Hardening Docker Containers

A reference for the workshop and for the discussion afterward

It is written for a mixed room. If you have never run a container, read it straight through; the terms are defined as they come up, and there is a glossary at the end. If you use Docker every day, skim the definitions and go to the commands.

The goal is not to hand you a list of flags to copy. It is to explain what a container actually is, so that each hardening step makes sense as a consequence of how the system works rather than as a rule you have to trust.

What a Container Actually Is

A common mental picture is that a container is a small virtual machine. It is not. A virtual machine boots its own kernel on emulated hardware. A container is just one or more normal processes running on the host's own kernel, with the kernel lying to those processes about what they can see and how much they can use. There is no second kernel and no hardware emulation. That single fact drives everything else in this document.

The lying is done with three kernel features. It is worth knowing their names because the rest of the packet refers back to them.

- **Namespaces** control what a process can *see*. A namespace gives a process its own private view of one kind of system resource. There are several kinds. The PID namespace gives the container its own process numbering, so the process inside thinks it is PID 1 and cannot see the host's processes. The mount namespace gives it its own filesystem view. The network namespace gives it its own network interfaces and port space. There are a few others (hostname, inter-process communication, and user). A container is really just a process placed inside a set of these namespaces. Isolation means the process can only see what its namespaces choose to show it.
- **Control groups**, usually written cgroups, control how *much* a process can use. A cgroup caps and accounts for resources: memory, CPU time, the number of process IDs, disk and network throughput. When you set a memory limit on a container later in this document, you are writing to a cgroup.
- **Capabilities** control what a privileged process is *allowed to do*. Traditional Unix had two classes of user: root, who could do anything, and everyone else. Linux broke root's powers into around forty separate capabilities, each covering one kind of privileged action. `CAP_NET_BIND_SERVICE` lets a process bind low-numbered

ports. `CAP_CHOWN` lets it change file ownership. `CAP_SYS_ADMIN` is a large catch-all that is close to full root. Docker gives a new container a subset of these by default and lets you drop or add individual ones.

The kernel underneath all of this is shared with the host and with every other container on the machine. That is the reason container security is a real subject. If a process breaks out of its namespaces, or exploits a bug in the kernel, or is simply handed too much privilege to begin with, it is now acting on the same kernel your host runs on. So the whole job is to narrow what a container can see, use, and do, so that a compromise of the application inside stays contained.

One more piece of vocabulary before the practical part, because it explains why "root in a container" is dangerous. Every Linux user has a numeric user ID, and root is UID 0. By default the container's UID 0 is the same UID 0 as the host's. Docker does remove several of the most dangerous capabilities from that user, so container root is not quite the same as host root out of the box, but it is close enough that you should not rely on the gap. There is a feature called user-namespace remapping (`userns-remap`) that maps container UID 0 to an ordinary unprivileged UID on the host, which closes most of that gap, but it is off by default.

Quick demonstration that the container's root really is the host's UID 0

```
docker run --rm alpine id
# uid=0(root) gid=0(root)

mkdir -p /tmp/demo
docker run --rm -v /tmp/demo:/data alpine sh -c 'id > /data/x'
ls -l /tmp/demo/x
# the file is owned by root on your host, written by a process you thought was sandboxed
```

Images, Layers, and the Daemon

Two more ideas make sense of the build sections that follow.

An **image** is a read-only template: a snapshot of a filesystem plus some metadata that says which command to run and with what settings. A **container** is a running instance of an image, with a thin writable scratch layer added on top. One image can start many containers. You build an image from a **Dockerfile**, which is a recipe of instructions.

Images are built in **layers**. Each instruction in a Dockerfile that changes the filesystem produces one layer, and the layers are stacked and presented to the container as a single filesystem. This has three consequences that matter for security:

- **Caching:** Layers are cached, which is why ordering instructions well makes rebuilds fast.
- **Sharing:** Layers are shared between images, which saves space.
- **Immutability:** Layers are immutable once built. If you add a secret in one layer and delete it in a later layer, the secret is still sitting in the earlier layer, and anyone with the image can read it. Deleting is not removing.

Finally, the daemon. When you type a `docker` command, the command-line tool is a thin client. It sends your request to a background service called `dockerd`, the Docker daemon, which runs as root and does the actual work of building and running containers. The client and daemon talk over a socket at `/var/run/docker.sock`. Hold on to that fact; it is the reason one of the worst mistakes later in this document is as bad as it is.

