ARMO

The Makers of Kubescape

Kubernetes Security Best Practices: Definitive Guide for Security Professionals

Ben Hirschberg
CTO and Co-Founder, ARMO

Table of contents

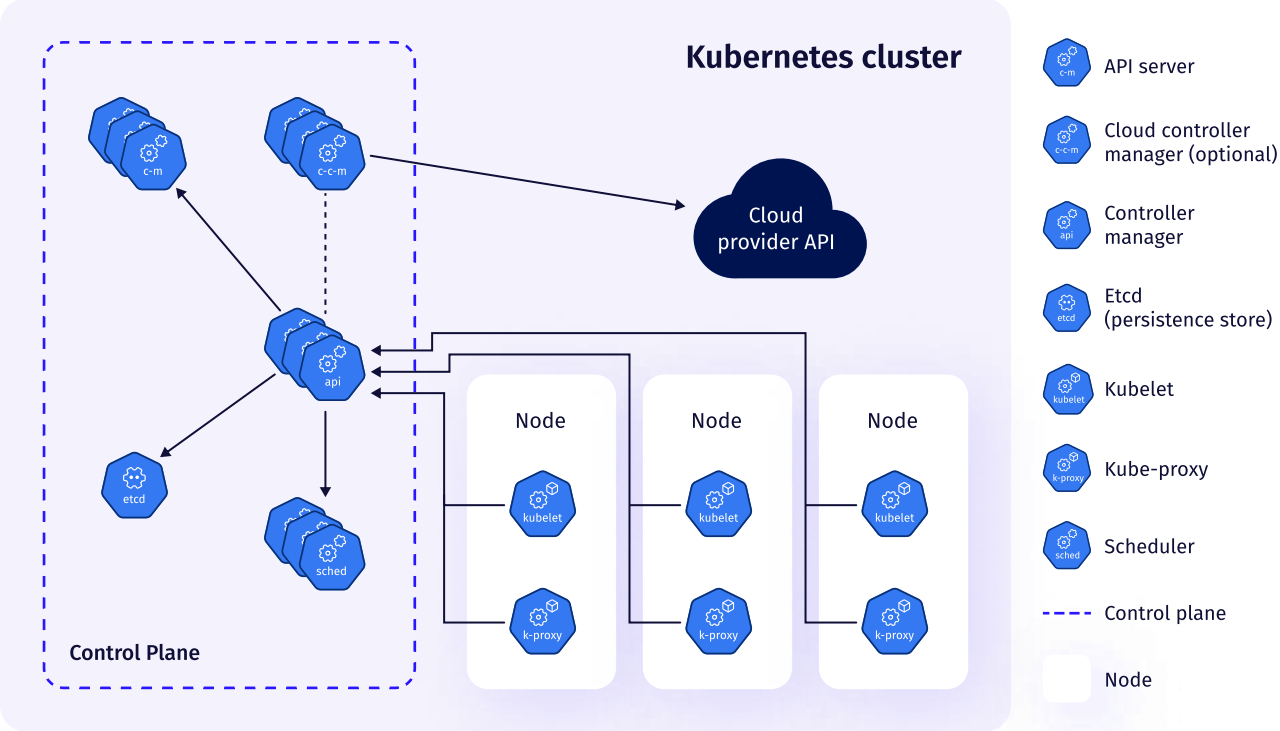
Opening Words

As containerized applications become the norm, the complexities of securing these dynamic, scalable environments demand a fresh perspective on traditional security practices. While Kubernetes streamlines deployment and management, it also introduces a new layer of attack surface, necessitating a nuanced approach to threat mitigation. We must navigate these complexities by drawing upon the collective knowledge and tools offered by the open-source Kubernetes community to build a robust Kubernetes security posture. So, buckle up, security architects and engineers, because in this blog, we'll be diving deep into the evolving landscape of Kubernetes security, exploring best practices, pointing out potential pitfalls, and charting a course towards securing the ever-expanding containerized frontier.

Introduction

Kubernetes, an open-source container orchestration engine, is well known for its ability to automate the deployment, management, and, most importantly, scaling of containerized applications.

Running an individual microservice in a single container is almost always safer than running it as processes in the same VM. To run a container, Kubernetes use the Pod concept (Point Of Deployment) which is a non-empty set of containers. When a Pod is launched in Kubernetes, it is hosted in a Kubernetes Node (the abstraction of a machine). A Kubernetes Node can behave as a “Worker” or a “Master” or both. A Worker node(s) hosts the application Pods in the cluster, the Master Node(s) hosts the Kubernetes control plane components like API server, Scheduler and Controller manager. Together, they constitute a cluster.



Source: kubernetes.io

Nodes provide CPU, memory, storage, and networking resources on which the control plane can place the pods. Kubernetes nodes must also run a variety of components supporting the control plane's management of the node and its workloads, such as the kubelet, kube-proxy, and the container runtime.

What is Kubernetes Security?

Kubernetes orchestrates containerized applications running in the cloud or on-premises. As a result, the security of a cluster is determined not only by the configuration of the cluster, but also by the security of the infrastructure on which the cluster is deployed. Applications and business processes used in and around the cluster, also affect its security. These concerns span several security disciplines, ranging from application security and access control to network security and vulnerability management. Furthermore, each discipline is important at different levels of a Kubernetes cluster stack. These levels can be thought of abstractly as the Four Cs: Cloud, Cluster, Container, and Code.

