Securing Container Runtimes

(and why path resolution keeps me up at night.)

Aleksa Sarai

Senior Software Engineer

`@lordcyphar`

`<cyphar@cyphar.com>`

PLEASE USE USER NAMESPACES

(Folks who did were *not vulnerable to most of these bugs*…)

container_runtime.pdf

- Download and extract image archive into rootfs.
- Fork (and re-exec) to create proto-pid1.
  - Child will exec() pid1’s code at end.
  - Parent assists during setup.

container_runtime (2).pdf

- Parent’s job:
  - Move child process into correct cgroup.
  - Set up and container’s veth or other network devices (if configured).
  - Signal child to start.

container_runtime (3).pdf

- Child's job:
  - Create or join namespaces (mount, pid, net, ipc, …, and *hopefully* user).
  - Configure mountpoints for container process, pivot_root(new root).
  - Configure seccomp filters, LSM labels, no_new_privs, process credentials, …
  - Wait for parent signal, then execve(user's code).

container_runtime (4).pdf

- Other jobs:
  - Spawn a new process inside the container **while it's running**.
    - Rather that creating and configuring namespaces, join existing ones.
  - Modify existing container state (cgroup limits, network devices, ...).
  - Many more uninteresting things.

CVE-2014-????

- `docker cp` didn't do *any* path sanitisation.
  - Oops.
  - `docker cp container:<symlink to /etc/shadow> w00t_w00t`
- Lesson learned:
  - Maybe we should sanitise paths.

CVE-2015-{3627,3629,3630,3631}

- Mostly related to bad configuration or permitting bad configurations.
  - Oops.
- Lessons learned:
  - Don’t make those kinds of mistakes(?).
  - `VOLUME` was probably a mistake.
  - Containers are hard.

CVE-2016-9962

- We kept open a file descriptor to the root filesystem while joining the container.
  - Container could access host through `/proc/$pid/fd/$n`.
- Lessons learned:
  - `procfs` is a bit scary.
  - Make ourselves “non-dumpable” to block container process trickery.
    - Turns out there were some kernel bugs here too…

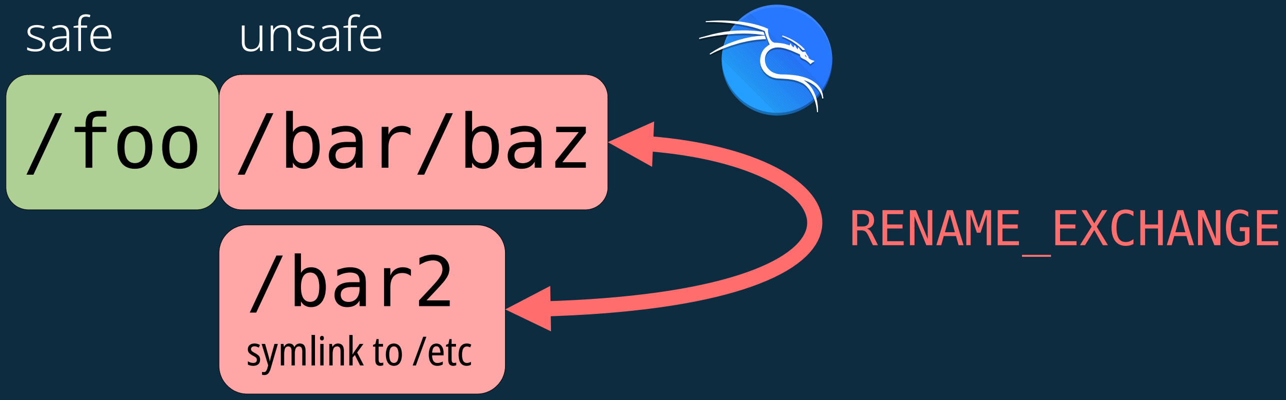
CVE-2017-????

- There were a few Kubernetes CVEs related to symlinks.
  - Basically, they weren’t properly handling symlinks at all.

CVE-2018-15664

- Path sanitisation isn’t enough if the attacker can change the paths underneath you.
  - `RENAME_EXCHANGE` can be used to swap (“symlink-exchange”) a component.

