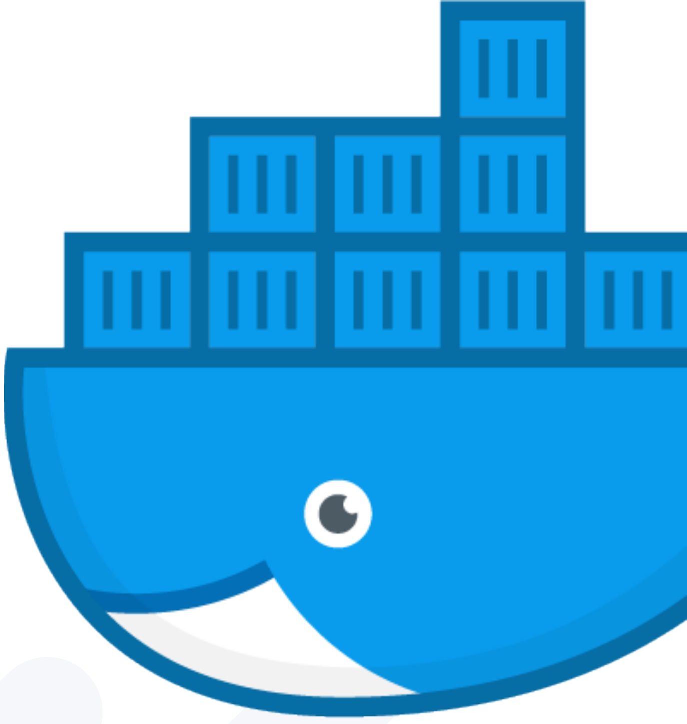


Docker Security Best Practices

Docker accelerates your development and deployment cycles, letting you push code out faster than ever before. But it also comes with an unexpected set of security implications that you should be aware of.

Below are five common scenarios where deploying Docker images open up new kinds of security issues you might not have considered, and some great tools and advice that you can use to ensure you aren't leaving the barn doors open when you deploy.

1. Image Authenticity

Let's start with an issue that is, perhaps, inherent in Docker's very nature: image authenticity.
If you've been using Docker for any length of time, you'll be familiar with the ability to base your containers around almost any image; whether it comes from an official one, such as NGINX, Redis, Ubuntu, or Alpine Linux; or one from a total stranger.

As a result, we have an enormous level of choice. If you don't like one container, because it doesn't quite suit your needs, then you can replace it with another one.
However, is that the most secure approach?

If you're not sure what I mean, let's consider this from a code perspective.
When developing an application, as a result of your language's package manager, just because you can use code from anyone, should you?
Or should you treat all code which you've not analyzed with a healthy level of suspicion?

I'd suggest that, if security holds any level of importance to you, that you'd always do due diligence on the code before integrating it into your application's code base.
Am I right?

Well, the same level of suspicion and skepticism should be given to Docker containers.
If you don't know the developer or organization, can you be confident that the container you've chosen doesn't contain compromised binaries of other malicious code?
It'd be hard to be sure.

Given that, here are three things that I suggest you do.

USE PRIVATE OR TRUSTED REPOSITORIES

Firstly, you can use private and trusted repositories such as Docker Hub's official repositories.
The official repositories have base images for:

- Operating systems such as Ubuntu.

- Programming languages such as PHP and Ruby.
- Servers such as MySQL, PostgreSQL, and Redis.

What makes Docker Hub special, among other things, is that the images are always scanned and reviewed by Docker's Security Scanning Service. If you've not heard of the service, quoting the documentation:

> Docker Cloud and Docker Hub can scan images in private repositories to verify that they are free from known security vulnerabilities or exposures, and report the results of the scan for each image tag.

As a result, by using the official repositories, you can know that your containers are safe to use and don't contain malicious code.

The feature is available to free plans for a limited time, and available to all paid plans.
If you already have a paid plan, make use of the scanning service to ensure that your custom container images are also safe and secure, and don't contain any vulnerabilities which you're not aware of.
In so doing, you can then create a private repository for use within your organization.

USE DOCKER CONTENT TRUST

Another tool that you should make use of is Docker Content Trust.
This is a new feature introduced in Docker Engine 1.8, which allows you to verify Docker image publishers.
To quote the release article by Diogo Mónica, Docker's security lead:

> Before a publisher pushes an image to a remote registry, Docker Engine signs the image locally with the publisher's private key. When you later pull this image, Docker Engine uses the publisher's public key to verify that the image you are about to run is exactly what the publisher created, has not been tampered with and is up to date.

