National Security Agency
Cybersecurity and Infrastructure Security Agency

Cybersecurity Technical Report

Kubernetes Hardening Guide

August 2022

U/OO/168286-21
PP-22-0324
Version 1.2

Notices and history

Document change history

Date	Version	Description
August 2021	1.0	Initial publication
March 2022	1.1	Updated guidance based on industry feedback
August 2022	1.2	Corrected automountServiceAccountToken (Authentication and Authorization), clarified ClusterRoleBinding (Appendix K)

Disclaimer of warranties and endorsement

The information and opinions contained in this document are provided "as is" and without any warranties or guarantees. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, and this guide shall not be used for advertising or product endorsement purposes.

Trademark recognition

Kubernetes is a registered trademark of The Linux Foundation. ▪ SELinux is a registered trademark of the National Security Agency. ▪ AppArmor is a registered trademark of SUSE LLC. ▪ Windows and Hyper-V are registered trademarks of Microsoft Corporation. ▪ ETCD is a registered trademark of CoreOS, Inc. ▪ Syslog-ng is a registered trademark of One Identity Software International Designated Activity Company. ▪ Prometheus is a registered trademark of The Linux Foundation. ▪ Grafana is a registered trademark of Raintank, Inc. dba Grafana Labs ▪ Elasticsearch and ELK Stack are registered trademarks of Elasticsearch B.V.

Copyright recognition

Acknowledgements

NSA and CISA acknowledge the feedback received from numerous partners and the cybersecurity community on the previous version of this report, and thank them for their help in making it better. Changes have been incorporated where appropriate.

Publication information

Author(s)

National Security Agency (NSA)
Cybersecurity Directorate
Endpoint Security

Cybersecurity and Infrastructure Security Agency (CISA)

Contact information

Client Requirements / General Cybersecurity Inquiries:
Cybersecurity Requirements Center, 410-854-4200, Cybersecurity_Requests@nsa.gov

Media inquiries / Press Desk:
Media Relations, 443-634-0721, MediaRelations@nsa.gov

For incident response resources, contact CISA at CISAServiceDesk@cisa.dhs.gov.

Purpose

NSA and CISA developed this document in furtherance of their respective cybersecurity missions, including their responsibilities to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

Executive summary

Kubernetes® is an open-source system that automates the deployment, scaling, and management of applications run in containers, and is often hosted in a cloud environment. Using this type of virtualized infrastructure can provide several flexibility and security benefits compared to traditional, monolithic software platforms. However, securely managing everything from microservices to the underlying infrastructure introduces other complexities. This report is designed to help organizations handle Kubernetes-associated risks and enjoy the benefits of using this technology.

Three common sources of compromise in Kubernetes are supply chain risks, malicious threat actors, and insider threats. Supply chain risks are often challenging to mitigate and can arise in the container build cycle or infrastructure acquisition. Malicious threat actors can exploit vulnerabilities and misconfigurations in components of the Kubernetes architecture, such as the control plane, worker nodes, or containerized applications. Insider threats can be administrators, users, or cloud service providers. Insiders with special access to an organization's Kubernetes infrastructure may be able to abuse these privileges.

This guide describes the security challenges associated with setting up and securing a Kubernetes cluster. It includes strategies for system administrators and developers of National Security Systems, helping them avoid common misconfigurations and implement recommended hardening measures and mitigations when deploying Kubernetes. This guide details the following mitigations:

- Scan containers and Pods for vulnerabilities or misconfigurations.
- Run containers and Pods with the least privileges possible.
- Use network separation to control the amount of damage a compromise can cause.
- Use firewalls to limit unneeded network connectivity and use encryption to protect confidentiality.
- Use strong authentication and authorization to limit user and administrator access as well as to limit the attack surface.
- Capture and monitor audit logs so that administrators can be alerted to potential malicious activity.
- Periodically review all Kubernetes settings and use vulnerability scans to ensure risks are appropriately accounted for and security patches are applied.

For additional security hardening guidance, see the Center for Internet Security Kubernetes benchmarks, the Docker and Kubernetes Security Technical Implementation Guides, the Cybersecurity and Infrastructure Security Agency (CISA) analysis report, and Kubernetes documentation [1], [2], [3], [6].

Contents

Figures

Tables

Introduction

Kubernetes, frequently abbreviated “K8s” because there are 8 letters between K and S, is an open-source container-orchestration system used to automate deploying, scaling, and managing containerized applications. As illustrated in the following figure, it manages all elements that make up a cluster, from each microservice in an application to entire clusters. Using containerized applications as microservices provides more flexibility and security benefits compared to monolithic software platforms, but also can introduce other complexities.

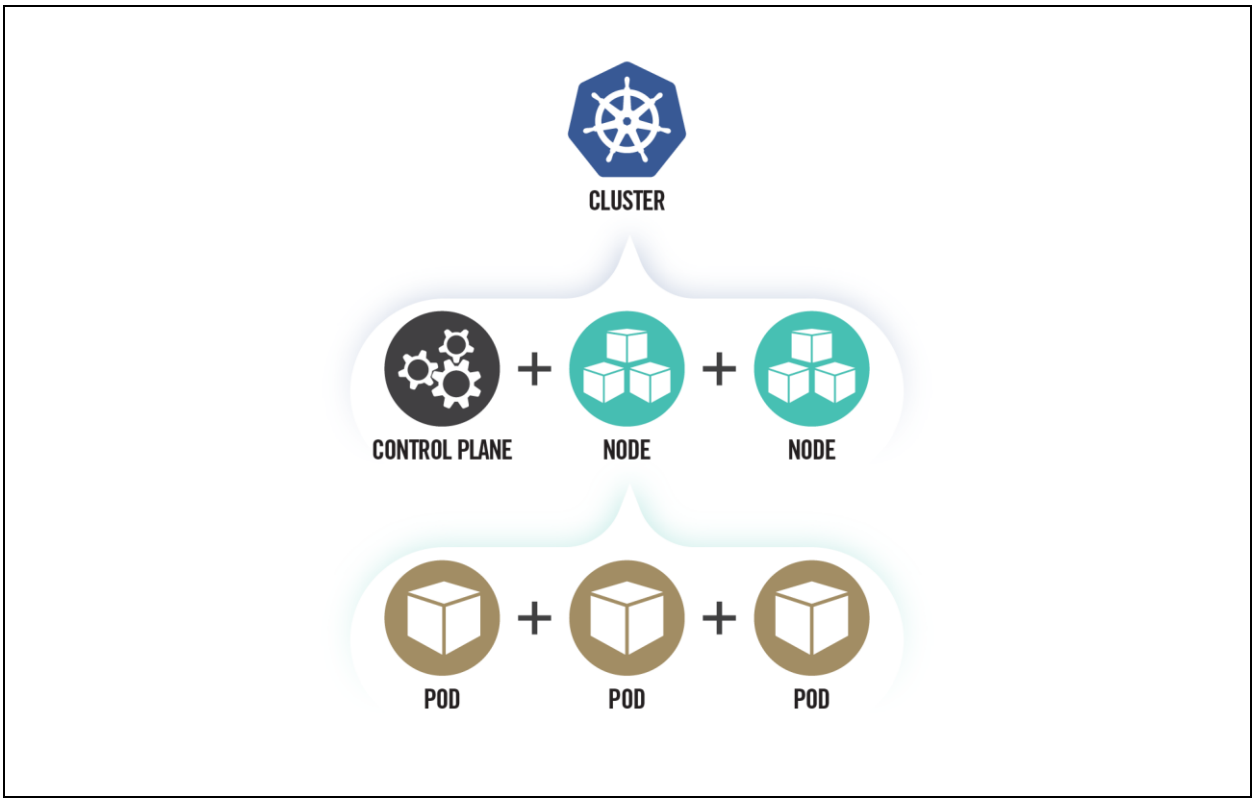

Figure 1: High-level view of Kubernetes cluster components

This guide focuses on security challenges and suggests hardening strategies for administrators of National Security Systems and critical infrastructure. Although this guide is tailored to National Security Systems and critical infrastructure organizations, NSA and CISA also encourage administrators of federal and state, local, tribal, and territorial (SLTT) government networks to implement the recommendations in this guide. Kubernetes clusters can be complex to secure and are often abused in compromises

that exploit their misconfigurations. This guide offers specific security configurations that can help build more secure Kubernetes clusters.

Recommendations

A summary of the key recommendations from each section are:

- Kubernetes Pod security
  - Use containers built to run applications as non-root users.
  - Where possible, run containers with immutable file systems.
  - Scan container images for possible vulnerabilities or misconfigurations.
  - Use a technical control to enforce a minimum level of security including:
    - Preventing privileged containers.
    - Denying container features frequently exploited to breakout, such as hostPID, hostIPC, hostNetwork, allowedHostPath.
    - Rejecting containers that execute as the root user or allow elevation to root.
    - Hardening applications against exploitation using security services such as SELinux®, AppArmor®, and secure computing mode (seccomp).
- Network separation and hardening
  - Lock down access to control plane nodes using a firewall and role-based access control (RBAC). Use separate networks for the control plane components and nodes.
  - Further limit access to the Kubernetes etcd server.
  - Configure control plane components to use authenticated, encrypted communications using Transport Layer Security (TLS) certificates.
  - Encrypt etcd at rest and use a separate TLS certificate for communication.
  - Set up network policies to isolate resources. Pods and services in different namespaces can still communicate with each other unless additional separation is enforced.
  - Create an explicit deny network policy.
  - Place all credentials and sensitive information encrypted in Kubernetes Secrets rather than in configuration files. Encrypt Secrets using a strong encryption method. Secrets are not encrypted by default.
- Authentication and authorization
  - Disable anonymous login (enabled by default).

- Use strong user authentication.
- Create RBAC policies with unique roles for users, administrators, developers, service accounts, and infrastructure team.

- Audit Logging and Threat Detection
  - Enable audit logging (disabled by default).
  - Persist logs to ensure availability in the case of node, Pod, or container-level failure.
  - Configure logging throughout the environment (e.g., cluster application program interface (API) audit event logs, cluster metric logs, application logs, Pod seccomp logs, repository audit logs, etc.).
  - Aggregate logs external to the cluster.
  - Implement a log monitoring and alerting system tailored to the organization's cluster.
- Upgrading and application security practices
  - Promptly apply security patches and updates.
  - Perform periodic vulnerability scans and penetration tests.
  - Uninstall and delete unused components from the environment.

Architectural overview

Kubernetes uses a cluster architecture. A Kubernetes cluster comprises many control planes and one or more physical or virtual machines called “worker nodes.” The worker nodes host Pods, which contain one or more containers.

