14 Docker and security

This chapter covers

- The security Docker offers out of the box
- What Docker has done to help make it more secure
- What other parties are doing about it
- What other steps can be taken to ameliorate security concerns
- How to manage user Docker permissions with an aPaaS, potentially in a multi-tenant environment

As Docker makes clear in its documentation, access to the Docker API implies access to root privileges, which is why Docker must often be run with sudo, or the user must be added to a user group (which might be called "docker", or "dockerroot") that allows access to the Docker API.

In this chapter we're going to look at the issue of security in Docker.

14.1 Docker access and what it means

You may be wondering what sort of damage a user can do if they can run Docker. As a simple example, the following command (don't run it!) would delete all the binaries in /sbin on your host machine (if you took out the bogus `--donotrunme` flag):

```
docker run --donotrunme -v /sbin:/sbin busybox rm -rf /sbin
```

It's worth pointing out that this is true even if you're a non-root user.

The following command will show you the contents of the secure shadow password file from the host system:

```
docker run -v /etc/shadow:/etc/shadow busybox cat /etc/shadow
```

Docker's insecurity is often misunderstood, partly due to a misunderstanding of the benefits of namespaces in the kernel. Linux namespaces provide isolation from other parts of the system, but the level of isolation you have in Docker is at your discretion (as seen in the preceding `docker run` examples). Furthermore, not all parts of the Linux OS have the ability to be namespaced. Devices and kernel modules are two examples of core Linux features that aren't namespaced.

> **TIP** Linux namespaces were developed to allow processes to have a different view of the system than other processes have. For example, *process namespacing* means that containers can only see processes associated with that container—other processes running on the same host are effectively invisible to them. *Network namespacing* means that containers appear to have their own network stack available to them. Namespaces have been part of the Linux kernel for a number of years.

Also, because you have the ability to interact with the kernel as root from within the container through syscalls, any kernel vulnerability could be exploited by root within the Docker container. Of course, VMs have a similar attack possible through access to the hypervisor, because hypervisors also have security vulnerabilities reported against them.

Another way to understand the risks here is to think of running a Docker container as being no different (from a security perspective) from being able to install any package via a package manager. Your requirement for security when running Docker containers should be the same as for installing packages. If you have Docker, you can install software as root. This is partly why some argue that Docker is best understood as a software packaging system.

> **TIP** Work is underway to remove this risk through user namespacing, which maps root in the container to a non-privileged user on the host.

14.1.1 Do you care?

Given that access to the Docker API is equivalent to root access, the next question is "do you care?" Although this might seem an odd line to take, security is all about trust, and if you trust your users to install software in the environment in which they operate, there should be no barrier to them running Docker containers there. Security difficulties primarily arise when considering multi-tenant environments. Because the root user inside your container is in key respects the same as root outside your container, having lots of different users being root on your system is potentially worrying.

TIP A multi-tenant environment is one in which many different users share the same resources. For example, two teams might share the same server with two different VMs. Multi-tenancy offers cost savings through sharing hardware rather than provisioning hardware for specific applications. But it can bring other challenges related to service reliability and security isolation that can offset the cost savings.

Some organizations take the approach of running Docker on a dedicated VM for each user. The VM can be used for security, operational, or resource isolation. Within the VM trust boundary, users run Docker containers for the performance and operational benefits they bring. This is the approach taken by Google Compute Engine, which places a VM between the user's container and the underlying infrastructure for an added level of security and some operational benefits. Google has more than a little compute resources at their disposal, so they don't mind the overhead of doing this.

14.2 Security measures in Docker

Various measures have already been taken by the Docker maintainers to reduce the security risks of running containers. For example,

- Certain core mount points (such as /proc and /sys) are now mounted as read-only.
- Default Linux capabilities have been reduced.
- Support for third-party security systems like SELinux and AppArmor now exists.

In this section, we'll look more deeply at these and at some of the measures you can take to reduce the risks of running containers on your system.

TECHNIQUE 93 Constraining capabilities

As we've already mentioned, the root user on the container is the same user as root on the host. But not all root users are created equal. Linux provides you with the ability to assign more fine-grained privileges to the root user within a process.

