aqua

Container Security **10 Things DevOps Need to Do**

Introduction

Few technologies have revolutionized software development the way containers have. According to Gartner, 65% of enterprises already have containers in production (end of 2017), with an additional 21% planning to go into production in 2018 and beyond.

The growth of containers has led to a revolution in the way organizations develop and deploy applications. Just a few years ago, most deployments were made up of isolated greenfield applications created and managed using Docker. Today, organizations have adopted tools such as Kubernetes for the purpose of transforming their application delivery process into an agile, scalable machine.

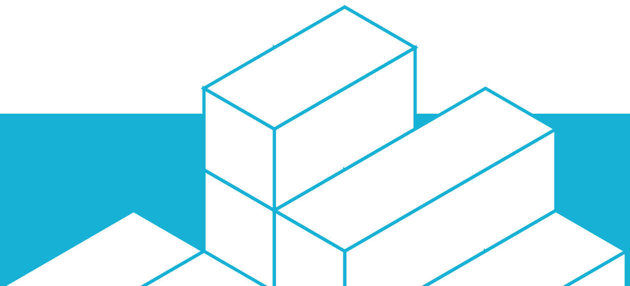
This has broad implications not just for development and operations, but also for security. With breaches such as the Equifax breach affecting millions of people at a time, the need for strong application security has never been more important.

For security departments, adoption of containerization presents a multi-faceted challenge:

Containerization introduces new infrastructure that operates dynamically and is open in nature, with more potential for cross-container activity.

- Containerization introduces new processes that move code through the software development pipeline at an accelerated pace and with greater flexibility (or, less oversight).
- Containerization introduces new identities accessing resources in the form of developers and IT operations that may have an immediate effect on runtime environments.

From a delivery perspective, these are all elements that contribute to reducing the time to deployment, application resource footprint – collapsing the friction between operations and development.



Dev

- Threats to the build environment
- Platform security
- Segmenting microservices network
- Segregation of duties

Test

- Tightening user access control
- Hardening the host

Run

- Managing vulnerabilities in images
- Reduce the container attack surface
- Secrets management
- Container runtime security

One of the challenges with this shortened pipeline is that it often leaves no room for security. The goal of DevOps is to deploy changes as quickly as possible and with as little maintenance as possible. As a result, processes that don't directly contribute to the rapid implementation of changes are seen by developers as a hindrance. 52% of developers worry that application security will delay development and threaten deadlines, leaving security teams with little to no visibility into the DevOps process.

In enterprise environments, trusting developers not to introduce vulnerabilities can only go so far. The product must meet stringent requirements for compliance, reporting, and threat mitigation, and the only way for security teams to verify this is through full visibility into the DevOps process. This blending of DevOps and security is known as DevSecOps, and already 63% of organizations have either a formal or informal DevSecOps team in house.

Here are ten key things DevOps should know about securing containerized applications:

1. Threats to the Build Environment
2. Managing Vulnerabilities in the Container Image
3. Reducing the Container Attack Surface
4. Tightening User Access Controls
5. Secrets Management
6. Hardening the Host
7. Container Runtime Security
8. Platform Security
9. Segmenting Microservices Network
10. Segregation of Duties

1. Threats to the Build Environment

Despite being a critical piece of the DevOps pipeline, the build environment is often overlooked as a security liability. Traditionally the build environment was less of a security risk, since it was not directly linked to production environments. With companies becoming more agile and adopting Continuous Integration and Continuous Development (CI/CD) practices, build environments now need to push changes to production several times a day. This requires direct access to secure resources, making it a more valuable target for attackers, who are also finding it harder to penetrate highly secure production environments directly. Attackers who infiltrate this environment could not only introduce malicious code, but also gain access to privileged accounts, internal services, and company secrets.

With many steps involved in the build process, vulnerabilities can be introduced anywhere: compromised source code, malicious libraries, loose access credentials, or insecure build servers. Containers compound this issue by merging applications with their operating environments. While this simplifies the deployment process, it creates an abstraction that makes it harder for security teams to audit the underlying code. Security teams must now analyze the application, the container environment, the images used to build the container, the build tools, and the build environment. The only way to do this without significantly impacting turnaround time is to give security teams insight into the build process from beginning to end.

DevOps teams need to secure the resources used during the build process to ensure the application remains untainted from source code to final artifact. Privileged access to the build environment should be restricted to only a few authorized users. Build tools that require access to code repositories, company servers, and other secure resources should also be restricted to user accounts with limited access, as well as isolated from other systems.

2. Managing Vulnerabilities in the Container Image

Images are the static code files from which containers are instantiated. As a result, any vulnerabilities present in an image are guaranteed to be present in each container instantiated from it. From a security perspective, images must be as secure as possible or the entire application is at risk. It is also much easier and more effective to secure images at the source, rather than chase an impossibly large and elusive set of running containers.

