Container Security – Who contains the Containers?

Chemnitzer Linux-Tage 2021

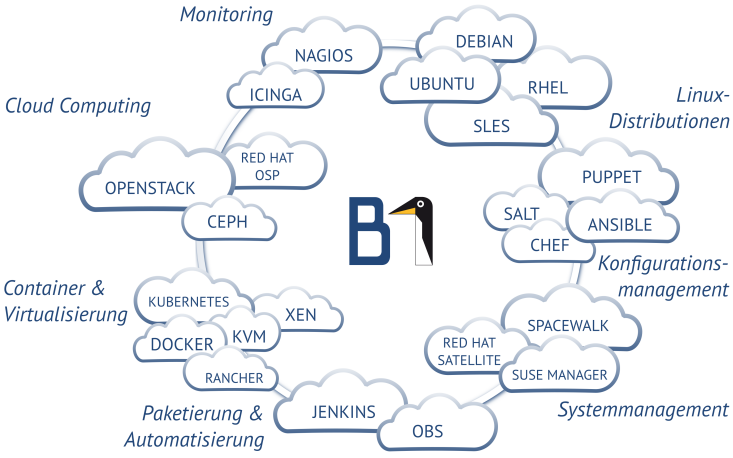
14. März 2021

Tilman Kranz
Linux Consultant & Trainer
B1 Systems GmbH
kranz@b1-systems.de

Vorstellung B1 Systems

- gegründet 2004
- Linux/Open Source-Themen
- national & international tätig
- über 125 Mitarbeiter
- unabhängig von Soft- & Hardware-Herstellern
- Leistungsangebot:
  - Beratung & Consulting
  - Support
  - Training
  - Managed Service & Betrieb
  - Lösungen & Entwicklung
- Standorte in Rockolding, Köln, Berlin & Dresden

Schwerpunkte



Einleitung

Ziele von Containern

- Portabilität
- Skalierbarkeit
- Prozess-Isolation
- hier im Fokus: Prozess-Isolation für Container auf Linux

Security für Linux-Prozesse

Versuch einer grafischen Darstellung

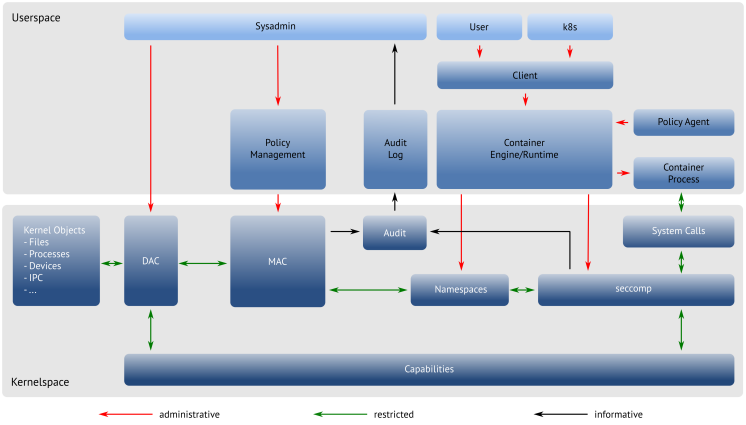


Abbildung: Linux Kernel Security

DAC – *Discretionary Access Control*

- Identitäten: „user“, „group“ (NSS)
- Prozesse haben (effektive) UID/GID
- Dateien haben Eigentümer und Berechtigungen
- Prozess-Signalisierung ist reglementiert
- bei Verletzung: `EACCESS` oder `EPERM`

MAC – *Mandatory Access Control*

- Policy Management (Userspace) steuert ...
- ... *Linux Security Module*, LSM (Kernelspace)
- LSM prüft jeden Zugriff jedes Prozesses gegen die Policy
- bei Verletzung: `EACCESS`, Meldung im Audit-Log

