StackRox

Docker Security Cheat Sheet

Containers, along with orchestrators such as Kubernetes, have ushered in a new era of application development methodology, enabling microservices architectures as well as continuous development and delivery. Docker is by far the most dominant container runtime engine, with a 91% penetration according to our latest State of the Container and Kubernetes Security Report.

Containerization has many benefits and as a result has seen wide adoption. According to Gartner, by 2020, more than 50% of global organizations will be running containerized applications in production. However, building apps using Docker containers also introduces new security challenges and risks. A single compromised Docker container can threaten all other containers as well as the underlying host, underscoring the importance of securing Docker.

Securing Docker can be loosely categorized into two areas: securing and hardening the host so that a container breach doesn't also lead to host breach, and securing Docker containers. This cheat sheet focuses on container security by highlighting Docker container security risks and challenges as well as providing best practices for securing your environment during the build and deploy phases and protecting your Docker containers during runtime.

We also share best practices for securing Kubernetes, given its massive adoption and critical role in orchestrating containers. Finally, we provide you with 11 key security questions you should be able to answer to ensure you are running containers securely in production.

8 container security challenges you must address for Docker

Companies have long deployed applications on virtual machines (VMs) or bare metal servers. Security for that infrastructure involved securing your application and the host it's running on and then protecting the application as it runs. Containerization introduces several new challenges that must be addressed.

1. Containers enable microservices, which increases data traffic and network and access control complexity.

2. Containers rely on a base image, and knowing whether the image comes from a secure or insecure source can be challenging. Images can also contain vulnerabilities that can spread to all containers that use the vulnerable image.

3. Containers have short life spans, so monitoring them, especially during runtime, can be extremely difficult. Another security risk arises from a lack of visibility into an ever-changing container environment.

4. Containers, unlike VMs, aren't necessarily isolated from one another. A single compromised container can lead to other containers being compromised.

5. Containerized environments have many more components than traditional VMs, including the Kubernetes orchestrator that poses its own set of security challenges. Can you tell which deployments or clusters are affected by a high-severity vulnerability? Are any exposed to the Internet? What's the blast radius if a given vulnerability is exploited? Is the container running in production or a dev/test environment?

6. Container configuration is yet another area that poses security risks. Are containers running with heightened privileges when they shouldn't? Are images launching unnecessary services that increase the attack surface? Are secrets stored in images?

7. As one of the biggest security drivers, compliance can be a particular challenge given the fast-moving nature of container environments. Many of the traditional components that helped demonstrate compliance, such as firewall rules, take a very different form in a Docker environment.

8. Finally, existing server workload security solutions are ill-equipped to address container security challenges and risks.

26 Docker security best practices

What follows is a list of best practices derived from industry standards and StackRox customers for securely configuring your Docker containers and images.

1 Always use the most up to date version of Docker. The runC vulnerability from earlier this year, for example, was quickly patched soon after its discovery with the release of Docker version 18.09.2.

2. Allow only trusted users control of the Docker daemon by making sure only trusted users are members of Docker group. Check out this article for more information about decreasing your Docker daemon attack surface.

3. Make sure you have rules in place that give you an audit trail for:
    a. Docker daemon
    b. Docker files and directories:
        i. /var/lib/docker
        ii. /etc/docker
        iii. Docker.service
        iv. Docker.socket
        v. /etc/default/docker
        vi. /etc/docker/daemon.json
        vii. /etc/sysconfig/docker
        viii. /usr/bin/containerd
        ix. /usr/sbin/runc
    c. Check out this article for more details

4. Secure all Docker files and directories (see 4.2 above) by ensuring they are owned by the appropriate user (usually the root user) and their file permissions are set to a restrictive value (see the CIS benchmarks section on Docker daemon configuration files).

5. Use registries that have a valid registry certificate or ones that use TLS to minimize the risk of traffic interception.

6. If you are using containers without an explicit container user defined in the image, you should enable user namespace support, which will allow you to re-map container user to host user.

7. Disallow containers from acquiring new privileges. By default, containers are allowed to acquire new privileges so this configuration must be explicitly set. Another step you can take to minimize a privilege escalation attack is to remove the setuid and setgid permissions in the images.

