O'REILLY®

Docker Security

Using Containers Safely in Production

Adrian Mouat

Docker Security

Using Containers Safely in Production

Adrian Mouat

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Docker Security
by Adrian Mouat

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Brian Anderson
Production Editor: Shiny Kalapurakkel
Copyeditor: Sharon Wilkey
Proofreader: Marta Justak

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2015: First Edition

Revision History for the First Edition
2015-08-17: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491936610* for release details.

978-1-491-93661-0

[LSI]

Table of Contents

Foreword ..... **vii**

Security and Limiting Containers **1**
Things to Worry About 2
Defense in Depth 5
Segregate Containers by Host 6
Applying Updates 8
Image Provenance 12
Security Tips 18
Run a Hardened Kernel 31
Linux Security Modules 32
Auditing 37
Incident Response 38
Future Features 38
Conclusion 39

Foreword

Docker's introduction of the standardized image format has fueled an explosion of interest in the use of containers in the enterprise. Containers simplify the distribution of software and allow greater sharing of resources on a computer system. But as you pack more applications onto a system, the risk of an individual application having a vulnerability leading to a breakout increases.

Containers, as opposed to virtual machines, currently share the same host kernel. This kernel is a single point of failure. A flaw in the host kernel could allow a process within a container to break out and take over the system. Docker security is about limiting and controlling the attack surface on the kernel. Docker security takes advantage of security measures provided by the host operating system. It relies on Defense in Depth, using multiple security measures to control what the processes within the container are able to do. As Docker/containers evolve, security measures will continue to be added.

Administrators of container systems have a lot of responsibility to continue to use the common sense security measures that they have learned on linux and UNIX systems over the years. They should not just rely on whether the "containers actually contain."

- Only run container images from trusted parties.
- Container applications should drop privileges or run without privileges whenever possible.
- Make sure the kernel is always updated with the latest security fixes; the security kernel is critical.
- Make sure you have support teams watching for security flaws in the kernel.

- Use a good quality supported host system for running the containers, with regular security updates.
- Do not disable security features of the host operating system.
- Examine your container images for security flaws and make sure the provider fixes them in a timely manner.

—Dan Walsh
Consulting Engineer, Red Hat

Security and Limiting Containers

To use Docker safely, you need to be aware of the potential security issues and the major tools and techniques for securing container-based systems. This report considers security mainly from the viewpoint of running Docker in production, but most of the advice is equally applicable to development. Even with security, it is important to keep the development and production environments similar in order to avoid the issues around moving code between environments that Docker was intended to solve.

Reading online posts and news items[1] about Docker can give you the impression that Docker is inherently insecure and not ready for production use. While you certainly need to be aware of issues related to using containers safely, containers, if used properly, can provide a more secure and efficient system than using virtual machines (VMs) or bare metal alone.

This report begins by exploring some of the issues surrounding the security of container-based systems that you should be thinking about when using containers.

1 The best articles on Docker security include the series by Dan Walsh of Red Hat on opensource.com and Jonathan Rudenberg's article on image insecurity, but note that the issues in Jonathan's article have been largely addressed by the development of digests and the Notary project.

Disclaimer!

The guidance and advice in this report is based on my opinion. I am not a security researcher, nor am I responsible for any major public-facing system. That being said, I am confident that any system that follows the guidance in this report will be in a better security situation than the majority of systems out there. The advice in this report does not form a complete solution and should be used only to inform the development of your own security procedures and policy.

Things to Worry About

So what sorts of security issues should you be thinking about in a container-based environment? The following list is not comprehensive, but should give you food for thought:

Kernel exploits
: Unlike in a VM, the kernel is shared among all containers and the host, magnifying the importance of any vulnerabilities present in the kernel. Should a container cause a kernel panic, it will take down the whole host. In VMs, the situation is much better: an attacker would have to route an attack through both the VM kernel and the hypervisor before being able to touch the host kernel.

Denial-of-service attacks
: All containers share kernel resources. If one container can monopolize access to certain resources—including memory and more esoteric resources such as user IDs (UIDs)—it can starve out other containers on the host, resulting in a denial-of-service (DoS), whereby legitimate users are unable to access part or all of the system.

Container breakouts
: An attacker who gains access to a container should not be able to gain access to other containers or the host. Because users are not namespaced, any process that breaks out of the container will have the same privileges on the host as it did in the container; if you were `root` in the container, you will be `root` on the

host.[2] This also means that you need to worry about potential *privilege escalation* attacks—whereby a user gains elevated privileges such as those of the `root` user, often through a bug in application code that needs to run with extra privileges. Given that container technology is still in its infancy, you should organize your security around the assumption that container breakouts are unlikely, but possible.

Poisoned images
: How do you know that the images you are using are safe, haven't been tampered with, and come from where they claim to come from? If an attacker can trick you into running his image, both the host and your data are at risk. Similarly, you want to be sure that the images you are running are up-to-date and do not contain versions of software with known vulnerabilities.

Compromising secrets
: When a container accesses a database or service, it will likely require a secret, such as an API key or username and password. An attacker who can get access to this secret will also have access to the service. This problem becomes more acute in a microservice architecture in which containers are constantly stopping and starting, as compared to an architecture with small numbers of long-lived VMs. This report doesn't cover how to address this, but see the Deployment chapter of *Using Docker* (O'Reilly, 2015) for how to handle secrets in Docker.

2 Docker developers are working on methods of automatically mapping the `root` user in a container to a nonprivileged user on the host. This would dramatically reduce the capabilities of an attacker in the event of a breakout, but creates problems with the ownership of volumes.

Containers and Namespacing

In a much-cited article, Dan Walsh of Red Hat wrote, "Containers Do Not Contain." By this, he primarily meant that not all resources that a container has access to are *namespaced*. Resources that *are* namespaced are mapped to a separate value on the host; for example, PID 1 inside a container is not PID 1 on the host or in any other container. By contrast, resources that are not namespaced are the same on the host and in containers.

Resources that are not namespaced include the following:

UIDs
: If a user is `root` inside a container and breaks out of the container, that user will be `root` on the host. Mapping the `root` user to a high-numbered user is a work in progress, but this hasn't landed yet.

The kernel keyring
: If your application or a dependent application uses the kernel keyring for handling cryptographic keys or something similar, it's *very* important to be aware of this. Keys are separated by UID, meaning any container running with a user of the same UID will have access to the same keys.

The kernel itself and any kernel modules
: If a container loads a kernel module (which requires extra privileges), the module will be available across all containers and the host. This includes the Linux Security Modules discussed later.

Devices
: Including disk drives, sound-cards, and graphics processing units (GPUs).

The system time
: Changing the time inside a container changes the system time for the host and all other containers. This is possible only in containers that have been given the `SYS_TIME` capability, which is not granted by default.

The simple fact is that both Docker and the underlying Linux kernel features it relies on are still young and nowhere near as battle-hardened as the equivalent VM technology. For the time being at

least, do not consider containers to offer the same level of security guarantees as VMs.[3]

Defense in Depth

So what can you do? Assume vulnerability and build defense in depth. Consider the analogy of a castle, which has multiple layers of defense, often tailored to thwart various kinds of attacks. Typically, a castle has a moat, or exploits local geography, to control access routes to the castle. The walls are thick stone, designed to repel fire and cannon blasts. There are battlements for defenders and multiple levels of keeps inside the castle walls. Should an attacker get past one set of defenses, there will be another to face.

The defenses for your system should also consist of multiple layers. For example, your containers will most likely run in VMs so that if a container breakout occurs, another level of defense can prevent the attacker from getting to the host or other containers. Monitoring systems should be in place to alert admins in the case of unusual behavior. Firewalls should restrict network access to containers, limiting the external attack surface.

