Operating System Security:

Attacking Container Runtimes

Aleksa Sarai

Senior Software Engineer [SUSE]

<cyphar@cyphar.com>

The Plan

- A brief introduction to containers (20’).
- Filesystems and path resolution attacks (20’).
- Other miscellaneous container runtime attacks (10’).
- Q & A.

Containers?

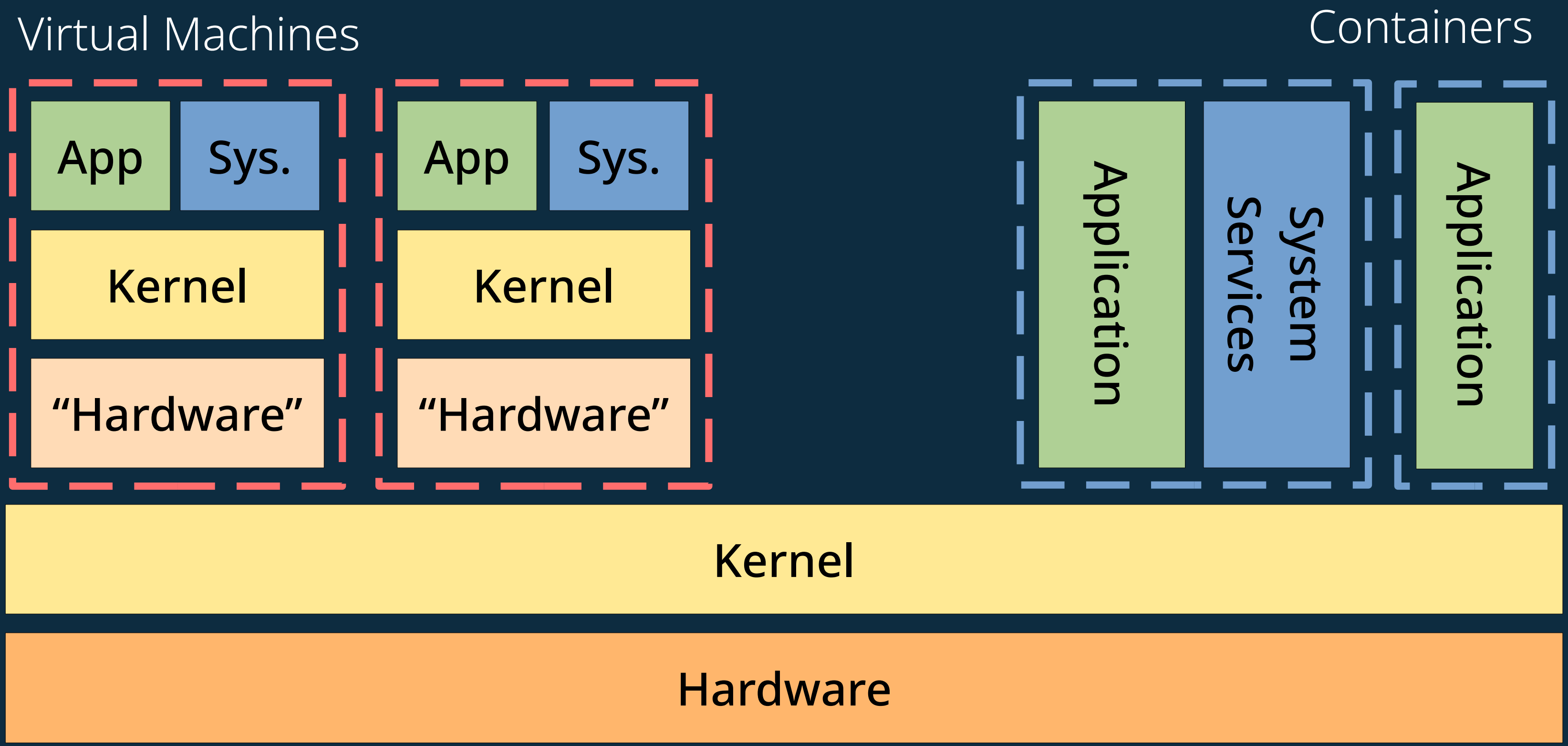
tl;dr

- Containers are a secure[citation needed] way of running multiple *isolated* services on a single machine without incurring the performance and resource overhead of VMs.

  Virtual machines → *hardware virtualisation* → the *hardware* is a **lie**.
  Containers → *operating system virtualisation* → the *OS environment* is a **lie**.