To summarize, the service protects against image forgery, replay attacks, and key compromises.
I *strongly* encourage you to check out the article, as well as the official documentation.

Docker Bench Security

Another tool that I've recently been using is Docker Bench Security, which:

Checks for dozens of common best practices around deploying Docker containers in production.

The tool was based on the recommendations in the CIS Docker 1.13 Benchmark, and run checks against the following six areas:

1. Host configuration
2. Docker daemon configuration
3. Docker daemon configuration files
4. Container images and build files
5. Container runtime
6. Docker security operations

To install it, clone the repository by running:

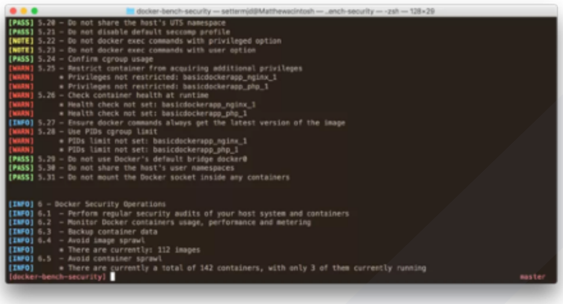
```
git clone git@github.com:docker/
docker-bench-security.git
```

Then, `cd docker-bench-secutity` and run the following command:

```
docker run -it --net host --pid host
--cap-add audit_control \
-e
DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_T
```

```
RUST \
-v /var/lib:/var/lib \
-v /var/run/docker.sock:/var/run/
docker.sock \
-v /etc:/etc --label
docker_bench_security \
docker/docker-bench-security
```

Doing so builds the container and starts the script running checks against the host machine and its containers.
Below is a sample of the output which it produces.



```
docker-bench-security — settermjd@Matthewacintosh — ..ench-security — -zsh — 128×29
[PASS] 5.20 - Do not share the host's UTS namespace
[PASS] 5.21 - Do not disable default seccomp profile
[NOTE] 5.22 - Do not docker exec commands with privileged option
[NOTE] 5.23 - Do not docker exec commands with user option
[PASS] 5.24 - Confirm cgroup usage
[WARN] 5.25 - Restrict container from acquiring additional privileges
[WARN]      * Privileges not restricted: basicdockerapp_nginx_1
[WARN]      * Privileges not restricted: basicdockerapp_php_1
[WARN] 5.26 - Check container health at runtime
[WARN]      * Health check not set: basicdockerapp_nginx_1
[WARN]      * Health check not set: basicdockerapp_php_1
[INFO] 5.27 - Ensure docker commands always get the latest version of the image
[WARN] 5.28 - Use PIDs cgroup limit
[WARN]      * PIDs limit not set: basicdockerapp_nginx_1
[WARN]      * PIDs limit not set: basicdockerapp_php_1
[PASS] 5.29 - Do not use Docker's default bridge docker0
[PASS] 5.30 - Do not share the host's user namespaces
[PASS] 5.31 - Do not mount the Docker socket inside any containers


[INFO] 6 - Docker Security Operations
[INFO] 6.1  - Perform regular security audits of your host system and containers
[INFO] 6.2  - Monitor Docker containers usage, performance and metering
[INFO] 6.3  - Backup container data
[INFO] 6.4  - Avoid image sprawl
[INFO]      * There are currently: 112 images
[INFO] 6.5  - Avoid container sprawl
[INFO]      * There are currently a total of 142 containers, with only 3 of them currently running
[docker-bench-security]                                                                      master
```

You can see that it provides quite clear, color-coded, output detailing the checks that it's run and their results.
In my case, you can see that it's in need of some improvement.

What I particularly like about this feature is that it can be automated.
As a result, it is included as part of a CI process, helping to ensure that containers are operating as securely as possible.

2. Excess Privileges

Next, ask yourself: `For as long as I can remember, the question of excess privileges has been a constant. Whether that was back in the days of installing Linux distros on bare-metal servers or more recently when installing them as a guest operating system inside a virtual machine.

Well, just because we're installing it inside a Docker container doesn't make it any more inherently secure.
Plus, Docker adds an extra layer of complexity, as the wall between the guest and host is no longer as clear.

