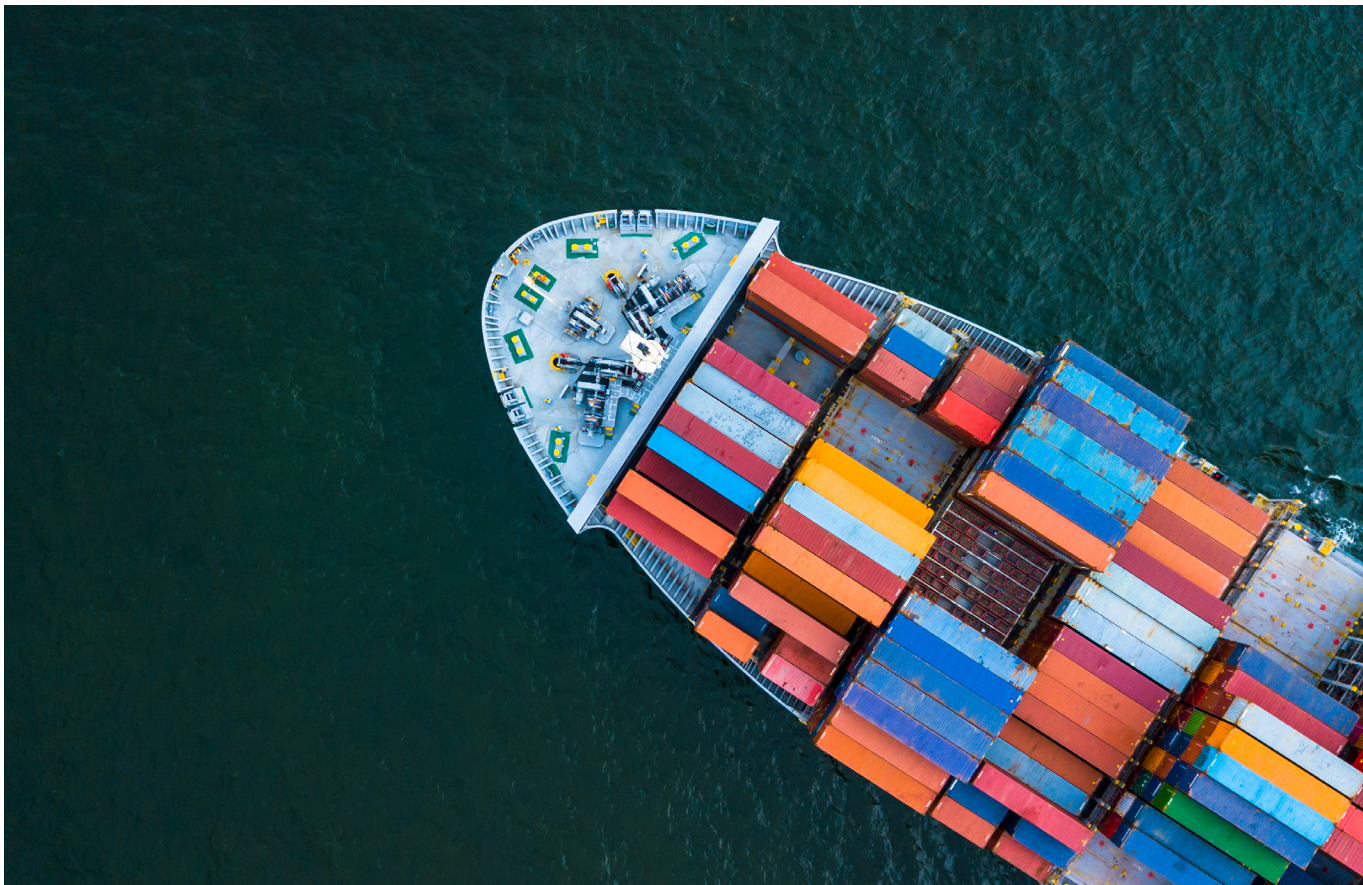
Google Cloud

July 2020

Why Container Security Matters to your Business

Understanding the container security concepts that impact your organization

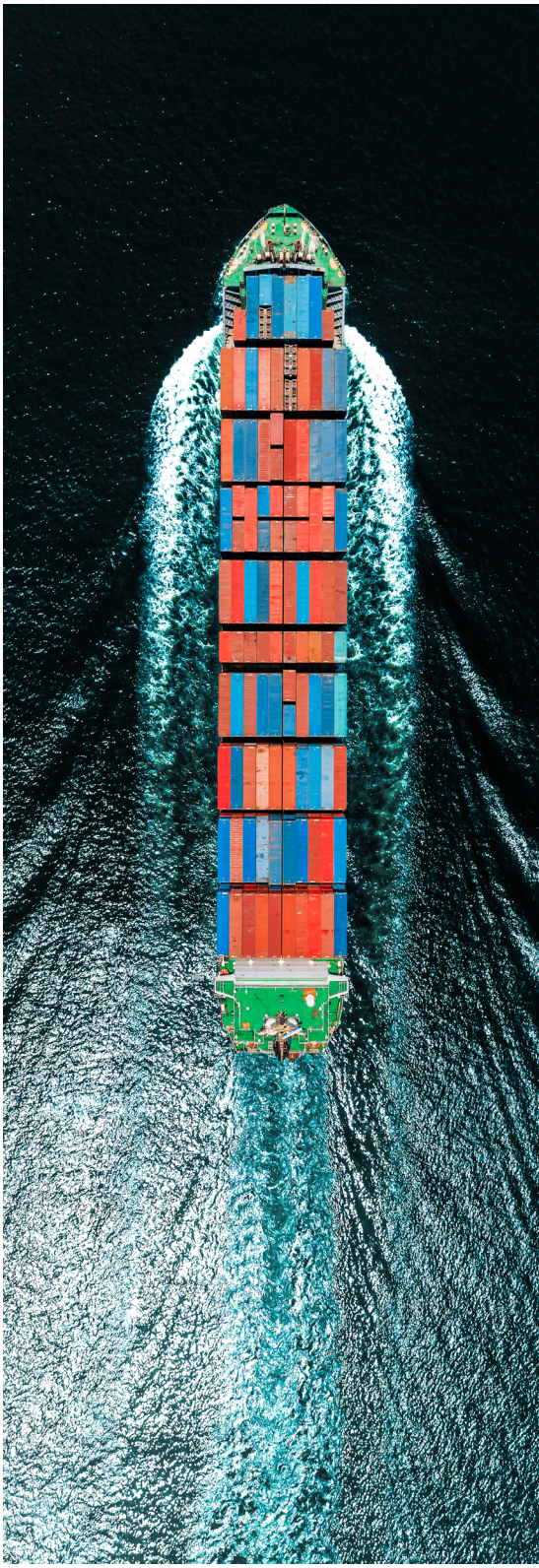


Google Cloud

Table of Contents

Introduction

Why container security matters to your business



All around you, whether you know it or not, are containerized applications. They provide Wifi in your local cafe, process your purchase at the grocery store, and serve up your online banking. You might send your doctor a message through a mobile app and then play your favorite mobile game, all running in containers.

With real workloads come real consequences. Your business is your data, and running your business-critical data in containers has elevated this once-buzzword to something business leaders are forced to make decisions about. If you're evaluating a cloud provider and your security lead says, "Cloud A offers managed images; for Cloud B, we'd be on our own," how important is that? Should it influence your decision?

The goal of this book is to teach you the fundamentals of container security so when it comes time to make business decisions, you'll have the context you need to help keep your business safe.

What actually is a container?

