The Complete Guide to Kubernetes Security

CROWDSTRIKE

What You'll Learn in This Guide

In the rapidly evolving landscape of cloud-native technologies, Kubernetes has emerged as the orchestration platform of choice for deploying, scaling and managing containerized applications.

This guide is a resource for those tasked with securing Kubernetes deployments, providing a foundation for the knowledge and tools necessary to fortify Kubernetes clusters against common threats. It provides attack examples to illustrate the vast attack surface Kubernetes presents and shows how neglecting hardening measures can lead to breaches.

Readers can expect to learn what attackers look for in Kubernetes, the importance of both shifting left and shielding right, and attributes of a complete and effective solution for securing Kubernetes clusters.

Table of Contents

Introduction

Kubernetes has become the de facto standard for container orchestration following the rise in adoption of cloud-native technologies and DevOps practices. This is something we can expect to remain true for a long time to come, especially since 84% of organizations surveyed in the Cloud Native Computing Foundation (CNCF) Annual Survey 2023 were using or evaluating Kubernetes, up from 81% in 2022.

With Kubernetes' widespread adoption comes the need to ensure robust security measures are in place to protect the infrastructure, applications and data it hosts. The Red Hat 2024 State of Kubernetes security report revealed that two-thirds of organizations experience delayed deployments due to Kubernetes security concerns.

Although Kubernetes provides inherent security advantages — such as enabling isolation (if one container is compromised, the other containers remain unaffected if the attacker is unable to break out) and version control (the ability to easily and quickly roll back an image if a vulnerability is found in new code) — its distributed and ephemeral nature poses unique security challenges.

Since Kubernetes is based on cloud-native architecture, CrowdStrike has structured its Kubernetes security guidance to follow the life cycle phases of cloud-native application development, as detailed by the CNCF. This framework for Kubernetes security will also help illuminate how a robust Kubernetes security program involves both shifting left and shielding right through the lens of the four life cycle phases. In the following pages, you will see examples of ways an attacker might exploit various points of vulnerability, followed by risk mitigation recommendations. This is not meant to be a fully comprehensive technical guide; instead, it is meant to be a point of reference for why and how to infuse Kubernetes security practices from development to runtime.

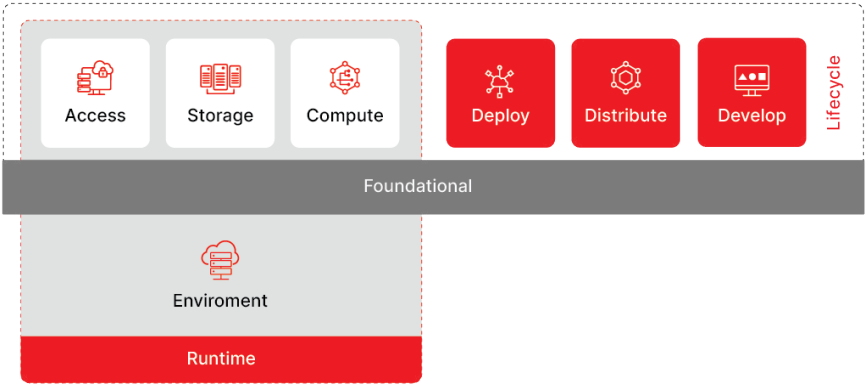
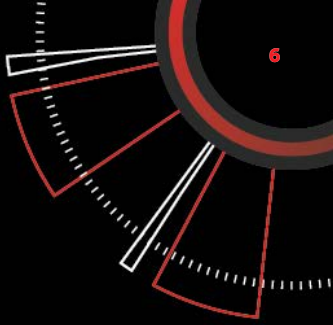


Figure 1. The cloud-native stack — CNCF Cloud Native Security White Paper

Whether you're a seasoned DevOps engineer, a security professional or a newcomer tasked with securing Kubernetes deployments, this guide will serve as a foundation for the knowledge and tools necessary to fortify your Kubernetes clusters against common threats.

If you would like more information, see these recommended resources:

- Kubernetes: kubernetes.io
- Cloud Native Computing Foundation: cncf.io
- OWASP: owasp.org/www-project-kubernetes-top-ten/ and cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html
- CrowdStrike: crowdstrike.com/platform/cloud-security/container-kubernetes/



Chapter 1

Understanding Kubernetes and Its Place in Cloud-Native Architecture

Exploring Cloud-Native Architecture: Foundations and Benefits

In the rapidly evolving technology landscape, cloud-native architecture has emerged as a pivotal paradigm, fundamentally transforming how applications are built, deployed and managed. At its core, cloud-native architecture enables organizations to develop resilient, scalable and agile applications. Unlike traditional monolithic architectures, cloud-native approaches foster an environment where continuous integration and continuous delivery (CI/CD) can thrive. This shift accelerates development cycles and enhances operational efficiency and reliability.

Figure 2. The CI/CD pipeline — What is CI/CD?

One of the primary benefits of cloud-native architecture is its inherent scalability. In a cloud-native environment, applications are designed to scale horizontally, allowing individual components to grow independently based on demand. This flexibility ensures optimal resource utilization, cost-effectiveness and the ability to handle varying workloads seamlessly. Moreover, by adopting microservices, organizations can decouple application components, making it easier to update, maintain and deploy specific parts of an application without affecting the entire system. This modularity enhances the overall agility of the development process, enabling faster innovation and time to market.

Other significant advantages of cloud-native architecture include its resilience and fault tolerance. Cloud-native applications are built with redundancy and failover mechanisms, ensuring high availability even in the face of infrastructure failures. By leveraging container orchestration platforms like Kubernetes, these applications can automatically manage and recover from failures, providing a robust and reliable user experience. Ultimately, cloud-native architecture empowers organizations to deliver high-quality and scalable applications, meeting the demands of modern digital enterprises.

