KubeCon | CloudNativeCon

North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

KubeCon | CloudNativeCon
North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

Building Container Images in Kubernetes: It’s Been a Journey!

Laurent Bernaille & Eric Mountain

@ericmountain
@lbernail

About Datadog

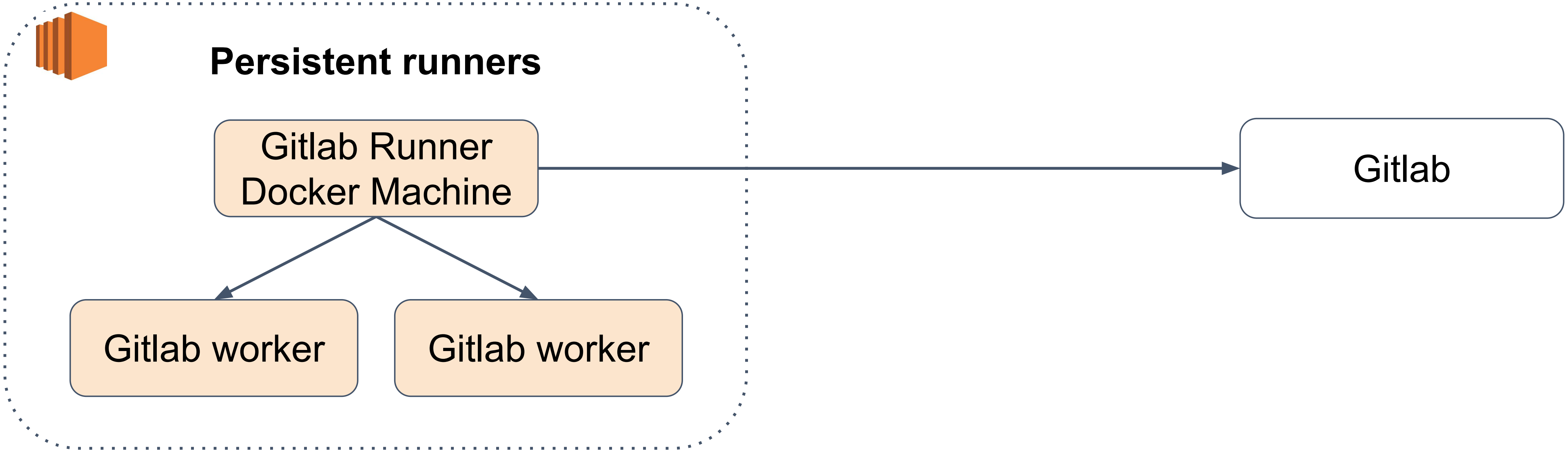
Over 500 integrations
Over 3000 employees
Over 18,500 customers
Millions of hosts reporting
Trillions of data points per day

