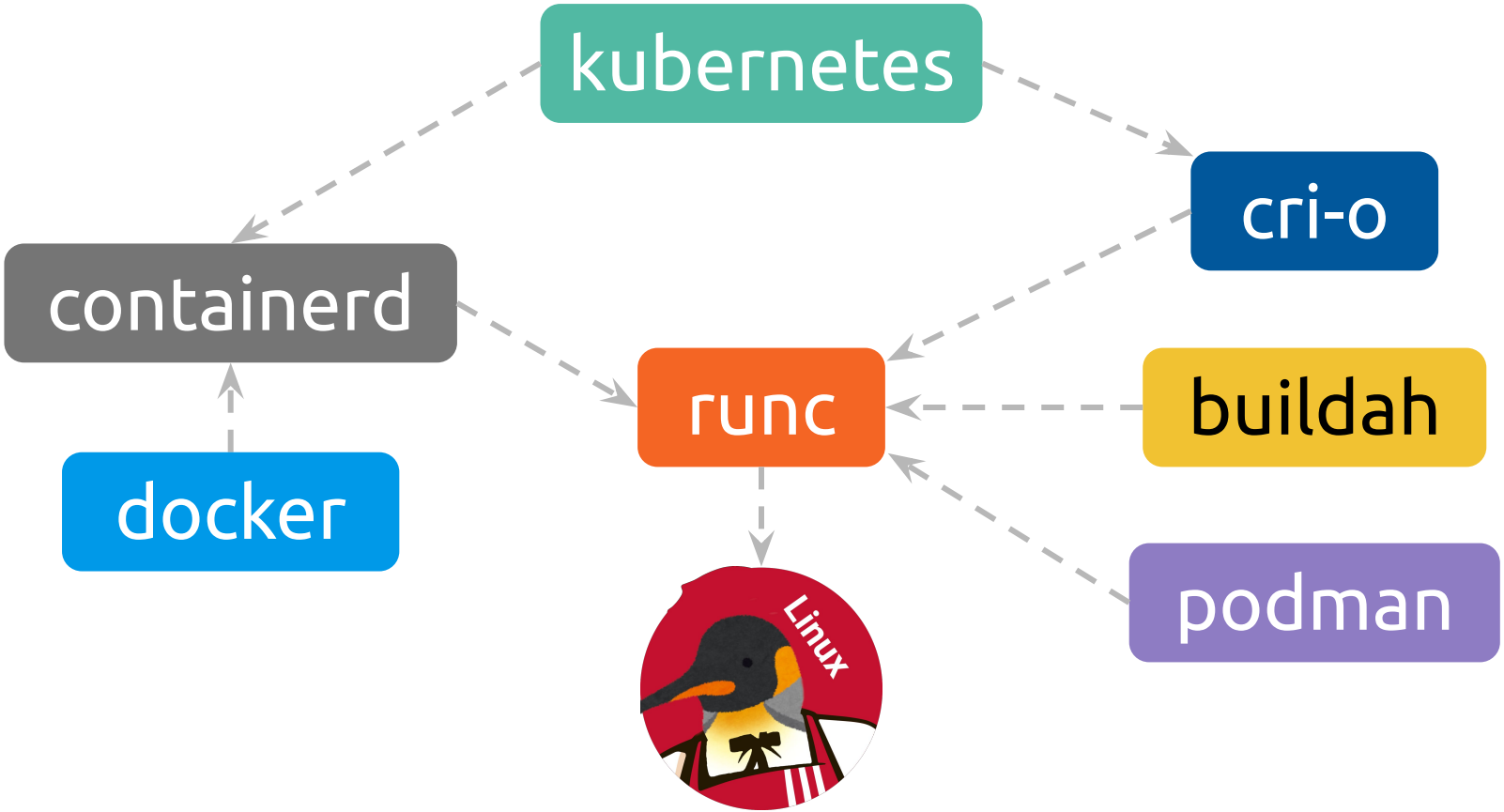
AMUTABLE

Path Safety “in the Trenches”

