Kubernetes and cloud applications let complex systems run reliably in unreliable environments. In doing so, however, they can expose an organization to a multitude of security issues. While some organizations do a great job of securing their infrastructure, many fall into one of two categories:

- Those who are unaware that their clusters are insecure.
- Those who know their clusters are insecure, but have no idea where to start securing them.

Fortunately, organizations can easily mitigate most Kubernetes security issues by taking action in four main areas:

- **Secure the cluster**, its underlying infrastructure, and administrative tools.
- **Control access** by keeping out unauthorized people and workloads.
- **Provide a secure software development process** that lets developers build and improve software quickly while consuming only appropriate components, and that prevents erroneous or malicious execution of untrusted workloads on your clusters.
- **Build a secure DevOps culture** by adopting coding best-practices, architectural patterns, standardization, and tools that make your code run more efficiently and securely on Kubernetes, while helping to identify and mitigate vulnerabilities and reduce the attack surface as much as possible.
Secure the cluster

A properly-hardened specification for Kubernetes environments — and the ability to deploy and lifecycle-manage self-similar environments on a wide range of infrastructures — is a powerful security enabler. Establishing such a standard, hardened Kubernetes cluster configuration and deployment specification delivers many benefits, including:

- **One single source of infrastructure-as-code truth** for deploying, scaling, managing and updating consistent clusters anywhere. This saves DevOps time and enables continual improvement of the now-portable DevOps infra-as-codebase. Lost cycles are minimized with these extraneous requirements and dependencies removed.

- **One single source of automation truth** for deploying, scaling, managing, and updating applications written to exploit this standardized, consistent infrastructure.

- **Independence from infrastructure dependencies**, conferring greater agility, and infrastructure mobility, and limiting cloud provider lock-in.

Hardening Kubernetes begins with selecting a Kubernetes platform that implements the Kubernetes project’s best-practices checklist for cluster security. This entails, among other things:

- **Hardening nodes**: Whether deploying on bare-metal or virtual machines, node hardening details changes that need to be made (and maintained) to harden selected host operating systems and ensure appropriate, non-disruptive, and timely patching.

- **Securing etcd**: Kubernetes uses the etcd key-value store to persist information about the current state of the cluster and all of its objects. An attacker who gains access to it can wreak havoc, stealing sensitive data and causing significant destruction.

- **Securing the Kubernetes API**: As the gateway into your cluster, the Kubernetes API is an essential security chokepoint, and securing it involves a combination of certificates and careful configuration. In some cases, your installer will handle most of the heavy lifting, but you should be aware of what is required.

- **Securing the Kubernetes dashboard**: While it’s important to have a way to see what’s going on in your cluster, the upstream Kubernetes Dashboard does not meet the necessary security standards most organizations require for production use. You will need to either take additional steps to mitigate problems or use a different tool, such as the open source Lens Kubernetes IDE.

- **Runtime security**: Within the cluster, applications run in pods, which themselves run containers. While containerd (with and without Docker) is the most common container runtime, multiple implementations of the Container Runtime Interface (CRI) exist, and most provide opportunities for attackers to use malicious containers to attack your host system if the container runtime isn’t properly secured.

- **East-West traffic**: Transport Layer Security (TLS) involves certificates to ensure that a hacker cannot, for example, bypass the Kubernetes API and attack the Kubernetes Scheduler directly in order to compromise your cluster.

- **Securing Ingress**: Finally, while you will, of course, have legitimate traffic entering your cluster, you must ensure that only that legitimate traffic gets past your firewall or other perimeter protection.
• **CIS Benchmarks:** There are well-defined, community-produced best practices for several major components of a container environment, including Kubernetes, Docker, and several flavors of Linux. Given the rapid pace of change with Kubernetes, the benchmarks themselves change quickly, so having a process and tools that run these benchmarks against your environment will help ensure you’re implementing industry-accepted security settings.

### Control access

Once you’ve hardened your cluster, you still need to prevent unauthorized access to it. This is a vast topic unto itself, but some of the basic topics you need to consider include:

- **Infrastructure as code and immutable clusters:** One of the most important ways to keep your cluster in good working order — and to keep it secure — is to make sure that node configuration doesn’t drift over time. In that respect, you must treat your infrastructure as you treat every other part of your process: as code. If changes need to be made, you need to make them at the configuration file level and re-create the nodes rather than tinkering on the nodes themselves.