10000s hosts in our infra
Dozens of k8s clusters
Multi-cloud
Rapid growth

DATADOG

A Quick history of our build system

History: Building our applications



Persistent runners

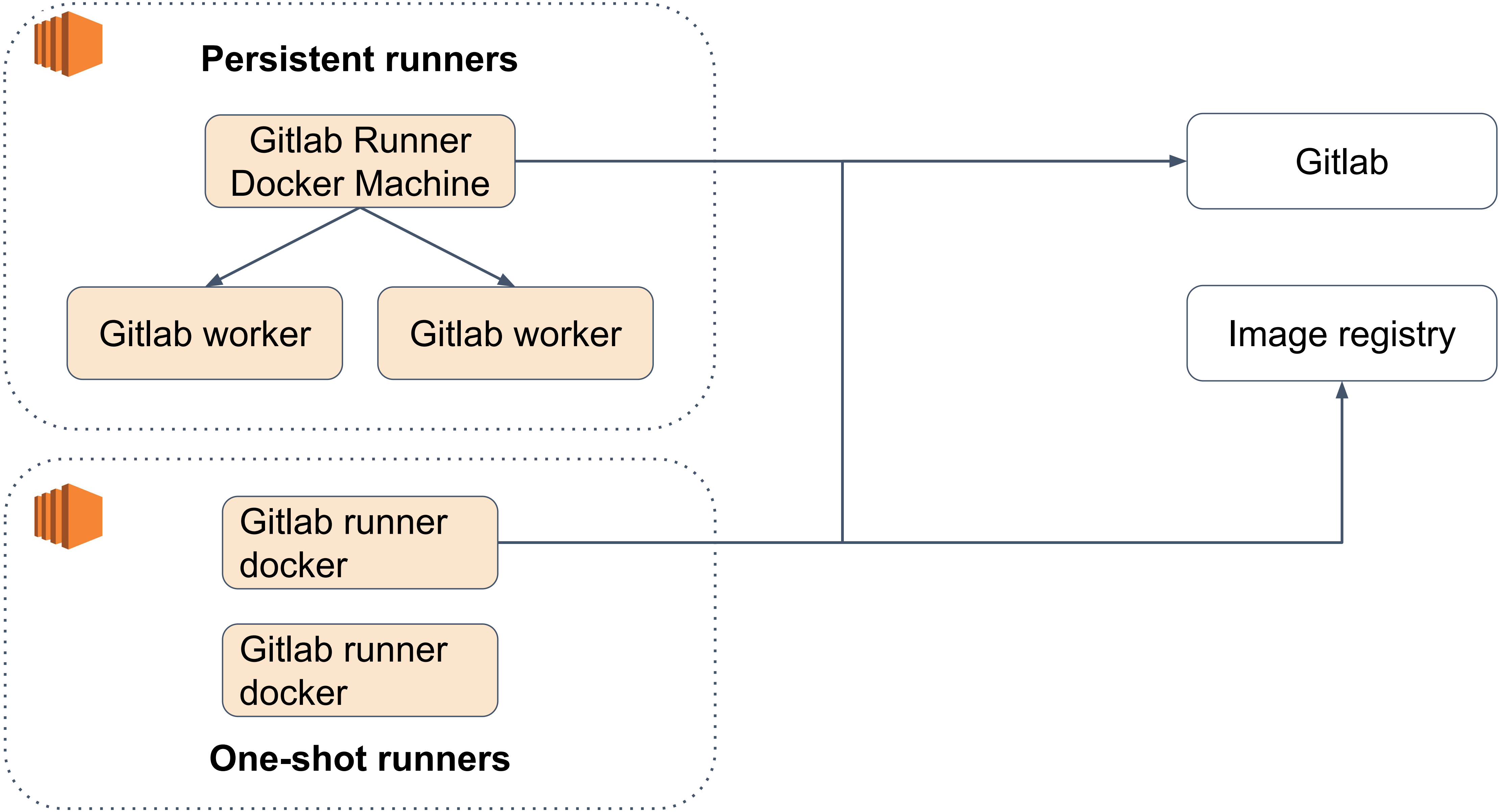
Gitlab Runner
Docker Machine

Gitlab worker

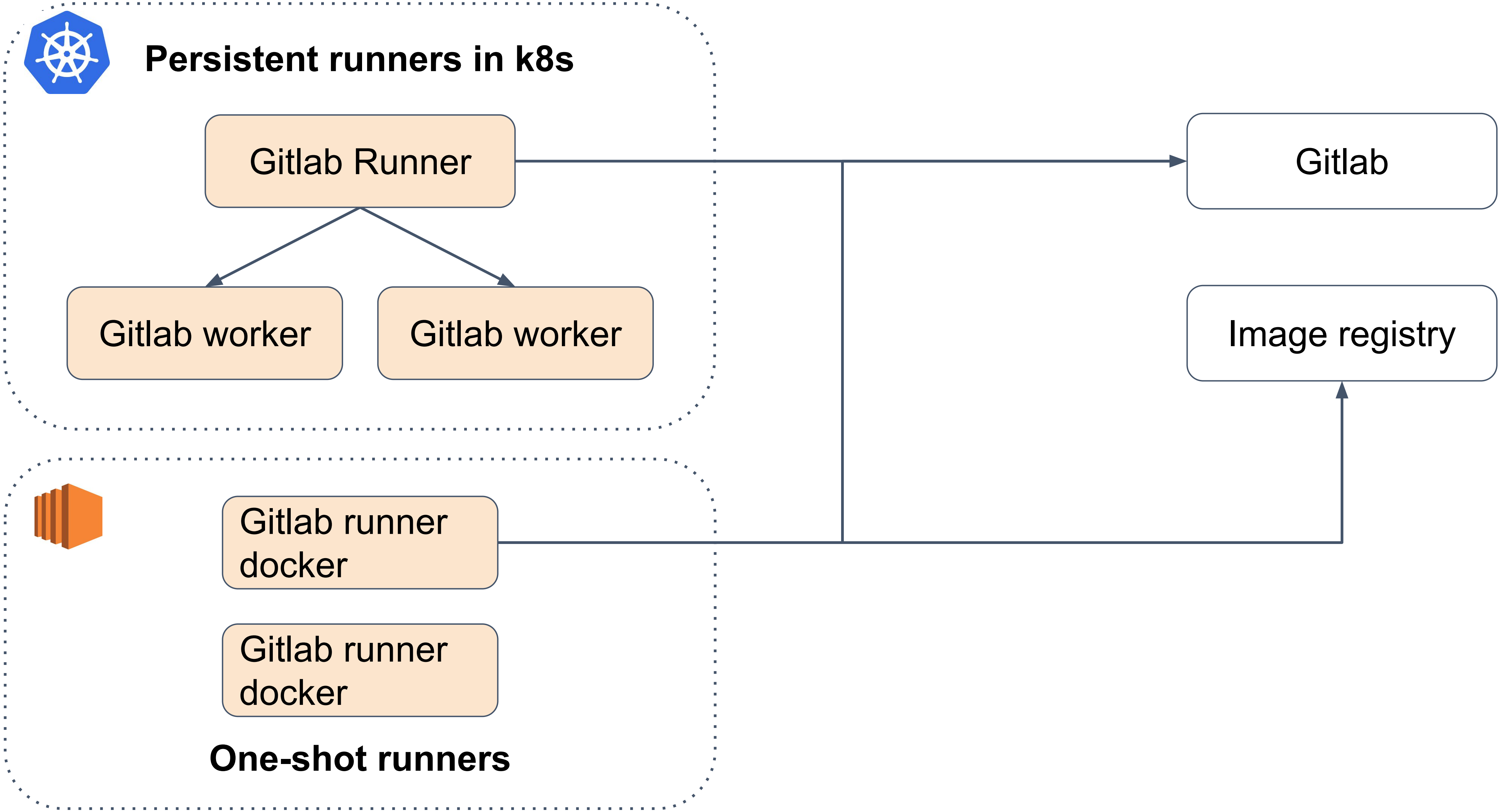
Gitlab worker

Gitlab

History: Building docker images



History: Using kubernetes builders



Persistent runners in k8s

Gitlab Runner

Gitlab worker

Gitlab worker

Gitlab runner docker

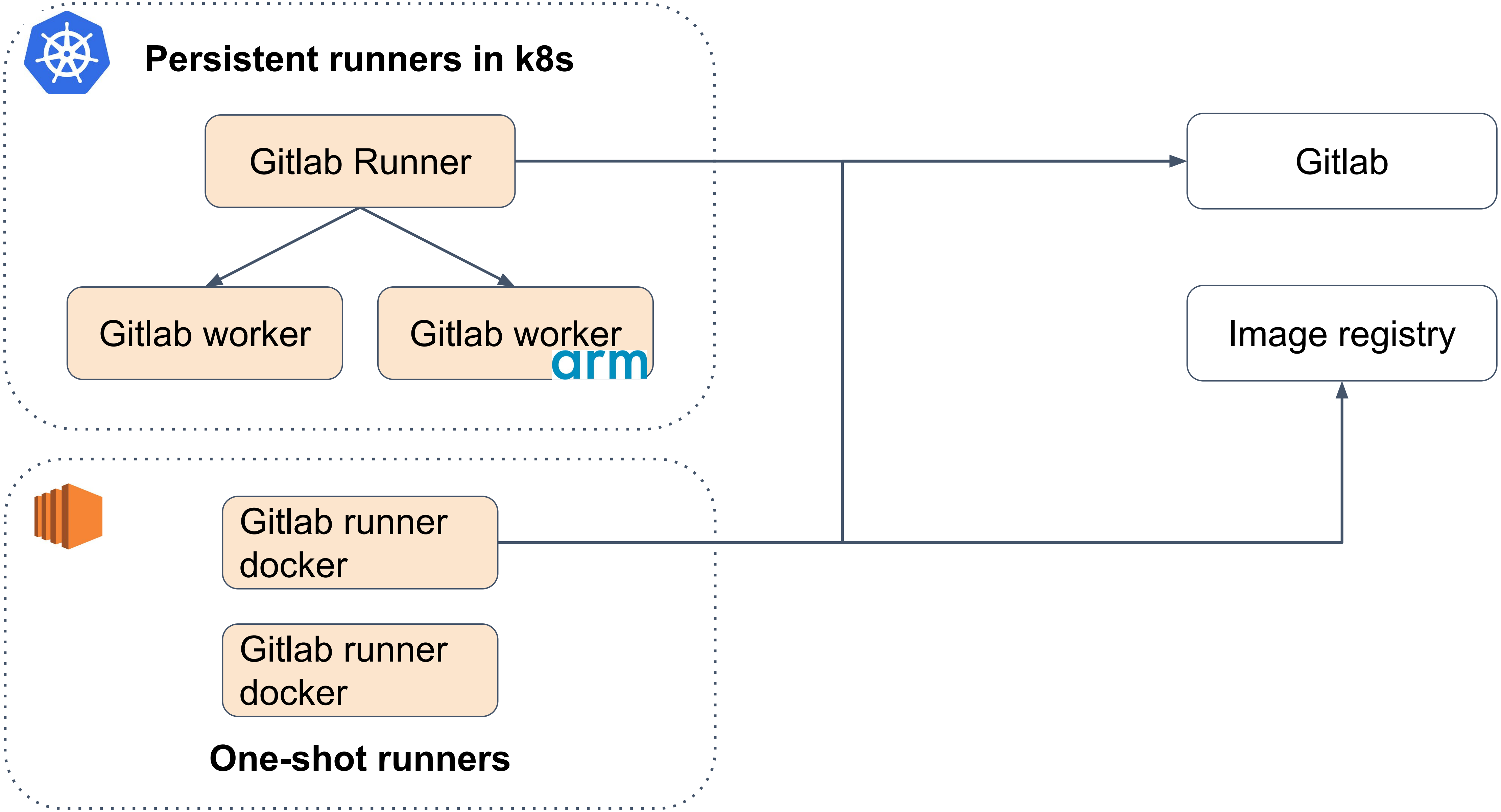
Gitlab runner docker

One-shot runners

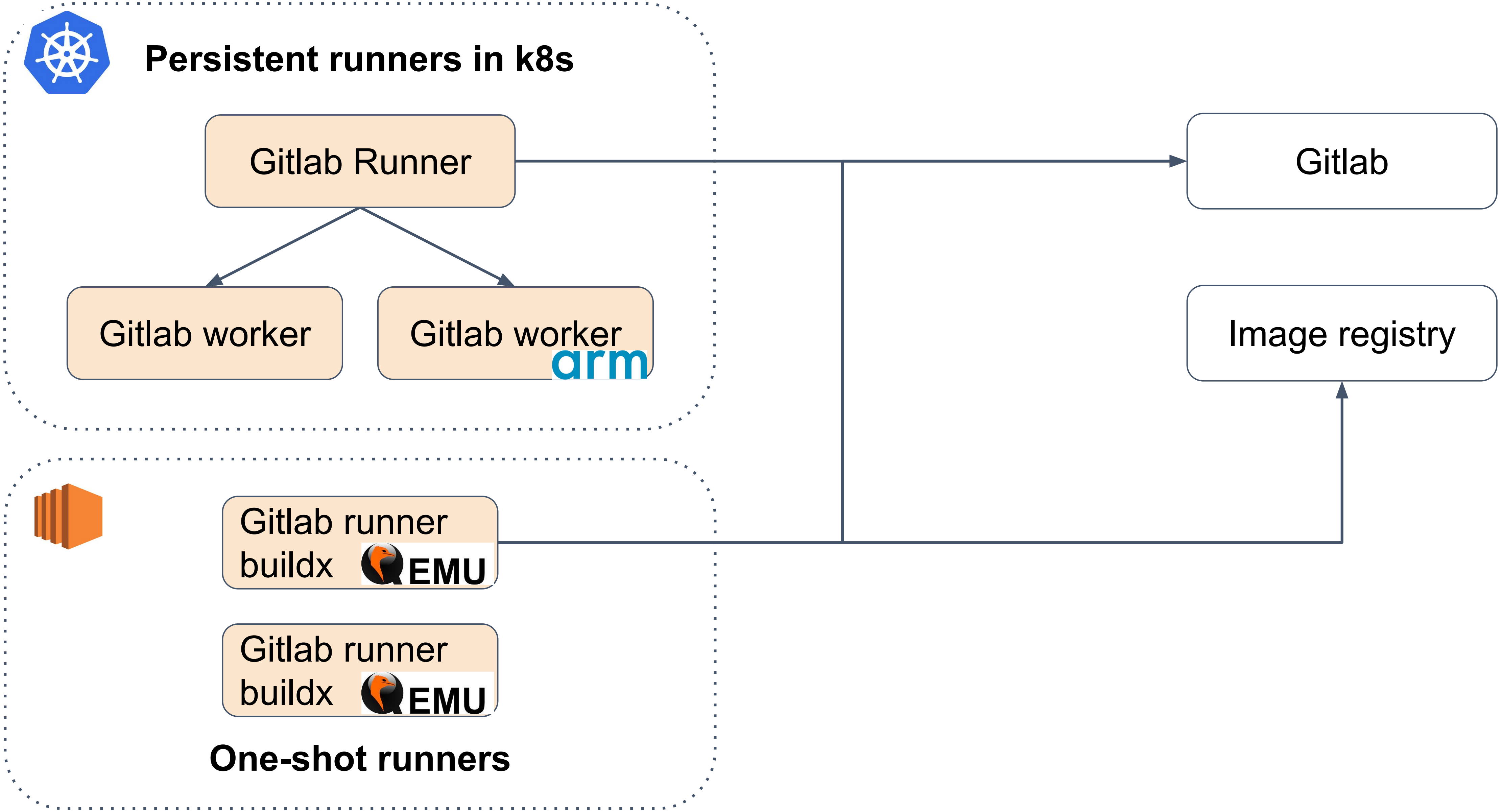
Gitlab

Image registry

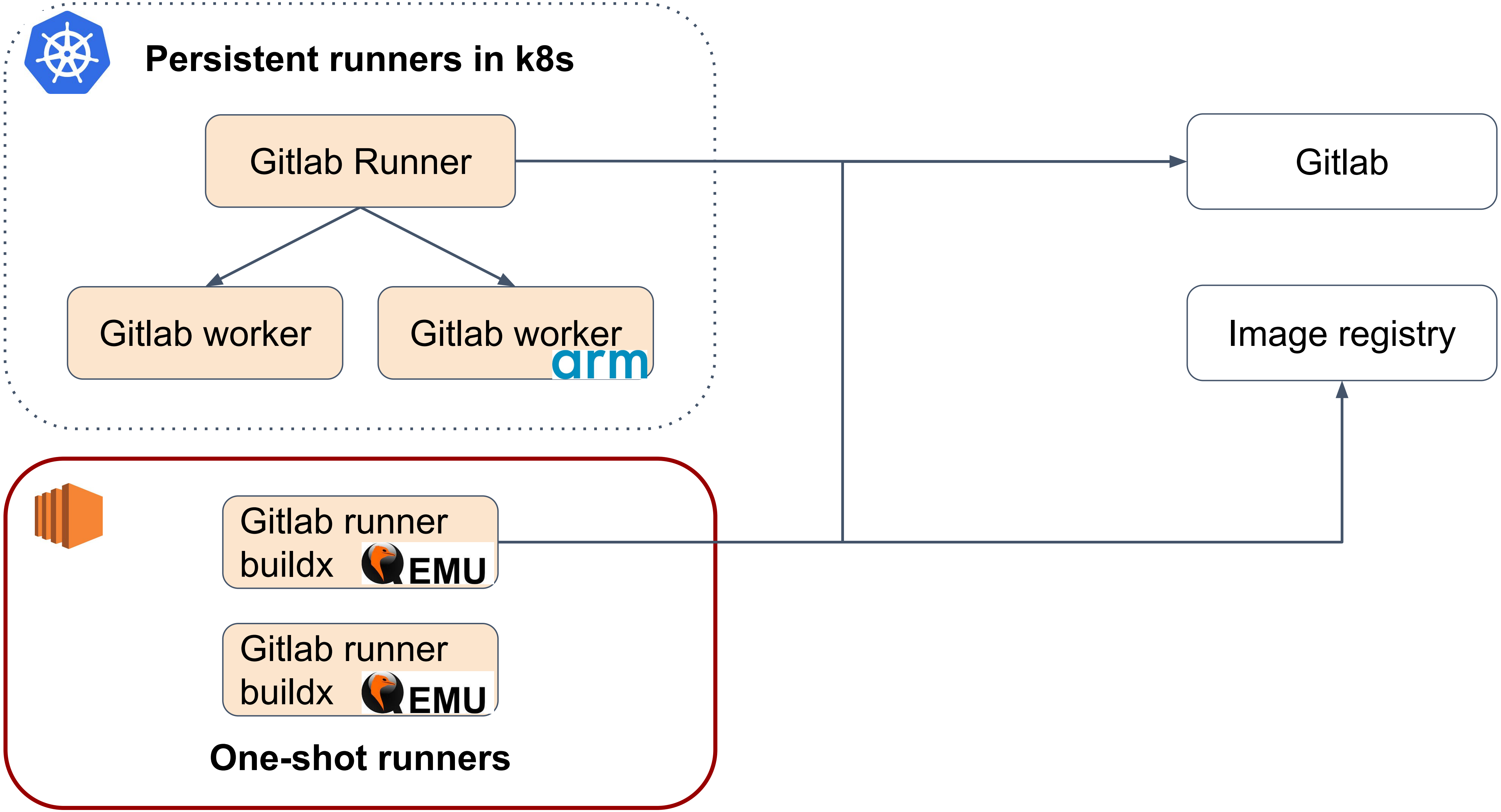
History: Building ARM binaries



History: Building multiarch images



History: Building multiarch images



Limits of this system

- Last workload outside of Kubernetes
- Several new needs requiring investment in legacy platform
- Challenging in terms of security

What if we could build images

in Kubernetes?

Can we build images in Kubernetes?

- docker-in-docker

- Standalone builders
  - buildah
  - kaniko
  - img

- Dedicated build daemons: buildkitd

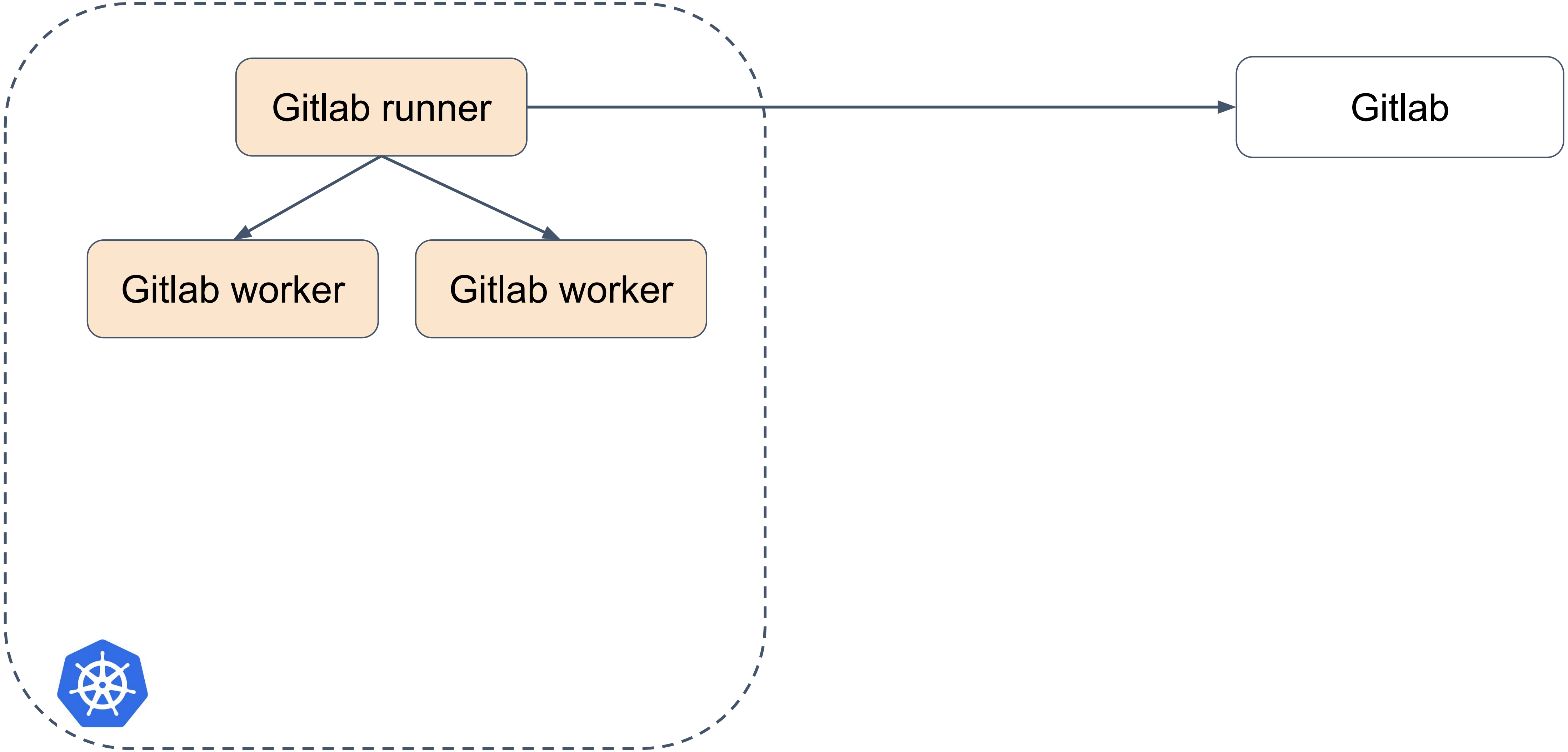
Can we use them?

- **docker-in-docker**
  => Security implications

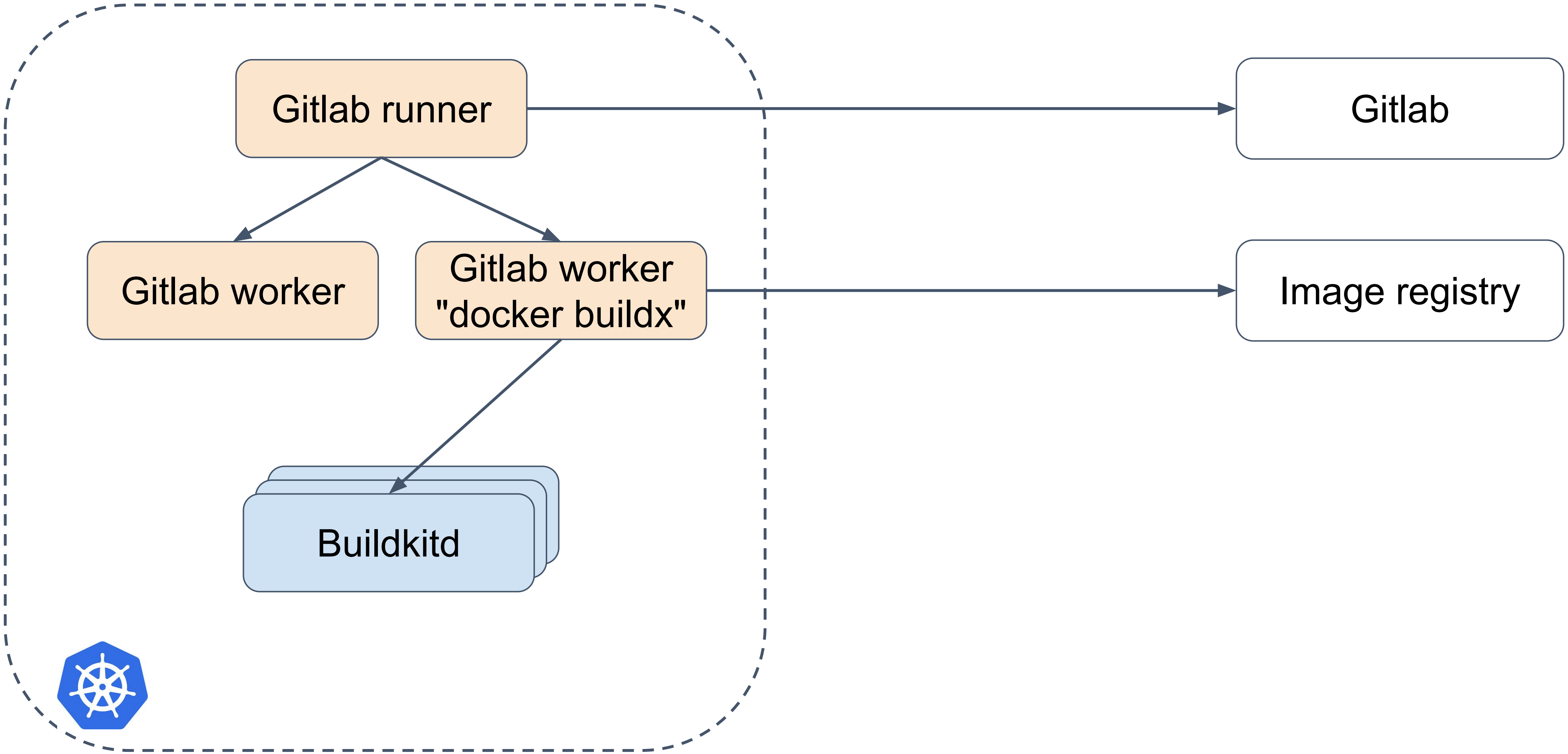
- **Standalone builders**
  => Complex setup: Distribute jobs / Assemble multiarch images

- Dedicated build daemons: **buildkitd**
  => Great UX, promising

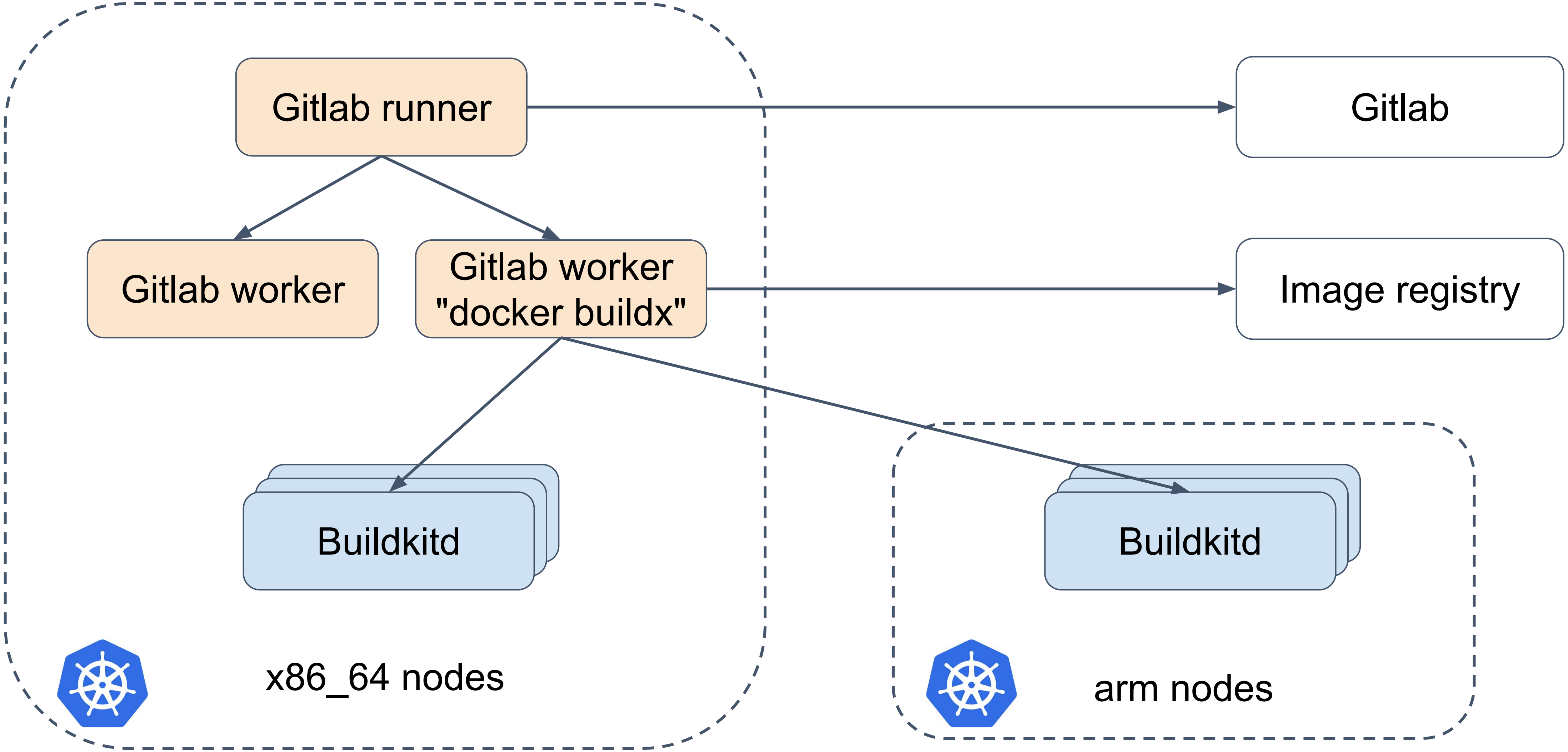
What would it look like?



What would it look like?



What would it look like?



Can we make this safe?

- Image builds require privileges
  - e.g. package installation
- Containers should run not run as root
- → “rootless” builds

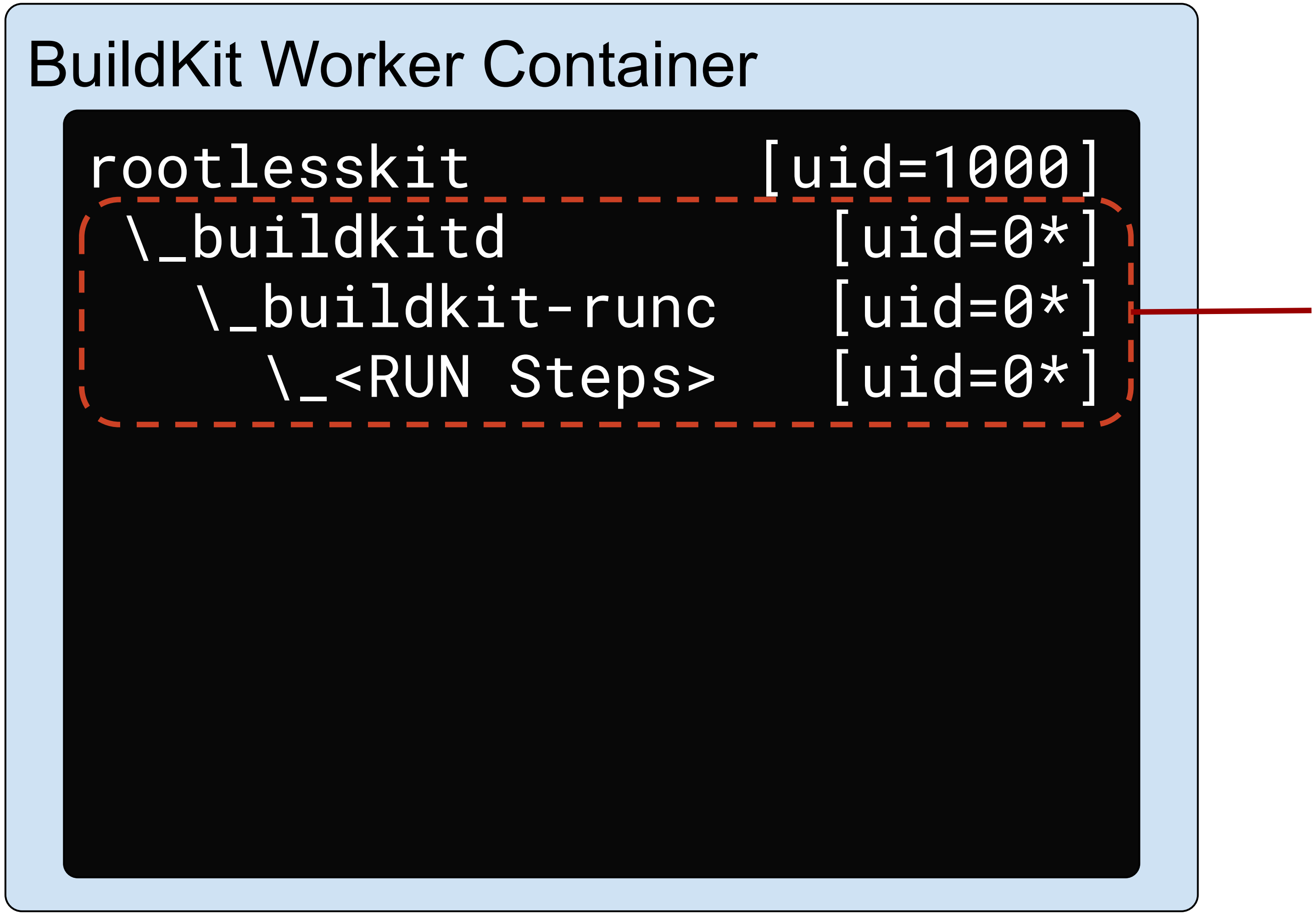
“rootless”?

