Kubernetes and Containerized Security

August 2021
Brendan Francis O'Connor, Risk and Advisory Services

Table of Contents

1 Introduction

We have had several clients approach us regarding how best to create a holistic information security program as they spin up a new-to-them Kubernetes infrastructure for the first time. Particularly for regulated entities, this can seem like a significant challenge; given the rapid changes in infrastructure that Kubernetes makes possible, how can organizations ensure that appropriate security controls are still in place?

This paper provides several specific recommendations around securing Kubernetes itself---that is, securing the control plane which provides container orchestration services to the cluster. In our experience, however, organizations that approach us with questions around "Kubernetes security" have more material concerns around "containerized security"—that is, creating effective security controls that take into account their newly-containerized applications, regardless of what orchestration layer (Kubernetes, OpenShift, or others) is being used. Therefore, in addition to recommendations around security of the control plane, we provide the following six major recommendations to secure an organization's containerized production systems:

1. Automated Testing
2. Private Registry
3. Hardened Base Images
4. Continuous Integration/Deployment
5. Centralized Production Monitoring
6. Container-Specific Incident Response Tooling

This is not a complete guide to Kubernetes or containerized security; many other organizations have published important work in this field, and we have no wish to duplicate it. In particular, we recommend that organizations spend time studying best practices, including the CIS Benchmarks[1] and other standardized recommendation sets for Kubernetes, before committing to a containerized infrastructure.

Throughout this document, we name both paid and open-source products as examples of particular functionality. We have not evaluated, and do not endorse, these products; companies needing these capabilities should make their own evaluation of relevant features, costs, and security benefits.

[1] https://www.cisecurity.org/cis-benchmarks/

2 Control Plane Security

The Center for Internet Security provides CIS Benchmarks,[2] available under a Creative Commons license, which contain excellent baseline configuration guides for a variety of software. For Kubernetes, the version 1.6.1 Benchmark (applicable to Kubernetes 1.18) contains well over 250 pages of information, including 58 automated and 28 manual controls applicable to the primary control nodes, and 10 automated and 13 manual controls applicable to worker nodes to achieve Level 1 compliance. (Automated controls are those verifiable with configuration scanning tools; manual controls are those that are not verifiable in an automated fashion.) 12 additional controls provide Level 2 compliance, some of which may be desirable, though Level 2 controls often come with certain negative effects upon the utility of the cluster. Since such a well-written basic resource exists, this section will focus on controls that are inadequately covered by CIS. We recommend that anyone implementing Kubernetes in production implement all Level 1 controls in any final deployment unless they have specific operational contraindications that outweigh the security benefits. The Level 2 set should be considered best practices, but not requirements if they adversely impact operational abilities (as many will, depending on your particular context).

Authentication: Authentication for human users to the Kubernetes cluster can be difficult to manage at scale; to avoid security pitfalls, it is essentially required to use an external user store that can communicate with Kubernetes via OpenID Connect.[3] Most companies either have an identity store that includes OpenID Connect (for instance, Google Workspaces) or can set up an intermediate OpenID Connect provider to communicate with their primary identity store. Once the Kubernetes cluster is integrated with the identity store, ensure that access control groups (e.g., ClusterRoleBinding, RoleBinding) are bound to groups managed in the identity store; this will decrease the complexity of creating true role-based access control (RBAC) in the Kubernetes clusters.

Secrets: Kubernetes Secrets[4] are the built-in system for managing sensitive data such as TLS private keys, OAuth tokens, and the like. Unfortunately, despite the name, Kubernetes Secrets are not well-protected by default; they are simply base64-encoded to prevent avoidable escaping and newline-type issues. Kubernetes does provide optional encryption at rest[5] for secrets, and basic authorization support for secrets as part of its larger RBAC system; however, due to the implementation of encrypted secrets and the human-opaque appearance of base64 encoding, human error frequently results in unencrypted secrets in practice even when the native encryption is intended. To provide more robust and verifiable secret protections (including usable separate authentication and authorization schemes for human versus automated access), consider using a dedicated secrets management solution such as HashiCorp Vault[6] if appropriate.

