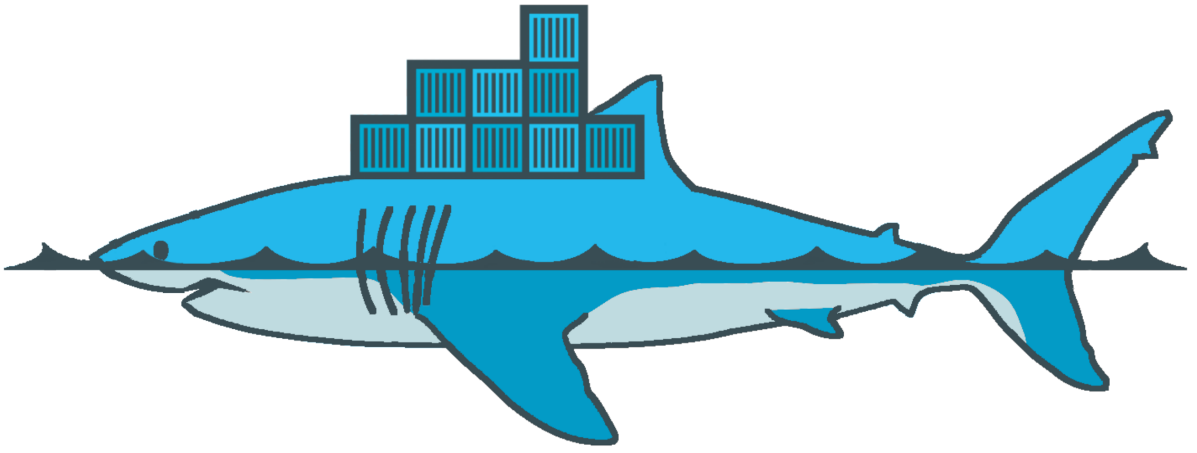
Docker Security
Quick Reference



For
DevOps Engineers

Kim Carter

Docker Security - Quick Reference

For DevOps Engineers

Kim Carter

This book is for sale at http://leanpub.com/dockersecurity-quickreference

This version was published on 2018-04-28

Tweet This Book!

Please help Kim Carter by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Kims new book #DockerSecurityQuickReference looks pretty good!

The suggested hashtag for this book is #dockersecurityquickreference.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#dockersecurityquickreference

Contents

Preface

Description

I usually use bruce schneier's Sensible Security Model when threat modelling. With this book I've focussed on identifying and addressing the risks and countermeasures only, that I see most commonly in Docker deployments. This book is targeted toward the DevOps and Software Engineer in a hurry.

Purpose

My intention with this book is for it to serve as a reference to help you increase the likelihood of resisting attacks from your adversaries targeting your Docker based applications and services, at the same time not getting bogged down in hundreds of pages of content.

I've found the content of this book most useful when read in conjunction with the Cisecurity Benchmark for Docker, I would recommend you also read both items together.

Please contact me if you don't understand something within the book at: binarymist.io/#contact. If you find any errors, please:

- Submit a pull request at github.com/binarymist/dockersecurity-quickreference/pulls or
- Create an issue at github.com/binarymist/dockersecurity-quickreference/issues

I would also love to hear how the book has helped you. Please leave a comment on this book's blog post: https://binarymist.io/blog/2018/03/31/docker-security/#comments

Reason

Colleagues from my tech communities encouraged me to put on paper what was in my head around Docker security. I also hosted an interview with Docker Security Team Lead Diogo Monica, in which I performed a couple of weeks research. I thought it made sense

to consolidate as much of my knowledge and experience around Docker security as I had quickly accessible, then to pass it on to you to help improve the security posture of your Docker deployments.

Introduction

Docker Security - Quick Reference addresses the issues that I have had most commonly with Docker deployments requiring a decent level of security.

One aspect, in-fact, almost certainly the most important regarding Docker security, is that Software Developers often neglect application security. I've written about application security extensively in other books, so I've taken the assumption that you understand it to be of utmost importance when it comes to defending you and your organisations assets, and that you have already invested significantly into application security..

I hope this book serves you well in providing the knowledge you will need to harden your Docker deployments.

I blog at binarymist.io/blog.

If you require any help with improving the security stature of your Development Teams, review the services that I can offer at binarymist.io/#services

1. Habitat

With the continual push for shorter development cycles, combined with continuous delivery, as well as cloud and virtual based infrastructure, containers have become an important part of the continuous delivery pipeline. Docker has established itself as a top contender in this space.

Many of Docker's defaults favour ease of use over security, in saying that, Docker's security considerations follow closely. After working with Docker, the research I have performed in writing these sections on Docker security, while having the chance to discuss many of my concerns and ideas with the Docker Security team lead, Diogo Mónica, it is my belief that, by default, Docker containers, infrastructure and orchestration provide better security than running your applications in Virtual Machines (VMs). Just be careful when comparing containers with VMs, as this is analogous with comparing apples to oranges.

Docker security provides immense configurability to improve its security posture many times over better than defaults. In order to do this properly, you will have to invest some time and effort into learning about the possible issues, features, and how to configure them. I have attempted to illuminate this specifically in these sections on Docker security.

Docker security is similar to VPS security, except there is a much larger attack surface. This is most noteworthy when running many containers with different packages, many of which do not receive timely security updates, as noted by banyan and the morning paper.

A monolithic kernel, such as the Linux kernel, which contains tens of millions of lines of code, and can be reached by untrusted applications via all sorts of networking, USB, and driver APIs, has a huge attack surface. Adding Docker into the mix has the potential to expose all these vulnerabilities to each and every running container, and its applications within, thus making the attack surface of the kernel grow exponentially.

Docker leverage's many features that have been in the Linux kernel for years, which provide many security enhancements out of the box. The Docker Security Team are working hard to add additional tooling and techniques to further harden their components, this has become obvious as I have investigated many of them. You still need to know what all the features, tooling and techniques are, and how to use them, in order to determine whether your container security is adequate for your needs.

From the Docker overview, it states: "*Docker provides the ability to package and run an application in a loosely isolated environment*". Later in the same document it says: "*Each*

container is an isolated and secure application platform, but can be given access to resources running in a different host or container" leaving the "loosely" out. It continues to say: "*Encapsulate your applications (and supporting components) into Docker containers*". The meaning of encapsulate is to enclose, but if we are only loosely isolating, then we're not really enclosing are we? I will address this concern in the following Docker sections and subsections.

To start with, I am going to discuss many areas where we can improve container security. At the end of this Docker section I will discuss why application security is of far more concern than container security.

It is my intent to provide a high level over view of the concepts you will need to know in order to create a secure environment for the core Docker components, and your containers. There are many resources available, and the Docker security team is hard at work constantly trying to make the task of improving security around Docker easier.

Do not forget to check the Additional Resources section for material to be consumed in parallel with the Docker Countermeasures, such as the excellent CIS Docker Benchmark, and the interview I conducted with the Docker Security Team Lead Diogo Mónica.

CISecurity has an excellent resource for hardening docker images, which the Docker Security team helped with.

Consumption from Registries

Exploitability: AVERAGE | Prevalence: VERY WIDESPREAD | Detectability: EASY | Impact: MODERATE

This is a similar concept to that of consuming free and open source, which the OWASP A9 (Using Components with Known Vulnerabilities) addresses. Many of us trust the images on Docker hub without much consideration for the possibly defective packages within. There have been quite a few reports with varying numbers of vulnerable images as noted by Banyan and "the morning paper" mentioned above.

The Docker Registry project is an open-source server side application that lets you store and distribute Docker images. You could run your own registry as part of your organisation's Continuous Integration (CI) / Continuous Delivery (CD) pipeline. Some of the public known instances of the registry are:

- Docker Hub
- EC2 Container Registry

- Google Container Registry
- CoreOS quay.io

Prevention: AVERAGE

"*Docker Security Scanning is available as an add-on to Docker hosted private repositories on both Docker Cloud and Docker Hub.*". You also have to opt in and pay for it. Docker Security Scanning is also now available on the new Enterprise Edition. The scan compares the SHA of each component in the image with those in an up to date CVE database for known vulnerabilities. This is a good start, but not free and does not do enough. Images are scanned on push and the results indexed so that when new CVE databases are available, comparisons can continue to be made.

It's up to the person consuming images from docker hub to assess whether or not they have vulnerabilities. Whether unofficial or official, it is your responsibility. Check the Hardening Docker Host, Engine and Containers section for tooling to assist with finding vulnerabilities in your Docker hosts and images.

Your priority before you start testing images for vulnerable contents, is to understand the following:

1. Where your image originated from
2. Who created it
3. Image provenance: Is Docker fetching the image we think it is?
    1. Identification: How Docker uses secure hashes, or digests.
       Image layers (deltas) are created during the image build process, and also when commands within the container are run, which produce new or modified files and/or directories.
       Layers are now identified by a digest which looks like: `sha256:<the-hash>`
       The above hash element is created by applying the SHA256 hashing algorithm to the layers content.
       The image ID is also the hash of the configuration object which contains the hashes of all the layers that make up the images copy-on-write filesystem definition, also discussed in my Software Engineering Radio show with Diogo Mónica.
    2. Integrity: How do you know that your image has not been tampered with?
       This is where secure signing comes in with the Docker Content Trust feature. Docker Content Trust is enabled through an integration of Notary into the Docker Engine. Both the Docker image producing party and image consuming

party need to opt-in to use Docker Content Trust. By default, it is disabled. In order to do that, Notary must be downloaded and setup by both parties, and the `DOCKER_CONTENT_TRUST` environment variable must be set to `1`, and the `DOCKER_CONTENT_TRUST_SERVER` must be set to the URL of the Notary server you setup.
Now the producer can sign their image, but first, they need to generate a key pair. Once they have done so, when the image is pushed to the registry, it is signed with their private (tagging) key.
When the image consumer pulls the signed image, Docker Engine uses the publisher's public (tagging) key to verify that the image you are about to run is cryptographically identical to the image the publisher pushed.
Docker Content Trust also uses the Timestamp key when publishing the image, this makes sure that the consumer is getting the most recent image on pull.
Notary is based on a Go implementation of The Update Framework (TUF)

3. By specifying a digest tag in a `FROM` instruction in your `Dockerfile`, when you `pull` the same image will be fetched.

Doppelganger images

Exploitability: AVERAGE | Prevalence: COMMON | Detectability: AVERAGE | Impact: SEVERE

Beware of doppelganger images that will be available for all to consume, similar to doppelganger packages that I discuss in the Web Applications chapter of Fascicle 1 of my book Holistic Info-Sec for Web Developers, these can contain a huge number of packages and code that can be used to hide malware in a Docker image.

People often miss-type what they want to install. Attackers often take advantage of this by creating malicious packages with very similar names. Some of the actions could be: having consumers of your package destroy or modify their systems, send sensitive information to the attacker, or any number of other malicious activities.

Prevention: AVERAGE

If you are already performing step 3 from above, then fetching an image with a very similar name becomes highly unlikely, but it pays to be aware of these types of techniques that attackers use.