A container is a runtime environment containing a software package and all its dependencies. Container images are standalone collections of the executable code and content that are used to populate a container environment as illustrated in the following figure:

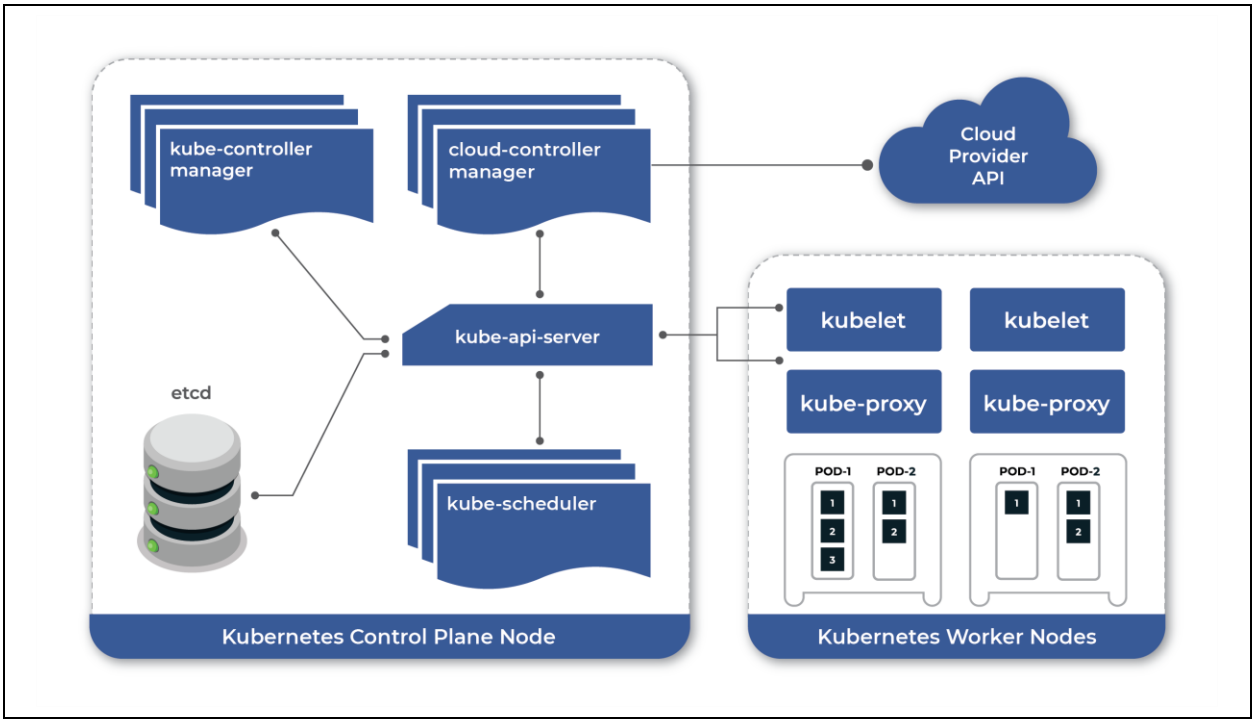

Figure 2: Kubernetes architecture

The control plane makes decisions about the cluster. This includes scheduling containers to run, detecting/responding to failures, and starting new Pods when the number of replicas listed in a Deployment file is unsatisfied. The following logical components are all part of the control plane:

- **Controller manager** – Monitors the Kubernetes cluster to detect and maintain several aspects of the Kubernetes environment including joining Pods to services, maintaining the correct number of Pods in a set, and responding to the loss of nodes.

- **Cloud controller manager** – An optional component used for cloud-based deployments. The cloud controller interfaces with the cloud service provider (CSP) to manage load balancers and virtual networking for the cluster.
- **Kubernetes application programming interface (API) server** – The interface through which administrators direct Kubernetes. As such, the API server is typically exposed outside of the control plane. It is designed to scale and may exist on multiple control plane nodes.
- **Etcd®** – The persistent backing store where all information regarding the state of the cluster is kept. Etcd is not intended to be manipulated directly but should be managed through the API server.
- **Scheduler** – Tracks the status of worker nodes and determines where to run Pods. Kube-scheduler is intended to be accessible only from within the control plane.

Kubernetes worker nodes are physical or virtual machines dedicated to running containerized applications for the cluster. In addition to running a container engine, worker nodes host the following two services that allow orchestration from the control plane:

- **Kubelet** – Runs on each worker node to orchestrate and verify Pod execution.
- **Kube-proxy** – A network proxy that uses the host's packet filtering capability to ensure correct packet routing in the Kubernetes cluster.

Clusters are commonly hosted using a CSP Kubernetes service or an on-premises Kubernetes service; CSPs often provide additional features. They administer most aspects of managed Kubernetes services; however, organizations may need to handle some Kubernetes service aspects, such as authentication and authorization, because default CSP configurations are typically not secure. When designing a Kubernetes environment, organizations should understand their responsibilities in securely maintaining the cluster.

▲*Return to Contents*

Threat model

Kubernetes can be a valuable target for data or compute power theft. While data theft is traditionally the primary motivation, cyber actors seeking computational power (often for cryptocurrency mining) are also drawn to Kubernetes to harness the underlying infrastructure. In addition to resource theft, cyber actors may target Kubernetes to cause a denial of service. The following threats represent some of the most likely sources of compromise for a Kubernetes cluster:

- **Supply Chain** – Attack vectors to the supply chain are diverse and challenging to mitigate. The risk is that an adversary may subvert any element that makes up a system. This includes product components, services, or personnel that help supply the end product. Additional supply chain risks can include third-party software and vendors used to create and manage the Kubernetes cluster. Supply chain compromises can affect Kubernetes at multiple levels including:
  - Container/application level – The security of applications and their third-party dependencies running in Kubernetes rely on the trustworthiness of the developers and the defense of the development infrastructure. A malicious container or application from a third party could provide cyber actors with a foothold in the cluster.
  - Container runtime – Each node has a container runtime installed to load container images from the repository. It monitors local system resources, isolates system resources for each container, and manages container lifecycle. A vulnerability in the container runtime could lead to insufficient separation between containers.
  - Infrastructure – The underlying systems hosting Kubernetes have their own software and hardware dependencies. Any compromise of systems used as worker nodes or as part of the control plane could provide cyber actors with a foothold in the cluster.

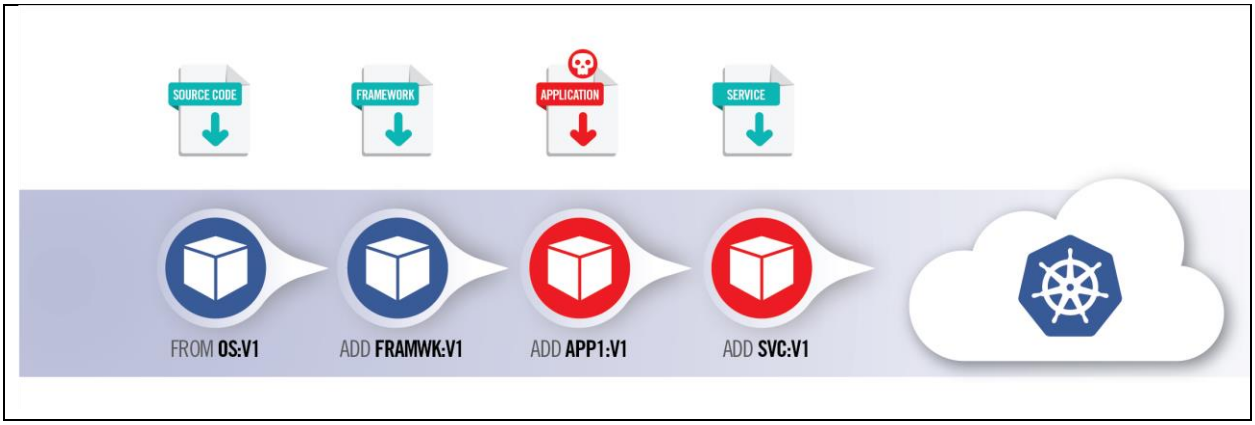

Figure 3: Example of container supply chain dependencies introducing malicious code into a cluster

- **Malicious Threat Actor** – Malicious actors often exploit vulnerabilities or stolen credentials from social engineering to gain access from a remote location. Kubernetes architecture exposes several APIs that cyber actors could potentially leverage for remote exploitation including:
    - Control plane – The Kubernetes control plane has many components that communicate to track and manage the cluster. Cyber actors frequently take advantage of exposed control plane components lacking appropriate access controls.
    - Worker nodes – In addition to running a container engine, worker nodes host the kubelet and kube-proxy service, which are potentially exploitable by cyber actors. Additionally, worker nodes exist outside of the locked-down control plane and may be more accessible to cyber actors.
    - Containerized applications – Applications running inside the cluster are common targets. They are frequently accessible outside of the cluster, making them reachable by remote cyber actors. An actor can then pivot from an already compromised Pod or escalate privileges within the cluster using an exposed application's internally accessible resources.
- **Insider Threat** – Threat actors can exploit vulnerabilities or use privileges given to the individual while working within the organization. Individuals from within the organization have special knowledge and privileges that can be used against Kubernetes clusters.
    - Administrator – Kubernetes administrators have control over running containers, including executing arbitrary commands inside containerized environments. Kubernetes-enforced RBAC authorization can reduce the risk by restricting access to sensitive capabilities. However, because

Kubernetes lacks two-person integrity controls, at least one administrative account must be capable of gaining control of the cluster. Administrators often have physical access to the systems or hypervisors, which could also be used to compromise the Kubernetes environment.

- User – Containerized application users may know and have credentials to access containerized services in the Kubernetes cluster. This level of access could provide sufficient means to exploit either the application itself or other cluster components.
- Cloud service or infrastructure provider – Access to physical systems or hypervisors managing Kubernetes nodes could be used to compromise a Kubernetes environment. CSPs often have layers of technical and administrative controls to protect systems from privileged administrators.

▲*Return to Contents*

Kubernetes Pod security

Pods are the smallest deployable Kubernetes unit and consist of one or more containers. Pods are often a cyber actor's initial execution environment upon exploiting a container. For this reason, Pods should be hardened to make exploitation more difficult and to limit the impact of a successful compromise. The following figure illustrates the components of a Pod and possible attack surface.

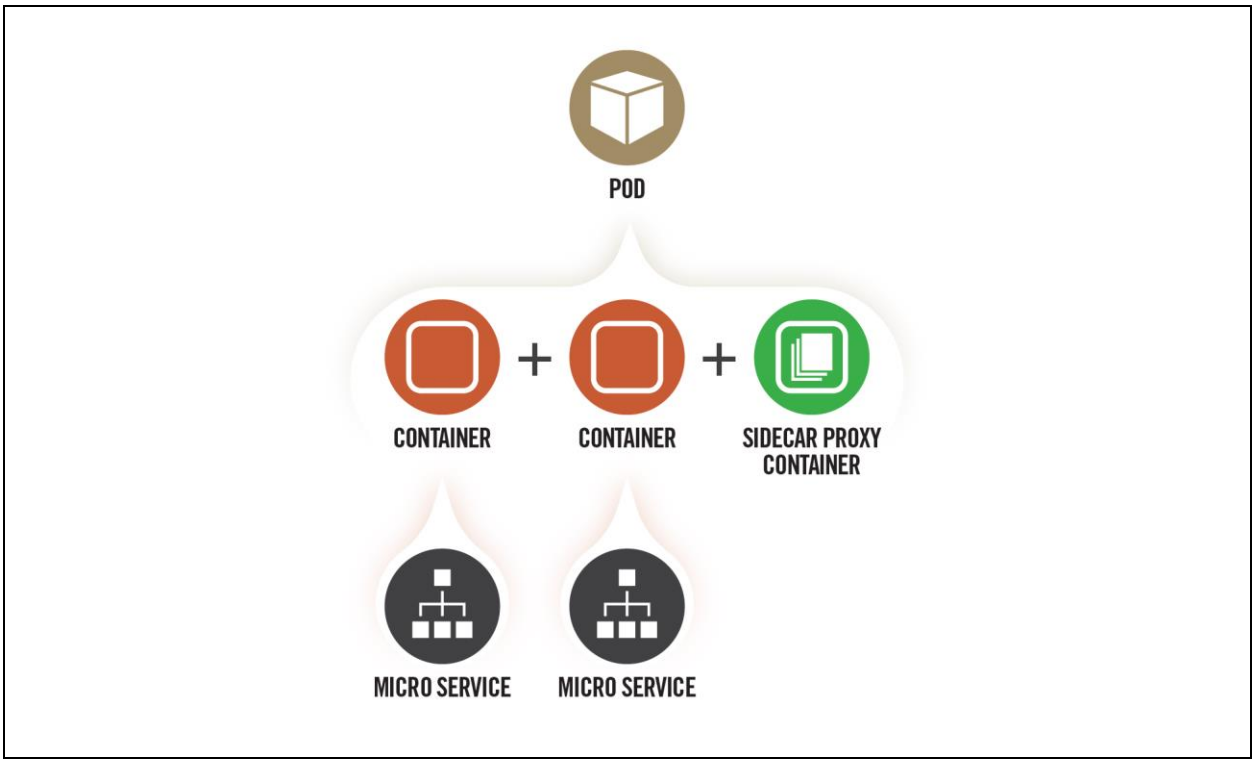

Figure 4: Pod components with sidecar proxy as logging container

"Non-root" containers and "rootless" container engines

By default, many container services run as the privileged root user, and applications execute inside the container as root despite not requiring privileged execution. Preventing root execution by using non-root containers or a rootless container engine limits the impact of a container compromise. Both methods affect the runtime environment significantly, so applications should be thoroughly tested to ensure compatibility.

