Rethinking Security from Scratch

The Case for Shifting Container Security from the Edge to the Core

Chris Milsted

Staff Field Engineer

cmilsted@vmware.com

+44 7795 316 723

Agenda

$ whoami

Refresher - OCI image specification

Options - Distroless and Scratch

Demo - Building from scratch

Looking at our current security model

Shifting towards “build-based” security

$ whoami

$ whoami

- Chris is based in the UK and is a Staff Field Engineer for VMware working in the Cloud Native team. He spends most of his work wrangling Kubernetes and most of his spare time playing field hockey badly and being a taxi driver for two children who are growing up rapidly.

- He has spent the last 5 years working with containers and kubernetes and Linux

- Most of this has been with large Financial Services Customers

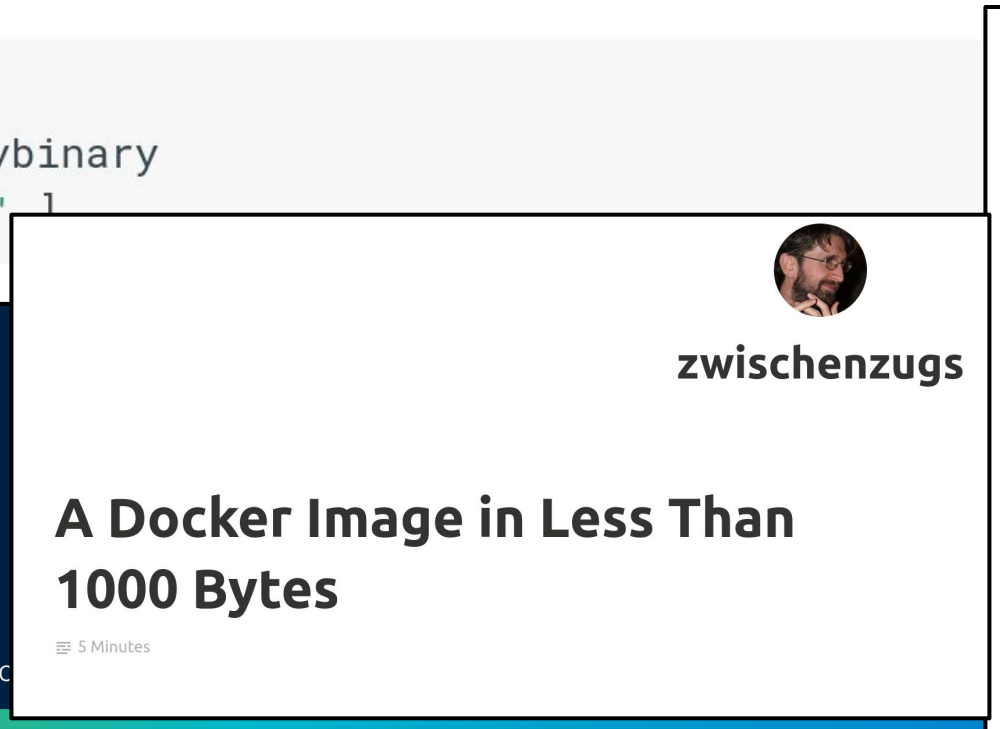
What inspired this talk

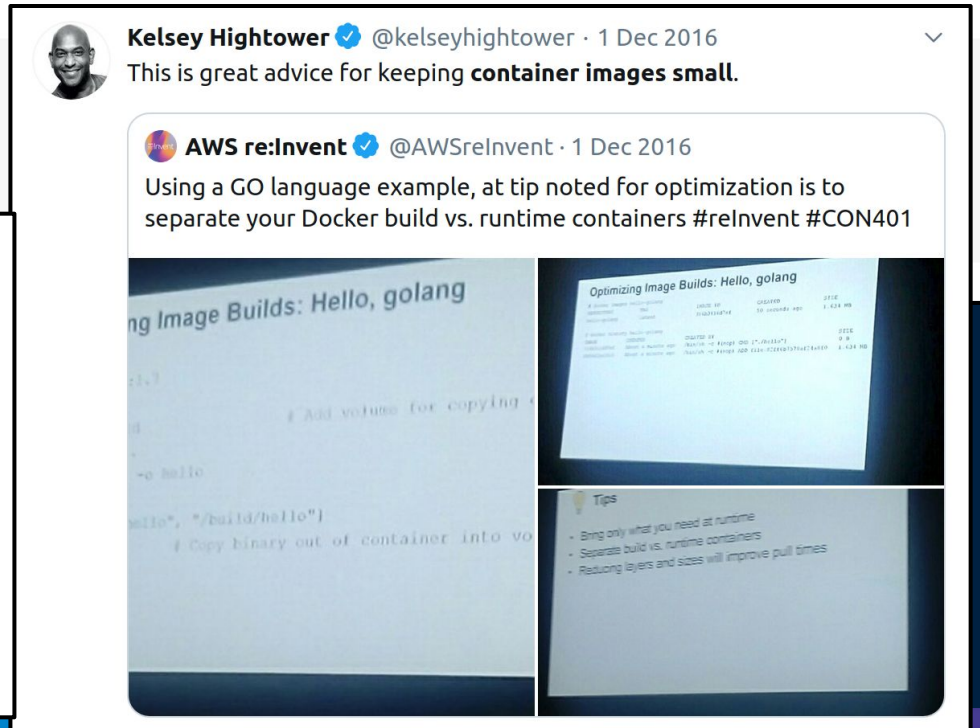
Use the smallest base image possible

The base image is the one referenced in the `FROM` instruction in your Dockerfile. Every other instruction in the Dockerfile builds on top of this image. The smaller the base image, the smaller the resulting image is, and the more quickly it can be downloaded. For example, the alpine:3.7 image is 71 MB smaller than the centos:7 image.

You can even use the scratch base image, which is an empty image on which you can build your own runtime environment. If your app is a statically linked binary, it's easy to use the scratch base image:

```
FROM scratch
COPY mybinary /mybinary
CMD [ "/mybinary" ]
```