These fine-grained privileges are called *capabilities*, and they allow you to limit the damage a user can do, even if they're root. This technique shows you how to manipulate these capabilities when running Docker containers, particularly if you don't fully trust their contents.

PROBLEM

You want to reduce the ability of containers to perform damaging actions on your host machine.

SOLUTION

Drop the capabilities available to the container by using the `--drop-cap` flag.

THE UNIX TRUST MODEL

To understand what "dropping capabilities" means and does, a little bit of background is required. When the Unix system was designed, the trust model wasn't sophisticated. You had admins who were trusted (root users) and users who weren't. Root users could do anything, whereas standard users could only affect their own files. Because the system was typically used in a university lab and was small, this model made sense.

As the Unix model grew and the internet arrived, this model made less and less sense. Programs like web servers needed root permissions to serve content on port 80, but they were also acting effectively as proxies for running commands on the host. Standard patterns were established to handle this, such as binding to port 80 and dropping the effective user ID to a non-root user. Users performing all sorts of roles, from sysadmins to database administrators through to application support engineers and developers, could all potentially need fine-grained access to different resources on a system. Unix groups alleviated this to some degree, but modeling these privilege requirements—as any systems admin will tell you—is a nontrivial problem.

LINUX CAPABILITIES

In an attempt to support a more fine-grained approach to privileged user management, the Linux kernel engineers developed *capabilities*. This was an attempt to break down the monolithic root privilege into slices of functionality that could be granted discretely. You can read about them in more detail by running `man 7 capabilities` (assuming you have the man page installed).

Docker has helpfully switched off certain capabilities by default. This means that even if you have root in the container, there are things you won't be able to do. For example, the `CAP_NET_ADMIN` capability, which allows you to affect the network stack of the host, is disabled by default.

Table 14.1 lists Linux capabilities, gives a brief description of what they allow, and indicates whether they're permitted by default in Docker containers.

Table 14.1 Linux capabilities in Docker containers

Capability	Description	Switched on?
`CHOWN`	Make ownership changes to any files	Y
`DAC_OVERRIDE`	Override read, write, and execution checks	Y
`FSETID`	Don't clear suid and guid bits when modifying files	Y
`FOWNER`	Override ownership checks when saving files	Y
`KILL`	Bypass permission checks on signals	Y
`MKNOD`	Make special files with `mknod`	Y

Table 14.1 Linux capabilities in Docker containers *(continued)*

Capability	Description	Switched on?
`NET_RAW`	Use raw and packet sockets, and bind to ports for transparent proxying	Y
`SETGID`	Make changes to group ownership of processes	Y
`SETUID`	Make changes to user ownership of processes	Y
`SETFCAP`	Set file capabilities	Y
`SETPCAP`	If file capabilities aren't supported, apply capability limits to and from other processes	Y
`NET_BIND_SERVICE`	Bind sockets to ports under 1024	Y
`SYS_CHROOT`	Use `chroot`	Y
`AUDIT_WRITE`	Write to kernel logs	Y
`AUDIT_CONTROL`	Enable/disable kernel logging	N
`BLOCK_SUSPEND`	Employ features that block the ability of the system to suspend	N
`DAC_READ_SEARCH`	Bypass file permission checks on reading files and directories	N
`IPC_LOCK`	Lock memory	N
`IPC_OWNER`	Bypass permissions on interprocess communication objects	N
`LEASE`	Establish leases (watches on attempts to open or truncate) on ordinary files	N
`LINUX_IMMUTABLE`	Set the `FS_APPEND_FL` and `FS_IMMUTABLE_FL` i-node flags	N
`MAC_ADMIN`	Override mandatory access control (related to the Smack Linux Security Module (SLM))	N
`MAC_OVERRIDE`	Mandatory access control changes (related to SLM)	N
`NET_ADMIN`	Various network-related operations, including IP firewall changes and interface configuration	N
`NET_BROADCAST`	Unused	N
`SYS_ADMIN`	A range of administrative functions—see `man capabilities` for more information	N
`SYS_BOOT`	Rebooting	N
`SYS_MODULE`	Load/unload kernel modules	N
`SYS_NICE`	Manipulate nice priority of processes	N
`SYS_PACCT`	Turn on or off process accounting	N
`SYS_PTRACE`	Trace processes' system calls and other process manipulation capabilities	N