Least Privilege

Another important principle to adhere to is *least privilege*: each process and container should run with the minimum set of access rights and resources it needs to perform its function.[4] The main benefit of this approach is that if one container is compromised, the attacker

3 An interesting argument exists about whether containers will ever be as secure as VMs. VM proponents argue that the lack of a hypervisor and the need to share kernel resources mean that containers will always be less secure. Container proponents argue that VMs are more vulnerable because of their greater attack surface, pointing to the large amounts of complicated and privileged code in VMs required for emulating esoteric hardware (as an example, see the recent VENOM vulnerability that exploited code in floppy drive emulation).

4 The concept of least privilege was first articulated as "Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job," by Jerome Saltzer in "Protection and the Control of Information Sharing in Multics." Recently, Diogo Mónica and Nathan McCauley from Docker have been championing the idea of "least-privilege microservices" based on Saltzer's principle.

should still be severely limited in being able to perform actions that provide access to or exploit further data or resources.

In regards to least privilege, you can take many steps to reduce the capabilities of containers:

- Ensure that processes in containers do not run as `root`, so that exploiting a vulnerability present in a process does not give the attacker root access.
- Run filesystems as read-only so that attackers cannot overwrite data or save malicious scripts to file.
- Cut down on the kernel calls that a container can make to reduce the potential attack surface.
- Limit the resources that a container can use to avoid DoS attacks whereby a compromised container or application consumes enough resources (such as memory or CPU) to bring the host to a halt.

Docker Privileges = Root Privileges

This report focuses on the security of running containers, but it is important to point out that you also have to be careful about who you give access to the Docker daemon. Any user who can start and run Docker containers effectively has root access to the host. For example, consider that you can run the following:

```
$ docker run -v /:/homeroot -it debian bash
...
```

And you can now access any file or binary on the host machine.

If you run remote API access to your Docker daemon, be careful about how you secure it and who you give access to. If possible, restrict access to the local network.

Segregate Containers by Host

If you have a multitenancy setup, running containers for multiple users (whether these are internal users in your organization or external customers), ensure that each user is placed on a separate Docker host, as shown in Figure 1-1. This is less efficient than shar-

ing hosts between users and will result in a higher number of VMs and/or machines than reusing hosts, but is important for security. The main reason is to prevent container breakouts resulting in a user gaining access to another user's containers or data. If a container breakout occurs, the attacker will still be on a separate VM or machine and unable to easily access containers belonging to other users.

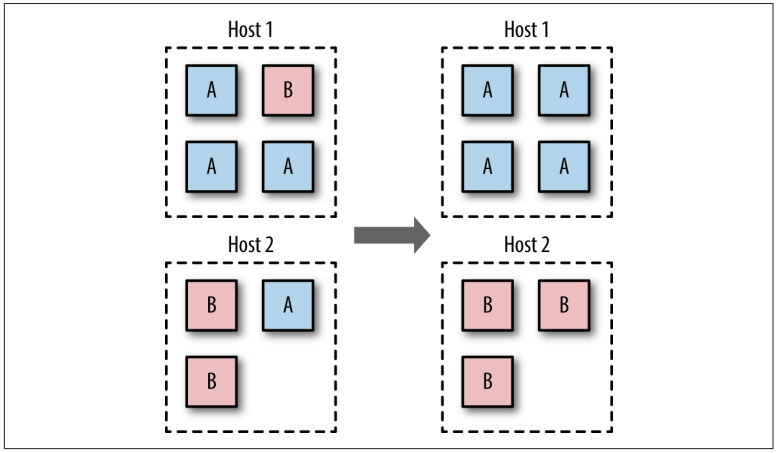

Figure 1-1. Segregating containers by host

Similarly, if you have containers that process or store sensitive information, keep them on a host separate from containers handling less-sensitive information and, in particular, away from containers running applications directly exposed to end users. For example, containers processing credit-card details should be kept separate from containers running the Node.js frontend.

Segregation and use of VMs can also provide added protection against DoS attacks; users won't be able to monopolize all the memory on the host and starve out other users if they are contained within their own VM.

In the short to medium term, the vast majority of container deployments will involve VMs. Although this isn't an ideal situation, it does mean you can combine the efficiency of containers with the security of VMs.

Applying Updates

The ability to quickly apply updates to a running system is critical to maintaining security, especially when vulnerabilities are disclosed in common utilities and frameworks.

The process of updating a containerized system roughly involves the following stages:

1. Identify images that require updating. This includes both base images and any dependent images. See "Getting a List of Running Images" on page 9 for how to do this with the Docker client.
2. Get or create an updated version of each base image. Push this version to your registry or download site.
3. For each dependent image, run `docker build` with the `--no-cache` argument. Again, push these images.
4. On each Docker host, run `docker pull` to ensure that it has up-to-date images.
5. Restart the containers on each Docker host.
6. Once you've ascertained that everything is functioning correctly, remove the old images from the hosts. If you can, also remove them from your registry.

Some of these steps sound easier than they are. Identifying images that need updating may require some grunt work and Shell-fu. Restarting the containers assumes that you have in place some sort of support for rolling updates or are willing to tolerate downtime. At the time of writing, functionality to completely remove images from a registry and reclaim the disk space is still being worked on.[5]

If you use Docker Hub to build your images, note that you can set up *repository links*, which will kick off a build of your image when

5 A work-around is to `docker save` all the required images and load them into a fresh registry.

any linked image changes. By setting a link to the base image, your image will automatically get rebuilt if the base image changes.

Getting a List of Running Images

The following gets the image IDs for all running images:

```
 $ docker inspect -f "{{.Image}}" $(docker ps -q)
42a3cf88f3f0cce2b4bfb2ed714eec5ee937525b4c7e0a0f70daff18c...
41b730702607edf9b07c6098f0b704ff59c5d4361245e468c0d551f50...
```

You can use a little more Shell-fu to get some more information:

```
$ docker images --no-trunc | grep \
  $(docker inspect -f "-e {{.Image}}" $(docker ps -q))
nginx   latest  42a3cf88f...  2 weeks ago  132.8 MB
debian  latest  41b730702...  2 weeks ago  125.1 MB
```

To get a list of all images and their base or intermediate images (use `--no-trunc` for full IDs):

```
$ docker inspect -f "{{.Image}}" $(docker ps -q) | \
    xargs -L 1 docker history -q
41b730702607
3cb35ae859e7
42a3cf88f3f0
e59ba510498b
50c46b6286b9
ee8776c93fde
439e7909f795
0b5e8be9b692
e7e840eed70b
7ed37354d38d
55516e2f2530
97d05af69c46
41b730702607
3cb35ae859e7
```

And you can extend this again to get information on the images:

```
$ docker images | grep \
  $(docker inspect -f "{{.Image}}" $(docker ps -q) | \
       xargs -L 1 docker history -q | sed "s/^/\-e /")
nginx   latest  42a3cf88f3f0  2 weeks ago  132.8 MB
debian  latest  41b730702607  2 weeks ago  125.1 MB
```

If you want to get details on the intermediate images as well as named images, add the `-a` argument to the `docker images` command. Note that this command includes a significant gotcha: if your host doesn't have a tagged version of a base image, it won't show up in the list. For example, the official Redis image is based

on `debian:wheezy`, but the base image will appear as `<None>` in `docker images -a` unless the host has separately and explicitly pulled the `debian:wheezy` image (and it is exactly the same version of that image).

When you need to patch a vulnerability found in a third-party image, including the official images, you are dependent on that party providing a timely update. In the past, providers have been criticized for being slow to respond. In such a situation, you can either wait or prepare your own image. Assuming that you have access to the Dockerfile and source for the image, rolling your image may be a simple and effective temporary solution.

This approach should be contrasted with the typical VM approach of using configuration management (CM) software such as Puppet, Chef, or Ansible. In the CM approach, VMs aren't re-created but are updated and patched as needed, either through SSH commands or an agent installed in the VM. This approach works, but means that separate VMs are often in different states and that significant complexity exists in tracking and updating the VMs. This is necessary to avoid the overhead of re-creating VMs and maintaining a master, or *golden*, image for the service. The CM approach can be taken with containers as well, but adds significant complexity for no benefit—the simpler golden image approach works well with containers because of the speed at which containers can be started and the ease of building and maintaining images.[6]

6 This is similar to modern ideas of *immutable infrastructure*, whereby infrastructure—including bare metal, VMs, and containers—is never modified and is instead replaced when a change is required.