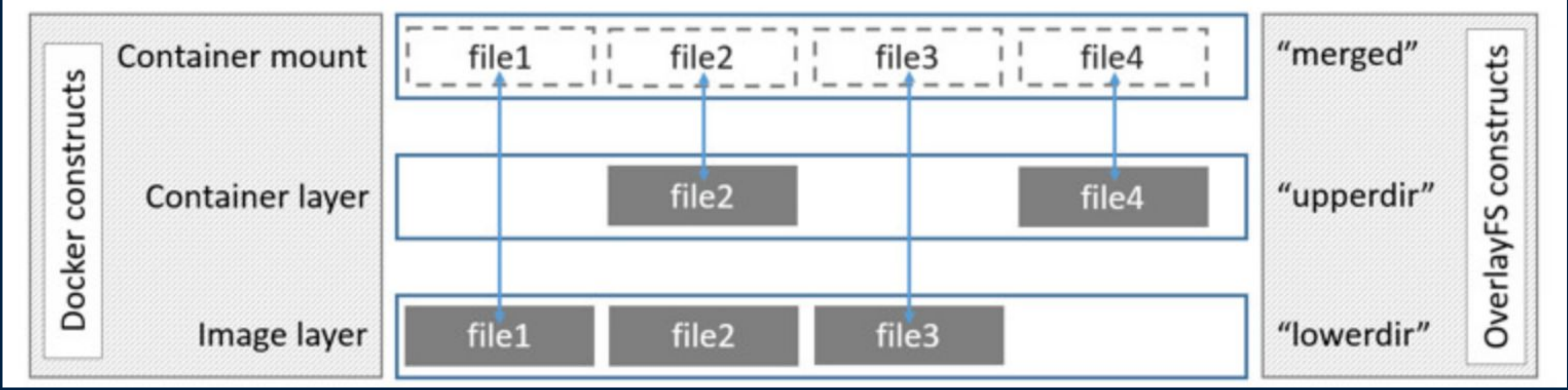




Refresher - OCI image specification

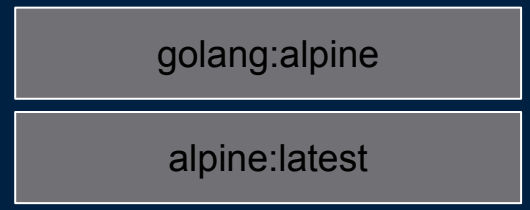
What is a container

Overlay2 graph driver



OCI images

Building container images and inheritance




```
$ sudo docker inspect --format='{{.GraphDriver.Data}}' alpine:latest |tr ' ' '\n'
map[MergedDir:/var/lib/docker/overlay2/784af3f8492d8d7ade0a82bbaa6dace2bd694d4c0f1a4ab1510cd43cec0c67d9/merged
UpperDir:/var/lib/docker/overlay2/784af3f8492d8d7ade0a82bbaa6dace2bd694d4c0f1a4ab1510cd43cec0c67d9/diff
WorkDir:/var/lib/docker/overlay2/784af3f8492d8d7ade0a82bbaa6dace2bd694d4c0f1a4ab1510cd43cec0c67d9/work]
```

```
$ sudo docker inspect --format='{{.GraphDriver.Data}}' golang:alpine |tr ' ' '\n'
