www.javaspektrum.de

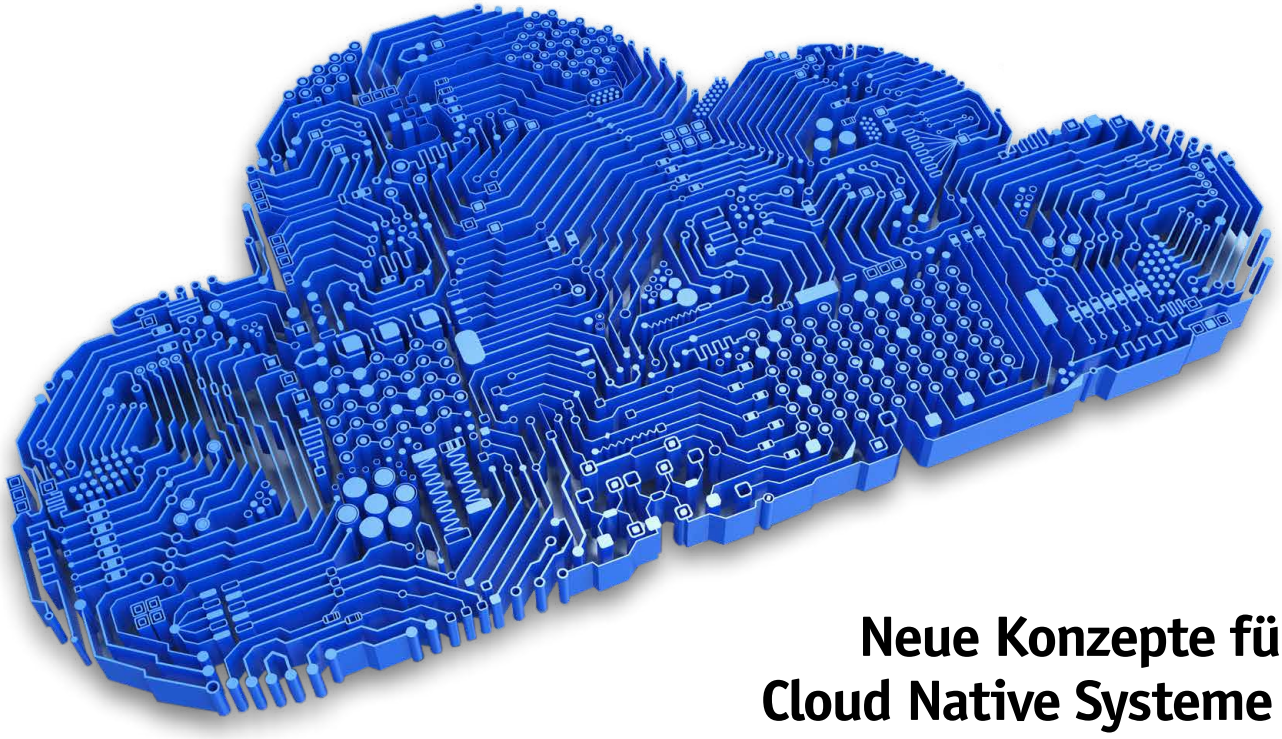
Ausgabe 1/2020
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 01

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Cloud Native – Architektur & Integration



Neue Konzepte für Cloud Native Systeme

Mit Kubernetes-Operators den Betrieb automatisieren

Interview

Maximilian Hille von Crisp Res…
über beschleunigte Softw…
und mehr Agilität fü…
durch Low Cod…

…BSTORMS®-Lernlabor –
…nd Testen lernen

Wie Stakeholder mit Analysetools besser zusammenarbeiten

Sonderdruck aus JavaSpektrum 1/2020

Cloud Native – Architektur & Integration

G30325

Kubernetes AppOps Security

Security Context – Teil 1: Good Practices

Johannes Schnatterer

Werden Anwendungen auf managed Kubernetes-Clustern betrieben, ist auch der Betreiber des Clusters (beispielsweise Amazon oder „der Betrieb") für die Sicherheit zuständig, oder? Nicht ganz! Zwar abstrahiert Kubernetes von der Hardware, sein API bietet dennoch viele Möglichkeiten, die Sicherheit der darauf betriebenen Anwendungen gegenüber der Standardeinstellung zu verbessern. Dieser Artikel zeigt, welche Einstellungen im Security Context von Pods und Containern empfehlenswert sind.