Table 14.1 Linux capabilities in Docker containers *(continued)*

Capability	Description	Switched on?
SYS_RAWIO	Perform I/O on various core parts of the system, such as memory and SCSI device commands	N
SYS_RESOURCE	Control and override various resource limits	N
SYS_TIME	Set the system clock	N
SYS_TTY_CONFIG	Privileged operations on virtual terminals	N

NOTE If you aren't using Docker's default container engine (libcontainer), these capabilities may be different on your installation. If you have a sysadmin and want to be sure, ask them.

Unfortunately the kernel maintainers only allocated 32 capabilities within the system, so capabilities have grown in scope as more and more fine-grained root privileges have been carved out of the kernel. Most notably, the vaguely named `CAP_SYS_ADMIN` capability covers actions as varied as changing the host's domain name to exceeding the system-wide limit on the number of open files.

One extreme approach is to remove all the capabilities that are switched on in Docker by default from the container, and see what stops working. Here we start up a bash shell with the capabilities that are enabled by default removed:

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \
--cap-drop=FSETID --cap-drop=FOWNER --cap-drop=KILL --cap-drop=MKNOD \
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

If you run your application from this shell, you can see where it fails to work as desired, and re-add the required capabilities. For example, you may need the capability to change file ownership, so you'll need to lose the dropping of the `FOWNER` capability in the preceding code to run your application:

```
$ docker run -ti --cap-drop=CHOWN --cap-drop=DAC_OVERRIDE \
--cap-drop=FSETID  --cap-drop=KILL --cap-drop=MKNOD \
--cap-drop=NET_RAW --cap-drop=SETGID --cap-drop=SETUID \
--cap-drop=SETFCAP --cap-drop=SETPCAP --cap-drop=NET_BIND_SERVICE \
--cap-drop=SYS_CHROOT --cap-drop=AUDIT_WRITE debian /bin/bash
```

TIP If you want to enable or disable all capabilities, you can use `all` instead of a specific capability, such as `docker run -ti --cap-drop=all ubuntu bash`.

DISCUSSION

If you run a few basic commands in the bash shell with all capabilities disabled, you'll see that it's quite usable. Your mileage may vary when running more complex applications, though.

WARNING It's worth making clear that many of these capabilities relate to the root capabilities to affect *other* users' objects on the system, not root's own objects. A root user could still chown root's files on the host if they were host in the container and had access to the host's files through a volume mount, for example. Therefore, it's still worth ensuring that applications drop to a non-root user as soon as possible to protect the system, even if all these capabilities are switched off.

This ability to fine-tune the capabilities of your container means that using the `--privileged` flag to `docker run` should be unnecessary. Processes that require capabilities will be auditable and under the control of the administrator of the host.

TECHNIQUE 94 A "bad" Docker image to scan

One issue quickly recognized in the Docker ecosystem was that of vulnerabilities—if you have an unchanging image, you also won't get any security fixes. This may not be a problem if you're following the Docker best practices of image minimalism, but it can be hard to tell.

Image scanners were created as a solution to this problem—a way to identify issues with an image—but that still leaves open the question of how to evaluate them.

PROBLEM

You want to determine how effective an image scanner is.

SOLUTION

Create a "known-bad" image to test your scanners on.

We were faced with this problem while at work. Plenty of Docker image scanners exist (such as Clair), but commercial offerings claim to go deeper into the image to determine any potential issues lurking within it.

But no image existed that contained known and documented vulnerabilities that we could use to test the efficacy of these scanners. Hardly surprising, as most images don't advertise their own insecurity!

We therefore invented a known bad image. The image is available to download:

```
$ docker pull imiell/bad-dockerfile
```

The principle is simple: create a Dockerfile to build an image riddled with documented vulnerabilities, and point that image at your candidate scanner.

The latest version of the Dockerfile is available at https://github.com/ianmiell/bad-dockerfile. It's still in flux, so it's not printed here. The form of it is, however, quite simple:

Various RUN/COPY/ADD commands install software to the image that are known to be vulnerable.

