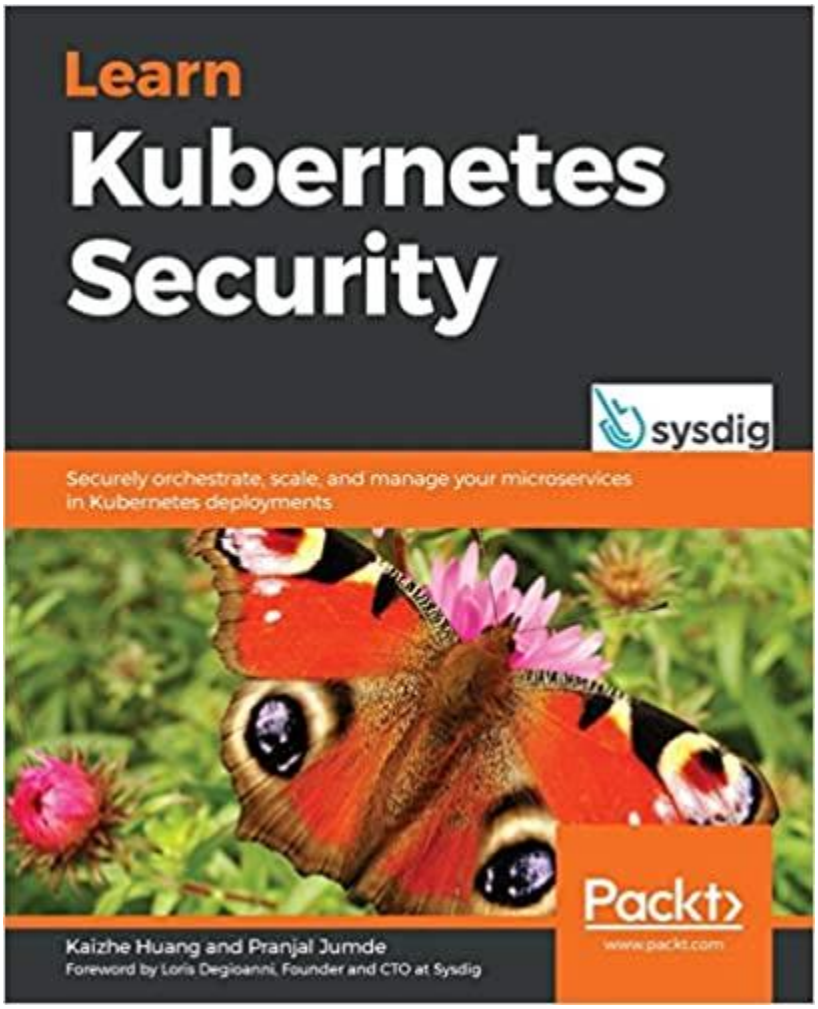
Book review for Learn Kubernetes Security

By Kaizhe Huang, Pranjal Jumde

July 2020

URL = https://www.amazon.com/Learn-Kubernetes-Security-orchestrate-microservices/dp/1839216506/ref=sr_1_1_sspa

https://www.packtpub.com/product/learn-kubernetes-security/9781839216503



Summary: Good k8s security book. I like Chap 11 (Falco, Vault, and Audit Policy) most. Section 2 (chap 6-11) provides a lot of tips to secure cluster components, AuthN, AuthZ, Admission control, secure pods, image scans, PSP, OPA, Anchore Engine. Section 3 (chap 12+13) shows some CVEs and Crypto-Mining attacks. Good to know more in k8s security!

Some interesting topics, e.g. what is a Pause Container? Top 3 challenges for Kubernetes Users ? Compare 3 Major clouds k8s engine (EKS vs GKE vs AKS), Threat Actors and Modeling, security boundaries, …etc.

Suggestions: some links and comparisons are old and then outdated, e.g. CNI comparison table was updated recently in 2020.

K8s as a Service: good cloud providers compare, but quite out-dated because this was done in 2018, see https://kubedex.com/google-gke-vs-microsoft-aks-vs-amazon-eks/
And https://docs.google.com/spreadsheets/d/1U0x4-NQegEPGM7eVTKJemhkPy18LWuHW5vX8uZzqzYo/edit#gid=0

For more details. Still good to see the compare criteria but readers need to adjust to update with latest.

free open source project.

Kubernetes and cloud providers

A lot of people believe that Kubernetes is the future of infrastructure, and there are some people who believe that everything will end up on the cloud. However, this doesn't mean you have to run Kubernetes on the cloud, but it does work really well with the cloud.

Kubernetes as a service

Containerization makes applications more portable so that locking down with a specific cloud provider becomes unlikely. Although there are some great open source tools, such as `kubeadm` and `kops`, that can help DevOps create Kubernetes clusters, Kubernetes as a service offered by a cloud provider still sounds attractive. As the original creator of Kubernetes, Google has offered Kubernetes as a service since 2014. It is called **Google Kubernetes Engine** (**GKE**). In 2017, Microsoft offered its own Kubernetes service, called **Azure Kubernetes Service** (**AKS**). AWS offered **Elastic Kubernetes Service** (**EKS**) in 2018.

Kubedex (`https://kubedex.com/google-gke-vs-microsoft-aks-vs-amazon-eks/`) have carried out a great comparison of the cloud Kubernetes services. Some of the differences between the three are listed in the following table:

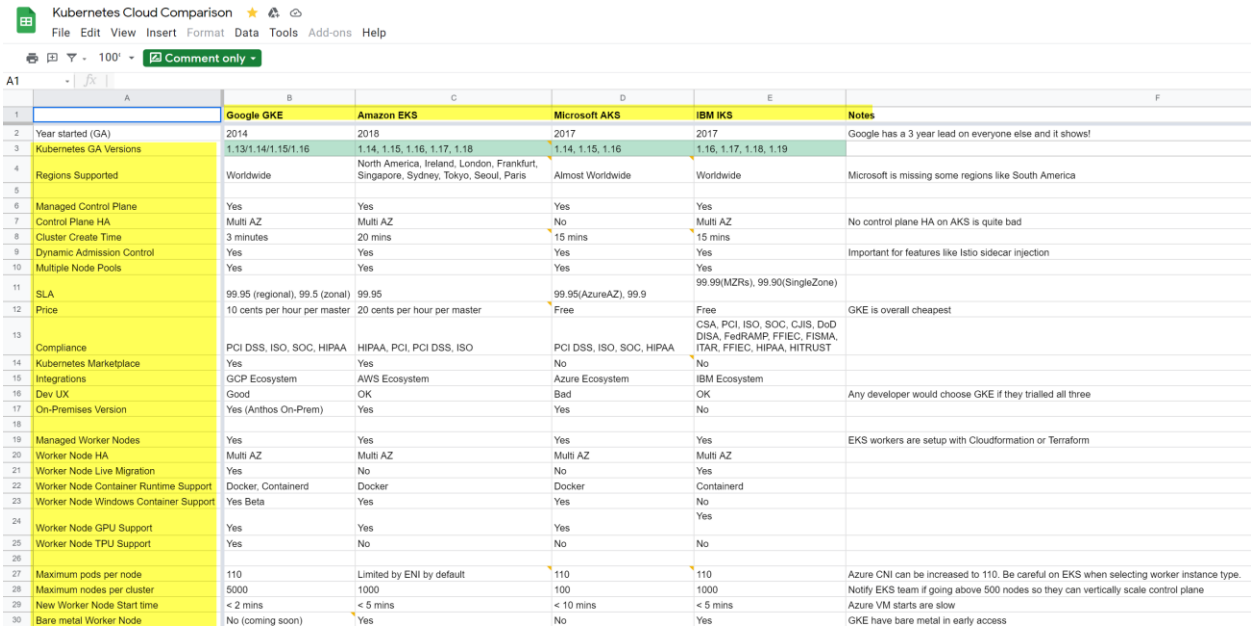
Parameters	Google GKE	Amazon EKS	Microsoft AKS
Year started (GA)	2014	2018	2017
Kubernetes GA Versions	1.14/1.15	1.14	1.13 and 1.14
Regions Supported	Worldwide	North America, Ireland, London, Frankfurt, Singapore, Sydney, Tokyo, Seoul, and Paris	Almost worldwide
Cluster Create Time	3 minutes	20 minutes	15 minutes
Price	Free	20 cents per hour per master	Free

Parameters	Google GKE	Amazon EKS	Microsoft AKS
Kubernetes Marketplace	Yes	No	No
Integrations	GCP Ecosystem	AWS Ecosystem	Azure Ecosystem
Managed Worker Nodes	Yes	No	Yes
Application Layer Secret Encryption	Yes	No	No
Network Policy Support	Yes (Calico)	Yes (Calico)	Yes (`kube-router` and Calico)
Monitoring Integration	Yes (Stackdriver)	Yes (Container Insights)	Yes
Sandbox support (for example, gVisor)?	Yes (beta)	No	No
Binary Authorization Support	Yes (beta)	No	No
Ingress managed SSL Certificate	Yes	No	No
Release Channels	Yes (stable, regular, and rapid)	No	No
Compliance	PCI DSS, ISO, SOC, HIPAA	HIPAA, PCI, PCI DSS, ISO	PCI DSS, ISO, SOC, HIPAA
Maximum nodes per cluster	5000	500+	100
Cross region load balancing	Yes	No	No

Some highlights worth emphasizing from the preceding list are as follows:

- **Scalability**: GKE supports up to 5,000 nodes per cluster, while AKS and EKS only support a few hundred nodes or less.