Securing a Kubernetes cluster means ensuring that each of the Four Cs is configured and used in accordance with best practices and organizational security policies. This includes things like when a workload is allowed to run, what behavior, permissions, and capabilities it has throughout its lifecycle. It also covers who is allowed to access or configure the cluster or its underlying infrastructure.

Clusters' security posture must also be constantly re-evaluated in order to keep up with the pace of healthy development practices. Automated controls must enable developers to identify and remediate security issues quickly and independently in order to meet business needs for developer autonomy and allow security to scale with the organization.

Importance of Kubernetes Security

The risk posed by a compromise in any single improperly contained cluster workload is often equal to the risk posed by complete cluster compromise. An attacker who gains control of one cluster node can potentially compromise any other cluster workload. Using credentials stored on each cluster node, it is possible to start additional malicious workloads and access secrets stored in the cluster. This can develop into lateral movement to other types of environments, databases, or other cloud or external services. Such an attacker can also gain network access to any networks reachable from any cluster node, for example to peered on-premises data centers.

In short, clusters are valuable targets. Each workload, user, and configuration should be held to high security standards to prevent the compromise of all cluster resources.

Kubernetes Security Best Practices: The 4C Model

When constructing a defense-in-depth strategy, it is necessary to incorporate numerous security barriers in various areas; cloud-native security operates similarly and suggests implementing the same approach. The security techniques of Cloud Native Systems are divided into four different layers, which is referred to as “The 4C Security Model”: Cloud, Cluster, Container, Code. Addressing all these layers ensures comprehensive security coverage from development to deployment. The best practices for Kubernetes can also be classified into these four categories of the cloud-native approach.

Cloud / Colocated

The cloud layer refers to the server infrastructure, whether the provider is a public cloud, a private cloud, an on-premises datacenter, or some combination of the three. Preparing the infrastructure for running a secure Kubernetes environment involves deploying and configuring various services. While public CSPs are primarily responsible for safeguarding such services (e.g., operating system, platform management, and network implementation), the default configurations are often not suitable for production, and customers are still responsible for many aspects of their own security, like managing credentials, correctly configuring the infrastructure, and monitoring and securing their data.

`_Prevent Unwanted Access to the API Server`

The best practice is to limit the network layer access to Kubernetes API server. Depending on cloud provider, this is usually done by limiting the API server access to a single VPC. When users need to access the API from outside the VPC, they usually create a VPN service or SSH server in that specific VPC and using it as a gateway to Kubernetes API.

Cluster

Since Kubernetes is the most widely used container orchestration tool, we will concentrate our attention on it while discussing cluster security in general. Therefore, all security recommendations in this section are limited to safeguarding the cluster itself.

_Controlling Access to the API Server

In addition to restricting API server access at the infrastructure level, there are additional considerations for controlling access to the Kubernetes API, which limit who can use it, and what they are able to use it for.

Each request to the API server is first authenticated to identify the entity making the request, then subsequently authorized to ensure the entity has permission to perform the requested action., Finally, the content of the request is subjected to admission control before it is accepted.

It is important that only TLS connections are used for connections to the API server, internal communication within the control plane, and communication between the control plane and the kubelet. To accomplish this, you can provide a TLS certificate and a TLS private key file to the API server. You can do this either through a command line option or a configuration file. This is a basic security measure and nearly all Kubernetes distributions (Kind, Minikube, Rancher, OpenShift, etc.) and managed Kubernetes services (EKS, GKE, and AKS) come with this best practice setup.

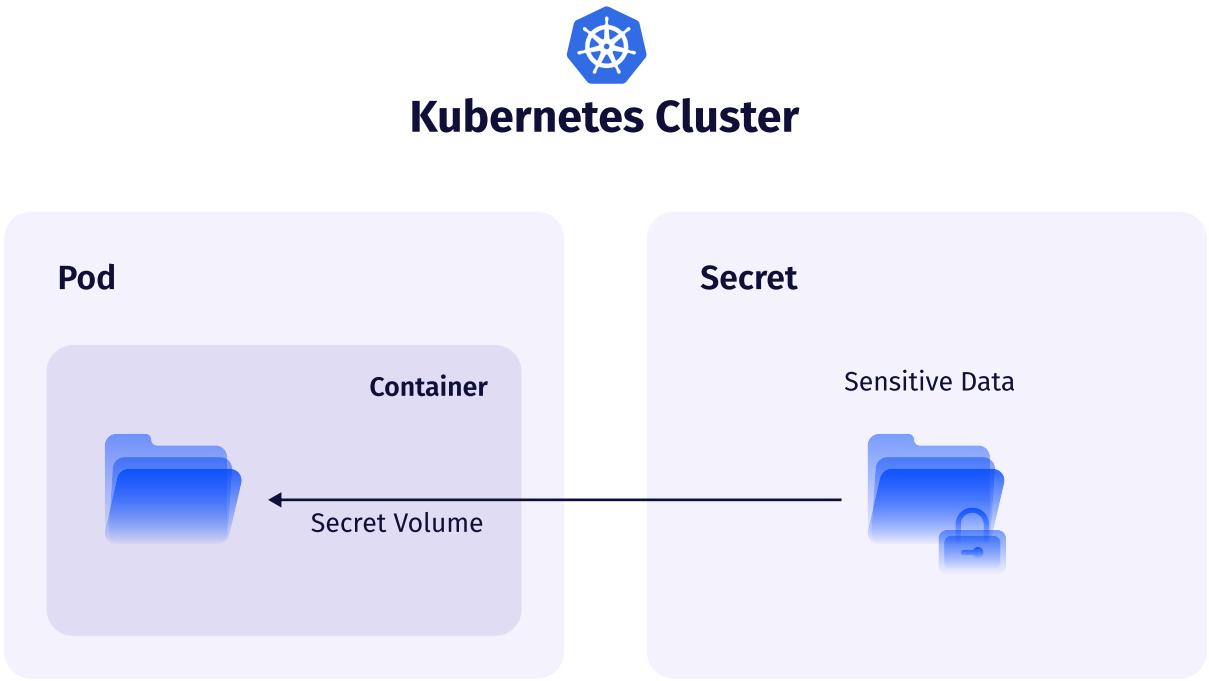
The Kubernetes API uses two HTTP ports, designated as localhost and secure port, to communicate. The localhost port does not require TLS, so requests made through this port will bypass the authentication and authorization components. Therefore, you must make sure this port is not enabled outside of the Kubernetes cluster's test configuration.

