Container Sicherheit für Kritische Infrastrukturen

Bruno Kreyßig

04.12.2022

Contents

1. Einleitung

Containerisierte Anwendungen haben mit der Einführung von Docker (2013) einen großen Aufschwung erlebt. Schnelle Deployment-Zeiten, hohe Portabilität nach dem Prinzip “Build once, run anywhere” und Ressourceneffizienz machen die neue Technologie attraktiver als herkömmliche Virtualisierung. Insbesondere der geringe Speicherbedarf eines einzelnen Containers ermöglicht erst praktikable MicroService-Architekturen.

Während ein Großteil der Privatwirtschaft bereits seit einigen Jahren von den Vorzügen der Containerisierung profitiert, haben Betreiber Kritischer Infrastrukturen , mangels Sicherheitsvorgaben des BSI, sich von der Thematik fern gehalten. Wie sich in der Ausarbeitung herausstellen wird, sind Container nur oberflächlich mit Virtuellen Maschinen vergleichbar.

Erst in der 2022 Version des IT-Grundschutzkompendiums wurden Sicherheitsvorgaben für Containerisierung in einem Baustein (SYS.1.6) erfasst. Umsetzungshinweise gibt es bisher nicht.

Das Ziel dieser Recherchearbeit liegt folglich einerseits in der Erarbeitung der technischen Besonderheiten einer Container-Infrastruktur (s. Kapitel 2) und deren Sicherheitsimplikationen, soll andererseits gleichzeitig die Anforderungen des neuen BSI-Bausteins berücksichtigen und erfüllen. Dennoch bleibt das primäre Anliegen eine operative IT-Sicherheit für eine Container-Infrastruktur zu erarbeiten (Kapitel 3-5). In Kapitel 6 werden die Baustein-Anforderungen, bezugnehmend auf die bisherigen Abschnitte, analysiert und abgebildet.

Im Rahmen der Arbeit werden sowohl Aspekte der Sicherheit von Containern zur Laufzeit (Kapitel 3), als auch deren sichere Bereitstellung im Rahmen der Software-Supply-Chain (Kapitel 4) untersucht. Zusätzlich verdeutlicht Kapitel 5 in Referenzangriffsszenarien, wie böswillige Akteure undurchdachte Container-Infrastrukturen ausnutzen könnten. Abschließend werden in Kapitel 7 eine Reihe hilfreicher Tools mit deren Anwendungsmöglichkeiten aufgeführt.

2. Grundlegende Konzepte

Dieses Kapitel gibt einen kurzen Überblick zu den fundamentalen Prinzipien und der technischen Implementierung, die mit der Containerisierung von Anwendungen einhergeht. Leser, die mit Linux *Capabilities*, *cgroups* und *namespaces* vertraut sind können mit 3. Containersicherheit zur Laufzeit[1] fortfahren.

In seinem Kern ist ein Container nur ein Prozess, der auf einem Linux Kernel läuft. Gerade deshalb sind Container so effizient mit Deployment-Zeiten im Millisekundenbereich und in ihrer Ressourcennutzung. Das steht im harten Kontrast zu Virtuellen Maschinen, die jeweils mit einem eigenen Betriebssystem gestartet werden.

Aus dem Winkel der IT-Sicherheit betrachtet ergibt sich mit Containern wiederum eine wesentlich höhere Komplexität zur Erreichung der vermutlich wichtigsten Grundbedingung für den Einsatz containerisierter Anwendungen - die Isolation. Ein Hypervisor kommt mit gerade einmal 50.000 Zeilen Quelltext aus und hat eine wesentlich einfachere Aufgabe bzw. ist es nicht einmal vorgesehen, dass VMs in irgendeiner Weise direkt miteinander interagieren (Netzwerkverbindung ausgenommen). [Rice20], [Xen19]

Je nach Version des verwendeten Linux Kernels besteht dieser aus 20 - 35 Millionen Zeilen Quelltext. Es ist durchaus möglich und unter Umständen auch gewollt Containern geteilte Ressourcen zur Verfügung zu stellen. Prozesse können i.A. auch andere Prozesse sehen. Zusammengefasst bedeutet das, dass mit Zunahme der Konfigurationsmöglichkeiten das Potenzial für eine Schwachstelle im Kernel Code oder eine Fehlkonfiguration steigt. [Rice20], [WiLK]

Aus diesem Grund werden zunächst die Mechanismen zur Erfüllung der Container Isolation in Kapitel 2.1 vorgestellt (weitergehende Härtungsmaßnahmen s. 3.). Kapitel 2.2 befasst sich mit der Unveränderlichkeit von Containern, einer weiteren wünschenswerten Eigenschaft von Containern und in 2.3 wird ein grober Einblick in die Terminologie von Kubernetes, der marktführenden Container-Orchestrierungslösung gegeben.

[1] Doc/03_RuntimeContainerSecurity.md

2.1 Container Isolation

Eine funktionierende Container Isolation setzt voraus, dass ein Container keine anderen auf dem gleichen Kernel laufenden Container oder sonstige Host-Prozesse negativ beeinflussen kann. Zusammengefasst lässt sich diese erreichen durch

Linux (Befehl/Konzept)	Zweck	Beschreibung
cgroups	Ressourcenbeschränkung	Limitierung des Speicher-, Netzwerk-, CPU-Verbrauchs oder auch Beschränkung der maximalen Anzahl an Kindprozessen. *Cgroups* werden in `/sys/fs/cgroup` erstellt. Das Anlegen eines neuen Ordners in bspw. `/sys/fs/cgroup/memory` und Schreiben der PID im darin befindlichen `cgroup.procs` bindet einen Prozess an diese *cgroup*.
namespaces	Sichtbarkeitsbeschränkung	Mit dem Befehl `unshare` lassen sich Kindprozesse erstellen, die nicht den *namespace* des Elternprozesses übernehmen. Hierüber erhält ein Prozess (Container) u.a. ein vom Host unabhängiges Netzwerk-Interface, Prozess-Nummerierung und Mount-Points