The Default User is Root

Exploitability: EASY | Prevalence: COMMON | Detectability: VERY EASY | Impact: MODERATE

What is worse, Docker's default is to run containers, and all commands / processes within a container as root. This can be seen by running the following command from the CIS_-Docker_1.13.0_Benchmark:

Query User running containers

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: User={{ .Config.User }}'
```

If you have two containers running, and the user has not been specified, you will see something like the below, which means your two containers are running as root.

Result of user running containers output

```
<container n Id>: User=
<container n+1 Id>: User=
```

Images derived from other images inherit the same user defined in the parent image explicitly or implicitly, so unless the image creator has specifically defined a non-root user, the user will default to root. That means all processes within the container will run as root.

Prevention: VERY EASY

In order to run containers as a non-root user, the user needs to be added in the base image (`Dockerfile`) if it is under your control, and set before any commands you want run as a non-root user. Here is an example of the NodeGoat image:

NodeGoat Dockerfile

```
1  FROM node:4.4
2
3  # Create an environment variable in our image for the non-root user we want to use.
4  ENV user nodegoat_docker
5  ENV workdir /usr/src/app/
6
7  # Home is required for npm install. System account with no ability to login to shell
8  RUN useradd --create-home --system --shell /bin/false $user
9
10 RUN mkdir --parents $workdir
11 WORKDIR $workdir
12 COPY package.json $workdir
13
14 # chown is required by npm install as a non-root user.
15 RUN chown $user:$user --recursive $workdir
16 # Then all further actions including running the containers should
17 # be done under non-root user, unless root is actually required.
18 USER $user
19
20 RUN npm install
21 COPY . $workdir
22
23 # Permissions need to be reapplied, due to how docker applies root to new files.
24 USER root
25 RUN chown $user:$user --recursive $workdir
26 RUN chmod --recursive o-wrx $workdir
27
28 RUN ls -liah
29 RUN ls ../ -liah
30 USER $user
```

As you can see on line 4 we create our `nodegoat_docker` user.
On line 8 we add our non-root user to the image with no ability to login.
On line 15 we change the ownership of the `$workdir` so our non-root user has access to do the things that we normally have permissions to do without root, such as installing npm packages and copying files, as we see on line 20 and 21. But first we need to switch to our non-root user on line 18. On lines 25 and 26 we need to reapply ownership and permissions due to the fact that docker does not `COPY` according to the user you are set to run commands as.

Without reapplying the ownership and permissions of the non-root user as seen above on lines 25 and 26, the container directory listings would look like this:

No reapplication of ownership and permissions

```
Step 12 : RUN ls -liah
 ---> Running in f8692fc32cc7
total 116K
13 drwxr-xr-x   9 nodegoat_docker nodegoat_docker 4.0K Sep 13 09:00 .
12 drwxr-xr-x   7 root            root            4.0K Sep 13 09:00 ..
65 drwxr-xr-x   8 root            root            4.0K Sep 13 08:59 .git
53 -rw-r--r--   1 root            root             178 Sep 12 04:22 .gitignore
69 -rw-r--r--   1 root            root            1.9K Nov 21  2015 .jshintrc
61 -rw-r--r--   1 root            root              55 Nov 21  2015 .nodemonignore
58 -rw-r--r--   1 root            root             715 Sep 13 08:59 Dockerfile
55 -rw-r--r--   1 root            root            6.6K Sep 12 04:16 Gruntfile.js
60 -rw-r--r--   1 root            root             11K Nov 21  2015 LICENSE
68 -rw-r--r--   1 root            root              48 Nov 21  2015 Procfile
64 -rw-r--r--   1 root            root            5.6K Sep 12 04:22 README.md
56 drwxr-xr-x   6 root            root            4.0K Nov 21  2015 app
66 -rw-r--r--   1 root            root             527 Nov 15  2015 app.json
54 drwxr-xr-x   3 root            root            4.0K May 16 11:41 artifacts
62 drwxr-xr-x   3 root            root            4.0K Nov 21  2015 config
57 -rw-r--r--   1 root            root             244 Sep 13 04:51 docker-compose.yml
67 drwxr-xr-x 498 root            root             20K Sep 12 03:50 node_modules
63 -rw-r--r--   1 root            root            1.4K Sep 12 04:22 package.json
52 -rw-r--r--   1 root            root            4.6K Sep 12 04:01 server.js
59 drwxr-xr-x   4 root            root            4.0K Nov 21  2015 test
 ---> ad42366b24d7
Removing intermediate container f8692fc32cc7
Step 13 : RUN ls ../ -liah
 ---> Running in 4074cc02dd1d
total 12K
12 drwxr-xr-x  7 root            root            4.0K Sep 13 09:00 .
11 drwxr-xr-x 32 root            root            4.0K Sep 13 09:00 ..
13 drwxr-xr-x  9 nodegoat_docker nodegoat_docker 4.0K Sep 13 09:00 app
```

With reapplication of the ownership and permissions of the non-root user, as the `Dockerfile` is currently above, the container directory listings look like the following:

With reapplication of ownership and permissions

```
Step 15 : RUN ls -liah
 ---> Running in 8662e1657d0f
total 116K
13 drwxr-x---   21 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 .
12 drwxr-xr-x    9 root            root            4.0K Sep 13 08:51 ..
65 drwxr-x---   20 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 .git
53 -rw-r-----    1 nodegoat_docker nodegoat_docker  178 Sep 12 04:22 .gitignore
69 -rw-r-----    1 nodegoat_docker nodegoat_docker 1.9K Nov 21  2015 .jshintrc
61 -rw-r-----    1 nodegoat_docker nodegoat_docker   55 Nov 21  2015 .nodemonignore
58 -rw-r-----    1 nodegoat_docker nodegoat_docker  884 Sep 13 08:46 Dockerfile
55 -rw-r-----    1 nodegoat_docker nodegoat_docker 6.6K Sep 12 04:16 Gruntfile.js
60 -rw-r-----    1 nodegoat_docker nodegoat_docker  11K Nov 21  2015 LICENSE
68 -rw-r-----    1 nodegoat_docker nodegoat_docker   48 Nov 21  2015 Procfile
64 -rw-r-----    1 nodegoat_docker nodegoat_docker 5.6K Sep 12 04:22 README.md
56 drwxr-x---   14 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 app
66 -rw-r-----    1 nodegoat_docker nodegoat_docker  527 Nov 15  2015 app.json
54 drwxr-x---    5 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 artifacts
62 drwxr-x---    5 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 config
57 -rw-r-----    1 nodegoat_docker nodegoat_docker  244 Sep 13 04:51 docker-compose.yml
67 drwxr-x--- 1428 nodegoat_docker nodegoat_docker  20K Sep 13 08:51 node_modules
63 -rw-r-----    1 nodegoat_docker nodegoat_docker 1.4K Sep 12 04:22 package.json
52 -rw-r-----    1 nodegoat_docker nodegoat_docker 4.6K Sep 12 04:01 server.js
59 drwxr-x---    8 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 test
 ---> b88d816315b1
Removing intermediate container 8662e1657d0f
Step 16 : RUN ls ../ -liah
 ---> Running in 0ee2dcc889a6
total 12K
12 drwxr-xr-x  9 root            root            4.0K Sep 13 08:51 .
11 drwxr-xr-x 34 root            root            4.0K Sep 13 08:51 ..
13 drwxr-x--- 21 nodegoat_docker nodegoat_docker 4.0K Sep 13 08:51 app
```

An alternative to setting the non-root user in the `Dockerfile` is to set it in the `docker-compose.yml`, provided that the non-root user has been added to the image in the `Dockerfile`. In the case of NodeGoat, the mongo `Dockerfile` is maintained by DockerHub, and it adds a user called `mongodb`. In the NodeGoat projects `docker-compose.yml`, we just need to set the user, as seen on line 13 below:

NodeGoat docker-compose.yml

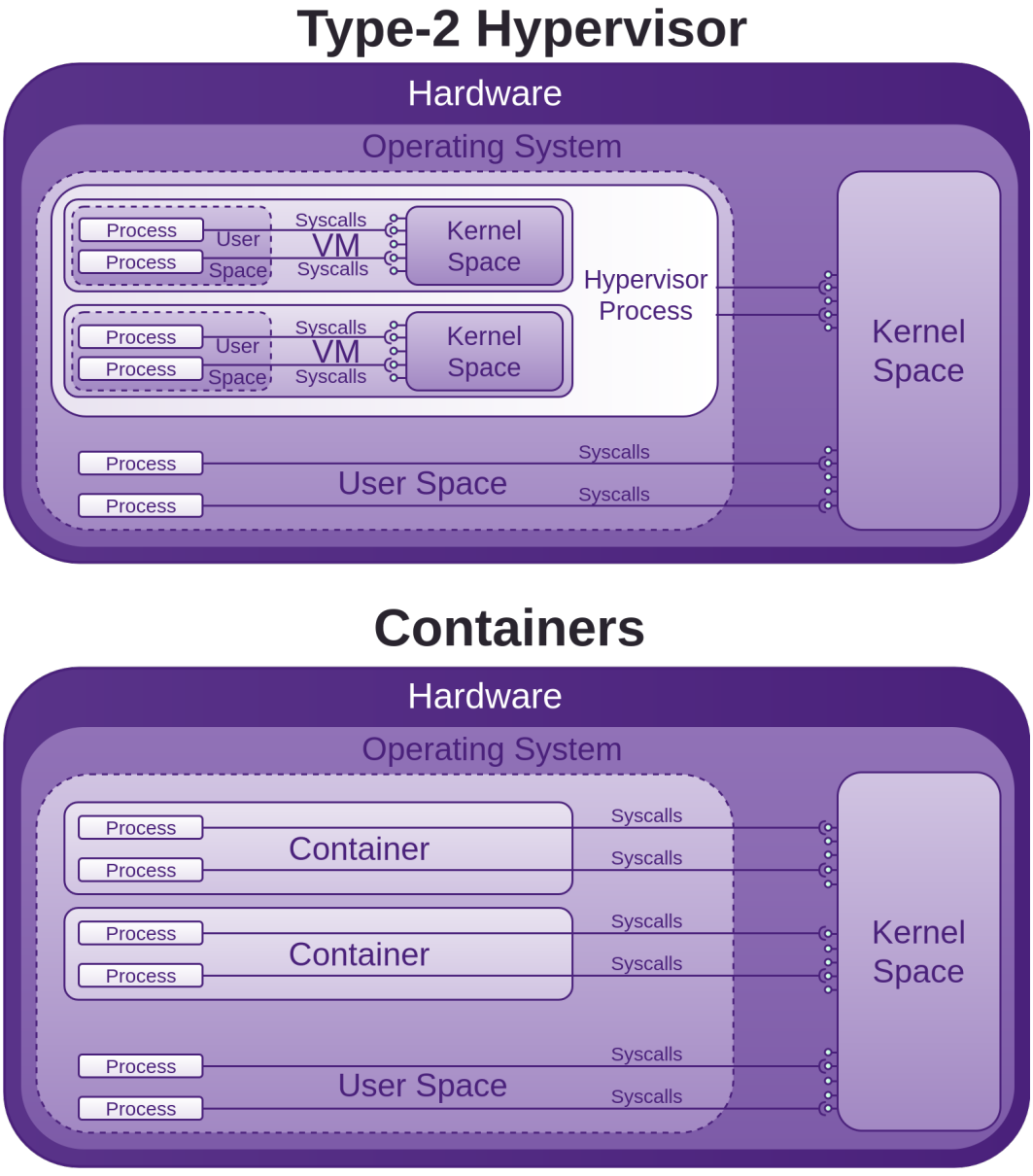
```
version: "2.0"

