Kubernetes Security Essentials

Securing Cloud-Native Applications

CONTENTS

YITAEK HWANG
SR. SOFTWARE ENGINEER II, AXONI

By every measure, Kubernetes is the de-facto standard for automating the deployment and management of cloud-native applications. Its adoption is transforming the ways in which organizations of every size, in every industry, develop and release software using technologies such as containers, microservices, and declarative APIs. In parallel, these new technologies and architectures give rise to broad security risks and challenges that organizations must protect. Kubernetes introduces a new threat environment — one that is as dynamic, fast-moving, and as active as containerized applications themselves.

Kubernetes — with its breadth of cluster components and associated tooling — also introduces complexity for an organization's end users. It requires teams to learn new skills and adopt new security workflows across development and operations. This complexity can expose organizations to a potentially expansive set of attack vectors throughout Kubernetes environments that stem from vulnerabilities, misconfigurations, or other operational issues.

As Kubernetes increasingly becomes a foundational infrastructure platform that underpins modern software delivery, securing Kubernetes itself and the cloud-native applications that run on top become critical. The broader Kubernetes community has undertaken several efforts to increase security awareness, such as conducting a security audit of the Kubernetes platform, publishing a Kubernetes attack matrix based on the MITRE ATT&CK framework, publishing a security whitepaper on best practices, and establishing various industry-standard security benchmarks.

In addition, the Open Source Security Foundation (OpenSSF) is now leading the Alpha-Omega Project to improve software supply chain security. On the application side, the OWASP Top 10 now includes "Insecure Design" and "Security Misconfiguration" to further shift security left. These efforts can help development, operations, and security leaders develop effective strategies for implementing new security measures to protect both applications and the Kubernetes infrastructure on which they run.

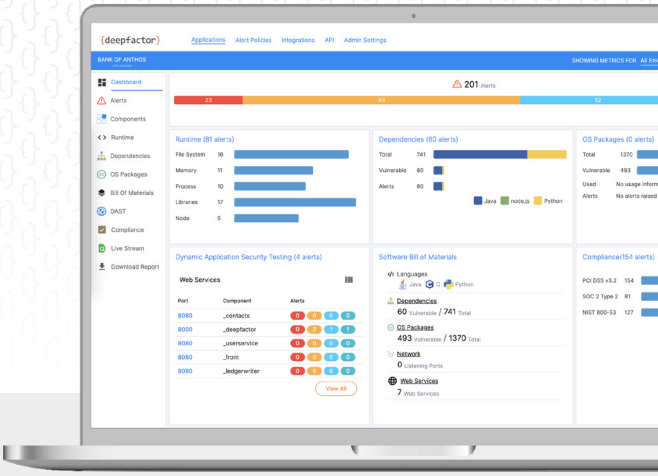
THE KUBERNETES ATTACK SURFACE

To understand how to secure cloud-native applications, it is imperative to know how to protect the underlying Kubernetes environment and its relevant attack surface. The attack surface within a Kubernetes cluster consists of three main areas:

- The software supply chain for building the immutable artifacts used to deploy and run containers
- Infrastructure components that must be provisioned and configured to run Kubernetes
- Deployed and running containers that make up individual Kubernetes applications

Developer-Led Security Starts Here

Deepfactor Developer Security helps your engineering teams automatically discover, prioritize, and remediate application security risks early in development and test.



// Why Deepfactor?

Detect Security Risks Before Shipping

Observe running applications in development and testing to uncover critical security risks in custom and third-party code.