Linux (Befehl/Konzept)	Zweck	Beschreibung
chroot	Sichtbarkeitsbeschränkung	*Namespaces* alleine reichen nicht aus, um eine vollständige Sichtbarkeitsbeschränkung zu erreichen. Das liegt daran, dass Prozesse nach wie vor aus den Verzeichnissen `/proc` und `/mnt` lesen. Mit `chroot` wird das Wurzelverzeichnis eines Prozesses verlegt, sodass dieser nicht mehr auf die Verzeichnisse des Hosts zugreifen kann. In diesem Schritt ist jedoch zugleich der `/bin`-Ordner unsichtbar geworden, wodurch innerhalb des Prozesses keine weiteren Befehle mehr ausgeführt werden können. Genau hierfür verwendet man *Container Images*, eine rudimentäre Verzeichnisstruktur, welche im Idealfall nur die notwendigen Befehle zur Ausführung der darin befindlichen Anwendung enthält.

Linux (Befehl/Konzept)	Zweck	Beschreibung
capabilities	Fähigkeitsbeschränkung	*Capabilities* limitieren die *Syscalls*, die ein Prozess ausführen darf. Eine Liste gefährlicher Capabilities kann [HTCap] entnommen werden. Darunter bspw. `CAP_SYS_ADMIN` mit zahlreichen administrativen Berechtigungen, die trivial für Privilege Escalation genutzt werden können.

2.2 Immutable Containers

Bei einer Gruppe von Containern handelt es sich genau dann um *Immutable Container*, wenn diese vom gleichen Image stammen und zur Laufzeit identisches Verhalten aufweisen. Somit sollten Container unter keinen Umständen:

- neue Code-Versionen oder Abhängigkeiten zur Laufzeit herunterladen
  - das ist sowohl aus Sicherheitsgründen, als auch aus Gründen der Wartbarkeit und Fehlerreproduzierbarkeit untragbar.
- Prozesse starten, die nicht für die Ausführung der Anwendung benötigt werden
  - das schließt insbesondere die schlechte Praxis zu Wartungszwecken eine Shell auf einem Container zu starten mit ein.

Unter der Voraussetzung eines *Immutable Container* müssen Schwachstellenscans (bzw. Image Scans) nur in der Container Registry ausgeführt werden.

Viele Container-Infrastrukturen nehmen die Unveränderlichkeit von Containern als Prämisse an, ohne diese technisch garantieren zu können. Zu diesem Zweck sollten *Container Image Profiles* eingesetzt werden (s. Kapitel 3). Hierdurch offenbart sich ein maßgeblicher Sicherheitsgewinn bei der Entwicklung von Microservice-Architekturen gegenüber Monolithen. Es ist viel einfacher möglich festzustellen, was ein Microservice (Container) machen soll und darf und dementsprechend genau diese Aktivitäten in einer Whitelist zu erfassen. [Rice20]

2.3 Container Runtime

Der Begriff *Container Runtime* ist überladen und wird oftmals synonym für high-level *Container Runtimes* (wie containerd, CRI-O) verwendet, welche wiederum eine low-level *Container Runtime* (in der Regel runC) einbeziehen.

Die high-level *Container Runtime* ist für das herunterladen von Container Images aus einer Registry, die Verwaltung von Mounts und Speichern, sowie das Ausführen von Containern über eine OCI-konforme *low-level Container Runtime* zuständig. Anschließend erstellt und führt die low-level *Container Runtime* die containerisierten Prozesse aus. [Dono21]

Um die dafür notwendigen Aufgaben (s. 2.1) zu erledigen, muss eine *Container Runtime* zwingend mit Root-Rechten laufen.

Was nun noch fehlt ist eine Schnittstelle zur Interaktion mit der *Container Runtime*, also das *Container Runtime Interface* (CRI). Das CRI kann in Form einer Kommandozeile (`docker`) oder über eine API gegeben sein. Genau hierein fügt sich die Implikation, dass unprivilegierte Nutzer mit Zugriff auf das CRI faktisch privilegierte Nutzer sind (s. Kapitel 3). [Rice20]

2.4 Container-Orchestrierung

Sobald etwas komplexere Microservice-Architekturen oder einfacher ausgedrückt der Bedarf an containerisierten Anwendungen im Unternehmen zunimmt, gelangt man aus administrativer Sicht schnell an Grenzen. Das wird unter anderem deutlich bei der Aktualisierung einer laufenden Container-Umgebung, wo das Vorgehen schematisch wie folgt abläuft:

```
1 docker ps
2 docker stop <containerName>
3 docker rm <containerName>
4 docker run <neuesImage> --name <containerName>
5 # Vorgehen für jeden Container im Deployment wiederholen
```

Mit Kubernetes ist lediglich eine Anpassung in der zugehörigen `deployment.yaml` vorzunehmen und diese anschließend mit `kubectl apply -f deployment.yaml` auszurollen. Die Orchestrierung garantiert, dass das zugrundeliegende redundant aufgesetzte Deployment während des Rollouts stets laufende Container enthält. Des Weiteren erkennt und behandelt Kubernetes automatisch abgestürzte Container und startet diese wieder.

Zur Realisierung solcher Aufgaben basiert Kubernetes auf einer komplexen Architektur aus **Nodes** und darauf laufenden Diensten (s. Abbildung). Auf den **Nodes** laufen letztendlich die **Pods**, eine Abstraktionsschicht für mehrere Container im gleichen *namespace*.