---

[2] https://www.cisecurity.org/cis-benchmarks/
[3] https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens
[4] https://kubernetes.io/docs/concepts/configuration/secret/
[5] https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/
[6] https://www.vaultproject.io/

Scheduling: Kubernetes uses its scheduling engines to assign workloads to appropriate worker nodes, including both compute and storage (Persistent Volume Claims). Kubernetes has the ability to maintain a single cluster that has worker and storage nodes with different security levels (for instance, running some workers on dedicated hosts, or using Persistent Volumes with differing resilience configurations) and to assign workloads appropriately based on labeling. However, ensuring that high-security workloads are assigned to high-security nodes is complex in practice, and requires that labels be assigned correctly 100% of the time, imposing a difficult (and excessively manual) process upon engineering teams. To provide a more robust solution, production deployments should use separate Kubernetes clusters for workloads that need substantially different underlying security arrangements. For instance, if a company maintains both regulated and unregulated workloads, it should either run all workloads on clusters conforming to the regulated level of security, or separate the workloads into separate clusters, rather than attempting to maintain perfection when assigning work to different security levels within one cluster; to do otherwise courts disaster with little operational benefit.

Deploy-Time Security: Kubernetes provides two major types of internal controls for security around the container deployment event: Admission Controllers and Pod Security Policies. Admission Controllers perform certain checks within the API Server binary before a deployment can proceed, and are configured at the API Server startup. The default configuration specifies five Admission Controllers:

7. NodeRestriction: Prevents workloads from modifying certain attributes that control how they are deployed.
8. AlwaysPullImages: Ensures that containers are only started by users with the permissions to access the underlying container image; in addition, ensures that containers are not started based on outdated images.
9. ServiceAccount: A default configuration that provides the interfaces necessary for non-human interaction with the API Server.
10. NamespaceLifecycle: Ensures that namespaces must exist and not be in the middle of deletion if a new workload is started in them.
11. EventRateLimit: Allows administrators to configure rate limiting on API events (e.g., starting workloads).

These Admission Controllers are excellent as a baseline, but as one moves toward production-ready clusters, it is worth examining the full list of Admission Controllers[7] to check for others that may be useful depending on specific details of how the cluster is configured, such as CertificateSigning, ImagePolicyWebhook, and SecurityContextDeny.

Pod Security Policies (PSPs) specify additional requirements specific to creating pods. It is common to create PSPs that allow privileged mode; these should not ever be used in production. Privileged mode allows a container to do nearly anything that its host node could do. It is the containerization equivalent of running software as root and has the same substantial downside—that an attacker who is able to exploit the software now has full root privileges. Where there is a need for a container to

[7] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

take a specific action that is ordinarily denied to unprivileged containers, such as manipulating the host clock, the pod specification can assign a specific capability[8] to the container. The difference is substantial; allowing a container to run as privileged is the equivalent of giving twenty-seven capabilities to that container, plus some additional abilities. There are few, if any, reasons that privileged containers should ever be run in production or staging environments; the limited situations in which more-than-default access is necessary can be resolved by capabilities, which should be used as needed.

Configuration Source Control: Kubernetes can be configured in a variety of ways, including various graphical user interfaces, the first-party command-line interfaces (kubectl and kubeadm), and YAML configuration files. Regardless of how a given configuration was first created, every Kubernetes configuration can be output back to a YAML file. We strongly recommend that companies deploying Kubernetes configure its clusters, workloads, and settings using a YAML-first configuration methodology wherever possible; when this is not possible, they should use other tools as needed, then output the resulting configuration to YAML files. These YAML files should be stored in source control. This will ensure that the cluster, or the workloads running on it, does not "drift" from its original configuration in a way that is not visible without direct interrogation of the cluster. This, in turn, will help to achieve compliance and security goals, as well as to ensure that should a cluster or workload need to be recreated, it can occur quickly and without the need for people to remember post-deployment changes.

Local Deployment Sources: For much the same reasons as storing configurations in source control, we recommend that companies disallow Kubernetes deployments from sources not kept within their own systems—that is, that they deploy only container images stored in their private container repository (rather than deploying using, e.g., public Docker Hub images), and that they not use externally-hosted Helm charts or similar shortcuts to deploy resources onto the cluster. This is not because we believe that the public resources are inherently untrustworthy; instead, this recommendation is to allow cluster administrators to be able to view and recreate not just what is currently running in the cluster, but what was previously running at any given point in time. This enables security and audit teams to ensure that they have all relevant data when investigating software defects or security incidents. This point-in-time audit capability is very straightforward if everything is preserved in local source control, but difficult, or impossible, when deploying straight from the Internet, as third parties may change files without notice. For more on this, see the following section on private registries.

