Mechanismen für Containersicherheit unter Linux

Lorenz Kofler

BACHELORARBEIT

Nr. S1810239018

eingereicht am
Fachhochschul-Bachelorstudiengang

Sichere Informationssysteme

in Hagenberg

im Mai 2021

Betreuung:

Dieter Vymazal, BSc MSc

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 17. Mai 2021

Lorenz Kofler

Inhaltsverzeichnis

Kurzfassung

Container sind eine leichtgewichtige Alternative zu virtuellen Maschinen und werden dadurch immer beliebter. Um Container zu ermöglichen, sind unter Linux verschiedenste Mechanismen verfügbar. Einerseits sind Mechanismen vorhanden, welche die vom Container benötigten Systemressourcen isolieren, und andererseits Mechanismen, welche den Host vor unerlaubten Zugriffen aus dem Container beschützen. Das Ziel dieser Bachelorarbeit ist es, den Stand der Technik der vorhandenen Mechanismen zur Isolation von Containern aufzuzeigen. Ebenso werden die eventuell vorhandenen Probleme geschildert.

Um die Forschungsfragen zu beantworten, werden zuerst die zur Isolation von Containern verwendeten Linux Kernel Features und anschließend die dazu verwendeten Linux Security Modules behandelt. Die Funktionalität der Mechanismen wird mithilfe von Beispielen demonstriert.

Es zeigt sich, dass alle nötigen Mechanismen für eine erfolgreiche Isolation von Containern vorhanden sind. Die Verwendung von Linux Security Modules ist ein besonders wichtiger Mechanismus, um Hosts vor einem Container zu beschützen. Jedoch weisen vorhanden Linux Security Modules Mängel bei Verwendung mit Containern auf. Aktuelle Forschungen fokussieren sich auf die vorhanden Mängel und versuchen diese zu beheben.

Abstract

Containers are more lightweight than virtual machines and therefore gaining popularity. Under Linux there are several mechanisms available to make containers possible. On one hand there are mechanisms needed to isolate the necessary system resources and on the other hand there are mechanisms needed to protect the host from attacks out of the container.

The goal of this thesis is to show the current state-of-the-art of mechanisms to successfully isolate containers. To answer the research questions the necessary Linux Kernel Features are explained first. The second part is the explanation of the necessary Linux Security Modules. Additionally, the functionality of the mechanism is demonstrated with examples.

The result of this thesis shows that all necessary mechanisms, to successfully isolate containers, are available. The use of Linux Security Modules is a very important mechanism to protect the host from containers. However, existing Linux Security Modules are not best suited for this use case. Current development and research is focused on improving usability of Linux Security Modules with containers.

Kapitel 1

Einleitung

1.1 Ausgangssituation

Container sind eine leichtgewichtige Alternative zu hypervisorbasierter Virtualisierung und bieten in beinahe allen Anwendungsfällen gleiche oder bessere Performance [4]. Dadurch wird die Verwendung von Containern in den verschiedensten Einsatzgebieten immer beliebter [3].

Der Unterschied zu hypervisorbasierter Virtualisierung besteht darin, dass diese die nötige Hardware emulieren. Das Gast-, sowie das Host-Betriebssystem verwenden jeweils einen eigenen Kernel. Container im Gegensatz bieten eine isolierte Umgebung an, welche den Anschein einer Virtualisierung erweckt. Es werden bestimmte Prozesse und Ressourcen wie RAM[1], CPU[2] et cetera vom Host und anderen Containern isoliert. Dabei teilen Container und Host sich den gleichen Kernel.

1.2 Problemstellung

Bei der Verwendung von Containern entsteht aufgrund des geteilten Kernels ein Problem. Damit Container sicher genützt werden können, müssen diese Mechanismen verwenden, um die Ressourcen des Containers vollständig vom Hostsystem zu isolieren. Ebenso müssen jegliche unerlaubte Zugriffe auf das Hostsystem blockiert werden.

Somit werden unterschiedliche Linux Kernel Features und Linux Security Modules verwendet, um den Container zu isolieren und das Hostsystem zu schützen. Genau genommen sind Mechanismen vorhanden, welche die Isolation von allen nötigen Ressourcen des Containers ermöglichen und dadurch den Anschein einer Virtualisierung innerhalb des Containers erwecken. Des Weiteren sind Mechanismen vorhanden, welche das Host-Betriebssystem vor Angriffen aus dem Container schützen, indem die Angriffsoberfläche auf den Kernel limitiert und der unberechtigte Zugriff auf Dateien außerhalb des Containers blockiert wird.

[1] Random-Access Memory
[2] Central Processing Unit

1.3 Zielsetzung

Es wird der Stand der Technik der unter Linux vorhandenen Mechanismen zur Isolation von Containern aufgezeigt. Ebenso werden die eventuell vorhandenen Probleme dieser Mechanismen geschildert. Des Weiteren wird ein Überblick über den Stand der Forschung gegeben und wie dieser die Probleme des Stands der Technik lösen kann.

1.4 Forschungsfragen

Was sind der Stand der Technik und der Stand der Forschung von Mechanismen, die unter Linux zur Verfügung stehen, um einen Container zu isolieren und das Hostsystem vor diesem zu schützen?

- Welche Features bietet der Linux-Kernel zum Isolieren von Containern und was sind deren Limits und Probleme?
- Welche Linux Security Modules bietet der Linux-Kernel zum Isolieren von Containern und was sind deren Limits und Probleme?
- Was ist der Stand der Forschung von Mechanismen unter Linux zum Isolieren von Containern?

1.5 Methodik

Bei dieser Arbeit handelt es sich um eine Literaturstudie. Hierbei wird der Stand der Technik und der Stand der Forschung von Mechanismen, welche unter Linux verfügbar sind, aufgezeigt. Hauptsächlich werden dabei Linux Kernel Features und Linux Security Modules betrachtet. Im Laufe der Arbeit werden die Probleme dieser Mechanismen genannt. Es wird ein Überblick über den Stand der Forschung geschaffen und wie dieser die Probleme der Stand der Technik löst. Schlussendlich wird der Stand der Technik der Mechanismen und deren Probleme zusammengefasst.

1.6 Erwartete Ergebnisse

Das erwartete Ergebnis ist eine vollständige Sammlung von Mechanismen, welche dem Stand der Technik entsprechen, um Container unter Linux zu isolieren und deren Probleme aufzuzeigen.

Kapitel 2

Linux Kernel Features

Das folgende Kapitel beschäftigt sich mit den Features des Linux-Kernels[1], welche zur Isolation von Containern verwendet werden können.

2.1 Namespaces

2.1.1 Erklärung

Die Manpage[2] von Namespaces besagt Folgendes:

> *A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. [42]*

Unter Linux sind acht verschiedene Namespace-Typen verfügbar, wovon jeder Typ unterschiedliche Systemressourcen abstrahiert. Jeder Prozess ist jeweils einem Namespace von jedem Typen zugeordnet. Prozessen ist es nur möglich, Ressourcen zu sehen bzw. zu verwenden, welche sich im gleichen Namespace befinden. Mithilfe des proc-Dateisystems[3] werden Informationen über vorhandene Namespaces mit dem Kernel ausgetauscht. Jeder Prozess besitzt unter dem Pfad */proc/PID/ns/*[4] ein Verzeichnis. Dieses Verzeichnis enthält einen Eintrag für jeden Namespace-Typen, dem der Prozess zugehörig ist [42]. In folgendem Beispiel (siehe Programm 2.1) werden die Namespaces, welchen der Prozess mit der PID 1 (systemd[5]) zugehörig ist, aufgelistet.

[1] https://www.kernel.org/linux.html (letzter Zugriff 28.02.2021)
[2] https://man7.org/linux/man-pages/man1/man.1.html (letzter Zugriff 11.05.2021)
[4] PID muss jene des Prozesses sein
[5] https://man7.org/linux/man-pages/man1/systemd.1.html (letzter Zugriff 05.02.2021)

Programm 2.1: Namespaces denen der Prozess mit der PID 1 zugehörig ist

```
1 # ls -l /proc/1/ns/
2 total 0
3 lrwxrwxrwx 1 root root 0 Jan 24 18:59 cgroup -> 'cgroup:[4026531835]'
4 lrwxrwxrwx 1 root root 0 Jan 24 18:59 ipc -> 'ipc:[4026531839]'
5 lrwxrwxrwx 1 root root 0 Jan 24 18:59 mnt -> 'mnt:[4026531840]'
6 lrwxrwxrwx 1 root root 0 Jan 24 18:59 net -> 'net:[4026532008]'
7 lrwxrwxrwx 1 root root 0 Jan 24 18:59 pid -> 'pid:[4026531836]'
8 lrwxrwxrwx 1 root root 0 Jan 24 18:59 pid_for_children -> 'pid:[4026531836]'
9 lrwxrwxrwx 1 root root 0 Jan 24 18:59 time -> 'time:[4026531834]'
10 lrwxrwxrwx 1 root root 0 Jan 24 18:59 time_for_children -> 'time:[4026531834]'
11 lrwxrwxrwx 1 root root 0 Jan 24 18:59 user -> 'user:[4026531837]'
12 lrwxrwxrwx 1 root root 0 Jan 24 18:59 uts -> 'uts:[4026531838]'
```

In Programm 2.1 sind die acht verschiedenen Namespace-Typen zu erkennen. Um Namespaces von einem Prozess aufzulisten, kann auch das Programm *lsns*[6] verwendet werden, welches die Informationen aus dem proc-Dateisystem[7] liest [59].

2.1.2 Typen von Namespaces

Um die Funktionalität der acht Namespace-Typen zu demonstrieren, wird der Befehl *unshare*[8] verwendet. Dieser erstellt die per Argument angeführten Namespaces und führt das angegebene Programm innerhalb des neuen Namespaces aus [38]. Das Programm *unshare* ist Teil des Pakets *util-linux*[9].

Network Namespace

Network Namespaces (seit Kernel 2.6.24[10]) ermöglichen die Isolation von Netzwerkressourcen [43]. Jeder Network Namespace ist eine Kopie des Netzwerk-Stacks mit eigenen Routen, Firewallregeln und Netzwerkschnittstellen [48].

Programm 2.2: Beispiel für Network Namespace

```
1 # ip route
2 default via 192.168.0.1 dev wlp0s20f3 proto dhcp metric 600
3 192.168.0.0/24 dev wlp0s20f3 proto kernel scope link src 192.168.0.131 metric 600
4 # ip netns add newns
5 # PS1="newns# " ip netns exec newns /bin/bash --norc
6 newns# ip route
7 Error: ipv4: FIB table does not exist.
8 Dump terminated
```

Programm 2.2 zeigt, dass die Routingtabelle innerhalb des neuen Network Namespaces (Zeile 6-8) nicht der Routingtabelle außerhalb des neuen Network Namespaces (Zeile 1-3) entspricht. Erstellt wird der Network Namespace mit dem *ip-netns*[11] (Zeile 4) Befehl.