Service accounts in Kubernetes are used to provide an identity for processes that run in a Pod. They are used to authenticate and authorize applications and processes that need to interact with the Kubernetes API. This identity is critical for the security of a Kubernetes cluster, as it controls access to resources and operations. Due to the potential for security breaches, service accounts should be thoroughly and regularly audited, particularly if they are associated with privileged accounts. To ensure security and follow the principle of least privilege, each application should have its own service account. This service account should only have the necessary permissions for the specific deployment. It is best to avoid using the default service account, as it is automatically mapped to every Pod by default, going against the principle of least privilege. To prevent potential security risks, it is important to disable the automatic mounting of default service account tokens when creating new pods, especially if no individual service account is provided. This helps minimize the risk of unauthorized access and reduces the chances of creating additional vulnerabilities.

Access to signed certificates (kubeconfigs) and service account tokens should be tightly controlled. While newer Kubernetes versions use the TokenRequest API to issue short-lived, revocable tokens, it is still possible to manually create tokens which never expire. Similarly, kubeconfigs can be created without an expiration. Once a user's keypair has been signed by the cluster's root certificate authority, it cannot be easily revoked. If such a credential is stolen or leaked, the only way to ensure cluster security is to reinstall the whole cluster with a new root certificate.

_Kubernetes Secrets

Secrets in Kubernetes are objects used to hold sensitive information, such as passwords, keys, tokens, and many other types of information. Since they are different from other configuration objects like "ConfigMaps", they are handled differently on multiple levels. This limits the exploitable attack surface. Secrets are intended to decouple sensitive values from non-sensitive configuration which accompanies a workload. This approach reduces the chances of a developer unintentionally accessing or storing sensitive data. Secrets are RBAC-controlled, namespaced objects (maximum length: 1 MB), which, unlike ConfigMaps, are kept in tmpfs on the nodes in order to prevent ever writing the data to persistent storage.



Source: https://livebook.manning.com/book/gitops-and-kubernetes/chapter-7/v-6/57

The API server stores Secrets in etcd in plain text by default, so it's important to enable encryption in the API server configuration. This way, if an attacker were to access the etcd data, they wouldn't be able to read it because they would also need the key used for encryption. Kubernetes supports multiple types of encryption schemes for stored objects, including local key pairs and CSP-hosted key management systems. Although Secret values are stored base64-encoded, this is for data (de-)serialization purposes and not (!) a substitute for proper encryption.

_Protect Nodes

Nodes are responsible for the operations of your containerized apps. As such, it is essential that nodes are properly hardened to provide a secure computing environment.

There are numerous individual settings and configuration options to consider when hardening a node. Many of these are included in the CIS Benchmark and provide guidelines for protecting nodes. These are:

Enabling SELinux: SELinux adds an extra layer of security by enforcing mandatory access controls. It helps protect nodes from unauthorized access and protects against privilege escalation attacks.

Disabling unnecessary services: By disabling unnecessary services and daemons, organizations reduce the attack surface and minimize the potential for vulnerabilities.

Configuring firewall rules: Implementing proper firewall rules allows organizations to control network traffic, block unauthorized access, and protect the nodes from network-based attacks.

Enabling automatic updates: Enabling automatic updates ensures that the nodes receive the latest security patches and bug fixes, protecting against known vulnerabilities.

Configuring the system time, hostname, DNS servers, NTP servers, and proxy server: Properly configuring these settings ensures secure communication, accurate time synchronization, and reliable network access for the nodes.