The reference bad-dockerfile repository uses a centos image, but you might want to replace this with one closer to your base image.

```
FROM <base image>
 RUN <install 'bad' software>
 COPY <copy 'bad' software in>
 [...]
CMD echo 'Vulnerable image' && /bin/false
```

The CMD directive for the image tries its best never to allow itself to be run, for obvious reasons.

The image contains a spectrum of vulnerabilities designed to exercise a scanner to its limits.

At its simplest, the image installs software known to be vulnerable using the package manager. Within each category, the Docker image attempts to contain vulnerabilities of varying degrees of severity.

More sophisticated placement of vulnerabilities is performed by (for example) `COPY`ing vulnerable JavaScript, using language-specific package managers (such as npm for JavaScript, gem for Ruby, and pip for Python) to install vulnerable code, and even compiling a specific version of bash (one with the infamous Shellshock bug) and placing it in an unexpected location to avoid many scanning techniques.

DISCUSSION

You might think that the best scanning solution is one that catches the most CVEs. But this isn't necessarily the case. Obviously, it's good if a scanner can spot that an image has a vulnerability within it. Beyond this, however, scanning for vulnerabilities can become more of an art than a science.

> **TIP** A Common Vulnerability Exposure (CVE) is an identifier for a specific vulnerability discovered in generally available software. An example of a CVE might be CVE-2001-0067, where the first four-digit number is the year of discovery, and the second is the count of the identified vulnerability for that year.

For example, a vulnerability might be very severe (such as gaining root on your host server), but extremely difficult to exploit (such as requiring the resources of a nation-state). You (or the organization you're responsible for) might be less worried about this than about a vulnerability that's less severe, but easy to exploit. If, for example, there's a DoS attack on your system, there's no risk of data leakage or infiltration, but you could be put out of business by it, so you'd be more concerned about patching that than some obscure cipher attack requiring tens of thousands of dollars' worth of computing power.

> **WHAT IS A DOS ATTACK?** DoS stands for "denial of service." This means an attack that results in a reduction in the ability of your system to cope with demand. A denial of service attack could overwhelm your web server to the point where it can't respond to legitimate users.

It's also worth considering whether the vulnerability is actually available on the running container. An old version of the Apache web server may exist on the image, but if it's never actually run by the container, the vulnerability is effectively ignorable. This happens often. Package managers regularly bring in dependencies that aren't really needed just because it makes managing dependencies simpler.

If security is a big concern, this can be another reason to have small images (see chapter 7)—even if a piece of software is unused, it can still show up on a security scan, wasting time as your organization tries to work out whether it needs patching.

This technique hopefully gave you food for thought when considering which scanner is right for you. As always, it's a balance between cost, what you need, and how much you're willing to work to get the right solution.

14.3 Securing access to Docker

The best way to prevent insecure use of a Docker daemon is to prevent any use at all.

You probably first encountered restricted access when you installed Docker and needed to use `sudo` to run Docker itself. Technique 41 describes how to selectively permit users on the local machine to use Docker without this restriction.

But this doesn't help you if you have users connecting to a Docker daemon from another machine. We'll look at a couple of ways to provide a bit more security in those situations.

TECHNIQUE 95 HTTP auth on your Docker instance

In technique 1 you saw how to open up access to your daemon to the network, and in technique 4 you saw how to snoop the Docker API using socat.

This technique combines those two: you'll be able to access your daemon remotely and view the responses. Access is restricted to those with a username/password combination, so it's slightly safer. As a bonus, you don't have to restart your Docker daemon to achieve it—start up a container daemon.

PROBLEM

You'd like basic authentication with network access available on your Docker daemon.

SOLUTION

Use HTTP authentication to share your Docker daemon with others temporarily.

Figure 14.1 lays out the final architecture of this technique.

> **NOTE** This discussion assumes your Docker daemon is using Docker's default Unix socket method of access in /var/run/docker.sock.

The code in this technique is available at https://github.com/docker-in-practice/docker-authenticate. The following listing shows the contents of the Dockerfile in this repository, used to create the image for this technique.