[6] https://man7.org/linux/man-pages/man8/lsns.8.html (letzter Zugriff 05.02.2021)
[7] https://man7.org/linux/man-pages/man5/procfs.5.html (letzter Zugriff 25.03.2021)
[8] https://www.man7.org/linux/man-pages/man1/unshare.1.html (letzter Zugriff 05.02.2021)
[9] https://www.kernel.org/pub/linux/utils/util-linux/ (letzter Zugriff 03.02.2021)
[10] https://kernelnewbies.org/Linux_2_6_24 (letzter Zugriff 05.02.2021)
[11] https://man7.org/linux/man-pages/man8/ip-netns.8.html (letzter Zugriff 05.02.2021)

Network Namespaces ermöglichen Containern, ihre eigenen Netzwerkressourcen zu verändern, ohne dabei die Netzwerkressourcen des restlichen Systems zu beeinflussen. Jeder Container besitzt eigene Netzwerkschnittstellen und einen eigenen Portnummernraum. Beispielsweise ist es somit möglich, mehrere Webapplikation in Containern zu betreiben, welche alle innerhalb ihres eigenen Namespaces auf Port 80 gebunden sind.

Mount Namespace

Mount Namespaces (seit Kernel 2.4.19) ermöglichen die Sicht auf die Liste der Mountpunkte, welche eine Gruppe an Prozessen sieht, zu isolieren [41]. Es ist möglich, dass Prozesse in verschiedenen Mount Namespaces eine unterschiedliche Sicht auf das Dateisystem haben. In folgendem Beispiel wird dies demonstriert.

Programm 2.3: Beispiel für Mount Namespace

```
1 # which cp
2 /usr/bin/cp
3 # PS1="newns# " unshare --mount /bin/bash --norc
4 newns# mount --bind /usr/bin/ /mnt/
5 newns# ls /mnt/cp
6 /mnt/cp
7 newns# exit
8 # ls /mnt/cp
9 "/mnt/cp": No such file or directory (os error 2)
```

In Programm 2.3 wird eine neue Shell in einem neuen Mount Namespace gestartet (Zeile 3). Innerhalb des neuen Mount Namespaces wird das Verzeichnis */usr/bin* nach */mnt* eingehängt (Zeile 4). Es ist zu erkennen, dass innerhalb des Namespaces das eingehängte Verzeichnis in der Dateisystem-Hierarchie sichtbar ist (Zeile 5-6). Außerhalb des Namespaces sind keine Auswirkungen auf die Dateisystem-Hierarchie (Zeile 8-9) zu erkennen. Dieser Mechanismus ermöglicht, dass Container eine andere Sicht auf das Dateisystem haben.

PID Namespace

PID Namespaces (seit Kernel 2.6.24[12]) ermöglichen die Isolation des PID-Nummernraums, wobei neue Namespaces bei PID 1 zu zählen beginnen [44]. Dieser Mechanismus wird von Containern zur Prozessisolation verwendet. Ein Prozess kann nur jene Prozesse sehen, welche sich im selben PID Namespace oder in einem Kind-Namespace davon befinden. So kann zum Beispiel verhindert werden, dass ein Prozess aus einem Container *kill*[13] Signale zu Prozessen außerhalb des Containers sendet. Ebenso können PIDs innerhalb des Containers immer dieselben bleiben, egal welche PIDs auf dem jeweiligen Hostsystem in Verwendung sind. Ein Beispiel dafür wird in Abschnitt 2.1.3 gegeben.

[12] https://kernelnewbies.org/Linux_2_6_24 (letzter Zugriff 05.02.2021)
[13] https://man7.org/linux/man-pages/man2/kill.2.html (letzter Zugriff 05.02.2021)

Time Namespaces

Time Namespaces (seit Kernel 5.6[14]) ermöglichen je Namespace unterschiedliche Zeit Offsets für die Systemuhren CLOCK_MONOTONIC und CLOCK_BOOTTIME[15] zu bestimmen [45]. Für den jeweiligen Namespace wird die Uhrzeit des initialen Namespaces mit dem Offset addiert. Das folgende Beispiel zeigt, wie durch einen Offset die Systemuhr CLOCK_BOOTIME innerhalb eines Namespaces verändert werden kann.

Programm 2.4: Beispiel für Time Namespace

```
 1 # uptime --pretty
 2 up 2 days, 18 hours, 5 minutes
 3 # cat /proc/self/timens_offsets
 4 monotonic           0         0
 5 boottime            0         0
 6 # PS1="newns# " sudo unshare --time /bin/bash --norc
 7 newns# echo "boottime  $((7*24*60*60)) 0" > /proc/self/timens_offsets
 8 newns# cat /proc/self/timens_offsets
 9 monotonic           0         0
10 boottime       604800         0
11 newns# uptime --pretty
12 up 1 week, 2 days, 18 hours, 5 minutes
13 newns# exit
14 # uptime --pretty
15 up 2 days, 18 hours, 5 minutes
```

In Programm 2.4 wird ein neuer Time Namespace erstellt (Zeile 6). Innerhalb des Time Namespaces wird ein Offset von sieben Tagen zur Systemuhr CLOCK_BOOTIME hinzugefügt (Zeile 7). Es zeigt, dass die Betriebszeit innerhalb des Namespaces (Zeile 11-12) verändert werden kann, ohne dabei die Zeit im initialen Namespace zu beeinflussen (Zeile 14-15). Mit diesem Mechanismus ist es möglich, die Zeit innerhalb eines Containers unabhängig vom Hostsystem zu ändern.

User Namespaces

Mithilfe User Namespaces (seit Kernel 3.8[16]) ist es möglich, sicherheitsrelevante Identifizierungsmerkmale und Attribute zu isolieren [46]. Unter anderem UID[17] und GID[18], was ermöglicht, dass UID und GID je nach User Namespaces unterschiedlich sein können. Ein Prozess kann eine unprivilegierte UID außerhalb eines User Namespaces besitzen, während der Prozess innerhalb des User Namespaces die UID 0 besitzt und somit volle Privilegien innerhalb des Namespaces hat.

[14] https://kernelnewbies.org/Linux_5.6#Time_Namespaces (letzter Zugriff 05.02.2021)
[15] https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html (letzter Zugriff 05.02.2021)
[16] https://kernelnewbies.org/Linux_3.8 (letzter Zugriff 05.02.2021)
[17] User Identifier
[18] Group Identifier

Programm 2.5: Beispiel für User Namespace

```
1 # cat /proc/$$/status | egrep '^[UG]id'
2 Uid:    1000    1000    1000    1000
3 Gid:    985     985     985     985
4
5 # PS1="newns# " unshare --user --map-root-user /bin/bash --norc
6 newns#  cat /proc/$$/status | egrep '^[UG]id'
7 Uid:    0       0       0       0
8 Gid:    0       0       0       0
```

In Programm 2.5 ist zu erkennen, dass die UID und die GID außerhalb des neuen Namespaces (Zeile 1-3) unterschiedlich zu der UID und der GID innerhalb des Namespaces (Zeile 6-8) sind. Erstellt wird der neue User Namespace in Zeile 5. Mit dem Argument *--map-root-user* wird angegeben, dass die aufrufende UID (1000, Zeile 2) zu der UID des *root* Users (0, Zeile 7) umgesetzt wird [38].

Container verwenden diesen Mechanismus, wenn ein Prozess mit privilegierten Rechten ausgeführt werden muss, jedoch der Container selbst nur mit unprivilegierten Rechten [53]. Somit kann ein Prozess innerhalb es Containers Superuser-Rechte besitzen, wobei diese Rechte auf den jeweiligen Namespace limitiert sind. Hierdurch ermöglichen User Namespaces das Konzept von unprivilegierten Containern.

Ebenso ermöglichen User Namespaces Rootless Docker[56]. Damit ein Namespace erstellt werden kann, sind privilegierte Capabilities (siehe Abschnitt 3.2) nötig. Dies bedeutet beispielsweise der Docker-Dämon[19] muss mit Superuser-Rechte ausgeführt werden, um Namespaces erstellen zu können. Damit das umgangen werden kann, wird zuerst ein User Namespace erstellt und der Docker-Dämon in diesem ausgeführt. Dadurch besitzt der Dämon privilegierte Rechte innerhalb seines User Namespaces und kann somit Namespaces für neue Container erstellen, ohne Superuser-Rechte zu benötigen. Hierdurch wird möglicher Schaden von kompromittierten Docker-Containern weiter verringert.

UTS Namespaces

UTS[20] Namespaces ermöglichen die Isolation des Hostnamens und des NIS[21] Namens.

Programm 2.6: Beispiel für UTS Namespace

```
1 # PS1="newns# " unshare --uts /bin/bash --norc
2 newns# hostname linux2
3 newns# hostname
4  linux2
5 newns# exit
6 # hostname
7 linux
```

In Programm 2.6 ist zu erkennen, dass der Hostname außerhalb des neuen UTS Namespaces (Zeile 6-7) nicht durch die Veränderung des Hostnamens innerhalb des neuen Namespaces (Zeile 2-4) beeinflusst wird. Container verwenden den Mechanismus, um ihren Hostnamen unabhängig vom Hostsystem zu verändern [52].

[19] https://docs.docker.com/get-started/overview/#the-docker-daemon (letzter Zugriff 28.02.2021)
[20] UNIX Time Sharing
[21] Network Information System

IPC Namespace

IPC[22] ist die Kommunikation von Prozessen untereinander. Mittels des IPC Namespaces (seit Kernel 2.6.19[23]) werden System-V-IPC-Mechanismen[24] und POSIX-Nachrichtenwarteschlangen[25] isoliert [40].