Beispiel für MAC – SELinux

- Labels für Benutzer, Prozesse und Dateien
- Zuordnung von Dateien mit Label Y zu Prozessen mit Label X
- Dateien müssen richtig gelabelt sein (`restorecon(8)`)
- Datei-Labels sind *Extended Attributes*

SELinux-Labels mit Docker – 1/3

- Host-Verzeichnis für Bind-Mount erstellen
- SELinux-Label mit Kategorien c1 und c2 setzen

SELinux-Label setzen

```
root@host:~# mkdir -p /container/share1
root@host:~# chcon -R \
  system_u:object_r:container_file_t:s0:c1,c2 \
  /container/share1
```

SELinux-Labels mit Docker – 2/3

Prozess mit passenden Kategorien c1 und c2 starten

```
root@host:~# docker run --rm -it --name test-container \
  -v container/share1:/share1 \
  --security-opt label=level:s0:c1,c2 \
  fedora sh
sh-5.0# ls -lZ /share1/
```

→ funktioniert

SELinux-Labels mit Docker – 3/3

Prozess mit Kategorien c2 und c3 starten

```
root@host:~# docker run --rm -it --name test-container \
  -v container/share1:/share1 \
  --security-opt label=level:s0:c2,c3 \
  fedora sh
sh-5.0# ls -lZ /share1/
```

→ funktioniert nicht

Beispiel für MAC – Apparmor

- 1 Profil bezieht sich auf 1 Executable
- Quellcode in `/etc/apparmor.d`; „übersetzen“ mit `apparmor_parser`
- Profil kann:
  - Zugriffe auf Dateien/Wildcards reglementieren
  - Netzwerk einschränken (z. B. „Nur TCP ist erlaubt“)
  - Capabilities zuweisen

Apparmor mit Docker 1/4

Standardverhalten:

```
user@host:~$ docker container run \
  --rm --name apparmor-test -it \
  alpine cat /proc/sysrq-trigger
```

→ funktioniert nicht („Permission denied“)

Apparmor mit Docker 2/4

Angepasste AppArmor-Policy erstellen:

/etc/apparmor.d/containers/test-profile

```
#include <tunables/global>
profile test-profile {
  #include <abstractions/base>
  capability dac_override,
  @{PROC}/sysrq-trigger r,
}
```

Apparmor mit Docker 3/4

Angepasste Policy übersetzen

```
root@host:~# apparmor_parser \
  --replace --write-cache \
  /etc/apparmor.d/containers/test-profile
```

Apparmor mit Docker 4/4

Angepasste Policy testen

```
user@host:~$ docker container run \
  --rm --name apparmor-test -it \
  --security-opt apparmor=test-profile \
  alpine cat /proc/sysrq-trigger
```

→ funktioniert („I/O error“)

Capabilities

- Eigenschaft von Threads
- unterteilen Admin-Rechte
- ziemlich grobe Unterteilung; siehe z. B. `CAP_SYS_ADMIN`
- können mit `setcap(8)` Dateien zugewiesen werden
- File-Capabilities sind *Extended Attributes*

Capabilities mit Docker 1/2

Starte Host-Prozess und ermittle Capabilities

```
root@host:~# sh -c 'getpcaps $$'
```

Capabilities mit Docker 2/2

Starte Container-Prozess und ermittle Capabilities

```
user@host:~$ docker run --rm -it \
    --cap-drop cap_fowner \
    --name test-container \
    fedora sh -c 'getpcaps $$'
```

System Calls

- Userspace löst im Kernel Aktionen aus
- Beispiele: `open`, `read`, `chown`, `execve`, `mount`, ...
- Linux-Kernel-Dev führt neue Syscalls ein, ...
- ... versucht aber, die alten funktionstüchtig zu lassen

Seccomp

- kann Verwendung von System Calls untersagen
- Feinsteuerung mit *Berkeley Packet Filter* (State Machine)
  - persistente BPF-Programme im Kernel
  - Docker: *Default Policy* untersagt Containern das Laden von BPF
- Filter lesen/bearbeiten benötigt `CAP_SYS_ADMIN`
- vor Kernel 4.8 angreifbar durch `CAP_PTRACE`