NOTE

Label Your Images

Identifying images and what they contain can be made a lot easier by liberal use of labels when building images. This feature appeared in 1.6 and allows the image creator to associate arbitrary key/value pairs with an image. This can be done in the Dockerfile:

```
FROM debian
LABEL version 1.0
LABEL description "Test image for labels"
```

You can take things further and add data such as the Git hash that the code in the image was compiled from, but this requires using some form of templating tool to automatically update the value.

Labels can also be added to a container at runtime:

```
$ docker run -d --name label-test -l group=a \
             debian sleep 100
1d8d8b622ec86068dfa5cf251cbaca7540b7eaa6766...
$ docker inspect -f '{{json .Config.Labels}}'\
         label-test
{"group":"a"}
```

This can be useful when you want to handle certain events at runtime, such as dynamically allocating containers to load-balancer groups.

At times, you will need to update the Docker daemon to gain access to new features, security patches, or bug fixes. This will force you to either migrate all containers to a new host or temporarily halt them while the update is applied. It is recommended that you subscribe to either the *https://groups.google.com/forum/* !forum/docker-user[docker-user] or docker-dev Google groups to receive notifications of important updates.

Avoid Unsupported Drivers

Despite its youth, Docker has already gone through several stages of development, and some features have been deprecated or are unmaintained. Relying on such features is a security risk, because they will not be receiving the same attention and updates as other parts of Docker. The same goes for drivers and extensions depended on by Docker.

In particular, do not use the legacy LXC execution driver. By default, this is turned off, but you should check that your daemon isn't running with the `-e lxc` argument.

Storage drivers are another major area of development and change. At the time of writing, Docker is moving from aufs to Overlay as the preferred storage driver. The aufs driver is being taken out of the kernel and no longer developed. Users of aufs are encouraged to move to Overlay in the near future.

Image Provenance

To safely use images, you need to have guarantees about their *provenance*: where they came from and who created them. You need to be sure that you are getting exactly the same image that the original developer tested and that no one has tampered with it, either during storage or transit. If you can't verify this, the image may have become corrupted or, much worse, replaced with something malicious. Given the previously discussed security issues with Docker, this is a major concern; you should assume that a malicious image has full access to the host.

Provenance is far from a new problem in computing. The primary tool in establishing the provenance of software or data is the secure hash. A *secure hash* is something like a fingerprint for data—it is a (comparatively) small string that is unique to the given data. Any changes to the data will result in the hash changing. Several algorithms are available for calculating secure hashes, with varying degrees of complexity and guarantees of the uniqueness of the hash. The most common algorithms are SHA (which has several variants) and MD5 (which has fundamental problems and should be avoided). If you have a secure hash for some data and the data itself, you can recalculate the hash for the data and compare. If the hashes match, you can be certain that the data has not been corrupted or tampered with. An issue remains, however: why should you trust the hash? What's to stop an attacker from modifying both the data and the hash? The best answer to this is *cryptographic signing*.

This is perhaps best explained with reference to Debian's Package manager, which faces the same issues with establishing provenance. When a user installs the Debian OS, it will also install the public

part of the Debian Release signing key.[7] Before downloading software, the user needs to obtain the *Release* files, which contain lists of *Package* files along with their hashes. The *Release* files contain a signature created with the private part of the Debian key. This signature can be verified by using the public part of the Debian key that the user already has. By doing so, the user can verify the contents of the *Release* file. Because the *Release* file contains the hashes of the *Package* files, these can be verified by checking the hash, and similarly any packages can be verified by checking the appropriate hash in the *Package* file.

Be Wary of Old Images

Any images pushed from a Docker client prior to 1.6, or stored in the old (v1) version of the registry, do not have associated digests. You have no guarantees of the contents of those images; they may have been corrupted or tampered with, and you will have no way of telling.

The only level of security here is at the transport layer, which uses HTTPS by default. This does rule out man-in-the-middle attacks during transit, provided you trust the certificates used during the connection.

At the time of writing, Docker has implemented secure hashes, but is still working on the signing and verification infrastructure. The secure hashes are known as *digests* in Docker parlance. A digest is a SHA-256 hash of a filesystem layer or manifest, where a manifest is metadata file describing the constituent parts of a Docker image. Because the manifest contains a list of all the constituent layers identified by digest, if you can verify that the manifest hasn't been tampered with, you can safely download and trust the layers, even over untrustworthy channels (for example, HTTP). The specification for manifests includes a field for a cryptographic signature, which in the future will allow Docker to implement a fully trustworthy image-distribution mechanism.

7 A full discussion of public-key cryptography is beyond the scope of this book, but it is a fascinating area. For more information, see *Applied Cryptography* by Bruce Schneier (John Wiley & Sons).

Until image signing and verification is fully implemented, it is possible (theoretically, at least) that an attacker could gain access to a registry and change both the data and digest for any image, without the user or owner being aware. Because images are transferred using HTTPS, it shouldn't be possible for an attacker to tamper with images in transit (but this does require you to trust the certificates used to verify and negotiate the connection).

If you want to be certain[8] that the image you are pulling has not been tampered with, you can pull your images by hash rather than tag; for example:

```
$ docker pull debian@sha256:f43366bc755696485050ce1\
4e1429c481b6f0ca04505c4a3093dfdb4fafb899e
```

This pulls the `debian:jessie` image as of the time of writing. Unlike the `debian:jessie` tag, using the digest is guaranteed to always pull exactly the same image (or none at all). If the hash can be securely transferred and authenticated in some manner (for example, sent via a cryptographically signed email from a trusted party), you can guarantee the authenticity of the image.

Notary

The Docker Notary project is working on the secure signing and verification infrastructure. The Docker developers recognized that signing and verification of content in general was a common problem, and that it was largely unsolved outside package managers. Rather than produce a highly specialized tool that works only for Docker images, the Notary developers are building a generic server-client framework for publishing and accessing content in a trustworthy and secure manner.

A major use-case for Notary is to improve the security and trustworthiness of the common `+curl | sh` approach, which is typified by the current Docker installation instructions:

```
$ curl -sSL https://get.docker.com/ | sh
```

If such a download is compromised either on the server or in transit, the attacker can run arbitrary commands on the victim's

8 Some would say paranoid, but plenty of enterprise developers and sysadmins will fall in this category.

computer. The equivalent example using Notary looks something like this:

```
$ curl http://get.docker.com/ | \
  notary verify docker.com/scripts v1 | sh
```

The call to `notary` compares a checksum for the script with the checksum in Notary's trusted collection for `docker.com`. If it passes, you have verified that the script does indeed come from `docker.com` and has not been tampered with. If it fails, Notary will bail out and no data will be passed to `sh`. What's also notable is that the script itself can be transferred over insecure channels—in this case, `http`—without worry; if the script is altered in transit, the checksum will change and Notary will throw an error.

Notary's integration with Docker for image verification is transparent; there is no need to run manual Notary commands to verify images.

Notary is an exciting development, but it's unclear how long it will be before it is ready to be relied on in production use. At any rate, it looks set to improve the state of the art in trusted deployment of software across the industry.

If you don't trust either a private registry or the Docker Hub to distribute your images, you can always use the `docker load` and `docker save` commands to export and import images. The images can be distributed by an internal download site or simply by copying files. Of course, if you go down this route, you are likely to find yourself re-creating many of the features of the Docker registry component.