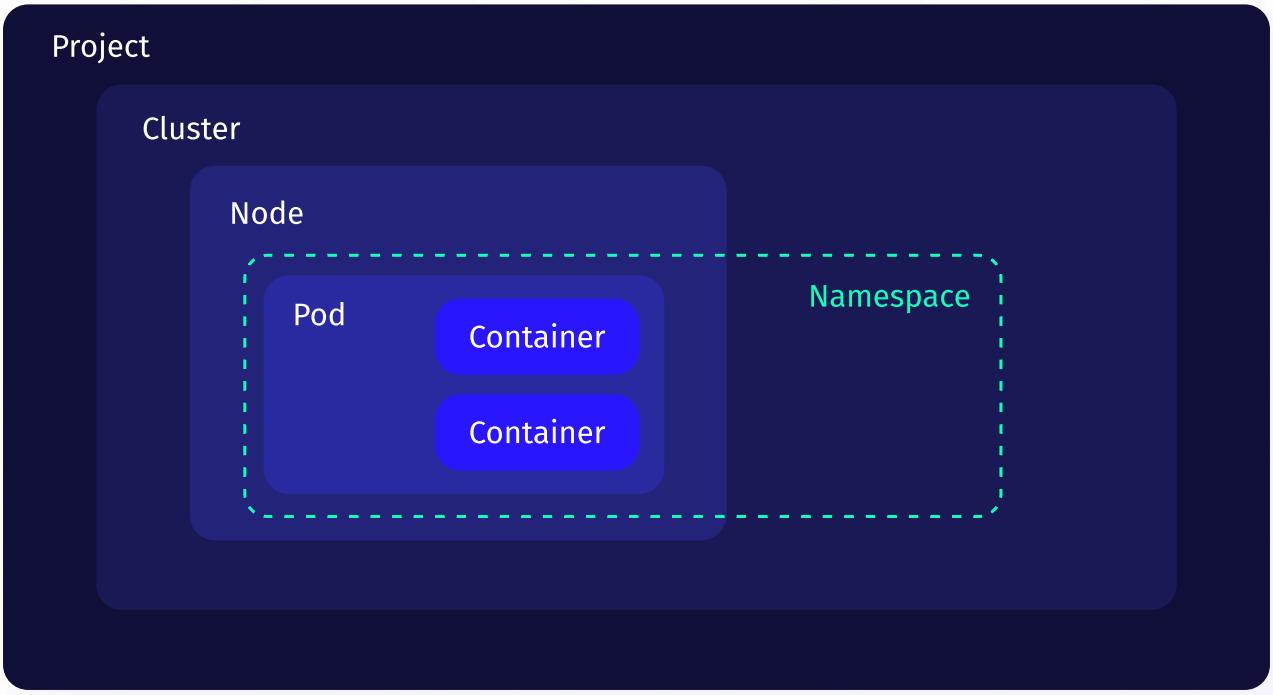
Configuring the system network interfaces: Properly configuring network interfaces helps prevent unauthorized access, isolate network traffic, and protect against attacks targeting the nodes' network connectivity.

Node hardening in Kubernetes primarily takes place at or above the operating system level. This means that the nodes, which are the individual servers in a Kubernetes cluster, can be either virtual or physical. The choice between a virtual or physical environment is often dictated by the specific needs of the organization or any regulatory compliance requirements. Regardless of the environment, the objective of node hardening remains the same - to enhance the security of the Kubernetes cluster. However, care should be taken to evaluate how the environment changes the security requirements for a node. Factors like physical environments which are likely to be accessed by untrusted parties, or virtualization in a multi-tenant environment may impose additional requirements versus running in a trusted datacenter.

The topology and distribution of workloads across the already hardened nodes also matters for ensuring cluster availability. For example, a single node serving both control plane components and application workloads can be used in a Kubernetes test environment to reduce the cost of the testing infrastructure. In a real-world production environment, it's best to separate application workloads from control plane components. This separation is crucial to minimize risks and support resilience. It protects the control plane from threats like malicious activities, malfunctions, resource-intensive processes, or disruptive behaviours from other components.

_Multi-tenancy and Workload Isolation

In a typical Kubernetes cluster, multiple users, clients, and applications share the cluster's resources. To ensure the protection of these resources, it becomes necessary to enforce boundaries between distinct logical groups of users and their associated cluster resources. The degree of separation required can be thought of as a spectrum from tenants having full and complete trust of one another to tenants having absolutely zero trust and assuming malicious intent by other tenants. At the most extreme end, the only sure option is to offer separate Kubernetes clusters to each tenant. This is sometimes referred to as “hard multi-tenancy”.