How Kubernetes Unlocks the Full Potential of Cloud-Native Architecture

Manually managing numerous components in the cloud — such as microservices and containers — would be an arduous and error-prone task. This is where Kubernetes, an open-source container orchestration platform, comes into play.

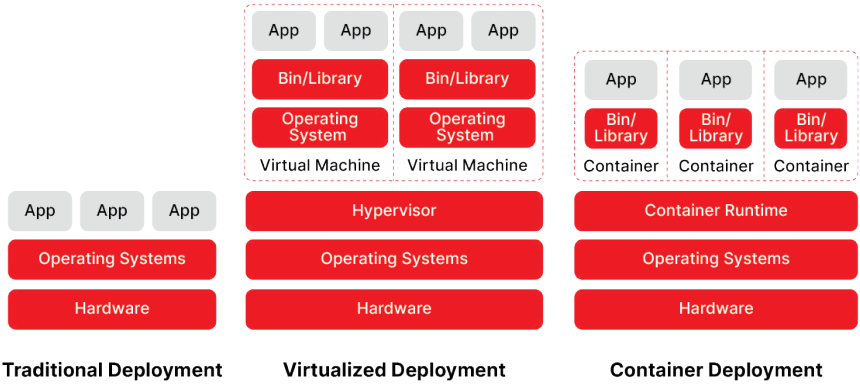


Figure 3. Containers are considered lightweight and portable, compared to virtual machines and traditional deployment mechanisms – Kubernetes Overview

Kubernetes is fundamentally designed to support cloud-native architecture by orchestrating containers in a scalable, resilient and efficient manner. Here are a few ways Kubernetes amplifies the principles of cloud-native architecture:

- **Microservices Architecture:** Cloud-native applications are typically composed of loosely coupled microservices that communicate with each other over the network. Kubernetes excels at managing these microservices by deploying them in containers, which are lightweight and isolated units that encapsulate application code, libraries and dependencies.

- **Containerization:** Containers are a cornerstone of cloud-native architecture, providing consistency across different environments. Kubernetes helps automate the deployment, scaling and operations of containerized applications. It manages container life cycles, handles container networking and ensures containers are always in the desired state.

- **Scalability and Flexibility:** Cloud-native applications require the ability to scale dynamically based on demand. Kubernetes facilitates horizontal scaling by automatically adding or removing container instances and worker nodes in response to load changes. This capability ensures applications can handle varying workloads efficiently without manual intervention.

- **Resilience and High Availability:** Kubernetes inherently supports the resilience and high availability needs of cloud-native applications. It continuously monitors the health of containers and automatically replaces or restarts failed containers. Kubernetes also supports rolling updates and rollbacks, allowing for seamless application updates without downtime, which is crucial for maintaining availability in a cloud-native environment.

Despite its powerful capabilities, deploying and managing Kubernetes clusters can be challenging, particularly from a security perspective. The distributed nature of Kubernetes introduces complexities in securing communications between microservices, managing access controls and protecting sensitive data. Additionally, the dynamic nature of Kubernetes environments means security policies and practices must continuously evolve to address emerging threats. The fundamental nature of Kubernetes opens it up to various security risks, such as:

- **Complex Access Controls:** Kubernetes employs a sophisticated access control system based on role-based access control (RBAC), which can be challenging to configure correctly. Misconfigurations can lead to overly permissive access, allowing unauthorized users to perform actions that could compromise the cluster's security. Enforcing the principle of least privilege requires careful planning and regular audits.

- **Container Image Vulnerabilities:** Containers are built from images that may contain known vulnerabilities if not properly vetted. Additionally, traditional "patching" workflows are not compatible with containers, where the container image is meant to be immutable once it is deployed.

- **Insecure Networks:** Network policies in Kubernetes are essential for controlling the traffic between pods and services. Without proper network segmentation or restrictions of unnecessary communication paths, the cluster's services and internal communications can get exposed.

- **Improper Secrets Management:** Although secrets are a security feature of Kubernetes designed to enable you to store and manage sensitive information securely, it's important that security controls — such as encryption and secret rotation — are implemented to ensure the secrets remain secure throughout their life cycle.

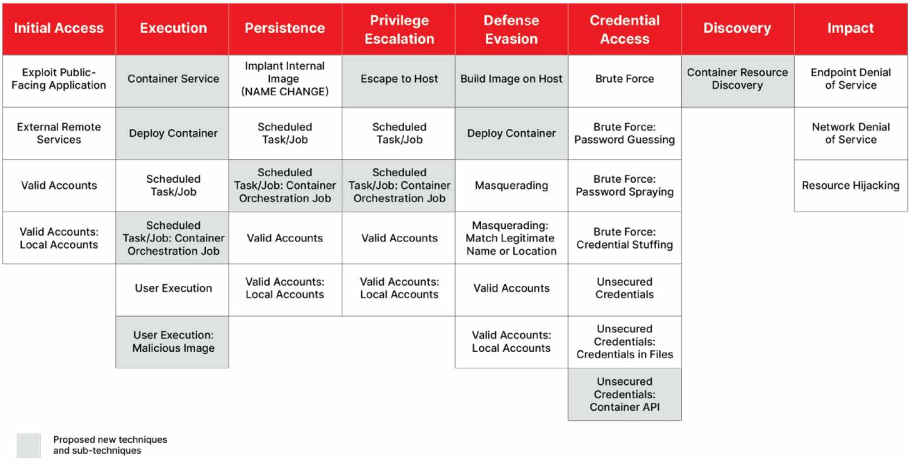
Ensuring that each component within the cluster is properly configured and secured requires a deep understanding of both Kubernetes and cloud security principles.

