Security in Kubernetes

Overview

Best practices in Kubernetes security are rapidly evolving. Many security problems in early versions of Kubernetes are now resolved by default in recent versions, but like any complex system there are still risks you should understand before you trust it with your production data. In this document, we've tried to summarize the most important things you should have in mind when you host sensitive workloads on Kubernetes.

The topics in this guide are meant to help you understand potential risks in your cluster. The risk in your environment will depend on your threat model and the types of applications you run in your cluster. You'll have to consider how best to invest in security controls and hardening based on the sensitivity of your data, the amount of time and staff you're able to dedicate to security, and your company's particular compliance requirements (PCI DSS/HIPAA/FISMA/etc).

Control plane authentication/authorization

The Kubernetes "control plane" consists of the Kubernetes API server, the kubelet, and other core components that cooperate to schedule and run the workloads in your cluster. Access to the control plane is similar to SSH access or physical access to a machine, and should be carefully controlled. The flexibility of the Kubernetes access control system lets you customize access controls to support your environment's particular needs, but that flexibility can be daunting.

At a minimum, you should enable role-based access control (RBAC) in your cluster. RBAC is enabled by default in most recent installers and provides a framework for implementing the principle of least privilege for humans and applications accessing the Kubernetes API.

To get the most benefit from RBAC, you need an appropriate configuration:

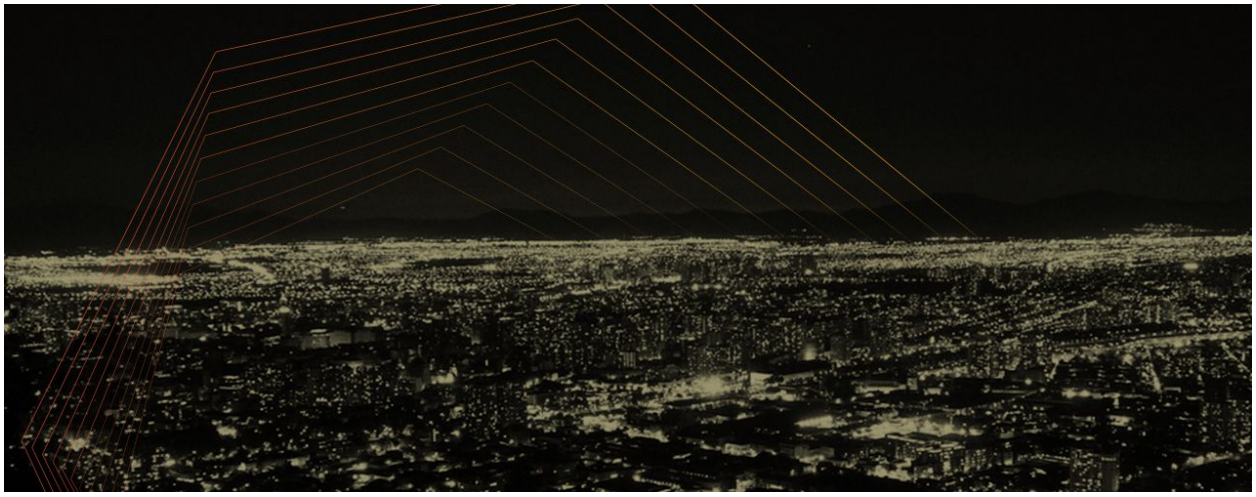
- Run each component with the most restrictive permissions that still allow for expected functionality. Most applications in your cluster will need little or no access to the Kubernetes API. System components such as an ingress controller or monitoring system may need more access, but can often be limited to read-only access or access within a particular namespace. The Kubernetes API audit logs are a useful tool for discovering which APIs a particular application is using.

- Make sure that trusted components don't act as "pivots" that allow less privileged users to escalate privileges. The Kubernetes Dashboard and Helm "tiller" daemon are examples that deserve special attention. Isolate these components with application-level authentication/authorization or network access controls to prevent unauthorized access.

For human users, consider integrating Kubernetes authentication with your existing corporate identity system. Kubernetes ships with the ability to authenticate with any compliant OpenID Connect provider (for example GitHub or Google). You can also

extend Kubernetes authentication and authorization with webhook-based plugins to create a custom identity integration.

You'll also need to secure access to the infrastructure beneath Kubernetes. This might include SSH access or cloud provider access control (for example, AWS IAM). These systems are a potential weak point because they circumvent the Kubernetes access control system. For example, an attacker with SSH access to the Kubernetes etcd host could maliciously insert pod definitions to run arbitrary code elsewhere on the cluster.