Aleksa Sarai – github.com/cyphar
Founding Engineer @ Amutable
FOSDEM 2026-01-31

runc


kubernetes
containerd
cri-o
runc
buildah
docker
podman
Linux

```
$ cat config.json
{
    "process": { ... },
    "root": { "path": "path/to/rootfs" },
    "mounts": [
        { "destination": "/proc", "type": "proc" },
        ...
    ],
    "linux": {
        "resources": { "devices": [ ... ] },
        "namespaces": [ ... ],
        ...
    }
}
```

```
$ cat config.json
{ ... }
$ runc run ctr-name
ctr# ...
```

path safety

“**regular**” path safety

“**strict**” path safety (procfs)

regular path safety

- When operating on a path, a path component might be swapped with a symlink or moved.
- Classic time-of-check-to-time-of-use attacks abound.

example: regular path (un)safety

```
int fd1 = open("/rootfs/etc/foo", O_CREAT|O_TRUNC|O_RDWR, 0755);
int fd2 = open("/rootfs/etc/hosts", O_NOFOLLOW|O_RDONLY);

mkdir("/rootfs/foo", 0755);
mkdir("/rootfs/foo/bar", 0755);

link("/rootfs/foo/bar/baz", "/rootfs/foo/bar/boop");

unlink("/rootfs/foo/bar/baz");
```

example: regular path (un)safety

```
int fd1 = open("/rootfs/etc/foo", O_CREAT|O_TRUNC|O_RDWR, 0755);
int fd2 = open("/rootfs/etc/hosts", O_NOFOLLOW|O_RDONLY);

mkdir("/rootfs/foo", 0755);
mkdir("/rootfs/foo/bar", 0755);

link("/rootfs/foo/bar/baz", "/rootfs/foo/bar/boop");

unlink("/rootfs/foo/bar/baz");
```

surely this isn't *that* common…

surely this isn't *that* common…

- CVE-2017-1002101
- CVE-2018-15664
- CVE-2019-16884
- CVE-2019-19921
- CVE-2021-30465
- CVE-2023-27561
- CVE-2023-28642
- CVE-2024-1753
- CVE-2024-45310
- CVE-2024-0132
- CVE-2024-0133
- CVE-2024-9676
- CVE-2025-31133
- CVE-2025-52565
- … and so on …

regular path safety – openat2

```
int openat2(int dirfd, const char *path,
            struct open_how *how, size_t size);

struct open_how {
    u64 flags;    /* O_* flags */
    u64 mode;     /* O_CREAT file mode */
    u64 resolve;  /* resolution flags */
};
```

regular path safety – openat2

- Most programs can make do with one of the following:
  - `RESOLVE_IN_ROOT` – `chroot(2)`-like lookups
  - `RESOLVE_BENEATH` – restricted lookups
  - `RESOLVE_NO_SYMLINKS` – better `O_NOFOLLOW` (/ still escapes)
- Requires the program to primarily use file descriptors.

example: regular path safety (i)

```
int root = open("/rootfs", O_DIRECTORY|O_PATH);

struct open_how how = { .resolve = RESOLVE_IN_ROOT };

how.flags = O_CREAT|O_TRUNC|O_RDWR;
how.mode = 0755;
int fd1 = openat2(root, "/etc/foo", &how, sizeof(how));

how.flags = O_NOFOLLOW|O_TRUNC|O_RDWR;
how.mode = 0;
int fd2 = openat2(root, "/etc/hosts", &how, sizeof(how));
```

example: regular path safety (ii)

```
int root = open("/rootfs", O_DIRECTORY|O_PATH);

struct open_how how =
    { .flags = O_DIRECTORY|O_PATH,
      .resolve = RESOLVE_IN_ROOT };
int dfd = openat2(root, "/foo", &how, sizeof(how));
mkdirat(dfd, "bar", 0755); /* mkdir(/rootfs/foo/bar) */
```

regular path safety – `O_PATH`

- Implement per-component lookups in userspace (very finicky).
- Still requires file-descriptor-based code.
- Usually needs `readlink("/proc/self/fd/$n")` verification.
- *See:* systemd's `chaseat`, Go's `os.Root`, `libpathrs`.

libpathrs

- Rust library that wraps the most commonly needed filesystem operations on a root filesystem with friendly™ C FFI interfaces.
  - Also has Go and Python bindings.
  - `pathrs-lite` – pure-Go port.
- Transparently supports `openat2` and the `O_PATH` fallback.

