On this page:

Docker security

Kernel namespaces

Control groups

Docker daemon attack surface

Linux kernel capabilities

Other kernel security features

Conclusions

Install

Docker Fundamentals

Use Docker

About Docker (../../../engine/misc/)

Apply custom metadata (../../../engine/userguide/labels-custom-metadata/)

Docker Deprecated Features (../../../engine/misc/deprecated/)

Understand the architecture (../../../engine/introduction/understanding-docker/)

Provision & set up Docker hosts

Create multi-container applications

Cluster Docker containers

Administrate Docker

Automatically start containers (../../../engine/articles/host_integration/)

Docker security (../../../engine/articles/security/)

Configuring and running Docker (../../../engine/articles/configuring/)

Runtime metrics (../../../engine/articles/runmetrics/)

Protect the Docker daemon socket (../../../engine/articles/https/)

Link via an ambassador container (../../../engine/articles/ambassador_pattern_linking/)

Control and configure Docker with systemd (../../../engine/articles/systemd/)

Logging

Applications and Services

Integrate with Third-party Tools

Docker storage drivers

Network configuration

Applied Docker

Manage image repositories

Extend Docker

Command and API references

Open Source at Docker

About

Docs archive

Docker security

There are three major areas to consider when reviewing Docker security:

- the intrinsic security of the kernel and its support for namespaces and cgroups;
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers.

Kernel namespaces

Docker containers are very similar to LXC containers, and they have similar security features. When you start a container with `docker run`, behind the scenes Docker creates a set of namespaces and control groups for the container.

Namespaces provide the first and most straightforward form of isolation: processes running within a container cannot see, and even less affect, processes running in another container, or in the host system.

Each container also gets its own network stack, meaning that a container doesn’t get privileged access to the sockets or interfaces of another container. Of course, if the host system is setup accordingly, containers can interact with each other through their respective network interfaces — just like they can interact with external hosts. When you specify public ports for your containers or use *links* (../../../engine/userguide/networking/default_network/dockerlinks/) then IP traffic is allowed between containers. They can ping each other, send/receive UDP packets, and establish TCP connections, but that can be restricted if necessary. From a network architecture point of view, all containers on a given Docker host are sitting on bridge interfaces. This means that they are just like physical machines connected through a common Ethernet switch; no more, no less.

How mature is the code providing kernel namespaces and private networking? Kernel namespaces were introduced between kernel version 2.6.15 and 2.6.26 (http://lxc.sourceforge.net/index.php/about/kernel-namespaces/). This means that since July 2008 (date of the 2.6.26 release, now 7 years ago), namespace code has been exercised and scrutinized on a large number of production systems. And there is more: the design and inspiration for the namespaces code are even older. Namespaces are actually an effort to reimplement the features of OpenVZ

(http://en.wikipedia.org/wiki/OpenVZ) in such a way that they could be merged within the mainstream kernel. And OpenVZ was initially released in 2005, so both the design and the implementation are pretty mature.

Control groups

Control Groups are another key component of Linux Containers. They implement resource accounting and limiting. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

So while they do not play a role in preventing one container from accessing or affecting the data and processes of another container, they are essential to fend off some denial-of-service attacks. They are particularly important on multi-tenant platforms, like public and private PaaS, to guarantee a consistent uptime (and performance) even when some applications start to misbehave.

Control Groups have been around for a while as well: the code was started in 2006, and initially merged in kernel 2.6.24.

Docker daemon attack surface

Running containers (and applications) with Docker implies running the Docker daemon. This daemon currently requires `root` privileges, and you should therefore be aware of some important details.

First of all, **only trusted users should be allowed to control your Docker daemon**. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container. This means that you can start a container where the `/host` directory will be the `/` directory on your host; and the container will be able to alter your host filesystem without any restriction. This is similar to how virtualization systems allow filesystem resource sharing. Nothing prevents you from sharing your root filesystem (or even your root block device) with a virtual machine.

This has a strong security implication: for example, if you instrument Docker from a web server to provision containers through an API, you should be even more careful than usual with parameter checking, to make sure that a malicious user cannot pass crafted parameters causing Docker to create arbitrary containers.

For this reason, the REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (the latter being prone to cross-site-scripting attacks if you happen to run Docker directly on your local machine, outside of a VM). You can then use traditional UNIX permission checks to limit access to the control socket.

You can also expose the REST API over HTTP if you explicitly decide to do so. However, if you do that, being aware of the above mentioned security implication, you should ensure that it will be reachable only from a trusted network or VPN; or protected with e.g., `stunnel` and client SSL certificates. You can also secure them with HTTPS and certificates (../../../engine/articles/https/).

The daemon is also potentially vulnerable to other inputs, such as image loading from either disk with 'docker load', or from the network with 'docker pull'. This has been a focus of improvement in the community, especially for 'pull' security. While these overlap, it should be noted that 'docker load' is a mechanism for backup and restore and is not currently considered a secure mechanism for loading images. As of Docker 1.3.2, images are now extracted in a chrooted subprocess on Linux/Unix platforms, being the first-step in a wider effort toward privilege separation.