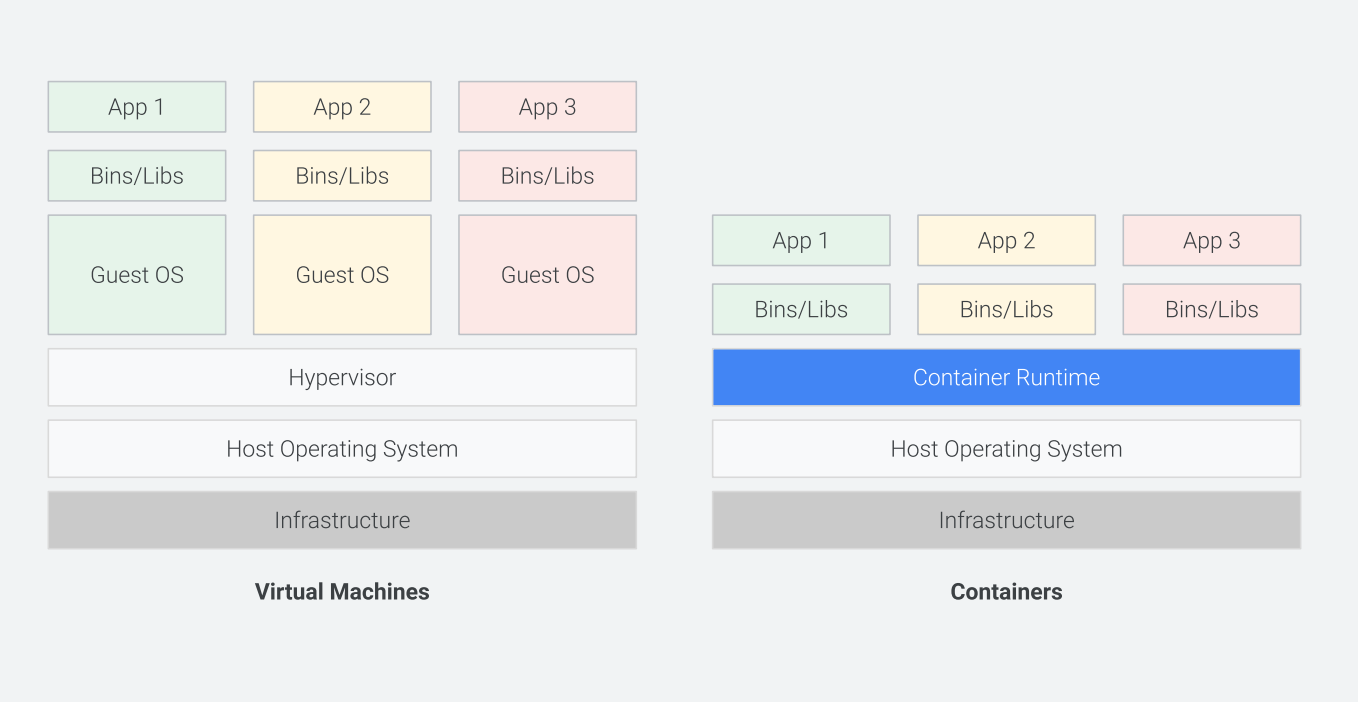
A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of portability -- or, more precisely, the lack thereof. Applications are built and tested using specific language, runtime, package, and library versions. When Developer A hands off work to Developer B, who merges it for testing and into production, inconsistencies between these environments can cause the application to break. (Operating system versions, for example, can be hard to keep in sync between development and production; OS and application upgrades risk accidentally pushing incompatible changes.) Also, if you want to run multiple applications on the same host, these applications may require incompatible OS versions.

Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure.

But while a container can be considered a boundary, it's a boundary with limitations. Just like VMs, containers can still be compromised through various attacks, or left vulnerable through misconfigurations or unpatched components. In "Running Kubernetes Securely," we'll talk more about threats to containers, but a compromised or misconfigured container can lead to unauthorized access to your workloads and your compute resources, and even the potential to recreate your application (and its data) somewhere else.

The TL;DR

- Don't let the shipping container imagery fool you; containers are not a special security boundary
- Containers use primitives of the Linux kernel (cgroups, namespaces) to isolate processes in an environment
- A "container image" is your application and its dependencies, and uses a "base image" as the basis for the container image
- Container registries host your container images. It's important that you be able to trust your base and container images, and that you use a private, trusted registry.



Containers

Core kernel container mechanisms and privilege restrictions

Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests).

The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process.

Along with cgroups and namespaces, you can also use a Linux Security Module (LSM) to configure a container's capabilities. Two LSMs common in containers are AppArmor and SELinux. Both deny undesirable default capabilities, like the ability to write to the proc filesystem. Another kernel feature, Secure Computing with filters (**seccomp**) is a system call filter which prohibits certain syscalls from being made to the kernel, which reduces the kernel's attack surface.

Cgroups, namespaces, LSMs and seccomp are the parameters which define what the process can do when it's running, and what creates our containerized environment. But there's much more to containers. With these tools we can isolate our processes on one host, but our end goal is to package our applications to run in any environment, and in order to accomplish that we'll need a container runtime and image.

Container runtimes, images and registries

A container runtime is responsible for executing the container's specifications. Google Cloud Developer Advocate Ian Lewis explores container runtimes' mechanisms and functionalities in this four part series. For now, we can think of the runtime as what configures the container isolation primitives and runs a process inside them. Different runtimes have different security capabilities, particularly in the area of container isolation (covered in "Understanding Isolation").

The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.

A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.

Container images are static, which is part of what makes them a security benefit; when you need to make a change to a deployed container, you should build and deploy a new image rather than changing the running container itself. Deploying your containers with read-only filesystems in order to prevent intruders from changing files is one way you can use containers' inherent architecture properties as a security tool.

Container images are stored in **container registries**. You can pull your container image from a common repository like Docker Hub or a private repository like Google Container Registry. Either way, it's

critical that you be able to trust this image, since it's going in your environment. You can easily imagine the risks of pulling from a widely available but unknown source on the public internet (we'll discuss this further in "Supply Chain Security").

You don't need to understand how to configure SELinux and namespaces yourself, but you should know that containers alone are not a hardened security boundary; there are many necessary components for running them -- like base images, container images, and registries -- each of which comes with its own set of security considerations.

Supply Chain Security

Adopting containers and container orchestration tools like Kubernetes can sound intimidating. But in fact, you can use containers to improve your overall security posture.

You can use containers to improve your overall security posture:

1. Containers are short-lived and frequently re-deployed; **you can constantly be patching**.
2. Containers are intentionally immutable; a modified container is a **built-in security alert**.
3. Good security defaults are one line changes; **setting secure configurations is easy**.
4. With isolation technologies, **you can increase security without adding resources**.

Containers give you a software supply chain

With a monolithic application running on a virtual machine, developers usually make changes by remotely logging in to the machine or pushing code changes manually. This is not only hard to debug, but it's also a very informal process; the next time the developer needs to make a change, they can just remotely log in to the VM again to debug, patch, update, restart, or otherwise adjust the app. That's not a great security story, and it's really tough on the ops team, because they don't know what exactly is running anymore.

With containers, things are a bit different. Containers have a defined development pipeline, also known as a software supply chain. You write your code and can ensure that it meets your requirements for build, test, scan, and whatever else, before you deploy it. Further, code can be intercepted at any step in the chain if it doesn't meet your requirements.

The TL;DR

There are properties inherent to containers that can be security advantages:

- Containers have a defined development pipeline
- Containers are not patched live; patches can be rolled out as part of your regular pipeline
- Container images are meant to be immutable; if a new vulnerability is disclosed, you know if that image is affected

Containers let you patch continuously, automatically

Even today, many security attacks that occur in the wild, especially for containers, are 'drive-by' attacks in which bad actors look for deployments with known vulnerabilities that they can exploit. And those vulnerabilities are rarely freshly disclosed vulnerabilities that haven't been patched—we're talking about problems that have been left unfixed for years. Like wearing sunscreen, scanning for and patching vulnerabilities is one of those boring best practices you really should be doing (may we recommend Container Registry Vulnerability Scanner?).

Even today, many security attacks that occur in the wild, especially for containers, are 'drive-by' attacks in which bad actors look for deployments with known vulnerabilities that they can exploit.

But patching containers is different than patching VMs. Containers are meant to be immutable, meaning they don't change once they're deployed; instead of remotely logging in to the machine, you rebuild, and redeploy, the whole image. This happens quite often, since containers are short-lived; Sysdig estimates that 95% of containers live for less than a week.

But wait…isn't that's really often? If you look at traditional patch management, Patch Tuesday comes just once a month. Maybe if you're extra busy, you might also have to manage some weekly patch sets. You might still need Sunday 2 a.m. maintenance windows to apply your patches (and there's a poor soul who has to stay up for this), but there's simply not enough time in the day or coffee in the world to deal with deployments that only live one week!