Chapter 2

Initial Entry Points for Container Compromise

How might an attacker gain access to a container in the first place? Before delving into Kubernetes security best practices, it's important to understand the vectors for initial access for container compromise. The MITRE ATT&CK® framework highlights three common techniques attackers can use to compromise containers:

- **Exploit Public-Facing Application:** Public-facing applications are software applications or services that are accessible to external users over the internet. This exposure to the internet is what typically makes these applications a popular target. Should an attacker successfully gain shell access, they would have access to the underlying container.
- **External Remote Services:** Attackers can leverage exposed Kubernetes services, and some may not even require authentication. Examples include an exposed Docker API or web application such as the Kubernetes dashboard.
- **Valid Accounts:** Valid cloud credentials can provide an attacker with the means to bypass access controls and gain access to the cluster's management layer undetected. Valid cloud credentials can be obtained in various ways, such as through phishing or exposure in a public code repository.



Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Impact
Exploit Public-Facing Application	Container Service	Implant Internal Image (NAME CHANGE)	Escape to Host	Build Image on Host	Brute Force	Container Resource Discovery	Endpoint Denial of Service
External Remote Services	Deploy Container	Scheduled Task/Job	Scheduled Task/Job	Deploy Container	Brute Force: Password Guessing		Network Denial of Service
Valid Accounts	Scheduled Task/Job	Scheduled Task/Job: Container Orchestration Job	Scheduled Task/Job: Container Orchestration Job	Masquerading	Brute Force: Password Spraying		Resource Hijacking
Valid Accounts: Local Accounts	Scheduled Task/Job: Container Orchestration Job	Valid Accounts	Valid Accounts	Masquerading: Match Legitimate Name or Location	Brute Force: Credential Stuffing		
	User Execution	Valid Accounts: Local Accounts	Valid Accounts: Local Accounts	Valid Accounts	Unsecured Credentials		
	User Execution: Malicious Image			Valid Accounts: Local Accounts	Unsecured Credentials: Credentials in Files		
					Unsecured Credentials: Container API		

Proposed new techniques and sub-techniques

Figure 4. The ATT&CK for Containers matrix – MITRE Engenuity

Managed vs. Unmanaged Kubernetes Clusters: Do These Techniques Apply to Both?

These three attack techniques remain valid threats to both managed and unmanaged clusters. One key concept to remember when securing Kubernetes is the shared responsibility model. Typically, in a cloud-native environment, security responsibilities are shared among the cloud provider, platform operator and application developer. Although managed Kubernetes clusters, like those provided by Google Kubernetes Engine (GKE) or Amazon Elastic Kubernetes Service (EKS), handle many operational tasks and can help ease the burden of securing the control plane, teams should still adopt the mindset of "trust, but verify." Though you can trust cloud providers to secure their infrastructure, verification ensures that no gaps exist in areas you are responsible for, such as application security, network policies and user access controls.

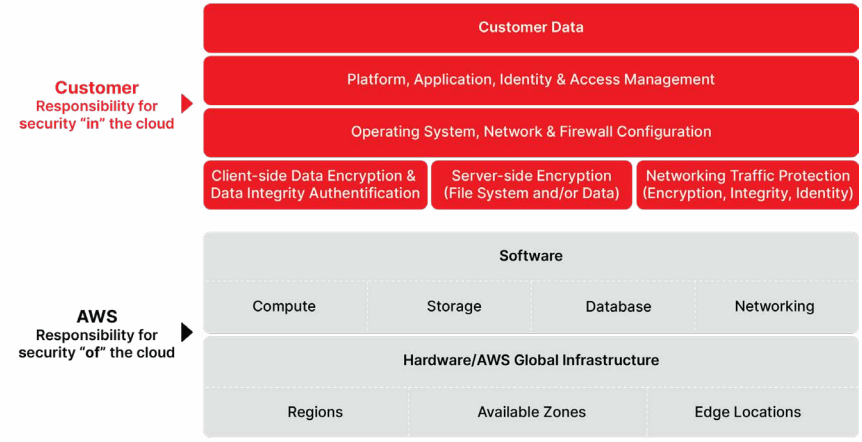


Figure 5. The AWS Shared Responsibility Model

Depending on how you choose to host your Kubernetes clusters, the default configuration for the control plane API may change slightly. Many examples in the following sections assume you are managing your Kubernetes cluster yourself (with defaults lacking guardrails), but they still serve as a reminder to review your default configurations no matter how your clusters are hosted.

Chapter 3

Kubernetes Security Aligned to the Cloud-Native Application Development Life Cycles

The Kubernetes platform offers built-in security functionalities — such as namespace isolation, RBAC and network policies — that help enforce security boundaries and manage permissions. These capabilities provide a good starting foundation for a secure cloud-native environment (if configured properly), but they are not comprehensive solutions on their own, as Kubernetes itself is not a security tool. The security of a Kubernetes cluster requires a layered approach, addressing both the orchestrator and the applications it manages.

Security in Kubernetes is not a feature that can simply be toggled on or off. It requires a continuous set of practices integrated across every level of the system and throughout the software development life cycle (SDLC). This holistic approach is essential for maintaining a secure and resilient Kubernetes environment.

Understanding the Kubernetes Architecture

Before jumping into the phases of the cloud-native application development life cycle, it's important to first understand the core architecture of Kubernetes clusters.

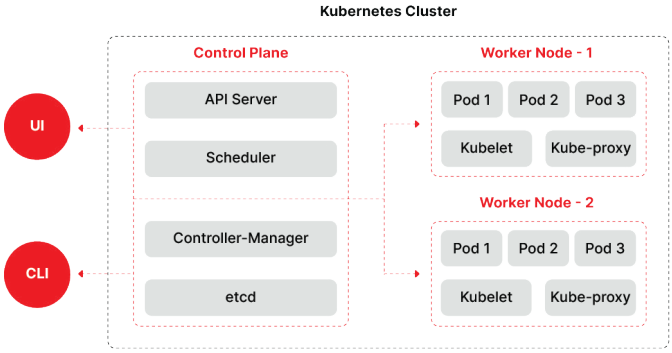


Figure 6. A high-level diagram of Kubernetes architecture

A typical Kubernetes cluster is made up of:

The Control Plane

The control plane is responsible for managing the entire Kubernetes cluster. It makes decisions about resource allocation, manages workloads and ensures everything runs smoothly. Key components include:

- **API Server:** This is the central communication hub. It authenticates users and services, ensuring only authorized actions are allowed.
- **Controller Manager:** Monitors the cluster's health and ensures that everything is functioning as expected by aligning the current state with the desired state.
- **etcd:** A storage system that keeps all of the important information about the cluster's configuration and state.
- **Scheduler:** Decides which resources will be used for tasks, making sure workloads are distributed efficiently across the cluster.