- Widely adopted in the past few years (Docker, Kubernetes, et al).

tl;dr


Virtual Machines
App
Sys.
Kernel
"Hardware"
App
Sys.
Kernel
"Hardware"
Containers
Application
System Services
Application
Kernel
Hardware

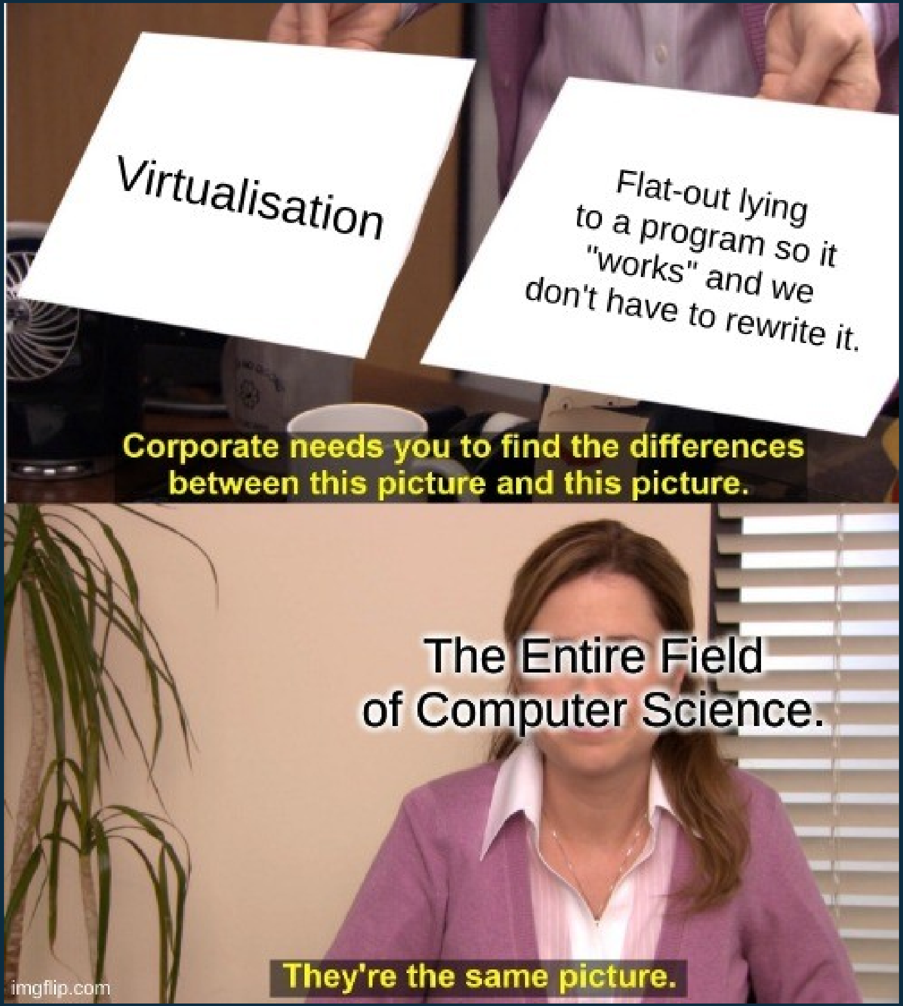
tl;dr in meme form

Virtual Machines

Always has been.
Wait, it's all virtualised?
The "Hardware".
VM GUEST
HYPERVISOR
imgflip.com

Containers

Always has been.
Wait, it's all virtualised?
The "Operating System".
CONTAINER GUEST
OPERATING SYSTEM
imgflip.com


Virtualisation
Flat-out lying to a program so it "works" and we don't have to rewrite it.
Corporate needs you to find the differences between this picture and this picture.
The Entire Field of Computer Science.
They're the same picture.
imgflip.com

A Brief History Lesson.

- 1967: IBM CP-40 is “first real VM”
- 1972: IBM System/370 can host VMs.

  Required additional support for hardware accelerated virtualisation.
- 2000s: Intel VT-x (‘05), AMD-V (‘06).

  First “full virtualisation” support for x86 by adding instructions to “fake” ring-0.

- 1960s: Time-sharing systems.
- 1970s: Multi-user systems (Unix).

  chroot(2) (‘79).
- 2000s: Introduction of containers.

  FreeBSD Jails (‘00), Linux vServer (‘01), Solaris Zones (‘05), OpenVZ (‘05), LXC (‘08), Docker (‘13).

Interesting talk on (part of) this history: https://youtu.be/hgN8pCMLI2U

chroot

- First “container” concept – introduced in Unix v7.
- Run a normal program, but pretend a host directory is the root ("/").
  - (In theory) service cannot access anything outside "/".
  - Operating system virtualises (lies about) the filesystem layout.
- Widely used for running file sharing and web servers for decades.

chroot

[good_program.gif]

Yo, can you pass me a handle to /var/www/htdocs/index.php?