Wer schon einmal einer Container-Runtime wie Docker unter die Haube gesehen oder gar Anwendungen auf einer Container-Runtime in Produktion betrieben hat, weiß, dass das virtuelle Konstrukt „Container" im Kern ein normaler Linux-Prozess ist, der durch bestimmte Kernel-Komponenten großteils isoliert vom Rest des Systems läuft. Dies macht Container leichtgewichtiger, aber auch angreifbarer als virtuelle Maschinen (VMs).

Um diese Angriffsfläche zu reduzieren, bieten Container-Runtimes vielfältige Einstellungen, deren Standardwerte einen Kompromiss zwischen Benutzbarkeit und Sicherheit darstellen. An dieser Stelle können Entwickler durch einige good Practices die Angriffsfläche verkleinern. Eine gute Übersicht für die weit verbreitete Container-Runtime Docker bietet die Benchmark des Center for Internet Security (CIS) [cis-docker], in der neben Host- und Container-Runtime-Konfiguration auch good Practices für Image und Container in Bezug auf Sicherheit spezifiziert sind.

Für den Betrieb von Containern auf Kubernetes gilt dies in mindestens gleichem Umfang: Als Container-Orchestrator abstrahiert Kubernetes von unterliegenden Container-Runtimes, auf der die eigentlichen Container ausgeführt werden. Viele der good Practices in Bezug auf Container, die für Docker gelten, gelten damit auch für Kubernetes. Hinzu kommt, dass Kubernetes teilweise die Standardeinstellungen der Container-Runtimes verändert. Es ist allerdings möglich, auf verschiedenen Ebenen in Kubernetes Einstellungen an die Container-Runtime weiterzureichen und so die Sicherheit zu erhöhen. Kubernetes bietet darüber hinaus noch weitere empfehlenswerte Sicherheitsmechanismen, wie Network Policies, in den ersten beiden Teilen dieser Artikelserie beschrieben sind [Sch19-a, Sch19-b].

Der Artikel beschreibt im Folgenden zunächst, welche Einstellungen in Kubernetes vorhanden sind. Anschließend wird eine pragmatische good Practice vorgestellt, welche die Sicherheit erhöht, ohne den Aufwand zu sehr in die Höhe zu treiben. Wer sich mit Docker auskennt, wird hier viele Punkte aus der CIS-Benchmark für Docker wiederfinden. Die CIS-Benchmark für Kubernetes [cis-k8s] hingegen fokussiert sich auf den Betrieb des Clusters (API-Server, Kubelet usw.) und enthält leider wenig Empfehlungen für den Betrieb von Anwendungen auf dem Cluster.

Johannes Schnatterer ist Continuous-Delivery-Enthusiast, fokussiert auf Softwarequalität und hat eine Passion für Open Source. Er ist begeistert von allen Cloud-native-Themen und deren lebhafter Community. Johannes Schnatterer arbeitet für die Cloudogu GmbH als Solution Architect und Trainer. E-Mail: johannes.schnatterer@cloudogu.com

Security Context

Der direkteste Weg, sicherheitsrelevante Einstellungen in Kubernetes an die Container-Runtime weiterzureichen, ist der Security Context. Diesen gibt es auf zwei Ebenen, pro Pod und pro Container. Einige Einstellungen sind auf beiden Ebenen möglich. In diesem Fall hat die Einstellung auf Container-Ebene Vorrang. Ein Beispiel in YAML zeigt Listing 1: Hier wird ein Pod spezifiziert, dessen Container nicht mit dem User „root“ ausgeführt werden dürfen. Eine Ausnahme wird für einen speziellen Initialisierungs-Container gemacht.

```
apiVersion: v1
kind: Pod
# ...
spec:
  securityContext:
    runAsNonRoot: true
  containers:
    - name: mustNotRunAsRoot
  initContainers:
    - name: isAllowedToRunAsRoot
      securityContext:
        runAsNonRoot: false
```