Worker nodes

Worker nodes are the part of the Kubernetes cluster that handle all of the actual processing and running of applications. They host your workloads and handle the traffic between them. Key components include:

- **Pods:** These are the smallest building blocks in Kubernetes and contain the application or service that runs within the cluster.
- **Services:** Help group together related pods and make them available over the network, allowing them to communicate with each other.
- **Kubelet:** Ensures that the containers in each pod are running correctly and talks to the control plane to manage workloads.
- **Kube-proxy:** Maintains the networking rules that allow communication between services and pods.
- **Ingress:** Manages how external traffic, like HTTP and HTTPS requests, reaches services inside the cluster.

After examining each of the Kubernetes components, you can see how each part plays a role in the security of the cluster. For instance, etcd stores sensitive cluster data, and the API server allows fine-tuned authorization. However, misconfiguration of any component (not to mention risk brought on by the supply chain) can expose your environment to significant vulnerabilities, making it prone to various types of attacks.

We will now review the four phases of the cloud-native application development lifecycle — Develop, Distribute, Deploy and Runtime — to understand how a lack of security practices at each phase can facilitate either initial access or persistence for an attacker in Kubernetes environments. The main goal of Chapter 3a and Chapter 3b is to provide insight into how security gaps at each stage in the DevOps life cycle can provide low-cost opportunities for attackers to wreak havoc.

Note: This is not meant to be a comprehensive guide on penetration testing Kubernetes environments, and we do not recommend trying any attack techniques without having explicit permission to do so. This is simply meant to show examples of how an attacker can take advantage of various weaknesses one might find in a Kubernetes environment to encourage proper security hardening and monitoring.

Chapter 3a: Develop and Distribute

Security practices must be introduced in the "Develop" phase at the start of the software development life cycle. In the "Distribute" phase, injecting security practices ensures applications are packaged, stored and tested securely to prevent tampering, unauthorized access and other potential threats.

Embedding security in these early stages is critical because it establishes the configurations that form the foundation of the risk posture for cloud-native applications on Kubernetes, including application and container manifests, application code, runtime specifications, etc. Additionally, many organizations find it to be the most cost-effective way to reduce security risk, since identifying and fixing points of vulnerability during the early stages of development is significantly less expensive than addressing them in production. For instance, according to the CNCF, it can be up to 640 times more expensive to address issues in production compared to fixing them during development. Despite the advantages of bringing security best practices early in the DevOps life cycle, according to Gartner®,[1] "Security teams are often isolated from the DevOps continuous integration/continuous delivery (CI/CD) value stream, leading to bottlenecks and wait states that inhibit agility and improved security."

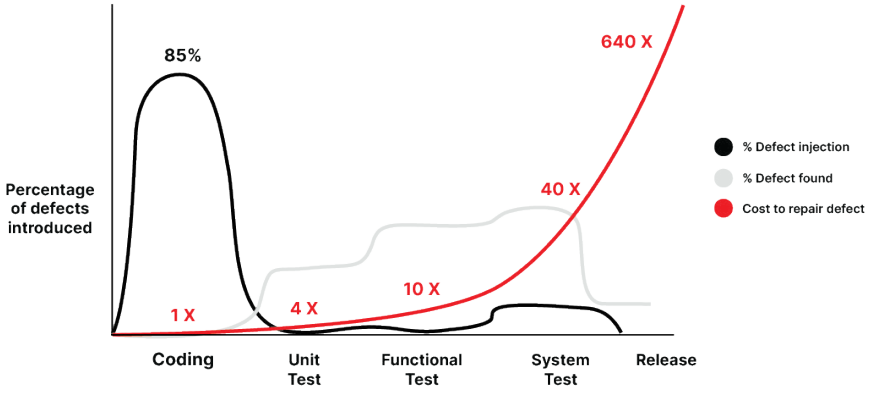


Figure 7. The Caper Jones graph, which shows the percentage of defects introduced during each phase of the development life cycle and how the cost to repair the defects goes from 1x when coding to 640x in production — The Cost of a Kubernetes Repair in Development vs. Production

[1] Gartner, 3 Essential Steps to Enable Security in DevOps, Daniel Betts, Manjunath Bhat, et al., 1 October 2024 https://www.gartner.com/document-reader/document/4145599?ref=TypeAheadSearch (Report accessible to Gartner subscriber only)

Building security checkpoints in between development and deployment is a necessary process, especially since within Kubernetes environments, we tend to see the following patterns:

- Increased dependency on third-party components

  Kubernetes environments often involve a vast array of third-party components (images, libraries, etc.), each of which can introduce vulnerabilities if compromised. Additionally, since Kubernetes environments are highly dynamic, these components are frequently updated, increasing the attack surface.

- Increased automation and the use of CI/CD pipelines

  CI/CD pipelines enable teams to become more efficient through automation, but they can be a double-edged sword: Increased automation can make it much easier for a compromised component to propagate quickly through the environment.

Let's explore a few example attacks that could be possible if security-focused checks are not embedded ahead of deployment.

Example Attacks

Exploiting Infrastructure as Code (IaC) Misconfigurations

In a Kubernetes environment, IaC plays a critical role in defining and managing infrastructure, including the Kubernetes cluster configurations themselves. Teams can use IaC tools to automatically apply Kubernetes configurations across multiple clusters instead of manually creating or updating deployments. IaC template security vulnerabilities are crucial to secure since IaC security risks can quickly become widespread.