Reproducible and Trustworthy Dockerfiles

Ideally, Dockerfiles should produce exactly the same image each time. In practice, this is hard to achieve. The same Dockerfile is likely to produce different images over time. This is clearly a problematic situation, as again, it becomes hard to be sure what is in your images. It is possible to at least come close to entirely reproducible builds, by adhering to the following rules when writing Dockerfiles:

- Always specify a tag in `FROM` instructions. `FROM redis` is bad, because it pulls the `latest` tag, which changes over time and

can be expected to move with major version changes. `FROM redis:3.0` is better, but can still be expected to change with minor updates and bug fixes (which may be exactly what you want). If you want to be sure you are pulling exactly the same image each time, the only choice is to use a digest as described previously; for example:

```
FROM
redis@sha256:3479bbcab384fa343b52743b933661335448f8166...
```

Using a digest will also protect against accidental corruption or tampering.

- Provide version numbers when installing software from package managers. `apt-get install cowsay` is OK, as `cowsay` is unlikely to change, but `apt-get install cowsay=3.03+dfsg1-6` is better. The same goes for other package installers such as pip—provide a version number if you can. The build will fail if an old package is removed, but at least this gives you warning. Also note that a problem still remains: packages are likely to pull in dependencies, and these dependencies are often specified in `>=` terms and can hence change over time. To completely lock down the version of things, have a look at tools like aptly, which allow you to take snapshots of repositories.
- Verify any software or data downloaded from the Internet. This means using checksums or cryptographic signatures. Of all the steps listed here, this is the most important. If you don't verify downloads, you are vulnerable to accidental corruption as well as attackers tampering with downloads. This is particularly important when software is transferred with HTTP, which offers no guarantees against man-in-the-middle attacks. The following section offers specific advice on how to do this.

Most Dockerfiles for the official images provide good examples of using tagged versions and verifying downloads. They also typically use a specific tag of a base image, but do not use version numbers when installing software from package managers.

Securely Downloading Software in Dockerfiles

In the majority of cases, vendors will make signed checksums available for verifying downloads. For example, the Dockerfile for the official Node.js image includes the following:

```
RUN gpg --keyserver pool.sks-keyservers.net \
        --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D \
                    114F43EE0176B71C7BC219DD50A3051F888C628D ❶

ENV NODE_VERSION 0.10.38
ENV NPM_VERSION 2.10.0
RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/node-v\
$NODE_VERSION-linux-x64.tar.gz" \ ❷
&& curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/\
SHASUMS256.txt.asc" \ ❸
&& gpg --verify SHASUMS256.txt.asc \ ❹
&& grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" \
        SHASUMS256.txt.asc | sha256sum -c - ❺
```

❶ Gets the GNU Privacy Guard (GPG) keys used to sign the Node.js download. Here, we do have to trust that these are the correct keys.

❷ Downloads the Node.js tarball.

❸ Downloads the checksum for the tarball.

❹ Uses GPG to verify that the checksum was signed by whoever owns the keys we obtained.

❺ Tests that the checksum matches the tarball by using the `sha256sum` tool.

If either the GPG test or the checksum test fails, the build will abort.

In some cases, packages are available in third-party repositories, which means they can be installed securely by adding the given repository and its signing key. For example, the Dockerfile for the official Nginx image includes the following:

```
RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 \
    --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/\
 jessie nginx" >> /etc/apt/sources.list
```

The first command obtains the signing key for Nginx (which is added to the keystore), and the second command adds the Nginx package repository to the list of repositories to check for software. After this, Nginx can be simply and securely installed with `apt-get install -y nginx` (preferably with a version number).

Assuming no signed package or checksum is available, creating your own is easy. For example, to create a checksum for a Redis release:

```
$ curl -s -o redis.tar.gz \
  http://download.redis.io/releases/redis-3.0.1.tar.gz
$ sha1sum -b redis.tar.gz ❶
fe1d06599042bfe6a0e738542f302ce9533dde88 *redis.tar.gz
```

❶ Here, we're creating a 160-bit SHA-1 checksum. The `-b` flag tells the `sha1sum` utility that we are dealing with binary data, not text.

Once you've tested and verified the software, you can add something like the following to your Dockerfile:

```
RUN curl -sSL -o redis.tar.gz \
        http://download.redis.io/releases/redis-3.0.1.tar.gz \
    && echo "fe1d06599042bfe6a0e738542f302ce9533dde88\
 *redis.tar.gz" | sha1sum -c -
```

This downloads the file as *redis.tar.gz* and asks `sha1sum` to verify the checksum. If the check fails, the command will fail and the build will abort.

Changing all these details for each release is a lot of work if you release often, so automating the process is worthwhile. In many of the official image repositories, you can find `update.sh` scripts for this purpose (for example, *https://github.com/docker-library/wordpress/blob/master/update.sh)*.

Security Tips

This section contains actionable tips on securing container deployments. Not all the advice is applicable to all deployments, but you should become familiar with the basic tools you can use.

Many of the tips describe various ways in which containers can be limited so that containers are unable to adversely affect other containers or the host. The main issue to bear in mind is that the host kernel's resources—CPU, memory, network, UIDs, and so forth—are shared among containers. If a container monopolizes any of these, it will starve out other containers. Worse, if a container can exploit a bug in kernel code, it may be able to bring down the host or gain access to the host and other containers. This could be caused either accidentally, through some buggy programming, or maliciously, by an attacker seeking to disrupt or compromise the host.

Set a USER

Never run production applications as `root` inside the container. That's worth saying again: *never run production applications as root inside the container*. An attacker who breaks the application will have full access to the container, including its data and programs. Worse, an attacker who manages to break out of the container will have `root` access on the host. You wouldn't run an application as root in a VM or on bare metal, so don't do it in a container.

To avoid running as `root`, your Dockerfiles should always create a nonprivileged user and switch to it with a `USER` statement or from an entrypoint script. For example:

```
RUN groupadd -r user_grp && useradd -r -g user_grp user
USER user
```

This creates a group called `user_grp` and a new user called `user` who belongs to that group. The `USER` statement will take effect for all following instructions and when a container is started from the image. You may need to delay the `USER` instruction until later in the Dockerfile if you need to first perform actions that need root privileges such as installing software.

Many of the official images create an unprivileged user in the same way, but do not contain a `USER` instruction. Instead, they switch users in an entrypoint script, using the *gosu* utility. For example, the entry-point script for the official Redis image looks like this:

```
#!/bin/bash
set -e
if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

This script includes the line `chown -R redis .`, which sets the ownership of all files under the images data directory to the `redis` user. If the Dockerfile had declared a `USER`, this line wouldn't work. The next line, `exec gosu redis "$@"`, executes the given `redis` command as the redis user. The use of `exec` means the current shell is replaced with `redis`, which becomes PID 1 and has any signals forwarded appropriately.

Use gosu, not sudo

The traditional tool for executing commands as another user is sudo. While sudo is a powerful and venerable tool, it has some side effects that make it less than ideal for use in entry-point scripts. For example, you can see what happens if you run `sudo ps aux` inside an Ubuntu[9] container:

```
$ docker run --rm ubuntu:trusty sudo ps aux
USER       PID ... COMMAND
root         1     sudo ps aux
root         5     ps aux
```

You have two processes, one for sudo and one for the command you ran.

By contrast, say you install gosu into an Ubuntu image:

```
$ docker run --rm amouat/ubuntu-with-gosu gosu
root ps aux
USER       PID ... COMMAND
root         1     ps aux
```

You have only one process running—gosu has executed the command and got ten out of the way completely. Importantly, the command is running as PID 1, meaning that it will correctly receive any signals sent to the container, unlike the sudo example.

If you have an application that insists on running as `root` (and you can't fix it), consider using tools such as sudo, SELinux (see "SELinux" on page 32), and fakeroot to constrain the process.

Limit Container Networking

A container should open only the ports it needs to use in production, and these ports should be accessible only to the other containers that need them. This is a little trickier than it sounds, because by default, containers can talk to each other whether or not ports have

9 I'm using Ubuntu instead of Debian here, as the Ubuntu image includes sudo by default.

been explicitly published or exposed. You can see this by having a quick play with the `netcat` tool:[10]

```
$ docker run --name nc-test -d \
             amouat/network-utils nc -l 5001 ❶
f57269e2805cf3305e41303eafefaba9bf8d996d87353b10d0ca577acc7...
$ docker run \
   -e IP=$(docker inspect -f \
           {{.NetworkSettings.IPAddress}} nc-test) \
   amouat/network-utils \
           sh -c 'echo -n "hello" | nc -v $IP 5001' ❷
Connection to 172.17.0.3 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ Tells the `netcat` utility to listen to port 5001 and echo any input.

