

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-Namespaces 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-Namespaces 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?
```

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**.

Created 2026-09-25 06:00:16 UTC by Admin

Updated 2026-09-25 09:16:35 UTC by Admin