An engineer might, for example, inadvertently assign overly broad permissions to an identity and access management (IAM) role, leave a cloud storage bucket open to the public, include hard-coded secrets or allow unrestricted inbound/outbound network traffic in their IaC template. All of this would provide an attacker with the means to breach the environment.

```
Unset
provider "aws" {
  region = "us-east-1"
  access_key = "AKIAEXAMPLEACCESSKEY"  # Hardcoded secret
  secret_key = "abc123exampleSecretKey"  # Hardcoded secret
}

resource "aws_instance" "example_instance" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = "ExampleInstance"
  }
}
```

Figure 8. This example Terraform file includes AWS credentials directly in the code, which is a major security risk. If the file is shared, stored in version control (like GitHub), or inadvertently exposed, the secrets can be compromised, leading to unauthorized access to AWS resources.

Typosquatting

A typosquatting attack in Kubernetes is a form of supply chain attack where an attacker creates malicious packages or containers with names that are very similar to legitimate ones, with the aim of having developers accidentally download and use them due to typographical errors. It should be noted that this is a relatively low-cost attack for threat actors to try.

Let's say we have an attacker, Alice, who publishes a malicious container image with a name similar to a frequently used image in Kubernetes environments (for example, "ngnix" instead of "nginx"). This container image is designed to deploy malware, and Alice publishes it to a public container registry.

```
Unset
FROM ngnix:latest
```

Figure 9. Typo in a Dockerfile that pulls a malicious image

Should a developer accidentally build a container from a compromised base image, which could contain a legitimate nginx server and malicious code, Alice could have the means to successfully create a reverse shell. With shell access, Alice can further her attack in a variety of ways. For example, she could target the CI/CD pipeline with a script that is meant to modify configurations or insert additional malicious software into builds, infecting all future builds with the malicious image. The compromised build pipeline can now inadvertently push infected images to production.

Compromising Source Code Repositories

Attackers can gain unauthorized access to source code repositories (e.g., GitHub, GitLab) and inject malicious code into the application. This can lead to the deployment of compromised applications within the Kubernetes environment. Techniques for compromising source code repositories include exploiting weak passwords, gaining the trust of the open-source community (as seen with CVE-2024-3094), phishing attacks to steal credentials or exploiting vulnerabilities in the repository hosting service.

Security Implementation Recommendations

Develop Life Cycle Phase

- Define and test security policies (e.g., Pod Security Standards) to ensure security controls are enforced when deployed
- Define secure IaC configurations and integrate IaC scans into existing pre-deployment DevOps workstreams
- Establish a code review process that involves both developers and platform engineers to provide opportunities to double-check code prior to merging into the codebase

Distribute Life Cycle Phase

- Always run the latest stable version of Kubernetes or the latest stable version of your preferred vendor's Kubernetes distribution (e.g., Red Hat OpenShift or SUSE Rancher)
- Always verify the source of packages and container images before installing them
- Use trusted repositories and official sources

- Use strict naming policies and enforce them across the development and operations teams to minimize the risk of typographical errors
- Employ automated tools to scan for and detect malicious packages; tools like container image scanners can help identify known vulnerabilities and suspicious behavior
- Scan container images and check the results against pipeline compliance rules to prevent insufficiently patched applications from deploying to production
- Ensure images are up to date
- Use minimized base container images that reduce the attack surface, such as Google's Distroless, Red Hat's UBI Micro, Canonical's Chiselled Ubuntu or empty images when possible
- Subscribe to feeds that announce new vulnerability patches or dependency updates
- Leverage private registries for storing container images or to access remote registries (like vendors' registries)
- Digitally sign image content during build, and implement checkpoints for validation before deployment
- Periodically re-scan container images running in your cluster — not just during the build pipeline — to ensure emerging vulnerabilities can be detected and remediated

Chapter 3b: Deploy and Runtime

The "Deploy" phase provides the opportunity to do two things from a security perspective: (1) verify the container images passed all checks that were implemented during the "Distribute" life cycle phase and (2) set the security foundation for the application's runtime environment. Within this phase, users can create various objects that will handle the life cycle of a Kubernetes-based application — such as pods, services and ingresses — in manifest files that Kubernetes knows how to manage. For the "Runtime" phase, as stated in the Kubernetes documentation, security monitoring should cover three main components: **compute**, **access** and **storage**.

Although it may seem redundant to implement security checks right before deployment and after, both phases are necessary. The former is a critical last effort to ensure anything being deployed to production is going as intended; the latter is a fail-safe for security weaknesses that might slip through to production. Unlike proactive hardening measures, which aim to reduce risk before deployment, runtime security ensures the system remains secure during operation.

Although shifting left is a key part of secure development in Kubernetes, it would be a huge security gap to not implement continuous, real-time security monitoring at runtime. This must also include advanced response capabilities, since threats and vulnerabilities can emerge at any stage of the application life cycle (including post-deployment), requiring immediate detection and mitigation to prevent potential breaches.

The following example attacks become possible if there are security gaps within these life cycle phases.

Example Attacks

Sidecar Container Injection

Pods are composed of one or more containers that share the same network namespace, IP address and storage. A pod is meant to represent an instance of an application, but it can also include sidecar containers. These are secondary containers that run alongside the main application container to extend its functionality without altering the logic of the main application container. They are typically leveraged for supplementary functionality such as logging or metrics monitoring.

An attacker can either manipulate the functionality of an existing sidecar container or inject their own, allowing them to avoid deploying a new pod in the cluster. Since sidecar containers run on legitimate pods, they can serve as a mechanism for an attacker to maintain stealth.

