Docker Security

Dokument Initiator	Donzyk, Oliver (Devops)
Dokumenten Status	UPTODATE
Letzte Überprüfung	31.07.2021 Donzyk, Oliver
ℹ weitere wichtige Seiten unter:	

Content:

Docker Security

Kernel Exploits

Durch gemeinsame Kernel Nutzung des Host-Systems kann unter anderem ein Kernel-Panic-Fehler hervorgerufen werden.

Denial-of-Service-(DoS-)Angriffe

Container teilen sich ihre Kernel-Ressourcen des Hosts. Falls ein Container alle Ressourcen für sich beansprucht (Speicher, UIDs etc.) würden alle anderen Container Probleme bekommen.

Container-Breakouts

Ein Angreifer eines Containers sollte besten falls keinen Zugriff auf andere Container oder den Host zugreifen können.

Da die Benutzer nicht über Namensräume getrennt sind, bekommen alle Prozesse, die aus dem Container ausbrechen, auf dem Host die gleichen Privilegien wie im Container

Vergiftete Images

Herkunft und Sicherheit des Images sollten geprüft sein!
Images aus nicht verifizierbaren Quellen bergen Gefahren das Sie mit Schadsoftware und sonstigen manipulativen Mechanismen bestückt sein können!
Genauso ist es wichtig das ausgeführte Images aktuell sind und keine Softwareversionen mit bekannten Sicherheitslücken enthalten.

Docker Digests, Docker Content Trust - Ist Content Trust aktiviert, arbeitet die Docker Engine nur mit signierten Images und verweigert das Ausführen anderer Images, deren Signaturen oder Digests nicht passen.

Um Content-Trust für eine lokale Registry zu verwenden, müsste zusätzlich noch einen Notary-Server (https://github.com/docker/notary) konfigurieren und bereitgestellt werden.

Verratene Geheimnisse

Container die auf Datenbanken oder Services zugreifen benötigen Passwörter oder API Schlüssel welche möglichst nicht in Images gelagert werden sollten.
Hierzu gibt es sichere Auslagerungen in externe Konfigurationsdateien.

- Einsatz von Umgebungsvariablen
    - Umgebungsvariablen sind für alle Kind-Prozesse, docker inspect und verlinkte Container sichtbar. Keiner davon hat einen guten Grund, diese Geheimnisse sehen zu müssen.
      Das ist einer der Gründe, warum man in der Version 1.10 von Docker das Link-Konzept verändert hat und nun eben nicht mehr die ENV-Variable in die gelinkten Container spiegelt.
    - Die Umgebung wird häufig aus Protokollierungs- oder Debugging-Gründen gesichert. Das Risiko ist groß, dass die Variablen in Debug-Logs oder IssueTrackern einzusehen sind.
    - Sie können nicht gelöscht werden. Idealerweise würden wir das Geheimnis nach seinem Einsatz überschreiben oder auslöschen, aber bei Docker-Containern geht das nicht.
- Eine etwas bessere – wenn auch bei weitem nicht perfekte – Lösung ist der Einsatz von Volumes, um Geheimnisse zu übergeben.
- Die wohl beste Lösung ist der Einsatz eines Key/Value-Store, um Geheimnisse zu sichern und sie zur Laufzeit im Container auslesen zu können. Damit haben Sie Steuerungsmechanismen, die bei den vorigen Lösungen nicht zur Verfügung stehen – allerdings ist das Einrichten aufwendiger, und Sie müssen dem Key/Value-Store vertrauen. zB: KeyWhiz, Vault, Crypt.

Least Privilege

Jeder Prozess und Container sollte nur mit so viel Zugriffsrechten und Ressourcen laufen, wie er gerade braucht, um seine Aufgaben zu erfüllen.

- sicherstellen, dass Prozesse in Containern nicht als root laufen, so dass das Ausnutzen von Sicherheitslücken in einem Prozess dem Angreifer keine root-Berechtigungen geben
- Dateisysteme schreibgeschützt einsetzen, so dass Angreifer keine Daten überschreiben oder böswillige Skripten speichern können
- die Kernel-Aufrufe, die ein Container ausführen kann, einschränken, um die Angriffsoberfläche zu verringern
- die Ressourcen begrenzen, die ein Container nutzen kann, um DoS-Angriffe zu verhindern, bei denen ein kompromittierter Container oder eine Anwendung so viele Ressourcen auf braucht (wie zum Beispiel Speicher oder CPU-Zeit), dass der Host zum Halten kommt

Netzwerkzugriffe von Containern beschränken

Ein Container sollte in der Produktivumgebung nur die Ports öffnen, die er tatsächlich benötigt, und diese sollten auch nur für die anderen Container erreichbar sein, die sie brauchen.
Eine weitere Sicherheitsebene kann hier über Networks mit Traefik eingerichtet werden. So wird die Kommunikation der Container in jeweilige Stacks über eine oder mehre bridges Segmentiert.

Den Speicher begrenzen

Durch die Begrenzung des verfügbaren Speichers schützt man sich vor DoS Angriffen und Anwendungen mit Speicherlecks, die nach und nach den Speicher auf dem Host auffressen (solche Anwendungen können automatisch neu gestartet werden, um den Service beizubehalten).

Den CPU-Einsatz beschränken

Kann ein Angreifer einen Container – oder eine ganze Gruppe – dazu bringen, die CPU des Host vollständig aus zulasten, werden andere Container auf dem Host nicht mehr arbeiten können, und man hat es mit einem DoS-Angriff zu tun.

Neustarts begrenzen

Stirbt ein Container immer wieder und wird dann neu gestartet, muss das System nicht unerheblich Zeit und Ressourcen aufwenden, was im Extremfall auch zu einem DoS führen kann. Das lässt sich leicht mit der Neustart-Vorgabe on-failure statt always vermeiden.

Zugriffe auf die Dateisysteme begrenzen

Wenn verhindert wird, dass Angreifer in Dateien schreiben, stören Sie eine ganze Reihe von Angriffen und machen das Leben von Hackern ganz allgemein schwerer. Sie können kein Skript in eine Datei schreiben und Ihre Anwendung dazu bringen, diese auszuführen, oder kritische Daten oder Konfigurationsdateien überschreiben.

Die allermeisten Anwendungen müssen aber in Dateien schreiben können und sind nicht dazu in der Lage, in einer vollständig schreibgeschützten Umgebung zu arbeiten. In solchen Fällen können Sie die Ordner und Dateien herausfinden, für die die Anwendung schreibenden Zugriff benötigt, und nur diese Dateien als Volumes mounten.

Nicht unterstützte Treiber vermeiden

Der Einsatz diverser Features ist ein Sicherheitsrisiko, da sie nicht die gleiche Aufmerksamkeit und Anzahl an Updates erhalten wie andere Teile von Docker. Das Gleiche gilt für Treiber und Erweiterungen, die von Docker abhängen.

Insbesondere sollten Sie nicht den veralteten LXC-Execution-Treiber verwenden. Standardmäßig ist er schon abgeschaltet!

Ressourcenbeschränkungen (ulimits) anwenden

Der Linux-Kernel definiert Ressourcenbeschränkungen, die für Prozesse gesetzt werden können – zum Beispiel die Anzahl der Kind-Prozesse, die sich forken lassen, oder die Anzahl der zulässigen offenen File-Deskriptoren. Diese lassen sich auch auf Docker-Container anwenden – entweder durch Übergabe des Flags --ulimit an docker run oder durch das Setzen containerübergreifender Standards per --default-ulimit beim Start des Docker Daemon

Von besonderem Interesse sind die folgenden Werte: cpu, nofile, nproc

User Namespaces (userns-remap)

User Namespaces sind vor allem dann empfehlenswert, wenn Sie in Ihren Containern als root -Benutzer arbeiten müssen.
Docker stellt durch sogenannte Namespaces sicher, dass jeder Container eigene UIDs, GIDs und PIDs hat (also eigene User-, Gruppen- und Prozessnummern) und somit die Prozesse des Hosts oder anderer Container nicht sieht.

Mount-Namespaces vermitteln jedem Container seine eigene Sichtweise auf das Dateisystem. Der Container kann außer seinem bereits beschriebenen Overlay-Dateisystem und eventuell gemeinsam genutzten Volumes nicht auf andere Verzeichnisse des Host-Dateisystems zugreifen.

