

System

- [systemd](#)
- [cgroups](#)
- [namespaces](#)
- [System Load](#)

systemd

`systemd` ist der Standard-Init-Prozess und System-Manager in fast allen modernen Linux-Distributionen (z. B. Ubuntu, Debian, RHEL, Fedora, Arch Linux). Es wurde 2010 von Lennart Poettering und Kay Sievers entwickelt, um das alte, sequenzielle SysVinit-System abzulösen.

1. Was ist systemd und welche Aufgaben hat es?

Der Prozess `systemd` startet direkt nach dem Linux-Kernel und erhält immer die **Process ID (PID)** 1. Als Mutter aller Prozesse verfolgt und verwaltet es den gesamten Zustand des Betriebssystems.

Haupteigenschaften

- **Parallelisierung:** Dienste werden beim Systemstart parallel gestartet, was die Bootzeit drastisch verkürzt.
- **Socket- und D-Bus-Aktivierung:** Dienste werden erst gestartet, wenn auf ein Socket zugegriffen wird oder ein D-Bus-Aufruf eingeht (Lazy Loading).
- **Automatischer Dienst-Neustart:** Stürzt ein Prozess ab, kann `systemd` ihn sofort neu starten.
- **Prozess-Tracking mit `cgroups`:** Prozess-Gruppen werden sauber isoliert. Stoppt man einen Dienst, werden auch alle Tochterprozesse verlässlich beendet.
- **Einheitliches Logging:** Zentrale Protokollierung aller System- und Dienstmeldungen über den Daemon `journald`.

2. Das Konzept der "Units"

In `systemd` wird alles über sogenannte **Units** verwaltet. Jede Unit repräsentiert eine Systemressource oder eine Aufgabe. Es gibt verschiedene Unit-Typen, erkennbar an ihrer Dateiendung:

Unit-Typ	Endung	Beschreibung
Service	<code>.service</code>	Verwaltet Hintergrundprozesse / Daemons (z. B. <code>nginx.service</code> , <code>ssh.service</code>).
Target	<code>.target</code>	Gruppiert andere Units (entspricht den alten Runleveln, z. B. <code>multi-user.target</code>).
Socket	<code>.socket</code>	Hört auf Netzwerksockets oder IPC-Pipes zur Socket-Aktivierung.
Timer	<code>.timer</code>	Zeitgesteuerter Ausführungstrigger (Ersatz für klassische Cronjobs).
Mount / Automount	<code>.mount</code> / <code>.automount</code>	Steuert das Einhängen von Dateisystemen (<code>/etc/fstab</code> -Ersatz).
Path	<code>.path</code>	Überwacht Dateien/Ordner auf Änderungen und startet daraufhin Services.
Slice / Scope	<code>.slice</code> / <code>.scope</code>	Verwaltet Ressourcenbeschränkungen (CPU, RAM) via cgroups.

3. Dateipfade & Speicherorte für Konfigurationen

Konfigurationsdateien sind in einer strikten Hierarchie organisiert:

1. `/lib/systemd/system/` (**oder** `/usr/lib/systemd/system/`)

- Hier liegen die vom Paketmanager (APT, DNF, Pacman) installierten Standard-Units.
- **Wichtig:** Diese Dateien niemals manuell bearbeiten, da sie bei Updates überschrieben werden!

2. `/run/systemd/system/`

- Zur Laufzeit generierte Units (z. B. temporäre Einstellungen).

3. `/etc/systemd/system/`

- **Der Ort für eigene Konfigurationen.**
- Hier abgelegte Dateien haben Vorrang vor den Dateien in `/lib/systemd/system/`.

4. Aufbau einer `.service`-Unit-Datei

Eine Service-Datei besteht typischerweise aus drei Hauptabschnitten: `[Unit]`, `[Service]` und `[Install]`.

