# Kubernetes Security

TIPS, TRICKS & BEST PRACTICES

DevOps.com

sponsored by StackRox

# CONTENTS

The drive toward digital transformation has introduced unprecedented change in IT systems and processes, which is saying a lot in a field that has been growing and changing non-stop since its inception. Today, organizations are using DevOps and Agile practices, coding in containers and microservices, and adopting Kubernetes at a record pace to help manage all these components. Even five years ago, the level of agility, speed, and flexibility the cloud-native stack enables was but a dream. Since Google first introduced Kubernetes in June 2014, enterprise adoption has been unparalleled — a staggering 78% of respondents to a 2020 survey conducted by the Cloud Native Computing Foundation cite running Kubernetes in production.

Like any other advance in technology, though, adding multiple layers of technology — with both containers and Kubernetes — adds a whole new set of security and stability concerns that organization must address, and few are comfortable they've met these concerns to date.

Organizations are on a steep learning curve — to understand both the infrastructure and security ramifications of running Kubernetes. Microsoft recently added to the conversation by developing an Attack Matrix for Kubernetes that provides a helpful framework for understanding the risks.

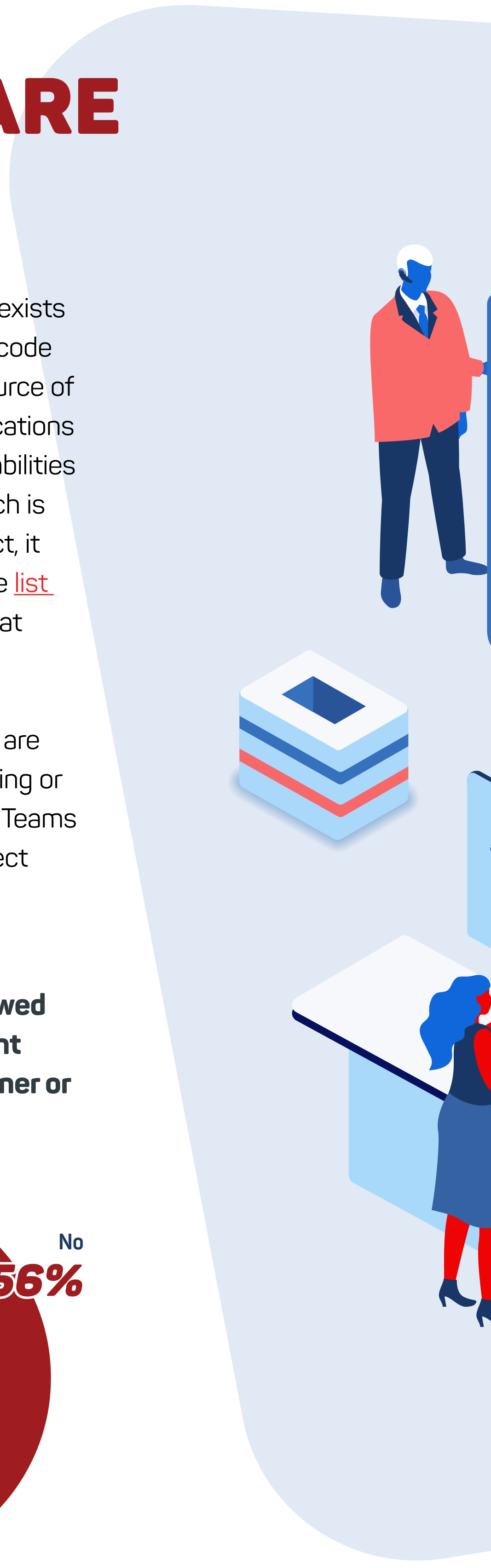
The industry has also seen plenty of evidence of security incidents. Tesla made headlines when its Kubernetes dashboard running in AWS was hacked to enable cryptomining, and Shopify educated the world on the risks of exposed Kubernetes metadata. If you count dangerous misconfigurations, nearly everyone who responded to a recent StackRox survey had experienced a security incident. And organizations are clearly wary of this risk — in the same StackRox survey, nearly half of respondents said their application deployment timelines were delayed due to container or Kubernetes security concerns.