- User namespaces
  - root user of host != root user in container

- Affects for instance:
  - Capabilities
  - Mount namespaces, VFS, filesystems
  - Linux Security Modules

- Building images efficiently and securely on Kubernetes with BuildKit
  Akihiro Suda, Kubecon EU, 2019

Overview

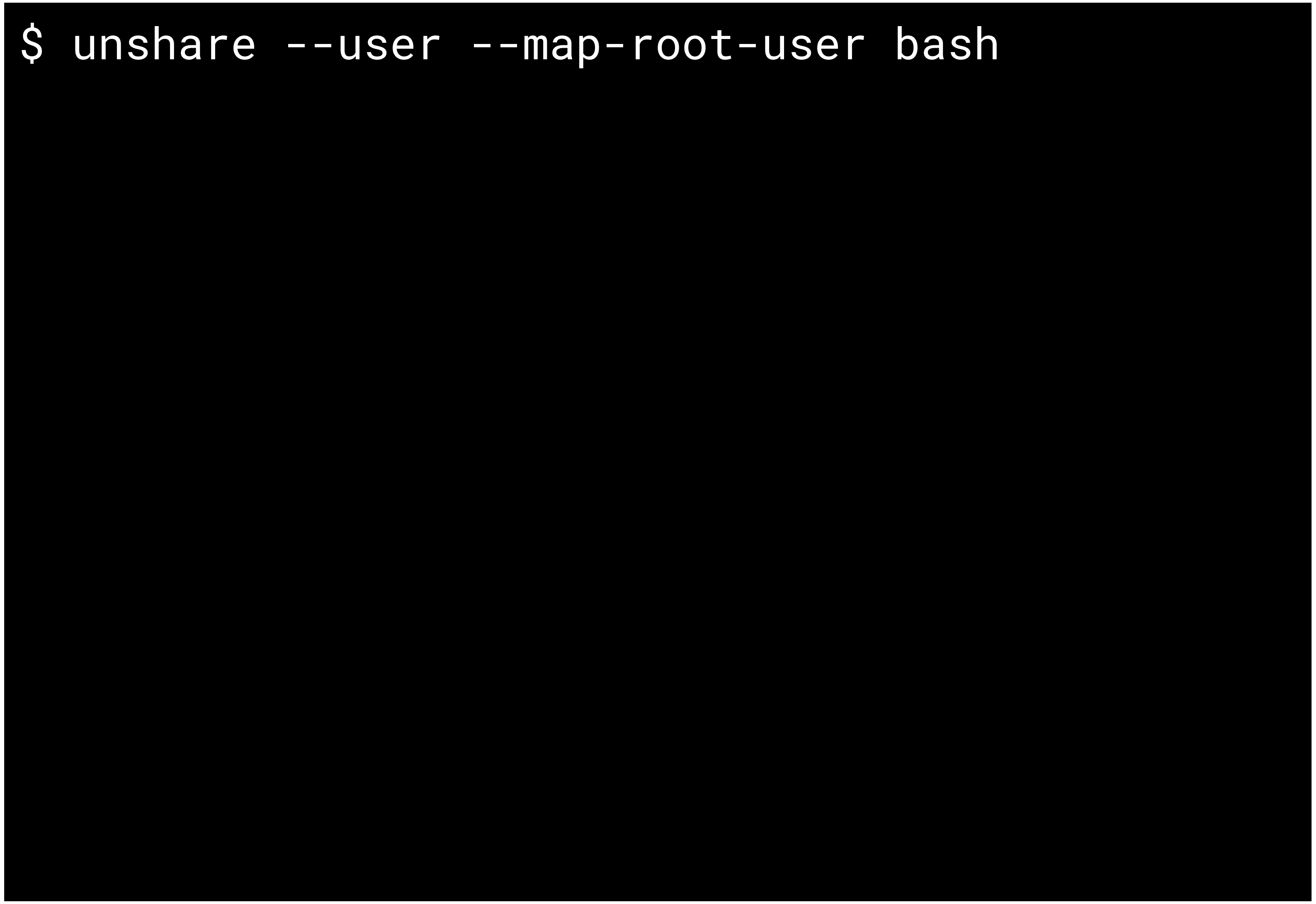


BuildKit Worker Container

```
rootlesskit              [uid=1000]
 \_buildkitd               [uid=0*]
   \_buildkit-runc         [uid=0*]
     \_<RUN Steps>         [uid=0*]
```

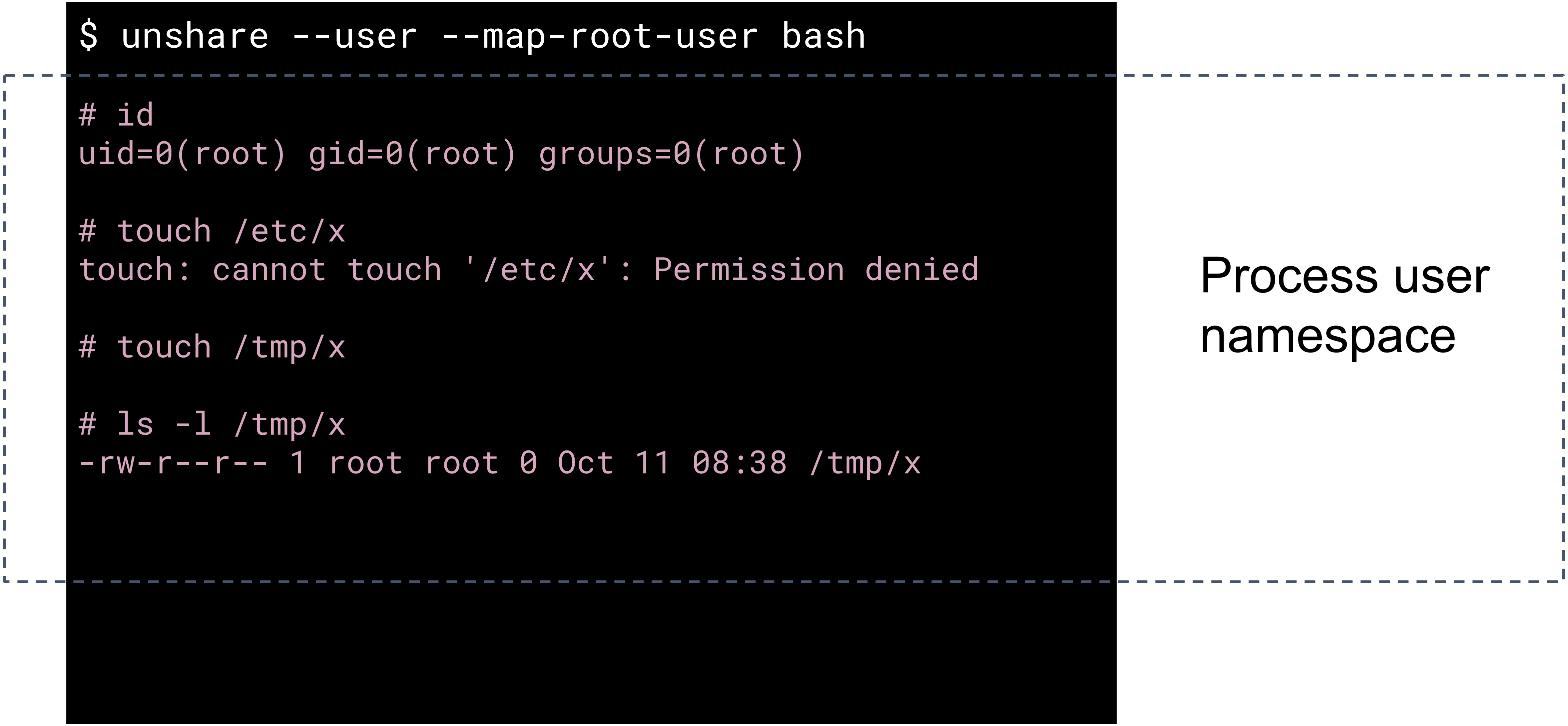
User namespace
uid 0 != uid 0 on the host

Example



```
$ unshare --user --map-root-user bash
```

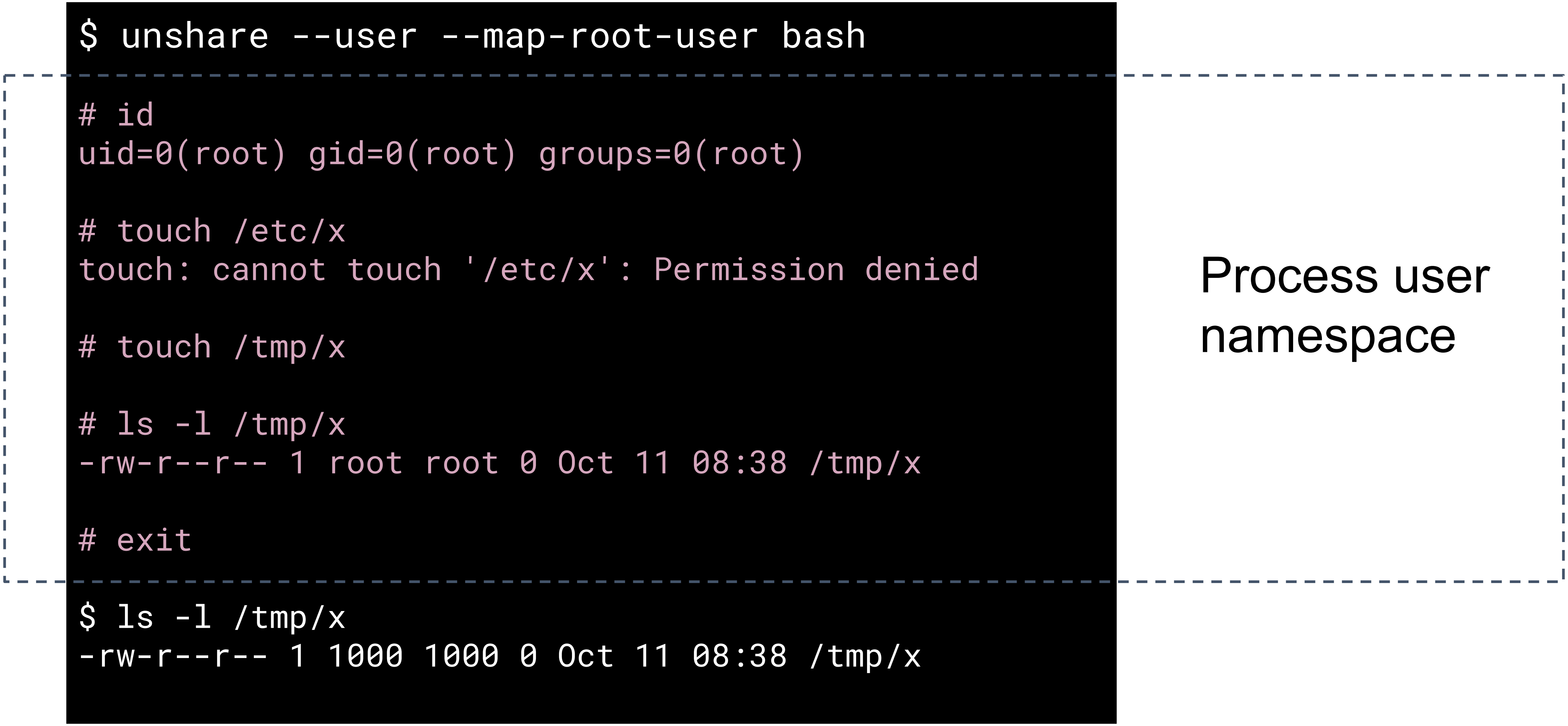
Example



```
$ unshare --user --map-root-user bash

# id
uid=0(root) gid=0(root) groups=0(root)

# touch /etc/x
touch: cannot touch '/etc/x': Permission denied

# touch /tmp/x

# ls -l /tmp/x
-rw-r--r-- 1 root root 0 Oct 11 08:38 /tmp/x
```

Process user namespace

Example



```
$ unshare --user --map-root-user bash

# id
uid=0(root) gid=0(root) groups=0(root)

# touch /etc/x
touch: cannot touch '/etc/x': Permission denied

# touch /tmp/x

# ls -l /tmp/x
-rw-r--r-- 1 root root 0 Oct 11 08:38 /tmp/x

# exit

$ ls -l /tmp/x
-rw-r--r-- 1 1000 1000 0 Oct 11 08:38 /tmp/x
```

Process user namespace

Before we engage

- Most images built fine on buildkit/kube from the start
- We’ll only talk about the tricky bits though…

Before we engage

- Most images built fine on buildkit/kube from the start
- We’ll only talk about the tricky bits though…
- **Where’s the "fun" otherwise?**

Issue #1: Why is RUN broken?

Time to Fun*: a few hours

* Time between "Wait what???" and "Oh that's fun"

A complex Dockerfile

```
FROM gitlab/gitlab-runner-helper:arm64-v14.0.0
RUN echo "test"
```

A complex Dockerfile

```
FROM gitlab/gitlab-runner-helper:arm64-v14.0.0
RUN echo "test"
```

```
$ docker buildx ...

#5 [2/2] RUN echo "test"
#5 ERROR: mount callback failed on
/run/user/1000/containerd-mount4xx: operation not permitted
```

A complex Dockerfile

```
FROM gitlab/gitlab-runner-helper:arm64-v14.0.0
RUN echo "test"
```

```
$ docker buildx ...

#5 [2/2] RUN echo "test"
#5 ERROR: mount callback failed on
/run/user/1000/containerd-mount4xx: operation not permitted
```

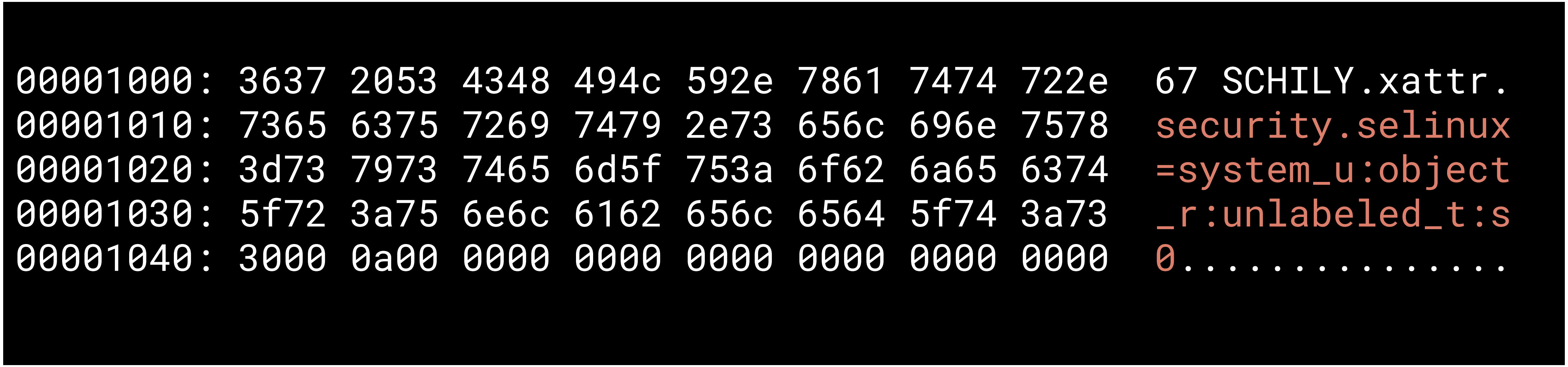
Surprising, right?

Which Operation is Denied?

```
# strace -f -p <buildkitd_pid> -e trace=%file |& grep EPERM


... lsetxattr("...",
              "security.selinux",
              "system_u:..."...,
              33, 0)
      = -1 EPERM (Operation not permitted)
```

It's always ~~DNS~~ SELinux!