With that, the rest is practical.

Before We Start

Image downloads are slow on conference wifi, so start these now while we talk:

```
docker version && docker run --rm hello-world
docker pull python:3.12-slim
docker scout version       # comes with Docker Desktop; otherwise install trivy
brew install mintoolkit    # image slimming tool; check the installed binary name (section 5)
```

1 Start from a Safe Image

A **base image** is the image your own image is built on top of, named in the first `FROM` line of your Dockerfile. Everything in that base becomes part of your image, including its shell, its package manager, and any libraries it ships, whether your app uses them or not. Every one of those files is code that could contain a vulnerability, so the base image is the first security decision you make.

Ask four things about any base image: where it came from, how much is in it, whether it is pinned to a fixed version, and whether it has known vulnerabilities today.

- **Prefer a small base.** A full `python` image is about a gigabyte and includes a shell and package managers your app never calls. The `python:3.12-slim` variant is a fraction of that. A **distroless** image goes further: it contains your language runtime and your app and essentially nothing else, no shell and no package manager, which removes most of the tools an attacker would want to use after breaking in. Less in the image also means fewer known vulnerabilities to track and patch.
- **Pin the tag and the digest.** A tag such as `python:3.12-slim` is a human-friendly label, and it is mutable: the same tag can point at a new image tomorrow. A **digest** is a cryptographic hash of the exact image contents, written `sha256:...`, and it cannot change. Pinning the digest gives you a reproducible build and means a tampered or unexpectedly updated image will be rejected rather than silently used.

Get the digest of an image you have pulled:

```
docker inspect --format='{{index .RepoDigests 0}}' python:3.12-slim
```

That field is only filled in for images you pulled from a registry, so it will be empty for an image you built locally or have not pulled yet. In that case, read the digest from the registry directly with `docker buildx imagetools inspect IMAGE` or `crane digest IMAGE`. A **registry** is the server that stores and hands out images, such as Docker Hub or GitHub's registry; pulling and pushing are how images move between it and your machine.

Use the digest in your Dockerfile:

```
FROM python:3.12-slim@sha256:PASTE_DIGEST_HERE
```

Then scan before you trust it. A **scanner** reads the packages inside an image and looks each one up against public vulnerability databases. A **CVE** (Common Vulnerabilities and Exposures) is the standard public identifier for one known flaw, so a scan report is largely a list of CVEs and the packages they affect. Run a scan on both the slim and the full image and compare:

```
docker scout cves python:3.12-slim
docker scout quickview python:3.12
# or: trivy image python:3.12-slim
```

Two related tools are worth knowing exist:

- `syft IMAGE` produces a **software bill of materials** (SBOM), which is a full inventory of the packages and versions in an image; it is what lets you answer "are we affected?" quickly the next time a major CVE is announced.
- `cosign`, part of the sigstore project, is how you verify that an image was actually published by who you think, using a cryptographic signature.

2 Add Your Code Without Leaking Anything

Here is the sample app. `app.py` is a small Flask web service that returns `{"status":"ok"}`, and `requirements.txt` contains one line: `flask==3.0.3`.

This is the Dockerfile many of us wrote years ago. Every line has a problem:

```
FROM python:latest           # unpinned and huge
WORKDIR /app
COPY . .                     # copies .git, env, keys, everything
RUN pip install -r requirements.txt
CMD ["python", "app.py"]     # runs as root
```

To see why the `COPY . .` line is dangerous, you need one term: the **build context** is the set of files the client sends to the daemon when you run a build, which is normally your whole current directory. The `COPY` instruction can only copy from that context, and `COPY . .` copies all of it into the image. That means your `.git` directory with its full history, any local `.env` file, and any stray keys go straight into the image, and because of the layer

rule from earlier, they stay there even if a later line deletes them.

Fix it in three steps.

Step 1: Add a .dockerignore file

It works like `.gitignore` but for the build context: files listed here are never sent to the daemon and so can never be copied into the image. It is the single most forgotten security control in Docker.

```
.git
.env
*.env
.venv
venv/
__pycache__/
*.pyc
.ssh/
.aws/
```

Step 2: Copy selectively

Copy only the files you need, and copy the requirements before the application code. Because layers are cached in order, putting the rarely-changing dependency install before the frequently-changing code means editing your app does not force a slow reinstall on every build.