Performing a Denial-of-Service (DoS) Attack with a Fork Bomb

A fork bomb is a DoS attack where a process continually replicates itself to deplete system resources, leading to a system crash. If an attacker gains shell access through vulnerabilities in applications or services running on the pods, they have an opportunity to perform a DoS attack.

```
Unset
:(){ :|:& };:
```

Figure 10. Commonly used fork bomb code

Attacking the API Server

The API server is the highest priority for an attacker to target, since it is the main function used to control the Kubernetes platform. Attackers can search for publicly available, unauthenticated kubelet APIs using freely available tools, such as Shodan. Since the kubelet API is exposed on port 10250 by default, the following search query can be used to search for servers listening on 10250 and return a 404 error without a URL path.

```
Unset
port:10250 ssl:true 404
```

Figure 11. Shodan example search query for publicly available, unauthenticated kubelet APIs

If any IP addresses are returned with the above search query, an attacker can then query the runningpods API to find a detailed list of the running containers.

```
Unset
$ curl -s -k https://${IP_ADDRESS}:10250/runningpods/
```

Figure 12. Query **runningpods** API for a detailed list of running containers

Security Implementation Recommendations

Deploy Life Cycle Phase

- Define and apply secure Kubernetes network policies to restrict traffic and ensure these policies are properly enforced with a network policy provider
- Encrypt traffic that will travel on the wire
- Avoid the use of tags and use the image sha256 hash instead
- Set namespaces to isolate Kubernetes resources (and block the usage of the default namespace)
- Use container-optimized operating systems for your Kubernetes nodes, as these operating systems limit the surface attack and ensure your environment is optimized for security, performance and reliability

- Use the admission controller ImagePolicyWebhook to ensure only approved images are deployed to production
- Leverage projects such as the Open Policy Agent (OPA) to enforce centralized policy management
- Apply security context to pods and containers with the principle of least privilege
- Block access to network ports and limit access to the Kubernetes API server
- According to OWASP, if vulnerabilities are found in running containers, always update the source image and redeploy the containers — don't directly update the running container, as it can break the image-container relationship
- Create separate namespaces for different applications and environments to limit the scope of any potential sidecar injection
- Verify the integrity of any artifacts
- Restrict service accounts used by applications and ensure they do not have unnecessary permissions that could allow them to modify other pods or inject sidecars
- Apply restrictive Pod Security Standards to limit what containers can do within the cluster, and ensure only trusted and necessary containers are allowed to run
- Follow best practices for hardening your Kubernetes nodes, such as disabling unused services, applying security patches, and using minimal and secure base images
- Check for any changes or regressions that may have occurred in the CI/CD pipeline

Runtime Life Cycle Phase

- Design Zero Trust into the architecture of microservices running on containers
- Configure containers to use read-only root file systems wherever possible to minimize the risk of tampering with the container environment during runtime, and use container-optimized operating systems that provide limited writable locations on disk
- Define short lifetimes for certificates and automate rotation

- Define resource limits and quotas for CPU and memory in your pod specifications to prevent any single pod from consuming excessive resources
- Use Pod Security Standards to restrict privileged access and capabilities
- Prevent ServiceAccount's API credentials from being automounted on services and pods
- When dealing with multi-tenant or highly untrusted clusters, practice container sandboxing or isolate running containers from the host kernel
- Prevent containers from loading unwanted kernel modules by using a tool like SELinux
- Leverage an external storage plugin that can provide encryption at rest for volumes
- Design authentication mechanisms between cluster nodes and storage on the network
- Review your Kubernetes cluster against the Kubernetes Security Checklist
- Leverage an advanced container security solution that can monitor for any threats or vulnerabilities at runtime
- Track runtime activity across pods in the same deployments to identify anomalies

Chapter 4

Putting It All into Practice: Building an Effective Kubernetes Security Program

Now that we've walked through what the Kubernetes attack surface looks like and we've reviewed recommendations for securing Kubernetes clusters across the cloud-native application life cycle, it's time to discuss how to build a robust Kubernetes security program that is evergreen, capable of spanning multiple teams and sustainable in an ever-changing threat landscape. Kubernetes environments are characterized by their scalability, frequent changes and use of numerous interconnected components, which can create a broad attack surface. Hardening best practices are not sufficient for full coverage.

We recommend that alongside the Kubernetes hardening best practices throughout this guide, you should leverage a tool that can help with the following:

Aligning Security and DevOps

"Shifting left" or "shielding right" is not effective if both occur in silos. For Kubernetes security, it is imperative for security and DevOps teams to collaborate to go from detecting a high-priority security issue to fixing it in a timely manner. This is because both teams provide different perspectives that need to be considered together to effectively prioritize and respond to security risks. DevOps teams have ownership of — and deep knowledge about — the development and deployment of their Kubernetes clusters, and security teams have insight into the threat landscape.

Reporting and Dashboards to Provide Shared Understanding

To streamline visibility and drive alignment across teams, organizations should look for a security solution that creates a common language between the DevOps and security perspectives.

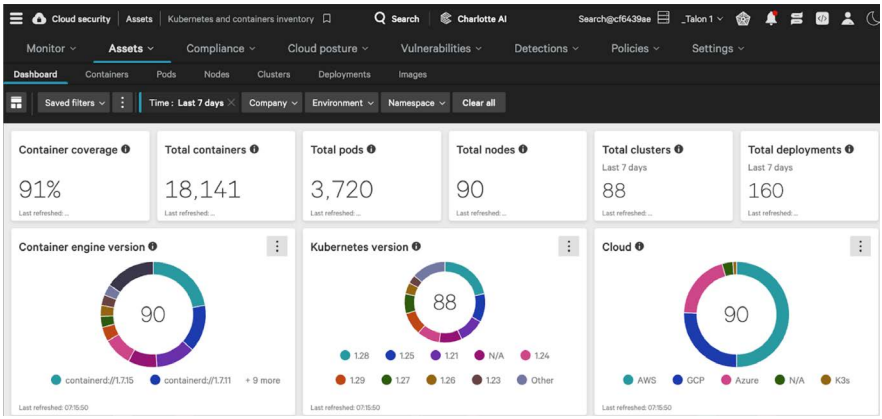


Figure 13. The Kubernetes dashboard in CrowdStrike Falcon® Cloud Security provides full visibility into all Kubernetes clusters deployed in production