Ini, TOML

```
[Unit]
Description=Mein eigener Python-Webservice
After=network.target remote-fs.target
Wants=postgresql.service

[Service]
Type=simple
User=appuser
Group=appuser
WorkingDirectory=/var/www/myextension
ExecStart=/usr/bin/python3 /var/www/myextension/app.py
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure
RestartSec=5s

# Sicherheits-Hardening (Optional)
ProtectSystem=full
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

Die Abschnitte im Detail:

- `[Unit]`
 - `Description`: Menschlich lesbarer Name.
 - `After`: Der Dienst startet erst **nach** den hier angegebenen Units.
 - `Requires`: Strikte Abhängigkeit. Schlägt eine hier genannte Unit fehl, startet auch dieser Dienst nicht.

- `Wants`: Schwache Abhängigkeit. Versucht die genannte Unit zu starten, bricht aber bei Fehler nicht ab.
- **[Service]**
 - `Type`:
 - `simple` (Standard): Der Befehl in `ExecStart` ist der Hauptprozess.
 - `forking`: Für klassische Daemons, die sich im Hintergrund abspalten (forken).
 - `oneshot`: Kurzläufer-Skripte, die einmalig laufen und sich beenden.
 - `ExecStart`: Der exakte Befehl zum Starten (immer mit absolutem Pfad!).
 - `Restart`: Wann soll neu gestartet werden? (`always`, `on-failure`, `no`).
 - `RestartSec`: Wartezeit vor dem Neustart.
- **[Install]**
 - `WantedBy`: Bestimmt, in welchem Target die Unit beim Systemstart geladen wird (z. B. `multi-user.target` entspricht dem alten Runlevel 3 - Konsolenmodus).

5. Wichtige Befehle zur Verwaltung

Die Steuerung von `systemd` erfolgt über das Befehlszeilenwerkzeug `systemctl`.

Steuerung von Diensten

Bash

```
# Status eines Dienstes prüfen
systemctl status mein-dienst.service

# Dienst sofort starten / stoppen / neustarten
systemctl start mein-dienst
systemctl stop mein-dienst
systemctl restart mein-dienst

# Konfiguration neu laden ohne den Dienst zu stoppen
systemctl reload mein-dienst
```

```
# Autostart beim Booten aktivieren / deaktivieren
systemctl enable mein-dienst
systemctl disable mein-dienst

# Autostart aktivieren UND sofort starten
systemctl enable --now mein-dienst
```

Konfiguration bearbeiten und laden

Bash

```
# Erstellt/Editiert einen Override in /etc/systemd/system/
systemctl edit mein-dienst.service

# systemd-Daemon anweisen, Geänderte Unit-Dateien neu einzulesen (SEHR WICHTIG!)
systemctl daemon-reload
```

Targets und Systemzustände

Bash

```
# Aktuelles Standard-Target anzeigen
systemctl get-default

# Auf grafische Oberfläche (Runlevel 5) umstellen
systemctl set-default graphical.target

# Auf Konsolenmodus (Runlevel 3) umstellen
systemctl set-default multi-user.target
```

6. Logging mit `journald`

`systemd-journald` sammelt zentral alle Systemnachrichten (Kernel-Logs, stdout/stderr von Services, Syslog). Der Zugriff erfolgt über `journalctl`.

Bash

```
# Gesamtes Protokoll seit dem letzten Bootvorgang anzeigen
journalctl -b
```

```
# Live-Logs eines spezifischen Dienstes verfolgen (wie tail -f)
journalctl -u mein-dienst.service -f

