



Red Hat Enterprise Linux Atomic Host 7 Container Security Guide

Container Security Guide

Red Hat Atomic Host Documentation Team

Container Security Guide

Legal Notice

Copyright © 2016 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Building secure containers, security analysis of containers, containers and SELinux

Table of Contents

PART I. CONTAINER SECURITY GUIDE	3
CHAPTER 1. OVERVIEW	4
CHAPTER 2. BUILDING AND SECURING CONTAINERS	5
CHAPTER 3. KEEPING CONTAINERS FRESH AND UPDATEABLE	6
3.1. NEVER PUT NAKED PULLS IN FROM INSTRUCTIONS	6
3.2. USE YUM UPDATE TOOLS	6
3.3. USING DOCKER CACHING TO YOUR ADVANTAGE	6
3.4. ENSURING THE PROVENANCE OF CONTAINERS	7
3.5. LEVERAGING KUBERNETES AND OPENSIFT TO ENSURE THAT CONTAINERS ARE IMMUTABLE	7
CHAPTER 4. CONTAINER ANALYSIS TOOLS	9
4.1. ATOMIC COMMAND	9
CHAPTER 5. LOCKED-DOWN, SECURE FIREFOX IN A CONTAINER	10
CHAPTER 6. DOCKER SELINUX SECURITY POLICY	12
6.1. LIBVIRT SELINUX	12
6.2. MCS - MULTI-CATEGORY SECURITY	12
6.3. LEVERAGING THE DOCKER SELINUX SECURITY MODEL	12
CHAPTER 7. CONTAINER SECURITY PRACTICES	15
7.1. DROPPING KERNEL CAPABILITIES	15
7.2. DROPPING ROOT	15
7.3. EXERCISE CARE IN USING THE --PRIVILEGED FLAG.	15
7.4. SUID CONTENT	15
7.5. TMPFILE	15
CHAPTER 8. LEVERAGING EXTERNAL TECHNOLOGIES	16
8.1. BUILDING SECURITY ZONES WITH VIRTUALIZATION	16
8.2. KUBERNETES AUTHORIZATION AND SECURITY	16
8.3. KUBERNETES NETWORK SECURITY AND ROUTING	16

PART I. CONTAINER SECURITY GUIDE

CHAPTER 1. OVERVIEW

This guide explains how to make your Docker workflow more secure. There is no one-size-fits-all solution to the problem of securing a workflow.

Because there is no single solution for securing your Docker workflow, you need to understand the tools and strategies that you can use to improve your Docker workflow's security. The general strategy for doing this is to reduce the number of potential attack vectors in your Docker infrastructure available to malicious agents.

This guide will provide you with the fundamentals you need to understand the interaction between security practices and the Docker ecosystem. This guide will furnish you with the tools and strategies you need to find the security solution that works best with your Docker workflow.

This is the general strategy for securing containers: avoid untrusted content, keep containers up-to-date, leverage SELinux, minimize attack surface, and use virtualization.

The containerization ecosystem is developing, and it changes fast. This guide is intended to make the containerization ecosystem legible to you in order to help you make the best choices for your own implementation of containerization.

CHAPTER 2. BUILDING AND SECURING CONTAINERS

This chapter describes security concerns involving the building and distribution of Docker containers.

Docker's Unsigned Binary

Docker's official binary installation is not signed.

The Dangers of Untrusted Content

The process of installing RPMs involves two phases: (1) the retrieval phase, and (2) the installation phase. This separation between retrieval and installation does not exist in the Docker workflow. There are a number of CVEs related to this issue. Docker images are stored as tars, and they can escape the docker daemon without your knowing it.

- ✦ docker pull is an active process - unlike RPMs, there is no separation between the retrieval phase of installation and the installation phase
- ✦ docker containers run as root - you should run Docker content that has originated only from trusted vendors (that includes Red Hat, and in fact is primarily Red Hat).

CHAPTER 3. KEEPING CONTAINERS FRESH AND UPDATEABLE

This section describes processes and practices that ensure that containers are fresh and updateable.