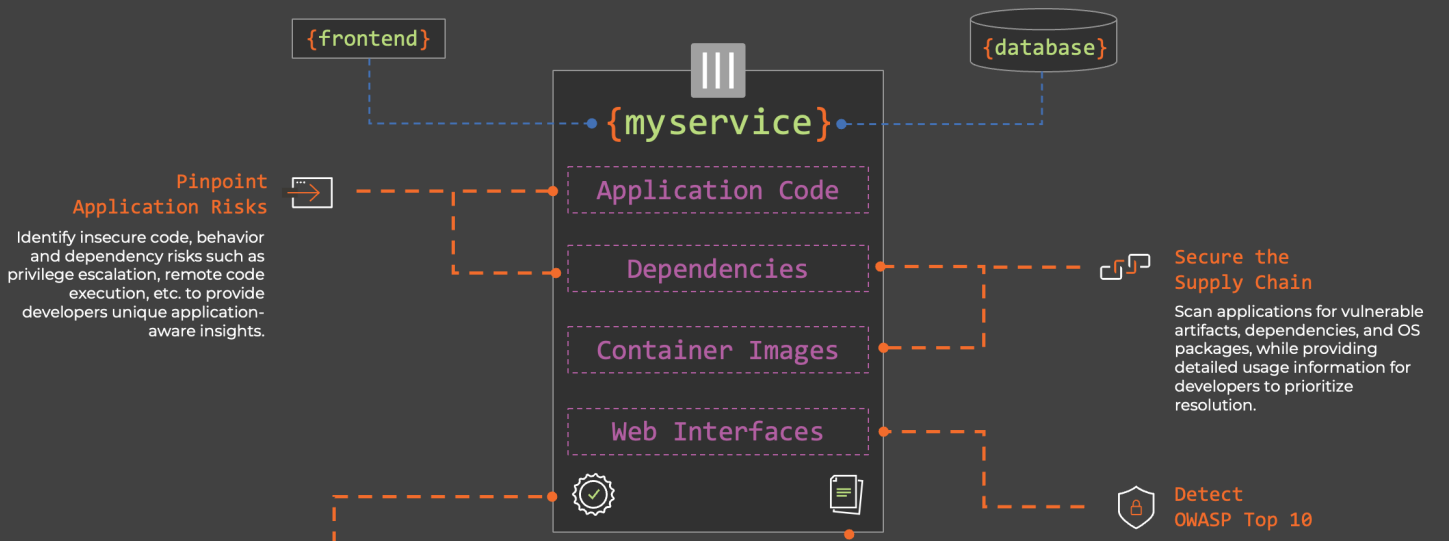
Provide Contextual and Actionable Alerts

Pinpoint insecure code, streamline and prioritize remediation, analyze drift between releases, and learn the impact on compliance.

Instrumentation Purpose-Built for Kubernetes

Observe every thread, process, container, and pod without requiring intrusive agents and privileged sidecars.

Deepfactor Developer Security Insights



Pinpoint Application Risks

Identify insecure code, behavior and dependency risks such as privilege escalation, remote code execution, etc. to provide developers unique application-aware insights.

Secure the Supply Chain

Scan applications for vulnerable artifacts, dependencies, and OS packages, while providing detailed usage information for developers to prioritize resolution.

Detect OWASP Top 10

Scan web interfaces and APIs, including hidden URIs, during QA testing to detect OWASP Top 10 critical security risks.

Understand Compliance Risk

Map application security risks to various compliance controls (PCI DSS, SOC 2, NIST) to help engineering teams understand potential impact.

Dynamic Bill of Materials

Analyze licensing, file usage, code interactions, and network behavior in addition to dependencies, OS packages, and components.

Integrated security insights spanning code, dependencies, container images, APIs, and compliance

Request a demo today!

{deepfactor}

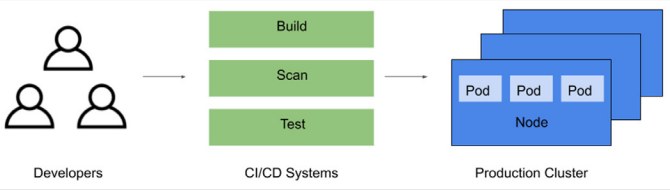
Nearly all Kubernetes threat vectors can be mapped into one of these three categories. This Refcard utilizes these categories as a framework to describe key security concepts that comprehensively span the Kubernetes infrastructure and applications.

SECURING THE SOFTWARE SUPPLY CHAIN

In Kubernetes environments, the software supply chain is a centralized place to make any software changes for propagation into production environments. It also serves as a bottleneck where users can incorporate security measures that impact the rest of the application lifecycle.

Container images constitute the standard application delivery format in Kubernetes environments. Building these images is the primary goal of a cloud-native software supply chain, so securing the supply chain should primarily focus on image security. The wide distribution and deployment of these container images requires a well-thought-out strategy for ensuring their security.

