Community Experience Distilled

Securing Docker

Learn how to secure your Docker environment and keep your environments secure irrespective of the threats out there

Scott Gallagher

[PACKT] PUBLISHING open source* community experience distilled

Securing Docker

Learn how to secure your Docker environment and keep your environments secure irrespective of the threats out there

Scott Gallagher

[PACKT] open source*
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Securing Docker

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2016

Production reference: 1230316

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-885-4

www.packtpub.com

Credits

Author
Scott Gallagher

Reviewer
Harald Albers

Commissioning Editor
Priya Singh

Acquisition Editor
Prachi Bisht

Content Development Editor
Arshiya Ayaz Umer

Technical Editor
Suwarna Patil

Copy Editor
Vibha Shukla

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Nilesh Mohite

Cover Work
Nilesh Mohite

About the Author

Scott Gallagher has been fascinated with technology since he was in elementary school, when he used to play Oregon Trail. His love continued through middle school, working on more Apple IIe computers. In high school, he learned how build computers and program in BASIC! His college years were all about server technologies such as Novell, Microsoft, and Red Hat. After college, he continued to work on Novell, all while keeping an interest in all the technologies. He then moved into managing Microsoft environments and eventually into what he is the most passionate about, Linux environments, and now his focus is on Docker and cloud environments.

I would like to thank my family for the support they have given me, not only throughout the work on this book, but throughout my life and career. I would like to thank my wife, who is my soulmate, the love of my life, and the most important person in my life and the reason I push myself to be the best I can be each day. I would also like to thank my kids, who are the most amazing kids in this world, for being able to watch them grow each day; I truly am blessed. Finally, I would like to thank my parents, who have helped me become the person I am today.

About the Reviewer

Harald Albers works as a Java developer and security engineer in Hamburg, Germany.

In addition to developing distributed web applications, he also sets up and maintains the build infrastructure, staging, and production environments for these applications.

Most of his work is only possible because of Docker's simple and elegant solutions for the challenges of provisioning, deployment, and orchestration.

He started using Docker and contributing to the Docker project in mid-2014. He is a member of the Docker Governance Advisory Board, 2015-2016.

Table of Contents

Preface

Docker is the hottest buzzword in technology these days! This book helps you to ensure that you are securing all the pieces in the Docker ecosystems of tools. Keeping your data and systems safe is of utmost importance these days, and with Docker, it's the same exception. Learn how Docker is inherently secure and how to secure the pieces around it even more and be on the lookout for potential vulnerabilities as they take place.

What this book covers

Chapter 1, *Securing Docker Hosts*, starts off the book by discussing how to secure the first part of getting your Docker environment up and running, and that is by focusing on your Docker hosts. The Docker hosts are the platform that your containers will run on. Without securing these first, it's like leaving the front door to your house wide open.

Chapter 2, *Securing Docker Components,* focuses on securing the components of Docker, such as the registry you can use, the containers that run on your hosts, and how to sign your images.

Chapter 3, *Securing and Hardening Linux Kernels,* explains hardening guides that are out there as well as different security measures/methods you can use to help secure the kernel that is being used to run your containers as it's important to secure it.

Chapter 4, *Docker Bench for Security,* informs how well you have set up your Docker environment with the Docker Bench Security application, get recommendations for where you should focus your efforts to fix right away, and what you don't really have to fix right now, but should keep yourself aware of.

Chapter 5, Monitoring and Reporting Docker Security Incidents, covers how to stay on top of the items that Docker has released regarding the security findings to help keep you aware of your environments. Also, we will take a look at how to safely report any security findings you come across to ensure that Docker has a chance to alleviate the concern before it becomes public and widespread.

Chapter 6, Using Docker's Built-in Security Features, introduces the use of Docker tools to help secure your environment. We will go over all of them to give you a baseline of what you can use that is provided by Docker itself. You can learn what command-line and GUI tools you can use for your security needs.

Chapter 7, Securing Docker with Third-party Tools, covers the third-party tools that are out there to help you keep your Docker environment secure. You will learn about command line, but we'll focus on third-party tools. We will take a look at traffic authorization, summon, and sVirt with SELinux.

Chapter 8, Keeping up on Security, explains the means that you can use to keep up to date with Docker-related security issues that are out there for the version of the Docker tools you might be running now, how to stay ahead of any security issues, and keep your environments secure even with threats out there.

What you need for this book

The book will walk you through the installation of any tools that you will need. You will need a system with Windows, Mac OS, or Linux installed; preferably, the latter one, as well as an Internet connection.

Who this book is for

This book is intended for those developers who will be using Docker as their testing platform as well as security professionals who are interested in securing Docker containers. Readers must be familiar with the basics of Docker.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You will need `pass phrase` you entered earlier for `ca-key.pem`."

Any command-line input or output is written as follows:

```
$ docker run -it scottpgallagher/chef-server /bin/bash
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The next section, **Security** settings, is probably one of the most important ones."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1
Securing Docker Hosts

Welcome to the *Securing Docker* book! We are glad you decided to pick up the book and we want to make sure that the resources you are using are being secured in proper ways to ensure system integrity and data loss prevention. It is also important to understand why you should care about the security. If data loss prevention doesn't scare you already, thinking about the worst possible scenario—a full system compromise and the possibility of your secret designs being leaked or stolen by others—might help to reinforce security. Throughout this book, we will be covering a lot of topics to help get your environment set up securely so that you can begin to start deploying containers with peace of mind knowing that you took the right steps in the beginning to fortify your environment. In this chapter, we will be taking a look at securing Docker hosts and will be covering the following topics:

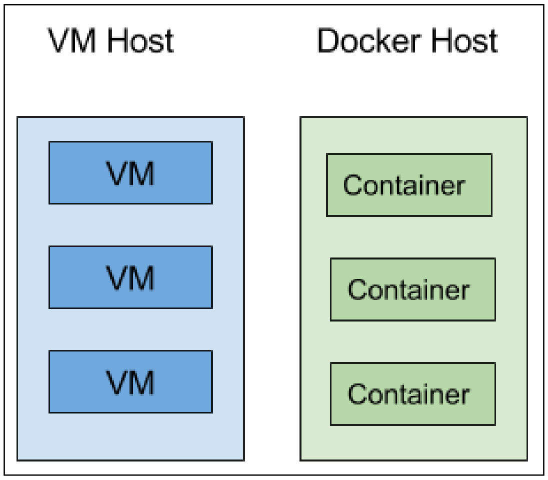
- Docker host overview
- Discussing Docker host
- Virtualization and isolation
- Attack surface of Docker daemon
- Securing Docker hosts
- Docker Machine
- SELinux and AppArmor
- Auto-patching hosts

Docker host overview

Before we get in depth and dive in, let's first take a step back and review exactly what the Docker host is. In this section, we will look at the Docker host itself to get an understanding of what we are referring to when we are talking about the Docker host. We will also be looking at the virtualization and isolation techniques that Docker uses to ensure security.

Discussing Docker host

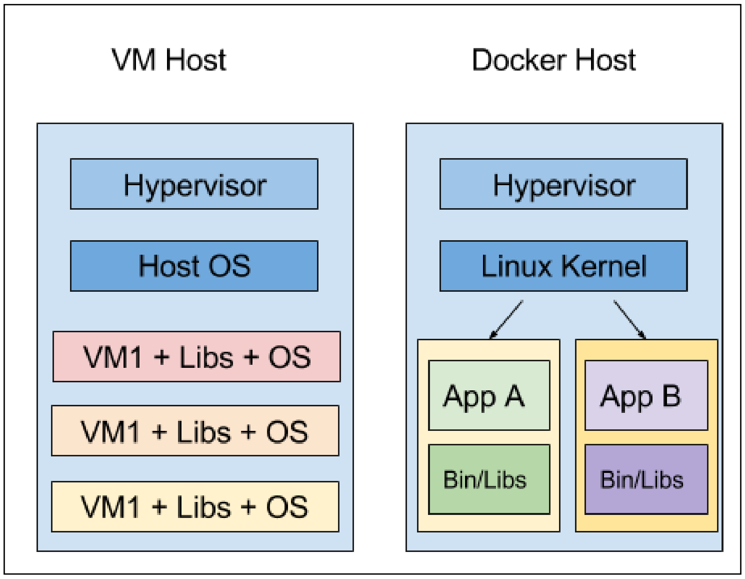
When we think of a Docker host, what comes to our mind? If you put it in terms of virtual machines that almost all of us are familiar with, let's take a look at how a typical VM host differs from a Docker host. A **VM host** is what the virtual machines actually run on top of. Typically, this is something like **VMware ESXi** if you are using VMware or **Windows Server** if you are using **Hyper-V**. Let's take a look at how they are as compared so that you can get a visual representation of the two, as shown in the following diagram:



The preceding image depicts the similarities between a **VM host** and **Docker host**. As stated previously, the host of any service is simply the system that the underlying virtual machines or containers in Docker run on top of. Therefore, a host is the operating system or service that contains and operates the underlying systems that you install and set up a service on, such as web servers, databases, and more.

Virtualization and isolation

To understand how Docker hosts can be secured, we must first understand how the **Docker host** is set up and what items are contained in the **Docker host**. Again, like VM hosts, they contain the operating system that the underlying service operates on. With VMs, you are creating a whole new operating system on top of this **VM host** operating system. However, on Docker, you are not doing that and are sharing the **Linux Kernel** that the **Docker host** is using. Let's take a look at the following diagram to help us represent this:



As we can see from the preceding image, there is a distinct difference between how items are set up on a **VM host** and on a **Docker host**. On a **VM host**, each virtual machine has all of its own items inclusive to itself. Each containerized application brings its own set of libraries, whether it is Windows or Linux. Now, on the **Docker host**, we don't see that. We see that they share the **Linux Kernel** version that is being used on the **Docker host**. That being said, there are some security aspects that need to be addressed on the **Docker host** side of things. Now, on the **VM host** side, if someone does compromise a virtual machine, the operating system is isolated to just that one virtual machine. Back on the **Docker host** side of things, if the kernel is compromised on the **Docker host**, then all the containers running on that host are now at high risk as well.

So, now you should see how important it is that we focus on security when it comes to Docker hosts. Docker hosts do use some isolation techniques that will help protect against kernel or container compromises in a few ways. Two of these ways are by implementing **namespaces** and **cgroups**. Before we can discuss how they help, let's first give a definition for each of them.

Kernel namespaces, as they are commonly known as, provide a form of isolation for the containers that will be running on your hosts. What does this mean? This means that each container that you run on top of your Docker hosts will be given its own network stack so that it doesn't get privileged access to another container's socket or interfaces. However, by default, all Docker containers are sitting on the bridged interface so that they can communicate with each other easily. Think of the bridged interface as a network switch that all the containers are connected to.

Namespaces also provide isolation for processes and mount isolation. Processes running in one container can't affect or even see processes running in another Docker container. Isolation for mount points is also on a container by container basis. This means that mount points on one container can't see or interact with mount points on another container.

On the other hand, control groups are what control and limit resources for containers that will be running on top of your Docker hosts. What does this boil down to, meaning how will it benefit you? It means that cgroups, as they will be called going forward, help each container get its fair share of memory disk I/O, CPU, and much more. So, a container cannot bring down an entire host by exhausting all the resources available on it. This will help to ensure that even if an application is misbehaving that the other containers won't be affected by this application and your other applications can be assured uptime.

Attack surface of Docker daemon

While Docker does ease some of the complicated work in the virtualization world, it is easy to forget to think about the security implications of running containers on your Docker hosts. The largest concern you need to be aware of is that Docker requires root privileges to operate. For this reason, you need to be aware of who has access to your Docker hosts and the Docker daemon as they will have full administrative access to all your Docker containers and images on your Docker host. They can start new containers, stop existing ones, remove images, pull new images, and even reconfigure running containers as well by injecting commands into them. They can also extract sensitive information like passwords and certificates from the containers. For this reason, make sure to also separate important containers if you do need to keep separate controls on who has access to your Docker daemon. This is for containers where people have a need for access to the Docker host where the containers are running. If a user needs API access then that is different and separation might not be necessary. For example, keep containers that are sensitive on one Docker host, while keeping normal operation containers running on another Docker host and grant permissions for other staff access to the Docker daemon on the unprivileged host. If possible, it is also recommended to drop the setuid and setgid capabilities from containers that will be running on your hosts. If you are going to run Docker, it's recommended to only use Docker on this server and not other applications. Docker also starts containers with a very restricted set of capabilities, which works in your favor to address security concerns.

> To drop the setuid or setgid capabilities when you start a Docker container, you will need to do something similar to the following:
>
> ```
> $ docker run -d --cap-drop SETGID --cap-drop SETUID nginx
> ```
>
> This would start the `nginx` container and would drop the `SETGID` and `SETUID` capabilities for the container.

Docker's end goal is to map the root user to a non-root user that exists on the Docker host. They also are working towards allowing the Docker daemon to run without requiring root privileges. These future improvements will only help facilitate how much focus Docker does take when they are implementing their feature sets.

Protecting the Docker daemon

To protect the Docker daemon even more, we can secure the communications that our Docker daemon is using. We can do this by generating certificates and keys. There are are few terms to understand before we dive into the creation of the certificates and keys. A **Certificate Authority** (**CA**) is an entity that issues certificates. This certificate certifies the ownership of the public key by the subject that is specified in the certificate. By doing this, we can ensure that your Docker daemon will only accept communication from other daemons that have a certificate that was also signed by the same CA.

Now, we will be looking at how to ensure that the containers you will be running on top of your Docker hosts will be secure in a few pages; however, first and foremost, you want to make sure the Docker daemon is running securely. To do this, there are some parameters you will need to enable for when the daemon starts. Some of the things you will need beforehand will be as follows:

1. Create a CA.

   ```
   $ openssl genrsa -aes256 -out ca-key.pem 4096
   Generating RSA private key, 4096 bit long modulus
   ..................................................................
   ..................................................................
   ..................................................................
   ................++
   ..................................................................
   ..++
   e is 65537 (0x10001)
   Enter pass phrase for ca-key.pem:
   Verifying - Enter pass phrase for ca-key.pem:
   ```

You will need to specify two values, `pass phrase` and `pass phrase`. This needs to be between `4` and `1023` characters. Anything less than `4` or more than `1023` won't be accepted.

```
$ openssl req -new -x509 -days <number_of_days> -key ca-key.pem
-sha256 -out ca.pem

Enter pass phrase for ca-key.pem:

You are about to be asked to enter information that will be
incorporated

into your certificate request.

What you are about to enter is what is called a Distinguished Name
or a DN.

There are quite a few fields but you can leave some blank

For some fields there will be a default value,

If you enter '.', the field will be left blank.

-----

Country Name (2 letter code) [AU]:US

State or Province Name (full name) [Some-State]:Pennsylvania

Locality Name (eg, city) []:

Organization Name (eg, company) [Internet Widgits Pty Ltd]:

Organizational Unit Name (eg, section) []:

Common Name (e.g. server FQDN or YOUR name) []:

Email Address []:
```

There are a couple of items you will need. You will need `pass phrase` you entered earlier for `ca-key.pem`. You will also need the `Country`, `State`, `city`, `Organization Name`, `Organizational Unit Name`, **fully qualified domain name** (**FQDN**), and `Email Address` to be able to finalize the certificate.

2. Create a client key and signing certificate.

```
$ openssl genrsa -out key.pem 4096

$ openssl req -subj '/CN=<client_DNS_name>' -new -key key.pem -out
client.csr
```

3. Sign the public key.

```
$ openssl x509 -req -days <number_of_days> -sha256 -in client.csr
-CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.em
```

4. Change permissions.

```
$ chmod -v 0400 ca-key.pem key.pem server-key.em
$ chmod -v 0444 ca.pem server-cert.pem cert.em
```

Now, you can make sure that your Docker daemon only accepts connections from the other Docker hosts that you provide the signed certificates to:

```
$ docker daemon --tlsverify --tlscacert=ca.pem --tlscert=server-
certificate.pem --tlskey=server-key.pem -H=0.0.0.0:2376
```

Make sure that the certificate files are in the directory you are running the command from or you will need to specify the full path to the certificate file.

On each client, you will need to run the following:

```
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem
--tlskey=key.pem -H=<$DOCKER_HOST>:2376 version
```

Again, the location of the certificates is important. Make sure to either have them in a directory where you plan to run the preceding command or specify the full path to the certificate and key file locations.

You can read more about using **Transport Layer Security** (**TLS**) by default with your Docker daemon by going to the following link:

`http://docs.docker.com/engine/articles/https/`

For more reading on **Docker Secure Deployment Guidelines**, the following link provides a table that can be used to gain insight into some other items you can utilize as well:

`https://github.com/GDSSecurity/Docker-Secure-Deployment-Guidelines`

Some of the highlights from that website are:

- Collecting security and audit logs
- Utilizing the privileged switch when running Docker containers
- Device control groups
- Mount points
- Security audits

Securing Docker hosts

Where do we start to secure our hosts? What tools do we need to start with? We will take a look at using Docker Machine in this section and how to ensure the hosts that we are creating are being created in a secure manner. Docker hosts are like the front door of your house, if you don't secure them properly, then anybody can just walk right in. We will also take a look at **Security-Enhanced Linux** (**SELinux**) and **AppArmor** to ensure that you have an extra layer of security on top of the hosts that you are creating. Lastly, we will take a look at some of the operating systems that support and do auto patching of their operating systems when a security vulnerability is discovered.

Docker Machine

Docker Machine is the tool that allows you to install the Docker daemon onto your virtual hosts. You can then manage these Docker hosts with Docker Machine. Docker Machine can be installed either through the **Docker Toolbox** on Windows and Mac. If you are using Linux, you will install Docker Machine through a simple `curl` command:

```
$ curl -L https://github.com/docker/machine/releases/download/v0.6.0/
docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \
$ chmod +x /usr/local/bin/docker-machine
```

The first command installs Docker Machine into the `/usr/local/bin` directory and the second command changes the permissions on the file and sets it to executable.

We will be using Docker Machine in the following walkthrough to set up a new Docker host.

Docker Machine is what you should be or will be using to set up your hosts. For this reason, we will start with it to ensure your hosts are set up in a secure manner. We will take a look at how you can tell if your hosts are secure when you create them using the Docker Machine tool. Let's take a look at what it looks like when you create a Docker host using Docker Machine, as follows:

```
$ docker-machine create --driver virtualbox host1


Running pre-create checks...
Creating machine...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
```

```
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...

Setting Docker configuration on the remote daemon...
```

From the preceding output, as the create is running, Docker Machine is doing things such as creating the machine, waiting for SSH to become available, performing actions, copying the certificates to the correct location, and setting up the Docker configuration, we will see how to connect Docker to this machine as follows:

```
$ docker-machine env host1


export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/scottpgallagher/.docker/machine/machines/
host1"
export DOCKER_MACHINE_NAME="host1"
# Run this command to configure your shell:
# eval "$(docker-machine env host1)"
```

The preceding command output shows the commands that were run to set this machine up as the one that Docker commands will now run against:

```
 eval "$(docker-machine env host1)"
```

We can now run the regular Docker commands, such as `docker info`, and it will return information from `host1`, now that we have set it as our environment.

We can see from the preceding highlighted output that the host is being set up securely from the start from two of the export lines. Here is the first highlighted line by itself:

```
export DOCKER_TLS_VERIFY="1"
```

From the other highlighted output, `DOCKER_TLS_VERIFY` is being set to `1` or `true`. Here is the second highlighted line by itself:

```
export DOCKER_HOST="tcp://192.168.99.100:2376"
```

We are setting the host to operate on the secure port of `2376` as opposed to the insecure port of `2375`.

We can also gain this information by running the following command:

```
$ docker-machine ls

NAME      ACTIVE   DRIVER       STATE     URL
SWARM

host1              *        virtualbox     Running
tcp://192.168.99.100:2376
```

Make sure to check the TLS switch options that can be used with Docker Machine if you have used the previous instructions to set up your Docker hosts and Docker containers to use TLS. These switches would be helpful if you have existing certificates that you want to use as well. These switches can be found in the highlighted section by running the following command:

```
$ docker-machine --help


Options:

  --debug, -D      Enable debug mode

  -s, --storage-path "/Users/scottpgallagher/.docker/machine"
  Configures storage path [$MACHINE_STORAGE_PATH]

  --tls-ca-cert      CA to verify remotes against [$MACHINE_TLS_CA_CERT]

  --tls-ca-key      Private key to generate certificates [$MACHINE_TLS_
CA_KEY]

  --tls-client-cert     Client cert to use for TLS [$MACHINE_TLS_CLIENT_
CERT]

  --tls-client-key       Private key used in client TLS auth [$MACHINE_
TLS_CLIENT_KEY]

  --github-api-token     Token to use for requests to the Github API
[$MACHINE_GITHUB_API_TOKEN]

  --native-ssh      Use the native (Go-based) SSH implementation.
[$MACHINE_NATIVE_SSH]

  --help, -h      show help

  --version, -v      print the version
```

You can also regenerate TLS certificates for a machine using the `regenerate-certs` subcommand in the event that you want that peace of mind or that your keys do get compromised. An example command would look similar to the following command:

```
$ docker-machine regenerate-certs host1


Regenerate TLS machine certs?  Warning: this is irreversible. (y/n): y
```

```
Regenerating TLS certificates
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
```

SELinux and AppArmor

Most Linux operating systems are based on the fact that they can leverage SELinux or AppArmor for more advanced access controls to files or locations on the operating system. With these components, you can limit a container's ability to execute a program as the root user with root privileges.

Docker does ship a security model template that comes with AppArmor and Red Hat comes with SELinux policies as well for Docker. You can utilize these provided templates to add an additional layer of security on top of your environments.

For more information about SELinux and Docker, I would recommend visiting the following website:

`https://www.mankier.com/8/docker_selinux`

While, on the other hand, if you are in the market for some more reading on AppArmor and Docker, I would recommend visiting the following website:

`https://github.com/docker/docker/tree/master/contrib/apparmor`

Here you will find a `template.go` file, which is the template that Docker ships with its application that is the AppArmor template.

Auto-patching hosts

If you really want to get into advanced Docker hosts, then you could use **CoreOS** and **Amazon Linux AMI**, which perform auto-patching, both in a different way. While CoreOS will patch your operating system when a security update comes out and will reboot your operating system, the Amazon Linux AMI will complete the updates when you reboot. So, when choosing which operating system to use when you are setting up your Docker hosts, make sure to take into account the fact that both of these operating systems implement some form of auto-patching, but in a different way. You will want to make sure you are implementing some type of scaling or failover to address the needs of something that is running on CoreOS so that there is no downtime when a reboot occurs to patch the operating system.

Summary

In this chapter, we looked at how to secure our Docker hosts. The Docker hosts are the first line of defense as they are the starting point where your containers will be running and communicating with each other and end users. If these aren't secure, then there is no purpose of moving forward with anything else. You learned how to set up the Docker daemon to run securely running TLS by generating the appropriate certificates for both the host and the clients. We also looked at the virtualization and isolation benefits of using Docker containers, but make sure to remember the attack surface of the Docker daemon too.

Other items included how to use Docker Machine to easily create Docker hosts on secure operating systems with secure communication and ensure that they are being set up using secure methods when you use it to set up your containers. Using items such as SELinux and AppArmor also help to improve your security footprint as well. Lastly, we covered some Docker host operating systems that you can use for auto-patching as well, such as CoreOS and Amazon Linux AMI.

In the next chapter, we will be looking at securing the components of Docker. We will focus on securing the components of Docker such as the registry you can use, containers that run on your hosts, and how to sign your images.

2

Securing Docker Components

In this chapter, we will be taking a look at securing some Docker components using things such as image signing tools. There are tools that will help secure the environments where we are storing our images, whether they are signed or not. We will also look at using tools that come with commercial level support. Some of the tools (image signing and commercial level support tools) we will be looking at are:

- **Docker Content Trust**: Software that can be used to sign your images. We will look at all the components and go through an example of signing an image.
- **Docker Subscription**: Subscription is an all inclusive package that includes a location to store your images, and Docker Engine to run your containers, all while providing commercial level support for all those pieces, plus for the applications and their life cycle that you plan to use.
- **Docker Trusted Registry** (**DTR**): Trusted Registry gives you a secure location to store and manage your images either on premises or in the cloud. It also has a lot of integration into your current infrastructure as well. We will take a look at all the pieces available.

Docker Content Trust

Docker Content Trust is a means by which you can securely sign your Docker images that you have created to ensure that they are from who they say they are from, that being you! In this section, we will take a look at the components of **Notary** as well as an example of signing images. Lastly, we will take a peek at the latest means of using Notary with regards to hardware signing capabilities that are now available. It is a very exciting topic, so let's dive in head first!

Docker Content Trust components

To understand how Docker Content Trust works it is beneficial to be familiar with all the components that make up its ecosystem.

The first part of that ecosystem is **The Update Framework** (**TUF**) piece. TUF, as we will refer to it from now on, is the framework that Notary is built upon. TUF solves the problem with software update systems in that they can often be hard to manage. It enables users to ensure that all applications are secure and can survive any key compromises. However, if an application is insecure by default, it won't help to secure that application until it has been brought up to a secure compliance. It also enables trusted updates over untrusted sources and much more. To learn more about TUF, please visit the website:

`http://theupdateframework.com/`

The next piece of the Content Trust ecosystem is that of Notary. Notary is the key underlying piece that does the actual signing using your keys. Notary is open source software, and can be found here:

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

This has been produced by those at Docker for the use of publishing and verifying content. Notary consists of a server piece as well as a client piece. The client piece resides on your local machine and handles the storing of the keys locally as well as the communication back with the Notary server to match up timestamps as well.

Basically, there are three steps to the Notary server.

1. Compile the server
2. Configure the server
3. Run the server

Since the steps may change in the future, the best location for that information would be on the GitHub page for Docker Notary. For more information about compiling and setting up the server side of Notary, please visit:

`https://github.com/docker/notary#compiling-notary-server`

Docker Content Trust utilizes two distinct keys. The first is that of a tagging key. The tagging key is generated for every new repository that you publish. These are keys that can be shared with others and exported to those who need the ability to sign content on behalf of the registry. The other key, the offline key, is the important key. This is the key that you want to lock away in a vault and never share with anyone… *ever*! Like the name implies, this key should be kept offline and not stored on your machine or anything on a network or cloud storage. The only times you need the offline key are if you are rotating it out for a new one or if you are creating a new repository.

So, what does all this mean and how does it truly benefit you? This helps in protecting against three key, no pun intended, scenarios.

- Protects against image forgery, for instance if someone decides to pretend one of your images is from you. Without that person being able to sign that image with the repository specific key, remember the one you are to keep *offline*!, they won't be able to pass it off as actually coming from you.
- Protects against replay attacks; replay attacks are ones in which a malicious user tries to pass off an older version of an application, which has been compromised, as the latest legitimate version. Due to the way timestamps are utilized with Docker Content Trust, this will ultimately fail and keep you and your users safe.
- Protects against key compromise. If a key is compromised, you can utilize that offline key to do a key rotation. That key rotation can only be done by the one with the offline key. In this scenario, you will need to create a new key and sign it with your offline key.

The major take away from all of this is that the offline key is meant to be kept offline. Never store it on your cloud storage, on GitHub, or even a system that is always connected to the Internet such as that of your local machine. It would be best practice to store it on a thumb drive that is encrypted and keep that thumb drive stored in a a secure location.

To learn more about Docker Content Trust, please visit the following blog post:

`http://blog.docker.com/2015/08/content-trust-docker-1-8/`

Signing images

Now that we have covered all the components of Docker Content Trust, let's take a look at how we can sign an image and what all steps are involved. These instructions are just for development purposes. If you are going to want to run a Notary server in production, you will want to use your own database and compile Notary yourself using the instructions at their website:

`https://github.com/docker/notary#compiling-notary-server`

This will allow you to use your own keys for your situation to your own backend registry. If you are using the Docker Hub, it is very simple to use Docker Content Trust.

```
$ export DOCKER_CONTENT_TRUST=1
```

The most important piece is that you need to put a tag on all images you push, which we see in the next command:

```
$ docker push scottpgallagher/ubuntu:latest


The push refers to a repository [docker.io/scottpgallagher/ubuntu] (len:
1)
f50e4a66df18: Image already exists
a6785352b25c: Image already exists
0998bf8fb9e9: Image already exists
0a85502c06c9: Image already exists
latest: digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495a
c34c4c16eacd57 size: 8008
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This
passphrase
will be used to protect the most sensitive key in your signing system.
Please
choose a long, complex passphrase and be careful to keep the password and
the
key file itself secure and backed up. It is highly recommended that you
use a
password manager to generate the passphrase and keep it safe. There will
be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id d792b7a:
Repeat passphrase for new root key with id d792b7a:
```

```
Enter passphrase for new repository key with id docker.io/
scottpgallagher/ubuntu (46a967e):
Repeat passphrase for new repository key with id docker.io/
scottpgallagher/ubuntu (46a967e):
Finished initializing "docker.io/scottpgallagher/ubuntu"
```

The most important line from the code above is:

```
latest: digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495a
c34c4c16eacd57 size: 8008
```

This gives you the SHA hash that is used to verify the image is what it says it is and not created by someone else, as well as its size. This will be used later when someone goes to run that `image/container`.

If you were to do a `docker pull` from a machine that doesn't have this image, you can see it has been signed with that hash.

```
$ docker pull scottpgallagher/ubuntu


Using default tag: latest
latest: Pulling from scottpgallagher/ubuntu
Digest: sha256:98002698c8d868b03708880ad2e1d36034c79a6698044b495ac34c4c16
eacd57
Status: Downloaded newer image for scottpgallagher/ubuntu:latest
```

Again, we see the SHA value being presented when we do the `pull` command.

So, what this means is when you go to run this container, it won't run locally without first comparing the local hash to that on the registry server to ensure it hasn't changed. If they match, it will run, if they don't match, it won't run and will give you an error message about the hashes not matching.

With the Docker Hub you aren't essentially signing images with your own key, unless you manipulate the keys that are in your `~/.docker/trust/trusted-certificates/` directory. Remember that, by default, when you install Docker you are given a set of certificates that you can use.

Hardware signing

Now that we have looked at being able to sign images, which other security measure have been put in place to help make that process even more secure? Enter YubiKeys! YubiKeys is a form of two factor authentication that you can utilize. The way YubiKey works is that it has the root key on it, built into the hardware. You enable Docker Content Trust, then push your image. Upon using your image, Docker notes that you have enabled Content Trust and asks you to touch the YubiKey, yes, physically touch it. This is to ensure that you are a person and not a robot or just a script. You then need to provide a passphase to use and then, once again, touch the YubiKey. Once you have done this, you won't need the YubiKey anymore, but you will need that passphrase that you assigned earlier.

My description of this really doesn't do it justice. At DockerCon Europe 2015 (`http://europe-2015.dockercon.com`), there was a great play-by-play of this in operation between two Docker employees, Aanand Prasad and Diogo Mónica.

To view the video, please visit the following URL:

`https://youtu.be/fLfFFtOHRZQ?t=1h21m33s`

Docker Subscription

Docker Subscription is a service for your distributed applications that will help you support those applications as well as deploy them. The Docker Subscription package includes two critical software pieces and a support piece:

- **Docker Registry** — where you store and manage your images (locally hosted or hosted in the cloud)
- **The Docker Engine** — to run those images
- **Docker Universal Control Plane** (**UCP**)
- **Commercial support** — pick up the phone or shoot off an email for some assistance

If you are a developer, sometimes the operations side of things are either a little difficult to get set up and manage or they require some training to get going. With Docker Subscription you can off load some of those worries by utilizing the expertise that is out there with commercial level support. With this support you will get responsive turn around on your issues. You will receive any hot fixes that are available or have been made available to patch your solution. Assistance with future upgrades is also part of the added benefit of choosing the Docker Subscription plan. You will get assistance with upgrading your environments to the latest and most secure Docker environments.

Pricing is broken down based on where you want to run your environment whether it is on a server of your choosing or if it's in a cloud environment. It is also based upon how many Docker Trusted Registries and/or how many commercially supported Docker Engines you wish to have as well. All of these solutions provide you with integration into your existing **LDAP** or **Active Directory** environments. With this added benefit, you can use items such as group policies to manage access to such resources. The last thing that you will have to decide is how quick a response time you want from the support end. All of those will result in the price you pay for the subscription service. No matter what you do pay though the money spent will be well worth it, not only in respect of the peace of mind you will get but the knowledge you will gain is priceless.

You can also change your plans on a monthly or annual basis as well as upgrade, in increments of ten, your Docker Engine instances. You can also upgrade in packs of ten the number of **Docker Hub Enterprise** instances. Switching between an on premises server to the cloud, or vice-versa, is also possible.

To break some things down so as to not be confused, the Docker Engine is the core of the Docker ecosystem. It is the command line tools that you use to run, build, and manage your containers or images. The Docker Hub Enterprise is the location where you store and manage your images. These images can be public or made private. We will learn more about DTR in the next section of this chapter.

For more information about Docker Subscription, please visit the link below. You can sign up for a free 30 day trial, view subscription plans, and contact sales for additional assistance or questions. The subscription plans are flexible enough to conform to your operating environment whether it is something you want support for 24/7 or maybe just half of that:

`https://www.docker.com/docker-subscription`

You can also view the breakdown for commercial support here:

`https://www.docker.com/support`

Bringing this all back to the main topic of the book, Securing Docker, this is by far the most secure you can get with your Docker environment that you will be using to manage your images and containers, as well as managing the location they are all stored and run from. A little extra help never hurts and with this option, a little help will defiantly go a long way.

The latest part to be added is the Docker Universal Control Plane. The Docker UCP provides a solution for management of applications and infrastructure that is Dockerized regardless of where they might be running. This could be running on premises or in the cloud. You can find out more information about Docker UCP at the following link:

`https://www.docker.com/products/docker-universal-control-plane`

You can also get a demo of the product using the above URL. Docker UCP is scalable, easy to set up, and you can manage users and access control through integrations into your existing LDAP or Active Directory environments.

Docker Trusted Registry

The DTR is a solution that provides a secure location where you can store and manage your Docker images either on premises or in the cloud. It also provides some monitoring to let you get insight into usage so you can tell what kind of load is being passed to it. DTR, unlike Docker Registry, is not free and does come with a pricing model. As we saw earlier with Docker Subscription, the pricing plan for DTR is the same. Don't fret as we will go over Docker Registry in the next section of the book so you can understand it as well and have all the options available to you for image storage.

The reason we separate it out into its own section is that there are a lot of moving pieces involved and it's critical to understand how they all function not only as a whole to the Docker Subscription piece, but as it stands by itself, the DTR piece where all your images are being maintained and stored.

Installation

There are two ways you can install DTR, or rather there are two locations where you can install DTR. The first is that you can deploy it in house on a server you manage. The other is deploying it to a cloud provider environment like that of **Digital Ocean**, **Amazon Web Services** (**AWS**), or **Microsoft Azure**.

The first part you will need is a license for the DTR. Currently, they do offer a trial license that you can use, which I highly recommend you do. This will allow you to evaluate the software on your selected environment without having to fully commit to that environment. If there is something that you find doesn't work in a particular environment or you feel another location may suit you better, you can then switch without having to be tied to a particular location or having to move your existing environment around to a different provider or location. If you do choose to use AWS, they do have a pre-baked **Amazon Machine Image** (**AMI**) that you can utilize to get your Trusted Registry set up much quicker. This avoids having to do it all manually by hand.

Before you can install the Trusted Registry, you first need to have Docker Engine installed. If you don't already have it installed, please see the documentation located with the link below for more information on doing so.

`https://docs.docker.com/docker-trusted-registry/install/install-csengine/`

You will notice there is a difference in installing the normal Docker Engine from the **Docker CS Engine**. The Docker CS Engine stands for commercially supported Docker Engine. Be sure to check the documentation as the list of recommended or supported Linux versions are shorter than the regular list for Docker Engine.

If you are installing using the AMI, then please follow the instructions here:

`https://docs.docker.com/docker-trusted-registry/install/dtr-ami-byol-launch/`

If you are installing on Microsoft Azure, then please follow the instructions here:

`https://docs.docker.com/docker-trusted-registry/install/dtr-vhd-azure/`

Once you do have Docker Engine installed, it's time to install the DTR piece. If you are reading to this point we will be assuming that you aren't installing to AWS or Microsoft Azure. If you are using either of those two methods, please see the links from above. The installation is very straightforward:

```
$ sudo bash -c '$(sudo docker run docker/trusted-registry install)'
```

> Note: You may have to remove the `sudo` options from the above command when running on Mac OS.

Once this has been run, you can navigate in your browser to the IP address of your Docker host. You will then be setting the domain name for your Trusted Registry as well applying the license. The web portal will guide you through the rest of the setup process.

In accessing the portal you can set up authentication through your existing LDAP or Active Directory environments as well, but this can be done at anytime.

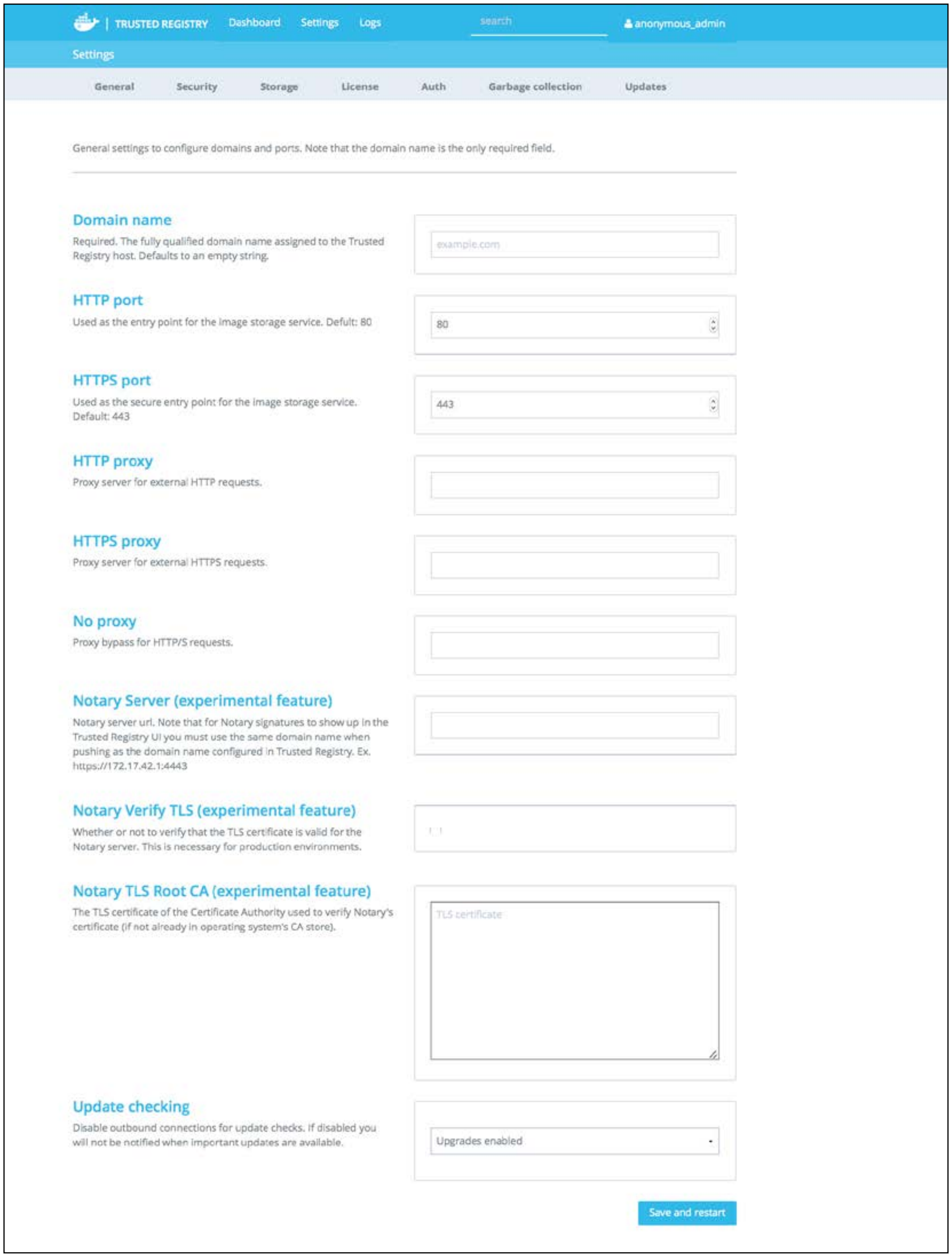
Once that is done, it is time for *Securing Docker Trusted Registry*, which we will cover in the next section.

Securing Docker Trusted Registry

Now that we have our Trusted Registry set up, we need to make it secure. Before making it secure you will need to create an administrator account to be able to perform actions. Once you have your Trusted Registry up and running, and are logged into it, you will be able to see six areas under **Settings**. These are:

- **General** settings
- **Security** settings
- **Storage** settings
- **License**
- **Auth** settings
- **Updates**

The **General** settings are mainly focused around settings such as **HTTP port** or **HTTPS port**, the **Domain name** to be used for your Trusted Registry, and proxy settings, if applicable.



General settings to configure domains and ports. Note that the domain name is the only required field.

Domain name
Required. The fully qualified domain name assigned to the Trusted Registry host. Defaults to an empty string.
example.com

HTTP port
Used as the entry point for the image storage service. Defult: 80
80

HTTPS port
Used as the secure entry point for the image storage service. Default: 443
443

HTTP proxy
Proxy server for external HTTP requests.

HTTPS proxy
Proxy server for external HTTPS requests.

No proxy
Proxy bypass for HTTP/S requests.

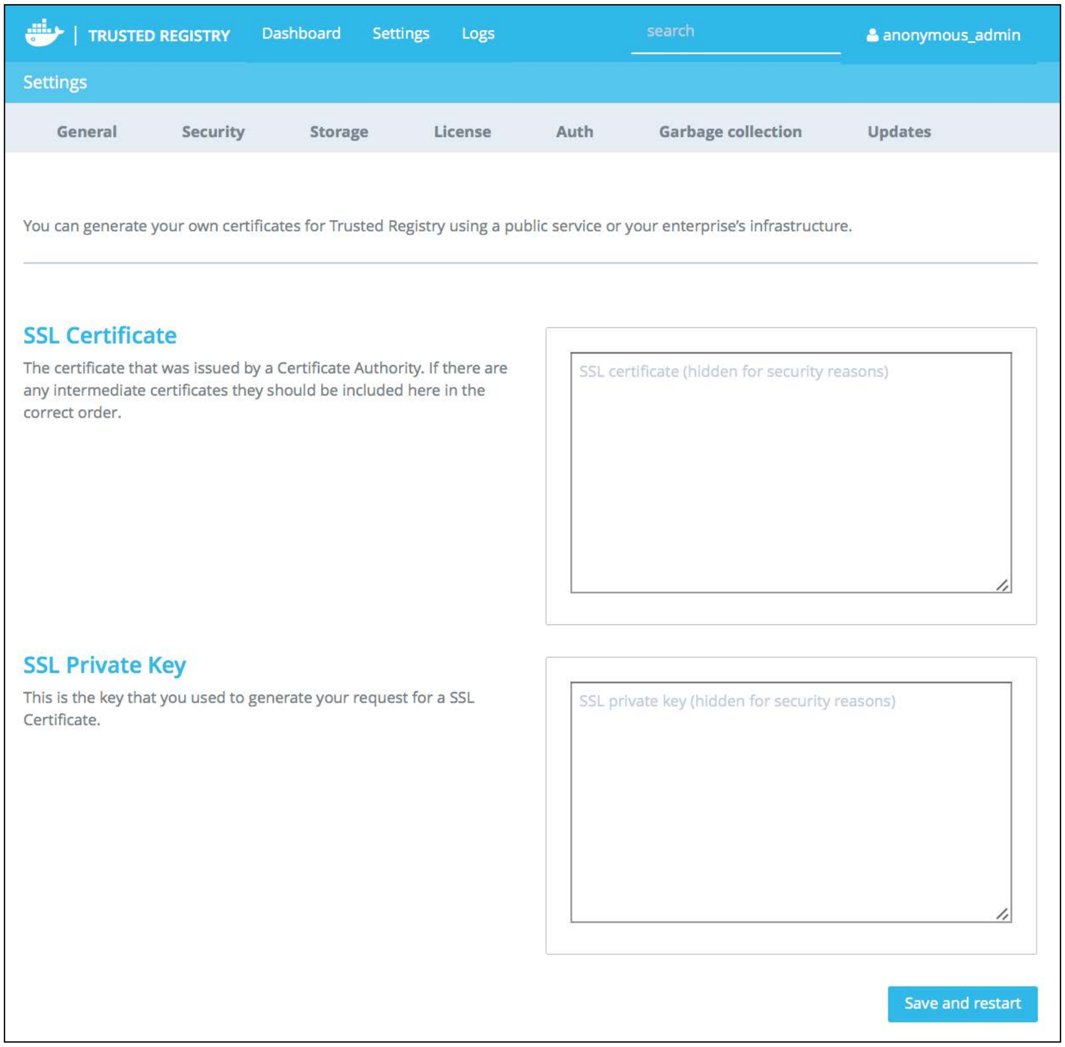
Notary Server (experimental feature)
Notary server url. Note that for Notary signatures to show up in the Trusted Registry UI you must use the same domain name when pushing as the domain name configured in Trusted Registry. Ex. https://172.17.42.1:4443

Notary Verify TLS (experimental feature)
Whether or not to verify that the TLS certificate is valid for the Notary server. This is necessary for production environments.

Notary TLS Root CA (experimental feature)
The TLS certificate of the Certificate Authority used to verify Notary's certificate (if not already in operating system's CA store).
TLS certificate

Update checking
Disable outbound connections for update checks. If disabled you will not be notified when important updates are available.
Upgrades enabled

The next section, **Security** settings, is probably one of the most important ones. Within this **Dashboard** pane you are able to utilize your **SSL Certificate** and **SSL Private Key**. These are what make the communication between your Docker clients and the Trusted Registry secure. Now, there are a few options for those certificates. You can use the self signed ones that are created when installing the Trusted Registry. You can also do self signed ones of your own, using a command line tool such as **OpenSSL**. If you are in an enterprise organization, they more than likely have a location where you can request certificates such as the one that can be used with the registry. You will need to make sure that the certificates on your Trusted Registry are the same ones being used on your clients to ensure secure communications when doing `docker pull` or `docker push` commands:



You can generate your own certificates for Trusted Registry using a public service or your enterprise's infrastructure.

SSL Certificate

The certificate that was issued by a Certificate Authority. If there are any intermediate certificates they should be included here in the correct order.

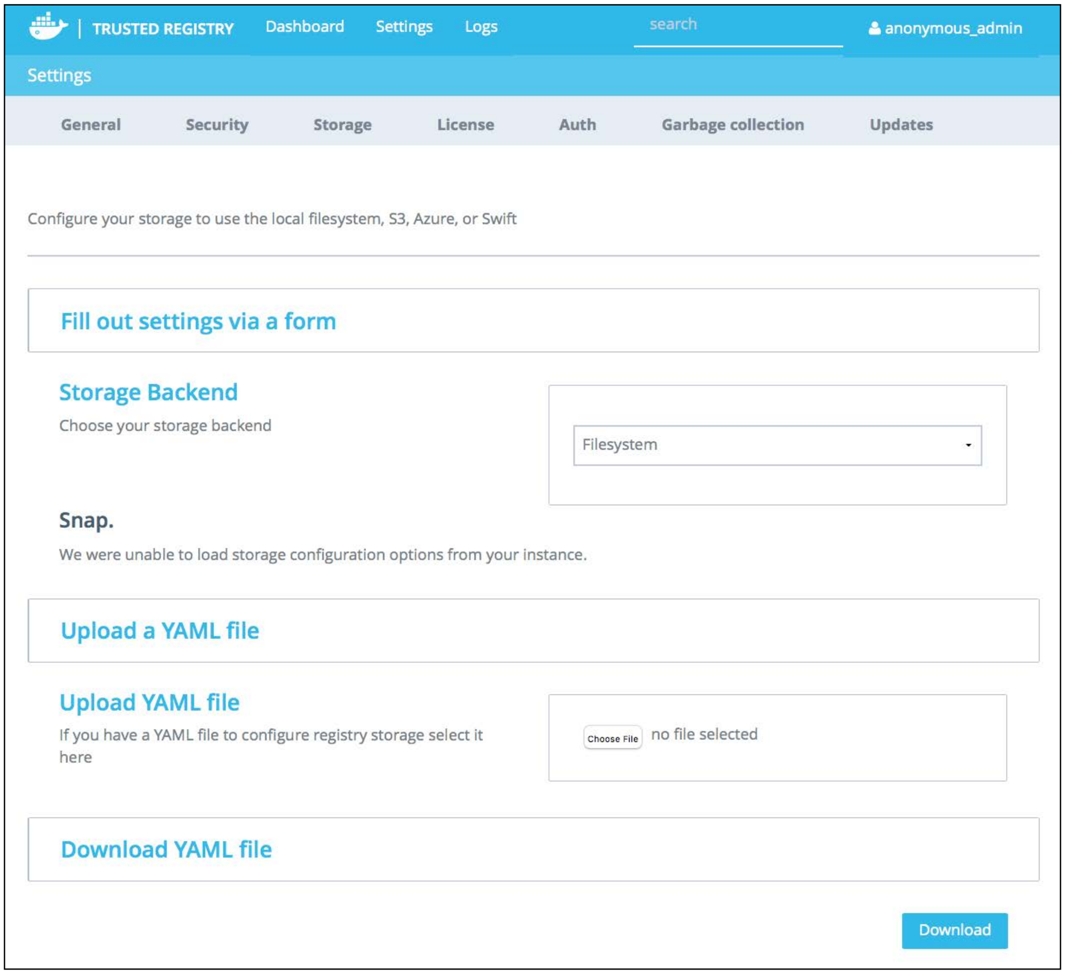
SSL certificate (hidden for security reasons)

SSL Private Key

This is the key that you used to generate your request for a SSL Certificate.

SSL private key (hidden for security reasons)

The next section deals with image storage settings. Within this **Dashboard** pane, you can set where your images are stored on the backend storage. Options for this might include an NFS share you are using, local disk storage of the Trusted Registry server, an S3 bucket from AWS, or another cloud storage solution. Once you have selected your **Storage Backend** option, you can then set the path from within that **Storage** to store your images:



Configure your storage to use the local filesystem, S3, Azure, or Swift

Fill out settings via a form

Storage Backend

Choose your storage backend

Filesystem

Snap.

We were unable to load storage configuration options from your instance.

Upload a YAML file

Upload YAML file

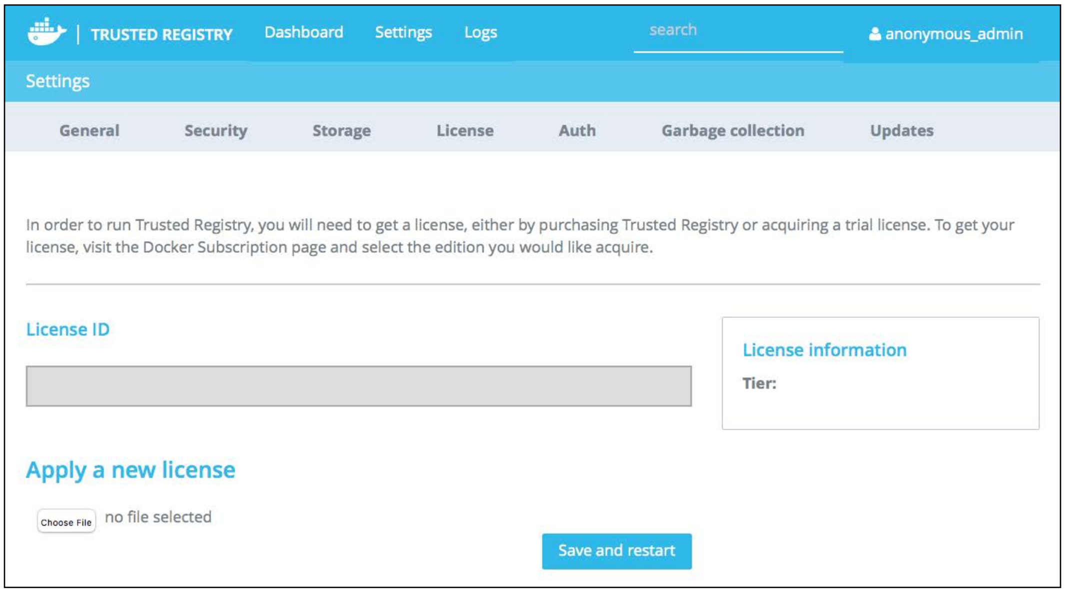
If you have a YAML file to configure registry storage select it here

Choose File no file selected

Download YAML file

Download

The **License** section is very straightforward as this is where you update your license when it's time to renew a new one or when you upgrade a license that might include more options:



In order to run Trusted Registry, you will need to get a license, either by purchasing Trusted Registry or acquiring a trial license. To get your license, visit the Docker Subscription page and select the edition you would like acquire.

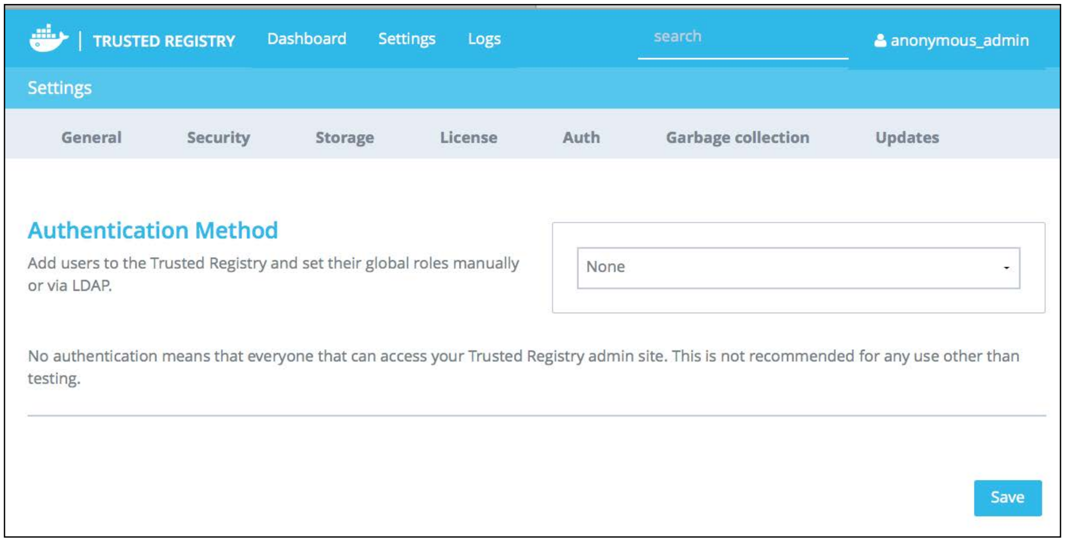
License ID

License information

Tier:

Apply a new license

Authentication settings allow you to tie the login to the Trusted Registry into your existing authentication environment. Your options here are: **None** or a **Managed** option. **None** is not recommended except for testing purposes. The **Managed** option is where you would set up usernames and passwords and manage them from there. The other option would be to use an **LDAP** service, one that you might already be running as well, so that users have the same login credentials for their other work appliances such as email or web logins.



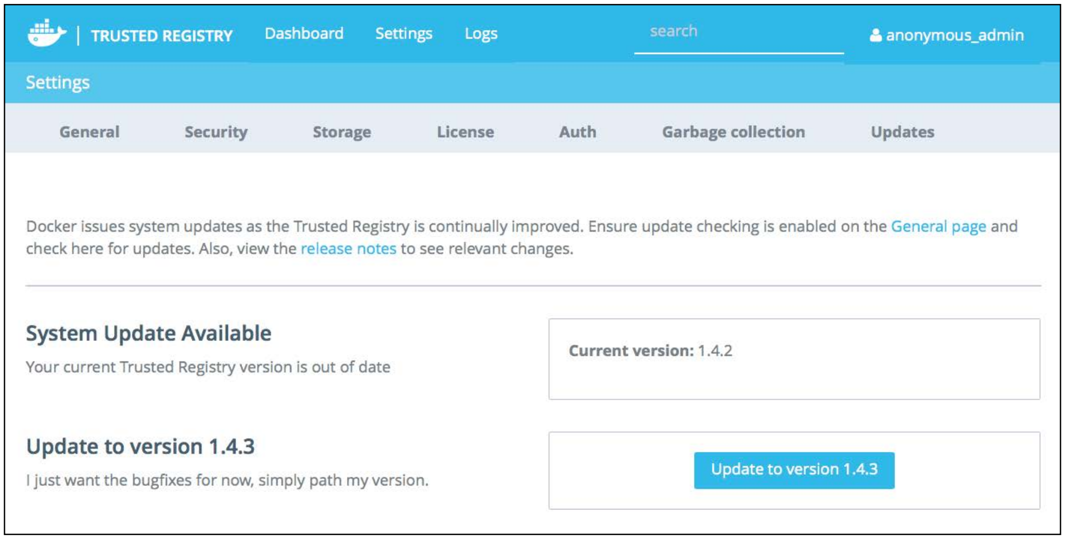
Authentication Method

Add users to the Trusted Registry and set their global roles manually or via LDAP.

None

No authentication means that everyone that can access your Trusted Registry admin site. This is not recommended for any use other than testing.

The last section, **Updates**, deals with how you manage updates for the DTR. These settings would be totally up to you how you handle updates, but be sure if you are doing an automated form of updates that you are also utilizing backups for restoring purposes in the event that something goes wrong during the update process.



Docker issues system updates as the Trusted Registry is continually improved. Ensure update checking is enabled on the General page and check here for updates. Also, view the release notes to see relevant changes.

System Update Available

Your current Trusted Registry version is out of date

Current version: 1.4.2

Update to version 1.4.3

I just want the bugfixes for now, simply path my version.

Administering

Now that we have covered the items that help you secure your Trusted Registry, we might as well take a few minutes to cover the other items that are within the console to help you administer it. Beyond the **Settings** tab within the registry, there are four other tabs that you can navigate and gather information about your registry. Those are:

- **Dashboard**
- **Repositories**
- **Organizations**
- **Logs**

The **Dashboard** is the main landing page you are taken to when you log in via your browser to the console. This will display information about your registry in one central location. The information you will be seeing is more hardware related information about the registry server itself as well as the Docker host that the registry server is running on. The **Repositories** section will allow you to control which repositories, either **Public** or **Private**, your users are able to pull images from. The **Organizations** section allows you to control access, that is, who on the system can push, pull, or do other Docker related commands against the repositories that you have elected to configure. The last section, the **Logs** section, will allow you to view logs based upon your containers that are being used from your registry. The logs are rotated every two weeks with a maximum size of *64 mb*. You are able to filter through the logs as well based on a container as well as being able to search for a date and/or time.

Workflow

In this section, let's pull an image, manipulate it, and then place it on our DTR for access by others within our organization.

First, we need to pull an image from the **Docker Hub**. Now, you could start from scratch with a **Dockerfile** and then do a Docker build and then push, but let's, for this demonstration, say we have the `mysql` image and we want to customize it in some way.

```
$ docker pull mysql


Using default tag: latest
```

```
latest: Pulling from library/mysql

1565e86129b8: Pull complete
a604b236bcde: Pull complete
2a1fefc8d587: Pull complete
f9519f46a2bf: Pull complete
b03fa53728a0: Pull complete
ac2f3cdeb1c6: Pull complete
b61ef27b0115: Pull complete
9ff29f750be3: Pull complete
ece4ebeae179: Pull complete
95255626f143: Pull complete
0c7947afc43f: Pull complete
b3a598670425: Pull complete
e287fa347325: Pull complete
40f595e5339f: Pull complete
0ab12a4dd3c8: Pull complete
89fa423a616b: Pull complete
Digest: sha256:72e383e001789562e943bee14728e3a93f2c3823182d14e3e01b3
fd877976265
Status: Downloaded newer image for mysql:latest

$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE

mysql               latest              89fa423a616b        20 hours ago
359.9 MB
```

Now, let's assume we made a customization to the image. Let's say that we set up the container to ship its logs off to a log stash server that we are using to collect our logs from all our containers that we are running. We now need to save those changes.

```
$ docker commit be4ea9a7734e <dns.name>/mysql
```

When we go to do the commit, we need a few tidbits of information. The first is the container ID, which we can get from running a `docker ps` command. We also need the DNS name of our registry server that we set up earlier, and lastly a unique image name to give it. In our case, we will keep it as `mysql`.

We are now ready to push the updated image to our registry server. The only information we need is the image name that we want to push, which will be the `<dns.name>/mysql`.

```
$ docker push <dns.name>/mysql
```

The image is now ready to be used by the other users in our organization. Since the image is in our Trusted Registry, we can control access to that image from our clients. This could mean that our clients would need our certificate and keys to be able to push and pull this image, as well as permissions set up within the organization settings we previously went over in the last section.

```
$ docker pull <dns.name>/mysql
```

We can then make run the image, make changes if needed, and push the newly created image back to the Trusted Registry server as necessary.

Docker Registry

The Docker Registry is an open source option if you want to totally go at it on your own. If you totally want hands off, you can always use the Docker Hub and rely on public and private repositories, which will run you a fee on the Docker Hub though. This can be hosted locally on a server of your choosing or on a cloud service.

Installation

The installation of the Docker Registry is extremely simply as it runs in a Docker container. This allows you to run it virtually anywhere, on a virtual machine in your own server environment or in a cloud environment. The typical port that is used is port `5000`, but you can change it to suit your needs:

```
$ docker run -d -p 5000:5000 --restart=always  --name registry
registry:2.2
```

One of the other items you will notice from above is that we are specifying a version to use instead of leaving it blank and pulling the latest version. That is because as of writing this book, the latest version for that registry tag is still at version 0.9.1. Now, while this might be suitable for some, version 2 is stable enough to be considered and to run your production environment on. We are also introducing the `--restart=always` flag for that as in the event of something happening to the container, it will restart and be available to serve out or accept images.

Once you have run the command above, you will have a running container registry on the IP address of the Docker host you ran the command on along with the port selection that you used in your `docker run` command above.

Now it is time to put some images up on your new registry. The first thing we need is an image to push to the registry and we can do that in two ways. We can build images based on Docker files that we have created or we can pull down an image from another registry, in our case we will be using the Docker Hub, and then push that image to our new registry server. First, we need an image to choose and again, we will default back to the `mysql` image since it's a more popular image that most people might be using in their environments at some point along the way.

```
$ docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql

1565e86129b8: Pull complete
a604b236bcde: Pull complete
2a1fefc8d587: Pull complete
f9519f46a2bf: Pull complete
b03fa53728a0: Pull complete
ac2f3cdeb1c6: Pull complete
b61ef27b0115: Pull complete
9ff29f750be3: Pull complete
ece4ebeae179: Pull complete
95255626f143: Pull complete
0c7947afc43f: Pull complete
b3a598670425: Pull complete
e287fa347325: Pull complete
40f595e5339f: Pull complete
0ab12a4dd3c8: Pull complete
89fa423a616b: Pull complete
```

```
Digest: sha256:72e383e001789562e943bee14728e3a93f2c3823182d14e3e01b3
fd877976265
Status: Downloaded newer image for mysql:latest
```

Next, you need to tag the image so it will now be pointing to your new registry so you can do push it to the new location:

```
$ docker tag mysql <IP_address>:5000/mysql
```

Let's break down that command above. What we are doing is applying the tag of `<IP_address>:5000/mysql` to the `mysql` image that we pulled from the Docker Hub. Now that `<IP_address>` piece will be replaced by the IP address from the Docker host that is running the registry container. This could also be a DNS name as well, as long as the DNS points to the correct IP that is running on the Docker host. We also need to specify the port number for our registry server, and in our case we left it with port `5000`, so we include: `5000` in the tag. Then, we are going to give it the same same of `mysql` at the end of the command. We are now ready to push this image to our new registry.

```
$ docker push <IP_address>:5000/mysql
```

After it has been pushed, you can now pull it down from another machine that is configured with Docker and has access to the registry server.

```
$ docker pull <IP_address>:5000/mysql
```

What we have looked at here are the defaults and while it could work if you want to use firewalls and such to secure the environment or even internal IP address, you still might want to take security to the next level and that is what we will look at in the next section. How can we make this even more secure?

Configuration and security

It's time to tighten up our running registry with some additional features. The first method would be to run your registry using TLS. Using TLS allows you to apply certificates to the system so that people who are pulling from it know that it is who you say it is by knowing that someone hasn't comprised the server or is doing a man in the middle attack by supplying you with compromised images.

To do that, we will need to rework the Docker `run` command we were running in the last section. This is going to assume you have gone through some of the process of obtaining a certificate and key from your enterprise environment or you have self signed one using another piece of software.

Our new command will look like this:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
  -v <certificate folder>/<path_on_container> \
  registry:2.2.0
```

You will need to be in the directory where the certificates are or specify the full path to them in the above command. Again, we are keeping the standard port of `5000`, along with the name of registry. You could alter that too to something that might suit you better. For the sake of this book we will keep it close to that in the official documentation in the event that you look there for more reference. Next, we add two additional lines to the `run` command:

```
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
```

This will allow you to specify the certificate and key file that you will be using. These two files will need to be in the same directory that you are running the run command from as the environmental variables will be looking for them upon run. Now you could also add a volume switch to the run command to make it a little cleaner if you like and put the certificate and key in that folder and run the registry server that way.

The other way you can help with security is by putting a username and password on the registry server. This will help when users want to push or pull an item as they will need the username and password information. The catch with this is that you have to be using TLS in conjunction with this method. This method of username and password is not a standalone option.

First, you need to create a password file that you will be using in your `run` command:

```
$ docker run --entrypoint htpasswd registry:2.2.0 -bn <username>
<password> > htpasswd
```

Now, it can be a little confusing to understand what is happening here, so let's clear that up before we jump to the `run` command. First, we are issuing a `run` command. This command is going to run the `registry:2.2.0` container and its entry point specified means to run the `htpasswd` command along with the `-bn` switches, which will inject the `username` and `password` in an encrypted fashion into a file called `htpasswd` that you will be using for authentication purposes on the registry server. The `-b` means to run in batch mode while the `-n` means to display the results, and the `>` means to put those items into a file instead of to the actual output screen.

Now, onto our newly enhanced and totally secure Docker `run` command for our registry:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Name" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=htpasswd \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
  registry:2.20
```

Again, it's a lot to digest but let's walk through it. We have seen some of these lines before in:

```
  -e REGISTRY_HTTP_TLS_CERTIFICATE=server.crt \
  -e REGISTRY_HTTP_TLS_KEY=server.key \
```

The new ones are:

```
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Name" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=htpasswd \
```

The first one tells the registry server to use `htpasswd` as its authentication method to verify clients. The second gives your registry a name and can be changed at your own discretion. The last one tells the registry server the location of the file that is to be used for the `htpasswd` authentication. Again, you will need to use volumes and put the `htpasswd` file in its own volume inside the container so it allows for easier updating down the road. You also need to remember the `htpasswd` file needs to be placed in the same directory as the certificate and key file when you execute the Docker `run` command.

Summary

In this chapter, we have looked at being able to sign your images using the components of Docker Content Trust as well as hardware signing using Docker Content Trust along with the third party utilities in the form of YubiKeys. We also took a look at Docker Subscription that you can utilize to your advantage to help set up not only secure Docker environments but also ones that are supported by those at Docker itself. We then looked at DTR as a solution that you can use to store your Docker images. Lastly, we looked at the Docker Registry, which is a self hosted registry that you can use to store and manage your images. This chapter should help give you enough configuration items to chew on to help you make the right decision as to where to store your images.

In the next chapter we will be looking at securing/hardening Linux kernels. As the kernel is what is used to run all your containers, it is important that it is secured in the proper way to help alleviate any security related issues. We will be covering some hardening guides that you can use to accomplish this goal.

3
Securing and Hardening Linux Kernels

In this chapter, we will turn our attention to securing and hardening the one key piece that every container running on your Docker host relies on: the Linux kernel. We will focus on two topics: guides that you can follow to harden the Linux kernel and tools that you can add to your arsenal to help harden the Linux kernel. Let's take a brief look at what we will be covering in this chapter before diving in:

- Linux kernel hardening guides
- Linux kernel hardening tools
  - **Grsecurity**
  - **Lynis**

Linux kernel hardening guides

In this section, we will be looking at the SANS Institute hardening guide for the Linux kernel. While a lot of this information is outdated, I believe that it is important for you to understand how the Linux kernel has evolved and become a secure entity. If you were to step into a time machine and go back to the year 2003 and attempt to do the things you want to do today, this is everything you would have to do.

First, some background information about the SANS Institute. It is a private US-based company that specializes in cybersecurity and information technology-related training and education. These trainings prepare professionals to defend their environments against attackers. SANS also offers a variety of free security-related content via their SANS Technology Institute Leadership Lab. More information about this can be found at `http://www.sans.edu/research/leadership-laboratory`.

To help alleviate against this widespread attack base, there needs to be security focus on every aspect of your IT infrastructure and software. Based upon this, the first place to start would be at the Linux kernel.

SANS hardening guide deep dive

As we have already covered the background of the SANS Institute, let's go ahead and jump into the guide that we will be following to secure our Linux kernel(s).

For reference, we will be using the following URL and highlighting the sticking points that you should be focusing on and implementing in your environments to secure the Linux kernel:

`https://www.sans.org/reading-room/whitepapers/linux/linux-kernel-hardening-1294`

The Linux kernel is an always-developing and maturing piece of the Linux ecosystem and for this reason, it's important to get a firm grasp on the Linux kernel as it stands currently, which will help when looking to lockdown the new feature sets that might come in future releases.

The Linux kernel allows loading modules without having to recompile or reboot, which is great when you are looking to eliminate downtime. Some various operating systems require reboots when trying to apply updates to a certain operating system/ application criteria. This can also be a bad thing with regards to the Linux kernel as the attackers can inject harmful material into the kernel and wouldn't need to reboot the machine, which might be caught by someone noticing the reboot of the system. For this reason, it is suggested that a statically compiled kernel with the load option be disabled to help prevent against attack vectors.

Buffer overflows are another way attackers can compromise a kernel and gain entry. Applications have a limit, or buffer, on how much a user can store in memory. An attacker overflows this buffer with specially crafted code, which could let the attacker gain control of the system that, in turn, will empower them to do whatever they want at that point. They could add backdoors to the system, send logs off to a nefarious place, add additional users to the system, or even lock you out of the system. To prevent these type of attacks, there are three areas of focus that the guide hones in on.

The first is the **Openwall** Linux kernel patch that was a patch created to address this issue. This patch also includes some other security enhancements that might be attributed to your running environments. Some of these items included restricted links and file reads/writes in the `/tmp` folder location and restricted access to the `/proc` locations on the filesystem. It also includes enhanced enforcement for a number of user processes that you could control as well as the ability to destroy shared memory segments, which were not in use, and lastly, some other enhancements for those of you that are running kernel versions older than version 2.4.

If you are running an older version of the Linux kernel, you will want to check out the Openwall hardened Linux at `http://www.openwall.com/Owl/` and Openwall Linux at `http://www.openwall.com/linux/`.

The next piece of software is called **Exec Shield** and it takes a similar approach to the Openwall Linux kernel patch, which implements a non-executable stack, but Exec Shield extends this by attempting to protect any and all segments of virtual memory. This patch is limited to the prevention of attacks against the Linux kernel address space. These address spaces include stack, buffer, or function pointer overflow spaces.

More information about this patch can be found at `https://en.wikipedia.org/wiki/Exec_Shield`.

The last one is **PaX**, which is a team that creates a patch for the Linux kernel to prevent against a variety of software vulnerabilities. As this is something we will be talking about in-depth in the next section, we will just discuss some of its features. This patch focuses on the following three address spaces:

- **PAGEEXEC**: These are paging-based, non-executable pages
- **SEGMEXEC**: These are segmentation–based, non-executable pages
- **MPROTECT**: These are `mmap()` and `mprotect()` restrictions

To learn more about PaX, visit `https://pax.grsecurity.net`.

Now that you have seen how much efforts you had to put in, you should be glad that security is now at the forefront for everyone, especially, the Linux kernel. In some of the later chapters, we will be looking at some of the following new technologies that are being used to help secure environments:

- Namespaces
- cgroups
- sVirt
- Summon

There are also a lot of capabilities that can be accomplished through the `--cap-ad and --cap-drop` switches on your `docker run` command.

Even like the days before, you still need to be aware of the fact that the kernel is shared throughout all your containers on a host, therefore, you need to protect this kernel and watch out for vulnerabilities when necessary. The following link allows you to view **Common Vulnerabilities and Exposures** (**CVE**) in the Linux kernel:

`https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html`

Access controls

There are various levels of access controls that you can layer on top of Linux as well as recommendations that you should follow with reference to certain users, and these would be the superusers on your system. Just to give some definition to superusers, they are the accounts on the system that have unfettered access to do anything and everything. You should include the root user when you are layering on these access controls.

These access control recommendations will be the following:

- Restricting usage of the root user
- Restricting its ability to SSH

  By default, on some systems, root has the ability to SSH to machine if SSH is enabled, which we can see from a portion of the `/etc/ssh/sshd_config` file on some Linux systems, as follows:

  ```
  # Authentication:

  #LoginGraceTime 2m
  #PermitRootLogin no
  #StrictModes yes
  #MaxAuthTries 6
  #MaxSessions 10
  ```

  From what you can see here, the section for `PermitRootLogin no` is commented out with the `#` symbol so that means this line won't be interpreted. To change this, simply remove the `#` symbol and save the file and restart the service. The section of this file should now be similar to the following code:

  ```
  # Authentication:

  #LoginGraceTime 2m
  ```

```
PermitRootLogin no
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
```

Now, you may want to restart the SSH service for these changes to take affect, as follows:

```
$ sudo service sshd restart
```

- Restrict its ability to log in beyond the console. On most Linux systems, there is a file in `/etc/default/login` and in that file, there is a line that is similar to the following:

  ```
  #CONSOLE=/dev/console
  ```

  Similar to the preceding example, we need to uncomment this line by removing `#` for this to take affect. This will only allow the root user to log in at `console` and not via SSH or other methods.

- Restrict `su` command

  The `su` commands allow you to login as the root user and be able to issue root-level commands, which gives you full access to the entire system. To restrict access to who can use this command, there is a file located at `/etc/pam.d/su`, and in this file, you will see a line similar to the following:

  ```
  auth required /lib/security/pam_wheel.so use_uid
  ```

  You can also choose the following line of code here, depending upon your Linux flavor:

  ```
  auth required pam_wheel.so use_uid
  ```

  The check for wheel membership will be done against the current user ID for the ability to use the `su` command.

- Requiring `sudo` to run commands
- Some other access controls that are remanded are the use of the following controls:
  - **Mandatory Access Controls** (**MAC**): Restricting what users can do on systems
  - **Role-Based Access Controls**: Using groups to assign the roles that these groups can perform
  - **Rule Set Based Access Controls** (**RSBAC**): Rule sets that are grouped in the request type and performs actions based on set rule(s)

    - **Domain and Type Enforcement** (**DTE**): Allow or restrict certain domains from performing set actions or preventing domains from interacting with each other

You can also utilize the following:

- SELinux (RPM-based systems (such as Red Hat, CentOS, and Fedora)
- AppArmor (apt-get-based systems (such as Ubuntu and Debian)

These RSBAC, as we discussed earlier, allow you to choose methods of control that are appropriate for what your system is running. You can also create your own access control modules that can help enforce. By default, on most Linux systems, these type of environments are enabled or in enforcing mode. Majority of people will turn these off when they create a new system, but it comes with security drawbacks, therefore, it's important to learn how these systems work and use them in the enabled or enforcement mode to help mitigate further risks.

More information about each can be found at the following:

- **SELinux**: `https://en.wikipedia.org/wiki/Security-Enhanced_Linux`
- **AppArmor**: `https://en.wikipedia.org/wiki/AppArmor`

Distribution focused

There are many Linux distributions, or flavors as they call them, in the Linux community that have been *pre-baked* to be already hardened. We referenced one earlier, the **Owlwall** flavor of Linux, but there are others out there as well. Out of the other two, one that is no longer around is **Adamantix** and the other is **Gentoo Linux**. These Linux flavors feature some baked-in Linux kernel hardening as standards of their operating system builds.

Linux kernel hardening tools

There are some Linux kernel hardening tools out there, but we will focus on only two of them in this section. The first being Grsecurity and the second being Lynis. These are tools that you can add to your arsenal to help in increasing the security of the environments that you will be running your Docker containers on.

Grsecurity

So, what exactly is Grsecurity? According to their website, Grsecurity is an extensive security enhancement for the Linux kernel. This enhancement contains a wide range of items that help in defending against various threats. These threats might include the following components:

- **Zero day exploits**: This mitigates and keeps your environment protected until a long-term solution can be made available through the vendor.
- **Shared host or container weaknesses**: This protects you against kernel compromises that various technologies, and very much so containers, use for each container on the host.
- **It goes beyond basic access controls**: Grsecurity works with the PaX team to introduce complexity and unpredictability to the attacker, while responding and denying the attacker any more chances.
- **Integrates with you existing Linux distribution:** As Grsecurity is kernel-based, it can be used with any Linux flavors such as Red Hat, Ubuntu, Debian, and Gentoo. Whatever your Linux flavor is, it doesn't matter, as the focus is on the underlying Linux kernel.

More information can be found at `https://grsecurity.net/`.

To directly get to the good stuff and see the feature set that is offered by utilizing a tool like Grsecurity, you will want to go to the following link:

`http://grsecurity.net/features.php`

On this page, items will be grouped into the following five categories:

- Memory Corruption Defenses
- Filesystem Hardening
- Miscellaneous Protections
- RBAC
- GCC Plugins

Lynis

Lynis is an open source tool that is used to audit your systems for security. It runs directly on the host so that it has access to the Linux kernel itself, as well as various other items. Lynis runs on almost every Unix operating system including the following:

- AIS
- FreeBSD
- Mac OS
- Linux
- Solaris

Lynis was written as a shell script, therefore, it's just as easy as copying and pasting on your system and running a simple command:

```
./lynis audit system
```

While it is running, the following actions are being taken:

- Determining the OS
- Performing a search for available tools and utilities
- Checking for any Lynis update
- Running tests from enabled plugins
- Running security tests per category
- Reporting status of security scan

More information can be found at `https://rootkit.nl/projects/lynis.html` and `https://cisofy.com/lynis/`.

Summary

In this chapter, we took a look at hardening and securing Linux kernels. We first looked at some hardening guides followed by a deep dive of an overview of the SANS Institute Hardening Guide. We also took a look at how to prevent buffer overflows in our kernels and applications through various patches. We also looked at various access controls, SELinux, and AppArmor. Lastly, we took a look at two hardening tools that we can add to our toolbox of software in the form of Grsecurity and Lynis.

In the next chapter, we will take a look at the Docker Bench application for security. This is an application that can look at the various Docker items, such as host configuration, Docker daemon configuration, daemon configuration files, container images and build files, container runtime, and lastly, Docker security operations. It will contain hands-on examples with a lot of code outputs.

4
Docker Bench for Security

In this chapter, we will be looking at the **Docker Bench for Security**. This is a tool that can be utilized to scan your Docker environments, start the host level and inspect all the aspects of this host, inspect the Docker daemon and its configuration, inspect the containers running on the Docker host, and review the Docker security operations and give you recommendations across the board of a threat or concern that you might want to look at in order to address it. In this chapter, we will be looking at the following items:

- **Docker security** – best practices
- **Docker** – best practices
- **Center for Internet Security** (**CIS**) guide
    - Host configuration
    - Docker daemon configuration
    - Docker daemon configuration files
    - Container images/runtime
    - Docker security operations
- The Docker Bench Security application
    - Running the tool
    - Understanding the output

Docker security – best practices

In this section, we will take a look at the best practices when it comes to Docker as well as the CIS guide to properly secure all the aspects of your Docker environment. You will be referring to this guide when you actually run the scan (in the next section of this chapter) and get results of what needs to or should be fixed. The guide is broken down into the following sections:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images/runtime
- Docker security operations

Docker – best practices

Before we dive into the CIS guide, let's go over some of the following best practices when using Docker:

- **One application per container**: Spread your applications to one per container. Docker was built for this and it makes everything easy at the end of the day. The isolation that we talked about earlier is where this is the key.
- **Review who has access to your Docker hosts**: Remember that whoever has the access to your Docker hosts has the access to manipulate all your images and containers on the host.
- **Use the latest version**: Always use the latest version of Docker. This will ensure that all the security holes have been patched and you have the latest features as well.
- **Use the resources**: Use the resources available if you need help. The community within Docker is huge and immensely helpful. Use their website, documentation, and the **Internet Relay Chat** (**IRC**) chat rooms to your advantage.

CIS guide

The CIS guide is a document (`https://benchmarks.cisecurity.org/tools2/docker/cis_docker_1.6_benchmark_v1.0.0.pdf`) that goes over the aspects of the Docker pieces to help you securely configure the various pieces of your Docker environment. We will cover these in the following sections.

Host configuration

This part of the guide is about the configuration of your Docker hosts. This is that part of the Docker environment where all your containers run. Thus, keeping it secure is of the utmost importance. This is the first line of defense against the attackers.

Docker daemon configuration

This part of the guide recommends securing the running Docker daemon. Everything you do to the Docker daemon configuration affects each and every container. These are the switches you can attach to the Docker daemon that we saw previously and items you will see in the following section when we run through the tool.

Docker daemon configuration files

This part of the guide deals with the files and directories that the Docker daemon uses. This ranges from permissions to ownerships. Sometimes, these areas may contain information you don't want others to know about, which could be in a plain text format.

Container images/runtime

This part of the guide contains both the information for securing the container images as well as the container runtime.

The first part contains images, cover base images, and build files that were used. You need to be sure about the images you are using not only for your base images, but also for any aspect of your Docker experience. This section of the guide covers the items you should follow while creating your own base images to ensure they are secure.

The second part, the container runtime, covers a lot of security-related items. You have to take care of the runtime variables that you are providing. In some cases, attackers can use them to their advantage, while you think you are using them to your own advantage. Exposing too much in your container can compromise the security of not only that container, but also the Docker host and other containers running on this host.

Docker security operations

This part of the guide covers the security areas that involve deployment. These items are more closely tied to the best practices and recommendations of items that are to be followed.

The Docker Bench Security application

In this section, we will cover the Docker Benchmark Security application that you can install and run. The tool will inspect the following components:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- Docker security operations

Looks familiar? It should, as these are the same items that we reviewed in the previous section, only built into an application that will do a lot of heavy lifting for you. It will show you what warnings arise with your configurations and provide information on other configuration items and even the items that have passed the test.

We will look at how to run the tool, a live example, and what the output of the process will mean.

Running the tool

Running the tool is simple. It's already been packaged for us inside a Docker container. While you can get the source code and customize the output or manipulate it in some way (say, e-mail the output), the default may be all that you need.

The code is found here: `https://github.com/docker/docker-bench-security`

To run the tool, we will simply copy and paste the following into our Docker host:

```
$ docker run -it --net host --pid host --cap-add audit_control \
-v /var/lib:/var/lib \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /usr/lib/systemd:/usr/lib/systemd \
-v /etc:/etc --label docker_bench_security \
docker/docker-bench-security
```

If you don't already have the image, it will first download the image and then start the process for you. Now that we've seen how easy it is to install and run it, let's take a look at an example on a Docker host to see what it actually does. We will then take a look at the output and take a dive into dissecting it.

There is also an option to clone the Git repository, enter the directory from the `git clone` command, and run the provided shell script. So, we have multiple options!

Let's take a look at an example and break down each section, as shown in the following command:

```
# -----------------------------------------------------------------------
-------
# Docker Bench for Security v1.0.0
#
# Docker, Inc. (c) 2015
#
# Checks for dozens of common best-practices around deploying Docker
containers in production.
# Inspired by the CIS Docker 1.6 Benchmark:
# https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.6_
Benchmark_v1.0.0.pdf
# -----------------------------------------------------------------------
-------


Initializing Sun Jan 17 19:18:56 UTC 2016
```

Running the tool – host configuration

Let's take a look at the output of the host configuration runtime:

```
[INFO] 1 - Host configuration
[WARN] 1.1  - Create a separate partition for containers
[PASS] 1.2  - Use an updated Linux Kernel
[PASS] 1.5  - Remove all non-essential services from the host - Network
[PASS] 1.6  - Keep Docker up to date
[INFO]       * Using 1.9.1 which is current as of 2015-11-09
[INFO]       * Check with your operating system vendor for support and
security maintenance for docker
```

```
[INFO] 1.7  - Only allow trusted users to control Docker daemon
[INFO]      * docker:x:100:docker
[WARN] 1.8  - Failed to inspect: auditctl command not found.
[INFO] 1.9  - Audit Docker files and directories - /var/lib/docker
[INFO]      * Directory not found
[WARN] 1.10 - Failed to inspect: auditctl command not found.
[INFO] 1.11 - Audit Docker files and directories - docker-registry.
service
[INFO]      * File not found
[INFO] 1.12 - Audit Docker files and directories - docker.service
[INFO]      * File not found
[WARN] 1.13 - Failed to inspect: auditctl command not found.
[INFO] 1.14 - Audit Docker files and directories - /etc/sysconfig/docker
[INFO]      * File not found
[INFO] 1.15 - Audit Docker files and directories - /etc/sysconfig/docker-
network
[INFO]      * File not found
[INFO] 1.16 - Audit Docker files and directories - /etc/sysconfig/docker-
registry
[INFO]      * File not found
[INFO] 1.17 - Audit Docker files and directories - /etc/sysconfig/docker-
storage
[INFO]      * File not found
[INFO] 1.18 - Audit Docker files and directories - /etc/default/docker
[INFO]      * File not found
```

Running the tool – Docker daemon configuration

Let's take a look at the output for the Docker daemon configuration runtime, as shown in the following command:

```
[INFO] 2 - Docker Daemon Configuration
[PASS] 2.1  - Do not use lxc execution driver
[WARN] 2.2  - Restrict network traffic between containers
[PASS] 2.3  - Set the logging level
[PASS] 2.4  - Allow Docker to make changes to iptables
[PASS] 2.5  - Do not use insecure registries
[INFO] 2.6  - Setup a local registry mirror
```

```
[INFO]      * No local registry currently configured
[WARN] 2.7  - Do not use the aufs storage driver
[PASS] 2.8  - Do not bind Docker to another IP/Port or a Unix socket
[INFO] 2.9  - Configure TLS authentication for Docker daemon
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.10 - Set default ulimit as appropriate
[INFO]      * Default ulimit doesn't appear to be set
```

Running the tool – Docker daemon configuration files

Let's take a look at the output for the Docker daemon configuration files runtime, as follows:

```
[INFO] 3 - Docker Daemon Configuration Files
[INFO] 3.1  - Verify that docker.service file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.2  - Verify that docker.service file permissions are set to 644
[INFO]      * File not found
[INFO] 3.3  - Verify that docker-registry.service file ownership is set
to root:root
[INFO]      * File not found
[INFO] 3.4  - Verify that docker-registry.service file permissions are
set to 644
[INFO]      * File not found
[INFO] 3.5  - Verify that docker.socket file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.6  - Verify that docker.socket file permissions are set to 644
[INFO]      * File not found
[INFO] 3.7  - Verify that Docker environment file ownership is set to
root:root
[INFO]      * File not found
[INFO] 3.8  - Verify that Docker environment file permissions are set to
644
[INFO]      * File not found
[INFO] 3.9  - Verify that docker-network environment file ownership is
set to root:root
```

```
[INFO]      * File not found
[INFO] 3.10 - Verify that docker-network environment file permissions are
set to 644
[INFO]      * File not found
[INFO] 3.11 - Verify that docker-registry environment file ownership is
set to root:root
[INFO]      * File not found
[INFO] 3.12 - Verify that docker-registry environment file permissions
are set to 644
[INFO]      * File not found
[INFO] 3.13 - Verify that docker-storage environment file ownership is
set to root:root
[INFO]      * File not found
[INFO] 3.14 - Verify that docker-storage environment file permissions are
set to 644
[INFO]      * File not found
[PASS] 3.15 - Verify that /etc/docker directory ownership is set to
root:root
[PASS] 3.16 - Verify that /etc/docker directory permissions are set to
755
[INFO] 3.17 - Verify that registry certificate file ownership is set to
root:root
[INFO]      * Directory not found
[INFO] 3.18 - Verify that registry certificate file permissions are set
to 444
[INFO]      * Directory not found
[INFO] 3.19 - Verify that TLS CA certificate file ownership is set to
root:root
[INFO]      * No TLS CA certificate found
[INFO] 3.20 - Verify that TLS CA certificate file permissions are set to
444
[INFO]      * No TLS CA certificate found
[INFO] 3.21 - Verify that Docker server certificate file ownership is set
to root:root
[INFO]      * No TLS Server certificate found
[INFO] 3.22 - Verify that Docker server certificate file permissions are
set to 444
[INFO]      * No TLS Server certificate found
[INFO] 3.23 - Verify that Docker server key file ownership is set to
root:root
```

```
[INFO]      * No TLS Key found
[INFO] 3.24 - Verify that Docker server key file permissions are set to
400
[INFO]      * No TLS Key found
[PASS] 3.25 - Verify that Docker socket file ownership is set to
root:docker
[PASS] 3.26 - Verify that Docker socket file permissions are set to 660
```

Running the tool – container images and build files

Let's take a look at the output for the container images and build files runtime, as shown in the following command:

```
[INFO] 4 - Container Images and Build Files
[INFO] 4.1  - Create a user for the container
[INFO]      * No containers running
```

Running the tool – container runtime

Let's take a look at the output for the container runtime, as follows:

```
[INFO] 5  - Container Runtime
[INFO]      * No containers running, skipping Section 5
```

Running the tool – Docker security operations

Let's take a look at the output for the Docker security operations runtime, as shown in the following command:

```
[INFO] 6  - Docker Security Operations
[INFO] 6.5 - Use a centralized and remote log collection service
[INFO]      * No containers running
[INFO] 6.6 - Avoid image sprawl
[INFO]      * There are currently: 23 images
[WARN] 6.7 - Avoid container sprawl
[WARN]      * There are currently a total of 51 containers, with only 1
of them currently running
```

Wow! A lot of output and tons to digest; but what does all this mean? Let's take a look and break down each section.

Understanding the output

There are three types of output that we will see, as follows:

- `[PASS]`: These items are solid and good to go. They don't need any attention, but they are good to read to make you feel warm inside. The more of these, the better!
- `[INFO]`: These are items that you should review and fix if you feel that they are pertinent to your setup and security needs.
- `[WARN]`: These are items that need to be fixed. These are the items we don't want to be seeing.

Remember, we had the six main topics that were covered in the scan, as shown in the following:

- The host configuration
- The Docker daemon configuration
- The Docker daemon configuration files
- Container images and build files
- Container runtime
- The Docker security operations

Let's take a look at what we are seeing in each section of the scan. These scan results are from a default Ubuntu Docker host, with no tweaks made to the system at this point. We want to again focus on the `[WARN]` items in each section. Other warnings may come up when you run yours, but these will be the ones that come up the most, if not for everyone at first.

Understanding the output – host configuration

Let's take a look at the following output for the host configuration runtime output:

```
[WARN] 1.1 - Create a separate partition for containers
```

For this one, you will want to map `/var/lib/docker` to a separate partition.

```
[WARN] 1.8 - Failed to inspect: auditctl command not found.
[WARN] 1.9 - Failed to inspect: auditctl command not found.
[WARN] 1.10 - Failed to inspect: auditctl command not found.
[WARN] 1.13 - Failed to inspect: auditctl command not found.
[WARN] 1.18 - Failed to inspect: auditctl command not found.
```

Understanding the output – the Docker daemon configuration

Let's take a look at the following output for the Docker daemon configuration output:

```
[WARN] 2.2 - Restrict network traffic between containers
```

By default, all the containers running on the same Docker host have access to each other's network traffic. To prevent this, you would need to add the `--icc=false` flag to the Docker daemon's start up process:

```
[WARN] 2.7 - Do not use the aufs storage driver
```

Again, you can add a flag to your Docker daemon start up process that will prevent Docker from using the `aufs` storage driver. Using `-s <storage_driver>` on your Docker daemon startup, you can tell Docker not to use `aufs` for storage. It is recommended that you use the best storage driver for the OS on the Docker host that you are using.

Understanding the output – the Docker daemon configuration files

If you are using the stock Docker daemon, you should not see any warnings. If you have customized the code in some way, you may get a few warnings here. This is one area where you should hope to never see any warnings.

Understanding the output – container images and build files

Let's take a look at the following output for the container images and build files runtime output:

```
[WARN] 4.1 - Create a user for the container
[WARN] * Running as root: suspicious_mccarthy
```

This states that the `suspicious_mccarthy` container is running as the root user and it is recommended to create another user to run your containers.

Understanding the output – container runtime

Let's take a look at the output for the container runtime output, as follows:

```
[WARN] 5.1: - Verify AppArmor Profile, if applicable
[WARN] * No AppArmorProfile Found: suspicious_mccarthy
```

This states that the suspicious_mccarthy container does not have AppArmorProfile, which is the additional security provided in Ubuntu in this case.