# Logs der letzten 2 Stunden anzeigen
journalctl --since "2 hours ago"

# Nur Fehler-Nachrichten anzeigen
journalctl -u mein-dienst.service -p err
```

7. Weitere wichtige Komponenten des systemd-Ökosystems

`systemd` ist längst mehr als nur ein Init-System, sondern eine Suite von Werkzeugen:

- `timedatectl`: Verwaltung von Zeitzone und NTP-Ressourcen.
- `hostnamectl`: Verwaltung des Hostnamens und System-Metadaten.
- `localectl`: Verwaltung von Tastaturlayouts und System-Locales.
- `resolved`: Ein lokaler DNS-Resolver Daemon (`resolvectl`).
- `networkd`: Netzwerkkonfiguration für Interfaces.
- `logind`: Verwaltet Benutzer-Sitzungen und Power-Management (Reboot, Suspend).

cgroups

Control Groups (cgroups) spielen eine zentrale Schlüsselrolle in `systemd`. Sie bilden das fundamentale Kernel-Feature, mit dem `systemd` Prozesse **gruppiert, verfolgt, isoliert und bezüglich ihrer Systemressourcen beschränkt**.

Vor `systemd` nutzten klassische Init-Systeme (wie SysVinit) Prozess-IDs (PIDs) und PID-Dateien (`.pid`), um zu verfolgen, welche Prozesse zu welchem Dienst gehörten. Wenn ein Dienst Tochterprozesse erzeugte (z. B. ein Apache-Webserver mit vielen Worker-Prozessen) und abstürzte oder unsauber beendet wurde, blieben verwaiste Prozesse („Zombie-Prozesse“) im System zurück.

Mit cgroups gelingt `systemd` die vollständige und lückenlose Prozesskontrolle.

1. Die Kernaufgaben von cgroups in systemd

A. Lückenloses Prozess-Tracking (Keine "Verwaisten Prozesse")

Jeder Dienst, den `systemd` startet, wird in seine eigene cgroup platziert.

- Wenn ein Prozess neue Tochterprozesse startet (`fork()`), erben diese **automatisch** dieselbe cgroup.
- Selbst wenn sich ein Dämon versucht zu tarnen oder von seinem Elternprozess absplattet, bleibt er in der cgroup gefangen.
- **Ergebnis:** Wenn Sie `systemctl stop mein-dienst` ausführen, sendet `systemd` das Stopp-Signal an die *gesamte cgroup*. Dadurch werden verlässlich **alle** dazugehörigen Prozesse beendet.

B. Ressourcenbeschränkung und -kontrolle (Resource Limiting)

Über cgroups kann `systemd` genau festlegen, wie viel Hardware-Ressourcen eine Unit oder eine Gruppe von Units verbrauchen darf. Sie können das direkt in der Unit-Datei konfigurieren:

- **Arbeitsspeicher (RAM):**

- `MemoryMax=500M` – Der Dienst darf maximal 500 MB RAM nutzen. Wächst er darüber hinaus, greift der Out-Of-Memory (OOM) Killer gezielt für diesen Dienst.
- `MemoryHigh=400M` – Drosselt den Prozess sanft ab 400 MB, bevor das harte Limit erreicht wird.

- **CPU-Leistung:**

- `CPUQuota=50%` – Beschränkt den Dienst auf maximal 50% eines CPU-Kerns.
- `CPUWeight=100` – Priorisierung der CPU-Zeit im Vergleich zu anderen cgroups (Standard ist 100).

- **Festplatten-I/O:**

- `IOReadBandwidthMax=/dev/sda 10M` – Beschränkt das Lesen von der Festplatte auf 10 MB/s.

- **Tasks/Prozesse (Fork-Bomb-Schutz):**

- `TasksMax=100` – Der Dienst darf maximal 100 Unterprozesse/Threads gleichzeitig öffnen.

2. Die cgroup-Hierarchie in systemd

`systemd` baut beim Systemstart eine baumartige Struktur unter dem virtuellen Dateisystem `/sys/fs/cgroup/` auf. Diese Struktur unterteilt die Last des Systems in drei Hauptzweige (Slices):

Plaintext

```
/sys/fs/cgroup/
├─ system.slice    --> Alle Systemdienste (z. B. nginx, sshd, docker)
├─ user.slice      --> Angemeldete Benutzer & deren Sitzungen/Prozesse
└─ machine.slice   --> Virtuelle Maschinen (KVM/QEMU) & LXC/Systemd-Container
```

- **Slices (`.slice`):** Logische Gruppen, um Ressourcen hierarchisch zu verteilen (z. B. "Systemdienste bekommen insgesamt maximal 70% CPU, angemeldete User 30%").
- **Scopes (`.scope`):** Automatisch erstellte Gruppen für Prozesse, die nicht von `systemd` selbst gestartet wurden (z. B. eine SSH-Benutzersitzung).

- **Services** (`.service`): Die einzelnen Dienste innerhalb der Slices.

3. Befehle zum Anzeigen und Steuern von cgroups

Der Befehl `systemd-cgls` (cgroup list)

Zeigt den gesamten cgroup-Baum des Systems inklusive aller laufenden PIDs an:

Bash