CVE-2018-15664



safe

unsafe

/foo

/bar/baz

/bar2

symlink to /etc

RENAME_EXCHANGE

/foo/bar/passwd

CVE-2018-15664

- Lessons learned:
  - Plain path sanitisation (as used to fix the 2014 bugs) is insufficient.
  - Solving this properly is non-trivial (see how LXD has done it).
  - In Docker, this was solved by fixing some bugs in the chrootarchive implementation.
    - But the underlying bug still remains.

CVE-2019-5736

- We could be tricked into re-executing ourselves, pinning `/proc/self/exe`.
  - This clears the “non-dumpable” bit, but maintains `/proc/self/exe`.
  - `open(“/proc/self/exe”, O_RDONLY)` then re-open it after the process dies.
- Lessons learned:
  - `procfs` is fairly terrifying.
  - Make a copy of the runc binary each time, so overwriting does nothing.
  - Maybe we should do some kernel work to block re-opens like this…

CVE-2019-16884

- With `VOLUME`, you can configure mounts that shadow `/proc`.
  - This means the container runtime can be tricked into not setting security labels.
- Lessons learned:
  - `VOLUME` was still a mistake.
  - `procfs` might be fake while being horrifying.

CVE-2019-19921

- With custom images, you can use the symlink-exchange trick to mess with `/proc`.
  - This means the container runtime can be tricked into not setting security labels.
  - `/proc/self/sched` can be used as a no-op writeable `procfs` target.
    - Ditto for `/proc/self/environ`.
- Lessons learned:
  - `procfs` is like staring into a bottomless abyss, filled with pain and CVEs.
  - `VOLUME` were a mistake, as were several of my life decisions at this point.

(almost) CVE-2020-????

- Our devices cgroup handling was … fairly questionable.
  - We would temporarily allow all device access during `runc update`.
    - Luckily this was never in a released version of runc.
    - And it required using `--systemd-cgroup`.
  - Our default devices policy was *allow-all*.
    - Luckily all users (including Docker) already had deny-by-default policies.
- Lessons learned:
  - How is it possible for us to have legacy code in such a young codebase!?

what is the common theme?

- Don’t be tricked into misconfiguring containers.
- Filesystem operations are really easy to screw up.
- **`procfs`**

let’s make filesystem operations safe!

the problem

`/foo/bar/baz`

- `baz` might be a symlink. *(Just use* ***`O_NOFOLLOW`****!)*
- `bar` might be a symlink. *(Uhhh… sanitise it in userspace?)*
- `foo` might be attacker-controlled and thus `bar` can become a symlink. *(Dammit.)*
- This *is* a solveable problem in userspace, but almost nobody does it correctly.

the (old) solution

`/foo/bar/baz`

- For each component:
  - Open the next component (with `O_NOFOLLOW`) relative to the current one.
  - Handle symlinks in userspace by keeping track of the “text” path.
  - Do some double-checking along the way through *`/proc`* and hope it works.
- Very hard to get right, and it looks like nobody is actually doing it.

the new solution

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


struct open_how {
  u64 flags;          // openat(2) flags
  u64 mode;           // openat(2) mode
  u64 resolve;        // RESOLVE_* flags
  // future fields go here
};
```

openat2

```
#define RESOLVE_NO_SYMLINKS   … /* Don’t traverse symlinks. */
#define RESOLVE_NO_MAGICLINKS … /* Don’t traverse magiclinks. */
#define RESOLVE_NO_XDEV       … /* Don’t cross mounts. */
#define RESOLVE_IN_ROOT       … /* Resolve within a root. */
```

so, are we done?

- Not by a long shot.
- It’s hard to get this stuff right, and even with `openat2`:
  - Programs on old kernels still need to be hardened.
  - Users need to be **exceptionally** careful when doing other VFS operations.
  - Programs need to be restructured to use file descriptors everywhere.

a library to make path resolution safe.

lib **path r** **s**

libpathrs

libpathrs

(a **lib**rary to make **path r**esolution **s**afe.)

libpathrs

(a **lib**rary to make **path r**esolution **s**afe.)

(it’s also written in rust.)

introducing libpathrs!

- Rust library (with C bindings, usable from almost any language).
- Emulates `openat2`'s `RESOLVE_IN_ROOT` on older kernels.
- Implements helpers that match most VFS syscalls (which are correctly written).
- Includes some additional hardening (related to p̭r̝͈o͔̠̹ç̩ș͔͔).

usage