```
[WARN] 5.3 - Verify that containers are running only a single main
process
[WARN] * Too many processes running: suspicious_mccarthy
```

This error is pretty straightforward. You will want to make sure that you are only running one process per container. If you are running more than one process, you will want to spread them out across multiple containers and use container linking, as shown in the following command:

```
[WARN] 5.4 - Restrict Linux Kernel Capabilities within containers
[WARN] * Capabilities added: CapAdd=[audit_control] to suspicious_
mccarthy
```

This states that the audit_control capability has been added to this running container. You can use --cap-drop={} from your docker run command to remove the additional capabilities from a container, as follows:

```
[WARN] 5.6 - Do not mount sensitive host system directories on containers
[WARN] * Sensitive directory /etc mounted in: suspicious_mccarthy
[WARN] * Sensitive directory /lib mounted in: suspicious_mccarthy
[WARN] 5.7 - Do not run ssh within containers
[WARN] * Container running sshd: suspicious_mccarthy
```

This is straight to the point. No need to run SSH inside your containers. You can do everything you want to with your containers using the tools provided by Docker. Ensure that SSH is not running in any container. You can utilize the docker exec command to execute the items against your containers (see more information here: https://docs.docker.com/engine/reference/commandline/exec/), as shown in the following command:

```
[WARN] 5.10 - Do not use host network mode on container
[WARN] * Container running with networking mode 'host':
suspicious_mccarthy
```

The issue with this one is that, when the container was started, the `--net=host` switch was passed along. It is not recommended to use this as it allows the container to modify the network configuration and open low port numbers as well as access networking services on the Docker host, as follows:

```
[WARN] 5.11 - Limit memory usage for the container
[WARN] * Container running without memory restrictions:
suspicious_mccarthy
```

By default, the containers don't have memory restrictions. This can be dangerous if you are running multiple containers per Docker host. You can use the `-m` switch while issuing your `docker run` commands to limit the containers to a certain amount of memory. Values are set in megabytes (that is, 512 MB or 1024 MB), as shown in the following command:

```
[WARN] 5.12 - Set container CPU priority appropriately
[WARN] * The container running without CPU restrictions:
suspicious_mccarthy
```

Like the memory option, you can also set the CPU priority on a per-container basis. This can be done using the `--cpu-`shares switch while issuing your `docker run` command. The CPU share is based off of the number 1,024. Therefore, half would be 512 and 25% would be 256. Use 1,024 as the base number to determine the CPU share, as follows:

```
[WARN] 5.13 - Mount container's root filesystem as readonly
[WARN] * Container running with root FS mounted R/W:
suspicious_mccarthy
```

You really want to be using your containers as immutable environments, meaning that they don't write any data inside them. Data should be written out to volumes. Again, you can use the `--read-`only switch, as follows:

```
[WARN] 5.16 - Do not share the host's process namespace
[WARN] * Host PID namespace being shared with: suspicious_mccarthy
```

This error arises when you use the `--pid=host` switch. It is not recommended to use this switch as it breaks the isolation of processes between the container and Docker host.

Understanding the output – Docker security operations

Again, another section you should hope to never see are the warnings if you are using stock Docker. Mostly, here you will see the information and should review this to make sure it's all kosher.

Summary

In this chapter, we took a look at the CIS guidelines for Docker. This guide will assist you in setting up multiple aspects of your Docker environment. Lastly, we looked at the Docker Bench for Security. We looked at how to get it up and running and went through an example of what the output would look like once it has been run. We then took a look at the output to see what all it meant. Remember the six items that the application covered: host configuration, Docker daemon configuration, Docker daemon configuration files, container images and build files, container runtime, and Docker security operations.

In the next chapter, we will be taking a look at how to monitor as well as report any Docker security issues that you come across. This will help you know where to look for anything related to the security that may pertain to your existing environment. If you are to come across security-related issues that you find yourself, there are best practices for reporting these issues to give time to Docker to fix them before allowing the public community time to know about the issue, which will allow the hackers to use these vulnerabilities to their advantage.

5
Monitoring and Reporting Docker Security Incidents

In this chapter, we will take a look at how to stay on top of the items that Docker has released, regarding the security findings in order to be aware of your environments. Also, we will take a look at how to safely report any security findings that you come across in order to ensure that Docker has a chance to alleviate the concern before it becomes public and widespread. In this chapter, we will be covering the following topics:

- Docker security monitoring
- Docker **Common Vulnerabilities and Exposures** (**CVE**)
- Mailing lists
- Docker security reporting
    - Responsible disclosure
    - Security reporting
- Additional Docker resources
    - Docker Notary
    - Hardware signing
    - Reading materials

Docker security monitoring

In this section, we will take a look at some ways of monitoring security issues that relate to any Docker products you may be using. While you are using the various products, you need to be able to be aware of, if any, security issues that arise so that you can mitigate these risks to keep your environments and data safe.

Docker CVE

To understand what a Docker CVE is, you need to first know what is CVE. CVEs are actually a system that is maintained by the MITRE Corporation. These are used as a public way of providing information based on a CVE number that is dedicated to each vulnerability for easy reference. This allows a national database of all the vulnerabilities that are given a CVE number from the MITRE Corporation. To learn more about CVEs, you can find it on the Wikipedia article here:

`https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures`

The Wikipedia article explains things such as how they go about giving CVE numbers and the format that they all follow.

Now that you know what CVEs are, you probably have already pieced together what Docker CVEs are. They are CVEs that are directly related to Docker security incidents or issues. To learn more about Docker CVEs or see a list of current Docker CVEs, visit `https://www.docker.com/docker-cve-database`.

This listing will be updated anytime a CVE is created for a Docker product. As you can see, the list is very small, therefore, this is probably a list that will not grow on a day-to-day, or even a month-to-month, basis frequency.

Mailing lists

Another method for following or discussing security-related issues of any Docker products in the ecosystem is to join their mailing lists. Currently, they have two mailing lists that you can either join or follow along with.

The first is a developer list that you can join or follow along with. This is a list for those who are either helping in assisting with contributing the code to the Docker products or developing products using the Docker code base provided in the following:

`https://groups.google.com/forum/#!forum/docker-dev`

The second list is a user list. This list is for those who, you guessed it, are the users of the various Docker products that might have security-related questions. You can search from the already submitted discussions, join existing conversations, or ask new questions that will be answered by those who are also on the mailing lists at the following forum:

`https://groups.google.com/forum/#!forum/docker-user`

Before asking some security-related questions, you will want to read the following section to ensure that you are not exposing any existing security issues that might tempt an attacker out there.

Docker security reporting

Reporting Docker security issues is just as important as monitoring security issues with regards to Docker. While it is important to report these issues, there are certain standards that you should follow when you find security issues and are going to, hopefully, report them.

Responsible disclosure

When disclosing security-related issues, not only for Docker, but for any product out there, there is a term called **responsible disclosure** that everyone should follow. Responsible disclosure is an agreement that allows the developer or maintainer of the product ample time to provide a fix for the security issue before disclosing the issue to the general public.

To learn more about responsible disclosure, you can visit `https://en.wikipedia.org/wiki/Responsible_disclosure`.

Remember to put yourself in the shoes of the group that is responsible for the code. If it were your code, wouldn't you want someone to give you a notice of a vulnerability so that you had ample time to fix the issue before it was disclosed, causing widespread panic and flooding the inbox with e-mails from the masses.

Security reporting

Currently, the method for reporting security issues is to e-mail the Docker security team and give them as much information as you can provide about the security issue. While these are not the exact items that Docker might recommend, there are general guidelines that most other security professionals like to see when reporting security issues, such as the following:

- Product and version, where the security issue was discovered
- Method to reproduce the issue
- Operating system that was being used at the time, plus the version
- Any additional information you can provide

Remember, the more information you provide from the beginning, the quicker the team has to react from their end by being on top of the issue and attack it more aggressively from the start.

To report a security issue for any Docker-related product, make sure to e-mail any information to `security@docker.com`

Additional Docker security resources

If you are looking for some other items to look into, there are some additional items that we have covered in *Chapter 1*, *Securing Docker Hosts* that are worthwhile to conduct a quick review. Make sure to look back at *Chapter 1*, *Securing Docker Hosts* to get more details on the next couple of items or links that will be provided in each section.

Docker Notary

Let's take a quick look at **Docker Notary**, but for more information about Docker Notary, you can look back at the information in *Chapter 2*, *Securing Docker Components* or the following URL:

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

Docker Notary allows you to publish your content by signing it with private keys that you are recommended to keep offline. Using these keys to sign your content helps in ensuring others to know that the content they are using is, in fact, from who it says it is—you—and that the content can be trusted, assuming the users trust you.

Docker Notary has a few key goals that I believe are important to point out in the following:

- Survivable key compromise
- Freshness guarantee
- Configurable trust thresholds
- Signing delegation
- Use of existing distribution
- Untrusted mirrors and transport

It is important to know that Docker Notary has a server and client component as well. To use Notary, you will have to be familiar with the command-line environment. The preceding link will break it down for you and give you walkthroughs on setting up and using each component.

Hardware signing

Similar to the previous *Docker Notary* section, let's take a quick look at the hardware signing as it's a very important feature that must be understood fully.

Docker also allows hardware signing. What does this mean? From the previous section, we saw that you can use highly secure keys to sign your content, allowing others to verify that the information is from who it says it is, which ultimately provides everyone great peace of mind.

Hardware signing takes this to a whole new level by allowing you to add yet another layer of code signing. By introducing a hardware device, Yubikey—a USB piece of hardware—you can use your private keys (remember to keep them secure and offline somewhere) as well as a piece of hardware that requires you to tap it when you sign your code. This proves that you are a human by the fact of having to physically touch the YubiKey when you are signing your code.

For more information about the hardware signing part of Notary, it is worthwhile to read their announcement when they released this feature from the following URL:

`https://blog.docker.com/2015/11/docker-content-trust-yubikey/`

For a video demonstration of using **YubiKeys** and Docker Notary, please visit the following YouTube URL:

`https://youtu.be/fLfFFtOHRZQ?t=1h21m23s`

To find out more information about YubiKeys, visit their website at the following URL:

```
https://www.yubico.com
```

Reading materials

There are also some additional reading materials that can assist with ensuring your focus is on monitoring the security aspect of the entire Docker ecosystem.

Looking back at *Chapter 4*, *Docker Bench for Security*, we covered the Docker Bench, which is a scanning application for your entire Docker environment. This is highly useful to help in pointing out any security risks that you might have.

There is also a great free Docker security eBook that I found. This book will cover potential security issues along with tools and techniques that you can utilize to secure your container environments. Not bad for free, right?! You can find this book at the following URL:

```
https://www.openshift.com/promotions/docker-security.html
```

You can refer to the following *Introduction to Container Security* whitepaper for more information:

```
https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/
WP_Intro_to_container_security_03.20.2015.pdf
```

You can also refer to *The Definitive Guide To Docker Containers* whitepaper, as follows:

```
https://www.docker.com/sites/default/files/WP-%20Definitive%20
Guide%20To%20Containers.pdf
```

The last two items—*Introduction to Container Security* whitepaper and *The Definitive Guide To Docker Containers*—are directly created from Docker, therefore, they contain information that is directly related to understanding how containers are structured and they breakdown a lot of the Docker information into a central location, which you can download or print out and have at hand at any point of time. They also help you to understand the various layers of containers and how they help keep your environment and applications secure from each other.

Awesome Docker

While this is not a security-related tool, it is a Docker tool that is very useful and is updated quite frequently. Awesome Docker is a curated list of any and all Docker projects. It allows others to contribute with pull requests to the curated list. The list includes topics for those who are looking to get started with Docker; useful articles; deep-dive articles; networking articles; and articles on using multi-server Docker environments, cloud infrastructure, tips, and newsletters, the list just keeps going on. To view the project as well as the *awesomeness* of everything that it includes, visit the following URL:

```
https://github.com/veggiemonk/awesome-docker
```

Summary

In this chapter, we looked at a number of ways to monitor and report Docker security issues. We looked at some mailing lists that you can join monitoring the Docker CVE list. We also reviewed using both Docker Notary to sign your images as well as hardware signing to utilize hardware items such as YubiKeys. We also looked at using responsible disclosure, which is giving Docker a chance to fix any security-related issue prior to releasing them to the public.

In the next chapter, we will be looking at working with some Docker tools. These tools can be used to secure the Docker environment. We will look at both command-line tools as well as GUI tools that you can use to your advantage. We will be looking at utilizing TLS in your environments using read-only containers, utilizing kernel namespaces and control groups, and mitigating against the risk, while being aware of the Docker daemon attack surface.

6
Using Docker's Built-in Security Features

In this chapter, we will take a look at working with Docker tools that can be used to secure your environment. We will be taking a look at both command-line tools as well as GUI tools that you can utilize to your advantage. We will cover the following items in this chapter:

- Docker tools
    - Using TLS in your environments to help ensure that pieces are communicating securely
    - Using read-only containers to help protect the data in a container from being manipulated in some form
- Docker security fundamentals
    - Kernel namespaces
    - Control groups
    - Linux kernel capabilities

Docker tools

In this section, we will cover the tools that can help you secure your Docker environment. These are options that are built into the Docker software, which you are already using. It's time to learn how to enable or utilize these such features to help give you the peace of mind in order to be sure that the communication is secure; this is where we will cover enabling TLS, which is a protocol that ensures privacy between applications. It ensures that nobody is listening in on the communication. Think of it as when you are watching a movie and people on the phone say, *is this line secure?* It's the same kind of idea when it comes to network communication. Then, we will look at how you can utilize the read-only containers to ensure that the data you are serving up can't be manipulated by anyone.

Using TLS

It is highly recommended to use the Docker Machine to create and manage your Docker hosts. It will automatically set up the communication to use TLS. Here's how you can verify that the *default* host that was created by `docker-machine` indeed uses TLS.

One of the important factors is knowing if you are using TLS or not and then adjusting to use TLS if you are, in fact, not using TLS. The important thing to remember is that, nowadays, almost all the Docker tools ship with the TLS enabled, or if they don't, they appear to be working towards this goal. One of the commands that you can use to check in order to see if your Docker host is utilizing the TLS is with the Docker Machine `inspect` command. In the following, we will take a look at a host and see if it is running with the TLS enabled:

```
docker-machine inspect default


{
    "ConfigVersion": 3,
    "Driver": {
      "IPAddress": "192.168.99.100",
      "MachineName": "default",
      "SSHUser": "docker",
      "SSHPort": 50858,
      "SSHKeyPath": "/Users/scottgallagher/.docker/
        machine/machines/default/id_rsa",
      "StorePath": "/Users/scottgallagher/.docker/machine",
      "SwarmMaster": false,
      "SwarmHost": "tcp://0.0.0.0:3376",
      "SwarmDiscovery": "",
      "VBoxManager": {},
      "CPU": 1,
      "Memory": 2048,
      "DiskSize": 204800,
      "Boot2DockerURL": "",
      "Boot2DockerImportVM": "",
      "HostDNSResolver": false,
      "HostOnlyCIDR": "192.168.99.1/24",
      "HostOnlyNicType": "82540EM",
      "HostOnlyPromiscMode": "deny",
      "NoShare": false,
      "DNSProxy": false,
      "NoVTXCheck": false
    },
    "DriverName": "virtualbox",
    "HostOptions": {
      "Driver": "",
      "Memory": 0,
      "Disk": 0,
      "EngineOptions": {
        "ArbitraryFlags": [],
        "Dns": null,
        "GraphDir": "",
        "Env": [],
        "Ipv6": false,
        "InsecureRegistry": [],
        "Labels": [],
        "LogLevel": "",
        "StorageDriver": "",
        "SelinuxEnabled": false,
        "TlsVerify": true,
        "RegistryMirror": [],
        "InstallURL": "https://get.docker.com"
      },
      "SwarmOptions": {
        "IsSwarm": false,
        "Address": "",
        "Discovery": "",
        "Master": false,
        "Host": "tcp://0.0.0.0:3376",
        "Image": "swarm:latest",
        "Strategy": "spread",
        "Heartbeat": 0,
        "Overcommit": 0,
        "ArbitraryFlags": [],
        "Env": null
      },
      "AuthOptions": {
        "CertDir": "/Users/scottgallagher/.docker/machine/certs",
        "CaCertPath": "/Users/scottgallagher/.docker/
          machine/certs/ca.pem",
        "CaPrivateKeyPath": "/Users/scottgallagher/.docker/
           machine/certs/ca-key.pem",
        "CaCertRemotePath": "",
        "ServerCertPath": "/Users/scottgallagher/.docker/
          machine/machines/default/server.pem",
        "ServerKeyPath": "/Users/scottgallagher/.docker/
          machine/machines/default/server-key.pem",
        "ClientKeyPath": "/Users/scottgallagher/.docker/
          machine/certs/key.pem",
        "ServerCertRemotePath": "",
        "ServerKeyRemotePath": "",
        "ClientCertPath": "/Users/scottgallagher/.docker/
          machine/certs/cert.pem",
        "ServerCertSANs": [],
        "StorePath": "/Users/scottgallagher/.docker/
          machine/machines/default"
      }
    },
    "Name": "default"
}
```

From the preceding output, we can focus on the following line:

```
    "SwarmHost": "tcp://0.0.0.0:3376",
```

This shows us that if we were running **Swarm**, this host would be utilizing the secure `3376` port. Now, if you aren't using Docker Swarm, then you can disregard this line. However, if you are using Docker Swarm, then this line is important.

Just to take a step back, let's identify what Docker Swarm is. Docker Swarm is native clustering within Docker. It helps in turning multiple Docker hosts into an easy-to-manage single virtual host:

```
 "AuthOptions": {
            "CertDir": "/Users/scottgallagher/.docker/machine/certs",
            "CaCertPath": "/Users/scottgallagher/.docker/machine/certs/
ca.pem",
            "CaPrivateKeyPath": "/Users/scottgallagher/.docker/machine/
certs/ca-key.pem",
            "CaCertRemotePath": "",
            "ServerCertPath": "/Users/scottgallagher/.docker/machine/
machines/default/server.pem",
            "ServerKeyPath": "/Users/scottgallagher/.docker/machine/
machines/default/server-key.pem",
            "ClientKeyPath": "/Users/scottgallagher/.docker/machine/
certs/key.pem",
            "ServerCertRemotePath": "",
            "ServerKeyRemotePath": "",
            "ClientCertPath": "/Users/scottgallagher/.docker/machine/
certs/cert.pem",
            "ServerCertSANs": [],
            "StorePath": "/Users/scottgallagher/.docker/machine/machines/
default"
        }
```

This shows us that this host is, in fact, using the certificates so we know that it is using TLS, but how do we know from just that? In the following section, we will take a look at how to tell that it is, in fact, using TLS for sure.

Docker Machine also has the option to run everything over TLS. This is the most secure way of using Docker Machine in order to manage your Docker hosts. This setup can be tricky if you start using your own certificates. By default, Docker Machine stores your certificates that it uses in `/Users/<user_id>/.docker/machine/certs/`. You can see the location on your machine where the certificates are stored at from the preceding output.

Let's take a look at how we can achieve the goal of viewing if our Docker host is utilize TLS:

```
docker-machine ls

NAME      ACTIVE   URL          STATE     URL SWARM    DOCKER    ERRORS

default   *        virtualbox   Running   tcp://192.168.99.100:2376
v1.9.1
```

This is where we can tell that it is using TLS. The insecure port of Docker Machine hosts is the `2375` port and this host is using `2376`, which is the secure TLS port for Docker Machine. Therefore, this host is, in fact, using TLS to communicate, which gives you the peace of mind in knowing that the communication is secure.

Read-only containers

With respect to the `docker run` command, we will mainly focus on the option that allows us to set everything inside the container as read-only. Let's take a look at an example and break down what it exactly does:

```
$ docker run --name mysql --read-only -v /var/lib/mysql v /tmp --e MYSQL_
ROOT_PASSWORD=password -d mysql
```

Here, we are running a `mysql` container and setting the entire container as read-only, except for the `/var/lib/mysql` directory. What this means is that the only location the data can be written inside the container is the `/var/lib/mysql` directory. Any other location inside the container won't allow you to write anything in it. If you try to run the following, it would fail:

```
$ docker exec mysql touch /opt/filename
```

This can be extremely helpful if you want to control where the containers can write to or not write to. Make sure to use this wisely. Test thoroughly, as it can have consequences when the applications can't write to certain locations.

Remember the Docker volumes we looked at in the previous chapters, where we were able to set the volumes to be read-only. Similar to the previous command with `docker run`, where we set everything to read-only, except for a specified volume, we can now do the opposite and set a single volume (or more, if you use more `-v` switches) to read-only. The thing to remember about volumes is that when you use a volume and mount it in a container, it will mount as an empty volume over the top of that directory inside the container, unless you use the `--volumes-from` switch or add data to the container in some other way after the fact:

```
$ docker run -d -v /opt/uploads:/opt/uploads:/opt/uploads:ro nginx
```

This will mount a volume in `/opt/uploads` and set it to read-only. This can be useful if you don't want a running container to write to a volume in order to keep the data or configuration files intact.

The last option that we want to look at, with regards to the `docker run` command is the `--device=` switch. This switch allows us to mount a device from the Docker host into a specified location inside the container. For doing so, there are some security risks that we need to be aware of. By default, when you do this, the container will get full the access: read, write, and the `mknod` access to the device's location. Now, you can control these permissions by manipulating `rwm` at the end of the switch command.

Let's take a look at some of these and see how they work:

```
$ docker run --device=/dev/sdb:/dev/sdc2 -it ubuntu:latest /bin/bash
```

The previous command will run the latest Ubuntu image and mount the `/dev/sdb` device inside the container at the `/dev/sdc2` location:

```
$ docker run --device=/dev/sdb:/dev/sdc2:r -it ubuntu:latest /bin/bash
```

This command will run the latest Ubuntu image and mount the `/dev/sdb1` device inside the container at the `/dev/sdc2` location. However, this one has the `:r` tag at the end of it that specifies that it's read-only and can't be written.

Docker security fundamentals

In the previous sections, we looked into some Docker tools that you can use, such as TLS for communication, and using read-only containers to help ensure data isn't changed or manipulated. In this section, we will focus on some more options that are available from within the Docker ecosystem that can be used to help secure up your environments to another level. We will take a look at the kernel namespaces that provide another layer of abstraction by providing the running process to its own resources that appear only to the process itself and not to other processes that might be running. We will cover more about kernel namespaces in this section. We will then take a look at the control groups. Control groups, more commonly known as cgroups, give you the ability to limit the resources that a particular process has. We will then cover the Linux kernel capabilities. By that, we will look at the restrictions that are placed on containers, by default, when they are run using Docker. Lastly, we will take a look at the Docker daemon attack surface, risks that exist with the Docker daemon that you need to be aware of, and mitigation of these risks.

Kernel namespaces

Kernel namespaces provide a form of isolation for containers. Think of them as a container wrapped inside another container. Processes that are running in one container can't disrupt the processes running inside another container or let alone run on the Docker host that the container is operating on. The way this works is that each container gets its own network stacks to operate with. However, there are ways to link these containers together in order to be able to interact with each other; however, by default, they are isolated from each other. Kernel namespaces have been around for quite a while too, so they are a tried and true method of isolation protection. They were introduced in 2008 and at the time of writing this book, it's 2016. You can see that they will be eight years old, come this July. Therefore, when you issue the `docker run` command, you are benefiting from a lot of heavy lifting that is going on behind the scenes. This heavy lifting is creating its own network stack to operate on. This is also shielding off the container from other containers being able to manipulate the container's running processes or data.

Control groups

Control groups, or more commonly referred to as cgroups, are a Linux kernel feature that allows you to limit the resources that a container can use. While they limit the resources, they also ensure that each container gets the resources it needs as well as that no single container can take down the entire Docker host.

With control groups, you can limit the amount of CPU, memory, or disk I/O that a particular container gets. If we look at the `docker run` command's help, let's highlight the items that we can control. We will just be highlighting a few items that are particularly useful for the majority of users, but please review them to see if any others fit your environment, as follows:

```
$ docker run --help


Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]


Run a command in a new container


  -a, --attach=[]                  Attach to STDIN, STDOUT or STDERR
  --add-host=[]                    Add a custom host-to-IP mapping
(host:ip)
  --blkio-weight=0                 Block IO (relative weight), between 10
and 1000
  --cpu-shares=0                   CPU shares (relative weight)
  --cap-add=[]                     Add Linux capabilities
  --cap-drop=[]                    Drop Linux capabilities
  --cgroup-parent=                 Optional parent cgroup for the
container
  --cidfile=                       Write the container ID to the file
  --cpu-period=0                   Limit CPU CFS (Completely Fair
Scheduler) period
  --cpu-quota=0                    Limit CPU CFS (Completely Fair
Scheduler) quota
  --cpuset-cpus=                   CPUs in which to allow execution (0-3,
0,1)
  --cpuset-mems=                   MEMs in which to allow execution (0-3,
0,1)
  -d, --detach=false               Run container in background and print
container ID
  --device=[]                      Add a host device to the container
  --disable-content-trust=true     Skip image verification
  --dns=[]                         Set custom DNS servers
  --dns-opt=[]                     Set DNS options
  --dns-search=[]                  Set custom DNS search domains
  -e, --env=[]                     Set environment variables
  --entrypoint=                 Overwrite the default ENTRYPOINT of the
image
  --env-file=[]                 Read in a file of environment variables
  --expose=[]                   Expose a port or a range of ports
  --group-add=[]                Add additional groups to join
  -h, --hostname=               Container host name
  --help=false                  Print usage
  -i, --interactive=false       Keep STDIN open even if not attached
  --ipc=                        IPC namespace to use
  --kernel-memory=              Kernel memory limit
  -l, --label=[]                Set meta data on a container
  --label-file=[]               Read in a line delimited file of labels
  --link=[]                     Add link to another container
  --log-driver=                 Logging driver for container
  --log-opt=[]                  Log driver options
  --lxc-conf=[]                 Add custom lxc options
  -m, --memory=                 Memory limit
  --mac-address=                Container MAC address (e.g.
92:d0:c6:0a:29:33)
  --memory-reservation=         Memory soft limit
  --memory-swap=                Total memory (memory + swap), '-1' to
disable swap
  --memory-swappiness=-1        Tuning container memory swappiness (0
to 100)
  --name=                       Assign a name to the container
  --net=default                 Set the Network for the container
  --oom-kill-disable=false      Disable OOM Killer
  -P, --publish-all=false       Publish all exposed ports to random
ports
  -p, --publish=[]              Publish a container's port(s) to the
host
  --pid=                        PID namespace to use
  --privileged=false            Give extended privileges to this
container
  --read-only=false             Mount the container's root filesystem
as read only
  --restart=no                  Restart policy to apply when a
container exits
```

```
  --rm=false                       Automatically remove the container when
it exits
  --security-opt=[]                Security Options
  --sig-proxy=true                 Proxy received signals to the process
  --stop-signal=SIGTERM            Signal to stop a container, SIGTERM by
default
  -t, --tty=false                  Allocate a pseudo-TTY
  -u, --user=                      Username or UID (format:
<name|uid>[:<group|gid>])
  --ulimit=[]                      Ulimit options
  --uts=                           UTS namespace to use
  -v, --volume=[]                  Bind mount a volume
  --volume-driver=                 Optional volume driver for the
container
  --volumes-from=[]                Mount volumes from the specified
container(s)
  -w, --workdir=                   Working directory inside the container
```

As you can see from the preceding highlighted portions, these are just a few items that you can control on a per-container basis.

Linux kernel capabilities

Docker uses the kernel capabilities to place the restrictions that Docker places on the containers when they are launched or started. Limiting the root access is the ultimate agenda with these kernel capabilities. There are a few services that typically run as root, but can now be run without these permissions. Some of these include `SSH`, `cron`, and `syslogd`.

Overall, what this means is that you don't need root in the server sense you typically think of. You can run with a reduced capacity set. This means that your root user doesn't need the privilege it typically needs.

Some of the things that you might not need to enable anymore are shown in the following:

- Performing mount operations
- Using raw sockets, which will help to prevent spoofing of packets
- Creating new devices
- Changing the owner of files
- Altering attributes

This helps due to the fact that if someone does compromise a container, then they can't escalate any more than what you are providing them. It will be much harder, if not impossible, to escalate their privileges from a running container to running Docker host. Due to such complexity, the attackers will probably look elsewhere than your Docker environments to try to attack. Docker also supports the addition and removal of capabilities, therefore, it's recommend to remove all the capabilities, except the ones that you intend to use. An example would be to use the `-cap-add net_bind_service` switch on your `docker run` command.

Containers versus virtual machines

Hopefully, you trust your organization and all those who have access to these systems. You will most likely be setting up virtual machines from scratch. It is probably impossible to get the virtual machine from someone else due to its sheer size. Therefore, you will be aware of what is inside the virtual machine and what isn't. This being said, with the Docker containers, you will not be aware of what is there inside the image that you may be using for your container(s).

Summary

In this chapter, we looked at deploying TLS to all the pieces of our Docker environment so that we can ensure that everything is communicating securely and the traffic can't be intercepted and then interpreted. We also understood how to utilize the read-only containers to our advantage in order to ensure the data that is being served up can't be manipulated. We then took a look at how to provide processes with their own abstraction of items, such as networks, mounts, users, and more. We then dove into control groups, or cgroups as their more commonly referred to as, as a way to limit the resources that a process or container has. We also took a look at the Linux kernel capabilities, that is, the restrictions that are placed on a container when it is started or launched. Lastly, we dove into mitigating risks against the Docker daemon attack surface.

In the next chapter, we will look at securing Docker with third-party tools and learn which third-party tools, beyond those offered by Docker, are out there to help secure your environments to help keep your application(s) secure when running on Docker.

7

Securing Docker with Third-party Tools

In this chapter, let's take a look at securing Docker using third-party tools. These would be tools that are not part of the Docker ecosystem, which you can use to help secure your systems. We will be taking a look at the following three items:

- **Traffic Authorization**: This allows inbound and outbound traffic to be verified by the token broker in order to ensure that traffic between services is secure.
- **Summon**: Summon is a command-line tool that reads a file in the `secrets.yml` format and injects secrets as environment variables into any process. Once the process exits, the secrets are gone.
- **sVirt and SELinux**: sVirt is a community project that integrates **Mandatory Access Control** (**MAC**) security and Linux-based virtualization (**Kernel-base Virtual Machine** (**KVM**), lguest, and so on).

We will then add bonus material with regards to some extra third-party tools that are quite useful and powerful and deserve to get some recognition as useful third-party tools. These tools include **dockersh**, **DockerUI**, **Shipyard**, and **Logspout**. Without further ado, let's jump in and get started on our path to the most secure environments that we can obtain.

Third-party tools

So, what third-party tools will we focus on? Well from the preceding introduction, you learned that we will be looking at three tools in particular. These would be Traffic Authorization, Summon, and sVirt with SELinux. All the three tools help in different aspects and can be used to perform different things. We will learn the differences between them and help you to determine which ones to implement. You can decide whether you want to implement them all, only one or two of them, or maybe you feel that none of these would pertain to your current environment. However, it is good to know what is out there, in case, your needs change and the overall architecture of your Docker environments change over time.

Traffic Authorization

Traffic Authorization can be used to regulate HTTP/HTTPS traffic between services. This involves a forwarder, gatekeeper, and token broker. This allows inbound and outbound traffic to be verified by the token broker in order to ensure that traffic between services is secure. Each container runs a gatekeeper that is used to intercept all the HTTP/HTTPS inbound traffic and verifies its authenticity from a token that is found in the authorization header. The forwarder also runs on each container, and like the gatekeeper, this also intercepts traffic; however, instead of intercepting inbound traffic, it intercepts outbound traffic and places the token on the authorization header. These tokens are issues from the token broker. These tokens can also be cached to save time and minimize the impact of latency. Let's break it down into a series of steps, as shown in the following:

1. Service A initiates a request to Service B.
2. The forwarder on Service A will authenticate itself with the token broker.
3. The token broker will issue a token that Service A will apply to the authorization header and forward the request to Service B.
4. Service B's gatekeeper will intercept the request and verify the authorization header against the token broker.
5. Once the authorization header has been verified, it is then forwarded to Service B.

As you can see, this applies extra authorizations on both inbound and outbound requests. As we will see in the next section, you can also use Summon along with Traffic Authorization to use shared secrets that are available once they are used, but go away once the application has completed its actions.

For more information about Traffic Authorization and Docker, visit `https://blog.conjur.net/securing-docker-with-secrets-and-dynamic-traffic-authorization`.

Summon

Summon is a command-line tool and is used to help pass along secrets or things you don't want exposed, such as passwords or environmental variables and then these secrets are disposed upon exiting the process. This is great as once the secret is used and the process exits, the secret no longer exists. This means the secret isn't lingering around until it is either removed manually or discovered by an attacker for malicious use. Let's take a look at how to utilize Summon.

Summon typically uses three files: a `secrets.yml` file, script used to perform the action or task, and Dockerfile. As you have learned previously, or based on your current Docker experience, the Dockerfile is the basis of what helps in building your containers and has instructions on how to set up the container, what to install, what to configure, and so on.

One great example have for the usage of Summon is to be able to deploy your AWS credentials to a container. For utilizing AWS CLI, you need a few key pieces of information that should be kept secret. These two pieces of information are your **AWS Access Key ID** and **AWS Secret Access Key**. With these two pieces of information, you can manipulate someone's AWS account and perform actions within this account. Let's take a look at the contents of one of these files, the `secrets.yml` file:

```
secrets.yml
AWS_ACCESS_KEY_ID: !var $env/aws_access_key_id
AWS_SECRET_ACCESS_KEY: !var $env/aws_secret_access_key
```

The `-D` option is used to substitute values while `$env` is an example of a substitution variable, therefore, the options can be interchanged.

In the preceding content, we can see that we want to pass along these two values into our application. With this file, the script file you want to deploy, and the Dockerfile, you are now ready to build your application.

We simply utilize the `docker build` command inside the folder that has our three files in it:

```
$ docker build -t scottpgallagher/aws-deploy .
```

Next, we need to install Summon, which can be done with a simple `curl` command, as follows:

```
$ curl -sSL https://raw.githubusercontent.com/conjurinc/summon/master/
install.sh | bash
```

Now that we have Summon installed, we need to run the container with Summon and pass along our secret values (note that this will only work on OS X):

```
$ security add-generic-password -s "summon" -a "aws_access_key_id" -w
"ACESS_KEY_ID"
$ security add-generic-password -s "summon" -a "aws_secret_access_key" -w
"SECRET_ACCESS_KEY"
```

Now we are ready to run Docker with Summon in order to pass along these credentials to the container:

```
$ summon -p ring.py docker run —env-file @ENVFILE aws-deploy
```

You can also view the values that you have passed along by using the following `cat` command:

```
$ summon -p ring.py cat @SUMMONENVFILE
aws_access_key_id=ACESS_KEY_ID
aws_secret_access_key=SECRET_ACCESS_KEY
```

The `@SUMMONENVFILE` is a memory-mapped file that contains the values from the `secrets.yml` file.

For more information and to see other options to utilize Summon, visit `https://conjurinc.github.io/summon/#examples`.

sVirt and SELinux

sVirt is part of the SELinux implementation, but it is typically turned off as most view it as a roadblock. The only roadblock should be learning sVirt and SELinux.

sVirt is an open source community project that implements MAC security for Linux-based virtualization. A reason you would want to implement sVirt is to improve the security as well as harden the system against any bugs that might exist in the hypervisor. This will help in eliminating any attack vectors that might be aimed towards the virtual machine or host.

Remember that all containers on a Docker host share the usage of the Linux kernel that is running on the Docker host. If there is an exploit to this Linux kernel on the host, then all containers running on this Docker host have the potential to be easily compromised. If you implement sVirt and a container is compromised, there is no way for the compromise to reach your Docker host and then out to other Docker containers.

sVirt utilizes labels in the same way as SELinux. The following table is a list of these labels and their descriptions:

Type	**SELinux Context**	**Description**
Virtual machine processes	`system_u:system_r:svirt_t:MCS1`	`MCS1` is a randomly selected MCS field. Currently, approximately 500,000 labels are supported.
Virtual machine image	`system_u:object_r:svirt_image_t:MCS1`	Only processes labeled `svirt_t` with the same MCS fields are able to read/write these image files and devices.
Virtual machine shared read/write content	`system_u:object_r:svirt_image_t:s0`	All processes labeled `svirt_t` are allowed to write to the `svirt_image_t:s0` files and devices.
Virtual machine image	`system_u:object_r:virt_content_t:s0`	This is the system default label used when an image exits. No `svirt_t` virtual processes are allowed to read files/devices with this label.

Other third-party tools

There are some other third-party tools that do deserve a mention in this chapter and are worth exploring to see the value that they can add for you. It seems that these days, a lot of focus is on GUI applications to help with securing applications and infrastructures. The following utilities will give you a few options that could be pertinent to the environment you are running with the Docker tools.

> Note that you should use caution when implementing some of the following items as there could be unwanted repercussions. Make sure to use testing environments prior to production implementation.

dockersh

The dockersh was designed to be used as a login shell replacement on machines that support multiple interactive users. Why is this important? If you remember some of the general security warnings that you have when dealing with Docker containers on a Docker host, you will know that whoever has access to the Docker host has access to all the running containers on this Docker host. With dockersh, you can isolate the use on a per-container basis and only allow users access the containers that you want them to, while maintaining administrative control over the Docker host and keeping the security threshold minimum.

This is an ideal way to help isolate users on a per-container basis, while containers help eliminate the need for SSH by utilizing dockersh, you can remove some of these fears about providing everyone that needs container to access, the access to the Docker host(s) as well. There is a lot of information required to set up and invoke dockersh, therefore, if you are interested, it's recommended to visit the following URL to find more about dockersh, including how to set it up and use it:

`https://github.com/Yelp/dockersh`

DockerUI

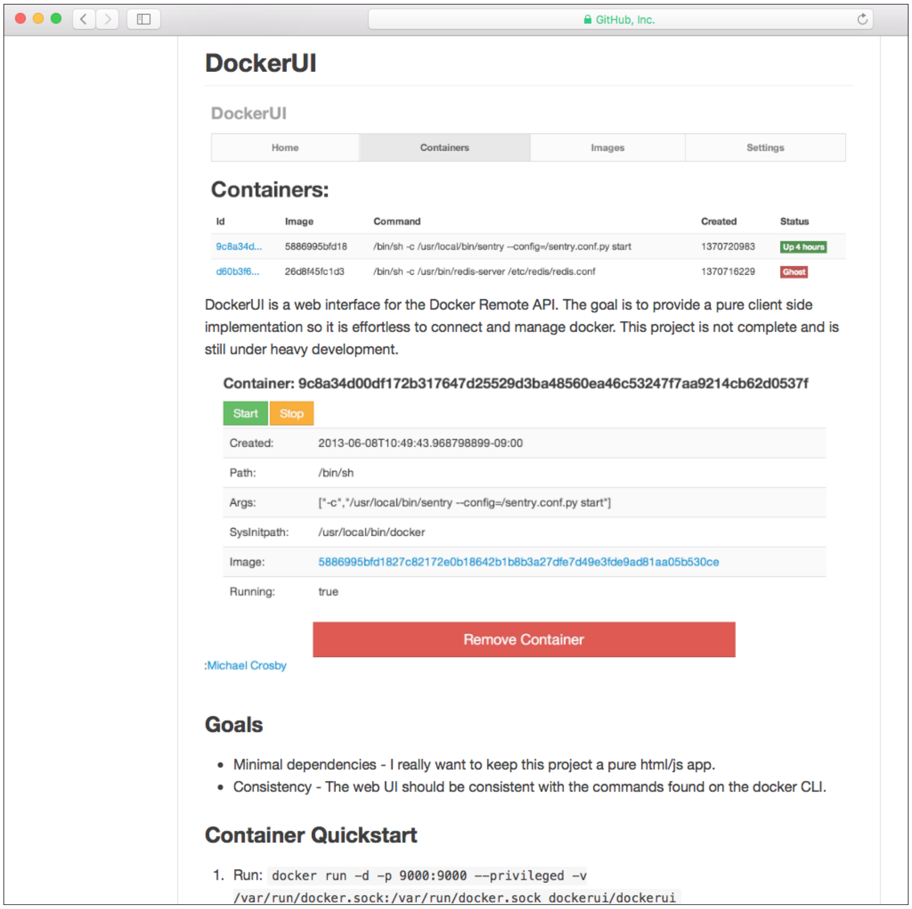
DockerUI is a simple way to view what is going on inside your Docker host. The installation of DockerUI is very straightforward and is done by running a simple `docker run` command in order to get started:

```
$ docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/
run/docker.sock dockerui/dockerui
```

To access the DockerUI, you simply open a browser and navigate to the following link:

```
http://<docker_host_ip>:9000
```

This opens your DockerUI to the world on port `9000`, as shown in the following screenshot:



DockerUI

DockerUI

Containers:

Id	Image	Command	Created	Status
9c8a34d...	5886995bfd18	/bin/sh -c /usr/local/bin/sentry --config=/sentry.conf.py start	1370720983	Up 4 hours
d60b3f6...	26d8f45fc1d3	/bin/sh -c /usr/bin/redis-server /etc/redis/redis.conf	1370716229	Ghost

DockerUI is a web interface for the Docker Remote API. The goal is to provide a pure client side implementation so it is effortless to connect and manage docker. This project is not complete and is still under heavy development.

Container: 9c8a34d00df172b317647d25529d3ba48560ea46c53247f7aa9214cb62d0537f

Created:	2013-06-08T10:49:43.968798899-09:00
Path:	/bin/sh
Args:	["-c","/usr/local/bin/sentry --config=/sentry.conf.py start"]
SysInitpath:	/usr/local/bin/docker
Image:	5886995bfd1827c82172e0b18642b1b8b3a27dfe7d49e3fde9ad81aa05b530ce
Running:	true

:Michael Crosby

Goals

- Minimal dependencies - I really want to keep this project a pure html/js app.
- Consistency - The web UI should be consistent with the commands found on the docker CLI.

Container Quickstart

1. Run: `docker run -d -p 9000:9000 --privileged -v /var/run/docker.sock:/var/run/docker.sock dockerui/dockerui`

You can get the general high-level view of your Docker host and its ecosystem and can do things such as manipulate the containers on the Docker host by restarting, stopping, or starting them from a stopped state. DockerUI takes some of the steep learning curve of running command-line items and places them into actions that you perform in a web browser using point and click.

For more information about DockerUI, visit `https://github.com/crosbymichael/dockerui`.

Shipyard

Shipyard, like DockerUI, allows you to use a GUI web interface to manage various aspects—mainly in your containers—and manipulate them. Shipyard is build on top of Docker Swarm so that you get to utilize the feature set of Docker Swarm, where you can manage multiple hosts and containers instead of having to just focus on one host and its containers at a time.

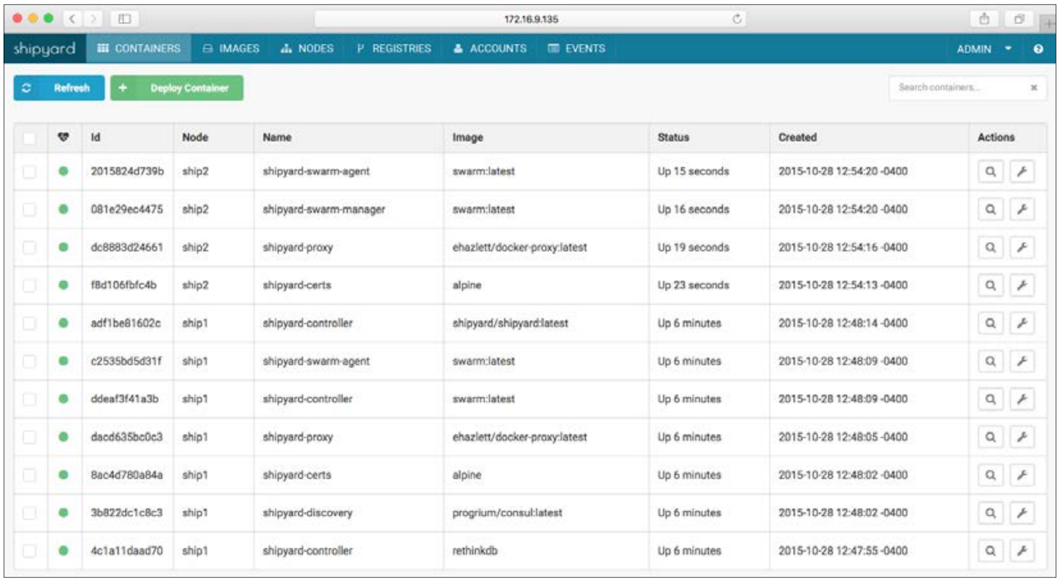
Using Shipyard is simple and the following `curl` command re-enters the picture:

```
$ curl -sSL https://shipyard-project.com/deploy | bash -s
```

To access the Shipyard once the set up is completed, you can simply open a browser and navigate to the following link:

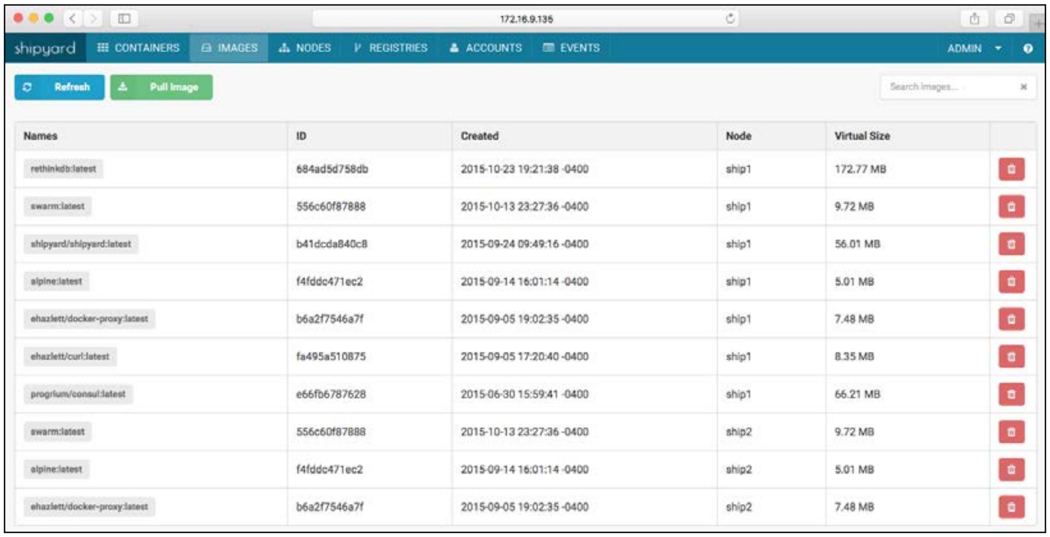
```
http://<docker_host_ip>:8080
```

As we can see in the following screenshot, we can view all the containers on our Docker host:



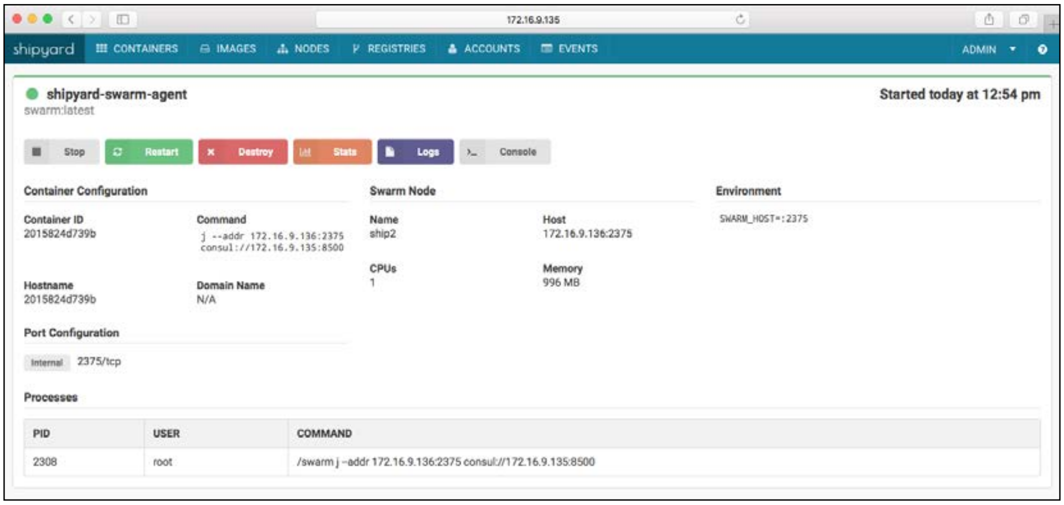
□	♥	Id	Node	Name	Image	Status	Created	Actions
□	●	2015824d739b	ship2	shipyard-swarm-agent	swarm:latest	Up 15 seconds	2015-10-28 12:54:20 -0400	
□	●	081e29ec4475	ship2	shipyard-swarm-manager	swarm:latest	Up 16 seconds	2015-10-28 12:54:20 -0400	
□	●	dc8883d24661	ship2	shipyard-proxy	ehazlett/docker-proxy:latest	Up 19 seconds	2015-10-28 12:54:16 -0400	
□	●	f8d106fbfc4b	ship2	shipyard-certs	alpine	Up 23 seconds	2015-10-28 12:54:13 -0400	
□	●	adf1be81602c	ship1	shipyard-controller	shipyard/shipyard:latest	Up 6 minutes	2015-10-28 12:48:14 -0400	
□	●	c2535bd5d31f	ship1	shipyard-swarm-agent	swarm:latest	Up 6 minutes	2015-10-28 12:48:09 -0400	
□	●	ddeaf3f41a3b	ship1	shipyard-controller	swarm:latest	Up 6 minutes	2015-10-28 12:48:09 -0400	
□	●	dacd635bc0c3	ship1	shipyard-proxy	ehazlett/docker-proxy:latest	Up 6 minutes	2015-10-28 12:48:05 -0400	
□	●	8ac4d780a84a	ship1	shipyard-certs	alpine	Up 6 minutes	2015-10-28 12:48:02 -0400	
□	●	3b822dc1c8c3	ship1	shipyard-discovery	progrium/consul:latest	Up 6 minutes	2015-10-28 12:48:02 -0400	
□	●	4c1a11daad70	ship1	shipyard-controller	rethinkdb	Up 6 minutes	2015-10-28 12:47:55 -0400	

We can also view all the images that are on our Docker host, as shown in the following screenshot:



Names	ID	Created	Node	Virtual Size
rethinkdb:latest	684ad5d758db	2015-10-23 19:21:38 -0400	ship1	172.77 MB
swarm:latest	556c60f87888	2015-10-13 23:27:36 -0400	ship1	9.72 MB
shipyard/shipyard:latest	b41dcda840c8	2015-09-24 09:49:16 -0400	ship1	56.01 MB
alpine:latest	f4fddc471ec2	2015-09-14 16:01:14 -0400	ship1	5.01 MB
ehazlett/docker-proxy:latest	b6a2f7546a7f	2015-09-05 19:02:35 -0400	ship1	7.48 MB
ehazlett/curl:latest	fa495a510875	2015-09-05 17:20:40 -0400	ship1	8.35 MB
progrium/consul:latest	e66fb6787628	2015-06-30 15:59:41 -0400	ship1	66.21 MB
swarm:latest	556c60f87888	2015-10-13 23:27:36 -0400	ship2	9.72 MB
alpine:latest	f4fddc471ec2	2015-09-14 16:01:14 -0400	ship2	5.01 MB
ehazlett/docker-proxy:latest	b6a2f7546a7f	2015-09-05 19:02:35 -0400	ship2	7.48 MB

We can also control our containers, as seen in the following screenshot:



● shipyard-swarm-agent
swarm:latest

Started today at 12:54 pm

Container Configuration

Container ID
2015824d739b

Command
j --addr 172.16.9.136:2375 consul://172.16.9.135:8500

Hostname
2015824d739b

Domain Name
N/A

Swarm Node

Name
ship2

Host
172.16.9.136:2375

CPUs
1

Memory
996 MB

Environment

SWARM_HOST=:2375

Port Configuration

Internal 2375/tcp

Processes

PID	USER	COMMAND
2308	root	/swarm j --addr 172.16.9.136:2375 consul://172.16.9.135:8500

Shipyard, like DockerUI, allows you to manipulate your Docker hosts and containers, by restarting them, stopping them, starting them from a failed state, or deploying new containers and having them join the Swarm cluster. Shipyard also allows you to view information such as port mapping information that is what port from the host maps to the container. This allows you to get a hold of important information like that when you need it quickly to address any security related issues. Shipyard also has user management where DockerUI lacks such capability.

For more information about Shipyard simply visit the following URLs:

- `https://github.com/shipyard/shipyard`
- `http://shipyard-project.com`

Logspout

Where do you go when there is an issue that needs to be addressed? Most people will first look at the logs of that application to see if it is outputting any errors. With Logspout, this becomes a much more manageable task with many multiple running containers. With Logspout, you can route all the logs for each and every container to a location of your choice. Then, you could parse these logs in one place. Instead of having to pull the logs from each container and review them individually you can instead have Logspout do that work for you.

Logspout is just as easy to set up as we have seen for other third-party solutions. Simply run the following command on each Docker host to start collecting the logs:

```
$ docker run --name="logspout" \
    --volume=/var/run/docker.sock:/tmp/docker.sock \
    --publish=127.0.0.1:8000:8080 \
    gliderlabs/logspout
```

Now that we have all the container logs collected in one area, we need to parse through these logs, but how do we do it?