Non-root containers – Container engines allow containers to run applications as a non-root user with non-root group membership. Typically, this non-default setting is configured when the container image is built. For an example Dockerfile that runs an application as a non-root user, refer to **Appendix A: Example Dockerfile for non-root application.** Alternatively, Kubernetes can load containers into a Pod with `SecurityContext:runAsUser` specifying a non-zero user. While the `runAsUser` directive effectively forces non-root execution at deployment, NSA and CISA encourage developers to build container applications to execute as a non-root user. Having non-root execution integrated at build time provides better assurance that applications will function correctly without root privileges.

Rootless container engines – Some container engines can run in an unprivileged context rather than using a daemon running as root. In this scenario, execution would appear to use the root user from the containerized application's perspective, but execution is remapped to the engine's user context on the host. While rootless container engines add an effective layer of security, many are currently released as experimental and should not be used in a production environment. Administrators should be aware of this emerging technology and adopt rootless container engines when vendors release a stable version compatible with Kubernetes.

Immutable container file systems

By default, containers are permitted mostly unrestricted execution within their own context. A cyber actor who has gained execution in a container can create files, download scripts, and modify the application within the container. Kubernetes can lock down a container's file system, thereby preventing many post-exploitation activities. However, these limitations also affect legitimate container applications and can potentially result in crashes or anomalous behavior.

To prevent damaging legitimate applications, Kubernetes administrators can mount secondary read/write file systems for specific directories where applications require write access. For an example immutable container with a writable directory, refer to **Appendix B: Example deployment template for read-only filesystem**.

Building secure container images

Container images are usually created by either building a container from scratch or by building on top of an existing image pulled from a repository. Repository controls within the developer environment can be used to restrict developers to using only trusted repositories. Specific controls vary depending on the environment but may include both platform-level restrictions, such as admission controls, and network-level restrictions. Kubernetes admission controllers, third-party tools, and some CSP-native solutions can restrict entry so that only digitally signed images can execute in the cluster.

In addition to using trusted repositories to build containers, image scanning is key to ensuring deployed containers are secure. Throughout the container build workflow, images should be scanned to identify outdated libraries, known vulnerabilities, or misconfigurations, such as insecure ports or permissions. Scanning should also provide the flexibility to disregard false positives for vulnerability detection where knowledgeable

cybersecurity professionals have deemed alerts to be inaccurate. As illustrated in the following figure, one approach to implementing image scanning is to use an admission controller. An admission controller is a Kubernetes-native feature that can intercept and process requests to the Kubernetes API prior to persistence of the object, but after the request is authenticated and authorized. A custom or proprietary webhook can be implemented to scan any image before it is deployed in the cluster. This admission controller can block deployments if the image does not comply with the organization's security policies defined in the webhook configuration [4].

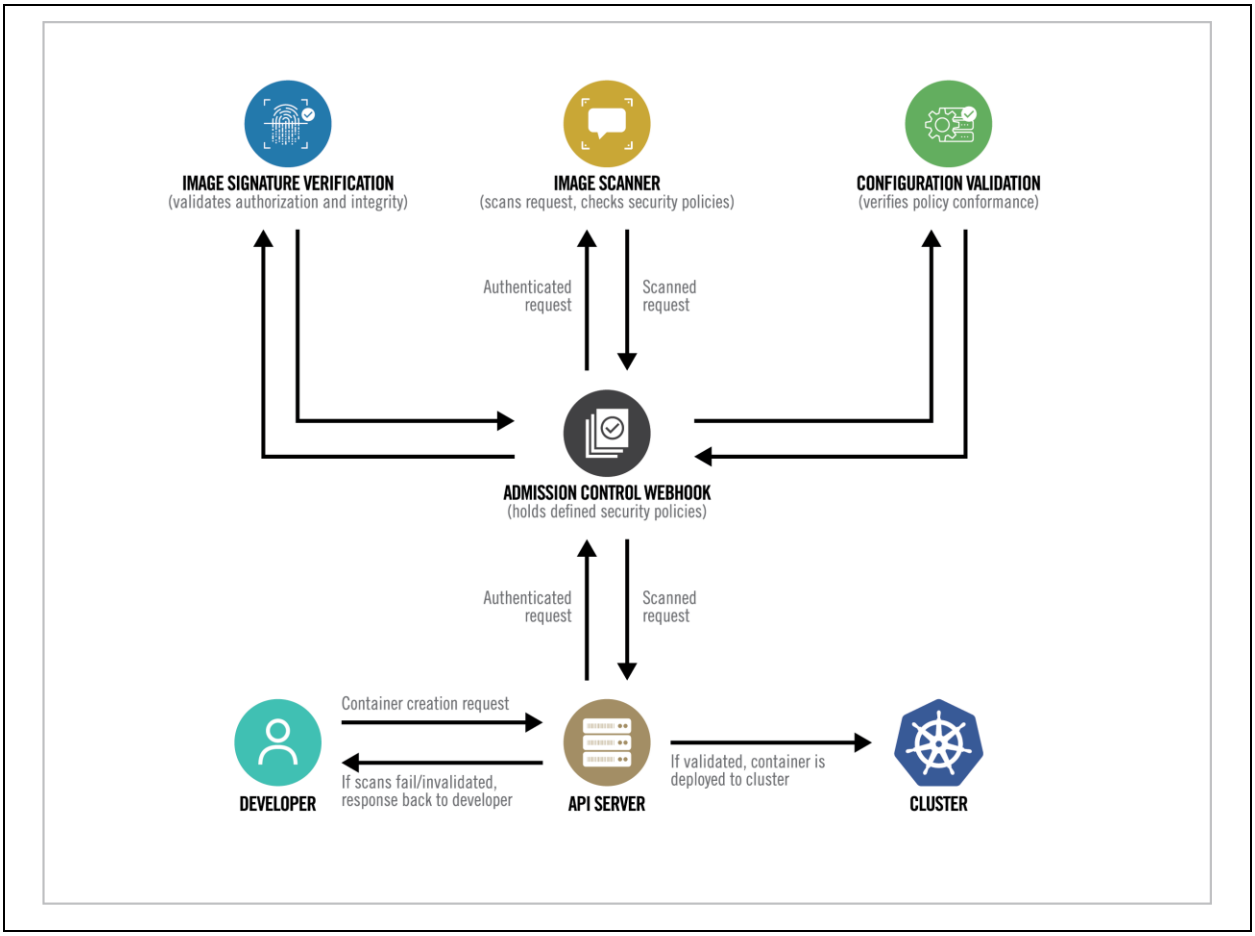

Figure 5: A hardened container build workflow

Pod security enforcement

Enforcing security requirements on Pods can be accomplished natively in Kubernetes through two mechanisms:

1. A beta[1] release feature called Pod Security Admission – Production Kubernetes administrators should adopt Pod Security Admission, as the feature is enabled by default in Kubernetes version 1.23. Pod Security Admission is based around categorizing pods as privileged, baseline, and restricted and provides a more straightforward implementation than PSPs. More information about Pod Security Admission is available in the online documentation[2].

2. A deprecated feature called Pod Security Policies (PSPs) – Administrators using PSPs while transitioning to Pod Security Admission can use information in **Appendix C: Pod Security Policies** to enhance their PSPs.

In addition to native Kubernetes solutions, third-party solutions often implemented as Kubernetes admission controllers can provide additional fine-grained policy control. While these solutions are beyond the scope of this document, administrators may explore the products available for their environment to determine the best solution for their needs.

Protecting Pod service account tokens

By default, Kubernetes automatically provisions a service account when creating a Pod and mounts the account's secret token within the Pod at runtime. Many containerized applications do not require direct access to the service account as Kubernetes orchestration occurs transparently in the background. If an application is compromised, account tokens in Pods can be gleaned by cyber actors and used to further compromise the cluster. When an application does not need to access the service account directly, Kubernetes administrators should ensure that Pod specifications disable the secret token being mounted. This can be accomplished using the "`automountServiceAccountToken: false`" directive in the Pod's YAML specification.

In some cases, containerized applications use provisioned service account tokens to authenticate to external services, such as cloud platforms. In these cases, it can be

[1] Beta releases of software have generally passed some level of quality assurance and contain most planned functionality

[2] https://kubernetes.io/docs/concepts/security/pod-security-admission/

infeasible to disable the account token. Instead, cluster administrators should ensure that RBAC is implemented to restrict Pod privileges within the cluster. For more information on RBAC, refer to the section on **authentication and authorization**.

Hardening container environments

Some platforms and container engines provide additional options or tools to harden containerized environments. For example:

- **Hypervisor-backed containerization** – Hypervisors rely on hardware to enforce the virtualization boundary rather than the operating system. Hypervisor isolation is more secure than traditional container isolation. Container engines running on the Windows® operating system can be configured to use the built-in Windows hypervisor, Hyper-V®, to enhance security. Additionally, some security-focused container engines natively deploy each container within a lightweight hypervisor for defense-in-depth. Hypervisor-backed containers mitigate container breakouts.
- **Kernel-based solutions** – The seccomp tool, which is disabled by default, can be used to limit a container's system call abilities, thereby lowering the kernel's attack surface. Seccomp can be enforced through a previously described Pod policy. For more information on Seccomp, refer to **Audit Logging and Threat Detection.**
- **Application sandboxes –** Some container engine solutions offer the option to add a layer of isolation between the containerized application and the host kernel. This isolation boundary forces the application to operate within a virtual sandbox thereby protecting the host operating system from malicious or destructive operations.

▲*Return to Contents*

Network separation and hardening

Cluster networking is a central concept of Kubernetes. Communication among containers, Pods, services, and external services must be taken into consideration. By default, Kubernetes resources are not isolated and do not prevent lateral movement or escalation if a cluster is compromised. Resource separation and encryption can be an effective way to limit a cyber actor's movement and escalation within a cluster.

Key points

- Use network policies and firewalls to separate and isolate resources.
- Secure the control plane.
- Encrypt traffic and sensitive data (such as Secrets) at rest.

Namespaces

Kubernetes namespaces are one way to partition cluster resources among multiple individuals, teams, or applications within the same cluster. *By default, namespaces are not automatically isolated*. However, namespaces do assign a label to a scope, which can be used to specify authorization rules via RBAC and networking policies. In addition to policies that limit access to resources by namespace, resource policies can limit storage and compute resources to provide better control over Pods at the namespace level.

There are three namespaces by default, and they cannot be deleted:

- kube-system (for Kubernetes components)
- kube-public (for public resources)
- default (for user resources)

User Pods should not be placed in kube-system or kube-public, as these are reserved for cluster services. A YAML file, shown in **Appendix E: Example namespace**, can be used to create new namespaces. Pods and services in different namespaces can still communicate with each other unless additional separation is enforced.

