libpathrs

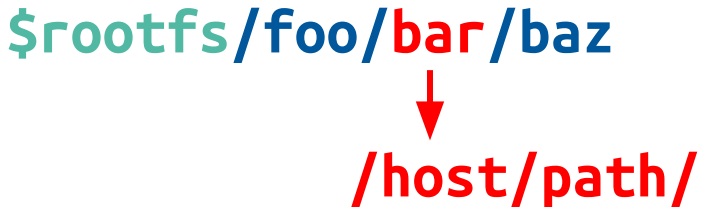
securing path operations for system tools

CC-BY-SA 4.0

Aleksa Sarai
github.com/cyphar

`$rootfs/foo/bar/baz`

- The vast majority of system tools need to interact with unsafe paths.



- Any component can be renamed or swapped to a symlink.

solutions

openat2

(since Linux 5.6)

- `chroot`-like `IN_ROOT` "just works" for most cases.
- `NO_SYMLINKS`, `BENEATH` for everything else.
- `NO_MAGICLINKS`, `NO_XDEV` are particularly useful.

openat(O_PATH)

(since Linux 2.6.39-ish)

- Manually do lookup with `O_PATH` handles, emulating what `openat2` does.
- `..` and `/` components are usually verified through `/proc/self/fd`.

prior art

- Most tools try to restrict path lookups in various ways.
  - LXC and Incus use `openat2` with an `O_PATH` fallback.
  - Docker and containerd use `chroot` for *some* things.
  - runc and umoci use filepath-securejoin for *most* things.
  - systemd has a custom `O_PATH` resolver (`chaseat`).
  - Golang are working on their own version.
- *If you're lucky,* they'll also verify the path using `procfs`.

resolution alone is not enough

- `*at(2)` syscalls make *most* things *mostly* painless, however:
  - Symlink following behaviour is inconsistent.
  - Some care is needed for syscalls without `AT_EMPTY_PATH`.
- Some operations are a bit more complicated to implement (`mkdir -p`, `rm -r`, etc).

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.
- Currently intended for `RESOLVE_IN_ROOT` users, but `RESOLVE_BENEATH` could easily be added.
- Newer kernel features are automatically used if available.

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)?;

// 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.
```

libpathrs (C)

```c
int root = pathrs_root_open("/path/to/root");
if (root < 0) {
	liberr = root;
	goto err;
}
int handle = pathrs_resolve(root, "/etc/passwd");
if (handle < 0) {
	liberr = handle;
	goto err;
}
int fd = pathrs_reopen(handle, O_RDONLY);
if (fd < 0) {
	liberr = fd;
	goto err;
}
```

```c
err:
if (liberr < 0) {
	pathrs_error_t *error =
		pathrs_errorinfo(liberr);
	fprintf(stderr,
		"Uh-oh: %s (errno=%d)\n",
		error->description,
		error->saved_errno);
	pathrs_errorinfo_free(error);
}

close(root);
close(handle);
/* ... do something with fd ... */
```

“**re**open”?

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

// Get a reusable (O_PATH) ptmx handle.
let ptmx = root
       .resolve("/dev/pts/ptmx")?;

// Create several new console instances.
// They are all independent instances.
let console1 = ptmx
       .reopen(OpenFlags::O_RDWR)?;
let console2 = ptmx
       .reopen(OpenFlags::O_RDWR)?;
let console3 = ptmx
       .reopen(OpenFlags::O_RDWR)?;
```

- Sometimes you need to re-open the same file multiple times.
- This is not just dup! It’s a proper *race-free* open.
- Implementing lookups entirely with O_PATH and re-opening is simpler…

`/proc/self/exe`
`/proc/self/fd/$fd`
`/proc/self/attr/exec`

- `procfs` is the only way to do some operations.
- Unlike regular filesystems, we care about which *specific* path is being opened.
- Almost all programs assume that `/proc` is implicitly trusted.

`procfs` attacks

- `CVE-2019-19921` and `CVE-2019-16884` both worked by creating a fake `procfs` so that AppArmor labels weren't applied.
- Bind-mounts are undetectable without `RESOLVE_NO_XDEV`.
- Over-mounted magic-links are even more undetectable.
    - Patches to block this in newer kernels.

`procfs` api

- Detecting attackers is the primary goal, followed by resiliency.
- Private procfs instance with `fsopen` and `open_tree` if possible.
  - Can’t be used for unprivileged programs…
- `openat2` or very restrictive `O_PATH` resolver for lookups.
  - *(This resolver doesn’t do reopens!)*
- Used internally by libpathrs to implement the main API.

procfs api (rust)

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

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

// Create your own private handle.
let handle = ProcfsHandle::new()?;
```

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

// Shorthand for readlinkat(fd, "").
let fd_path = GLOBAL_PROCFS_HANDLE.readlink(
        ProcfsBase::ProcThreadSelf,
        format!("fd/{}", file.as_raw_fd()),
        OpenFlags::O_RDONLY,
)?;
```

procfs api (C)

