Ace the Exam Series®

2022

FIRST EDITION

KUBERNETES SECURITY SPECIALIST (CKS)

EXAM CRAM NOTES

LAST MINUTE EXAM REVIEW MANUAL

UP TO DATE EXAM BLUEPRINT
LATEST EXAM QUESTIONS AND REGULARLY REFRESHED CONTENT FOR YOU TO MASTER YOUR EXAM CERTIFICATION.

ACE EXAMS WITH CONFIDENCE:
OUR PRECISE AND COMPREHENSIVE STUDY MATERIAL ENSURES PASSING GRADES.

IPSpecialist
Sharpen Your Skills for the Digital Future

Kubernetes Security Specialist (CKS)

Exam Cram Notes

First Edition

Chapter 01: Introduction

DevOps

DevOps is a combination of cultural principles, practices, and technologies that help organizations build high-speed applications and services, allowing them to innovate and enhance products faster than conventional software development and infrastructure management approaches. Because of this speed, businesses can better serve their customers and stay ahead of competitors.

Why DevOps Matters?

Software and the Internet have altered the environment and industries we live in, from commerce to entertainment to finance. Software is no longer only a way of sustaining a corporation; it has become an integral part of all aspects. Companies engage with their customers with software that is available as internet services or applications that can be utilized on various devices. To improve operational efficiencies, they also leverage software to revolutionize every value chain component, including logistics, communications, and operations. Firms in today's environment must adapt how they produce and distribute software in the same manner that physical goods companies revolutionized how they designed, constructed, and delivered things utilizing industrial automation during the twentieth century.

What is Kubernetes?

Kubernetes, often known as K8s, is an open-source system for automating containerized applications' deployment, scaling, and administration. In essence, it is a container orchestration system. It enables organizations to automate and orchestrate key IT processes and thus helps streamline and simplify business operations. Orchestration means your application follows microservices-based architecture and is already containerized. However, to put these separate images together into a manageable production-grade solution, we need to run the application via the orchestration tools. You must maintain control over the containers that run the applications and guarantee no downtime. While there are other orchestration tools available, Kubernetes is the most popular one.

Kubernetes in DevOps Space

Kubernetes and DevOps are, without a doubt, the cloud's power duo! For businesses trying to create complex applications, they work hand in hand.

How Can Kubernetes be a Strength For DevOps?

Developers can easily exchange their software and dependencies with IT operations using Kubernetes, and it reduces the burden while resolving problems across several settings. Container orchestration brings IT operations and developers closer together, making it easier for everyone to work together successfully and efficiently.

Kubernetes provides a platform for developers to adapt to consumer requests while offloading the work of operating applications to the cloud. This is accomplished by removing the manual chores associated with deploying and scaling containerized applications, allowing software to function more consistently when transferred from one environment to another.

For example, any number of containers may be scheduled and deployed onto a node (across public, private, and hybrid clouds), and Kubernetes handles those workloads, allowing you to perform what you set out to do. Container activities are eased with Kubernetes, including horizontal auto-scaling, rolling updates, and canary deployment.

As a result, using the Kubernetes process in DevOps may make the build/test/deploy pipelines easier.

Kubernetes: An Enabler for Enterprise DevOps

Kubernetes has several characteristics and features that make it ideal for designing, implementing and growing enterprise-grade DevOps pipelines. These features allow teams to automate the manual effort that would otherwise be required for orchestration. This form of automation is required when teams boost production or, more critically, quality.

Lineage of Kubernetes

Kubernetes was originally developed at Google, having emerged from Google's Borg project, but now it has been taken over by cloud Native Computing Foundation, where the Kubernetes project is not just open-source but openly managed by an open community.

Features of Kubernetes

1. **Automated rollouts and rollbacks:** Kubernetes deploy updates to your application or its settings in stages, monitoring application health to avoid killing all of your instances at once. Kubernetes will revert the update for you if something goes wrong. Profit from a growing ecosystem of configurations.
2. **Storage orchestration:** Automatically mount your preferred storage system, whether it is from local storage, a public cloud provider like GCP or AWS, or a network storage system like NFS, Gluster, iSCSI, Cinder, Ceph, or Flocker.
3. **Automatic bin packing:** Places containers automatically depending on resource requirements and other limitations without losing availability. You can mix essential and best-effort workloads to increase utilization and save even more resources.
4. **IPv4/IPv6 dual-stack:** Allocation of IPv4 and IPv6 addresses to Pods and Services
5. **Self-healing:** Kubernetes' ability to self-heal is one of its most appealing features. Kubernetes will automatically reload a containerized app or an application component if it goes down, matching the so-called ideal state.
6. **Service discovery and load balancing:** There is no need to change your app to use a new service discovery approach. Kubernetes provides each Pod its IP address and a single DNS name for a group of Pods, which may load-balance.
7. **Secret and configuration management:** Deploy and update secrets and application configuration without rebuilding your image or exposing secrets in your stack settings.
8. **Batch execution:** Kubernetes can also handle your batch and continuous integration workloads, replacing containers that fail if necessary.
9. **Horizontal scaling:** Scale up and down your program with a single command, a user interface, or automatically based on CPU consumption.
10. **Designed for extensibility:** Easily add new features to your Kubernetes

cluster without modifying the source code.

About the Course

The CKS (Certified Kubernetes Security Specialist) certification requires you first to earn your Certified Kubernetes Administrator (CKA) certification. You will need to be familiar with the topics and skills from the CKA in order to understand the concepts of CKS. This chapter will cover all of the subject domains that are part of the CKS curriculum, including cluster setup, where you will learn about securely setting up and configuring your cluster.

Certified Kubernetes Administrator (CKA)

Kubernetes essentially builds a layer of abstraction known as the Kubernetes Cluster. Multiple servers are joined together to make up the Kubernetes Cluster. When using Kubernetes, you do not have to think as much about what server to run your container on; instead, you just run the container on the cluster, and Kubernetes handles the process of running that container on an actual server.

The main feature and primary purpose of Kubernetes is container orchestration, which is used to manage containers across multiple host systems dynamically. Kubernetes also offers several features around application reliability, making it easier to build reliable, self-healing, and scalable applications. Moreover, Kubernetes makes it easy to build automation around your containers.

About the CKS Exam

You might want to be aware of a few things regarding the CKS exam. First, the exam is hands-on, which means that you do not need to memorize a lot of information and facts to prepare for the exam. Thus, you only need to do the tasks expected of a Certified Kubernetes Security Specialist in the real world. Another thing to be aware of is that the exam is open documentation, which means that you can view certain approved documentation sites in one additional browser tab. You cannot just look at anything out there on the internet, but there are certain documentation sites that you will be able to access.

When preparing for the exam, you do not necessarily need to focus on memorizing every little detail that you can look up in the documentation during the exam. You want to focus more on the general knowledge of performing different hands-on tasks.

Another thing to be aware of is the time limit. You will have a total of 2 hours to go through all of the different performance-based aspects of the exam. There are specific documentation sites that are allowed during the exam; for example, you can access the Kubernetes GitHub and the Kubernetes Blog. There are also some external tools for which you can view documentation, such as Trivy, a container image vulnerability scanning tool. You can access the GitHub documentation for that, and the documentation for Sysdig and Falco as well. These tools can help you detect security threats in real time and access the AppArmor documentation. AppArmor is a Linux module that allows you to restrict what individual processes can do on your Linux server, including Kubernetes containers. You can access the documentation for Kubernetes and the external tools that you might have to interact with as you are taking the exam.

This course includes various resources that can help you prepare for the exam. First, it has chapters covering background knowledge and walks you through completing tasks in the cluster using hands-on demonstrations.

The purpose of the chapters is to give you the background knowledge and context, and show you the steps of how to complete some of the required tasks practically using a real Kubernetes cluster. Moreover, you have labs, which cover similar content as the lessons. However, the purpose of the labs is to allow you to complete these tasks on your own essentially.

What is Kubernetes?

Kubernetes, also known as K8S, is an open-source system for programming deployment, scaling, and operating containerized applications. Kubernetes helps you deploy containers across a pool of computing resources, such as servers. The core feature of Kubernetes is that it makes it easy to deploy and manage containers. Through Kubernetes, you can quickly deploy your container and get it up and running on one of the servers. Kubernetes does a lot more than just help you deploy a container to a server since it allows you to manage multiple application replicas within your pool of servers easily. It makes it easy to manage containers running across a pool of servers or other computing resources.

Kubernetes handles networking by providing a networking framework that helps you manage and control network communication between containers. It also provides various security features to build more secure applications within your Kubernetes infrastructure. It also offers some configuration management features that help you manage application configuration and pass configuration data to your containers.

The Kubernetes Cluster

A Kubernetes cluster is simply a collection of worker machines that run containers. You have multiple machines running your containers; that collection of machines is your Kubernetes cluster.

Building a Kubernetes Cluster

There are several ways to form a Kubernetes cluster, one of them is using a tool called kubeadm. Kubeadm is the tool for “first-paths” when forming your first Kubernetes cluster.

As a part of your account access, you have a total of nine units, so creating three medium servers will be just enough to hit that capacity. A three-node cluster will be sufficient for most practice clusters, which allows you to create all the resources you want.

Log in and Start Building

Once the servers show a “ready " status, go ahead and get logged in. Select the server to expand and view the details of each server. Click on the page icon next to the temporary password and click on the terminal to open it in a new browser tab. Type the username and paste it into the temporary password box.

The kubelet is the node agent that will run all the pods for you, including the kube-system pods. The kubeadm is a tool for installing multi-node Kubernetes clusters, while the kubectl is the command-line tool for communicating with Kubernetes.

Kubernetes Security Overview

Cyber-attacks are are considered as more and more prevalent, and it is becoming everyday news. This makes security more important than ever to keep our systems and information secure, including Kubernetes systems and the information stored within Kubernetes systems.