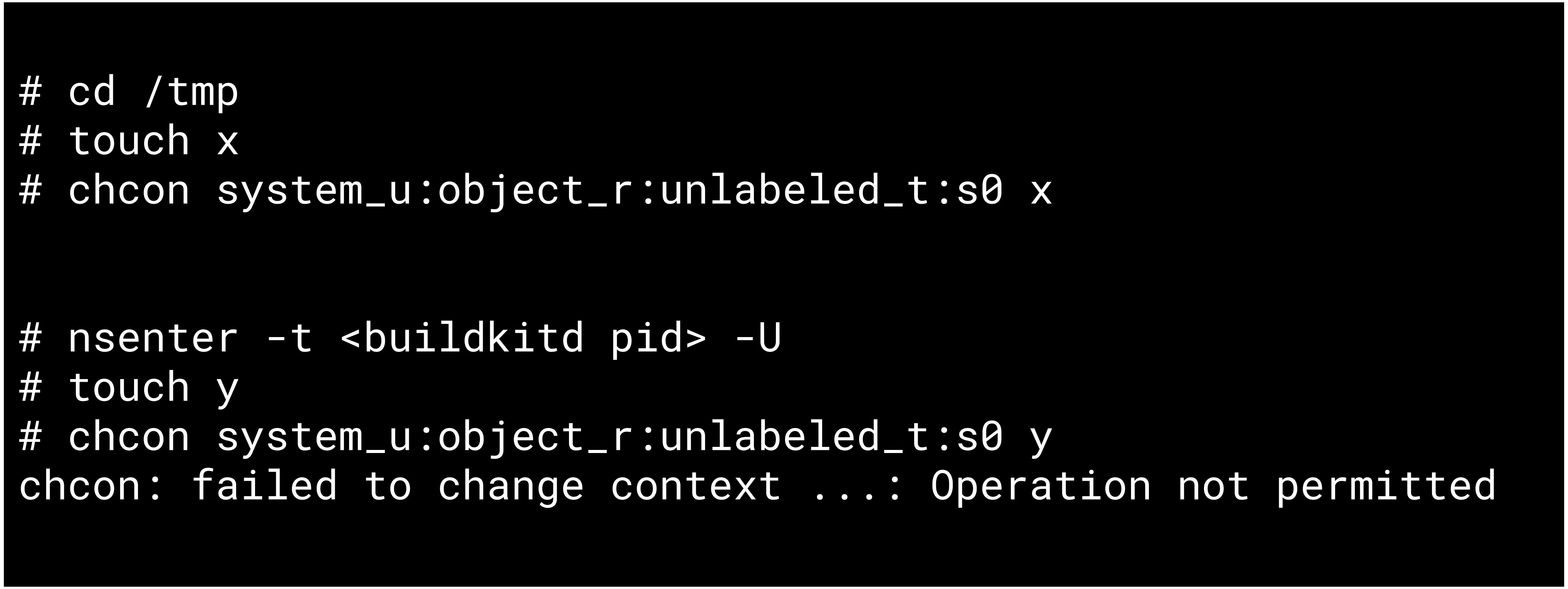
Delving Into Image Layer



```
00001000: 3637 2053 4348 494c 592e 7861 7474 722e  67 SCHILY.xattr.
00001010: 7365 6375 7269 7479 2e73 656c 696e 7578  security.selinux
00001020: 3d73 7973 7465 6d5f 753a 6f62 6a65 6374  =system_u:object
00001030: 5f72 3a75 6e6c 6162 656c 6564 5f74 3a73  _r:unlabeled_t:s
00001040: 3000 0a00 0000 0000 0000 0000 0000 0000  0...............
```

⇒ Files in tarball have SELinux extended attributes

SELinux in User Namespaces



```
# cd /tmp
# touch x
# chcon system_u:object_r:unlabeled_t:s0 x


# nsenter -t <buildkitd pid> -U
# touch y
# chcon system_u:object_r:unlabeled_t:s0 y
chcon: failed to change context ...: Operation not permitted
```

Host User NS

Process User NS

Kernel disallows setting SELinux context because
root in user namespace ≠ root in initial user namespace

Issue

- Upstream issue: https://github.com/moby/buildkit/issues/2407

- No real solution but easy enough to mitigate
  - Pull / Push to create an image without SELinux labels
  - Use an image without SELinux labels (gitlab-runner-helper:v14.1.0)

Issue #2: Who stole my port?

Time to Fun: a few days

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
          && dpkg -i some_app.deb \
          && rm some_app.deb
```

- Build times out

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
          && dpkg -i some_app.deb \
          && rm some_app.deb
```

- Build times out
- Let's retry!

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
         && dpkg -i some_app.deb \
         && rm some_app.deb
```

- Build times out
- Let's retry!
- RUN fails but **with an error this time**

```
#2 [2/2] RUN curl...
#2 ERROR: Bind failed: Address already in use
```

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
         && dpkg -i some_app.deb \
         && rm some_app.deb
```

- Build times out
- Let's retry!
- Let's be scientific and retry **again**

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
         && dpkg -i some_app.deb \
         && rm some_app.deb
```

- Build times out
- Let's retry!
- Let's be scientific and retry **again**

```
#2 [2/2] RUN curl...
#2 ERROR: Bind failed: Address already in use
```

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
          && dpkg -i some_app.deb \
          && rm some_app.deb
```

- Build times out
- Let's retry!
- Let's be scientific and retry again
- What if we delete the buildkitd pod?

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
          && dpkg -i some_app.deb \
          && rm some_app.deb
```

- Build times out
- Let's retry!
- Let's be scientific and retry again
- What if we delete the buildkitd pod?
- Build times out

A more complex Dockerfile

```
RUN curl -OL https://.../some_app.deb \
         && dpkg -i some_app.deb \
         && rm some_app.deb
```

- Build times out
- Let's retry!
- Let's be scientific and retry again
- What if we delete the buildkitd pod?
- Build times out
- Following builds get the error

```
#2 [2/2] RUN curl...
#2 ERROR: Bind failed: Address already in use
```

Let's debug!

```
RUN netstat -tunlp &&
    curl -OL https://.../some_app.deb \
        && dpkg -i some_app.deb \
        && rm some_app.deb \
        && netstat -tunlp
```

With a brand new buildkitd pod

- **1st** netstat: no port bound
- **2nd** netstat: **"some_app" port is bound by some_appd**
- Build hangs

Let's debug!

```
RUN netstat -tunlp &&
    curl -OL https://.../some_app.deb \
        && dpkg -i some_app.deb \
        && rm some_app.deb \
        && netstat -tunlp
```

Next build

- **1st** netstat: **"some_app" port is bound by some_appd**
- dpkg -i fails with "**Address already in use**"

What's happening?

```
RUN netstat -tunlp &&
    curl -OL https://.../some_app.deb \
         && dpkg -i some_app.deb \
         && rm some_app.deb
```

=> some_app.deb starts a daemon on install

=> Looks like the daemon is still running when we do build #2

What's happening?

```
RUN netstat -tunlp &&
    curl -OL https://.../some_app.deb \
        && dpkg -i some_app.deb \
        && rm some_app.deb
```

=> some_app.deb starts a daemon on install
=> Looks like the daemon is still running when we do build #2

Can we reproduce?

Basic Reproducer

```
FROM ubuntu
ADD script.sh /
RUN /script.sh
RUN echo "Dockerfile done"
```

```
#!/bin/bash
set -x
sleep infinity &
echo "Script done"
```

```
=> [3/4] RUN /script.sh
=> => # + echo 'Script done'
=> => # + sleep infinity
=> => # Script done
...hangs...
```

Basic Reproducer

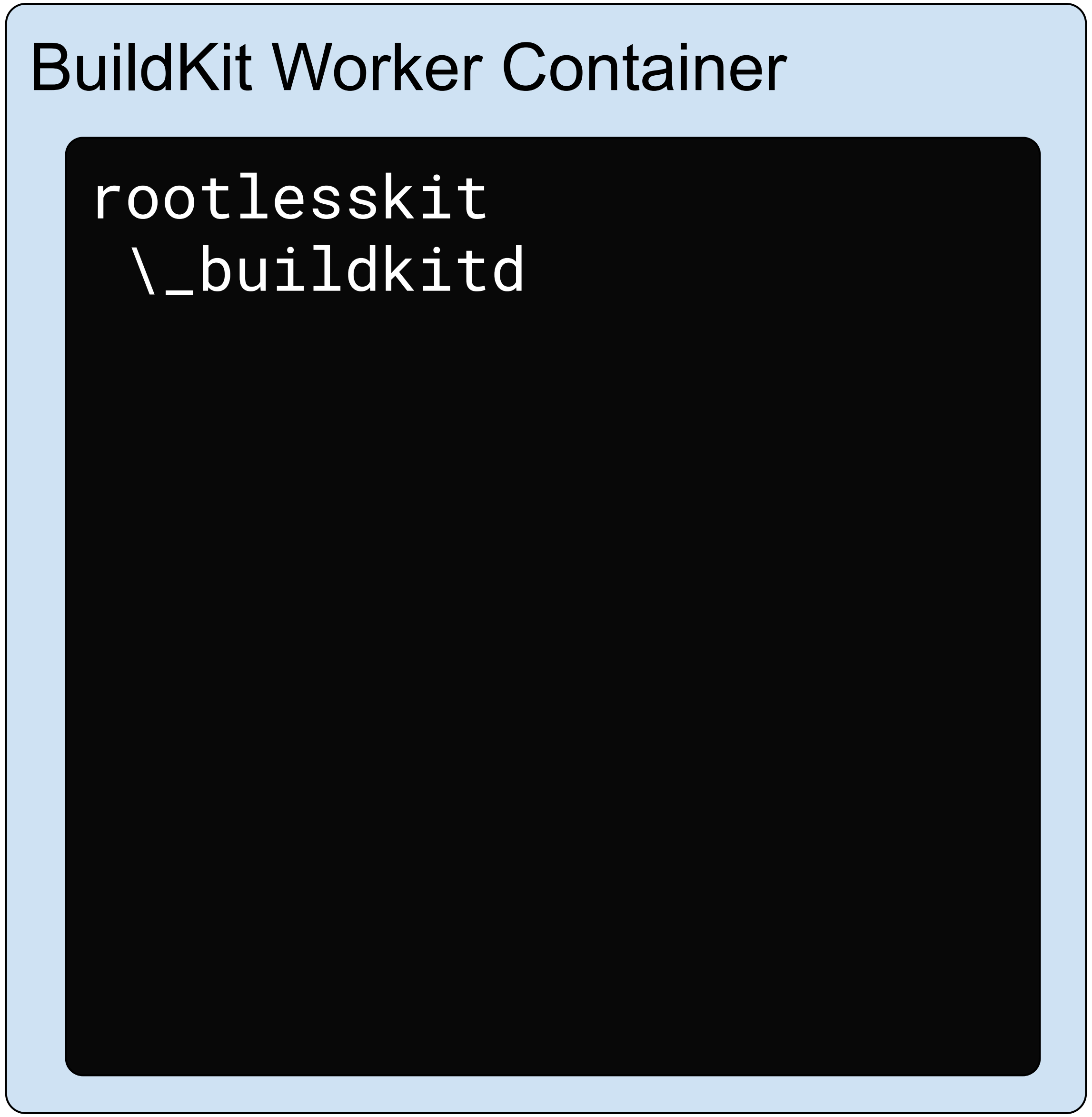
```
FROM ubuntu
ADD script.sh /
RUN /script.sh
RUN echo "Dockerfile done"
```

```
#!/bin/bash
set -x
sleep infinity &
echo "Script done"
```

```
=> [3/4] RUN /script.sh
=> => # + echo 'Script done'
=> => # + sleep infinity
=> => # Script done
...hangs...
```

=> Process leaked when build times out or is interrupted

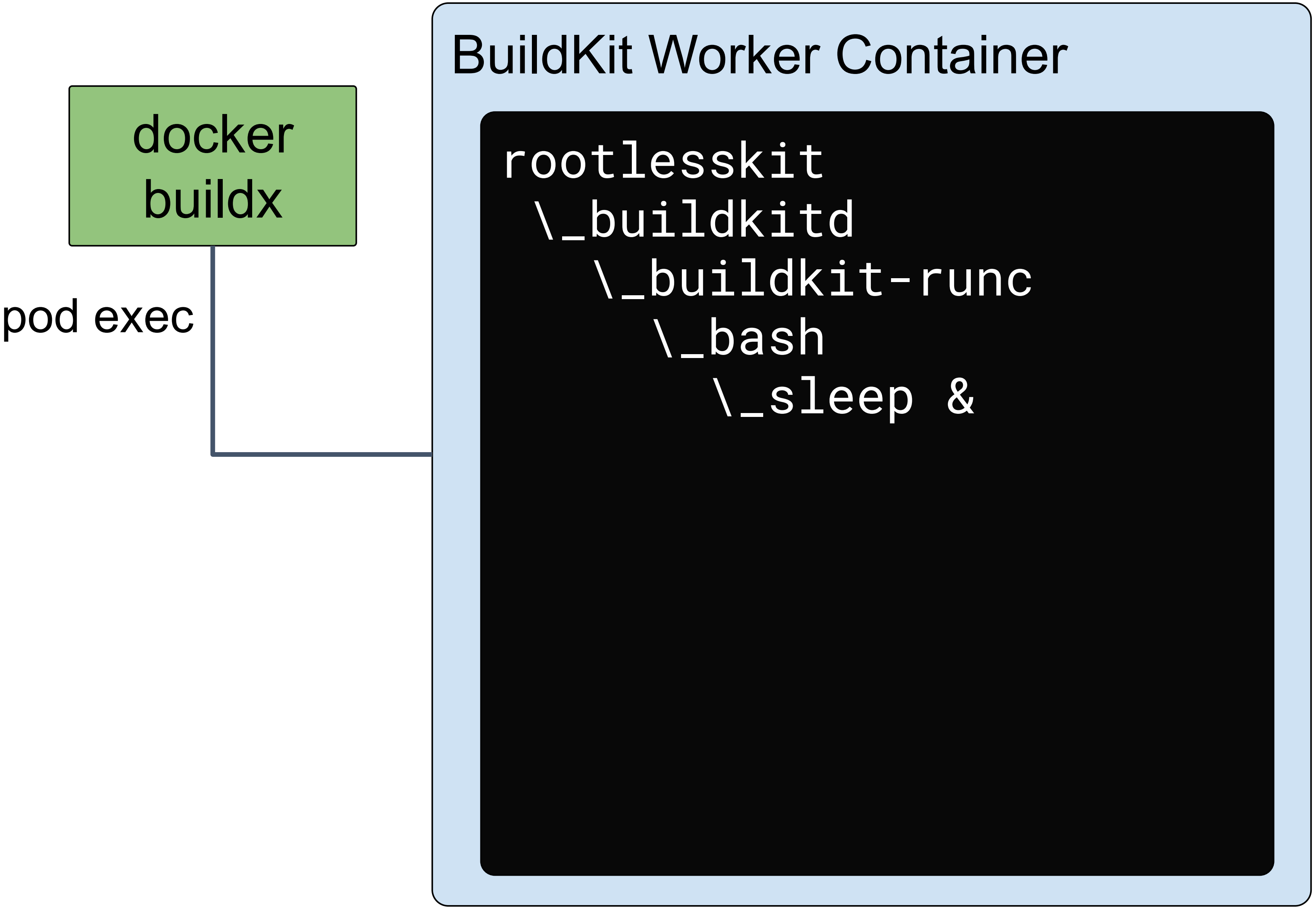
Let’s Peek Under the Hood



BuildKit Worker Container

```
rootlesskit
 \_buildkitd
```

Building our image



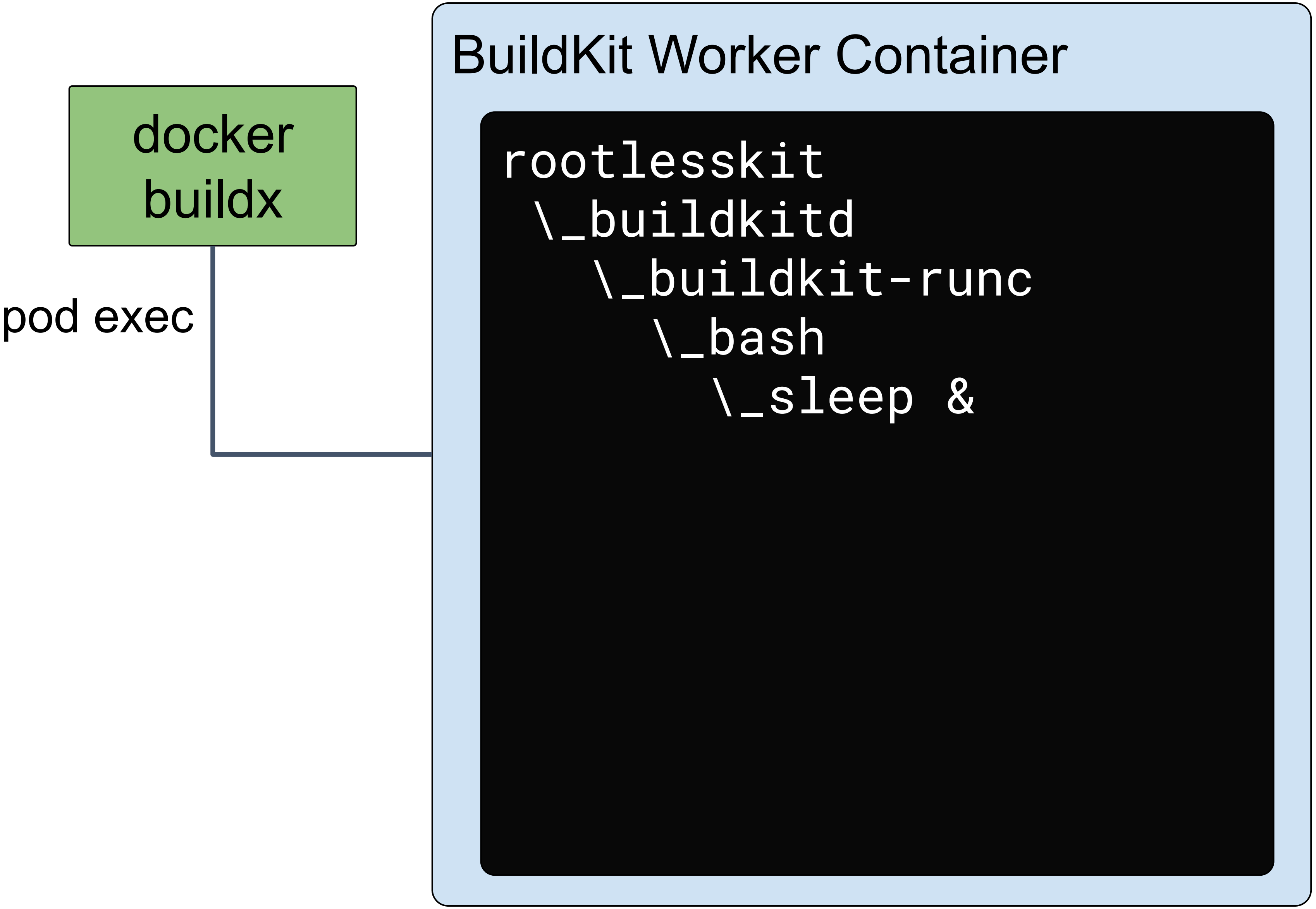
docker buildx

pod exec

BuildKit Worker Container