[8] https://kubernetes.io/docs/tasks/configure-pod-container/security-context/#set-capabilities-for-a-container

3 Container Security: Automated Testing

The security of an application deployed in Kubernetes is reliant on being able to leverage the particular advantages Kubernetes provides (such as rapid redeployment) against the disadvantages of Kubernetes and containerization more broadly (such as an additional layer of abstraction with its own patching and vulnerability management needs). To make this possible, it is critical to have robust, automated tests in place that can provide a deployable/not deployable status for a given container, including both the application and the container OS.

To be effective in aiding deployment, these tests should be integrated into the build pipeline, and should complete quickly enough to allow many cycles per day, when necessary, of bugfixing and retesting; as a rule of thumb, 30 minutes is a good maximum wall-clock time. If a test suite requires more time than this to execute, allocating additional compute resources to the test environment may be necessary. Alternately, identifying a minimum set of tests from the suite that can be used for a basic "smoke test"[9] and running only those during the build process may be preferred; the longer test suite can still be run periodically (e.g., daily) to ensure that all unit and regression tests continue to pass.

If your application does not have sufficient automated, reliable testing to achieve this goal at this time, our recommendation would be to invest in building tests before investing in containerized deployment. The investment in testing will help to ensure that you can leverage the maximum benefits from containerized or other rapid deployment paradigms confidently, and without testing, most of the following recommendations will be unworkable.

Recommendation

Build a targeted suite of automated smoke tests capable of determining whether your application(s) are working properly, and use them as a central part of an automated build process.

[9] https://en.wikipedia.org/wiki/Smoke_testing_(software)

4 Container Security: Private Registry

A registry is the repository from which Kubernetes pulls specific container images. There are many public registries; the largest is Docker Hub,[10] due to its creation by the same team that built Docker, but GitHub,[11] GitLab,[12] and other services provide public registries as well. They are the container implementation of open-source package hosting, and they are extraordinarily helpful for quickly testing containerized software.

However, in a context in which security, privacy, or data protection matter, they can be problematic. First, like many[13] package[14] registries,[15] public container registries provide a high-value target for attackers---but the security of an individual package (rather than the platform itself) is up to the security of an individual account, leading to a potential for widespread compromise due to an account takeover (ATO) attack on a single developer. In addition, an issue for container registries specifically is that a container contains a significant amount of software other than the principal object of the container.

For instance, the Docker Hub node images[16] may contain, depending on selected tags, Alpine Linux or any of several different versions of Debian. While the Node project updates its containers frequently, many developers may not rebuild containers outside their own software's release cadence. This means that if a software project only publishes new releases every six months, the published container image provided by the project may not have received patches for its container OS for six months; this can lead to making old, well-known vulnerabilities unintentionally available and exploitable in production systems due to the "hidden" unpatched system.

Recommendation

We recommend that any Kubernetes deployment deploy from a private registry exclusively (including network rules preventing accidental pulls from Docker Hub). Container images should, as described in the next section, be built by a trusted build process before being pushed to the private registry.

[10] https://hub.docker.com/
[11] https://github.com/features/packages
[12] https://docs.gitlab.com/ee/user/packages/container_registry/
[13] https://www.helpnetsecurity.com/2019/08/21/backdoored-ruby-gems/
[14] https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes
[15] https://threatpost.com/supply-chain-hack-paypal-microsoft-apple/163814/
[16] https://hub.docker.com/_/node

5 Container Security: Hardened Base Images

Container images, like virtual machine images and bare-metal images before them, should be built on top of a trusted base image. This practice ensures that general hardening guidelines are applied to all systems in production, provides stability for tools in use and supporting infrastructure, and ensures reasonable upgrade pathways are available to enable patching.

When moving applications to a containerized platform, it is common to adopt a "lift and shift" mentality and make only the smallest possible changes to the application. This leads to the use of full operating systems, such as Red Hat Enterprise Linux (RHEL), Ubuntu, or Debian, inside containers. In keeping with the general hardening methodology of "do not install unnecessary software or services," and owing to the unique symbiosis between a running container and its host machine's operating system, it is preferable to build container images on top of a minimal base OS, rather than on a full-featured OS. Alpine Linux[17] is often used for this purpose, but alternatives include Photon OS,[18] which is maintained by VMWare, Fedora CoreOS,[19] or others provided by individual container distributions.