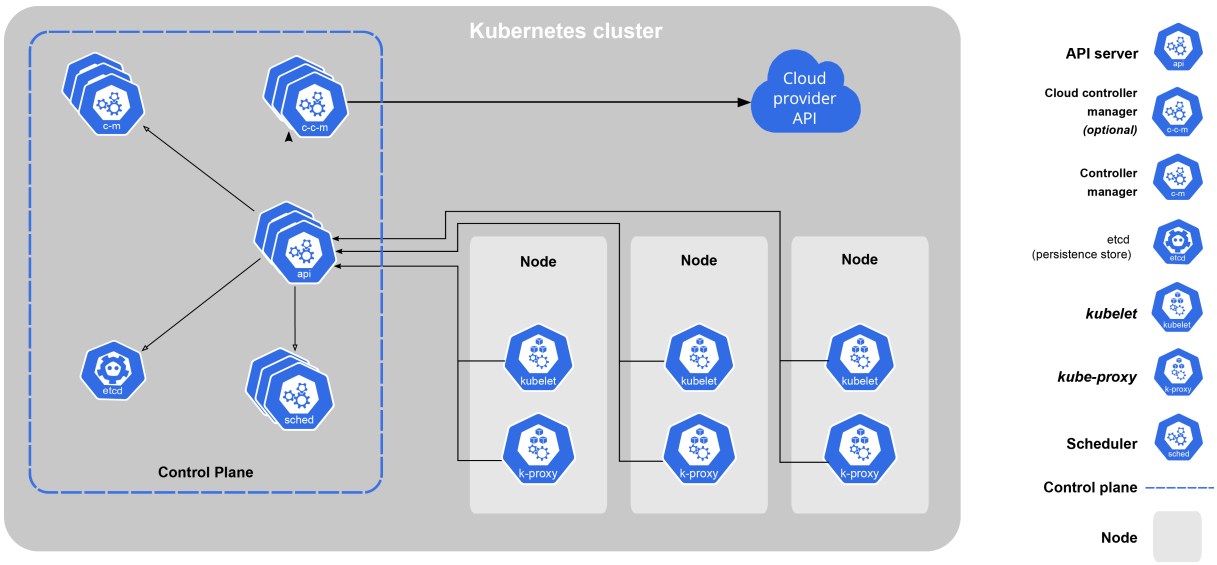

Figure 1: Abbildung: Kubernetes Node-Architektur [K8S_Arc]

In der Control Plane bzw. auf dem sogenannten Master-Node laufen folgende Dienste:

- **API-Server**: Schnittstelle zur Interaktion mit dem Cluster
- **Controller Manager**: Monitoring auf Abweichungen vom Soll-Zustand des Clusters und Propagierung von Maßnahmen zur Wiederherstellung an die Worker-Nodes
- **Scheduler**: Verteilung neuer Pods auf Worker-Nodes basierend auf deren Auslastung
- **etcd**: Key-Value-Store, welcher den Zustand des Clusters abspeichert, sodass der Controller Manager Änderungen erkennen kann

Controller Manager und Scheduler interagieren mit dem **kubelet**-Dienst auf den Worker-Nodes. Dieser setzt die angefragten Änderungen in der vorliegenden **Container Runtime** (bspw. containerd oder CRI-O) um. Abschließend läuft auf den Worker-Nodes noch der **kube-proxy**, welcher die Netzwerkregeln zur Kommunikation der Pods untereinander (mittels *services*) oder mit der Außenwelt (mittels *ingress*) geltend macht. Dabei greift der kube-proxy auf den Paketfilter des Betriebssystems zurück, also bei Unix-Derivaten *iptables*. [K8S_Arc]

Auf Grundlage dieser Architektur können Deployments basierend auf einer Vielzahl von Konzepten wie *ReplicaSets*, *Services*, *Ingress*, *Volumes*, *PersistentVolumeClaims*, *Secrets*, *ConfigMaps*, *LimitRanges*, *ResourceQuotas*, *Namespaces* und *Policies* konfiguriert werden. *Namespaces* haben im Kontext von Kubernetes übrigens nichts mit den zuvor erwähnten Linux-Namespaces zu tun. Kubernetes-Namespaces dienen der Isolierung von Nutzerrechten und Ressourcen innerhalb eines Clusters.

Von besonderer Relevanz für die Sicherheit des Clusters sind dabei *LimitRanges* und *ResourceQuotas*, welche die Ressourcenbeschränkungs-Konzept von *cgroups* auf Pod- bzw. Namespace-Ebene

durchsetzen.

Falls ein tiefergreifendes Verständnis für die Komponenten eines Kubernetes-Clusters erforderlich ist, kann in der offiziellen Kubernetes-Dokumentation nachgelesen werden. Die folgenden Kapitel beschreiben ausschließlich Möglichkeiten innerhalb des Clusters Sicherheitsmaßnahmen zur Gewährleistung der Container-Isolation und von *Immutable Containers* einzubringen.

3. Containersicherheit zur Laufzeit

In diesem Kapitel werden einige Mechanismen zur Gewährleistung der Containerisolation und Unveränderlichkeit von Containern zur Laufzeit betrachtet. Letzeres ist auch unter dem Begriff *Drift Prevention* bekannt. Die Analyse erfolgt anhand eines lokalen Minikube-Clusters (s. Kapitel 7.2 zur Referenz).

Seit einigen Jahren ist es möglich mit dem extended Berkeley Packet Filter (eBPF) beliebige Events im Kernel auszuwerten und Funktionalität basierend auf diesen hinzuzufügen. Während es komplex ist eigenhändig eBPF-Programme für den Kernel zu schreiben, gibt es bereits eine Vielzahl von Programmen, welche Detailwissen über den Kernel selbst über eine abstrahierte Schnittstelle verbergen. Das Open-Source-Projekt *Cilium* bietet ein breites Anwendungsspektrum für eBPF (s. Kapitel 3.3).

Dennoch lohnt es sich einige herkömmliche Ansätze zur Berechtigungsrestriktion, wie *seccomp*, *AppArmor* und *SELinux* anzusehen (Kapitel 3.2 und 3.3).