Network policies

Every Pod gets its own cluster-private IP address and can be treated similarly to virtual machines (VMs) or physical hosts with regard to port allocation, naming, service discovery, and load balancing. Kubernetes can shift Pods to other nodes and recreate Pods in a Deployment that have died. When that happens, the Pod IP addresses can change, which means applications should not depend on the Pod IP being static.

A Kubernetes Service is used to solve the issue of changing IP addresses. A Service is an abstract way to assign a unique IP address to a logical set of Pods selected using a label in the Pod configuration. The address is tied to the lifespan of the Service and will not change while the Service is alive. The communication to a Service is automatically load-balanced among the Pods that are members of the Service.

Services can be exposed externally using NodePorts or LoadBalancers, and internally. To expose a Service externally, configure the Service to use TLS certificates to encrypt traffic. Once TLS is configured, Kubernetes supports two ways to expose the Service to the Internet: NodePorts and LoadBalancers.

Adding `type: NodePort` to the Service specification file will assign a random port to be exposed to the Internet using the cluster's public IP address. The NodePort can also be assigned manually if desired. Changing the type to LoadBalancer can be used in conjunction with an external load balancer. Ingress and egress traffic can be controlled with network policies. Although Services cannot be selected by name in a network policy, the Pods can be selected using the label that is used in the configuration to select the Pods for the Service.

Network policies control traffic flow between Pods, namespaces, and external IP addresses. By default, no network policies are applied to Pods or namespaces, resulting in unrestricted ingress and egress traffic within the Pod network. Pods become isolated through a network policy that applies to the Pod or the Pod's namespace. Once a Pod is selected in a network policy, it rejects any connections that are not specifically allowed by any applicable policy object.

Network Policies Checklist

- ✓ Use a CNI plugin that supports NetworkPolicy API
- ✓ Create policies that select Pods using podSelector and/or the namespaceSelector
- ✓ Use a default policy to deny all ingress and egress traffic. Ensures unselected Pods are isolated to all namespaces except kube-system
- ✓ Use LimitRange and ResourceQuota policies to limit resources on a namespace or Pod level

To create network policies, a container network interface (CNI) plugin that supports the NetworkPolicy API is required. Pods are selected using the `podSelector` and/or the `namespaceSelector` options. For an example network policy, refer to **Appendix F: Example network policy**.

Network policy formatting may differ depending on the CNI plugin used for the cluster. Administrators should use a default policy selecting all Pods to deny all ingress and egress traffic and ensure any unselected Pods are isolated. Additional policies could then relax these restrictions for permissible connections.

External IP addresses can be used in ingress and egress policies using `ipBlock`, but different CNI plugins, cloud providers, or service implementations may affect the order of NetworkPolicy processing and the rewriting of addresses within the cluster.

Network policies can also be used in conjunction with firewalls and other external tools to create network segmentation. Splitting the network into separate sub-networks or security zones helps isolate public-facing applications from sensitive internal resources. One of the major benefits to network segmentation is limiting the attack surface and opportunity for lateral movement. In Kubernetes, network segmentation can be used to separate applications or types of resources to limit the attack surface.

Resource policies

LimitRanges, ResourceQuotas, and Process ID Limits restrict resource usage for namespaces, nodes, or Pods. These policies are important to reserve compute and storage space for a resource and avoid resource exhaustion.

A LimitRange policy constrains individual resources per Pod or container within a particular namespace, e.g., by enforcing maximum compute and storage resources. Only one LimitRange constraint can be created per namespace. For an example YAML file, refer to **Appendix G: Example LimitRange**.

Unlike LimitRange policies that apply to each Pod or container individually, ResourceQuotas are restrictions placed on the aggregate resource usage for an entire namespace, such as limits placed on total CPU and memory usage. For an example ResourceQuota policy, refer to **Appendix H: Example ResourceQuota**. If a user tries to create a Pod that violates a LimitRange or ResourceQuota policy, the Pod creation fails.

Process IDs (PIDs) are a fundamental resource on nodes and can be exhausted without violating other resource limits. PID exhaustion prevents host daemons (such as `kubelet` and `kube-proxy`) from running. Administrators can use node PID limits to reserve a specified number of PIDs for system use and Kubernetes system daemons. Pod PID limits are used to limit the number of processes running on each Pod. Eviction policies can be used to terminate a Pod that is misbehaving and consuming abnormal resources. However, eviction policies are calculated and enforced periodically and do not enforce the limit.

Control plane hardening

Steps to secure the control plane

1. Set up TLS encryption
2. Set up strong authentication methods
3. Disable access to internet and unnecessary, or untrusted networks
4. Use RBAC policies to restrict access
5. Secure the etcd datastore with authentication and RBAC policies
6. Protect kubeconfig files from unauthorized modifications

The control plane is the core of Kubernetes and allows users to view containers, schedule new Pods, read Secrets, and execute commands in the cluster. Because of these sensitive capabilities, the control plane should be highly protected. In addition to secure configurations such as TLS encryption, RBAC, and a strong authentication method, network separation can help prevent unauthorized users from accessing the control plane. The Kubernetes API server runs on port 6443, which should be protected by a firewall to accept only expected traffic. The Kubernetes API server should not be exposed to the Internet or an untrusted network. Network policies can be applied to the kube-system namespace to limit internet access to the kube-system. If a default deny policy is implemented to all namespaces, the kube-system namespace must still be able to communicate with other control plane segments and worker nodes.

The following table lists the control plane ports and services:

Table I: Control plane ports

Protocol	Direction	Port Range	Purpose
TCP	Inbound	6443	Kubernetes API server
TCP	Inbound	2379-2380	etcd server client API
TCP	Inbound	10250	kubelet API
TCP	Inbound	10259	kube-scheduler
TCP	Inbound	10257	kube-controller-manager

Etcd

The etcd backend database stores state information and cluster Secrets. It is a critical control plane component, and gaining write access to etcd could give a cyber actor root access to the entire cluster. The etcd server should be configured to trust only certificates assigned to the API server. Etcd can be run on a separate control plane node, allowing a firewall to limit access to only the API servers. This limits the attack surface when the API server is protected with the cluster's authentication method and RBAC policies to restrict users. Administrators should set up TLS certificates to enforce Hypertext Transfer Protocol Secure (HTTPS) communication between the etcd server and API servers. Using a separate certificate authority (CA) for etcd may also be beneficial, as it trusts all certificates issued by the root CA by default.

The etcd backend database is a critical control plane component and the most important piece to secure within the control plane.

Kubeconfig Files

The `kubeconfig` files contain sensitive information about clusters, users, namespaces, and authentication mechanisms. `Kubectl` uses the configuration files stored in the `$HOME/.kube` directory on the worker node and control plane local machines. Cyber actors can exploit access to this configuration directory to gain access to and modify configurations or credentials to further compromise the cluster. The configuration files should be protected from unintended changes, and unauthenticated non-root users should be blocked from accessing the files.

Worker node segmentation

A worker node can be a virtual or physical machine, depending on the cluster's implementation. Because nodes run the microservices and host the web applications for the cluster, they are often the target of exploits. If a node becomes compromised, an administrator should proactively limit the attack surface by separating the worker nodes from other network segments that do not need to communicate with the worker nodes or Kubernetes services.

Depending on the network, a firewall can be used to separate internal network segments from the external-facing worker nodes or the entire Kubernetes service. Examples of services that may need to be separated from the possible attack surface of the worker nodes are confidential databases or internal services that would not need to be internet accessible.

The following table lists the worker node ports and services:

Table II: Worker node ports

Protocol	Direction	Port Range	Purpose
TCP	Inbound	10250	kubelet API
TCP	Inbound	30000-32767	NodePort Services

Encryption

Administrators should configure all traffic in the Kubernetes cluster—including between components, nodes, and the control plane—to use TLS 1.2 or 1.3 encryption. Encryption can be set up during installation or afterward using TLS bootstrapping, detailed in the Kubernetes documentation, to create and distribute certificates to nodes. For all methods, certificates must be distributed among nodes to communicate securely.

Secrets

Kubernetes Secrets maintain sensitive information, such as passwords, OAuth tokens, and Secure Shell (SSH) keys. Storing sensitive information in Secrets provides greater access control than storing passwords or tokens in YAML files, container images, or environment variables. By default, Kubernetes stores Secrets as unencrypted base64-encoded strings that can be retrieved by anyone with API access. Access can be restricted by applying RBAC policies to the *secrets* resource.

Secrets can be encrypted by configuring data-at-rest encryption on the API server or by using an external key management service (KMS), which may be available through a cloud provider. To enable Secret data-at-rest encryption using the API server, administrators should change the `kube-apiserver` manifest file to execute using the `--encryption-provider-config` argument.

By default, Secrets are stored as unencrypted base64-encoded strings and can be retrieved by anyone with API access.

For an example `encryption-provider-config` file, refer to **Appendix I: Example encryption**. Using a KMS provider prevents the raw encryption key from being stored on the local disk. To encrypt Secrets with a KMS provider, the `encryption-provider-config` file should specify the KMS provider. For an example, refer to **Appendix J: Example KMS configuration**.

After applying the `encryption-provider-config` file, administrators should run the following command to read and encrypt all Secrets:

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Protecting sensitive cloud infrastructure

Kubernetes is often deployed on VMs in a cloud environment. As such, administrators should carefully consider the attack surface of the VMs on which the Kubernetes worker nodes are running. In many cases, Pods running on these VMs have access to sensitive cloud metadata services on a non-routable address. These metadata services provide cyber actors with information about the cloud infrastructure and possibly even short-lived credentials for cloud resources.

Cyber actors abuse these metadata services for privilege escalation [5]. Kubernetes administrators should prevent Pods from accessing cloud metadata services by using network policies or through the cloud configuration policy. Because these services vary based on the cloud provider, administrators should follow vendor guidance to harden these access vectors.

▲*Return to Contents*

Authentication and authorization

Authentication and authorization are the primary mechanisms to restrict access to cluster resources. Cyber actors can scan for well-known Kubernetes ports and access the cluster's database or make API calls without being authenticated if the cluster is misconfigured. Several user authentication mechanisms are supported but not enabled by default.

Authentication

Kubernetes clusters have two types of users:

- Service accounts
- Normal user accounts

Service accounts handle API requests on behalf of Pods. Authentication is typically managed automatically by Kubernetes through the ServiceAccount Admission Controller using bearer tokens. When the admission controller is active, it checks whether Pods have an attached service account. If the Pod definition does not specify a service account, the admission controller attaches the default service account for the namespace. The admission controller will not attach the default service account if the Pod definition prohibits the addition of the service token by setting `automountServiceAccountToken` or `automountServiceAccounttoken` to `false`. Service accounts can also be individually created to grant specific permissions. When Kubernetes creates the service account, it creates a service account Secret and automatically modifies the Pod to use the Secret. The service account token Secret contains credentials for accessing the API. If left unsecured or unencrypted, service account tokens could be used from outside of the cluster by attackers. Because of this risk, access to Pod Secrets should be restricted to those with a need to view them, using Kubernetes RBAC.

For normal users and admin accounts, there is not an automatic authentication method. Administrators must implement an authentication method or delegate authentication to a

third-party service. Kubernetes assumes that a cluster-independent service manages user authentication. The Kubernetes documentation lists several ways to implement user authentication including X509 client certificates, bootstrap tokens, and OpenID tokens. At least one user authentication method should be implemented. When multiple authentication methods are implemented, the first module to successfully authenticate the request short-circuits the evaluation. Administrators should not use weak methods, such as static password files, as weak methods could allow cyber actors to authenticate as legitimate users.

Kubernetes assumes that a cluster-independent service manages user authentication.