Figure 1: Software Supply Chain



BASE IMAGES

Security of container images often starts with the base operating system (OS) images. Consider using the official language-specific base image from a reputable provider (e.g., OpenJDK) over installing the package on a generic Linux image. Opt for alpine or distroless images from official image registries to keep the base image small. Finally, the base images that are used should also be updated frequently to address any newly disclosed vulnerabilities or other security concerns.

IMAGE COMPONENTS

Any container images that are built should be kept as minimal as possible. To avoid additional avenues of exploitation, utilize multistage builds in the containerized application for binaries, libraries, and configuration files only. In particular, the following should be avoided in production environments whenever possible:

Table 1

IMAGE CONTENTS	EXAMPLES
Package managers	`apt`, `yum`, `apk`
Network tools and clients	`wget`, `curl`, `netcat`, `ssh`
Unix shells	`sh`, `bash`
Secrets	TLS certificate keys, cloud provider credentials, SSH private keys, database passwords

Secrets should not be embedded in images since anyone with access to the image — either by downloading it from a registry or once it is built — would be able to view them in plain text, and because it provides unnecessary exposure prior to when the secret needs to be used. In Kubernetes clusters, you can use Kubernetes secrets or tools like Vault or external-secrets to pass this sensitive data to pods.

Another way to manage image components is to utilize software bill of materials (SBOM), as recommended in the President's Executive Order on Improving the Nation's Cybersecurity.

SBOMs are critical to securing the supply chain, particularly in augmenting dependency and OS package information, with additional attributes such as licensing, network data, and usage information. Common pitfalls such as using restrictive open-source licenses (e.g., GPL) and end-of-life (EOL) packages can be avoided by maintaining an SBOM.

IMAGE SCANNING

Once images are built, they must be scanned to avoid introducing vulnerabilities into your running Kubernetes clusters. Image scanners are available as standalone tools (e.g., trivy, anchore, clair), or in some cases, are integrated with image registries. It is critical to utilize an image scanner with several or all of the capabilities listed here:

Table 2

SECURITY AREAS	SCANNER CAPABILITIES
• Installed OS packages • Installed runtime libraries • Secrets or other sensitive data	• Per-layer scanning • Binary fingerprinting • File contents testing • Open-source license checking
VULNERABILITY TYPES	**VIOLATION TYPES**
• OS-level vulnerabilities • Programming language-specific vulnerabilities	• Database secret embedded in image • Critical severity base image vulnerabilities • Library with unwanted license type used

Image scanning should be a requirement for passing image builds. Results can be used to implement policies that determine whether a build should pass or fail based on the number, severity, and type of vulnerabilities detected in a given image.

As the last line of defense, Kubernetes admission controllers can be configured to implement an `ImagePolicyWebhook` to scan all images before it is deployed to the cluster to prevent manual deployments that circumvent CI/CD pipelines.

BUILD SYSTEMS

The infrastructure, namely build and CI systems and pipelines, used to create these images must also be secured. As the number of tools used to deploy infrastructure and cloud-native applications onto Kubernetes grows, remember to secure the entire CI/CD pipeline. Below are essential security measures to take:

- Limit administrative access to the build infrastructure
- Allow only required network ingress
- Manage any necessary secrets carefully, granting only minimal required permissions
- Use network firewalls to allow access only from trusted sites used to retrieve sources or other files
- Automate vulnerability scanning, license management, and static code analysis for potential security issues in the pipeline

CONTAINER REGISTRY

Once an image has been built, it must also be stored securely in an appropriate image registry. A private, internal registry can provide greater security but imposes the added operational overhead of managing registry infrastructure and relevant access controls. An alternative is to use a registry managed by a cloud platform or other provider. Finally, configure alerts and reports for license violations and vulnerabilities to remediate issues on new and existing images.

SECURING THE KUBERNETES INFRASTRUCTURE

Kubernetes is the critical foundation for how cloud-native applications are deployed and managed. Therefore, security measures to protect the components that make up Kubernetes itself, including remediating vulnerabilities or preventing misconfigurations, are essential to protecting your clusters. Every Kubernetes cluster contains a set of infrastructure components needed to run the platform and applications on it. These components may require administrator or user configuration when provisioning clusters, and understanding them can help focus efforts on valuable security mitigations.

They can be categorized as:

- **Control plane components** – manage operations throughout the cluster.
- **Worker node components** – run containerized applications in pods.