Step 3: Use a multi-stage build with a non-root user

A **multi-stage build** uses more than one `FROM` line. Earlier stages can carry heavy build tools, and the final image copies only the finished artifacts out of them, so the tools themselves never ship. Here the first stage builds a Python virtual environment (a self-contained directory holding the installed dependencies), and the final stage copies just that directory:

```
# syntax=docker/dockerfile:1
FROM python:3.12-slim@sha256:DIGEST AS build
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.12-slim@sha256:DIGEST
RUN useradd -u 10001 app
COPY --from=build /opt/venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
WORKDIR /app
COPY --chown=app:app app.py .
USER 10001
CMD ["python", "app.py"]
```

The build stage has `pip` and any compilers; the final image gets only the finished virtual environment, so the toolchain never ships. The `USER 10001` line is the important one for this section: it sets the default user for the running container to an ordinary account instead of root, which is the single change that does the most to limit damage if the app is compromised. Build it and check:

```
docker build -t myapp:2.0 .
docker run --rm myapp:2.0 id
# no longer uid=0
```

One note on secrets that the app needs at build time, such as a token to fetch a private package. Do not pass them with `ARG` or `ENV`, because those values are recorded in the image layers and `docker history` will show them even if a later line removes the file. **BuildKit**, the modern Docker build engine, has a dedicated mechanism that mounts a secret only for the duration of one instruction and never writes it to a layer:

```
RUN --mount=type=secret,id=token sh -c 'use $(cat /run/secrets/token)'
```

> **Runtime** secrets, such as a database password, are a separate problem: inject those when you start the container, never bake them into the image.

3 Run with the Fewest Privileges That Still Work

Sections 1 and 2 hardened the image. This section is about how you *run* it, and it is where the kernel features from the opening pay off. A perfectly clean image, run as root with every capability and no limits, is still a serious risk. The goal is to let the container do what the application needs and nothing more. One run command applies the main controls:

```
docker run --rm \
  --user 10001:10001 \
  --cap-drop ALL \
  --security-opt=no-new-privileges \
  --read-only --tmpfs /tmp \
  --memory=256m --cpus=0.5 --pids-limit=100 \
  -p 5000:5000 myapp:2.0
```

Flag breakdown

- `--user 10001:10001` — Runs the process as an ordinary user ID rather than root, so a break-in starts with far fewer powers. This is the runtime counterpart to the `USER` line in the Dockerfile, and setting it here means it holds even if the image forgot to.
- `--cap-drop ALL` — Removes every one of the Linux capabilities described at the start. Most applications need none of them. If you find something genuinely requires one, add just that one back with `--cap-add`.

Starting from zero and adding back is the safe direction; starting from the default set and hoping is not.

- `--security-opt=no-new-privileges` — Tells the kernel that this process and its children can never gain more privileges than they start with. It closes a classic escalation path where a program marked with the setuid bit runs as a more powerful user. It costs nothing and is worth setting everywhere.
- `--read-only` and `--tmpfs /tmp` — `--read-only` makes the container's filesystem read-only, and `--tmpfs /tmp` gives back a small writable area in memory for temporary files. With a read-only filesystem, an attacker who gets code execution cannot drop a web shell, cannot overwrite your program, and cannot leave anything behind that survives a restart. You find the few paths that genuinely need to be writable by running read-only and seeing what fails.
- `--memory=256m`, `--cpus=0.5`, `--pids-limit=100` — These last three flags are cgroup limits. They stop a single container from starving the host, whether from a bug or a crypto-miner an attacker installed. `--pids-limit=100` caps how many processes it can create, which stops a fork bomb. These are security controls as much as operational ones.

One thing that trips people up: an ordinary non-root user cannot bind to network ports below 1024, which is a kernel rule. That is why the sample app listens on 5000. If you need to serve on port 80, map it on the host side with `-p 80:5000` rather than running as root. Some recent kernels and Docker Desktop setups relax the rule with a setting called `net.ipv4.ip_unprivileged_port_start`, but do not count on it being in place.

If you ship with Docker Compose rather than raw `docker run`, the same controls exist as keys: `user`, `cap_drop: [ALL]`, `security_opt: [no-new-privileges: true]`, `read_only: true`, `tmpfs`, `pids_limit`, `mem_limit`, and `cpus`.