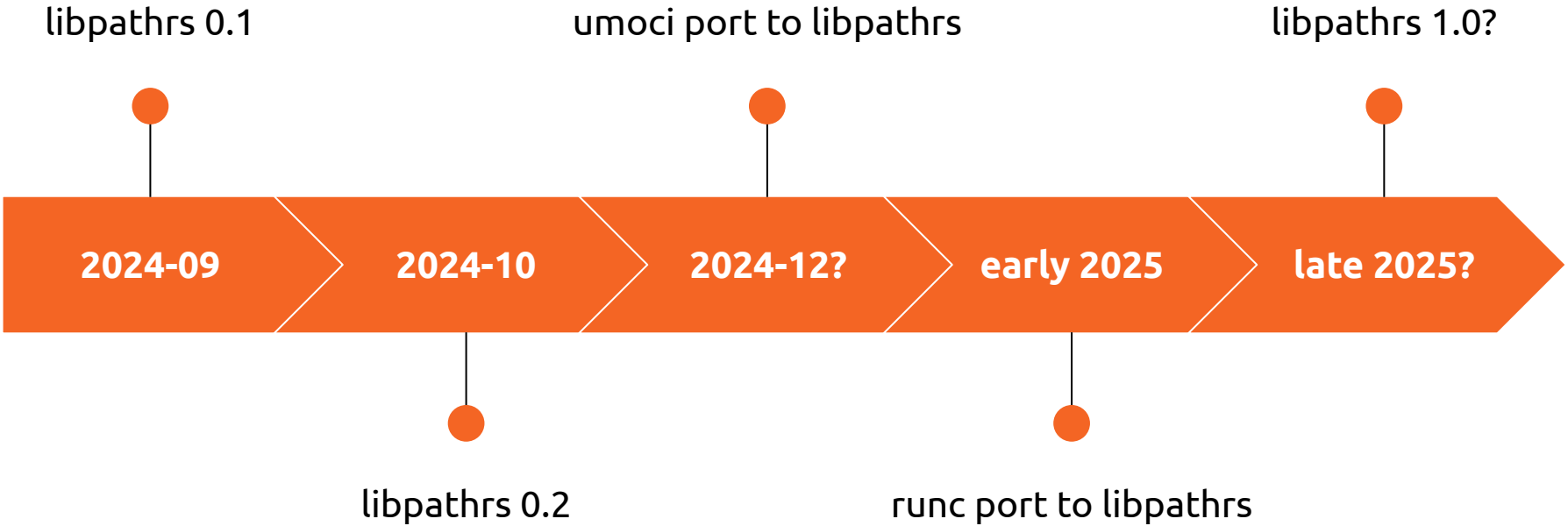
```c
int write_apparmor_label(const char *label)
{
    /* Open *regular* file. */
    int fd = pathrs_proc_open(
        PATHRS_PROC_THREAD_SELF,
        "attr/apparmor/exec",
        O_WRONLY|O_NOFOLLOW
    );
    if (fd < 0) {
        pathrs_error_t *e = pathrs_errorinfo(fd);
        /* ... print the error ... */
        pathrs_errorinfo_free(e);
        return -1;
    }
    int err = write(fd, label, strlen(label));
    close(fd);
    return err;
}
```

```c
int get_self_exe(void)
{
    /* Follows the magic-link! */
    int fd = pathrs_proc_open(PATHRS_PROC_SELF,
                              "exe", O_PATH);
    if (fd < 0) {
        pathrs_error_t *e = pathrs_errorinfo(fd);
        /* ... print the error ... */
        pathrs_errorinfo_free(e);
        return -1;
    }
    return fd;
}
```

`procfs` limitations

- For non-magic-links, `openat2` (Linux 5.6) is sufficient.
  - (`openat2` might be blocked due to seccomp limitations.)
  - (`O_PATH` resolver needs Linux 5.8 to be safe.)
- For magic-links, we need:
  - `statx` mount ID support (Linux 5.8) for bind-mounts.
  - `fsopen` or `open_tree` (Linux 5.1) for race safety.

next steps


libpathrs 0.1
umoci port to libpathrs
libpathrs 1.0?
2024-09
2024-10
2024-12?
early 2025
late 2025?
libpathrs 0.2
runc port to libpathrs

remaining work

- Minor C and Rust API work.
  - Other filesystem APIs we should provide wrappers for?
- Add support for `NO_XDEV` if users need it?
  - We can use `name_to_handle_at` for pre-`openat2` kernels.
- *Your idea goes here!*

questions?

(rants, pitchforks…?)

CC-BY-SA 4.0

kernel hardening – magic-links

- Magic-links can be used to break out of containers.
  - Userspace needs to be careful about leaks.
  - Container runtimes need to use `PR_SET_DUMPABLE`!
- `CVE-2019-5736`: Overwrite host binary with `/proc/self/exe`.
  - Solution: restrict reopening with extra permissions.
  - (Also allow users to specify fd restrictions with `openat2`.)
- Restrict mounts on top of all magic-links.

kernel work – openat2

- Some more things we might want to add:
  - `RESOLVE_NO_BLOCK` (`NO_REMOTE`?) to avoid DoSes.
  - Restrict types of files we want to open (another DoS).
  - `RESOLVE_NO_DOTDOT` for extreme lookup restrictions.
- What about atomic `mknod` combined with `open`? (`O_MKNOD`?)
  - If it could take `RESOLVE_*` flags that would be even nicer!