Listing 1: Ein Pod mit Security Context auf Pod- und Container-Ebene

Good Practices

Unter dem jeweiligen „securityContext“ stehen viele Einstellungen für Container zur Auswahl, wobei es mehr Optionen auf Container-Ebene gibt. Welche der Einstellungen empfiehlt es sich hier zu setzen und welcher Aufwand entsteht dabei?

Listing 2 enthält eine Reihe gegenüber den Standardwerten einschränkender Einstellungen auf Container-Ebene, die erfahrungsgemäß ein guter Startpunkt sind, da sie einige Angriffsvektoren verhindern, ohne zu aufwendig zu sein. Sollte eine Anwendung mit diesen Einstellungen nicht laufen, können sie später abgeschwächt werden.

```
apiVersion: v1
kind: Pod
# ...
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
spec:
  containers:
  - name: restricted
    securityContext:
    runAsNonRoot: true
    runAsUser: 100000
    runAsGroup: 100000
    readOnlyRootFilesystem: true
    allowPrivilegeEscalation: false
    capabilities:
      drop:
        - ALL
```

Listing 2: Ein Pod mit einschränkenderen Einstellungen

Die Einstellungen und welche Auswirkungen sie auf die Anwendung im Container haben, wird im Folgenden für jeden einzelnen Punkt besprochen. Wer diese selbst in einer definierten Umgebung ausprobieren will, findet vollständige Beispiele mit Anleitung im Repository „cloudogu/k8s-security-demos“ bei GitHub [k8s-sec-demo].

Container mit unprivilegiertem User ausführen

Durch das Ausnutzen von Fehlkonfiguration oder Schwachstellen im Kernel oder Container-Runtime ist es denkbar, dass ein Angreifer aus der Isolation des Containers ausbricht („container escape“). Wenn dies gelingt, hat der Angreifer außerhalb des Containers die Rechte des Users, mit dem der Prozess auf dem Host System ausgeführt wird.

Insofern ist es unbedingt zu vermeiden, dass der Container mit dem User „root“ (User-ID 0, Group-ID 0) ausgeführt wird, da der Angreifer sonst alle Rechte auf dem Host-System erlangt. Darüber hinaus ist es empfehlenswert, Containern nicht die User-ID oder Group-ID eines bestehenden Users auf dem System zu geben, da auch diese beispielsweise Zugriffsrechte auf Dateien auf dem System haben könnten, die Angreifer ausnutzen könnten.

Dies lässt sich auf mehreren Ebenen lösen:

- Im Image (beispielsweise mit der Dockerfile-Anweisung „USER 100000“).
- Im Kubernetes Security Context mit „runAsNonRoot: true“ wird die Ausführung des Containers verhindert, wenn dieser als User-ID 0 („root“) starten will.
- Im Kubernetes Security Context mit „runAsUser“ und „runAsGroup“. Die Wahl eines Wertes größer 10 000 ist empfehlenswert, da es hier unwahrscheinlicher wird, dass dieser auf dem Host-System vergeben ist. In Dockerfiles wird häufig 1000 gesetzt, was meist dem ersten auf dem System angelegten User entspricht. Das ist bequem für die Entwicklung, aber in Produktion ein Angriffsvektor.