```
rootlesskit
 \_buildkitd
   \_buildkit-runc
     \_bash
       \_sleep &
```

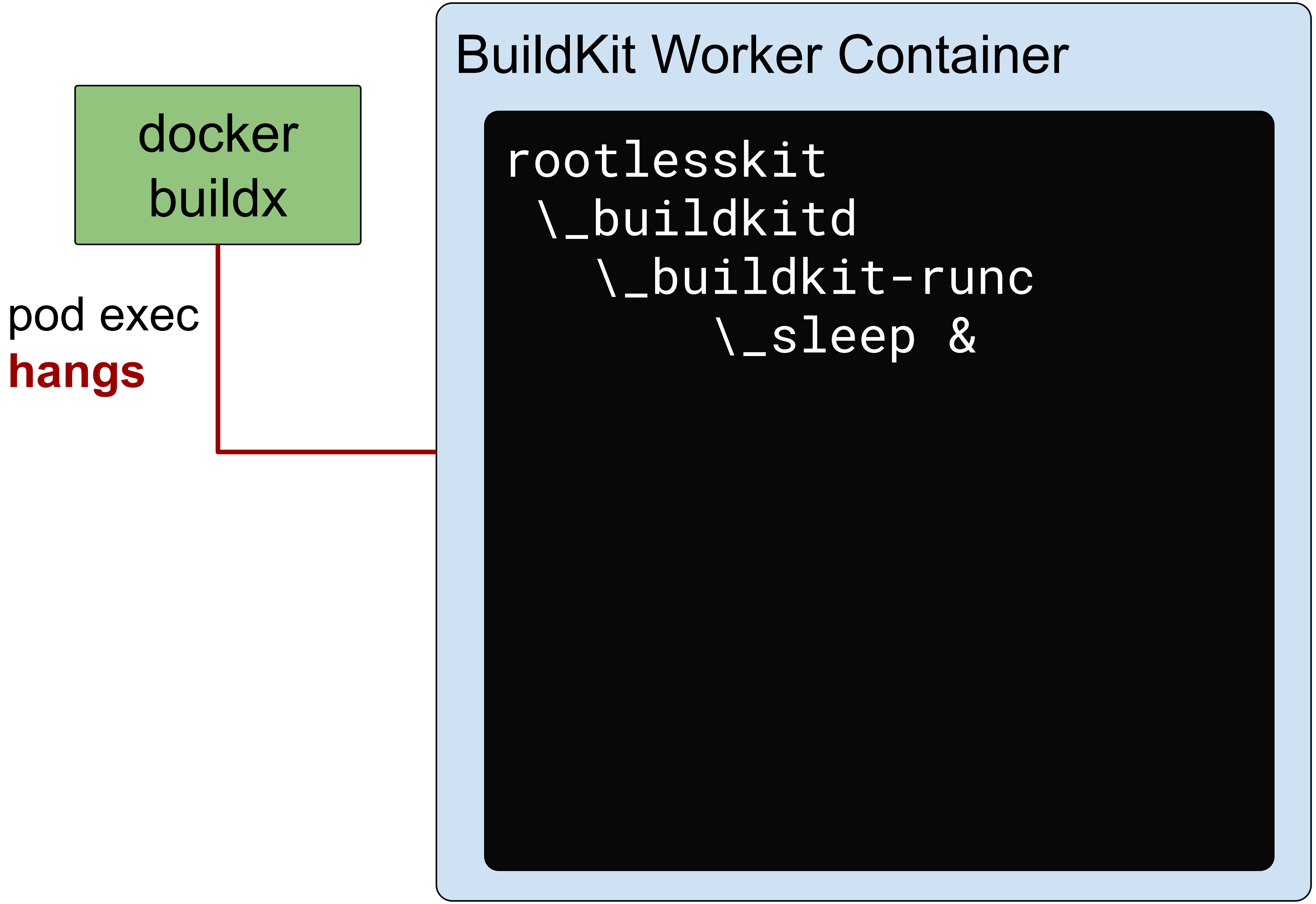
Building our image



```
rootlesskit
 \_buildkitd
   \_buildkit-runc
     \_bash
       \_sleep &
```

No process sandbox
Shared process view

bash exits



docker buildx

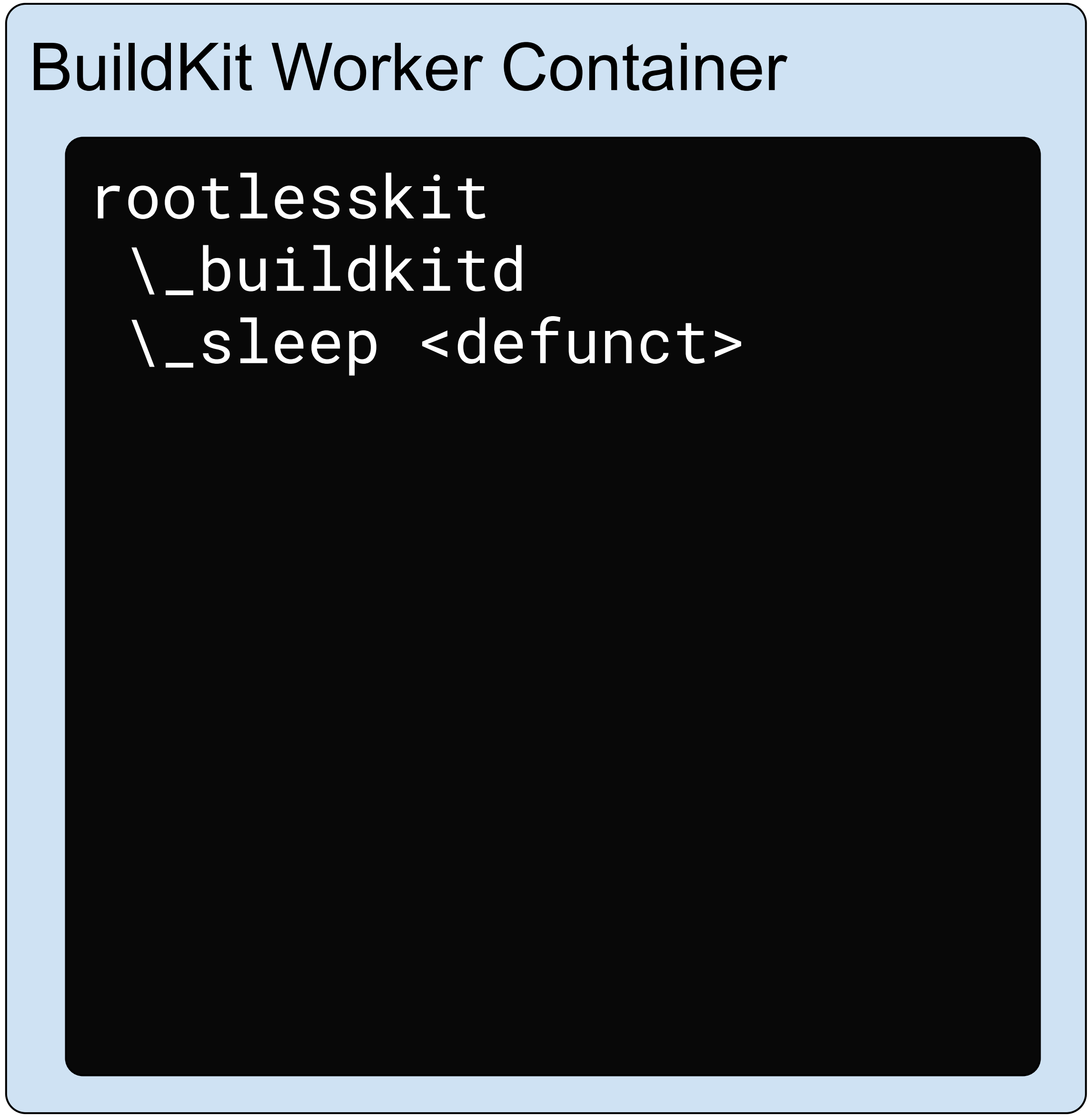
pod exec **hangs**

BuildKit Worker Container

```
rootlesskit
 \_buildkitd
   \_buildkit-runc
       \_sleep &
```

Process clean-up fails

What if we kill sleep?



BuildKit Worker Container

```
rootlesskit
 \_buildkitd
 \_sleep <defunct>
```

We get a Zombie

How does it work usually?

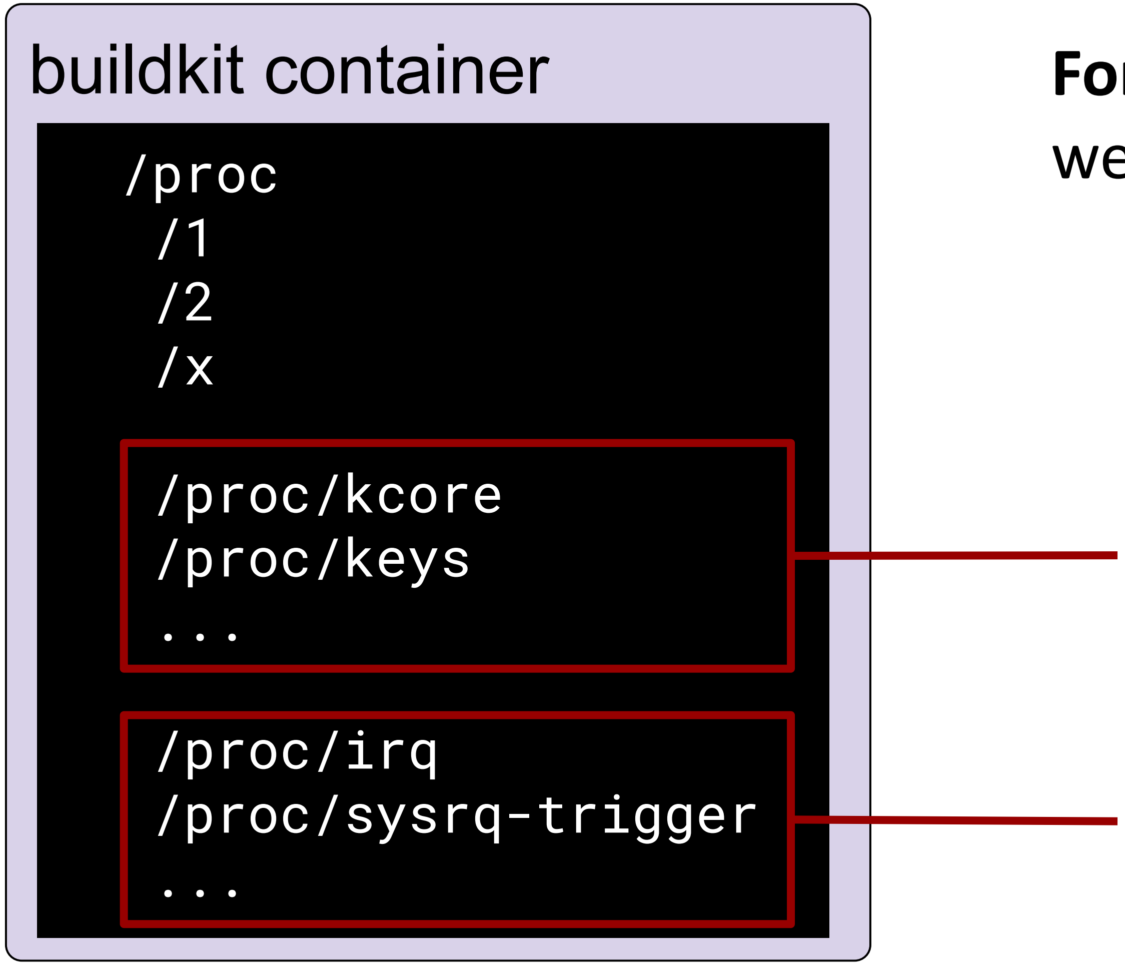
- Build steps run in a process sandbox

- When the step finishes, all processes in the sandbox are killed

=> We don't have this in rootless mode (**--oci-worker-no-process-sandbox**)
=> We can't fully keep track of processes started in the build steps

Why do we need this flag?

/proc in containers



buildkit container

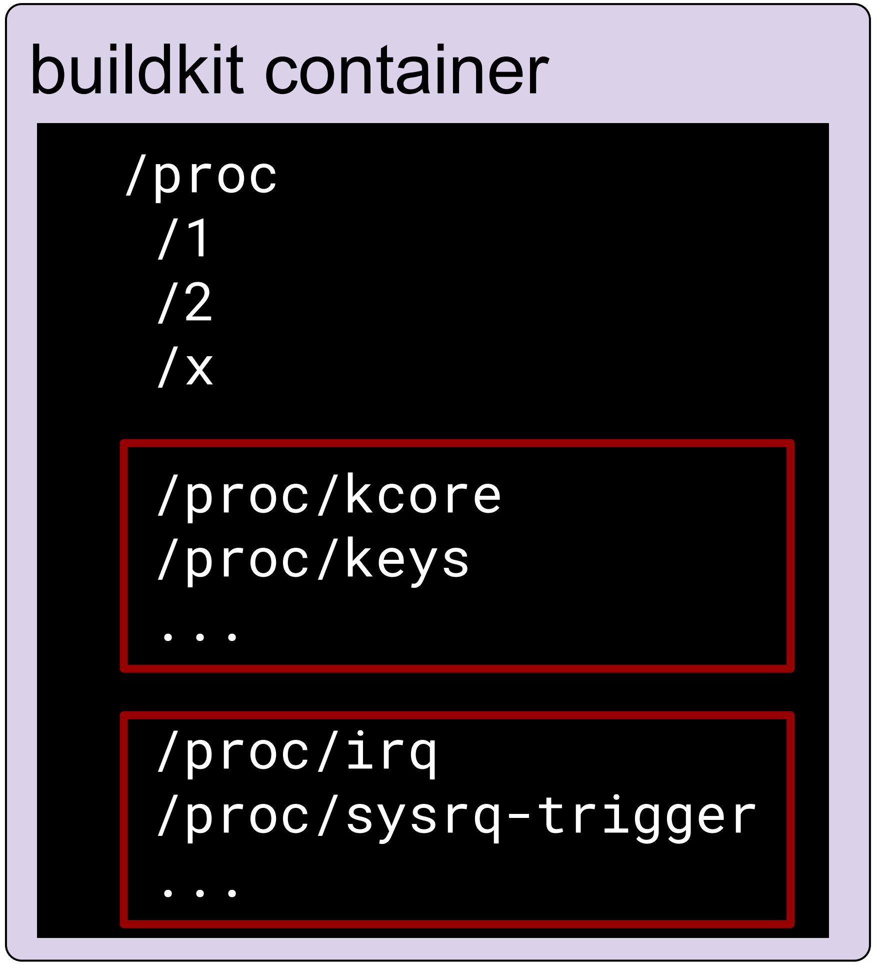
```
/proc
 /1
 /2
 /x

 /proc/kcore
 /proc/keys
 ...

 /proc/irq
 /proc/sysrq-trigger
 ...
```

For security reasons,
we can't fully expose /proc to containers

masked: bind mounted with empty dir
(prevents access to sensitive data)

readonly: bind mounted read-only
(prevents system-wide modifications)

/proc in containers



buildkit container

```
  /proc
   /1
   /2
   /x

   /proc/kcore
   /proc/keys
   ...

