The DevSec Guide to Kubernetes

Kubernetes® is the de facto container orchestration system, offering development teams incredible scale, flexibility, and speed when deploying and managing cloud native applications. For all its benefits, however, it also brings new complexity and risk.

In this guide, we'll explore the unique considerations Kubernetes presents for cloud native application security and how to build on top of its built-in security foundation to embrace DevSecOps.

We will focus mostly on securing Kubernetes infrastructure up to the container by leveraging its built-in security features and securing infrastructure as code (IaC) templates used to configure clusters and their components.

Table of Contents

Basic Kubernetes Security Considerations

To help manage containerization at scale, Kubernetes has risen in popularity as the de facto container orchestrator. It offers development teams incredible scale, flexibility, and speed when deploying and managing cloud native applications. For all its benefits, however, it also brings new complexity and security considerations.

Kubernetes Security Across Each Layer

The biggest reason Kubernetes security can be challenging is that it isn't a single, simple framework. It's a complex, multilayered beast. Each layer poses its own set of security challenges and requires unique solutions. And because the layers are interconnected, any issue at one layer, such as a container image vulnerability, gets amplified if there's a security weakness in another layer, such as an infrastructure misconfiguration. These are the basic components to address:

- **Cluster:** Securing Kubernetes deployments requires securing the underlying infrastructure (nodes, load balancers, etc.), configurable components, and the applications which run in the cluster, including maintaining the posture of underlying nodes and controlling access to the API and kubelet. It's also important to prevent malicious workloads from running in the cluster and isolate workload communication through strict network controls.
- **Control plane:** Also known as parent nodes, control planes include schedulers, API servers, and other components that make global decisions for worker nodes within clusters. Minimizing admin-level access to control planes and ensuring your API server isn't publicly exposed are the most important security basics.
- **Pod:** Kubernetes nodes host pods or workloads, which are collections of containers that share common configurations. Isolating non-dependent pods from talking to one another using network policies is important to prevent lateral movement across containers in the event of a breach. Kubernetes also has built-in features to define how pods behave and what they can access.
- **Containers:** Container security depends on the security of the images that make them up. This requires using trusted registries and libraries and scanning them for vulnerabilities in build time. It's also important to monitor runtime containers for suspicious activity such as a shell running inside, or sensitive data being leaked.
- **Code:** Application and infrastructure code is almost entirely controlled by the end user and thus often presents attackers with the most penetrable parts of the attack surface. Minimizing known vulnerabilities and attack vectors reduces the ways an application can be exploited.

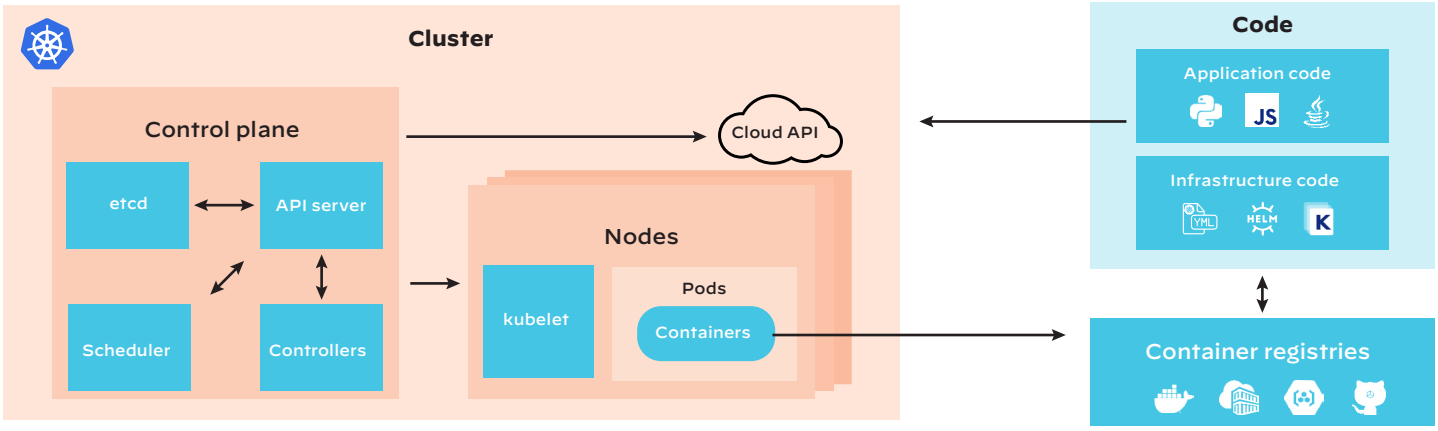

Figure 1: Diagram of Kubernetes infrastructure

Built-in Kubernetes Security Features

To help navigate its complex and interconnected attack surface, Kubernetes provides these security features:

- **Kubernetes Role-Based Access Control (RBAC):** RBAC allows you to limit access to services based on roles and identities. It controls resource access to users and applications through authorization to improve the security of a cluster by limiting who has access to what.