Source: ARMO

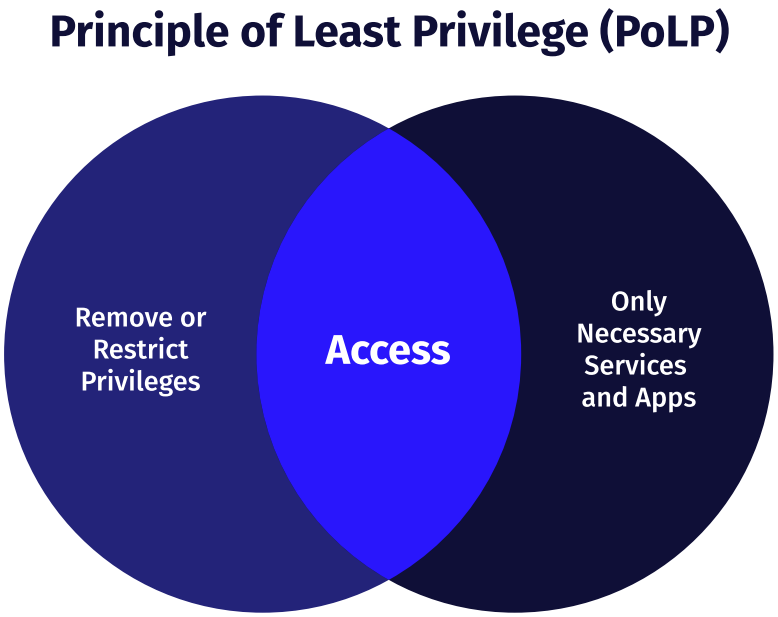
For clusters that can tolerate “softer” isolation, there are a number of Kubernetes features for separating workloads to ensure their security and performance. Namespaces allow grouping resources of many different types, and serve as a foundationfor other types of policy enforcement. Role-Based Access Control (RBAC) can limit members of one logical tenant to making API requests only for their own resources. Network traffic permitted in and out of a pod can be controlled with NetworkPolicies, which are also namespace-scoped. Additional requirements like requiring resource requests and limits, spreading workloads across particular groups of nodes, and preventing the creation of Pods with excessive privileges can be enforced by admission controllers.

_Using Kubernetes Secrets for Application Credentials

A Secret is an object in Kubernetes that holds sensitive information like passwords or tokens for authentication. It's important to know where and how these sensitive data are stored and accessed. This is especially important for applications that are accessible to the public, as they may be more vulnerable to security threats. Instead of passing secrets as environment variables, it's recommended to mount them as read-only volumes in containers for increased security. Reading the secret from a file is less likely to inadvertently result in the secret being written to a log file or being accessible to other applications sharing the environment.

_Apply Least Privilege on Access

Least privilege is a security concept in which a user, application or service is given the minimum levels of access necessary to complete a function. This principle is a proactive step towards protecting system information and functionality from faults and malicious behavior.



Source: ARMO

Most Kubernetes actions require authorization. Once a user has been logged in (authenticated), their request to access the cluster's resources will undergo authorization to determine if the user has the necessary permissions. Role-based Access Control (RBAC) determines whether an entity can call the Kubernetes API to perform a specific action on a specific resource.

RBAC authorization makes use of the rbac.authorization.k8s.io API group to make authorization decisions. Activating RBAC requires an authorization flag set to a comma-separated list that contains RBAC and then restarting the API server.

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

Once RBAC has been enabled, it is managed via several types of Kubernetes resources. Roles and their associated permissions are stored as objects in the cluster, meaning they can be stored and versioned as code and deployed like any other cluster resource. Once a role has been defined, it is "bound" to one or more subjects using second Kubernetes object called a

RoleBinding. Roles can be associated with different types of entities, like ServiceAccounts, individual users, or groups from an external identity provider.