   /proc/irq
   /proc/sysrq-trigger
   ...
```

Creating a new procfs in this container fails

=> Kernel check: "mount_too_revealing"
If existing procfs is partially masked, disallow mount

=> Explains need for --oci-worker-no-process-sandbox

Details:
https://github.com/opencontainers/runc/issues/1658

Conclusion

- No real solution to date: https://github.com/moby/buildkit/issues/2417

- Mitigations
  - **Make sure no daemons are started or stop them**
  - Potentially: ProcMountType=Unmasked
  - Use jobs for builds

- Security/stability implications
  - Leaked processes/zombies
  - Affect subsequent builds (e.g. bound ports)

Issue #3: Ghosts in the filesystem

Time to Fun: more than a week

Let’s Build Some Go

```
RUN git clone https://[...]/sig-storage-local-static-provisioner

RUN git -C ... checkout $TAG

RUN go build -o ... cmd/local-volume-provisioner/main.go
```

Compiler Error

```
vendor/k8s.io/utils/io/read.go:36:6:
  ConsistentRead redeclared in this block

previous declaration at
  vendor/k8s.io/utils/io/consistentread.go:28:61
```

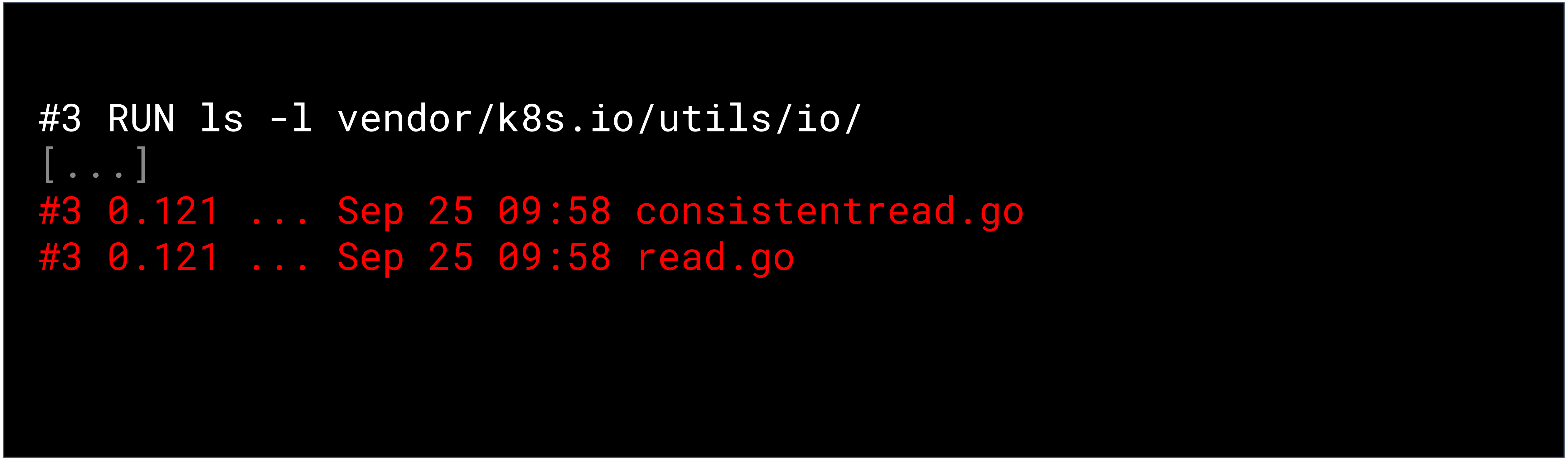
Compiler Error

```
vendor/k8s.io/utils/io/read.go:36:6:
  ConsistentRead redeclared in this block

previous declaration at
  vendor/k8s.io/utils/io/consistentread.go:28:61
```

This makes no sense:

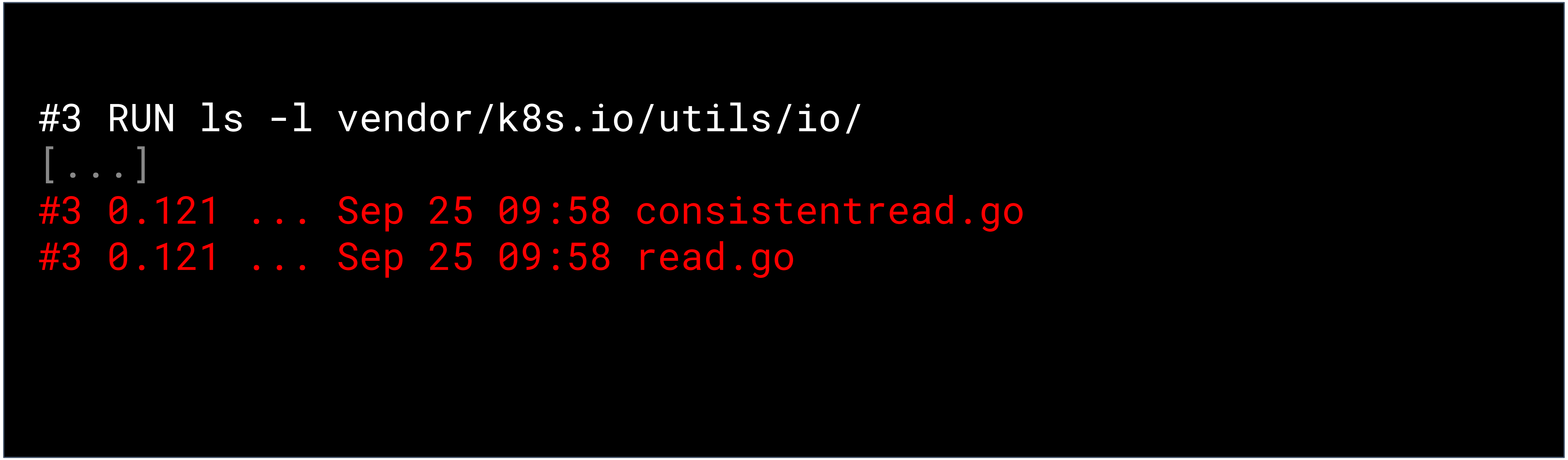
- Same Dockerfile builds fine with docker
- **Why would rootless buildkit trigger this???**

Let’s Check the Directory



```
#3 RUN ls -l vendor/k8s.io/utils/io/
[...]
#3 0.121 ... Sep 25 09:58 consistentread.go
#3 0.121 ... Sep 25 09:58 read.go
```

Let’s Check the Directory

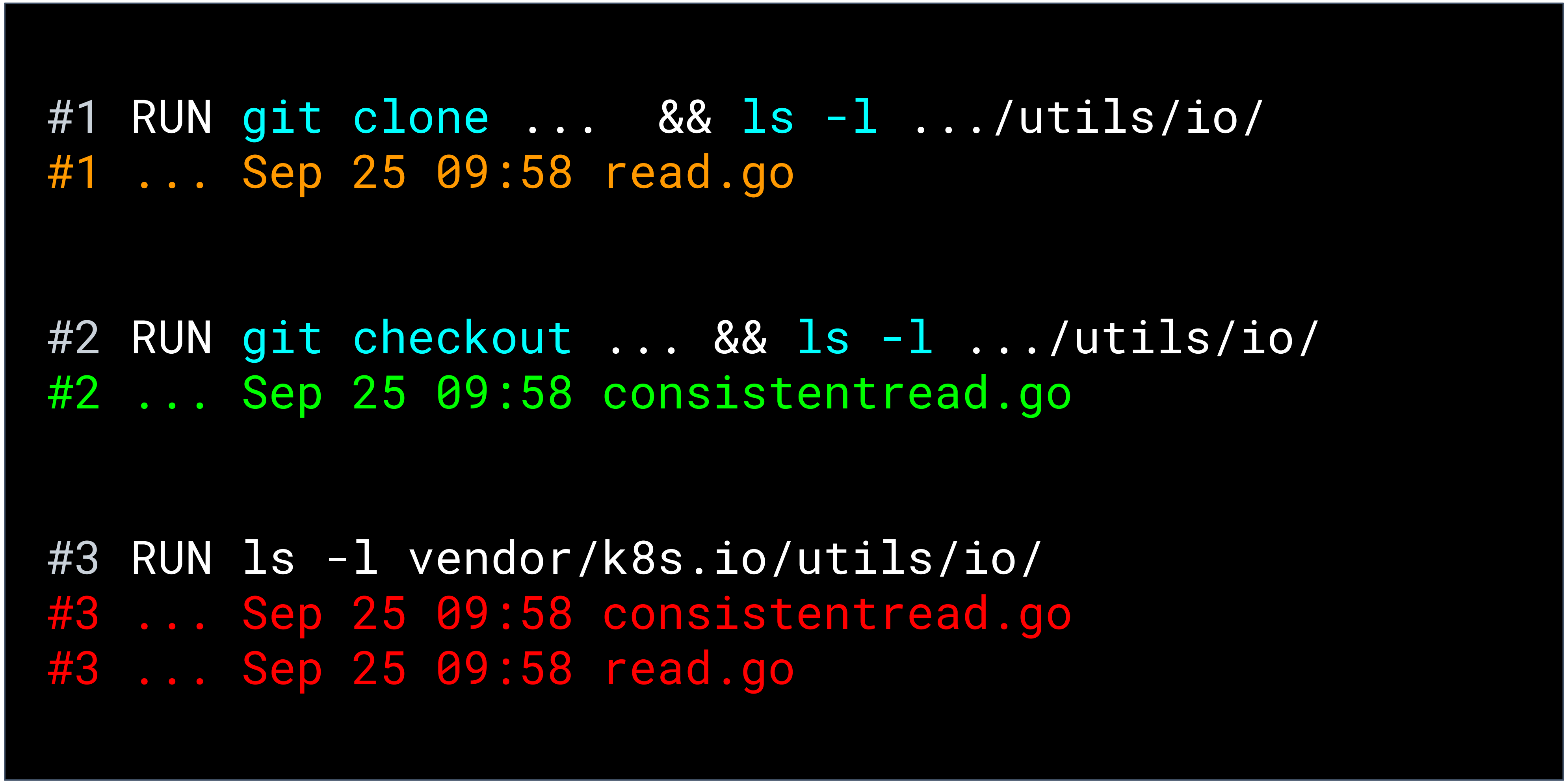


```
#3 RUN ls -l vendor/k8s.io/utils/io/
[...]
#3 0.121 ... Sep 25 09:58 consistentread.go
#3 0.121 ... Sep 25 09:58 read.go
```

In master, directory only contains "read.go"
In tag, directory only contains "consistentread.go"

How can we have both files in this directory?

Check Each Layer



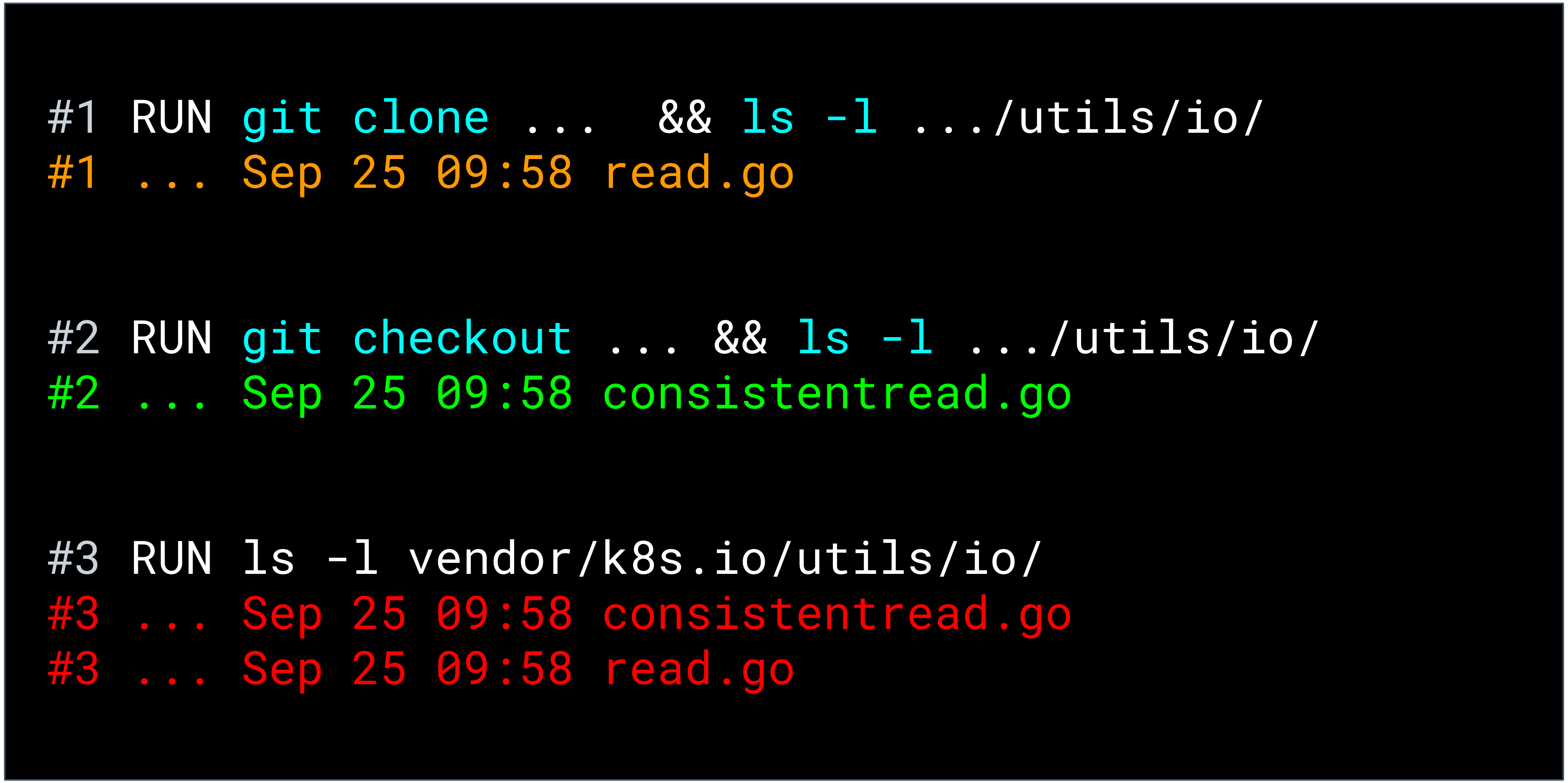
```
#1 RUN git clone ...  && ls -l .../utils/io/
#1 ... Sep 25 09:58 read.go


#2 RUN git checkout ... && ls -l .../utils/io/
#2 ... Sep 25 09:58 consistentread.go


#3 RUN ls -l vendor/k8s.io/utils/io/
#3 ... Sep 25 09:58 consistentread.go
#3 ... Sep 25 09:58 read.go
```

Expected

Expected

Wait. What?

Check Each Layer



```
#1 RUN git clone ...  && ls -l .../utils/io/
#1 ... Sep 25 09:58 read.go


#2 RUN git checkout ... && ls -l .../utils/io/
#2 ... Sep 25 09:58 consistentread.go


#3 RUN ls -l vendor/k8s.io/utils/io/
#3 ... Sep 25 09:58 consistentread.go
#3 ... Sep 25 09:58 read.go
```

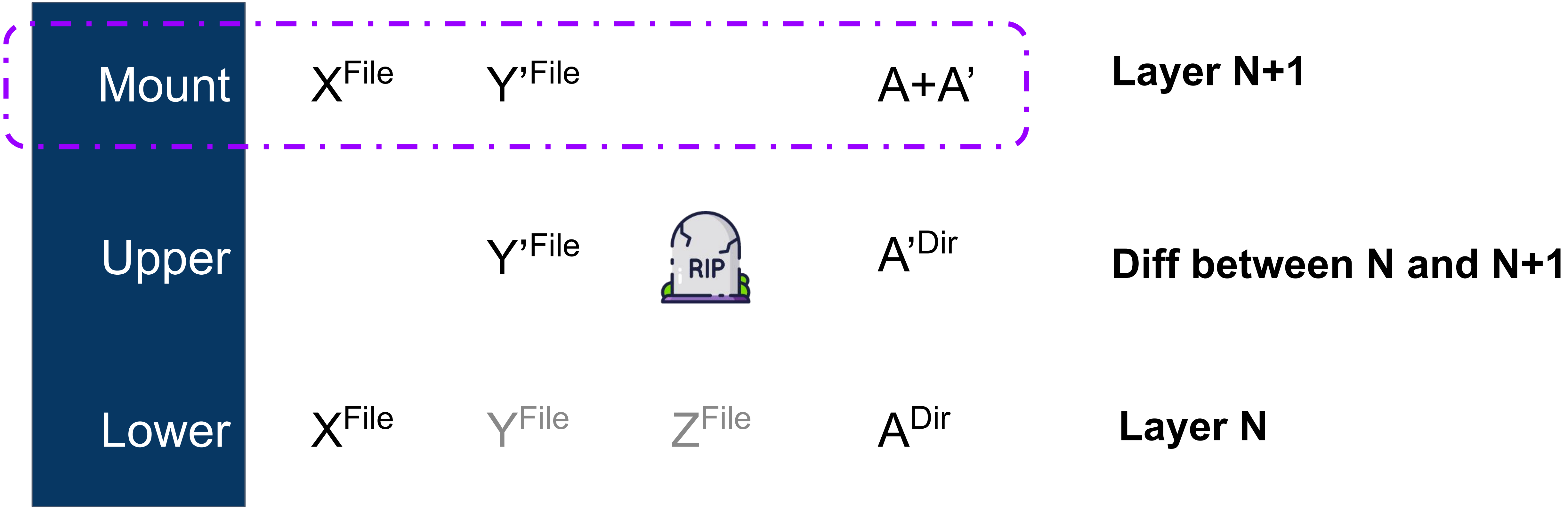
Expected

Expected

Wait. What?

Something is very wrong with our filesystem

We use Overlayfs. How does it work?



Mount	X^{File}	Y'^{File}		A+A'	**Layer N+1**
Upper		Y'^{File}	RIP	A'^{Dir}	**Diff between N and N+1**
Lower	X^{File}	Y^{File}	Z^{File}	A^{Dir}	**Layer N**

Reproducing the buildkit steps

```
$ unshare --mount --user --map-root-user bash
# mkdir layer1 layer2 l2diff work
```

Init build env

```
# git clone <...> layer1
```

Run step #1

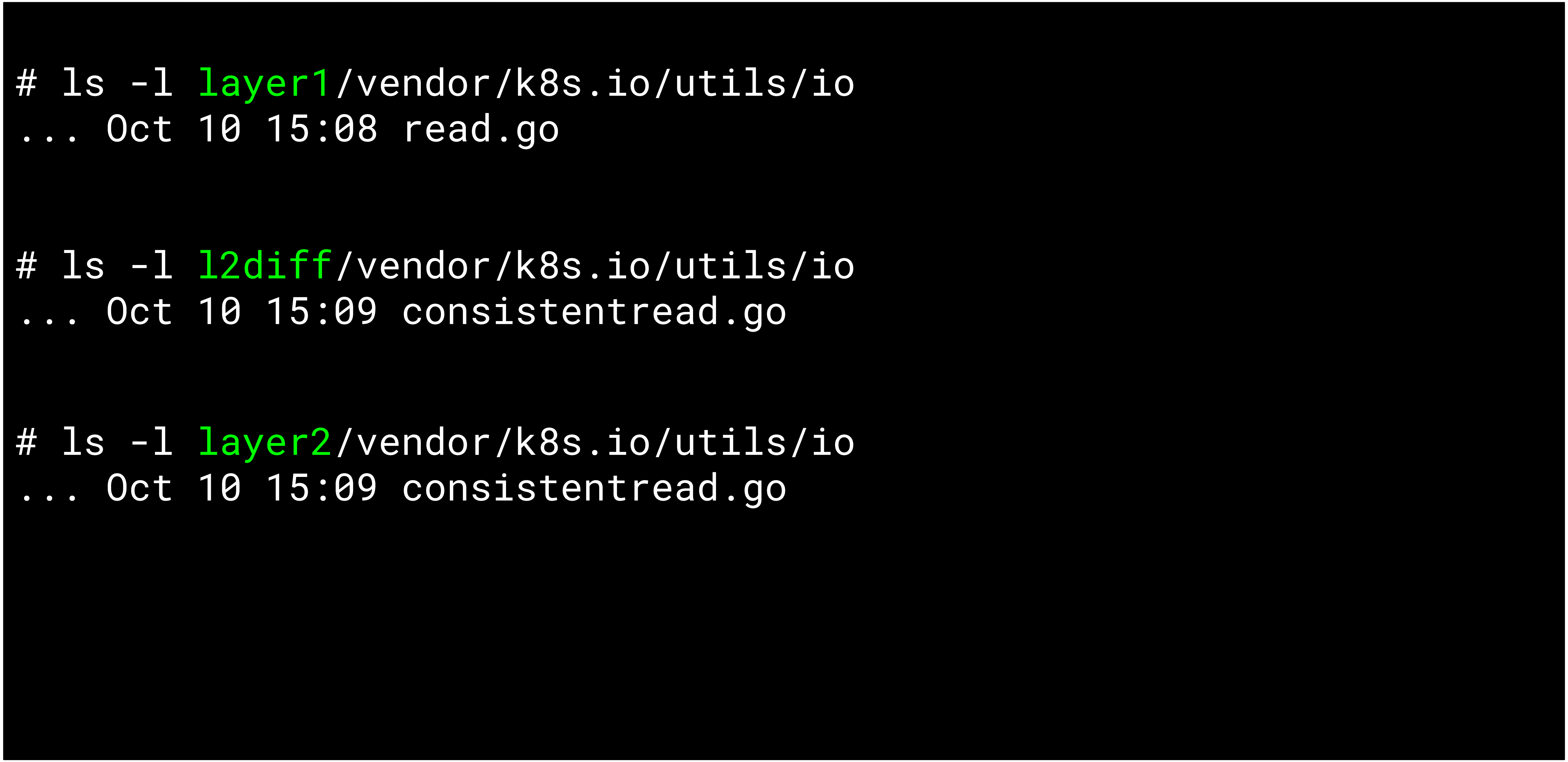
```
# mount -t overlay overlay \
      -olowerdir=layer1,upperdir=l2diff,workdir=work \
      layer2
```

Prepare step #2