Eventually, it is expected that the Docker daemon will run restricted privileges, delegating operations well-audited sub-processes, each with its own (very limited) scope of Linux capabilities, virtual network setup, filesystem management, etc. That is, most likely, pieces of the Docker engine itself will run inside of containers.

Finally, if you run Docker on a server, it is recommended to run exclusively Docker in the server, and move all other services within containers controlled by Docker. Of course, it is fine to keep your favorite admin tools (probably at least an SSH server), as well as existing monitoring/supervision processes (e.g., NRPE, collectd, etc).

Linux kernel capabilities

By default, Docker starts containers with a restricted set of capabilities. What does that mean?

Capabilities turn the binary "root/non-root" dichotomy into a fine-grained access control system. Processes (like web servers) that just need to bind on a port below 1024 do not have to run as root: they can just be granted the `net_bind_service` capability instead. And there are many other capabilities, for almost all the specific areas where root privileges are usually needed.

This means a lot for container security; let's see why!

Your average server (bare metal or virtual machine) needs to run a bunch of processes as root. Those typically include SSH, cron, syslogd; hardware management tools (e.g., load modules), network configuration tools (e.g., to handle DHCP, WPA, or VPNs), and much more. A container is very different, because almost all of those tasks are handled by the infrastructure around the container:

- SSH access will typically be managed by a single server running on the Docker host;
- `cron`, when necessary, should run as a user process, dedicated and tailored for the app that needs its scheduling service, rather than as a platform-wide facility;
- log management will also typically be handed to Docker, or by third-party services like Loggly or Splunk;
- hardware management is irrelevant, meaning that you never need to run `udevd` or equivalent daemons within containers;
- network management happens outside of the containers, enforcing separation of concerns as much as possible, meaning that a container should never need to perform `ifconfig`, `route`, or ip commands (except when a container is specifically engineered to behave like a router or firewall, of course).

This means that in most cases, containers will not need "real" root privileges *at all*. And therefore, containers can run with a reduced capability set; meaning that "root" within a container has much less privileges than the real "root". For instance, it is possible to:

- deny all "mount" operations;
- deny access to raw sockets (to prevent packet spoofing);
- deny access to some filesystem operations, like creating new device nodes, changing the owner of files, or altering attributes (including the immutable flag);
- deny module loading;
- and many others.

This means that even if an intruder manages to escalate to root within a container, it will be much harder to do serious damage, or to escalate to the host.

This won't affect regular web apps; but malicious users will find that the arsenal at their disposal has shrunk considerably! By default Docker drops all capabilities except those needed (https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/defa a whitelist instead of a blacklist approach. You can see a full list of available capabilities in Linux manpages (http://man7.org/linux/man-pages/man7/capabilities.7.html).

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.

Docker supports the addition and removal of capabilities, allowing use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

Other kernel security features

Capabilities are just one of the many security features provided by modern Linux kernels. It is also possible to leverage existing, well-known systems like TOMOYO, AppArmor, SELinux, GRSEC, etc. with Docker.

While Docker currently only enables capabilities, it doesn't interfere with the other systems. This means that there are many different ways to harden a Docker host. Here are a few examples.

- You can run a kernel with GRSEC and PAX. This will add many safety checks, both at compile-time and run-time; it will also defeat many exploits, thanks to techniques like address randomization. It doesn't require Docker-specific configuration, since those security features apply system-wide, independent of containers.
- If your distribution comes with security model templates for Docker containers, you can use them out of the box. For instance, we ship a template that works with AppArmor and Red Hat comes with SELinux policies for Docker. These templates provide an extra safety net (even though it overlaps greatly with capabilities).
- You can define your own policies using your favorite access control mechanism.

Just like there are many third-party tools to augment Docker containers with e.g., special network topologies or shared filesystems, you can expect to see tools to harden existing Docker containers without affecting Docker's core.

Recent improvements in Linux namespaces will soon allow to run full-featured containers without root privileges, thanks to the new user namespace. This is covered in detail here (http://s3hh.wordpress.com/2013/07/19/creating-and-using-containers-without-privilege/). Moreover, this will solve the problem caused by sharing filesystems between host and guest, since the user namespace allows users within containers (including the root user) to be mapped to other users in the host system.

Today, Docker does not directly support user namespaces, but they may still be utilized by Docker containers on supported kernels, by directly using the clone syscall, or utilizing the 'unshare' utility. Using this, some users may find it possible to drop more capabilities from their process as user namespaces provide an artificial capabilities set. Likewise, however, this artificial capabilities set may require use of 'capsh' to restrict the user-namespace capabilities set when using 'unshare'.

Eventually, it is expected that Docker will have direct, native support for user-namespaces, simplifying the process of hardening containers.

Conclusions

Docker containers are, by default, quite secure; especially if you take care of running your processes inside the containers as non-privileged users (i.e., non-`root`).

You can add an extra layer of safety by enabling AppArmor, SELinux, GRSEC, or your favorite hardening solution.

Last but not least, if you see interesting security features in other containerization systems, these are simply kernels features that may be implemented in Docker as well. We welcome users to submit issues, pull requests, and communicate via the mailing list.

References:

- Docker Containers: How Secure Are They? (2013) (http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/).
- On the Security of Containers (2014) (https://medium.com/@ewindisch/on-the-security-of-containers-2c60ffe25a9e).