Programm 2.7: Beispiel für IPC Namespace

```
1 # ipcs --shmems
2 ------ Shared Memory Segments --------
3 key        shmid      owner      perms      bytes      nattch     status
4 0x00000000 589838     lorenz     600        67108864   2          dest
5 0x00000000 196654     lorenz     600        524288     2          dest
6 0x00000000 196657     lorenz     600        524288     2          dest
7 0x00000000 196666     lorenz     600        1048576    2          dest
8 # PS1="newns# " unshare --ipc /bin/bash --norc
9 newns# ipcs -m
10 ------ Shared Memory Segments --------
11 key        shmid      owner      perms      bytes      nattch     status
```

Mithilfe des Befehls *ipcs*[26] ist es möglich, Informationen über System-V-IPC anzuzeigen [31]. In Programm 2.7 werden mittels des *--shmems* Arguments alle aktiven Shared Memory Segmente angezeigt (Zeile 1-7). Zu erkennen ist, dass diese innerhalb eines neuen IPC Namespaces nicht mehr vorhanden sind (Zeile 9-11). Der IPC Namespace stellt sicher, dass ein Container weder auf die IPC Ressourcen von anderen Containern, noch auf die des Hosts zugreifen kann.

Cgroup Namespace

Eine Erklärung zu Cgroups kann in Abschnitt 2.2 gefunden werden. Mittels des Cgroup Namespaces wird verhindert, dass Informationen über die am Host vorhanden Cgroups an den Container gelangen [26].

Programm 2.8: Beispiel für Cgroup Namespace

```
1 # cgcreate -g freezer:foo
2 # echo $$ > /sys/fs/cgroup/freezer/foo/cgroup.procs
3 # grep freezer /proc/self/cgroup
4 7:freezer:/foo
5 # PS1="newns# " unshare --cgroup /bin/bash --norc
6 newns# grep freezer /proc/self/cgroup
7 7:freezer:/
8 newns# echo $$
9 1093614
10 # exit
11 # grep freezer /proc/1093614/cgroup
12 7:freezer:/foo
```

[22] Inter-Process Communciation
[23] https://kernelnewbies.org/Linux_2_6_19 (letzter Zugriff 20.02.2021)
[24] https://man7.org/linux/man-pages/man7/sysvipc.7.html (letzter Zugriff 21.02.2021)
[25] https://man7.org/linux/man-pages/man7/mq_overview.7.html (letzter Zugriff 21.02.2021)
[26] https://man7.org/linux/man-pages/man1/ipcs.1.html (letzter Zugriff 18.04.2021)

In Programm 2.8 ist zu erkennen, dass die Sicht auf die Cgroups-Hierarchie innerhalb des neuen Namespaces (Zeile 6-7) nicht der Sicht außerhalb des Namespaces entspricht (Zeile 11-12).

2.1.3 Implementierung

Um mit Namespaces zu arbeiten, werden folgende Systemcalls benötigt [42]:

- *clone()*[27]: Erstellt einen neuen Prozess und die angegeben Namespaces.
- *setns()*[28]: Der aufrufende Prozess wird zu einem angegeben, bereits bestehenden, Namespace hinzugefügt.
- *unshare()*[29]: Der aufrufende Prozess wird einem Namespace hinzugefügt. Wenn dieser Namespace nicht vorhanden ist, wird ein neuer erstellt.

Um welchen Namespace-Typen es sich handelt, wird mit einer eindeutigen Flag für jeden Namespace-Typen angegeben [42]. Folgendes C-Programm[30] demonstriert die Erstellung eines neuen PID Namespaces mittels des *clone()* Systemcalls.

Programm 2.9: Implementierung eines PID Namespaces in C

```
1 #define _GNU_SOURCE
2 #include <sched.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7
8 #define CHILD_STACK (0x100000)
9
10 int isolated(){
11   printf("Child pid: %d\n", getpid());
12   return 0;
13 }
14
15 int main(){
16   void *stack = malloc(CHILD_STACK);
17   pid_t child_pid = clone( &isolated, stack + CHILD_STACK, CLONE_NEWPID, NULL);
18   if (child_pid == -1) {
19     fprintf(stderr, "clone() failed: %m\n");
20     return 0;
21   }
22   printf("clone() = %ld\n", (long)child_pid);
23   waitpid(child_pid, NULL, 0);
24   free(stack);
25   return 0;
26 }
```

In Programm 2.9 wird mittels des Systemcalls *clone()* ein Kindprozess erstellt (Zeile 18). Übergeben wird die Flag *CLONE_NEWPID* zum Erstellen eines PID Namespaces. Der Kindprozess führt die Funktion *isolated()* in Zeile 10 aus. In Zeile 11 wird die PID

[27] https://man7.org/linux/man-pages/man2/clone.2.html (letzter Zugriff 05.02.2021)
[28] https://man7.org/linux/man-pages/man2/setns.2.html (letzter Zugriff 05.02.2021)
[29] https://man7.org/linux/man-pages/man2/unshare.2.html (letzter Zugriff 05.02.2021)
[30] https://en.cppreference.com/w/ (letzter Zugriff 20.02.2021)

aus der Sicht des Kindprozesses auf *stdout*[31] geschrieben. In Zeile 22 wird die PID des Kindprozesses aus der Sicht des Elternprozesses auf *stdout* geschrieben.

Programm 2.10: Kompilieren und Ausführen des Programms 2.9

```
1 # id
2 uid=0(root) gid=0(root) groups=0(root)
3 # gcc pid_namespace.c -o pid_namespace
4 # ./pid_namespace
5 clone() = 1233830
6 Child pid: 1
```

In der Konsolenausgabe (Programm 2.10) ist zu erkennen, dass der Kindprozess aus der Sicht des Elternprozesses die PID *1233830* besitzt. Hingegen besitzt der Kindprozess aus dessen eigener Sicht die PID *1*, was der Funktionalität des PID Namespaces entspricht.

2.1.4 Probleme

Die Entwicklung von Namespaces findet seit Linux 2.4.19 (2002) statt und begann mit der Einführung des Mount Namespaces. Ab 2006 begann die Implementierung der restlichen Namespaces [2]. Namespaces wurden nicht von Grunde auf geplant, sondern im Laufe der Zeit zum Kernel hinzugefügt. Eric Biederman schrieb 2006 [2], dass zirka 6% bis 15% des Kernel-Sourcecodes verändert werden müssen, was zirka 130.000 Zeilen Code entspricht. Dadurch wurden die Änderungen aufgeteilt und jeder Namespace einzeln nacheinander implementiert. Diese Änderungen bewirkten, dass der Namespaces-Code für verschiedenste Sicherheitsschwachstellen verantwortlich war. Besonders nach der Einführung von User Namespaces (siehe Abschnitt 2.1.2) in Kernel 3.8[32] wurden mehrere Schwachstellen gefunden [6, S. 26]. Dies hatte schlussendlich zur Folge, dass eine Funktion eingebaut wurde, welche ermöglicht, User Namespaces zu deaktivieren [22]. Mittlerweile wird der Namespaces-Code jedoch als ausgereift betrachtet [25]. Auch User Namespaces sind seit 2019 unter Docker verwendbar [56].

2.1.5 Verwendung

Namespaces sind ein essenzieller Mechanismus zur Isolation von Containern. Durch die Abstrahierung von Ressourcen in unterschiedliche Namespaces wird in Containern der Anschein der Virtualisierung erreicht. Mithilfe des User Namespaces werden unprivilegierte Container möglich gemacht. Somit wird das Risiko, dass ein Container Schaden am Hostsystem anrichten kann, zusätzlich verringert.

Namespaces werden unter anderem von den Container-Management-Programmen LXC[33] und Docker[34] verwendet, um Systemressourcen zu isolieren. Wie aus der Docker Dokumentation [25] zu entnehmen ist, sind Namespaces die erstrangige und einfachste Form der Isolation. Auch der Browser Chromium[35] verwendet Namespaces, um seine Prozesse vor Angriffen aus dem Internet besser zu schützen [21].

[31] https://man7.org/linux/man-pages/man3/stdout.3.html (letzter Zugriff 06.02.2021)
[32] https://kernelnewbies.org/Linux_3.8 (letzter Zugriff 05.02.2021)
[33] https://linuxcontainers.org/ (letzter Zugriff 05.02.2021)
[34] https://www.docker.com/ (letzter Zugriff 05.02.2021)
[35] https://www.chromium.org/ (letzter Zugriff 05.02.2021)

2.2 Control Groups

2.2.1 Erklärung

Control Groups – kurz Cgroups – erlauben es, den Ressourcenverbrauch für eine Gruppe von Prozessen zu überwachen und zu limitieren [39]. Ressourcen, für welche dies möglich ist, sind unter anderem Prozessorzeit, Speicherverbrauch und Netzwerkbandbreite. Cgroups sind verfügbar seit Linux-Kernel 2.4.26[36].

2.2.2 Implementierung

Eine Cgroup ist eine Gruppe von Prozessen, welche an bestimmte Limits und Parameter gebunden ist [39]. Ein Subsystem ist eine Kernel-Komponente, welche das Verhalten der Prozesse einer Cgroup verändert. Verschiedenste Subsysteme machen es möglich, unterschiedlichste Ressourcen zu überwachen und zu limitieren. Neu erstellte Prozesse übernehmen automatisch die Cgroups der Elternprozesse. Die Kommunikation mit dem Kernel findet über das Pseudo-Dateisystem cgroupfs statt: */sys/fs/cgroup/*. Der Linux-Kernel 5.11 unterstützt 13 verschiedene Subsysteme [39]. Folgende Subsysteme sind beispielsweise vorhanden.

- *cpu*: Limitieren von CPU Ressourcen
- *memory*: Limitieren des Speicherverbrauchs
- *pids*: Limitieren der Anzahl an Prozessen
- *freezer*: Aussetzen und Fortsetzen von Prozessen

Programm 2.11 demonstriert die Anwendung von Cgroups. Verwendet werden dafür die Tools *cgcreate*[37] und *cgexec*[38] aus dem Paket *libcgroup*[39].

Programm 2.11: Beispiel für Control Groups

```
1 # cgcreate -g memory:foo
2 # cgexec -g memory:foo ping 8.8.8.8 > /dev/null &
3 [1] 1008678
4 # cat /sys/fs/cgroup/memory/foo/cgroup.procs
5 1008678
6 # cat /sys/fs/cgroup/memory/foo/memory.usage_in_bytes
7 438272
8 # kill 1008678
9 # echo 50000 > /sys/fs/cgroup/memory/foo/memory.limit_in_bytes
10 # cgexec -g memory:foo ping 8.8.8.8
11 Killed
```

In Programm 2.11 wird mittels des Befehls *cgcreate* eine Cgroup mit dem Namen *foo* im *memory* Subsystem erstellt (Zeile 1). Mithilfe *cgexec* wird der Befehl *ping 8.8.8.8*[40] ausgeführt und der Cgroup hinzugefügt (Zeile 2-3). Die Datei *cgroup.procs* enthält die PIDs, welche der Cgroup zugehörig sind (Zeile 4-5). Wenn das Speicherlimit der Cgroup niedriger gesetzt wird (Zeile 9) als der benötigte Speicher (Zeile 6-7), ist es nicht mehr

[36] https://kernelnewbies.org/Linux_2_6_24#Task_Control_Groups (letzter Zugriff 22.02.2021)
[37] https://linux.die.net/man/1/cgcreate (letzter Zugriff 26.02.2021)
[38] https://linux.die.net/man/1/cgexec (letzter Zugriff 26.02.2021)
[39] https://github.com/matsumotory/libcgroup (letzter Zugriff 26.02.2021)
[40] https://man7.org/linux/man-pages/man8/ping.8.html (letzter Zugriff 26.02.2021)

möglich, *ping 8.8.8.8* auszuführen (Zeile 10-11). Dies ist ein Beispiel dafür, wie Cgroups verwendet werden können, um den Speicherverbrauch von Prozessen zu überwachen (Zeile 6-7) und zu limitieren (Zeile 9-11).