Consider the following illustration. This Role allows an entity bound to that Role to perform several types of read actions on all Pod objects in the default namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: read-user
rules:
- apiGroups: [“”]
  resources: [“pods”]
  verbs: [“get”, “watch”, “list”]
```

_Use Admission Controllers

Admission controllers are used by Kubernetes to govern and enforce how the cluster is used by restricting the conditions under which a resource can be created. This process acts as a gatekeeper for intercepting (authenticated and authorized) API requests. Then, either allowing, modifying or rejecting them after authentication and authorization.

An admission controller can be considered either internal or external, from the perspective of the API server. Internal admission controllers run as code inside the API server itself to allow customizing the admission process. External admission controllers sit outside the API server and are configured as a webhook to which the API server forwards incoming requests for validation.

A common use case for admission controllers is to require incoming resources to comply with established security policies or best practices. For example, it is common to deny admission for Pods which attempt to run as the root user, or to require that Pods mount their root filesystems in read-only mode. These types of controls can greatly reduce the impact of a potential pod compromise, and should be considered a matter of best practice.

While Pods are the predominant focus of current security policies, other resources are security-relevant, and may also be targeted for validation. External admission controllers, such as Kyverno and OPA Gatekeeper, serve as policy systems. Their main function is to target specific resources for validation against custom logic. For instance, it can prevent insecure Service configurations. It can also require organization-specific metadata, such as an asset owner label.

_Evolution of built-in Pod-level security controls

In past versions of Kubernetes (from 1.3 through 1.24), the need to enforce Pod-level security settings was met by a built-in PodSecurityPolicy (PSP) admission controller. Over time, the PSP controller became the de facto standard for Pod security in a Kubernetes cluster. However, in order to improve the API design and extend admission-time validation to other resource types, the PSP type and associated admission controller were removed in Kubernetes v1.25. Lacking a built-in direct replacement, several parallel alternatives are now available to cluster administrators.

Starting from version 1.25, a similar mechanism known as Pod Security Admission (PSA) was introduced as an internal admission controller which can enforce one of three predefined “profiles,” or levels of hardening for all Pods within a particular namespace. These profiles are collectively referred to as Pod Security Standards (PSS), and include several distinct policies grouped in a way that reflects Kubernetes SIG Auth’s understanding of common Pod security context use cases.

As of the time of this writing, new features are under development or recently available which are likely to replace PSA in future Kubernetes versions. Validating Admission Policy (VAP) was

introduced in v1.28, offering a built-in option for applying security and other custom policies more granularly than possible with PSA. This option supports Common Expression Language policies that the API server uses to evaluate incoming resources.

Since the removal of PodSecurityPolicy, cluster administrators must weigh the available options for their cluster versions against their risk appetite and policy needs. This may mean adopting PSA, writing VAP policies, using an external admission controller, or a combination of those controls.

_Enforce Restrictive Network Policies

Kubernetes network policies allow you to control the network traffic between pods in a cluster, providing an additional layer of security. It helps to isolate the microservice applications from each other and allows developers to focus on application development without requiring a deep understanding of low-level networking.

These policies let you decide how a pod communicates with other pods, namespaces, or IP addresses.

Network policies cover the following:

Entities You Control: You can control communication between your pods, different namespaces, and specific IP addresses.

Selectors: You can use selectors to decide the allowed behavior in specific pods or namespaces. An example is specifying which pods can communicate with each other.

IP Blocks: When using IP-based policies, you define rules based on ranges of IP addresses (CIDR ranges).

Remember, there is a minimum of required communication and traffic to and from the node where a pod is running is always allowed. The Kubernetes Network policy works by allowing you to define rules that control how pods communicate with each other and other network

endpoints in a cluster. If selectors match a pod in one or more NetworkPolicy objects, then the pod will accept only connections that are allowed by at least one of those NetworkPolicy objects. A pod that is not selected by any NetworkPolicy objects is fully accessible.

Container

Kubernetes is an orchestration system for containerized workloads, typically using containerd, CRI-O or Docker as the underlying Container Runtime. While many container security options are configurable through the Kubernetes API, there are controls and behaviors which can further reduce a container's inherent risk which can't be directly controlled from within the Kubernetes API.

_Container Runtime Hardening

Container runtime hardening is a critical aspect of securing containerized applications. It involves implementing various security measures to protect the container runtime environment from potential vulnerabilities and attacks. One important step in container runtime hardening is ensuring that only trusted container images are used. Regular scanning and verification can detect known vulnerabilities. Additionally, implementing strong access controls and isolation mechanisms helps prevent unauthorized access and lateral movement within the container at runtime. Regular updates and patches to the container runtime software are also essential to address any newly discovered security vulnerabilities. By prioritizing container runtime hardening, organizations can enhance the overall security posture of their containerized applications.

_Use Trusted Images with Proper Tags

Pods and Pod-controlling resources like Deployments and Jobs make it easy to start a containerized application by simply specifying the name of an image to run. However, just as care should be taken not to run untrusted software on a local workstation, images should be evaluated for trustworthiness before being run in a cluster.