Kubernetes security entails safeguarding container images and runtime host, platform, and application layers in your build pipeline. By incorporating security into the continuous delivery life cycle, your company will be able to reduce risk and vulnerabilities across an ever-growing attack surface.

Chapter 02: Cluster Setup

Cluster Setup Introduction

In this chapter, we will discuss the security and configurations of the Kubernetes cluster. First, we will look into network policies, in which we will discuss locking down the network access, and only allowing the necessary network access between your pods using network policies.

Then, we will discuss CIS Benchmark while using kube-bench to run CIS benchmark tests to verify how well our cluster conforms to the security standards set out by the Center of Internet Security.

Securing Cluster

This section discusses how to safeguard a cluster from unintentional or malicious access, as well as security suggestions.

Controlling Access to Kubernetes APIs

Because Kubernetes is totally API-driven, the first line of protection is to restrict and limit who can access the cluster and what actions they are authorized to execute.

For all API traffic, use Transport Layer Security (TLS).

The majority of installation methods will allow the appropriate certificates to be issued and distributed to the cluster components, as Kubernetes expects all API interactions in the cluster to be encrypted by default with TLS. Note that some components and installation methods may enable local ports through HTTP, and administrators should become familiar with each component's settings to discover potentially unprotected traffic.

API Authentication

When you install a cluster, choose an authentication mechanism for the API servers that matches the common access patterns. Small, single-user clusters, for example, may want to employ a simple certificate or static Bearer token solution. Larger clusters might want to integrate an existing OIDC or LDAP server that divides users into groups.

All API clients, including those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins, must be authorized. These clients are usually service accounts or employ x509 client certificates, and they are created automatically or as part of the cluster setup.

API Authorization

Every API call must also pass an authorization check once it has been authenticated. An integrated Role-Based Access Control (RBAC) component ships with Kubernetes, which matches an incoming user or group to a set of permissions grouped into roles. These rights integrate verbs (get, create, delete) with resources (pods, services, nodes) and might be namespace or cluster specific. A collection of pre-defined roles is offered that provide a sensible default separation of responsibilities depending on the actions a client may want to conduct. The Node and RBAC authorizers, as well as the NodeRestriction admission plugin, are advised to be used together.

Controlling access to the kubelet

kubelets expose HTTPS endpoints that provide extensive management over nodes and containers. Unauthenticated access to this API is enabled by default in kubelets.

kubelet authentication and authorization should be enabled in production clusters.

Controlling a workload's or user's capabilities in real-time

Kubernetes authorization is purposefully high level, focusing on coarse resource actions. More powerful policies exist to limit how those objects behave on the cluster, themselves, and other resources based on use case.

Limiting Resource Usage

The number and capacity of resources available to a namespace are limited by resource quota. This is most commonly used to limit how much CPU, RAM, or persistent disc a namespace can allocate, but it can also be used to limit how many pods, services, or volumes each namespace can have.

Limit ranges limit the maximum or minimum size of certain of the above resources, preventing users from demanding arbitrarily high or low amounts for commonly reserved resources such as memory or providing default limits when none are provided.

Network access restrictions

Application writers can use a namespace's network policies to limit which pods from other namespaces can access pods and ports within their namespace. Many of the Kubernetes networking providers that are supported now adhere to network policies.

Quota and limit ranges can also be used to govern whether users can request node ports or load-balanced services, which can affect whether those users' applications are visible outside of the cluster on many clusters.

Additional precautions, such as per-node firewalls, physically separating cluster nodes to avoid cross-talk or advanced networking policies, may be available that manage network rules on a per-plugin or per-environment basis.

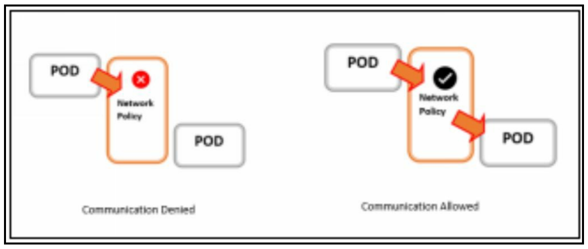
Restricting Cloud Metadata API access

Metadata services are frequently exposed locally to instances by cloud platforms (AWS, Azure, GCE, and so on). By default, pods running on an instance can access these APIs, which can contain cloud credentials for that node or provisioning data like kubelet credentials. This account's credentials can be used to elevate within the cluster or to other cloud services.

Limit the permissions given to instance credentials when running Kubernetes on a cloud platform, use network policies to restrict pod access to the metadata API, and avoid using provisioning data to deliver secrets.

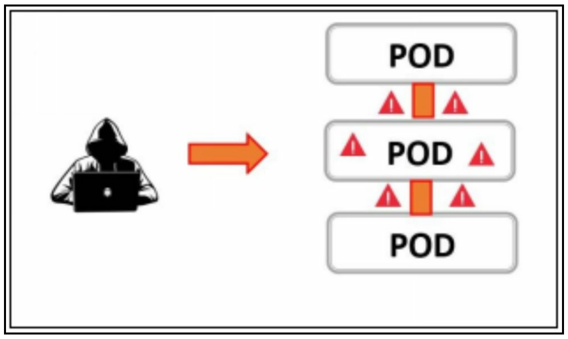
Restricting Default Access with Network Policies

Network policies allow you to prevent or restrict network communication to and from pods. They are like firewalls within the Kubernetes cluster network. As depicted in the diagram below, pods can communicate with each other using a cluster network, but network policies allow us to control that.

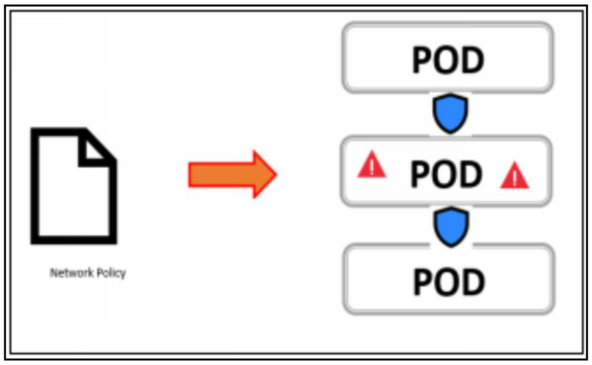


Default Deny

Suppose a hacker tries to take control of one of your pods and then uses that pod to reach all the other pods via a cluster network. You can use the default deny network policy to block all traffic except for what is needed to avoid potential malicious traffic.



The communication between pods will be blocked using network policy. So even if the attacker compromises one of the pods, they would not be able to use that pod to compromise other pods via the cluster network. Default deny network blocks all traffic by default unless we explicitly allow network communication.



Allowing Limited Access with Network Policies

You might use Kubernetes NetworkPolicies for certain applications in your cluster if you wish to manage traffic flow at the IP address or port level (OSI layer 3 or 4). NetworkPolicies are an application-centric construct that allows you to declare how a pod can communicate across the network with various network "entities" (we use the term "entity" here to avoid overusing terms like "endpoints" and "services," which have unique Kubernetes implications). Other connections are not affected by NetworkPolicies since they have a pod on one or both ends.

Network policies do not compete with one another; rather, they complement one another. If a policy applies to a given pod for a given direction, the connections allowed in that direction from that pod are the sum of the policies that apply. As a result, the order of evaluation has no bearing on the policy outcome. Even if we deny default traffic in a default policy, we can allow traffic explicitly in a different policy. With a default deny policy in place, you can allow necessary traffic with an additional targeted policy.

Mandatory Fields

A NetworkPolicy, like any other Kubernetes configuration, requires the apiVersion, kind, and metadata fields.

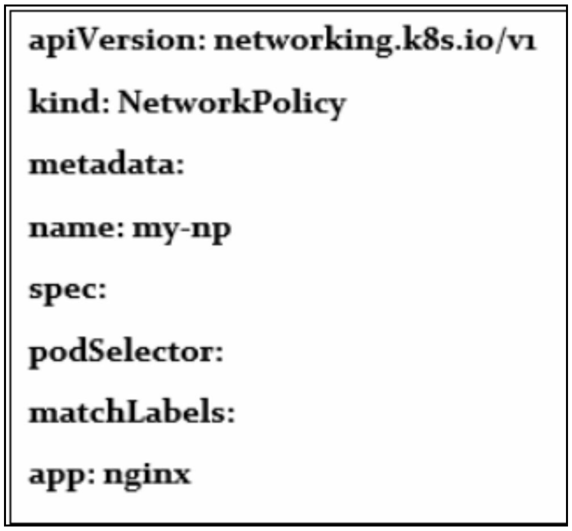
spec

The NetworkPolicy standard contains all of the necessary information to specify a specific network policy in the provided namespace.

podSelector

A podSelector is included in each NetworkPolicy, which picks the pod grouping to which the policy applies. The policy in this example chooses pods with the label "role=db." All pods in the namespace are selected by an empty podSelector.

It is important to note that the policy will not affect pods that do not match the podSelector. So, if you allow traffic for one pod, the policy does not apply to other pods that do not match that selector.