Seccomp mit Docker 1/4

Standardverhalten

```
user@host:~$ docker container run --rm -it \
  alpine sh -c 'grep Seccomp "/proc/$$/status"'
```

→ Default-Filter von Docker

Seccomp mit Docker 2/4

Verhalten mit `seccomp:unconfined`

```
user@host:~$ docker container run --rm -it \
  --security-opt seccomp:unconfined \
  alpine sh -c 'grep Seccomp "/proc/$$/status"'
```

→ kein Filter

Seccomp mit Docker 3/4

Angepasstes Profil `./no-mkdir.json` erstellen

```
{ "defaultAction": "SCMP_ACT_ALLOW",
  "syscalls": [
    { "name": "mkdir", "action": "SCMP_ACT_ERRNO" }
  ]
}
```

Seccomp mit Docker 4/4

Angepasstes Profil `./no-mkdir.json` verwenden

```
user@host:~$ docker container run --rm -it \
  --security-opt seccomp:no-mkdir.json \
  alpine sh -c 'mkdir /test123'
```

→ funktioniert nicht

Linux-Namespaces

Allgemeines zu Namespaces

- neuer Thread in neuen Namespaces: `clone(2)`
- nachträglich in Namespaces verschieben: `setns(2)`, `unshare(2)`
- Namespaces erstellen benötigt `CAP_SYS_ADMIN`
- Ausnahme u. U. User-Namespace
- letzter Prozess verlässt Namespace: Namespace wird entfernt

Mount-Namespaces

- initial sind alle Mounts aus Herkunfts-Namespace sichtbar
- `mount`, `umount` haben außerhalb keine Wirkung
- Ausnahme: „Shared mounts“

PID-Namespaces

- erster Prozess in Namespace hat dort PID 1
- beachte `/proc/sys/kernel/ns_last_pid`:
  PIDs sind *nicht* zufällig!
- in PID-Namespace `/proc` neu mounten
  → nur PIDs aus aktuellem Namespace sichtbar

Beispiel – Neuer PID- und Mount-Namespace

unshare startet Prozess in neuem Namespace

```
root@host:~# unshare -f -p -m bash
root@host:~# ps awxu
```

→ Prozessliste aus Host-„proc“

/proc im Mount-Namespace neu mounten

```
root@host:~# mount -t proc proc /proc
root@host:~# ps awxu
```

→ Prozessliste aus Container-„proc“

Network-Namespaces

- jedes Netzwerkgerät ist genau einem Network-Namespace zugeordnet
- Inter-Namespace-Networking: Paare von virtuellen Interfaces
- Namespace hat eigene Routingtabelle, ...
- ... und eigene iptables-Regeln (erfordert `CAP_NET_ADMIN`)

Beispiel – Network-Namespace erstellen

Neuen Network-Namespace erstellen mit `unshare`

```
root@host:~# unshare -n ip link show
```

→ nur ein Loopback-Device vorhanden

Network-Namespace mit Paketfilter

- kann Netzwerk-Traffic unterbinden
- Userspace: `iptables`, `nft`
- Kernelspace: Netlink-Socket, Kernel-Modul (z. B. `ip_tables`)
- Capability `NET_ADMIN` erforderlich

Beispiel – Network-Namespace in Docker

Container mit Capability NET_ADMIN starten

```
user@host:~$ docker container run --rm -it \
  --cap-add=NET_ADMIN \
  alpine sh
/ # apk add iptables
/ # iptables -I OUTPUT -d 8.8.8.8 -j REJECT
/ # ping 8.8.8.8
```

→ funktioniert nicht

User-Namespaces

- Namespace hat eigene Liste von UIDs und GIDs
- `clone(2)` mit Flag `CLONE_NEWUSER`:
  - erzeugt neuen Child-Namespace vom Typ `user`
  - neuer Prozess hat alle Capabilities in diesem Namespace
  - User-Namespaces können nicht-User-Namespaces „besitzen“

User-Namespace ohne Remapping

- Problem: UID x in Container = UID x auf Host
- Container-„root“ hat per Bind-Mount Zugriff auf Host-Dateien
- selbes Problem für Keyring (`keyctl(1)`)
- mögliche Lösung: MAC (erfordert Policy, Labels, ...)