3.1. NEVER PUT NAKED PULLS IN FROM INSTRUCTIONS

When writing Dockerfiles, always list the registry from which you're pulling in the FROM commands you use. In Red Hat's case, that means that you have to include the whole name of the Red Hat Container registry.

This is a naked pull:

```
$ docker pull rhel7
```

This is not a naked pull:

```
$ docker pull registry.redhat.com/rhel7
```

3.2. USE YUM UPDATE TOOLS

In Dockerfiles, always run **yum update && yum clean all** to ensure that you have the freshest base image. These commands update the RPM content associated with the base image. This reduces the exposed attack surface.

3.3. USING DOCKER CACHING TO YOUR ADVANTAGE

This section explains how to use Docker caching to make your Dockerfiles more efficient for your workflow.

3.3.1. Order Instructions to Take Advantage of Caching

Docker assumes that each instruction is deterministic. Docker assumes that these instructions are not associative. If it encounters the same instructions in the same order, it caches the results. This means that if you have the instruction **FROM foo: dnf -y update** in the same two Dockerfiles in the same order, Docker will create the same base image from that point.

Always perform installations in Dockerfiles in the same order to take advantage of Docker caching. Break tasks into thematic components (for example "adding users" or "updating software") in order to take best advantage of Docker's caching utilities.

3.3.2. Deliberately Break Caching In Some Situations

Sometimes it is advantageous to subvert Docker's default caching behavior.

There are three methods to force "yum update" to run:

- (1) On a single machine, delete and recreate the image.
- (2) Insert a nonce command (for example "echo nonce") in order to break caching

(3) Use a buildsystem like the OpenShift buildsystem

3.4. ENSURING THE PROVENANCE OF CONTAINERS

Running a private registry is the easiest way to ensure container provenance. OpenShift and Satellite are Red Hat products that have built-in image service options. Avoid including sensitive information in images. If anyone overhears them, it won't be a problem. This means that you should always use TLS (transport layer security), but you don't need to use authentication. Things that need to remain confidential should be abstracted up a level to the orchestration level, which we will discuss in the Kubernetes and OpenShift sections of this document.

3.4.1. Distributing Docker Images Without a Registry

Docker images can be distributed as tarballs that are created with the "docker save" command and loaded with the "docker load" command. These

3.5. LEVERAGING KUBERNETES AND OPENSIFT TO ENSURE THAT CONTAINERS ARE IMMUTABLE

Immutable containers are containers that have no state.

Immutability improves security by decreasing the damage that can be done by a local compromise. Immutable images themselves have no secrets and save no state that could get corrupted. Immutable containers are trivial to verify because they never change.

3.5.1. Ways to Leverage Kubernetes and OpenShift

This section describes ways to leverage Kubernetes and OpenShift to create immutable container images.

1. **Using Volume Mounting** - Use Kubernetes to bring in external data that is mutable (for instance, WordPress content or a database).
2. **Using Services** - Kubernetes and OpenShift have matchmaking services. A container can be designed to depend generically on a database, and the details of logging into that database can be provided at runtime.
3. **Using Templating** - This is the same idea as using the matchmaking services of Kubernetes and Openshift applied to buildtime. Dockerfiles provide no way of making users with a certain UID run on a certain cluster. Use plugins to the OpenShift buildsystem to customize a given build.
4. **Using Github Repositories** - Use docker pull to pull in live content at runtime, for example from a private or public git repository. OpenShift has features that take this to another level: they allow you to ignore all the container details and have an application all of whose details are hosted in a github repo.

3.5.2. Leveraging Kubernetes to Ensure that Containers Do Not Store Secrets or Sensitive Information

Kubernetes has a "secrets" functionality that allows memory-hosted secrets to be inserted as virtual files at runtime. This should be used for all confidential information such as authentication details and encryption keys.

CHAPTER 4. CONTAINER ANALYSIS TOOLS

This section describes tools for the analysis of containers.