- **Network Policies:** These policies enable you to regulate traffic flow for specific applications in your cluster at the IP address and port level by defining how a pod can communicate with different network elements (endpoints and services) over the network.
- **Admission Controllers:** These plugins serve as gatekeepers, intercepting API requests and determining if they violate a predefined policy before rejecting or modifying them to meet policy. When configured correctly, Admission Controllers are used to provide several basic Kubernetes best practices, such as limits and requests, and ensure pods are not overly privileged.
- **Transport Layer Security (TLS) for Kubernetes Ingress:** You can configure access to your Kubernetes Ingress by defining which inbound connections reach which services using a set of rules, allowing you to combine all of your routing rules into a single resource. A secret that includes a TLS private key and certificate can be used to secure a Kubernetes app. Only one TLS port, 443, is supported by Ingress, which assumes TLS termination. The TLS secret must have the keys named tls. crt and tls. keys, which contain the TLS certificate and private key, respectively.
- **Pod Security Policies:** PSPs let you control pod and container behavior by specifying a set of requirements that pods must follow in order to be accepted by the cluster; if a request to create or update a pod fails to match the requirements, the request is rejected, and an error is returned. PSP is far from comprehensive, which is why it is currently being deprecated.

These built-in capabilities create some guardrails for limiting access and controlling pod and container behavior. That said, a truly secure Kubernetes environment requires a holistic and automated approach.

Kubernetes and IaC

Manually provisioning Kubernetes clusters and their add-ons is time-consuming and error-prone. That's where infrastructure as code (IaC) comes in. IaC utilizes machine-readable files to manage resources such as data servers, storage, and networks programmatically, allowing for version control, auditability, and easier collaboration and testing.

Kubernetes manifest files are an example of IaC. Written in JSON or YAML format, manifests specify the desired state of an object that Kubernetes will maintain when you apply the manifest. Here is an example manifest for NGINX deployment into a Kubernetes cluster:

```
apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: nginx-deployment
    labels:
      app: nginx
  spec:
    replicas: 3
    selector:
      matchLabels:
        app: nginx
    template:
      metadata:
        labels:
          app: nginx
      spec:
        containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
          - containerPort: 80
```

Additionally, packaged manifests such as Helm charts and Kustomize files simplify Kubernetes even further by reducing complexity and duplication.

The biggest security advantage with IaC is that it enables you to scan earlier in the development lifecycle to catch easy-to-miss misconfigurations before they're deployed.

Takeaway: IaC Security Is Key

Automated and continuous IaC scanning is a huge component of DevSecOps and is crucial for securing cloud native apps.

Basic Kubernetes Security Best Practices

When implementing DevSecOps for securing cloud native applications, it's essential to understand basic Kubernetes infrastructure pitfalls and how IaC can help mitigate each challenge.

These are some of the most common Kubernetes security errors and misconfigurations:

- Leaving host infrastructure vulnerable
- Granting overly permissive access to clusters and registries
- Running containers in privileged mode and allowing privilege escalation
- Pulling "latest" container images
- Failing to isolate pods and encrypt internal traffic
- Forgetting to specify resource limits and enable audit logging
- Using the default namespace
- Incorporating insecure open source components

Kubernetes Host Infrastructure Security

Let's start at the most basic layer of a Kubernetes environment: the host infrastructure. This is the bare metal and/or virtual server that serve as Kubernetes nodes. Securing this infrastructure starts with ensuring that each node (whether it's a worker or a parent) is hardened against security risks. An easy way to do this is to provision each node using IaC templates that enforce security best practices at the configuration and operating system level.

When you write your IaC templates, ensure that the image template and/or the startup script for your nodes are configured to run only the strictly necessary software to serve as nodes. Extraneous libraries, packages, and services should be excluded. You may also want to provision nodes with a kernel-level security hardening framework, as well as employ basic hygiene like encrypting any attached storage.