Images are stored using a layer-based file system. Each layer represents a change from the layer below it, such as the addition of a file. Layers are reusable between images and containers to reduce download and deployment times. However, each layer also increases the risk of a vulnerability finding its way into the final container. A vulnerability in a common base layer can easily find its way into numerous containers.

Additional risk is posed by the use of public image registries such as Docker Hub, which allows developers to download and share images with millions of other users. Many of these images – especially base Linux images such as Ubuntu – form the base layer of many other images, which then go on to form the base of other images. Securing all of these layers can quickly become unmanageable, unless access to the original sources is controlled.

When using public registries, DevOps teams should only download signed images from trusted sources not tampered with in transit. Images should undergo comprehensive security scans and be pinned to a specific version, preventing unverified updates from introducing vulnerabilities. It's also recommended to only allow designated developers to access public registries, and have an internal registry that only stores trusted base images for other developers to use.

3. Reducing the Container Attack Surface

Although containers provide a degree of isolation, vulnerable or improperly configured containers can still be leveraged to gain access to the host system. For example, Docker containers run as the root user by default. While this root user has fewer privileges than the host root user, it can still access mounted resources as well as the host kernel. An attacker could leverage these entry points to access secure files or exploit vulnerabilities to gain administrative access on the host.

One of the strengths of containerization is the isolation of the user namespace from the host system. This isolation allows for users and groups to be defined and managed independently of those of the host. A user account created within a container can still gain access to container resources and files but has a significantly smaller attack surface than a root user.

DevOps must follow the principle of least privilege and only have the permissions necessary to run the containerized application. The benefit is twofold: if an attacker gains access to the container, their ability to interact with the container is limited. And as a result, the attacker will find it more difficult to break out of the container onto the host.

4. Tightening User Access Control

Restricting user access to containers is essential, since containers of varying trust levels or sensitivity, as well as those associated with different applications, may run within the same cluster or on the same host.

Both authentication and authorization should be controlled, so as to limit which users have access to which groups or types of containers, and, once they do have access, what their privileges allow them to do. For example, containers that make up PCI-compliant applications need to have access controlled separately from containers that make up other, non-PCI, applications. Additionally, an auditor or compliance team member may require visibility into an environment (seeing what processes are running, viewing audit logs), but should not have the ability to start or stop containers, or alter the Docker environment configuration.

These controls can be resolved through the use of external user access control systems, allowing you to set granular permissions for users rather than root level access. For example, both Docker EE[1] and Kubernetes[2] support role-based access control to resources in each of their respective environments, but they must be governed using a centralized policy.

Without a centralized approach, it's difficult to determine whether the privileges assigned to users are consistent with their functional role, especially if those roles change over time.

1 https://docs.docker.com/ee/ucp/authorization

2 https://kubernetes.io/docs/reference/access-authn-authz/rbac

5. Secrets Management

Some containers require access to sensitive data during their operation. This sensitive data – known as secrets – can include credentials, tokens, and passwords. Solutions already exist for distributing secrets securely, but the transitory nature of containers has made the process more challenging.

All too often, secrets are hard-coded into the source code, images, or the build process. While this is convenient for testing, it has unintended side-effects, since there's no telling in advance where an image would end up. Secrets embedded in an image can be spread to any user with access to the image, even when the image is stored in a private registry. And restricting secrets to build tools limits the ability for developers to test the application on their local systems.

A good secrets management solution allows DevOps teams to store secrets in a secure centralized vault while also ensuring that the relevant containers have access to the secrets they need in a way that makes the secret inaccessible anywhere along the way. This reduces the chance of an unsecured container leaking secrets, or abusing a secret to gain access to a restricted resource.

6. Hardening the Host

Although containers provide their own isolated environments, they still typically run on a host OS, sharing the kernel resources. As a result, hosts should be hardened against possible attacks originating from the container or the container runtime.

Host security depends heavily on the operating system (OS). It is highly recommended to dedicate hosts that run containers to only run containers, and not mix containerized with non-containerized workloads. This make it easier to control access and work with orchestrators. For running containers, it's recommended to use a "thin OS", typically a Linux distribution that is optimized for running containers and doesn't include many of the unneeded capabilities of a full enterprise Linux distro - examples include Red Hat CoreOS, Rancher OS, and VMware's Photon OS. A thin OS not only reduces the attack surface but offers improved speed and lower resource usage than a full OS. Certain minimal OSs may also offer features tailored to containers such as native clustering tools, a read-only root filesystem, and native support for Docker and other runtimes.

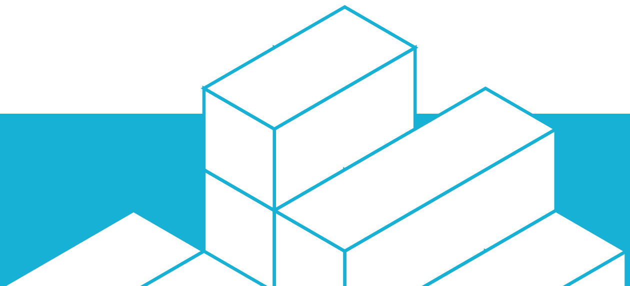
Most container runtimes provide additional hardening by leveraging Linux access limitation processes. The key components are namespaces and control groups (or cgroups). In essence, cgroups determine how much of the shared kernel and system resources a container can consume, while namespaces define which resources the container is allowed to access. It's also recommended to use seccomp profiles, which limit access to Linux kernel system calls and can improve container-to-host isolation and prevent kernel exploits via containers.