4.1. ATOMIC COMMAND

If you use the atomic command, you'll know which layers went into your images, and if any of those layers have been updated, you will know that you should rebuild your image.

CHAPTER 5. LOCKED-DOWN, SECURE FIREFOX IN A CONTAINER

This section explains how to deploy a secure container that runs Firefox. This container gives you an instance of Firefox, containerized, with the following features:

- ✧ Completely unprivileged — needs no extra SELinux tweaking
- ✧ Only the list of cgroups is passed into the container from the host
- ✧ No port redirection because the container is available only to the host
- ✧ No X11 clipboard events or X events shared with your real host
- ✧ No shared sound hardware
- ✧ Everything runs with normal, non-elevated user permissions except for systemd (and systemd runs only to reap the other processes)
- ✧ unsynced sound, flash, and interactivity.

Running Firefox Securely in a Container

1. Retrieve the base image that we use to build this container:

```
$ curl -o Fedora-Docker-Base-22-20150521.x86_64.tar.xz -L
https://download.fedoraproject.org/pub/fedora/linux/releases/22/D
ocker/x86_64/Fedora-Docker-Base-22-20150521.x86_64.tar.xz
```

2. Load the base image you just downloaded into the local Docker registry:

```
$ sudo docker load < Fedora-Docker-Base-22-20150521.x86_64.tar.xz
```

3. Create a directory to hold the Dockerfile that will map out this container:

```
$ mkdir -p isolated_firefox
```

4. Retrieve the Dockerfile by using this curl command:

```
$ curl -o isolated_firefox/Dockerfile -L
http://pastebin.com/raw.php?i=cgYXQvJu
```

5. Build the container and tag it with a tag called **isolated_firefox**:

```
$ sudo docker build -t isolated_firefox isolated_firefox .
```

6. Run the container:

```
$ sudo docker run -v /sys/fs/cgroup:/sys/fs/cgroup:ro
isolated_firefox
```

7. Retrieve the CONTAINER_ID by using the docker ps command:

```
$ sudo docker ps
```

- Retrieve the IP address of the container:

```
$ sudo docker inspect CONTAINER_ID | grep IPAddress\":
```

- Open the container in vncviewer:

```
$ vncviewer CONTAINER_IP
```

- To hear the audio associated with this container, open a browser and go to the following location:

```
http://CONTAINER_IP:8000/firefox.ogg
```

**Note**

Do not forget to include the port in the URL. That means that you should not forget to type **:8000** after the URL. You can also send the address of the container to VLC to play the content in VLC.

- Run the following command to launch the VLC instance:

```
$ vlc http://CONTAINER_IP:8000/firefox.ogg
```

CHAPTER 6. DOCKER SELINUX SECURITY POLICY

The Docker SELinux security policy is similar to the libvirt security policy and is based on the libvirt security policy.

The libvirt security policy is a series of SELinux policies that defines two ways of isolating virtual machines. Generally, virtual machines are prevented from accessing parts of the network. Specifically, individual virtual machines are denied access to one another's resources. Red Hat extends the libvirt-SELinux model to Docker. The Docker SELinux role and Docker SELinux types are based on libvirt. For example, by default, Docker has access to `/usr/var/` and some other locations, but it has complete access to things that are labeled with `svirt_sandbox_file_t`.

https://www.mankier.com/8/docker_selinux - this explains the entire Docker SELinux policy. It is not in layman's terms, but it is complete.

`svirt_sandbox_file_t`

```
system_u:system_r:svirt_lxc_net_t:s0:c186,c641
```

```

^          ^          ^          ^          ^--- unique category
|          |          |          |----- secret-level 0
|          |          |----- a shared type
|          |---SELinux role
|----- SELinux user

```

If a file is labeled `svirt_sandbox_file_t`, then by default all containers can read it. But if the containers write into a directory that has `svirt_sandbox_file_t` ownership, they write using their own category (which in this case is "c186,c641"). If you start the same container twice, it will get a new category the second time (a different category than it had the first time). The category system isolates containers from one another.