map[LowerDir:/var/lib/docker/overlay2/c34762b4720a23034e53c9f35be009071921086ca140db65d13a82940b9ebf35/diff:/var/lib/docker/overlay2/9988592682bef4c04e144f6d153b9116686e8c0a879e71e41cfc79e07037a19d/diff:/var/lib/docker/overlay2/96f7522bc8c562a266b524645dd43931d6bdf37560f940fe50cb85177f08fe02/diff:/var/lib/docker/overlay2/784af3f8492d8d7ade0a82bbaa6dace2bd694d4c0f1a4ab1510cd43cec0c67d9/diff
MergedDir:/var/lib/docker/overlay2/71495101f982ea3ee928f5c00798c7ee6b6ae21378d40f252e9283b856c792d6/merged
UpperDir:/var/lib/docker/overlay2/71495101f982ea3ee928f5c00798c7ee6b6ae21378d40f252e9283b856c792d6/diff
WorkDir:/var/lib/docker/overlay2/71495101f982ea3ee928f5c00798c7ee6b6ae21378d40f252e9283b856c792d6/work]
```

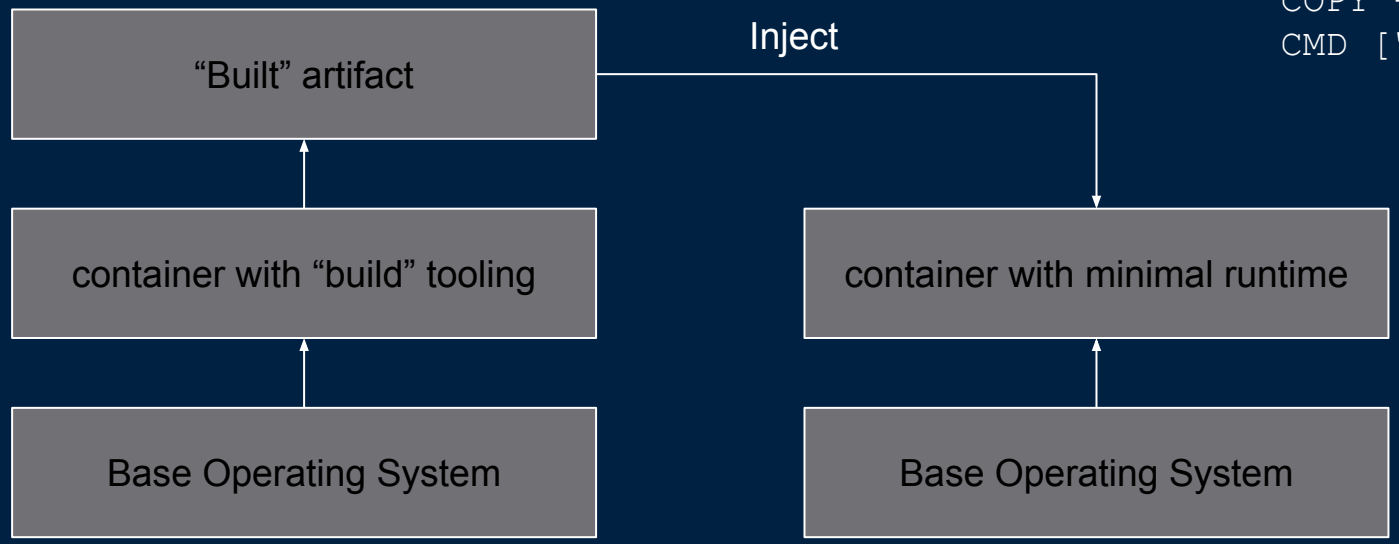
https://github.com/opencontainers/image-spec/blob/master/layer.md

