CONTAINERDAYS 2026 · HAMBURG

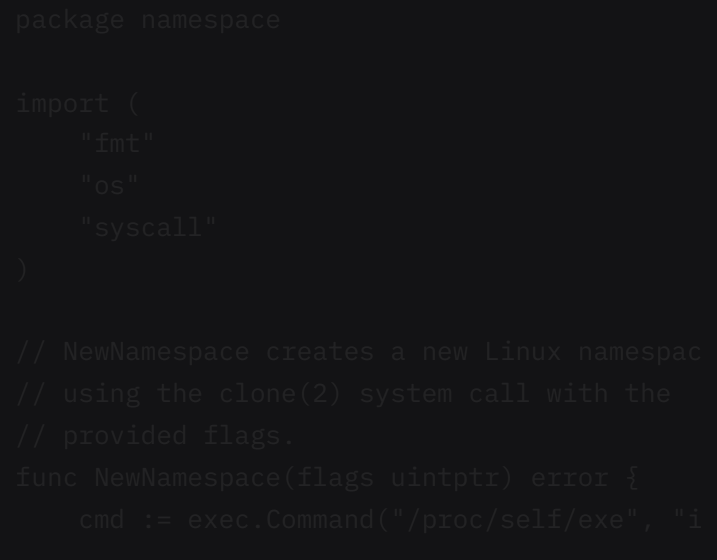
The Reality of Rootless Containers

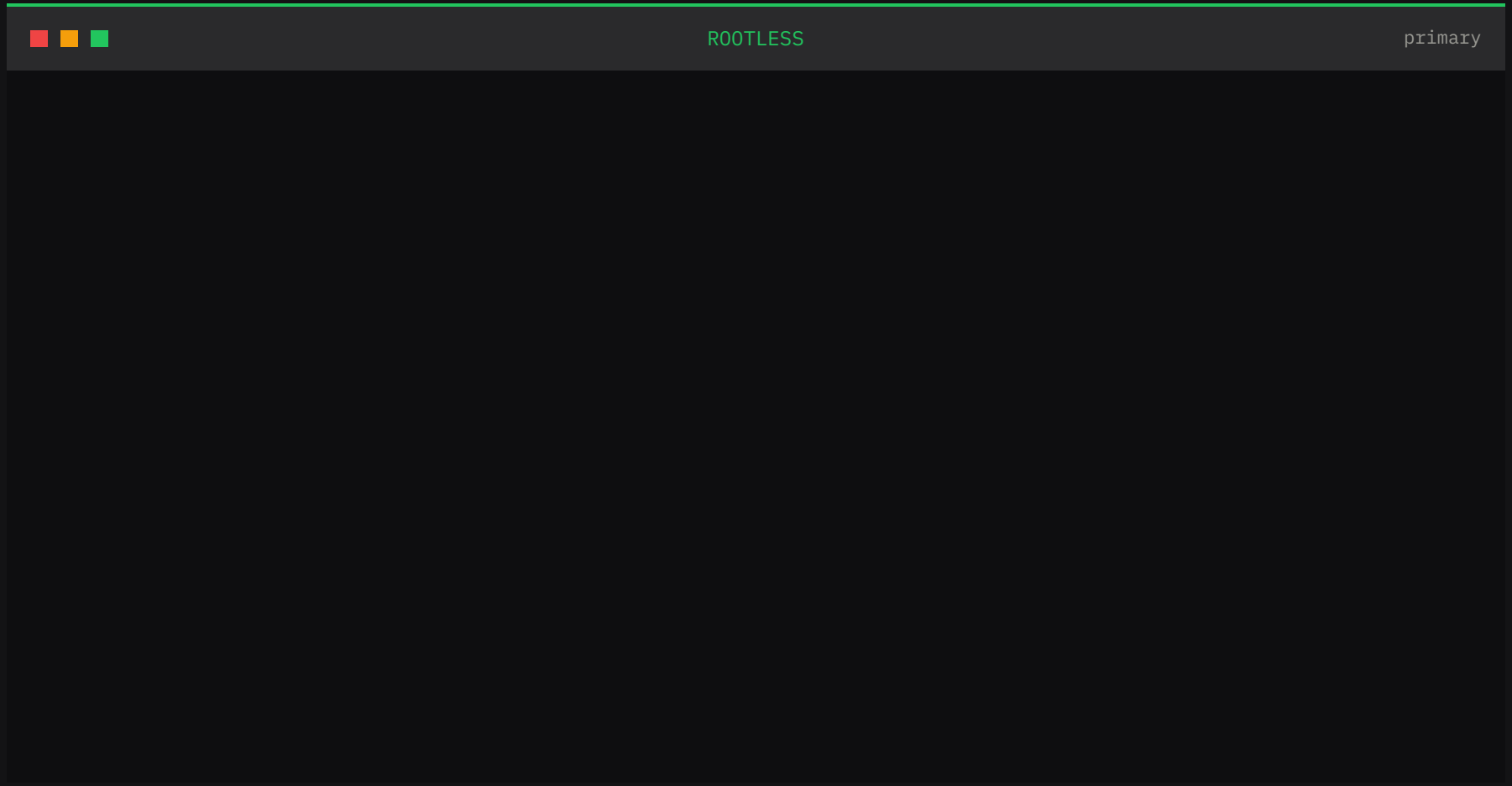
What actually changes, what does not, and what it costs