2.2.3 Probleme

Xing Gao et al. [5] haben Möglichkeiten gefunden, trotz Ressourcenlimitierung mit Cgroups, Out-of-Band Arbeitslast zu erstellen. Hierbei werden mithilfe unterschiedlicher Strategien Prozesse gestartet, welche nicht der entsprechenden Cgroup zugehörig sind und somit auch nicht dessen Limitierungen besitzen. In einem Experiment gelang es ihnen, dass ein Container 200 Mal mehr als sein Limit verbrauchte.

2.2.4 Verwendung

Cgroups sind, neben Namespaces, ein weiterer essenzieller Mechanismus zur Isolation der Ressourcen eines Containers. Mit Cgroups wird der Ressourcenverbrauch von Containern kontrolliert und somit verhindert, dass diese mehr Ressourcen verwenden als erlaubt. Durch die richtige Verwendung von Cgroups können mehrere bekannte Denial-of-Service-Angriffe abgewehrt werden [11]. Cgroups werden von Docker verwendet, um verschiedenste Laufzeitmetriken zu überwachen [24]. Ebenso kann Docker damit Ressourcen, wie Speicher- und CPU-Verbrauch limitieren [54].

2.3 Seccomp-BPF

2.3.1 Erklärung

Der Linux-Kernel 5.11 bietet zirka 450 Systemcalls an [49]. Jeder dieser kann als Angriffsvektor genutzt werden. Die Mehrheit der Programme benötigt jedoch nur einen kleinen Teil der verfügbaren Systemcalls. Durch ein Limitieren der verwendbaren Systemcalls werden mögliche Auswirkungen von kompromittierten Prozessen reduziert. Ebenso wird die Angriffsoberfläche auf den Linux-Kernel verringert [51].

Der Seccomp[41] Modus ist seit Kernel 2.6.12[42] verfügbar. Dieser verbietet einem Prozess die Verwendung jeglicher Systemcalls außer *read()*[43], *write()*[44], *_exit()*[45] und *sigreturn()*[46] [34]. Der Secure Computing-Modus wurde ursprünglich dafür entwickelt, CPU-Zyklen über ein Linux basiertes, weltweites Netzwerk zu vermieten [23]. Um zu garantieren, dass der ausgeführte Code keinen Schaden am System anrichten kann, wurde der Seccomp Modus verwendet.

Seccomp bietet jedoch nicht die Möglichkeit, beliebige Systemcalls zu erlauben bzw. zu verbieten. Aus diesem Grund wurde mit Kernel 3.5[47] diese Funktionalität mit Seccomp-BPF[48] (Secure Computing with filters) erweitert. Der Systemcall-Filter wird

[41] Secure Computing
[42] https://kernelnewbies.org/Linux_2_6_12 (letzter Zugriff 22.02.2021)
[43] https://man7.org/linux/man-pages/man2/read.2.html (letzter Zugriff 27.02.2021)
[44] https://man7.org/linux/man-pages/man2/write.2.html (letzter Zugriff 27.02.2021)
[45] https://man7.org/linux/man-pages/man2/exit.2.html (letzter Zugriff 27.02.2021)
[46] https://man7.org/linux/man-pages/man2/sigreturn.2.html (letzter Zugriff 27.02.2021)
[47] https://kernelnewbies.org/Linux_3.5 (letzter Zugriff 22.02.2021)

als Berkley Paket Filter Programm[49] angegeben [51]. Wenn beispielsweise eine Anwendung den *open()* Systemcall nicht benötigt, kann dieser blockiert werden. Dadurch wird der Schaden, welche eine kompromittierte Anwendung anrichten kann, und die Angriffsfläche auf den Kernel reduziert.

2.3.2 Implementierung

Folgendes ist eine Implementierung zur Demonstration des Secure Computing-Moduses in C.

Programm 2.12: Implementierung des Secure Computing-Moduses in C

```
1 #include <stdio.h>
2 #include <sys/prctl.h>
3 #include <linux/seccomp.h>
4 #include <unistd.h>
5
6 int main() {
7   printf("1. Uneingeschränkt\n");
8   prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
9   printf("2. SECCOMP_MODE_STRICT aktiv\n");
10  fork();
11  printf("3. Programm ist schon beendet\n");
12  return 0;
13 }
```

In Programm 2.12 wird der Prozess mit dem Systemcalls *prctl*[50] und der Flag *PR_SET_SECCOMP* in den Secure Computing-Modus gebracht (Zeile 8). Durch die Flag *SECCOMP_MODE_STRICT* entsteht das unter Abschnitt 2.3.1 erklärte Verhalten. Die Mehrheit der Systemcalls wird somit deaktiviert. Wird nun ein verbotener Systemcall verwendet, hier *fork()*[51] (Zeile 10), wird das Programm beendet.

Programm 2.13: Kompilieren und Ausführen des Programms 2.12

```
1 # gcc seccomp_example.c -o seccomp_example
2 ./seccomp_example
3 1. Uneingeschränkt
4 2. SECCOMP_MODE_STRICT aktiv
5 [1]    1797387 killed     ./seccomp_example
```

Wie in Programm 2.13 zu erkennen, spiegelt dies das gewünschte Verhalten wider. Es wird ein verbotener Systemcall verwendet, somit wird das Programm beendet (Zeile 5).

Folgendes Programm ist eine Implementierung in C zur Demonstration von Seccomp-BPF. Damit das Verwenden von BPF vereinfacht wird, werden von Chromium OS[52] Autoren entwickelte BPF Makros verwendet. Die C-Header-Datei ist im Github-Repo[53] des Benutzers *ahupowerdn* zu finden.

[49] https://www.freebsd.org/cgi/man.cgi?query=bpf (letzter Zugriff 27.02.2021)
[50] https://man7.org/linux/man-pages/man2/prctl.2.html (letzter Zugriff 27.02.2021)
[51] https://man7.org/linux/man-pages/man2/fork.2.html (letzter Zugriff 27.02.2021)
[52] https://www.chromium.org/chromium-os (letzter Zugriff 16.05.2021)
[53] https://github.com/ahupowerdns/secfilter/blob/master/seccomp-bpf.h (letzter Zugriff 28.02.2021)

Programm 2.14: Implementierung von Seccomp-BPF in C

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include "seccomp-bpf.h"
4
5 static int install_filter(void) {
6   struct sock_filter filter[] = {
7       VALIDATE_ARCHITECTURE,
8       EXAMINE_SYSCALL,
9       ALLOW_SYSCALL(write),
10      ALLOW_SYSCALL(exit_group),
11      KILL_PROCESS,
12  };
13  struct sock_fprog prog = {
14      .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
15      .filter = filter,
16  };
17  if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
18    perror("prctl(NO_NEW_PRIVS)");
19    return 1;
20  }
21  if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
22    perror("prctl(PR_SET_SECCOMP)");
23    return 1;
24  }
25  return 0;
26 }
27
28 int main() {
29  printf("1. Uneingeschraenkt\n");
30  install_filter();
31  printf("2. write() Systecall ist erlaubt\n");
32  FILE *fp;
33  fp = fopen("/etc/passwd", "r");
34  fclose(fp);
35  printf("3. openat() Systecall ist nicht erlaubt\n");
36  return 0;
37 }
```

Zu Beginn ist das Programm 2.14 uneingeschränkt (Zeile 29). Die nötigen Instruktionen für den Berkley Packet Filter führt die Funktion *install_filter()* (Zeile 5) aus. Mit den Makros von *seccomp-bpf.h* werden nur die Systemcalls *exit_group()*[54] und *write()* erlaubt (Zeile 9-10). Der Systemcall *exit_group* wird für die *return*-Anweisung benötigt [10, S.532]. Der Systemcall *write()* wird für die Funktion *printf*[55] benötigt [10, S.30]. Jedoch wird der für die Funktion *fopen()*[56] (Zeile 33) benötigte Systemcall *openat()*[57] nicht erlaubt. Durch *prctl()* und der Flag *SECCOMP_MODE_FILTER* wird der angegebene Berkley Packet Filter *prog* aktiv (Zeile 21).

[54] https://man7.org/linux/man-pages/man2/exit_group.2.html (letzter Zugriff 28.02.2021)
[55] https://man7.org/linux/man-pages/man3/printf.3.html (letzter Zugriff 28.02.2021)
[56] https://man7.org/linux/man-pages/man3/fopen.3.html (letzter Zugriff 27.02.2021)
[57] https://man7.org/linux/man-pages/man2/openat.2.html (letzter Zugriff 27.02.2021)

Programm 2.15: Kompilieren und Ausführen des Programms 2.14

```
1 # gcc seccomp_bpf_example.c -o seccomp_bpf_example
2 ./seccomp_bpf_example
3 1. Uneingeschraenkt
4 2. write() Systecall ist erlaubt
5 [1]    394602 invalid system call (core dumped)  ./seccomp_bpf_example
```

Programm 2.15 zeigt, dass sobald ein nicht erlaubter Systemcall ausgeführt wird, das Programms beendet wird.

2.3.3 Verwendung

Durch das Blockieren von Systemcalls in einem Container mit Seccomp-BPF werden mögliche Auswirkungen von kompromittierten Prozessen verringert und die Angriffsoberfläche auf den Kernel reduziert. Standardmäßig werden bei Docker zirka 44 Systemcalls deaktiviert [55]. Ebenso der Browser Chromium verwendet Seccomp-BPF, um zu verhindern, dass bösartiger Code ausgeführt werden kann [21].

Kapitel 3

Linux Security Modules

Das folgende Kapitel beschäftigt sich mit den Linux Security Modules (LSM), welche zur Isolation von Containern verwendet werden können.

3.1 Einleitung

3.1.1 DAC versus MAC

Die standardmäßige Zugriffskontrolle unter Linux ist DAC[1], auch bekannt als Dateiberechtigungen [15, S. 269]. Hierbei entscheidet die*der Besitzer*in einer Ressource, welche Zugriffsrechte andere Benutzer*innen auf dieselbe Ressource haben.

Programm 3.1: Dateiberechtigungen von */etc/shadow*