Schließlich erhält jeder Container seinen eigenen Netzwerk-Stack.

https://docs.docker.com/engine/security/userns-remap/

Auditing

Um zu kontrollieren, dass Ihr System sauber und aktuell ist, und um ganz sicherzugehen, dass keine erfolgreichen Angriffe vorgenommen wurden, sind regelmäßige Audits oder Reviews für Ihre Container und Images eine sehr gute Idee. Bei einem Audit für ein containerbasiertes System wird geprüft, dass alle laufenden Container aktuelle Images nutzen und dass diese Images aktuelle und sichere Software einsetzen. Jeder Unterschied eines Containers zu seinem Image, aus dem er erstellt wurde, sollte erkannt und kontrolliert werden. Zusätzlich sollten Audits andere Bereiche abdecken, die nicht spezifisch für Containersysteme sind, wie zum Beispiel das Prüfen der Zugriffsprotokolle, der Dateiberechtigungen und der Datenintegrität. Wenn Audits weitgehend automatisiert werden können, ist es problemlos möglich, sie regelmäßig laufen zu lassen, um Probleme so schnell wie möglich erkennen zu können.

Zukünftige und umgesetzte Features

Seccomp

Das Prinzip ist simpel: Seccomp beschränkt die erlaubten Systemaufrufe (Syscalls) auf das absolut notwendige Minimum.
Seccomp basiert auf dem Prinzip von Profilen: In einem solchen Profil legt der Administrator fest, auf welche Systemaufrufe ein Programm Zugriff hat. Das Zielprogramm muss Seccomp unterstützen, denn es muss das Profil, mit dem es assoziiert sein möchte, selbst auswählen und durch den »seccomp()«-Syscall setzen. Versucht ein Programm, das durch ein Seccomp-Profil eingeschränkt ist, einen im Profil nicht ausdrücklich erlaubten Syscall auszuführen, schickt der Kernel des Hostbetriebssystems ihm kurzerhand das »SIGKILL«-Signal und befördert es so ins Nirwana. De facto agiert die Seccomp-Funktionalität also als Erweiterung des schon beschriebenen Linux-Capability-Systems: Nicht alle Operationen, die man als Admin einem Docker-Container verbieten möchte, lassen sich über Capabilities abdecken – hier springt Seccomp in die Bresche. Noch eine gute Nachricht: Seit Version 1.10 beherrscht Docker das Setzen von Seccomp-Profilen auf Basis von einzelnen Containern. Das Standardprofil ist in (https://docs.docker.com/engine/security/seccomp) ausgiebig erklärt, dort steht auch, welche Funktionen dieses Standardprofil blockiert

AppArmor-Alternative SELinux

Leider steht AppArmor nur unter wenigen Linux-Distributionen zur Verfügung, insbesondere unter Ubuntu und (open)SUSE. Linux-Distributionen aus dem Red-Hat-Umfeld, insbesondere also RHEL, Fedora, CentOS & Co., vertrauen dagegen mit SELinux auf eine andere Sicherheitstechnik.

Verschlüsselte Übertragung

Jegliche Kommunikation intern und extner sollte mit Zertifikaten verschlüsselt stattfinden.
Hierzu bedarf es einem Zertifikat welches durch alle Systeme evaluiert und angewendet werden kann.
Sollte kein eigenes echtes Root-Zertifikat bereitstehen kann ggf. Letsencrypt Anwendung finden.

Zum Schluss sei angemerkt das Docker schnelllebig weiter entwickelt wird und sich hier bereits neue Lösungen in der Planungsphase abzeichnen.

Die Absicherung für Systeme sollte auch aus mehreren Ebenen bestehen. Die Container werden zum Beispiel sehr wahrscheinlich in VMs laufen – geschieht ein Container-Breakout, kann eine weitere Verteidigungsebene dafür sorgen, dass Angreifer nicht auf den Host oder an Container kommen, die anderen Benutzern gehören. Das System sollte überwacht werden, um Administratoren bei ungewöhnlichem Verhalten alarmieren zu können. Firewalls sollten den Netzwerkzugriff der Container begrenzen und damit die externe Angriffsoberfläche minimieren.

Verwendung möglicher Applikationen zur Sicherung, Prüfung und Verwaltung von Applikationen, Containern und Plattform:

- APTLY - Mirror
- Jenkins
- Selenium
- Gitlab
- Sonartype / Nexus
- Sonarqube
- Docker / Docker-Compose
- Kubernetes
- Traefik
- Portainer
- Graylog
- DNS Cloud Domain
- Letsencrypt
- OpenVAS
- Netdata
- Graphana / Graphite
- Rkhunter
- Clair / falco / anchore / Docker-bench
- logspout
- watchtower
- dockly

- VSphere
- Vagrant
- Terraform
- Ansible

Diagramm: Docker Client, Daemon und Registry

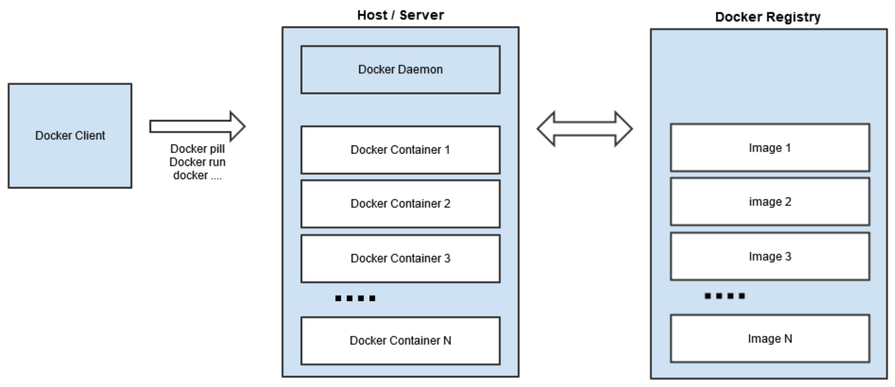
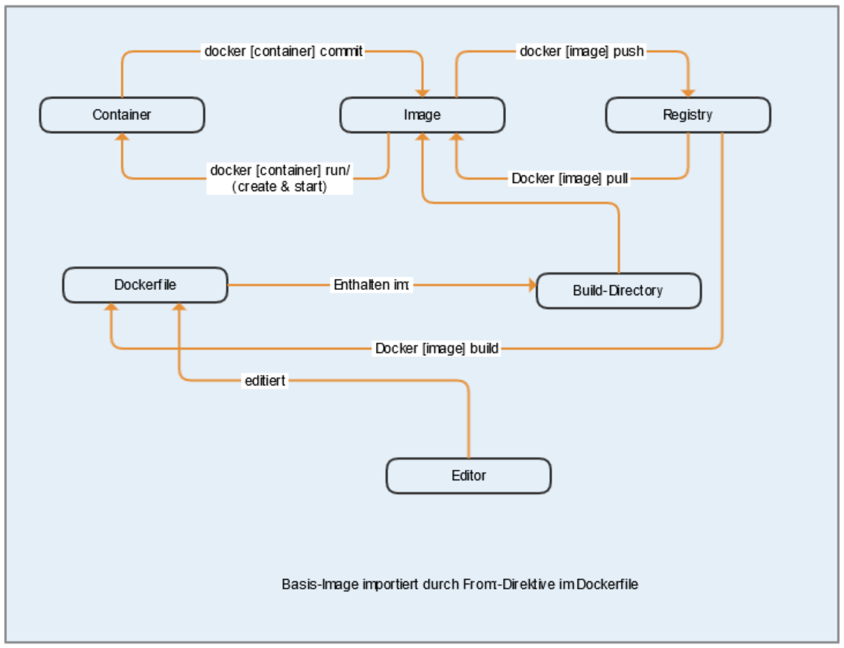


Diagramm: Docker Workflow (vereinfacht)



Funktionales Diagramm: Docker Host / Server, Client und Registry

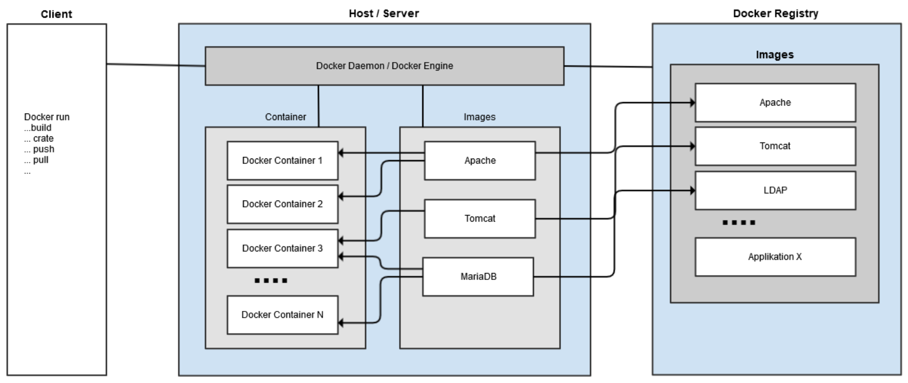


Diagramm Workflow Containers:

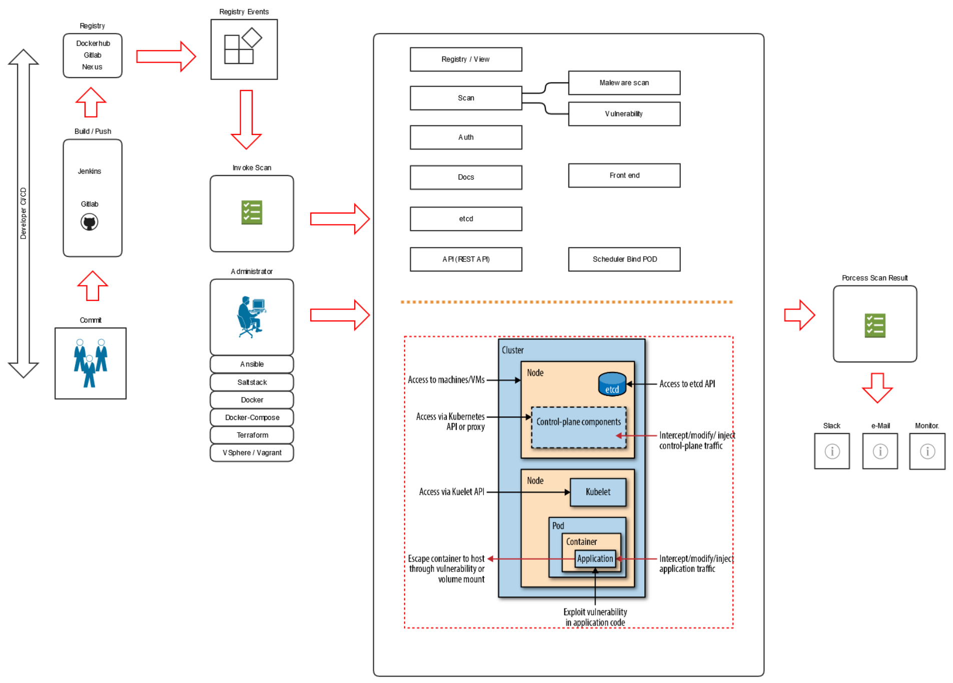


Diagramm vereinfachte Container Cluster / Orchestrierung:

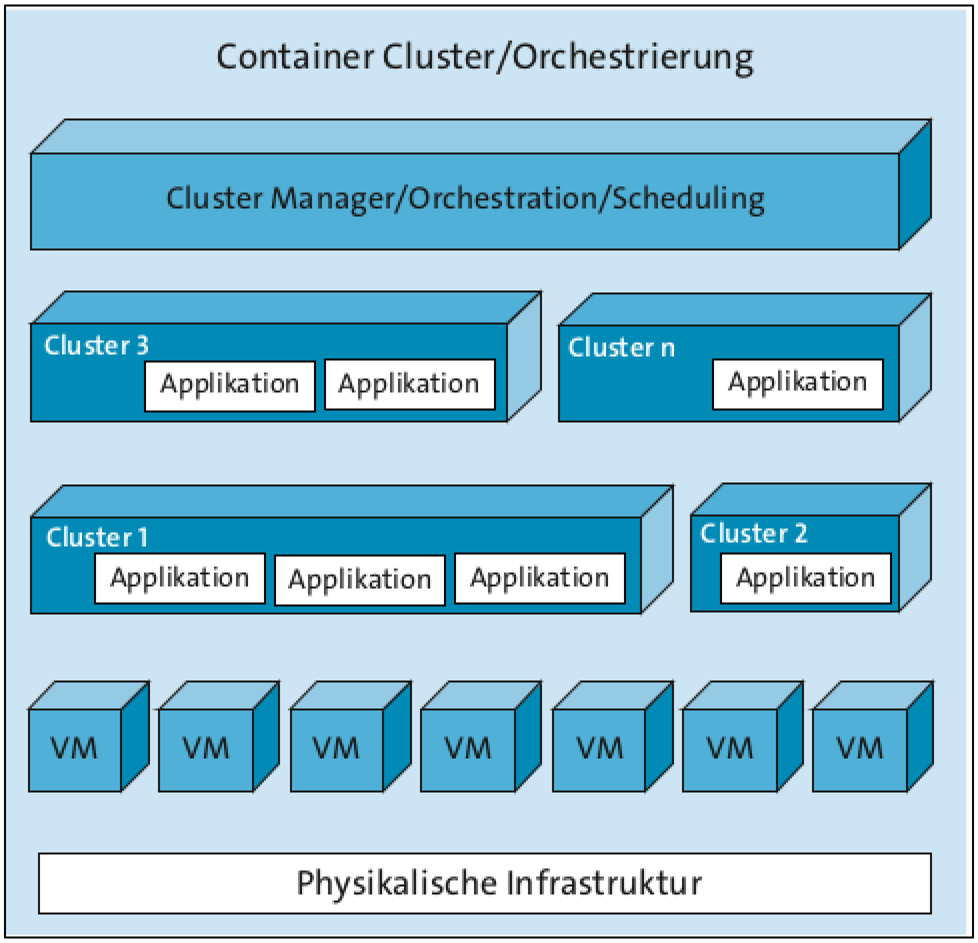


Diagramm Docker DataCenter - DDC:

Security Checklist:

		Item	Must have	Beschreibung
1		**General Configuration**		
2		Ensure the container host has been Hardened	⚠	This test is just a note to remind you to consider hardening your host. Hardening usually involves setting up a firewall, locking down various services, setting up auditing and logging, and implementing other security measures
3		Ensure that the version of Docker is up to date	⚠	Be care use the newest Version
4		**Linux Hosts Specific Configuration**		

5		Swap limit support?		▪ Auditing can generate large log files. You should ensure that these are rotated and archived periodically. A separate partition should also be created for audit logs to avoid filling up any other critical partition
				▪ By default, Docker related files and directories are not audited
				
				Docker Flags zB:
				-m 128m --memory-swap 128m
				
				Für Docker-Compose zB:
				`deploy:`
				`      resources:`
				`        limits:`
				`          cpus: '0.001'`
				`          memory: 50M`
				`        reservations:`
				`          cpus: '0.0001'`
				`          memory: 20M`
				
				Prüfen mit "free -mh"

6		Ensure a separate partition for containers has been created		▪ By default, /var/lib/docker is mounted under the / or /var partitions dependent on how the OS is configured ▪ All Docker containers and their data and metadata are stored in the /var/lib/docker directory. By default, /var/lib/docker should be mounted under either the / or /var partitions depending on how the Linux operating system in use is configured ▪ Docker depends on /var/lib/docker as the default directory where all Docker related files, including the images, are stored. This directory could fill up quickly causing both Docker and the host to become unusable. For this reason, you should create a separate partition (logical volume) for storing Docker files. ▪ For new installations, you should create a separate partition for the /var/lib/docker mount point. For systems that have already been installed, you should use the Logical Volume Manager (LVM) within Linux to create a new partition. ⚠️
7		Ensure only trusted users are allowed to control Docker daemon	⚠️	

8		Ensure auditing is configured for the Docker daemon	⚠️	ℹ️
				sudo apt-get install auditd
				sudo nano /etc/audit/audit.rules
				
				Paste the following snippet at the bottom of the file, then save and exit the editor:
				`-w /usr/bin/docker -p wa`
				`-w /var/lib/docker -p wa`
				`-w /etc/docker -p wa`
				`-w /lib/systemd/system/docker.service -p wa`
				`-w /lib/systemd/system/docker.socket -p wa`
				`-w /etc/default/docker -p wa`
				`-w /etc/docker/daemon.json -p wa`
				`-w /usr/bin/docker-containerd -p wa`
				`-w /usr/bin/docker-runc -p wa`
				
				sudo systemctl restart auditd

9		Ensure auditing is configured for Docker files and directories		▪ /var/lib/docker ▪ /etc/docker ▪ docker.service ▪ docker.socket ▪ /etc/default/docker ▪ /etc/sysconfig/docker ▪ /etc/docker/daemon.json ▪ /usr/bin/containerd ▪ /usr/sbin/runc
10		**Docker daemon configuration**		
11		Ensure network traffic is restricted between containers on the default bridge		▪ You can restrict all inter-container communication and link specific containers together that require communication ▪ you can create a custom network and only join containers that need to communicate to that custom network ▪ inter-container communication is disabled: "icc": false or dockerd --icc=false ▪ By default, all inter-container communication is allowed on the default network bridge. ⚠️
12		Ensure the logging level is set to 'info'		"log-level": "info"or dockerd --log-level="info" By default, Docker daemon is set to log level of info.
13		Ensure Docker is allowed to make changes to iptables		▪ The iptables firewall is used to set up, maintain, and inspect the tables of IP packet filter rules within the Linux kernel. The Docker daemon should be allowed to make changes to the iptables ruleset ▪ Do not run the Docker daemon with --iptables=false parameter ⚠️ ▪ The Docker daemon service requires iptables rules to be enabled before it starts. Any restarts of iptables during Docker daemon operation may result in losing Docker-created rules. Adding iptables-persistent to your iptables install can mitigate ▪ By default, iptables is set to true