services:
  web:
    build: .
    command: bash -c "node artifacts/db-reset.js && npm start"
    ports:
      - "4000:4000"
    links:
      - mongo
  mongo:
    image: mongo:latest
    user: mongodb
    expose:
      - "27017"
```

Alternatively, a container may be run as a non-root user by
`docker run -it --user lowprivuser myimage`
but this is not ideal, the specific user should usually be part of the build.

2. Hardening Docker Host, Engine and Containers

Exploitability: DIFFICULT	Prevalence: UNCOMMON	Detectability: AVERAGE	Impact: MODERATE

Considering that these processes run as root, and have indirect access to most of the Linux Kernel (20+ million lines of code written by humans) APIs, such as networking, USB, storage stacks, and others via System calls, the situation may look bleak.



System calls are how programmes access the kernel to perform tasks. This attack surface is huge, and all before any security is added on top in the form of LXC, libcontainer (now opencontainers/runc), or Linux Security Modules (LSM) such as AppArmor or SELinux. These are often seen as an annoyance and just disabled like many other forms of security.

If you run a container, you may have to install `kmod`, then run `lsmod` in the container, and also on the host system. You will see that the same modules are loaded, this is because as mentioned, the container shares the host kernel, so there is not a lot between processes within the container and the host kernel. As mentioned above, the processes within the container may be running as root as well, it pays for you to have a good understanding of the security features Docker provides, and how to employ them.

The Seccomp section below discusses Docker's attempt to put a stop to some System calls accessing the kernel APIs. There are also many other features that Docker has added or leveraged in terms of mitigating a lot of this potential abuse. Although the situation initially looks bad, Docker has done a lot to improve it.

As you can see in the above image, the host kernel is open to receiving potential abuse from containers. Make sure you keep it patched. We will now walk though many areas of potential abuse.

Prevention: DIFFICULT

Make sure you keep your host kernel well patched, as it is a huge attack surface, with all of your containers accessing it via System calls.

The space for tooling to help find vulnerabilities in code, packages, etc within your Docker images has been noted, and tools provided. The following is a sorted list of what feels like does the least and is the simplest in terms of security/hardening features to what does the most, not understating tools that do a little, but do it well.

These tools should form a part of your secure and trusted build pipeline, or software supply-chain.

Haskell Dockerfile Linter

"*A smarter Dockerfile linter that helps you build best practice Docker images.*"

Lynis

Lynis is a mature, free and open source auditing tool for Linux/Unix based systems. There is a Docker plugin available which allows one to audit Docker, its configuration and containers, but an enterprise license is required, although it is inexpensive.

Docker Bench

Docker Bench is a shell script that can be downloaded from GitHub and executed immediately, run from a pre-built container, or using Docker Compose after Git cloning. Docker Bench tests many host configurations and Docker containers against the CIS Docker Benchmark.

CoreOS Clair

CoreOS is an open source project that appears to do a similar job to Docker Security Scanning, but it is free. You can use it on any image you pull, to compare the hashes of the packages from every container layer within, with hashes of the CVE data sources. You could also use Clair on your CI/CD build to stop images being deployed if they have packages with hashes that match those of the CVE data sources. quay.io was the first container registry to integrate with Clair.

Banyanops collector

Banyanops is a free and open source framework for static analysis of Docker images. It does more than Clair, it can optionally communicate with Docker registries, private or Docker Hub, to obtain image hashes, and it can then tell Docker Daemon to pull the images locally. Collector then `docker run`'s each container in turn to be inspected. Each container runs a banyan or user-specified script which outputs the results to stdout. Collector collates the containers output, and can send this to Banyan Analyser for further analysis. Collector has a pluggable, extensible architecture. Collector can also: enforce policies, such as no unauthorised user accounts, etc. Make sure components are in their correct location. Banyanops was the organisation that blogged about the high number of vulnerable packages on Docker Hub. They have really put their money where their mouth was now.

Anchore

Anchore is a set of tools that provide visibility, control, analytics, compliance and governance for containers in the cloud or on-prem for a fee.
There are two main parts, a hosted web service, and a set of open source CLI query tools.
The hosted service selects and analyses popular container images from Docker Hub and other

registries. The metadata it creates is provided as a service to the on-premise CLI tools.
It performs a similar job to that of Clair, but does not look as simple. It also looks for source code secrets, API keys, passwords, etc. in images.

It's designed to integrate into your CI/CD pipeline and integrates with Kubernetes, Docker, Jenkins, CoreOS, Mesos

TwistLock

TwistLock is a fairly comprehensive and complete proprietary offering with a free developer edition. The following details were taken from TwistLock marketing pages:

Features of Trust:

- Discover and manage vulnerabilities in images
- Uses CVE data sources similar to CoreOS Clair
- Can scan registries: Docker Hub, Google Container Registry, EC2 Container Registry, Artifactory, Nexus Registry, and images for vulnerabilities in code and configuration
- Enforce and verify standard configurations
- Hardening checks on images based on CIS Docker benchmark
- Real-time vulnerability and threat intelligence
- Provide out-of-box plugins for vulnerability reporting directly into Jenkins and TeamCity
- Provides a set of APIs for developers to access almost all of the TwistLock core functions

Features of Runtime:

- Policy enforcement
- Detect anomalies, uses open source CVE feeds, commercial threat and vulnerability sources, as well as TwistLock's own Lab research
- Defend and adapt against active threats and compromises using machine learning
- Governs access control to individual APIs of Docker Engine, Kubernetes, and Docker Swarm, providing LDAP/AD integration.

Possible contenders to watch

- Drydock is a similar offering to Docker Bench, but not as mature at this stage
- Actuary is a similar offering to Docker Bench, but not as mature at this stage. I discussed this project briefly with its creator Diogo Monica, and it sounds like the focus is on creating a better way of running privileged services on swarm, instead of investing time into this.

Namespaces (Risks)

The first place to read for solid background on Linux kernel namespaces is the man-page, otherwise I'd just be repeating what is there. A lot of what follows about namespaces requires some knowledge from the namespaces man-page, so do yourself a favour and read it first.

Linux kernel namespaces were first added between 2.6.15 (January 2006) and 2.6.26 (July 2008).

According to the namespaces man page, IPC, network and UTS namespace support was available from kernel version 3.0, while mount, PID and user namespace support was available from kernel version 3.8 (February 2013), and cgroup namespace support was available from kernel version 4.6 (May 2016).

Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker leverages the Linux (kernel) namespaces which provide an isolated workspace wrapped with a global system resource abstraction. This makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. When a container is run, Docker creates a set of namespaces for that container, providing a layer of isolation between containers:

1. `mnt`: (Mount) Provides filesystem isolation by managing filesystems and mount points. The `mnt` namespace allows a container to have its own isolated set of mounted filesystems, the propagation modes can be one of the following: [`r`]`shared`, [`r`]`slave` or [`r`]`private`. The `r` means recursive.

   If you run the following command, then the host's mounted `host-path` is shared with all others that mount `host-path`. Any changes made to the mounted data will be propagated to those that use the `shared` mode propagation. Using `slave` means only the master (`host-path`) is able to propagate changes, not vice-versa. Using `private` which is the default, will ensure no changes can be propagated.

Mounting volumes in shared mode propagation

```
docker run <run arguments> --volume=[host-path:]<container-path>:[z][r]shared <container ima\
ge name or id> <command> <args...>
```

If you omit the `host-path` you can see the host path that was mounted when running the following command:

Query

```
docker inspect <name or id of container>
```

Find the "Mounts" property in the JSON produced. It will have a "Source" and "Destination" similar to:

Result

```
...
"Mounts": [
  {
    "Name": "<container id>",
    "Source": "/var/lib/docker/volumes/<container id>/_data",
    "Destination": "<container-path>",
    "Mode": "",
    "RW": true,
    "Propagation": "shared"
  }
]
...
```

An empty string for Mode means that it is set to its read-write default. For example, a container can mount sensitive host system directories such as `/`, `/boot`, `/etc`, `/lib`, `/proc`, `/sys`, along with the rest as I discuss in the Lock Down the Mounting of Partitions section of my book Fascicle 1 of Holistic Info-Sec for Web Developers, particularly if that advice is not followed. If it is followed, you have some defence in depth working for you, and although Docker may have mounted a directory as read-write, the underlying mount may be read-only, thus stopping the container from being able to modify files in these locations on the host system. If the host does not have the above directories mounted with constrained permissions, then we are relying on the user running any given Docker container that mounts a sensitive host volume to mount it as read-only. For example, after the following command has been run, users within the container can modify files in the hosts `/etc` directory:

Vulnerable mount

```
docker run -it --rm -v /etc:/hosts-etc --name=lets-mount-etc ubuntu
```

Query

```
docker inspect -f "{{ json .Mounts }}" lets-mount-etc
```

Result

```
[
  {
    "Type":"bind",
    "Source":"/etc",
    "Destination":"/hosts-etc",
    "Mode":"",
    "RW":true,
    "Propagation":""
  }
]
```

Also keep in mind that, by default, the user in the container, unless otherwise specified, is root, the same root user as on the host system.

Labelling systems such as Linux Security Modules (LSM) require that the contents of a volume mounted into a container be labelled. This can be done by adding the `z` (as seen in above example) or `Z` suffix to the volume mount. The `z` suffix instructs Docker to share the mounted volume with other containers, and in so doing, Docker applies a shared content label. Alternatively, if you provide the `Z` suffix, Docker applies a private unshared label, which means only the current container can use the mounted volume. Further details can be found at the dockervolumes documentation. This is something to keep in mind if you are using LSM, and have a process inside your container that is unable to use the mounted data.
`--volumes-from` allows you to specify a data volume from another container.

You can also mount your Docker container mounts on the host by doing the following:

```
mount --bind /var/lib/docker/<volumes>/<container id>/_data </path/on/host>
```

2. `PID`: (Process ID) Provides process isolation, separates container processes from host and other container processes.

   The first process that is created in a new `PID` namespace is the "init" process with `PID` 1, which assumes parenthood of the other processes within the same `PID` namespace.

When PID 1 is terminated, so are the rest of the processes within the same PID namespace.

PID namespaces are hierarchically nested in ancestor-descendant relationships to a depth of up to 32 levels. All PID namespaces have a parent namespace, other than the initial root PID namespace of the host system. That parent namespace is the PID namespace of the process that created the child namespace.

Within a PID namespace, it is possible to access (make system calls to specific PIDs) all other processes in the same namespace, as well as all processes of descendant namespaces. However, processes in a child PID namespace cannot see processes that exist in the parent PID namespace or further removed ancestor namespaces. The direction any process can access another process in an ancestor/descendant PID namespace is one way.

Processes in different PID namespaces can have the same PID, because the PID namespace isolates the PID number space from other PID namespaces.

Docker takes advantage of PID namespaces. Just as you would expect, a Docker container can not access the host system processes, and process Ids that are used in the host system can be reused in the container, including PID 1, by being reassigned to a process started within the container. The host system can however access all processes within its containers, because as stated above, PID namespaces are hierarchically nested in parent-child relationships. Processes in the hosts PID namespace can access all processes in their own namespace down to the PID namespace that was responsible for starting the process, such as the process within the container in our case.

The default behaviour can however be overridden to allow a container to be able to access processes within a sibling container, or the hosts PID namespace. Example:

Syntax

```
--pid=[container:<name|id>],[host]
```

Example

```
# Provides access to the `PID` namespace of container called myContainer
# for container created from myImage.
docker run --pid=container:myContainer myImage
```

Example

```
# Provides access to the host `PID` namespace for container created from myImage
docker run --pid=host myImage
```

As an aside, PID namespaces give us the functionality of: "*suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.*" with a handful of commands:

Example

```
docker container pause myContainer [mySecondContainer...]
docker export [options] myContainer
# Move your container to another host.
docker import [OPTIONS] file|URL|- [REPOSITORY[:TAG]]
docker container unpause myContainer [mySecondContainer...]
```

3. net: (Networking) Provides network isolation by managing the network stack and interfaces. It's also essential to allow containers to communicate with the host system and other containers. Network namespaces were introduced into the kernel in 2.6.24, January 2008, with an additional year of development they were considered largely done. The only real concern here is understanding the Docker network modes and communication between containers. This is discussed in the Countermeasures.
4. UTS: (Unix Timesharing System) Provides isolation of kernel and version identifiers.

   UTS is the sharing of a computing resource with many users, a concept introduced in the 1960s/1970s.

   A UTS namespace is the set of identifiers returned by uname, which include the hostname and the NIS domain name. Any processes which are not children of the process that requested the clone will not be able to see any changes made to the identifiers of the UTS namespace.

   If the CLONE_NEWUTS constant is set, then the process being created will be created in a new UTS namespace with the hostname and NIS domain name copied and able to be modified independently from the UTS namespace of the calling process.

   If the CLONE_NEWUTS constant is not set, then the process being created will be created in the same UTS namespace of the calling process, thus able to change the identifiers returned by uname.

   When a container is created, a UTS namespace is copied (CLONE_NEWUTS is set)(--uts="") by default, providing a UTS namespace that can be modified independently from the target UTS namespece it was copied from.

When a container is created with `--uts="host"`, a UTS namespace is inherited from the host, the `--hostname` flag is invalid.

5. `IPC`: (InterProcess Communication) manages access to InterProcess Communications). `IPC` namespaces isolate your container's System V IPC and POSIX message queues, semaphores, and named shared memory from those of the host and other containers, unless another container specifies on run that it wants to share your namespace. It would be a lot safer if the producer could specify which consuming containers could use its namespace. IPC namespaces do not include IPC mechanisms that use filesystem resources such as named pipes.

   According to the namespaces man page: "*Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces.*"

   Although sharing memory segments between processes provide Inter-Process Communications at memory speed, rather than through pipes or worse, the network stack, this produces a significant security concern.

   By default a container does not share the host's or any other container's IPC namespace. This behaviour can be overridden to allow a (any) container to reuse another container's or the host's message queues, semaphores, and shared memory via their IPC namespace. Example:

Syntax

```
# Allows a container to reuse another container's IPC namespace.
--ipc=[container:<name|id>],[host]
```

Example

```
docker run -it --rm --name=container-producer ubuntu
root@609d19340303:/#

# Allows the container named container-consumer to share the IPC namespace
# of container called container-producer.
docker run -it --rm --name=container-consumer --ipc=container:container-producer ubuntu
root@d68ecd6ce69b:/#
```

Now find the Ids of the two running containers:

Query

```
docker inspect --format="{{ .Id }}" container-producer container-consumer
```

Result

```
609d193403032a49481099b1fc53037fb5352ae148c58c362ab0a020f473c040
d68ecd6ce69b89253f7ab14de23c9335acaca64d210280590731ce1fcf7a7556
```

You can see from using the command supplied by the CIS_Docker_1.13.0_Benchmark that `container-consumer` is using the IPC namespace of `container-producer`:

Query

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: IpcMode={{ .HostConfig.I\
pcMode }}'
```

Result

```
d68ecd6ce69b89253f7ab14de23c9335acaca64d210280590731ce1fcf7a7556: IpcMode=container:containe\
r-producer
609d193403032a49481099b1fc53037fb5352ae148c58c362ab0a020f473c040: IpcMode=
```

When the last process in an IPC namespace terminates, the namespace will be destroyed along with all IPC objects in the namespace.