❷ Sends "hello" to the first container using `netcat`.

The second container is able to connect to `nc-test` despite there being no ports published or exposed. You can change this by running the Docker daemon with the `--icc=false` flag. This turns off intercontainer communication completely; no network traffic will be allowed between containers, but they will still be able to publish ports to the host.

If you don't need your containers to communicate at all, this is a great setting, because it prevents an attacker from using a compromised container to attack other containers.

In most cases,[11] you will still want certain groups of containers to communicate. If in addition to `--icc=false`, you pass the `--iptables` flag, then Docker will dynamically add IPtables rules to allow only *linked* containers to communicate on exposed ports.[12] For example, assuming the daemon has been launched with `--icc=false` and `--iptables`:

10 We're using the OpenBSD version here.

11 By this, I mean practically every case.

12 There have been a few bugs related to `--icc=false`, including a couple that hit me while writing, but seem to be solved upstream. If you have problems, try upgrading and manually checking the IPtables rules.

```
$ cat /etc/default/docker | grep DOCKER_OPTS=
DOCKER_OPTS="--iptables=true --icc=false" ❶
$ docker run --name nc-test -d --expose 5001 \
  amouat/network-utils nc -l 5001
d7c267672c158e77563da31c1ee5948f138985b1f451cd2222cf248006491139
$ docker run \
    -e IP=$(docker inspect -f \
            {{.NetworkSettings.IPAddress}} nc-test)
    amouat/network-utils sh -c 'echo -n "hello" \
            | nc -w 2 -v $IP 5001' ❷
nc: connect to 172.17.0.10 port 5001 (tcp) timed out: Ope...
$ docker run \
   --link nc-test:nc-test \
   amouat/network-utils sh -c 'echo -n "hello" \
            | nc -w 2 -v nc-test 5001'
Connection to nc-test 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ On Ubuntu, the Docker daemon is configured by setting `DOCKER_OPTS` in `/etc/default/docker`.

❷ The `-w 2` flag tells `netcat` to time out after 2 seconds.

The first connection fails, as intercontainer communication is off and no link is present. The second command succeeds, as you have added the link. If you want to understand how this works under the hood, try running `sudo iptables -L -n` on the host with and without linked containers.

When publishing ports to the host, Docker publishes to all interfaces (0.0.0.0) by default. You can instead specify the interface you want to bind to explicitly; for example:

```
$ docker run -p 87.245.78.43:8080:8080 -d myimage
```

This reduces the attack surface by allowing only traffic from the given interface.

Remove setuid/setgid Binaries

Chances are that your application doesn't need any `setuid` or `setgid` binaries.[13] If you can disable or remove such binaries, you stop any chance of them being used for privilege escalation attacks.

To get a list of such binaries in an image, try running `find / -perm +6000 -type f -exec ls -ld {} \;`—for example:

```
$ docker run debian find / -perm +6000 -type f -exec \
                         ls -ld {} \; 2> /dev/null
-rwsr-xr-x 1 root root 10248 Apr 15 00:02 /usr/lib/pt_chown
-rwxr-sr-x 1 root shadow 62272 Nov 20  2014 /usr/bin/chage
-rwsr-xr-x 1 root root 75376 Nov 20  2014 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 53616 Nov 20  2014 /usr/bin/chfn
-rwsr-xr-x 1 root root 54192 Nov 20  2014 /usr/bin/passwd
-rwsr-xr-x 1 root root 44464 Nov 20  2014 /usr/bin/chsh
-rwsr-xr-x 1 root root 39912 Nov 20  2014 /usr/bin/newgrp
-rwxr-sr-x 1 root tty 27232 Mar 29 22:34 /usr/bin/wall
-rwxr-sr-x 1 root shadow 22744 Nov 20  2014 /usr/bin/expiry
-rwxr-sr-x 1 root shadow 35408 Aug  9  2014 /sbin/unix_chkpwd
-rwsr-xr-x 1 root root 40000 Mar 29 22:34 /bin/mount
-rwsr-xr-x 1 root root 40168 Nov 20  2014 /bin/su
-rwsr-xr-x 1 root root 70576 Oct 28  2014 /bin/ping
-rwsr-xr-x 1 root root 27416 Mar 29 22:34 /bin/umount
-rwsr-xr-x 1 root root 61392 Oct 28  2014 /bin/ping6
```

You can then "defang" the binaries with `chmod a-s` to remove the suid bit. For example, you can create a defanged Debian image with the following Dockerfile:

```
FROM debian:wheezy

RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true ❶
```

❶ The `|| true` allows you to ignore any errors from `find`.

13 `setuid` and `setgid` binaries run with the privileges of the owner rather than the user. These are normally used to allow users to temporarily run with escalated privileges required to execute a given task, such as setting a password.

Build and run it:

```
$ docker build -t defanged-debian .
...
Successfully built 526744cf1bc1
docker run --rm defanged-debian \
  find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null \
  | wc -l
0
$
```

It's more likely that your Dockerfile will rely on a `setuid`/`setgid` binary than your application. Therefore, you can always perform this step near the end, after any such calls and before changing the user (removing `setuid` binaries is pointless if the application runs with root privileges).

Limit Memory

Limiting memory protects against both DoS attacks and applications with memory leaks that slowly consume all the memory on the host (such applications can be restarted automatically to maintain a level of service).

The `-m` and `--memory-swap` flags to `docker run` limit the amount of memory and swap memory a container can use. Somewhat confusingly, the `--memory-swap` argument sets the *total* amount of memory (memory *plus* swap memory rather than just swap memory). By default, no limits are applied. If the `-m` flag is used but not `--memory-swap`, then `--memory-swap` is set to double the argument to `-m`. This is best explained with an example. Here, you'll use the `amouat/stress` image, which includes the Unix stress utility that is used to test what happens when resources are hogged by a process. In this case, you will tell it to grab a certain amount of memory:

```
$ docker run -m 128m --memory-swap 128m amouat/stress \
     stress --vm 1 --vm-bytes 127m -t 5s ❶
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
$ docker run -m 128m --memory-swap 128m amouat/stress \
     stress --vm 1 --vm-bytes 130m -t 5s ❷
stress: FAIL: [1] (416) <-- worker 6 got signal 9
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 0s
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
$ docker run -m 128m amouat/stress \
```

```
    stress --vm 1 --vm-bytes 255m -t 5s ❸
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
```

❶ These arguments tell the stress utility to run one process that will grab 127 MB of memory and time out after 5 seconds.

❷ This time you try to grab 130 MB, which fails because you are allowed only 128 MB.

❸ This time you try to grab 255 MB, and because `--swap-memory` has defaulted to 256 MB, the command succeeds.

Limit CPU

If an attacker can get one container, or one group of containers, to start using all the CPU on the host, the attacker will be able to starve out any other containers on the host, resulting in a DoS attack.

In Docker, CPU share is determined by a *relative* weighting with a default value of 1024, meaning that by default all containers will receive an equal share of CPU.

The way it works is best explained with an example. Here, you'll start four containers with the `amouat/stress` image you saw earlier, except this time they will all attempt to grab as much CPU as they like, rather than memory.