```
systemd-cgls
```

Der Befehl `systemd-cgtop` (cgroup top)

Ähnlich wie der normale `top`-Befehl, zeigt aber den Live-Ressourcenverbrauch (CPU, RAM, I/O) aufgeteilt nach cgroups/Services an:

Bash

```
systemd-cgtop
```

Ressourcen zur Laufzeit anpassen

Sie können Limits eines laufenden Dienstes dynamisch anpassen, ohne die Unit-Datei manuell zu bearbeiten:

Bash

```
# Setzt das Arbeitsspeicher-Limit für Nginx sofort auf 1 GB
systemctl set-property nginx.service MemoryMax=1G
```

4. Die Bedeutung von cgroups für Container (Docker, Podman)

Cgroups sind zusammen mit Linux **Namespaces** die fundamentale Grundlage moderner Container-Technologien:

- **Namespaces** sorgen für die *Isolierung* (Der Container sieht nur seine eigenen Dateien, Netzwerke und Prozesse).
- **CGroups** sorgen für die *Ressourcenbegrenzung* (Der Container kann das Wirtssystem nicht lahmlegen).

Da `systemd` cgroups nativ verwaltet, integrieren sich Container-Runtimes wie Docker, Podman oder `containerd` meist direkt mit dem `systemd`-cgroup-Treiber.

namespaces

Linux Namespaces sind ein fundamentales Feature des Linux-Kernels, das die Grundlage für moderne Container-Technologien (wie Docker, Podman oder Kubernetes) bildet.

Einfach ausgedrückt: **Namespaces isolieren Systemressourcen für bestimmte Prozesse.**

Ein Prozess (oder eine Gruppe von Prozessen), der sich in einem eigenen Namespace befindet, sieht nur die Ressourcen, die diesem Namespace zugewiesen sind. Aus der Perspektive des Prozesses sieht es so aus, als hätte er ein eigenes, völlig isoliertes Linux-System für sich allein.

1. Die Analogie: Das Bürogebäude

Stell dir das Linux-Betriebssystem wie ein großes Bürogebäude vor:

- **Ohne Namespaces (Klassisches Linux):** Alle Mitarbeiter (Prozesse) arbeiten im selben Open-Space-Büro. Jeder sieht jeden, nutzt dieselbe Kaffeemaschine (Dateisystem), dieselbe Telefonanlage (Netzwerk) und sieht die gleichen Namensschilder (PIDs).
- **Mit Namespaces (Container-Prinzip):** Das Bürogebäude wird in Einzelbüros mit Milchglas und eigenen Telefonanlagen unterteilt. Mitarbeiter in *Büro A* wissen nicht einmal, dass Mitarbeiter in *Büro B* existieren. Sie haben ihre eigene Kaffeemaschine und eigene Durchwahlnummern – obwohl sie sich alle im selben Gebäude befinden und das gleiche Fundament (den Kernel) teilen.

2. Die 8 Typen von Linux Namespaces