Sure, here you go!

Host Filesystem

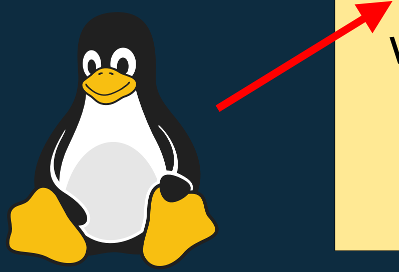
```
/
 etc/
  shadow
 var/run/
  bank-records.csv
 home/pedram/
  test-answers.txt
 var/www/htdocs/
  index.php
  forum.php
```

chroot

[evil_program.gif]

Yo, can you pass me a handle to /home/pedram/test-answers.txt?

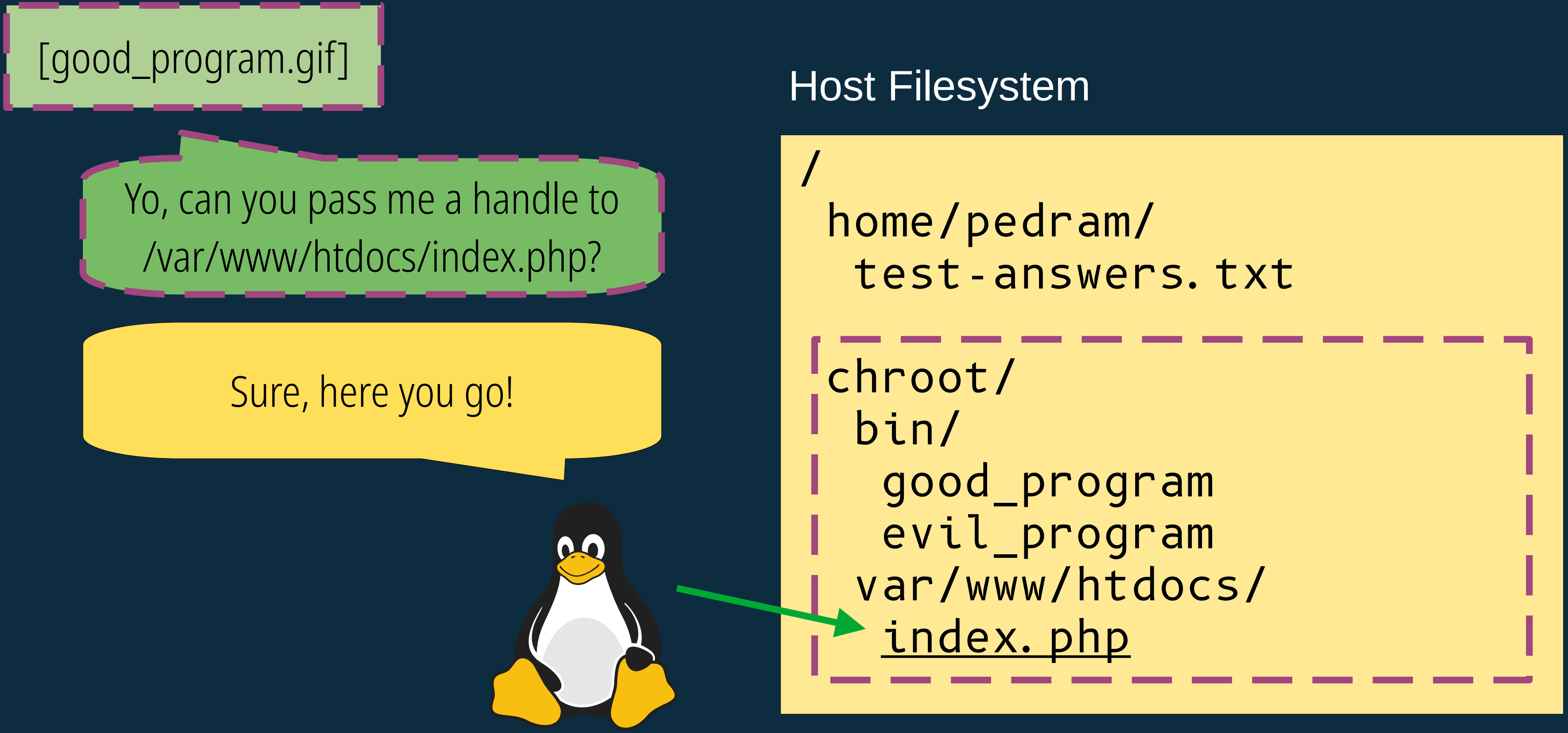
Sure, here you go!