Types can be applied to processes and to files.

6.1. LIBVIRT SELINUX

libvirt SELinux — RH is extending libvirt SELinux for Docker. The extension of libvirt SELinux to Docker is substantially RH-contributed work.

6.2. MCS - MULTI-CATEGORY SECURITY

MCS - Multi-Category Security - this is similar to Multi-Level Authentication. Each container is given a unique ID at startup, and each file that a container writes carries that unique ID. Although this is an opt-in system, failure to make use of it means that you will have no isolation between containers. If you do not make use of MCS, you will have isolation between containers and the host, but you will not have isolation of containers from one another. That means that one container could access another container's files.

<https://securityblog.redhat.com/2015/04/29/container-security-just-the-good-parts/> - this will be used later to build the MCS example that we will include in the MCS.

6.3. LEVERAGING THE DOCKER SELINUX SECURITY MODEL

properly labeling content - By default, docker gets access to everything in **/usr** and most things in **/etc**. To give docker access to more than that, relabel content on the host. To restrict access to things in **/usr** or things in **/etc**, relabel them. If you want to restrict access to only one or two containers, then you'll need to use the opt-in MCS system.

important booleans and other restrictions - "privileged" under docker is not really privileged. Even privileged docker processes cannot access arbitrary socket files. An selinux boolean, **docker_connect_any**, makes it possible for privileged docker processes to access arbitrary socket files. Even if run privileged, docker is restricted by the booleans that are in effect.

docker_connect_any - see "important booleans and other restrictions"

<https://github.com/rhatdan/docker-selinux> - dwalsh's github repository.

restricting kernel capabilities - docker supports two commands as part of "docker run": (1) "--cap-add=" and (2) "--cap-drop=". these allow us to add and drop kernel capabilities to and from the containers. root powers have been broken up into a number of groups of capabilities (for instance "cap-chown", which lets you change the ownership of files). by default, docker has a very restricted list of capabilities (provide this restricted list here as well as a datestamp communicating the date of the list's compilation).

<http://linuxmanpages.net/manpages/fedora21/man7/capabilities.7.html> - This provides more information about capabilities. Capabilities constitute the heart of the isolation of containers. If you have used capabilities in the manner described in this guide, an attacker who does not have a kernel exploit will be able to do nothing even if they have root on your system.

restricting kernel calls with seccomp - This is a kernel call that renounces capabilities. A seccomp call that has no root capabilities will make the call to the kernel. A capable process creates a restricted process that makes the kernel call. "seccomp" is an abbreviation of "secure computing mode".

<http://man7.org/linux/man-pages/man2/syscalls.2.html> - **seccomp** is even more fine-grained than capabilities. This feature restricts the kernel calls that containers can make. This is useful for general security reasons, because (for instance) you can prevent a container from calling "cd". Almost all kernel exploits rely on making kernel calls (usually to rarely used parts of the kernel). With seccomp you can drop lots of kernel calls, and dropped kernel calls can't be exploited as attack vectors.

docker network security and routing - By default, docker creates a virtual ethernet card for each container. Each container has its own routing tables and iptables. When specific ports are forwarded, docker creates certain host iptables rules. The docker daemon itself does some of the proxying. If you map applications to containers, you provide flexibility to yourself by limiting network access on a per-application basis. Because containers have their own routing tables, they can be used to limit incoming and outgoing traffic: use the ip route command in the same way you would use it on a host.

scenario: using a containerized firewall to segregate a particular kind of internet traffic from other kinds of internet traffic. This is an easy and potentially diverting exercise for the reader, and might involve concocting scenarios in which certain kinds of traffic are to be kept separate from an official network (one that is constrained, for instance, by the surveillance of a spouse or an employer).

cgroups - "control groups". cgroups provides the core functionality that permits docker to work. In its original implementation, cgroups controlled access only to resources like the CPU. You could put a process in a cgroup, and then instruct the kernel to give that cgroup only up to 10 percent of the cpu. This functions as a kind of a way of providing SLA or quota.