Protecting your software assets has never been more important, but many companies are struggling to understand and mitigate the new sources of risk the cloud-native stack introduces.  This eBook offers a guide to protecting containerized applications throughout their life cycle. From securing your supply chain through to your workloads and even your Kubernetes infrastructure, this eBook will help you understand and apply the steps needed to protect your organization's mission-critical applications.

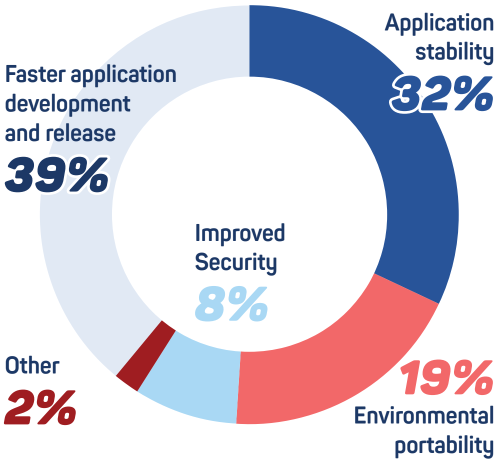
## SECURING THE SOFTWARE SUPPLY CHAIN

Modern software systems are immensely complex. Much of this complexity exists in underlying libraries and dependencies, and developers today are as much code assemblers as they are code producers. Organizations must consider the source of these software components as part of the overall security state of the applications they're building. It's critical that DevOps teams stay up to date on the vulnerabilities that exist in open source toolsets or in API services. Consider OpenSSH, which is used in applications at nearly every organization. As a security-related project, it is developed with processes designed to minimize vulnerabilities. Even so, the list of bugs on the project page shows that occasionally something comes up that impacts most organizations in the world.

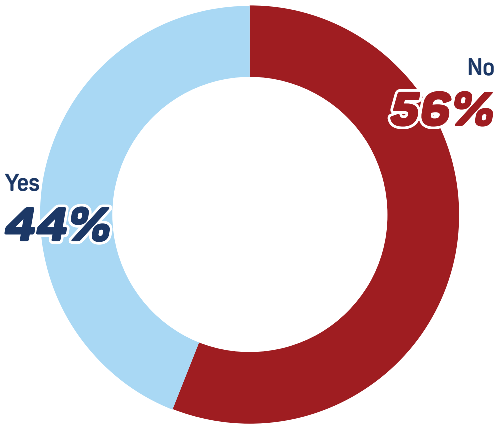
Organizations need a process in place to catch these issues as soon as they are known — whether these vulnerabilities appear in new images teams are building or previously unknown vulnerabilities surface that affect running deployments. Teams also need fast remediation once such issues become public — failing to protect systems at this most basic level only invites attacks.