Network access control

Many existing applications assume that network-level access implies a level of authorization. Even if your applications include strong application-layer authentication and authorization, network-level access control provides defense in depth. It provides important protection against pre-auth vulnerabilities such as the Heartbleed (CVE-2014-0160) vulnerability in OpenSSL.

Restricting pod-to-pod traffic

Kubernetes provides powerful core data types for specifying network access controls between pods. Network policy in Kubernetes can limit inbound traffic to a pod based on the source pod's namespace and labels, plus the IP address for traffic that originates outside your cluster. In Kubernetes 1.8, network policy can also limit

outbound traffic with the same set of selectors. A good starting point is to restrict ingress to your application namespace by default as described in the Kubernetes Network Policy documentation.

The enforcement of network policy relies on your CNI provider. Not all CNI providers implement these controls. Without them, Kubernetes "fails open" — the API happily accepts your network policies, but the policies are not enforced. If network access controls are important to you, make sure to run a provider like Calico that implements the controls.

Network border controls

Depending on your environment you may also want to enforce some ingress and egress controls at the network border in addition to the pod-level controls enforced by Kubernetes. In particular, in the scenario where an attacker has compromised one of your applications and exploited your container runtime/kernel, you can't trust the node to enforce network access controls.

Limitations

Network access controls have some limitations in dynamic environments like Kubernetes. Difficulties include:

- Federating Kubernetes network policy across multiple clusters.
- Integrating Kubernetes network-level controls and granular network-level controls expressed outside of the pod networking layer (for example, in AWS EC2 Security Groups).

If you encounter these issues, consider whether you can use a more coarse-grained network policy and rely on the application layer for fine-grained access control.

Application-layer access control

One solution to the problem of network-level access controls is strong application-layer authentication such as mutual TLS. A cryptographic application

identity is powerful because it allows that identity to be efficiently expressed across network boundaries.

Securely provisioning certificates for applications is still a hard problem in Kubernetes. Heptio is an active contributor to projects like SPIFFE and the Kubernetes Container Identity working group, which are working to make it easier.

Sidecars and the service mesh approach

Even with application identities provisioned, retrofitting existing applications to implement mutual TLS can be frustrating, time consuming, and error prone. One solution is to implement L7 auth using the sidecar pattern. In this model, each application has an adjacent proxy daemon that terminates and authenticates inbound connections and transparently authenticates outbound connections. Kubernetes makes this easy by allowing multiple containers to run together with a shared localhost network for the pod. When this pattern is used throughout your cluster, it's called a "service mesh". Istio is an early implementation of this approach.

Node and container runtime hardening

You should consider the security of the container-to-host boundary. This is important even in single-tenant environments since a remote code execution vulnerability like Shellshock (CVE-2014-6271) or the Ruby YAML parsing vulnerability (CVE-2013-0156) can turn your otherwise trusted workload into a malicious agent. Without proper hardening, it's possible for that single remote code execution vulnerability to escalate into a whole-node or whole-cluster takeover.

Current container runtimes don't provide the strongest possible sandboxing, but there are some steps you can take to help mitigate the risk of container escape vulnerabilities:

- Segment your Kubernetes clusters by integrity level — a simple but very effective way to limit your exposure to container escape vulnerabilities. For

example, your dev/test environments might be hosted in a different cluster than your production environment.

- Invest in painless host/kernel patching. Make sure that you have a way to test new system updates (for example, a staging environment) and that your applications can tolerate a rolling upgrade of the cluster without affecting application availability.

  Kubernetes shines at orchestrating these upgrades. Once you build confidence in letting Kubernetes dynamically rebalance application pods, patch management at the node level becomes relatively easy. You can automate a rolling upgrade that gracefully drains each node and either upgrades it in place or (in a IaaS environment) replaces it with a fresh node. Investments in this area also improve your overall resiliency to node-level outages.

- Run your applications as a non-root user. Root (UID 0) in a Linux container is still the same user as root on the node. A combination of sandboxing mechanisms restrict what code running in the container can do, but future Linux kernel vulnerabilities are more likely to be exploitable by a root user than by a non-privileged user.

- Enable and configure extra Linux security modules like SELinux and AppArmor. These tools let you enforce more restrictive sandboxing on particular containers. They are valuable in many situations, but building and maintaining appropriate configurations requires a time investment. They may not be appropriate for every application or environment.