Von den in Listing 2 gezeigten Einstellungen zeigen diese die größte Wirkung in Bezug auf die Reduktion der Angriffsfläche. Sie verursachen allerdings auch den größten Aufwand. Denn:

- Viele offizielle Images basieren auf der Ausführung als „root“ (beispielsweise NGINX oder Postgresql). Hier gibt es meistens Alternativen von Drittanbietern, beispielsweise von bitnami [bitnami-docker]. Von NGINX gibt es sogar ein dediziertes Image, das einen unprivilegierten User zur Ausführung setzt [nginx-unpriv]. Generell ist bei der Auswahl der Images Vorsicht geboten, denn nicht alle Images im Internet sind vertrauenswürdig [kromtech-cryptojacking]. Eine weitere Alternative ist es, selbst ein Image für die gewünschte Anwendung zu bauen, das explizit einen User setzt. Der einfachste Ansatz hierzu ist, das offizielle Image als Basis zu nehmen.
- „runAsNonRoot“ funktioniert nur mit einer numerischen ID. Wenn im Dockerfile also beispielsweise „USER node“ steht, scheitert die Ausführung des Containers. Dies kann durch explizites Setzen von „runAsUser“ umgangen werden.
- Je nachdem, welchen Besitzer und Zugriffsmodus („change mode“) Dateien im Container oder in Volumes haben, kann es dazu kommen, dass der Container zur Laufzeit keinen Zugriff hat. Dies kann behoben werden, indem Besitzer, Gruppe oder Zugriffsmodus (mittels des Befehls „chmod“) angepasst werden. Je nach Anwendungsfall erfordert dies eine Anpassung beim Starten des Pods oder ein eigenes Image. Bei Volumes kann dies mit einem „initContainer“ erreicht werden, der mehr Rechte hat

als der eigentliche Anwendungs-Container. Eine Möglichkeit zeigt Listing 1.

Ein aktuelles Beispiel einer Sicherheitslücke, deren Ausnutzung sich mit diesen Einstellungen verhindern lässt, ist [CVE-2019-5736]. Hier können Angreifer durch eine Sicherheitslücke in der low level Container-Runtime runc (wird auch von Docker verwendet) aus dem Container ausbrechen, wenn der Container mit dem User „root" ausgeführt wird.

Read-only root Filesystem

Ein Angreifer kann auch ohne Ausbruch aus dem Container Schaden anrichten. Er kann beispielsweise den Code der Anwendung kompromittieren und so Daten der Benutzer abgreifen. Des Weiteren kann er Programme aus dem Internet in den Container herunterladen, um von dort seinen Angriff in nicht öffentliche, aber vom Container aus erreichbare Netzwerk auszuweiten. Insbesondere der erste Fall kann komplett verhindern werden, indem das Dateisystem des Containers als read-only eingebunden wird.

Um einen möglichst schnellen Start und effiziente Speicherplatznutzung zu ermöglichen, arbeiten Container mit einem Copy-on-Write-Algorithmus: Das Dateisystem des Image wird beim Start nicht kopiert, sondern nur überlagert: Jeder Container, der aus einem Image gestartet wird, bekommt eine eigene Schicht („Layer"), in der zur Laufzeit geschriebene Dateien abgelegt werden. Wird eine Datei geändert, die im Image vorhanden ist, wird sie zunächst in die Container-Schicht kopiert und dann verändert. Ein Lesezugriff prüft zunächst, ob die Datei in der Container-Schicht vorhanden ist. Falls nicht, wird sie aus dem Image geliefert.

Mit „readOnlyRootFilesystem: true" wird die Container-Schicht deaktiviert, was dazu führt, dass zur Laufzeit nicht mehr ins Dateisystem geschrieben werden kann. Dadurch kann der Code nicht mehr verändert werden. Auch wenn Package-Manager im Container installiert sind, funktionieren diese meist nicht mehr. Das Nachladen von Anwendungen wird dadurch erschwert. Ein netter Nebeneffekt ist eine bessere Performanz beim Lesen und Schreiben.

Die meisten Anwendungen können allerdings nicht von Haus aus mit einem read-only Dateisystem umgehen. Webserver benötigen beispielsweise temporäre Ordner fürs Caching (häufig „/tmp"). Diese benötigten Ordner können im Kubernetes-Pod als eigenes Volume, beispielsweise als „emptyDir", zur Verfügung gestellt werden.