14		Ensure insecure registries are not used		▪ Docker considers a private registry either secure or insecure. By default, registries are considered secure ▪ You should ensure that no insecure registries are in use ⚠ ▪ By default, Docker assumes all registries except local ones are secure
15		Ensure aufs storage driver is not used		▪ Do not use aufs as the storage driver for your Docker instance ▪ The aufs storage driver is the oldest storage driver used on Linux systems. It is based on a Linux kernel patch-set that is unlikely in future to be merged into the main OS kernel. The aufs driver is also known to cause some serious kernel crashes. aufs has only legacy support within systems using Docker. Most importantly, aufs is not a supported driver in many Linux distributions using latest Linux kernels. ▪ Do not explicitly use aufs as storage driver. For example, do not start Docker daemon with the --storage-driver aufs flag. ▪ By default, Docker uses devicemapper as the storage driver on most of the platforms. The default storage driver can vary based on your OS vendor. You should use the storage driver that is recommended by your preferred vendor and which is in line with policy around the applications which are being deployed.
16		Ensure TLS authentication for Docker daemon is configured		▪ Bby default, Docker runs via a non-networked Unix socket and TLS must be enabled in order to have the Docker client and the daemon communicate securely over HTTPS. TLS ensures authenticity of the registry endpoint and that traffic to/from registry is encrypted
17		Ensure the default ulimit is configured appropriately		▪ --default-ulimit ulimit Default ulimits for containers (default [])
18		Enable user namespace support		▪ The best way to prevent privilege-escalation attacks from within a container is to configure your container's applications to run as unprivileged users. For containers whose processes must run as the root user within the container, you can re-map this user to a less-privileged user on the Docker host. The mapped user is assigned a range of UIDs which function within the namespace as normal UIDs from 0 to 65536, but have no privileges on the host machine itself. ▪ You can start dockerd with the --userns-remap flag or follow this procedure to configure the daemon using the daemon.json configuration file. The daemon.json method is recommended ▪ dockerd --userns-remap="testuser:testuser" ▪ https://docs.docker.com/engine/security/userns-remap/
19		Ensure the default cgroup usage has been confirmed		▪ It is possible to attach to a particular cgroup when a container is instantiated. Confirming cgroup usage would ensure that containers are running in defined cgroups. ▪ System administrators typically define cgroups in which containers are supposed to run. If cgroups are not explicitly defined by the system administrator, containers run in the docker cgroup by default. ▪ At run time, it is possible to attach a container to a different cgroup other than the one originally defined. This usage should be monitored and confirmed, as by attaching to a different cgroup, excess permissions and resources might be granted to the container and this can therefore prove to be a security risk ▪ You should not use the --cgroup-parent option within the docker run command unless strictly required
20		Ensure base device size is not changed until needed		▪ Likewise, there is a default limit of 10 GB size set for the containers. This is the maximum size up to which a container can grow, and is defined by the parameter 'Base Device Size'. Increasing the container size limit is possible, as long as its within the pool limit. ▪ When this default limit for docker container size is increased in Docker, it will impact the size all newly created containers. It is also possible to increase the storage pool size to above 100GB.
21		Ensure that authorization for Docker client commands is enabled		▪ Docker socket is protected by requiring membership in the docker group so this can be safely ignored
22		Ensure centralized and remote logging is configured		▪ By default, Docker uses the json-file driver, which writes JSON-formatted logs to a container-specific file on the host where the container is running ▪ The local logging driver also writes logs to a local file, compressing them to save space on the disk ▪ Docker also provides built-in drivers for forwarding logs to various endpoints ▪ Regardless of which logging driver you choose, you can configure your container's logging in either blocking or non-blocking delivery mode. The mode you choose determines how the container prioritizes logging operations relative to its other tasks
23		Ensure live restore is enabled		▪ By specifying "live-restore": true in the daemon config, we allow containers to continue running when the Docker daemon is not. This improves container uptime during updates of the host system and other stability issues.
24		Ensure Userland Proxy is Disabled		▪ The "userland-proxy": false line fixes this warning. This disables the docker-proxy userland process that by default handles forwarding host ports to containers, and replaces it with iptables rules. If hairpin NAT is available, the userland proxy is not needed and should be disabled to reduce the attack surface of your host.
25		Ensure that a daemon-wide custom seccomp profile is applied if appropriate		▪ https://docs.docker.com/engine/security/seccomp/
26		Ensure that experimental features are not implemented in production		▪ By default, experimental flag is turned off. To see the experimental flag, check Docker version Experimental: false
27		Ensure containers are restricted from acquiring new privileges		▪ The "no-new-privileges": true line in the daemon config prevents privilege escalation from inside containers. This ensures that containers cannot gain new privileges using setuid or setgid binaries.
28		**Docker daemon configuration files**		