Here's the thing though. With containers, you don't patch live containers, you patch the images in your container registry. By doing so, the fully patched container image can be rolled out or rolled back as one unit, so the patch rollout process becomes the same as your (obviously very frequent) code rollout process, complete with monitoring, canarying, and testing. This way, your patch rolls out using your normal, predictable process. An alternative (though less preferable because it happens on an unpredictable schedule) is to let the rollout happen ad hoc. Then the next time your container dies, Kubernetes spins up another one to compensate, and any patches you've applied will naturally roll out to your infrastructure. Depending on your containers' lifespan, you should be fully patched in a matter of days.

Containers mean you can actually tell if you're affected by a new vulnerability

Since containers are immutable, they give you content addressability—they're stored in such a way that you're able to retrieve a container based on its contents. This means you actually know what's running in your environment -- for example, which images you've deployed.

What does this mean for security? Suppose that when you scan your image, it's fully patched, so you deploy it. Later on, a new vulnerability is found. Rather than scanning your production clusters directly, you can just check your registry to see which versions are susceptible.

This also simplifies your patch management by decoupling decisions and processes about when to patch from actual patching. Instead of trying to answer, "Is my container patched?" your security team can ask, "Is my container image patched?" Then your ops team can ask, "Is my (patched) image running?" This also lets you answer the inevitable question from your CISO: "Are we affected?"

Containers made Google more secure, and more reliable

Thankfully, you don't have to take our word for it. Google's infrastructure is containerized, based on our Borg container orchestration system (the inspiration for Kubernetes), and we use it to deploy services and security patches on billions of containers per week.

By now it should be obvious how that's possible: by patching continuously, and deploying patched containers. In the event of a disruptive incident, for example hardware maintenance or a critical security patch, we use something called live migration. For GCP workloads, live migration is basically a blue/green deployment, where the new workload is deployed alongside the existing workload, and a load balancer gradually moves traffic over until it's fully

handled by the new instance. This means you can effectively patch a running containerized workload, with no downtime, without the user noticing. This is what let us patch Heartbleed in 2014 with no downtime, and more recently Spectre/Meltdown.

In short, using containers allows you to easily patch your infrastructure, with no downtime, and do so quickly in the event that you're affected by a newly discovered vulnerability. Better yet, you can automate all the boring patching stuff you never liked doing anyway. If you're serious about your production system's security, your infrastructure team can use containers to make patching your production environment safer, faster and easier.

But what if you want to run multiple containers? Enter container orchestration platforms. Next we'll focus on the predominant platform, **Kubernetes**, which helps you run containers at scale.

Running Kubernetes Securely

Upstream Kubernetes, the open source version that you get from the GitHub repository, wasn't designed to be a locked-down security environment out of the box. Rather, its defaults solve for flexibility and usability; it is designed to be very extensible, and much of its security relies on using those extension points to integrate with other systems like identity and authorization.

And that's okay! It means Kubernetes can fit lots of use cases. But it also means that you can't assume that upstream Kubernetes' defaults are correct for you. If you want to deploy Kubernetes with a "security first" mindset, there are several core components to keep in mind.

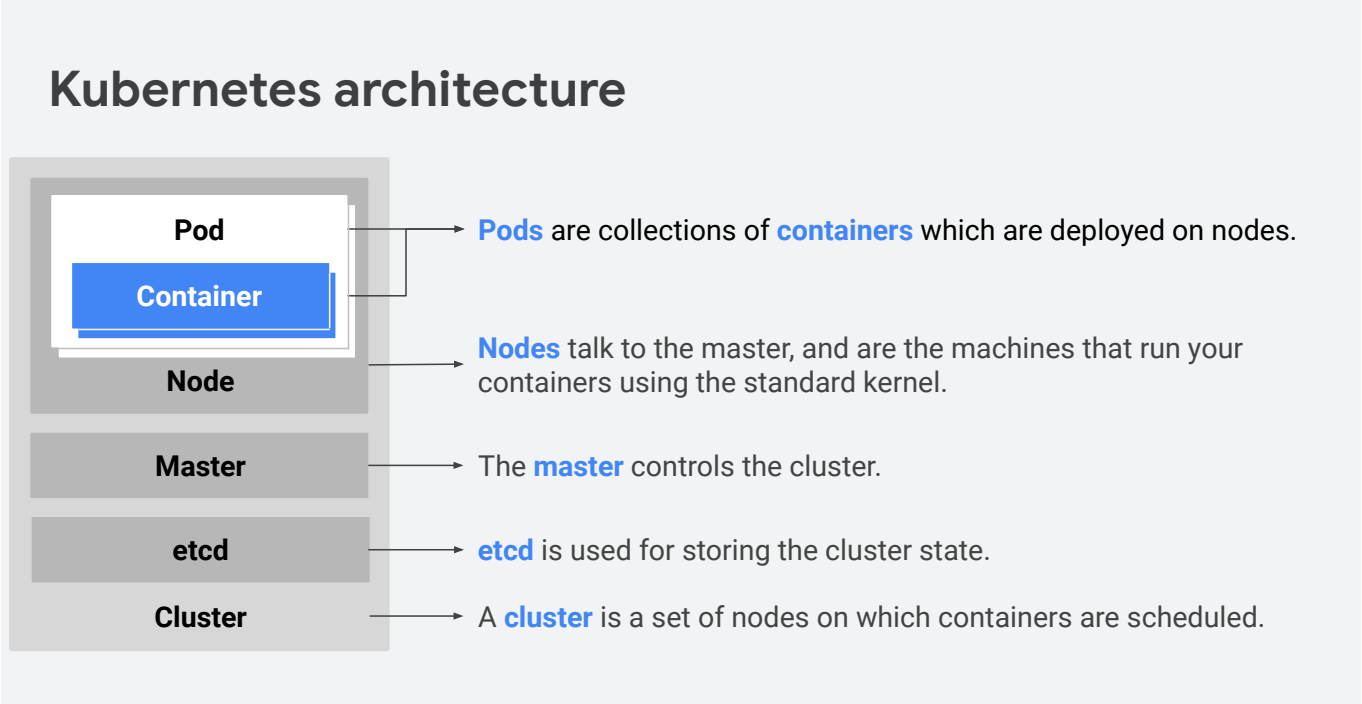
Just as we established a mental model for what a container is (and isn't), it's important to do the same for Kubernetes. Kubernetes has many components, and from an attacker's point of view, each one comes with a different reward if compromised. Understanding this will help you understand what your security team can do to protect your Kubernetes instance and your applications.

Kubernetes from the attacker's view

In Kubernetes, a container runs in a **pod**, which in turn runs on a **node**, a virtual or physical machine. The nodes running pods are called **worker nodes**, which contain the container runtime, have their own operating system, and are managed by the **Kubernetes control plane**. Finally, **etcd** is a key-value store that keeps the state of the control plane. All of these pieces together make up your **cluster**.

The TL;DR

- There are multiple components to Kubernetes that should be protected
- Reasons for attacking containers include abusing compute resources, accessing workload data, or gaining access to application code
- The defaults in upstream Kubernetes should not be assumed to provide adequate protection based on your use case



Kubernetes architecture

Why would an attacker want to compromise these components?

Let's start with the container itself. A common reason for attacking containers today is to abuse compute resources, for example, for cryptocurrency mining. But attacking the container can also offer access to customer or workload data.

Attackers could also try to escape the container in order to get at the node. Compromised Kubernetes nodes give malicious actors numerous attack opportunities, including a chance to propagate to other nodes in the cluster and also gain persistent access to valuable user code, compute and/or data. "Container escape" is a type of privilege escalation attack that uses the fact that containers share a host kernel. If a malicious actor compromises a container and receives privileged access, they could potentially access information running in the other containers. We'll cover more about this attack and how to prevent it in "Understanding Isolation."

The Kubernetes master controls your cluster. An attacker that can compromise the master can control the environment, including the ability to take it offline. And a compromised etcd can mean the ability to modify or destroy the cluster, steal secrets and credentials, or gain enough information about the application it's running to go recreate it somewhere else.