Anonymous requests are requests that are not rejected by other configured authentication methods and are not tied to any individual user or Pod. In a server setup for token authentication with anonymous requests enabled, a request without a token present would be performed as an anonymous request. In Kubernetes 1.6 and newer, anonymous requests are enabled by default. When RBAC is enabled, anonymous requests require explicit authorization of the `system:anonymous` user or `system:unauthenticated` group. Anonymous requests should be disabled by passing the `--anonymous-auth=false` option to the API server. Leaving anonymous requests enabled could allow a cyber actor to access cluster resources without authentication.

Role-based access control

RBAC, enabled by default, is one method to control access to cluster resources based on the roles of individuals within an organization. RBAC can be used to restrict access for user accounts and service accounts. To check if RBAC is enabled in a cluster using kubectl, execute `kubectl api-version`. The API version for `.rbac.authorization.k8s.io/v1` should be listed if RBAC is enabled. Cloud Kubernetes services may have a different way of checking whether RBAC is enabled for the cluster. If RBAC is not enabled, start the API server with the `--authorization-mode` flag in the following command:

```
kube-apiserver --authorization-mode=RBAC
```

Leaving authorization-mode flags, such as `AlwaysAllow`, in place allows all authorization requests, effectively disabling all authorization and limiting the ability to enforce least privilege for access.

Two types of permissions can be set:

- Roles – Set permissions for particular namespaces
- ClusterRoles – Set permissions across all cluster resources regardless of namespace

Both Roles and ClusterRoles can only be used to add permissions. There are no deny rules. If a cluster is configured to use RBAC and anonymous access is disabled, the Kubernetes API server will deny permissions not explicitly allowed. For an example RBAC Role, refer to **Appendix K: Example pod-reader RBAC Role**.

A Role or ClusterRole defines a permission but does not tie the permission to a user. As illustrated in the following figure, RoleBindings and ClusterRoleBindings are used to tie a Role or ClusterRole to a user, group, or service account. RoleBindings grant permissions in Roles or ClusterRoles to users, groups, or service accounts in a defined namespace. ClusterRoles are created independent of namespaces and can be used multiple times in conjunction with a RoleBinding to limit the namespace scope.

This is useful when users, groups, or service accounts require similar permissions in multiple namespaces. One ClusterRole can be used several times with different RoleBindings to limit scope to different individual users, groups, or service accounts. ClusterRoleBindings grant users, groups, or service accounts ClusterRoles across all cluster resources. For an example of RBAC RoleBinding and ClusterRoleBinding, refer to **Appendix L: Example RBAC RoleBinding and ClusterRoleBinding**.

Figure 6: Possible Role, ClusterRole, RoleBinding, and ClusterRoleBinding combinations to assign access

To create or update Roles and ClusterRoles, a user must have the permissions contained in the new role at the same scope or possess explicit permission to perform the escalate verb on the Roles or ClusterRoles resources in the `rbac.authorization.k8s.io` API group. After a binding is created, the Role or ClusterRole is immutable. The binding must be deleted to change a role.

Privileges assigned to users, groups, and service accounts should follow the principle of *least privilege*, allowing only required permissions to complete tasks. User groups can make creating Roles easier to manage. Unique permissions are required for different

groups, such as users, administrators, developers, and the infrastructure team. Each group needs access to different resources and should not have permissions to edit or view other groups' resources. Users, user groups, and service accounts should be limited to interact and view specific namespaces where required resources reside. Access to the Kubernetes API is limited by creating an RBAC Role or ClusterRole with the appropriate API request verb and desired resource on which the action can be applied. Tools exist that can help audit RBAC policies by printing users, groups, and service accounts with their associated assigned Roles and ClusterRoles.

▲*Return to Contents*

Audit Logging and Threat Detection

Audit logs capture attributed activity in the cluster. An effective logging solution and log reviewing are necessary, not only for ensuring that services are operating and configured as intended, but also for ensuring the security of the system. Systematic security audit requirements mandate consistent and thorough checks of security settings to help identify compromises. Kubernetes is capable of capturing audit logs for tracking attributed cluster actions, and monitoring basic CPU and memory usage information; however, it does not natively provide full featured monitoring or alerting services.

Key points

- Establish Pod baselines at creation to enable anomalous activity identification.
- Perform logging at all levels of the environment.
- Integrate existing network security tools for aggregate scans, monitoring, alerts, and analysis.
- Set up fault-tolerant policies to prevent log loss in case of a failure.

Logging

System administrators running applications within Kubernetes should establish an effective logging and monitoring system for their environment. Logging Kubernetes events alone is not enough to provide a full picture of the actions occurring on the system. Logging should be performed at all levels of the environment, including on the host, application, container, container engine, image registry, api-server, and the cloud, as applicable. Once captured, these logs should all be aggregated to a single service to provide security auditors, network defenders, and incident responders a full view of the actions taken throughout the environment.

Within the Kubernetes environment, some events that administrators should monitor/log include the following:

- API request history
- Performance metrics
- Deployments
- Resource consumption

- Operating system calls
- Protocols, permission changes
- Network traffic
- Pod scaling
- Volume mount actions
- Image and container modification
- Privilege changes
- Scheduled job (cronjob) creations and modifications

When administrators create or update a Pod, they should capture detailed logs of the network communications, response times, requests, resource consumption, and any other relevant metrics to establish a baseline. RBAC policy configurations should also be reviewed periodically and whenever personnel changes occur in the organization's system administrators. Doing so ensures access controls remain in compliance with the RBAC policy-hardening guidance outlined in the role-based access control section of this guide.

Routine system security audits should include comparisons of current logs to the baseline measurements of normal activities to identify significant changes in any of the logged metrics and events. System administrators should investigate significant changes to determine the root cause. For example, a significant increase in resource consumption could be indicative of a change in application usage or the installation of malicious processes such as a cryptominer.

Audits of internal and external traffic logs should be conducted to ensure all intended security constraints on connections have been configured properly and are working as intended. Administrators can also use these audits as systems evolve to evaluate where external access may be restricted.

Streaming logs to an external logging service will help to ensure availability to security professionals outside of the cluster, enabling them to identify abnormalities in as close to real time as possible. If using this method, logs should be encrypted during transit with TLS 1.2 or 1.3 to ensure cyber actors cannot access the logs in transit and gain valuable information about the environment.

Another precaution to take when utilizing an external log server is to configure the log forwarder within Kubernetes with append-only access to the external storage. This protects the externally stored logs from being deleted or overwritten from within the cluster.

Kubernetes native audit logging configuration

The `kube-apiserver` resides on the Kubernetes control plane and acts as the front end, handling internal and external requests for a cluster. Each request, whether generated by a user, an application, or the control plane, produces an audit event at each stage in its execution. When an audit event registers, the `kube-apiserver` checks for an audit policy file and applicable rule. If such a rule exists, the server logs the event at the level defined by the first matched rule. Kubernetes' built-in audit logging capabilities perform no logging by default.

Kubernetes audit logging capabilities are disabled by default

Cluster administrators must write an audit policy YAML file to establish the rules and specify the desired audit level at which to log each type of audit event. This audit policy file is then passed to the `kube-apiserver` with the appropriate flags. For a rule to be considered valid, it must specify one of the four audit levels:

- `None`
- `Metadata`
- `Request`
- `RequestResponse`

Logging all events at the `RequestResponse` level will give administrators the maximum amount of information available for incident responders should a breach occur. However, this may result in capturing base64-encoded Secrets in the logs. NSA and CISA recommend reducing the logging level of requests involving Secrets to the `Metadata` level to avoid capturing Secrets in logs.

Additionally, logging all other events at the highest level will produce a large quantity of logs, especially in a production cluster. If an organization's constraints require it, the audit policy can be tailored to the environment, reducing the logging level of non-critical, routine events. The specific rules necessary for such an audit policy will vary by deployment. It is vital to log all security-critical events, paying close attention to the

organization's specific cluster configuration and threat model to indicate where to focus logging. The goal of refining an audit policy should be to remove redundancy, while still providing a clear picture and attribution of the events occurring in the cluster.

For some examples of general critical and non-critical audit event types and stages, as well as an example of an audit policy file that logs Secrets at the `metadata` level, and all other events at the `RequestResponse` level, refer to **Appendix M: Audit Policy**

For an example where the `kube-apiserver` configuration file is located and an example of the flags by which the audit policy file can be passed to the `kube-apiserver`, refer to **Appendix N: Example Flags to Enable Audit Logging**. For directions on how to mount the volumes and configure the host path, if necessary, refer to **Appendix N: Example Flags to Enable Audit Logging**.

The `kube-apiserver` includes configurable logging and webhook backends for audit logging. The logging backend writes the audit events specified to a log file, and the webhook backend can be configured to send the file to an external HTTP API. The `--audit-log-path` and `--audit-log-maxage` flags, set in the example in **Appendix N: Example Flags to Enable Audit Logging,** are two examples of the flags that can be used to configure the logging backend, which writes audit events to a file. The `log-path` flag is the minimum configuration required to enable logging and the only configuration necessary for the logging backend. The default format for these log files is Java Script Object Notation (JSON), though this can also be changed if necessary. Additional configuration options for the logging backend can be found in the Kubernetes documentation. Kubernetes also provides a webhook backend option that administrators can manually configure via a YAML file submitted to the `kube-apiserver` to push logs to an external backend. An exhaustive list of the configuration options, which can be set in the `kube-apiserver` for the webhook backend, can be found in the Kubernetes documentation. Further details on how the webhook backend works and how to set it up can also be found in the Kubernetes documentation. There are also many external tools available to perform log aggregation, some of which are discussed briefly in the following sections.

Worker node and container logging

There are many ways for logging capabilities to be configured within a Kubernetes architecture. In the built-in method of log management, the `kubelet` on each node is

responsible for managing logs. It stores and rotates log files locally based on its policies for individual file length, storage duration, and storage capacity. These logs are controlled by the `kubelet` and can be accessed from the command line. The following command prints the logs of a container within a Pod:

```
kubectl logs [-f] [-p] POD [-c CONTAINER]
```

The `-f` flag may be used if the logs are to be streamed, the `-p` flag may be used if logs from previous instances of a container exist and are desired, and the `-c` flag can be used to specify a container if there are more than one in the Pod. If an error occurs that causes a container, Pod, or node to die, the native logging solution in Kubernetes does not provide a method to preserve logs stored in the failed object. NSA and CISA recommend configuring a remote logging solution to preserve logs should a node fail. Options for remote logging include:

Table III: Remote logging configuration

Remote logging option	Reason to use	Configuration implementation
Run a logging agent on every node to push logs to a backend	Gives the node the ability to expose logs or push logs to a backend, preserving them outside of the node in the case of a failure.	Configure an independent container in a Pod to run as a logging agent, giving it access to the node's application log files and configuring it to forward logs to the organization's SIEM.
Use a sidecar container in each Pod to push logs to an output stream	Used to push logs to separate output streams. This can be a useful option when application containers write multiple log files of different formats.	Configure a sidecar container for each log type and use them to redirect these log files to their individual output streams, where the `kubelet` can manage them. The node-level logging agent can then forward these logs onto the SIEM or other backend.
Use a logging agent sidecar in each Pod to push logs to a backend	When more flexibility is needed than the node-level logging agent can provide.	Configure for each Pod to push logs directly to the backend. This is a common method for attaching third-party logging agents and backends.
Push logs directly to a backend from within an application	Allows logs to go directly to the aggregation platform. Can be useful if the organization has separate teams responsible for managing application security vs the Kubernetes platform security.	Kubernetes does not have built-in mechanisms for exposing or pushing logs to a backend directly. Organizations will need to either build this functionality into their application or attach a reputable third-party tool to enable this.

To ensure continuity of logging agents across worker nodes, it is common to run them as a DaemonSet. Configuring a DaemonSet for this method ensures that there is a copy of the logging agent on every node at all times and that any changes made to the logging agent are consistent across the cluster.

Large organizations with multiple teams running their own Kubernetes clusters should establish logging requirements and a standard architecture to ensure that all teams have an effective solution in place.