Um diese Ordner zu identifizieren, bietet sich das Docker-Kommando „diff" an. Es zeigt, was in der Container-Schicht steht. Ein mögliches Vorgehen ist wie folgt:

- Anwendung lokal als Docker-Container starten (nicht mit read-only Dateisystem),
- Tests auf dem System ausführen, möglichst alle technischen Anwendungsfälle durchspielen (sonst könnte das Ergebnis unvollständig sein),
- „docker diff <containerId>" ausführen.

Das Ergebnis ist eine Liste von Dateien, die während der Laufzeit des Containers geschrieben wurden. Wenn alle betroffenen Ordner in Kubernetes als Volume gemountet werden, kann der Container dort ohne Probleme mit „readOnlyRootFilesystem" ausgeführt werden.

Bei den meisten Anwendungen ist dies einfach umzusetzen und bietet mehr Sicherheit. Manche Fälle, beispielsweise wenn die Anwendung so entworfen ist, dass sie beim Starten oder zur Laufzeit Dateien aus dem Image verändert, ist dies umständlicher. Hier gibt es mehrerer Möglichkeiten, Abhilfe zu schaffen: Im Dockerfile kann das betroffene Verzeichnis als „VOLUME" deklariert werden. Die Dateien werden dann von der Container-Engine zur Laufzeit dorthin kopiert. Alternativ lässt sich dies auch in Kubernetes lösen: Im Pod teilen sich „initContainer" und die anderen Container die Volumes. So kann ein „initContainer" Dateien aus dem Image in das Volume kopieren. In beiden Fällen ist es wichtig, Besitzer, Gruppe und Zugriffsmodus so zu setzen, dass der Anwendungs-Container Zugriff hat.

Privilege Escalation verhindern

Wenn der Container nicht mit dem User „root" ausgeführt wird, ist es immer noch denkbar, dass ein Angreifer seine Rechte erweitert. Der Mechanismus dürfte den meisten Linux-Anwendern bekannt sein, denn „sudo" macht (ohne weitere Parameter) nichts anderes als einen Befehl mit den Rechten des Users „root" auszuführen. Der „sudo"-Befehl sollte also auf keinen Fall im Container installiert und konfiguriert sein. Dies trifft auf die meisten Container zu.

Eine andere Möglichkeit, Rechte zu erweitern, sind Schwachstellen, beispielsweise im Kernel [CVE-2015-3339]. Mit der Einstellung „allowPrivilegeEscalation: false" wird dies einfach verhindert. Dabei sind keine ungewollten Nebenwirkungen für die Anwendung im Container zu erwarten. Wenn ein Image dafür entworfen wurde, ohne „root"-Rechte ausgeführt zu werden, braucht es zur Laufzeit keine erweiterten Rechte.

Capabilities einschränken

In Linux ist es möglich, die Rechte von Prozessen feingranular zu erweitern, ohne sie gleich als User „root" (der alles darf) auszuführen. Dazu gibt es definierte Capabilities, die Prozessen gewährt werden können. Bekannte Beispiele sind der Zugriff auf Sockets (beispielsweise für die Anwendung „ping", Capability „NET_RAW") oder das Binden an Ports < 1024 (beispielsweise für Webserver, Capability „NET_BIND_SERVICE").

Container-Runtimes starten die Container-Prozesse mit ausgewählten Capabilities, die auch hier einen Kompromiss zwischen Benutzbarkeit und Sicherheit darstellen (Beispiel: Docker [docker-capabilities]). Häufig werden diese Capabilities von Anwendungen nicht benötigt, gewähren aber einem Angreifer mehr Rechte und erhöhen damit die Angriffsfläche. Beispielsweise kann mit der Capability „NET_RAW" ein Man-In-The-Middle-Angriff auf die Kommunikation aller Container auf einem Host mittels DNS-Spoofing ausgeführt werden [aqua-dns-spoofing].

Um dies sicherer zu gestalten, bietet sich ein Whitelisting-Ansatz an: Starten ohne Capabilities und nur bei Bedarf ausgewählte erlauben.