```
1 # ls -l /etc/shadow
2 -rw------- 1 root root 962 Dec  2 21:47 /etc/shadow
```

In Programm 3.1 ist zu erkennen, dass der Besitzer der Datei */etc/shadow*[2] *root* ist und dieser Lese- und Schreibzugriff auf die Datei besitzt. Dadurch ist es jedem privilegierten Prozess[3] möglich, auf sensitive Dateien wie zum Beispiel *shadow* zuzugreifen.

Bei MAC[4] im Gegensatz, passiert die Zugriffskontrolle mithilfe von Sicherheitsrichtlinien, welche ausschließlich von dem*der Administrator*in definiert werden und vom Betriebssystem erzwungen werden [16, S. 9]. User*innen und Prozesse haben keine Möglichkeit diese Sicherheitsrichtlinien zu umgehen. Um unter Linux unterschiedliche Sicherheitsmodule zu nutzen, wird das Linux Security Module Framework verwendet [17].

Wird MAC zusätzlich zu DAC eingesetzt, ist es gegebenenfalls möglich, den Schaden, welche kompromittierte Prozesse auf dem System anrichten können, zu minimieren. Somit ist es zum Beispiel mit MAC möglich, Prozessen mit Superuser-Rechte den Zugriff auf */etc/shadow* zu verbieten.

[1] Discretionary Access Control

[2] https://man7.org/linux/man-pages/man5/shadow.5.html (letzter Zugriff 28.02.2021)

[3] Prozesse welche von *root* ausgeführt werden

[4] Mandatory Access Control

3.1.2 LSM Framework

Das LSM Framework ist seit dem Linux-Kernel 2.6 verfügbar. Es bietet die nötigen Mechanismen im Linux-Kernel an, um Sicherheitschecks durchzuführen. Der Kernel verfügt über LSM Hooks, welche an unterschiedlichsten Stellen vorhanden sind. Bei Zugriffe auf Ressourcen des Kernels werden LSM Hooks aktiv. Diese befragen das Linux Security Module, ob die jeweilige Aktion erlaubt ist, welches mit *Ja* oder *Nein* antwortet [17]. Hauptsächlich wird das LSM Framework von MAC Erweiterungen, wie SELinux[5], AppArmor[6], Tomoyo[7] oder Smack[8], verwendet [33]. Das standardmäßig aktive LSM ist das Linux Capabilities[9] System. Welche Linux Security Modules ein Linux-System verwendet, wird mit Kernel Boot-Parameter angegeben. Zur Isolation von Containern werden Capabilities, AppArmor und SELinux verwendet, welche folgend genauer betrachtet werden.

3.2 Capabilities

3.2.1 Erklärung

Klassisch unterscheidet Linux zwischen unprivilegierten[10] und privilegierten[11] Usern. Die Rechte eines unprivilegierten Users sind sehr limitiert. Dies hat zur Folge, dass Programme, welche mehr Rechte benötigen als ein unprivilegierter User (zum Beispiel *passwd*[12], NMAP), im Superuser-Kontext ausgeführt werden müssen. Das Problem dabei ist, dass Programme, welche nur einen Teil der Superuser-Rechte benötigen, trotzdem mit vollen Superuser-Rechten ausgeführt werden müssen [7]. Somit haben diese mehr Rechte als sie benötigen würden und können dadurch, wenn diese kompromittiert werden, mehr Schaden am System anrichten.

Capabilities ermöglichen, die Rechte des Superusers in kleinere Einheiten zu unterteilen [7]. Privilegierte Kernel Calls sind in Gruppen, welche ähnliche Funktionalitäten besitzen, aufgeteilt [35]. Somit kann ein Prozess, wenn dieser die nötigen Capabilities besitzt, bestimmte privilegierte Operationen ausführen, ohne dabei alle Superuser-Rechte zu benötigen.

Capabilities können auf ausführbare Dateien sowie auf Prozesse gesetzt werden. Folgend in Auszug 3.2 befinden sich drei Beispiele für Capabilities – inklusive Beschreibung entnommen aus der Manpage.

[5] https://selinuxproject.org/page/Main_Page (letzter Zugriff 20.04.2021)
[6] https://apparmor.net/ (letzter Zugriff 20.04.2021)
[7] https://tomoyo.osdn.jp/ (Letzter Zugriff 20.04.2021)
[8] http://schaufler-ca.com/ (letzter Zugriff 20.04.2021)
[9] https://man7.org/linux/man-pages/man7/capabilities.7.html (letzter Zugriff 20.04.2021)
[10] regulärer User
[11] Superuser
[12] https://man7.org/linux/man-pages/man1/passwd.1.html (letzter Zugriff 20.04.2021)

Auszug 3.2: Capabilities Manpage[35]

```
CAP_CHOWN
       Make arbitrary changes to file UIDs and GIDs (see chown(2)).

CAP_KILL
       Bypass permission checks for sending signals (see kill(2)).  This includes
    use  of  the ioctl(2) KDSIGACCEPT operation.

CAP_NET_RAW
       * Use RAW and PACKET sockets;
       * bind to any address for transparent proxying.
```

3.2.2 Implementierung

Folgendes Beispiel (siehe Programm 3.3) demonstriert, wie Capabilities ermöglichen, dass das Tool NMAP[13] nicht mehr die Rechte eines Superusers benötigt. NMAP ist ein Netzwerk-Analysewerkzeug und Sicherheits-/Portscanner. Wenn dieses mit dem Argument *--privileged* (Zeile 3) ausgeführt wird, nimmt das Programm an, dass der*die ausführende User*in volle Privilegien besitzt. Dies hat zur Folge, dass NMAP Raw Sockets[14] verwendet, welche wiederum die Capability *CAP_NET_RAW* benötigen. Damit NMAP so ausgeführt werden kann, muss es entweder mit Superuser-Rechte gestartet werden oder von einem*einer unprivilegiertem*n User*in, wobei dann die Capability *CAP_NET_RAW* (siehe Auszug 3.2) benötigt wird.

Programm 3.3: Beispiel für Capabilities

```
 1 # id -u
 2 1000
 3 # nmap --privileged -sn 8.8.8.8
 4 Starting Nmap 7.91 ( https://nmap.org ) at 2021-02-04 13:23 CET
 5 dnet: Failed to open device wlp0s20f3
 6 QUITTING!
 7 # strace -e socket nmap --privileged -sn 8.8.8.8
 8 ...
 9 socket(AF_INET, SOCK_RAW, IPPROTO_RAW)  = -1 EPERM (Operation not permitted)
10 Couldn't open a raw socket. Error: Operation not permitted (1)
11 # getcap /usr/bin/nmap
12 <no output>
13 # sudo setcap cap_net_raw+ep /usr/bin/nmap
14 # getcap /usr/bin/namp
15 /usr/bin/nmap cap_net_raw=ep
16 # nmap --privileged -sn 8.8.8.8
17 Starting Nmap 7.91 ( https://nmap.org ) at 2021-02-04 13:27 CET
18 Nmap scan report for google.com (216.58.207.174)
19 Host is up (0.020s latency).
20 Other addresses for google.com (not scanned): 2a00:1450:4016:800::200e
21 rDNS record for 216.58.207.174: muc11s04-in-f14.1e100.net
22 Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
```

In Programm 3.3 wird gezeigt, dass es nicht möglich ist, NMAP mit Argument *--privileged* und unprivilegierten Rechten (Zeile 1-2) auszuführen (Zeile 3-6). Der Grund

[13] https://nmap.org/ (letzter Zugriff 05.02.2021)
[14] https://man7.org/linux/man-pages/man7/raw.7.html (letzter Zugriff 05.02.2021)

ist, dass NMAP Raw Sockets verwendet (Zeile 7-10). Der Befehl *getcap*[15] zeigt die Datei-Capabilities an (Zeile 11). Es sind keine Capabilities vorhanden (Zeile 12). Mittels des Befehls *setcap*[16] werden die Datei-Capabilities gesetzt (Zeile 13). Der Ausdruck *+ep* (Zeile 13) bewirkt, dass die angegebene Capability als Effective[17] und Permitted[18] hinzugefügt wird. Das Setzen der Capability war erfolgreich (Zeile 14-15). Durch die neu erlangte Capability ist es möglich NMAP, ohne volle Superuser-Rechte auszuführen (Zeile 16-22).

3.2.3 Verwendung

Docker führt standardmäßig alle Container mit reduzierten Capabilities aus [57]. Auch wenn ein Angreifer *root* Rechte innerhalb eines Containers besitzt, ist es aufgrund der fehlenden Capabilities schwerer, wirklichen Schaden am System anzurichten. Welche Capabilities innerhalb von Docker-Containern erlaubt sind, wird mit einer Zulassungsliste festgelegt.

Obwohl Capabilities heutzutage noch eine wichtige Rolle in Containern spielen, wird dies durch das Verwenden von unprivilegierten Containern mit User Namespaces weniger relevant. Der Grund dafür ist, dass ein User Namespace mit der UID 0, alle Capabilities eines echten *root* Users besitzt. Welche sich jedoch nur auf den jeweiligen User Namespace beziehen und nicht auf den initialen User Namespace (siehe Abschnitt 2.1.2).

Des Weiteren können Capabilities zur Systemhärtung eingesetzt werden. Anwendungen, welche das SUID[19] Bit gesetzt haben, sind gegebenenfalls ein Sicherheitsrisiko. Das SUID Bit hat zur Folge, dass ausführbare Dateien mit den Rechten des Dateibesitzers und nicht mit den Rechten des Anwenders ausgeführt werden. Beispielsweise anstatt das SUID Bit von NMAP (wo der Dateibesitzer *root* ist) zu setzen, sollte nur die jeweilige nötige Capability hinzugefügt werden (siehe Programm 3.3).

[15] https://man7.org/linux/man-pages/man8/getcap.8.html (letzter Zugriff 20.04.2021)
[16] https://man7.org/linux/man-pages/man8/setcap.8.html (letzter Zugriff 20.04.2021)
[17] aktive Capabilities
[18] erlaubte Capabilities
[19] Set User ID

3.3 AppArmor

3.3.1 Erklärung

AppArmor[20], ein MAC-Modell (siehe Abschnitt 3.1.1) basiertes LSM, ist seit Linux-Kernel 2.6.36[21] verfügbar. Das LSM macht es dem Systemadministrator möglich, Profile zu definieren, um die Aktionen von Programmen einzuschränken. AppArmor soll hierdurch das Betriebssystem und Anwendungen von internen und externen Gefahren, als auch Zero-Day-Angriffe schützen [19].

3.3.2 Implementierung

Folgendes Beispiel demonstriert die Anwendung und Funktionalität von AppArmor.

Programm 3.4: *readShadow.c*: Programm liest */etc/shadow*