example: libpathrs (C)

```c
#include "<pathrs.h>"

int root = pathrs_open_root("/rootfs"); /* or open(2) */

int fd1 = pathrs_inroot_creat(root, "/etc/foo",
                              O_RDWR|O_TRUNC, 0644);
int fd2 = pathrs_inroot_open(root, "/etc/hosts",
                             O_NOFOLLOW|O_RDONLY);

pathrs_inroot_hardlink(root, "/foo/bar/baz", "/foo/bar/boop");

pathrs_inroot_unlink(root, "/foo/bar/baz");
```

example: libpathrs (rust)

```rust
use pathrs::{Root, flags::OpenFlags};
let root = Root::open("/path/to/root")?;

// Resolve and open a file.
let passwd = root
        .resolve("/etc/passwd")?
        .reopen(OpenFlags::O_RDONLY)?;
// ... or ...
let passwd = root.open_subpath(
        "/etc/passwd", OpenFlags::O_RDONLY
)?;

// Create a new file and open it (O_CREAT).
let newfile = root.create_file(
        "/etc/newfile",
        OpenFlags::O_RDWR,
        &Permissions::from_mode(0o755),
)?;
```

```rust
// Create a symlink.
let newfile = root.create(
        "/link",
        &InodeType::Symlink("/target".into()),
)?;

// mkdir -p
let dir = root.mkdir_all(
        "/foo/bar/baz",
        &Permissions::from_mode(0o755),
)?;

// rm -r
root.remove_all("/foo/bar")?;

// See the docs for more info.
```

strict path safety

`/proc/self/attr/exec`,
`/proc/self/mountinfo`,
`/proc/sys/$sysctl`, etc.

strict path safety

- For certain pseudo-filesystems we need to ensure we are operating on an exact path.
    - `procfs` is most critical.
- Overmounts or fake mounts can trick us into doing dangerous operations or make operations a no-op.

example: strict path (un)safety

```
int lfd = open("/proc/self/attr/exec", O_RDWR);
dprintf(lfd, "exec docker-default\n");

int pfd = open("/proc/sys/net/ipv4/ping_group_range", O_RDWR);
dprintf(pfd, "0 0\n");

int reopen = open("/proc/thread-self/fd/123", O_RDWR);

execve("/proc/self/exe", ...);
```

surely not!

okay, it's a *bit* less common…

- CVE-2019-16884
- CVE-2019-19921
- CVE-2025-52881

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_WRONLY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int lfd = openat2(procfd, "self/attr/exec", &how, sizeof(how));
dprintf(lfd, "exec docker-default\n");
```

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_WRONLY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int lfd = openat2(procfd, "self/attr/exec", &how, sizeof(how));
dprintf(lfd, "exec docker-default\n");
```

magic-links?

`/proc/self/exe`,
`/proc/self/fd/$n`,
`/proc/self/root`, etc.

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_RDWR,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int reopen = openat2(procfd, "thread-self/fd/123",
                     &how, sizeof(how));
```

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_RDWR,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int reopen = openat2(procfd, "thread-self/fd/123",
                     &how, sizeof(how));
```

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: strict path (un)safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

```
man 2 fsopen
man 2 fspick
man 2 fsconfig
man 2 fsmount
man 2 open_tree*
man 2 move_mount
```

example: strict path safety

```
int procfd = open("/proc", O_DIRECTORY|O_PATH);
/* check PROC_SUPER_MAGIC and PROC_ROOT_INO */

struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: strict path safety

```
int fsfd = fsopen("proc", 0);
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, 0, 0);
int procfd = fsmount(fsfd, 0, MOUNT_ATTR_NOEXEC|...);

struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* for open_tree(2) -- validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: strict path safety

```
int fsfd = fsopen("proc", 0);
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, 0, 0);
int procfd = fsmount(fsfd, 0, MOUNT_ATTR_NOEXEC|...);
struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* for open_tree(2) -- validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: strict path safety