AS

Ajitem Sahasrabuddhe

Automattic · @asahasrabuddhe




ROOTLESS
primary

Both answers are true

Three claims

Three claims

1. Root is not UID 0. Root is a set of capabilities, evaluated against a namespace.

Three claims

1. Root is not UID 0. Root is a set of capabilities, evaluated against a namespace.

2. Rootless shrinks the blast radius of an escape. It does not prevent the escape.

Three claims

1. Root is not UID 0. Root is a set of capabilities, evaluated against a namespace.

2. Rootless shrinks the blast radius of an escape. It does not prevent the escape.

3. You pay for it: in features, in throughput, and in one new attack surface.

Three claims

1. Root is not UID 0. Root is a set of capabilities, evaluated against a namespace.

2. Rootless shrinks the blast radius of an escape. It does not prevent the escape.

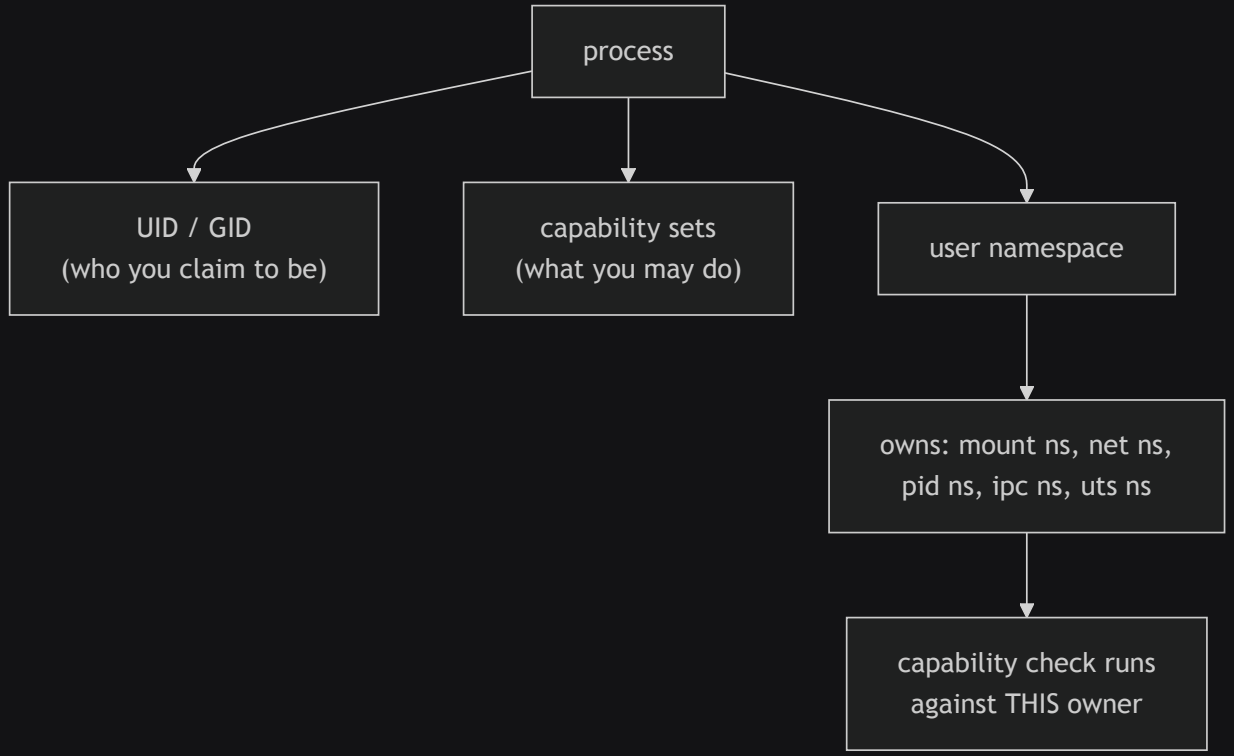
3. You pay for it: in features, in throughput, and in one new attack surface.

I run rootless. I also think most of what people believe about it is wrong.

UID 0 is a shorthand

Since Linux 2.2 the kernel has not asked ”is this UID 0?”

The credential model



A user namespace is the only namespace
that owns other namespaces

That ownership is how an unprivileged user configures a network
interface, mounts a filesystem, and sets a hostname.

Five moving parts

1	2	3	4	5
Identity	Filesystem	cgroups	Network	Execution

1. Identity: two boring files

```
$ cat /etc/subuid
ajitem:100000:65536

$ cat /etc/subgid
ajitem:100000:65536
```

1. Identity: two boring files

```
$ cat /etc/subuid
ajitem:100000:65536

$ cat /etc/subgid
ajitem:100000:65536
```

The administrator has delegated 65,536 UIDs, starting at 100000, to me.

I may map them however I like inside a namespace I create.

The map has two lines

```
/ # cat /proc/self/uid_map
         0       1000          1
         1     100000      65536
```

The map has two lines

```
/ # cat /proc/self/uid_map
         0       1000          1
         1     100000      65536
```

Container UID 0 is your own host UID. Not the start of the range.