Integrating Security Practices Within Existing DevOps Toolchains

Leveraging existing DevOps toolchains for security ensures security measures are scalable and adaptable and that they can keep pace with the rapid deployment cycles typical of Kubernetes environments. By embedding security tools and practices directly into the DevOps workflow:

- DevOps teams don't have to context switch or disrupt their daily operations to bring a security perspective to their work
- Security teams become closer to the tools and knowledge used for development and deployment, which enhances their understanding of various security risks detected

Outsmarting Modern Adversaries

Attackers are constantly innovating to devise new ways to automate and scale attacks. To stay a step ahead, those tasked with securing their Kubernetes clusters must understand adversary motivations and anticipate their tactics, techniques and procedures (TTPs). To achieve this, look for a solution that can help you with (1) holistic visibility across multi-cloud and hybrid deployments to ensure adversaries can't hide in preventable gaps and (2) reliable real-time threat and adversary intelligence to make sure you recognize indicators of compromise (IOCs) for known and zero-day attacks.

Visibility to Avoid Dangerous Blind Spots

In Kubernetes environments, lacking holistic visibility coverage across multi-cloud and hybrid deployments can create significant blind spots that adversaries can exploit. These gaps in visibility mean that security teams might miss critical IOCs or fail to detect subtle signs of an ongoing attack. Without comprehensive monitoring, malicious activities can go unnoticed until it's too late. Additionally, the dynamic nature of Kubernetes — with its rapid scaling and frequent changes — exacerbates these risks, making it even more challenging to maintain an effective security posture without full visibility.

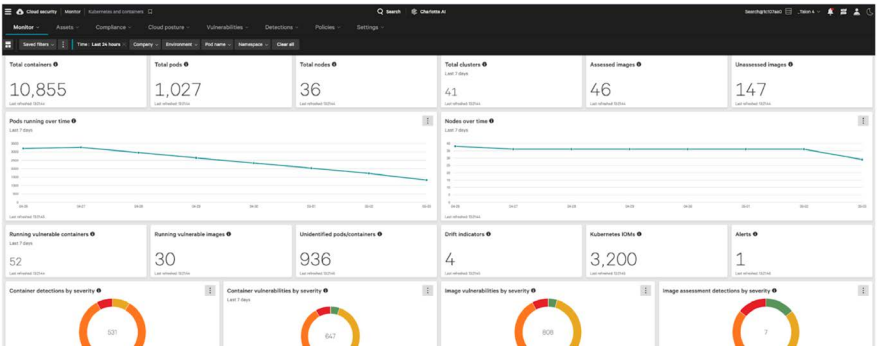


Figure 14. CrowdStrike Falcon Cloud Security provides unified visibility with a single, lightweight agent, illuminating security risks from code to control plane to cloud. Additionally, Falcon Cloud Security captures relevant details, such as container start and stop, image and runtime information, unidentified and rogue containers, and events generated inside containers.

Threat Intelligence to Understand Modern Cloud Adversaries

In a modern cloud environment, threat intelligence is critical for staying ahead of cloud attacks. Adversaries are constantly evolving their TTPs, making it essential for organizations to have real-time visibility into emerging threats. Effective threat intelligence and hunting enable security operations teams to rapidly identify and respond to threats by providing critical context, such as the attribution of attackers, their motives and the infrastructure they leverage. With this insight, teams can prioritize responses and anticipate potential attack paths before they escalate, reducing the time it takes to mitigate threats. By embedding this intelligence directly into cloud-native security solutions, organizations gain continuous, automated protection. Integrated threat intelligence enables cloud security detection engines to adapt dynamically to evolving threats.

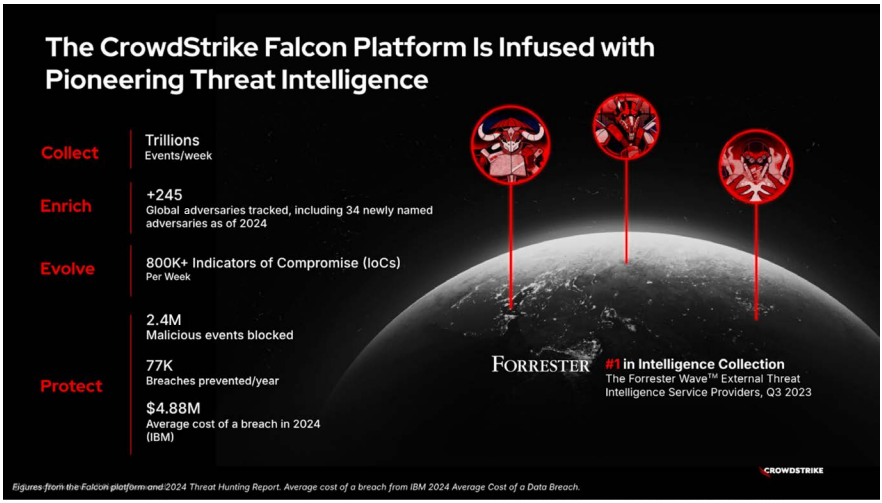


Figure 15. The CrowdStrike Falcon® platform processes trillions of events per week from millions of endpoints worldwide. CrowdStrike's intelligence team tracks 245+ threat actors, publishing more than 800,000 indicators of compromise weekly. This constant flow of real-time data enables CrowdStrike to detect and stop even the most advanced threats before they can impact the organization.

Moving Faster than Sophisticated Attacks

To accelerate detection and response, organizations must proactively hunt for threats and prevent incidents before they occur. Look for a tool that consolidates all of the security solutions necessary to secure Kubernetes across the SDLC. The tool should also employ automation to eliminate the manual effort required to operationalize and contextualize data and take the right action.