**Of the following container and Kubernetes benefits, which has benefited your organization the most?**

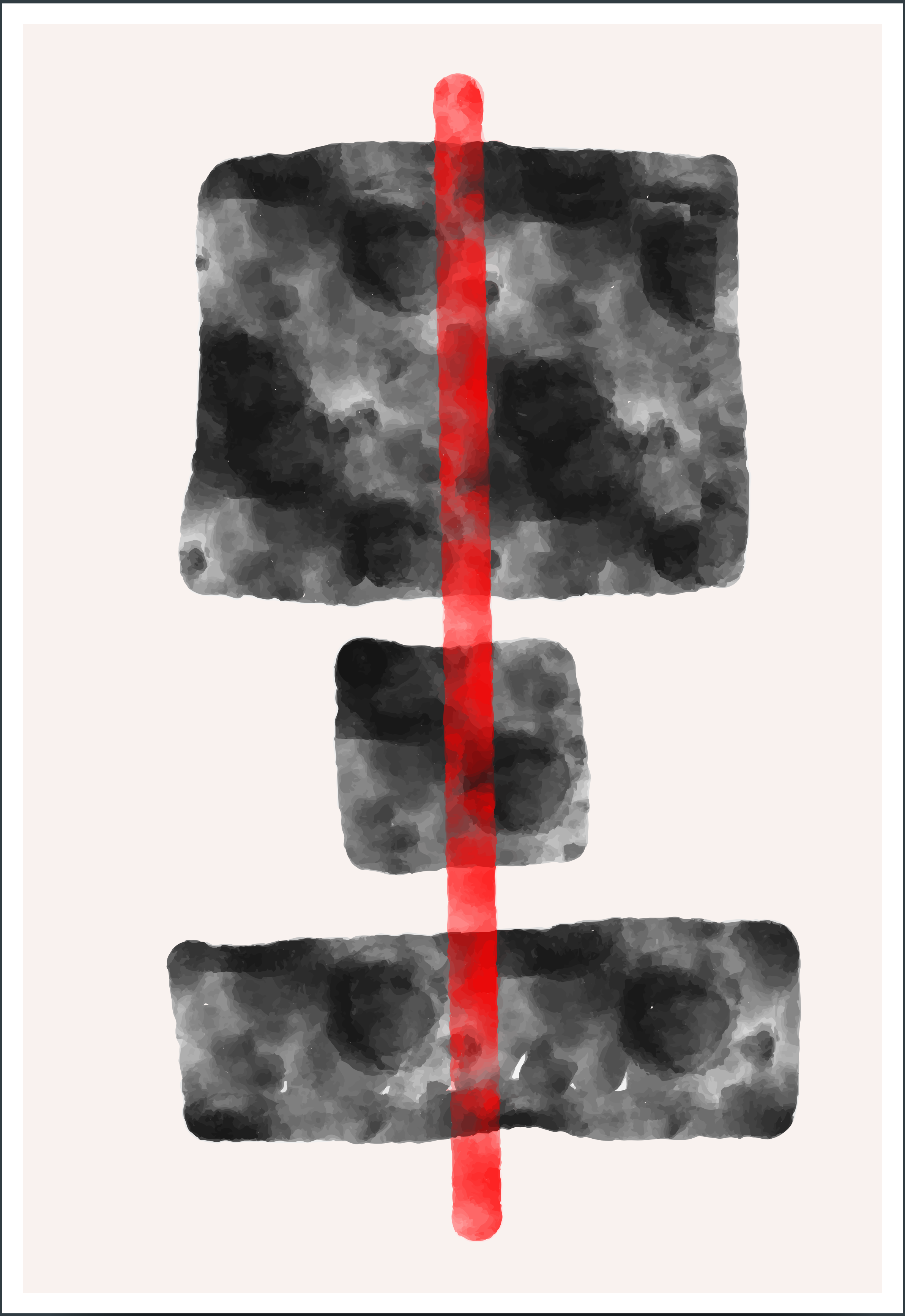


**Have you ever delayed or slowed down application development into production due to container or Kubernetes concerns?**



**Source: State of Container and Kubernetes Report**

The operational cost of not catching vulnerabilities until late in the life cycle is well-documented. The speed of change in software systems and the increasing complexity of environments make the threat larger than it has been historically. The StackRox survey found that 44% of respondents had delayed going to production because of security concerns or issues. Failing to deploy to production, or fixing issues in a rush while in production, have massive business impacts. What is needed is a proactive approach that can keep IT apprised of vulnerabilities and even take steps to help protect against them.



## Use secure, trusted, and minimalist base images

The base images that applications are based on will be running in several places across the infrastructure. Be they all internal, all cloud-hosted or mixed, these images will run in the dozens of copies in a given environment, and they must be trustworthy. A vulnerability or back door in the base image can open the entirety of the organization's environment to attackers. Ensure you use registries known to be trustworthy to obtain base images, and consider implementing a private registry.

One step you can take is to use the admission control capability in Kubernetes to restrict access to image repositories as one step to enforcing corporate image standards and ensuring the images running in a given environment are approved.

Another best practice is to keep images as slim as possible. The less unnecessary software in an image, the less that can go wrong and the smaller the attack surface for malicious actors to exploit. Start with the most stripped-down image possible, and then add only what is needed for the application to function. Distro-less images are a good starting point, as they are minimalist, containing only the application and its runtime dependencies.

From the distro-less base image, adding what is needed by the application — and only what is needed — is easy. This process will yield a final container configuration that is functional but and has the smallest attack surface possible. "Only what is needed" does not include debugging tools—those packages are indispensable in development but should never make their way into production. In general, if an application requires debugging tools in production, something is wrong with the application or the process that created it.