Seccomp: audit mode

In addition to the node and container logging previously described, it can be highly beneficial to log system calls. One method for auditing container system calls in Kubernetes is to use the seccomp tool. This tool is disabled by default but can be used to limit a container's system call abilities, thereby lowering the kernel's attack surface. Seccomp can also log what calls are being made by using an audit profile.

A custom seccomp profile defines which system calls are allowed, denied, or logged, and default actions for calls not specified. To enable a custom seccomp profile within a Pod, Kubernetes admins can write their seccomp profile JSON file to the `/var/lib/kubelet/seccomp/` directory and add a `seccompProfile` to the Pod's `securityContext`.

A custom `seccompProfile` should also include two fields: `Type: Localhost` and `localhostProfile: myseccomppolicy.json`. Logging all system calls can help administrators know what system calls are needed for standard operations allowing them to restrict the seccomp profile further without losing system functionality. It can also help administrators establish a baseline for a Pod's standard operation patterns, allowing them to identify any major deviances from this pattern that could be indicative of malicious activity.

Syslog

Kubernetes, by default, writes `kubelet` logs and container runtime logs to journald if the service is available. If organizations wish to utilize syslog utilities—or to collect logs from across the cluster and forward them to a syslog server or other log storage and aggregation platform—they can configure that capability manually. Syslog protocol defines a log message-formatting standard. Syslog messages include a header and a

message written in plaintext. Syslog daemons such as syslog-ng® and rsyslog are capable of collecting and aggregating logs from across a system in a unified format. Many Linux operating systems by default use rsyslog or journald—an event logging daemon that optimizes log storage and output logs in syslog format via journalctl. The syslog utility logs events, on nodes running certain Linux distributions by default at the host level.

Containers running these Linux distributions will, by default, collect logs using syslog as well. Syslog utilities store logs in the local file system on each applicable node or container unless a log aggregation platform is configured to collect them. The syslog daemon or another such tool should be configured to aggregate both these and all other logs being collected across the cluster and forward them to an external backend for storage and monitoring.

SIEM platforms

Security information and event management (SIEM) software collects logs from across an organization's network. It brings together firewall logs, application logs, and more, parsing them out to provide a centralized platform from which analysts can monitor system security. SIEM tools have variations in their capabilities. Generally, these platforms provide log collection, aggregation, threat detection, and alerting capabilities. Some include machine-learning capabilities, which can better predict system behavior and help to reduce false alerts. Organizations using these platforms in their environment should integrate them with Kubernetes to better monitor and secure clusters. Open-source platforms for managing logs from a Kubernetes environment exist as alternatives to SIEM platforms.

Containerized environments have many interdependencies between nodes, Pods, containers, and services. In these environments, Pods and containers are constantly being deleted and redeployed on different nodes. This type of environment presents an extra challenge for traditional SIEMs, which typically use IP addresses to correlate logs. Even next-generation SIEM platforms may not always be suited to the complex Kubernetes environment. However, as Kubernetes has emerged as the most widely used container orchestration platform, many of the organizations developing SIEM tools have developed variations of their products specifically designed to work with the Kubernetes environment, providing full monitoring solutions for these containerized environments. Administrators should be aware of their platform's capabilities and

ensure that their logging sufficiently captures the environment to support future incident responses.

Service meshes

Service meshes are platforms that streamline micro-service communications within an application by allowing for the logic of these communications to be coded into the service mesh rather than within each micro-service. Coding this communication logic into individual micro-services is difficult to scale, difficult to debug as failures occur, and difficult to secure. Using a service mesh can simplify this coding for developers. Log collection at this level can also give cluster administrators insight into the standard service-to-service communication flow throughout the cluster. The mesh can:

- Redirect traffic when a service is down,
- Gather performance metrics for optimizing communications,
- Allow management of service-to-service communication encryption,
- Collect logs for service-to-service communication,
- Collect logs from each service,
- Help developers diagnose problems and failures of micro-services or communication mechanisms, and
- Help with migrating services to hybrid or multi-cloud environments.

While service meshes are not necessary, they are an option that is highly suitable to the Kubernetes environment. Their logging capabilities can also be useful in mapping the service-to-service communications, helping administrators see what their standard cluster operation looks like and identify anomalies easier. Managed Kubernetes services often include their own service mesh; however, several other platforms are also available and, if desired, are highly customizable.

Another major benefit of modern service meshes is encryption of service-to-service communications. Many service meshes manage keys and generate and rotate certificates, allowing for secure TLS authentication between services, without requiring developers to set this up for each individual service and manage it themselves. Some service meshes even perform this service-to-service encryption by default. If administrators deploy a service mesh within a Kubernetes cluster, it is important to keep up with updates and security alerts for the service mesh as illustrated in the following figure:

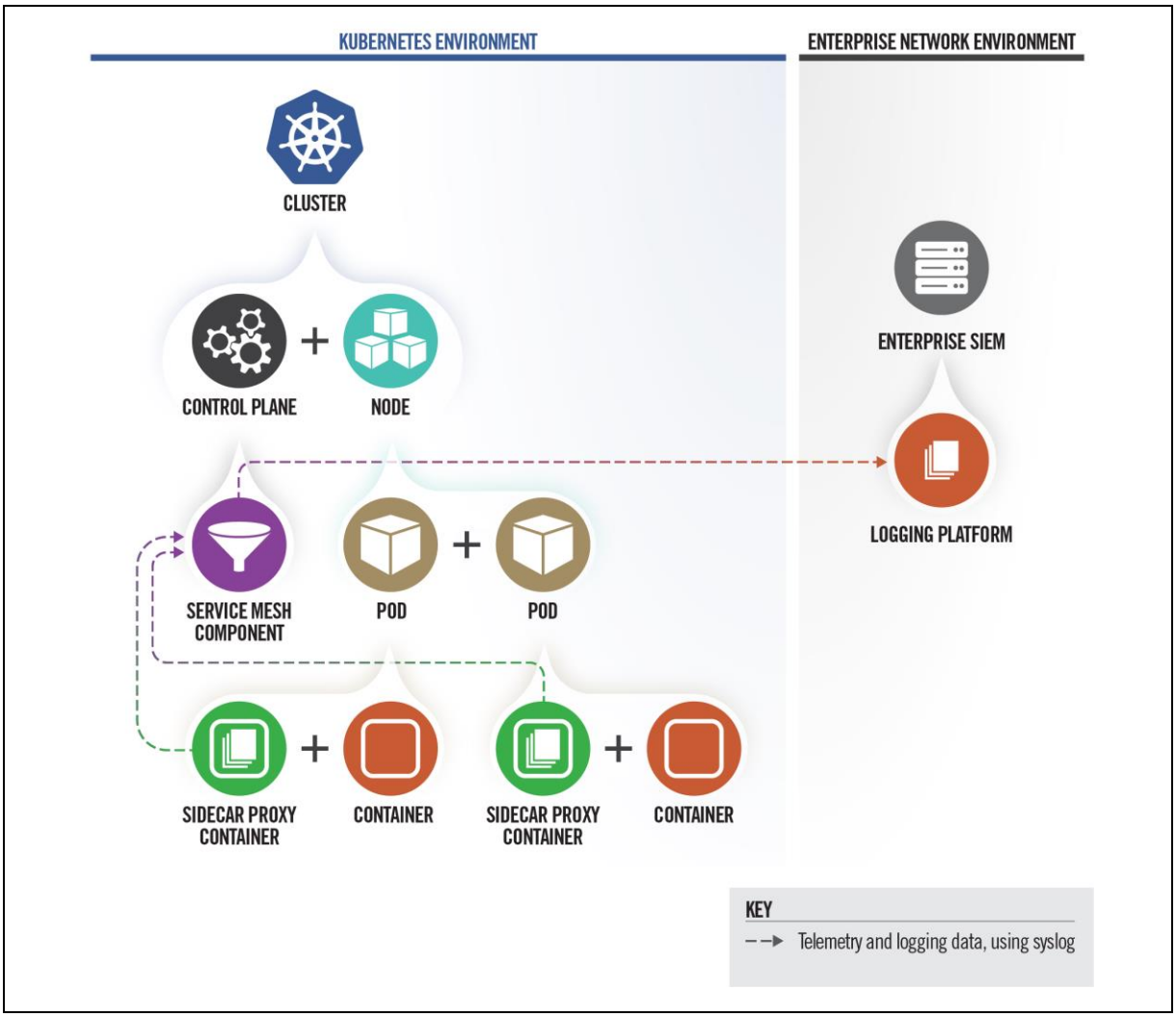

Figure 7: Cluster leveraging service mesh to integrate logging with network security

Fault tolerance

Organizations should put fault tolerance policies in place. These policies could differ depending on the specific Kubernetes use case. One such policy is to allow new logs to overwrite the oldest log files, if absolutely necessary, in the event of storage capacity being exceeded. Another such policy that can be used if logs are being sent to an external service is to establish a place for logs to be stored locally if a communication loss or an external service failure occurs. Once communication to the external service is restored, a policy should be in place for the locally stored logs to be pushed up to the external server.

Threat Detection

An effective logging solution comprises two critical components: collecting all necessary data and then actively monitoring the collected data for red flags in as close to real time as possible. The best logging solution in the world is useless if the data is never examined. Much of the process of log examination can be automated; however, when either writing log parsing policies or manually examining logs, it is vital to know what to look for. When attackers try to exploit the cluster, they will leave traces of their actions in the logs.

The following table contains some of the ways attackers may try to exploit the cluster and how that may present in the logs. (Caveat: This table lists *some* known suspicious indicators. Administrators should also be aware of, and alert to, specific concerns and emerging threats in their environments. The most effective alerts are tailored to identify abnormal activity for a specific cluster.)

Table IV: Detection recommendations

Attacker Action	Log Detection
Attackers may try to deploy a Pod or container to run their own malicious software or to use as a staging ground/pivot point for their attack. Attackers may try to masquerade their deployment as a legitimate image by copying names and naming conventions. They may also try to start a container with root privileges to escalate privileges.	Watch for atypical Pod and container deployments. Use image IDs and layer hashes for comparisons of suspected image deployments against the valid images. Watch for Pods or application containers being started with root permissions
Attackers may try to import a malicious image into the victim organization's registry, either to give themselves access to their image for deployment, or to trick legitimate parties into deploying their malicious image instead of the legitimate ones.	This may be detectable in the container engine or image repository logs. Network defenders should investigate any variations from the standard deployment process. Depending on the specific case this may also be detectible through changes in containers' behavior after being redeployed using the new image version.
If an attacker manages to exploit an application to the point of gaining command execution capabilities on the container, then depending on the configuration of the Pod, they may be able to make API requests from within the Pod, potentially	Unusual API requests (from the Kubernetes audit logs) or unusual system calls (from seccomp logs) originating from inside a Pod. This could also show as pod creation requests registering a Pod IP address as its source IP.

Attacker Action	Log Detection
escalating privileges, moving laterally within the cluster, or breaking out onto the host.	
Attackers who have gained initial access to a Kubernetes cluster will likely start attempting to penetrate further into the cluster, which will require interacting with the kube-apiserver.	While they work to determine what initial permissions they have, they may end up making several failed requests to the API server. Repeated failed API requests and request patterns that are atypical for a given account would be red flags.
Attackers may attempt to compromise a cluster in order to use the victim's resources to run their own cryptominer (i.e., a cryptojacking attack).	If an attacker were to successfully start a cryptojacking attack it would likely show in the logs as a sudden spike in resource consumption.
Attackers may attempt to use anonymous accounts to avoid attribution of their activities in the cluster.	Watch for any anonymous actions in the cluster.
Attackers may try to add a volume mount to a container they have compromised or are creating, to gain access to the host	Volume mount actions should be closely monitored for abnormalities.
Attackers with the ability to create scheduled jobs (aka Kubernetes CronJobs) may attempt to use this to get Kubernetes to automatically and repetitively run malware on the cluster [8].	Scheduled job creations and modifications should be closely monitored.