8. As a best practice, run your containers as a non-root user (UID not 0). By default, containers run with root privileges as the root user inside the container.

9. Use only trusted base images when building your containers. This tip might seem like an obvious one, but third-party registries often don't have any governance policies for the images stored in them. It's important to know which images are available for use on the Docker host, understand their provenance, and review the content in them. You should also enable Content trust for Docker for image verification and install only verified packages into images.

10. Use minimal base images that don't include unnecessary software packages that could lead to a larger attack surface. Having fewer components in your container reduces the number of available attack vectors, and a minimal image also yields better performance because there are fewer bytes on disk and less network traffic for images being copied. BusyBox and Apline are two options for building minimal base images.

11. Implement a strong governance policy that enforces frequent image scanning. Stale images or images that haven't been scanned recently should be rejected or rescanned before moving to build stage.

12. Build a workflow that regularly identifies and removes stale or unused images and containers from the host.

13. Don't store secrets in images/Dockerfiles. By default, you're allowed to store secrets in Dockerfiles, but storing secrets in an image gives any user of that image access to the secret. When a secret is required, use a secrets management tool.

14. When running containers, remove all capabilities not required for the container to function as needed. You can use Docker's CAP DROP capability to drop a specific container's capabilities (also called Linux capability), and use CAP ADD to add only those capabilities required for the proper functioning of the container.

15. Don't run containers with –privileged flag, as this type of container will have most of the capabilities available to the underlying host. This flag also overwrites any rules you set using CAP DROP or CAP ADD.

16. Don't mount sensitive host system directories on containers, especially in writable mode that could expose them to being changed maliciously in a way that could lead to host compromise.

17. Don't run sshd within containers. By default, the ssh daemon will not be running in a container, and you shouldn't install the ssh daemon to simplify security management of the SSH server.

18. Don't map any ports below 1024 within a container as they are considered privileged because they transmit sensitive data. By default, Docker maps container ports to one that's within the 49153 - 65525 range, but it allows the container to be

mapped to a privileged port. As a general rule of thumb, ensure only needed ports are open on the container.

19. Don't share the host's network namespace, process namespace, IPC namespace, user namespace, or UTS namespace, unless necessary, to ensure proper isolation between Docker containers and the underlying host.

20. Specify the amount of memory and CPU needed for a container to operate as designed instead of relying on an arbitrary amount. By default, Docker containers share their resources equally with no limits.

21. Set the container's root filesystem to read-only. Once running, containers don't need changes to the root filesystem. Any changes made to the root filesystem will likely be for a malicious objective. To preserve the immutable nature of containers – where new containers don't get patched but rather recreated from a new image – you should not make the root filesystem writable.

22. Impose PID limits. One of the advantages of containers is tight process identifier (PID) control. Each process in the kernel carries a unique PID, and containers leverage Linux PID namespace to provide a separate view of the PID hierarchy for each container. Putting limits on PIDs effectively limits the number of processes running in each container. Limiting the number of processes in the container prevents excessive spawning of new processes and potential malicious lateral movement. Imposing PID limits also prevents fork bombs (processes that continually replicate themselves) and anomalous processes. Mostly, the benefit here is if your service always runs a specific number of processes, then setting the PID limit to that exact number mitigates many malicious actions, including reverse shells and remote code injection – really, anything that requires spawning a new process.

23. Don't configure your mount propagation rules as shared. Sharing mount propagation means that any changes made to the mount will propagate to all instances of that mount. Instead set the mount propagation in slave or private mode so that a necessary change made to a volume isn't shared with (or propagated to) containers that don't require that change.

24. Don't use docker exec command with privileged or user=root option, since this setting could give the container extended Linux capabilities.

25. Don't use the default bridge "docker0." Using the default bridge leaves you open to ARP spoofing and MAC flooding attacks. Instead containers should be on a user-defined network and not the default "docker0" bridge.

26. Don't mount Docker socket inside containers, since this approach would allow a process within the container to execute commands that give it full control of the host.