Linux stellt verschiedene Arten von Namespaces zur Verfügung. Jeder Typ isoliert eine ganz bestimmte Systemressource:

Namespace-Typ	Isolierte Ressource	Was der Prozess sieht
PID (Process ID)	Prozess-IDs	Einen eigenen Prozess-Baum. Prozess <code>#1</code> im Namespace ist die "Initiator-PID", obwohl er auf dem Host eine ganz andere PID hat.

Namespace-Typ	Isolierte Ressource	Was der Prozess sieht
NET (Network)	Netzwerk-Stack	Eigene Netzwerk-Interfaces (<code>eth0</code> , <code>lo</code>), eigene Routing-Tabellen, Firewall-Regeln (iptables) und Port-Belegungen.
MNT (Mount)	Dateisystem-Mountpoints	Einen eigenen Mount-Baum. Änderungen an Dateisystem-Mounts betreffen nur diesen Namespace.
UTS (UNIX Timesharing)	Hostname & Domain	Einen eigenen Hostnamen (z. B. <code>my-container</code>) unabhängig vom tatsächlichen Host-Namen.
IPC (Inter-Process Comm.)	Shared Memory & Queues	Eigene System V IPC-Objekte und POSIX Message Queues (Prozesse aus verschiedenen IPC-Namespace können nicht direkt über Shared Memory kommunizieren).
USER (User IDs)	Benutzer & Gruppen (UID/GID)	Ein normaler Benutzer auf dem Hostsystem kann Root (UID 0) innerhalb seines User-Namespace sein (wichtig für die Sicherheit!).
CGROUP	Control Groups	Sichtbarkeit der eigenen Cgroup-Hierarchie (Ressourcen-Limits wie CPU/RAM).
TIME	Systemzeit	Abweichende Systemzeiten/Uhrzeiten (z. B. für Testumgebungen mit simulierten Datumswerten).

3. Wie Namespaces technisch funktionieren

Im Kernel ist jeder Prozess durch eine Struktur namens `task_struct` repräsentiert. Diese Struktur enthält Zeiger auf die jeweiligen Namespaces, zu denen der Prozess gehört.

System Calls zur Steuerung:

Es gibt drei primäre Systemaufrufe (System Calls) in C, um mit Namespaces zu interagieren:

1. `clone()`: Erstellt einen neuen Kind-Prozess (ähnlich wie `fork()`), kann aber über Flags (z. B. `CLONE_NEWNET`, `CLONE_NEWPID`) direkt neue Namespaces für das Kind anlegen.
2. `unshare()`: Erlaubt einem bereits laufenden Prozess, sich von seinen bisherigen Namespaces zu lösen und neue zu erstellen.

3. `setns()`: Klinkt einen Prozess in einen bereits existierenden Namespace ein (das nutzt z. B. `docker exec`, um sich in den Namespace eines laufenden Containers einzuklinken).

4. Praxistest: Namespaces auf der Kommandozeile ausprobieren

Du kannst Namespaces direkt mit dem Utility-Befehl `unshare` in deiner Linux-Shell testen.

Beispiel 1: Einen eigenen Hostnamen-Namespace (UTS) erstellen

Bash