3.1 Grundlegende Konfiguration

3.2 Secure Computing Mode (seccomp)

Der Linux-Kernel stellt mit Secure Computing Mode (seccomp) ein Feature zur Beschränkung der von einem Prozess ausführbaren syscalls bereit.

- default docker seccomp Profile –> 44 syscalls blockiert
- 2022 ca. 400 syscalls
- laut aquasec benötigt Container zw. 40 und 70 syscalls –> default Profile unzureichend
- docker seccomp json dokument
  - SCMP_ACT_KILL, SCMP_ACT_TRAP, SCMP_ACT_ERRNO, trace, allow, log
- strace verwenden um syscalls eines containers zu profilen `strace -qc time`, `strace -c -f -S name time 2>1&1 1>/dev/null | tail -n +3 | head -n -2 | awk '{print $(NF)}'`

```
# cyberbit training
sudo docker run --security-opt seccomp=/home/cyberuser/profiles/
    violation.json --name cyberbit -dit busybox:latest

strace -c -f -S name <command line name> 2>&1 1>/dev/null | tail -n +3
    | head -n -2 | awk '{print $(NF)}'
```

3.3 Linux Security Modules

3.4 Extended Berkeley Packet Filter mit Cilium

- Konfiguration
- Netzwerk
- Side-Car Container
- im Kontext eines Kubernetes-Clusters

4. Container Images

Container Images bilden die Verzeichnisstruktur ab, auf die eine containerisierte Anwendung zur Gewährleistung von dessen Funktionalität zurückgreift. Es ist dabei gängige Praxis die Abhängigkeiten der Anwendungen (bspw. Laufzeitumgebung) aus einem Basis Image zu beziehen und dieses gemeinsam mit der Anwendung in ein neues Container Image zu bündeln (**build**).

Das fertige Container Image wird anschließend entweder direkt ausgerollt oder zunächst in einer Container Registry abgelegt (s. Kapitel 4.2). Container Images sind somit das zentrale Artefakt in der CI/CD-Pipeline und sind in jedem Schritt bis hin zum Deployment in ein Cluster Bedrohungen ausgesetzt (s. Abbildung). Die Kürzel CICD-SEC-x nehmen Bezug auf die **Top 10 OWASP CI/CD Security Risks**. Von besonderer Bedeutung ist das Risiko CICD-SEC-1, welches bei Nichtbeachtung Angreifern ermöglicht, von einem beliebigen System im Build-Prozess aus, schadhaften Quelltext unkontrolliert in die Produktion auszurollen. [OWASPCD], [Rice20]

Dementsprechend sind den Verifikationsmaßnahmen von Container-Images ein hohes Gewicht beizumessen. In den folgenden Unterkapiteln werden 4 Maßnahmen beschrieben, um sowohl das Deployment schadhafter Container zu verhindern als auch Schwachstellen und Fehlkonfigurationen zu vermeiden.

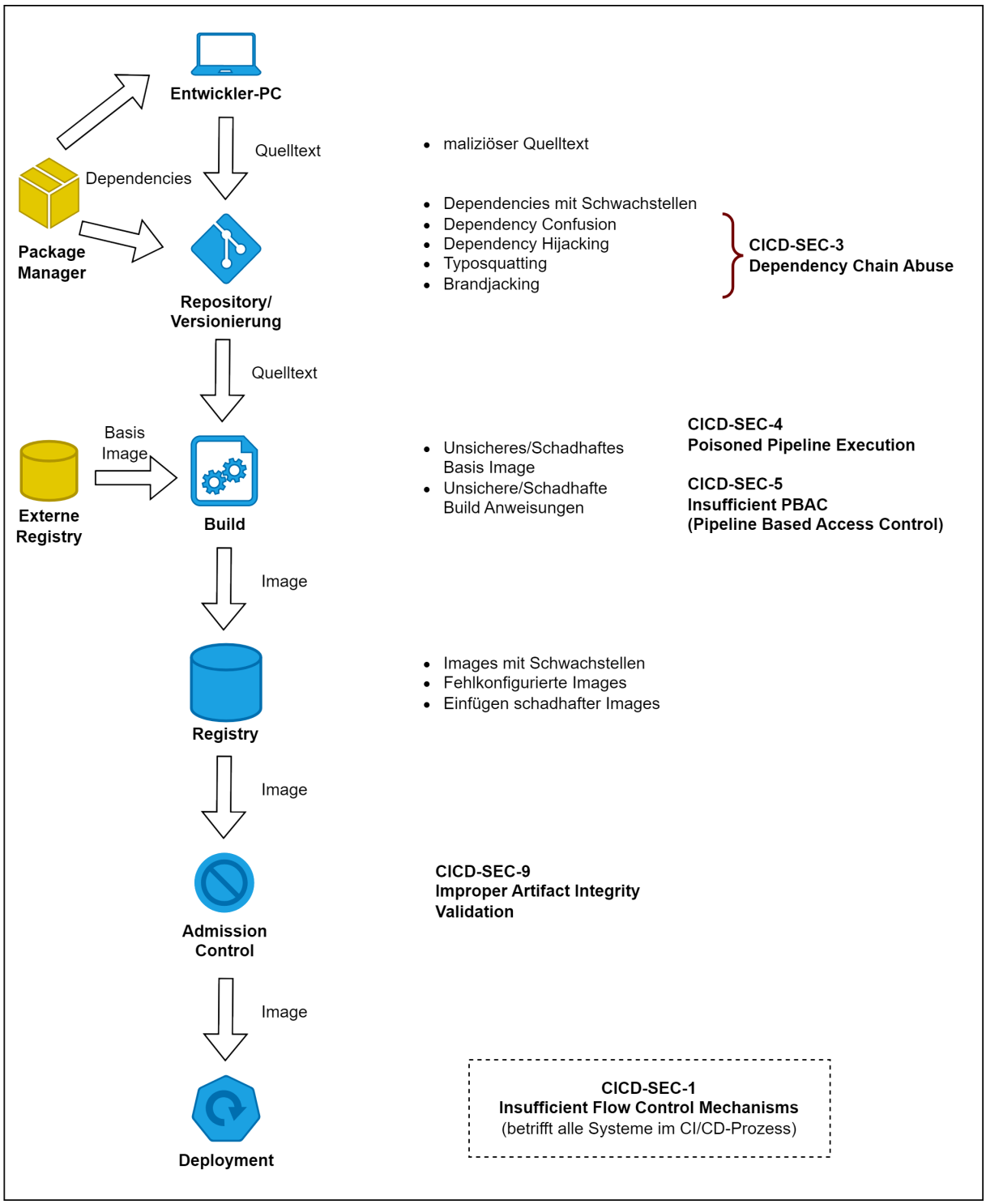

Figure 1: Abbildung: Container Images in CI/CD

4.1 Image Signatur und Verifikation

(TUF, Notary)