7 Kubernetes security best practices

As the de facto standard for container orchestration, Kubernetes plays a pivotal role in ensuring your applications are secure. To effectively secure containerized applications, you must leverage the contextual information from Kubernetes as well as its native policy enforcement capabilities.

For example, Kubernetes has several built-in security features that make it easier to operationalize full life cycle container security, including Kubernetes RBAC, Network Policies, and Admission Controllers. Tap into the power of these inherent control capabilities in Kubernetes to protect your containerized environments.

Below are some Kubernetes security best practices that help operationalize full life cycle container security.

1. For RBAC, specify your Roles and ClusterRoles to specific users or groups of users instead of granting cluster-admin privileges to any user or groups of users.

2. Avoid duplication of permissions when using Kubernetes RBAC, as doing so could create operational issues.

3. Remove unused or inactive RBAC roles so that you can focus your attention on the active roles when troubleshooting or investigating security incidents.

4. Use Kubernetes network policies to isolate your pods and explicitly allow only the communication paths required for the application to function. Otherwise you're exposing yourself to both lateral and north-south threats.

5. If your pods need Internet access (either ingress or egress), create the appropriate network policy that enforces the right network segmentation/firewalling rule, then create a label that's targeted by the said network policy, and lastly associate your pods to that label.

6. Use the PodSecurityPolicy admission controller to ensure proper governance policies are being enforced. The PodSecurityPolicy controller can prevent containers from running as root or make sure a container's root filesystem is mounted read-only (these recommendations should sound familiar, given they're both on the previous list of Docker measures to take).

7. Use the Kubernetes admission controller to enforce image registry governance policies such that all images taken from untrusted registries are automatically denied.

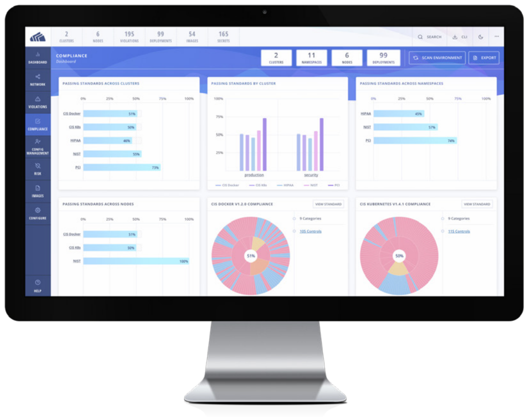
11 security questions you should be able to answer about your Docker container environment

To help you quickly assess your security posture, we've compiled a list of questions your security, DevSecOps, or DevOps teams should readily be able to answer if your cloud-native stack has been architected with appropriate security measures.

1. How many images are on a host where the last scan date exceeds 60 days?
2. How many images/containers have a high-severity vulnerability?
3. Which deployments are impacted by these high-severity vulnerable containers?
4. Are there any containers in the impacted deployments that have secrets stored in them?
5. Are any of the vulnerable containers running as root or with privileged flag?
6. Are any of the vulnerable containers in a Pod that doesn't have a network policy associated with it (meaning it allows all communication)?
7. Are any containers running in production impacted by this vulnerability?
8. Where are the images we're using coming from?
9. How are we blocking images that are being pulled from untrusted registries?
10. Are we able to see which processes are executing during container runtime?
11. Which clusters, namespaces, and nodes are non-compliant with CIS benchmarks for Docker and Kubernetes?

Final thoughts

Follow the best practices compiled in this cheat sheet and you'll have taken the most important steps to successfully securing your Docker and Kubernetes environments and protecting your critical business applications.



Ready to see StackRox in action?

Get a personalized demo tailored for your business, environment, and needs.

REQUEST DEMO

StackRox helps enterprises secure their containers and Kubernetes environments at scale. The StackRox Kubernetes Security Platform enables security and DevOps teams to enforce their compliance and security policies across the entire container life cycle, from build to deploy to runtime. StackRox integrates with existing DevOps and security tools, enabling teams to quickly operationalize container and Kubernetes security. StackRox customers span cloud-native start- ups Global 2000 enterprises, and government agencies.

LET'S GET STARTED

Request a demo today!

info@stackrox.com

+1 (650) 385-8329

www.stackrox.com