4 Two Things to Never Do

This section is short, but it covers the mistakes behind the worst container incidents. Both of them effectively hand the container root on the host, which undoes everything in section 3.

> **1. Never use `--privileged`**
>
> This flag switches off almost all of the isolation the opening described at once: it grants all capabilities, disables the syscall filter, and exposes the host's devices to the container. A privileged container is close to a process running directly on the host as root. It shows up in tutorials because it makes awkward things work on the first try, so if you see advice that says to add it, treat that as a warning sign and look for the narrower fix, which is usually a single `--cap-add`, a single `--device`, or one specific mount.

2. Never mount `/var/run/docker.sock`

This is the quieter danger, because it looks harmless and appears in a lot of continuous-integration and management setups. Recall that the socket is how a client sends commands to the root daemon. A container that can reach the socket can therefore tell the daemon to start another container, and that second container can mount the host's entire filesystem:

```
# Demonstrate ONLY on a throwaway VM, NEVER on a machine you care about
docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock docker sh

# Inside the container:
docker run -it -v /:/host alpine chroot /host
```

Access to the daemon is access to root on the host, with no exploit required. Treat a mounted `docker.sock` the way you would treat giving out the root password.

If your real need is to build images inside a container, which is common in CI, do not reach for either of these. Use one of the tools designed for it: **Kaniko**, which builds an image from a Dockerfile without a daemon and without privileged mode; **BuildKit in rootless mode**; or **Sysbox**, a container runtime that lets you run Docker inside a container safely.

5 Cut the Image Down to What It Uses

The theme from section 1 was that less in the image means less to attack and less to patch. A small base and a multi-stage build get you far. This tool goes further by watching what your application actually uses at runtime and rebuilding an image from only that.

> **A note on the name first**, because it will bite you at the terminal. The tool began as `docker-slim` and is now maintained as SlimToolkit (at github.com/slimtoolkit/slim) with the command-line binary `slim`. The same author also runs a newer `mintoolkit/mint` line, and Homebrew packages it under the formula name `mintoolkit`. Because the naming is still in flux, check what actually installed, for example with `slim --version` or by tab-completing the binary. The subcommands below have stayed the same across the renames.

The important thing is the verb: you minify with the `build` command, not with a `slim` command. The tool starts your container, records which files and which system calls it actually uses, and rebuilds a minimal image from only those. A **system call**, or syscall, is the way a program asks the kernel to do something it cannot do on its own, such as open a file or a network connection; the set of syscalls a program makes is a good description of what it really does. The rebuilt image is often ten to thirty times smaller and, because the shell and package manager are gone, offers an attacker very little to work with. The `build` command can also write a seccomp profile.

Seccomp, short for secure computing mode, is a kernel feature that restricts which syscalls a process is allowed to make. Docker already applies a default seccomp profile that blocks a few dozen dangerous calls. A profile generated from your specific application is much tighter, because it allows only the calls your app was seen to use and the kernel kills the process if it tries anything else.

```
slim xray myapp:2.0                # inspect what is actually in the image, layer by layer
slim build --http-probe myapp:2.0   # run and profile the app, then rebuild it minimally
docker run --rm -p 5000:5000 myapp.slim
docker scout quickview myapp.slim  # the vulnerability count usually drops too
```

One tie-in to the previous section: `slim` needs to talk to the Docker daemon to do its work, so run it on a build machine you trust, not by mounting the socket into some long-lived container.

Now the honest caveat, because this technique has a real failure mode. It works by **dynamic analysis**, meaning it only keeps what it saw the application do while it was watching. A code path you did not exercise during profiling, such as an admin route, an error handler, or a scheduled job, can be stripped out and then fail in production. So drive real traffic or your test suite at it during profiling, use `--include-path` to force-keep files you know are needed, and always test the slimmed image before shipping it. If that feels like too much for your stack, a distroless or Wolfi base image gets you most of the size and surface reduction with fewer surprises.