To illustrate the difference in potential for exploitation, a RHEL NodeJS 12 base image (rhscl/nodejs-12-rhel7:1-24) is 534.3 megabytes in size; the most directly equivalent Alpine-based image (including NodeJS version 12, as the RHEL image does), node:12-alpine3.12,[20] is just 89.3MB. That 83% reduction in size translates not simply to significant bandwidth savings both intra-cluster (as images are transmitted from the control plane to worker nodes) and extra-cluster (when container images are pulled from the registry), but a significant reduction in how much code is available for bugs and, in turn, exploitation by attackers.[21]

While there are potentially some pieces of software that require more features from a container than Alpine can provide, there are many container-specific "lightweight" OS distributions available to choose from, any of which might meet the needs of a particular piece of containerized software, and all of which would, inherently, require less hardening than a full OS installation.

Recommendation

We recommend that all container images deployed in production be built on a container-specific OS designed for minimal attack surface. They should also be hardened following best practices for the specific OS; this hardened image should then form the base image layer for application and other necessary containers built for and deployed to production.

---

[17] https://alpinelinux.org/
[18] https://vmware.github.io/photon/
[19] https://getfedora.org/coreos?stream=stable
[20] See https://hub.docker.com/_/node;
[21] To be clear, we are not saying that there are known vulnerabilities in containerized RHEL; instead, we are applying the software security adage of "the most secure code is the code that is no longer shipped." We would make the same recommendation for any other full-featured OS being used inside a container.

6 Container Security: CI/CD

Continuous integration (CI) is the practice of continually building and testing software to ensure that software defects are detected and eliminated in a rapid, responsive cycle, rather than queuing a significant number of changes to await testing all at once. CI is most-often applied to first-party code (i.e., the software that an organization writes itself), which creates an exceptionally robust and flexible deployment paradigm; establishing this is based on a robust suite of smoke tests (as discussed above) to ensure that when the automated tests say that the software under test has passed, it can be deployed without further intervention. (Whether this Continuous Deployment (CD) is adopted or not depends upon the needs of the organization, but the CI system's output should be immediately deployable.)

In addition to first-party application testing, the same CI idea can be applied to ensuring that base container images are rebuilt frequently (e.g., daily, or multiple times per day), including the installation of available software patches (via "apt update && apt upgrade" or similar) and appropriate hardening steps (as discussed above). Given this rapidly updated base image, an organization's dependent container images, containing its own code, can be rebuilt on top of the base image just as frequently as the base image is updated. Combined with automated testing, this would allow an organization to move beyond "approved base container image versions" and simply apply all available patches as soon as they are published by the operating system maintainer. Potential incompatibilities will be caught by the testing system and can be addressed by humans as needed; with continuous integration, however, this will take the form of small bug fixes more frequently, rather than the need to fix an enormous number of bugs all at once, when a new OS version is approved by the appropriate parties.

Whether or not Continuous Deployment (CD) is adopted, the organization should restrict production deployments such that they can only be made by the automated system (with or without human initiation). This requirement can be enforced with continuous monitoring for "rogue" containers (see the following section on monitoring) to ensure that no such containers are allowed to persist. This ensures that only containers built on images that meet organizational security requirements can exist in the production environment. This also has the beneficial effect of providing another point of multiparty control to SDLC-driven code deployment compliance frameworks, since no single employee will be able to deploy code to production without the required approvals.

Recommendation

We recommend that organizations with sufficient testing capabilities adopt both CI and CD for all systems; in the alternative, adopting CI plus human-initiated deployment (where the automated tools handle all aspects of deployment, but the command is given by authorized personnel) is acceptable if organizational requirements prohibit automated deployment. In either case, monitoring should be adopted to ensure that only container images built by the CI system can be run in production.

7 Container Security: Production Monitoring

Many of the general rules of production monitoring apply in equal force to monitoring a containerized infrastructure; it is critical that as an organization moves to Kubernetes, it does not disregard these principles.

First, implement universal, centralized logging. Every Kubernetes node (both primary and worker) and every container should be sending logs to a centralized analysis platform where monitoring and alerting can occur. Two typical solutions for this are Splunk[22] and Elastic Stack;[23] however, given the latter's recent pivot to licensing that is hostile to intellectual property,[24] we no longer recommend it. If an organization has an existing centralized log management and analysis platform, there is absolutely no need to change it; simply make sure that all logs flow to it.

Second, use production-wide monitoring to look for improperly deployed resources. As discussed in the section above, no containers should ever be deployed other than by the automated systems, and containers should be destroyed and redeployed frequently. In a public cloud environment, tools like Cloud Custodian[25] can implement rules like this; in private datacenters, developing custom tooling may be preferable.