Options - Distroless and Scratch

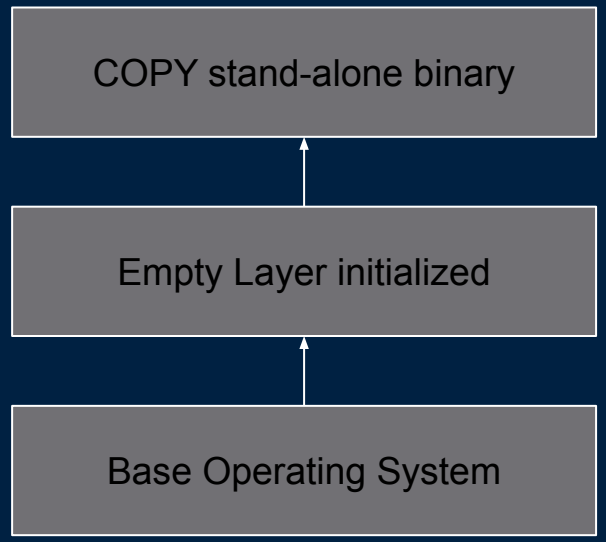
Distroless

Minimising container but still based on distro userspace

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/<pathtocode>
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a
-installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder ./app .
CMD ["./app"]
```



Scratch



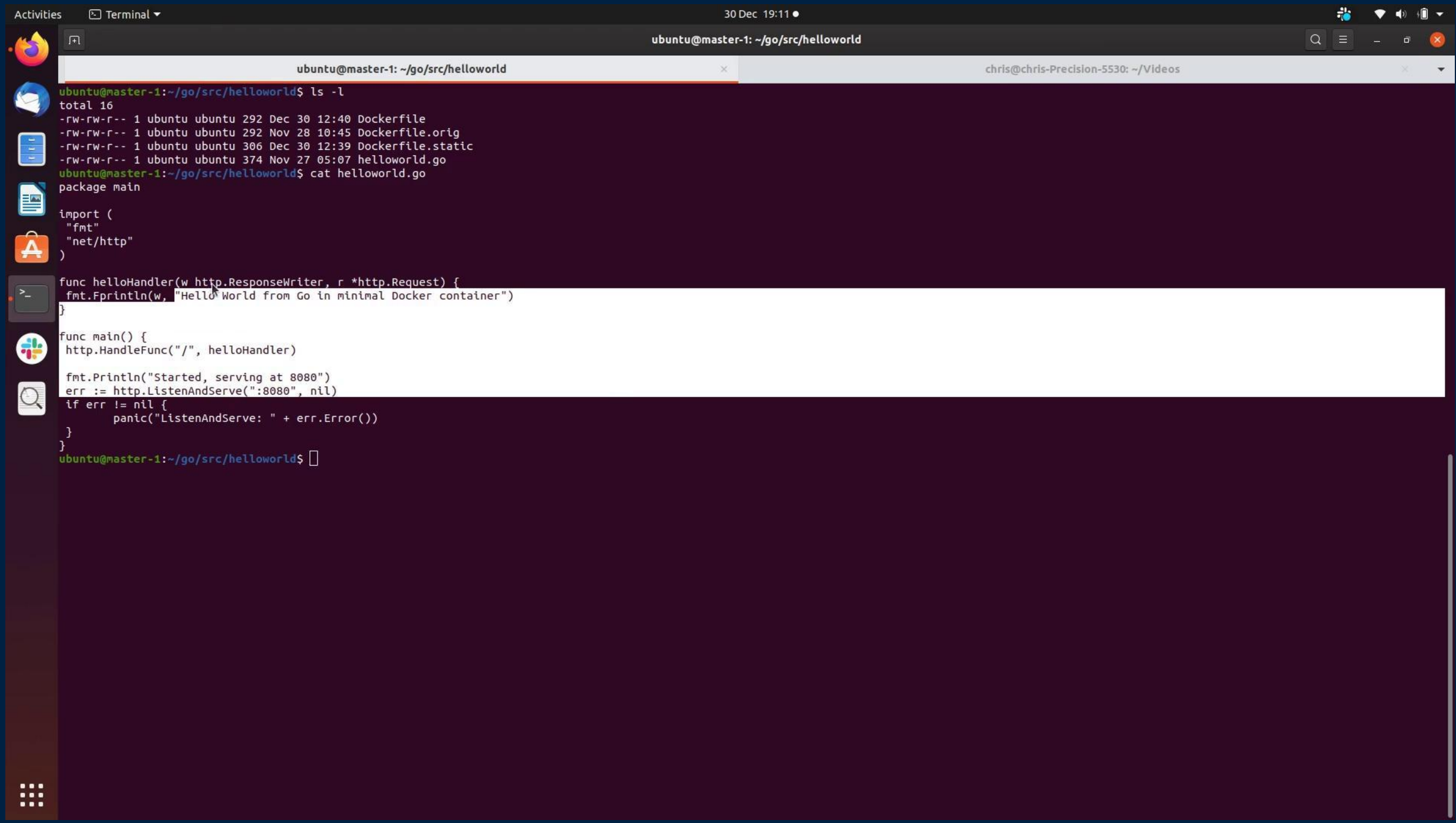