4.2 Container Registry

Bei der Entwicklung containerisierter Anwendungen stößt man als erstes vermutlich auf die öffentliche Container Registry Dockerhub, um von dort aus die benötigten Basis-Images zu beziehen. Dieser Ansatz ähnelt der ungeprüften Verwendung von Quelltextausschnitten, Bibliotheken oder sonstigen Abhängigkeiten aus einem offen verfügbaren Code-Repository. Ohne die Verwendung einer eigenen privaten Container Registry ist es folglich schwierig die Kontrolle darüber zu behalten welche (Basis-)Images im eigenen Cluster verwendet werden. Ferner reduziert eine private Container Registry Angriffsvektoren wie Typosquatting und DNS-Spoofing. [Rice20]

Zugleich ist die eigene Container Registry eng mit den Baustein-Anforderungen SYS.1.6.A6 Verwendung sicherer Images, SYS.1.6.A12 Verteilung sicherer Images und SYS.1.6.A14 Aktualisierung von Images verbunden. [BSI22]

Es gibt eine Vielzahl von Lösungen zur Realisierung einer privaten Container Registry. Aus diesem Grund sollten zunächst einige Qualitätsmerkmale für die Auswahl betrachtet werden:

1. Kompatibilität zu anderen Komponenten in der eingesetzten CI/CD-Pipeline (Gitlab-Runner, Jenkins, Github Actions, Admission Controler, etc.)
2. *On-Premise* Bereitstellung (insbesondere relevant für Kritische Infrastrukturen)
3. integriertes Image Scanning
4. integriertes Image Signing

Einige Container Registries, die diese Bedingungen erfüllen wären:

- Harbor Container Registry[1]
  - Opensource
  - hohe Kompatibilität zu anderen Repositories (Replication Adapters) und Image Scannern (Scanner Adapters)
- Nexus Repository[2]
  - vor allem interessant als Gesamtlösung im CI/CD-Prozess (Bereitstellung von language Packages, SBOM-Validierung, Nexus Container Security)

[1] https://goharbor.io/
[2] https://de.sonatype.com/products/container?topnav=true

- Red Hat Quay[3]
- Docker private registry server[4]
  - erfordert hohes Maß an eigenständiger Konfiguration um 3 und 4 zu erfüllen

4.3 Helm Repository

Auch das Sammelsurium an YAML-Konfigurationsdateien zur Definition von Deployments, Services, Nutzern, Volumes und weiteren Kubernetes-Komponenten sollte zentral abgespeichert werden, sodass einerseits deren Konfiguration auditiert und andererseits die Wiederverwendbarkeit von Kubernetes-Komponenten verbessert wird.

Helm Charts haben sich als Format für Kubernetes-YAML-Dateien etabliert. Genauso wie Dockerhub von der Allgemeinheit genutzt werden kann um Container Images zu teilen, erlaubt Artifacthub die Bereitstellung von Helm Charts. Somit müssen öffentliche Helm Charts gleichermaßen geprüft werden, bevor sie in ein lokales Repository gezogen werden. [Helm]

Sowohl Harbor, als auch Nexus können mit als Helm Repository verwendet werden.

4.4 Admission Control

(Connaiseur)

4.5 Image Scanning

aqua, trivy

```
1 trivy fs
2 trivy image
3 trivy repo
```

- Container Registry

[3] https://www.redhat.com/en/technologies/cloud-computing/quay
[4] https://docs.docker.com/registry/deploying/

4.6 Hinweise zum Build-Prozess und der Gestaltung von Images

In Kapitel 3 wurde bereits das Thema **Immutable Containers** und Möglichkeiten für die Durchsetzung dieses Prinzips zur Laufzeit besprochen. Die beschriebenen Maßnahmen können tiefergreifend verstärkt werden, indem ein Container Image auch nur die Laufzeitumgebung und Bibliotheken enthält, die die Anwendung benötigt. Selbst das minimalistische Basis-Image **Alpine** enthält eine große Menge von typischen Linux-Befehlen wie `ls`, `cat`, `mount` und `sh`, welche einem Angreifer genügend Möglichkeiten bieten, um sich auf dem System umzusehen und ggf. seine Privilegien zu eskalieren. **Reverse Shells** greifen üblicherweise darauf zurück einen Shell-Prozess zu starten. Das heißt, würde man die Shell-Binary garnicht erst im Container bereithalten, ist es auch wesentlich schwieriger die Anwendung als solche zu kompromittieren.