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
name: my-np
spec:
podSelector:
matchLabels:
app: nginx
```

policyTypes

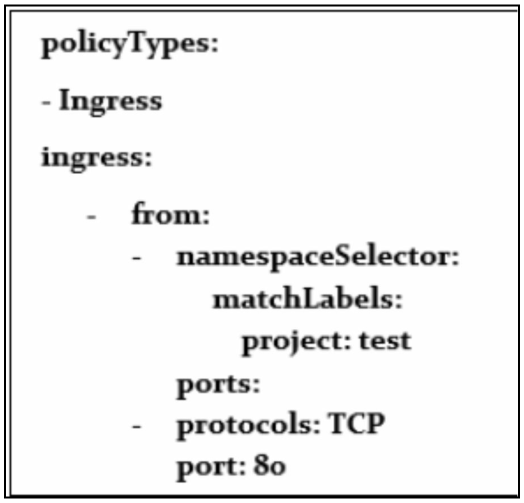
The policyTypes list in each NetworkPolicy can include either Ingress, Egress, or both. The policyTypes parameter specifies whether a policy applies to ingress traffic to a specific pod, egress traffic from a specific pod, or both. Ingress will always be set if no policyTypes are given on a NetworkPolicy, and Egress will be set if the NetworkPolicy includes any egress rules.

Ingress/Egress Rules

Ingress rules allow control of incoming traffic, and egress rule allows control of outgoing traffic. The port section determines which port will allow traffic for the rule, so in the example below, we are allowing traffic to only one port 80. To/from selectors determine the sources and destinations of allowed traffic.

A list of acceptable ingress rules may be included in each NetworkPolicy. Each rule enables traffic that matches the sections from and ports. The sample policy has a single rule that matches traffic on a single port from one of three sources: an ipBlock, a namespaceSelector, or a podSelector.

A list of acceptable egress rules may be included in each NetworkPolicy. Traffic that matches both the to and ports portions is allowed by each regulation. A single rule in the sample policy matches traffic on a single port to any destination in the 10.0.0.0/24 subnet.



```
policyTypes:
- Ingress
ingress:
    -   from:
        -   namespaceSelector:
                matchLabels:
                    project: test
            ports:
        -   protocols: TCP
            port: 80
```

Running a CIS Benchmark with kube-bench

CIS is the "Center of Internet Security," a non-profit organization dedicated to promoting digital security. CIS benchmarks are a set of consensus-driven community standards and best practices for securing systems.

kube-bench

kube-bench is a tool that checks your cluster to see how well it confirms the CIS Kubernetes Benchmark. If you check out the CIS Kubernetes benchmark document, you will notice it is very long and includes many different things.

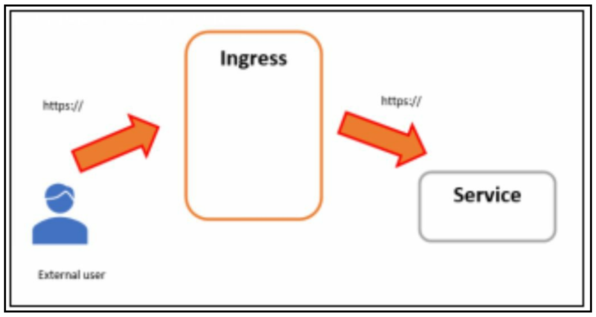
Checking all those things manually would take a lot of time. In this case, kube-bench is a useful tool for insight into how well your cluster conforms to the standards set by the CIS benchmark. It generates a report that will tell you if your cluster has some security issues based on CIS recommendations.

Implementing TLS with Ingress

Ingress is a Kubernetes object that manages access to Services from outside the cluster. Ingresses can provide features on top of services such as load balancing and TLS termination.

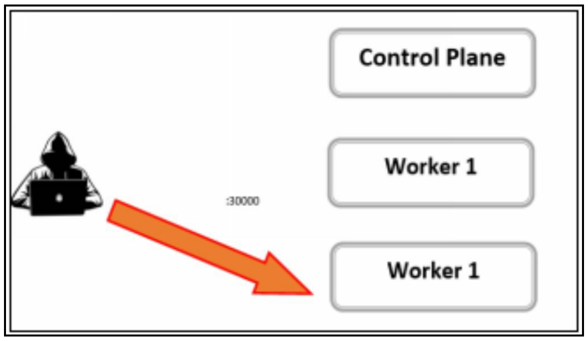
Ingresses and TLS

Ingresses can be used to provide TLS termination for services. In the diagram below, we have a service here and an external user, and we can put an ingress in the middle, which would allow the user to communicate securely using HTTPS. The ingress will communicate using plain old HTTP with the service. It prevents you from worrying about HTTPS or TLS termination at the service level. You can abstract that and worry about it at the ingress level.



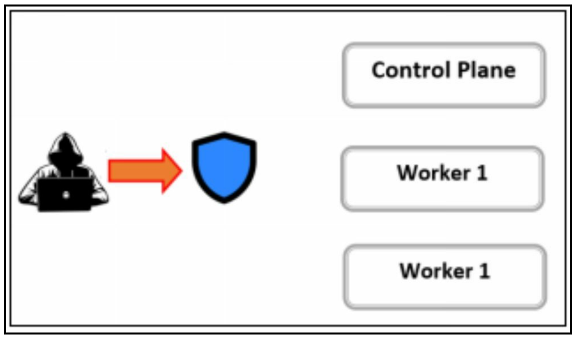
Securing Node Endpoints

Consider an attacker checking all the ports that open on all our nodes to see if one of these ports is vulnerable. Kubernetes nodes use certain ports to communicate. Therefore, we need to be aware of what ports the Kubernetes nodes are using, which is also known as an attack surface. The attack surface is all different components of your system that could potentially be attacked.



Minimizing Attack Surfaces

We can mitigate a lot of these attacks to using network segmentation and firewalls to keep those parts safe. As shown in the diagram, if we set up some firewall attacker would not be able to try out all the different ports on our control plane and our two worker nodes because he would not even be able to communicate with those ports from outside.



Deploy and Access to Kubernetes Dashboard

The dashboard is a web-based user interface for Kubernetes. The dashboard lets you deploy containerized applications to a Kubernetes cluster, troubleshoot them, and control cluster resources. The dashboard can be used to receive an overview of your cluster's apps and create and change specific Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc.). For example, you can use a deploy wizard to scale a Deployment, start a rolling update, restart a pod, or deploy additional apps.

The dashboard also shows the status of Kubernetes resources in your cluster, as well as any faults that have occurred.

Deploying the Dashboard UI

The Dashboard User Interface is not enabled by default. Run the following command to deploy it:

```
kubectl                                    apply                                    -f
```

```
https://raw.githubusercontent.com/kubernetes/dashboard/v2.5.0/aio/deploy/re
```

Accessing the Dashboard UI

Dashboard deploys with a simple RBAC configuration by default to safeguard your cluster data. The dashboard currently only accepts Bearer Tokens for login. Follow our instructions for creating a sample user to create a token for this experiment.

> **Note:** The sample user will have administrative capabilities and is strictly for educational purposes.

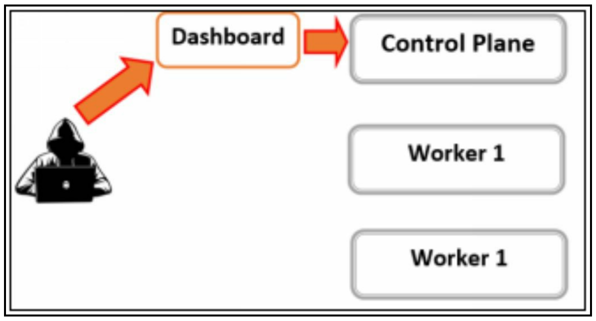
Command-line proxy

By running the following command with the kubectl command-line tool, you can allow access to the Dashboard:

```
kubectl proxy
```

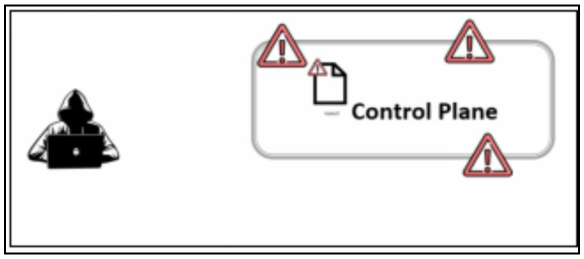
Securing GUI Elements

Kubernetes GUI (Graphical User Interface) provides a visual interface for managing your cluster. Such as the Kubernetes dashboard is one tool that provides a graphical interface to manage the cluster. If an attacker can control your GUI tool, they can control your cluster. Therefore, if you are using a GUI tool, make sure you keep it secure. GUI tool presents an additional attack surface that an attacker can use to gain access or control over your cluster. Using firewalls or network segmentation, you can use role-based access control or lock down network access to the tool to secure your GUI tool.

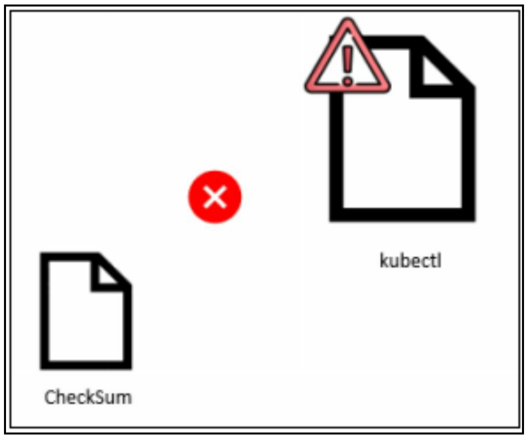


Verifying Kubernetes Platform Binaries

Binaries are the executable files used to run Kubernetes components. kubectl, kubeadm, and kubelet all run on our servers using binaries. Attackers can put malicious codes in our binaries before we download them. In this case, whenever we run kubectl, we will be running some malicious code, and the attacker will gain control of the control plane.



If we use packaged installation, it automatically verifies the binaries. When manually installing binaries, you can verify that the binary has not been tampered with before running it. Kubernetes provides checksum files for their binaries. The checksum contains a cryptographic hash calculated from the contents of a valid binary. If your binaries have been tampered with, they will not match the checksum.



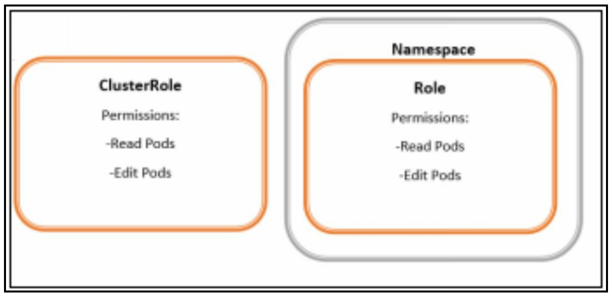
Chapter 03: Cluster Hardening

Cluster Hardening Introduction

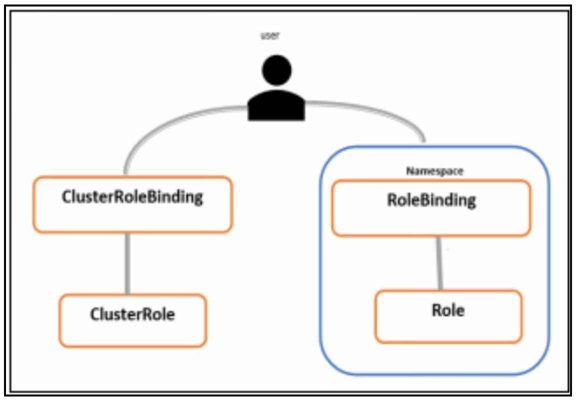
In this chapter, we will focus on service accounts and how to limit their permissions and security risks. We will review role-based access control and discuss some basic principles of protecting the API from attack and Kubernetes versioning.

Exploring Service Accounts

Roles and ClusterRoles essentially define a set of permissions; for example, a role or ClusterRole might define the permission to read or edit pods or a combination of two. Thus, they define a permission that an entity can perform within your cluster. The difference between ClusterRole and Role is that Roles live in a namespace, while ClusterRole works across all namespaces. Suppose you have a set of permissions that you want to apply all over the cluster, regardless of the namespace, use a ClusterRole. However, if it is specific to a particular namespace, use a Role.

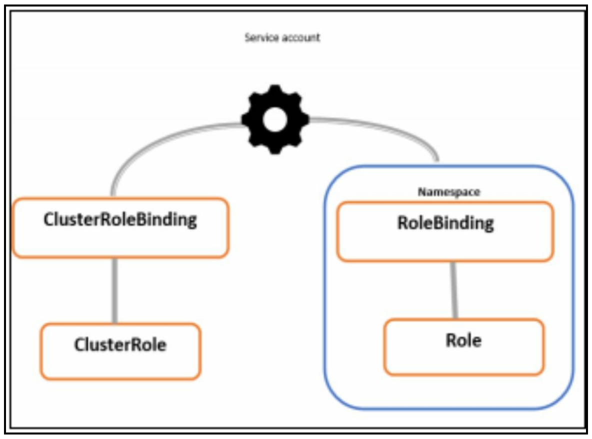


Once we have our Roles and ClusterRoles set up to define permissions, we need to attach those permissions to a particular user. This is what RoleBindings and ClusterRoleBindings do, as they attach Roles and ClusterRoles to users. ClusterRoleBindings is not specific to a particular namespace, but RoleBinding is specific to a particular namespace.



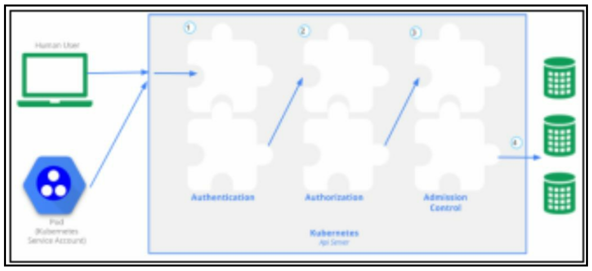
Service Accounts

Service Accounts give pods access to the Kubernetes API, and their permissions are governed by regular role-based access control objects. One can use ClusterRole and ClusterRoleBinding and just Roles or regular RoleBindings to provide permissions to a service account. The service account will provide a way for the pod's internal process to access and authenticate with the Kubernetes API.



Restricting Access to the Kubernetes API

Users can use kubectl, client libraries, or REST requests to access the Kubernetes API. API access can be granted to both human users and Kubernetes service accounts. When a request arrives at the API, it travels through various phases, as shown in the picture below:



Transport Security

The API in a typical Kubernetes cluster runs on port 443, which is secured using TLS. A certificate is presented by the API server. This certificate could be signed by a private certificate authority (CA) or a public key infrastructure linked to a well-known CA.

If your cluster employs a private certificate authority, you'll need a copy of that CA certificate in your client's /.kube/config so you can trust the connection and know it wasn't intercepted.

At this point, your client can offer a TLS client certificate.

Authentication

The HTTP request moves to the Authentication phase after TLS is established. Step 1 is depicted in the diagram. The API server is configured to run one or more Authenticator modules by the cluster formation script or cluster admin. Authentication goes into greater information about authenticators.

The authentication phase takes the full HTTP request as input, but it usually looks at the headers and/or client certificate.

Client certificates, passwords, plain tokens, bootstrap tokens, and JSON Web Tokens are among the authentication modules available (used for service accounts).

Multiple authentication modules can be supplied, and each one will be tried in turn until one succeeds.

The request is denied with HTTP status code 401 if it cannot be authenticated. Otherwise, the user is authorised as a certain username, and following processes can utilize that username to make decisions. Some authenticators additionally show the user's group memberships, whereas others do not.

While Kubernetes makes use of usernames for access control and request logging, it lacks a User object and does not store usernames or other user information in its API.

Authorization

The request must be authorised after it has been verified as originating from a specific user. Step 2 is depicted in the diagram.

The requester's username, the desired action, and the item affected by the action must all be included in the request. If an existing policy says that the user has authority to perform the requested action, the request is allowed.

To interface with current organization-wide or cloud-provider-wide access control systems, Kubernetes authorization requires the use of standard REST characteristics. Because these control systems may interface with APIs other than the Kubernetes API, REST formatting is essential.

ABAC mode, RBAC mode, and Webhook mode are among the authorization modules supported by Kubernetes. When a cluster is created, the authorisation modules that should be utilized in the API server are configured. If more than one authorization module is specified, Kubernetes evaluates each module and allows the request to proceed if any module authorises it. The request will be refused if all of the modules deny it (HTTP status code 403).

Admission Control

Software modules that can modify or deny requests are known as Admission Control Modules. Admission Control modules have access to the contents of the object being created or updated in addition to the characteristics exposed to Authorization modules.

Requests to create, change, remove, or connect to (proxy) an object is handled by admission controllers. Requests to read objects are ignored by admission controllers. When several admission controllers are configured, they are called in the order in which they were configured.

Step 3 is depicted in the diagram.

If any admission controller module rejects, unlike the authentication and authorization modules, the request is immediately rejected.

Admission controllers can provide elaborate defaults for fields in addition to rejecting objects.

Admission Controllers describe the available Admission Control modules.

A request is validated using the validation methods for the associated API object after passing all admission controllers and then written to the object storage (shown as step 4).

Auditing

Kubernetes auditing creates a chronological set of records that document the course of events in a cluster. Users, applications that use the Kubernetes API, and the control plane itself all create activity, which the cluster audits.

RBAC Authorization

RBAC (role-based access control) is a means of controlling access to computer and network resources based on the responsibilities of individual users within your company.

The rbac.authorization.k8s.io API group is used to drive authorization decisions in RBAC authorization, allowing you to dynamically set policies using the Kubernetes API.

Start the API server with the --authorization-mode flag set to a comma-separated list that contains RBAC, such as:

```
kube-apiserver --authorization-mode=Example,RBAC --other-options --more-options
```

API Objects

Role, ClusterRole, RoleBinding, and ClusterRoleBinding are the four types of Kubernetes objects declared via the RBAC API. Objects, like any other Kubernetes object, can be described or amended using tools like kubectl.

Role and ClusterRole

The rules of an RBAC Role or ClusterRole represent a collection of permissions. Permissions are only added together (there are no "deny" rules).

A Role always sets permissions within a specific namespace; while creating a Role, you must define which namespace it belongs to.

ClusterRole, on the other hand, is a nameless resource. Because a Kubernetes object can only be either namespaced or not namespaced, the resources have different names (Role and ClusterRole).

ClusterRoles have a variety of applications. A ClusterRole can be used to:

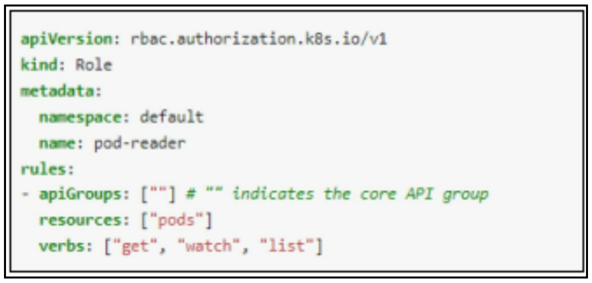
1. Permissions on namespaced resources can be defined and granted within each namespace (s)

2. Permissions on named resources can be defined and granted across all namespaces.

3. define cluster-scoped resource permissions

Use a Role to define a role within a namespace and a ClusterRole to define a role that spans the entire cluster.

Role Example



```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

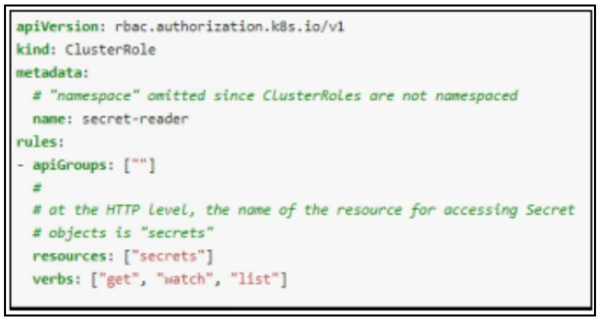
ClusterRole Example

The same permissions as a Role can be granted by a ClusterRole. Because ClusterRoles are cluster-scoped, you can use them to grant access to the following:

1. resources with a clustered scope (like nodes)
2. /healthz and other non-resource endpoints
3. spanning all namespaces, namespaced resources (like Pods)

You can use a ClusterRole to allow a certain user to run kubectl obtain pods, for example. --all-namespaces