Follow the link, then can see more compare including IBM cloud, IKS. This can be outdated, so check before you use it!


Kubernetes Cloud Comparison

File Edit View Insert Format Data Tools Add-ons Help

100' Comment only

A1

	A	B	C	D	E	F
1		**Google GKE**	**Amazon EKS**	**Microsoft AKS**	**IBM IKS**	**Notes**
2	Year started (GA)	2014	2018	2017	2017	Google has a 3 year lead on everyone else and it shows!
3	Kubernetes GA Versions	1.13/1.14/1.15/1.16	1.14, 1.15, 1.16, 1.17, 1.18	1.14, 1.15, 1.16	1.16, 1.17, 1.18, 1.19	
4	Regions Supported	Worldwide	North America, Ireland, London, Frankfurt, Singapore, Sydney, Tokyo, Seoul, Paris	Almost Worldwide	Worldwide	Microsoft is missing some regions like South America
5						
6	Managed Control Plane	Yes	Yes	Yes	Yes	
7	Control Plane HA	Multi AZ	Multi AZ	No	Multi AZ	No control plane HA on AKS is quite bad
8	Cluster Create Time	3 minutes	20 mins	15 mins	15 mins	
9	Dynamic Admission Control	Yes	Yes	Yes	Yes	Important for features like Istio sidecar injection
10	Multiple Node Pools	Yes	Yes	Yes	Yes	
11	SLA	99.95 (regional), 99.5 (zonal)	99.95	99.95(AzureAZ), 99.9	99.99(MZRs), 99.90(SingleZone)	
12	Price	10 cents per hour per master	20 cents per hour per master	Free	Free	GKE is overall cheapest
13	Compliance	PCI DSS, ISO, SOC, HIPAA	HIPAA, PCI, PCI DSS, ISO	PCI DSS, ISO, SOC, HIPAA	CSA, PCI, ISO, SOC, CJIS, DoD DISA, FedRAMP, FFIEC, FISMA, ITAR, FFIEC, HIPAA, HITRUST	
14	Kubernetes Marketplace	Yes	Yes	No	No	
15	Integrations	GCP Ecosystem	AWS Ecosystem	Azure Ecosystem	IBM Ecosystem	
16	Dev UX	Good	OK	Bad	OK	Any developer would choose GKE if they trialled all three
17	On-Premises Version	Yes (Anthos On-Prem)	Yes	Yes	No	
18						
19	Managed Worker Nodes	Yes	Yes	Yes	Yes	EKS workers are setup with Cloudformation or Terraform
20	Worker Node HA	Multi AZ	Multi AZ	Multi AZ	Multi AZ	
21	Worker Node Live Migration	Yes	No	No	Yes	
22	Worker Node Container Runtime Support	Docker, Containerd	Docker	Docker	Containerd	
23	Worker Node Windows Container Support	Yes Beta	Yes	Yes	No	
24	Worker Node GPU Support	Yes	Yes	Yes	Yes	
25	Worker Node TPU Support	Yes	No	No	No	
26						
27	Maximum pods per node	110	Limited by ENI by default	110	110	Azure CNI can be increased to 110. Be careful on EKS when selecting worker instance type.
28	Maximum nodes per cluster	5000	1000	100	1000	Notify EKS team if going above 500 nodes so they can vertically scale control plane
29	New Worker Node Start time	< 2 mins	< 5 mins	< 10 mins	< 5 mins	Azure VM starts are slow
30	Bare metal Worker Node	No (coming soon)	Yes	No	Yes	GKE have bare metal in early access

Why worry about Kubernetes' security?

Kubernetes was in general availability in 2018 and is still evolving very fast. There are features that are still under development and are not in a GA state (either alpha or beta). This is an indication that Kubernetes itself is far from mature, at least from a security standpoint. But this is not the main reason that we need to be concerned with Kubernetes security.

Bruce Schneier summed this up best in 1999 when he said '*Complexity is the worst enemy of security*' in an essay titled *A Plea for Simplicity*, correctly predicting the cybersecurity problems we encounter today (`https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html`). In order to address all the major orchestration requirements of stability, scalability, flexibility, and security, Kubernetes has been designed in a complex but cohesive way. This complexity no doubt brings with it some security concerns.

Configurability is one of the top benefi https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html developers. Developers and cloud providers are free to configure their clusters to suit their needs. This trait of Kubernetes is one of the major reasons for increasing security concerns among enterprises. The ever-growing Kubernetes code and components of a Kubernetes cluster make it challenging for DevOps to understand the correct configuration. The default configurations are usually not secure (the openness does bring advantages to DevOps to try out new features).

Top 3 challenges for Kubernetes Users:

A recent survey by The New Stack (`https://thenewstack.io/top-challenges-kubernetes-users-face-deployment/`) shows that security is the primary concern of enterprises running Kubernetes:

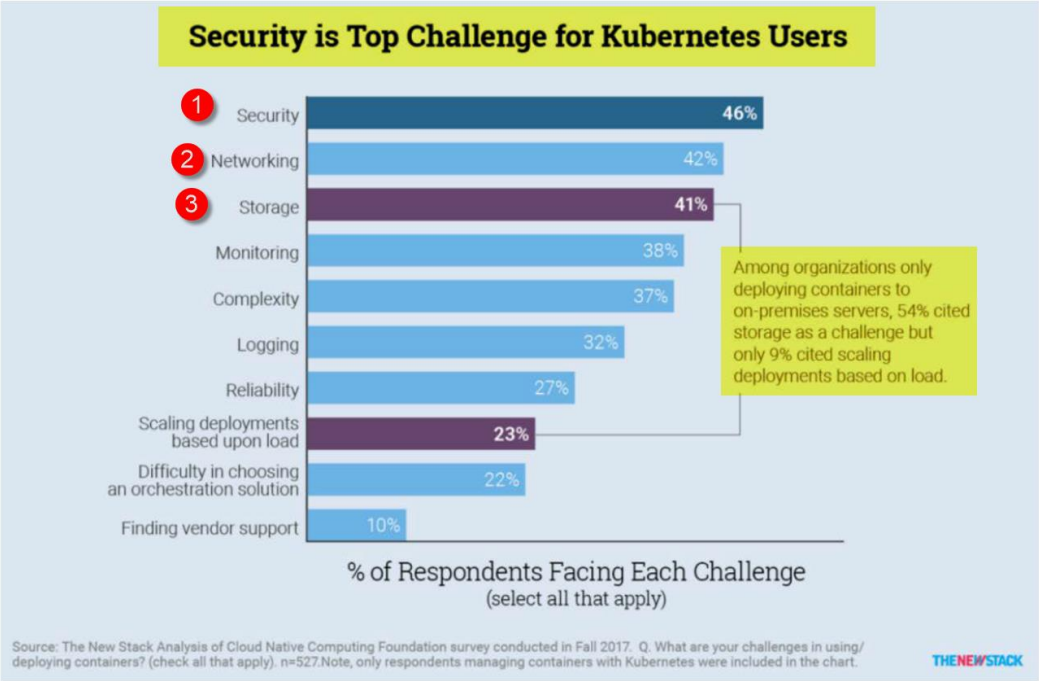


Figure 1.3 – Top concerns for Kubernetes users

What is a Pause container?