By default, docker creates a unique cgroup for each container. If you have existing cgroup policy on the docker daemon host, you can make use of that existing cgroup policy to control the resource consumption of the specified container.

freezing and unfreezing a container - You can completely stall a container in the state that it is in a any given moment, and then restart it at that point later. This is done by giving the container zero percent cpu. cgroups is the protection that docker provides against ddos attacks. We could host a service on a machine and give it a cgroup priority so that the service can never get less than ten percent of the cpu: then if other services became compromised, they would be unable to stall out the service, because the service is guaranteed to get a minimum of ten percent of the cpu. This makes it possible to ensure that essential processes need never relinquish control of a part of the cpu, no matter how strongly they are attacked.

CHAPTER 7. CONTAINER SECURITY PRACTICES

This chapter describes measures that you can take to protect the security of your containers.

7.1. DROPPING KERNEL CAPABILITIES

Kernel capabilities can be dropped from the CLI of Docker and in the `.json` file of Kubernetes. Start with nothing and add what you need — this is the safest method of determining which kernel capabilities you need.

7.2. DROPPING ROOT

Never have long-running containers running as root, with the exception of **systemd**. There are a couple of approaches for doing this: if you don't need `systemd`, you can use `exec` to become root and drop your user privileges. Or you can use `systemd` to launch the application you want inside the container with a unit file, and the unit file can be used to designate the desired services as non-root.

7.3. EXERCISE CARE IN USING THE `--PRIVILEGED` FLAG.

Unless your container needs access to the host's hardware, you should not use `--privileged`.

7.4. SUID CONTENT

Sticky-bit content (sticky UID content). This is a word-to-the-wise-type note. It is possible to make use of `suid` content without a privileged container, but it is not easy to create `suid` content without a privileged container because marking something as `suid` is a privileged operation.

7.5. TMPFILE

A `tmpfile` is a file that gets created in `/tmp`. This section refers to `tmpfile` attacks. Containers are susceptible to the same race conditions and attacks that non-container `tmpfiles` are susceptible to. Make sure that `tmpfile` content ownership is properly restricted.

CHAPTER 8. LEVERAGING EXTERNAL TECHNOLOGIES

This chapter explains how to build security zones with virtualization and using Kubernetes Network Security and Routing.

8.1. BUILDING SECURITY ZONES WITH VIRTUALIZATION

Dan Walsh had this idea, and there might eventually be a blog post about it, but I'm disinclined to bug him about it. As we all know from Dan's teachings, "containers don't contain" (meaning that containers are open to local privilege escalation attacks, and VMs are not). We will use Kubernetes to create "apartment complexes" (I know that this is a non-standard phrase, but this is from Dan's three-little-pigs example) that can keep different kinds of running code separate. For instance, you will have a VM for applications of a certain security level. If you are a law firm, you might have a VM per law-firm client (meaning an entity that hires a law firm, not the "client" role in a "client-server" architecture) with whom you work, and your associates work within containers running inside that VM (this means that if someone breaks into the VM, the target surface is only the data in that VM alone, not the data pertaining to other cases). Mesos is a resource management application from Apache that can run on top of Kubernetes that we may deal with here.

8.2. KUBERNETES AUTHORIZATION AND SECURITY

This is an area of investigation. Authorization, logging, and client certificate verification must all be investigated.

8.3. KUBERNETES NETWORK SECURITY AND ROUTING

The biggest concern facing Kubernetes is protecting the Kubernetes cluster from the containers themselves. The undesirable situation involves a container using its network access to ask Kubernetes to spawn an unauthorized container, and Kubernetes haplessly complying with that request. As part of "defense in depth", we'll investigate putting passwords and logs and certificates on Kubernetes as well as stopping these kinds of undesired attacks at the network level. If the container never needs to access Kubernetes features, then Kubernetes should be blocked from making networking-related requests. You can create containers that get what they need from Kubernetes early in their lifecycles and then close off their own firewalls, thereby closing down a potential attack vector and minimizing the attack surface of the container and of the environment.