```
# Erstelle ein neues UTS-Namespace und starte darin eine Bash
sudo unshare --uts bash

# Ändere den Hostnamen innerhalb des Namespaces
hostname isolierter-host

# Überprüfe den Namen:
hostname

# Output: isolierter-host
```

Wenn du nun ein zweites Terminal auf deinem Rechner öffnest und `hostname` eingibst, siehst du weiterhin deinen ursprünglichen Hostnamen.

Beispiel 2: Einen eigenen Prozess-Namespace (PID) erstellen

Bash

```
# Erstelle einen neuen PID- und Mount-Namespace
sudo unshare --pid --mount-proc --fork bash

# Welches ist die PID dieses Prozesses im Namespace?
ps aux
```

Du wirst sehen, dass deine `bash` plötzlich die **PID 1** hat und du keine anderen Prozesse deines echten Rechners mehr sehen kannst.

5. Namespaces vs. Cgroups vs. Chroot: Wo liegt der Unterschied?

Oft werden diese Begriffe im Zusammenhang mit Containern zusammengeworfen. Sie erfüllen jedoch unterschiedliche Aufgaben:

- **Namespaces = Isolierung (Was kann ich sehen?):** Sie bestimmen, welche Ressourcen (Netzwerke, Prozesse, Dateien) ein Prozess überhaupt wahrnehmen kann.
- **Cgroups (Control Groups) = Limitierung (Was darf ich nutzen?):** Sie begrenzen, wie viel CPU, Arbeitsspeicher, I/O-Bandbreite oder Disk-Performance ein Prozess nutzen darf.
- **Chroot = Pfad-Sperre (Wo darf ich hin?):** Ein älterer Mechanismus, der lediglich das Root-Verzeichnis (`/`) für einen Prozess ändert. (Mount-Namespaces sind die moderne, viel sicherere Weiterentwicklung davon).

“Ein Docker-Container ist im Grunde nichts anderes als ein ganz normaler Linux-Prozess, für den der Kernel mehrere Namespaces und Cgroups kombiniert hat.

Linux Namespaces sind die Zauberei hinter der Container-Revolution. Sie sorgen dafür, dass Prozesse gegeneinander abgeschottet werden, ohne dass dafür ein schwerfälliger Hypervisor oder ein zweites Betriebssystem (wie bei einer virtuellen Maschine) geladen werden muss. Das macht Container extrem leichtgewichtig und schnell.

User Namespaces gehören zu den mächtigsten Sicherheitsfeatures des Linux-Kernels. Sie ermöglichen die Trennung und Zuordnung (Mapping) von Benutzer- und Gruppen-IDs (UIDs/GIDs) zwischen dem Host-System und dem isolierten Namespace.

Kurz gesagt: **Ein Prozess kann innerhalb seines User Namespaces volle Root-Rechte (UID 0) besitzen, ist aber auf dem eigentlichen Host-System ein völlig gewöhnlicher, privilegienloser Benutzer (z. B. UID 1000).**

1. Das Kernprinzip: UID/GID Mapping

Das Herzstück der User Namespaces ist das sogenannte **UID Mapping**. Der Linux-Kernel übersetzt dabei die Benutzer-IDs dynamisch an den Grenzen des Namespaces.

Ein vereinfachtes Mapping-Schema sieht so aus:

Plaintext

[Innerhalb des User Namespaces]	---	[Auf dem Host-System]
UID 0 (Root im Container)	---	UID 1000 (Dein normaler User)
UID 1 (Sub-User 1)	---	UID 100001
UID 2 (Sub-User 2)	---	UID 100002

Wenn ein Prozess im Container versucht, eine Datei zu erstellen, sieht das für den Prozess im Container so aus, als gehöre sie `root` (UID 0). Schaust du dir dieselbe Datei jedoch vom Host-System aus an, gehört sie in Wahrheit dem Benutzer `1000`.

Wo wird das konfiguriert?

Der Kernel verwaltet dieses Mapping über zwei Dateien im `/proc`-Dateisystem des jeweiligen Prozesses:

- `/proc/<PID>/uid_map`
- `/proc/<PID>/gid_map`

Dort ist nach folgendem Muster hinterlegt, wie die UIDs übersetzt werden:

ID-inside-NS	ID-outside-NS	Length
--------------	---------------	--------

Beispiel: `0 100000 65536` bedeutet:

- UID `0` im Namespace entspricht UID `100000` auf dem Host.
- Es werden insgesamt `65536` fortlaufende UIDs gemappt (also von UID 0–65535 im Container auf UID 100000–165535 auf dem Host).

2. Warum das herkömmliche Docker ein Sicherheitsrisiko war

In der klassischen Docker-Architektur läuft der Docker-Daemon (`dockerd`) als echter **Root-User auf dem Host-System**.