Listing 14.1 Dockerfile

```
FROM debian
RUN apt-get update && apt-get install -y \
nginx apache2-utils
 RUN htpasswd -c /etc/nginx/.htpasswd username
 RUN htpasswd -b /etc/nginx/.htpasswd username password
 RUN sed -i 's/user .*;/user root;/' \
/etc/nginx/nginx.conf
 ADD etc/nginx/sites-enabled/docker \
/etc/nginx/sites-enabled/docker
 CMD service nginx start && sleep infinity
```

Ensures the required software is updated and installed

Creates a password file for the user called username

Sets the password for the user called username to "password"

Nginx will need to run as root to access the Docker Unix socket, so you replace the user line with the "root" user details.

Copies in Docker's nginx site file (listing 14.8)

By default, starts the nginx service and waits indefinitely

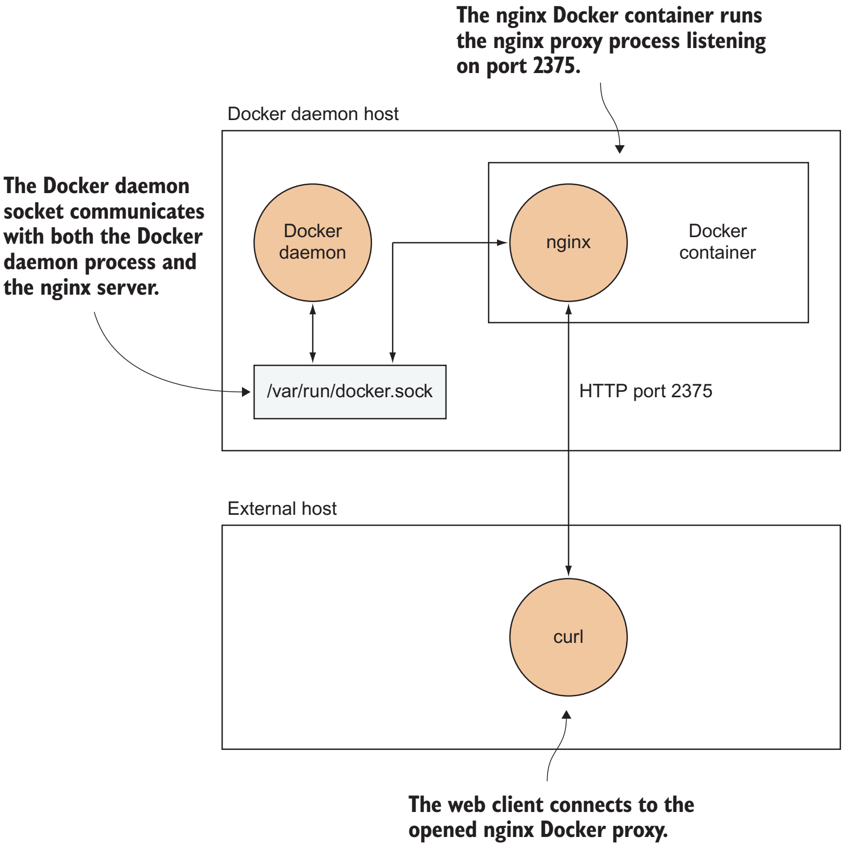

Figure 14.1 The architecture of a Docker daemon with basic authentication

The .htpasswd file set up with the `htpasswd` command contains the credentials to be checked before allowing (or rejecting) access to the Docker socket. If you're building this image yourself, you'll probably want to alter `username` and `password` in those two steps to customize the credentials with access to the Docker socket.

> **WARNING** Be careful not to share this image, as it will contain the password you've set!

The nginx site file for Docker is shown in the following listing.

Listing 14.2 /etc/nginx/sites-enabled/docker

```
upstream docker {
  server unix:/var/run/docker.sock;
 }

server {
  listen 2375 default_server;
```

Defines the docker location in nginx as pointing to Docker's domain socket

Listens on port 2375 (the standard Docker port)

Proxies these requests to and from the docker location defined earlier

Defines the password file to use

Restricts access by password

```
location / {
 proxy_pass http://docker;
  auth_basic_user_file /etc/nginx/.htpasswd;
  auth_basic "Access restricted";
 }
}
```

Now run the image as a daemon container, mapping the required resources from the host machine:

```
$ docker run -d --name docker-authenticate -p 2375:2375 \
  -v /var/run:/var/run dockerinpractice/docker-authenticate
```

This will run the container in the background with the name `docker-authenticate` so you can refer to it later. Port 2375 of the container is exposed on the host, and the container is given access to the Docker daemon by mounting the default directory containing the Docker socket as a volume. If you're using a custom-built image with your own username and password, you'll need to replace the image name here with your own.