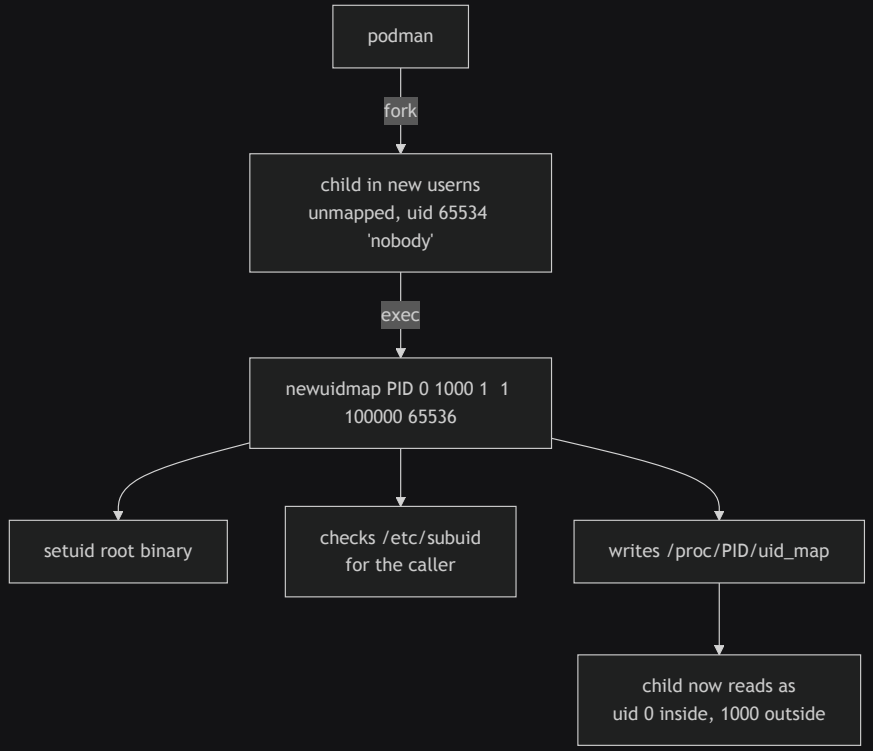
The map has two lines

```
/ # cat /proc/self/uid_map
         0       1000          1
         1     100000      65536
```

Container UID 0 is your own host UID. Not the start of the range.

A file written by "root" inside a rootless container lands on disk owned by you. A file written by UID 1000 inside lands on disk owned by 100999.

Who writes that map?



Two setuid-root binaries sit in your trust path

”Rootless” describes the runtime, not the installation.

2. Filesystem: three eras

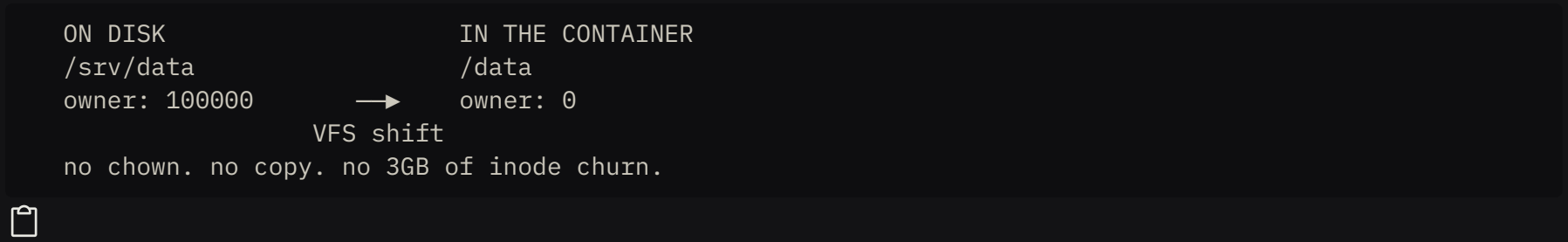
Era	Mechanism	Cost
Before 5.11	`fuse-overlayfs` in userspace	Every metadata op crosses to userspace
Kernel 5.11+	overlayfs inside a userns	Native speed for images
Kernel 5.12+	**idmapped mounts**	Shared volumes with no recursive `chown`

2. Filesystem: three eras

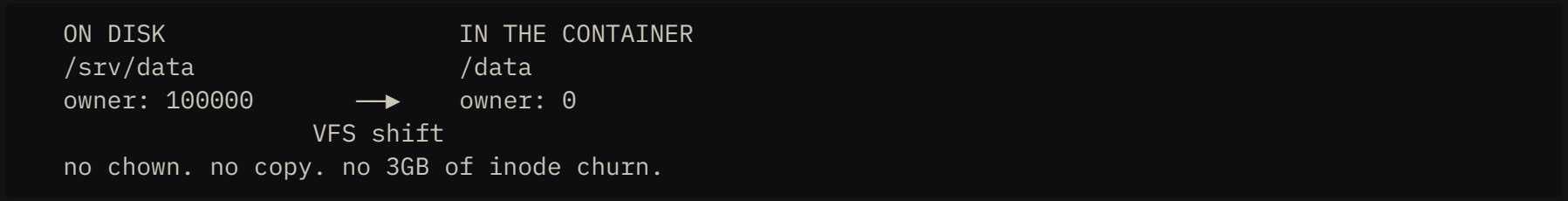
Era	Mechanism	Cost
Before 5.11	`fuse-overlayfs` in userspace	Every metadata op crosses to userspace
Kernel 5.11+	overlayfs inside a userns	Native speed for images
Kernel 5.12+	**idmapped mounts**	Shared volumes with no recursive `chown`

Most rootless folklore you have heard dates from the first row.