Though each of these namespaces is powerful and serves an isolation purpose on different resources, not all of them are adopted for containers inside the same pod. Containers inside the same pod share at least the same IPC namespace and network namespace; as a result, K8s needs to resolve potential conflicts in port usage. There will be a loopback interface created, as well as the virtual network interface, with an IP address assigned to the pod. A more detailed diagram will look like this:

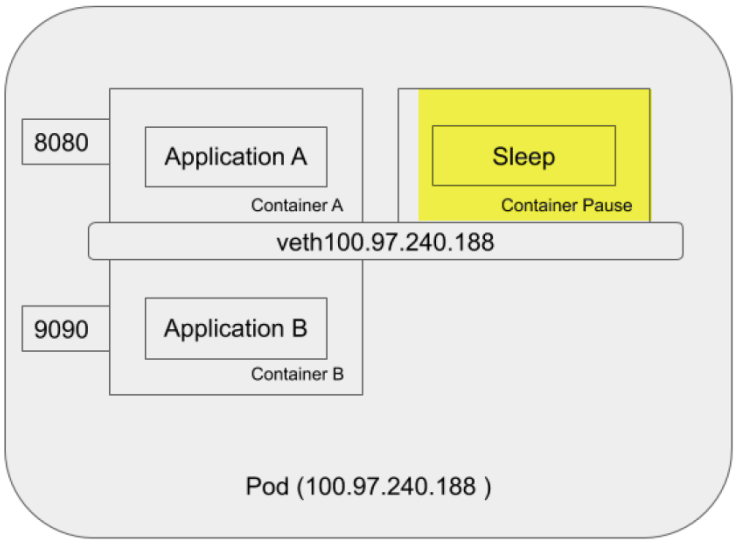


Figure 2.4 – Containers inside a pod

In this diagram, there is one **Pause** container running inside the pod alongside containers **A** and **B**. If you **Secure Shell** (**SSH**) into a Kubernetes cluster node and run the `docker ps` command inside the node, you will see at least one container that was started with the `pause` command. The `pause` command suspends the current process until a signal is received. Basically, these containers do nothing but sleep. Despite the lack of activity, the **Pause** container plays a critical role in the pod. It serves as a placeholder to hold the network namespace for all other containers in the same pod. Meanwhile, the **Pause** container acquires an IP address for the virtual network interface that will be used by all other containers to communicate with each other and the outside world.

CNI compare

A little outdated, e.g. Calico has wire guard now to do encryption. So, read it with care.

See updated CNI compare at https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-august-2020-6e1b757b9e49

There are a handful of CNI plugins available to choose—just to name a few: Calico, Cilium, WeaveNet, Flannel, and so on. The CNI plugins' implementation varies, but in general, what CNI plugins do is similar. They carry out the following tasks:

- Manage network interfaces for containers
- Allocate IP addresses for pods. This is usually done via calling other **IP Address Management** (**IPAM**) plugins such as `host-local`
- Implement network policies (optional)

The network policy implementation is not required in the CNI specification, but when DevOps choose which CNI plugins to use, it is important to take security into consideration. Alexis Ducastel's article (`https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-36475925a560`) did a good comparison of the mainstream CNI plugins with the latest update in April 2019. The security comparison is notable, as can be seen in the following screenshot:

CNI	Encryption	Network Policies
Calico	No	Ingress + Egress
Canal	No	Ingress + Egress
Cilium	Yes	Ingress + Egress
Flannel	No	No
Kube-router	No	Ingress only
WeaveNet	Yes	Ingress + Egress

Figure 2.9 – CNI plugins comparison

You may notice that the majority of the CNI plugins on the list don't support encryption. Flannel does not support Kubernetes network policies, while `kube-router` supports ingress network policies only.

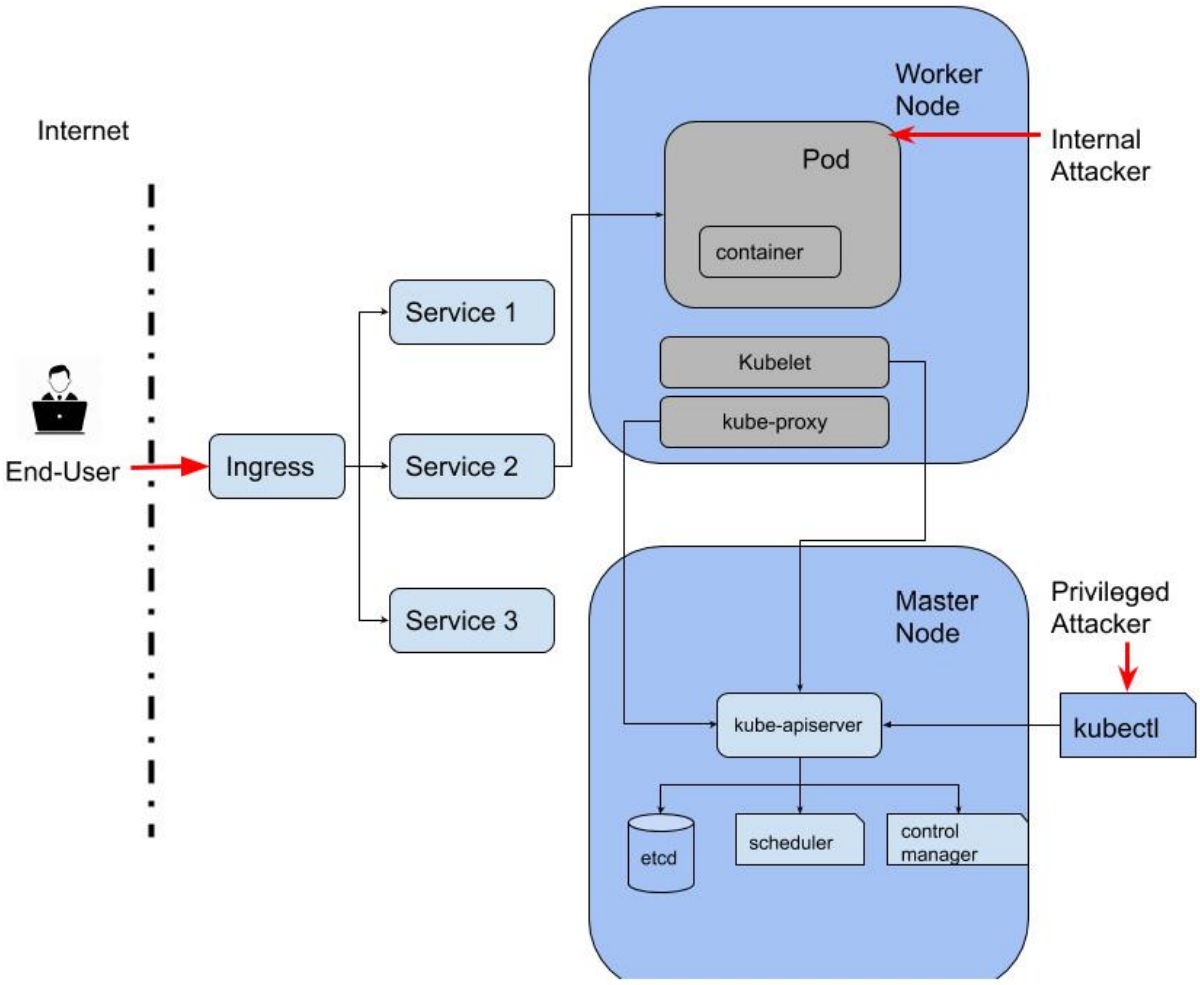
Threat actors in Kubernetes environments

A threat actor is an entity or code executing in the system that the asset should be protected from. From a defense standpoint, you first need to understand who your potential enemies are, or your defence strategy will be too vague. Threat actors in Kubernetes environments can be broadly classified into three categories:

1. **End user**: An entity that can connect to the application. The entry point for this actor is usually the load balancer or ingress. Sometimes, pods, containers, or NodePorts may be directly exposed to the internet, adding more entry points for the end user.
2. **Internal attacker**: An entity that has limited access inside the Kubernetes cluster. Malicious containers or pods spawned within the cluster are examples of internal attackers.
3. **Privileged attacker**: An entity that has administrator access inside the Kubernetes cluster. Infrastructure administrators, compromised `kube-apiserver` instances, and malicious nodes are all examples of privileged attackers.

Examples of threat actors include script kiddies, hacktivists, and nation-state actors. All these actors fall into the three aforementioned categories, depending on where in the system the actor exists.

The following diagram highlights the different actors in the Kubernetes ecosystem:


Internet
End-User
Ingress
Service 1
Service 2
Service 3
Worker Node
Pod
container
Kubelet
kube-proxy
Internal Attacker
Master Node
kube-apiserver
etcd
scheduler
control manager
Privileged Attacker
kubectl

Good to secure api-servers, etcd, etc…

Threats in Kubernetes clusters

With our new understanding of Kubernetes components and threat actors, we're moving on to the journey of threat modeling a Kubernetes cluster. In the following table, we cover the major Kubernetes components, nodes, and pods. Nodes and pods are the fundamental Kubernetes objects that run workloads. Note that all these components are assets and should be protected from threats. Any of these components getting compromised could lead to the next step of an attack, such as privilege escalation. Also, note that `kube-apiserver` and `etcd` are the brain and heart of a Kubernetes cluster. If either of them were to get compromised, that would be game over.