The Kubernetes control plane makes global decisions regarding a cluster's operations. As a result, guarding against threats to its components is paramount since these could compromise the entire cluster environment. Tables 3 and 4 list the control plane and worker node components with corresponding threat vectors and security measures to implement.

Table 3

CONTROL PLANE COMPONENTS			
COMPONENT	**DESCRIPTION**	**EXAMPLE THREAT VECTORS**	**SECURITY MEASURES**
`kube-apiserver`	• Core control plane component that handles API requests to manage the state of cluster resource objects • Primary interface that controls plane and worker node components, workloads, and Kubernetes clients communicate with for ongoing operations	• Privilege escalation vulnerabilities that allow access to cluster-wide resources • Denial-of-Service (DoS) attacks based on using manipulated YAML files	• *Upgrade to a recent version or apply available patches* • *Restrict access to the Kubernetes API by ensuring client authentication, configuring RBAC authorization, and encrypting API traffic* • *Turn on audit logging*
`etcd`	Distributed key-value store that Kubernetes clusters use to store all their data, including data about their resources, configurations, metadata, and overall state	Access allows exposure of full details, including critical and sensitive operational data, about cluster environments	• *Encrypt sensitive information (e.g., secrets in etcd)* • *Apply strong authentication and access control; limit read/write permissions on data*
`kube-scheduler`	Handles pod scheduling and placement across available worker nodes in clusters by considering several parameters (e.g., available resources, affinity and anti-affinity rules, taints, and tolerations)	Compromise can lead to the scheduling of new pods or placement that increases the risk to critical applications (non-sensitive and sensitive workloads running on the same node)	• *Restrict permissions on pod specifications and apply RBAC to apply minimum privileges* • *Configure to only serve HTTPS*
`kube-controller-manager`	Manages the state of resources including nodes, pod replicas, and services	Privilege escalation on control loops can occur when individual service account credentials for each controller are not used	• *Restrict permissions on pod specifications and apply RBAC to apply minimum privileges* • *Configure to only serve HTTPS*
`cloud-controller-manager`	Relevant for load balancing, node termination, and routing in cloud provider environments		

Table 4

WORKER NODE COMPONENTS			
COMPONENT	**DESCRIPTION**	**EXAMPLE THREAT VECTORS**	**SECURITY MEASURES**
`kubelet`	Primary node agent that manages individual containers running in pods and their state based on pod specifications	• Access to the kubelet is unauthenticated by default • Its API endpoints can be used to gain additional access and privileges on nodes and pods	• *Upgrade to the latest version or apply available patches* • *Ensure strong authentication and authorization*
`kube-proxy`	Forwards traffic addressed to virtual IP addresses of Kubernetes services and corresponding pods; serves as a network proxy that applies network rules and protocols (e.g., TCP, UDP)	• Several of its operating modes enable cluster services to be exposed externally • May use relevant `kubeconfig` file without permissions	• *Restrict permissions on the `kubeconfig` file*
Container runtime	Manages and executes containers that deploy and run on individual nodes; handles functions that require interfacing with kernel	Interfaces with kernel for multiple system-level functions, so compromise or container breakouts can result in risks to the underlying node	• *Upgrade to the latest version* • *Harden nodes using Linux security module* • *Apply strong access controls regarding who can control the container runtime*

SECURING DEPLOYED AND RUNNING WORKLOADS

Once the software supply chain (including the images that are built using it) and Kubernetes cluster infrastructure are adequately secured, the remaining focus is on security controls for the Kubernetes pods that are deployed and run. Pods are the smallest units deployed into clusters and collectively form Kubernetes applications, so they are ultimately the targets subject to individual exploits. Securing pods and their containers requires substantial attention — doing so enables more granular security controls that better scope the requirements of individual application components.

Deploy-time and runtime security involve separate security measures but are related in that they should primarily focus on first setting parameters that restrict what containerized applications can do, and then subsequently monitor for any deviations (or attempts to deviate) from those restrictions.

ADMISSION CONTROLLERS

The primary mechanisms for restricting the deployment of pods in Kubernetes are called admission controllers.