Idmapped mounts



Idmapped mounts



Remember this one. It is the exact kernel feature that let user-namespaced pods use volumes in Kubernetes, and we come back to it at the end.

3. cgroups: three outcomes

cgroup v2 + systemd delegation → limits work

cgroup v2, no delegation → limits silently unavailable

cgroup v1 → "not supported and ignored"

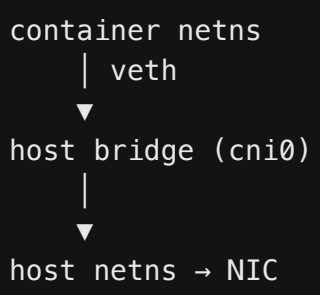
```
$ systemctl show user@$(id -u).service -p Delegate
$ cat /sys/fs/cgroup/user.slice/user-1000.slice/cgroup.controllers
cpu memory pids
```

`--memory=512m` is not an error.
It is a no-op with a warning.

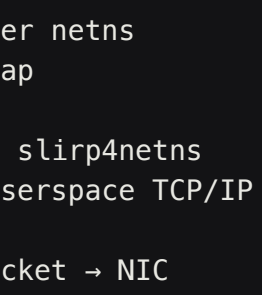
A container you believed was capped is not capped.

4. Network: nobody gave you a tap device

ROOTFUL



ROOTLESS



4. Network: nobody gave you a tap device

ROOTFUL

```
container netns
    | veth
    ▼
host bridge (cni0)
    |
    ▼
host netns → NIC
```

ROOTLESS

```
container netns
    | tap
    ▼
pasta / slirp4netns
    | userspace TCP/IP
    ▼
host socket → NIC
```

One end of a veth pair lives in the host netns. You cannot put it there.

What userspace networking costs

What userspace networking costs

- Packets are copied through a userspace process. Throughput and latency both suffer.

What userspace networking costs

- Packets are copied through a userspace process. Throughput and latency both suffer.
- Ports below 1024 refused, unless the admin lowers `net.ipv4.ip_unprivileged_port_start`

What userspace networking costs

- Packets are copied through a userspace process. Throughput and latency both suffer.
- Ports below 1024 refused, unless the admin lowers `net.ipv4.ip_unprivileged_port_start`
- `ping` works only if your GID is in `net.ipv4.ping_group_range`

What userspace networking costs

- Packets are copied through a userspace process. Throughput and latency both suffer.
- Ports below 1024 refused, unless the admin lowers `net.ipv4.ip_unprivileged_port_start`
- `ping` works only if your GID is in `net.ipv4.ping_group_range`
- The source IP the container sees is often the gateway, not the client. Your IP allow-lists and access logs are wrong.

What userspace networking costs

- Packets are copied through a userspace process. Throughput and latency both suffer.
- Ports below 1024 refused, unless the admin lowers `net.ipv4.ip_unprivileged_port_start`
- `ping` works only if your GID is in `net.ipv4.ping_group_range`
- The source IP the container sees is often the gateway, not the client. Your IP allow-lists and access logs are wrong.
- Most CNI plugins assume host privileges and simply do not apply

pasta vs slirp4netns

	slirp4netns	pasta
Addressing	Invents a subnet (10.0.2.0/24)	Copies the host’s addresses and routes
Throughput	Lower	Meaningfully better
Default in	Docker rootless	Podman 5.0+

pasta vs slirp4netns

	slirp4netns	pasta
Addressing	Invents a subnet (10.0.2.0/24)	Copies the host’s addresses and routes
Throughput	Lower	Meaningfully better
Default in	Docker rootless	Podman 5.0+

If you benchmarked rootless networking before 2024, benchmark it again.

Membership of the `docker` group
has always been root on that host

Rootless deletes that entire category of mistake.

CHAPTER

Demo A

Podman did this in one command.
What did it actually do?

A Go program cannot unshare itself

```
cmd := newChild(modeChild)
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUSER | syscall.CLONE_NEWNS,
    UidMappings: []syscall.SysProcIDMap{
        {ContainerID: 0, HostID: uid, Size: 1},
    },
    GidMappings: []syscall.SysProcIDMap{
        {ContainerID: 0, HostID: gid, Size: 1},
    },
    GidMappingsEnableSetgroups: false,
}
```