The following table highlights the threats in the default Kubernetes configuration. This table also highlights how developers and cluster administrators can protect their assets from these threats:

Assets	Threat	Security control	Mitigation strategy
`kube-apiserver`	No default audit policy. This prevents forensic analysis after an attack.	Audit policy	Enable audit policy. The metadata level is recommended across the ecosystem.
	`--anonymous-auth` is set to `true` by default, which basically allows any connection to `kube-apisever`.	Authentication/ authorization	Ensure `--anonymous-auth` is not set to `false`.
	Weak authentication because of the use of self-signed certificates.	Enable client CA using `--client-ca-file`	Monitor ingress connections to `kube-apiserver` to check for anomalies.
`etcd`	Data is not encrypted at rest by default.	Encrypt data using `--encryption-provider-config`	Pass the configuration parameter `--encryption-provider-config` to `kube-apiserver` by default.
	Authentication is not enabled by default.	Authentication/ authorization	Ensure authentication is enabled and use TLS to avoid access to `etcd` by malicious components or objects in the cluster.

Many good reminders in the next few pages!

Traditional 3-tier web application

Threat modeling application in Kubernetes

Now that we have looked at threats in a Kubernetes cluster, let's move on to discuss how threat modeling will differ for an application deployed on Kubernetes. Deployment in Kubernetes adds additional complexities to the threat model. Kubernetes adds additional considerations, assets, threat actors, and new security controls that need to be considered before investigating the threats to the deployed application.

Let's look at a simple example of a three-tier web application:

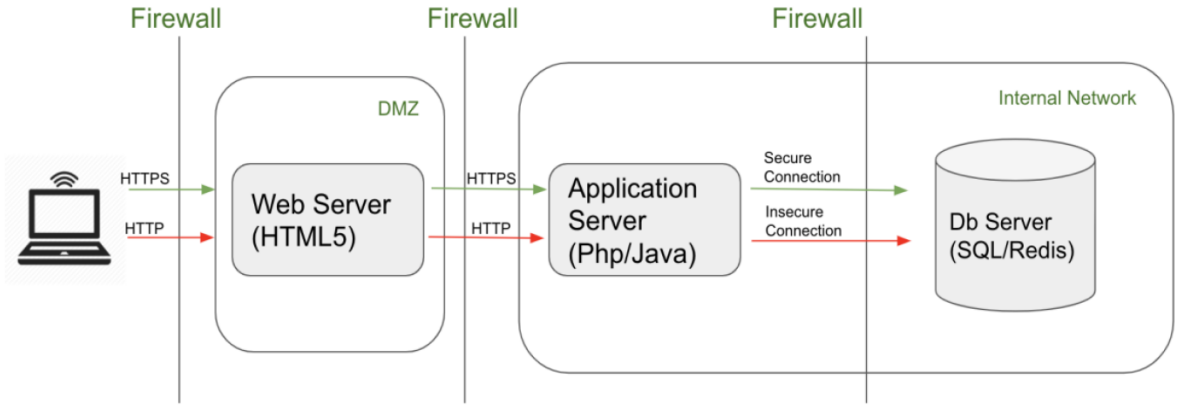


Figure 3.4 – Threat model of a traditional web application

Very different now with k8s:

The same application looks a little different in the Kubernetes environment:

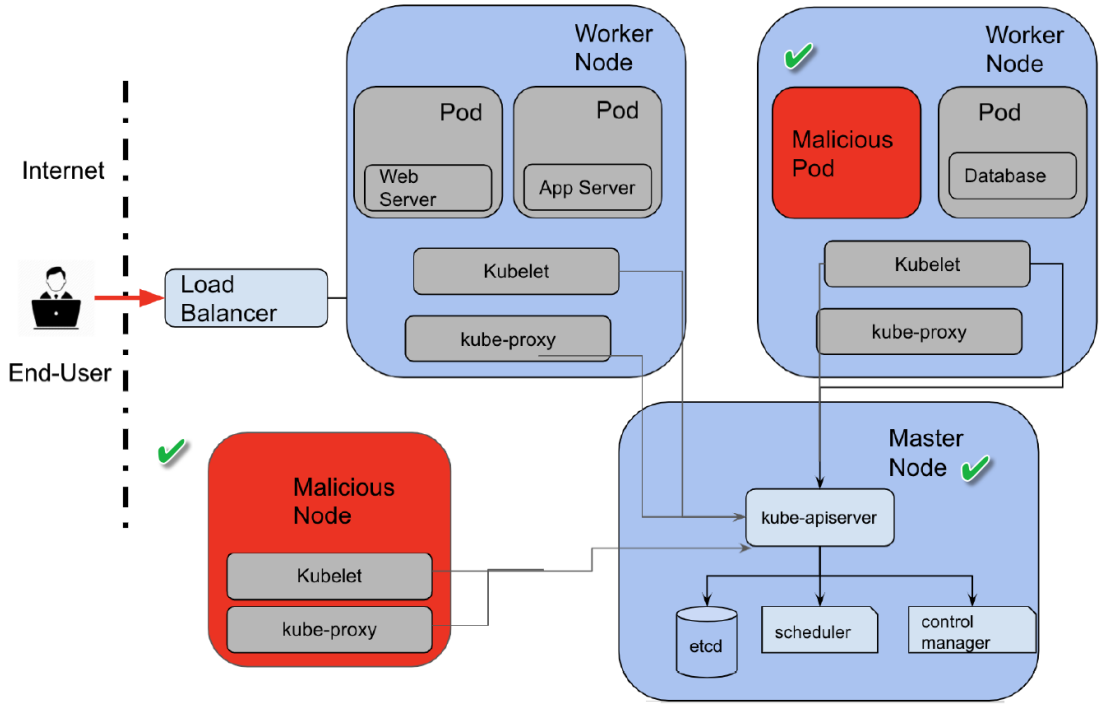


Figure 3.5 – Threat model of the three-tier web application in Kubernetes

As shown in the previous diagram, the web server, application server, and databases are all running inside pods. Let's do a high-level comparison of threat modeling between traditional web architecture and cloud-native architecture:

	Traditional web architecture	**Web application on Kubernetes**
Assets	Web server	Web server
	Application server	Application server
	Database server	Database server
	Hosts	Nodes (worker and master)
		Pods
		Persistent volumes
		Kubernetes components (`api-server`, `etcd`, `proxy`, `kubelet`, `scheduler`, `controller-manager`)

	Traditional web architecture	Web application on Kubernetes
Threat Actors	Internet/end users	Internet/end users
	Internal attackers	Internal attackers
	Admins	Admins
		Malicious/compromised nodes
		Malicious/compromised pods
		Compromised Kubernetes components
		Applications running inside the cluster
Security Controls	Firewall	Network policies
	DMZ	TLS, mTLS
	Internal network	Pod security policy
	Web application firewall	Web application firewall
	TLS connections	Pod isolation
	File encryption	File encryption
	Database authorization	Database authorization
	Database encryption	Database encryption
		Admission controllers
		Kubernetes authorization

To summarize the preceding comparison, you will find that more assets need to be protected in a cloud-native architecture, and you will face more threat actors in this space. Kubernetes provides more security controls, but it also adds more complexity. More security controls doesn't necessarily mean more security. Remember: complexity is the enemy of security.

Least privilege for accessing system resources

Recall that a microservice running inside a container or pod is nothing but a process on a worker node isolated in its own namespace. A pod or container may access different types of resources on the worker node based on the configuration. This is controlled by the security context, which can be configured both at the pod level and the container level. Configuring the pod/container security context should be on the developers' task list (with the help of security design and review), while pod security policies—the other way to limit pod/container access to system resources at the cluster level—should be on DevOps's to-do list. Let's look into the concepts of security context, PodSecurityPolicy, and resource limit control.

Security context

A security context offers a way to define privileges and access control settings for pods and containers with regard to accessing system resources. In Kubernetes, the security context at the pod level is different from that at the container level, though there are some overlapping attributes that can be configured at both levels. In general, the security context provides the following features that allow you to apply the principle of least privilege for containers and pods:

- **Discretionary Access Control (DAC)**: This is to configure which **user ID** (**UID**) or **group ID** (**GID**) to bind to the process in the container, whether the container's root filesystem is read-only, and so on. It is highly recommended not to run your microservice as a root user (*UID = 0*) in containers. The security implication is that if there is an exploit and a container escapes to the host, the attacker gains the root user privileges on the host immediately.

Least privilege for accessing system resources

Recall that a microservice running inside a container or pod is nothing but a process on a worker node isolated in its own namespace. A pod or container may access different types of resources on the worker node based on the configuration. This is controlled by the security context, which can be configured both at the pod level and the container level. Configuring the pod/container security context should be on the developers' task list (with the help of security design and review), while pod security policies—the other way to limit pod/container access to system resources at the cluster level—should be on DevOps's to-do list. Let's look into the concepts of security context, PodSecurityPolicy, and resource limit control.

Security context

A security context offers a way to define privileges and access control settings for pods and containers with regard to accessing system resources. In Kubernetes, the security context at the pod level is different from that at the container level, though there are some overlapping attributes that can be configured at both levels. In general, the security context provides the following features that allow you to apply the principle of least privilege for containers and pods:

- **Discretionary Access Control (DAC)**: This is to configure which **user ID** (**UID**) or **group ID** (**GID**) to bind to the process in the container, whether the container's root filesystem is read-only, and so on. It is highly recommended not to run your microservice as a root user (*UID = 0*) in containers. The security implication is that if there is an exploit and a container escapes to the host, the attacker gains the root user privileges on the host immediately.