IAM Security for Kubernetes Clusters

In addition to managing internal access controls within clusters using Kubernetes RBAC and depending on where and how you run Kubernetes, you may use an external cloud service to manage access controls for your Kubernetes environment. For example, with AWS EKS, you'll use AWS IAM to grant varying access levels to Kubernetes clusters based on individual users' needs.

Using a least-privilege approach to manage IAM roles and policies in IaC minimizes the risk of manual configuration errors that could grant overly permissive access to the wrong user or service. You can also scan your configurations with IaC scanning tools (such as our open source tool, Checkov) to automatically catch overly permissive or unused IAM roles and policies.

Container Registry Security

Although they're not part of native Kubernetes, container registries are widely used as part of a Kubernetes-based application deployment pipeline to store and host the images deployed into a Kubernetes environment. Access control frameworks vary between container registries. With some, you can manage access via public cloud IAM frameworks. Regardless, you can typically define, apply, and manage them using IaC.

In doing so, you'll want to ensure that container images are only accessible by registry users who need to access them. You should also prevent unauthorized users from uploading images to a registry, as insecure registries are an excellent way for threat actors to push malicious images into your environment.

Avoid Pulling "Latest" Container Images

When you tell Kubernetes to pull an image from a container registry, it will automatically pull the version of the specified image labeled with the "latest" tag in the registry.

While this may seem logical—after all, you typically want the latest version of an application—it can be risky from a security perspective because relying on the "latest" tag makes it more difficult to track the specific version of a container that you are using. In turn, you may not know whether your containers are subject to security vulnerabilities or image-poisoning attacks that impact specific images on an application.

To avoid this mistake, specify image versions, or better yet, the image manifest when pulling images, and audit your Kubernetes configurations to detect instances that lack specific version selection for images. It's a little more work, but it's worth it from a Kubernetes security perspective.

Avoid Privileged Containers and Escalation

At the container level, a critical security consideration is ensuring that containers can't run in privileged mode. Running containers in privileged mode—which gives them unfettered access to host-level resources—grants way too much power but, unfortunately, is an easy mistake to make.

Disallowing privilege escalation is easy to do using IaC. Simply write a security context that denies privilege escalation, and make sure to include the context when defining a pod:

```
securityContext:
        allowPrivilegeEscalation: false
```

Then, ensure that privilege isn't granted directly with the "privilege" flag or by granting CAP_SYS_ADMIN. Here again, you can use IaC scanning tools to check for the absence of this security context and to catch any other privilege escalation settings within pod settings.

Isolate Pods at the Network Level

By default, any pod running in Kubernetes can talk to any other pod over the network. That's why, unless your pods actually need to talk to each other (which is usually only the case if they are part of a related workload), you should isolate them to achieve a higher level of segmentation between workloads.

As long as you have a Kubernetes networking layer that supports Network Policies (most do, but some Kubernetes distributions default to CNIs that lack this support), you can write a Network Policy to specify which other pods the selected pod can connect to for both ingress and egress. The value of defining all of this in code is that you can scan the code to check for configuration mistakes or oversights that may grant more network access than intended.

Encrypt Internal Traffic

Another small but critical setting is the *--kubelet-https=...* flag. It should be set to "true." If it's not, traffic between your API server and kubelets won't be encrypted.

Omitting the setting entirely also usually means that traffic will be encrypted by default. But because this behavior could vary depending on which Kubernetes version and distribution you use, it's best practice to be explicit about requiring kubelet encryption, at least until the Kubernetes developers encrypt traffic universally by default.

Specifying Resource Limits

You may think of resource limits in Kubernetes as a way to control infrastructure costs and prevent "noisy neighbor" issues between pods by restricting how much memory and CPU a pod can consume. In addition, resource limits are crucial from a security perspective, helping to mitigate the risk of denial-of-service (DoS) attacks. A compromised pod can do more damage when it can suck up the entire cluster's resources, depriving other workloads of functioning properly. From a security perspective, then, it's a good idea to define resource limits.

Avoiding the Default Namespace

By default, every Kubernetes cluster contains a namespace named "default." As the name implies, the default namespace is where workloads will reside by default unless you create other namespaces.

Using the default namespace presents two security concerns. First, your namespace is a significant configuration value, and if it's publicly known, it's that much easier for attackers to exploit your environment. The other, more substantial concern with default namespaces is that if everything runs there, that means your workloads aren't segmented. It's better to create separate namespaces for separate workloads, making it harder for a breach against one workload to escalate into a cluster-wide issue.