```rust
// Get a root handle for resolution.
let root = Root::open("/path/to/root")?;
// Resolve the path.
let handle = root.resolve("/etc/passwd")?;
// Upgrade the handle to a full std::fs::File.
let file = handle.reopen(libc::O_RDONLY)?;

// Or, in one line:
let file = root.resolve("/etc/passwd")?
               .reopen(libc::O_RDONLY)?;
```

docs.rs/pathrs

usage

```
    root = pathrs_open("/path/to/root");
    error = pathrs_error(PATHRS_ROOT, root);
    if (error)
        goto err;

    handle = pathrs_resolve(root, "/etc/passwd");
    error = pathrs_error(PATHRS_ROOT, root);
    if (error) /* or (!handle) */
        goto err;

    fd = pathrs_reopen(handle, O_RDONLY);
    error = pathrs_error(PATHRS_HANDLE, handle);
    if (error) /* or (fd < 0) */
        goto err;

err:
    if (error)
        fprintf(stderr, "Uh-oh: %s (errno=%d)\n", error->description, error->saved_errno);
    pathrs_free(PATHRS_ROOT, root);
    pathrs_free(PATHRS_HANDLE, handle);
    pathrs_free(PATHRS_ERROR, error);
```

docs.rs/pathrs

demo time.

great!
now we’re all done, right?

let's have a chat about procfs

the other problem

`/proc/self/attr/exec`

- How do I make sure that I’m writing to the real `procfs` file?
  - You can grab a `/proc` handle which is definitely real (the inode is 1).
  - You can check if the target is a procfs file (but you aren’t sure it’s the right one).
  - You can disable all symlink crossings a-la `openat2` (or emulate it).
    - Wait … how on earth do you check for bind-mounts?

yeah, what about bind-mounts?

There is **no way** on Linux to be verify if you’ve crossed a bind-mount (until openat2).

and then there's magic-links

magic-links

```
/proc/self/fd/$n
/proc/self/exe
```

magic-links

```
/proc/self/fd/$n
/proc/self/exe
```

YOU CAN BIND-MOUNT OVER SYMLINKS

incomprehensible rambling

next steps

- Stabilise the base libpathrs C API.
- Start porting programs to libpathrs.
- Continue kernel hardening work (which libpathrs can support opportunisically).
  - Lots of work needed to make procfs safe to use.

links

- `openat2` (in Linux 5.6)
  - `lwn.net/Articles/767547`
  - `lwn.net/Articles/796868`
  - `man 2 openat2`
- libpathrs
  - `github.com/openSUSE/libpathrs`
  - `docs.rs/pathrs`
- `github.com/cyphar/talks`

questions?

magic-link restriction

- Don’t allow a read-only magic-link to be re-opened as read-write.
  - Requires lots of fun semantics with `O_PATH`.
  - Doesn’t break userspace (based on my testing).
  - Needs to cover up a **lot** of different holes.

O_EMPTYPATH?

```
openat(fd, “”, O_EMPTYPATH | O_RDWR);
```

built-in procfs handle?

```
openat(AT_PROCFD, “self/fd/$n”, O_RDWR);


setupfd = fsopen(“procfs”, FSOPEN_CLOEXEC);
procfd = fsmount(setupfd, FSMOUNT_CLOEXEC, 0);
openat(procfd, “self/fd/$n”, O_RDWR);
```

pidfd-based /proc/self ??

```
selffd = pidfd_open(getpid(), 0);

pidfd_get_resource(selffd, PIDFD_EXE,
                   O_RDONLY);         // ???

openat(selffd, “exe”, O_RDONLY);      // ???
```