- **Admission controllers:** Admission controllers are cluster-wide checks and restrictions that control how requests to the cluster are treated. These controllers can enforce policies or prevent actions such as pulling container images from unknown registries.

- **Pod Security Policy:** While many threats come from outside your cluster, there is always the danger that workloads can intentionally, or even unintentionally, cause problems. Pod Security Policy enables you to control access to features such as the ability to run in a “privileged” context, and to control that access at the cluster level.

- **Audit logging:** Kubernetes not only provides the ability to log every request and action, it also enables you to control, at a granular level, which of those actions you want to log. Think carefully about where you’re vulnerable and ensure that you’re watching those areas and checking the logs on a regular basis.

- **Penetration testing:** You’ll never know how secure your system really is until someone tries to break into it. Fortunately, it doesn’t have to be someone who intends to do you harm; penetration testing uses hackers who are on your side to look for vulnerabilities and report them back to you so you can fix them.

- **Limiting resource usage:** One way to prevent, or at least mitigate, a Denial of Service attack is to ensure that no single user or workload can monopolize all of your cluster’s resources. You can do that by limiting the amount of resources, such as memory, a container can request.

- **Content Trust policies:** Ultimately your cluster is only as secure as the containers that run on it. Content trust policies enable you to choose whether and when to allow unsigned images to be used on your cluster, or whether to require that all containers be instantiated from images that have been signed — and even signed by a particular person or process, such as a security scan.
Provide a secure software development process

Creating, updating, integrating, and lifecycle-managing containerized workloads for Kubernetes is considerably more flexible, and in some ways, fundamentally easier, than creating and managing conventional applications. Most organizations transitioning to container-based development do so partly in order to gain speed — to deliver better software, faster, by using container technology to package software components and make them independent of one another, making them easy to consume, combine, extend, and run in self-scaling fleets for improved resilience and performance.

The need to secure containerized applications, however, can throw sand in the gears. How do you know whether the convenient base images, downloaded by your developers, contain malware, vulnerabilities, or protected intellectual property? How do you ensure that all the elements of an application have been sourced, tested, curated, and approved appropriately, before pushing to production? How do you ensure that your home-grown applications comply with governance requirements?

Automation is the best solution. Best practice demands creating a secure software supply chain with a CI/CD automation envelope ensuring that:

- Applications can only be built from fully-curated, trusted, authorized components.
- Components and applications undergo complete testing in self-similar environments from developer desktop through QA, staging, and production. Steps are never skipped and requirements for human review and deliberate promotion are enforced in every stage.
- Risky errors (such as deploying QA code to production) are deterministically prevented at points of execution. Containers are only allowed to run in appropriate contexts.

Tools that play roles in such a supply chain include:

- **Static code analysis**: True static code analysis solutions work before containers are built, identifying security issues and vulnerabilities in code and dependencies before packaging. This kind of analytic tool is often closely bound to particular languages and development environments, such as Node.js.

- **Private container registries**: One cornerstone of enterprise container security and related policy is the private registry as a source for authorized base images, component images, and built container images.

- **Mirroring of vendor-certified components**: Popular and important components (such as vendor-certified database container images) are made available in public registries (such as Docker Hub). For the sake of speed and convenience, it’s important that developers are able to use this kind of component. Equally important, however, is that no component should be made usable by developers without appropriate analysis and approval. Being able to mirror specific public registry images into a private registry helps make the ingestion process more seamless.

- **Automated image scanning and vulnerability flagging**: Once imported, automated container image scanning can analyze candidate container images to expose issues and vulnerabilities. The best scanning tools draw vulnerability signatures from multiple security and vulnerability databases, providing comprehensive protection.

- **Image signing and execution prevention/content trust**: Cryptographic signing of binary container images ensures that policy steps have been followed and approvals granted before containers can be consumed by later stages in the supply chain or executed in inappropriate contexts. Container engine ‘content trust’ capability prevents execution of unsigned or wrongly-signed containers in inappropriate environments.
• **DevOps tool integration to CI/CD and other components:** Adopting a Kubernetes IDE or similar integration point enables connection to code editors, repositories and registries, CI/CD front-ends, metrics harnesses, test automation, security scanning and other tools to provide visibility into and facilitate management of all the moving parts in an enterprise-grade dev/test/production environment.