Wenn du einen Standard-Container startest, ist der Prozess im Container standardmäßig ebenfalls als Root (UID 0) unterwegs – und zwar **ohne** User-Namespace-Mapping:

- **Container Root (UID 0) == Host Root (UID 0)**

Das Gefahrenszenario (Container Escape):

Gelingt es einem Angreifer durch eine Sicherheitslücke im Anwendungscode oder im Kernel, aus dem Container auszubrechen (ein sogenannter *Container Escape*), landet er direkt auf dem Host-System. Da seine UID auf dem Host ebenfalls `0` ist, hat er sofort die **vollständige Kontrolle über den gesamten Server**.

3. Wie Rootless Docker mit User Namespaces maximale Sicherheit schafft

Bei **Rootless Docker** (sowie nativ bei Podman) läuft schon der Docker-Daemon selbst als unprivilegierter Benutzer (z. B. UID 1000) auf dem Host. Wenn nun ein Container gestartet wird, kommt zwingend ein neuer **User Namespace** zum Einsatz.

Das bietet drei entscheidende Sicherheitsvorteile:

A. Keine Host-Root-Rechte bei einem Container-Escape

Bricht ein Angreifer aus einem Rootless-Container aus, sieht der Host-Kernel seine Aktionen nicht als `root`, sondern als gewöhnlicher Benutzer (z. B. UID 1000 oder UID 100000). Der Angreifer kann:

- Keine Systemdateien wie `/etc/shadow` lesen oder verändern.
- Keine neuen Kernel-Module laden.
- Keine Festplatten neu partitionieren oder das System herunterfahren.
- Keine Prozesse anderer Host-Benutzer manipulieren.

B. Unprivilegierte Erstellung von Namespaces

Normalerweise erfordert das Erstellen von Netzwerk- oder Mount-Namespaces Root-Rechte. **User Namespaces sind die einzige Ausnahme:** Ein normaler Benutzer darf ein neues User Namespace erstellen und erlangt *innerhalb* dieses Namespaces die Rechte (Capabilities) zum Erstellen aller anderen Namespace-Typen (NET, MNT, PID etc.).

Dadurch können normale Benutzer vollwertige Container starten, **ohne dass dafür jemals ein Root-Prozess beteiligt sein muss.**

C. Schutz vor Privilege Escalation über Suid-Binaries oder Sockets

Sollte im Container eine SUID-Datei (wie `passwd` oder `sudo`) missbraucht werden, um im Container Root-Rechte zu erlangen, bleibt dieser Rechtegewinn strikt auf die Grenzen des User Namespaces beschränkt.

4. Einschränkungen von User Namespaces & Rootless Docker

Obwohl User Namespaces die Sicherheit drastisch erhöhen, bringen sie bauartbedingte Einschränkungen mit sich:

1. **Privilegierte Ports (< 1024):** Ein Rootless-Container kann auf dem Host standardmäßig keine Ports unterhalb von 1024 (z. B. Port 80 oder 443) direkt anbinden, da dies auf dem Host weiterhin dem echten Root vorbehalten ist (lässt sich via Kernel-Parameter `net.ipv4.ip_unprivileged_port_start` anpassen).
2. **Dateirechte auf Mounts:** Wenn Host-Verzeichnisse in den Container gemountet werden (*Bind Mounts*), muss das UID-Mapping berücksichtigt werden. Dateien, die auf dem Host dem normalen User gehören, werden im Container eventuell als fremde UID angezeigt, wenn das Mapping nicht exakt passt.
3. **Cgroup v2 erforderlich:** Für eine vollständige Ressourcenbegrenzung (CPU, RAM) in Rootless-Umgebungen wird zwingend ein modernes Linux-System mit **Cgroup v2** vorausgesetzt.

User Namespaces entkoppeln die Macht der Root-Rechte innerhalb einer isolierten Umgebung von den tatsächlichen Rechten auf dem zugrundeliegenden Betriebssystem. Sie verwandeln das Konzept von Docker vom "*Alles-oder-Nichts-Prinzip*" (Root-Daemon) hin zu einem echten **Least-Privilege-Modell**.

System Load

Die **System Load** (oder *Load Average*) eines Linux-Servers gibt an, wie stark die Systemressourcen – primär die CPU und I/O-Wartezeiten – über einen bestimmten Zeitraum ausgelastet sind. Sie misst die **durchschnittliche Anzahl von Prozessen, die sich im Zustand "ausführbar" (runnable) oder "nicht unterbrechbar" (uninterruptible) befinden**.

Die 3 Werte der Load Average

Wenn man Befehle wie `uptime`, `top` oder `w` ausführt, sieht man drei Zahlenwerte für die Load Average:

Plaintext