## Implement robust vulnerability scanning

While starting with trusted image sources and limiting software installed to a given container are good starting points, the world of security is not static, and new threats evolve over time. To counter this changing threat environment, final images should be scanned as a part of the CI/CD process. Scanning against the organization's security policy and failing any image that does not pass the scan cleanly enables a stronger and more dynamic security posture. As corporate security policy changes, scans will be updated also.

At a minimum, image scanning needs to look for vulnerabilities in the OS, along with any loadable modules used by applications running on the image. This approach gives a more complete picture of the security posture of the overall container image than scanning just the OS alone. While an OS vulnerability generally threatens a larger number of organizations, runtime libraries generally introduce more vulnerabilities.

Any external images brought into the organization to serve specific purposes should be scanned as well — even those from a trusted source. Image repositories are prime targets for attack, precisely because so many organizations download images from them for application development. By scanning them each time CI/CD is run, the risk that a recent compromise will result in attack vectors being installed on an organization's network is minimized.

To enable effective security operations, image scanning needs to be integrated with CI/CD tooling and processes. Image build policies will likely reside in a security policy engine, and failing one of the policies should trigger a failed build directly within the CI/CD system and should provide context and remediation steps. That way, developers learn immediately about a failure, in the pane of glass they're already in, and they should at the same time get an explanation of the rationale for the failure as well as the steps they must take to successfully complete the build. Such steps might include checking to see if an updated version of the image or library is available or changing a configuration such as not running as root.

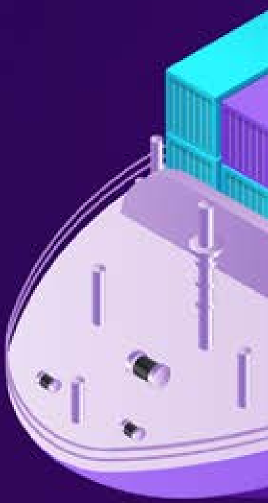
# SECURING THE CONTAINER WORKLOAD

Starting with known-good images and libraries is a solid step toward application security in a modern container environment, but it is just the first step. The next attack surfaces that need to be covered before the application can be considered secure are the configuration of Kubernetes deployment itself and having runtime policy enforcement enabled. Securing the deployment and management services of Kubernetes clusters is critical to security posture. Just as repositories are tempting targets for attackers because they provide a central location to plant attacks into many organizations, Kubernetes itself provides a central point for launching attacks into many containers, microservices, and applications. Once this portion of your infrastructure is hardened, you'll then need to configure runtime protections so that if someone is still able to penetrate your systems, the malicious activity will become immediately apparent and software can take automated actions to stop the breach.

The following sections will walk you through the highlights of hardening your Kubernetes environment directly — a game plan for Kubernetes security posture management, if you will.

## NAMESPACES

Namespaces in Kubernetes are much the same as namespaces in other IT contexts. With Kubernetes, the compelling reason to use them is that they help you separate logically distinct work. Many organizations use namespaces to allow the same cluster(s) to support dev, test and production, which means the only real difference in the three environments is which namespace is used. Applying Kubernetes role-based access control (RBAC) on a per-namespace level lets you restrict access to each environment to just those who need access to a given Kubernetes service. Kubernetes network policies are also namespace-scoped, so proper network segmentation will require you to use namespaces diligently.



## ROLE-BASED ACCESS CONTROL (RBAC)

Speaking of RBAC, a solid configuration of RBAC is critical to the security posture of an organization. Cluster-level access (via the use of cluster-admin role) should be limited only to those who need it and proper namespace usage will minimize which accounts need cluster-level access. Another area to look at is role aggregation. In Kubernetes, you can simplify privilege grants by combining new privileges into an existing role. Pay attention to how the new aggregated role impacts your security posture to make sure you're not creating overly privileged roles. Lastly, in order to streamline RBAC management, remove unused or inactive roles, and whenever possible, minimize duplicated role grants.

# NETWORK SEGMENTATION

Network segmentation is certainly not new, and Kubernetes implements the same types of segmentation over its internal network as is done in any other network. The key is that Kubernetes introduces a layer of complexity in the form of pods, so policies must be written with pods and namespaces in mind. That being said, most of what is needed to create network segmentation within Kubernetes should be familiar to networking and security staff.