To avoid these issues, create new namespaces using kubectl or define them in a YAML file. You can also scan existing YAML files to detect instances where workloads are configured to run in the default namespace.

Enable Audit Logging

If a security incident (or, for that matter, a performance incident) does occur, Kubernetes audit logs tend to be helpful in researching it. This is because audit logs record every request to the Kubernetes API server and its outcome. Unfortunately, audit logs aren't enabled by default in most Kubernetes distributions. To turn this feature on, add lines like these to your kube-apiserver policy file to tell Kubernetes where to store audit logs and where to find the policy file that configures what to audit:

```
--audit-log-path=/var/log/kubernetes/
apiserver/audit.log
  --audit-policy-file=/etc/kubernetes/audit-
policies/policy.yaml
```

Because audit logs offer critical security information, it's worth including a rule in your Kubernetes configuration scans to check whether audit logs are enabled.

Securing Open Source Kubernetes Components

Leveraging open source components, such as container images and Helm charts allows developers to move fast without reinventing the wheel. Open source components, however, are not typically secure-by-default, so you should never assume a container image or Helm chart is secure.

Our research shows that around half of all open source Helm charts within Artifact Hub contained misconfigurations, highlighting the gap in how and where Kubernetes security is being addressed.

Before using open source Helm charts from Artifact Hub, GitHub, or elsewhere, you should scan them for misconfigurations. Similarly, before deploying container images, you should scan them using tools to identify vulnerable components within them. Identifying security risks within Kubernetes before putting them into production is crucial, especially when it comes to integrating open source components into your environment.

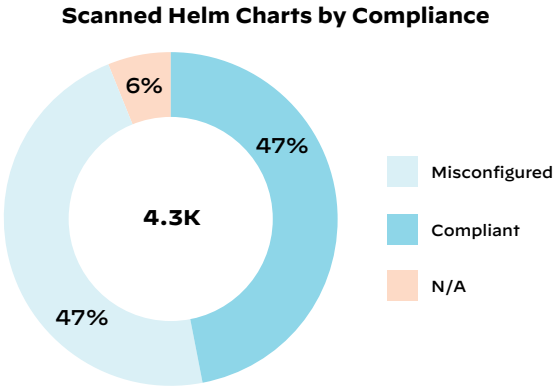

Figure 2: High-level findings from Bridgecrew's open source Helm security research

Kubernetes Security Across the DevOps Lifecycle

Identifying common Kubernetes security issues early is crucial to building a repeatable and efficient DevSecOps strategy as feedback cycles are faster and cheaper. Managing Kubernetes security risks at every stage is also key to securing cloud native applications as they become more representative throughout the lifecycle.

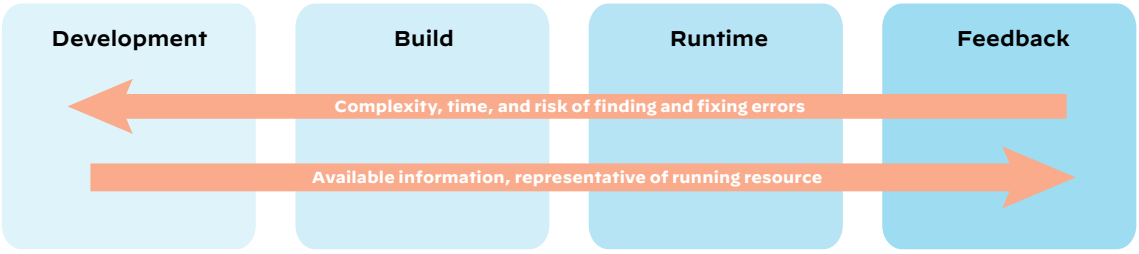

Figure 3: Kubernetes security considerations across development phases

Development

During the development stage, engineers are writing code for applications and infrastructure that will later be deployed into Kubernetes. These are the three main types of security flaws to avoid in this phase:

- Misconfigurations within IaC files
- Vulnerabilities in container images
- Hard-coded secrets

You can check for misconfigurations using IaC scanners which can be deployed from the command line or via IDE extensions. IaC scanners work by identifying missing or incorrect security configurations that may later lead to security risks when the files are applied.