```
int fsfd = fsopen("proc", 0);
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, 0, 0);
int procfd = fsmount(fsfd, 0, MOUNT_ATTR_NOEXEC|...);
struct open_how how =
     { .flags = O_DIRECTORY,
       .resolve = RESOLVE_BENEATH|RESOLVE_NO_XDEV };
int fdfd = openat2(procfd, "thread-self/fd", &how, sizeof(how));
/* for open_tree(2) -- validate no overmounts */
int reopen = openat(fdfd, "123", O_RDWR);
```

example: libpathrs (C)

```c
#include "<pathrs.h>"

int lfd = pathrs_proc_open(PATHRS_PROC_SELF,
                           "attr/exec", O_WRONLY);
dprintf(lfd, "exec docker-default\n");

int pfd = pathrs_proc_open(PATHRS_PROC_ROOT,
                           "sys/net/ipv4/ping_group_range",
                           O_RDWR);
dprintf(pfd, "0 0\n");

int reopen = pathrs_reopen(123, O_RDWR); // or pathrs_proc_open
```

example: libpathrs (rust)

```rust
use pathrs::{flags::OpenFlags, procfs::*};

// Open *regular* file.
let attr_file = ProcfsHandle::new()?
        .open(
                ProcfsBase::ProcThreadSelf,
                "attr/exec",
                OpenFlags::O_WRONLY,
        )?;

// Create your own private handle.
let handle =
        ProcfsHandleBuilder::new()
                .unmasked(true)
                .build()?;
```

```rust
// Open a magic-link.
let exe = ProcfsHandle::new()?.open_follow(
        ProcfsBase::ProcSelf,
        "exe",
        OpenFlags::O_RDONLY,
)?;

// Equivalent to readlinkat(fd, "").
let fd_path = ProcfsHandle::new()?.readlink(
        ProcfsBase::ProcThreadSelf,
        format!("fd/{}", file.as_raw_fd()),
        OpenFlags::O_RDONLY,
)?;
```

“wait, why do we care about this?”

runc

threat models

- runc (currently) has no real threat model.
- Most vulnerabilities have been “misconfiguration” bugs.
- Higher-level runtimes let unprivileged users do wacky things.
- Most people **still** don’t use user namespaces.

pop quiz!

is this a vulnerability? (i)

```
$ cat config.json
{
    "linux": { "namespaces": [] },
    ...
}
```

is this a vulnerability? (i)

```
$ cat config.json
{
    "linux": { "namespaces": [] },
    ...
}
```

is this a vulnerability? (ii)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/host",
            "source": "/",
            "type": "bind",
            "options": [ "rbind" ] }
     ],
     ...
}
```

is this a vulnerability? (ii)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/host",
            "source": "/",
            "type": "bind",
            "options": [ "rbind" ] }
     ],
     ...
}
```

is this a vulnerability? (iii)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/volume",
            "source": "/some/volume/path",
            "type": "bind",
            "options": [ "rbind" ] },
          ...
     ],
     ...
}
```

is this a vulnerability? (iii)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/volume",
            "source": "/some/volume/path",
            "type": "bind",
            "options": [ "rbind" ] },
          ...
     ],
     ...
}
```

CVE-2025-31133 / CVE-2025-52565

```
$ cat config.json
{
      "mounts": [
            ...,
            { "destination": "/volume",
              "source": "/some/volume/path",
              "type": "bind",
              "options": [ "rbind" ] },
            ...
      ],
      ...
}
```

CVE-2025-31133 / CVE-2025-52565

- `/volume` is a symlink to `/dev`.
- Racing process swaps files in `/some/volume/path` with symlink to `/proc/sys/kernel/core_pattern`:
  - CVE-2025-31133 – `/dev/null` (masked files)
  - CVE-2025-52565 – `/dev/pts/0` (`/dev/console`)
- Bind-mount *source* becomes `/proc/sys/kernel/core_pattern`, creating a `rw` bind-mount to a masked procfs file.

solutions

- Added much stricter validation of special inodes we use.
  - *Takeaway:* “Safe” `major:minor` numbers are very handy.
- Mountpoint creation was moved to `libpathrs` (`pathrs-lite`).
  - Everything is now (mostly) file-descriptor-based.
  - *Takeaway:* Wow, regular path safety is a good idea!
- `TIOCGPTPEER` for consoles.

is this a vulnerability? (iv)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/proc",
            "source": "/fake/procfs",
            "type": "bind",
            "options": [ "rbind" ] },
     ],
     ...
}
```