6. `user`: Not enabled by default. Allows a process within a container to have a unique range of user and group Ids within the container, known as the subordinate user and group Id feature in the Linux kernel. These do not map to the same user and group Ids of the host, container users to host users are remapped. For example, if a user within a container is root, which it is by default unless a specific user is defined in the image hierarchy, it will be mapped to a non-privileged user on the host system.
   Docker considers user namespaces to be an advanced feature. There are currently some Docker features that are incompatible with using user namespaces, and according to the CIS Docker 1.13.0 Benchmark, functionalities that are broken if user namespaces are used. the Docker engine reference provides additional details around known restrictions of user namespaces.
   If your containers have a predefined non-root user, then, currently, user namespaces should not be enabled, due to possible unpredictable issues and complexities, according to "2.8 Enable user namespace support" of the CIS Docker Benchmark.
   The problem is that these mappings are performed on the Docker daemon rather than at a per-container level, so it is an all or nothing approach. This may change in the

future though.
As mentioned, user namespace support is available, but not enabled by default in the Docker daemon.

Namespaces (Countermeasures)

1. `mnt`: Keep the default propagation mode of `private` unless you have a very good reason to change it. If you do need to change it, think about defence in depth and employ other defence strategies.

   If you have control over the Docker host, lock down the mounting of the host systems partitions as I discussed in the Lock Down the Mounting of Partitions subsection of the VPS chapter of my book Fascicle 1 Holistic Info-Sec for Web Developers.

   If you have to mount a sensitive host system directory, mount it as read-only:

   ```
   docker run -it --rm -v /etc:/hosts-etc:ro --name=lets-mount-etc ubuntu
   ```

   If any file modifications are now attempted on `/etc` they will be unsuccessful.

Query

```
docker inspect -f "{{ json .Mounts }}" lets-mount-etc
```

Result

```
[
  {
    "Type":"bind",
    "Source":"/etc",
    "Destination":"/hosts-etc",
    "Mode":"ro",
    "RW":false,
    "Propagation":""
  }
]
```

   Also, as discussed previously, lock down the user to non-root.

   If you are using LSM, you will probably want to use the `Z` option as discussed in the risks section.
2. `PID`: By default enforces isolation from the containers `PID` namespace, but not from the host to the container. If you are concerned about host systems being able to access your containers, as you should be, consider putting your containers within a VM

3. `net`: A network namespace is a virtualisation of the network stack, with its own network devices, IP routing tables, firewall rules and ports.
   When a network namespace is created the only network interface that is created is the loopback interface, which is down until brought up.
   Each network interface, whether physical or virtual, can only reside in one namespace, but can be moved between namespaces.

   When the last process in a network namespace terminates, the namespace will be destroyed, destroy any virtual interfaces within it, and move any physical network devices back to the initial network namespace, not the process parent.

   Docker and Network Namespaces

   A Docker network is analogous to a Linux kernel network namespace.

   When Docker is installed, three networks are created `bridge`, `host` and `null`, which you can think of as network namespaces. These can be seen by running: `docker network ls`

```
NETWORK ID    NAME              DRIVER   SCOPE
9897a3063354  bridge            bridge   local
fe179428ccd4  host              host     local
a81e8669bda7  none              null     local
```

   When you run a container, if you want to override the default network of `bridge`, you can specify which network in which you want to run the container with the `--network` flag as the following:
   `docker run --network=<network>`

   The bridge can be seen by running `ifconfig` on the host:

```
docker0   Link encap:Ethernet  HWaddr 05:22:bb:08:41:b7
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:fbff:fe80:57a5/64 Scope:Link
```

   When the Docker engine (CLI) client or API tells the Docker daemon to run a container, part of the process allocates a bridged interface, unless specified otherwise, that allows processes within the container to communicate to the system host via the virtual Ethernet bridge.

   Virtual Ethernet interfaces, when created, are always created as a pair. You can think of them as one interface on each side of a namespace wall with a tube through the wall connecting them. Packets come in one interface and exit the other, and vice versa.

   Creating and Listing Network NameSpaces

Some of these commands you will need to run as root.

Create:

Syntax

```
ip netns add <yournamespacename>
```

Example

```
ip netns add testnamespace
```

This ip command adds a bind mount point for the `testnamespace` namespace to `/var/run/netns/`. When the `testnamespace` namespace is created, the resulting file descriptor keeps the network namespace alive and persisted. This allows system administrators to apply configuration to the network namespace without fear that it will disappear when no processes are within it.

Verify it was added

```
ip netns list
```

Result

```
testnamespace
```

However, a network namespace added in this way cannot be used for a Docker container. In order to create a Docker network called `kimsdockernet` run the following command:

```
# bridge is the default driver, so not required to be specified
docker network create --driver bridge kimsdockernet
```

You can then follow this with a
`docker network ls`
to confirm that the network was added. You can base your network on one of the existing network drivers created by Docker, the bridge driver is used by default.

`bridge`: As seen above with the `ifconfig` listing on the host system, an interface is created called docker0 when Docker is installed. A pair of veth (Virtual Ethernet) interfaces are created when the container is run with this `--network` option. The `veth` on the outside of the container will be attached to the bridge, the other `veth` is put inside the container's namespace, along with the existing loopback interface.

`none`: There will be no networking in the container other than the loopback interface which was created when the network namespace was created, and has no routes to external traffic.
`host`: Uses the network stack that the host system uses inside the container. The `host` mode is more performant than the `bridge` mode due to using the hosts native network stack, but also less secure.
`container`: Allows you to specify another container to use its network stack.

When running
`docker network inspect kimsdockernet`
before starting the container, and then again after, you will see the new container added to the `kimsdockernet` network.

Now you can run your container using your new network:

```
docker run -it --network kimsdockernet --rm --name=container0 ubuntu
```

When one or more processes, Docker containers in this case, uses the `kimsdockernet` network, it can also be seen opened by the presence of its file descriptor at:

`/var/run/docker/netns/<filedescriptor>`

You can also see that the container named `container0` has a network namespace by running the following command, which shows the file handles for the namespaces, and not just the network namespace:

Query Namespaces

```
sudo ls /proc/`docker inspect -f '{{ .State.Pid }}' container0`/ns -liah
```

Result

```
total 0
1589018 dr-x--x--x 2 root root 0 Mar 14 16:35 .
1587630 dr-xr-xr-x 9 root root 0 Mar 14 16:35 ..
1722671 lrwxrwxrwx 1 root root 0 Mar 14 17:33 cgroup -> cgroup:[4026531835]
1722667 lrwxrwxrwx 1 root root 0 Mar 14 17:33 ipc -> ipc:[4026532634]
1722670 lrwxrwxrwx 1 root root 0 Mar 14 17:33 mnt -> mnt:[4026532632]
1589019 lrwxrwxrwx 1 root root 0 Mar 14 16:35 net -> net:[4026532637]
1722668 lrwxrwxrwx 1 root root 0 Mar 14 17:33 pid -> pid:[4026532635]
1722669 lrwxrwxrwx 1 root root 0 Mar 14 17:33 user -> user:[4026531837]
1722666 lrwxrwxrwx 1 root root 0 Mar 14 17:33 uts -> uts:[4026532633]
```

If you run
`ip netns list`

again, you may think that you should be able to see the Docker network, but you will not, unless you create the following symlink:

```
ln -s /proc/`docker inspect -f '{{.State.Pid}}' container0`/ns/net /var/run/netns/container0
# Don't forget to remove the symlink once the container terminates,
# else it will be dangling.
```

If you want to run a command inside of the Docker network of a container, you can use the `nsenter` command of the `util-linux` package:

```
# Show the ethernet state:
nsenter -t `docker inspect -f '{{ .State.Pid }}' container0` -n ifconfig
# Or
nsenter -t `docker inspect -f '{{ .State.Pid }}' container0` -n ip addr show
# Or
nsenter --net=/var/run/docker/netns/<filedescriptor> ifconfig
# Or
nsenter --net=/var/run/docker/netns/<filedescriptor> ip addr show
```

Deleting Network NameSpaces

The following command will remove the bind mount for the specified namespace. The namespace will continue to persist until all processes within it are terminated, at which point any virtual interfaces within it will be destroyed and any physical network devices if they were assigned, would be moved back to the initial network namespace, not the process parent.

Syntax

```
ip netns delete <yournamespacename>
```

Example

```
ip netns delete testnamespace
```

To remove a docker network

```
docker network rm kimsdockernet
```

If you still have a container running, you will receive an error:
`Error response from daemon: network kimsdockernet has active endpoints`
Stop your container and try again.

It also pays to understand container communication with each other.

Also checkout the Additional Resources.

4. `UTS` Do not start your containers with the `--uts` flag set to `host`
   As mentioned in the CIS_Docker_1.13.0_Benchmark "*Sharing the UTS namespace with the host provides full permission to the container to change the hostname of the host. This is insecure and should not be allowed.*". You can test that the container is not sharing the host's UTS namespace by making sure that the following command returns nothing, instead of `host`:

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: UTSMode={{ .HostConfig.U\
TSMode }}'
```

5. `IPC`: In order to stop another untrusted container sharing your containers IPC namespace, you could isolate all of your trusted containers in a VM, or if you are using some type of orchestration, that will usually have functionality to isolate groups of containers. If you can isolate your trusted containers sufficiently, then you may still be able to share the IPC namespace of other near by containers.
6. `user`: If you have read the risks section and still want to enable support for user namespaces, you first need to confirm that the host user of the associated containers `PID` is not root by running the following CIS Docker Benchmark recommended commands:

```
ps -p $(docker inspect --format='{{ .State.Pid }}' <CONTAINER ID>) -o pid,user
```

   Or, you can run the following command and make sure that the `userns` is listed under the `SecurityOptions`

```
docker info --format '{{ .SecurityOptions }}'
```

   Once you have confirmed that your containers are not being run as root, you can look at enabling user namespace support on the Docker daemon.

   The `/etc/subuid` and `/etc/subgid` host files will be read for the user and optional group supplied to the `--userns-remap` option of `dockerd`.

   The `--userns-remap` option accepts the following value types:
   - `uid`
   - `uid:gid`
   - `username`
   - `username:groupname`

   The username must exist in the `/etc/passwd` file, the `sbin/nologin` users are also valid. Subordinate user Id and group Id ranges need to be specified in `/etc/subuid` and `/etc/subuid` respectively.

"*The UID/GID we want to remap to does not need to match the UID/GID of the username in `/etc/passwd`*". It is the entity in the `/etc/subuid` that will be the owner of the Docker daemon and the containers it runs. The value you supply to `--userns-remap` if numeric Ids, will be translated back to the valid user or group names of `/etc/passwd` and `/etc/group` which must exist, if username, groupname, they must match the entities in `/etc/passwd`, `/etc/subuid`, and `/etc/subgid`.

Alternatively, if you do not want to specify your own user and/or user:group, you can provide the `default` value to `--userns-remap`, and a default user of `dockremap` along with subordinate uid and gid ranges that will be created in `/etc/passwd` and `/etc/group` if it does not already exist. Then the `/etc/subuid` and `/etc/subgid` files will be populated with a contiguous 65536 length range of subordinate user and group Ids respectively, starting at the offset of the existing entries in those files.