Apply the generated profile at runtime like this:

```
docker run --rm --security-opt seccomp=./myapp-seccomp.json \
  -p 5000:5000 myapp.slim
```

Checklist for Your Own Images

- ☐ Small base, tag and digest pinned, scanned before use
- ☐ A `.dockerignore` file present, no `COPY . .`, a multi-stage build, and a non-root user
- ☐ Run with `--cap-drop ALL`, `--security-opt=no-new-privileges`, `--read-only` plus `--tmpfs`, and memory, CPU, and PID limits
- ☐ No `--privileged`, and no mounted `docker.sock`
- ☐ Image reduced with `slim` (SlimToolkit) or a distroless base, with a seccomp profile applied

Cheat Sheet

Category	Key practices & commands
Images	Small base; `FROM img:tag@sha256:...` Scan: `docker scout cves IMG` or `trivy image IMG`
Build	`.dockerignore`; copy only what you need; multi-stage; `USER` non-root Build-time secrets: `--mount=type=secret` (never `ARG` or `ENV`)
Run	`--user 10001 --cap-drop ALL --security-opt no-new-privileges` `--read-only --tmpfs /tmp --memory 256m --cpus=0.5 --pids-limit=100`
Avoid	`--privileged` `-v /var/run/docker.sock:/var/run/docker.sock`
Shrink	Building inside a container? Use Kaniko, rootless BuildKit, or Sysbox `slim xray IMG` → `slim build --http-probe IMG` → apply seccomp; test first (Binary is `slim` from slimtoolkit/slim; verify the name after install)

Glossary

Base image: The image named in your first `FROM` line, which your own image is built on top of. Everything in it becomes part of yours.

Build context: The files sent to the daemon for a build, normally the current directory. `COPY` can only pull from here, and `.dockerignore` decides what is excluded.

Capability: One of about forty separate powers that Linux split out of the all-or-nothing root user, so privilege can be granted piece by piece.

cgroup (control group): The kernel feature that limits and measures a process group's use of memory, CPU, process count, and other resources. Behind `--memory`, `--cpus`, and `--pids-limit`.

Container: A running instance of an image, with a thin writable layer on top. Really just host processes inside a set of namespaces and cgroups.

CVE: Common Vulnerabilities and Exposures: the standard public identifier for a single known security flaw. Scanners report these.

Daemon (dockerd): The background service, running as root, that actually builds and runs containers. The `docker` command is only a client that talks to it.

Digest: A cryptographic hash of an exact image, written `sha256:...`. Immutable, unlike a tag.

Distroless: An image containing only an app and its runtime dependencies, with no shell or package manager, which reduces what an attacker can use.

Dockerfile: The recipe of instructions used to build an image.

Image: A read-only, layered filesystem snapshot plus metadata, used as the template for containers.

Layer: The unit an image is built in; one per filesystem-changing instruction. Layers are cached, shared, and immutable, which is why a deleted secret can still be recovered from an earlier layer.

Least privilege: The practice of granting only the access strictly needed. Most of this document is applied least privilege.

Multi-stage build: A Dockerfile with more than one `FROM`, where heavy build tools live in earlier stages and only finished artifacts are copied into the final image.

Namespace: The kernel feature that gives a process its own private view of a resource such as processes, filesystem mounts, or the network. The basis of container isolation.

Registry: The server that stores and distributes images, such as Docker Hub or GHCR.

Rootless mode: Running the Docker daemon or a build tool as a non-root user, so even a container breakout is confined to an unprivileged account.

SBOM: Software Bill of Materials: a full inventory of the packages and versions in an image, useful for answering whether you are exposed to a new CVE.

seccomp: A kernel feature that restricts which system calls a process may make. Docker applies a default profile; a per-application profile is tighter.

System call (syscall): A request from a program to the kernel to perform a privileged action, such as opening a file or a socket. The boundary between an app and the kernel.

Tag: A mutable human-friendly label on an image, such as `3.12-slim`. Can be repointed, so pin a digest for anything you depend on.

User namespace remapping (userns-remap): A setting that maps the container's root (UID 0) to an unprivileged user ID on the host, so container root is not host root. Off by default.

Where to Read More

Docker Bench for Security, the CIS Docker Benchmark, and the OWASP Docker Security Cheat Sheet cover this ground in more depth. For the tools named above: Slim Toolkit lives at github.com/slimtoolkit/slim (originally docker-slim, with a newer mintoolkit/mint line), distroless at github.com/GoogleContainerTools/distroless, and Chainguard publishes the Wolfi images. Trivy, Grype, and Docker Scout scan images; Syft builds SBOMs; Cosign handles signing; Kaniko, rootless BuildKit, and Sysbox handle unprivileged builds.