The web service will now be up and running. If you `curl` the service with the username and password you set, you should see an API response:

The JSON response from the Docker daemon

Puts the username: password in the URL to curl, and the address after the @ sign. This request is to the /info endpoint of the Docker daemon's API.

```
$ curl http://username:password@localhost:2375/info
 {"Containers":115,"Debug":0, >
 "DockerRootDir":"/var/lib/docker","Driver":"aufs", >
"DriverStatus":[["Root Dir","/var/lib/docker/aufs"], >
["Backing Filesystem","extfs"],["Dirs","1033"]], >
"ExecutionDriver":"native-0.2", >
"ID":"QSCJ:NLPA:CRS7:WCOI:K23J:6Y2V:G35M:BF55:OA2W:MV3E:RG47:DG23", >
"IPv4Forwarding":1,"Images":792, >
"IndexServerAddress":"https://index.docker.io/v1/", >
"InitPath":"/usr/bin/docker","InitSha1":"", >
"KernelVersion":"3.13.0-45-generic", >
"Labels":null,"MemTotal":5939630080,"MemoryLimit":1, >
"NCPU":4,"NEventsListener":0,"NFd":31,"NGoroutines":30, >
"Name":"rothko","OperatingSystem":"Ubuntu 14.04.2 LTS", >
"RegistryConfig":{"IndexConfigs":{"docker.io": >
{"Mirrors":null,"Name":"docker.io", >
"Official":true,"Secure":true}}, >
"InsecureRegistryCIDRs":["127.0.0.0/8"]},"SwapLimit":0}
```

When you're done, remove the container with this command:

```
$ docker rm -f docker-authenticate
```

Access is now revoked.

USING THE DOCKER COMMAND?

Readers may be wondering whether other users will be able to connect with the docker command—for example, with something like this:

```
docker -H tcp://username:password@localhost:2375 ps
```

At the time of writing, authentication functionality isn't built into Docker itself. But we have created an image that will handle the authentication and allow Docker to connect to a daemon. Simply use the image as follows:

```
$ docker run -d --name docker-authenticate-client \
    -p 127.0.0.1:12375:12375 \
    dockerinpractice/docker-authenticate-client \
    192.168.1.74:2375 username:password
```

Exposes a port to connect a Docker daemon to, but only for connections from the local machine

Runs the client container in the background and gives it a name

The image we've made to allow authenticated connections with Docker

The two arguments to the image: a specification of where the other end of the authenticated connection should be, and the username and password (both of these should be replaced as appropriate for your setup)

Note that `localhost` or `127.0.0.1` won't work for specifying the other end of the authenticated connection—if you want to try it out on one host, you must use `ip addr` to identify an external IP address for your machine.

You can now use the authenticated connection with the following command:

```
docker -H localhost:12375 ps
```

Be aware that interactive Docker commands (`run` and `exec` with the `-i` argument) won't work over this connection due to some implementation limitations.

DISCUSSION

In this technique we showed you how to set up basic authentication for your Docker server in a trusted network. In the next technique we'll look at encrypting the traffic so snoopers can't take a peek at what you're up to, or even inject evil data or code.

> **WARNING** This technique gives you a basic level of *authentication*, but it doesn't give you a serious level of *security* (in particular, someone able to listen to your network traffic could intercept your username and password). Setting up a server secured with TLS is rather more involved and is covered in the next technique.

TECHNIQUE 96 Securing your Docker API

In this technique we'll show how you can open up your Docker server to others over a TCP port while at the same time ensuring that only trusted clients can connect. This is achieved by creating a secret key that only trusted hosts will be given. As long as that trusted key remains a secret between the server and client machines, the Docker server should remain secure.

PROBLEM

You want your Docker API to be served securely over a port.