Third, use the same type of configuration monitoring tools to look for improper configurations at the orchestration layer. If desirable, tools like the Tenable suite[26] can add another layer of configuration monitoring to prevent unintended drift. It is likely that some points of configuration will require custom scripted or manual checks to verify, particularly with regard to checking that intra-Kubernetes networking is set to "deny by default" at all times, with specific business reasons given for each allow rule.[27]

Finally, embrace immutable infrastructure. Containers are spawned from images which are built by an automated process; they should not need to be manipulated manually (e.g., SSHed into, or having manual commands run via Kubernetes exec) as part of the deployment lifecycle. Debugging can take place, in most cases, by using the logs emitted by the container and simulation on development machines. Where it becomes necessary to run commands on a container in production, the container should be automatically marked for destruction in a short time period so that an accidental side effect of the commands cannot negatively affect production traffic. This also helps to prevent development of the post-deployment-configuration antipattern, where it requires less effort on the part of the relevant engineer simply to make changes on running containers, rather than making the change to the image build process—which creates siloed knowledge and leads to increased debugging complexity.

---

22 https://www.splunk.com/
23 https://www.elastic.co/elastic-stack
24 https://aws.amazon.com/blogs/opensource/stepping-up-for-a-truly-open-source-elasticsearch/
25 https://cloudcustodian.io/
26 https://www.tenable.com/products/tenable-io or similar products.
27 This would be a container-specific reflection of the common firewall rules control required by PCI-DSS (1.2.1), SOC 2 (TSP CC6.6), ISO 27001 (A.13.1.3) NIST 800-53r5 (SC-7(11)(21)(25)), etc.

7.1 Container Compliance Monitoring

In addition to preserving and centralizing logs from all running containers, as discussed above, an optional, but useful, monitoring technique is to use tools to find differences between running containers. If ten containers, spawned from the same image, are running in production and one is different from the others in any way, it is reasonable to assume some security-impactful failure has occurred (whether it is a software defect causing a resilience failure or an active exploitation). Accordingly, periodically (but frequently; for instance, every 5-10 minutes) use diffing tools to examine similar running containers and, when an outlier is found, trigger containerized incident response tools (detailed below) and respawn a fresh container from the base image.

One such tool is built into Docker,[28] though the Docker command-line tools will soon be removed from a default Kubernetes installation.[29] (They can of course still be installed separately.) Some incident response tools for containerized workflows also include this functionality.[30] Alternately, it may be more useful to develop customized in-house tools to achieve this.

Recommendation

Centralized, monitored logs (with appropriate alerting) are table stakes for an operational security posture, and that requirement does not change in a containerized infrastructure. Any organization not already doing this should address it with the highest possible priority, as lacking logs means that no effective incident response or even performance analysis can exist.

[28] https://docs.docker.com/engine/reference/commandline/diff/
[29] https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/
[30] https://github.com/coinbase/dexter

8 Container Security: Incident Response

The general virtual infrastructure incident response flow is as follows:

1. Quarantine the affected infrastructure (disable its ability to send or receive information to/from other systems or the Internet).
2. Obtain useful data from the quarantined infrastructure.
3. Terminate the affected infrastructure and respawn unaffected infrastructure, using monitoring to watch for attempted re-exploitation until the underlying defect has been found and fixed.
4. Once the defect has been fixed, redeploy all infrastructure that contained the vulnerability, or which may have been exposed to harm from the exploited system.

Given the rapid spawning and reaping of containerized infrastructure, using tooling for incident response and forensics is even more critical than in a virtual machine-based infrastructure. If a container's maximum lifetime is measured in minutes or hours, manual processes will not be able to keep pace.[31] What specific tooling to use will be determined by what integrates best with an organization's existing processes and tools. Regardless of what tools are selected, it is critical to establish a runbook for the security operations team that covers how to use the tools to collect relevant information in a short amount of time, and how to refer the investigation to appropriate personnel (from security, development, or SRE) to track down the issue and remediate it in a timely fashion.

Recommendation

Establish and frequently test containerized incident response runbooks, including specific and customized tooling and commands, to ensure that your security operations team can rapidly handle a containerized incident.

[31] For more background on this, an excellent (if somewhat dated) conference presentation is available at https://www.youtube.com/watch?v=I1HU67Vd7ec .