29		Ensure that the docker. service file ownership is set to root:root		▪ You should verify that the docker.service file ownership and group ownership are correctly set to root. ▪ The docker.service file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore be individually and group owned by the root user in order to ensure that it is not modified or corrupted by a less privileged user. **1.** Find out the file location: systemctl show -p FragmentPath docker.service **2.** If the file does not exist, this recommendation is not applicable. If the file does exist, you should execute the command below, including the correct file path, in order to set the ownership and group ownership for the file to root. For example, chown root:root /usr/lib/systemd/system/docker.service
30		Ensure that docker.service file permissions are appropriately set		▪ You should verify that the docker.service file permissions are either set to 644 or to a more restrictive value. ▪ The docker.service file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore not be writable by any other user other than root in order to ensure that it can not be modified by less privileged users. **1.** Find out the file location: systemctl show -p FragmentPath docker.service **2.** If the file does not exist, this recommendation is not applicable. If the file exists, execute the command below including the correct file path to set the file permissions to 644. For example, chmod 644 /usr/lib/systemd/system/docker.service
31		Ensure that docker.socket file ownership is set to root: root		▪ You should verify that the Docker socket file is owned by root and group owned by docker. ▪ The Docker daemon runs as root. The default Unix socket therefore must be owned by root. If any other user or process owns this socket, it might be possible for that non-privileged user or process to interact with the Docker daemon. Additionally, in this case a non-privileged user or process might be able to interact with containers which is neither a secure nor desired behavior. Additionally, the Docker installer creates a Unix group called docker. You can add users to this group, and in this case, those users would be able to read and write to the default Docker Unix socket. The membership of the docker group is tightly controlled by the system administrator. However, ff any other group owns this socket, then it might be possible for members of that group to interact with the Docker daemon. Such a group might not be as tightly controlled as the docker group. Again, this is not in line with good security practice. For these reason, the default Docker Unix socket file should be owned by root and group owned by docker to maintain the integrity of the socket file. ▪ Run the following command: chown root:docker /var/run/docker.sock This sets the ownership to root and group ownership to docker for the default Docker socket file.
32		Ensure that docker.socket file permissions are set to 644 or more restrictive		▪ You should verify that the file permissions on the docker.socket file are correctly set to 644 or more restrictively ▪ The docker.socket file contains sensitive parameters that may alter the behavior of the Docker remote API. It should therefore be writeable only by root in order to ensure that it is not modified by less privileged users **1.** Find out the file location: systemctl show -p FragmentPath docker.socket **2.** If the file does not exist, this recommendation is not applicable. If the file does exist, you should execute the command below, including the correct file path to set the file permissions to 644. For example, chmod 644 /usr/lib/systemd/system/docker.socket
33		Ensure that the /etc/docker directory ownership is set to root:root		▪ You should verify that the /etc/docker directory ownership and group ownership is correctly set to root. ▪ The /etc/docker directory contains certificates and keys in addition to various other sensitive files. It should therefore be individual owned and group owned by root in order to ensure that it can not be modified by less privileged users. ▪ To resolve this issue, run the following command: chown root:root /etc/docker This sets the ownership and group ownership for the directory to root
34		Ensure that /etc /docker directory permissions are set to 755 or more		▪ You should verify that the /etc/docker directory permissions are correctly set to 755 or more restrictively ▪ The /etc/docker directory contains certificates and keys in addition to various sensitive files. It should therefore only be writeable by root to ensure that it can not be modified by a less privileged user. ▪ Run the following command: chmod 755 /etc/docker This sets the permissions for the directory to 755 ▪ By default, the permissions for this directory are set to 755.
35		Ensure that registry certificate file ownership is set to root:root		▪ You should verify that all the registry certificate files, usually found under the /etc/docker/certs.d/<registry-name> directory, are individually owned and group owned by root ▪ The /etc/docker/certs.d/<registry-name> directory contains Docker registry certificates. These certificate files must be individually owned and group owned by root to ensure that less privileged users are unable to modify the contents of the directory. ▪ Execute the following command: chown root:root /etc/docker/certs.d/<registry-name>/* This sets the individual ownership and group ownership for the registry certificate files to root ▪ By default, the individual ownership and group ownership for registry certificate files is correctly set to root
36		Ensure that registry certificate file permissions are set to 444 or more		▪ You should verify that all the registry certificate files, usually found under /etc/docker/certs.d/<registry-name> directory, have permissions of 444 or are set more restrictively. ▪ The /etc/docker/certs.d/<registry-name> directory contains Docker registry certificates. These certificate files must have permissions of 444or more restrictive permissions in order to ensure that unprivileged users do not have full access to them. ▪ Run the following command: chmod 444 /etc/docker/certs.d/<registry-name>/* This sets the permissions for the registry certificate files to 444. ▪ By default, the permissions for registry certificate files might not be 444. The default file permissions are governed by the system or user specific umask values which are defined within the operating system itself.
37		Ensure that TLS CA certificate file ownership is set to root:root		▪ You should verify that the TLS CA certificate file, the file that is passed along with the --tlscacert parameter, is individually owned and group owned by root ▪ The TLS CA certificate file should be protected from any tampering. It is used to authenticate the Docker server based on a given CA certificate. It must be therefore be individually owned and group owned by root to ensure that it cannot be modified by less privileged users. ▪ Run the following command: chown root:root <path to TLS CA certificate file> This sets the individual ownership and group ownership for the TLS CA certificate file to root ▪ By default, the ownership and group-ownership for TLS CA certificate file is correctly set to root
38		Ensure that TLS CA certificate file permissions are set to 444 or more restrictively		▪ You should verify that the TLS CA certificate file, the file that is passed along with the --tlscacert parameter, has permissions of 444 or is set more restrictively. ▪ The TLS CA certificate file should be protected from any tampering. It is used to authenticate the Docker server based on a given CA certificate. It must therefore have permissions of 444, or more restrictive permissions to ensure that the file cannot be modified by a less privileged user. ▪ Run the following command: chmod 444 <path to TLS CA certificate file> This sets the file permissions on the TLS CA file to 444. ▪ By default, the permissions for the TLS CA certificate file might not be 444. The default file permissions are governed by the operating system or user specific umask values
39		Ensure that Docker server certificate file ownership is set to root:root		▪ You should verify that the Docker server certificate file, the file that is passed along with the --tlscert parameter, is individual owned and group owned by root ▪ The Docker server certificate file should be protected from any tampering. It is used to authenticate the Docker server based on the given server certificate. It must therefore be individually owned and group owned by root to prevent modification by less privileged users. ▪ Run the following command: chown root:root <path to Docker server certificate file> This sets the individual ownership and the group ownership for the Docker server certificate file to root. ▪ By default, the ownership and group-ownership for Docker server certificate file is correctly set to root.

40		Ensure that the Docker server certificate file permissions are set to 444 or more		▪ You should verify that the Docker server certificate file, the file that is passed along with the --tlscert parameter, has permissions of 444 or more restrictive permissions. ▪ The Docker server certificate file should be protected from any tampering. It is used to authenticate the Docker server based on the given server certificate. It should therefore have permissions of 444 to prevent its modification. ▪ Run the command below: chmod 444 <path to Docker server certificate file> This sets the file permissions of the Docker server certificate file to 444. ▪ By default, the permissions for the Docker server certificate file might not be 444. The default file permissions are governed by the operating system or user specific umask values.
41		Ensure that the Docker server certificate key file ownership is set to root: root		▪ You should verify that the Docker server certificate key file, the file that is passed along with the --tlskey parameter, is individually owned and group owned by root ▪ The Docker server certificate key file should be protected from any tampering or unneeded reads/writes. As it holds the private key for the Docker server certificate, it must be individually owned and group owned by root to ensure that it cannot be accessed by less privileged users. ▪ Run the following command: chown root:root <path to Docker server certificate key file> This sets the individual ownership and group ownership for the Docker server certificate key file to root
42		Ensure that the Docker server certificate key file permissions are set to 400		▪ You should verify that the Docker server certificate key file, the file that is passed along with the --tlskey parameter, has permissions of 400 ▪ The Docker server certificate key file should be protected from any tampering or unneeded reads. It holds the private key for the Docker server certificate. It must therefore have permissions of 400 to ensure that the certificate key file is not modified ▪ You should execute the following command: chmod 400 <path to Docker server certificate key file> This sets the Docker server certificate key file permissions to 400
43		Ensure that the Docker socket file ownership is set to root: docker		▪ You should verify that the Docker socket file is owned by root and group owned by docker ▪ The Docker daemon runs as root. The default Unix socket therefore must be owned by root. If any other user or process owns this socket, it might be possible for that non-privileged user or process to interact with the Docker daemon. Additionally, in this case a non-privileged user or process might be able to interact with containers which is neither a secure nor desired behavior. Additionally, the Docker installer creates a Unix group called docker. You can add users to this group, and in this case, those users would be able to read and write to the default Docker Unix socket. The membership of the docker group is tightly controlled by the system administrator. However, ff any other group owns this socket, then it might be possible for members of that group to interact with the Docker daemon. Such a group might not be as tightly controlled as the docker group. Again, this is not in line with good security practice. For these reason, the default Docker Unix socket file should be owned by root and group owned by docker to maintain the integrity of the socket file ▪ Run the following command: chown root:docker /var/run/docker.sock This sets the ownership to root and group ownership to docker for the default Docker socket file
44		Ensure that the Docker socket file permissions are set to 660 or more restrictively		▪ You should verify that the Docker socket file has permissions of 660 or are configured more restrictively. ▪ Only root and the members of the docker group should be allowed to read and write to the default Docker Unix socket. The Docker socket file should therefore have permissions of 660 or more restrictive permissions. ▪ Run the command chmod 660 /var/run/docker.sock This sets the file permissions of the Docker socket file to 660
45		Ensure that the daemon. json file ownership is set to root:root		▪ You should verify that the daemon.json file individual ownership and group ownership is correctly set to root. ▪ The daemon.json file contains sensitive parameters that could alter the behavior of the docker daemon. It should therefore be owned and group owned by root to ensure it can not be modified by less privileged users ▪ Run chown root:root /etc/docker/daemon.json This sets the ownership and group ownership for the file to root ▪ Default: This file may not be present on the system, and in that case, this recommendation is not applicable
46		Ensure that daemon.json file permissions are set to 644 or more restrictive		▪ You should verify that the daemon.json file permissions are correctly set to 644 or more restrictively ▪ The daemon.json file contains sensitive parameters that may alter the behavior of the docker daemon. Therefore it should be writeable only by root to ensure it is not modified by less privileged users. ▪ Run chmod 644 /etc/docker/daemon.json This sets the file permissions for this file to 644.
47		Ensure that the /etc/default /docker file ownership is set to root:root		▪ You should verify that the /etc/default/docker file ownership and group-ownership is correctly set to root ▪ The /etc/default/docker file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore be individually owned and group owned by root to ensure that it cannot be modified by less privileged users. ▪ Execute the following command: chown root:root /etc/default/docker This sets the ownership and group ownership of the file to root. ▪ Default:This file may not be present on the system, and in this case, this recommendation is not applicable
48		Ensure that the /etc /sysconfig /docker file ownership is set to root:root		▪ You should verify that the /etc/sysconfig/docker file individual ownership and group ownership is correctly set to root. ▪ The /etc/sysconfig/docker file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore be individually owned and group owned by root to ensure that it is not modified by less privileged users. ▪ Run the following command: chown root:root /etc/sysconfig/docker This sets the ownership and group ownership for the file to root. ▪ Default:This file may not be present on the system, and in this case, this recommendation is not applicable
49		Ensure that the /etc /sysconfig /docker file permissions are set to 644 or more restrictively		▪ You should verify that the /etc/sysconfig/docker file permissions are correctly set to 644 or more restrictively. ▪ The /etc/sysconfig/docker file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore be writeable only by root in order to ensure that it is not modified by less privileged users. ▪ Run the following command: chmod 644 /etc/sysconfig/docker This sets the file permissions for this file to 644. ▪ Default:This file may not be present on the system, and in this case, this recommendation is not applicable
50		Ensure that the /etc/default /docker file permissions are set to 644 or more restrictively		▪ You should verify that the /etc/default/docker file permissions are correctly set to 644 or more restrictively ▪ The /etc/default/docker file contains sensitive parameters that may alter the behavior of the Docker daemon. It should therefore be writeable only by root in order to ensure that it is not modified by less privileged users. ▪ Run the following command: chmod 644 /etc/default/docker This sets the file permissions for this file to 644 ▪ Default:This file may not be present on the system, and in this case, this recommendation is not applicable
51		**Container Images and Build File**		