The enormous quantity of logs generated in an environment such as this makes it infeasible for administrators to review all of the logs manually and even more important for administrators to know what indicators to look for. This knowledge can be used to configure automated responses and refine the criteria for triggering alerts.

Alerting

Kubernetes does not natively support alerting; however, several monitoring tools with alerting capabilities are compatible with Kubernetes. If Kubernetes administrators choose to configure an alerting tool to work within a Kubernetes environment, administrators can use several metrics to monitor and configure alerts.

Examples of actionable events that could trigger alerts include but are not limited to:

- Low disk space on any of the machines in the environment,

- Available storage space on a logging volume running low,
- External logging service going offline,
- A Pod or application running with root permissions,
- Requests being made by an account for resources they do not have permission for,
- Anonymous requests being submitted to the API server,
- Pod or Worker Node IP addresses being listed as the source ID of a Pod creation request,
- Unusual system calls or failed API calls,
- User/admin behavior that is abnormal (i.e. at unusual times or from an unusual location), and
- Significant deviations from the standard operation metrics baseline.

In their 2021 Kubernetes blog post, contributors to the Kubernetes project made the following three additions to this list [7]:

- Changes to a Pod's securityContext,
- Updates to admission controller configs, and
- Accessing certain sensitive files/URLs.

Where possible, systems should be configured to take steps to mitigate compromises while administrators respond to alerts. In the case of a Pod IP being listed as the source ID of a Pod creation request, automatically evicting the Pod is one mitigation that could be implemented to keep the application available but temporarily stop any compromises of the cluster. Doing so would allow a clean version of the Pod to be rescheduled onto one of the nodes. Investigators could examine the logs to determine if a breach occurred and, if so, how the malicious actors executed the compromise so that a patch can be deployed. Automating such responses can help improve security professionals' response time to critical events.

Tools

Kubernetes does not natively include extensive auditing capabilities. However, the system is built to be extensible, allowing users the freedom to develop their own custom solution or to choose an existing add-on that suits their needs. Kubernetes cluster administrators commonly connect additional backend services to their cluster to perform additional functions for users, such as extended search parameters, data mapping

features, and alerting functionality. Organizations that already use SIEM platforms can integrate Kubernetes with these existing capabilities. Open-source monitoring tools—such as the Cloud Native Computing Foundation's Prometheus®, Grafana Labs' Grafana®, and Elasticsearch's Elastic Stack (ELK)®—are also available. The tools can conduct event monitoring, run threat analytics, manage alerting, and collect resource isolation parameters, historical usage, and network statistics on running containers. Scanning tools can be used when auditing the access control and permission configurations to identify risky permission configurations in RBAC.

NSA and CISA encourage organizations utilizing Intrusion Detection Systems (IDSs) on their existing environment to consider integrating that service into their Kubernetes environment as well. This integration would allow an organization to monitor for—and potentially kill containers showing signs of—unusual behavior so the containers can be restarted from the initial clean image. Many CSPs also provide container monitoring services for those wanting more managed and scalable solutions.

▲*Return to Contents*

Upgrading and application security practices

Following the hardening guidance outlined in this document is a step toward ensuring the security of applications running on Kubernetes orchestrated containers. However, security is an ongoing process, and it is vital to keep up with patches, updates, and upgrades. The specific software components vary depending on the individual configuration, but each piece of the overall system must be kept as secure as possible. This includes updating Kubernetes, hypervisors, virtualization software, plugins, operating systems on which the environment is running, applications running on the servers, all elements of the organization's continuous integration/continuous delivery (CI/CD) pipeline and any other software hosted in the environment. Companies who need to maintain 24/7 uptime for their services can consider using high-availability clusters, so that services can be off-loaded from physical machines one at a time, allowing for firmware, kernel, and operating system updates to be deployed in a timely manner while still maintaining service availability.

The Center for Internet Security (CIS) publishes benchmarks for securing software. Administrators should adhere to the CIS benchmarks for Kubernetes and any other relevant system components. Administrators should periodically check to ensure their system's security is compliant with the current cybersecurity best practices. Periodic vulnerability scans and penetration tests should be performed on the various system components to proactively look for insecure configurations and zero-day vulnerabilities. Any discoveries should be promptly remediated before potential cyber actors can discover and exploit them.

As administrators deploy updates, they should also keep up with uninstalling any old, unused components from the environment and deployment pipeline. This practice will help reduce the attack surface and the risk of unused tools remaining on the system and falling out of date. Using a managed Kubernetes service can help to automate upgrades and patches for Kubernetes, operating systems, and networking protocols. However, administrators must still ensure that their deployments are up to date and developers properly tag new images to avoid accidental deployments of outdated images.

▲*Return to Contents*

Works cited

[1] Center for Internet Security, "CIS Benchmarks Securing Kubernetes," 2021. [Online]. Available: https://cisecurity.org/benchmark/kubernetes/.

[2] DISA, "Kubernetes STIG," 2021. [Online]. Available: https://public.cyber.mil/stigs/downloads/.

[3] The Linux Foundation, "Kubernetes Documnetation," 2021. [Online]. Available: https://kubernetes.io/docs/ . [Accessed 02 2021].

[4] The Linux Foundation, "11 Ways (Not) to Get Hacked," 18 07 2018. [Online]. Available: https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/#10-scan-images-and-run-ids. [Accessed 03 2021].

[5] MITRE, "MITRE ATT&CK," 2021. [Online]. Available: https://attack.mitre.org/techniques/T1552/005/. [Accessed 7 May 2021].

[6] CISA, "Analysis Report (AR21-013A)," 14 January 2021. [Online]. Available: https://www.cisa.gov/uscert/ncas/analysis-reports/ar21-013a. [Accessed 26 May 2021].

[7] Kubernetes, "A Closer Look at NSA/CISA Kubernetes Hardening Guidance," 5 October 2021. [Online]. Available: https://www.kubernetes.io/blog/2021/10/05/nsa-cisa-kubernetes-hardening-guidance/ 2021.

[8] MITRE ATT&CK, "Scheduled Task/Job: Container Orchestration Job," 27 7 2021. [Online]. Available: https://attack.mitre.org/techniques/T1053/007/. [Accessed 9 11 2021].

[9] The Kubernetes Authors, "Pod Security Admission," [Online]. Available: https://kubernetes.io/docs/concepts/security/pod-security-admission/.

Appendix A: Example Dockerfile for non-root application

The following example is a Dockerfile that runs an application as a non-root user with non-group membership. The lines highlighted in red below are the portion specific to using non-root.

```
FROM ubuntu:latest

#Update and install the make utility
RUN apt update && apt install -y make

#Copy the source from a folder called "code" and build the application with
the make utility
COPY . /code
RUN make /code

#Create a new user (user1) and new group (group1); then switch into that
user's context
RUN useradd user1 && groupadd group1
USER user1:group1

#Set the default entrypoint for the container
CMD /code/app
```

Appendix B: Example deployment template for read-only file system

The following example is a Kubernetes deployment template that uses a read-only root file system. The lines highlighted in red below are the portion specific to making the container's filesystem read-only. The lines highlighted in blue are the portion showing how to create a writeable volume for applications requiring this capability.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: web
  name: web
spec:
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
      name: web
    spec:
      containers:
      - command: ["sleep"]
        args: ["999"]
        image: ubuntu:latest
        name: web
        securityContext:
          readOnlyRootFilesystem: true
        volumeMounts:
          - mountPath: /writeable/location/here
            name: volName
      volumes:
      - emptyDir: {}
        name: volName
```

Appendix C: Pod Security Policies (deprecated)

A Pod Security Policy (PSP) is a cluster-wide policy that specifies security requirements/defaults for Pods to execute within the cluster. While security mechanisms are often specified within Pod/deployment configurations, PSPs establish a minimum security threshold to which all Pods must adhere. Some PSP fields provide default values used when a Pod's configuration omits a field. Other PSP fields are used to deny the creation of non-conformant Pods. PSPs are enforced through a Kubernetes admission controller, so PSPs can only enforce requirements during Pod creation. PSPs do not affect Pods already running in the cluster.

PSPs are useful technical controls to enforce security measures in the cluster. PSPs are particularly effective for clusters managed by admins with tiered roles. In these cases, top-level admins can impose defaults to enforce requirements on lower-level admins. NSA and CISA encourage organizations to adapt the Kubernetes hardened PSP template in **Appendix D: Example Pod Security Policy** to their needs. The following table describes some widely applicable PSP components.

Table V: Pod Security Policy components[3]

Field Name(s)	Usage	Recommendations
privileged	Controls whether Pods can run privileged containers.	Set to false.
hostPID, hostIPC	Controls whether containers can share host process namespaces.	Set to false.
hostNetwork	Controls whether containers can use the host network.	Set to false.
allowedHostPaths	Limits containers to specific paths of the host file system.	Use a "dummy" path name (such as "/foo" marked as read-only). Omitting this field results in no admission restrictions being placed on containers.
readOnlyRootFilesystem	Requires the use of a read only root file system.	Set to true when possible.
runAsUser, runAsGroup, supplementalGroups, fsGroup	Controls whether container applications can run with root privileges or with root group membership.	- Set runAsUser to MustRunAsNonRoot. - Set runAsGroup to non-zero (See the example in Appendix D: Example Pod Security Policy). - Set supplementalGroups to non-zero (see example in appendix D).

[3] https://kubernetes.io/docs/concepts/policy/pod-security-policy

Field Name(s)	Usage	Recommendations
		- Set fsGroup to non-zero (See the example in Appendix D: Example Pod Security Policy).
allowPrivilegeEscalation	Restricts escalation to root privileges.	Set to false. This measure is required to effectively enforce “runAsUser: MustRunAsNonRoot” settings.
seLinux	Sets the SELinux context of the container.	If the environment supports SELinux, consider adding SELinux labeling to further harden the container.
AppArmor annotations	Sets the AppArmor profile used by containers.	Where possible, harden containerized applications by employing AppArmor to constrain exploitation.
seccomp annotations	Sets the seccomp profile used to sandbox containers.	Where possible, use a seccomp auditing profile to identify required syscalls for running applications; then enable a seccomp profile to block all other syscalls.

Note: PSPs do not automatically apply to the entire cluster for the following reasons:

- First, before PSPs can be applied, the PodSecurityPolicy plugin must be enabled for the Kubernetes admission controller, part of `kube-apiserver`.
- Second, the policy must be authorized through RBAC. Administrators should verify the correct functionality of implemented PSPs from each role within their cluster’s organization.

Administrators should be cautious in environments with multiple PSPs as Pod creation adheres to the *least restrictive* authorized policy. The following command describes all Pod Security Policies for the given namespace, which can help to identify problematic overlapping policies:

```
kubectl get psp -n <namespace>
```

Appendix D: Example Pod Security Policy

The following example is a Kubernetes Pod Security Policy that enforces strong security requirements for containers running in the cluster. This example is based on official Kubernetes documentation: https://kubernetes.io/docs/concepts/policy/pod-security-policy/. Administrators are encouraged to tailor the policy to meet their organization's requirements.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames:
'docker/default,runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames:
'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:
'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:
'runtime/default'
spec:
  privileged: false # Required to prevent escalations to root.
    allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim' # Assume persistentVolumes set up by admin
are safe
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot' # Require the container to run without root
  seLinux:
    rule: 'RunAsAny' # This assumes nodes are using AppArmor rather than
SELinux
  supplementalGroups:
    rule: 'MustRunAs'
    ranges: # Forbid adding the root group.
      - min: 1
        max: 65535
  runAsGroup:
    rule: 'MustRunAs'
    ranges: # Forbid adding the root group.
      - min: 1
        max: 65535
  fsGroup:
```

```
    rule: 'MustRunAs'
    ranges: # Forbid adding the root group.
      - min: 1
        max: 65535
readOnlyRootFilesystem: true
```

Appendix E: Example namespace

The following example is for each team or group of users, a Kubernetes namespace can be created using either a `kubectl` command or YAML file. Any name with the prefix `kube-` should be avoided as it may conflict with Kubernetes system reserved namespaces.