SOLUTION

Create a self-signed certificate, and run the Docker daemon with the `--tls-verify` flag.

This method of security depends on so-called *key files* being created on the server. These files are created using special tools that ensure they're difficult to copy if you don't have the *server key*. Figure 14.2 gives an overview of this how this works.

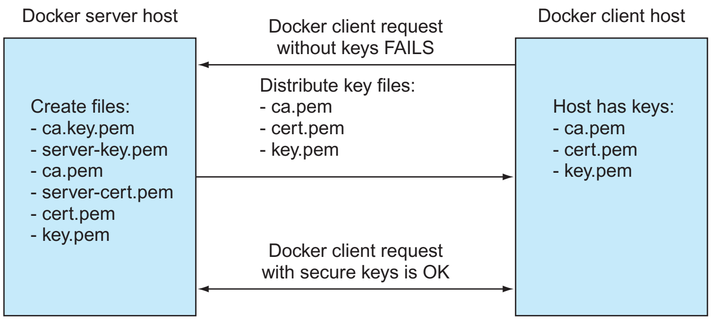

Figure 14.2 Key setup and distribution

> **TIP** The *server key* is a file that holds a secret number known only to the server, and which is required to read messages encrypted with the secret key files given out by the owner of the server (the so-called *client keys*). Once the keys have been created and distributed, they can be used to make the connection between client and server secure.

SETTING UP THE DOCKER SERVER CERTIFICATE

First you create the certificates and keys.

Generating keys requires the OpenSSL package, and you can check whether it's installed by running `openssl` in a terminal. If it's not installed, you'll need to install it before generating the certificates and keys with the following code.

Listing 14.3 Creating certificates and keys with OpenSSL

```
$ sudo su
 $ read -s PASSWORD
 $ read SERVER
 $ mkdir -p /etc/docker
 $ cd /etc/docker
 $ openssl genrsa -aes256 -passout pass:$PASSWORD \
-out ca-key.pem 2048
 $ openssl req -new -x509 -days 365 -key ca-key.pem -passin pass:$PASSWORD \
-sha256 -out ca.pem -subj "/C=NL/ST=./L=./O=./CN=$SERVER"
```

Ensure you are root.

Type in your certificate password and the server name you'll use to connect to the Docker server.

Create the docker configuration directory if it doesn't exist, and move into it.

Generate certificate authority (CA) .pem file with 2048-bit security.

Sign the CA key with your password and address for a period of one year.

Generate a client key with 2048-bit security.

Generate a server key with 2048-bit security.

Process the server key with the name of your host.

Sign the key with your password for a period of one year.

```
$ openssl genrsa -out server-key.pem 2048
$ openssl req -subj "/CN=$SERVER" -new -key server-key.pem \
-out server.csr
$ openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem
-passin "pass:$PASSWORD" -CAcreateserial \
-out server-cert.pem
$ openssl genrsa -out key.pem 2048
$ openssl req -subj '/CN=client' -new -key key.pem\
-out client.csr
$ sh -c 'echo "extendedKeyUsage = clientAuth" > extfile.cnf'
$ openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem \
-passin "pass:$PASSWORD" -CAcreateserial -out cert.pem \
-extfile extfile.cnf
$ chmod 0400 ca-key.pem key.pem server-key.pem
$ chmod 0444 ca.pem server-cert.pem cert.pem
$ rm client.csr server.csr
```

Process the key as a client key.

Sign the key with your password for a period of one year.

Change the permissions to read-only by root for the server files.

Remove leftover files.

Change the permissions of the client files to read-only by everyone.

TIP A script called CA.pl may be installed on your system that makes this process simpler. Here we've exposed the raw `openssl` commands because they're more instructive.

SETTING UP THE DOCKER SERVER

Next you need to set the Docker opts in your Docker daemon config file to specify which keys are used to encrypt the communications (see appendix B for advice on how to configure and restart your Docker daemon).