```
# As root, run:
dockerd --userns-remap=default
```

If `dockremap` does not already exist, it will be created:

/etc/subuid and /etc/subgid

```
<existinguser>:100000:65536
dockremap:165536:65536
```

There are rules about providing multiple range segments in the `/etc/subuid`, `/etc/subgid` files, but that is beyond the scope of what I am providing here. For those advanced scenario details, check out the Docker engine reference. The simplest scenario is to use a single contiguous range as seen in the above example, this will cause Docker to map the hosts user and group Ids to the container process using as much of the `165536:65536` range as necessary. For example, the host's root user would be mapped to `165536`, the next host user would be mapped to container user `165537`, and so on until the 65536 possible Ids are all mapped. Processes run as root inside the container are owned by the subordinate uid outside of the container.

Disabling user namespace for specific containers

In order to disable user namespace mapping, on a per container basis, once enabled for the Docker daemon, you could supply the `--userns=host` value to either of the `run`, `exec` or `create` Docker commands. This would mean the default user within the container was mapped to the host's root.

Control Groups (Risks)

When a container is started with `docker run` without specifying a cgroup parent, as well as creating the namespaces discussed above, Docker also creates a Control Group (or cgroup) with a set of system resource hierarchies, nested under the default parent `docker` cgroup, also created at container runtime, if not already present. You can see how this hierarchy looks in the `/sys/fs/cgroup` pseudo-filesystem in the Countermeasures section. Cgroups have been available in the Linux kernel since January 2008 (2.6.24), and continue to improve. Cgroups track, provide the ability to monitor, and configure, fine-grained limitations on how much of any resource a set of processes, or in the case of Docker or pure LXC, any given container can use, such as CPU, memory, disk I/O, and network. Many aspects of these resources can be controlled, but by default, any given container can use all of the system's resources, allowing potential DoS.

Fork Bomb from Container

If an attacker gains access to a container, or, in a multi-tenanted scenario where being able to run a container by an arbitrary entity is expected, by default, there is nothing stopping a fork bomb
`:(){:|:&};:`
launched in a container from bringing the host system down. This is because, by default, there is no limit to the number of processes a container can run.

Control Groups (Countermeasures)

Use cgroups to limit, track and monitor the resources available to each container at each nested level. Docker makes applying resource constraints very easy. Check the runtime constraints on resources Docker engine run reference documentation, which covers applying constraints such as:

- User memory
- Kernel memory
- Swappiness
- CPU share
- CPU period
- Cpuset
- CPU quota

- Block IO bandwidth (Blkio)

For additional details on setting these types of resource limits, also refer to the Limit a container's resources Admin Guide for Docker Engine. Basically, when you `run` a container, you simply provide any number of the runtime configuration flags that control the underlying cgroup system resources. Cgroup resources cannot be set if a process is not running, that is why we optionally pass the flag(s) at runtime or alternatively, manually change the cgroup settings once a process (or Docker container in our case) is running. We can make manual changes on the fly by directly modifying the cgroup resource files. These files are stored in the container's cgroup directories shown in the output of the `/sys/fs/cgroup find -name "4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24"` command below. These files are ephemeral for the life of the process (Docker container in our case).

By default Docker uses the cgroupfs cgroup driver to interface with the Linux kernel's cgroups. You can see this by running `docker info`. The Linux kernel's cgroup interface is provided through the cgroupfs pseudo-filesystem `/sys/fs/cgroup` on the host filesystem of recent Linux distributions. The `/proc/cgroups` file contains the information about the systems controllers compiled into the kernel. This file on my test system looks like the following:

```
#subsys_name    hierarchy       num_cgroups     enabled
cpuset          4               9               1
cpu             5               106             1
cpuacct         5               106             1
blkio           11              105             1
memory          6               170             1
devices         8               105             1
freezer         3               9               1
net_cls         7               9               1
perf_event      2               9               1
net_prio        7               9               1
hugetlb         9               9               1
pids            10              110             1
```

The fields represent the following:

- `subsys_name`: The name of the controller
- `hierarchy`: Unique Id of the cgroup hierarchy
- `num_cgroups`: The number of cgroups in the specific hierarchy using this controller
- `enabled`: 1 == enabled, 0 == disabled

If you run a container as follows:

```
docker run -it --rm --name=cgroup-test ubuntu
root@4f1f200ce13f:/#
```

Cgroups for your containers and the system resources controlled by them will be stored as follows:

/sys/fs/cgroup pseudo-filesystem

```
/sys/fs/cgroup   find -name "4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24"
./blkio/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./pids/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./hugetlb/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./devices/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./net_cls,net_prio/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./memory/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./cpu,cpuacct/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./cpuset/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./freezer/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./perf_event/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
./systemd/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/sys/fs/cgro\
up` pseudo-filesystem
```

Docker also keeps track of the cgroups in
`/sys/fs/cgroup/[resource]/docker/[containerId]`
You will notice that Docker creates cgroups using the container Id.

If you want to manually create a cgroup, and have your containers hierarchically nested within it, you just need to `mkdir` within:
`/sys/fs/cgroup/`
You will likely need to be root for this.

```
/sys/fs/cgroup mkdir cg1
```

This makes and populates the directory, and also sets up the cgroup like the following:

```
/sys/fs/cgroup   find -name "cg1"
./cg1
./blkio/system.slice/docker.service/cg1
./pids/system.slice/docker.service/cg1
./hugetlb/cg1
./devices/system.slice/docker.service/cg1
./net_cls,net_prio/cg1
./memory/system.slice/docker.service/cg1
./cpu,cpuacct/system.slice/docker.service/cg1
./cpuset/cg1
./freezer/
./perf_event/cg1
./systemd/system.slice/docker.service/cg1
```

Now you can run a container with `cg1` as your cgroup parent:

```
docker run -it --rm --cgroup-parent=cg1 --name=cgroup-test1 ubuntu
root@810095d51702:/#
```

Now that Docker has your container named `cgroup-test1` running, you will be able to see the nested cgroups:

```
/sys/fs/cgroup   find -name "810095d51702*"
./blkio/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90\
961ee528903
./pids/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b909\
61ee528903
./hugetlb/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90961ee528903
./devices/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b\
90961ee528903
./net_cls,net_prio/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90961ee528903
./memory/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b9\
0961ee528903
./cpu,cpuacct/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883\
306b90961ee528903
./cpuset/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90961ee528903
./freezer/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90961ee528903
./perf_event/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b90961ee528903
./systemd/system.slice/docker.service/cg1/810095d517027737a0ba4619e108903c5cc74517907b883306b\
90961ee528903
```

You can also run containers nested below already running containers cgroups, let's take the container named `cgroup-test` for example:

```
/sys/fs/cgroup/cpu/docker/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24
```

```
docker run -it --rm --cgroup-parent=4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e55\
1f4eb24 --name=cgroup-test2 ubuntu
root@93cb84d30291:/#
```

Now your new container named `cgroup-test2` will have a set of nested cgroups within each of the:
`93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270`
directories shown here:

```
/sys/fs/cgroup   find -name "93cb84d30291*"
./blkio/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e55\
1f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./pids/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551\
f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./hugetlb/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a84\
d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./devices/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e\
551f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./net_cls,net_prio/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30\
291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./memory/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e5\
51f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./cpu,cpuacct/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5\
ee1e551f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./cpuset/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a84d\
5676545015220696dbcc72a65a12a0c96cda01dd1d270
./freezer/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a84\
d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./perf_event/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201\
a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
./systemd/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e\
551f4eb24/93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270
```

You should see the same result if you have a look in the running container's `/proc/self/cgroup` file.

Within each cgroup resides a collection of files specific to the controlled resource, some of which are used to limit aspects of the resource, and some of which are used for monitoring aspects of the resource. They should be fairly obvious what they are based on their names. You can not exceed the resource limits of the cgroup that your cgroup is nested within. There

are ways in which you can get visibility into any containers resource usage. One quick and simple way is with the:

```
docker stats [containerId]
```

command, which will give you a line with your containers CPU usage, Memory usage and Limit, Net I/O, Block I/O, Number of PIDs. There are so many other sources of container resource usage. Check the Docker engine runtime metrics documentation for additional details.

The most granular information can be found in the statistical files within the cgroup directories listed above.
The `/proc/[pid]/cgroup` file provides a description of the cgroups that the process with the specified PID belongs to. You can see this in the following `cat` output. The information provided is different for cgroups version 1 and version 2 hierarchies, for this example, we are focussing on version 1. Docker abstracts all of this anyway, so it is just to show you how things hang together:

```
cat /proc/`docker inspect -f '{{ .State.Pid }}' cgroup-test2`/cgroup
11:blkio:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e\
551f4eb24/93cb84d30291201a84d567654501522069 6dbcc72a65a12a0c96cda01dd1d270
10:pids:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e5\
51f4eb24/93cb84d30291201a84d567654501522069 6dbcc72a65a12a0c96cda01dd1d270
9:hugetlb:/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a8\
4d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
8:devices:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1\
e551f4eb24/93cb84d30291201a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
7:net_cls,net_prio:/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d3\
0291201a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
6:memory:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e\
551f4eb24/93cb84d30291201a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
5:cpu,cpuacct:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b\
5ee1e551f4eb24/93cb84d30291201a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
4:cpuset:/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a84\
d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
3:freezer:/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d30291201a8\
4d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
2:perf_event:/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7b5ee1e551f4eb24/93cb84d3029120\
1a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
1:name=systemd:/system.slice/docker.service/4f1f200ce13f2a7a180730f964c6c56d25218d6dd40b027c7\
b5ee1e551f4eb24/93cb84d30291201a84d56765450152206 96dbcc72a65a12a0c96cda01dd1d270
```

Each row of the above file depicts one of the cgroup hierarchies that the process, or Docker container in our case, is a member of. The row consists of three fields separated by colon, in the form:

`hierarchy-Id:list-of-controllers-bound-to-hierarchy:cgroup-path`
If you remember back to our review of the `/proc/cgroups` file above, you will notice that the:

1. hierarchy unique Id is represented here as the `hierarchy-Id`
2. subsys_name is represented here in the comma separated list-of-controllers-bound-to-hierarchy
3. Unrelated to `/proc/cgroups`, the third field contains relative to the mount point of the hierarchy the pathname of the cgroup in the hierarchy to which the process belongs. You can see this reflected with the
   `/sys/fs/cgroup find -name "93cb84d30291*"`
   from above

Fork Bomb from Container

With a little help from the CIS Docker Benchmark we can use the `PIDs` cgroup limit:

Run the containers with `--pids-limit` (kernel version 4.3+) and set a sensible value for maximum number of processes that the container can run, based on what the container is expected to be doing. By default the PidsLimit value displayed with the following command will be 0. 0 or -1 means that any number of processes can be forked within the container:

Query

```
docker inspect -f '{{ .Id }}: PidsLimit={{ .HostConfig.PidsLimit }}' cgroup-test2
```

Result

```
93cb84d30291201a84d5676545015220696dbcc72a65a12a0c96cda01dd1d270: PidsLimit=0
```

```
docker run -it --pids-limit=50 --rm --cgroup-parent=4f1f200ce13f2a7a180730f964c6c56d25218d6dd\
40b027c7b5ee1e551f4eb24 --name=cgroup-test2 ubuntu
root@a26c39377af9:/#
```

Query

```
docker inspect -f '{{ .Id }}: PidsLimit={{ .HostConfig.PidsLimit }}'
```

Result

```
cgroup-test2 a26c39377af9ce6554a1b6a8bffb2043c2c5326455d64c2c8a8cfe53b30b7234: PidsLimit=50
```

Capabilities (Risks)

According to the Linux man page for capabilities, "*Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled*". This is on a per thread basis. Root, with all capabilities, has privileges to do everything. According to the man page, there are currently 38 capabilities.

By default, the following capabilities are available to the default user of root within a container, check the man page for the full descriptions of the capabilities. The very knowledgeable Dan Walsh, who is one of the experts when it comes to applying least privilege to containers, also discusses these: `chown`, `dac_override`, `fowner`, `fsetid`, `kill`, `setgid`, `setuid`, `setpcap`, `net_bind_service`, `net_raw`, `sys_chroot`, `mknod`, `audit_write`, `setfcap`. `net_bind_service` for example allows the superuser to bind a socket to a privileged port <1024 if enabled. The Open Container Initiative (OCI) runC specification is considerably more restrictive, only enabling three capabilities: `audit_write`, `kill`, `net_bind_service`

As stated on the Docker Engine security page: "*One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.*"

Capabilities (Countermeasures)

There are several ways you can minimise your set of capabilities that the root user of the container will run. `pscap` is a useful command from the `libcap-ng-utils` package in Debian and some other distributions. Once installed, you can check which capabilities your container built from the `<amazing>` image runs with, by:

```
docker run -d <amazing> sleep 5 >/dev/null; pscap | grep sleep
# This will show which capabilities sleep within container is running as.
# By default, it will be the list shown in the Identify Risks section.
```

In order to drop capabilities `setfcap`, `audit_write`, and `mknod`, you could run:

```
docker run -d --cap-drop=setfcap --cap-drop=audit_write --cap-drop=mknod <amazing> sleep 5 > \
/dev/null; pscap | grep sleep
# This will show that sleep within the container no longer has enabled:
# setfcap, audit_write, or mknod
```

Or just drop all capabilities and only add what you need:

```
docker run -d --cap-drop=all --cap-add=audit_write --cap-add=kill --cap-add=setgid --cap-add=\
setuid <amazing> sleep 5 > /dev/null; pscap | grep sleep
# This will show that sleep within the container is only running with
# audit_write, kill, setgid and setuid.
```

Another way of auditing the capabilities of your container is with the following command from CIS Docker Benchmark:

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: CapAdd={{ .HostConfig.CapAdd }}\
 CapDrop={{ .HostConfig.CapDrop }}'
```

Alternatively you can modify the container manifest directly. See the runC section for this.