Public container registries like Docker Hub can contain outdated, unmaintained, and even malicious images in addition to the many legitimate and trustworthy images hosted there. Developers should ensure that any public images they use are maintained in accordance with their organization's security policies. For example, images should only be pulled from registries controlled by trusted entities, to reduce the likelihood of pulling a malicious image. This might mean that only approved third-party registries may be used (e.g. "only images from the nginx organization on Docker Hub may be used"), or that all third-party images must first be copied into a central private registry before being used. Admission controllers can be used to restrict which registries are permissible for pulling cluster images.

Regardless of where an image is pulled from, it is important for software to be uniquely and immutably identifiable. In a Kubernetes-orchestrated container context, this means that the image used in a Pod should include a tag which uniquely identifies the image to be pulled. This requires at least eliminating the use of so-called floating tags, like "latest", which are updated periodically to point to new versions of the image. An even better practice is to identify images based on their SHA (Secure Hash Algorithm), instead of a tag version. This is a bit less readable for humans, but ensures that the same image will be pulled even if tags are modified or removed from the registry. Admission controllers can also be used to enforce the use of SHAs as image identifiers, as well as to mutate workloads to replace tags with SHAs, if desired.

While not yet universally imposed, cryptographic image provenance is steadily progressing from optional measure to mandatory requirement. As industry standards, regulations, and security frameworks evolve, verifying image origins is increasingly necessary before deployment in today's ecosystem. Ensuring software integrity requires verifying not just the image itself, but its entire journey, which is generally referred to as supply chain security. This means checking the signature attached to the image, as well as other build artifacts involved in its creation. This chain of trust begins with developers and extends through the build environment, ending in the container runtime where the journey ends. Securing each step fosters confidence in the software's authenticity and prevents unauthorized modifications along the way. While this adds complexity for maintainers, security-conscious developers can take the lead by signing their own images and artifacts. Tools like cosign

can be used to create digital signatures, and admission controllers can require and verify these signatures at cluster admission time.

_Reduce Container Attack Surface

Despite the name, containers are not virtual machines, and do not inherently isolate an application enough to prevent it from abusing its underlying node or other workloads. Containers describe an environment to be used for starting a process on the underlying node, but that process shares the memory, CPU, and certain OS and filesystem resources of the node itself. As such, containers should be crafted in a way that protects the node and other workloads if the container were to become compromised or malfunction.

Many of these concerns can be configured and enforced through the Kubernetes API, as described in the previous section on using admission controllers. Other factors, however, must be addressed at image creation time, beyond the scope of Kubernetes API controls.

One major threat: running containers as the root user. In this scenario, the container gains the same powerful privileges as the node itself, making it much harder to contain and potentially allowing privilege escalation if vulnerable. Workloads typically should not need to run as root, but the default user of many common base images is still root, so workload creators must take care to change the user when crafting their container image.

Code

The code layer is also referred to as application security, with emphasis on the security of the actual logic used by any of the applications running in the container. It is also the layer over which businesses have the greatest control. Attackers often target application code because it's frequently updated, less scrutinized, and directly exposed to the internet. If other system components are secure, this becomes their primary focus.

_Scan For Vulnerabilities

Most applications rely heavily on open-source packages, libraries, and other third-party components. As a result, a vulnerability in any one of these dependencies can affect the security of the whole application. The more dependencies an app has, the higher the odds: at least one is probably already vulnerable.

Attackers frequently target workloads by exploiting known vulnerabilities in widely used dependency code. Therefore, some mechanism is needed to periodically verify that those dependencies are up to date and vulnerability-free, to the extent possible. In practice, this often means including scanners at various points in the lifecycle of a container image. These scanners will scan the image to identify the software contained in it, and issue some form of warning if the software version in use contains a known vulnerability. This warning may come in the form of a blocked CI/CD pipeline, rejected cluster admission, or an issue opened in a bug tracking system.

Scanning can be made more compute-efficient and precise by generating and including one or more Software Bills of Materials (SBOMs) alongside an image at build time. An SBOM gives you a clear, exact record of software in your image, thanks to precise versions from package management tools. This improves scan time because an image does not need to be pulled and have its filesystem examined to identify packages. The packages are already listed in the SBOM, so a scanner needs only to retrieve the SBOM and compare the listed components against its vulnerability database.

SBOM scanning is becoming mainstream, and is not without trade-off's. For example, SBOMs must contain a complete and accurate list of packages to correctly assess the vulnerabilities contained in the associated image. SBOMs are only as trustworthy as their creators. Tools must accurately capture all software in the image, and the list must remain tamper-proof. While signatures can verify integrity, completeness still depends on the tools and image structure.

Implementing Kubernetes Security Best Practices

There is a lot to do on many levels to secure Kubernetes infrastructure. It is a continuous process of patching and hardening that will keep your workloads, business and customers safe.

Security Frameworks

There are a variety of security frameworks that are applicable to Kubernetes. Many prominent pre-Kubernetes frameworks, like MITRE and DISA STIGs, have been updated to include Kubernetes-specific concerns. Additionally, a number of existing governing bodies have published new guidance specifically covering Kubernetes security, including the Center for Internet Security (CIS), and the US National Institute of Standards and Technology (NIST). Other existing security standards like PCI-DSS and SOC may be relevant for certain industries or types of Kubernetes users.