The most crucial risk to understand about Kubernetes networking is that, by default, Kubernetes has no networking policies enabled — all resources can talk to all other resources. This approach suits the needs of developers, enabling them to get their services interworking as quickly as possible. However, it's easy for developers to forget to enable network segmentation when applications move into production.

## INGRESS POLICIES

Ingress policies control access to services on a container or containers. Ingress controllers process requests coming in and route them to the correct server/port/path. You can choose from a number of third-party ingress controllers to perform functions such as load balancing and security gateways.

An ingress policy tells the ingress controller what to do with traffic. Traffic aimed for a specific URL can be redirected to a server/port/path combination on the back end to service requests.

## EGRESS POLICIES

Egress policies control what connections going out to the Internet—more specifically, outside the pod—are allowed for a given container, pod, or group of containers. They reduce the chance of malware initiating connections from within a corporate network and granting unauthorized access. Egress controllers are typically used to block IP ranges, white-list acceptable servers for systems to open connections to, and control access to other services.

## BUILDING NETWORK POLICIES

Developing network policies can be as simple or as complex as your organization desires—it can be fine-grained and lock down exactly which servers can talk to specific locales outside the pod, or it can be coarse and allow general access from inside the network/VPN and be more restrictive outside the network.

While learning and developing network policies, reference a couple technical guides — the StackRox Guide to Network Policies and the Example Kubernetes Network Policies doc on GitHub.

## RUNTIME PRIVILEGES

Privilege escalation has been a concern since the early days of multi-user OSes. Any process that can run as root creates an attractive target for attackers. Containers increase concerns in this area because they share resources with the host OS, making a given container a possible gateway to the host.

In addition to other detailed advice that follows, adhere to this short list of simple best practices to move your organization toward an increased security posture:

- **Do not run application processes as root.**
- **Use a read-only root filesystem.**
- **Use the default (masked) /proc filesystem mount.**
- **Do not use the host network or process space.**
- **Use SELinux options for more fine-grained process controls.**
- **Give each application its own Kubernetes service account.**
- **Do not mount the service account credentials in a container if it does not need to access the Kubernetes API.**

Use pod security policies to enforce security requirements on an entire Kubernetes instance. The ability to control configurations such as privilege escalation for an entire pod makes the environment more secure and reduces management overhead. Pod security policy enforcement is managed through the PodSecurityPolicy Admission Controller, which allows infrastructure-wide enforcement of best practices by enabling the controller.

Anomaly detection and alerting at runtime are critical to a proactive security posture. Securing cloud-native technology depends in large part on shifting left with security and applying controls throughout the build and deploy phases of the container life cycle. But ultimately, organizations will also need runtime controls. To deliver safe, reliable uptime and protection against both malicious and accidental management issues, look for threat detection capabilities that include:

- **RUNTIME VISIBILITY: Activity occurring within the running application such as process execution, network connections, privilege escalation and storage activity within each container.**

- **ANOMALY DETECTION AND PREVENTION: Whitelisting of activities combined with ML-based behavioral modeling to monitor for and alert on anomalous activity.**

- **MANAGEMENT MONITORING: Use Kubernetes-native capabilities to detect and respond to runtime issues such as using kubectl scale to set instances to zero or killing a running pod from the host OS.**

## HARDENING THE KUBERNETES INFRASTRUCTURE

In addition to securing the software supply chain and your container workloads, you must also harden the Kubernetes infrastructure. A single misconfigured component can expose you to myriad threat vectors.

As with any software, keeping Kubernetes up to date is critical. As bugs and vulnerabilities are discovered, they are resolved in subsequent releases. Kubernetes is both more complex and more mission-critical than most software in the data center.

As a control point for dozens or hundreds of servers, Kubernetes is a tempting target to attackers. As such, it needs to be a priority for updates and maintenance. Consider some of the more severe vulnerabilities uncovered and fixed in subsequent releases.

The runC vulnerability, for example, hit just about every container management platform on the market, because runC, itself a "low level" container runtime, is used by container runtimes such as Docker to spawn and run containers.

This particular vulnerability in runC allowed an attacker to gain host-level code execution by breaking out of a running container, making it highly risky. Kubernetes code was not vulnerable, but it "inherited" the vulnerability since most Kubernetes installations make extensive use of runC. Kubernetes also inherited the fix: if you were using a managed Kubernetes service such as EKS, GKE, or AKS, patches were released to fix the vulnerability.