Chapter 5: Configuring Kubernetes Security Boundaries

We will cover the following topics in this chapter:

1. Introduction to security boundaries
2. Security boundaries versus trust boundaries
3. Kubernetes security domains
4. Kubernetes entities as security boundaries
5. Security boundaries in the system layer
6. Security boundaries in the network layer

E.g. network policy boundary:

Note that there is no - in front of the `podSelector` attribute. This means the ingress source can only be pods with the label `from: good` in the namespace with the label `from: good`. This network policy protects Pods with the label `app: web` in the default namespace:

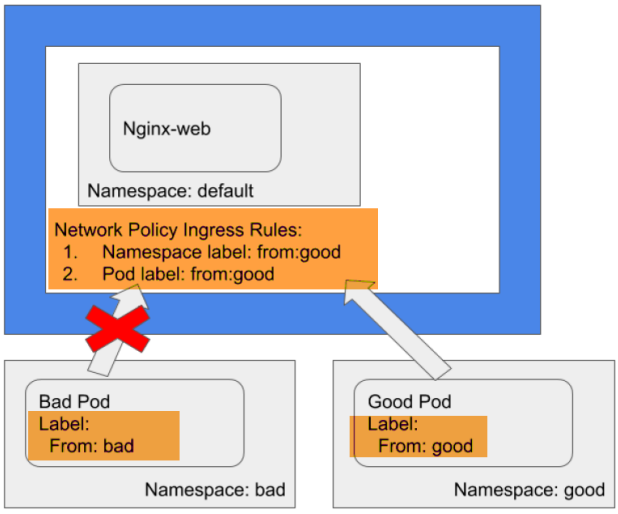


Figure 5.1 – Network policy restricting incoming traffic by Pod and namespace labels

In the preceding diagram, the `good` namespace has the label `from: good` while the `bad` namespace has the label `from: bad`. It illustrates that only Pods with the label `from: good` in the namespace with the label `from: good` can access the `nginx-web` service in the default namespace. Other Pods, no matter whether they're from the `good` namespace but without the label `from: good` or from other namespaces, cannot access the `nginx-web` service in the default namespace.

Section 2: Securing Kubernetes Deployments and Clusters

In this section, you will learn through hands-on exercises how to secure Kubernetes deployments/clusters in two ways: you will learn how to secure a DevOps pipeline in build, deployment, and runtime stages, and you will learn about defense in depth, looking at compliance, configuration, identity, authorization, resource management, logging and monitoring, detection, and incident response.

The following chapters are included in this section:

1. *Chapter 6, Securing Cluster Components*
2. *Chapter 7, Authentication, Authorization, and Admission Control*
3. *Chapter 8, Securing Kubernetes Pods*
4. *Chapter 9, Image Scanning in DevOps Pipelines*
5. *Chapter 10, Real-Time Monitoring and Resource Management of a Kubernetes Cluster*
6. *Chapter 11, Defense in Depth*

Chapter 6: Securing Cluster Components

For each master and node component, we briefly discuss the function of components with a security-relevant configuration in a Kubernetes cluster and look in detail at each configuration. We look at the possible settings for these configurations and highlight the recommended practices. Finally, we introduce `kube-bench` and walk through how this can be used to evaluate the security posture of your cluster.

In this chapter, we will cover the following topics:

1. Securing kube-apiserver
2. Securing kubelet
3. Securing etcd
4. Securing kube-scheduler
5. Securing kube-controller-manager
6. Securing CoreDNS
7. Benchmarking a cluster's security configuration

E.g. a very long list to secure apiserver:

To secure the API server, you should do the following:

1. **Disable anonymous authentication**: Use the `anonymous-auth=false` flag to set anonymous authentication to `false`. This ensures that requests rejected by all authentication modules are not treated as anonymous and are discarded.
2. **Disable basic authentication**: Basic authentication is supported for convenience in `kube-apiserver` and should not be used. Basic authentication passwords persist indefinitely. `kube-apiserver` uses the `--basic-auth-file` argument to enable basic authentication. Ensure that this argument is not used.
3. **Disable token authentication**: `--token-auth-file` enables token-based authentication for your cluster. Token-based authentication is not recommended. Static tokens persist forever and need a restart of the API server to update. Client certificates should be used for authentication.
4. **Ensure connections with kubelet use HTTPS**: By default, `--kubelet-https` is set to `true`. Ensure that this argument is not set to `false` for `kube-apiserver`.
5. **Disable profiling**: Enabling profiling using `--profiling` exposes unnecessary system and program details. Unless you are experiencing performance issues, disable profiling by setting `--profiling=false`.
6. **Disable AlwaysAdmit**: `--enable-admission-plugins` can be used to enable admission control plugins that are not enabled by default. `AlwaysAdmit` accepts the request. Ensure that the plugin is not in the `--enabled-admission-plugins` list.
7. **Use AlwaysPullImages**: The `AlwaysPullImages` admission control ensures that images on the nodes cannot be used without correct credentials. This prevents malicious pods from spinning up containers for images that already exist on the node.
8. **Use SecurityContextDeny**: This admission controller should be used if `PodSecurityPolicy` is not enabled. `SecurityContextDeny` ensures that pods cannot modify `SecurityContext` to escalate privileges.
9. **Enable auditing**: Auditing is enabled by default in `kube-apiserver`. Ensure that `--audit-log-path` is set to a file in a secure location. Additionally, ensure that the `maxage`, `maxsize`, and `maxbackup` parameters for auditing are set to meet compliance expectations.
10. **Disable AlwaysAllow authorization**: Authorization mode ensures that requests from users with correct privileges are parsed by the API server. Do not use `AlwaysAllow` with `--authorization-mode`.
11. **Enable RBAC authorization**: RBAC is the recommended authorization mode for the API server.

… and more on page 96 and 97!

- **Do not disable the ServiceAccount admission controller**: This admission control automates service accounts. Enabling `ServiceAccount` ensures that custom `ServiceAccount` with restricted permissions can be used with different Kubernetes objects.
- **Do not use self-signed certificates for requests**: If HTTPS is enabled for `kube-apiserver`, a `--tls-cert-file` and a `--tls-private-key-file` should be provided to ensure that self-signed certificates are not used.
- **Secure connections to etcd**: Setting `--etcd-cafile` allows `kube-apiserver` to verify itself to `etcd` over **Secure Sockets Layer** (**SSL**) using a certificate file.
- **Use secure TLS connections**: Set `--tls-cipher-suites` to strong ciphers only. `--tls-min-version` is used to set the minimum-supported TLS version. TLS 1.2 is the recommended minimum version.
- **Enable advanced auditing**: Advanced auditing can be disabled by setting the `--feature-gates` to `AdvancedAuditing=false`. Ensure that this field is present and is set to `true`. Advanced auditing helps in an investigation if a breach happens.

Chapter 7 : Authentication, Authorization, and Admission Control

AuthN and AuthZ, RBAC, Admission controllers, Webhooks, OPA, etc.

OPA is deployed as a service alongside your other services. To make authorization decisions, the microservice makes a call to OPA to decide whether the request should be allowed or denied. Authorization decisions are offloaded to OPA, but this enforcement needs to be implemented by the service itself. In Kubernetes environments, it is often used as a validating webhook:

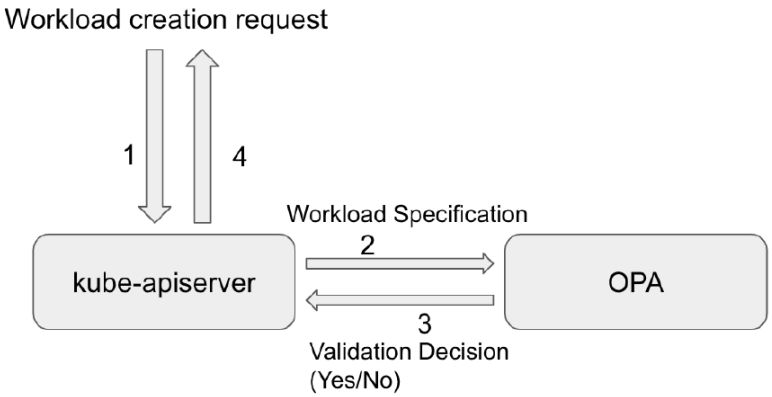


Figure 7.2 – Open Policy Agent

To make a policy decision, OPA needs the following:

- **Cluster information**: The state of the cluster. The objects and resources available in the cluster are important for OPA to make a decision about whether a request should be allowed or not.
- **Input query**: The parameters of the request being parsed by the policy agent are analyzed by the agent to allow or deny the request.
- **Policies**: The policy defines the logic that parses cluster information and input query to return the decision. Policies for OPA are defined in a custom language called Rego.