The good news: you can proactively harden your Kubernetes deployment to increase your container security. If you're using a managed service, your provider may have implemented some of these for you (in Google Kubernetes Engine (GKE), you can see what we offer to protect these Kubernetes components). There are also tools, like the CIS Benchmark, that will help you compare where your Kubernetes deployment is now, and where you want to be.

The CIS Kubernetes Benchmark
Gauging your security

The Center for Internet Security (CIS) Kubernetes Benchmark is a set of recommendations for configuring Kubernetes to support a strong security posture. The CIS Benchmark is tied to a specific Kubernetes release; the first Benchmark was for Kubernetes 1.6, at the time of writing, the latest is for the 1.15 Kubernetes release. These CIS Benchmarks are meant to be widely applicable to many Kubernetes distributions.

Each CIS Benchmark is community-contributed, and written by experts in a variety of disciplines in order to reflect multiple perspectives, including security consulting, development, compliance and operations. These experts are volunteering their time, so if you meet a CIS editor, thank them for developing this resource!

CIS Benchmark recommendations are defined as either Level 1 -- a fundamental security configuration with an immediate benefit -- or Level 2, an extension of a Level 1 recommendation that, while improving security, may inhibit performance or harm compatibility, and thus requires evaluation before implementation.

The CIS Kubernetes Benchmark is written for the open source Kubernetes distribution and intended to be as universally applicable across distributions as possible. So you shouldn't read it as a step-by-step required configuration manual, and it isn't fully applicable to hosted distributions like GKE. "With any security standard, it is very important for companies to consider their threat model," says Rory McCune, Principal Consultant, NCC Group PLC, and CIS Benchmark editor. "Some security controls will always have trade-offs in terms of performance impact or usability, so organizations should not treat methodologies like the CIS Benchmark as an 'all or nothing' exercise, but instead should consider each recommendation and whether it makes sense in the context of their environment."

The TL;DR

- The CIS Kubernetes Benchmark contains a set of recommendations for a strong Kubernetes security posture
- It is not written to be a must-do checklist, but rather recommendations that you should evaluate
- If you are using a managed service, some recommendations in the CIS Benchmark may not apply

If you're using a managed service, not all items on the CIS Benchmark are your responsibility, nor are they directly exposed to or configurable by you, as they fall under your provider's purview. In GKE, for example, etcd -- the key value store that keeps the state of your cluster -- is part of what Google hardens per GKE's shared responsibility model. If you were to run a tool like kube-bench, an open source tool that checks for the CIS Benchmark's recommended configurations, you wouldn't be able to inspect certain elements, such as the control plane, and might see false Benchmark item "FAILs" for other items due to that limitation.

How to apply the CIS Benchmark to your deployment

How you will want to apply the CIS Benchmark will depend on how you consume Kubernetes and what other CIS Benchmarks you also plan to use.

If you're running open source Kubernetes

If you're running Kubernetes straight from upstream, you can see how you're doing against the CIS Benchmark by using kube-bench, a line-by-line list of recommendations, each with a PASS/FAIL. The output includes the corresponding recommendation number in the Benchmark guide. If you fail item 1.1.7, for example, you can simply check 1.1.7 in the guide for their recommendation and implementation.

"It is impossible to inspect the master nodes of managed clusters, e.g. GKE, EKS and AKS, using kube-bench as one does not have access to such nodes, although it is still possible to use kube-bench to check worker node configuration in these environments."

If you're using a managed service

As the kube-bench docs call out, "It is impossible to inspect the master nodes of managed clusters, e.g. GKE, EKS and AKS, using kube-bench as one does not have access to such nodes, although it is still possible to use kube-bench to check worker node configuration in these environments."

One benefit of a managed service is that, depending on your provider's shared responsibility model, the security of certain components (the master nodes, for example) isn't your responsibility. You can still use the CIS Benchmark and kube-bench to test your security posture, but the inspection limitations means you shouldn't expect to see an "all pass" status. For components you don't control (and might not be able to test directly), your service provider should be able to tell you what measures they've taken to harden them -- and indeed, what measures they take to protect your workloads. (For GKE, see Control plane security and Cluster trust).

One benefit of a managed service is that, depending on your provider's shared responsibility model, the security of certain components (the master nodes, for example) isn't your responsibility.

For components that you're responsible for protecting, your provider should be able to offer further distribution-specific steps you can take to improve your security, such as, for example, the GKE hardening guide.

Finally: there's an ongoing effort to develop distribution-specific benchmarks based on the general CIS Kubernetes Benchmark. If you're interested, you can follow progress and contribute directly in the CIS WorkBench tool.

Combining multiple CIS Benchmarks

Some tools attempt to analyze Kubernetes nodes against multiple CIS Benchmarks (e.g. Linux, Docker, and Kubernetes) and combine the results. Because those benchmarks weren't designed to be combined and applied in a Kubernetes environment, This often results in confusing and potentially contradictory advice. For example, it doesn't make much sense to invest deeply in a recommendation from the CIS Docker Benchmark, like configuring

Docker authorization plugins, when Kubernetes will be handling the authorization and scheduling of containers. In fact, it makes the most sense to remove much of the unused Docker functionality and run using the GKE container node image as recommended in the GKE hardening guide.

Similarly, the CIS Linux Benchmark is designed for a different serving environment that doesn't translate completely to Kubernetes and hosted cloud computing. For example, some advice centers around not being able to modify security sensitive configuration without a reboot. This makes sense in traditional server or desktop deployments where reboots would be fairly visible and unusual. Contrast that with a Kubernetes cluster whose nodes are largely disposable and may be created and destroyed regularly in response to application needs. A hosted cloud computing environment may freeze certain configurations and leave others open for users to configure It might also be undesirable to have the cloud provider freeze configuration in some cases if customers need configurability to meet their security needs.

If you're using tools that combine benchmarks in this way, you should consider each recommendation on its merits and decide if it makes sense and is applicable to your environment, rather than treating it as established best practice.

The shared responsibility model in GKE

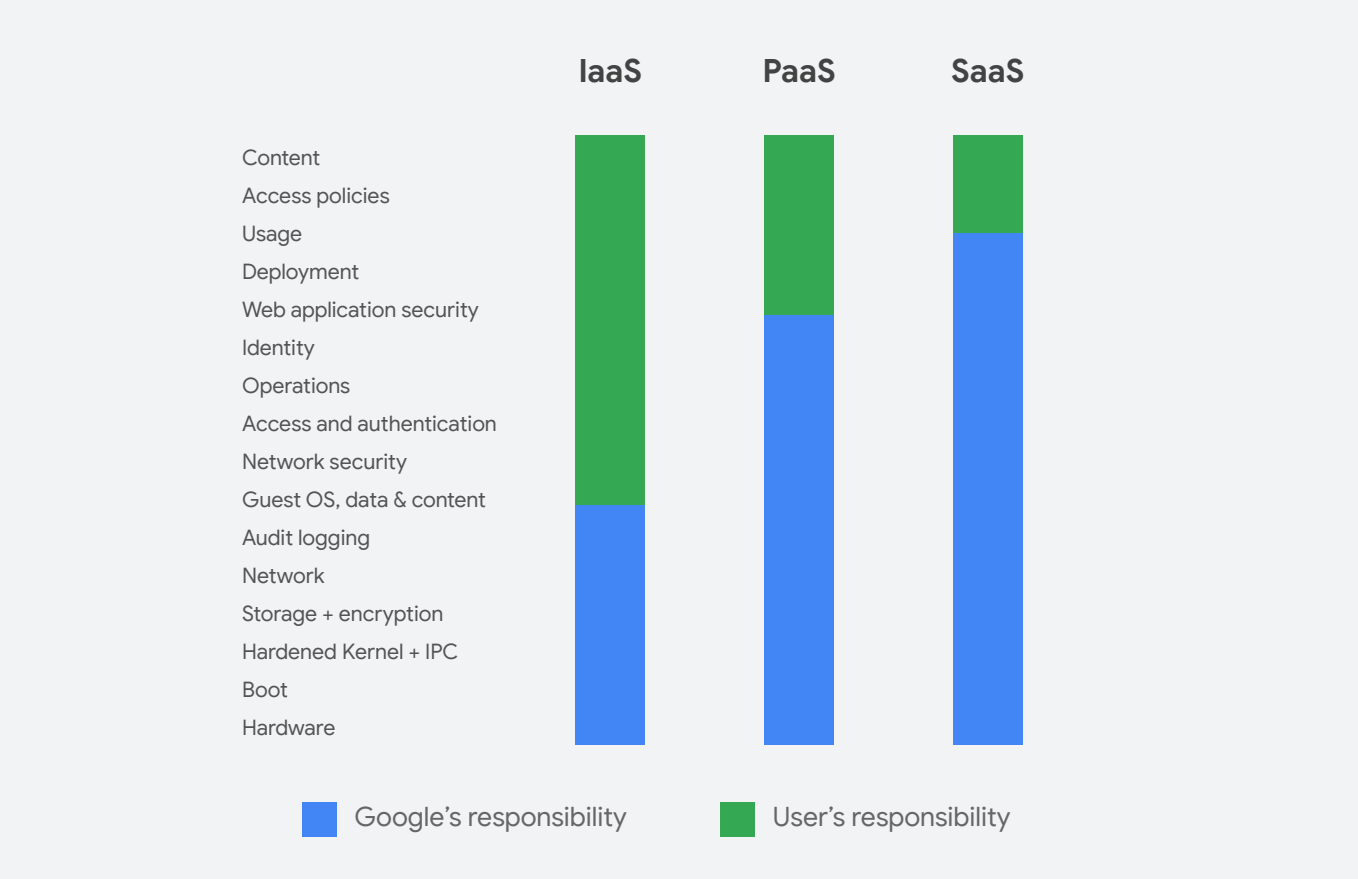
Security in the cloud is a responsibility shared between the cloud provider and its customer. Google Cloud is committed to doing our part to protect the underlying infrastructure, like encryption at rest by default, and provide capabilities you can use to protect your workloads, like access controls in Cloud Identity and Access Management (IAM). As newer infrastructure models emerge, though, it's not always easy to figure out what exactly you and your provider are each responsible for. Here we'll we clarify what Google Kubernetes Engine (GKE) does and doesn't do—and where to look for resources to lock down the rest.