Linux Security Modules (Risks)

A little history to start with: In the early 1990s, Linux was developed as a clone of the Unix Operating system. The core Unix security model, which is a form of Discretionary Access Control (DAC), was inherited by Linux. I have provided a glimpse of some of the Linux kernel security features that have been developed since the inception of Linux. The Unix DAC remains at the core of Linux. The Unix DAC allows a subject and/or the group of an identity to set the security policy for a specific object. The canonical example is a file, and having a user set the different permissions on who can do what with it. The Unix DAC was designed in 1969, and a lot has changed since then.

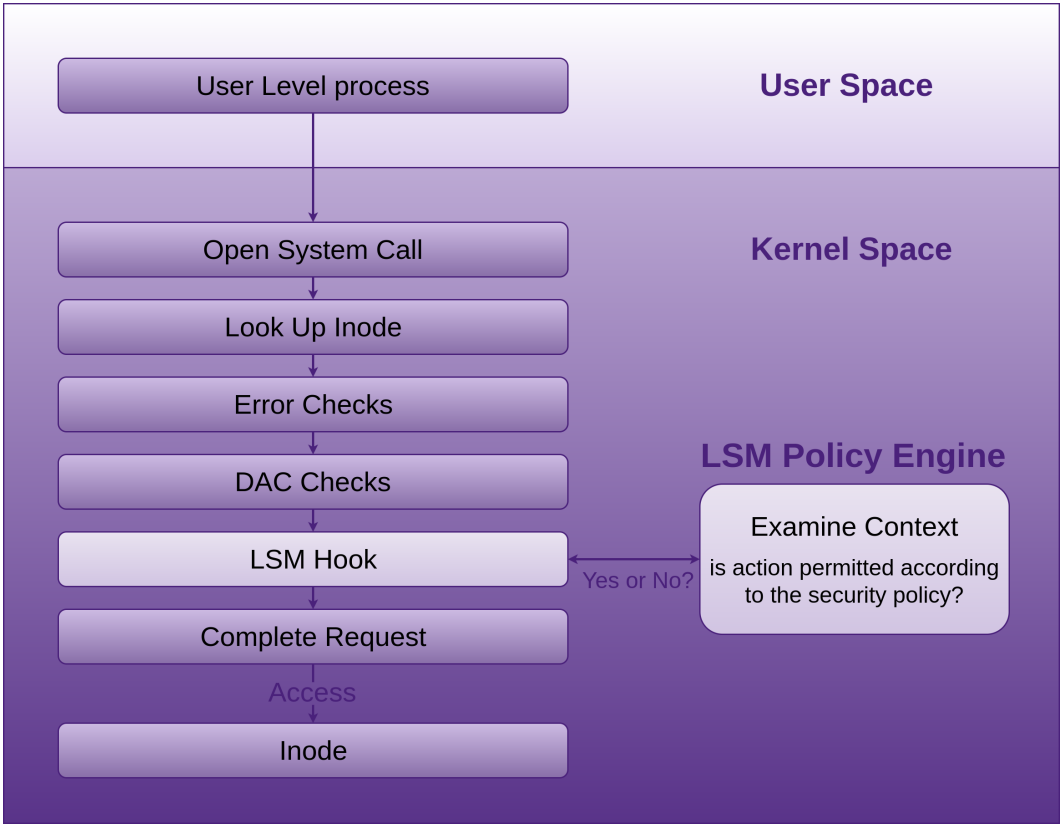
Capabilities vary in granularity, attain an understanding of both capabilities and Linux Security Modules (LSMs). Many of the DACs can be circumvented by users. Finer grained control is often required along with Mandatory Access Control (MAC).

Linux Security Modules (Countermeasures)

Linux Security Modules (LSM) is a framework that has been part of the Linux kernel since 2.6, that supports security models implementing Mandatory Access Control (MAC). The currently accepted modules are AppArmor, SELinux, Smack and TOMOYO Linux.

At the first Linux kernel summit in 2001, "*Peter Loscocco from the National Security Agency (NSA) presented the design of the mandatory access control system in its SE Linux distribution.*" SE Linux had implemented many check points where authorization to perform a particular task was controlled, and a security manager process which implements the actual authorization policy. "*The separation of the checks and the policy mechanism is an important aspect of the system - different sites can implement very different access policies using the same system.*" The aim of this separation is to make it harder for the user to not adjust or override policies.

It was realised that there were several security related projects trying to solve the same problem. It was decided to have the developers interested in security create a "*generic interface which could be used by any security policy. The result was the Linux Security Modules (LSM)*" API/framework, which provides many hooks at security critical points within the kernel.



LSMs can register with the API and receive callbacks from these hooks when the Unix Discretionary Access Control (DAC) checks succeed, allowing the LSMs Mandatory Access Control (MAC) code to run. The LSMs are not loadable kernel modules, but are instead selectable at build-time via `CONFIG_DEFAULT_SECURITY` which takes a comma separated list of LSM names. Commonly multiple LSMs are built into a given kernel and can be overridden at boot time via the `security=...` kernel command line argument, while also taking a comma separated list of LSM names.

If no specific LSMs are built into the kernel, the default LSM will be the Linux capabilities. "*Most LSMs choose to extend the capabilities system, building their checks on top of the defined capability hooks.*" A comma separated list of the active security modules can be found in `/sys/kernel/security/lsm`. The list reflects the order in which checks are made, the capability module will always be present and be the first in the list.

AppArmor LSM in Docker

If you intend to use AppArmor, make sure it is installed, and you have a policy loaded (`apparmor_parser -r [/path/to/your_policy]`) and enforced (`aa-enforce`). AppArmor policy's are created using the profile language. Docker will automatically generate and load a default AppArmor policy `docker-default` when you run a container. If you want to override the policy, you do this with the `--security-opt` flag, like:
`docker run --security-opt apparmor=your_policy [container-name]`
provided that your policy is loaded as mentioned above. There are further details available on the apparmor page of Dockers Secure Engine.

SELinux LSM in Docker

Red Hat, Fedora, and some other distributions ship with SELinux policies for Docker. Many other distros such as Debian require an install. SELinux needs to be installed and configured on Debian.

SELinux support for the Docker daemon is disabled by default and needs to be enabled with the following command:

```
#Start the Docker daemon with:
dockerd --selinux-enabled
```

Docker daemon options can also be set within the daemon configuration file
`/etc/docker/daemon.json`
by default or by specifying an alternative location with the `--config-file` flag.

Label confinement for the container can be configured using `--security-opt` to load SELinux or AppArmor policies as shown in the Docker `run` example below:

SELinux Labels for Docker consist of four parts:

Syntax

```
# Set the label user for the container.
--security-opt="label:user:USER"
# Set the label role for the container.
--security-opt="label:role:ROLE"
# Set the label type for the container.
--security-opt="label:type:TYPE"
# Set the label level for the container.
--security-opt="label:level:LEVEL"
```

Example

```
docker run -it --security-opt label=level:s0:c100,c200 ubuntu
```

SELinux can be enabled in the container using `setenforce 1`.

SELinux can operate in one of three modes:

1. `disabled`: not enabled in the kernel
2. `permissive` or `0`: SELinux is running and logging, but not controlling/enforcing permissions
3. `enforcing` or `1`: SELinux is running and enforcing policy

To change at runtime: Use the `setenforce [0|1]` command to change between `permissive` and `enforcing`. Test this, set to `enforcing` before persisting it at boot.
To persist on boot: In Debian, set `enforcing=1` in the kernel command line
`GRUB_CMDLINE_LINUX` in `/etc/default/grub`
and run `update-grub`
SELinux will be enforcing after a reboot.

To audit what LSM options you currently have applied to your containers, run the following command from the CIS Docker Benchmark:

```
docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: SecurityOpt={{ .HostConfi\
g.SecurityOpt }}'
```

SecComp (Risks)

Secure Computing Mode (SecComp) is a security facility that reduces the attack surface of the Linux kernel by reducing the number of System calls that can be made by a process. Any System calls made by the process, outside of the defined set, will cause the kernel to terminate the process with `SIGKILL`. In so doing, the SecComp facility stops a process from accessing the kernel APIs via System calls.

The first version of SecComp was merged into the Linux kernel mainline in version 2.6.12 (March 8 2005). If enabled for a given process, only four System calls could be made: `read()`, `write()`, `exit()`, and `sigreturn()`, thus significantly reducing the kernel's attack surface.

In order to enable SecComp for a given process, you would write a `1` to `/proc/<PID>/seccomp`. This would cause the one-way transition into the restrictive state.

There have been a few revisions since 2005, such as the addition of "seccomp filter mode", which allowed processes to specify which System calls are allowed. There was also the addition of the `seccomp()` System call in 2014 to kernel version 3.17. Like other popular applications such as Chrome/Chromium and OpenSSH, Docker uses SecComp to reduce the attack surface on the kernel APIs.

Docker has disabled about 44 system calls in its default (seccomp) container profile (default.json) out of well over 300 available in the Linux kernel. Docker calls this "*moderately protective while providing wide application compatibility*". It appears that ease of use is the first priority. Again, plenty of opportunity here for reducing the attack surface on the kernel APIs. For example, the `keyctl` System call was removed from the default Docker container profile after vulnerability CVE-2016-0728 was discovered, which allows privilege escalation or denial of service. CVE-2014-3153 is another vulnerability accessible from the `futex` System call which is white listed in the default Docker profile.

If you are looking to attack the Linux kernel via its APIs from a Docker container, you still have plenty of surface area here to play with.

SecComp (Countermeasures)

First, you need to make sure your Docker instance was built with Seccomp. Using the recommended command from the CIS Docker Benchmark:

```
docker ps --quiet | xargs docker inspect --format '{{ .Id }}: SecurityOpt={{ .HostConfig.Secu\
rityOpt }}'
# Should return without a value, or your modified seccomp profile, discussed soon.
# If [seccomp:unconfined] is returned, it means the container is running with
# no restrictions on System calls.
# Which means the container is running without any seccomp profile.
```

Confirm that your kernel is configured with `CONFIG_SECCOMP`:

```
cat /boot/config-`uname -r` | grep CONFIG_SECCOMP=
# Should return the following if it is:
CONFIG_SECCOMP=y
```

To add system calls to the list of syscalls you want to block for your container, take a copy of the default seccomp profile for containers (`default.json`) which contains a whitelist of the allowed system calls, and remove the system calls you want blocked. Then, run your container with the `--security-opt` option to override the default profile with a copy that you have modified:

```
docker run --rm -it --security-opt seccomp=/path/to/seccomp/profile.json hello-world
```

Read-only Containers

In order to set up read-only hosts, physical or virtual, there is a lot of work to be done, and in some cases, it becomes challenging to stop an Operating System writing to some files. I discussed this in depth in the subsections "Partitioning on OS Installation" and "Lock Down the Mounting of Partitions" in the VPS chapter of my book: Fascicle 1 of Holistic Info-Sec for Web Developers. In contrast, running Docker containers as read-only is trivial.

Running a container with the `--read-only` flag stops writes to the container.