Further reading

You can refer to the following links for more information:

- Admission controllers: `https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do`
- OPA: `https://www.openpolicyagent.org/docs/latest/`
- Kubernetes RBAC: `https://rbac.dev/`
- audit2RBAC: `https://github.com/liggitt/audit2rbac`
- KubiScan: `https://github.com/cyberark/KubiScan`

https://github.com/cyberark/KubiScan

"A tool for scanning Kubernetes cluster for risky permissions in Kubernetes's Role-based access control (RBAC) authorization model. The tool was published as part of the "Securing Kubernetes Clusters by Eliminating Risky Permissions" research https://www.cyberark.com/threat-research-blog/securing-kubernetes-clusters-by-eliminating-risky-permissions/."

Chapter 8: Securing Kubernetes Pods

CIS Docker benchmarks, configuring the security attributes of pods, Security Context, Linux capabilities, PSP …etc

Configuring the security attributes of pods

As we mentioned in the previous chapter, application developers should be aware of what privileges a microservice must have in order to perform tasks. Ideally, application developers and security engineers work together to harden the microservice at the pod and container level by configuring the security context provided by Kubernetes.

We classify the major security attributes into four categories:

1. Setting host namespaces for pods
2. Security context at the container level
3. Security context at the pod level
4. AppArmor profile

By employing such a means of classification, you will find them easy to manage.

Setting host-level namespaces for pods

The following attributes in the pod specification are used to configure the use of host namespaces:

5. **hostPID**: By default, this is `false`. Setting it to `true` allows the pod to have visibility on all the processes in the worker node.
6. **hostNetwork**: By default, this is `false`. Setting it to `true` allows the pod to have visibility on all the network stacks in the worker node.
7. **hostIPC**: By default, this is `false`. Setting it to `true` allows the pod to have visibility on all the IPC resources in the worker node.

PSP advisor:

Kubernetes PodSecurityPolicy Advisor

Kubernetes PodSecurityPolicy Advisor (also known as `kube-psp-advisor`) is an open source tool from Sysdig. It scans the security attributes of running workloads in the cluster and then, on this basis, recommends pod security policies for your cluster or workloads.

First, let's install `kube-psp-advisor` as a `kubectl` plugin. If you haven't installed `krew`, a `kubectl` plugin management tool, please follow the instructions (`https://github.com/kubernetes-sigs/krew#installation`) in order to install it. Then, install `kube-psp-advisor` w

https://github.com/kubernetes-sigs/krew#installation

```
$ kubectl krew install advise-psp
```

Then, you should be able to run the following command to verify the installation:

```
$ kubectl advise-psp
A way to generate K8s PodSecurityPolicy objects from a live
K8s environment or individual K8s objects containing pod
specifications

Usage:
  kube-psp-advisor [command]

Available Commands:
  convert     Generate a PodSecurityPolicy from a single K8s
Yaml file
  help        Help about any command
```

Managing vulnerabilities

When you have a vulnerability management strategy, you won't panic. In general, every vulnerability management strategy will start with understanding the exploitability and impact of the vulnerability based on the CVE detail. NVD provides a vulnerability scoring system also known as **Common Vulnerability Scoring System** (**CVSS**) to help you better understand how severe the vulnerability is.

The following information needs to be provided to calculate the vulnerability score based on your own understanding of the vulnerability:

1. **Attack vector**: Whether the exploit is a network attack, local attack, or physical attack
2. **Attack complexity**: How hard it is to exploit the vulnerability
3. **Privileges required**: Whether the exploit requires any privileges, such as root or non-root

4. **User interaction**: Whether the exploit requires any user interaction
5. **Scopes**: Whether the exploit will lead to cross security domain
6. **Confidentiality impact**: How much the exploit impacts the confidentiality of the software
7. **Integrity impact**: How much the exploit impacts the integrity of the software
8. **Availability impact**: How much the exploit impacts the availability of the software

The CVSS calculator is available at `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`:

This is why an image scanning tool is critical to your CI/CD pipeline. It's not realistic to cover vulnerability management in one section, but I think a basic understanding of vulnerability management will help you make the most use of any image scanning tool. There are a few popular open source image scanning tools available, such as Anchore, Clair, Trivvy, and so on. Let's look at one such image scanning tool with examples.

Scanning images with Anchore Engine

Anchore Engine is an open source image scanning tool. It not only analyzes Docker images but also allows users to define an acceptance image scanning policy. In this section, we will first give a high-level introduction to Anchore Engine, then we will show how to deploy Anchore Engine and the basic image scanning use case of Anchore Engine by using Anchore's own CLI tool, `anchore-cli`.

Introduction to Anchore Engine

When an image is submitted to Anchore Engine for analysis, Anchore Engine will first retrieve the image metadata from image registry, then download the image and queue the image for analysis. The following are the items that Anchore Engine will analyze:

1. Image metadata
2. Image layers
3. Operating system packages such as `deb`, `rpm`, `apkg`, and so on
4. File data
5. Application dependency packages:
   - Ruby gems
   - Node.js NPMs
   - Java archives
   - Python packages
6. File content

To deploy Anchore Engine in a Kubernetes cluster with **Helm**—CNCF project which is a package manage tool for the Kubernetes cluster, run the following command:

```
$ helm install anchore-demo stable/anchore-engine
```

Anchore Engine is composed of a few microservices. When deployed in a Kubernetes cluster, you will find the following workloads are running:

```
$ kubectl get deploy
NAME                                        READY   UP-TO-DATE
AVAILABLE   AGE
anchore-demo-anchore-engine-analyzer        1/1     1
1           3m37s
anchore-demo-anchore-engine-api             1/1     1
1           3m37s
anchore-demo-anchore-engine-catalog         1/1     1
1           3m37s
```

Introduction to Anchore Engine

When an image is submitted to Anchore Engine for analysis, Anchore Engine will first retrieve the image metadata from image registry, then download the image and queue the image for analysis. The following are the items that Anchore Engine will analyze:

1. Image metadata
2. Image layers
3. Operating system packages such as `deb`, `rpm`, `apkg`, and so on
4. File data
5. Application dependency packages:
   - Ruby gems
   - Node.js NPMs
   - Java archives
   - Python packages
6. File content

To deploy Anchore Engine in a Kubernetes cluster with **Helm**—CNCF project which is a package manage tool for the Kubernetes cluster, run the following command:

```
$ helm install anchore-demo stable/anchore-engine
```

Anchore Engine is composed of a few microservices. When deployed in a Kubernetes cluster, you will find the following workloads are running:

```
$ kubectl get deploy
NAME                                        READY   UP-TO-DATE
AVAILABLE   AGE
anchore-demo-anchore-engine-analyzer        1/1     1
1           3m37s
anchore-demo-anchore-engine-api             1/1     1
1           3m37s
anchore-demo-anchore-engine-catalog         1/1     1
1           3m37s
```

Sample image scanning and outputs:

So the image `docker.io/kaizheh/nginx-docker:latest` failed the default policy evaluation. This means that there must be some vulnerabilities at a high or critical level. Use the following command to list all the vulnerabilities in the image:

```
root@anchore-cli:/# anchore-cli image vuln docker.io/kaizheh/
nginx-docker:latest all
```

And the output should be like the following:

```
Vulnerability ID        Package
Severity          Fix                                  CVE Refs
Vulnerability URL
CVE-2019-9636           Python-2.7.16
Critical          None                                 CVE-2019-
9636           https://nvd.nist.gov/vuln/detail/CVE-2019-9636
CVE-2020-7598           minimist-0.0.8
Critical          None                                 CVE-2020-
7598           https://nvd.nist.gov/vuln/detail/CVE-2020-7598
CVE-2020-7598           minimist-1.2.0
Critical          None                                 CVE-2020-
7598           https://nvd.nist.gov/vuln/detail/CVE-2020-7598
CVE-2020-8116           dot-prop-4.2.0
Critical          None                                 CVE-2020-
8116           https://nvd.nist.gov/vuln/detail/CVE-2020-8116
CVE-2013-1753           Python-2.7.16
High              None                                 CVE-2013-
1753           https://nvd.nist.gov/vuln/detail/CVE-2013-1753
CVE-2015-5652           Python-2.7.16
High              None                                 CVE-2015-
5652           https://nvd.nist.gov/vuln/detail/CVE-2015-5652
CVE-2019-13404          Python-2.7.16
High              None                                 CVE-2019-
13404          https://nvd.nist.gov/vuln/detail/CVE-2019-13404
CVE-2016-8660           linux-compiler-gcc-8-x86-
4.19.67-2+deb10u1             Low               None
CVE-2016-8660           https://security-tracker.debian.org/
tracker/CVE-2016-8660
CVE-2016-8660           linux-headers-4.19.0-6-amd64-
4.19.67-2+deb10u1           Low               None
CVE-2016-8660           https://security-tracker.debian.org/
tracker/CVE-2016-8660
```

Chapter 10: Real-Time Monitoring and Resource Management of a Kubernetes Cluster

Dashboard, metrics server, Prometheus, Grafana, etc…

Chapter 11: Defense in Depth

In this chapter, we will talk about Kubernetes auditing, then we will introduce the concept of high availability and talk about how we can apply high availability in the Kubernetes cluster. Next, we will introduce Vault, a handy secrets management product for the Kubernetes cluster. Then, we will talk about how to use Falco to detect anomalous activities in the Kubernetes cluster. Last but not least, we will introduce Sysdig Inspect and **Checkpoint and Resource In Userspace** (also known as **CRIU**) for forensics.