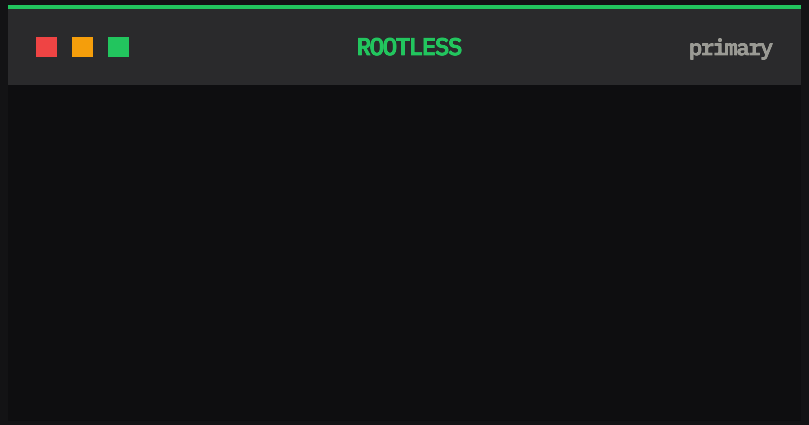
A Go program cannot unshare itself

```
cmd := newChild(modeChild)
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWUSER | syscall.CLONE_NEWNS,
    UidMappings: []syscall.SysProcIDMap{
        {ContainerID: 0, HostID: uid, Size: 1},
    },
    GidMappings: []syscall.SysProcIDMap{
        {ContainerID: 0, HostID: gid, Size: 1},
    },
    GidMappingsEnableSetgroups: false,
}
```

`CLONE_NEWUSER` is only legal for a single-threaded process. The Go runtime is multi-threaded before `main` starts. So it forks and execs. Exactly what runc does.

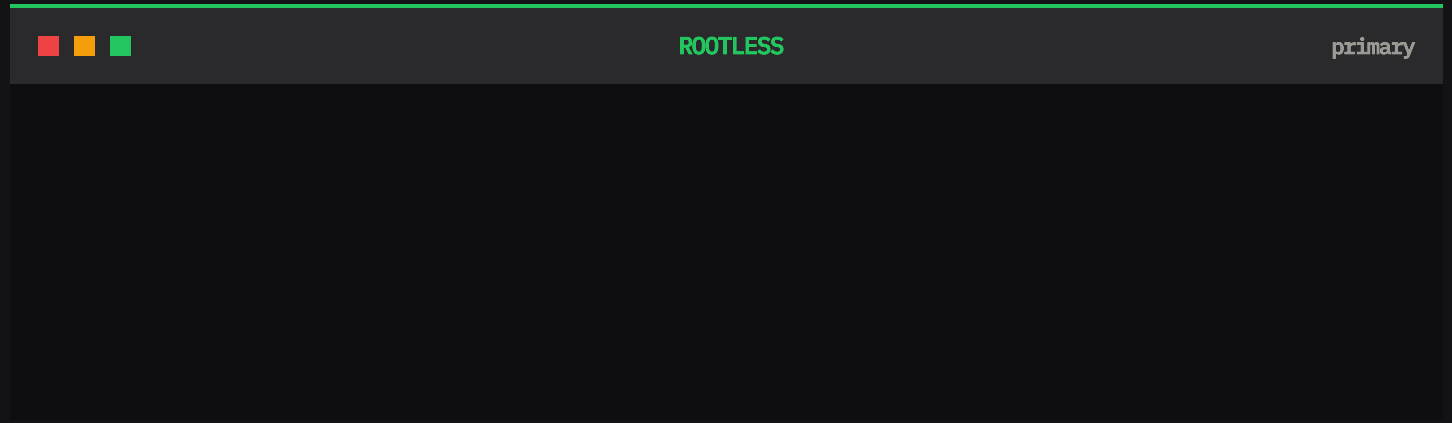
Full capability set.
Held by nobody.

```
uid=65534   uid_map: (empty)
CapEff: 0000000000000000   CapBnd: 000001ffffffffff
```



```
0 1000 1
```

Three fields. That is the entire security model.



Same PID. No setuid call.
The kernel changed its mind about who it is.

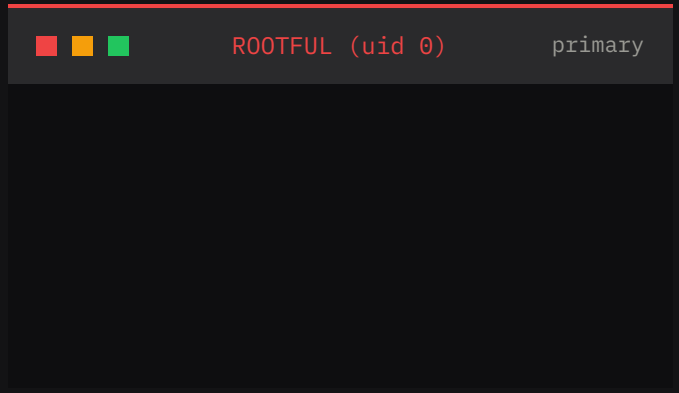
A text file, a setuid helper, and a kernel that has always cared
about namespaces more than it cared about UID 0.