Distroless Basis-Images greifen genau diese Problematik auf und reduzieren die Angriffsfläche dadurch, dass sie nur die notwendige Laufzeitumgebung beinhalten. Der Unterschied fällt besonders stark im Vergleich des `node:18` Basis-Images auf Dockerhub mit dem `gcr.io/distroless/nodejs18-debian11` Image von Distroless auf. [Distr], [Rice20]

```
1 docker images -a
2 # REPOSITORY                                TAG       IMAGE ID
      CREATED          SIZE
3 # ...
4 # node                                      18        e390ceb99781   13
      days ago      991MB
5 # gcr.io/distroless/nodejs18-debian11   latest    34e1fabd14c3   52
      years ago     160MB
```

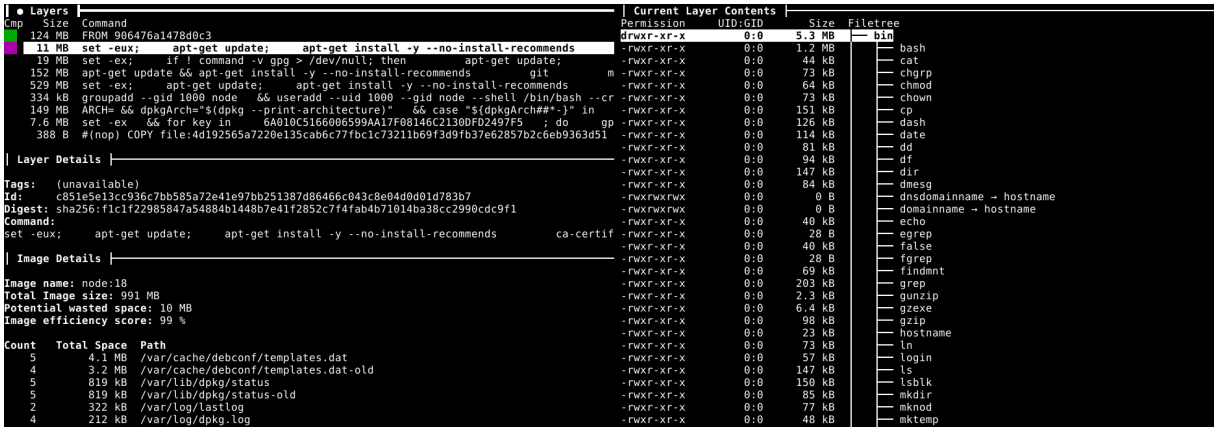

Figure 2: Abbildung 2: Layer Inhalt Node Basis-Image (`dive node:18`)

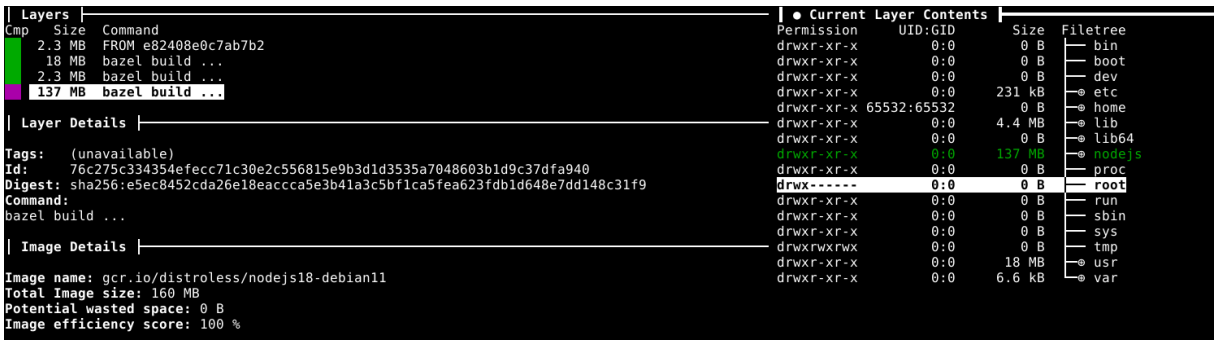

Figure 3: Abbildung 3: Layer Inhalt Distroless Node Basis-Image

Bei der Bereitstellung einer Anwendung in einem Container-Image sind in der Regel zusätzliche Build-Schritte notwendig. Um bei dem Beispiel einer Node-Anwendung zu bleiben, müssten in diesem Fall zunächst alle Abhängigkeiten über ein `npm install` installiert werden. Der Node Package Manager ist jedoch zur Laufzeit nicht mehr notwendig und sollte deswegen auch nicht auf dem finalen Image enthalten sein. Gleiches gilt für Compiler und ähnliche Werkzeuge zur Fertigung einer Binary (z.B. in Go, C, etc.). Aus diesem Grund sollte auf **Multi-Stage Builds** zurückgegriffen werden, welche ein temporäres Image für den Buildprozess (Kompilierung, Dependency-Installation) erstellen und die fertige Anwendung anschließend in ein minimalistisches Basis-Image (wie distroless) einfügen:

```
1  # Beispiel Multi-Stage Dockerfile von distroless
2
3  FROM node:18 AS build-env
4  ADD . /app
5  WORKDIR /app
6  RUN npm install --omit=dev
7
8  FROM gcr.io/distroless/nodejs18-debian11
9  COPY --from=build-env /app /app
10 WORKDIR /app
11 EXPOSE 3000
12 CMD ["hello_express.js"]
```

- kics, rootless builds, buildah

5. Angriffsszenarien

- Use Cases, mögliche Angriffspfade, Mitigation und Erkennung
- woran erkenne ich, dass ich in einem Container bin?
- bekannte Exploits
- ATT&CK-Framework

6. Der Baustein Containerisierung

- spezifisch für Betreiber Kritischer Infrastrukturen
- was gilt es noch zur Erfüllung zu beachten?
  - was noch nicht in den vorherigen Kapiteln behandelt wurde
  - organisatorische Maßnahmen

6.1 Anforderungen

SYS.1.6.A1 Planung des Container-Einsatzes

SYS.1.6.A2 Planung der Verwaltung von Containern

SYS.1.6.A3 Sicherer Einsatz containerisierter IT-Systeme

SYS.1.6.A4 Planung der Bereitstellung und Verteilung von Images

SYS.1.6.A5 Separierung der Administrations- und Zugangsnetze bei Containern

SYS.1.6.A6 Verwendung sicherer Images

Diese Anforderung ist eng gekoppelt mit SYS.1.6.A12 und SYS.1.6.A14 und wird mit den Hinweisen zu diesen zwei Anforderungen erfüllt.