The following topics will be covered in this chapter:

1. Introducing Kubernetes auditing
2. Enabling high availability in a Kubernetes cluster
3. Managing secrets with Vault
4. Detecting anomalies with Falco
5. Conducting forensics with Sysdig Inspect and CRIU

Good audit policy sample:

Kubernetes audit policy

As it is not realistic to record everything happening inside the Kubernetes cluster, an audit policy allows users to define rules about what kind of event should be recorded and how much detail of the event should be recorded. When an event is processed by `kube-apiserver`, it compares the list of rules in the audit policy in order. The first matching rules also dictate the audit level of the event. Let's take a look at what an audit policy looks like. Here is an example:

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Skip generating audit events for all requests in
RequestReceived stage. This can be either set at the policy
level or rule level.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    verbs: ["create", "update"]
    namespace: ["ns1", "ns2", "ns3"]
    resources:
    - group: ""
# Only check access to resource "pods", not the sub-resource of
pods which is consistent with the RBAC policy.
      resources: ["pods"]
# Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]
# Don't log authenticated requests to certain non-resource URL
paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs: ["/api*", "/version"]
```

Configuring the audit backend

In order to enable Kubernetes auditing, you need to pass the `--audit-policy-file` flag with your audit policy file when starting `kube-apiserver`. There are two types of audit backends that can be configured to use process audit events: a log backend and a webhook backend. Let's have a look at them.

Log backend

The log backend writes audit events to a file on the master node. The following flags are used to configure the log backend within `kube-apiserver`:

1. `--log-audit-path`: Specify the log path on the master node. This is the flag to turn ON or OFF the log backend.
2. `--audit-log-maxage`: Specify the maximum number of days to keep the audit records.
3. `--audit-log-maxbackup`: Specify the maximum number of audit files to keep on the master node.
4. `--audit-log-maxsize`: Specify the maximum size in megabytes of an audit log file before it gets rotated.

Let's take a look at the webhook backend.

Webhook backend

The webhook backend writes audit events to the remote webhook registered to `kube-apiserver`. To enable the webhook backend, you need to set the `--audit-webhook-config-file` flag with the webhook configuration file. This flag is also specified when starting `kube-apiserver`. The following is an example of a webhook configuration to register a webhook backend for the Falco service, which will be introduced later in more detail:

```
apiVersion: v1
kind: Config
clusters:
- name: falco
  cluster:
    server: http://$FALCO_SERVICE_CLUSTERIP:8765/k8s_audit
contexts:
- context:
    cluster: falco
    user: ""
  name: default-context
current-context: default-context
preferences: {}
users: []
```

The URL specified in the `server` field (`http://$FALCO_SERVICE_CLUSTERIP:8765/k8s_audit`) is the remote endpoint that the audit events will be sent to. Since version 1.13 of Kubernetes, the webhook backend can be configured dynamically via the `AuditSink` object, which is still in the alpha stage.

In this section, we talked about Kubernetes auditing by introducing the audit policy and audit backends. In the next section, we will talk about high availability in the Kubernetes cluster.

Then Vault!

And let's look at the specification of the pod (not the patched deployment)—you will find the following (marked in bold) were added, compared to the specification of the patched deployment:

```
    containers:
2   - image: jweissig/app:0.0.1
      ...
      volumeMounts:
      - mountPath: /vault/secrets
        name: vault-secrets
    - args:
      - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json &&
vault agent -config=/tmp/config.json
      command:
      - /bin/sh
      - -ec
      image: vault:1.3.2
      name: vault-agent
      volumeMounts:
      - mountPath: /vault/secrets
1       name: vault-secrets
  initContainers:
    - args:
      - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json &&
vault agent -config=/tmp/config.json
      command:
      - /bin/sh
      - -ec
      image: vault:1.3.2
      name: vault-agent-init
      volumeMounts:
      - mountPath: /vault/secrets
        name: vault-secrets
    volumes:
     - emptyDir:
        medium: Memory
      name: vault-secrets
```

A few things worth mentioning from the preceding changes listed: one `init` container named `vault-agent-init` and one sidecar container named `vault-agent` have been injected, as well as an `emptyDir` type volume named `vault-secrets`. That's why you saw two containers are running in the demo application pod after the patch. Also, the `vault-secrets` volume is mounted in the `init` container, the `sidecar` container, and the `app` container with the `/vault/secrets/` directory. The secret is stored in the `vault-secrets` volume. The pod specification modification is done by the `vault-agent-injector` pod through a predefined mutating webhook configuration (installed via `helm`), as follows:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  ...
  name: vault-agent-injector-cfg
webhooks:
- admissionReviewVersions:
  - v1beta1
  clientConfig:
    caBundle: <CA_BUNDLE>
    service:
      name: vault-agent-injector-svc
      namespace: demo
      path: /mutate
  failurePolicy: Ignore
  name: vault.hashicorp.com
  namespaceSelector: {}
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    - UPDATE
    resources:
    - pods
```

Then Falco:

Event sources for anomaly detection

Falco relies on two event sources to do anomalous detection. One is system calls and the other is the Kubernetes audit events. For system call events, Falco uses a kernel module to tap into the stream of system calls on a machine, and then passes those system calls to a user space (`ebpf` is recently supported as well). Within the user space, Falco also enriches the raw system call events with more context such as the process name, container ID, container name, image name, and so on. For Kubernetes audit events, users need to enable the Kubernetes audit policy and register the Kubernetes audit webhook backend with the Falco service endpoint. Then, the Falco engine checks any of the system call events or Kubernetes audit events matching any Falco rules loaded in the engine.

It's also important to talk about the rationale for using system calls and Kubernetes audit events as event sources to do anomalous detection. System calls are a programmatic way for applications to interact with the operating system in order to access resources such as files, devices, the network, and so on. Considering containers are a bunch of processes with their own dedicated namespaces and that they share the same operating system on the node, a system call is the one unified event source that can be used to monitor activities from containers. It doesn't matter what programming language the application is written in; ultimately, all the functions will be translated into system calls to interact with the operating system. Take a look at the following diagram:

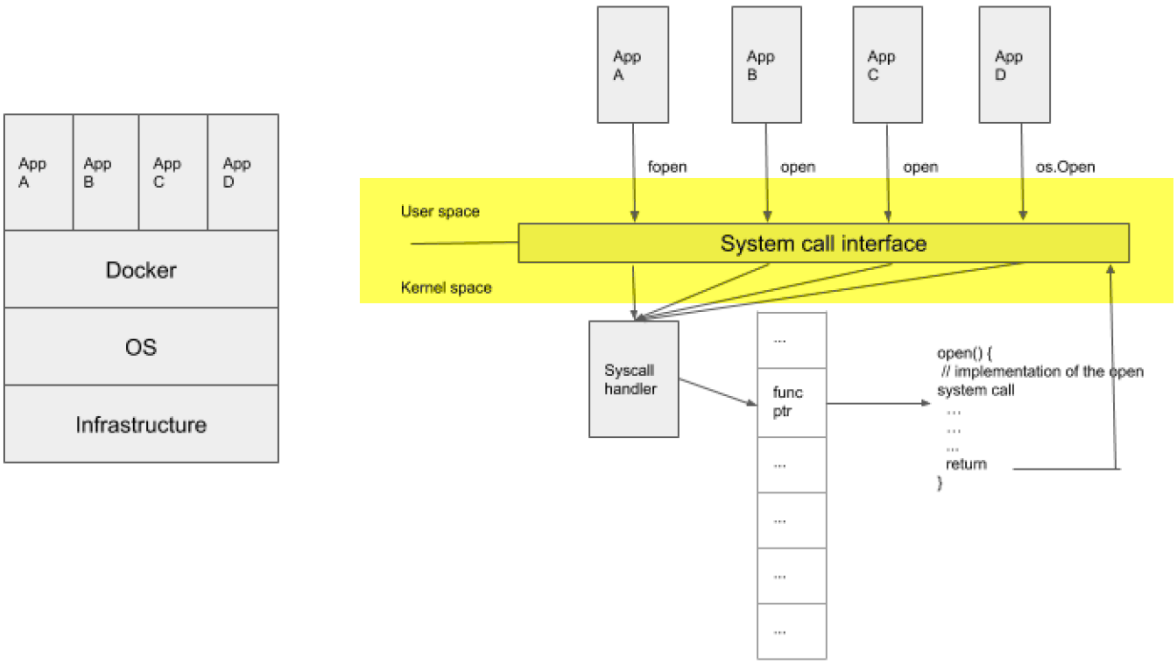


Figure 11.3 – Containers and system calls

On the other hand, with the help of Kubernetes audit events, Falco has full visibility into a Kubernetes object's life cycle. This is also important for anomalous detection. For example, it may be abnormal that there is a pod with a `busybox` image launched as a privileged pod in a production environment.