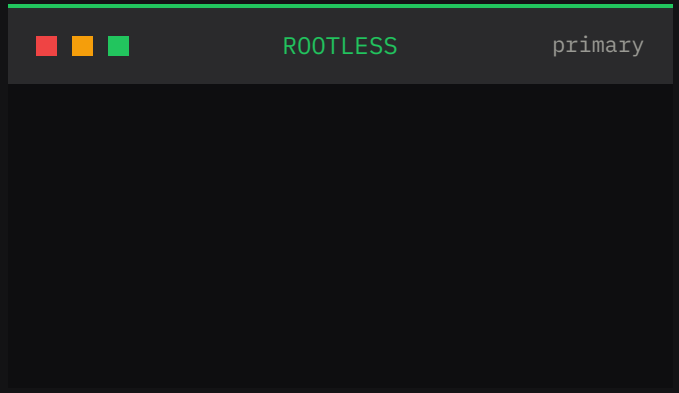
CHAPTER

Demos 1 to 5

One script. Two panes.
The only difference is where I typed it.

```
# sudo -i ; ./scripts/demo.sh 3
$         ./scripts/demo.sh 3
```





tmpfs yes. ext4 no.

`FS_USERNS_MOUNT` is a flag on the filesystem type. The capability is genuine. The set of objects it applies to is not.

The same mistake, two outcomes

```
-v /etc:/host   →   echo "backdoor::0:0::/root:/bin/sh" >> /host/passwd
```

LEFT

WROTE

That host now has an unauthenticated root account.

RIGHT

DENIED

Container UID 0 is host UID 1000. You.

ROOTFUL (uid 0) primary

ROOTLESS primary

The mount succeeded in both panes.

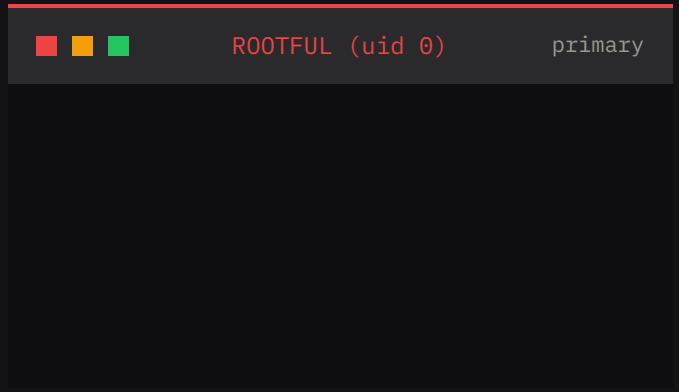
Rootless did not stop the mistake.
It made the mistake survivable.

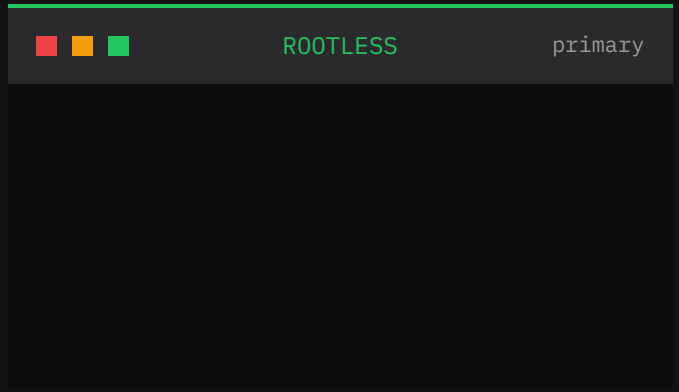
CHAPTER

Demo 4

Ask for a limit.
Then ask the container what it got.

```
67108864  =  the limit applied
anything else  =  it did not
```



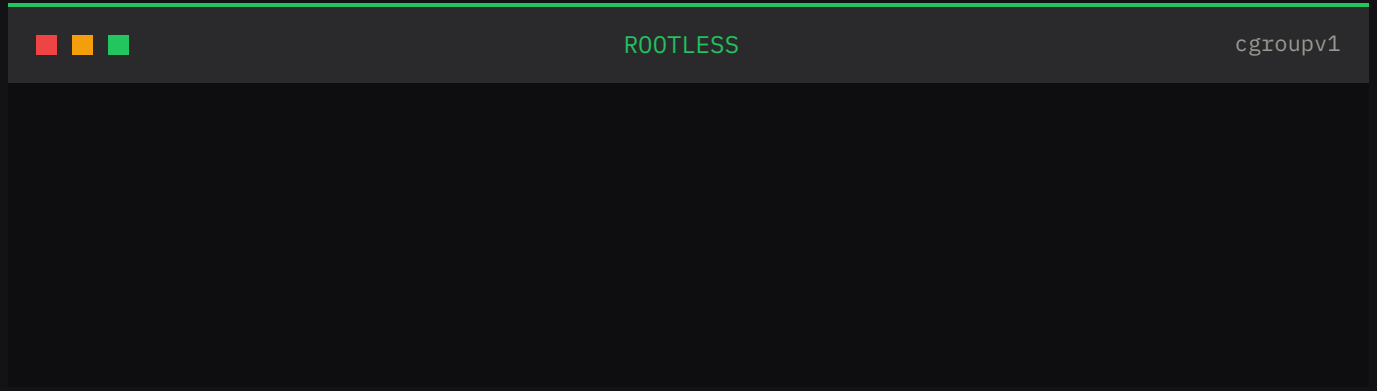


CHAPTER

Demo 4

Ask for a limit.
Then ask the container what it got.

```
67108864  =  the limit applied
anything else  =  it did not
```