Organizations can use these frameworks as a guide for implementing security measures on their Kubernetes infrastructure. Organizations should account for their risk tolerance and the inherent tradeoffs between security, cost, and accessibility/friction. The CIS benchmark is commonly used as a starting point to ensure that a Kubernetes cluster itself has been appropriately hardened to host production workloads.

Security Updates for the Environment

Containers running a variety of open-source software and other software packages, may be attacked via an exploit before it can be patched through routine updates. Hence, it is critical to scan images to identify high priority vulnerabilities and prioritize urgent workload updates,

while keeping up with software updates. Using Kubernetes' rolling update capability, it is possible to gradually upgrade a running application by updating it to the most recent version available on the platform.

Security Context

Kubernetes Security Context is a set of security settings that provides the ability to define privilege and access controls for pods or containers. Each pod and container has its own security context, which defines all of the privileges and access control settings that can be used by a container. These settings encompass a range of different configurations such as being able to run privileged, whether a container's root filesystem should be mounted as read-only, access control based on UID and GID, system-level capabilities, and whether built-in Linux security mechanisms such as seccomp, SELinux, and AppArmor should be leveraged.

Security context values can greatly reduce the impact of a successful compromise. As a result, Pods should be held to very high standards at admission time. In addition to preventing known unsafe values, newer policies also require explicitly setting preemptively safer configuration. For example: requiring a container to drop all unused capabilities, even those which are currently believed to be safe.

Moreover, a pod-level security context will also result in settings being applied to volumes when they are mounted. Pod-level security contexts will result in constraints being applied to all containers that run within the pod in question. In cases where the same settings are not ap[plicable to all containers in a given pod, Kubernetes allows you to specify security contexts for individual containers as well.

Resource Management

A major feature of Kubernetes is the ability to intelligently schedule workloads across nodes.

In determining an appropriate node for a particular Pod, the scheduler can account for the compute resources (CPU and memory) a Pod requires. To make use of this feature, a Pod must first specify how much CPU and memory it needs to run, and can optionally indicate an upper limit which it expects to reach. If no limits are specified, there is no implicit constraint on the resources available to a workload. This means that a compromised or inefficient workload can consume all the resources on a node, leading to a potential denial of service for other workloads.

Furthermore, the effectiveness of auto-scaling features like Horizontal or Vertical Pod Autoscalers (HPA and VPA) depends on the behavior of an application under load and the expected changes in its resource usage. To optimize the use of these features, it is recommended to define meaningful requests and limits for the target workload. This allows for better resource allocation and ensures that the auto-scaling features can function properly.

Having said that, Pod requests and limits are not required by Kubernetes itself. Many organizations choose to enforce this requirement using an admission controller, in order to ensure workloads can be properly scheduled.

Shift Left

Shift left security involves conducting security testing early in the container development process, allowing for earlier identification and resolution of vulnerabilities. By catching security issues early in the software development lifecycle (SDLC), it helps reduce the time spent fixing them later. This results in a shorter development cycle, improved overall security, and faster deployments.

It is good practice if the security controls in your Kubernetes security scanners can be enforced in a CI/CD pipeline. However the ultimate in shifting left is reaching developer workstations with the controls mentioned above. Thuse, providing timely feedback to developers and preventing insecure workloads from ever being committed or built.

Conclusion

Containerized applications have significantly shaped today's computing landscape. As enterprises increasingly migrate applications to cloud-native setups, Kubernetes emerges as a pivotal open-source orchestrator driving continuous growth. Its role in simplifying deployment, scaling, and orchestrating diverse workloads on shared infrastructure brings myriad advantages. However, this transition introduces security complexities that demand attention.

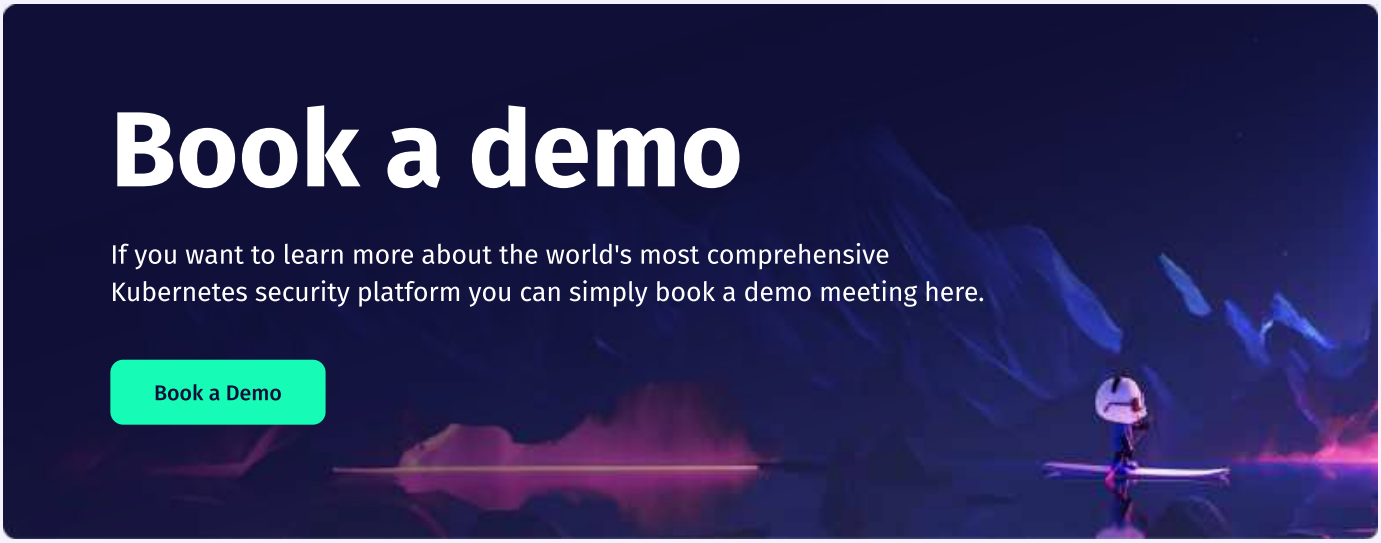
Ensuring security across the orchestration layer, underlying infrastructure, and applications themselves is a necessity. The thriving Kubernetes ecosystem, supported by a robust open-source community, offers many tools and evolving, user-centric processes tailored to managing complex applications at scale. Fortunately, most security processes and controls remain familiar, necessitating only minor adjustments in a Kubernetes context. Moreover, the transition to Kubernetes presents organizations with a prime opportunity to reassess their existing security protocols. It allows them to retain effective methods while updating or replacing outdated strategies unfit for the demands of a cloud-native environment.

About Armo

ARMO’s mission is to build an end-to-end Kubernetes security platform, powered by open source. A platform which covers all Kubernetes security issues without adding to engineers’ burden. ARMO focuses solely on open source based CI/CD & Kubernetes security, allowing organizations to be fully compliant and secure from code to production. Our solutions make security simple and frictionless for DevOps and are embraced by security.

ARMO Platform

ARMO Platform is an enterprise-grade Kubernetes Native Application Protection Platform (KNAPP) that’s widely used by DevOps teams. It is powered by Kubescape, the fastest growing open-source project for Kubernetes security. Armo Platform solves one of the main challenges in addressing vulnerabilities and misconfigurations - the extensive analysis required to ensure changes won’t disrupt the application. ARMO Security Co-Pilot is the first Kubernetes-specific solution that analyzes applications in real-time. It filters out irrelevant vulnerabilities and policies and generates safe remediation suggestions and security policies that won’t break applications. Leaving your applications secure and performant.



ARMO

The Makers of Kubescape

ARMO PLATFORM

Sign up for
ARMO Platform

Get involved
on **Github**

Follow us
on **X**

Join the discussion
on **Slack**