Container scanning checks for vulnerabilities inside container images. You can run scanners on individual images directly from the command line. However, most container registries also feature built-in scanners. The major limitation of container image scanning tools is that they can only detect known vulnerabilities—meaning those that have been discovered and recorded in public vulnerability databases.

When scanning container images, keep in mind that container scanners designed to validate the security of container images alone may not be capable of detecting risks that are external to container images. That's why you should also be sure to scan Kubernetes manifests and any other files that are associated with your containers. Along similar lines, if you produce Helm charts as part of your build process, you'll need security scanning tools that can scan Helm charts.

In addition to avoiding misconfigurations and vulnerabilities, it's crucial to avoid hard coding secrets such as passwords or API keys that threat actors can leverage to gain privileged access. Secrets scanning tools are key to checking for sensitive data like passwords and access keys inside source code. They can also be used to check for secret data in configuration files that developers write to govern how their application will behave once it is compiled and deployed.

The key to getting feedback in this stage is to integrate with developers' local tools and workflows—either via integrated development environments (IDEs) or command lines. As code gets integrated into shared repositories, it's also important to have guardrails in place that allow for team-wide feedback to collaboratively address.

Build and Deploy

After code is written for new features or updates, it gets passed on to the build and deploy stages. This is where code is compiled, packaged, and tested.

Security risks like over-privileged containers, insecure RBAC policies, or insecure networking configurations can arise when you deploy applications into production. These insecure configurations may be baked into the binaries themselves, but they could also originate from Kubernetes manifests that are created alongside the binaries in order to prepare the binaries for the deploy stage.

Due to the risk of introducing new security problems while making changes like these, it's a best practice to run a final set of scans on both your application and any configuration or IaC files used to deploy it just before you actually deploy.

> **Takeaway: Leverage CI/CD**
>
> **Integrating security checks into your CI/CD pipeline is the most consistent way to get security coverage on each Kubernetes deployment.**

This is your last chance to address security issues before they affect users in production. Doing so automatically is the only way to get continuous security coverage.

Runtime

Once your app has been deployed into production, it enters the final stage of the development lifecycle: runtime. If you've done the proper vetting during earlier stages of the DevOps lifecycle, you can be pretty confident that your runtime environment is secure.

However, there's never a guarantee that unforeseen vulnerabilities won't arise within a runtime environment. There is also the risk that developers or IT engineers could change configurations within a live production environment.

That's why it's important to scan production environments continuously in order to detect updates to Security Contexts, Network Policies, and other configuration data. You want to know as soon as possible if someone on your team makes a change that creates a security vulnerability, or worse, if attackers who have found a way to access your cluster are making changes in an effort to escalate the attack.

On top of scanning configuration rules, you should also take steps to secure the Kubernetes runtime environment by hardening and monitoring the nodes that host your cluster. Kernel-level security frameworks like AppArmor and SELinux can reduce the risk of successful attacks that exploit a vulnerability within the operating systems running on nodes. Monitoring operating system logs and processes can also allow you to detect signs of a breach from within the OS. Security information and event management (SIEM) and security orchestration, automation, and response (SOAR) platforms are helpful for monitoring your runtime environment.

Feedback and Planning

The DevOps lifecycle is a continuous loop in the sense that data collected from runtime environments should be used to inform the next round of application updates.

From a security perspective, this means that you should carefully log data about security issues that arise within production, or, for that matter, at any earlier stage of the lifecycle, and then find ways to prevent similar issues from recurring in the future.

For instance, if you determine that developers are making configuration changes between the testing and deployment stages that lead to unforeseen security risks, you may want to establish rules for your team that prevent these changes. Or, if you need a more aggressive stance, you can use access controls to restrict who is able to modify application deployments. The fewer people who have the ability to modify configuration data, the lower the risk of changes that introduce vulnerabilities.

DevSecOps Tips for Cloud Native Apps

Technology is core to implementing a Kubernetes-based DevSecOps strategy, but without the right culture, it can actually create the friction and bottlenecks that it's trying to avoid. Aligning development, security, and operations teams, however, is not an easy feat, as these teams' goals are often at odds with one another.

People

Having the right people sets the foundation for DevSecOps. Security training and fostering security champions have been the go-to solution for making security matter, but you can't stop there. DevSecOps requires bi-directional knowledge sharing to build true shared accountability for security.