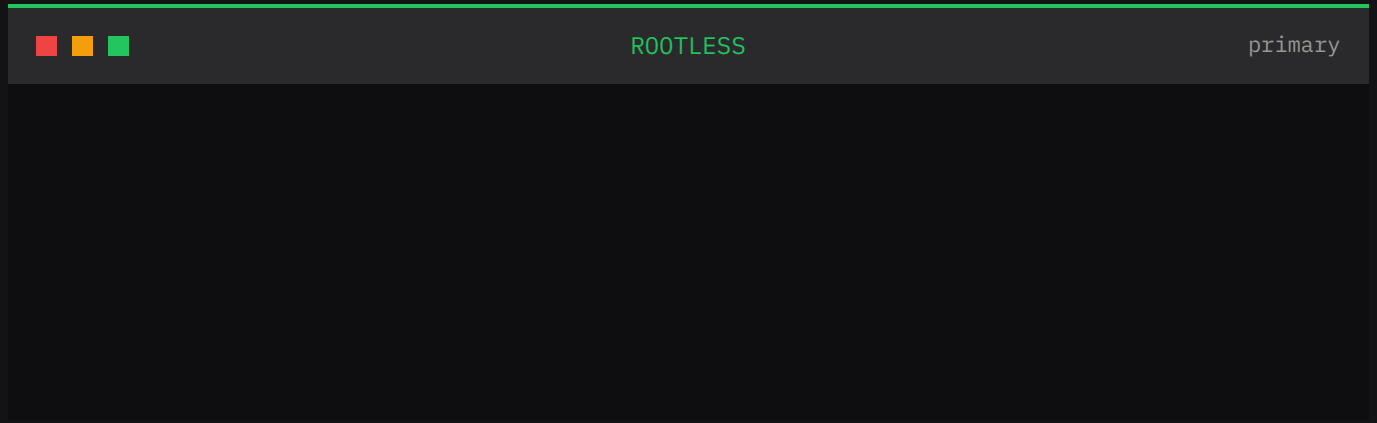


CHAPTER

Demo 5

Userspace networking, and its bill

measured beforehand. never benchmark live.



Measured on my hardware

iperf3 TCP	Rootful	pasta	slirp4netns
Gbit/s	128	82.3	28.4
vs veth	baseline	36% slower	78% slower

Rootless cold start to echo, 193 ms. Cold pull and extract of a 927 MB image.

Ubuntu 24.04, kernel 6.8, in a VM. There is no wire, so read the ratios, not the absolute numbers.

What rootless genuinely fixes

- `docker` **group equals root.** No privileged daemon, no root-owned socket.
- **Escape to host root (CVE-2019-5736).** The runtime binary is not writable by your UID.
- **Leaked fd escapes (CVE-2024-21626).** Reachable files are ones your UID already reached.
- **Careless** `-v /:/host`**.** Bounded by your UID's own permissions.
- **Multi-tenant CI runners.** Each job gets a distinct UID range.

What it does not touch

- **Kernel LPE.** One shared kernel. A kernel bug is a kernel bug.
- **Your own data.** SSH keys, cloud creds, source, `~/.kube/config` .
- **Egress abuse.** Mining, exfiltration, botnets. No root needed.
- **Supply chain.** A malicious image runs as you. That is enough.
- **Denial of service.** Especially where cgroup limits are ignored.
- **Lateral movement.** Same UID range, same everything.

Rootless *adds* an attack surface

The sysctls hardening guides argue about

```
# Debian family, the blunt instrument
$ sysctl kernel.unprivileged_userns_clone

# Ubuntu 23.10+, the surgical one
$ sysctl kernel.apparmor_restrict_unprivileged_userns
```

The sysctls hardening guides argue about

```
# Debian family, the blunt instrument
$ sysctl kernel.unprivileged_userns_clone

# Ubuntu 23.10+, the surgical one
$ sysctl kernel.apparmor_restrict_unprivileged_userns
```

A long line of local privilege-escalation bugs (filesystem mount parsers, nf_tables, overlayfs copy-up) were reachable by unprivileged users **only because** they could enter a user namespace first.

The sysctls hardening guides argue about

```
# Debian family, the blunt instrument
$ sysctl kernel.unprivileged_userns_clone

# Ubuntu 23.10+, the surgical one
$ sysctl kernel.apparmor_restrict_unprivileged_userns
```

A long line of local privilege-escalation bugs (filesystem mount parsers, nf_tables, overlayfs copy-up) were reachable by unprivileged users **only because** they could enter a user namespace first.

Ubuntu's is an allowlist, not a switch. Podman is on it. `unshare`, and the program from Demo A, are not.

The sysctls hardening guides argue about

```
# Debian family, the blunt instrument
$ sysctl kernel.unprivileged_userns_clone

# Ubuntu 23.10+, the surgical one
$ sysctl kernel.apparmor_restrict_unprivileged_userns
```

A long line of local privilege-escalation bugs (filesystem mount parsers, nf_tables, overlayfs copy-up) were reachable by unprivileged users **only because** they could enter a user namespace first.