• **CI/CD scripting:** Scripting pulls all of these features together, automating the secure supply chain of container build, scan, store, sign, as well as the promote process for application containers and operations automation codebases. A huge range of solutions are available, some following a classic coding model, others helping to automate the pipeline-creation process.

A secure software supply chain consumes authorized base images and properly-versioned code to produce candidate images that are pushed to a trusted registry, automatically scanned, signed, and manually or automatically promoted for use in QA / test, staging, or production. Content trust is implemented on container runtimes to ensure that only properly-signed images can be utilized. This illustration depicts Docker Enterprise Kubernetes Service / Universal Control Plane as an example of a secured Kubernetes cluster model, and Docker Trusted Registry as its associated private image registry. Docker Trusted Registry implements Notary in extended form under the name Docker Content Trust (DCT).
Build a secure DevOps culture

Once you’ve hardened your infrastructure, controlled who can access it (and what they can do), and built a secure supply chain for delivering applications, then comes the challenge of using all of this properly, every day. This is, by far, the most important and riskiest aspect of maintaining security over time — the operational security (a.k.a. ‘opsec’) part of the exercise — because it depends on people.

Establishing firm operational security around DevOps requires two levels of action and awareness:

• Establishing standards and adopting tools that further enhance application security — mostly by making it more ‘automatic’ and independent of people.

• Developing secure programming and operations best practices that are universally understood and embraced by your teams.

In the first category, you’ll need to think about some pretty deep stuff, such as:

• **The principle of least privilege**: People and application components should be given the least far-ranging, most-specific possible permissions required to do their jobs. For example, instead of admitting “all traffic” to a Kubernetes infrastructure node, you should lock down all but the (still quite a large number) of ports it requires. Least privilege applications and architectures are always more secure, on balance, as well as more auditable. Implementing least privilege complements creation of ‘zero trust’ architectures, where every element is treated as an untrusted, possibly-compromised resource, and where every operation is verified before being allowed to proceed.

• **Limiting the attack surface**: Apply ‘least privilege’ thinking everywhere to limit the attack surface presented by applications, infrastructure, operators, and users. If a server isn’t exposed to the Internet, it can’t be compromised (at least not directly) by internet attacks. If a developer doesn’t need access to critical infrastructure keys (because these are stored in a secrets repository - see below), these will not be a concern if the developer leaves their work laptop on a train.

• **Securing Kubernetes networking**: Kubernetes, its configured container network provider, ingress controller(s), components, and other plumbing collaborate to dynamically manage multiple virtual networks over the physical (and/or virtual) networks and protocols connecting nodes. All this needs to be secured, and ideally monitored, from the ground up — a big and complicated set of jobs within a tall stack of functionality. Best practices dictate treating every networked entity with ‘zero trust,’ applying the principle of least privilege.

• **Pod-to-pod communication**: Communications among containers running in separate pods and namespaces is the backbone of modern applications. Best practice dictates a range of methods for enforcing policies about where and how pods can execute, scale, discover needed services, authenticate, and communicate, and how their exchanges can be protected from threats.

• **Access control - authentication and authorization**: In principle, all communications within a Kubernetes cluster are mutually authenticated and authorized to comply with policy.

• **Secrets management**: Kubernetes contains a framework enabling encryption, safe storage, and use of secrets within a strict identity and policy framework.
• **Namespace management**: Namespaces are used to constrain access and visibility of resources.

• **Architecture standardization**: Embracing standardization wherever possible can be an effective vehicle for limiting the attack surface on known entities. This might include the use of a Service Mesh to enable self discovery of load balancing, security policies, and even proper circuit breaker patterning practices to eliminate the potential for injecting new attack points.

• **Encryption**: Encryption and cryptographic signing of binary container images, traffic, secrets, and other entities is a critical part of preventing untrusted software from executing on your cluster, controlling the context(s) in which given workloads can run, and enforcing other aspects of policy.

• **Open Policy Agent (OPA)**: OPA lets you model complex policies such as privileges, permissions, and requirements, and apply them to control traffic admission to critical resources such as the Kubernetes REST API, Docker Engine and its API, and other critical system components.

**Where to go from here**

Building out a secure container orchestration facility and then using it to modernize applications and streamline software development in your organization is, admittedly, a pretty tall order. Readers who are in the process of evaluating Kubernetes and related offerings and deciding on a Kubernetes strategy may want to consider a few common-sense tips:

• **Choose an enterprise Kubernetes solution**: Hardening and security engineering around the Kubernetes stack is complicated and meticulous work, so it makes sense to prefer Kubernetes solutions where most of the hard work is done for you, where vendors stand firmly behind the quality of their offerings (with firm SLAs, for example), and where support is prompt, expert, and methodical.