Here is an example of a ClusterRole that can be used to allow read access to secrets in any namespace (depending on how it is bound):



```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RoleBinding and ClusterRoleBinding

A role binding assigns a user or group of users the permissions indicated in a role. It contains a list of topics (users, groups, or service accounts) as well as a reference to the role that is being assigned. A RoleBinding offers access to a certain namespace, but a ClusterRoleBinding grants access to the entire cluster.

Any Role in the same namespace can be referenced by a RoleBinding. A RoleBinding can also refer to a ClusterRole and tie that ClusterRole to the RoleBinding's namespace. A ClusterRoleBinding is used to bind a ClusterRole to all the namespaces in your cluster.

Keeping K8s Updated

Now, consider that this malicious actor has found some vulnerabilities in an older version of Kubernetes, which they can use to compromise our cluster. If we are running one of those older versions of Kubernetes, they might be able to use those vulnerabilities to gain access or control over our cluster. New security flaws and vulnerabilities are constantly being discovered, and one needs to install newer versions that include security packages to prevent those vulnerabilities from being exploited.

Kubernetes Updates

You need to be aware of the difference between minor and patch versions in Kubernetes. Minor versions are regularly patched and contain your bug and security fixes. To take advantage of the latest security fixes that prevent those vulnerabilities from being exploited, you need to install those latest patch versions. New patch versions are constantly applied to minor versions to provide security fixes. You should be aware of the patch support period and that the latest three minor versions receive patch support. Minor versions will receive patches for approximately one year, so do not let your minor version fall more than three versions or one year behind. Be aware of the process of backporting security patches, as security patches are usually backported to active minor versions.

It is important to keep the cluster updated because there might be security patches that are not being backported to those earlier versions and are only present in the latest minor version of Kubernetes.

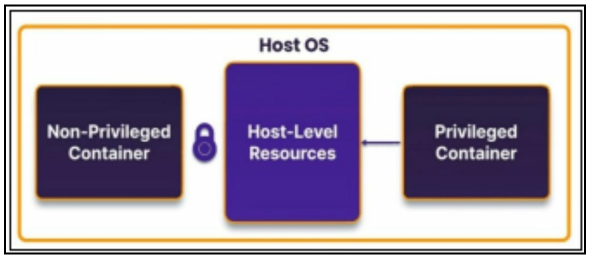
Chapter 04: System Hardening

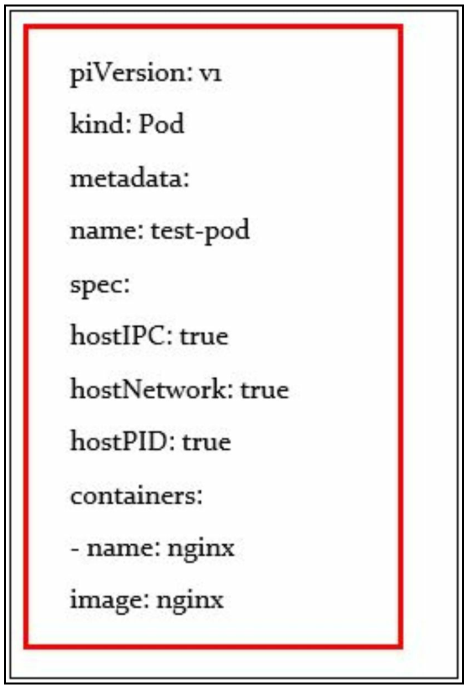
Introduction

This chapter will discuss the topics related to the actual host systems where our Kubernetes cluster is running. The first thing that we will focus on is a few security concerns and techniques related to the security of the host operating system. We will briefly discuss IAM roles related to Kubernetes when you are running in the context of Amazon Web Services. We will also discuss network security, the importance of securing Kubernetes from a networking standpoint, and use App Armor to provide some customized restrictions on what containers can do from a security perspective.

Host Namespace

The concept of operating system namespaces is completely different from the idea of Kubernetes namespaces. In our Kubernetes servers, we have host namespaces, which are the namespaces used by normal applications running directly on our host operating system. Containers run in completely separate container namespaces, which creates some separation between containers as well as between containers and the host operating system. Thus, if the container is compromised, an attacker can only operate within that container namespace, and this isolation limits their his to potentially compromise the host or other containers.