```
$ docker run -d --name load1 -c 2048 amouat/stress
912a37982de1d8d3c4d38ed495b3c24a7910f9613a55a42667d6d28e1da71fe5
$ docker run -d --name load2 amouat/stress
df69312a0c959041948857fca27b56539566fb5c7cda33139326f16485948bc8
$ docker run -d --name load3 -c 512 amouat/stress
c2675318fefafa3e9bfc891fa303a16e72caf221ec23a4c222c2b889ea82d6e2
$ docker run -d --name load4 -c 512 amouat/stress
5c6e199423b59ae481d41268c867c705f25a5375d627ab7b59c5fbfbcfc1d0e0
$ docker stats $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER           CPU %    ...
/load1              392.13%
/load2              200.56%
/load3              97.75%
/load4              99.36%
```

In this example, the container `load1` has a weighting of 2048, `load2` has the default weighting of 1024, and the other two containers have weightings of 512. On my machine with eight cores and hence a total of 800% CPU to allocate, this results in `load1` getting approxi-

mately half the CPU, `load2` getting a quarter, and `load3` and `load4` getting an eighth each. If only one container is running, it will be able to grab as many resources as it wants.

The relative weighting means that it shouldn't be possible for any container to starve the others with the default settings. However, you may have "groups" of containers that dominate CPU over other containers, in which case, you can assign containers in that group a lower default value to ensure fairness. If you do assign CPU shares, make sure that you bear the default value in mind so that any containers that run without an explicit setting still receive a fair share without dominating other containers.

Limit Restarts

If a container is constantly dying and restarting, it will waste a large amount of system time and resources, possibly to the extent of causing a DoS. This can be easily prevented by using the `on-failure` restart policy instead of the `always` policy, for example:

```
$ docker run -d --restart=on-failure:10 my-flaky-image
...
```

This causes Docker to restart the container up to a maximum of 10 times. The current restart count can be found as `.RestartCount` in `docker inspect`:

```
$ docker inspect -f "{{ .RestartCount }}" $(docker ps -lq)
0
```

Docker employs an exponential back-off when restarting containers. (It will wait for 100 ms, then 200 ms, then 400 ms, and so forth on subsequent restarts.) By itself, this should be effective in preventing DoS attacks that try to exploit the restart functionality.

Limit Filesystems

Stopping attackers from being able to write to files prevents several attacks and generally makes life harder for hackers. They can't write a script and trick your application into running it, or overwrite sensitive data or configuration files.

Starting with Docker 1.5, you can pass the `--read-only` flag to `docker run`, which makes the container's filesystem entirely read-only:

```
$ docker run --read-only debian touch x
touch: cannot touch 'x': Read-only file system
```

You can do something similar with volumes by adding `:ro` to the end of the volume argument:

```
$ docker run -v $(pwd):/pwd:ro debian touch /pwd/x
touch: cannot touch '/pwd/x': Read-only file system
```

The majority of applications need to write out files and won't operate in a completely read-only environment. In such cases, you can find the folders and files that the application needs write access to and use volumes to mount only those files that are writable.

Adopting such an approach has huge benefits for auditing. If you can be sure your container's filesystem is exactly the same as the image it was created from, you can perform a single offline audit on the image rather than auditing each separate container.

Limit Capabilities

The Linux kernel defines sets of privileges—called *capabilities*—that can be assigned to processes to provide them with greater access to the system. The capabilities cover a wide range of functions, from changing the system time to opening network sockets. Previously, a process either had full root privileges or was just a user, with no in-between. This was particularly troubling with applications such as ping, which required root privileges only for opening a raw network socket. This meant that a small bug in the ping utility could allow attackers to gain full root privileges on the system. With the advent of capabilities, it is possible to create a version of ping that has only the privileges it needs for creating a raw network socket rather than full root privileges, meaning would-be attackers gain much less from exploiting utilities like ping.

By default, Docker containers run with a subset of capabilities,[14] so, for example, a container will not normally be able to use devices such as the GPU and sound card or insert kernel modules. To give

14 These are `CHOWN`, `DAC_OVERRIDE`, `FSETID`, `FOWNER`, `MKNOD`, `NET_RAW`, `SETGID`, `SETUID`, `SETFCAP`, `SETPCAP`, `NET_BIND_SERVICE`, `SYS_CHROOT`, `KILL`, and `AUDIT_WRITE`. Dropped capabilities notably include (but are not limited to) `SYS_TIME`, `NET_ADMIN`, `SYS_MODULE`, `SYS_NICE`, and `SYS_ADMIN`. For full information on capabilities, see `man capabilities`.

extended privileges to a container, start it with the `--privileged` argument to `docker run`.

In terms of security, what you really want to do is limit the capabilities of containers as much as you can. You can control the capabilities available to a container by using the `--cap-add` and `--cap-drop` arguments. For example, if you want to change the system time (don't try this unless you want to break things!):

```
$ docker run debian \
         date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
date: cannot set date: Operation not permitted
$ docker run --cap-add SYS_TIME debian \
         date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
$ date
Tue Feb 10 10:00:03 GMT 1981
```

In this example, you can't modify the date until you add the `SYS_TIME` privilege to the container. As the system time is a non-namespaced kernel feature, setting the time inside a container sets it for the host and all other containers as well.[15]

A more restrictive approach is to drop all privileges and add back just the ones you need:

```
$ docker run --cap-drop all debian chown 100 /tmp
chown: changing ownership of '/tmp': Operation not permitted
$ docker run --cap-drop all --cap-add CHOWN debian \
         chown 100 /tmp
```

This represents a major increase in security; an attacker who breaks into a kernel will still be hugely restricted in which kernel calls she is able to make. However, some problems exist:

- How do you know which privileges you can drop safely? Trial and error seems to be the simplest approach, but what if you accidentally drop a privilege that your application needs only rarely? Identifying required privileges is easier if you have a full test suite you can run against the container and are following a

15 If you run this example, you'll have a broken system until you set the time correctly. Try running `sudo ntpdate` or `sudo ntpdate-debian` to change back to the correct time.

microservices approach that has less code and moving parts in each container to consider.

- The capabilities are not as neatly grouped and fine-grained as you may wish. In particular, the `SYS_ADMIN` capability has a lot of functionality; kernel developers seemed to have used it as a default when they couldn't find (or perhaps couldn't be bothered to look for) a better alternative. In effect, it threatens to re-create the simple binary split of admin user versus normal user that capabilities were designed to take us away from.

Apply Resource Limits (ulimits)

The Linux kernel defines resource limits that can be applied to processes, such as limiting the number of child processes that can be forked and the number of open file descriptors allowed. These can also be applied to Docker containers, either by passing the `--ulimit` flag to `docker run` or setting container-wide defaults by passing `--default-ulimit` when starting the Docker daemon. The argument takes two values, a soft limit and a hard limit separated by a colon, the effects of which are dependent on the given limit. If only one value is provided, it is used for both the soft and hard limit.

The full set of possible values and meanings are described in full in `man setrlimit`. (Note that the `as` limit can't be used with containers, however.) Of particular interest are the following values:

cpu
: Limits the amount of CPU time to the given number of seconds. Takes a soft limit (after which the container is sent a `SIGXCPU` signal) followed by a `SIGKILL` when the hard limit is reached. For example, again using the stress utility from "Limit Memory" on page 24 and "Limit CPU" on page 25 to maximize CPU usage:

```
$ time docker run --ulimit cpu=12:14 amouat/stress \
              stress --cpu 1
stress: FAIL: [1] (416) <-- worker 5 got signal 24
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 12s
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

real 0m12.765s
user 0m0.247s
sys 0m0.014s
```

The `ulimit` argument killed the container after it used 12 seconds of CPU.

This is potentially useful for limiting the amount of CPU that can be used by containers kicked off by another process—for example, running computations on behalf of users. Limiting CPU in such a way may be an effective mitigation against DoS attacks in such circumstances.

nofile
: The maximum number of file descriptors[16] that can be concurrently open in the container. Again, this can be used to defend against DoS attacks and ensure that an attacker isn't able to read or write to the container or volumes. (Note that you need to set `nofile` to *one more* than the maximum number you want.) For example:

```
$ docker run --ulimit nofile=5 debian cat /etc/hostname
b874469fe42b
$ docker run --ulimit nofile=4 debian cat /etc/hostname
Timestamp: 2015-05-29 17:02:46.956279781 +0000 UTC
Code: System error

Message: Failed to open /dev/null - open /mnt/sda1/var/...
```

Here, the OS requires several file descriptors to be open, although `cat` requires only a single file descriptor. It's hard to be sure of how many file descriptors your application will need, but setting it to a number with plenty of room for growth offers some protection against DoS attacks, compared to the default of 1048576.

16 A *file descriptor* is a pointer into a table recording information on the open files on the system. An entry is created whenever a file is accessed, recording the mode (read, write, etc.) the file is accessed with and pointers to the underlying files.

nproc
: The maximum number of processes that can be created by the user of the container. On the face of it, this sounds useful, because it can prevent fork bombs and other types of attack. Unfortunately, the `nproc` limits are not set per container but rather for the user of the container across all processes. This means, for example:

```
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
92b162b1bb91af8413104792607b47507071c52a2e3128f0c6c7659bfbb84511
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
158f98af66c8eb53702e985c8c6e95bf9925401c3901c082a11889182bc843cb
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
6444e3b5f97803c02b62eae601fbb1dd5f1349031e0251613b9ff80871555664
FATA[0000] Error response from daemon: Cannot start container
6444e3b5f9780...
[8] System error: resource temporarily unavailable
$ docker run --user 500 -d debian sleep 100
f740ab7e0516f931f09b634c64e95b97d64dae5c883b0a349358c5995806e503
```

The third container couldn't be started, because two processes already belong to UID 500. By dropping the `--ulimit` argument, you can continue to add processes as the user. Although this is a major drawback, `nproc` limits may still be useful in situations where you use the same user across a limited number of containers.

Also note that you can't set `nproc` limits for the `root` user.

Run a Hardened Kernel

Beyond simply keeping your host operating system up-to-date and patched, you may want to consider running a hardened kernel, using patches such as those provided by grsecurity and PaX. PaX provides extra protection against attackers manipulating program execution by modifying memory (such as buffer overflow attacks). It does this by marking program code in memory as nonwritable and data as nonexecutable. In addition, memory is randomly arranged to mitigate against attacks that attempt to reroute code to existing procedures (such as system calls in common libraries). grsecurity is designed to work alongside PaX, and it adds patches related to role-based access control (RBAC), auditing, and other miscellaneous features.

To enable PaX and/or grsec, you will probably need to patch and compile the kernel yourself. This isn't as daunting as it sounds, and plenty of resources are available online to help.

These security enhancements may cause some applications to break. PaX, in particular, will conflict with any programs that generate code at runtime. A small overhead also is associated with the extra security checks and measures. Finally, if you use a precompiled kernel, you will need to ensure that it is recent enough to support the version of Docker you want to run.

Linux Security Modules

The Linux kernel defines the Linux Security Module (LSM) interface, which is implemented by various modules that want to enforce a particular security policy. At the time of writing, several implementations exist, including AppArmor, SELinux, Smack, and TOMOYO Linux. These security modules can be used to provide another level of security checks on the access rights of processes and users, beyond that provided by the standard file-level access control.

The modules normally used with Docker are SELinux (typically with Red Hat-based distributions) and AppArmor (typically with Ubuntu and Debian distributions). We'll take a look at both of these modules now.

SELinux

The SELinux, or *Security Enhanced Linux*, module was developed by the United States National Security Agency (NSA) as an implementation of what they call mandatory access control (MAC), as contrasted to the standard Unix model of discretionary access control (DAC). In somewhat plainer language, there are two major differences between the access control enforced by SELinux and the standard Linux access controls:

- SELinux controls are enforced based on *types*, which are essentially labels applied to processes and objects (files, sockets, and so forth). If the SELinux policy forbids a process of type A to access an object of type B, that access will be disallowed, regardless of the file permissions on the object or the access privileges of the user. SELinux tests occur after the normal file permission checks.

- It is possible to apply multiple levels of security, similar to the governmental model of confidential, secret, and top-secret access. Processes that belong to a lower level cannot read files written by processes of a higher level, regardless of where in the filesystem the file resides or what the permissions on the file are. So a top-secret process could write a file to `/tmp` with `chmod 777` privileges, but a confidential process would still be unable to access the file. This is known as *multilevel security* (MLS) in SELinux, which also has the closely related concept of *multicategory security* (MCS). MCS allows categories to be applied to processes and objects and denies access to a resource if it does not belong to the correct category. Unlike MLS, categories do not overlap and are not hierarchical. MCS can be used to restrict access to resources to subsets of a type (for example, by using a unique category, a resource can be restricted to use by only a single process).

SELinux comes installed by default on Red Hat distributions and should be simple to install on most other distributions. You can check whether SELinux is running by executing `sestatus`. If that command exists, it will tell you whether SELinux is enabled or disabled and whether it is in permissive or enforcing mode. When in permissive mode, SELinux will log access-control infringements but will not enforce them.

The default SELinux policy for Docker is designed to protect the host from containers, as well as containers from other containers. Containers are assigned the default process type `svirt_lxc_net_t`, and files accessible to a container are assigned `svirt_sandbox_file_t`. The policy enforces that containers are able to read and execute files only from `/usr` on the host and cannot write to any file on the host. It also assigns a unique MCS category number to each container, intended to prevent containers from being able to access files or resources written by other containers in the event of a breakout.

NOTE

Enabling SELinux

If you're running a Red Hat-based distribution, SELinux should be installed already. You can check whether it's enabled and is enforcing rules by running `sestatus` on the command line. To enable SELinux and set it to enforcing mode, edit */etc/selinux/config* so that it contains the line `SELINUX=enforcing`.

You will also need to ensure that SELinux support is enabled on the Docker daemon. The daemon should be running with the flag `--selinux-enabled`. If not, it should be added to the file */etc/sysconfig/docker*.

You must be using the devicemapper storage driver to use SELinux. At the time of writing, getting SELinux to work with Overlay and Btrfs is an ongoing effort, but they are not currently compatible.

For installation of other distributions, refer to the relevant documentation. Note that SELinux needs to label all files in your filesystem, which takes some time. Do not install SELinux on a whim!

Enabling SELinux has an immediate and drastic effect on using containers with volumes. If you have SELinux installed, you will no longer be able to read or write to volumes by default:

```
$ sestatus | grep mode
Current mode:                   enforcing
$ mkdir data
$ echo "hello" > data/file
$ docker run -v $(pwd)/data:/data debian cat /data/file
cat: /data/file: Permission denied
```

You can see the reason by inspecting the folder's security context:

```
$ ls --scontext data
unconfined_u:object_r:user_home_t:s0 file
```

The label for the data doesn't match the label for containers. The fix is to apply the container label to the data by using the `chcon` tool, effectively notifying the system that you expect these files to be consumed by containers:

```
$ chcon -Rt svirt_sandbox_file_t data
$ docker run -v $(pwd)/data:/data debian cat /data/file
hello
$ docker run -v $(pwd)/data:/data debian \
         sh -c 'echo "bye" >> /data/file'
```

```
$ cat data/file
hello
bye
$ ls --scontext data
unconfined_u:object_r:svirt_sandbox_file_t:s0 file
```

Note that if you run `chcon` only on the file and not the parent folder, you will be able to read the file but not write to it.

From version 1.7 and on, Docker automatically relabels volumes for use with containers if the `:Z` or `:z` suffix is provided when mounting the volume. The `:z` labels the volume as usable by *all* containers (this is required for data containers that share volumes with multiple containers), and the `:Z` labels the volume as usable by only that container. For example:

```
$ mkdir new_data
$ echo "hello" > new_data/file
$ docker run -v $(pwd)/new_data:/new_data debian \
         cat /new_data/file
cat: /new_data/file: Permission denied
$ docker run -v $(pwd)/new_data:/new_data:Z debian \
         cat /new_data/file
hello
```

You can also use the `--security-opt` flag to change the label for a container or to disable the labeling for a container:

```
$ touch newfile
$ docker run -v $(pwd)/newfile:/file \
         --security-opt label:disable \
         debian sh -c 'echo "hello" > /file'
$ cat newfile
hello
```

An interesting use of SELinux labels is to apply a specific label to a container in order to enforce a particular security policy. For example, you could create a policy for an Nginx container that allows it to communicate on only ports 80 and 443.

Be aware that you will be unable to run SELinux commands from inside containers. Inside the container, SELinux appears to be turned off, which prevents applications and users from trying to run commands such as setting SELinux policies that will get blocked by SELinux on the host.

A lot of tools and articles are available for helping to develop SELinux policies. In particular, be aware of `audit2allow`, which can turn log messages from running in permissive mode into policies

that allow you to run in enforcing mode without breaking applications.

The future for SELinux looks promising; as more flags and default implementations are added to Docker, running SELinux secured deployments should become simpler. The MCS functionality should allow for the creation of secret or top-secret containers for processing sensitive data with a simple flag. Unfortunately, the current user experience with SELinux is not great; newcomers to SELinux tend to watch everything break with "Permission Denied" and have no idea what's wrong or how to fix it. Developers refuse to run with SELinux enabled, leading back to the problem of having different environments between development and production—the very problem Docker was meant to solve. If you want or need the extra protection that SELinux provides, you will have to grin and bear the current situation until things improve.

AppArmor

The advantage and disadvantage of AppArmor is that it is much simpler than SELinux. It should just work and stay out of your way, but cannot provide the same granularity of protection as SELinux. AppArmor works by applying profiles to processes, restricting which privileges they have at the level of Linux capabilities and file access.

If you're using an Ubuntu host, chances are that it is running right now. You can check this by running `sudo apparmor_status`. Docker will automatically apply an AppArmor profile to each launched container. The default profile provides a level of protection against rogue containers attempting to access various system resources, and it can normally be found at */etc/apparmor.d/docker*. At the time of writing, the default profile cannot be changed, as the Docker daemon will overwrite it when it reboots.

If AppArmor interferes with the running of a container, it can be turned off for that container by passing `--security-opt="apparmor:unconfined"` to `docker run`. You can pass a different profile for a container by passing `--security-opt="apparmor:PROFILE"` to `docker run`, where the `PROFILE` is the name of a security profile previously loaded by AppArmor.

Auditing

Running regular audits or reviews on your containers and images is a good way to ensure that your system is kept clean and up-to-date and to double-check that no security breaches have occurred. An audit in a container-based system should check that all running containers are using up-to-date images and that those images are using up-to-date and secure software. Any divergence in a container from the image it was created from should be identified and checked. In addition, audits should cover other areas nonspecific to container-based systems, such as checking access logs, file permissions, and data integrity. If audits can be largely automated, they can run regularly to detect any issues as quickly as possible.

Rather than having to log into each container and examine each individually, you can instead audit the image used to build the container and use `docker diff` to check for any drift from the image. This works even better if you use a read-only filesystem (see "Limit Filesystems" on page 26) and can be sure that nothing has changed in the container.

At a minimum, you should check that the versions of software used are up-to-date with the latest security patches. This should be checked on each image and any files identified as having changed by `docker diff`. If you are using volumes, you will also need to audit each of those directories.

The amount of work involved in auditing can be seriously reduced by running minimal images that contain only the files and libraries essential to the application.

The host system also needs to be audited as you would a regular host machine or VM. Making sure that the kernel is correctly patched becomes even more critical in a container-based system where the kernel is shared among containers.

Several tools are already available for auditing container-based systems, and you can expect to see more in the coming months. Notably, Docker released the Docker Bench for Security tool, which checks for compliance with many of the suggestions from the Docker Benchmark document from the Center for Internet Security (CIS). Also, the open source Lynis auditing tool contains several checks related to running Docker.

Incident Response

Should something bad occur, you can take advantage of several Docker features to respond quickly to the situation and investigate the cause of the problem. In particular, `docker commit` can be used to take a snapshot of the compromised system, and `docker diff` and `docker logs` can reveal changes made by the attacker.

A major question that needs to be answered when dealing with a compromised container is "Could a container breakout have occurred?" Could the attacker have gained access to the host machine? If you believe that this is possible or likely, the host machine will need to be wiped and all containers re-created from images (without some form of attack mitigation in place). If you are sure the attack was isolated to the container, you can simply stop that container and replace it. (*Never* put the compromised container back into service, even if it holds data or changes not in the base image; you simply can't trust the container anymore.)

Effective mitigation against the attack may be to limit the container in some way, such as dropping capabilities or running with a read-only filesystem.

Once the immediate situation has been dealt with and some form of attack mitigation put in place, the compromised image that you committed can be analyzed to determine the exact causes and extent of the attack.

For information on how to develop an effective security policy covering incident response, read CERT's Steps for Recovering from a UNIX or NT System Compromise and the advice given on the ServerFault website.

Future Features

Several Docker features related to security are in the works. Because these features have been prioritized by Docker, they will likely be available by the time you read this.

Seccomp
: The Linux seccomp (or *secure computing mode*) facility can be used to restrict the system calls that can be made by a process. Seccomp is most notably used by web browsers, including both Chrome and Firefox, to sandbox plug-ins. By integrating sec-

comp with Docker, containers can be locked down to a specified set of system calls. The proposed Docker seccomp integration would, by default, deny access to 32-bit system calls, old networks, and various system functions that containers don't typically need access to. In addition, other calls could be explicitly denied or allowed at runtime, for example:

```
$ docker run -d --security-opt seccomp:allow:clock_adjtime ntpd
```

This allows the container to make the `clock_adjtime` system call needed for syncing the system time by using the Network Time Protocol daemon.

User namespacing
: As mentioned previously, a few proposals exist for how to improve the issue of user namespacing, in particular with regard to the `root` user. You can expect to see support for mapping the `root` user to a nonprivileged user on the host soon.

In addition, I would expect to see some consolidation of the various security tools available to Docker, possibly in the form of a security profile for containers. At the moment, a lot of overlap exists between the various security tools and options (for example, file access can be restricted by using SELinux, dropping capabilities, or using the `--read-only` flag).

Conclusion

As you've seen in this report, there are many aspects to consider when securing a system. The primary advice is to follow the principles of defense-in-depth and least privilege. This ensures that even if an attacker manages to compromise a component of the system, that attacker won't gain full access to the system and will have to penetrate further defenses before being able to cause significant harm or access sensitive data.

Groups of containers belonging to different users or operating on sensitive data should run in VMs separate from containers belonging to other users or running publicly accessible interfaces. The ports exposed by containers should be locked down, particularly when exposed to the outside world, but also internally to limit the access of any compromised containers. The resources and functionality available to containers should be limited to only that required by their purpose, by setting limits on their memory usage, filesystem

access, and kernel capabilities. Further security can be provided at the kernel level by running hardened kernels and using security modules such as AppArmor or SELinux.

In addition, attacks can be detected early through the use of monitoring and auditing. Auditing, in particular, is interesting in a container-based system, as containers can be easily compared to the images they were created from in order to detect suspicious changes. In turn, images can be vetted offline to make sure they are running up-to-date and secure versions of software. Compromised containers with no state can be replaced quickly with newer versions.

Containers are a positive force in terms of security because of the extra level of isolation and control they provide. A system using containers properly will be more secure than the equivalent system without containers.

About the Author

Adrian Mouat is a chief scientist at Container Solutions. In the past he has worked on a wide range of software projects, from small web apps to large-scale data-analysis software. He has also authored the Docker book: *Using Docker* (O'Reilly, 2015).