```
# git -C layer2 checkout ...
```

Run step #2

At this point everything looks ok



```
# ls -l layer1/vendor/k8s.io/utils/io
... Oct 10 15:08 read.go


# ls -l l2diff/vendor/k8s.io/utils/io
... Oct 10 15:09 consistentread.go


# ls -l layer2/vendor/k8s.io/utils/io
... Oct 10 15:09 consistentread.go
```

What about step #3?

```
# umount layer2
# mkdir l3diff layer3 work3

# mount -t overlay overlay \
     -olowerdir=l2diff,layer1,upperdir=l3diff,workdir=work3 \
     layer3
```

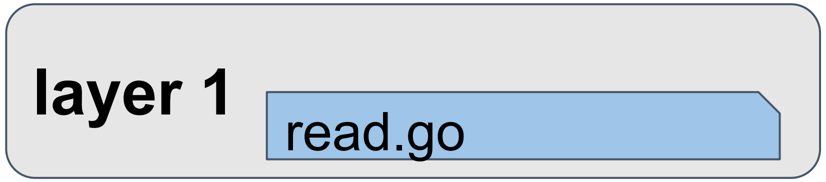
Prepare step #3

```
# ls layer3
 ... Oct 10 15:08 read.go
 ... Oct 10 15:09 consistentread.go
```

Run step #3

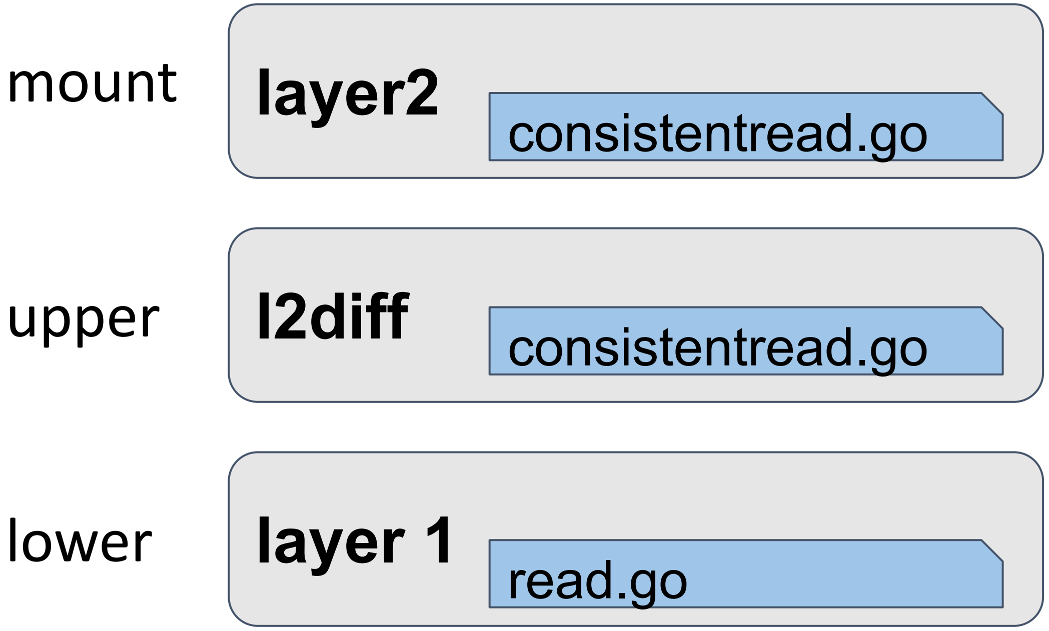
Definitely broken, we have reproduced!

Taking a step back



Step #1: git clone

Taking a step back



mount

layer2

consistentread.go

upper

l2diff

consistentread.go

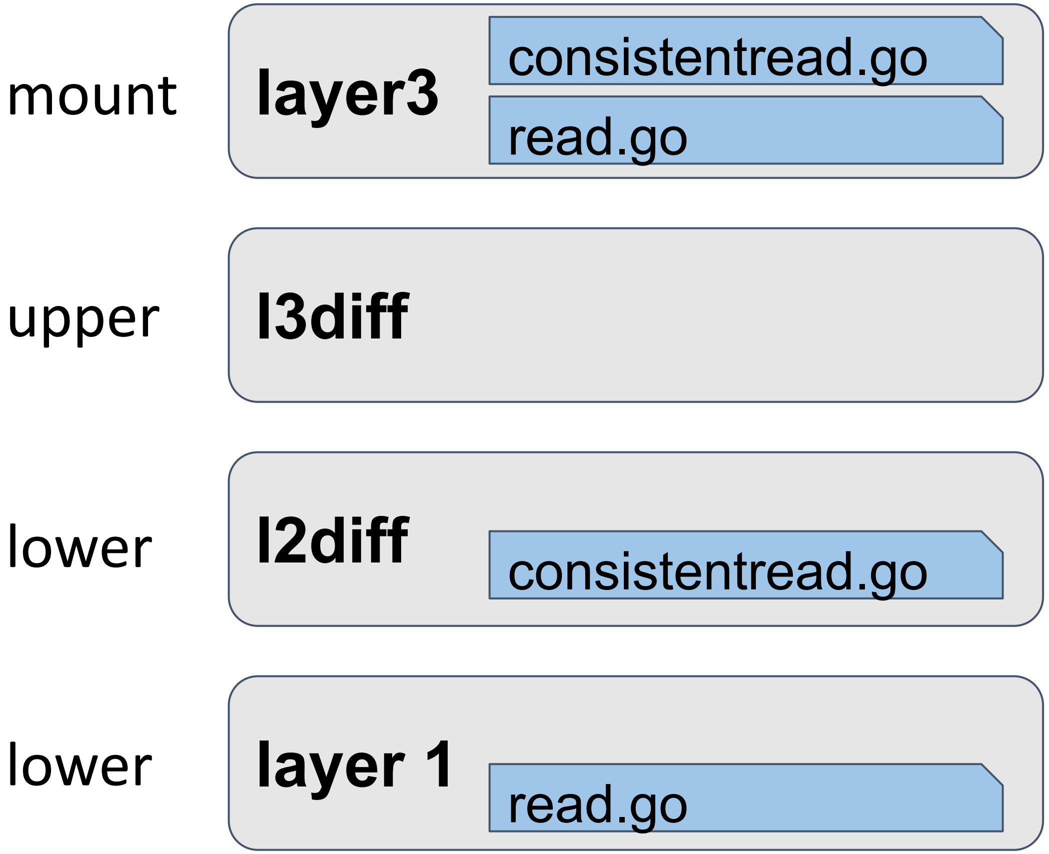
lower

layer 1

read.go

Step #2: git checkout

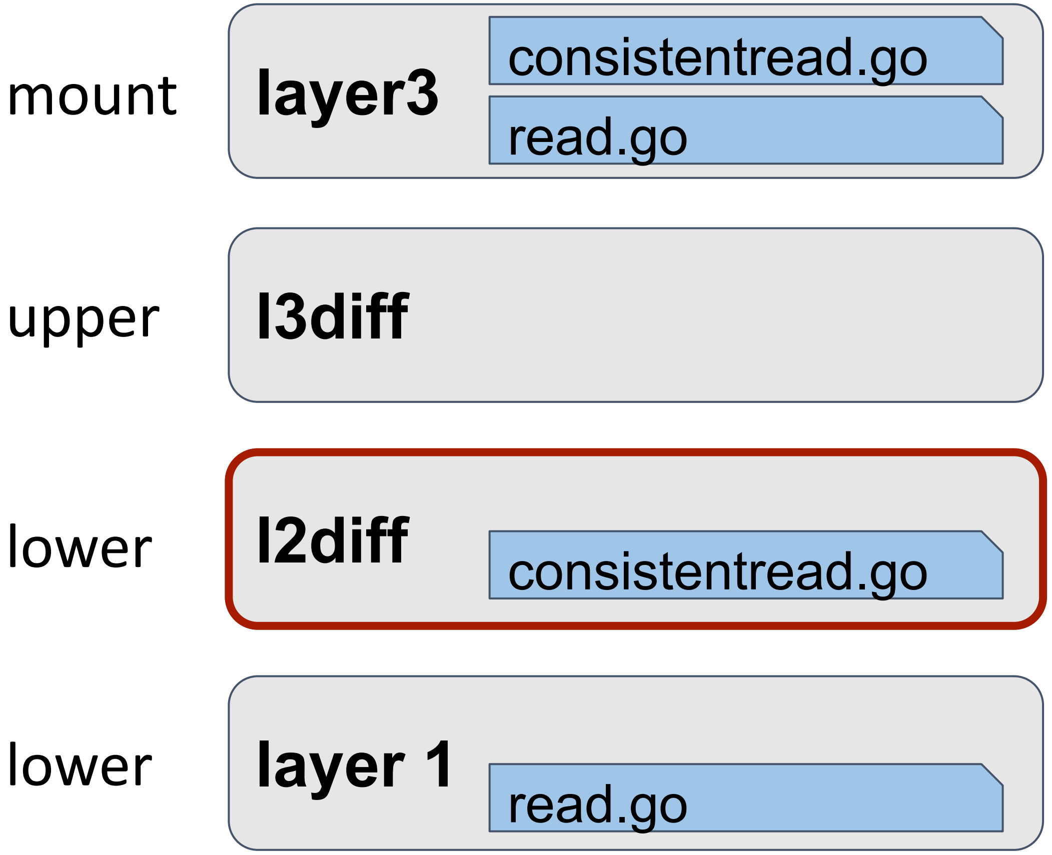
Taking a step back



mount	**layer3**	consistentread.go read.go
upper	**l3diff**	
lower	**l2diff**	consistentread.go
lower	**layer 1**	read.go

Step #3: ls

Something feels wrong with l2diff



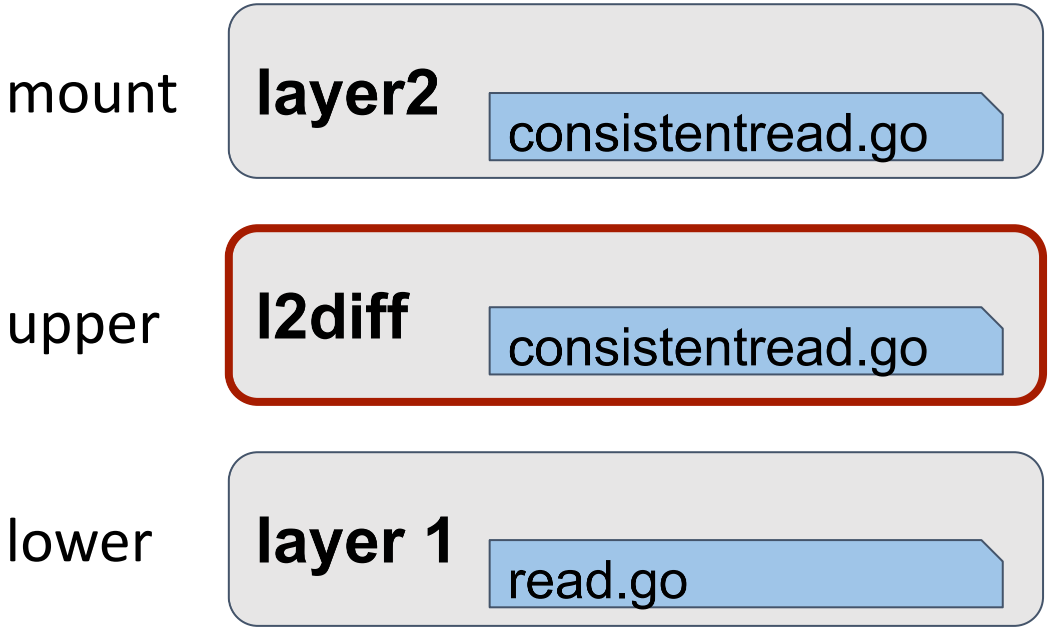
mount layer3 consistentread.go read.go

upper l3diff

lower l2diff consistentread.go

lower layer 1 read.go

Looking more closely at l2diff



mount

layer2

consistentread.go

upper

l2diff

consistentread.go

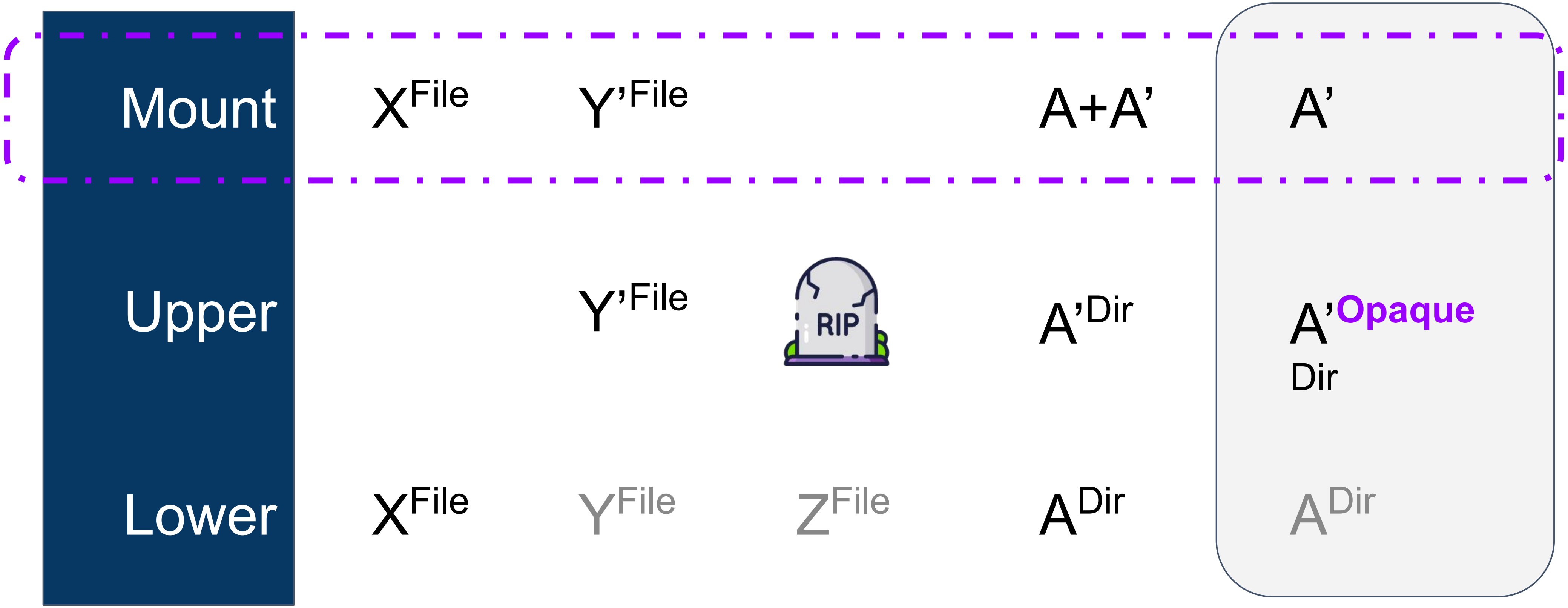
lower

layer 1

read.go

Where is the tombstone for read.go?

Overlayfs: we missed a bit

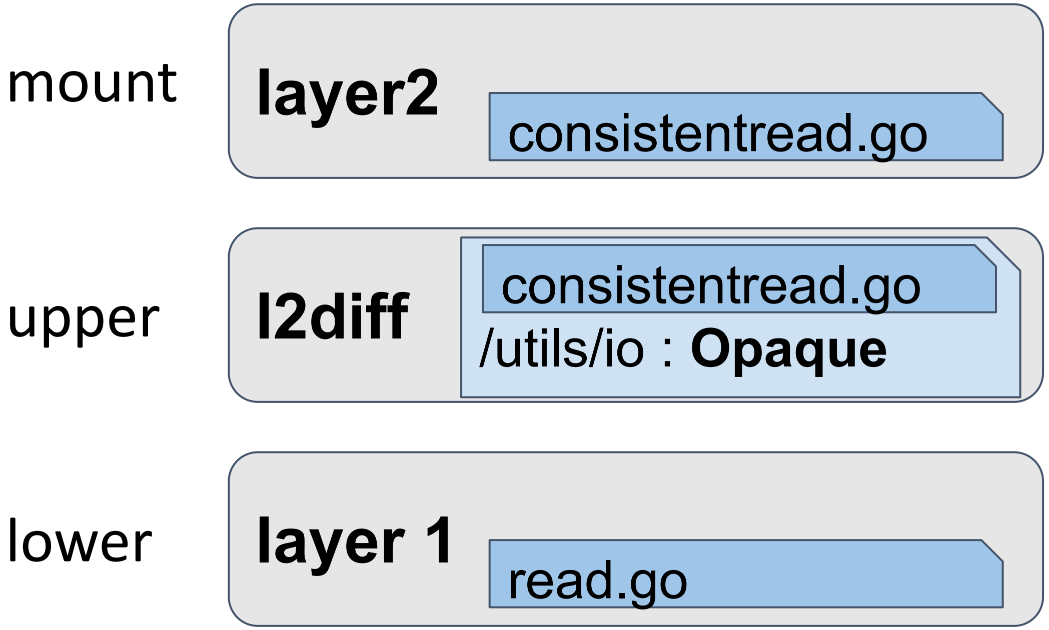


Mount	X^{File}	Y'^{File}		A+A’	A’
Upper		Y'^{File}	RIP	A'^{Dir}	A'^{Opaque} Dir
Lower	X^{File}	Y^{File}	Z^{File}	A^{Dir}	A^{Dir}

Opaque directory: mask directory from lower layers

=> Faster and simpler

Opaque directory



mount

layer2

consistentread.go

upper

l2diff

consistentread.go

/utils/io : **Opaque**

lower

layer 1

read.go

Step #2: git checkout

Opaque directory implementation

Uses directory xattr: **trusted.overlay.opaque**

Opaque directory implementation

Uses directory xattr: **trusted.overlay.opaque**

Can we see it?