```
piVersion: v1
kind: Pod
metadata:
name: test-pod
spec:
hostIPC: true
hostNetwork: true
hostPID: true
containers:
- name: nginx
image: nginx
```

As shown in Figure 4-02, if the setting called a hostIPC= true is set to true, containers will use the host's interprocess communication namespace. Interprocess communication is just a feature of Linux that allows processes to communicate. Normally, our containers use a separate IPC namespace, which means there is no way for a container process to communicate with other processes on the host or with other containers. This limits the potential for an attacker to utilize that to potentially interact with and compromise other system components. So, if we set host IPC to true, the containers in this Pod will be utilizing the host IPC namespace. However, we must avoid this setting because the isolation provided by having a separate namespace is beneficial to security. Another setting, called hostnetwork, controls the network namespace, where we have hostPID. If we set it to true, containers will use the host process ID namespace. All three of these settings instruct our containers to use the host namespace in each one of those different areas rather than using their own separate isolated namespace. As all of these settings are set to false, by default, if you do not specify any of those three settings, you can rest easy knowing that your containers are properly isolated using their namespaces rather than the host namespace. Thus, the most important thing to remember is to use hostIPC, hostNetwork, and hostPID only when necessary. Do not use settings unnecessarily because it is good from a security standpoint to have that isolation and not use the host namespace.

IAM Roles:

IAM roles help provide permissions and credentials that users and applications can use to access resources within the Amazon Web Services cloud platform.

Applications must use AWS credentials to sign API requests. As a result, if you are an application developer, you will need a strategy for managing credentials for your EC2 applications. For example, you can securely distribute your AWS credentials to instances, allowing those instances' applications to use your credentials for signing requests while keeping your credentials safe from other users. However, securely distributing credentials to each instance, particularly those created by AWS on your behalf, such as Spot instances or instances in Auto Scaling groups. When you rotate your AWS credentials, you must also update the credentials on each instance.

We created IAM roles so that your applications can make secure API requests from your instances without you having to manage their security credentials. You can delegate permission to make API requests using IAM roles instead of creating and disseminating your AWS credentials, as follows:

1. Make a role for IAM.
2. Define which AWS accounts or services are allowed to play the part.
3. After assuming the role, specify which API actions and resources the app can use.
4. When launching your instance, specify the role, or attach it to an existing one.
5. Use a set of temporary credentials that the application retrieves.

Instance profiles

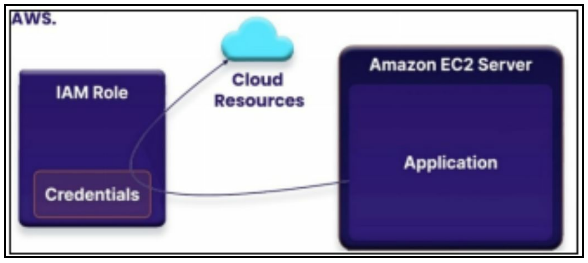
An instance profile on Amazon EC2 serves as a container for an IAM role. The role and the instance profile are created as separate actions with possibly different names when using the AWS CLI, API, or SDK to create a role. When you use the IAM console to create an IAM role, it automatically creates an instance profile with the same name as its corresponding role. You can select an IAM role from a list of instance profile names when using the Amazon EC2 console. This launches an instance with an IAM role or attaches an IAM role to an instance.

If you create a role with the AWS CLI, API, or SDK, the role and instance profile are created as separate actions with potentially different names. Specify the instance profile name if you use the AWS CLI, API, or an AWS SDK to launch an instance with an IAM role or attach an IAM role to an instance.

Why do we need to worry about IAM roles if we run Kubernetes in the Amazon cloud?

IAM roles are designed to allow applications to access Amazon cloud resources.

For example, we have an application in Figure 4-03 running inside an Amazon EC2 server. EC2 is a virtual machine that is running within the Amazon cloud. Our application is running on that server, and we have this IAM role with some credentials. The application can retrieve those credentials and use them to access Amazon cloud resources.



Why is this important for Kubernetes?

Kubernetes containers running on AWS may be able to access IAM credentials. Like any other application running on our Amazon EC2 server, our Kubernetes containers might be able to retrieve those credentials and interact with resources in the Amazon cloud.

Security best practices in IAM

- *Lock away your AWS account root user access keys*

To make programmatic requests to AWS, you need an access key (an access key ID and a secret access key). Instead, you can use your AWS account's root user access key, which grants full access to all your resources, including billing information, for all AWS services. The root user access key permissions associated with your AWS account cannot be reduced.

- *Use roles to delegate permissions*

You can get a temporary credentials role session by using AWS Security Token Service operations or switching to a role in the AWS Management Console. Using your long-term password or access key credentials is less secure. A session is only active for a certain amount of time, which reduces your risk if your credentials are stolen.

Use IAM role temporary credentials to access the resources you need to do your job as a best practice (granting least privilege). AWS Single Sign-On allows users from your external identity source to access AWS resources in your accounts. You can configure an IAM role within a single account to allow identities from a SAML or web identity source to assume the role.

- *Grant least privilege*

  When creating IAM policies, follow the standard security advice of granting the least privilege or only the permissions needed to complete a task. Determine what users (and roles) need to do, and then create policies that only allow them to do that.
  Begin with the bare minimum of permissions and gradually increase them as necessary. This is considered as an efficient approach.
  You can use AWS-managed policies or policies with wildcard * permissions as an alternative to least privilege to get started with policies. Consider the security implications of giving your principals more permissions than they require to perform their duties. Keep an eye on those principals to see which permissions they utilize, and write policies with the fewest privileges.

- *With AWS-managed policies, you can get started with permissions*

  We recommend using the least privileged policies or granting only the permissions needed to complete a task. Writing a custom policy with only your team's permissions is the safest way to grant the least privilege. You will need to set up a system that allows your team to request additional permissions as needed. Creating IAM customer-managed policies that give your team only the permissions they need takes time and expertise.
  You can use AWS-managed policies to add permissions to your IAM identities (users, groups of users, and roles). AWS-managed policies are available in your AWS account and cover common use cases but do not

grant the real east privilege permissions. You should think about the security implications of giving your principals more permission than they require to do their job.

AWS-managed policies, including job functions, can be attached to any IAM identity. You can use AWS Identity and Access Management Access Analyzer to monitor the principals with AWS-managed policies and switch to least privilege permissions. After you have figured out which permissions they are using, you can create a custom policy or generate a policy that only gives your team the permissions they need. This is less secure, but it gives you more flexibility as your team learns to use AWS.

- *Validate your policies*

  Validating the policies you create is one of the best practices that you should use. When you create and edit JSON policies, you can perform policy validation. IAM detects JSON syntax errors, while IAM Access Analyzer performs over 100 policy checks and provides actionable recommendations to assist you in writing secure and functional policies. According to our recommendations, all of your existing policies should be reviewed and validated.

- *Block access to IAM*

  If your Kubernetes applications do not need to use IAM, consider blocking access to IAM credentials. When it comes to Amazon EC2, you just want to block access to this IP address 169.254.169.254.

Kubernetes Cluster Network

Kubernetes uses a virtual cluster network that allows Pods to communicate with other Pods and Services freely and transparently regardless of their node. Although networking is an important part of Kubernetes, it can be difficult to grasp exactly how it works.

Kubernetes is all about sharing machines between applications. In most cases, sharing machines necessitates ensuring that no two applications use the same ports. Coordinating ports across multiple developers is extremely difficult at scale, and users are exposed to cluster-level issues beyond their control.

The system is complicated by dynamic port allocation because every application must use ports as flags. API servers must know how to insert dynamic port numbers into configuration blocks, and services must know how to find each other, etc. Kubernetes, rather than dealing with this, takes a different approach.

How to implement the Kubernetes networking model

The Kubernetes network model can be implemented in a variety of ways; a few of them are as follows:

ACI

Cisco Application Centric Infrastructure (ACI) supports containers, virtual machines, and bare-metal servers with an integrated overlay and underlay SDN solution and provides container networking integration.

Antrea

Project Antrea is a Kubernetes networking solution that is open-source and Kubernetes-native. The networking data plane is based on Open vSwitch, which is a high-performance programmable virtual switch with Linux and Windows support. Antrea can use Open vSwitch to implement Kubernetes Network Policies with high performance and efficiency. It can implement an extensive set of networking and security features and services on top of Open vSwitch, thanks to the "programmable" feature of Open vSwitch.

AWS VPC CNI for Kubernetes

For Kubernetes clusters, the AWS VPC CNI provides integrated AWS Virtual Private Cloud (VPC) networking. High throughput and availability, low latency, and minimal network jitter are all features of this CNI plugin. Users can also build Kubernetes clusters using existing AWS VPC networking and security best practices. This includes the ability to isolate network traffic using VPC flow logs, VPC routing policies, and security groups.

Kubernetes pods with this CNI plugin have the same IP address inside the pod as they do on the VPC network. The CNI assigns AWS Elastic Networking Interfaces (ENIs) to each Kubernetes node, with pods on the node using the secondary IP range from each ENI. For fast pod startup times, the CNI includes controls for pre-allocating ENIs and IP addresses and supports large clusters of up to 2,000 nodes.

AppArmor

AppArmor is a Linux security kernel module that provides granular access control for programs running on Linux systems. You can use it to control and limit what a program can do within the host operating system. You can essentially think of it as granular access control that applies to every process or program running within your Linux system. An AppArmor profile is a set of rules that define what a program can and cannot do.

Seccomp

Seccomp (secure computing mode) has been part of the Linux kernel since version 2.6.12. It can be used to sandbox a process's privileges by limiting the calls it can make from userspace to the kernel. Kubernetes allows you to apply seccomp profiles put onto a node to Pods and containers automatically.

Chapter 05: Minimizing Microservice Vulnerabilities

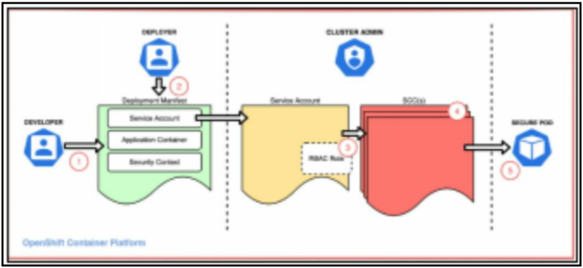
Introduction

You will learn about minimizing microservice vulnerabilities in this chapter. That means taking care of the security of your actual applications running inside your cluster, particularly microservice applications that consist of multiple individual Pods and containers talking to each other.

You will also learn about security contexts and some special security settings that you can set at the Pod and container levels. The chapter also focuses on Pod security policies and the Kubernetes objects that will allow you to enforce secure Pod configurations within your cluster.