Building your team based on formal titles isn't necessary for building the right culture; whether you already have the right building blocks or are looking to build out your teams, these are some of the skills you should look out for:

- **A knack for efficiency:** Regardless of department, efficiency and automation are key to DevSecOps success. When manual work inevitably crops up, teammates with productivity mindsets will invest the time to make that repeatable in the future despite the temptation to just complete the task at hand.
- **Balance individual focus and greater goals:** DevOps aims to break down the development process into smaller components and processes, isolating individual outcomes at each phase. DevSecOps requires striking the right balance between security and efficiency. To do that in practice, priorities need to be set, recognized, and constantly evaluated from the organization level to the individual contributors.
- **Continuous learning:** Although Kubernetes has been around for a while now, it's valuable for everyone to be constantly learning new things when it comes to building the most performant and innovative products. The same goes for security. Staying on top of the latest vulnerabilities and policies is essential to keeping your applications secure. Having natural curiosity is ideal, but with consistent processes for training and education, you can achieve the same outcome.

Processes

The DevSecOps paradigm necessitates new processes or perhaps improvements to existing ones that prioritize security at each step.

- **Development:** As code is being written and updated, securing feedback should be incorporated into workflows via IDE extensions or CLI tools. By surfacing security best practices earlier, it's easier to

address issues with the right context and quickly prevent them from progressing further. This is also a great way to foster continuous security education through actionable insights.

- **Build and deploy:** As you add checks to your pull/merge requests and CI/CD pipelines, ensure that all stakeholders are aware, and expectations are aligned. That way, friction won't arise when a build fails, or a deployment is blocked due to a critical misconfiguration or vulnerability. When issues do arise, make sure you have individuals responsible and on call to help keep things running smoothly.
- **Runtime:** Even with the most mature, proactive security guardrails in place, the work doesn't stop at deployment. Having the right visibility and processes for when security issues are exposed in runtime is a significant part of a comprehensive DevSecOps strategy.
- **Feedback and planning:** It's important for all stakeholders to understand the security impact new features and updates may have. Security training and awareness are also crucial at this phase, as work done in this phase will determine the security coverage throughout the rest of the development lifecycle.

Setting the right processes in place ensures that everyone is on the same page and sets the foundation for security consistency and cohesiveness.

Tools

The last component of implementing a successful DevSecOps strategy is tooling. The Kubernetes security landscape has numerous tools that tackle various layers and aspects of Kubernetes and cloud native security. Many of those tools have major shortfalls, however.

When researching and implementing security tooling, make sure to keep these criteria in mind:

- **In code, for code:** Whether you're looking for workload protection or Cloud Security Posture Management, having visibility at the code level is crucial. This is especially true if you develop and manage any infrastructure in code—which you likely do if you're leveraging Kubernetes. On a similar note, being able to address issues at the source is the best way to prevent issues from resurfacing and snowballing into hundreds of security alerts in runtime.
- **Integrated into dev tools and workflows:** For code feedback and fixes to be truly actionable, they need to be surfaced at the right time and in the right place. Whether that's on a developer's local workstation, in a pull/merge request, or during a CI/CD build, continuously enforcing security best practices is easiest when layered on top of existing tools and workflows.
- **Code to cloud coverage:** One of the big challenges of securing cloud native environments is that the state and connections of their components change throughout the development lifecycle. That's why coverage across stages is important and why having unified policies and visibility from code to cloud is so important. Only then can you address issues at the source and detect drift. Having that complete coverage also helps to bridge the gap between engineering, operations, and security.
- **Don't forget about compliance:** While many of the issues we've addressed so far relate to security best practices, getting compliance feedback and enforcing rules for maintaining compliance across different benchmarks is also a great use case for embedding security early and often.

Key Performance Indicators (KPIs)

One way to integrate DevSecOps into a team's day-to-day operations is to hold them accountable via shared KPIs. Metrics should take into consideration not only how secure applications are but also how quickly deployments occur and how reliable applications are.

Here are some sample KPIs that touch all development, operations, and security teams:

- **Volume of production issues over time and by severity:** Ideally, the number of misconfigurations in runtime should go down over time if issues are addressed earlier. By having end-to-end visibility, it should also be easier to prioritize higher severity issues, leading to fewer alerts and hardening infrastructure over time.
- **Mean time to remediation (MTTR):** As the volume of issues goes down over time, identified vulnerabilities and misconfigurations should be resolved faster over time. A shorter MTTR also indicates a stronger CI/CD pipeline and institutional knowledge when it comes to infrastructure being deployed and its security expectations.
- **Deployment speed and frequency:** As you bake security measures into your DevOps lifecycle, be sure to monitor how frequently and quickly you're deploying. At the end of the day, security checks are only valuable if you're able to deliver updates, so striking the right balance by tweaking levels of control is key.