With respect to Docker I'm specifically focused on two points:

• Containers running in privileged mode

• Excess privileges used by containers

Starting with the first point, you can run a Docker container with the `--privileged` switch.
What this does is give extended privileges to this container.
Quoting the documentation, it:

> gives all capabilities to the container, and it also lifts all the limitations enforced by the device cgroup controller. In other words, the container can then do almost everything that the host can do. This flag exists to allow special use-cases, like running Docker within Docker.

If the very idea of this capability doesn't give you pause, then I'd be surprised — even concerned.
Honestly, unless you have *a very particular* use case, then I don't know why you'd use this switch, not without a lot of caution.
Given that, please tread very carefully before ever doing so.
To quote Armin Braun:

> Don't use privileged containers unless you treat them the same way you treat any other process running as root.

But even if you're not running a container with the `--privileged` switch, one or more of your containers may have excess capabilities.

By default, Docker starts containers with a rather restricted set of capabilities.

However, that can be overridden via a non-default profile.
Depending on who you're hosting your Docker containers with, including vendors such as *DigitalOcean*, *sloppy.io*, *dotCloud*, and *Quay.io*, these defaults may not be what you have.
You may also be self-hosting, in which case, it's just as important to validate the privileges which your containers have.

DROP UNNECESSARY PRIVILEGES AND CAPABILITIES

Regardless of who you're hosting with, as the Docker Security guide says:

The best practice for users would be to remove all capabilities except those explicitly required for their processes.

Consider these questions:

• What kind of network connectivity does your application need?

• Does it need raw socket access?

• Does it need to send and receive UDP requests?

If not, then deny it those capabilities.

However, does it need capabilities that, by default, most of your applications don't?
If so, then provide it those capabilities.

By doing so, you will help limit the ability for malicious users to abuse your systems, because the ability for them to do so isn't available.
To do so, make use of the `--cap-drop` and `--cap-add` switches.

Let's say that your application doesn't need to be able to either modify process capabilities or to bind on privileged ports, but does need the ability to load and unload kernel modules.

Here's how you would remove and add the respective capabilities:

```
docker run \
--cap-drop SETPCAP \
--cap-drop NET_BIND_SERVICE \
--cap-add SYS_MODULE \
-ti /bin/sh
```

For in-depth coverage of these options, refer to the "Runtime privilege and Linux capabilities" section of the documentation.

3. System Security

Ok, you're using a verified base image and have reduced (or removed) the excess privileges available to your containers.
But how secure is the image you're using?
For example, what kind of permissions can rogue agents get access to if they access your container?
Or said another way, how much have your hardened your container(s)?
If attackers can breach your container, is it then quite easy for them to do anything else?
If so, it's time to harden your container.

While Docker is increasingly very safe by default, thanks to having namespaces and cgroups as its foundation, you don't need to only rely on these features.
You can go further and make use of other Linux security options, such as AppArmor, SELinux, grsecurity and Seccomp.

Each of these is a mature and well-tested tool capable of further extending the security of the security of your containers' host.
If you're not familiar with them, here's a brief overview.

AppArmor

is a Linux kernel security module that allows the system administrator to restrict programs' capabilities with per-program profiles. Profiles can allow capabilities like network access, raw socket access, and the permission to read, write, or execute files on matching paths. AppArmor supplements the traditional Unix discretionary access control (DAC) model by providing mandatory access control (MAC). It was included in the mainline Linux kernel since version 2.6.36 –
Source: Wikipedia.

SELinux

Security-Enhanced Linux (SELinux) is a Linux kernel security module that provides a mechanism for supporting access control security policies, including...(MAC). SELinux is a set of kernel modifications and user-space tools that have been added to various Linux distributions. Its architecture strives to separate enforcement of security decisions from the security policy itself and streamlines the volume of software charged with security policy enforcement – Source: Wikipedia.

grsecurity

A set of patches for the Linux kernel which emphasize security enhancements. The patches are typically used by computer systems which accept remote connections from untrusted locations, such as web servers and systems offering shell access to its users. Grsecurity provides a collection of security features to the Linux kernel, including address space protection, enhanced auditing and process control – Source: Wikipedia.

seccomp

short for secure computing mode, [seccomp] is a computer security facility in the Linux kernel. It was merged into the Linux kernel mainline in kernel version 2.6.12, which was released on March 8, 2005. seccomp allows a process to make a one-way transition into a "secure" state where it cannot make any system calls except exit(), sigreturn(), read() and write() to already-open file descriptors. Should it attempt any other system calls, the kernel will terminate the process with SIGKILL. In this sense, it does not virtualize the system's resources but isolates the process from them entirely –
Source: Wikipedia

It's beyond the scope of this article to provide working examples, nor a deeper coverage of these technologies.
But I strongly encourage you to learn more about them and to use them as part of your infrastructure.