Wie beim read-only Dateisystem kann auch empirisch auf dem lokalen Rechner mit Docker herausgefunden werden, welche Capabilities benötigt werden. Dazu wird das gewünschte Image mit „--cap-drop ALL" gestartet und führt Anwendungsfälle mit dem System aus. Aus den Fehlermeldungen ist meist schnell ersichtlich, welche Capabilities fehlen, die dann jeweils mit dem Parameter „--cap-add" hinzugefügt werden. Ein gutes Beispiel ist erneut NGINX, der ohne bestimmte Capabilities nicht startet. Dazu gehört die Capability „NET_BIND_SERVICE" fürs Binden an Port 80. Eine Alternative zum Hinzufügen der Capability ist in dem speziellen Fall ein eigenes Image, das NGINX so konfiguriert, dass dieser nicht an Port 80, sondern an einen Port größer 1024 bindet, siehe unter anderem [nginx-unpriv].

Seccomp default Profil aktivieren

Die Isolation von Containern ist im Linux Kernel durch Sicherheitsmechanismen wie Seccomp realisiert. Seccomp erlaubt das Einschränken von Syscalls, also Aufrufen von Funktionalitäten, die im Kernel realisiert sind. Docker hat hier nach ausgiebigem Test aller Dockerfiles auf GitHub [frazelle-container-security] im Jahr 2016 ein „default Profile" eingeführt, das 44 der über 300 Syscalls verhindert. Damit ist es „moderat schützend" und trotzdem mit den meisten Anwendungen kompatibel [docker-seccomp].

Bei Docker wird dies standardmäßig auf Container-Prozesse angewendet. Bei Kubernetes wurde es aus Sorge um Kompatibilität allerdings explizit deaktiviert [k8s-20870]. Dies zu ändern, ist zwar in Arbeit, aber Stand Kubernetes 1.16 noch nicht umgesetzt [k8s-enhancement-135]. Seccomp ist ein zentrales Sicherheits-Feature von Docker und dort seit Jahren im Einsatz. Das Fehlen von Seccomp bei Kubernetes wurde auch schon in einem Security Audit angemahnt [k8s-sec-audit].

Es drängt sich also auf, zu Beginn explizit ein Profil zu setzen und es nur im Bedarfsfall zu deaktivieren. Dies sollte nur in Ausnahmefällen zu Problemen führen. Beispielsweise wird „official images" im DockerHub, der Standard-Registry von Docker, nur für begründete Ausnahmen die Abweichung von den Standardeinstellungen gestattet [docker-lib-official].

In Kubernetes hat Seccomp es noch nicht in das offizielle API geschafft. Es kann daher noch nicht über den Security Context festgelegt werden, sondern über eine Annotation. Dies ist das übliche Vorgehen für „Alpha"-Features, bevor sie einem bestimmten API-Objekt zugewiesen werden. Die Annotation wird auf Pod-Ebene spezifiziert und gilt entweder für den ganzen Pod (s. Listing 2), kann aber auch pro Container spezifiziert werden.

Wer prüfen möchte, ob in einem Container ein Seccomp aktiv ist, kann das:

- Innerhalb des Containers mit „grep Seccomp /proc/1/status" erfragen. „Seccomp: 0" bedeutet, dass kein Profil aktiv ist. Im „proc"-Dateisystem werden Informationen zu allen Prozessen angezeigt und im Container sollte der Hauptprozess die Prozess-ID 1 haben.
- Außerhalb des Containers kann dies bei Docker mit „docker inspect" abgefragt werden, wobei dort nur bei Abweichung vom Standard etwas zu finden ist. Ein explizit deaktiviertes Seccomp (wie bei Kubernetes), wird als „seccomp:unconfined" angezeigt.

Fazit und Empfehlung

Dieser Artikel empfiehlt, die folgenden Einstellungen pro Container in Kubernetes vorzunehmen:

- Container mit unprivilegiertem User ausführen,
- ein read-only root Filesystem verwenden,
- Privilege Escalation verhindern,
- Capabilities einschränken und
- das Seccomp default Profil aktivieren.