```
FROM scratch

# List the maintainer
LABEL maintainer="Chris Milsted"

# Copy the Pre-built binary file from the
previous stage.
COPY  ./helloworld .

# Expose port 8080 to the outside world
EXPOSE 8080

#Command to run the executable
CMD ["./helloworld"]
```

Layer Isolation

An application fully isolated in a container must be self-contained

- Namespaces
- CGroups
- Kernel capabilities (seccomp profile)
- Linux Security Module (e.g. SELinux or apparmour)

Demo - Building from scratch

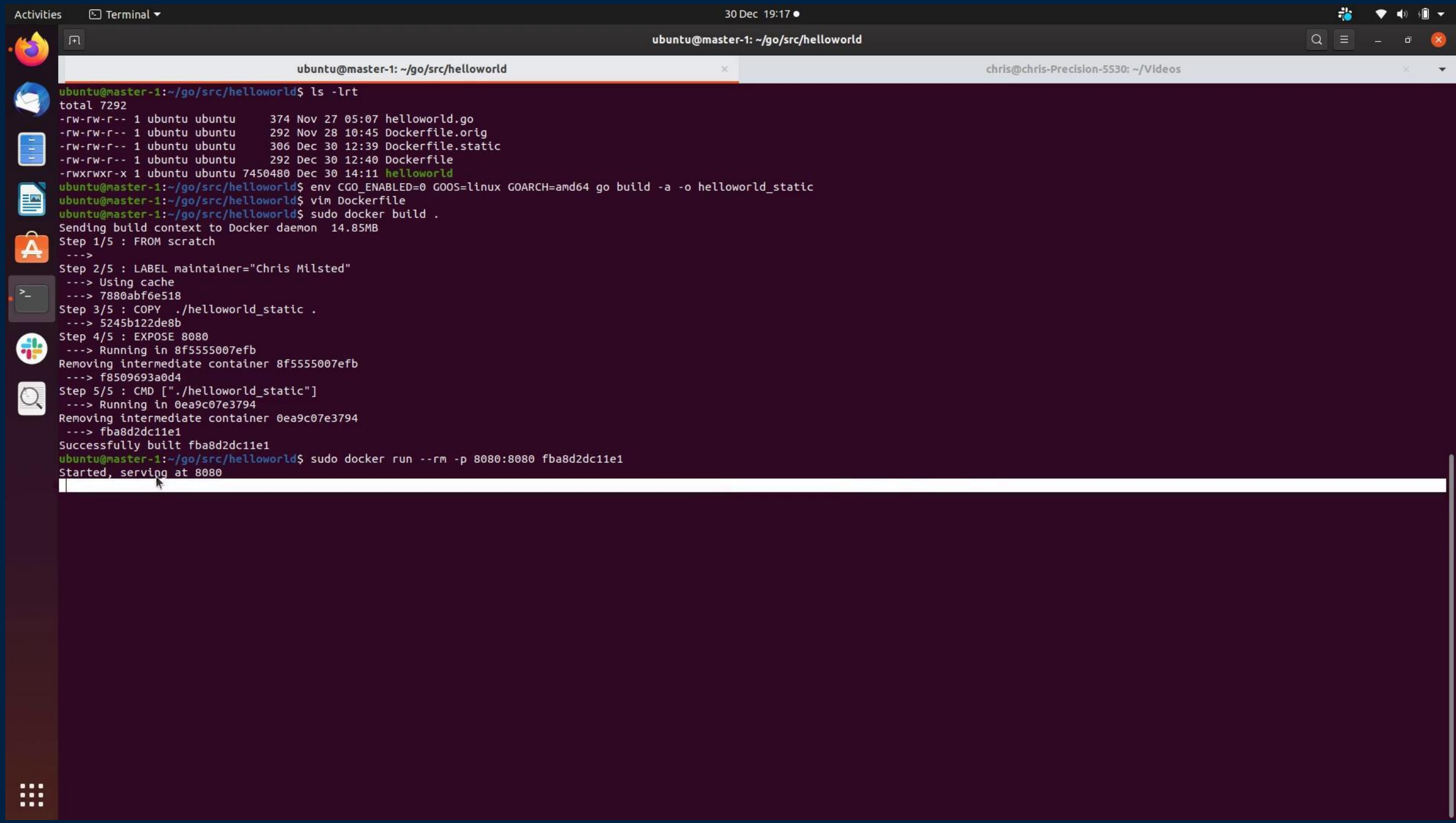

Activities
Terminal
30 Dec 19:11
ubuntu@master-1: ~/go/src/helloworld
ubuntu@master-1: ~/go/src/helloworld
chris@chris-Precision-5530: ~/Videos
ubuntu@master-1:~/go/src/helloworld$ ls -l
total 16
-rw-rw-r-- 1 ubuntu ubuntu 292 Dec 30 12:40 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu 292 Nov 28 10:45 Dockerfile.orig
-rw-rw-r-- 1 ubuntu ubuntu 306 Dec 30 12:39 Dockerfile.static
-rw-rw-r-- 1 ubuntu ubuntu 374 Nov 27 05:07 helloworld.go
ubuntu@master-1:~/go/src/helloworld$ cat helloworld.go
package main
import (
"fmt"
"net/http"
)
func helloHandler(w http.ResponseWriter, r *http.Request) {
fmt.Fprintln(w, "Hello World from Go in minimal Docker container")
}
func main() {
http.HandleFunc("/", helloHandler)
fmt.Println("Started, serving at 8080")
err := http.ListenAndServe(":8080", nil)
if err != nil {
panic("ListenAndServe: " + err.Error())
}
}
ubuntu@master-1:~/go/src/helloworld$