4. Limit Available Resource Consumption

What does your application need?
Is it a quite light application, requiring no more than 50MB of memory?
Then why let it access more?
Does it perform more intensive processing, requiring 4+ CPUs?
Then allow it to have access to that — **yet no more**.

Assuming that *analysis*, *profiling*, and *benchmarking* are part of your continuous development processes, then this information will be readily available.

As a result, when deploying your containers, ensure that they only have the resources that they need. To do so, use the applicable docker run switches, such as the following:

- `-m / --memory`: Set a memory limit
- `--memory-reservation`: Set a soft memory limit
- `--kernel-memory`: Set a kernel memory limit
- `--cpus`: Limit the number of CPUs
- `--device-read-bps`: Limit the read rate from a device

Here's an example of how to do some of these in a Docker compose configuration file, taken from the official documentation:

```
version: '3'
services:
redis:
image: redis:alpine
deploy:
resources:
limits:
cpus: '0.001'
memory: 50M
reservations:
memory: 20M
```

More information can be found by running `docker help run`, or by referring to the "Runtime constraints on resources" section of the Docker run documentation.

5. Large Attack Surfaces

The final aspect of security we should consider is one that comes about by consequence of how Docker works — a potentially quite large attack surface.
This can happen in any IT organization, but is exacerbated by the ephemeral nature of container-based infrastructure.

As Docker allows you to create and deploy applications so quickly — as well as to destroy them with equal ease — it can be difficult to know *exactly* what applications your organization has deployed.
Given that, the potential of a large attack surface grows.

Not sure about the deployment statistics of your organization?
To help you out, ask yourself the following questions:

- What applications are currently deployed by your organization?
- Who deployed them?
- When were they deployed?
- Why were they deployed?
- How long are they to be deployed for?
- Who's responsible for them?
- When was a security scan last run on them?

I hope that you're not getting too concerned as you're considering these questions.
Whether you are, or if you're not, let's consider some practical actions you can take.

IMPLEMENT AN AUDIT TRAIL WITH PROPER LOGGING

As with audit trails within applications, such as when a user created their account when it was activated, and when the user last updated their password, and those more widely within organizations, consider implementing an audit trail around every container that your organization creates and deploys.

This needn't be either sophisticated nor overly complex.
But it should implement something that records such specifics as:

- When an application was deployed
- Who deployed it
- Why it was deployed
- What its intent is
- When it should be deprecated

Most continuous development tools should support recording this information, whether directly in the tool or by supplying custom scripts in your preferred programming language of choice.

In addition to this, consider adding notifications, whether via email, or others such as *IRC*, *Slack*, and *HipChat*.
This extra layer should ensure that deployments are both visible and transparent to everyone.
That way, if something shouldn't have happened, it cannot be hidden from view.

I'm not suggesting that you shouldn't trust staff within your organization, but it is good to stay abreast of what's happening.

Before we finish up, please don't misunderstand me.
I'm not suggesting that you go overboard or become bogged down in the creation of a slew of new processes.
Doing so would likely only serve to wipe out many of the gains that using containers provides — as well as be completely unnecessary.

However, having the necessity to at least think through these questions, being able to review it on a regular basis, and being able to work from an informed perspective at all times should help reduce the likelihood of creating unknown attack surfaces against your organization.

In Conclusion

And that's been a look at five Docker security concerns, along with a range of potential solutions for them.
I hope that, if you're transitioning to Docker, considering transitioning, or already have, that you'll consider these areas and what you've undertaken to ensure your applications are protected against them.

Docker is an amazing technology, one which I wish had arrived *far* sooner than it did.
I hope that with the information presented here, you'll be in a position to ensure that now that it is here, it's working in your best interest and you're not exposed to any unexpected issues.

About the author

Matthew Setter is an independent software developer and technical writer. He specializes in creating test-driven applications and writing about modern software practices, including continuous development, testing, and security.

sqreen

Secure your container-based applications with Sqreen

Start your 14-day free trial and get protected in less than a minute!

web	www.sqreen.io
twitter	@sqreenIO