules apply to based on label selection. The ingress and egress rules define Pods, namespaces, IP addresses, and ports for allowing incoming and outgoing traffic. Network policies can be aggregated. A default deny rule may disallow ingress and/or egress traffic. An additional network policy can open up those rules with a more fine-grained definition.

Practice the use of kube-bench to detect cluster component vulnerabilities
: The Kubernetes CIS Benchmark is a set of best practices for recommended security settings in a production Kubernetes environment. You can automate the process of detecting security risks with the help of the tool kube-bench. The generated report from running kube-bench describes detailed remediation actions to fix a detected issue. Learn how to interpret the results and how to mitigate the issue.

Know how to configure Ingress with TLS termination
: An Ingress can be configured to send and receive encrypted data by exposing an HTTPS endpoint. For this to work, you need to create a TLS Secret object and assign it a TLS certificate and key. The Secret can then be consumed by the Ingress using the attribute `spec.tls[]`.

Know how to configure GUI elements for secure access
: GUI elements, such as the Kubernetes Dashboard, provide a convenient way to manage objects. Attackers can cause harm to your cluster if the application isn't protected from unauthorized access. For the exam, you need to know how to properly set up RBAC for specific stakeholders. Moreover, you are expected to have a rough understanding of security-related command line arguments. Practice the installation process for the Dashboard, learn how to tweak its command line arguments, and understand the effects of setting permissions for different users.

Know how to detect modified platform binaries
: Platform binaries like `kubectl` and `kubeadm` can be verified against their corresponding hash code. Know where to find the hash file and how to use a validation tool to identify if the binary has been tempered with.

Sample Exercises

Solutions to these exercises are available in the Appendix.

1. Create a network policy that denies egress traffic to any domain outside of the cluster. The network policy applies to Pods with the label `app=backend` and also allows egress traffic for port 53 for UDP and TCP to Pods in any other namespace.
2. Create a Pod named `allowed` that runs the `busybox:1.36.0` image on port 80 and assign it the label `app=frontend`. Make a `curl` call to `http://google.com`. The network call should be allowed, as the network policy doesn't apply to the Pod.
3. Create another Pod named `denied` that runs the `busybox:1.36.0` image on port 80 and assign it the label `app=backend`. Make a `curl` call to `http://google.com`. The network call should be blocked.
4. Install the Kubernetes Dashboard or make sure that it is already installed. In the namespace `kubernetes-dashboard`, create a ServiceAccount named `observer-user`. Moreover, create the corresponding ClusterRole and ClusterRoleBinding. The ServiceAccount should only be allowed to view Deployments. All other operations should be denied. As an example, create the Deployment named `deploy` in the `default` namespace with the following command: `kubectl create deployment deploy --image=nginx --replicas=3`.
5. Create a token for the ServiceAccount named `observer-user` that will never expire. Log into the Dashboard using the token. Ensure that only Deployments can be viewed and not any other type of resource.
6. Download the binary file of the API server with version 1.26.1 on Linux AMD64. Download the SH256 checksum file for the API-server executable of version 1.23.1. Run the OS-specific verification tool and observe the result.

About the Author

Benjamin Muschko is a software engineer, consultant, and trainer with more than 20 years of experience in the industry. He's passionate about project automation, testing, and continuous delivery. Ben is an author, a frequent speaker at conferences, and an avid open source advocate. He holds the CKAD, CKA, and CKS certifications and is a CNCF Ambassador Spring 2023.

Software projects sometimes feel like climbing a mountain. In his free time, Ben loves hiking Colorado's 14ers and enjoys conquering long-distance trails.

Colophon

The animal on the cover of *Certified Kubernetes Security Specialist (CKS) Study Guide* is a domestic goose. These birds have been selectively bred from wild greylag (*Anser anse*) and swan geese (*Anser cygnoides domesticus*). They have been introduced to every continent except Antarctica. Archaeological evidence shows the geese have been domesticated since at least 4,000 years ago.

Wild geese range in size from 7 to 9 pounds, whereas domestic geese have been bred for size and can weigh up to 22 pounds. The distribution of their fat deposits gives the domestic goose a more upright posture compared to the horizontal posture of their wild ancestors. Their larger size also makes them less likely to fly, although the birds are capable of some flight.

Historically, geese have been domesticated for use of their meat, eggs, and feathers. In more recent times, geese have been kept as backyard pets or even for yard maintenance since they eat weeds and leaves. Due to the loud and aggressive nature of geese, they have also been used to safeguard property, since they will make a lot of noise if they perceive a threat or an intruder.

Domestic animals are not assessed by the IUCN. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.