Host Filesystem

```
/
 etc/
  shadow
 var/run/
  bank-records.csv
 home/pedram/
  test-answers.txt
 var/www/htdocs/
  index.php
  forum.php
```

chroot



[good_program.gif]

Yo, can you pass me a handle to /var/www/htdocs/index.php?

Sure, here you go!

Host Filesystem

```
/
 home/pedram/
  test-answers.txt

 chroot/
  bin/
   good_program
   evil_program
  var/www/htdocs/
   index.php
```

chroot

[evil_program.gif]

Yo, can you pass me a handle to /home/pedram/test-answers.txt?

No idea what you're on about.

Host Filesystem

```
/
 home/pedram/
  test-answers.txt

 chroot/
  bin/
   good_program
   evil_program
  var/www/htdocs/
   index.php
```

chroot

- Sounds good, isn’t that enough?

chroot

[evil_program.gif]

Yo, can I listen to packets on 0.0.0.0:443?

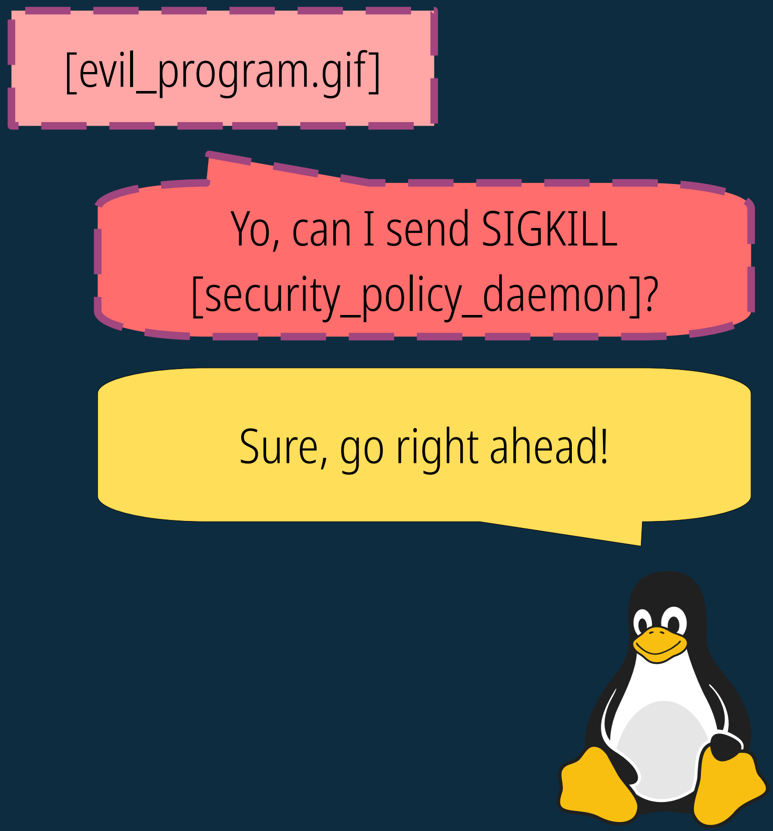
Sure, go right ahead!

[evil_program.gif]

Yo, can I connect to https://secret-network.local:1337?

Sure, go right ahead!

chroot



[evil_program.gif]

Yo, can I send SIGKILL [security_policy_daemon]?

Sure, go right ahead!

[evil_program.gif]

Yo, can I fork-bomb you?

Sure, go right ahe---*[crash]*

Containers

- Conceptually very similar to **chroot**(2), except now the kernel lies about:

  Whether any other processes are on the system. [PID Namespace]
  What network resources are available. [Network Namespace]
  What privileges the contained program has. [User Namespace]
  … [Time, IPC, UTS, …]

- And resource limits can be applied to these processes (akin to Unix's ulimits).

  No more fork-bombs or memory exhaustion tricks. [cgroups]

Container Runtimes

- Unlike other systems (FreeBSD Jails, Solaris Zones, ...) Linux Containers are a Rube Goldberg machine of security features and isolation primitives.
  - The operating system provides individual knobs to configure them, but the final configuration is entirely up to userspace.
- Container runtimes (runc, LXC, ...) perform this and similar management tasks.
  - Tools like Docker are built on top of lower-level tools like runc.
  - They (generally speaking) *run as root* and *interact with container resources*.

Attacking the Container Runtime

Filesystem Race Conditions

Container Filesystem

- Container filesystems are just regular directories – think back to **chroot**(2).

  (Though these days we actually use **pivot_root**(2).)

- Container runtimes have to do lots of operations on the container filesystem as root.

  Obvious examples are **docker cp**, but many other implicit operations too.
  Remember the container has *write access to its filesystem*!
  Confused deputy attacks.