User-Namespace mit Remapping

- bildet x Namespace-Benutzer auf Host-Benutzer/Gruppe X ab
- UIDs/GIDs in User-Namespace werden Sub-UIDs/GIDs von X
- unterstützt bei Docker & Podman
- TODO bei k8s (Issue #127, Stand 02/21)

User-Namespace-Remapping in Docker 1/6

Rechte von Host-„root“ via Bind-Mount testen

```
user@host:~$ docker container run --rm -it \
  -v /etc/shadow:/tmp/shadow \
  alpine \
  head -n1 /tmp/shadow
```

→ Datei ist lesbar

User-Namespace-Remapping in Docker 2/6

Eintrag in Host-Keyring erstellen

```
root@host:~# grep default_ccache_name /etc/krb5.conf
default_ccache_name = KEYRING:persistent:%{uid}
root@host:~# kinit admin@EXAMPLE.COM
root@host:~# klist
```

→ Ticket liegt im Keyring von Host-„root“

User-Namespace-Remapping in Docker 3/6

Container greift auf Host-Keyring zu

```
user@host:~$ docker container run --rm -it ubuntu
root@container:~# apt update ; apt install keyutils
root@container:~# s=$(keyctl get_persistent @s)
root@container:~# keyctl show $s
```

→ Container sieht Keyring von Host-„root“

User-Namespace-Remapping in Docker 4/6

Dedizierte Benutzer/Gruppe erstellen

```
root@host:~# adduser dockremap
root@host:~# grep dockremap /etc/sub*id
/etc/subgid:dockremap:493216:65536
/etc/subuid:dockremap:493216:65536
```

User-Namespace-Remapping in Docker 5/6

dockerd mit Option --userns-remap=... starten

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd
  --userns-remap=dockremap:dockremap ...

root@host:~# systemctl daemon-reload
root@host:~# systemctl restart docker.service
```

User-Namespace-Remapping in Docker 6/6

Test wiederholen

```
user@host:~$ docker container run --rm -it \
  -v /etc/shadow:/tmp/shadow \
  alpine head -n1 /tmp/shadow
```

→ funktioniert nicht

User-Namespace-Remapping in Podman 1/3

Standardverhalten als regulärer Benutzer

```
user@host:~$ podman container run -it --rm \
  -v /etc/shadow:/tmp/shadow \
  alpine head -n1 /tmp/shadow
```

→ funktioniert nicht

User-Namespace-Remapping in Podman 2/3

Standardverhalten als `root`

```
root@host:~# podman container run -it --rm \
  -v /etc/shadow:/tmp/shadow \
  alpine head -n1 /tmp/shadow
```

→ funktioniert

User-Namespace-Remapping in Podman 3/3

Verhalten als `root` mit Remapping