Other significant Kubernetes vulnerabilities have been publicized, including risks to the API Server and exposure of the dashboard and metadata — the runC vulnerability is but one to illustrate the criticality of protecting the Kubernetes infrastructure.

That said, simply updating Kubernetes is sometimes not enough. The billion laughs attack has been around since XML first became popular, and yet we seem to need to be reminded of its existence with each new self-referential file format. A billion laughs variant was found in the Kubernetes API Server that would result in a denial of service. Kubernetes patched the system and provided users with upgrades. At the time that the patch was released, if you created a new cluster with the patched upgrade, your system would be patched. However, if you upgraded a vulnerable cluster to the patched version, you still needed to modify your RBAC policy to adequately protect your systems. The moral of this story is to always check to make certain an upgrade actually resolves the issue and to take the recommended steps after upgrade to make your systems safe.

In addition, follow these best practices to keep key components of Kubernetes control plane and the worker nodes safe:

## KUBERNETES API SERVER

As one of the components of the Kubernetes control plane, Kubernetes API server (kube-apiserver) has a bewildering number of Configuration options. In most environments, those options will not be considered individually unless there is a specific need, which doesn't yield the most secure setup. As a first step, you must limit which users and service accounts have access to the API server by enabling and configuring Kubernetes RBAC. You must also make sure that:

- **All traffic between the API server and other infrastructure components (etcd, kubelet, etc.) are served over https (TLS).**
- **The API server is not serving requests on an insecure port.**
- **You are collecting and retaining API server audit logs.**

## KUBE-SCHEDULER

The Kubernetes scheduler (kube-scheduler) is another Kubernetes component within the control plan. kube-scheduler is responsible for selecting the node that a pod should run on, and carries its own deep configuration options. Thankfully, being part of the control plane means that if you're using a managed Kubernetes service provider such as GKE, EKS, or AKS, ensuring the scheduler is configured properly falls on the service provider, not the user. If you're self-managing your clusters, make sure you're scanning your environment against the Center for Internet Security (CIS) benchmarks for Kubernetes which contains several configuration checks for kube-scheduler.

## KUBE-CONTROLLER-MANAGER

The Kubernetes controller manager runs controller processes. Controllers monitor the state of a cluster via the apiserver to help ensure clusters remain in the desired state. Tasks such as detecting and responding to downed nodes are managed by controllers. The long list of kube-controller-manager options is also tough to weed through individually. If you're using a managed Kubernetes service, you will likely not have to worry about the configuration of kube-controller-manager as it falls under the responsibility of the service provider. If you're managing your own clusters, scan your environment against the CIS benchmarks to make sure you are passing the checks specific to kube-controller-manager.

## ETCD

As another control plane component, etcd stores key cluster information and locking it down is critical. Etcd is used internally by Kubernetes for data storage and exchange. As a result, any weakness in etcd configuration could leak information about the entirety of a Kubernetes installation. You must securely configure etcd and ensure all communication to it are protected using TLS encryption. As with the other control plane components, you will likely worry about Etcd configuration if you're managing your own clusters, in which case we advise you scan your environment against CIS checks specific to Etcd.

## CONFIGURATION FILES ON THE MASTER NODE

Should an attack obtain access to the local network, the goal must be to limit the damage. Locking down all critical configuration and PKI files on the master node is a critical step in protecting the system from attackers and is not a bad safety block against destructive accidental changes. You need to make sure that you have locked down the ownership and permission of these files, including the API pod spec file, controller manager pod spec file, scheduler pod spec file, etcd pod spec file, etc.

## KUBELET

Kubelets are referred to as the "node agents" in the Kubernetes documentation, but "node manager" is closer to the truth. As middle management, Kubelet is given a set of pods that are supposed to be running on its node (PodSpecs), and its job is to make certain the containers in those PodSpecs are "running and healthy." Make sure you've configured the kubelet config file such that it doesn't allow anonymous/unauthenticated requests to be served by the kubelet server. The default kubelet setting is to accept unauthenticated API requests.