Obvious Attacks

```
$ docker cp container:../../../etc/shadow shadow
$ docker cp container:symlink_to_etc_shad shadow
```

Filesystem Races

- We added some safe resolution checks to Docker in 2014, but they had a flaw:

  Time of Check, Time of Use (TOCTOU).
  We sanitised the path before opening it …
  … but there's a race window between sanitisation and opening.

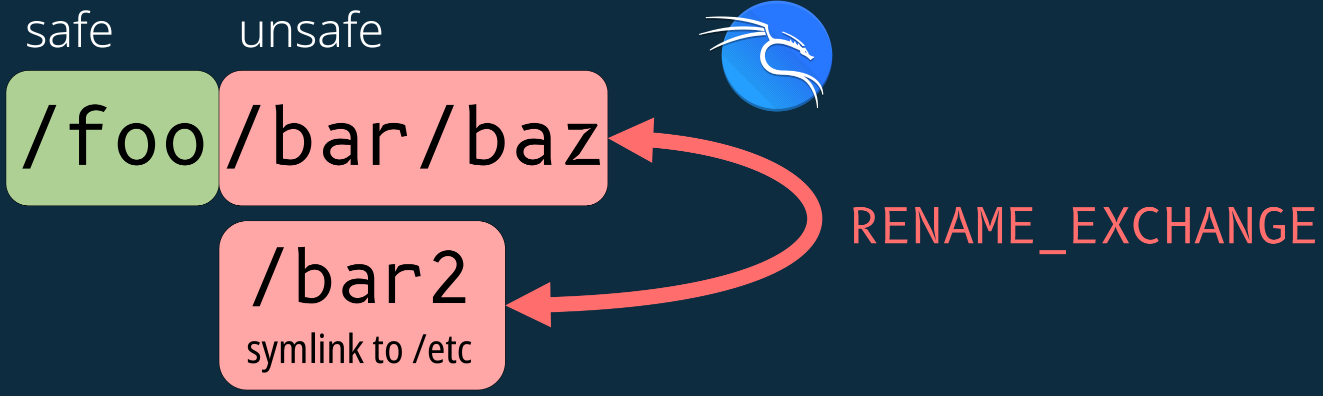
- This is actually a fairly common Unix bug which is quite hard to get right.

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, but almost nobody does it correctly.

CVE-2018-15664



safe

unsafe

/foo/bar/baz

/bar2

symlink to /etc

RENAME_EXCHANGE

```
docker cp
    ctr:/foo/bar/shadow shadow
```

Solution

- Always grab a stable handle (a file handle) to the target path and then check *that*.

  If the path moves, the file handle moves with it.

- Unfortunately we haven’t fully implemented this (it’s quite hard to get right).