Dies stellt einen pragmatischen Ansatz dar, der mit überschaubarem Aufwand die Rechte, mit denen ein Container ausgeführt wird, verringert. Aus Erfahrung reicht es, eine Volume für den Ordner „/tmp" zu erstellen, wodurch viele Webanwendungen (beispielsweise in Java mit Spring Boot) dann ohne weitere Probleme mit diesen Einstellungen laufen. Falls sich für eine spezielle Anwendung kein anderer Weg findet, ist es immer noch sicherer, einzelne Sicherheitseinstellungen aufzuweichen, als gleich von Beginn an mehr Rechte zu vergeben als benötigt werden. Dieser „least Privilege"-Ansatz verbessert die Sicherheit des ganzen Clusters, indem viele Angriffsvektoren auf Container unterbunden werden. Wie diese Angriffe im Detail aussehen, welche weiteren Sicherheitsoptionen es gibt und weitere fortgeschrittene Themen zum Security Context wird Teil des nächsten Artikels in dieser Serie sein.

Literatur und Links

[aqua-dns-spoofing] DNS Spoofing on Kubernetes Clusters | Aqua Blog, https://blog.aquasec.com/dns-spoofing-kubernetes-clusters

[bitnami-docker] bitnami's Profile - Docker Hub, https://hub.docker.com/u/bitnami

[cis-docker] CIS Docker Benchmarks, https://www.cisecurity.org/benchmark/docker/

[cis-k8s] CIS Kubernetes Benchmark, https://www.cisecurity.org/benchmark/kubernetes/

[CVE-2019-5736] Runc and CVE-2019-5736 - Kubernetes Blog, https://kubernetes.io/blog/2019/02/11/runc-and-cve-2019-5736/

[CVE-2015-3339] CVE-2015-3339 - privilege escalation via setuid binary, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3339

[docker-capabilities] Docker security | Docker Documentation, https://docs.docker.com/engine/security/security/#linux-kernel-capabilities

[docker-lib-official] docker-library/official-images: Source of truth for the Docker „Official Images", https://github.com/docker-library/official-images#runtime-configuration

[docker-seccomp] Seccomp security profiles for Docker | Docker Documentation, https://docs.docker.com/engine/security/seccomp/

[frazelle-container-security] Ramblings from Jessie: Containers, Security, and Echo Chambers, https://blog.jessfraz.com/post/containers-security-and-echo-chambers/

[k8s-enhancement-135] kubernetes/enhancements. Issue #135. Seccomp, https://github.com/kubernetes/enhancements/issues/135

[k8s-sec-audit] Open Sourcing the Kubernetes Security Audit - Cloud Native Computing Foundation, https://www.cncf.io/blog/2019/08/06/open-sourcing-the-kubernetes-security-audit/

[k8s-sec-demo] GitHub: cloudogu/k8s-security-demos, https://github.com/cloudogu/k8s-security-demos/tree/master/3-security-context

[kromtech-cryptojacking] Cryptojacking invades cloud. How modern containerization trend is exploited by attackers, https://kromtech.com/blog/security-center/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers

[kubernetes-20870] kubernetes Issue #20870. Add support for seccomp, https://github.com/kubernetes/kubernetes/issues/20870

[nginx-unpriv] nginxinc/nginx-unprivileged – Docker Hub, https://hub.docker.com/r/nginxinc/nginx-unprivileged

[Sch19-a] J. Schnatterer, Network Policies – Teil 1: Good Practices, in: JavaSPEKTRUM, 05/2019, siehe: https://cloudogu.com/de/blog/k8s-app-ops-teil-1

[Sch19-b] J. Schnatterer, Network Policies – Teil 2: Fortgeschrittene Themen und Tipps, in: JavaSPEKTRUM, 06/2019, siehe: https://cloudogu.com/de/blog/k8s-app-ops-teil-2