Google Cloud's shared responsibility model

The shared responsibility model depends on the workload—the more we manage, the more we can protect. This starts at the bottom of the stack and moves upward, from the infrastructure as a service (IaaS) layer, where only the hardware, storage, and network are the provider's responsibility, up to software as a service (SaaS), where almost everything except the content and its access are up to the provider. (For a deep dive check out the Google Infrastructure Security Design Overview whitepaper). Platform as a service (PaaS) layers like GKE fall somewhere in the middle; hence the ambiguity that arises.

The TL;DR

- The security of cloud services is a shared responsibility between the provider and the user
- It is important your security and incident response teams understand what they are responsible for hardening and protecting
- It is important that your team understand how your provider communicates in the event of an incident affecting a component they're responsible for



Shared responsibility on Google Cloud

For GKE, at a high level, we are responsible for protecting:

- The underlying infrastructure, including hardware, firmware, kernel, OS, storage, network, and more. This includes encrypting data at rest by default, encrypting data in transit, using custom-designed hardware, laying private network cables, protecting data centers from physical access, and following secure software development practices.

- The nodes' operating system, such as Container-Optimized OS (COS) or Ubuntu. GKE promptly makes all available patches to these images. If you have auto-upgrade enabled, this will happen automatically. This is the base layer of your container—it's not the same as the operating system running in your containers.

- The Kubernetes distribution. GKE provides the latest upstream versions of Kubernetes, and supports several minor versions. Providing updates to these, including patches, is our responsibility.

- The control plane. In GKE, we manage the control plane, which includes the master VMs, the API server and other components running on those VMs, as well as the etcd database. This includes upgrades and patching, scaling, and repairs, all backed by an SLO.

- Google Cloud integrations, for IAM, Operations, Cloud Key Management Service, Security Command Center, etc. These enable controls available for IaaS workloads across Google Cloud on GKE as well.

Now, here's what you are responsible for protecting:

- The nodes that run your workloads. You are responsible for any extra software installed on the nodes, or configuration changes made to the default. You're also responsible for keeping your nodes updated. We provide hardened VM images and configurations by default, manage the containers that are necessary to run GKE, and provide patches for your OS. You're just responsible for upgrading. If you use node auto-upgrade, it moves the responsibility of upgrading these nodes back to us.

- The workloads themselves, including your application code, Dockerfiles, container images, data, RBAC/IAM policy, and containers and pods that you're running. This means leveraging GKE features and other Google Cloud products to help protect your containers.

Hardening the control plane is Google's responsibility

Google is responsible for improving the security of the control plane — the component that manages how Kubernetes communicates with the cluster, and applies the user's desired state. The control plane includes the master VM, API server, scheduler, controller manager,

cluster CA, root-of-trust key material, IAM authenticator and authorizer, audit logging configuration, etcd, and various other controllers. All of your control plane components run on Compute Engine instances that we own and operate. These instances are single tenant, meaning each instance runs the control plane and its components for only one customer. (You can learn more about GKE control plane security here.)

We make changes to the control plane to further harden these components on an ongoing basis—as attacks occur in the wild, when vulnerabilities are announced, or when new patches are available. For example, we updated clusters to use RBAC rather than ABAC by default, and locked down and eventually disabled the Kubernetes dashboard.

How we respond to vulnerabilities depends on which component the vulnerability is found in:

- The kernel or an operating system: We apply the patch to affected components, including obtaining and applying the patch to the host images for Kubernetes, COS and Ubuntu. We automatically upgrade the master VMs, but you are responsible for upgrading nodes. Spectre/Meltdown and L1TF are examples of such vulnerabilities.

- Kubernetes: With Googlers on the Kubernetes Product Security Team, we often help develop and test patches for Kubernetes vulnerabilities when they're discovered. Since GKE is an official distribution, we receive the patch as part of the Private Distributors' List. We're responsible for rolling out these changes to the master VMs, but you are responsible for upgrading your nodes. Take a look at these security bulletins for the latest examples of such vulnerabilities, CVE-2017-1002101, CVE-2017-1002102, and CVE-2018-1002105.

- Components used in Kubernetes Engine's default configuration, like Calico components for Network Policy, or etcd. We don't control the open-source projects used in GKE,

but we select open-source projects that have demonstrated robust security practices and that take security seriously. For these projects, we may receive a patch from upstream Kubernetes, a partner, or the distributor list of another open-source project. We're responsible for rolling out these changes, and/or notifying you if there is action required. TTA-2018-001 is an example of such a vulnerability that we patched automatically.

- GKE: If a vulnerability is discovered in GKE, for example through our Vulnerability Reward Program, we are responsible for developing and applying the fix.

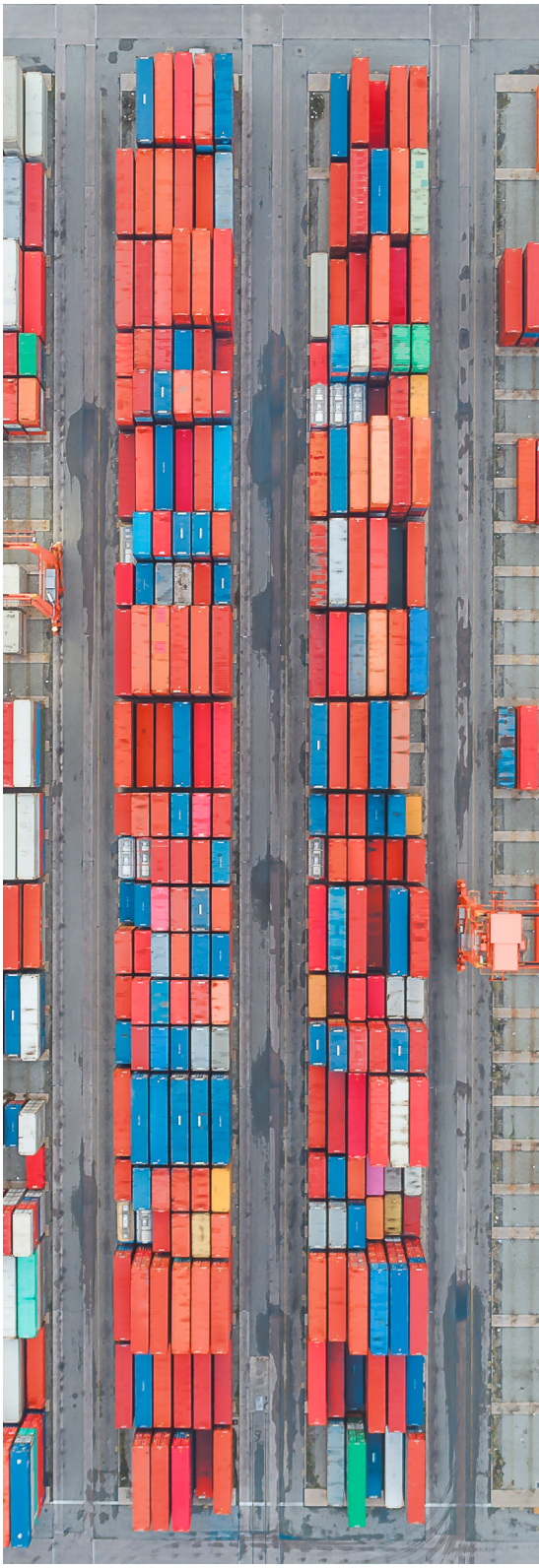
In all of these cases, we make these patches available as part of general GKE releases (patch releases and bug fixes) as soon as possible given the level of risk, embargo time, and any other contextual factors.In all of these cases, we make these patches available as part of general GKE releases (patch releases and bug fixes) as soon as possible given the level of risk, embargo time, and any other contextual factors.