Managing Container Access with Security Contexts

A security context is simply a portion of the Pod and container specification that allows you to provide special security and access control settings at the Pod and container level. When working with security contexts, always pay attention to whether you work at the Pod or container levels. It can get confusing because the Pod and container security contexts differ quite slightly regarding their options.



First, the security context offers a variety of security and access control-related settings. Set security context settings at the pod level; these settings apply to all containers in the Pod and spec. containers.securityContext sets the securityContext settings at the container level. These settings apply only to individual containers within the Pod. It is a good idea to make sure that you pay attention to the difference between the security context at the pod level directly under the spec and the security context at the container level within an individual container.

Governing Pod Configurations with Pod Security Policies

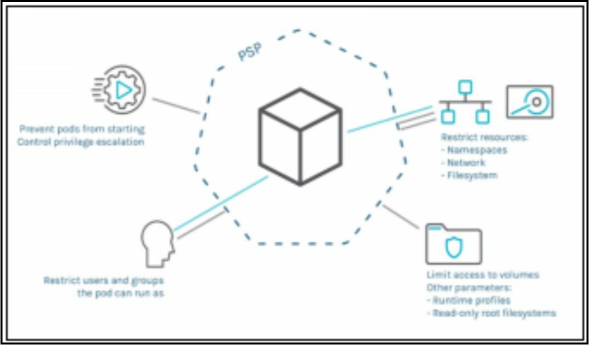
When you have multiple different people deploying applications to your cluster, what if not all of them are security specialists who accidentally use some insecure configuration when deploying Pods to the cluster?

This is the scenario that Pod security policies are designed to prevent. Pod security policies allow cluster administrators to control what security-related configurations Pods can run with. So essentially, you can use Pod security policies to automatically enforce the desired security configurations on your Pods within the cluster.

How Does PodSecurityPolicy Work?

Pods cannot use privileged mode for containers. They can allow privilege escalation, but they cannot run as root. There are certain allowed volume types that are not allowed to access the host network directly. They allow you to specify a list of dos and don'ts. Then, when users try to create Pods, the PodSecurityPolicy verifies that the Pods meet those criteria before allowing them to be created in the cluster. But Pod security policies can do a little more than that. They can also change Pods by providing defaults for certain values. This allows you to enforce certain configurations transparently for your users.

You can provide a default and prevent a user from doing something different from the default. But they can do things like allow or disallow running containers in privileged mode. They can prevent Pods from using host namespaces. Pod security policies are being deprecated, and they will be replaced by different Kubernetes functionality in the future.


PSP
Prevent pods from starting
Control privilege escalation
Restrict resources:
- Namespaces
- Network
- Filesystem
Restrict users and groups the pod can run as
Limit access to volumes
Other parameters:
- Runtime profiles
- Read-only root filesystems

You can use Pod security policies to enforce desired security configurations for new Pods. Pod security policies can reject Pods that do not meet the desired standard or modify Pods by applying default settings.

Pod Security Policy

A Pod Security Policy is a cluster-level resource that manages the pod specification's security-sensitive components. The PodSecurityPolicy objects establish a set of conditions that must be met by a pod in order for it to be admitted into the system, as well as defaults for the variables involved.

Using OPA Gatekeeper

The OPA stands for Open Policy Agent, and OPA Gatekeeper allows you to enforce highly customizable policies on any Kubernetes object at creation time. Policies are defined using the Open Policy Access Constraint Framework. Essentially, OPA is a framework for defining constraints and requirements around objects, and OPA Gatekeeper allows us to use that tool to define policies for object creation in Kubernetes.

Managing Kubernetes Secrets

Secrets store and manage sensitive passwords, API tokens, and certificates. This sensitive data can be passed into containers at runtime. If you have some sensitive configuration data that you need to pass into your containers, such as a password, you want to use a Secret to manage that data. This is the basic YAML for a Secret, and this particular Secret just contains some account credentials, a username, and a password. The type of the Secret is important, and it essentially tells Kubernetes what type of data is stored in the Secret. Opaque is just the default and allows arbitrary user-defined data. You can put any kind of data into an Opaque Secret.

Security Context

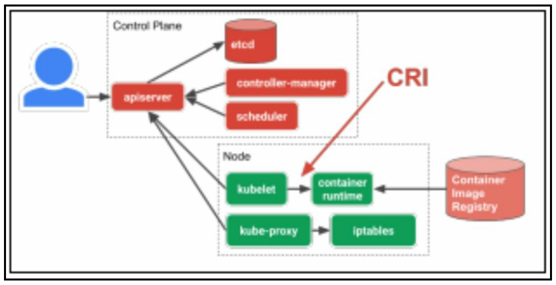
For a Pod or Container, a security context defines the privilege and access control settings. The following are examples of security context settings:

- Permission to access an item, such as a file, is granted based on the user ID (UID) and group ID (GID).
- Security Enhanced Linux (SELinux) assigns security labels to objects.
- Whether you are privileged or not.
- Give a process part, but not all, of the root user's rights using Linux capabilities.

- Use program profiles to limit the functionality of specific apps with AppArmor.
- Filter the system calls of a process with Seccomp.
- allowPrivilegeEscalation: Determines whether or not a process can earn greater privileges than its parent. This bool determines whether or not the container process's no_new_privs flag is set. when the container: allowPrivilegeEscalation is always true
  - run as privileged
  - has CAP_SYS_ADMIN
- readOnlyRootFilesystem: Mounts the root filesystem of the container as read-only.

Understanding Container Runtime Sandboxes

A container runtime sandbox is a specialized container runtime providing extra layers of process isolation and greater security. You have your normal container runtime, where most of your containers are running, and a runtime sandbox with additional security protections. Containers where you have some reason to be concerned about their security level, you can run those inside that special sandbox.



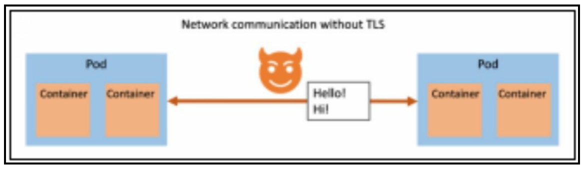
Creating a Container Runtime Sandbox

You can use the RuntimeClass to decide which Pods will use sandboxed runtime. Once you have created your RuntimeClass, you can use the RuntimeClass name attribute inside the Pod spec to instruct Kubernetes to utilize that particular RuntimeClass for your Pod and therefore cause the Pod to use the runtime sandbox.

You can use a RuntimeClass to define a specialized container runtime configuration, such as using gVisor/runsc. Also, you can set the runtimeClassName property in a Pod specification to make the Pod use that container runtime sandbox.

Understanding Pod-to-Pod mTLS

You can use something called mTLS, or Mutual Transport Layer Security. mTLS means that both communicating parties fully authenticate and encrypt all communications. That would block a user from listening in on those communications. You are probably used to seeing HTTPS on the web, a one-sided TLS, where the server has a certificate to authenticate, but the client does not have a certificate. With Mutual Transport Layer Security, both the client and the server have certificates. So both sides, both parties in that communication have a certificate, which prevents anyone from listening in or impersonating either the client or the server. mTLS is a way of preventing anyone from impersonating clients or servers or listening in on the communication between two Pods. If servers and clients both have certificates, you will need a lot of certificates in your cluster if you have a lot of different microservice applications communicating with each other.



Kubernetes API allows you to obtain certificates that you can use in your cluster applications. These certificates provided by the API are generated from a central Certificate Authority or CA, which you can use for trust purposes to make sure that all of your applications trust one another's certificates. Most importantly, you can programmatically access these certificates using the API. You can imagine building your tools and automation around these certificate-related features of Kubernetes to help manage certificates and implement Mutual TLS throughout your applications. Mutual Transport Layer Security, or mTLS, means clients and servers authenticate and encrypt their communications. You can obtain certificates using the Kubernetes API.

Signing Certificates

The process of requesting and signing certificates is simple. If you are familiar with the normal certificate management process, you create a Kubernetes object called a CertificateSigningRequest, which contains an actual standard CSR. That will be a Kubernetes object that represents a request for a new certificate. Then the CertificateSigningRequest can be approved or denied, and you can control who is allowed to approve or deny requests using normal role-based access control.

You can create a CertificateSigningRequest object in Kubernetes to request a new certificate. You can manage, approve, or deny requests via the command line with the kubectl certificate command. Once approved, the signed certificate can be retrieved from the status. certificate field of the Certificate Signing Request.

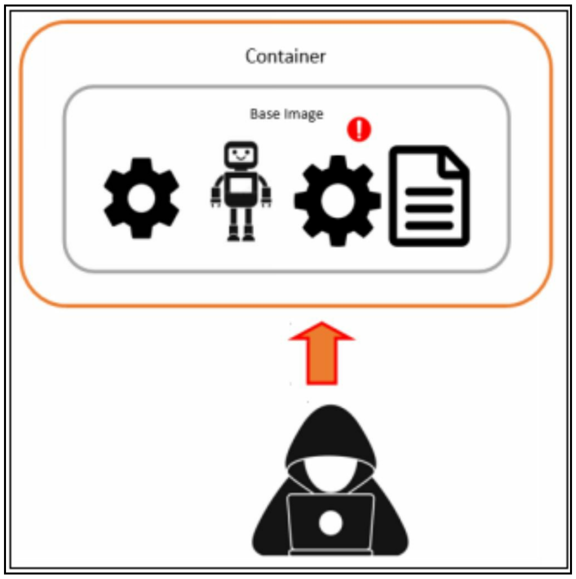
Chapter 06: Supply Chain Security

Supply Chain Security Introduction

In this lesson, we will discuss potential threats posed by images and some of the techniques for minimizing those threats. We will also look into the concept of whitelisting image registries, allowing images from only pre-approved registries. Furthermore, the chapter will focus on image validation, which is the process of validating signed images to ensure that the content is not tempered.

Minimizing Base Image Attack Surface

Our container images come with a lot of different software components. An attacker can exploit any of those components having vulnerabilities. If we want to protect ourselves, we need to ensure that the software components inside our images are not creating a method for a hacker to break in.



Software Vulnerabilities

A software vulnerability is a flaw or weakness in a piece of software that an attacker can use. When possible, use images that contain up-to-date software with the latest security patches. The creators of that software usually release patches that contain fixes for those software vulnerabilities that have been discovered.

Minimizing Unnecessary Software

Some images may not be well-designed and may contain additional, unneeded software. Additional software comes with an additional risk of vulnerabilities. It is a good idea to minimize the amount of unnecessary software in the images. Attackers may purposefully create images that contain malicious software. Ensure you are getting your software from a trusted source so that we are not using a container image with malicious software intentionally put there by an attacker.

Whitelisting Allowed Image Registries

Image registries are extremely convenient. They make it easy to download container images to your cluster. What if some of your users download images from registries you do not trust? An attacker might have control over some of the images on those untrusted registries. If an attacker has control of images and we are downloading and using those images, then we are running malicious code. This is a method for an attacker to gain some control over the cluster.

Image Registry

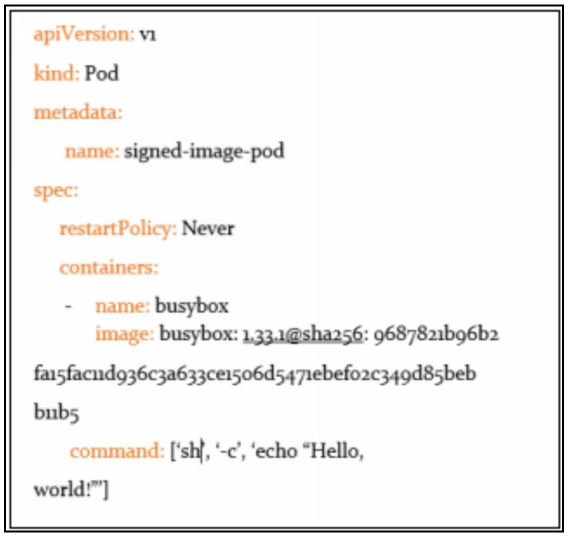
An image registry is a service that stores container images and provides them for download. Whenever we run a container in Kubernetes, the node automatically downloads the images from a registry. To ensure our users are not downloading images from untrusted registries that might have some malicious containers, we can restrict which registries users can download images from. One way to do this is to use OPA Gatekeeper.

Validating Signed Images

Container images can be signed using a hash generated from the unique contents of the image. These signatures can be used to verify that the contents of the image have not been tampered with. Image signatures allow us to ensure that an attacker has not added malicious software or code into an image after it was initially created.

Validating Images

To make use of image signatures, we need to validate our images. We can supply a SHA256 hash when we are referencing images in Kubernetes. The example below shows that we can append the hash to the end of the image reference with @sha256:<hash-value>. If the image fails validation, the pod will be created, but the node will fail to download the image. If that image has been tampered with, validation will prevent potentially malicious code from running in the cluster.