This can sometimes be a little to constraining, as your application may need to write some temporary data locally. You could volume mount a host directory into your container, but this would obviously expose that temporary data to the host, and also other containers that may mount the same host directory. To stop other containers sharing your mounted volume, you would have to employ labeling with the likes of LSM and apply the `Z` suffix at volume mount time.

A better, easier and simpler solution would be to apply the `--tmpfs` flag to one or more directories. `--tmpfs` allows the creation of tmpfs (appearing as a mounted file system, but stored in volatile memory) mounts on any local directory, which solves the problem of not being able to write to read-only containers.

If an existing directory is specified with the `--tmpfs` option, you will experience similar behaviour to that of mounting an empty directory onto an existing one. The directory is initially empty, any additions or modifications to the directories contents will not persist past container stop.

The following is an example of running a container as read-only with a writeable tmpfs `/tmp` directory:

```
docker run -it --rm --read-only --tmpfs /tmp --name=my-read-only-container ubuntu
```

The default mount flags with `--tmpfs` are the same as the Linux default `mount` flags, if you do not specify any `mount` flags the following will be used:

```
rw,noexec,nosuid,nodev,size=65536k
```

3. runC and Where it Fits in

Docker engine is now built on containerd and runC. Engine creates the image indirectly via containerd -> runC using libcontainer -> and passes it to containerd.

containerd (daemon for Linux or Windows):
containerd is based on the Docker engine's core container runtime. It manages the complete container life-cycle, managing primitives on Linux and Windows hosts such as the following, whether directly or indirectly:

- Image transfer and storage
- Container execution and supervision
- Management of network interfaces
- Local storage
- Native plumbing level API
- Full Open Container Initiative (OCI) support: image and runtime (runC) specification

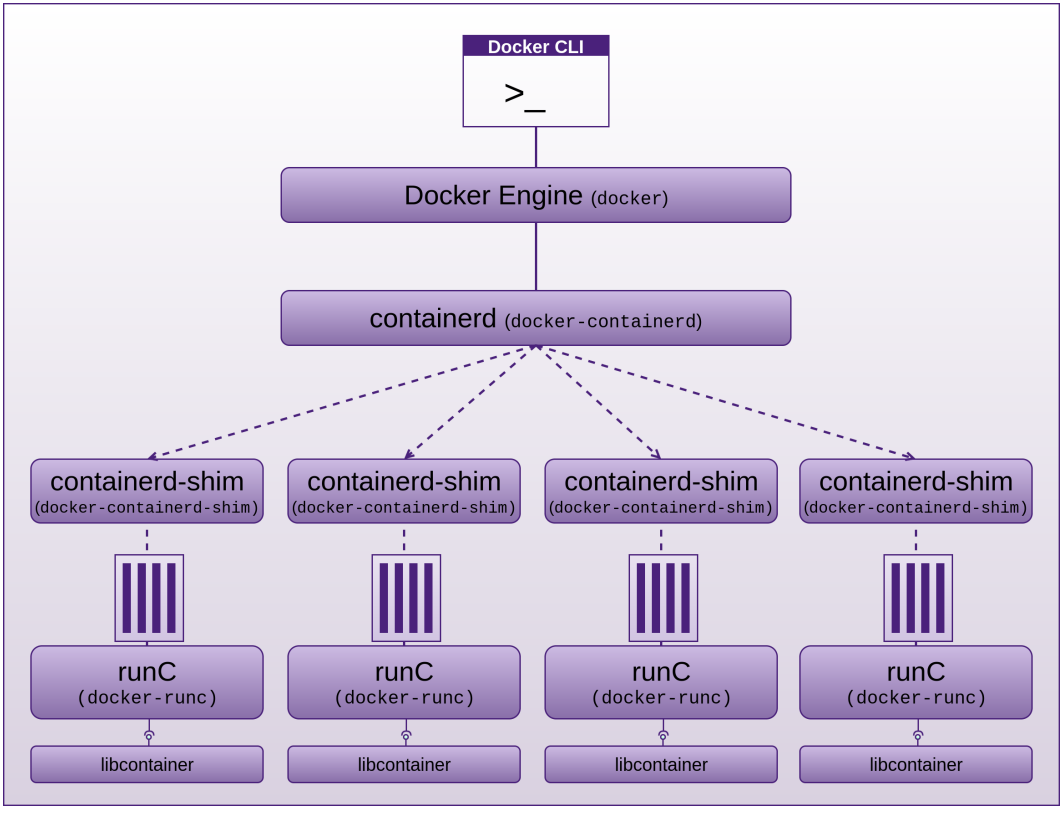
`containerd` calls `containerd-shim` which uses runC to run the container. `containerd-shim` allows the runtime, which is `docker-runc` in Docker's case, to exit once it has started the container, thus allowing the container to run without a daemon. You can see this if you run `ps aux | grep docker`
In fact, if you run this command you will see how all the components hang together. Viewing this output along with the diagram below, will help solidify your understanding of the relationships between the components.

runC is the container runtime that runs containers (think, run Container) according to the OCI specification, runC is a small standalone command line tool (CLI) built on and providing interface to libcontainer, which does most of the work. runC provides interface with:

- Linux Kernel Namespaces
- Cgroups
- Linux Security Modules
- Capabilities
- Seccomp

These features have been integrated into the low level, light weight, portable, container runtime CLI called runC, with libcontainer doing the heavy lifting. It has no dependency on the rest of the Docker platform, and has all the code required by Docker to interact with the container specific system features. More correctly, libcontainer is the library that interfaces with the above mentioned kernel features. runC leverages libcontainer directly, without the Docker engine being required in the middle.

runC was created by the OCI, whose goal is to have an industry standard for container runtimes and formats, attempting to ensure that containers built for one engine can run on other engines.



Using runC Standalone

runC can be installed separately, but it does come with Docker in the form of `docker-runc` as well. Just run it to see the available commands and options.

runC allows us to configure and debug many of the above mentioned points we have discussed. If you want, or need to get to a lower level with your containers, using `runC` (or if you have Docker installed, `docker-runc`), directly can be a useful technique to interact with your containers. It does require additional work that `docker run` commands already do for us. First, you will need to create an OCI bundle, which includes providing configuration for the host independent `config.json` and host specific `runtime.json` files. You must also construct or export a root filesystem, which if you have Docker installed you can export an existing containers root filesystem with `docker export`.

A container manifest (`config.json`) can be created by running:

```
runc spec
```

which creates a manifest according to the Open Container Initiative (OCI)/runc specification. Engineers can then add any additional attributes such as capabilities on top of the three specified within a container manifest created by the `runc spec` command.

4. Application Security

Exploitability: EASY | Prevalence: COMMON | Detectability: EASY | Impact: MODERATE

Application security is still our biggest weakness. I cover this in many other places, especially in the Web Applications chapter of my book: Fascicle 1, of Holistic Info-Sec for Web Developers.

Prevention: AVERAGE

Yes, container security is important, but in most cases, it is not the lowest hanging fruit for an attacker.

Application security is still the weakest point for compromise. It is usually much easier to attack an application running in a container, or anywhere for that matter, than it is to break container isolation or any security offered by containers and their infrastructure. Once an attacker has exploited any one of the commonly exploited vulnerabilities, such as any of the OWASP Top 10, still being introduced and found in our applications on a daily basis, and subsequently performs remote code execution, then exfils the database, no amount of container security is going to mitigate this.

During and before my interview of Diogo Monica on Docker Security for the Software Engineering Radio show, we discussed isolation concepts, many of which I have covered above. Diogo mentioned: "why does isolation even matter when an attacker already has access to your internal network?" There are very few attacks that require escaping from a container or VM in order to succeed, there are just so many easier approaches to compromise. Yes, this may be an issue for the cloud providers that are hosting containers and VMs, but for most businesses, the most common attack vectors are still attacks focussing on our weakest areas, such as people, password stealing, spear phishing, uploading and execution of web shells, compromising social media accounts, weaponised documents, and ultimately application security, as I have mentioned many times before.

Diogo and I also had a discussion about the number of container vs VM vulnerabilities, and it is pretty clear that there are far more vulnerabilities affecting VMs than there are affecting containers.

VMs have memory isolation, but many of the bugs listed in the Xen CVEs alone circumvent memory isolation benefits that VMs may have provided.

Another point that Diogo raised was the ability to monitor, inspect, and control the behaviour of applications within containers. In VMs there is so much activity that is unrelated to your applications, so although you can monitor activity within VMs, the noise to signal ratio is just too high to get accurate indications of what is happening in and around your application that actually matters to you. VMs also provide very little ability to control the resources associated with your running application(s). Inside of a container, you have your application and hopefully little else. With the likes of Control Groups you have many points at which you can monitor and control aspects of the application environment.

As mentioned above, Docker containers are immutable, and can be run read-only.

The Secure Developer podcast with Guy Podjarny interviewing Ben Bernstein (CEO and founder of Twistlock) - show #7 Understanding Container Security also echo's these same sentiments.

Also be sure to check the Additional Resources chapter for many excellent resources I collected along the way on Docker security.

Additional Resources

Cisecurity
has an excellent resource for hardening docker images, which the Docker Security team helped with. The CIS Benchmark for Docker should be consulted in parallel to reading this book

I also conducted an interview called "Docker Security"
for Software Engineering Radio in which Docker Security Team Lead Diogo Monica appeared as guest and provided some excellent advice, opinions, and food for thought, be sure to listen to it

Network Namespace source code
https://github.com/torvalds/linux/blob/master/net/core/net_namespace.c

IP-NETNS man page
http://man7.org/linux/man-pages/man8/ip-netns.8.html

Introducing Linux Network Namespaces
http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/

Network namespaces
https://blogs.igalia.com/dpino/2016/04/10/network-namespaces/

docker network
https://docs.docker.com/engine/reference/commandline/network/

Namespaces in operation
https://lwn.net/Articles/580893/

dockerscan may be worth keeping an eye on for offensive testing
https://github.com/cr0hn/dockerscan

Docker SELinux Man Page
https://www.mankier.com/8/docker_selinux

Increasing Attacker Cost using Immutable Infrastructure
https://diogomonica.com/2016/11/19/increasing-attacker-cost-using-immutable-infrastructure/

Diogo Monica on Mutual TLS
https://www.youtube.com/watch?v=apma_C24W58

Diogo Monica on Orchestrating Least Privilege

- https://www.youtube.com/watch?v=xpGNAiA3XW8
- https://www.slideshare.net/Docker/orchestrating-least-privilege-by-diogo-monica-67186063

Comparison of secrets across orchestrators
https://medium.com/on-docker/secrets-and-lie-abilities-the-state-of-modern-secret-management-2017-c82ec9136a3d#.f6yba66ti

Description of how PKI automatically gets setup in swarm
https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/

Image signing, and why it is important
https://blog.docker.com/2015/08/content-trust-docker-1-8/

Docker security scanning (content integrity)
https://blog.docker.com/2016/05/docker-security-scanning/

Attributions

Cover

The cover image was sourced from Kurzon with major changes made. Licensed under Creative Commons.