We do most of the hard work to protect nodes, but it's your responsibility to upgrade and reap the benefits

Your worker nodes in Kubernetes Engine consist of a few different surfaces that need to be protected, including the node OS, the container runtime, Kubernetes components like the kubelet and kube-proxy, and Google system containers for monitoring and logging. We're responsible for developing and releasing patches for these components, but you're responsible for upgrading your system to apply these patches.

Kubernetes components like kube-proxy and kube-dns, and Google-specific add-ons to provide logging, monitoring, and other services run in separate containers. We're responsible for these containers' control plane compatibility, scalability, upgrade testing, as well as security configurations. If these need to be patched, it's your responsibility to upgrade to apply these patches.

To ease patch deployment, you can use node auto-upgrade. Node auto-upgrade applies updates to nodes on a regular basis, including updates to the operating system and Kubernetes components from the latest stable version. This includes security patches. If you're using node auto-upgrade, upgrading becomes Google's responsibility. Notably, if a patch contains a critical fix and can be rolled out before the public vulnerability announcement without breaking embargo, your GKE environment will be upgraded before the vulnerability is even announced.



Protecting workloads is still your responsibility

What we've been talking about so far is the underlying infrastructure that runs your workload, but you, of course, are still responsible for application security and other protections to your workload itself.

You're also responsible for the Kubernetes configurations that pertain to your workloads. This includes setting up a NetworkPolicy to restrict pod to pod traffic and using a PodSecurityPolicy to restrict pod capabilities. For an up-to-date list of the best practices we recommend to protect your clusters, including node configurations, see Hardening your cluster's security.

If there's a vulnerability in your container image or application, it is also your responsibility to patch it. But there are tools you can use to help:

- Google-managed base images, which are regularly patched for known vulnerabilities.

- Container Registry vulnerability scanning to analyze your container images and packages for potential known vulnerabilities.
- Cloud Security Scanner to help you detect common application vulnerabilities.

Incident response in GKE

So what if you've done your part, we've done ours, and your cluster is still attacked?

Well, our first advice is, don't panic! Google Cloud takes the security of our infrastructure—including where user workloads run—very seriously, and we have documented processes for incident response. Our security team's job is to protect Google Cloud from potential attacks and protect the components outlined above. When it comes to the pieces for which you're responsible, Google Cloud already has a range of container security partners integrated with the Cloud Security Command Center. The alerting and remediation you can receive from Cloud Security Command Center and its partner integrations can help you respond to issues in the pieces you're responsible for protecting.

If you're responding to an incident, you can leverage Operations to help you reduce your time to incident mitigation, refer to sample queries for Kubernetes audit logs, and check out the Cloud Forensics 101 talk from Next '18 to learn more about conducting forensics.

What's the tl;dr of GKE security? For GKE, we're responsible for protecting the control plane, which includes your master VM, etcd, and controllers; and you're responsible for protecting your worker nodes, including deploying patches to the OS, runtime and Kubernetes components, and of course securing your own workload.

An easy way to do your part is to:

Google Cloud takes the security of our infrastructure—including where user workloads run—very seriously, and we have documented processes for incident response.

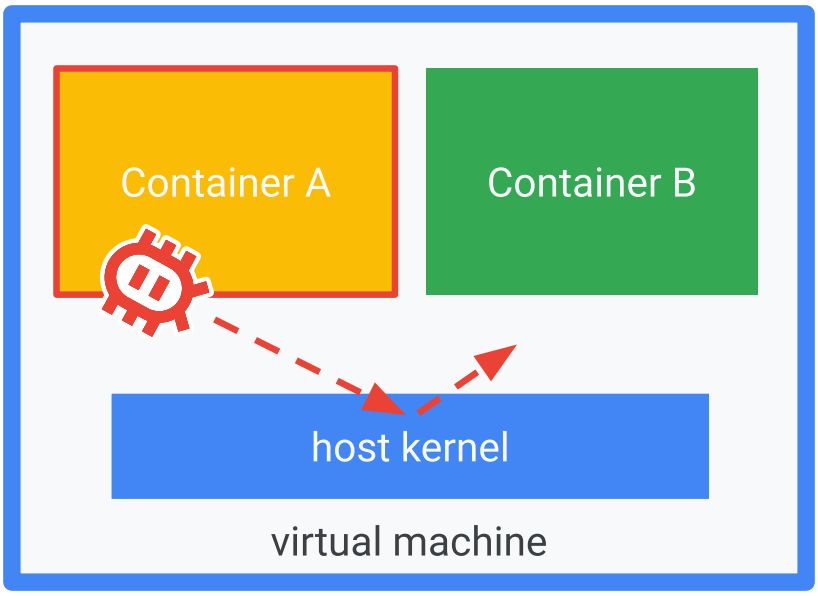
1. Use node-autoupgrade
2. Protect your workload from common image and application vulnerabilities, and
3. Follow the Google Kubernetes Engine hardening guide.

If you follow those three steps, together we can build GKE environments that are resilient to attacks and vulnerabilities, to deliver great uptime and performance.

Understanding Container Isolation

One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by “bin packing” multiple workloads onto each server. As such, the architecture of containers means that they’re deployed with multiple containers sharing the same kernel.

Unfortunately, while sharing a kernel between workloads enables higher density and efficiency, it also means that a single kernel bug can compromise the entire host. Container escapes are a type of attack that follow a specific pattern: a bad actor attacks one container, escalates their privileges, gains access to the host, then to a second container and its contents.



Containers don’t contain

The TL;DR

- The shared kernel in containers architecture introduces the threat of “container escape” attacks
- Emerging isolation open source projects, like gVisor and Kata Containers, provide defense-in-depth to prevent these attacks
- Consider a managed version of a project if your team cannot independently manage an open source tool

Lateral movement between containers remains a top-of-mind a threat for users running sensitive workloads, providing SaaS services, or otherwise running untrusted code. These users need to make a determination about the risk of their code and what sort of tradeoffs they're willing to tolerate, in terms of performance overhead or compatibility, in order to increase that code's isolation and their environment's security. (You can't always have your cake and eat it too!) For example, code written by your own teams, where you already have security controls in place for code review, and which accesses public data sets, would probably be classified as lower risk than code that either consumes sensitive data or was supplied by an external user, or both. In the latter case, an organization might decide to take a certain performance tax in order to increase the security measures they apply to that code.