Enforcing isolation policies

Pod Security Policies provide a policy-driven mechanism for requiring applications in your cluster to use container sandboxing in an approved way. For example you can require that all pods in a particular namespace run as non-root, do not mount host filesystems, and do not use host networking.

Patch management and CI/CD

Deployment pipelines

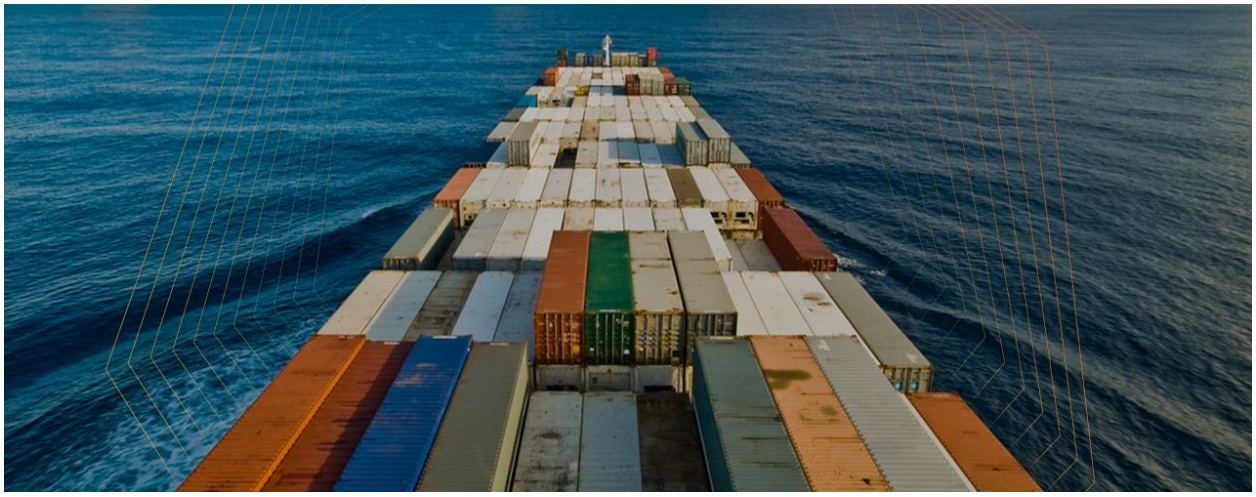
A successful pattern is to have most users interact with the production cluster only through a deployment pipeline. This pipeline consists of one or more automated systems that handle building code into a container image, running unit and integration tests, and other validation steps such as pausing for any manual approval. Depending on your needs, developers could still have direct read-only access to the Kubernetes API or have a way to "break the glass" and exec into pods during an incident.

A robust application deployment pipeline is also the key to remediating vulnerabilities in container images. You can use tools like Clair to identify known vulnerabilities in the libraries and packages you use, but to release patches in a timely manner you need a trusted, automated way of rebuilding and testing patched versions of the container.

Limiting churn

Healthy Kubernetes clusters are dynamic environments. New versions of applications are deployed, nodes disappear for kernel upgrades, deployments scale up and down, and (hopefully) the users of your application never notice. Making all this work in practice requires some diligence but it's critical to reaping all the benefits of Kubernetes.

One tool that can help put bounds on the amount of chaos introduced into your cluster is the Pod Disruption Budget. It's useful when you have multiple automated systems and you want to make sure they don't interact in unwanted ways. For example, an application-level bug might leave some pods of your application temporarily unavailable. A pod disruption budget could make sure that an automated rolling node upgrade doesn't terminate the remaining healthy copies of your application.



Overly privileged container builds

One Docker-specific anti-pattern to avoid in your build pipeline is mounting the host-level Docker control socket "`/var/run/docker.sock`" into a container during a build. Access to this socket is equivalent to root on the host, which means any running build could compromise the node. This is doubly true if your build system runs builds prior to manual code review (a common pattern).

Secrets management

Kubernetes has a core primitive for managing application secrets, appropriately called a Secret. Applications typically need secrets for two key reasons:

1. They need access to a credential that proves their identity to another system (for example, a database password or third-party API token).

2. They need a cryptographic secret for some intrinsic operation (for example, an HMAC signing key for issuing signed HTTP cookies).