```
$ curl http://127.0.0.1:8000/logs
```

Here's the `curl` command to the rescue again! Logs get prefixed with the container names and colorized in a manner in order to distinguish the logs. You can replace the loopback (`127.0.0.1`) address in the `docker run` invocations with the IP address of the Docker host so that it's easier to connect to in order to be able to get the logs as well as change the port from `8000` to something of your choice. There are also different modules that you can utilize to obtain and collect logs.

For more information about Logspout, visit `https://github.com/gliderlabs/logspout`.

Summary

In this chapter, we looked at some third-party tools in order to be able to help secure Docker environments. Mainly, we looked at three tools: Traffic Authorization, Summon, and sVirt with SELinux. All the three can be utilized in different ways to help secure your Docker environments to give you the peace of mind at end of the day to run your applications in the Docker containers. We learned what third-party tools, beyond those offered by Docker, are out there to help secure your environments to keep your application(s) secure when running on Docker.

We then took a look at some other third-party tools. These are extra tools that are worthwhile to some, given what your Docker environment setup looks like. Some of these tools include dockersh, DockerUI, Shipyard, and Logsprout. These tools, when carefully applied, layer on extra enhancements to help in the overall security of your Docker configurations.

In the next chapter, we will be looking at keeping up on security. With so much going on these days that surrounds the security, it's sometimes tough to know where to look for updated information and be able to apply quick fixes.

You will be learning to help enforce the idea of keeping security in the forefront of your mind and subscribing to things such as e-mail lists that not only include Docker, but also include items that are related to the environments you are running with Linux. Other items are keeping up on following what is going on with regards to items such as GitHub issues that relate to Docker security, following along in the IRC rooms, and watching websites such as the CVE.

8
Keeping up Security

In this chapter, we will be taking a look at keeping up with security as it relates to Docker. By what means you can use to help keep up to date on Docker-related security issues that are out there for the version of the Docker tools you might be running now? How do you stay ahead of any security issues and keep your environments secure even with threats? In this chapter, we will look at multiple ways in which you can keep up on any security issues that arise and the best way to obtain information as quickly as possible. You will cover learning to help enforce the idea of keeping security in the forefront of your mind and subscribing to things such as e-mail lists that not only include Docker, but also include items that are related to the environments you are running with Linux. Other items are keeping up on following what is going on with regards to items such as GitHub issues that relate to Docker security, following along with the **Internet Relay Chat** (**IRC**) rooms, and watching websites such as the CVE.

In this chapter, we will be covering the following topics:

- Keeping up with security
    - E-mail list options
    - GitHub issues
    - IRC rooms
    - CVE websites
- Other areas of interest

Keeping up with security

In this section, we will take a look at the multiple ways that you can obtain or keep up to date about the information related to the security issues that may occur in Docker products. While this isn't a complete list of tools that you can use to keep up on issues, this is a great start and consists of the most commonly used items that are used. These items include e-mail distribution lists, following the GitHub issues for Docker, IRC chat rooms for the multiple Docker products that exist, CVE website(s), and other areas of interest to follow on items that relate to Docker products, such as the Linux kernel vulnerabilities and other items you can use to mitigate the risks.

E-mail list options

Docker operates two mailing lists that users can sign up to be a part of. These mailing lists provide means to both gather information about the issues or projects others are working on and spark your thoughts into doing the same for your environment. You can also use them to help blanket the Docker community with questions or issues that you are running into when using various Docker products or even other products in relation to Docker products.

The two e-mail lists are as follows:

- Docker-dev
- Docker-user

What is the Docker-dev mailing list mostly geared towards? You guessed it, it is geared towards the developers! These are the people who are either interested in developer type roles and what others are developing or are themselves developing code for something that might integrate into various Docker products. This could be something such as creating a web interface around Docker Swarm. This list would be the one you want to post your questions at. The list consists of other developers and possibly even those that work at Docker itself that might be able to help you with any questions or issues that you have.

The other list, the Docker-user list, is geared towards the users of the various Docker products or services and have questions on either how to use the products/services or how they might be able to integrate third-party products with Docker. This might include how to integrate **Heroku** with Docker or use Docker in the cloud. If you are a user of Docker, then this list is the right one for you. You can also contribute to the list as well if you have advanced experience, or something comes across the list that you have experience in, or have dealt with previously.

There is no rule that says you can't be on both. If you want to get the best of both worlds, you can sign up for both and gauge the amount of traffic that comes across each one and then make the decision to only be on one, based on where your interests lie. You also have the option of not joining the lists and just following them on the Google Groups pages for each list.

The Google groups page for the Docker-dev list is `https://groups.google.com/forum/#!forum/docker-dev` and the Google groups page for the Docker-user list is `https://groups.google.com/forum/#!forum/docker-user`.

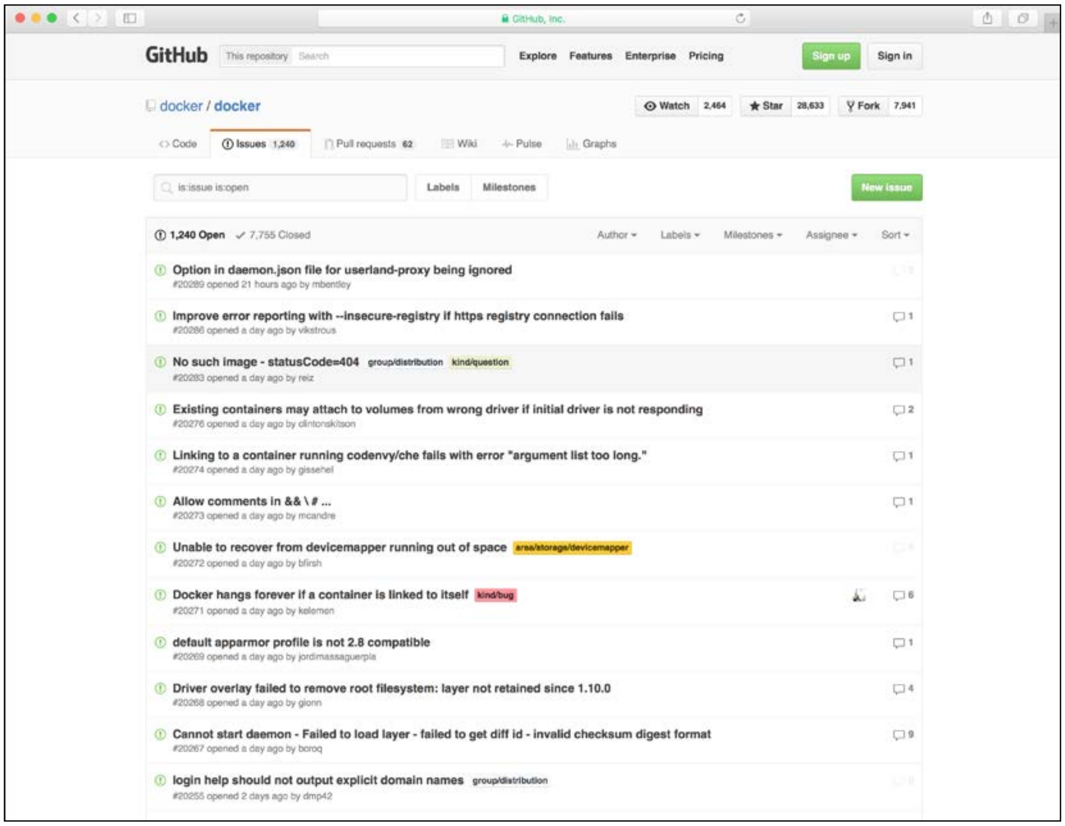
Don't forget that you can also search through these lists to see if your issue or questions might have already been answered. As this book is about security, don't forget that you can use these two mailing lists to discuss items that are security related—whether they be development or user related.

GitHub issues

Another method of keeping up with security-related issues is to follow the GitHub issues. As all the code for the Docker core and other various piece of Docker such as **Machine, Swarm**, **Compose**, and all others are stored on GitHub, it provides an area. What exactly are GitHub issues and why should I care about them is what you are probably asking yourself right now. GitHub Issues is a bug tracker system that GitHub uses. By tracking these issues, you can view the issues that others are experiencing and get ahead of them in your own environment, or it could solve the problem in your environment, knowing that others are having the same issue and it's not just on your end. You can stop pulling what is left of your hair.

As each GitHub repository has its own issues section, we don't need to look at each and every issues section, but I believe it is worthwhile to view one of the repositories issues section so that you know what exactly you are looking at in order to help decipher it all.

The following screenshot (which can be found at `https://github.com/docker/docker/issues`) shows all the current issues that exist with the Docker core software code:



1,240 Open ✓ 7,755 Closed

Option in daemon.json file for userland-proxy being ignored
#20289 opened 21 hours ago by mbentley

Improve error reporting with --insecure-registry if https registry connection fails 1
#20286 opened a day ago by vikstrous

No such image - statusCode=404 group/distribution kind/question 1
#20283 opened a day ago by reiz

Existing containers may attach to volumes from wrong driver if initial driver is not responding 2
#20276 opened a day ago by clintonskitson

Linking to a container running codenvy/che fails with error "argument list too long." 1
#20274 opened a day ago by gissehel

Allow comments in && \ # ... 1
#20273 opened a day ago by mcandre

Unable to recover from devicemapper running out of space area/storage/devicemapper
#20272 opened a day ago by bfirsh

Docker hangs forever if a container is linked to itself kind/bug 6
#20271 opened a day ago by kelemen

default apparmor profile is not 2.8 compatible 1
#20269 opened a day ago by jordimassaguerpla

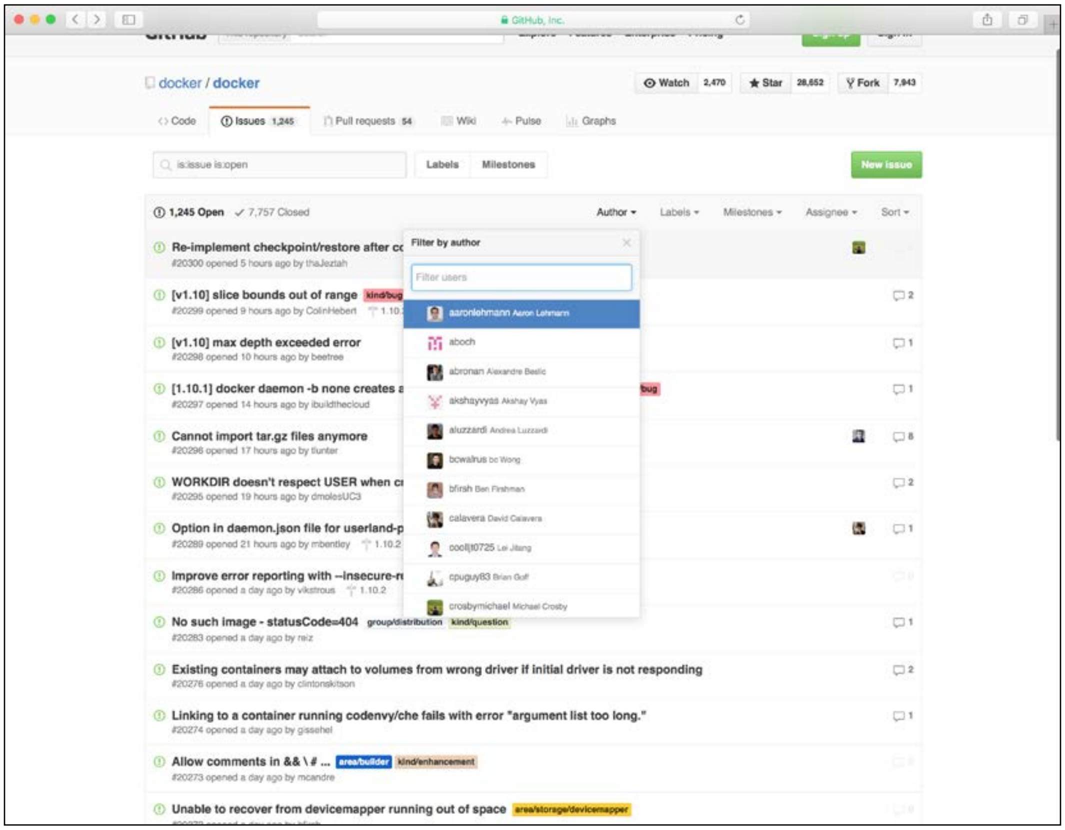
Driver overlay failed to remove root filesystem: layer not retained since 1.10.0 4
#20268 opened a day ago by glonn

Cannot start daemon - Failed to load layer - failed to get diff id - invalid checksum digest format 9
#20267 opened a day ago by boroq

login help should not output explicit domain names group/distribution
#20255 opened 2 days ago by dmp42

From this screen, we can not only see how many issues are open, but also know how many have been closed. These are issues that were once an issue and solutions were derived for them and now they have been closed. The closed ones are here for historic purposes in order to be able to go back in time and see what solution might have been provided to solve an issue.

In the following screenshot, we can filter the issue based on the author, that is, the person who submitted the issue:



Re-implement checkpoint/restore after cc
#20300 opened 5 hours ago by thaJeztah

[v1.10] slice bounds out of range kind/bug
#20299 opened 9 hours ago by ColinHebert 1.10.
2

[v1.10] max depth exceeded error
#20298 opened 10 hours ago by beetree
1

[1.10.1] docker daemon -b none creates a ... bug
#20297 opened 14 hours ago by ibuildthecloud
1

Cannot import tar.gz files anymore
#20296 opened 17 hours ago by tiunter
8

WORKDIR doesn't respect USER when cr
#20295 opened 19 hours ago by dmolesUC3
2

Option in daemon.json file for userland-p
#20289 opened 21 hours ago by mbentley 1.10.2
1

Improve error reporting with --insecure-r
#20286 opened a day ago by vikstrous 1.10.2

No such image - statusCode=404 group/distribution kind/question
#20283 opened a day ago by reiz
1

Existing containers may attach to volumes from wrong driver if initial driver is not responding
#20276 opened a day ago by clintonskitson
2

Linking to a container running codenvy/che fails with error "argument list too long."
#20274 opened a day ago by gissehel
1

Allow comments in && \ # ... area/builder kind/enhancement
#20273 opened a day ago by mcandre

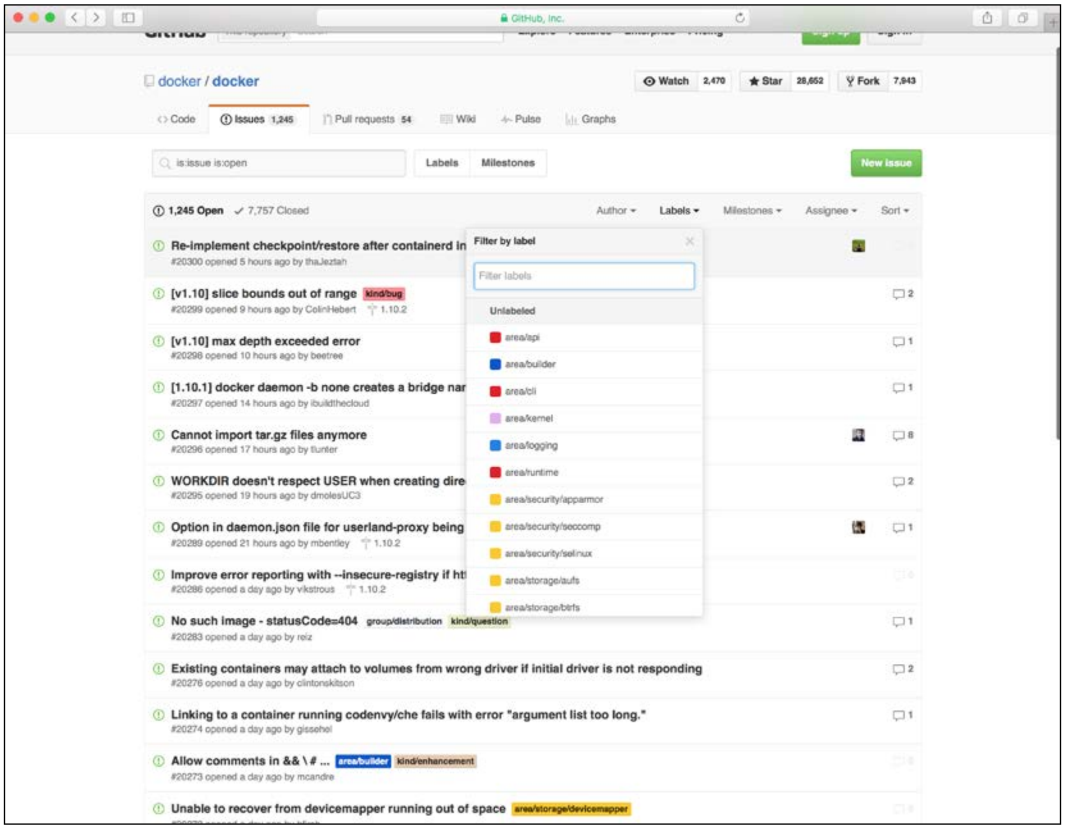
Unable to recover from devicemapper running out of space area/storage/devicemapper

Filter by author

Filter users

- aaronlehmann Aaron Lehmann
- aboch
- abronan Alexandre Beslic
- akshayvyas Akshay Vyas
- aluzzardi Andrea Luzzardi
- bcwalrus bc Wong
- bfirsh Ben Firshman
- calavera David Calavera
- coolljt0725 Lei Jitang
- cpuguy83 Brian Goff
- crosbymichael Michael Crosby

In the following screenshot, we can also filter the issue based on labels and these might include **api**, **kernel**, **apparmor**, **selinux**, **aufs**, and many more:



Re-implement checkpoint/restore after containerd in
#20300 opened 5 hours ago by thaJeztah

[v1.10] slice bounds out of range kind/bug 2
#20299 opened 9 hours ago by ColinHebert 1.10.2

[v1.10] max depth exceeded error 1
#20298 opened 10 hours ago by beetree

[1.10.1] docker daemon -b none creates a bridge nar 1
#20297 opened 14 hours ago by ibuildthecloud

Cannot import tar.gz files anymore 8
#20296 opened 17 hours ago by tlunter

WORKDIR doesn't respect USER when creating dire 2
#20295 opened 19 hours ago by dmolesUC3

Option in daemon.json file for userland-proxy being 1
#20289 opened 21 hours ago by mbentley 1.10.2

Improve error reporting with --insecure-registry if ht
#20286 opened a day ago by vikstrous 1.10.2

No such image - statusCode=404 group/distribution kind/question 1
#20283 opened a day ago by reiz

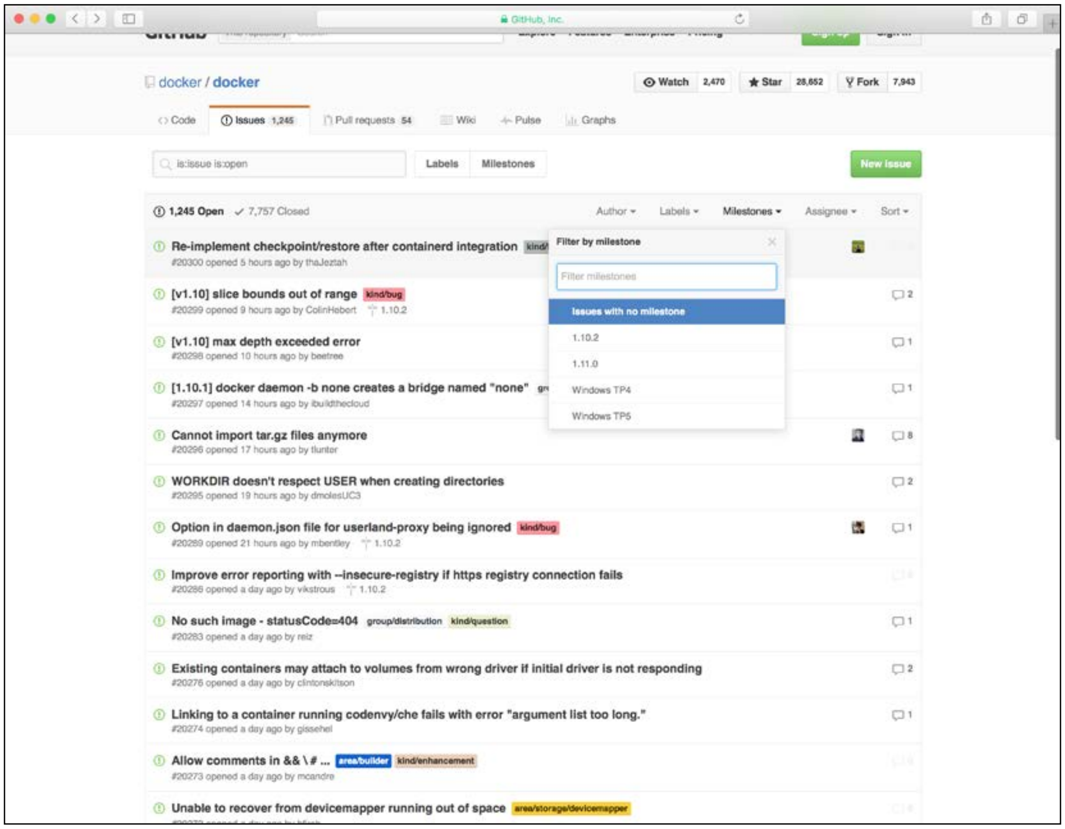
Existing containers may attach to volumes from wrong driver if initial driver is not responding 2
#20276 opened a day ago by clintonskitson

Linking to a container running codenvy/che fails with error "argument list too long." 1
#20274 opened a day ago by gissehel

Allow comments in && \ # ... area/builder kind/enhancement
#20273 opened a day ago by mcandre

Unable to recover from devicemapper running out of space area/storage/devicemapper

In the following screenshot, we see that we can also filter by milestone:



Re-implement checkpoint/restore after containerd integration kind/
#20300 opened 5 hours ago by thaJeztah

[v1.10] slice bounds out of range kind/bug 2
#20299 opened 9 hours ago by ColinHebert 1.10.2

[v1.10] max depth exceeded error 1
#20298 opened 10 hours ago by beetree

[1.10.1] docker daemon -b none creates a bridge named "none" gr 1
#20297 opened 14 hours ago by ibuildthecloud

Cannot import tar.gz files anymore 8
#20296 opened 17 hours ago by tlunter

WORKDIR doesn't respect USER when creating directories 2
#20295 opened 19 hours ago by dmolesUC3

Option in daemon.json file for userland-proxy being ignored kind/bug 1
#20289 opened 21 hours ago by mbentley 1.10.2

Improve error reporting with --insecure-registry if https registry connection fails
#20286 opened a day ago by vikstrous 1.10.2

No such image - statusCode=404 group/distribution kind/question 1
#20283 opened a day ago by reiz

Existing containers may attach to volumes from wrong driver if initial driver is not responding 2
#20276 opened a day ago by clintonskitson

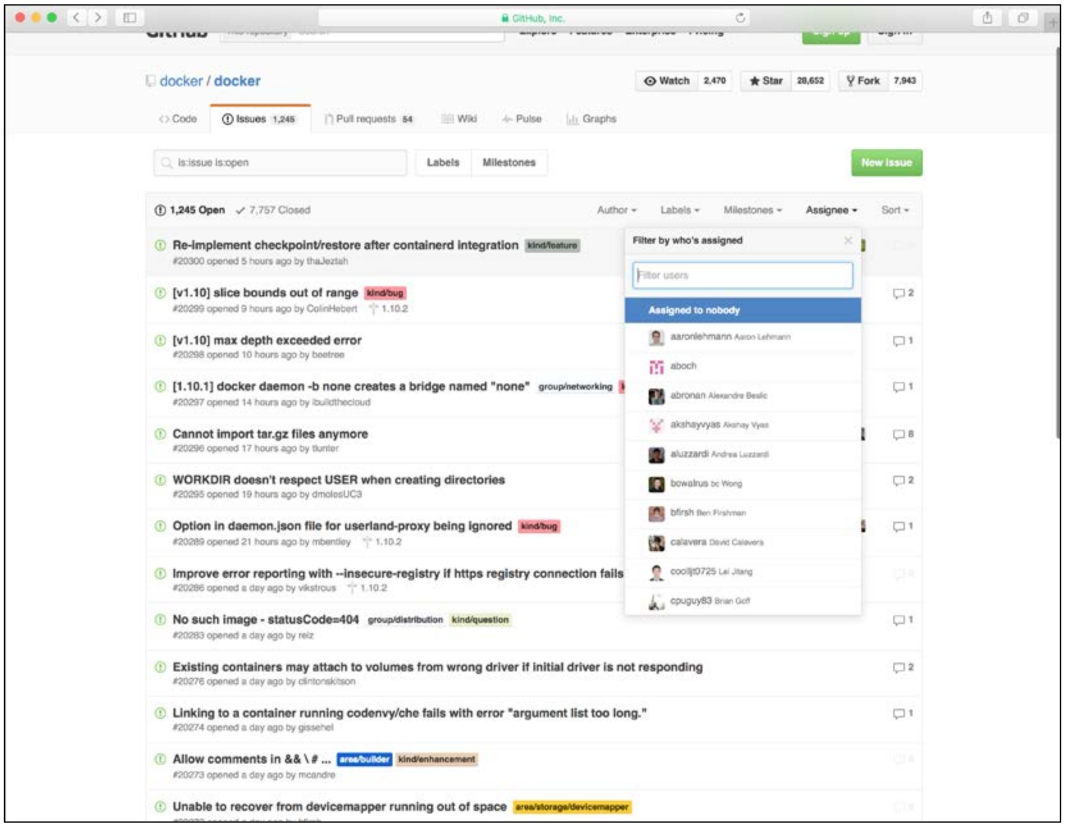
Linking to a container running codenvy/che fails with error "argument list too long." 1
#20274 opened a day ago by gissehel

Allow comments in && \ # ... area/builder kind/enhancement
#20273 opened a day ago by mcandre

Unable to recover from devicemapper running out of space area/storage/devicemapper

Milestones are essentially tags to help sort issues based on fixing an issue for a particular purpose. They can also be used to plan upcoming releases. As we can see here, some of these include **Windows TP4** and **Windows TP5**.

Lastly, we can filter issues based on assignee, that is, the person to whom it is assigned to fix or address the issue, as shown in the following screenshot:



Assigned to nobody
aaronlehmann Aaron Lehmann
aboch
abronan Alexandre Beslic
akshayvyas Akshay Vyas
aluzzardi Andrea Luzzardi
bowalrus bc Wong
bfirsh Ben Firshman
calavera David Calavera
coolljt0725 Lei Jitang
cpuguy83 Brian Goff

Re-implement checkpoint/restore after containerd integration kind/feature
#20300 opened 5 hours ago by thaJeztah

[v1.10] slice bounds out of range kind/bug
#20299 opened 9 hours ago by ColinHebert 1.10.2 — 2

[v1.10] max depth exceeded error
#20298 opened 10 hours ago by beetree — 1

[1.10.1] docker daemon -b none creates a bridge named "none" group/networking
#20297 opened 14 hours ago by ibuildthecloud — 1

Cannot import tar.gz files anymore
#20296 opened 17 hours ago by tlunter — 8

WORKDIR doesn't respect USER when creating directories
#20295 opened 19 hours ago by dmolesUC3 — 2

Option in daemon.json file for userland-proxy being ignored kind/bug
#20289 opened 21 hours ago by mbentley 1.10.2 — 1

Improve error reporting with --insecure-registry if https registry connection fails
#20286 opened a day ago by vikstrous 1.10.2

No such image - statusCode=404 group/distribution kind/question
#20283 opened a day ago by reiz — 1

Existing containers may attach to volumes from wrong driver if initial driver is not responding
#20276 opened a day ago by clintonskitson — 2

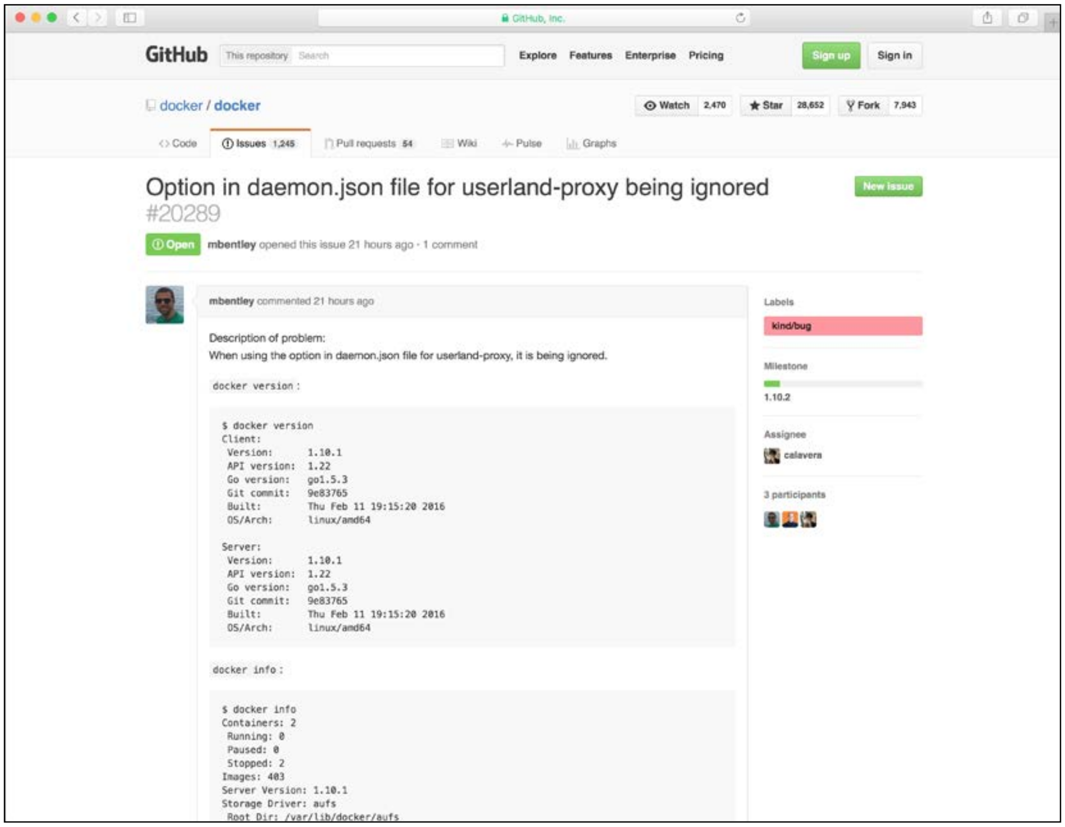
Linking to a container running codenvy/che fails with error "argument list too long."
#20274 opened a day ago by gissehel — 1

Allow comments in && \ # ... area/builder kind/enhancement
#20273 opened a day ago by mcandre

Unable to recover from devicemapper running out of space area/storage/devicemapper

As we can see, there are lot of ways in which we can filter the issues, but what does an issue actually look like and what does it contain? Let's take a look at that in the following section.

In the following screenshot, we can see what an actual issue looks like:



Option in daemon.json file for userland-proxy being ignored #20289

mbentley opened this issue 21 hours ago · 1 comment

mbentley commented 21 hours ago

Description of problem:
When using the option in daemon.json file for userland-proxy, it is being ignored.

`docker version` :

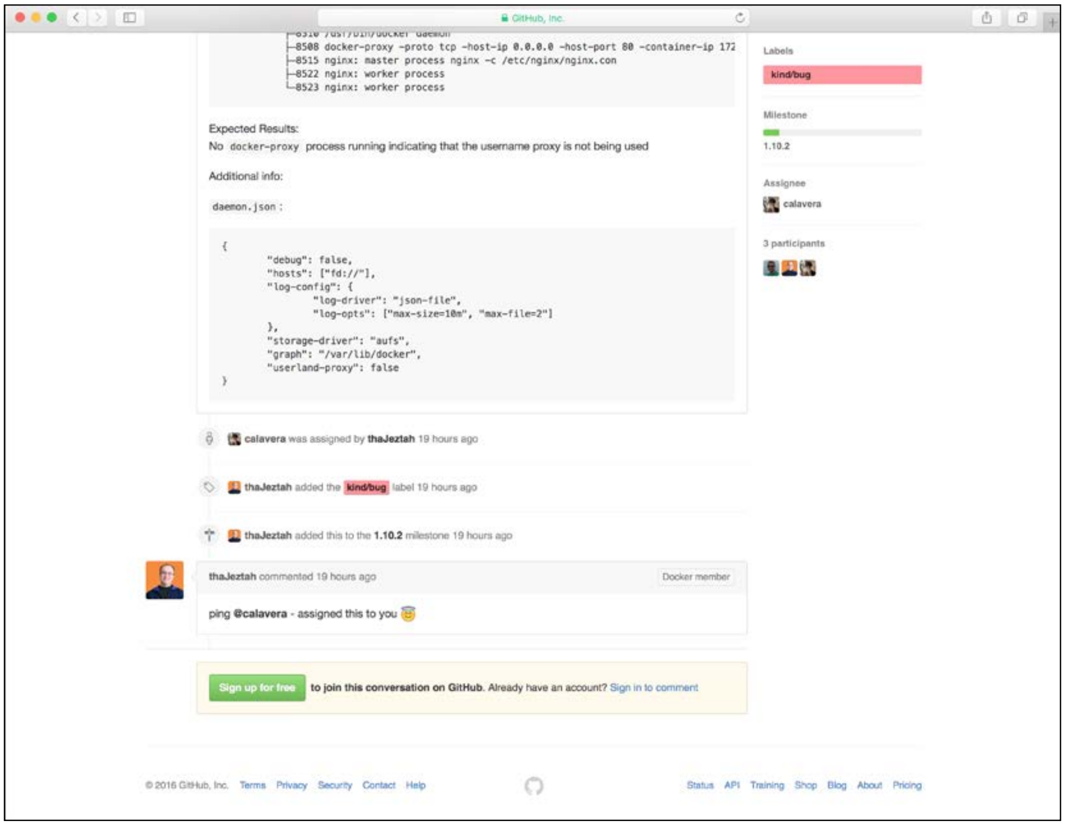
```
$ docker version
Client:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 19:15:20 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 19:15:20 2016
 OS/Arch:      linux/amd64
```

`docker info` :

```
$ docker info
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 403
Server Version: 1.10.1
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
```

Some of the information that we can see is the title of the issue and the unique issue number. We can then see that this particular issue is open, the person who reported the issue, and for how long it's opened. We can then see how many comments are there on the issue and then a large explanation of the issue itself. On the right-hand side, we can see what labels the issue has, what its milestone is, who it is assigned to, and how many participants are involved in the issue. Those involved are people who have commented on the issue in some way.

In the last image, which is at the bottom of the issue from the preceding image, we can see the timeline of the issue, such as who it was assigned to and when, as well as when it was assigned a label and any additional comments.



```
├─8510 /usr/bin/docker daemon
├─8508 docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 80 -container-ip 172
├─8515 nginx: master process nginx -c /etc/nginx/nginx.con
├─8522 nginx: worker process
└─8523 nginx: worker process
```

Expected Results:
No `docker-proxy` process running indicating that the username proxy is not being used

Additional info:

`daemon.json` :

```
{
        "debug": false,
        "hosts": ["fd://"],
        "log-config": {
                "log-driver": "json-file",
                "log-opts": ["max-size=10m", "max-file=2"]
        },
        "storage-driver": "aufs",
        "graph": "/var/lib/docker",
        "userland-proxy": false
}
```

Labels: kind/bug

Milestone: 1.10.2

Assignee: calavera

3 participants

calavera was assigned by **thaJeztah** 19 hours ago

thaJeztah added the kind/bug label 19 hours ago

thaJeztah added this to the **1.10.2** milestone 19 hours ago

thaJeztah commented 19 hours ago — Docker member

ping **@calavera** - assigned this to you 😇

to join this conversation on GitHub. Already have an account? Sign in to comment

© 2016 GitHub, Inc. Terms Privacy Security Contact Help
Status API Training Shop Blog About Pricing

IRC rooms

The first thing to understand is what exactly IRC is. If you think back to the older days, we probably all had some form of IRC rooms when we had AOL and had chat rooms that you could join based on your location or topic. IRC operates in the same way where there is a server that clients, such as yourself, connect to. These rooms are typically based on a topic, product, or service that people have in common that can come together to discuss. You can chat as a group but also utilize private chats with others in the same room or channel as you.

Docker utilizes IRC for discussion about its products. This allows not only end users of the products to engage in discussion, but also in the case of Docker, most of those who actually work for Docker and on these products tend to be in these rooms on a daily basis and will engage with you about issues you might be seeing or questions you have.

With IRC, there are multiple servers that you can use to connect to the hosted channels. Docker uses the `http://freenode.net` server (it is the server you would use if you were to use a third-party client to connect to IRC; however, you can also use `http://webchat.freenode.net`) and then all their channels for their products are things such as **#docker**, **#docker-dev**, **#docker-swarm**, **#docker-compose**, and **#docker-machine**. All channels start with the pound sign (#), followed by the channel name. Within these channels, there are discussion for each product. Beyond these channels, there are other channels where you can discuss Docker-related topics. In the previous chapter, we discussed the Shipyard project, which allows you to have a GUI interface that overlays on top of your Docker Swarm environment. If you had questions about this particular product, you could join the channel for that product, which is **#shipyard**. There are other channels you can join as well and more created daily. To get a list of channels, you will need to connect to your IRC client and issue a command. Follow the given link to find out how to do this:

`http://irc.netsplit.de/channels/?net=freenode`

Chat archives are also kept for each channel, therefore, you can search through them as well to find out whether discussions are happening around a question or issue that you may be experiencing. For example, if you wanted to see the logs of the **#docker** channel, you could find them here:

`https://botbot.me/freenode/docker/`

You can search for other channel archives on the following website:

`https://botbot.me`

CVE websites

In *Chapter 5, Monitoring and Reporting Docker Security Incidents*, we covered CVEs and Docker CVEs. A few things to remember about them are listed in the following:

- CVEs can be found at `https://cve.mitre.org/index.html`
- Docker-related ones can be found at `https://www.docker.com/docker-cve-database`
- To search for CVE's use the following URL: `https://cve.mitre.org/index.html`

- If you were to open this CVE from the preceding link, you will see that it gathers some information as shown in the following:
    - CVE ID
    - Description
    - References
    - Date entry created
    - Phase
    - Votes
    - Comments
    - Proposed

Other areas of interest

There are some areas of interest that you should keep in mind with regards to security. The Linux kernel, as we have talked about a lot during this book, is the key part of the Docker ecosystem. For this reason, it's very important to keep the kernel as up to date as possible. With regards to updates, it is also important to keep the Docker products you are using up to date too. Most updates include security updates, and for this reason, they should be updated when new product updates are released.

Twitter has become the social hotspot when you are looking to promote your products and Docker does the same. There are a few accounts that Docker operates for different purposes and they are listed in the following. Depending on what piece of Docker you are using, it would be wise to follow one or all of them, as shown in the following list:

- **@docker**
- **@dockerstatus**
- **@dockerswarm**
- **@dockermachine**

Twitter also utilizes hashtags that group the tweets together, based on their hashtags. For Docker, it's the same and they use the #docker hashtag, which you can search for on Twitter to gather tweets that all talk about Docker.

The last thing we want to cover is Stack Overflow. Stack Overflow is a question and answer website and uses votes to promote the answers that are provided to help you get the best answer in the quickest manner. Stack Overflow utilizes a method similar to Twitter with tagging questions so that you can search for all the questions on a particular topic. The following is the link that you can use to gather all the Docker questions into one search:

`http://stackoverflow.com/questions/tagged/docker`

When you visit the URL, you will see a list of questions as well as how many votes each question has, number of answers, number of views, and a green check mark on some of them. The checked answers are the answers that the person who asked them mark as accepted, meaning that it's the best answer. Some of the people who monitor Docker questions are those that work for Docker, doing the work behind the scenes and providing the best answers, therefore, it's a great place to pose any questions that you might have.

Summary

In this chapter, we looked at how to keep up with security-related issues that not only pertain to Docker products that you may be running now or in the near future, but they also pertain to security issues such as kernel issues. As Docker relies on the kernel for all Docker containers on a Docker host, the kernel is very important. We looked at multiple mailing lists that you can sign up for, getting notifications in this manner. Joining IRC chat rooms and following GitHub issues for anything security-related or anything that isn't currently working might affect your environments. It is very important to always keep security in the front of your mind when deploying anything and while the Docker is inherently secure, there are always people out there that will take advantage of any given vulnerability, therefore, keep all of your environments safe and as up to date as possible.

Index