is this a vulnerability? (iv)

```
$ cat config.json
{
      "mounts": [
            ...,
            { "destination": "/proc",
              "source": "/fake/procfs",
              "type": "bind",
              "options": [ "rbind" ] },
      ],
      ...
}
```

is this a vulnerability? (v)

```
$ cat config.json
{
      "mounts": [
            ...,
            { "destination": "/proc/1/attr/apparmor/exec",
              "source": "/proc/1/sched",
              "type": "bind",
              "options": [ "rbind" ] },
      ],
      ...
}
```

is this a vulnerability? (v)

```
$ cat config.json
{
      "mounts": [
            ...,
            { "destination": "/proc/1/attr/apparmor/exec",
              "source": "/proc/1/sched",
              "type": "bind",
              "options": [ "rbind" ] },
      ],
      ...
}
```

is this a vulnerability? (v)

```
$ cat config.json
{
      "mounts": [
            ...,
            { "destination": "/proc/1/attr/apparmor/exec",
              "source": "/proc/1/sched",
              "type": "bind",
              "options": [ "rbind" ] },
      ],
      ...
}
```

is this a vulnerability? (vi)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/foo", "source": "/some/cache/path",
            "type": "bind", "options": [ "rbind" ] },
          { "destination": "/foo/link/thread-self/attr/apparmor",
            "source": "/some/other-cache/path",
            "type": "bind", "options": [ "rbind" ] },
     ],
     ...
}
```

is this a vulnerability? (vi)

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/foo", "source": "/some/cache/path",
            "type": "bind", "options": [ "rbind" ] },
          { "destination": "/foo/link/thread-self/attr/apparmor",
            "source": "/some/other-cache/path",
            "type": "bind", "options": [ "rbind" ] },
     ],
     ...
}
```

CVE-2025-52881

```
$ cat config.json
{
     "mounts": [
          ...,
          { "destination": "/foo", "source": "/tmp/cache-foo",
            "type": "bind", "options": [ "rbind" ] },
          { "destination": "/foo/link/thread-self/attr/apparmor",
            "source": "/tmp/cache-apparmor",
            "type": "bind", "options": [ "rbind" ] },
     ],
     ...
}
```

CVE-2025-52881

- `/tmp/cache-apparmor/exec` is a symlink to a fun target:
  - `/proc/1/sched` – (no-op)
  - `/proc/sysrq-trigger` (crash – `"exec docker-default"`)
- Racing process swaps `/foo/link` symlink between `/proc` and dummy directory.
  - This bypassed our anti-`/proc` mount checks.
- `/proc/sys/kernel/core_pattern` allows a container escape.

solutions

- Additional hardening when doing mounts.
  - *Takeaway:* We really should've switched to the new mount API much earlier.
- Switch to `libpathrs` (`pathrs-lite`) procfs API for writes.
  - Also, audited all write paths for misdirectable writes.
  - *Takeaway:* Protecting against everything is quite hard.

runc todos

- Move our mount infrastructure to `fsopen` / `open_tree`.
  - Masked path application will finally be atomic.
- Do a deeper audit of all path-based code in runc.
- Switch to `libpathrs` for runc builds.
  - `pathrs-lite` can use `libpathrs` as a backend.

kernel todos

- Still some nice-to-have extensions for `openat2`.
  - `RESOLVE_NO_DOTDOT`?
  - `rootfd` / `cwdfd` split?
- Blocking all magic-link overmounts would help a lot.
  - Most have been blocked since 6.12.

general takeaways

- Use `openat2` or `libpathrs` (for Go, maybe `pathrs-lite`).
  - Switch to a more file-descriptor based design.
  - Doubly so if you need to touch `/proc`.
- *Every* pathname syscall is potentially dangerous.