## WORKER NODE CONFIGURATION FILES

Worker nodes get the job done. They are the equivalent of the "payload" in IT systems.  That makes them dangerous for two reasons: A node with several containers on it is more tempting than a single asset, and organizations typically have dozens of them, perhaps spread across architectures. You must lock down the ownership and permission of your worker node configuration files including kubelet service file, kubelet.conf file, proxy kubeconfig file, etc.

## DELIVERING ON THE TOP KUBERNETES SECURITY USE CASES

In the end, every organization desires a secure environment. The amount of time and effort dedicated to pursuing security varies from organization to organization, but the desire to be safe is ubiquitous. The fast pace, new technology, and complicated stack involved in cloud-native applications compounds the challenge.

Follow the steps outlined previously to protect organization's Kubernetes installation from attacks — and mistakes. They will help you lock down the most important resources in your deployments. They'll also help you protect against one of the greatest weaknesses of complex systems — human error.

As you move through your Kubernetes journey, your security needs will change, so look for tooling that can grow with you. The top Kubernetes security use cases have parallels in previous waves of infrastructure, but Kubernetes has unique needs to be met. To address the challenge, you should:

- **Look for deep, multi-faceted visibility into the Kubernetes system and issues, including logging, alerting, and dashboards.**
- **Focus on solutions that offer the ability to manage vulnerabilities proactively. In a DevOps world, feedback on vulnerabilities must come into the systems teams are already using, with appropriate rationale and remediation info so they can react quickly and intelligently.**
- **Network segmentation is different inside Kubernetes than in almost any other environment. Choose security tooling that automatically constructs policies that allow just the needed communications and generates the correlating YAML files to make setting policy easier.**
- **Configuration management in a highly complex, highly distributed system is not only difficult but also error-prone. Choose a tool that automatically identifies incorrect settings, prevents drift from initial configurations, and will highlight risks such as dangerous configuration changes.**
- **Risk-profiling can help alert an organization to issues before they become incidents. A tool that can offer a quick view on how important a given threat is and how the security posture of a given environment rates is necessary.**
- **Compliance has become a standard requirement in almost every industry. Choose a tool that can help meet the compliance requirements the organization must fulfill, with both visual insights about status and detailed logs auditors need.**
- **Catching threats as they develop has long been recognized as more effective than trying to clean up after a breach. Catching them can be difficult in simple environments. The distributed systems in play with Kubernetes installations are not simple, increasing the challenge. You'll need a tool to assist with runtime monitoring and threat detection.**

## KUBERNETES IS DIFFERENT

In case you missed it, Kubernetes introduces layers of complexity into an already complex environment. A security tool for a cloud-native environment must be Kubernetes-native and offer Kubernetes-specific tooling.

Kubernetes-native security refers to leveraging the context of and native controls in Kubernetes to protect the infrastructure. The rich context of Kubernetes' declarative data will yield meaningful insights into your environment that will help assess risk, speed troubleshooting, and enable faster analysis. Leveraging the native controls means security policies are embedded directly in your infrastructure, enabling security that's built in, not bolted on. Plus, that approach ensures developers, operations teams, and security staff — no matter where they sit in the organization — all share a common source of truth, speak the same language, and leverage the same set of policies.

Specifically, an ideal Kubernetes security tool should:

- **Reduce time and cost for teams via a reduced learning curve, familiar framework and a "configure once, run everywhere" model. Overworked teams are doing more all of the time; making it easy for them to leverage systems they're already using as part of their security solution is imperative.**

- **Offer better visibility into configuration, compliance, and workload isolation so that teams can see all of the important aspects of Kubernetes.**

- **Discover critical vulnerabilities and threat vectors such as Kube-specific configuration issues, vulnerabilities and ingress/egress communications configuration and monitoring.**

- **Minimize operational risk by eliminating operational conflict, such as where security tooling takes actions not reflected in Kubernetes, and enabling scalable enforcement. Rapid development/deployment and security can be at odds, which can create insecure situations and increase risk. While a Kubernetes security and configuration tool cannot resolve all such stresses, it can get all the teams using common infrastructure and policies, reducing that conflict.**

Build for success, maximizing both security and configuration stability, then move toward automation. In the end, securing Kubernetes infrastructure will enable more fundamental business protection securing any given application. Thankfully, StackRox has you covered.