Overall, the two event sources—system calls and Kubernetes audit events—are sufficient to cover all the meaningful activities happening in the Kubernetes cluster. Now, with an understanding of Falco event sources, let's wrap up our overview on Falco with a high-level architecture diagram.

High-level architecture

Falco is mainly composed of a few components, as follows:

- **Falco rules**: Rules that are defined to detect whether an event is an anomaly.
- **Falco engine**: Evaluate an incoming event with Falco rules and throw an output if an event matches any of the rules.
- **Kernel module/Sysdig libraries**: Tag system call events and enrich them before sending to the Falco engine for evaluation.
- **Web server**: Listen on Kubernetes audit events and pass on to the Falco engine for evaluation.

The following diagram shows Falco's internal architecture:

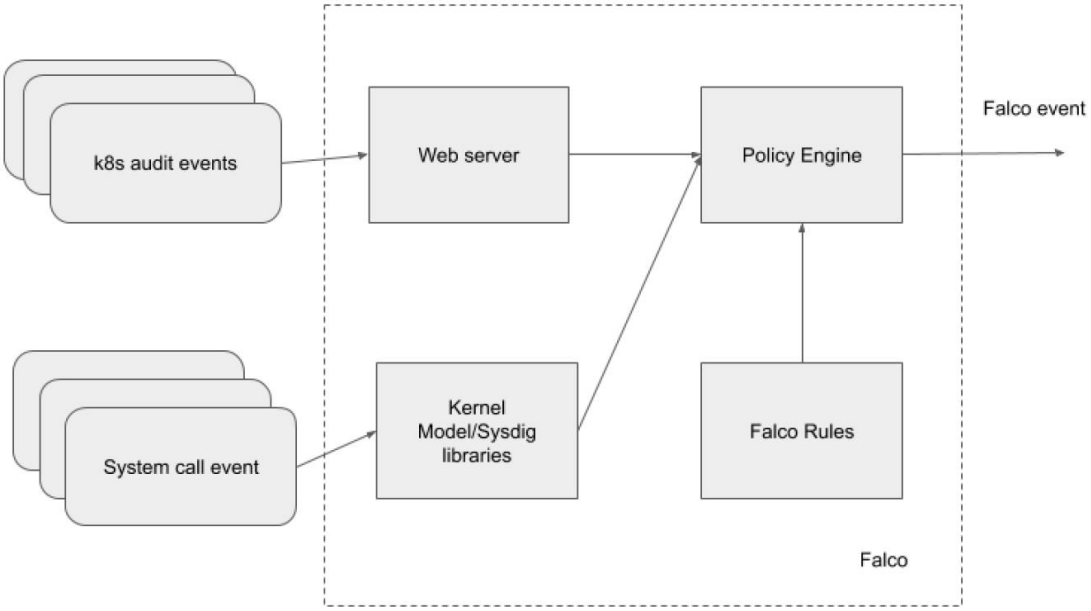


Figure 11.4 – Falco's internal architecture

Then Sysdig Inspect and CRIU:

Conducting forensics with Sysdig Inspect and CRIU

Forensics in cybersecurity means collecting, processing, and analyzing information in support of vulnerability mitigation and/or fraud, counterintelligence, or law enforcement investigations. The more data you can preserve and the faster the analysis you can conduct on the collected data, the quicker you will trace down an attack and respond to the incident better. In this section, we will show you how to use the CRIU and Sysdig open source tools to collect data, and then introduce Sysdig Inspect, an open source tool for analyzing data collected by Sysdig.

CRIU –

Using CRIU to collect data

CRIU is the abbreviation of **Checkpoint and Restore In Userspace**. It is a tool that can freeze a running container and capture the container's state on disk. Later on, the container's and application's data saved on the disk can be restored to the state it was at the time of the freeze. It is useful for container snapshots, migration, and remote debugging. From a security standpoint, it is especially useful to capture malicious activities in action in the container (so that you may kill the container right after the checkpoint) and then restore the state in a sandboxed environment for further analysis.

CRIU works as a Docker plugin and is still in experimental mode, and there is a known issue that CRIU is not working properly in the most recent few versions (`https://github.com/moby/moby/issues/37344`). For demo purposes, I have used an older Docker version (Docker CE 17.03) and will show how to use CRIU to checkpoint a running container and restore the state back as a new container.

To enable CRIU, you will need to enable the `experimental` mode in the Docker daemon, as follows:

https://github.com/moby/mot

```
echo "{\"experimental\":true}" >> /etc/docker/daemon.json
```

And then, after restarting the Docker daemon, you should be able to execute the `docker checkpoint` command successfully, like this:

```
# docker checkpoint
Usage:    docker checkpoint COMMAND
Manage checkpoints
Options:
      --help   Print usage
Commands:
  create      Create a checkpoint from a running container
  ls          List checkpoints for a container
  rm          Remove a checkpoint
```

Then, follow the instructions to install CRIU (`https://criu.org/Installation`). Next, let's see a simple example to show how powerful CRIU is. I have a simple `busybox` container running to increase the counter by `1` every second, as illustrated in the following code snippet:

CRIU is very useful for container forensics, especially when there are some suspicious activities happening in a container. You can checkpoint the container (assuming you have multiple replicas running within the cluster), let CRIU kill the suspicious container, and then restore the suspicious state of the container in a sandboxed environment for further analysis. Next, let's talk about another way to capture data for forensics.

Using Sysdig and Sysdig Inspect

Sysdig is an open source tool for Linux system exploration and troubleshooting with support for containers. Sysdig can also be used to create trace files for system activity through instrumenting into the Linux kernel and capturing system calls and other operating system events. The capture capability makes it an awesome forensics tool for a containerized environment. To support capture system calls in the Kubernetes cluster, Sysdig offers a `kubectl` plugin, `kubectl-capture`, which enables you to capture system calls of the target pods as simply as with some other `kubectl` commands. After the capture is finished, Sysdig Inspect, a powerful open source tool, can be used to do troubleshooting and security investigation.

Let's continue to use `insecure-nginx` as an example, since we've got a Falco alert, as illustrated in the following code snippet:

```
08:22:19.484698397: Warning Anomalous file read activity
in Nginx pod (user=<NA> process=nginx file=/etc/passwd
container_id=439e2e739868 image=kaizheh/insecure-nginx) k8s.
ns=insecure-nginx k8s.pod=insecure-nginx-7c99fdf44b-gffp4
container=439e2e739868 k8s.ns=insecure-nginx k8s.pod=insecure-
nginx-7c99fdf44b-gffp4 container=439e2e739868
```

By the time the alert was triggered, it is still possible the `nginx` pod was undergoing an attack. There are a few things you can do to respond. Starting a capture and then analyzing more context out of the Falco alert is one of them.

To trigger a capture, download `kubectl-capture` from `https://github.com/sysdiglabs/kubectl-capture` and place it with the other `kubectl` plugins, like this:

Summary

In this chapter, we went through a couple of the crypto-mining attacks that occurred over the last two years that brought a lot of attention to the need for securing containerized environments. Then, we showed you how to detect crypto-mining attacks with different open source tools. Last but not the least, we talked about how to defend your Kubernetes clusters against attacks in general by recapping what we discussed in previous chapters.

We hope you understand the core concepts of securing a Kubernetes cluster, which means securing the cluster provisioning, build, deployment, and runtime stages. You should also feel comfortable with starting to use Anchore, Prometheus, Grafana, and Falco.

As we know, Kubernetes is still evolving and it's not perfect. In the next chapter, we're going to talk about some known Kubernetes **Common Vulnerabilities and Exposures** (**CVEs**) and some mitigations that can protect your cluster against unknown variations. The purpose of the following chapter is to prepare you to be able to respond to handling any Kubernetes CVEs discovered in the future.

Chapter 13: Learning from Kubernetes CVEs

We will cover the following topics in this chapter:

1. The path traversal issue in kubectl cp—CVE-2019-11246
2. The DoS issue in JSON parsing—CVE-2019-1002100
3. The DoS issue in YAML parsing—CVE-2019-11253
4. The privilege-escalation issue in role parsing—CVE-2019-11247
5. Scanning known vulnerabilities using kube-hunter