9 Stretch Goal: Automated Vulnerability Management and Remediation

Some heavily-regulated organizations are required to use vulnerability scanning tools (such as Clair[32] or Sysdig Secure[33]) to ensure that "known vulnerabilities"—defined as vulnerabilities for which a patch is available—are not present for longer than a defined period in their infrastructure. In a containerized environment, this has led to organizations scanning container images on build and at deploy, creating significant performance issues for automatic scaling, and then periodically during runtime. However, given a private registry (whether run using the Docker registry image[34] or a paid product such as JFrog Artifactory[35]), an automated deployment system (such as Jenkins[36] or GitHub Actions[37]), and a vulnerability scanning system, an organization can create an extremely effective control system while removing these scanning-related performance problems by ensuring that only approved container images can be deployed, and then putting tight controls around what "approved" means in this context.

In summary, once a continuous integration system is, as discussed above, constantly rebuilding, checking, and tagging new versions of container images as ready for deployment, an organization can move the performance-intensive container vulnerability scan to "scan on build" exclusively, and eliminate "scan on deploy" and "scan during runtime." This would remove the performance penalty that scanning every image, every time it is deployed, imposes on starting new containers without sacrificing security. To prevent long-running containers from exceeding vulnerability constraints, use the production monitoring system to redeploy all long-lived containers automatically; the redeployment will use the newer approved image with, once again, no harms to security.

An example will help illustrate how this would work in practice. For the sake of discussion, we will say that existing security controls at a hypothetical company require that no package with a known vulnerability be used in production for more than 7 days (168 hours) after the release of a patch. With no automatic rebuilding of deployment container images, the hypothetical company currently needs to use a vulnerability scanner to scan at three separate points:

1. at container build, to ensure that packages included in the build have no known vulnerabilities for which patches have exceeded the deadline;
2. at container deploy, to ensure that packages in the deployed image have not exceeded the deadline between build and deploy time; and
3. during runtime, to ensure that packages in the running container have not exceeded the deadline while they have been running.

---

[32] https://github.com/quay/clair
[33] https://sysdig.com/products/secure/
[34] https://hub.docker.com/_/registry
[35] https://jfrog.com/artifactory/
[36] https://www.jenkins.io/
[37] https://github.com/features/actions

Now contrast a continuous integration system, which rebuilds the base container image, and all container images dependent upon the baseline, every six hours.[38] In general, then, a container image stored in a private registry and marked as the latest image will not contain a known vulnerability for which a patch has been available for more than six hours (plus the time it takes to actually build the container images). Should a patch cause a defect, it will be caught by the test suite running as part of the continuous integration and flagged for human intervention; as long as the fix is pushed to the code repository and picked up by the CI system within a reasonable amount of time (roughly, within six days), no image stored in the registry and marked as the latest image available will be able to exceed the 7-day control. **This eliminates the need to scan at container deploy time** (and suffer the significant hit to deployment performance).

Since images will be rebuilt so frequently, the organization can use Kubernetes to remove and redeploy running containers frequently. This is made possible by the inherent ephemerality of containers and their extremely low start-up costs, once deploy-time scanning has been eliminated. If orchestration automatically redeploys any containers that have exceeded six hours' runtime, no container in production will be more than twelve hours away from its build (and patching) under normal circumstances. If an image update was delayed due to a bug that required human intervention to fix, the control is still preserved; even if it takes six days for the fix to be identified, coded, and pushed to the code repository, there will still be sufficient time to allow a rebuild, redeploy, and reaping of all old containers before the seven-day deadline.[39] Since running containers therefore cannot exceed the seven-day deadline, **this removes the need to scan running containers for vulnerabilities** (with the attendant performance penalty), while maintaining the regulatory controls. While the specific timings might change based on a company's compliance requirements, an appropriate CI cadence and SLA for human bugfixes when needed can be constructed to ensure that the vulnerability scan's performance hit only needs to be endured by the CI system, and by no other component of the stack. This increases resilience and performance when deploying and scaling systems, and provides effective, low-impact vulnerability management within a highly adaptable infrastructure.

[38] Once again, this is entirely reliant on automated, reliable smoke testing as discussed in an earlier section.
[39] This would be while relying on the periodic automatic redeployment; a human could of course trigger an off-cadence rebuild and redeploy to decrease the time necessary for a fix to propagate from code repository to production while still utilizing the automated systems (and therefore not requiring a security exception).

10 Conclusion

While containerized production with advanced, large-scale orchestration engines such as Kubernetes is relatively new, the security implications of containers are a reflection of existing best practices, rather than a complete rewrite. Using Kubernetes adds new levels of abstraction to core concepts like networking and storage, just as using virtual machines did when moving from bare metal production systems; however, the same core controls of protecting data at rest, data in transit, secrets, and access apply at each new level. In general, companies should feel comfortable with continuing their high-level security guidance even as the underlying technology changes.