52		Ensure that a user for the container has been created		▪ Containers should run as a non-root user. ▪ It is good practice to run the container as a non-root user, where possible. This can be done either via the USER directive in the Dockerfile or through gosu or similar where used as part of the CMD or ENTRYPOINT directives. ▪ Ensure that the Dockerfile for each container image contains USER <username or ID> In this case, the user name or ID refers to the user that was found in the container base image. If there is no specific user created in the container base image, then make use of the useradd command to add a specific user before the USER instruction in the Dockerfile. For example, add the below lines in the Dockerfile to create a user in the container: RUN useradd -d /home/username -m -s /bin/bash username USER username Note: If there are users in the image that are not needed, you should consider deleting them. After deleting those users, commit the image and then generate new instances of the containers. Alternatively, if it is not possible to set the USER directive in the Dockerfile, a script running as part of the CMD or ENTRYPOINT sections of the Dockerfile should be used to ensure that the container process switches to a non-root user. ▪ Running as a non-root user can present challenges where you wish to bind mount volumes from the underlying host. In this case, care should be taken to ensure that the user running the contained process can read and write to the bound directory, according to their requirements. ▪ By default, containers are run with root privileges and also run as the root user inside the container
53		Ensure that containers use only trusted base images		▪ Docker images might be based on open source Linux distributions, and bundle within them open source software and libraries. A recent state of open source security research conducted by Snyk found that the top most popular docker images contain at least 30 vulnerabilities.
54		Ensure that unnecessary packages are not installed in the container		▪ https://medium.com/@gdiener/how-to-build-a-smaller-docker-image-76779e18d48a
55		Ensure images are scanned and rebuilt to include security patches		▪ Use tools tool that scans your Docker images for security vulnerabilities
56		Ensure Content trust for Docker is Enabled		▪ http://docs.docker.oeynet.com/engine/security/trust/content_trust/ ▪ export DOCKER_CONTENT_TRUST=1
57		Ensure that HEALTHCHECK instructions have been added to container images		▪ You should add the HEALTHCHECK instruction to your Docker container images in order to ensure that health checks are executed against running containers ▪ An important security control is that of availability. Adding the HEALTHCHECK instruction to your container image ensures that the Docker engine periodically checks the running container instances against that instruction to ensure that containers are still operational. Based on the results of the health check, the Docker engine could terminate containers which are not responding correctly, and instantiate new ones ▪ You should follow the Docker documentation and rebuild your container images to include the HEALTHCHECK instruction ▪ By default, HEALTHCHECK is not set
58		Ensure update instructions are not use alone in the Dockerfile		
59		Ensure setuid and setgid permissions are removed		
60		Ensure that COPY is used instead of ADD in Dockerfiles		▪ https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
61		Ensure secrets are not stored in Dockerfiles		▪ https://docs.docker.com/engine/swarm/secrets/
62		Ensure only verified packages are are installed		
63		**Container Runtime**		
64		Ensure that, if applicable, an AppArmor Profile is enabled		▪ AppArmor is an effective and easy-to-use Linux application security system. It is available on some Linux distributions by default, for example, on Debian and Ubuntu. ▪ AppArmor protects the Linux OS and applications from various threats by enforcing a security policy which is also known as an AppArmor profile. You can create your own AppArmor profile for containers or use Docker's default profile. Enabling this feature enforces security policies on containers as defined in the profile. ▪ If AppArmor is applicable for your Linux OS, enable it. 1. Verify AppArmor is installed. 2. Create or import a AppArmor profile for Docker containers. 3. Enable enforcement of the policy. 4. Start your Docker container using the customized AppArmor profile. For example: docker run --interactive --tty --security-opt="apparmor:PROFILENAME" ubuntu /bin/bash Alternatively, Docker's default AppArmor policy can be used. ▪ The container will have the security controls defined in the AppArmor profile. It should be noted that if the AppArmor profile is misconfigured, this may cause issues with the operation of the container ▪ By default, the docker-default AppArmor profile is applied to running containers. This profile can be found at /etc/apparmor.d/docker

65		Ensure that, if applicable, SELinux security options are set		▪ SELinux is an effective and easy-to-use Linux application security system. It is available by default on some distributions such as Red Hat and Fedora ▪ SELinux provides a Mandatory Access Control (MAC) system that greatly augments the default Discretionary Access Control (DAC) model. You can therefore add an extra layer of safety to your containers by enabling SELinux on your Linux host ▪ If SELinux is applicable for your Linux OS, you should use it. 1. Set the SELinux State. 2. Set the SELinux Policy. 3. Create or import a SELinux policy template for Docker containers. 4. Start Docker in daemon mode with SELinux enabled. For example: docker daemon --selinux-enabled 5. Start your Docker container using the security options. For example, docker run --interactive --tty --security-opt label=level:TopSecret centos /bin/bash ▪ Any restrictions defined in the SELinux policy will be applied to your containers. It should be noted that if your SELinux policy is misconfigured, this may have an impact on the correct operation of the affected containers ▪ By default, no SELinux security options are applied on containers
66		Ensure that Linux kernel capabilities are restricted within containers		▪ By default, Docker starts containers with a restricted set of Linux kernel capabilities. This means that any process can be granted the required capabilities instead of giving it root access. Using Linux kernel capabilities, processes in general do not need to run as the root user ▪ Docker supports the addition and removal of capabilities. Remove all capabilities not required for the correct function of the container. Specifically, in the default capability set provided by Docker, the NET_RAW capability should be removed if not explicitly required, as it can give an attacker with access to a container the ability to create spoofed network traffic ▪ Execute the command docker run --cap-add={"Capability 1","Capability 2"} <Run arguments> <Container Image Name or ID> <Command> to add required capabilities. ▪ Execute the command docker run --cap-drop={"Capability 1","Capability 2"} <Run arguments> <Container Image Name or ID> <Command> to remove unneeded capabilities ▪ Alternatively, remove all the currently configured capabilities and then restore only the ones you specifically use: docker run --cap-drop=all --cap-add={"Capability 1","Capability 2"} <Run arguments> <Container Image Name or ID> <Command> ▪ Restrictions on processes within a container are based on which Linux capabilities are in force. Removal of the NET_RAW capability prevents the container from creating raw sockets which is good security practice under most circumstances, but may affect some networking utilities ▪ By default, the capabilities below are applied to containers: AUDIT_WRITE CHOWN DAC_OVERRIDE FOWNER FSETID KILL MKNOD NET_BIND_SERVICE NET_RAW SETFCAP SETGID SETPCAP SETUID SYS_CHROOT
67		Ensure that privileged containers are not used		▪ Using the --privileged flag provides all Linux kernel capabilities to the container to which it is applied and therefore overwrites the --cap-add and --cap-drop flags. For this reason, ensure that it is not used ▪ The --privileged flag provides all capabilities to the container to which it is applied, and also lifts all the limitations enforced by the device cgroup controller. As a consequence this the container has most of the rights of the underlying host. This flag only exists to allow for specific use cases (for example running Docker within Docker) and should not generally be used. ▪ Do not run containers with the --privileged flag. For example, do not start a container using the command docker run --interactive --tty --privileged centos /bin/bash ▪ If you start a container without the --privileged flag, it will not have excessive default capabilities ▪ Default False
68		Ensure sensitive host system directories are not mounted on containers		
69		Ensure sshd is not run within containers		
70		Ensure privileged ports are not mapped within containers		▪ The TCP/IP port numbers below 1024are considered privileged ports. Normal users and processes are not allowed to use them for various security reasons. Docker does, however allow a container port to be mapped to a privileged port. ▪ By default, if the user does not specifically declare a container port to host port mapping, Docker automatically and correctly maps the container port to one available in the 49153-65535 range on the host. Docker does, however, allow a container port to be mapped to a privileged port on the host if the user explicitly declares it. This is because containers are executed with NET_BIND_SERVICE Linux kernel capability which does not restrict privileged port mapping. The privileged ports receive and transmit various pieces of data which are security sensitive and allowing containers to use them is not in line with good security practice ▪ Do not map container ports to privileged host ports when starting a container. You should also ensure that there is no such container to host privileged port mapping declarations in the Dockerfile ▪ By default, mapping a container port to a privileged port on the host is allowed. Note: There might be certain cases where you want to map privileged ports, because if you forbid it, then the corresponding application has to run outside of a container. For example: HTTP and HTTPS load balancers have to bind 80/tcp and 443/tcp respectively. Forbidding to map privileged ports effectively forbids from running those in a container, and mandates using an external load balancer. In such cases, those containers instances should be marked as exceptions for this recommendation.
71		Ensure that only needed ports are open on the container		
72		Ensure that the host's network namespace is not shared		▪ When the networking mode on a container is set to --net=host, the container is not placed inside a separate network stack. Effectively, applying this option instructs Docker to not containerize the container's networking. The consequence of this is that the container lives “outside” in the main Docker host and has full access to its network interfaces ▪ Selecting this option is potentially dangerous. It allows the container process to open reserved low numbered ports in the way that any other root process can. It also allows the container to access network services such as D-bus on the Docker host. A container process could potentially carry out undesired actions, such as shutting down the Docker host. This option should not be used unless there is a very specific reason for enabling it ▪ You should not pass the –net=host option when starting any container. ▪ By default, containers connect to the Docker bridge when starting and do not run in the context of the host's network stack.
73		Ensure that the memory usage for containers is limited		▪ https://docs.docker.com/config/containers/resource_constraints/