```
ubuntu@master-1:~/go/src/helloworld$ ldd helloworld
 linux-vdso.so.1 (0x00007ffcecff4000)
 libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f39aa77f000)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f39aa38e000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f39aa99e000)
```

Typically, compiled Go binaries have very few dependencies. You do not need any kind of runtime libraries or VMs, and all Go libraries that you use in your project are embedded directly into the resulting executable file. However, if you compile your application in Linux, the Go compiler will link the resulting binary against a few C standard libraries that are typically available on any Linux system. If you are on Linux, you can easily find out against which libraries your program was linked by invoking the `ldd` binary with one of your compiled Go binaries as argument. If your binary is linked against the C standard library, you will receive the following output:

```
$ ldd ./eventservice
    linux-vdso.so.1 (0x00007ffed09b1000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007fd523c36000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd52388b000)
    /lib64/ld-linux-x86-64.so.2 (0x0000564d70338000)
```

This means that your Go application actually needs these Linux libraries to run and that you cannot just arbitrarily delete them from your image to make it smaller.


Activities
Terminal
30 Dec 19:17
ubuntu@master-1: ~/go/src/helloworld
ubuntu@master-1: ~/go/src/helloworld
chris@chris-Precision-5530: ~/Videos
ubuntu@master-1:~/go/src/helloworld$ ls -lrt
total 7292
-rw-rw-r-- 1 ubuntu ubuntu   374 Nov 27 05:07 helloworld.go
-rw-rw-r-- 1 ubuntu ubuntu   292 Nov 28 10:45 Dockerfile.orig
-rw-rw-r-- 1 ubuntu ubuntu   306 Dec 30 12:39 Dockerfile.static
-rw-rw-r-- 1 ubuntu ubuntu   292 Dec 30 12:40 Dockerfile
-rwxrwxr-x 1 ubuntu ubuntu 7450480 Dec 30 14:11 helloworld
ubuntu@master-1:~/go/src/helloworld$ env CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -a -o helloworld_static
ubuntu@master-1:~/go/src/helloworld$ vim Dockerfile
ubuntu@master-1:~/go/src/helloworld$ sudo docker build .
Sending build context to Docker daemon 14.85MB
Step 1/5 : FROM scratch
 --->
Step 2/5 : LABEL maintainer="Chris Milsted"
 ---> Using cache
 ---> 7880abf6e518
Step 3/5 : COPY ./helloworld_static .
 ---> 5245b122de8b
Step 4/5 : EXPOSE 8080
 ---> Running in 8f5555007efb
Removing intermediate container 8f5555007efb
 ---> f8509693a0d4
Step 5/5 : CMD ["./helloworld_static"]
 ---> Running in 0ea9c07e3794
Removing intermediate container 0ea9c07e3794
 ---> fba8d2dc11e1
Successfully built fba8d2dc11e1
ubuntu@master-1:~/go/src/helloworld$ sudo docker run --rm -p 8080:8080 fba8d2dc11e1
Started, serving at 8080

Success?

A very very minimal container - which just contains a single executable