Ubuntu's is an allowlist, not a switch. Podman is on it. `unshare`, and the program from Demo A, are not.

The feature that makes rootless containers possible is the one hardening guides restrict, and what you are trusting instead is a list of 91 binaries.

The sysctls hardening guides argue about

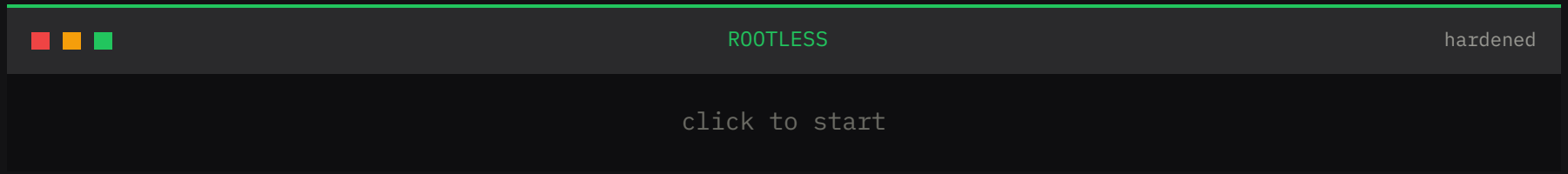
```
# Debian family, the blunt instrument
$ sysctl kernel.unprivileged_userns_clone

# Ubuntu 23.10+, the surgical one
$ sysctl kernel.apparmor_restrict_unprivileged_userns
```

A long line of local privilege-escalation bugs (filesystem mount parsers, nf_tables, overlayfs copy-up) were reachable by unprivileged users **only because** they could enter a user namespace first.

Ubuntu's is an allowlist, not a switch. Podman is on it. `unshare`, and the program from Demo A, are not.

The feature that makes rootless containers possible is the one hardening guides restrict, and what you are trusting instead is a list of 91 binaries.



Kubernetes finally caught up

```yaml
apiVersion: v1
kind: Pod
spec:
  hostUsers: false          # one field, open since 2016
  containers:
    - name: app
      image: ghcr.io/example/app:1.0
```

Kubernetes finally caught up

```yaml
apiVersion: v1
kind: Pod
spec:
  hostUsers: false          # one field, open since 2016
  containers:
    - name: app
      image: ghcr.io/example/app:1.0
```

GA in **v1.36**, April 2026. Alpha 1.25, beta 1.30, default 1.33. Persistent volumes under a user namespace only became practical once idmapped mounts landed in 5.12.

Kubernetes finally caught up

```yaml
apiVersion: v1
kind: Pod
spec:
  hostUsers: false          # one field, open since 2016
  containers:
    - name: app
      image: ghcr.io/example/app:1.0
```

GA in **v1.36**, April 2026. Alpha 1.25, beta 1.30, default 1.33. Persistent volumes under a user namespace only became practical once idmapped mounts landed in 5.12.

Caveats: modern kernel and CRI support required, some volume and device types restricted, each pod consumes a 65,536-UID range so pods-per-node is capped, and the container still runs as UID 0 inside.

So: should you?

Context	Verdict
CI runners, untrusted builds	**Strong yes.** The best argument rootless has.
Developer laptops	**Yes.** Loses almost nothing.
Shared multi-tenant hosts	**Yes**, with per-user subuid ranges.
Kubernetes workers	**Yes**, `hostUsers: false`, no longer exotic.
Network-heavy workloads	**Measure first.**
Host devices, GPUs, low ports	**Probably not.**
Hosts where userns is disabled	**No, and that is coherent.**

Five things to take away

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.

2. Rootless shrinks the blast radius. It does not prevent the mistake.

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.

2. Rootless shrinks the blast radius. It does not prevent the mistake.

3. Your own data is inside the blast radius. On a laptop or a CI runner, that is most of what matters.

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.
2. Rootless shrinks the blast radius. It does not prevent the mistake.
3. Your own data is inside the blast radius. On a laptop or a CI runner, that is most of what matters.
4. What you lose is concrete: cgroup limits on old hosts, network throughput, low ports, some devices.

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.

2. Rootless shrinks the blast radius. It does not prevent the mistake.

3. Your own data is inside the blast radius. On a laptop or a CI runner, that is most of what matters.

4. What you lose is concrete: cgroup limits on old hosts, network throughput, low ports, some devices.

5. Rootless needs unprivileged user namespaces, which is itself an attack surface. Know your threat model, then choose.

Five things to take away

1. Root is a capability set evaluated against a user namespace, not UID 0.
2. Rootless shrinks the blast radius. It does not prevent the mistake.
3. Your own data is inside the blast radius. On a laptop or a CI runner, that is most of what matters.
4. What you lose is concrete: cgroup limits on old hosts, network throughput, low ports, some devices.
5. Rootless needs unprivileged user namespaces, which is itself an attack surface. Know your threat model, then choose.

Not a security button. A smaller, sharper blast radius, bought with real trade-offs. That is a good deal, as long as you know you are making it.

// THAT'S A WRAP

Thank You

Slides, code, and demo, all on GitHub:

`cds-2026-hamburg-crl-slides` `cds-2026-hamburg-crl-code`

`@asahasrabuddhe`

scan for slides + repo