Habitat

Discuss many of my concerns and ideas
with the Docker Security team lead, Diogo Mónica
http://www.se-radio.net/2017/05/se-radio-episode-290-diogo-monica-on-docker-security/

As noted by banyan
https://www.banyanops.com/blog/analyzing-docker-hub/
and the morning paper
https://blog.acolyer.org/2017/04/03/a-study-of-security-vulnerabilities-on-docker-hub/

The Docker overview says: "*Docker provides the ability to package and run an application in a loosely isolated environment*"
https://docs.docker.com/engine/docker-overview/

Cisecurity has an excellent resource for hardening docker images which the Docker Security team helped with
https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.13.0_Benchmark_v1.0.0.pdf

Habitat - Risks

The Docker Registry project is an open-source server side application that lets you store and distribute Docker images
https://github.com/docker/distribution

Considering these processes run as root, and have indirect access to most of the Linux Kernel
https://theinvisiblethings.blogspot.co.nz/2012/09/how-is-qubes-os-different-from.html

All before any security is added on top in the form of LXC, or libcontainer (now opencontainers/runc)
https://github.com/opencontainers/runc

Hardening Docker Host, Engine and Containers - Risks

The first place to read for solid background on Linux kernel namespaces is the man-page
http://man7.org/linux/man-pages/man7/namespaces.7.html

The hosts mounted `host-path` is shared with all others that mount `host-path`
https://docs.docker.com/engine/reference/run/#volume-shared-filesystems

If you omit the `host-path` you can see the host path that was mounted
https://docs.docker.com/engine/tutorials/dockervolumes/#locating-a-volume

Further details can be found at the dockervolumes documentation
https://docs.docker.com/engine/admin/volumes/volumes/

`PID` namespaces are hierarchically nested in ancestor-descendant relationships to a depth of up to 32 levels
https://lwn.net/Articles/531419/

The default behaviour can however be overridden to allow a container to be able to access processes within a sibling container, or the hosts `PID` namespace
https://docs.docker.com/engine/reference/run/#pid-settings-pid

As an aside, `PID` namespaces give us the functionality of "*suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.*" http://man7.org/linux/man-pages/man7/pid_-namespaces.7.html
with a handful of commands
https://www.fir3net.com/Containers/Docker/the-essential-guide-in-transporting-your-docker-containers.html

A UTS namespace is the set of identifiers returned by `uname`
http://man7.org/linux/man-pages/man2/clone.2.html

When a container is created, a UTS namespace is copied (`CLONE_NEWUTS` is set)
https://github.com/docker/libcontainer/blob/83a102cc68a09d890cce3b6c2e5c14c49e6373a0/SPEC.md

When a container is created with `--uts="host"` a UTS namespace is inherited from the host
https://docs.docker.com/engine/reference/run/#uts-settings-uts

According to the namespaces man page "*Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces.*" http://man7.org/linux/man-pages/man7/namespaces.7.html

This behaviour can be overridden to allow a (any) container to reuse another containers or the hosts message queues, semaphores, and shared memory via their IPC namespace
https://docs.docker.com/engine/reference/run/#ipc-settings-ipc

You can see using the command supplied from the CIS_Docker_1.13.0_Benchmark
https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.13.0_Benchmark_v1.0.0.pdf

There are currently some Docker features that are incompatible with using user namespaces
https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-user-namespace-options

Docker engine reference provides additional details around known restrictions of user namespaces
https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-user-namespace-options

Cgroups have been available in the Linux kernel since January 2008 (2.6.24)
https://kernelnewbies.org/Linux_2_6_24#head-5b7511c1e918963d347abc8ed4b75215877d3aa3

According to the Linux man page for capabilities "*Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled*" http://man7.org/linux/man-pages/man7/capabilities.7.html

Dan Walsh who is one of the experts when it comes to applying least privilege to containers, also discusses these
http://rhelblog.redhat.com/2016/10/17/secure-your-containers-with-this-one-weird-trick/

Open Container Initiative (OCI) runC specification
https://github.com/opencontainers/runc/tree/6c22e77604689db8725fa866f0f2ec0b3e8c3a07#running-containers

As stated on the Docker Engine security page "*One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently, or when used in combination with kernel vulnerabilities.*" https://docs.docker.com/engine/security/security/

The core Unix security model which is a form of Discretionary Access Control (DAC) was inherited by Linux
https://en.wikipedia.org/wiki/Discretionary_access_control

The Unix DAC was designed in 1969
https://www.linux.com/learn/overview-linux-kernel-security-features

The first version of SecComp was merged into the Linux kernel mainline in version 2.6.12 (March 8 2005)
https://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=d949d0ec9c601f2b148bed3cdb5f87c052968554

In order to enable SecComp for a given process, you would write a `1` to `/proc/<PID>/seccomp`
https://lwn.net/Articles/656307/

Then the addition of the `seccomp()` System call in 2014 to the kernel version 3.17 along with popular applications such as Chrome/Chromium, OpenSSH
https://en.wikipedia.org/wiki/Seccomp

Docker has disabled about 44 system calls in its default (seccomp) container profile
https://docs.docker.com/engine/security/seccomp/
https://github.com/docker/docker/blob/master/profiles/seccomp/default.json

The `keyctl` System call was removed from the default Docker container profile after vulnerability CVE-2016-0728 was discovered, which allows privilege escalation or denial of service
https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728
https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153

Hardening Docker Host, Engine and Containers - Countermeasures

"***Docker Security Scanning*** *is available as an add-on to Docker hosted private repositories on both Docker Cloud and Docker Hub.*", you also have to opt in and pay for it
https://docs.docker.com/docker-cloud/builds/image-scan/#opt-in-to-docker-security-scanning

Docker Security Scanning is also now available on the new Enterprise Edition
https://blog.docker.com/2017/03/docker-enterprise-edition/

Whether un-official or official
https://github.com/docker-library/official-images

Docker Content Trust
https://blog.docker.com/2015/08/content-trust-docker-1-8/

Notary
https://github.com/docker/notary

`DOCKER_CONTENT_TRUST` environment variable must be set to `1`
https://docs.docker.com/engine/security/trust/content_trust/#enable-and-disable-content-trust-per-shell-or-per-invocation

`DOCKER_CONTENT_TRUST_SERVER` must be set to the URL of the Notary server you setup
https://docs.docker.com/engine/reference/commandline/cli/#environment-variables

They need to generate a key pair
https://docs.docker.com/engine/security/trust/trust_delegation/

Notary is based on a Go implementation of The Update Framework (TUF)
https://theupdateframework.github.io/

An example of the NodeGoat image
https://github.com/owasp/nodegoat

The space for tooling to help find vulnerabilities in code, packages, etc within your Docker images has been noted, and tools provided
https://community.alfresco.com/community/ecm/blog/2015/12/03/docker-security-tools-audit-and-vulnerability-assessment/

These tools should form a part of your secure and trusted build pipeline / software supply-chain
https://blog.acolyer.org/2017/04/03/a-study-of-security-vulnerabilities-on-docker-hub/

Dockerfile linter that helps you build best practice Docker images
https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

Free and open source auditing tool for Linux/Unix based systems
https://github.com/CISOfy/lynis

Docker plugin available which allows one to audit Docker
https://cisofy.com/lynis/plugins/docker-containers/

Hashes of the CVE data sources
https://github.com/coreos/clair/tree/f66103c7732c9a62ba1d3afc26437ae54953dc01#default-data-sources

Collector has a pluggable, extensible architecture
https://github.com/banyanops/collector/blob/master/docs/CollectorDetails.md

Banyanops was the organisation that blogged about the high number of vulnerable packages on Docker Hub
https://www.banyanops.com/blog/analyzing-docker-hub/

Seen by running `docker network ls`
https://docs.docker.com/engine/reference/commandline/network_ls/

Docker network
https://docs.docker.com/engine/userguide/networking/

Network drivers created by docker
https://docs.docker.com/engine/reference/run/#network-settings

`bridge`
https://docs.docker.com/engine/reference/run/#network-bridge

`none`
https://docs.docker.com/engine/reference/run/#network-none

`host`
https://docs.docker.com/engine/reference/run/#network-host

`container`
https://docs.docker.com/engine/reference/run/#network-container

`nsenter` command
http://man7.org/linux/man-pages/man1/nsenter.1.html

Understand container communication
https://docs.docker.com/v17.09/engine/userguide/networking/default_network/container-communication/

The username must exist in the `/etc/passwd` file, the `sbin/nologin` users are valid also
https://success.docker.com/KBase/Introduction_to_User_Namespaces_in_Docker_Engine

"***The UID/GID we want to remap to*** *does not need to match the UID/GID of the username in `/etc/passwd`*" https://success.docker.com/KBase/Introduction_to_User_Namespaces_in_Docker_Engine

Files will be populated with a contiguous 65536 length range of subordinate user and group Ids respectively
https://docs.docker.com/engine/security/userns-remap/

Check out the Docker engine reference
Updated URL: https://github.com/jquast/docker/blob/2fd674a00f98469caa1ceb572e5ae92a68b52f44/docs/reference/commandline/dockerd.md#detailed-information-on-subuidsubgid-ranges

Check the Runtime constraints on resources
https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources

Limit a container's resources Admin Guide for Docker Engine
https://docs.docker.com/engine/admin/resource_constraints/

By default Docker uses the cgroupfs cgroup driver to interface with the Linux kernel's cgroups
https://docs.docker.com/engine/reference/commandline/dockerd/#options-for-the-runtime

`docker stats` command, which will give you a line with your containers CPU usage, Memory usage and Limit, Net I/O, Block I/O, Number of PIDs
https://docs.docker.com/engine/reference/commandline/stats/

Docker engine runtime metrics
https://docs.docker.com/engine/admin/runmetrics/

With a little help from the CIS Docker Benchmark we can use the `PIDs` cgroup limit
https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.13.0_Benchmark_v1.0.0.pdf

There are several ways you can minimise your set of capabilities
http://rhelblog.redhat.com/2016/10/17/secure-your-containers-with-this-one-weird-trick/

First Linux kernel summit
https://lwn.net/2001/features/KernelSummit/

It was decided to have the developers interested in security create a "*generic interface which could be used by any security policy. The result was the Linux Security Modules (LSM)*" API/framework, which provides many hooks at security critical points within the kernel
http://www.hep.by/gnu/kernel/lsm/
https://lwn.net/Articles/180194/
https://www.linux.com/learn/overview-linux-kernel-security-features

Selectable at build-time via `CONFIG_DEFAULT_SECURITY`
https://www.kernel.org/doc/Documentation/security/LSM.txt

Overridden at boot-time via the `security=...` kernel command line argument
https://debian-handbook.info/browse/stable/sect.selinux.html#sect.selinux-setup

"***Most LSMs choose to extend the capabilities*** *system, building their checks on top of the defined capability hooks.*" https://www.kernel.org/doc/Documentation/security/LSM.txt

AppArmor policy's are created using the profile language
http://wiki.apparmor.net/index.php/ProfileLanguage

Apparmor page of Dockers Secure Engine
https://docs.docker.com/engine/security/apparmor/

SELinux needs to be installed and configured on Debian
https://wiki.debian.org/SELinux/Setup

SELinux support for the Docker daemon is disabled by default and needs to be enabled
https://github.com/GDSSecurity/Docker-Secure-Deployment-Guidelines
https://docs.docker.com/engine/reference/commandline/dockerd/

Docker daemon options can also be set within the daemon configuration file
https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file

Label confinement for the container can be configured using `--security-opt`
https://github.com/GDSSecurity/Docker-Secure-Deployment-Guidelines

SELinux Labels for Docker consist of four parts
https://www.projectatomic.io/docs/docker-and-selinux/

SELinux can be enabled in the container using `setenforce 1`
http://www.unix.com/man-page/debian/8/setenforce/

SELinux can operate in one of three modes
https://www.centos.org/docs/5/html/5.2/Deployment_Guide/sec-sel-enable-disable-enforcement.html

To persist on boot: In Debian
https://debian-handbook.info/browse/stable/sect.selinux.html#sect.selinux-setup

Kernel is configured with `CONFIG_SECCOMP`
https://docs.docker.com/engine/security/seccomp/

Default seccomp profile for containers (`default.json`)
https://github.com/docker/docker/blob/master/profiles/seccomp/default.json

Apply the `--tmpfs` flag
https://docs.docker.com/engine/reference/commandline/run/#mount-tmpfs—tmpfs

runC and Where it Fits in

libcontainer
https://github.com/opencontainers/runc/tree/master/libcontainer

containerd (daemon for Linux or Windows) is based on the Docker engine's core container runtime
https://containerd.io/

runC is the container runtime that runs containers
https://runc.io/

runC was created by the OCI
https://github.com/opencontainers/runc

runC can be installed separately
https://docker-saigon.github.io/post/Docker-Internals/#runc:cb6baf67dddd3a71c07abfd705dc7d4b

Host independent `config.json` and host specific `runtime.json` files
https://github.com/containerd/containerd/blob/0.0.5/docs/bundle.md#configs

You must also construct or export a root filesystem
https://github.com/opencontainers/runc#creating-an-oci-bundle

Application Security

The most common attack vectors are still attacks focussing on our weakest areas, such as people, password stealing, spear phishing, uploading and execution of web shells, compromising social media accounts, weaponised documents, and ultimately application security, as I have mentioned many times before
https://binarymist.io/talk/js-remote-conf-2017-the-art-of-exploitation/

It is pretty clear that there are far more vulnerabilities affecting VMs than there are affecting containers
https://xenbits.xen.org/xsa/
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=docker

Bugs listed in the Xen CVEs
https://xenbits.xen.org/xsa/

Show #7 Understanding Container Security
http://www.heavybit.com/library/podcasts/the-secure-developer/ep-7-understanding-container-security/