```
1 #include <stdio.h>
2
3 int main() {
4   FILE *fp;
5   char ch;
6   fp = fopen("/etc/shadow", "r");
7   while((ch = fgetc(fp)) != EOF)
8     printf("%c", ch);
9   fclose(fp);
10 }
```

Das Programm *readShadow.c* (siehe Programm 3.4) liest die Datei */etc/shadow* ein (Zeile 6) und gibt den Inhalt der Datei auf der Konsole aus (Zeile 7-8).

Programm 3.5: Kompilieren und Ausführen von *readShadow*

```
1 # gcc readShadow.c -o /usr/bin/readShadow
2 # id
3 uid=0(root) gid=0(root) groups=0(root)
4 # readShadow | head -2
5 root:$6$8QfqhzpvtlQuNh2u$kDSD7uBDaba5tPpvuKiDvGM0LdYGa3sSX33r2jFOdeeRRAfm1C8y.aS8eo.
      ogN.NeH8kOGxbM5vBfCQi0myr8.:18683:0:99999:7:::
6 daemon:*:18682:0:99999:7:::
```

In Programm 3.5 ist zu erkennen, dass *readShadow* die Datei */etc/shadow* ohne Probleme auslesen kann. Um dies zu verhindern, wird AppArmor verwendet. Wobei *readShadow* vom *root*-User ausgeführt wird.

Programm 3.6: Generierung des AppArmor-Profils

```
1 # aa-easyprof /usr/bin/readShadow > /etc/apparmor.d/usr.bin.readShadow
```

Mittels des Tools *aa-easyprof*[22] ist es möglich AppArmor-Profile zu erstellen (siehe Programm 3.6) [29].

[20] https://apparmor.net/ (letzter Zugriff 24.02.2021)
[21] https://kernelnewbies.org/Linux_2_6_36 (letzter Zugriff 28.02.2021)
[22] https://manpages.ubuntu.com/manpages/xenial/man8/aa-easyprof.8.html (letzter Zugriff 15.04.2021)

Programm 3.7: AppArmor-Profil[23]

```
1 #include <tunables/global>
2
3 "/usr/bin/readShadow" {
4   #include <abstractions/base>
5 }
```

Das entstandene Profil (siehe Programm 3.7) basiert auf einer Vorlage[24], welches jedoch den Zugriff auf */etc/shadow* nicht erlaubt. Zur Referenzierung des Programms verwendet AppArmor Dateipfade (Zeile 3).

Programm 3.8: Aktivierung des AppArmor-Profils

```
1 # apparmor_parser -r /etc/apparmor.d/usr.bin.readShadow
2 # id
3 uid=0(root) gid=0(root) groups=0(root)
4 # readShadow
5 Segmentation fault
6 #  /usr/bin/aa-notify -s 1 -v
7 Profile: /usr/bin/readShadow
8 Operation: open
9 Name: /etc/shadow
10 Denied: r
11 Logfile: /var/log/audit/audit.log
12
13 AppArmor denial: 1 (since Wed Feb 24 06:21:10 2021)
14 For more information, please see: https://wiki.ubuntu.com/DebuggingApparmor
```

Mit dem Befehl *apparmor_parser*][25] wird in Programm 3.8 das AppArmor-Profil in den Kernel geladen (Zeile 1)[28]. Wenn nun *readShadow* wieder ausgeführt wird, kommt es zu einem Absturz des Programms (Zeile 4-5). Mit dem Befehl *aa-notify* werden Informationen bezüglich AppArmor-Nachrichten angezeigt (Zeile 6-14)][32]. Zu erkennen ist, dass die *open* Operation (Zeile 8) des Programms */usr/bin/readShadow* (Zeile 7) auf */etc/shadow* (Zeile 9), entsprechend dem AppArmor-Profil, verweigert wurde.

Programm 3.9: Neues AppArmor-Profil

```
1 #include <tunables/global>
2
3 "/usr/bin/readShadow" {
4   #include <abstractions/base>
5   /etc/shadow r,
6 }
```

Um nun den Lesezugriff auf die Datei zu erlauben, muss dies im AppArmor-Profil angegeben werden. Dazu wird die Zeile 5 hinzugefügt, siehe Programm 3.9.

[23] Kommentare im Profil wurden entfernt

[24] https://gitlab.com/apparmor/apparmor/-/blob/master/utils/easyprof/templates/default (letzter Zugriff 28.02.2021)

[25] https://manpages.ubuntu.com/manpages/bionic/man8/apparmor_parser.8.html (letzter Zugriff 15.05.2021)

Programm 3.10: Aktivierung des neuen AppArmor-Profils

```
1 # apparmor_parser -r /etc/apparmor.d/usr.bin.readShadow
2 # readShadow | head -2
3 root:$6$8QfqhzpvtlQuNh2u$kDSD7uBDaba5tPpvuKiDvGM0LdYGa3sSX33r2jFOdeeRRAfm1C8y.aS8eo.
      ogN.NeH8kOGxbM5vBfCQi0myr8.:18683:0:99999:7:::
4 daemon:*:18682:0:99999:7:::
```

In Programm 3.10 wird das neue AppArmor-Profil (siehe Programm 3.9) in den Kernel geladen (Zeile 1). Wie zu erkennen, lässt sich das Programm jetzt ausführen (Zeile 2-4).

3.3.3 Verwendung

Seit Ubuntu Version 7.10[26] wird AppArmor standardmäßig installiert und mitgeliefert [58]. Eine Liste der vorhandenen AppArmor-Profile kann im Ubuntu Wiki[27] gefunden werden. Bei Docker-Containern ist es möglich AppArmor als zusätzlichen Schutzmechanismus zu verwenden [27].

3.4 SELinux

3.4.1 Erklärung

SELinux[28] wurde im März 2001 von der NSA[29] vorgestellt und basiert auf dem MAC-Modell [13, S. 6]. Infolge dieser Vorstellung zog der Linux Gründer Linus Torvalds[30] in Betracht, ein generelles Framework für solch Sicherheitsimplementierungen zu entwickeln und somit entstand das LSM Framework (siehe Abschnitt 3.1.2) [37].

3.4.2 Implementierung

SELinux beantwortet folgende Frage: „Darf <Subjekt> <Objekt> <Aktion>?“ [8, S. 7]. Subjekte sind jene die eine Aktion ausführen, wie zum Beispiel Prozesse oder User. Objekte sind Ressourcen, auf jenen die Aktion ausgeführt wird, wie zum Beispiel eine Datei. Eine Aktion ist beispielsweise das Lesen oder Schreiben. Hierzu folgendes Beispiel: „Darf das Programm *readShadow* die Datei */etc/shadow* lesen?“.

Im Gegensatz zu AppArmor verwendet SELinux Labels und nicht Pfade. Jedem Subjekt und Objekt wird ein Label zugeordnet. Mittels diesen Labels wird in der SElinux Policy definiert, wie Subjekte und Objekte miteinander interagieren dürfen [8]. Folgendes Beispiel demonstriert wie SELinux dem Programm *readShadow* (siehe Programm 3.4) den Lesezugriff auf */etc/shadow* verbietet.

[26] https://wiki.ubuntuusers.de/AppArmor/ (letzter Zugriff 20.02.2021)
[27] https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles (letzter Zugriff 20.02.2021)
[28] Security Enhanced Linux
[29] National Security Agency
[30] https://github.com/torvalds/linux (letzter Zugriff 11.05.2021)

Programm 3.11: SElinux Status

```
1 #  cat /etc/centos-release
2 CentOS Linux release 8.3.2011
3 # sestatus
4 SELinux status:                 enabled
5 SELinuxfs mount:                /sys/fs/selinux
6 SELinux root directory:         /etc/selinux
7 Loaded policy name:             mls
8 Current mode:                   permissive
9 ...
```

Für diese Demonstration (siehe Programm 3.11) wird die Linux Distribution CentOS[31] verwendet, da diese SELinux standardmäßig aktiviert hat (Zeile 1-2). SELinux befindet sich im *permissive* Modus (Zeile 8). Hierbei werden nicht erlaubte Zugriffe nur geloggt und nicht blockiert. Im Gegensatz dazu gibt es den *enforcing* Modus, wobei nicht erlaubte Zugriffe geloggt und blockiert werden. Aktiv ist die Multi-Level Security Policy[32]. Defaultmäßig werden durch SELinux alle Zugriffe blockiert, sofern keine Ausnahmeregeln vorhanden sind.

Programm 3.12: *readShadow* mit SELinux einschränken