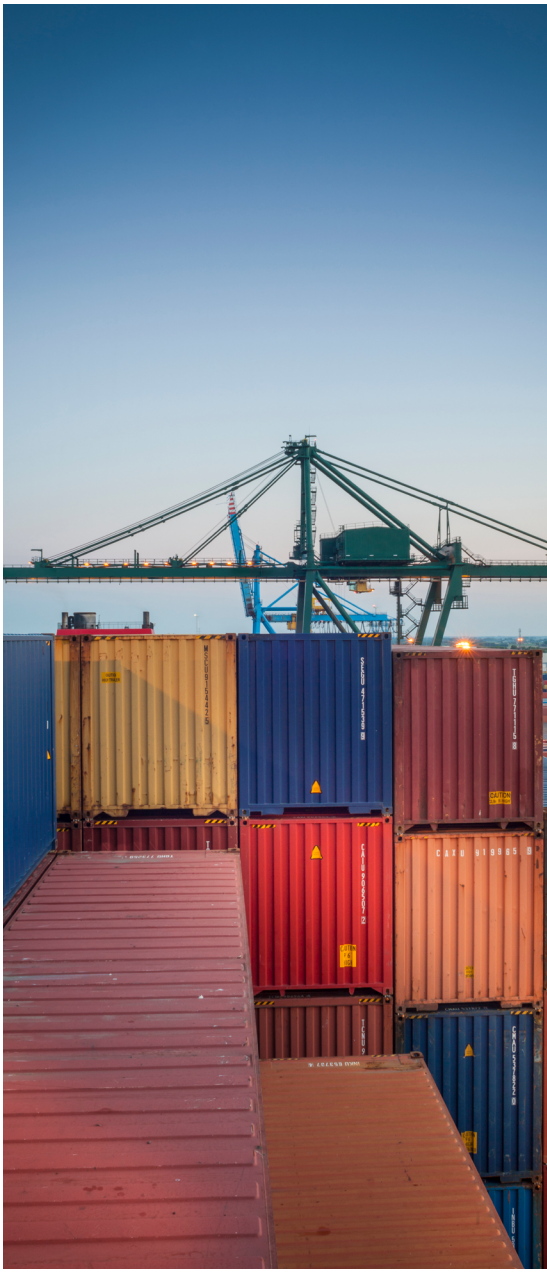
This use case has spawned multiple open source projects aimed at increasing container workloads' isolation, while retaining their density benefits as much as possible. Within these emerging projects, there are two types of approaches: isolation through a secondary kernel running in a virtual machine and isolation through a unikernel. Understanding each solution's approach and differences will help you choose the right isolation project to support your workload.

Lateral movement between containers remains a top-of-mind a threat for users running sensitive workloads, providing SaaS services, or otherwise running untrusted code.

The hypervisor approach

Hypervisors are a better understood boundary, and so generally considered stronger than merely a container boundary. Projects taking this approach are finding methods for generating lightweight virtual machines to act as a secondary boundary for each container or pod, and for each container to have its own kernel. This approach provides isolation of memory, network, I/O, and means that there's a breadth of options in this space for users with legacy virtualization technologies.

One of the projects that's using this approach is Kata Containers. The Kata Containers architecture has multiple components, which can increase start-up time and complexity. But the components are intentionally highly pluggable, which gives users flexibility. For example, users can decide to use QEMU, Firecracker, or Rust VMM as their virtual machine monitor. Because Kata Containers relies on hardware virtualization features, users must be using a cloud services provider that supports nested virtualization, offers bare metal, or Intel VTX, Arm HYP, IBM Power, or IBM Z mainframes.



The unikernel approach

The unikernel approach is about reducing the host kernel's attack surface by providing a stripped down kernel that only has the functionality necessary for a containerized workload. The reduced functionality means that an attacker has reduced methods for reaching the host, and reduced exploit opportunities.

Because the unikernel approach introduces only a minimal set of new components to the architecture, this approach can mean a rapid start-up time and minimal performance impact based on the workload. However, because the unikernel approach is about intentionally limiting the functionality of the kernel, kernel functionality needs to be compatible with the intended workload.

Projects like gVisor and Nabla Containers take the unikernel approach to isolation. gVisor is based on an internal system used by Google for isolating containers. gVisor uses not only a user space kernel, but also a filesystem proxy to give the user space kernel filtered filesystem access.

Determine your requirements first

Determining the isolation solution(s) that will meet your needs is a joint effort between your development and security teams. The first step is to determine the risk and threat profile of your organization and your various containerized workloads, and decide which workloads warrant additional isolation. From there, you can compare the requirements and priorities of those workloads with the isolation options available, as well as your team's tolerance for independently managing a project. If you decide you need a managed version, GKE Sandbox is based on gVisor and available to GKE users. These requirements will help you find the isolation solution that matches your application's needs.

Giving back to open source Kubernetes

As an open source project, the Kubernetes development lifecycle circles between users and developers. Users shape the project's direction through their needs and use cases; developers contribute the code that takes Kubernetes to its next iteration. As an organization that uses Kubernetes, even if you're consuming it as a managed service, there are many ways that you can give back to this worthy project.

User feedback

Developers rely on user feedback. If you're running Kubernetes and encounter a problem or have a feature request, you can open an issue on the Kubernetes GitHub and share what you found. Similarly, if something is working well for you, the community would love to hear about it. You can share your feedback with the community.

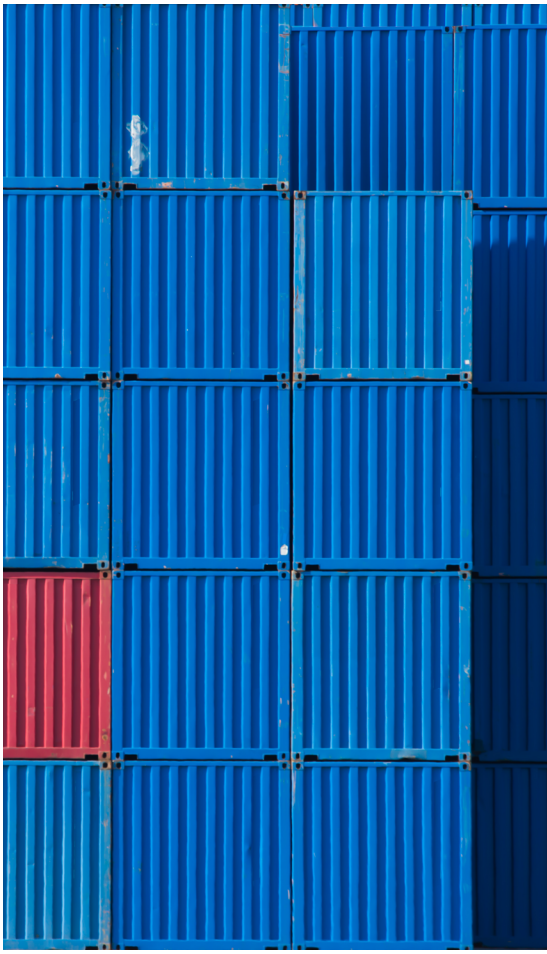
Set aside time for upstream contribution

Once your team is up and running, consider whether there's room in your organization for contributing upstream (meaning back to the open source project), even if you're consuming Kubernetes through a service. Particularly if your team has a specific expertise that derives from your use case (for example, you're in a regulated industry such as banking, or you run Kubernetes in a multi-tenant environment), sharing your knowledge as a contributor comes back around to benefit both you and the greater community.

The TL;DR

- Open source projects thrive when users contribute their stories, time, and talents to them
- Consider setting aside time for your teams to contribute upstream either through code or community involvement

But contributions don't have to consist of code. There are plenty of other ways to get involved, including documentation, release management, or architecture-specific working groups (such as the Multi-tenancy Working Group). You can learn how to get started contributing upstream with the Kubernetes Contributor Guide.



Submit a case study or blog

The community is always looking to share Kubernetes stories. If you have something to say about your migration journey, a problem you encountered, or how Kubernetes has helped your organization, you can submit your story to the Kubernetes blog . Or, if you have a larger success story, you can develop a case study with the help of the Cloud Native Computing Foundation.

Open source software thrives as more people and organizations adopt it. This collective effort and resulting community of collaboration provides tremendous learning, growth, and social opportunities. Once your organization is up and running with Kubernetes, consider what you can do to get involved with the open source project community. Your voice and experience will help make Kubernetes better for everyone.

Putting it all together

Modern applications run in containers; the data that matters to your company is tied to container technology. As a business leader, you might not need to know the ins and outs of Kubernetes security. What you do need to know is that Kubernetes out-of-the-box won't give you the protection you need.