```
REPOSITORY                                  TAG                 IMAGE ID            CREATED             SIZE
<none>                                      <none>              fba8d2dc11e1        3 minutes ago       7.39MB
<none>                                      <none>              92769f32e1f8        41 minutes ago      7.45MB
```

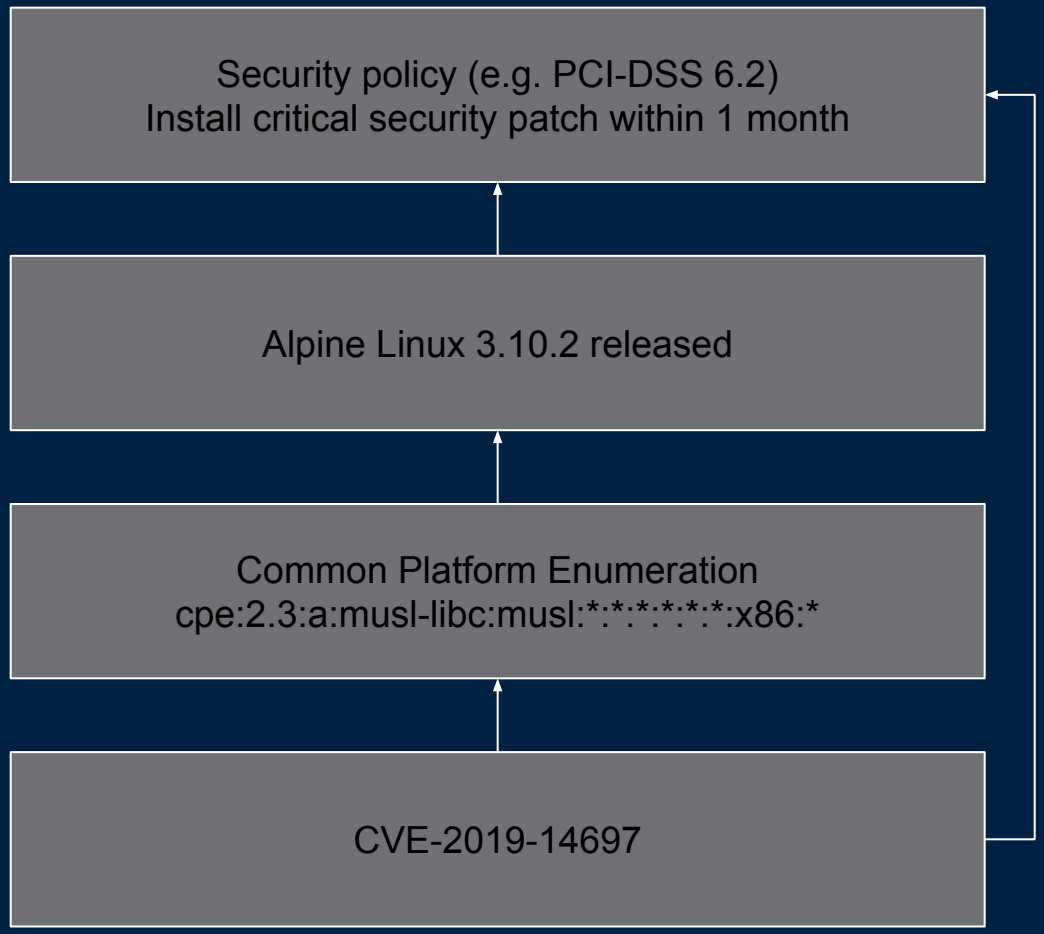
but…

```
ubuntu@master-1:~/go/src/helloworld$ sudo docker run --rm -p 8080:8080 fba8d2dc11e1 sh
docker: Error response from daemon: OCI runtime create failed: container_linux.go:346: starting container process caused "exec: \"sh\": executable file not found in $PATH": unknown.
ERRO[0001] error waiting for container: context canceled
```

What is inside our container when we scale this approach...