Table 5

WHAT THEY ARE	WHAT THEY DO	EXAMPLE
A group of plugins that govern and enforce what is allowed to run within a cluster.	Specify pod requirements before they can be deployed into a cluster; prevent pods that do not meet the conditions from deploying.	Resource policies such as `LimitRanges` and `ResourceQuotas` can be configured to enforce individual resource constraints (e.g., max 2 CPU per pod) or aggregate usage constraints (e.g., a total of 20 CPU in the dev namespace).

SECURITY CONTEXTS

The main Kubernetes controls that allow pod restrictions to be applied at runtime are referred to as **security contexts**. They form an important aspect of how pod runtime security works in Kubernetes.

Table 6

WHAT THEY ARE	WHAT THEY DO	EXAMPLE
Restrictions on individual running pods; these settings can be included in pod manifests.	Setting security contexts helps further harden pods and makes them less exposed to compromise; they can also be scoped to individual containers.	Whether pods run privileged, their filesystems are read-only if they use Linux security modules such as `seccomp`, `AppArmor`, and other system-level capabilities.

KUBERNETES NETWORK POLICIES

Additionally, **network segmentation** of pod-to-pod traffic is another substantial security measure that is required to ensure Kubernetes applications are adequately protected. It limits the impact of an attacker that is able to gain access to any particular group of deployed and running containers, including restricting the ability to move throughout a cluster laterally.

Table 7

WHAT THEY ARE	WHAT THEY DO	EXAMPLE
Policies that enable network isolation between pods.	Configure ingress and egress traffic to and from your applications by restricting pod-to-pod traffic; requires using a network plugin that actually enforces Network Policies.	Apply a "default-deny-all" network policy that isolates all pods; once in place, network policies to enable Internet access and pod-to-pod traffic can be added as needed.

RBAC AND SERVICE ACCOUNTS

Kubernetes uses role-based access control (RBAC) to authorize access to resources within the cluster. By default, Kubernetes mounts the default service account to every pod upon creation. Depending on its permissions, the pod may be granted more permission than required. If the application does not heed to Kubernetes roles, set `automountServiceAccountToken` to `false`.

Otherwise, exercise the principle of least privilege to only give the set of permissions the application needs to interact with Kubernetes or external components (e.g., cloud-specific services through service account mapping).

RUNTIME THREAT DETECTION

Finally, the security concepts outlined above must still be complemented with **runtime threat detection** since no security controls can be considered foolproof. System-level activity such as process execution and network communication can be monitored at the level of individual containers to determine potential indicators of compromise.

Tools such as Falco can monitor `syscalls` and Kubernetes API logs to trigger alerts. Such tools can be configured to alert when privilege escalation occurs or when write events are detected on protected directories. For example, Falco can be configured to detect and produce a warning when non-device files are written in `/dev`:

```
- rule: create_files_below_dev
  desc: creating any files below /dev other than
known programs that manage devices. Some rootkits
hide files in /dev.
  condition: (evt.type = creat or evt.arg.flags
contains O_CREAT) and proc.name != blkid and
fd.directory = /dev and fd.name != /dev/null
  output: "File created below /dev by untrusted
program (user=%user.name command=%proc.cmdline
file=%fd.name)"
  priority: WARNING
```

CONCLUSION

Kubernetes is a powerful yet complex system that is rapidly transforming how organizations build, ship, and run modern software applications. Its benefits come with associated security demands that must be addressed with a multitude of open-source and commercial tools to minimize risks and threats to your business. An effective approach to securing Kubernetes environments and the apps running inside is based on applying controls to secure the following key areas:

- Software supply chain used to build container images, including base images and image components
- Infrastructure components needed to run Kubernetes clusters, including its control plane and worker nodes
- Deployed and running containerized workloads made up of individual pods

In short, the entire software supply chain — from the application code to the deployed containers as well as the Kubernetes cluster — must be secured. Due to the complex nature of cloud-native applications, every single attack vector must be considered to securely deploy applications to Kubernetes. By implementing a multi-layered approach to security, organizations can scale their production usage of Kubernetes with confidence.

WRITTEN BY YITAEK HWANG,
SR. SOFTWARE ENGINEER II, AXONI

Yitaek Hwang is a Senior Software Engineer at Axoni focused on building infrastructure and developer tools for high-performance teams. He often writes about cloud, DevOps/SRE, and crypto topics.

DZone
600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.