Because Kubernetes is such a dynamic and complex system, it's even more crucial to implement a solid set of KPIs to help you assess your organization's success internally and externally. DevSecOps is getting more popular as a means to avoid costly (both in resources and reputation) breaches.

Bringing the right technologies, people, and processes together to establish baselines and measure success over time is necessary for any mature Kubernetes-based DevSecOps strategy.

Conclusion

Kubernetes is key to the cloud native ecosystem, providing a multilayered system for automating, deploying, scaling, and managing containerized applications. Thus, Kubernetes security requires a multipronged approach that addresses the security risks that exist across the various layers of Kubernetes.

By leveraging an IaC-based approach to defining security rules in Kubernetes, you can more easily configure and minimize the risk of configuration issues that will lead to security breaches within any layer of your Kubernetes stack. Leveraging Kubernetes' built-in security features, along with a holistic IaC security strategy, is a solid foundation for building security from the start. DevSecOps is the strategy that makes it all possible, integrating teams with common processes, metrics, and tools under a common goal—to deploy secure applications without being hindered by security best practices.

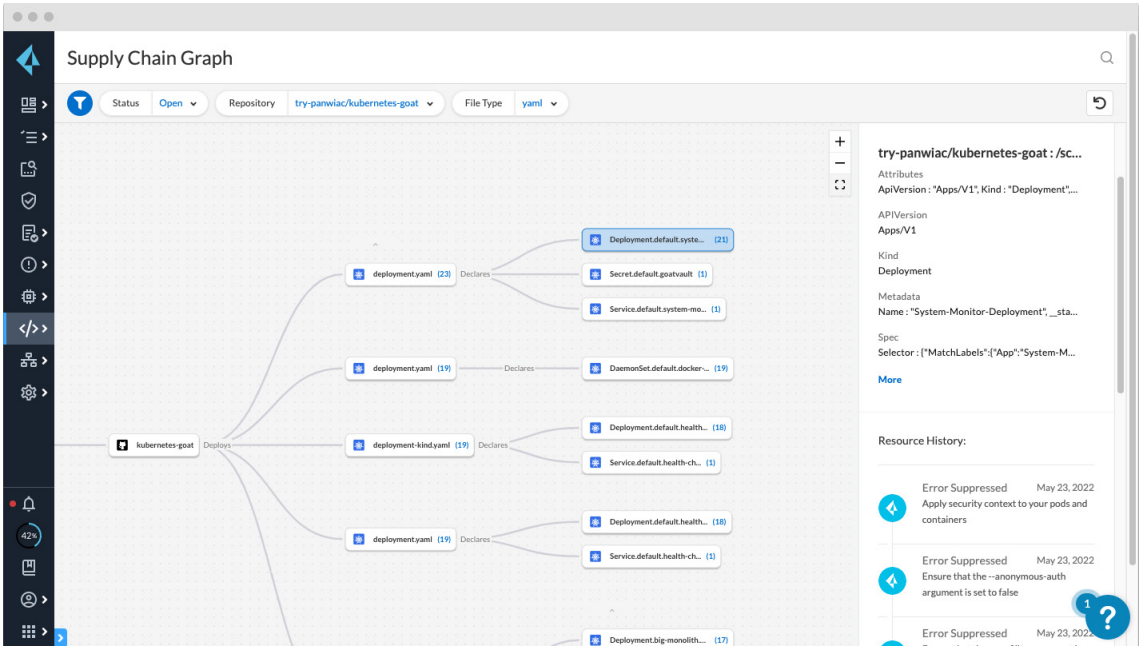

Figure 4: Prisma Cloud supply chain example with Kubernetes

Get started with Prisma® Cloud to:

- Find and fix misconfigurations in cloud resources and IaC.
- Enforce hundreds of built-in policies across security and compliance benchmarks.
- Embed guardrails via IDE plugins, pre-commit hooks, and native VCS and CI/CD integrations.

Request a trial.

paloalto NETWORKS

3000 Tannery Way
Santa Clara, CA 95054

Main: +1.408.753.4000
Sales: +1.866.320.4788
Support: +1.866.898.9087

www.paloaltonetworks.com