In production environments, user access should also be strictly limited to "break glass" scenarios. Under normal operations, there's no reason for any human admin to SSH into a host or configure it manually. Host should be managed through configuration management templates (e.g., using Chef or Ansible), and only the orchestrator should have ongoing access to run/stop containers.

7. Container Runtime Security

Containers can write files, start new processes, and increase their resource usage. An attacker who gains access to a container can abuse this resource usage, running unauthorized code and persisting changes across future instances of the container.

To reduce this risk, containers should be immutable. Immutability prevents a container from being modified once it's running, requiring an entirely new image to be deployed if changes need to be made. This ensures that all containers created from an image are identical on first start and that each container behaves in essentially the same way. It also



prevents the creation of “snowflake” deployments—deployments with custom or unique configurations—which are harder for DevOps teams to troubleshoot.

Even after ensuring immutability, containers should be monitored for modifications, exposed ports, open connections, and other signs of intrusion.

8. Platform Security

Containerized applications are only as secure as the container runtime itself. The steps involved will vary depending on the runtime, the host OS that the runtime is running on, and any additional features or plugins used. However, several organizations have published guidelines for optimal platform security.

Two of the more popular auditing tools is the Center for Internet Security (CIS) Docker Benchmark[3] and its Kubernetes Benchmark, a step-by-step checklist of industry best practices for managing Docker installations and Kubernetes clusters in a secure way. Each section describes not only the implications and risks of a security vulnerability but how to detect and remediate it. This includes updating OS components and applications, setting mandatory access control policies, and performing comprehensive logging and monitoring.

There are open source tools that can perform checks against these benchmarks - for example Aqua’s Kube-Bench runs the full list of checks against the CIS Kubernetes Benchmark.

3 https://kubernetes.io/docs/reference/access-authn-authz/rbac

9. Segmenting the Microservices Network

Monolithic applications typically have relatively static network deployments with reserved IP addresses, hostnames, and ports. Containers, however, get dynamic IP addresses from orchestrators, and may appear and disappear from nodes quite frequently, with a lifespan of hours rather than days. Instead of conceptualizing networks as physical connections hidden behind a gateway, container networks are software-defined networks that operate on logical connections based on the container service context, usually provided by the orchestrator or, in more advanced deployments, by a service mesh.

Microsegmentation involves splitting up a network into smaller zones and applying high-level IT security policies to each zone. Policies can be applied to specific types of traffic or even to specific containers. This allows DevOps teams to manage ports, resolve DNS addresses, create load balancers, and proxy requests across complex, distributed applications.

If an attacker does gain access to a networked container, segmentation ensures that the number of resources exposed to the attacker is much smaller than it would be otherwise. For security, preventive microsegmentation is often not enough, and additional firewall controls between containerized services should be implemented, in order to go beyond passive measures and actively monitor and block attackers that try to traverse the network.

10. Segregation of Duties

Segregation of Duties, often referred to as SoD, is a security principle that mandates that no single person or team should have unlimited privileged access to a system. Typically, this means that, for example, sysadmins who manage and run servers will not be responsible for the security of those servers, nor will they have access to change the security settings - that job will be performed by the security team.

With containers, the lines are often blurred due to the heavy use of DevOps methods and sometimes a form of DevSecOps approach that merges security processes into DevOps. However, this does not mean that SoD should not be enforced - in fact, it should be enforced more than ever, since the speed at which damage can be caused by both malicious insiders as well as simple mistakes is multiplied.

For examples, in many Kubernetes setups, there is no built-in SoD, and the cluster admin is "all powerful". This poses unacceptable risk in enterprise production environments, and there must be controls in place that allow security teams to monitor and enforce security policy, assess risk, and prevent admins from changing security controls on the cluster, the Kubernetes console, and access to nodes on the cluster.

About Aqua

Aqua Security enables enterprises to secure their container and cloud-native applications from development to production, accelerating application deployment and bridging the gap between DevOps and IT security.

Aqua's Container Security Platform provides full visibility into container activity, allowing organizations to detect and prevent suspicious activity and attacks in real time. Integrated with container lifecycle and orchestration tools, the Aqua platform provides transparent, automated security while helping to enforce policy and simplify regulatory compliance. Aqua was founded in 2015 and is backed by Lightspeed Venture Partners, Microsoft Ventures, TLV Partners, and IT security leaders, and is based in Israel and Boston, MA.

For more information, visit www.aquasec.com

or follow us on twitter.com/AquaSecTeam