Tool Consolidation to Streamline Operations

The traditional approach to cloud security relies on disparate tools from multiple vendors. This forces administrators to toggle between cloud workload protection (CWP), container image scanning, and other tools and screens to create a holistic view of risk. The complexity created by a patchwork approach increases the likelihood of visibility gaps and delays decision-making and response. Siloed tools provide a fragmented view that lacks sufficient context to prioritize threats. Operations also suffer, as having multiple tools generates an excessive number of alerts to investigate, increases the potential for vulnerabilities and misconfigurations to go unnoticed, and consumes more cycles to configure and maintain.

These inefficiencies add risk and drive up costs while making it harder to maintain compliance and strengthen your Kubernetes security posture.

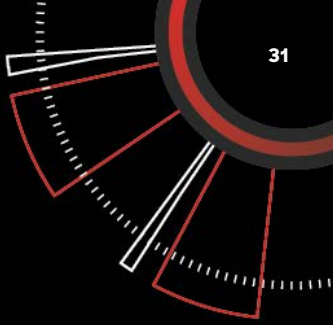
Automation Built to Scale

It's important to look for a solution that integrates and automates security throughout the cloud-native application life cycle. Manual efforts slow down security operations and application delivery while increasing the potential for human error. Without automated response, high volumes of alerts can delay decision-making among resource-constrained security staff. A modern cloud security platform should do more than tell administrators a problem exists — it should allow them to take steps to address weaknesses.

Deep Cloud and Security Expertise

To maximize the value of your investments in a security solution, teams must possess the expertise to leverage it fully. Organizations can acquire these skills by engaging with trusted managed detection and response (MDR) providers that bring deep cloud and security knowledge, allowing them to achieve 24/7 efficiency gains without having to hire and train in-house experts. An MDR provider should function as an extension of your team — working with the provider should be like adding a highly skilled analyst that brings a wealth of global knowledge and decades of experience. Choose a security leader with a proven history of security and cloud innovation and outsmarting clever cloud adversaries. Make sure the platform is backed by expert services to detect, respond to and prevent potential threats.

Figure 16. CrowdStrike possesses the breadth of knowledge and depth of skilled resources to meet enterprises wherever they are in their cloud maturity journey.



CrowdStrike Falcon Cloud Security: Secure Your Kubernetes Clusters Across the SDLC

CrowdStrike Falcon® Cloud Security delivers container, Kubernetes and host protection from build to runtime in AWS, Azure and Google Cloud while ensuring security in every step of the CI/CD pipeline. With Falcon Cloud Security, organizations can automate security and detect and stop suspicious activity, zero-day attacks and risky behavior to stay ahead of threats and reduce the attack surface. Falcon Cloud Security supports CI/CD workflows, allowing you to secure workloads at the speed of DevOps without sacrificing performance.

Effectively Secure Kubernetes Without Compromising on Innovation

Conventional approaches to security can't deliver the granular visibility and control needed to manage cloud risk, particularly risk associated with Kubernetes clusters and containers. Falcon Cloud Security ensures teams are able to move quickly to stay competitive by offering a Kubernetes security solution that includes:

- **Integrated Kubernetes security within DevOps workflows:** CrowdStrike provides many pre-runtime security capabilities. This includes the Falcon Cloud Security Kubernetes Admission Controller, which detects, alerts on and blocks Kubernetes objects when they are created or updated. CrowdStrike also offers vulnerability scans in container images and supports multiple IaC platforms.
- **Combined agent-based and agentless monitoring to eliminate gaps:** Where agentless-only approaches typically take snapshots of cloud risk once or twice per day, CrowdStrike delivers 24/7 continuous visibility coverage. This real-time insight is essential for stopping breaches.
- **World-renowned adversary intelligence:** Falcon Cloud Security integrates CrowdStrike's award-winning threat intelligence collected from protecting thousands of customer organizations worldwide.
- **Cloud-native detection and response on a single platform:** CrowdStrike offers comprehensive protection, from pre-runtime checks and misconfiguration identification to drift prevention and runtime threat mitigation using indicators of attack. Safeguard your operations seamlessly as you build and run your applications.
- **A complete security platform for Kubernetes and beyond:** The Falcon platform consolidates a wide range of point products used to protect and monitor endpoints, cloud workloads, identity and data to provide end-to-end coverage and eliminate complexity.

Conclusion

Kubernetes presents unique security challenges due to its dynamic and distributed nature, which involves managing numerous containers across different environments. Additionally, the complexity of its architecture — including components like the API server, etcd and kubelet — creates multiple potential attack surfaces that require diligent configuration and monitoring. However, building a Kubernetes security approach that aligns with the CNCF cloud-native application development life cycle provides not only an opportunity for reducing significant risk within your Kubernetes clusters but also effective protections in production. Although there are native Kubernetes security features, it's essential to adopt a solution that focuses on building and innovating for the sole outcome of security. Falcon Cloud Security scales security across your CI/CD pipelines by providing end-to-end protection with continuous image scanning, illuminating every Kubernetes cluster and container with discovery and mapping across public and private clouds, and automating testing for rapid detection of common Kubernetes threats (e.g., misconfigurations, loose permissions and vulnerable dependencies).

About CrowdStrike

CrowdStrike (Nasdaq: CRWD), a global cybersecurity leader, has redefined modern security with the world's most advanced cloud-native platform for protecting critical areas of enterprise risk — endpoints and cloud workloads, identity and data.

Powered by the CrowdStrike Security Cloud and world-class AI, the CrowdStrike Falcon® platform leverages real-time indicators of attack, threat intelligence, evolving adversary tradecraft and enriched telemetry from across the enterprise to deliver hyper-accurate detections, automated protection and remediation, elite threat hunting and prioritized observability of vulnerabilities.

Purpose-built in the cloud with a single lightweight-agent architecture, the Falcon platform delivers rapid and scalable deployment, superior protection and performance, reduced complexity and immediate time-to-value.

CrowdStrike: We stop breaches.

Learn more: https://www.crowdstrike.com/

Follow us: Blog | X | LinkedIn | Facebook | Instagram