```
1 # id
2 uid=0(root) gid=0(root) groups=0(root) context=user_u:user_r:user_t:s0
3 # ls -laZ /usr/bin/readShadow
4 -rwxr-xr-x. 1 root root system_u:object_r:readShadow_exec_t:s0 12912 Apr 11 08:46 /
      usr/bin/readShadow
5 # getenforce
6 Permissive
7 # readShadow
8 root:$1$EAinZ.tX$vCbVlkyAs7aPSCHsgSO5v/::0:99999:7:::
9 ...
10 # setenforce Enforcing
11 # readShadow
12 Segmentation fault (core dumped)
13 # ausearch -m AVC -ts recent | grep readShadow | tail -n 2
14 type=AVC msg=audit(1618156353.633:2454): avc:  denied  { open } for  pid=9260 comm="
      readShadow" path="/etc/shadow" dev="sda3" ino=19095863 scontext=root:sysadm_r:
      sysadm_t:s0-s15:c0.c1023 tcontext=system_u:object_r:shadow_t:s0 tclass=file
15 type=AVC msg=audit(1618156353.633:2454): avc:  denied  { read } for  pid=9260 comm="
      readShadow" name="shadow" dev="sda3" ino=19095863 scontext=root:sysadm_r:
      sysadm_t:s0-s15:c0.c1023 tcontext=system_u:object_r:shadow_t:s0 tclass=file
```

Programm 3.12 ist eine Root-Shell (Zeile 1-2). Der Befehl *getenforce*[33] zeigt, dass sich SELinux im *permissive* Modus (Zeile 5-6) befindet. Das Programm *readShadow* kann problemlos ausgeführt werden (Zeile 7-9). SELinux wird mit dem Befehl *setenforce*[34] in den *enforcing* Modus gebracht (Zeile 10). Nun wird *readShadow* bei der Ausführung blockiert (Zeile 11-12). Mittels des Befehls *ausearch*[35] werden die entsprechenden Logs

[31] https://www.centos.org/ (letzter Zugriff 15.05.2021)

[32] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/mls (letzter Zugriff 12.04.2021)

[33] https://man7.org/linux/man-pages/man8/getenforce.8.html (letzter Zugriff 14.04.2021)

[34] https://man7.org/linux/man-pages/man8/setenforce.8.html (letzter Zugriff 14.04.2021)

[35] https://man7.org/linux/man-pages/man8/ausearch.8.html

dafür angezeigt (Zeile 13-15).

3.4.3 Verwendung

SELinux ist unter anderem standardmäßig auf dem Mobile Betriebssystem Android[36] und mehreren Linux Distributionen, wie RHEL[37] [38] oder Fedora[39] aktiviert. Ebenso verwendet die Open Source Applikationscontainer-Plattform Openshift[40] SELinux. Mit der strengen SELinux-Policy unter Openshift, welche Out-of-the-Box vorhanden ist, werden erfolgreich Container Escape[41] Exploits abgewehrt [18].

3.5 Probleme

Das Problem der vorhandenen Linux Security Modules ist, dass diese nicht für Container entwickelt wurden. Der Zweck dieser ist es, auf einem gesamten System Sicherheitsrichtlinien zu erzwingen, welche von dem*der Systemadministrator*in definiert werden. Dadurch entstehen zwei Probleme bei der Verwendung zur Absicherung von Containern [14] [1].

- Systemadministrator: Um Sicherheitsrichtlinien zu definieren ist ein*e Systemadministrator*in notwendig. Ein unprivilegierter Container kann nicht selbstständig eigene Sicherheitsrichtlinien definieren. (SELinux, AppArmor)
- Gesamtes System: Die Mechanismen beziehen sich auf das gesamte System. Somit können keine Sicherheitsrichtlinien für eine limitierte Anzahl an Prozessen, wie zum Beispiel für Container, definiert werden. (SELinux)

Dies bedeutet, Container müssen sich auf den*die Systemadministrator*in verlassen, dass dieser*diese globale Sicherheitsrichtlinien erstellt, welche für die Sicherheitsbedürfnisse des jeweiligen Containers ausreichend sind. Hierbei das Problem: Es ist für Container nicht möglich, selbstständig Sicherheitsrichtlinien zu definieren, welche nur auf den eigenen Container angewendet werden. Somit ist es das Ziel, dass ein Container selbstständig Linux Security Modules verwenden und Richtlinien definieren kann.

Um dieses Problem zu lösen, sind mehrere Vorschläge vorhanden. Im folgenden Abschnitt werden Security Namespace, Landlock[42] und die Verbesserungen von AppArmor genauer betrachtet.

3.6 AppArmor Namespace und benutzerdefinierte Richtlinien

Mit dem AppArmor-Policy Namespace ist es möglich, unterschiedliche Richtlinien für unterschiedliche Teile des Systems zu definieren. Damit kann ein Container Richtlinien anwenden, ohne das Hostsystem und andere Container zu beeinflussen [20]. Diese

[36] https://source.android.com/security/selinux (letzter Zugriff 06.04.2021)
[37] Red Hat Enterprise Linux
[38] https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux (letzter Zugriff 06.04.2021)
[39] https://getfedora.org/ (letzter Zugriff 20.04.2021)
[40] https://www.openshift.com/ (letzter Zugriff 06.04.2021)
[41] unerlaubtes Ausbrechen aus dem Container
[42] https://landlock.io/ (letzter Zugriff 20.04.2021)

Funktion ist bereits vorhanden. Im Entwicklungsprozess befinden sich benutzerdefinierte Richtlinien. Diese sollen es möglich machen, dass Anwendungen und Benutzer*innen selbst Richtlinien definieren dürfen, ohne dass dafür ein*e Administrator*in notwendig ist [9] [30].

3.7 Security Namespace

Wie in Abschnitt 2.1 erklärt, ist es mit Namespaces möglich, unterschiedliche Systemressourcen zu abstrahieren. Mit dem von Yuqiong et al. [14] vorgeschlagenen Security Namespace soll es möglich sein, Linux Security Modules wie SELinux oder AppArmor zu abstrahieren. Damit wird erreicht, dass diese Frameworks auch nur auf eine begrenzte Anzahl an Prozessen, wie einem Container, angewendet werden können. Ebenfalls ist es dem*der Container Besitzer*in möglich, selbstständig Sicherheitsrichtlinien zu definieren.

3.8 Landlock

Das Ziel von Landlock ist es, einem*einer Entwickler*in die Möglichkeit zu geben, die Benutzer*innen ihrer Anwendung vor Attacken auf diese besser zu schützen [50]. Durch den unprivilegierten Sandboxmechanismus soll es möglich sein, dass sich Anwendungen selbstständig isolieren und somit die Auswirkungen von Sicherheitsschwachstellen limitieren [47]. Landlock befindet sich mittlerweile im Mainline[43] Kernel und wird Teil des Linux-Kernels 5.13 sein [36].

3.9 Zusammenfassung

Die Verwendung von Linux Security Modules ist ein wichtiger Mechanismus für die Isolation von Containern [11]. Mit vorhandenen Mechanismen ist es allerdings nicht möglich, dass Container selbstständig Linux Security Modules anwenden. Jedoch sind mehrere Projekte zur Forschung und Entwicklung vorhanden, um dieses Problem zu lösen [12] [14] [9].

[43] Git Repository von Linus Torvalds mit dem Linux Source-Code

Kapitel 4

Fazit und Ausblick

Die zwei wichtigsten Bausteine für Container sind Namespaces und Control Groups. Die acht verschiedenen Namespaces werden verwendet, um globale Systemressourcen für die Verwendung mit Containern zu abstrahieren. Zum Beispiel mittels des PID Namespaces werden die Prozesse des Containers vom restlichen System isoliert. Durch den erst seit kurzem als sicher gesehenen User Namespace kann die Verwendung von privilegierten Containern vermieden werden. Unprivilegierte Container bieten besseren Schutz vor Privilegien-Eskalation. Control Groups werden verwendet, um den Ressourcenverbrauch, wie zum Beispiel CPU und Arbeitsspeicher eines Containers zu limitieren. Durch Namespaces und Control Groups wird der Anschein der Virtualisierung innerhalb eines Containers erweckt.

Wenn der Linux-Kernel kompromittiert wird, können diese beiden Mechanismen einfach ausgehebelt werden. Deshalb ist es notwendig, zusätzliche Schutzmechanismen zu verwenden. Mit Seccomp-BPF können Systemcalls blockiert werden, um mögliche Auswirkungen von kompromittierten Prozessen zu reduzieren und die Angriffsoberfläche auf den Linux Kernel zu verringern. Capabilities teilen die Rechte des Superusers in kleine Einheiten. Durch das Entfernen von Capabilities können privilegierte Prozesse innerhalb von Containern weiter eingeschränkt werden, damit wird erreicht, dass diese weniger Schaden anrichten können. Linux Security Modules wie AppArmor oder SELinux werden als zusätzlicher Schutzmechanismus verwendet. Diese beiden Mechanismen erlauben dem*der Systemadministrator*in Sicherheitsrichtlinien zu erzwingen. Damit kann verhindert werden, dass Container Zugriff auf sensible Dateien haben.

Verbesserungswürdig sind die vorhandenen Linux Security Modules. Unprivilegierten Containern ist es mit AppArmor und SELinux nicht möglich, selbstständig benutzerdefinierte Sicherheitsrichtlinien anzuwenden. AppArmor arbeitet an einer Lösung zu diesem Problem. Ebenso sind Landlock und der Security Namespace Vorschläge um genanntes Problem zu lösen. Landlock wird in naher Zukunft Teil des Linux Kernels sein.

Aus diesen Neuerungen ergibt sich ein weiterer Forschungsbedarf. Mittels einem Vergleich der neuen Sicherheitsmodule soll evaluiert werden, welche Lösung am besten für die Verwendung mit Containern geeignet ist. Ebenso eine Gegenüberstellung von alternativen, nicht Linux basierten, Mechanismen zur Isolation von Container ist von Relevanz.

Quellenverzeichnis

Literatur

[1] M. Bélair, S. Laniepce und J. Menaud. „Leveraging Kernel Security Mechanisms to Improve Container Security: A Survey“. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES '19. Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019. ISBN: 9781450371643. URL: https://doi.org/10.1145/3339252.3340502 (siehe S. 24).

[2] E. Biederman. *Multiple Instances of the Global Linux Namespaces*. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf (besucht am 01.10.2020). Linux Networx, 2006 (siehe S. 10).

[3] Diamanti. *Container Adoption Benchmark Survey*. https://diamanti.com/wp-content/uploads/2019/06/Diamanti_2019_Container_Survey.pdf (besucht am 05.06.2020). 2019 (siehe S. 1).

[4] W. Felter u. a. „An Updated Performance Comparison of Virtual Machines and Linux Containers“. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://ieeexplore.ieee.org/document/7095802 (besucht am 01.10.2020). 2015, S. 171–172 (siehe S. 1).

[5] X. Gao u. a. „Houdini's Escape: Breaking the Resource Rein of Linux Control Groups“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, S. 1073–1086. ISBN: 9781450367479. URL: https://doi.org/10.1145/3319535.3354227 (siehe S. 12).