```
root@host:~# podman container run -it --rm \
  -v /etc/shadow:/tmp/shadow \
  --subuidname dockremap --subgidname dockremap \
  alpine head -n1 /tmp/shadow
```

→ funktioniert nicht

CGroup-Namespaces

- CG-Hierarchie nur sichtbar ab Container-Prozess
- In-Container `systemd(1)`:
  - Docker-Workarounds, z. B. Bind-Mount von `/sys/fs/cgroup` ...
  - ... nicht empfohlen (verletzt Isolation)
  - mit Podman einfacher
- seit Kernel 4.6

UTS- & Time-Namespaces

- UTS = *UNIX Timesharing*: abweichender Host- und Domainname
- Time = abweichende Systemzeit (seit Kernel 5.6)
- können DNS- und Zeit-basierte Verschlüsselung beeinflussen

Container Security

Container Runtime

- `runc` (Platzhirsch), Nvidia Container Runtime, Kata, ...
- erzeugt Container-Prozess
- richtet Prozess ein: Namespaces, Capabilities, TTY, ...
- startet gewünschtes Executable

Zugriff auf die Container Engine

- Authentifizierung – *Wer* darf die Engine benutzen?
- Docker: UNIX Domain Socket oder HTTPS mit TLS-Cert
- Podman: Container wird als der User gestartet, der `podman` ausführt
- k8s & Co.:
  - Benutzer, Gruppen, Rollen, Namespaces, ...
  - Identitäten aus LDAP, SAML, ...

Container Engine mit Policy-Agent

- Autorisierung – *Was* darf auf der Engine gemacht werden?
  - reglementiert Aktionen der Engine
  - inspiziert zu erstellende Objekte
- Docker: Authorization API
- k8s: Admission Control (u. a. OPA (*Open Policy Agent*))

OPA – *Open Policy Agent*

- Konfig-Sprache „Rego“ (Query/Assertion Language)
- Docker: OPA Docker Authorization Plugin (siehe Beispiel)
- außerdem: k8s, Terraform, Ceph, ...
- gefördert von *Cloud Native Computing Foundation* (CNCF)

OPA Authorization-Plugin (Docker) 1/4

Plugin installieren

```
user@host:~$ docker plugin install \
  --alias opa-docker-authz \
  openpolicyagent/opa-docker-authz-v2:0.6 \
  opa-args="-policy-file /opa/policies/authz.rego"
```

OPA Authorization-Plugin (Docker) 2/4

Plugin konfigurieren – /etc/docker/policies/authz.rego

```
package docker.authz
allow { not deny }

deny { unconfined }
unconfined { input.Body.HostConfig.SecurityOpt[_]
  == "seccomp:unconfined" }
```

OPA Authorization-Plugin (Docker) 3/4

Plugin verwenden

/etc/docker/daemon.json:

```
{ "authorization-plugins": ["opa-docker-authz:0.6"] }
```

Konfiguration einlesen

```
root@host:~# systemctl restart docker.service
```

OPA Authorization-Plugin (Docker) 4/4

Policy testen

```
user@host:~$ docker container run --rm \
  --security-opt seccomp:unconfined \
  hello-world
```

→ funktioniert nicht

Konsequenzen

Zugriff auf Container-Engine reglementieren

- automatische Pipelines / Orchestrierung verwenden
- Docker: rudimentäres Benutzermodell u. U. mit TLS-Auth+OPA

Container mit erweiterten Berechtigungen vermeiden

- Privilegien sollten generell vermieden werden
- nötige Ausnahmen punktuell und kontrolliert zulassen
- `CAP_SYS_ADMIN` ist nicht granular genug; erlaubt u. a. Deaktivieren von seccomp-Filtern

User-Namespace-Remapping verwenden

- Problem: manche Dinge gehen dann nicht mehr,
  z. B. Keyring, `mknod(2)`, ...
- falls nicht möglich:
  - Bind-Mounts reglementieren
  - MAC einrichten (falls möglich, u. U. Probleme mit Labels)
  - Gefahr von „Session Hijacking“ (via Keyring)

Container mit Kernel-Security einschränken

- seccomp filtert Syscalls, je weniger desto besser
- Capabilities: Je weniger, desto besser
- Host-SELinux kann Capabilities „überstimmen“

Policy Management für Container Engine verwenden

- bei Podman: Nicht in der Form verfügbar
- Umso mehr auf MAC & Co. achten!

Vielen Dank für Ihre Aufmerksamkeit!

Bei weiteren Fragen wenden Sie sich bitte an info@b1-systems.de oder +49 (0)8457 - 931096