• **Ensure freedom of choice**: Most enterprises now venturing into Kubernetes territory already have well-established IT standards, preferred operating systems, mature on-premises virtualization and private cloud setups, and public cloud strategies. Switching horses — for example, buying into a Kubernetes stack that obliges use of a particular Linux distribution, or that runs on only one public cloud — limits choice and adds costs. So does following the ostensible path of least resistance, and, for example, investing in the added-cost, semi-proprietary Kubernetes solution offered by your preferred public cloud provider, just because you now run virtual machines there. A better idea is to explore Kubernetes solutions that run anywhere, on any operating system and/or infrastructure.

• **Consistent clusters = portable policy and DevOps**: Operating one flavor of Kubernetes on developer desktops, perhaps a second flavor on your private cloud, and then possibly additional flavor(s) on public clouds, edge devices, and elsewhere isn’t efficient, and tends to work against security. You end up needing to maintain different policy and automation frameworks for different environments. You may end up dealing with bugs, dependencies, and issues that might not matter in one environment (i.e., "it works on my desktop!") but do matter in others. This said, you should adopt a complete Kubernetes solution that runs anywhere, letting you create consistent environments and administer them under one automation and policy regime — one focused on serving the needs of your business for security and agility.

• **Batteries included**: Many Kubernetes offerings are just that: a core offering that then needs fitting-out with solutions for functions such as ingress and networking, and for security that integrates with these choices. A better idea is to adopt a complete solution with community-preferred components already integrated and hardened (along with the freedom to integrate your own desired set of components, when required). This lets you begin your Kubernetes journey by focusing on building a secure software supply chain and operations automation that accelerates your business and reduces your risk.
About Mirantis Kubernetes Engine

Building out a secure container orchestration facility and then using it to modernize applications and streamline software development in your organization is, admittedly, a pretty tall order. Readers who are in the process of evaluating Kubernetes and related offerings and deciding on a Kubernetes strategy may want to consider a few common-sense tips:

Mirantis Kubernetes Engine (formerly Docker Enterprise Universal Control Plane) is an enterprise-grade Kubernetes container orchestration solution engineered to be “secure by default”, implementing all Kubernetes Community security recommendations, out of the box. Mirantis Kubernetes Engine uses Mirantis Container Runtime (formerly Docker Engine). Mirantis Container Runtime implements further security features such as optional FIPS-140-2 encryption and Mirantis Content Trust (formerly Docker Content Trust), an extended implementation of Docker Notary that enforces image-signing policies and prevents execution of unauthorized container workloads.

Mirantis Kubernetes Engine also ships with best-of-breed solutions in place for Kubernetes container networking and security (open-source Project Calico) and ingress (Istio), and with other Kubernetes features (internal DNS, for example) fully operational, so everything “just works.”

Calico Enterprise builds on top of open source Calico to provide additional higher-level features and capabilities for securing the cluster environment:

- Control access to endpoints outside the cluster
- Extend existing enterprise security controls to Kubernetes
- Extend firewalls to Kubernetes
- Zero trust network security
- Intrusion detection
- Cloud microsegmentation
- Self-service network security

Compliance controls and reporting

Mirantis Kubernetes Engine users can also install Mirantis Secure Registry (formerly Docker Trusted Registry) on any Mirantis Kubernetes Engine-managed Linux worker node, providing a complete private image registry solution with built-in best of breed image vulnerability scanning and other security features.

All Mirantis Kubernetes Engine and Mirantis Secure Registry features are independent of the underlying operating system, cloud provider, and infrastructure. As such, they enable freedom of operating system choice and let you quickly deploy, scale, and manage consistent Mirantis Kubernetes Engine clusters anywhere, from developer desktops to bare metal and VMs, on-premise and public clouds, as well as on edge devices. Consistent clusters means you can implement one set of security policy models that works everywhere, and use one set of dev and ops automation to build, test, and run clusters and applications wherever this is cost- and operationally-efficient. You get consistent security, accelerated development, streamlined operations, and reduced cost and risk.

Please reach out to Mirantis at www.mirantis.com/contact to request a free demonstration of Mirantis Kubernetes Engine and its numerous security features, and see our secure supply chain in operation.