```
[in user namespace]

# getfattr -R -d -m "" l2diff/.../utils/io
[nothing]
```

How is this working at step #2?

Opaque directory implementation

Uses directory xattr: **trusted.overlay.opaque**
Can we see it in the initial user namespace?

```
[in initial user namespace]

$ sudo getfattr -R -d -m "" l2diff/.../utils/io
trusted.overlay.opaque="y"
```

This makes more sense

Opaque directory implementation

Uses directory xattr: **trusted.overlay.opaque**
Can we see it in the initial user namespace?

```
[in initial namespace]

$ sudo getfattr -R -d -m "" l2diff/.../utils/io
trusted.overlay.opaque="y"
```

This makes more sense
But we should not be able to write **trusted.*** xattr from user namespace

Mysteries

- trusted.overlay.opaque set despite lacking CAP_SYS_ADMIN
- It seems that the issue
    - does not reproduce when the opaque directory is **“upper”** **(step #2)**
    - does reproduce when the opaque directory is a **“lower”** **(step #3)**

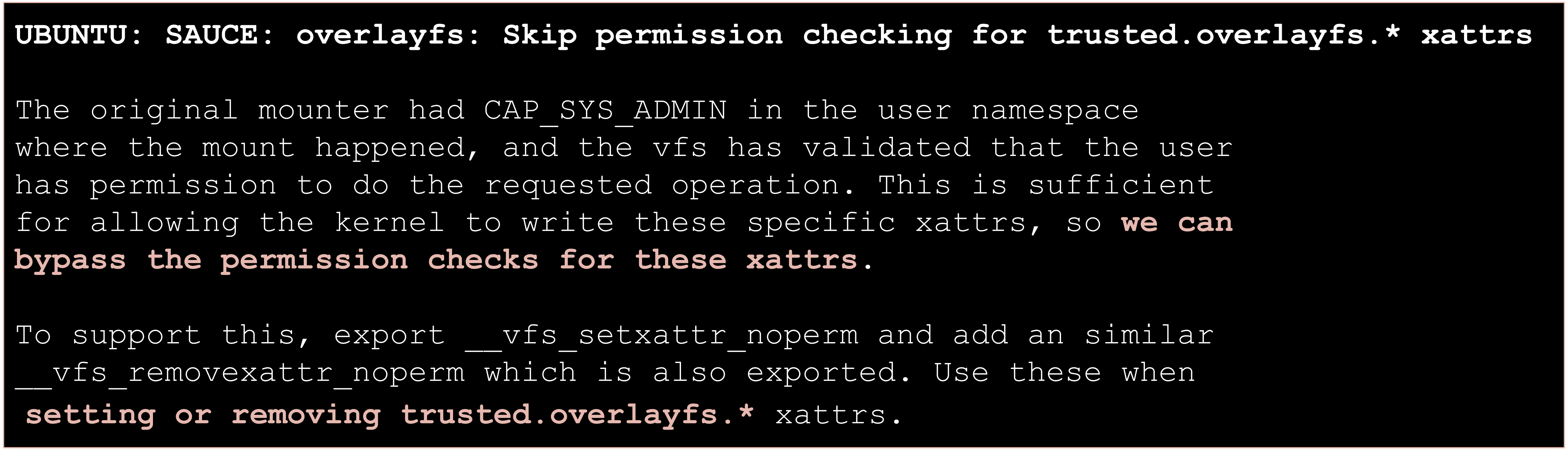
Kernel function tracing

Using ftrace to look at the git checkout operation

```
# trace-cmd record -p function -P <PID> -c -e all

      ovl_check_setxattr
         ...
         __vfs_setxattr_noperm
            __vfs_setxattr
               xattr_resolve_name
               ext4_xattr_trusted_set
               ...
```

vfs_setxattr_noperm

Specific Ubuntu patch to allow the use of overlayfs in user namespaces



UBUNTU: SAUCE: overlayfs: Skip permission checking for trusted.overlayfs.* xattrs

The original mounter had CAP_SYS_ADMIN in the user namespace where the mount happened, and the vfs has validated that the user has permission to do the requested operation. This is sufficient for allowing the kernel to write these specific xattrs, so **we can bypass the permission checks for these xattrs**.

To support this, export __vfs_setxattr_noperm and add an similar __vfs_removexattr_noperm which is also exported. Use these when **setting or removing trusted.overlayfs.*** xattrs.

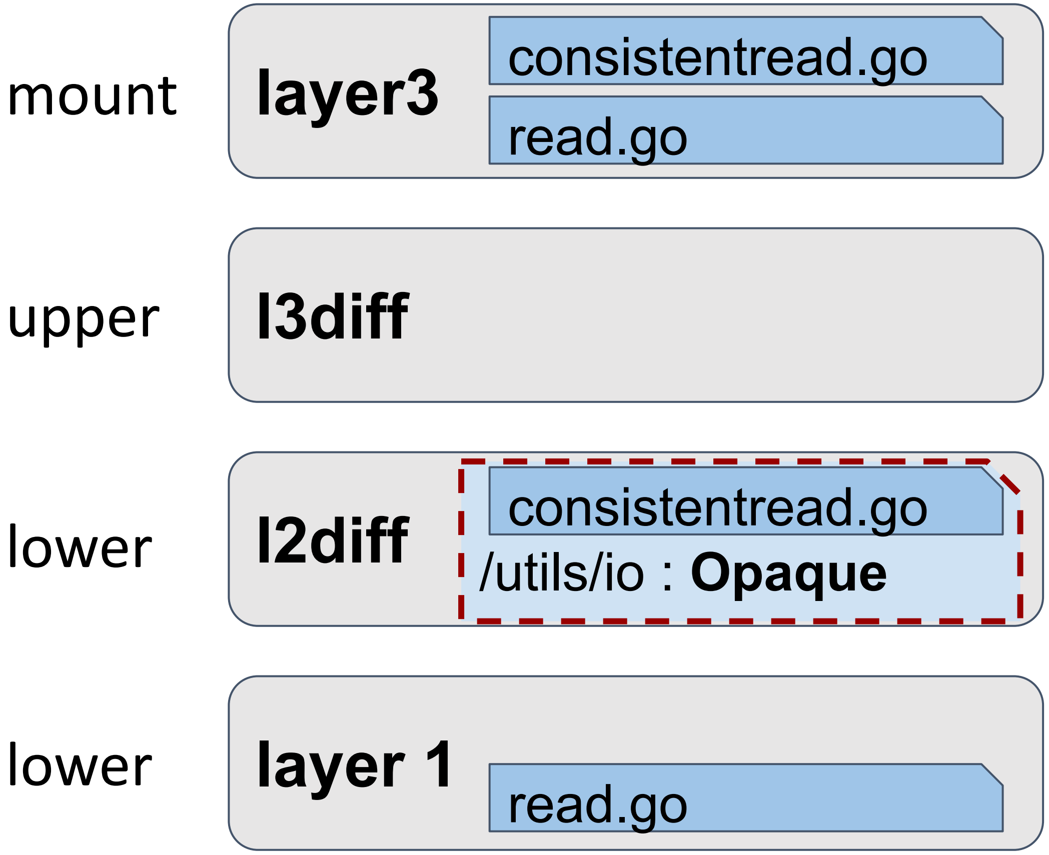
vfs_setxattr_noperm

Specific Ubuntu patch to allow the use of overlayfs in user namespaces

```
UBUNTU: SAUCE: overlayfs: Skip permission checking for trusted.overlayfs.* xattrs

The original mounter had CAP_SYS_ADMIN in the user namespace
where the mount happened, and the vfs has validated that the user
has permission to do the requested operation. This is sufficient
for allowing the kernel to write these specific xattrs, so we can
bypass the permission checks for these xattrs.

To support this, export __vfs_setxattr_noperm and add an similar
__vfs_removexattr_noperm which is also exported. Use these when
 setting or removing trusted.overlayfs.* xattrs.
```

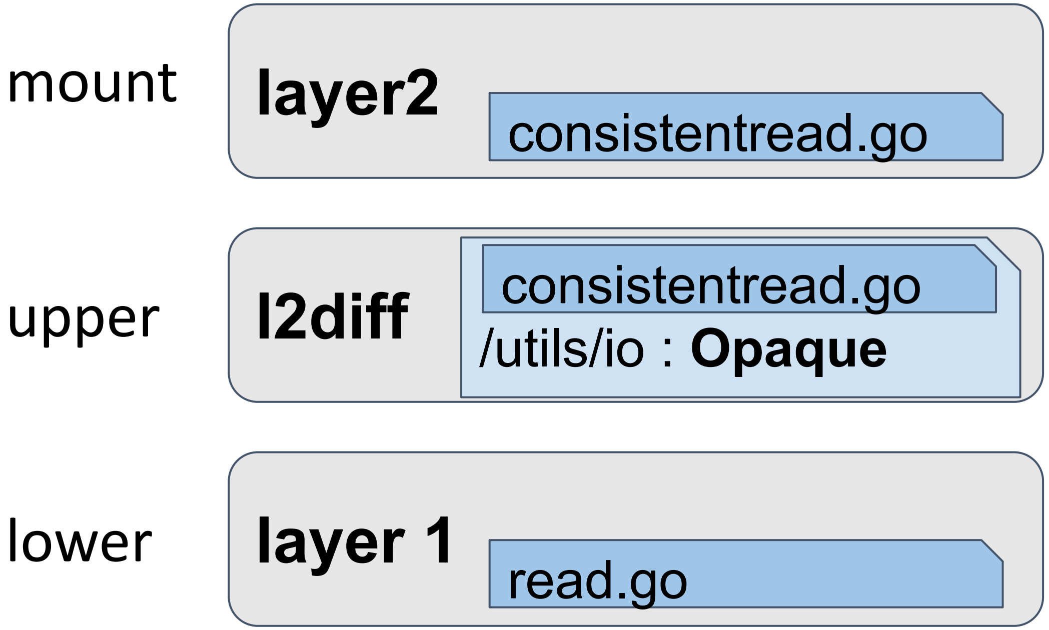
But no equivalent patch for reads!

Back to layers



We mount layer3 in userns
We can't read the Opaque xattr

But wait, it works for step #2



mount

layer2

consistentread.go

upper

l2diff

consistentread.go

/utils/io : **Opaque**

lower

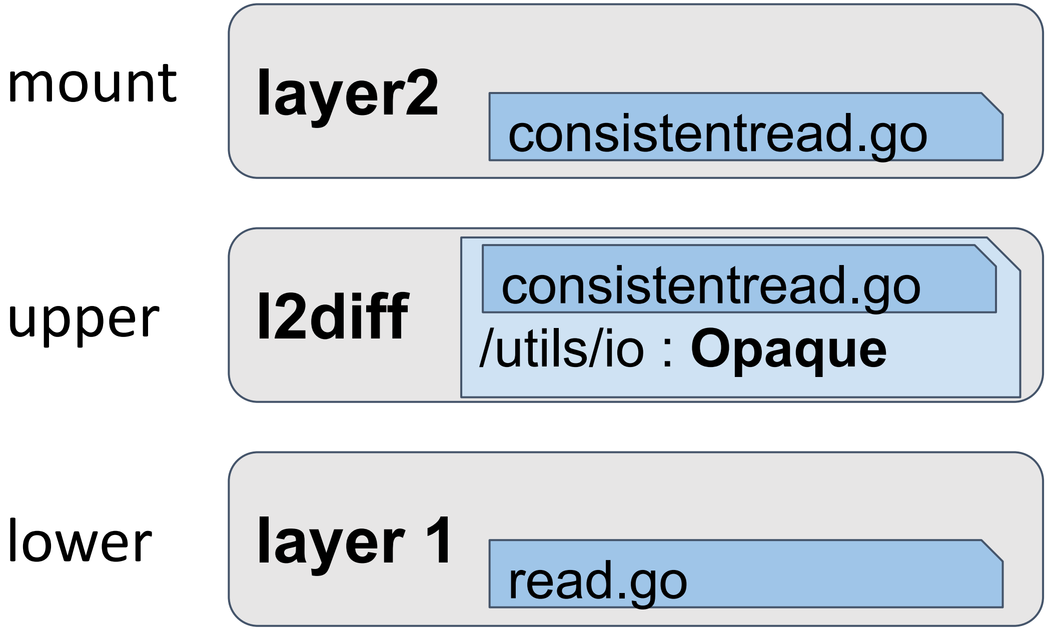
layer 1

read.go

We can write the xattr thanks to the noperm patch

But we should not be able to read it

But wait, it works for step #2



We can write the xattr thanks to the noperm patch

But we should not be able to read it

Filesystems cache heavily, could it be related?

Dropping the Cache

```
$ unshare --mount --user --map-root-user bash
# mkdir layer1 layer2 l2diff work
# git clone <...> layer1

# mount -t overlay overlay \
     -olowerdir=layer1,upperdir=l2diff,workdir=work \
     layer2
# git -C layer2 checkout ...
# ls
consistentread.go
```

Step #1

Step #2

Dropping the Cache

```
$ unshare --mount --user --map-root-user bash
# mkdir layer1 layer2 l2diff work
# git clone <...> layer1

# mount -t overlay overlay \
     -olowerdir=layer1,upperdir=l2diff,workdir=work \
     layer2
# git -C layer2 checkout ...
# ls
consistentread.go
   [In initial user namespace]
   $ sudo /bin/sh -c 'echo 3 > /proc/sys/vm/drop_caches'
# ls
consistentread.go
read.go
```

Step #1

Step #2

Summary

- Ubuntu added a patch to make overlayfs work in user namespaces

- But only patched setxattr and not getxattr

- Thanks to caching, it works until we have to read from disk

=> Leads to *interesting* behaviors

Fix

- New overlayfs mount option “userxattr” in kernel 5.11
  - "user.overlay.*" xattr namespace

- buildkit overlay implementation adds userxattr when available

=> Upgrading Kubernetes nodes to 5.11 just worked!

Details: https://github.com/moby/buildkit/issues/2381

Results!

Image Builds on Kube

- Starting with a mono-repo cleared most issues upfront

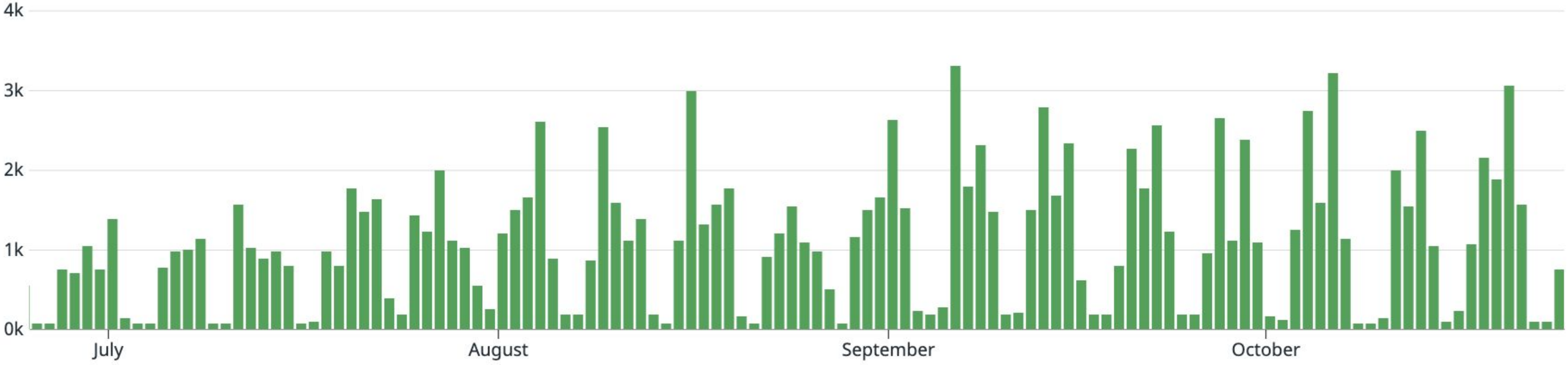
- Decommissioned dedicated Docker runners
  - Easier node-lifecycle management and tuning

- Native multi-arch builds
  - Emulation is (of course) simply too slow

Image Builds on Kube

- Several hundred distinct images now built on kube

- Handling >1k builds per day reliably



Takeaways

- Buildkit is great
    - Remote builds
    - Multiarch images

- Rootless Buildkit on Kube is bleeding edge
    - Overlayfs work fine in user namespaces with 5.11+
    - No great solution for process sandboxing
        - Maybe with "ProcMount: Unmasked"?

- After the initial hurdles it was really worth the effort for us!

We have many people to thank!

- Mayeul Blanzat for moving image builds to Kubernetes at Datadog
- Tõnis Tiigi and Akihiro Suda for
  - Making rootless buildkit possible
  - Answering our *many* questions
- Jess Frazelle and Alban Crequy for helping us make sense of procfs

Thank you

We’re hiring!
https://www.datadoghq.com/careers/

eric.mountain@datadoghq.com
laurent@datadoghq.com

Please scan the QR Code above to leave feedback on this session

🐦 **@ericmountain**
🐦 **@lbernail**