```
load average: 0.75, 1.20, 2.05
```

Diese drei Werte repräsentieren den geglätteten Durchschnitt der Load über drei verschiedene Zeitintervalle:

1. **1 Minute** (0.75)
2. **5 Minuten** (1.20)
3. **15 Minuten** (2.05)

Anhand der Entwicklung lässt sich erkennen, wohin die Tendenz geht:

- **Steigend** (z. B. $0.50 \rightarrow 1.20 \rightarrow 2.50$): Die Last nimmt aktuell zu.
- **Fallend** (z. B. $2.50 \rightarrow 1.20 \rightarrow 0.50$): Das System kühlt sich gerade ab.

Was bedeuten die Prozess-Zustände genau?

Linux zählt für die Berechnung der Load zwei Prozess-Zustände im Kernel:

1. **Running / Runnable (R)**: Prozesse, die entweder gerade auf einer CPU ausgeführt werden oder darauf warten, von der CPU Zeit zugewiesen zu bekommen (Prozessorschlange).
2. **Uninterruptible Sleep (D)**: Prozesse, die auf eine I/O-Ressource warten (z. B. Lesen/Schreiben auf der Festplatte/SSD, Dateizugriffe über NFS, Netzwerk-I/O). Während sie warten, können sie nicht unterbrochen werden.

“ **Wichtig:** Eine hohe Load bedeutet nicht zwangsläufig, dass die CPU zu 100 % ausgelastet ist. Eine langsame oder blockierte Festplatte kann viele Prozesse in den **D**-Zustand versetzen und die Load extrem in die Höhe treiben, obwohl die CPU fast nichts zu tun hat (*I/O Wait*).

Wie interpretiert man die Zahl? (CPU-Kerne berücksichtigen)

Eine nackte Load-Zahl lässt sich nur im Kontext der **Anzahl der verfügbaren CPU-Kerne (bzw. Threads)** bewerten.

- **1 CPU-Kern:**

- **Load = 1.00:** Der Kern ist zu genau 100 % ausgelastet. Jeder Prozess bekommt sofort Rechenzeit.
- **Load = 0.50:** Der Kern ist zu 50 % ausgelastet.
- **Load = 2.00:** Das System läuft über der Kapazität. Durchschnittlich wartet 1 Prozess darauf, verarbeitet zu werden (50 % Überlastung).

- **4 CPU-Kerne:**

- **Load = 4.00:** Alle 4 Kerne sind optimal zu 100 % ausgelastet.
- **Load = 2.00:** Die CPU-Kapazität ist etwa zu 50 % ausgelastet.
- **Load = 8.00:** Die CPU-Kapazität ist doppelt überlastet.

Zustand	1 CPU-Kern	4 CPU-Kerne	Interpretation
---------	------------	-------------	----------------

Unterauslastet	< 0.70	< 2.80	Das System hat reichlich Reserven.
Optimal	0.70 – 1.00	2.80 – 4.00	Hohe Effizienz ohne Wartezeiten/Staus.
Überlastet	> 1.00	> 4.00	Prozesse müssen warten; Reaktionszeiten steigen.

Die Anzahl der verfügbaren logischen Kerne lässt sich auf dem Server wie folgt abfragen:

Bash

```
nproc
```

Der Unterschied zwischen CPU-Auslastung (%) und Load

- **CPU-Auslastung (in %):** Misst rein die verbrauchte Rechenzeit der CPU in einem bestimmten Moment. Sie kann maximal 100 % pro Kern erreichen.
- **System Load:** Ist ein kontinuierlicher Indikator, der auch **Warteschlangen** abbildet. Sie ist nach oben offen (eine Load von 50.0 auf einem System mit 4 Kernen zeigt eine extreme Überlastung an).

Diagnose-Befehle zur Überprüfung

Um herauszufinden, warum die Load hoch ist, bieten sich folgende Terminal-Werkzeuge an:

1. `uptime` / `w`: Schneller Überblick über die 1-, 5- und 15-Minuten-Load.
2. `top` / `htop`: Zeigt Echtzeit-Prozesse, CPU-Auslastung sowie den Zustand `%wa` (I/O-Wait).
3. `dstat` / `iostat`: Hilft festzustellen, ob eine Festplatte oder SSD der Flaschenhals ist.