74		Ensure that CPU priority is set appropriately on containers		▪ By default, all containers on a Docker host share resources equally. By using the resource management capabilities of the Docker host you can control the host CPU resources that a container may consume. ▪ By default, CPU time is divided between containers equally. If you wish to control available CPU resources amongst container instances, you can use the CPU sharing feature. CPU sharing allows you to prioritize one container over others and prevents lower priority containers from absorbing CPU resources which may be required by other processes. This ensures that high priority containers are able to claim the CPU runtime they require. ▪ You should manage the CPU runtime between your containers dependent on their priority within your organization. To do so, start the container using the --cpu-shares argument. For example, you could run a container as docker run --interactive --tty --cpu-shares 512 centos /bin/bash The container is started with CPU shares of 50% of what other containers use. So if the other container has CPU shares of 80%, this container will have CPU shares of 40%. Every new container will have 1024 shares of CPU by default. However, this value is shown as 0 if you run the command mentioned in the audit section Alternatively: 1. Navigate to the /sys/fs/cgroup/cpu/system.slice/ directory. 2. Check your container instance ID using docker ps. 3. Inside the above directory (in step 1), call a directory. For example: docker-<Instance ID>.scope or docker-4acae729e8659c6be696ee35b2237cc1fe4edd2672e9186434c5116e1a6fbed6.scope. Navigate to this directory. 4. You will find a file named cpu.shares. Execute cat cpu.shares. This will always give you the CPU share value based on the system. Even if there are no CPU shares configured using the -c or --cpu-shares argument in the docker run command, this file will have a value of 1024. If you set one containers CPU shares to 512 it will receive half of the CPU time compared to the other containers. So if you take 1024 as 100% you can then derive the number that you should set for respective CPU shares. For example, use 512 if you want to set it to 50% and 256 if you want to set it 25% ▪ If you do not correctly assign CPU thresholds, the container process may run out of resources and become unresponsive. If CPU resources on the host are not constrained, CPU shares do not place any restrictions on individual resources ▪ By default, all containers on a Docker host share their resources equally. No CPU shares are enforced
75		Ensure that the container's root filesystem is mounted as read only		▪ The container's root filesystem should be treated as a 'golden image' by using Docker run's --read-only option. This prevents any writes to the container's root filesystem at container runtime and enforces the principle of immutable infrastructure. ▪ Enabling this option forces containers at runtime to explicitly define their data writing strategy to persist or not persist their data. This also reduces security attack vectors since the container instance's filesystem cannot be tampered with or written to unless it has explicit read-write permissions on its filesystem folder and directories. ▪ Add a --read-only flag at a container's runtime to enforce the container's root filesystem being mounted as read only. For example, docker run <Run arguments> --read-only <Container Image Name or ID> <Command> Enabling the --read-only option at a container's runtime should be used by administrators to force a container's executable processes to only write container data to explicit storage locations during its lifetime. Examples of explicit storage locations during a container's runtime include, but are not limited to: 1. Using the --tmpfs option to mount a temporary file system for non-persistent data writes. docker run --interactive --tty --read-only --tmpfs "/run" --tmpfs "/tmp" centos /bin/bash 2. Enabling Docker rw mounts at a container's runtime to persist container data directly on the Docker host filesystem. For example, docker run --interactive --tty --read-only -v /opt/app/data:/run/app/data:rw centos /bin/bash 3. Utilizing the Docker shared-storage volume plugin for Docker data volume to persist container data. For example, docker volume create -d convoy --opt o=size=20GB my-named-volume docker run --interactive --tty --read-only -v my-named-volume:/run/app/data centos /bin/bash 4. Transmitting container data outside of the Docker controlled area during the container's runtime for container data in order to ensure that it is persistent. Examples include hosted databases, network file shares and APIs. ▪ Enabling --read-only at container runtime may break some container OS packages if a data writing strategy is not defined. You should define what the container's data should and should not persist at runtime in order to decide which strategy to use. Example: Enable use --tmpfs for temporary file writes to /tmp Use Docker shared data volumes for persistent data writes ▪ By default, a container has its root filesystem writeable, allowing all container processes to write files owned by the container's actual runtime user
76		Ensure that incoming container traffic is bound to a specific host interface		▪ https://docs.docker.com/config/containers/container-networking/
77		Ensure that the 'on-failure' container restart policy is set to '5'		▪ By using the --restart flag in the docker run command you can specify a restart policy for how a container should or should not be restarted on exit. You should choose the on-failure restart policy and limit the restart attempts to 5 ▪ If you indefinitely keep trying to start the container, it could possibly lead to a denial of service on the host. It could be an easy way to do a distributed denial of service attack especially if you have many containers on the same host. Additionally, ignoring the exit status of the container and always attempting to restart the container, leads to non-investigation of the root cause behind containers getting terminated. If a container gets terminated, you should investigate on the reason behind it instead of just attempting to restart it indefinitely. You should use the on-failure restart policy to limit the number of container restarts to a maximum of 5 attempts ▪ If you wish a container to be automatically restarted, use docker run --detach --restart=on-failure:5 nginx ▪ If this option is set, a container will only attempt to restart itself 5 times ▪ By default, containers are not configured with restart policies
78		Ensure that the host's process namespace is not shared		▪ The Process ID (PID) namespace isolates the process ID space, meaning that processes in different PID namespaces can have the same PID. This creates process level isolation between the containers and the host. ▪ PID namespace provides separation between processes. It prevents system processes from being visible, and allows process ids to be reused including PID 1. If the host's PID namespace is shared with containers, it would basically allow these to see all of the processes on the host system. This reduces the benefit of process level isolation between the host and the containers. Under these circumstances a malicious user who has access to a container could get access to processes on the host itself, manipulate them, and even be able to kill them. This could allow for the host itself being shut down, which could be extremely serious, particularly in a multi-tenanted environment. You should not share the host's process namespace with the containers running on it ▪ You should not start a container with the --pid=host argument. For example, do not start a container with the command: docker run --interactive --tty --pid=host centos /bin/bash ▪ Container processes cannot see processes on the host system. In certain circumstances, you may want your container to share the host's process namespace. For example, you could build a container containing debugging tools such as strace or gdb, and want to use these tools when debugging processes on the host. If this is desired, then share specific host processes using the -p switch. For example: docker run --pid=host rhel7 strace -p 1234 ▪ By default, all containers have the PID namespace enabled and the therefore the host's process namespace is not shared with its containers
79		Ensure that the host's IPC namespace is not shared		▪ IPC (POSIX/SysV IPC) namespace provides separation of named shared memory segments, semaphores and message queues. The IPC namespace on the host should therefore not be shared with containers and should remain isolated ▪ The IPC namespace provides separation of IPC between the host and containers. If the host's IPC namespace is shared with the container, it would allow processes within the container to see all of IPC communications on the host system. This would remove the benefit of IPC level isolation between host and containers. An attacker with access to a container could get access to the host at this level with major consequences. The IPC namespace should therefore not be shared between the host and its containers ▪ Do not start a container with the --ipc=host argument. For example, do not start a container with the command docker run --interactive --tty --ipc=host centos /bin/bash ▪ Shared memory segments are used in order to accelerate interprocess communications, commonly in high-performance applications. If this type of application is containerized into multiple containers, you might need to share the IPC namespace of the containers in order to achieve high performance. Under these circumstances, you should still only share container specific IPC namespaces and not the host IPC namespace. A container's IPC namespace can be shared with another container. For example, docker run --interactive --tty --ipc=container:e3a7a1a97c58 centos /bin/bash ▪ By default, all containers have their IPC namespace enabled and host IPC namespace is not shared with any container