SYS.1.6.A7 Persistenz von Protokollierungsdaten der Container

SYS.1.6.A8 Sichere Speicherung von Zugangsdaten bei Containern

SYS.1.6.A9 Eignung für Container-Betrieb

SYS.1.6.A10 Richtlinie für Images und Container-Betrieb

SYS.1.6.A11 Nur ein Dienst pro Container

Aufgrund des geringen Speicherbedarfs des Container-Images gibt es kaum einen Grund mehr als einen Dienst auf einem Container laufen zu lassen. Gerade deswegen bringen Container einen besonderen Anreiz für die Realisierung von Micro-Service-Architekturen mit sich. Die Auflockerung dieser Anforderung hätte zur Folge, dass der Sicherheitsgewinn durch seccomp-Profile oder Capability-Beschränkungen abnimmt.

Mittels eines Admission Controllers lässt sich überprüfen, ob ein Container-Image nur einen Dienst startet. Insbesondere bedeutet das auch, dass nur einziger Port geöffnet wird.

SYS.1.6.A12 Verteilung sicherer Images

Hier empfiehlt sich ein Auftragsprozess zur Beantragung neuer Basis-Images, die für den Betrieb benötigt werden. Somit ist der Prprozess ordentlich dokumentiert und es sind stets nur Basis-Images im Einsatz, die zuvor auch geprüft wurden. Zugleich sollte im Freigabeprozess überprüft werden, ob nicht ein bereits verifiziertes Basis-Image verwendet werden kann. An diesem Prozess sollte das Security Operations Center beteiligt werden.

Sämtliche auf diesem Weg hinzugefügte Basis-Images werden mit einer Notary-Signatur versehen (s. Kapitel 4.1)

SYS.1.6.A13 Freigabe von Images

SYS.1.6.A14 Aktualisierung von Images

Container Images werden in der privaten Container Registry auf Schwachstellen gescannt und falls erforderlich aktuallisierte Basis-Images geprüft und heruntergeladen. Unabhängig davon, ob das Basis-Image, eine Dependency oder der Quelltext selbst eine Schwachstelle enthält, muss die selbstenwickelte Container-Anwendung neu gebaut werden und deren Funktionalität erneut getestet werden. Eine vollständige Automatisierung von Updates, wie es beim Patchmanagement externer

Anwendungen, Dienste und des Betriebssystems selbst üblich ist, kann für die CI/CD-Pipeline nicht sinnvoll umgesetzt werden. Schließlich sind die Entwickler (und kein externer Hersteller) in der Pflicht zu prüfen, ob mit einem Update der Abhängigkeiten die Schnittstelle des Containers semantisch gleich bleibt.

Schwachstellen und Updates im Basis-Image lassen sich mitunter komplett vermeiden, wenn in einem Multi-Stage-Build lediglich die Binary einer Anwendung im Container vorliegt. Ansonsten können diese zumindest maßgeblich durch die Verwendung minimaler Images (wie bspw. distroles) reduziert werden.

So verlockend es auch scheint in Kubernetes die `imagePullPolicy` auf `Always` zu setzen oder innerhalb eines Deplyoments (bzw. Pods) auf ein Image mit dem `:latest`-Tag zu verweisen, ist ein solches Vorgehen nicht empfehlenswert. Dieser Herstellerhinweis ist begründet durch damit einhergehende Erschwerung des Rollback-Mechanismus und Prüfung der aktuell verwendeten Image-Version im Cluster. [K8_IMG]

Für genauere Informationen zu den beschriebenen Konzepten, kann in Kapitel 4 nachgesehe werden.

SYS.1.6.A15 Limitierung der Ressourcen pro Container

Die Container-Orchestrierung Kubernetes stellt hierfür zwei Konzepte bereit **Resource Quotas** und **Limit Ranges**. Mit **Resource Quotas** lassen sich Obergrenzen für gesamte **namespaces** definieren. Innerhalb eines **namespace** könnte somit ein einzelner Pod die gesamten Ressourcen der zugewiesenen **Resource Quota** an sich reißen. Hier erlauben **Limit Ranges** eine Begrenzung der Ressourcen auf Granularität einzelner Pods.

Ohne Container-Orchestrierungs-Werkzeug könnten Beschränkungen mit cgroups oder Slice-Files festgelegt werden. Allerdings wäre es fahrlässig in einer kritischen Infrastruktur auf die Automatisierungs- und Protokollierungsmöglichkeiten einer Container-Orchestrierung zu verzichten.

SYS.1.6.A16 Administrativer Fernzugriff auf Container

Ein administrativer Fernzugriff auf Container darf unter keinen Umständen in einer Produktivumgebung erfolgen. Hierbei würde abermals das Prinzip **Immutable Containers** verletzt werden. Außerdem müssten nur zum Zweck der Administration übliche Linux-Befehle (`sh`, `ls`, etc.) im Image vorbehalten werden.

In einer Entwicklungsumgebung könnte man zum Debugging einen Fernzugriff erlauben. Wenn damit einhergeht, dass ein anderes Basis-Image, als in der Produktivumgebung verwendet werden muss (bspw. node:18 und distroles/node) kann wiederum keine identische Semantik der Anwendungen garantiert werden, weswegen auch dieser Punkt hinfällig wird.

SYS.1.6.A17 Ausführung von Containern ohne Privilegien

SYS.1.6.A18 Accounts der Anwendungsdienste

SYS.1.6.A19 Einbinden von Datenspeichern in Container

SYS.1.6.A20 Absicherung von Konfigurationsdaten

SYS.1.6.A21 Erweiterte Sicherheitsrichtlinien

SYS.1.6.A22 Vorsorge für Untersuchungen

SYS.1.6.A23 Unveränderlichkeit der Container

Diese Anforderung wird mit der Bedingung **Immutable Containers** erfüllt.