```
(gdb) p 'runtime.buildVersion'
$1 = 0x700d3a "go1.13.4"
```

Looking at our current security model



Process maps CVE and code change to package version and then onto higher level component such as Operating System.

Shifting towards “build-based” security

A quest for minimal build (technology) has impacted a process

Today we noticed that another Go team at CloudFlare, the Data team, had **a much smarter way to bake version numbers into binaries using the `-X` linker option**.

The `-X` Go linker option, which you can set with `-ldflags`, sets the value of a string variable in the Go program being linked. You use it like this: `-X main.version 1.0.0`.

Content-based non-linear versioning

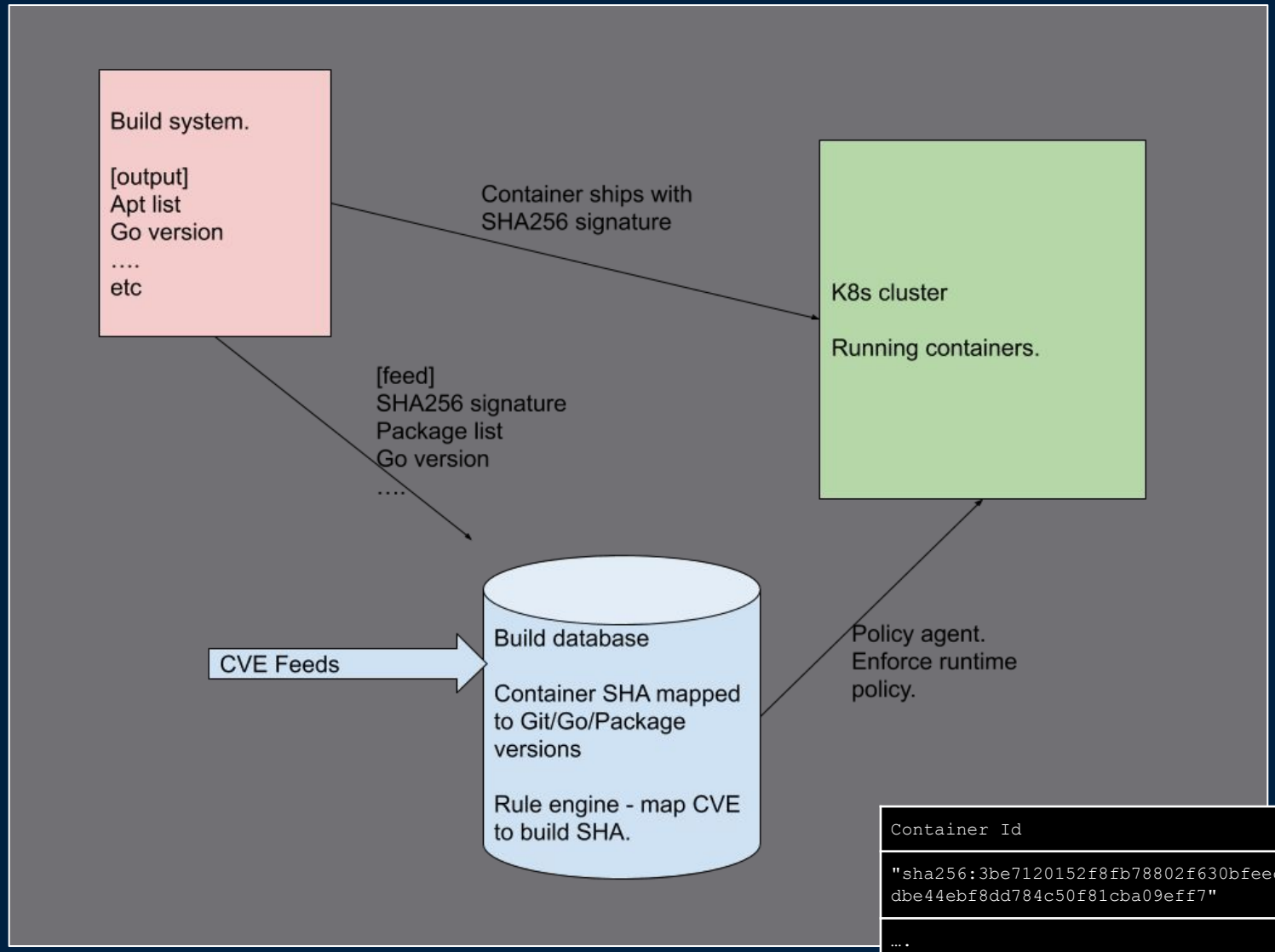
The very fundamental foundation of everything at Polyverse is content-based versioning. *A content-based version is a cryptographically secure hash over a set of bits, and that hash is used as the "version" for those bits.*

This is a premise you will find everywhere in our processes. If you want to tell your colleague what version of Router you want, you'd say something like: router@72e5e550d1835013832f64597cb1368b7155bd53. That is the version of the router you're addressing. It is unambiguous, and you can go to the Git repository that holds our router, and get PRECISELY what your colleague is using

https://blog.cloudflare.com/setting-go-variables-at-compile-time/
https://polyverse.com/blog/semantic-versioning-has-failed-agile-49bbd86d817/

What does the future hold?

We trust the build system, not the edge container



Container Id	Go	Git tag	libc
"sha256:3be7120152f8fb78802f630bfeec68af821dbe44ebf8dd784c50f81cba09eff7"	go1.13.4 linux/amd64	6a615ca	2.27-3ubuntu1
….			

Questions?

Thank You