Attacking the Container Runtime

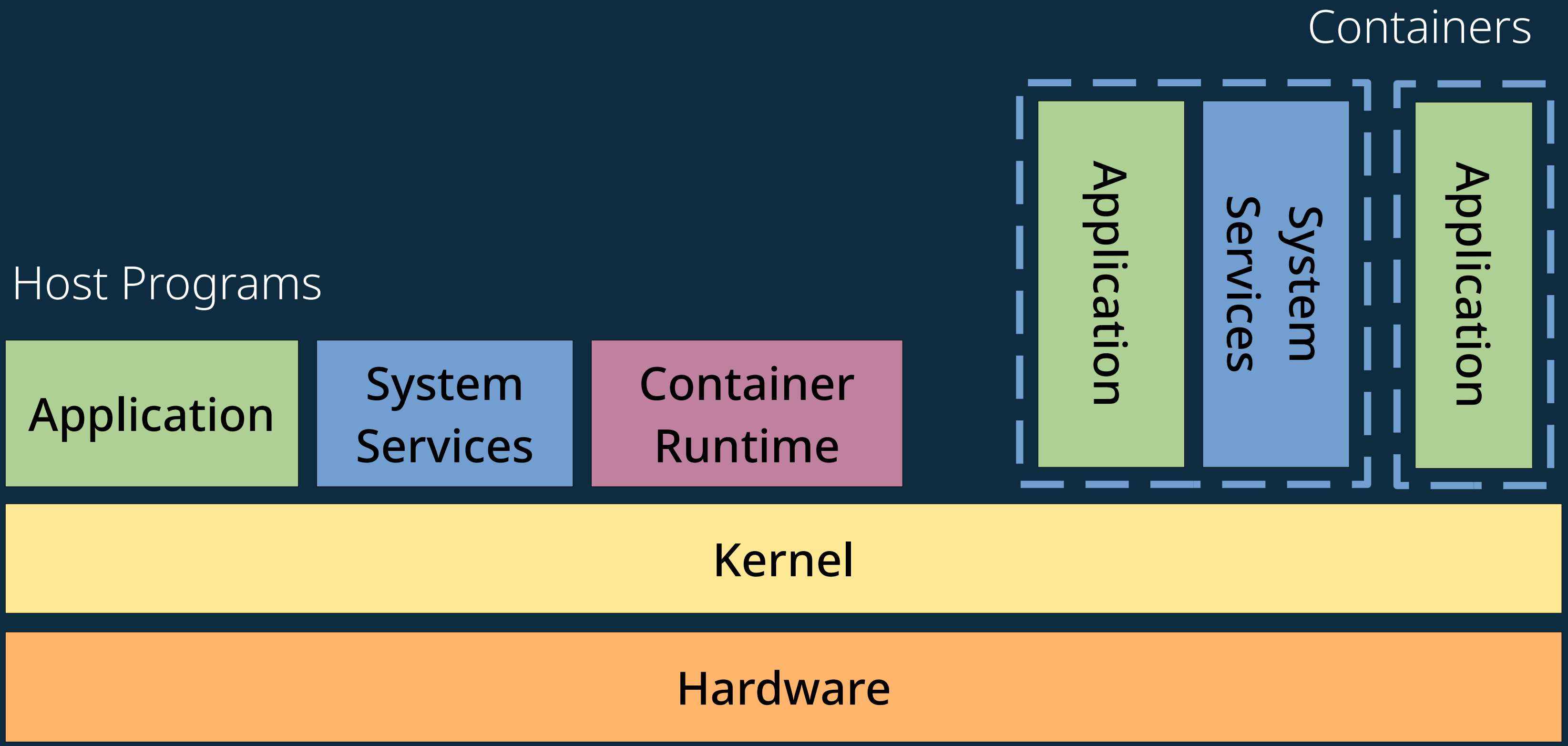
Some Other Fun Issues

procfs

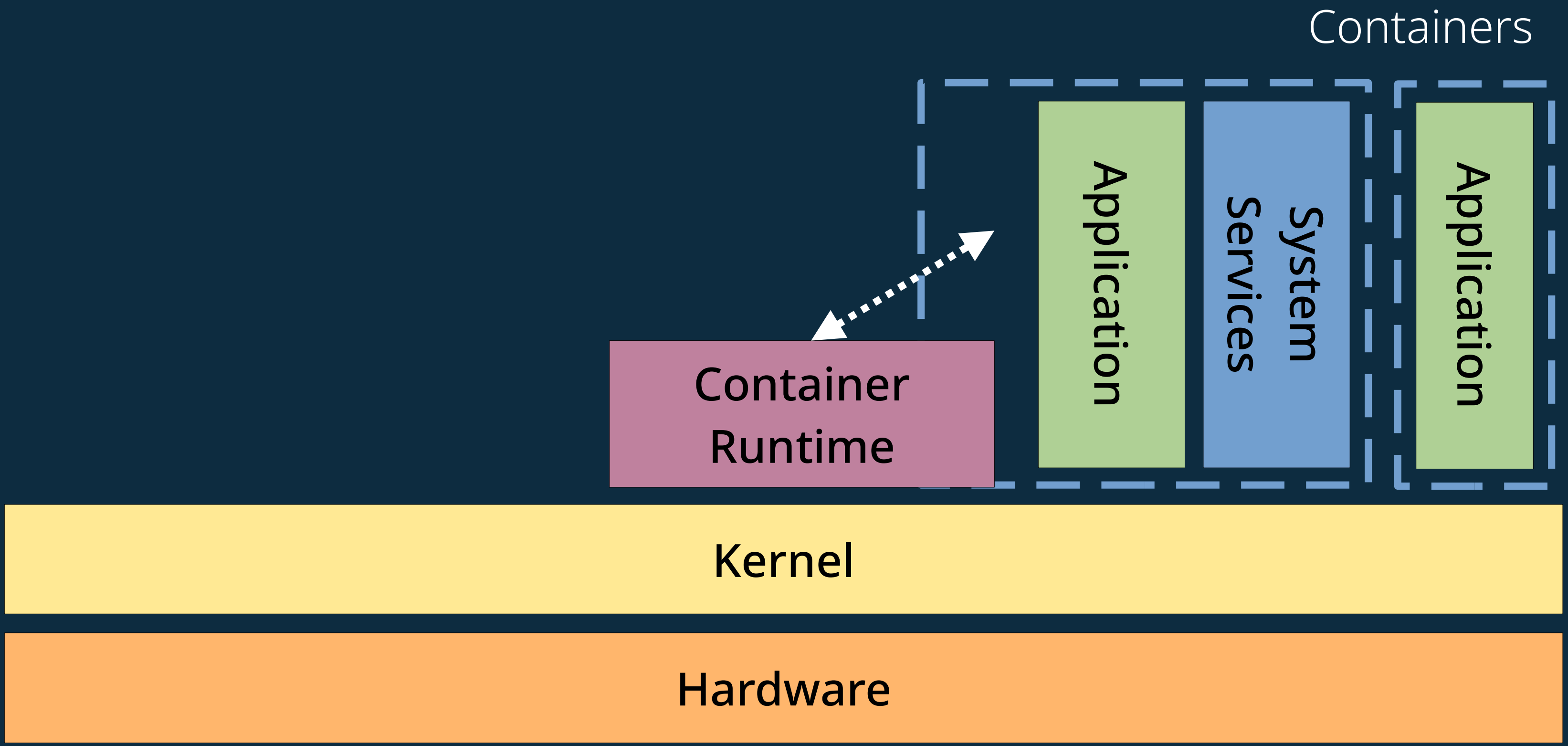
- `/proc` is a very scary and magical Linux filesystem.