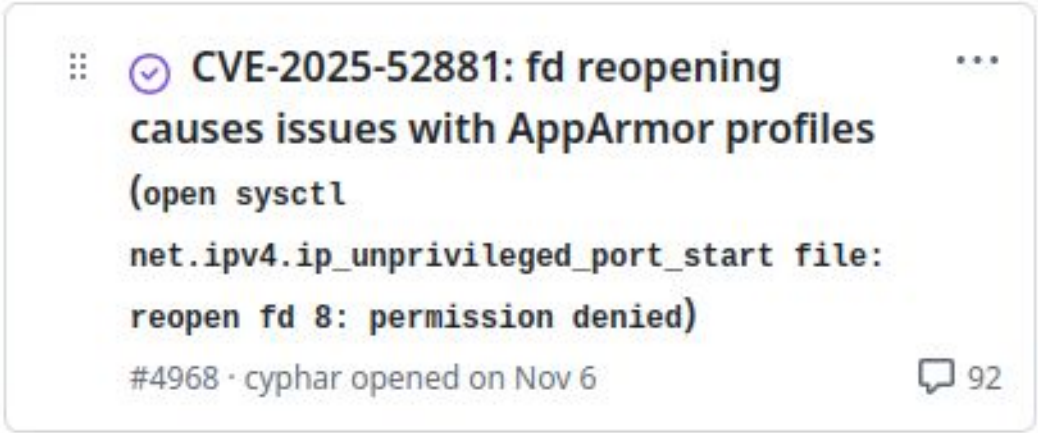
AMUTABLE

questions?

(rants, pitchforks…?)

fun issues

apparmor, d_path, pain



apparmor, d_path, pain

- Nested containers (under LXC).
- AppArmor returns `-EACCES` for `/proc/sys/net/...` write.
- Triggered by switch to `fsopen("proc")`.

apparmor, d_path, pain

```
deny /proc/sys/[^fknu]*{,/**} wklx,
deny /proc/sys/n[^e]*{,/**} wklx,
deny /proc/sys/ne[^t]*{,/**} wklx,
deny /proc/sys/net?*{,/**} wklx,
...
```

apparmor, d_path, pain

```
deny /proc/sys/[^fknu]*{,/**} wklx,
deny /proc/sys/n[^e]*{,/**} wklx,
deny /proc/sys/ne[^t]*{,/**} wklx,
deny /proc/sys/net?*{,/**} wklx,
...

deny /sys/[^fdck]*{,/**} wklx,
deny /sys/k[^e]*{,/**} wklx,
...
```

apparmor, d_path, pain

```
deny /proc/sys/[^fknu]*{,/**} wklx,
deny /proc/sys/n[^e]*{,/**} wklx,
deny /proc/sys/ne[^t]*{,/**} wklx,
deny /proc/sys/net?*{,/**} wklx,
...

deny /sys/[^fdck]*{,/**} wklx,
deny /sys/k[^e]*{,/**} wklx,
...
```

runc AppArmor

runc

```
int procfd = fsopen("proc");
```

AppArmor

```
aa_path_name() ⇒ "/"
```

—

runc

```
        int procfd = fsopen("proc");

int fd = openat(procfd,
                "sys/foo/bar", ...);
```

AppArmor

```
aa_path_name() ⇒ "/"

aa_path_name() ⇒ "/sys/foo/bar"
```

runc

```
        int procfd = fsopen("proc");

int fd = openat(procfd,
                "sys/foo/bar", ...);
```

AppArmor

```
aa_path_name() ⇒ "/"

aa_path_name() ⇒ "/sys/foo/bar"
```

apparmor, d_path, pain

```
int fsfd = fsopen("tmpfs", 0);
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, 0, 0);
int tmpfd = fsmount(fsfd, 0, MOUNT_ATTR_NOEXEC|...);

mkdirat(tmpfd, "sys", 0755);
openat(tmpfd, "sys/foo", O_CREAT, 0755); /* EACCES */
```

apparmor, d_path, pain

dangling symlinks

- Previously we would expand dangling symlinks.
  - `/foo/bar → /some/non-existant/path`
  - `/foo/bar/baz → /some/non-existant/path/baz`
- Some users depend on this behaviour…
  - Emulating it with file descriptors is quite hard.
- *Solution:* Expand the path and then use that as the user path.

AMUTABLE

questions?

(rants, pitchforks…?)