[6] A. Grattafiori. *Understanding and Hardening Linux Containers*. https://research.nccgroup.com/wp-content/uploads/2020/07/ncc_group_understanding_hardening_linux_containers-1-1.pdf (besucht am 01.10.2020). NCC Group Whitepaper, 2016 (siehe S. 10).

[7] S. Hallyn und A. Morgan. „Linux Capabilities: making them work“. In: *Proceedings of the Linux Symposium*. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/33528.pdf (besucht am 01.04.2021). 2008 (siehe S. 17).

[8] M. Jahoda u. a. *Red Hat Enterprise Linux 7: SELinux Users's and Administrator's Guide*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/pdf/selinux_users_and_administrators_guide/Red_Hat_Enterprise_Linux-7-SELinux_Users_and_Administrators_Guide-en-US.pdf (besucht am 02.04.2021). 2021 (siehe S. 22).

[9] J. Johansen. *Overview and Recent Developments AppArmor*. https://events19.linuxfoundation.org/wp-content/uploads/2017/12/lss-eu-apparmor-overview-2018.pdf (besucht am 27.02.2021). 2018 (siehe S. 25).

[10] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200 (siehe S. 14).

[11] X. Lin u. a. „A Measurement Study on Linux Container Security: Attacks and Countermeasures“. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, S. 418–429. ISBN: 9781450365697. URL: https://doi.org/10.1145/3274694.3274720 (siehe S. 12, 25).

[12] M. Salaün. *File access-control per container with Landlock*. https://landlock.io/talks/2018-02-04_landlock-fosdem.pdf (besucht am 05.06.2020). 2018 (siehe S. 25).

[13] S. Smalley, C. Vance und W. Salamon. *Implementing SElinux as a Linux Security Module*. http://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf (besucht am 02.04.2021). 2002 (siehe S. 22).

[14] Y. Sun u. a. *Security Namespace: Making Linux Security Frameworks Available to Containers*. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-sun.pdf (besucht am 11.05.2021). USENIX Security Symposium, 2018 (siehe S. 24, 25).

[15] John R. Vacca. *Cyber Security and IT Infrastructure Protection*. 1st. Syngress Publishing, 2013. ISBN: 0124166814 (siehe S. 16).

[16] S. Vermeulen. *SELinux System Administration*. Packt Publishing, 2013. ISBN: 1783283173 (siehe S. 16).

[17] C. Wright u. a. *Linux Security Module Framework*. https://www.kernel.org/doc/ols/2002/ols2002-pages-604-617.pdf (besucht am 31.03.2021). 2002 (siehe S. 16, 17).

Online-Quellen

[18] M. Barrett. *OpenShift Protects against Nasty Container Exploit*. 2019. URL: https://www.openshift.com/blog/openshift-protects-against-nasty-container-exploit (besucht am 11. 04. 2021) (siehe S. 24).

[19] S. Beattie. *AppArmor Home*. commit 4d9844c3. 2017. URL: https://gitlab.com/apparmor/apparmor/-/wikis/home (besucht am 27. 02. 2021) (siehe S. 20).

[20] S. Beattie. *AppArmorNamespaces*. commit 2b4ae5a9. 2017. URL: https://gitlab.com/apparmor/apparmor/-/wikis/AppArmorNamespaces (besucht am 27. 02. 2021) (siehe S. 24).

[21] A. Bonventre. *Linux Sandboxing*. commit 3322f76. 2021. URL: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/linux/sandboxing.md (besucht am 08. 05. 2021) (siehe S. 10, 15).

[22] K. Cook. *sysctl: allow CLONE_NEWUSER to be disabled*. 2016. URL: https://www.openwall.com/lists/kernel-hardening/2016/01/24/10 (besucht am 08. 01. 2021) (siehe S. 10).

[23] J. Corbet. *Securely renting out your CPU with Linux*. 2005. URL: https://lwn.net/Articles/120647/ (besucht am 08. 01. 2021) (siehe S. 12).

[24] S. Dowideit. *Control groups*. commit ac999a9cb2b. 2014. URL: https://github.com/docker/docker.github.io/blob/master/config/containers/runmetrics.md (besucht am 21. 02. 2021) (siehe S. 12).

[25] S. Dowideit. *Docker Security: Kernel namespaces*. commit ac999a9cb2b. 2014. URL: https://github.com/docker/docker.github.io/blob/master/engine/security/index.md (besucht am 08. 01. 2021) (siehe S. 10).

[26] J. Edge. *Control group namespace*. 2014. URL: https://lwn.net/Articles/621006/ (besucht am 21. 02. 2021) (siehe S. 8).

[27] J. Frazelle. *AppArmor security profiles for Docker*. commit 61553fc2f53. 2015. URL: https://github.com/docker/docker.github.io/blob/master/engine/security/apparmor.md (besucht am 08. 02. 2021) (siehe S. 22).

[28] J. Johansen. *apparmor_parser(8)*. commit 71e54288b. 2014. URL: https://gitlab.com/apparmor/apparmor/-/blob/master/parser/apparmor_parser.pod (besucht am 27. 02. 2021) (siehe S. 21).

[29] J. Strandboge. *aa-easyprof(8)*. commit 1db463f4d. 2012. URL: https://gitlab.com/apparmor/apparmor/-/blob/master/utils/aa-easyprof.pod (besucht am 27. 02. 2021) (siehe S. 20).

[30] J. Johansen. *AppArmorUserDefinedPolicy*. commit a07c3003. 2020. URL: https://gitlab.com/apparmor/apparmor/-/wikis/AppArmorUserDefinedPolicy (besucht am 13. 05. 2021) (siehe S. 25).

[31] K. Balasubramanian. *ipcs(1)*. commit 6d5b69483a. 2014. URL: https://github.com/karelzak/util-linux/blob/master/sys-utils/ipcs.1.adoc (besucht am 21. 05. 2021) (siehe S. 8).

[32] K. Cook. *aa-notify(8)*. commit 6717e2990. 2014. URL: https://gitlab.com/apparmor/apparmor/-/blob/master/utils/aa-notify.pod (besucht am 27. 02. 2021) (siehe S. 21).

[33] K. Cook. *Linux Security Module Usage*. commit e163bc8e4a0cd. 2011. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/LSM/index.rst (besucht am 28. 02. 2021) (siehe S. 17).

[34] K. Cook. *seccomp(2)*. commit e9519f4f2. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man2/seccomp.2 (besucht am 20. 01. 2021) (siehe S. 12).

[35] M. Kerrisk. *capabilities(7)*. commit 3dfe7e0ddd. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/capabilities.7 (besucht am 31. 12. 2021) (siehe S. 17, 18).

[36] L. Torvalds. *Merge tag 'landlock_v34'*. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=17ae69aba89dbfa2139b7f8024b757ab3cc42f59 (besucht am 11. 05. 2021) (siehe S. 25).

[37] M. C. Chehab. *Linux Security Modules: General Security Hooks for Linux*. commit 415008af32199. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/security/lsm.rst (besucht am 28. 02. 2021) (siehe S. 22).

[38] M. Gusarov and K. Zak. *unshare(1)*. commit b9d2ddfbb1. 2016. URL: https://github.com/karelzak/util-linux/blob/master/sys-utils/unshare.1.adoc (besucht am 09. 05. 2021) (siehe S. 4, 7).

[39] M. Kerrisk. *cgroups(7)*. commit 176a42118f. 2016. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/cgroups.7 (besucht am 21. 02. 2021) (siehe S. 11).

[40] M. Kerrisk. *ipc_namespaces(7)*. commit 1d36b4e17b. 2019. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/ipc_namespaces.7 (besucht am 21. 02. 2021) (siehe S. 8).

[41] M. Kerrisk. *mount_namespaces(7)*. commit 98c28960c3. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/mount_namespaces.7 (besucht am 31. 12. 2020) (siehe S. 5).

[42] M. Kerrisk. *namespaces(7)*. commit 020357e8e4. 2013. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/namespaces.7 (besucht am 20. 01. 2021) (siehe S. 3, 9).

[43] M. Kerrisk. *netwok_namespaces(7)*. commit 2685b303e3. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/network_namespaces.7 (besucht am 31. 12. 2020) (siehe S. 4).

[44] M. Kerrisk. *pid_namespaces(7)*. commit a79bacf5f1. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/pid_namespaces.7 (besucht am 31. 12. 2020) (siehe S. 5).

[45] M. Kerrisk. *time_namespaces(7)*. commit 5bed06a99a. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/time_namespaces.7 (besucht am 31. 12. 2020) (siehe S. 6).

[46] M. Kerrisk. *user_namespaces(7)*. commit 99f04bb1e9. 2020. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/user_namespaces.7 (besucht am 31. 12. 2020) (siehe S. 6).

[47] M. Salaün. *Landlock LSM: kernel documentation*. commit 5526b45083433. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/userspace-api/landlock.rst (besucht am 20. 01. 2021) (siehe S. 25).

[48] S. Hemminger. *ip-netns(8)*. commit 2a9721f1c. 2011. URL: https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/tree/man/man8/ip-netns.8.in (besucht am 20. 01. 2021) (siehe S. 4).

[49] M. Salaün. *__NR_syscalls*. commit a49f4f81cb489. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/asm-generic/unistd.h?h=v5.11 (besucht am 14. 05. 2021) (siehe S. 12).

[50] M. Salaün. *Landlock LSM: Unprivileged sandboxing*. 2016. URL: https://lwn.net/Articles/700607/ (besucht am 14. 04. 2021) (siehe S. 25).

[51] *Seccomp BPF (SECure COMPuting with filters)*. commit 8ac270d1e29f0. 2012. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/userspace-api/seccomp_filter.rst (besucht am 22. 02. 2021) (siehe S. 12, 13).

[52] D. Shepherd. *Docker run reference: UTS settings*. commit 02c900f4c56. 2015. URL: https://github.com/docker/cli/blob/master/docs/reference/run.md (besucht am 08. 01. 2021) (siehe S. 7).

[53] M. Stanley-Jones. *Isolate containers with a user namespace*. commit c1dbb5b012e. 2021. URL: https://github.com/docker/docker.github.io/blob/master/engine/security/userns-remap.md (besucht am 08. 01. 2021) (siehe S. 7).

[54] M. Stanley-Jones. *Runtime options with Memory, CPUs, and GPUs*. commit 97d8ab167de. 2021. URL: https://github.com/docker/docker.github.io/blob/master/config/containers/resource_constraints.md (besucht am 21. 02. 2021) (siehe S. 12).

[55] M. Stanley-Jones. *Seccomp security profiles for Docker*. commit 730ec4717d0). 2016. URL: https://github.com/docker/docker.github.io/blob/master/engine/security/seccomp.md (besucht am 21. 02. 2021) (siehe S. 15).

[56] T. Tiigi. *Experimenting with Rootless Docker*. 2019. URL: https://www.docker.com/blog/experimenting-with-rootless-docker (besucht am 08. 01. 2021) (siehe S. 7, 10).

[57] J. Turnbull. *Docker Security: Linux kernel capabilities*. commit bf69b773ec4. 2014. URL: https://github.com/docker/docker.github.io/blob/master/engine/security/index.md (besucht am 08. 01. 2021) (siehe S. 19).

[58] Contributors to the Ubuntu documentation wiki. *Ubuntu ApparmorProfiles*. 2020. URL: https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles (besucht am 08. 02. 2021) (siehe S. 22).

[59] K. Zak. *lsns(8)*. commit 6d5b69483a. 2015. URL: https://github.com/karelzak/util-linux/blob/master/sys-utils/lsns.8.adoc (besucht am 20. 01. 2021) (siehe S. 4).