80		Ensure that host devices are not directly exposed to containers		▪ https://docs.docker.com/engine/security/
81		Ensure that the default ulimit is overwritten at runtime if needed		▪ https://docs.docker.com/engine/reference/commandline/dockerd/
82		Ensure mount propagation mode is not set to shared		▪ https://docs.docker.com/engine/reference/run/
83		Ensure that the host's UTS namespace is not shared		▪ UTS namespaces provide isolation between two system identifiers: the hostname and the NIS domain name. It is used to set the hostname and the domain which are visible to running processes in that namespace. Processes running within containers do not typically require to know either the hostname or the domain name. The UTS namespace should therefore not be shared with the host ▪ Sharing the UTS namespace with the host provides full permission for each container to change the hostname of the host. This is not in line with good security practice and should not be permitted ▪ You should not start a container with the --uts=host argument. For example, do not start a container using the command docker run --rm --interactive --tty --uts=host rhel7.2 ▪ By default, all containers have the UTS namespace enabled and the host UTS namespace is not shared with any containers
84		Ensurethe default seccomp profile is not Disabled		▪ Seccomp filtering provides a means for a process to specify a filter for incoming system calls. The default Docker seccomp profile works on a whitelist basis and allows for a large number of common system calls, whilst blocking all others. This filtering should not be disabled unless it causes a problem with your container application usage ▪ A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. Most of applications do not need all these system calls and would therefore benefit from having a reduced set of available system calls. Having a reduced set of system calls reduces the total kernel surface exposed to the application and thus improvises application security. ▪ By default, seccomp profiles are enabled. You do not need to do anything unless you want to modify and use a modified seccomp profile ▪ With Docker 1.10 and greater, the default seccomp profile blocks syscalls, regardless of --cap-add passed to the container. You should create your own custom seccomp profile in such cases. You can also disable the default seccomp profile by passing --security-opt=seccomp:unconfined on docker run ▪ Default: When you run a container, it uses the default profile unless you override it with the --security-opt option
85		Ensure that docker exec commands are not used with the privileged option		▪ https://docs.docker.com/engine/reference/commandline/exec/
86		Ensure that docker exec commands are not used with the user=root option		▪ https://docs.docker.com/engine/reference/commandline/exec/
87		Ensure that cgroup usage is confirmed		▪ It is possible to attach to a particular cgroup when a container is instantiated. Confirming cgroup usage would ensure that containers are running in defined cgroups. ▪ System administrators typically define cgroups in which containers are supposed to run. If cgroups are not explicitly defined by the system administrator, containers run in the docker cgroup by default. At run time, it is possible to attach a container to a different cgroup other than the one originally defined. This usage should be monitored and confirmed, as by attaching to a different cgroup, excess permissions and resources might be granted to the container and this can therefore prove to be a security risk. ▪ You should not use the --cgroup-parent option within the docker run command unless strictly required. ▪ By default, containers run under docker cgroup
88		Ensure that the container is restricted from acquiring additional privileges		▪ You should restrict the container from acquiring additional privileges via SUID or SGID bits ▪ A process can set the no_new_priv bit in the kernel and this persists across forks, clones and execve. The no_new_priv bit ensures that the process and its child processes do not gain any additional privileges via SUID or SGID bits. This reduces the danger associated with many operations because the possibility of subverting privileged binaries is lessened ▪ Start your container with the options docker run --rm -it --security-opt=no-new-privileges ubuntu bash ▪ The no_new_priv option prevents LSMs like SELinux from allowing processes to acquire new privileges ▪ By default, new privileges are not restricted
89		Ensure that container health is checked at runtime		▪ If the container image does not have an HEALTHCHECK instruction defined, you should use the --health-cmd parameter at container runtime to check container health ▪ If the container image you are using does not have a pre-defined HEALTHCHECK instruction, use the --health-cmd parameter to check container health at runtime. Based on the reported health status, remedial actions can be taken if necessary ▪ You should run the container using the --health-cmd parameter. For example, docker run -d --health-cmd='stat /etc/passwd || exit 1' nginx ▪ By default, health checks are not carried out at container runtime
90		Ensure that Docker commands always make use of the latest version of their image		▪ https://docs.docker.com/engine/reference/commandline/build/
91		Ensure that the PIDs cgroup limit is used		▪ You should use the --pids-limit flag at container runtime ▪ Attackers could launch a fork bomb with a single command inside the container. This fork bomb could crash the entire system and would require a restart of the host to make the system functional again. Using the PIDs cgroup parameter –pids-limit would prevent this kind of attack by restricting the number of forks that can happen inside a container within a specified time frame ▪ Use --pids-limit flag with an appropriate value when launching the container. For example, docker run -it --pids-limit 100 <Image_ID> In the above example, the number of processes allowed to run at any given time is set to 100. After a limit of 100 concurrently running processes is reached, Docker would restrict any new process creation. ▪ Set the PIDs limit value as appropriate. Incorrect values might leave containers unusable ▪ The Default value for --pids-limit is 0 which means there is no restriction on the number of forks. Note that the PIDs cgroup limit works only for kernel versions 4.3 and higher.
92		Ensure that Docker's default bridge docker0 is not used		▪ https://docs.docker.com/engine/reference/commandline/dockerd/

93		Ensure that the host's user namespaces are not shared		▪ You should not share the host's user namespaces with containers running on it ▪ User namespaces ensure that a root process inside the container will be mapped to a non-root process outside the container. Sharing the user namespaces of the host with the container does not therefore isolate users on the host from users in the containers ▪ Do not share user namespaces between host and containers. For example, do not run the command docker run --rm -it --userns=host ubuntu bash ▪ By default, the host user namespace is shared with containers unless user namespace support is enabled
94		Ensure that the Docker socket is not mounted inside any containers		▪ The Docker socket docker.sock should not be mounted inside a container ▪ If the Docker socket is mounted inside a container it could allow processes running within the container to execute Docker commands which would effectively allow for full control of the host. ▪ You should ensure that no containers mount docker.sock as a volume. ▪ By default, docker.sock is not mounted inside containers
95		**Docker Security Operations**		
96		Ensure swarm mode is not Enabled, if not needed		▪ https://docs.docker.com/engine/swarm/swarm-mode/
97		Ensure that the minimum number of manager nodes have been created in a swarm		▪ https://docs.docker.com/engine/swarm/admin_guide/
98		Ensure that swarm services are bound to a specific host interface		▪ https://docs.docker.com/engine/swarm/services/
99		Ensure that all Docker swarm overlay networks are encrypted		▪ https://docs.docker.com/network/overlay/
100		Ensure that Docker's secret management commands are used for managing secrets in a swarm cluster		▪ https://docs.docker.com/engine/swarm/secrets/
101		Ensure that swarm manager is run in auto-lock mode		▪ https://docs.docker.com/engine/swarm/swarm_manager_locking/
102		Ensure that the swarm manager auto-lock key is rotated periodically		▪ https://docs.docker.com/engine/swarm/swarm_manager_locking/
103		Ensure that node certificates are rotated as appropriate		▪ https://kubernetes.io/docs/tasks/tls/manual-rotation-of-ca-certificates/
104		Ensure that CA certificates are rotated as appropriate		▪ https://kubernetes.io/docs/tasks/tls/manual-rotation-of-ca-certificates/
105		Ensure that management plane traffic is separated from data plane traffic		▪ use different systems
106				