`Kubectl` command to create a namespace:

```
kubectl create namespace <insert-namespace-name-here>
```

To create namespace using YAML file, create a new file called `my-namespace.yaml` with the contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

Apply the namespace using:

```
kubectl create -f ./my-namespace.yaml
```

To create new Pods in an existing namespace, switch to the desired namespace using:

```
kubectl config use-context <insert-namespace-here>
```

Apply new deployment using:

```
kubectl apply -f deployment.yaml
```

Alternatively, the namespace can be added to the `kubectl` command using:

```
kubectl apply -f deployment.yaml --namespace=<insert-namespace-here>
```

or specify `namespace: <insert-namespace-here>` under metadata in the YAML declaration.

Once created, resources cannot be moved between namespaces. The resource must be deleted, then created in the new namespace.

Appendix F: Example network policy

Network policies differ depending on the network plugin used. The following example is a network policy to limit access to the nginx service to Pods with the label access using the Kubernetes documentation: https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: example-access-nginx
 namespace: prod #this can any namespace or be left out if no
namespace is used
spec:
    podSelector:
        matchLabels:
            app: nginx
    ingress:
    -from:
        -podSelector:
            matchLabels:
                access: “true”
```

The new NetworkPolicy can be applied using:

```
kubectl apply -f policy.yaml
```

A default deny all ingress policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: deny-all-ingress
spec:
    podSelector: {}
    policyType:
    - Ingress
```

A default deny all egress policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: deny-all-egress
spec:
    podSelector: {}
    policyType:
    - Egress
```

Appendix G: Example LimitRange

LimitRange support is enabled by default in Kubernetes 1.10 and newer. The following YAML file specifies a LimitRange with a default request and limit, as well as a min and max request, for each container.

```
apiVersion: v1
kind: LimitRange
metadata:
   name: cpu-min-max-demo-lr
spec:
   limits
   -  default:
         cpu: 1
      defaultRequest:
         cpu: 0.5
      max:
         cpu: 2
      min:
         cpu 0.5
      type: Container
```

A LimitRange can be applied to a namespace with:

```
kubectl apply -f <example-LimitRange>.yaml --namespace=<Enter-Namespace>
```

After this example LimitRange configuration is applied, all containers created in the namespace are assigned the default CPU request and limit if not specified. All containers in the namespace must have a CPU request greater than or equal to the minimum value and less than or equal to the maximum CPU value or the container will not be instantiated.

Appendix H: Example ResourceQuota

ResourceQuota objects to limit aggregate resource usage within a namespace are created by applying a YAML file to a namespace or specifying requirements in the configuration file of Pods. The following example is based on official Kubernetes documentation: https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/

Configuration file for a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: example-cpu-mem-resourcequota
spec:
    hard:
        requests.cpu: "1"
        requests.memory: 1Gi
        limits.cpu: "2"
        limits.memory: 2Gi
```

This ResourceQuota can be applied with:

```
kubectl apply -f example-cpu-mem-resourcequota.yaml --
namespace=<insert-namespace-here>
```

This ResourceQuota places the following constraints on the chosen namespace:

- Every container must have a memory request, memory limit, CPU request, and CPU limit,
- Aggregate memory request for all containers should not exceed 1 GiB,
- Total memory limit for all containers should not exceed 2 GiB,
- Aggregate CPU request for all containers should not exceed 1 CPU, and
- Total CPU limit for all containers should not exceed 2 CPUs.

Appendix I: Example encryption

To encrypt Secret data at rest, the following encryption configuration file provides an example to specify the type of encryption desired and the encryption key. Storing the encryption key in the encryption file only slightly improves security. The Secrets will be encrypted, but the key will be accessible in the `EncryptionConfiguration` file. This example is based on official Kubernetes documentation: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
   -  resources:
      -  secrets
      providers:
      -  aescbc:
            keys:
            -  name: key1
               secret: <base 64 encoded secret>
      -  identity: {}
```

To enable encryption at rest with this encryption file, restart the API server with the `--encryption-provider-config` flag set with the location to the configuration file.

Appendix J: Example KMS configuration

To encrypt Secrets with a key management service (KMS) provider plugin, the following example encryption configuration YAML file can be used to set the properties for the provider. This example is based on official Kubernetes documentation: https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  -  resources:
       -  secrets
     providers:
       -  kms:
            name: myKMSPlugin
            endpoint: unix://tmp/socketfile.sock
            cachesize: 100
            timeout: 3s
       -  identity: {}
```

To configure the API server to use the KMS provider, set the *--encryption-provider-config* flag with the location of the configuration file and restart the API server.

To switch from a local encryption provider to KMS, add the KMS provider section of the EncryptionConfiguration file above the current encryption method, as shown below.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  -  resources:
       -  secrets
     providers:
       -  kms:
            name: myKMSPlugin
            endpoint: unix://tmp/socketfile.sock
            cachesize: 100
            timeout: 3s
       -  aescbc:
            keys:
              -  name: key1
                 secret: <base64 encoded secret>
```

Restart the API server and run the command below to re-encrypt all Secrets with the KMS provider.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Appendix K: Example pod-reader RBAC Role

To create the example *pod-reader* Role, create a YAML file with the following contents:

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

Apply the Role using:

```
kubectl apply --f role.yaml
```

To create the example *global-pod-reader* ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata: default
 # “namespace” omitted since ClusterRoles are not bound to a
namespace
 name: global-pod-reader
rules:
-  apiGroups: [“”] # “” indicates the core API group
   resources: [“pods”]
   verbs: [“get”, “watch”, “list”]
```

Apply the Role using:

```
kubectl apply --f clusterrole.yaml
```

Appendix L: Example RBAC RoleBinding and ClusterRoleBinding

To create a RoleBinding, create a YAML file with the following contents:

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read Pods in the "your-
namespace-name"
# namespace.
# You need to already have a Role names "pod-reader" in that
namespace.
kind: RoleBinding
metadata:
    name: read-pods
    namespace: your-namespace-name
subjects:
# You can specify more than one "subject"
-  kind: User
   name: jane # "name" is case sensitive
   apiGroup: rbac.authorization.k8s.io
roleRef:
   # "roleRef" specifies the binding to a Role/ClusterRole
   # kind: Role # this must be a Role or ClusterRole
   # this must match the name of the Role or ClusterRole you wish to
  bind
   # to
   name: pod-reader
   apiGroup: rbac.authorization.k8s.io
```

Apply the RoleBinding using:

```
kubectl apply --f rolebinding.yaml
```

To create a ClusterRoleBinding, create a YAML file with the following contents:

```
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binging allows anyone in the "manager" group to
read
# Pod information in any namespace.
kind: ClusterRoleBinding
metadata:
    name: global-pod-reader
subjects:
# You can specify more than one "subject"
-  kind: Group
   name: manager # Name is case sensitive
   apiGroup: rbac.authorization.k8s.io
roleRef:
   # "roleRef" specifies the binding to a Role/ClusterRole
   kind: ClusterRole # this must be a Role or ClusterRole
```

```
    name: global-pod-reader # this must match the name of the Role or
ClusterRole you wish to bind to
    apiGroup: rbac.authorization.k8s.io
```

Apply the ClusterRoleBinding using:

```
kubectl apply --f clusterrolebinding.yaml
```

Appendix M: Audit Policy

The following example is an Audit Policy that logs requests involving Kubernetes Secrets at the `Metadata` level, and all other audit events at the highest level:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
   -  level: Metadata
      resources:
         -  group:”” #this refers to the core API group
            resources: [“secrets”]
   -  level: RequestResponse

# This audit policy logs events involving secrets at the metadata level, and all other audit events at the RequestResponse level
```

If an organization has the resources available to store, parse, and examine a large number of logs, then logging all events, other than those involving Secrets, at the highest level is a good way of ensuring that, when a breach occurs, all necessary contextual information is present in the logs. If resource consumption and availability are a concern, then more logging rules can be established to lower the logging level of non-critical components and routine non-privileged actions, as long as audit logging requirements for the system are being met. As Kubernetes API events consist of multiple stages, logging rules can also specify stages of the request to omit from the log. By default, Kubernetes captures audit events at all stages of the request. The four possible stages of Kubernetes API request are:

- `RequestReceived`
- `ResponseStarted`
- `ResponseComplete`
- `Panic`

Because clusters in organizations expand to meet growing needs, it is important to ensure that the audit policy can still meet logging needs. To ensure that elements of the environment are not overlooked, the audit policy should end with a catch-all rule to log events that the previous rules did not log. Kubernetes logs audit events based on the first rule in the audit policy that applies to the given event; therefore, it is important to be aware of the order in which potentially overlapping rules are written. The rule regarding Secrets should be near the top of the policy file to ensure. This ensures that any overlapping rules do not inadvertently capture Secrets due to logging at a higher level

than the `Metadata` level. Similarly, the catch-all rule should be the last rule in the policy to ensure that all other rules are matched first.

What follows are some examples of critical event types that should be logged at the `Request` or `RequestResponse` level. In addition are examples of less critical event types and stages that can be logged at a lower level if necessary to reduce redundancy in the logs and increase the organization's ability to effectively review the logs as close to real time as possible.

Critical:

- Pod deployments and alterations
- Authentication requests
- Modifications to RBAC resources (clusterrolebindings, clusterroles, etc.)
- Scheduled job creations
- Edits to Pod Security Admissions or Pod Security Policies

Noncritical:

- `RequestRecieved` stage
- Authenticated requests to non-critical, routinely accessed resources

For an example of how to establish these rules, refer to the official Kubernetes documentation: https://kubernetes.io/docs/tasks/debug-application-cluster/audit/.

Appendix N: Example Flags to Enable Audit Logging

In the control plane, open the `kube-apiserver.yaml` file in a text editor. Editing the `kube-apiserver` configuration requires administrator privileges.

```
sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

Add the following text to the `kube-apiserver.yaml` file:

```
--audit-policy-file=/etc/kubernetes/policy/audit-policy.yaml
--audit-log-path=/var/log/audit.log
--audit-log-maxage=1825
```

The `audit-policy-file` flag should be set with the path to the audit policy, and the `audit-log-path` flag should be set with the desired secure location for the audit logs to be written to. Other additional flags exist, such as the `audit-log-maxage` flag shown here, which stipulates the maximum number of days the logs should be kept, and flags for specifying the maximum number of audit log files to retain, max log file size in megabytes, etc. The only flags necessary to enable logging are the `audit-policy-file` and `audit-log-path` flags. The other flags can be used to configure logging to match the organization's policies.

If a user's `kube-apiserver` is run as a Pod, then it is necessary to mount the volume and configure `hostPath` of the policy and log file locations for audit records to be retained. This can be done by adding the following sections to the `kube-apiserver.yaml` file as noted in the Kubernetes documentation: https://kubernetes.io/docs/tasks/debug-application-cluster/audit/

```
volumeMounts:
 -mountPath: /etc/kubernetes/audit-policy.yaml
  name: audit
  readOnly: true
 -mountPath: /var/log/audit.log
  name: audit-log
  readOnly: false

volumes:
- hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File
name: audit
 - hostPath:
    path: /var/log/audit.log
    type: FileOrCreate
name: audit-log
```