SYS.1.6.A24 Hostbasierte Angriffserkennung

SYS.1.6.A25 Hochverfügbarkeit von containerisierten Anwendungen

SYS.1.6.A26 Weitergehende Isolation und Kapselung von Containern

7. Referenz hilfreicher Werkzeuge

7.1 Linux-Befehle

7.1.1 Capabilities

```
cat /proc/<PID>/status | grep Cap
```

```
getcaps <PID>
```

7.1.2 Syscalls

7.2 Tools

7.2.1 Docker Installation

1. Installation Container Runtime (hier Docker) [DInst]

```
sudo apt-get remove docker docker-engine docker.io containerd runc
sudo apt-get install ca-certificates curl gnupg lsb-release

sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --
    dearmor -o /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
    keyrings/docker.gpg] https://download.docker.com/linux/debian $(
    lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.
    list > /dev/null

sudo chmod a+r /etc/apt/keyrings/docker.gpg
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
    compose-plugin
```

7.2.2 Minikube

Donwload Minikube über [MK8Inst].

```
minikube start --driver=virtualbox
minikube config set driver virtualbox
```

Buildah, Connaiseur, Dive, Grype, Skopeo, eBPF

Literaturverzeichnis

- [Abbassi], Building Container Images with Podman and Buildah[1], Puja Abbassi, 12.08.2019, GiantSwarm
- [ATT&CK], Containers Matrix[2], MITRE ATT&CK, 01.04.2022
- [BSI22], IT-Grundschutzkompendium Edition 2022[3]
- [Buildah], Buildah Image Builder[4], Containers Organisation
- [Cilium], eBPF-based Networking, Observability, Security[5], Isovalent
- [Connaiseur], Connaiseur Kubernetes Admission Controller[6], SSE Secure Systems
- [DInst], Install Docker Engine on Debian[7], docs.docker.com
- [Distr], “Distroless” Container Images[8], GoogleContainerTools, Github
- [Dive], Dive Image Explorer[9], Alex Goodman, Github
- [Dono21], Die Unterschiede zwischen Docker, containerd, CRI-O und runc[10], Tom Donohue, 12.07.2021
- [eBPF], eBPF[11]
- [Helm], Helm - The package manager for Kubernetes[12]
- [HTCap], Linux Capabilities[13], Carlos Polop
- [K8S_Arc], Kubernetes Components[14], 24.10.2022
- [K8S_IMG], Images[15], 13.11.2022
- [Knecht19], Using Multi-Stage Builds to Simplify And Standardize Build Processes[16], Sven Hans

[1] https://www.giantswarm.io/blog/building-container-images-with-podman-and-buildah
[2] https://attack.mitre.org/matrices/enterprise/containers/
[3] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2022.pdf?__blob=publicationFile&v=3
[4] https://buildah.io/
[5] cilium.io
[6] https://github.com/sse-secure-systems/connaisseur
[7] https://docs.docker.com/engine/install/debian/
[8] https://github.com/GoogleContainerTools/distroless
[9] https://github.com/wagoodman/dive
[10] https://www.kreyman.de/index.php/others/linux-kubernetes/232-unterschiede-zwischen-docker-containerd-cri-o-und-runc
[11] https://ebpf.io/
[12] https://helm.sh/
[13] https://book.hacktricks.xyz/linux-hardening/privilege-escalation/linux-capabilities
[14] https://kubernetes.io/docs/concepts/overview/components/
[15] https://kubernetes.io/docs/concepts/containers/images/
[16] https://medium.com/capital-one-tech/multi-stage-builds-and-dockerfile-b5866d9e2f84

Knecht, 14.03.2019
- [Grype], Grype Image Scanner[17], Alex Goodman, Github
- [Isov21], Detecting a Container Escape with Cilium and eBPF[18], Isovalent Blog, 16.11.2021
- [KICS], Keeting Infrastructure as Code secure[19]
- [MK8Inst], minikube start[20], minikube, 15.11.2022
- [Mouat19], Linux Capabilities in Practice[21], Adrian Mouat, 25.09.2019
- [MS21], Secure containerized environments with updated threat matrix for Kubernetes[22], Yossi Weizmann, 23.03.2021
- [OWASP], Docker Security Cheat Sheet[23], OWASP Cheat Sheet Series
- [OWASPCD], OWASP Top 10 CI/CD Security Risks[24], Daniel Krivelevich, Omer Gil, OWASP
- [Rice20], Container Security, Liz Rice, O'Reilly, 2020
- [Rice22], What is eBPF - An Introduction to a New Generation of Networking, Security, and Observability Tools, Liz Rice, O'Reilly, 13.04.2022
- [Skopeo], Skopeo[25], Github Containers Open Repository for Container Tools
- [SYS1.6], SYS.1.6 Containerisierung[26], Bundesamt für Sicherheit in der Informationstechnik, Februar 2022
- [WiLK], Linux Kernel[27], Wikipedia, 23. Oktober 2023
- [Xen19], Xen 4.12 shrinks code, beefs up security, rethinks x86 support[28], Max Smolaks, The Register, 04.04.2019

[17] https://github.com/anchore/grype
[18] https://isovalent.com/blog/post/2021-11-container-escape/
[19] https://kics.io/
[20] https://minikube.sigs.k8s.io/docs/start/
[21] https://blog.container-solutions.com/linux-capabilities-in-practice
[22] https://www.microsoft.com/en-us/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/
[23] https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
[24] https://owasp.org/www-project-top-10-ci-cd-security-risks/
[25] https://github.com/containers/skopeo
[26] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/IT-GS-Kompendium_Einzel_PDFs_2022/07_SYS_IT_Systeme/SYS_1_6_Containerisierung_Edition_2022.pdf?__blob=publicationFile&v=3
[27] https://en.wikipedia.org/wiki/Linux_kernel
[28] https://www.theregister.com/2019/04/04/xen_412_release/