Identity secrets

For the first use case of application identity, follow the efforts of SPIFFE and the Container Identity working group for a long term solution to dynamically provisioning unique application identities. In the near term, there is not a well established best practice in this area but some users have success integrating with

existing certificate provisioning workflows as part of a CI/CD pipeline. Simple Kubernetes-native solutions like cert-manager may also work for your use case.

Non-identity secrets

For the second use case, best practices are still evolving but many users are integrating with systems like Vault that perform cryptographic operations in a centralized service. Make sure you understand the entire chain of attestations involved in authenticating to a system like this, since often they will still depend on Kubernetes secret resources as one step in the chain. For example, the Vault Kubernetes auth backend authenticates pods by consuming a Kubernetes Service Account token, but that token is stored as a secret object before it's injected into the pod. This pattern also requires that you trust Vault not to replay your token and impersonate the pod to the Kubernetes API.

Caveats for Kubernetes Secrets

Unfortunately, Kubernetes Secrets come with some caveats. Most importantly, many commonly used components such as ingress controllers currently require permission to read all secrets in your cluster. Secrets are also not encrypted at rest by default. Alpha support for encryption is available in Kubernetes 1.7, but is not yet recommended for production use. One mitigation is to use volume-level encryption (for example, dm-crypt or cloud provider volume encryption) for your etcd data volumes.

Security monitoring and auditing

The right amount and type of security monitoring for your cluster depends largely on the amount of time and staffing you have to respond to alerts and keep an eye on things. As a general rule you shouldn't spend time building security monitoring systems that you don't have the time to maintain and tune. Start with the real time (alert-based) and periodic (audit review) analyst/operator workflows you want to enable and build the monitoring platform you need to enable those workflows.

Logging

The bedrock of security monitoring is logging. You should generally capture application logs, host-level logs, Kubernetes API audit logs, and cloud provider logs (if applicable). There are well established patterns for implementing log aggregation on common cluster configurations.

For security audit purposes, consider streaming your logs to an external location with append-only access from within your cluster. For example, on AWS you can create an S3 bucket in an isolated AWS account and give append-only access to your cluster log aggregator. This ensures your logs cannot be tampered with even in the case of a total cluster compromise.

Network monitoring

Network-based security monitoring tools such as network IDS and web application firewalls may work nearly out of the box, but making them work well takes some effort. The biggest hurdle is that many tools expect IP addresses to be a useful context for events. To integrate these tools with Kubernetes, consider enriching the collected events with Kubernetes namespace, pod name, and pod label metadata. This adds valuable context to the event that you can use for alerting or manual review and can make these traditional tools even more powerful in your cluster than they were in a more traditional environment.

Some monitoring tools can collect Kubernetes metadata already, but you can also write custom event enrichment code to add this kind of metadata integration to those that don't.

Host event monitoring

Host-based IDS such as file integrity monitoring and Linux system call logging (for example, auditd) are also possible to run directly with Kubernetes, but the results are hard to manage because the workload running on any particular node varies from hour to hour as applications deploy and Kubernetes orchestrates pods..

To make sense of host-based events, you'll again want to consider extending your existing tools to include Kubernetes pod/container metadata in the context of captured events. Newer systems like Sysdig Falco include this context out of the box.

Conclusions

What to do now

We keep saying it: how you secure your Kubernetes cluster depends in part on your available resources and on your application requirements. Consider each element in the larger security picture and spend some time up front assessing how important it is to your needs overall. At a very high level some of our recommendations fit nicely into larger best practices in deployment:

- **Automated deployment pipeline and scheduler.** Lets you simplify host and application patch management with rolling upgrades that are integrated into the rest of your overall development cycle.

- **Integrated access controls at appropriate levels**. (authz/authn with API … integration — refocus this point on what to do, not how it works)

- **Integrated logging and monitoring**. You log and monitor for performance and reliability — adding support for security-specific events and pod metadata is non-trivial but vital. Exactly what to monitor depends as always on your specific needs.

Planning for the future

Security is an increasing concern for everyone, and initiatives are well underway to improve the security landscape. Keep an eye out for developments on these fronts:

- More strongly encrypted identity specific to your hardware or cloud provider
- Stronger provenance for cryptographically signed binaries/images
- Automatically updated inventories
- More sophisticated alerts and monitoring

Staying in touch

There's a start on security in Kubernetes. If you're keen to continue the discussion, then please reach out to us via:
Heptio.com | @heptio