This isn't unique to containers and Kubernetes; as with any other technology, you'll need to determine your particular security requirements and invest in meeting them. For containers, this means establishing your threat model and exploring solutions in the following areas:

- **Infrastructure security**: How is the infrastructure that runs your containers protected? If you're using a managed containers service, what are you responsible for protecting? Does your team have an incident response plan?

- **Supply chain security:** How do you ensure trust throughout the develop, build and deploy lifecycle? Where do your container images come from? How do you verify that you trust what you're deploying?

- **Runtime security**: What applications warrant additional security beyond your organization's defaults? How can you increase the depth of their defense and further isolate risky applications? How can you be alerted to suspected incidents?

References such as the CIS Benchmark can help your development teams learn recommended practices. And encouraging your development teams to spend time in the upstream Kubernetes community will generate a plethora of valuable resources and connections for your organization while supporting the overall health of the Kubernetes project.

Security is an endless journey, not a one-time checklist. Just as the technology that runs your applications has evolved, so too must the steps you take to secure and protect the data that matters to your business.

User Story: How DroneDeploy achieved ISO-27001 certification on GKE

Editor's note: Aerial data mapping company DroneDeploy wanted to migrate its on-premises Kubernetes environment to Google Kubernetes Engine—but only if it would pass muster with auditors. Read on to learn how the firm leveraged GKE's native security capabilities to smooth the path to ISO-27001 certification.

At DroneDeploy, we put a lot of effort into securing our customers' data. We've always been proud of our internal security efforts, and receiving compliance certifications validates these efforts, helping us formalize our information security program, and keeping us accountable to a high standard. Recently, we achieved ISO-27001 certification— all from taking advantage of the existing security practices in Google Cloud and Google Kubernetes Engine (GKE). Here's how we did it.

As a fast-paced, quickly growing B2B SaaS startup in San Francisco, our mission is to make aerial data accessible and productive for everyone. We do so by providing our users with image processing, automated mapping, 3D modeling, data sharing, and flight controls through iOS and Android applications. Our Enterprise Platform provides an admin console for role-based access and monitoring of flights, mapped routes, image capture, and sharing. We serve more than 4,000 customers across 180 countries in the construction, energy, insurance, and mining industries, and ingest more than 50 terabytes of image data from over 30,000 individual flights every month.

Many of our customers and prospects are large enterprises that have strict security expectations of their third-party service providers. In an era of increased regulation (such as Europe's GDPR law) and data security concerns, the scrutiny on information security management has never been higher.. Compliance initiatives are one piece of the overall security strategy that help us communicate our commitment to securing customer data. At DroneDeploy, we chose

We achieved ISO-27001 certification— all from taking advantage of the existing security practices in Google Cloud and Google Kubernetes Engine (GKE)."

to start our compliance story with ISO-27001, an international information security standard that is for recognized across a variety of industries.

DroneDeploy's Architecture: Google Kubernetes Engine (GKE)

DroneDeploy was an early adopter of Kubernetes, and we have long since migrated all our workloads from virtual machines to containers orchestrated by Kubernetes. We currently run more than 150,000 Kubernetes jobs each month with run times ranging from a few minutes to a few days. Our tooling for managing clusters evolved over time, starting with hand-crafted bash and Ansible scripts, to the now ubiquitous (and fantastic) kops. About 18 months ago, we decided to re-evaluate our hosting strategy given the decreased costs of compute in the cloud. We knew that managing our own Kubernetes clusters was not a competitive advantage for our business and that we would rather spend our energy elsewhere if we could.

We investigated the managed Kubernetes offerings of the top cloud providers and did some technical due diligence before making our selection—comparing not only what was available at the time but also future roadmaps. We found that GKE had several key features that were missing in other providers such as robust Kubernetes-native autoscaling, a mature control plane, multi-availability zone masters, and extensive documentation. GKE's ability to run on pre-emptible node pools for ephemeral workloads was also a huge plus.

Proving our commitment to security hardening

But if we were going to make the move, we needed to document our information security management policies and process and prove that we were following best practices for security hardening.

Specifically, when it comes to ISO-27001 certification, we needed to follow the general process:

1. Document the processes you perform to achieve compliance
2. Prove that the processes convincingly address the compliance objectives
3. Provide evidence that you are following the process
4. Document any deviations or exceptions

While Google Cloud offers hardening guidance for GKE and several GCP blogs to guide our approach, we still needed to prove that we had security best practices in place for our critical systems. With newer technologies, though, it can be difficult to provide clear evidence to an auditor that those best practices are in place; they often live in the form of blog posts by core contributors and community leaders versus official, documented best practices. Fortunately, standards have begun to emerge for Kubernetes. The Center for Internet Security (CIS) recently published an updated compliance benchmark for Kubernetes 1.11 that is quite comprehensive. You can even run automated checks against the CIS benchmark using the excellent open source project kube-bench. Ultimately though, it was the fact that Google manages the underlying GKE infrastructure that really helped speed up the certification process.

Compliance with less pain thanks to GKE

As mentioned, one of the main reasons we switched from running Kubernetes in-house to GKE was to reduce our investment in manually maintaining and upgrading our Kubernetes clusters—including our compliance initiatives. GKE reduces the overall footprint that our team has to manage since Google itself manages

and documents much of the underlying infrastructure. We're now able to focus on improving and documenting the parts of our security procedures that are unique to our company and industry, rather than having to meticulously document the foundational technologies of our infrastructure.

For Kubernetes, here's a snippet of how we documented our infrastructure using the four steps described above:

1. We implemented security best practices within our Kubernetes clusters by ensuring all of them are benchmarked using the Kubernetes CIS guide. We use kube-bench for this process, which we run on our clusters once every quarter.

2. A well respected third-party authority publishes this benchmark, which confirms that our process addresses best practices for using Kubernetes securely.

3. We provided documentation that we assessed our Kubernetes clusters against the benchmark, including the tickets to track the tasks.

4. We provided the results of our assessment and documented any policy exceptions and proof that we evaluated those exceptions against our risk management methodology.

Similarly to the physical security sections of the ISO-27001 standard, the CIS benchmark has large sections dedicated to security settings for Kubernetes masters and nodes. Because we run on GKE, Google handled 95 of the 104 line items in the benchmark applicable to our infrastructure. For those items that could not be assessed against the benchmark (because GKE does not expose the masters), we provided links to Google's security documentation on those features (see Cluster Trust and Control Plane Security). Some examples include:

- Connecting kubelets to the masters

- Handling of config files on the masters (e.g. scheduler, controller manager, API server, etc.)
- Hardening the etcd database

Beyond GKE, we were also able to take advantage of many other Google Cloud services that made it easier for us to secure our cloud footprint (although the shared responsibility model for security means we can't rely on Google Cloud alone):

- For OS level security best practices, we we able to document strong security best practices for our OS security because we use Google's Container-Optimized OS (COS), which provides many security best practices by default by using things such as a read-only file system. All that was left for us to do was was follow best practices to help secure our workloads.
- We use node auto-upgrade on our GKE nodes to handle patch management at the OS layer for our nodes. For the level of effort, we found that node auto-upgrade provides a good middle ground patching and stability. To date, we have not had any issues with our software as a result of node auto-upgrade.
- We use Container Analysis (which is built into Google Container Registry) to scan for known vulnerabilities in our Docker images.
- ISO-27001 requires that you demonstrate the physical security of your network infrastructure. Because we run our entire infrastructure in the cloud, we were able to directly rely on Google Cloud's physical and network security for portions of the certification (Google Cloud is ISO-27001 certified amongst other certifications).

DroneDeploy is dedicated to giving our customers access to aerial imaging and mapping technologies quickly and easily. We handles vast amounts of sensitive information on behalf of our customers, and we want them to know that we are following best security practices even when the underlying technology gets complicated, like in the case of Kubernetes. For DroneDeploy, switching to GKE and Google Cloud has helped us reduce our operational overhead and increased the velocity with which we achieve key compliance certifications.

Further reading

- cloud.google.com/containers/security
- “Anthos: An opportunity to modernize security”
- Google Kubernetes Engine security documentation

Google Cloud