  It is an kernel API which has a lot of holes in it.
  Quite a few container breakouts were discovered by (ab)using `procfs`.

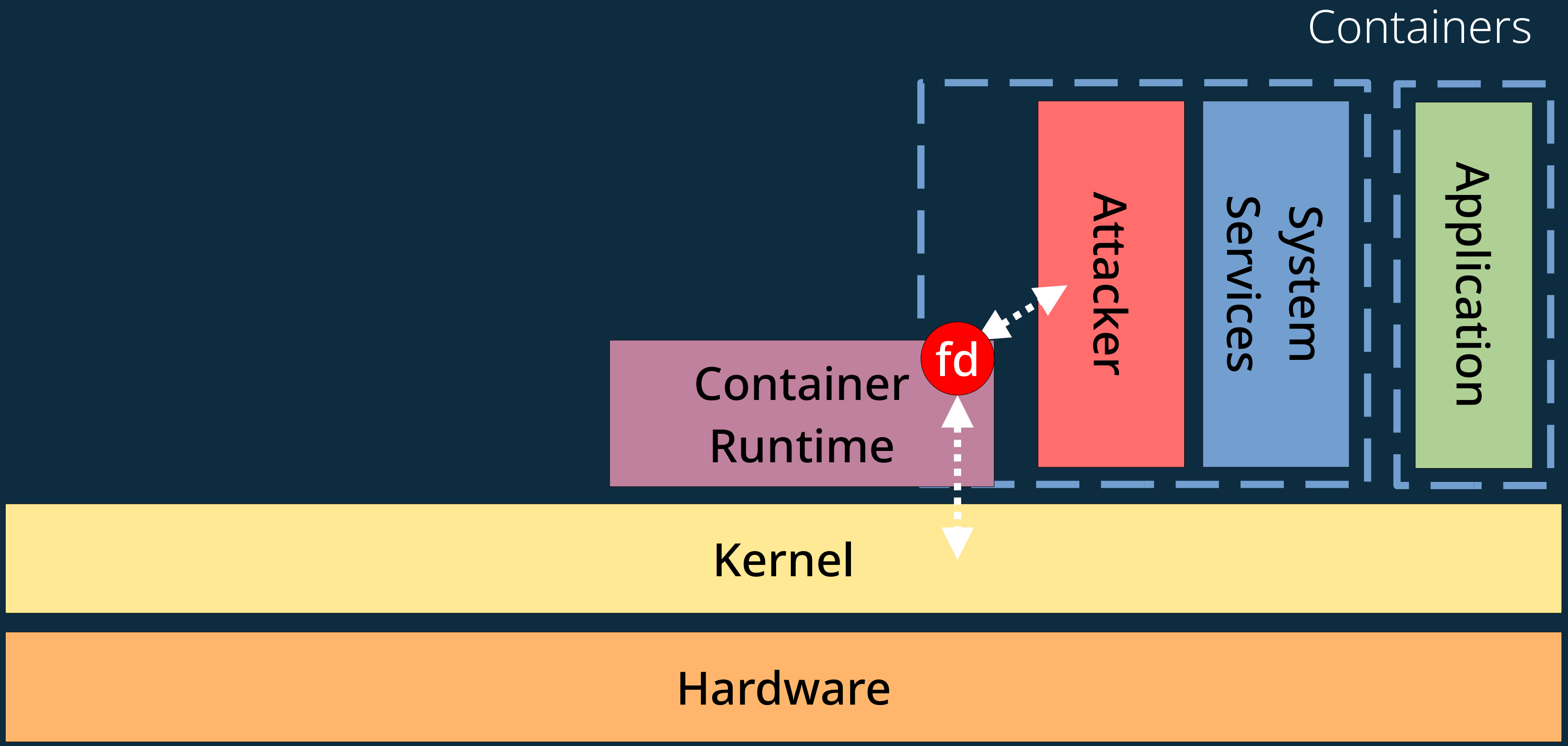
The Kernel Is Shared



The Kernel Is Shared



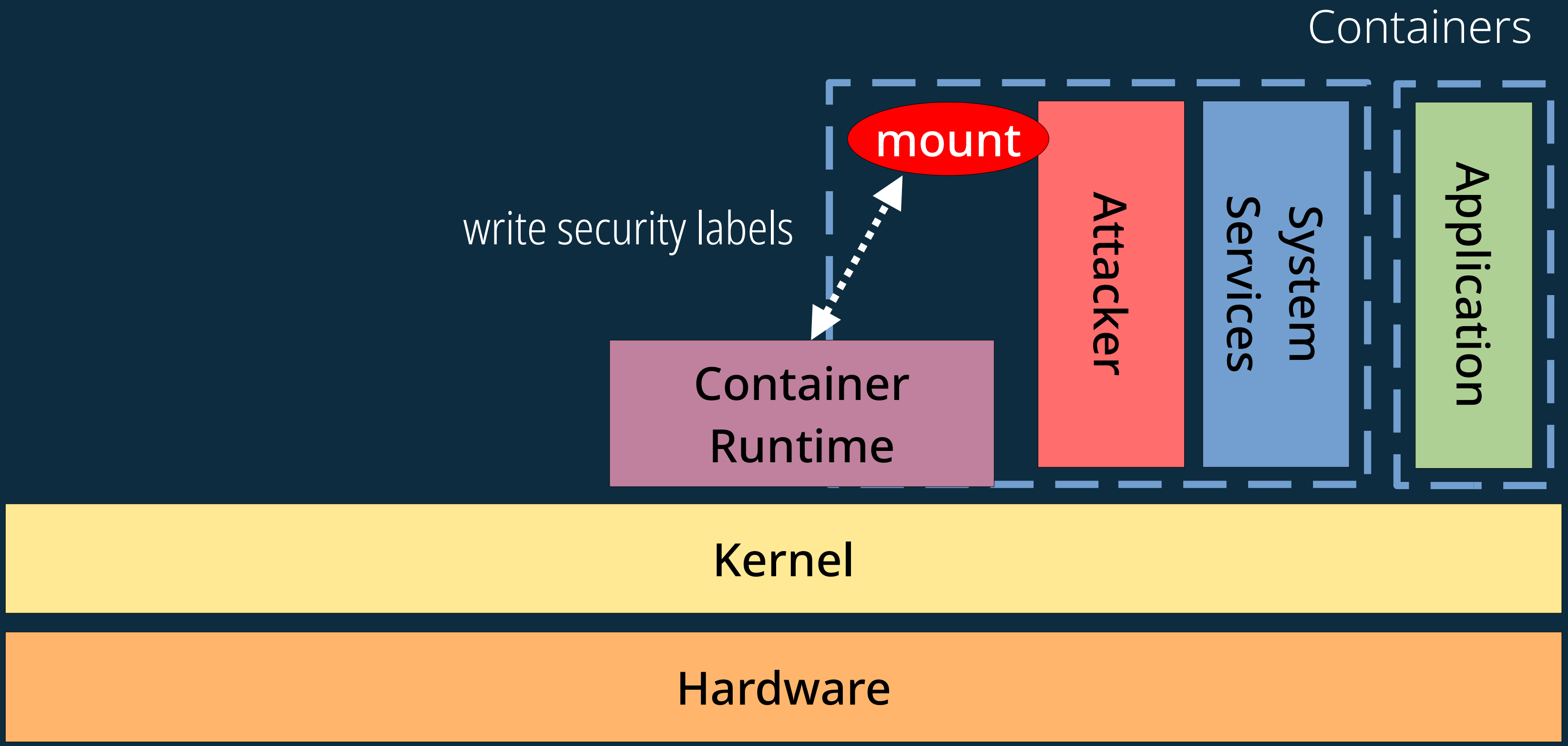
The Kernel Is Shared



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:
  - Make ourselves “non-dumpable” to block container process trickery.
    - Turns out there were some kernel bugs here too…

The Kernel Is Shared



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.

Any Questions?

… or I can discuss some defensive measures.

Slides are available at github.com/cyphar/talks.

let’s make filesystem operations safe!

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 p̭r̘͈o͔̱̹c̩fs͔͔).

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

```
/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);      // ???
```