```
apiVersion: v1
kind: Pod
metadata:
    name: signed-image-pod
spec:
   restartPolicy: Never
   containers:
   -    name: busybox
        image: busybox: 1.33.1@sha256: 9687821b96b2
fa15fac11d936c3a633ce1506d5471ebef02c349d85beb
b11b5
     command: ['sh', '-c', 'echo "Hello,
world!"']
```

Analyzing a DockerFile

Static Analysis is the process of looking at source code for configuration to identify potential security issues. One way to harden the security of their images is to analyze the DockerFiles to create them. Some tools do automatic static analysis, but we can perform it manually as well.

When you are looking at DockerFiles, look for user root. USER directive in DockerFile sets which user will be used to run the steps used to build the container image. There can be multiple USER directives. The final USER directive in DockerFile also sets which user the container will run as by default and at runtime. Another thing to avoid is :latest tag in the FROM directive at the top of a DockerFile that is where you are setting which base image you are building your new image off. It is generally a good idea to avoid using the :latest tag. When you use the :latest tag, it is not necessary for you to always be fully aware of which version of the base image you are using. Suppose an attacker somehow uploaded a new version of the image to the registry. In that case, you could be pulling that new version rather than an old one that you have already vetted and making sure that the specific version is secure. Generally, in a DockerFile, it is good practice to reference a specific version to tag. If you need an update, update that specific version rather than using the :latest tag.

Ensure the DockerFile does not install unnecessary software or tools in the final image that you are not using in your container. Verify that no sensitive data, such as passwords or API keys, are stored in the image. Instead, use Kubernetes secrets to pass sensitive data to the container at runtime.

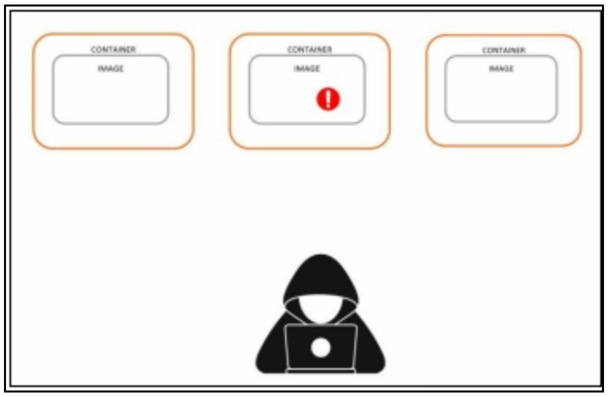
Analyzing Resource YAML Files

We can also perform static analysis on Kubernetes resources simply by examining those YAML manifest files that we often use to create objects in our Kubernetes cluster.

You need to look for host namespaces in YAML files, do not let containers use host namespaces if possible. Avoid using privileged mode for containers unless it is necessary. Avoid using :latest tag. You could potentially end up running a new container version that you have not vetted. You can set which user the container will run as, and you want to avoid using the root user or user ID 0, which also refers to the root.

Scanning Images for Known Vulnerabilities

If you have a lot of images, there are likely some vulnerabilities that a hacker can use.



Software Vulnerability

A software vulnerability is a flaw or weakness in a piece of software that an attacker can use. Security researchers are constantly working to discover and document vulnerabilities. We use the concept of vulnerability scanning to locate these known vulnerabilities. Vulnerability scanning means using tools to detect known vulnerabilities in your software. In Kubernetes, image scanning means scanning container images to see if they include vulnerable software.

Trivy

One tool that allows us to do image scanning in Kubernetes is Trivy. Trivy is a command-line tool that allows you to scan container images for vulnerabilities. We can use Trivy using trivy image <name-of-the-image>, and it will go and scan the image. The older version of Trivy uses trivy <name-of-the-image> command. If the newer version does not work, try an older one. Trivy generates a report that lists known vulnerabilities found in the software contained in a container image. We have a column called LIBRARY; it tells which piece of software is affected by the vulnerability that was located. The second column is of VULNERABILITY ID, a unique identifier for the vulnerability. You can use that to look for detailed information about vulnerability using that ID. We have a SEVERITY column that indicates the level of security risk created by the vulnerability.

Admission Controller

Admission controllers intercept requests to the Kubernetes API, and they can approve, deny, or modify the request before changes are made in the cluster. So essentially, when you are trying to change something through the Kubernetes API, an admission controller gives you a way to have some additional control of what happens in that scenario. You can prevent something from being created and allow it to be created, and you can even make changes to the request before making changes in the cluster.

Why Do We Need An Admission Controller?

An admission controller must be enabled to support many advanced features in Kubernetes properly. A Kubernetes API server that is not properly configured with the appropriate admission controllers is an incomplete server that will not provide all the features you require.

ImagePolicyWebhook

A backend webhook can use the ImagePolicyWebhook admission controller to make admission decisions.

What Does Image Policy Webhook Do?

Whenever you try to create a workload, like a Pod, it sends a request to an external webhook containing information about the image you are trying to use. This external webhook can then approve or deny the creation of the workload based upon the image. So essentially, this ImagePolicyWebhook allows you to control which images are allowed to run in your cluster. One of the things we can do with that, uses this functionality to automatically scan images and deny workloads if there are severe vulnerabilities.

Chapter 07: Monitoring, Logging, and Runtime Security

Introduction

In this chapter, we will be focusing on monitoring, logging, and runtime security. We will discuss the concepts of container immutability, how to configure immutable containers within Kubernetes and audit logging.

Behavioral Analytics

Behavioral analytics is the process of observing what is going on in a system, i.e., our Kubernetes cluster, and identifying abnormal or potentially malicious events.

Behavioral analytics could be done manually; we could log into our servers and see what is happening. However, it is not very efficient or effective, making it impossible to keep track of everything all the time manually. Therefore, with the help of tools, we can perform behavioral analytics. In this chapter, we will be focusing on a specific tool known as **Falco**.

Falco

Sysdig Inc. created the Falco Project, an open-source runtime security tool. Falco was given to the Cloud Native Computing Foundation (CNCF) and is now being developed as a CNCF incubating project.

What does Falco do?

Falco secures and monitors a system using system calls by:

- Using a powerful rules engine to assert the stream
- At runtime, parsing Linux system calls from the kernel
- When a rule is broken, an alert is sent out.

Falco is constantly watching everything that is going on in our system. We can build these rules that will alert us when suspicious activity occurs, allowing us to use Falco to detect and quickly respond to suspicious activity.

What does Falco check for?

Falco comes with a set of default rules that look for unusual behavior in the kernel, such as:

- Using privileged containers for privilege escalation
- Using tools like setns; you can change the namespace
- /etc, /usr/bin, /usr/sbin, and other well-known directories are read/written to
- Making a symlink
- Changes in ownership and mode
- Network connections or socket mutations that are not expected
- execve was used to spawn processes
- Shell binaries such as sh, bash, csh, zsh, and others are executed.
- SSH binaries such as ssh, scp, sftp, and others are executed
- Changing the coreutils executables in Linux
- Changing the login binaries
- Shadowutil and passwd executables can be modified, such as shadowconfig, pwck, chpasswd, getpasswd, change, useradd, and others

What are Falco rules?

The items Falco asserts against are called the rules, which represent the events you can check on the system and are defined in the Falco configuration file.

What are Falco alerts?

Alerts are user-configurable downstream actions ranging from logging to STDOUT to delivering a gRPC call to a client. Using the gRPC API, Falco Alerts has more information on configuration, understanding, and development. Falco can send alerts to the following addresses:

- Typical Output
- Syslog
- A file
- A program that has been spawned
- Endpoint for HTTP[s]

What are the components of Falco?

Falco is made up of three main parts:

- Falco's CLI tool, the userspace program, can interact with Falco. Signals are handled by the userspace program, which also parses information from a Falco driver and sends alerts.

- Falco's configuration determines how it is run, what rules are asserted, and how alerts are handled.
- A driver is a piece of software that follows the Falco driver specification and sends a stream of system call data. Falco will not run without a driver installed; the following drivers are currently supported by Falco:
    - (Default) Kernel module based on the C++ libraries libscap and libsinsp
    - BPF probe based on the same modules
    - Instrumentation in the userspace
- Plugins - allow users to extend the Falco libraries/falco executable functionality by adding new event sources and fields to extract data from events.

What is Container Immutability?

Immutable or stateless containers do not change during their lifetime, so they are replaced with new containers instead of being changed. The container file system remains static, and the container does not depend on non-immutable host resources that require privileged access.

Container immutability is a container that cannot change its code during runtime or depend on stateful host-level components that require privileged access.

Why is Immutability Important When it Comes to Containers?

Immutability has certain security benefits; for example, when a container cannot change its code, an attacker cannot download malicious software or tools into a container or alter the container's code during runtime. Essentially, container immutability is about limiting what an attacker might do if they get control of that container.

Immutability Best Practices

There are a few best practices when it comes to making sure and verifying that containers are immutable:

Privilege Mode

The first best practice is to avoid elevated privileges. If containers depend on host-level resources or elevated privileges, they can use those to maintain the state. Those things are bad for security in general, but they can also make your containers considered no longer immutable.

if you are using **securityContext.privileged: true**, your container may not be considered immutable.

Configuring Audit Logging

You can configure audit logging by passing flags to kube-apiserver.

• --**audit-policy-file** - Points to the audit policy config file

• **--audit-log-path**- The location output files

• --**audit-log-maxage** - The number of days to keep old log files

Host Namespaces

Settings like hostNetwork: true can make the container mutable.

Allow Privilege Escalation

This setting allows a container to gain more privileges than the parent process, making a container mutable.

Root User

Suppose you have **securityContext.runAsUser: root**, or user ID 0, which is the user ID for root, that makes the container process run as root. You should probably use a different user to maintain Immutability.

Some other best practices are as follows:

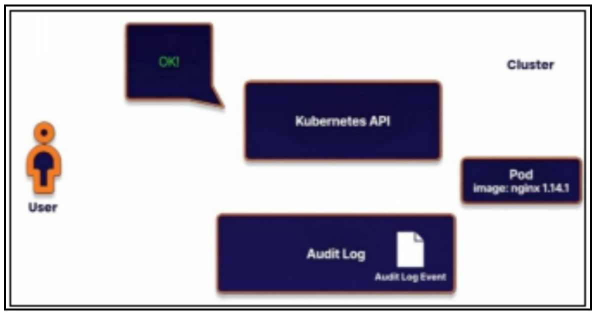
- Immutable containers do not change their code, so they should not write to the container file system
- Use **readOnlyRootFilesystem: true to** enforce this. A container without this setting might not be considered immutable
- If the application does need to write data, use volume mounts to support this

What Are Audit Logs?

Audit logs are chronological record actions performed through the Kubernetes API. They are useful for seeing what is happening in your cluster in real-time (i.e., threat detection) or examining what happened after the fact (i.e., post-mortem analysis).

How do Audit Logs Work?

Consider the following example to understand how audit logs work? We have a user coming to the Kubernetes API and creating a Pod. After the Kubernetes API, the Pod is created, and simultaneously, an audit log event is generated and logged to the audit log. That is going on as users or automated processes interact with the Kubernetes API. The API is logging everything happening to that audit log on the backend, and configuring audit logs; you will need to use an **audit policy**.



Audit Policy

The audit policy establishes guidelines for what events should be recorded and what information should be included. The audit.k8s.io API group defines the audit policy object structure, and when an event is processed, it is compared to a list of rules in chronological order. The first matching rule determines the event's audit level. The audit levels are as follows:

The audit policy includes a set of rules that determine which events are logged and how detailed the logs are.

Level 1 - How detailed are the rule's logs. They can be:

- **None** - Log nothing
- **Metadata** - Log only high-level data
- **Request** - Log the metadata and the request body
- **resources** - Matches Kubernetes object types with the applicable rule
- **namespaces** -(Optional) limits the rule to one or more namespaces