Listing 14.4 Docker options for using the new keys and certificates

Specifies the CA file for the Docker server

Tells the Docker daemon that you want to use TLS security to secure connections to it

```
DOCKER_OPTS="$DOCKER_OPTS --tlsverify"
DOCKER_OPTS="$DOCKER_OPTS \
--tlscacert=/etc/docker/ca.pem"
DOCKER_OPTS="$DOCKER_OPTS \
--tlscert=/etc/docker/server-cert.pem"
DOCKER_OPTS="$DOCKER_OPTS \
--tlskey=/etc/docker/server-key.pem"
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2376"
DOCKER_OPTS="$DOCKER_OPTS \
-H unix:///var/run/docker.sock"
```

Specifies the certificate for the server

Specifies the private key used by the server

Opens the Docker daemon to external clients over TCP on port 2376

Opens the Docker daemon locally via a Unix socket in the normal way

DISTRIBUTING CLIENT KEYS

Next you need to send the keys to the client host so it can connect to the server and exchange information. You don't want to reveal your secret keys to anyone else, so these need to be passed to the client securely. A relatively safe way to do this is to SCP (secure copy) them direct from the server to the client. The SCP utility uses essentially the same technique to secure the transmission of data that we're demonstrating here, only with different keys that will have already been set up.

On the client host, create the Docker configuration folder in /etc as you did earlier:

```
user@client:~$ sudo su
root@client:~$ mkdir -p /etc/docker
```

Then SCP the files from the server to the client. Make sure you replace "client" in the following commands with the hostname of your client machine. Also make sure that all the files are readable by the user that will run the `docker` command on the client.

```
user@server:~$ sudo su
root@server:~$ scp /etc/docker/ca.pem client:/etc/docker
root@server:~$ scp /etc/docker/cert.pem client:/etc/docker
root@server:~$ scp /etc/docker/key.pem client:/etc/docker
```

TESTING

To test your setup, first try making a request to the Docker server without any credentials. You should be rejected:

```
root@client~: docker -H myserver.localdomain:2376 info
FATA[0000] Get http://myserver.localdomain:2376/v1.17/info: malformed HTTP >
response "\x15\x03\x01\x00\x02\x02". Are you trying to connect to a >
TLS-enabled daemon without TLS?
```

Then connect with the credentials, which should return useful output:

```
root@client~: docker --tlsverify --tlscacert=/etc/docker/ca.pem \
--tlscert=/etc/docker/cert.pem --tlskey=/etc/docker/key.pem \
-H myserver.localdomain:2376 info
243 info
Containers: 3
Images: 86
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 92
Execution Driver: native-0.2
Kernel Version: 3.16.0-34-generic
Operating System: Ubuntu 14.04.2 LTS
CPUs: 4
Total Memory: 11.44 GiB
Name: rothko
ID: 4YQA:KK65:FXON:YVLT:BVVH:Y3KC:UATJ:I4GK:S3E2:UTA6:R43U:DX5T
WARNING: No swap limit support
```

DISCUSSION

This technique gives you the best of both worlds—a Docker daemon open to others to use, and one that's only accessible to trusted users. Make sure you keep those keys safe!

Key management is a critical aspect of most larger organizations' IT management processes. It's definitely a cost, so when it comes to implementing a Docker platform, it can become one that's brought into sharp focus. Deploying keys safely to containers is a challenge that may well need to be considered in most Docker platform designs.

14.4 Security from outside Docker

Security on your host doesn't stop with the `docker` command. In this section you're going to see some other approaches to securing your Docker containers, this time from outside Docker.

We'll start off with a couple of techniques that modify your image to reduce the surface area for external attack once they're up and running. The subsequent two techniques consider how to run containers in a restricted way.

Of these latter two techniques, the first demonstrates the application platform as a service (aPaaS) approach, which ensures Docker runs within a straightjacket set up and controlled by the administrator. As an example, we'll run an OpenShift Origin server (an aPaaS that deploys Docker containers in a managed way) using Docker commands. You'll see that the end user's powers can be limited and managed by the administrator, and access to the Docker runtime can be removed.

The second approach goes beyond this level of security to further limit the freedoms available within running containers using SELinux, a security technology that gives you fine-grained control over who can do what.

> **TIP** SELinux is a tool built and open-sourced by the United States' National Security Agency (NSA) that fulfills their need for strong access control. It has been a security standard for some time now, and it's very powerful. Unfortunately, many people simply switch it off when they encounter problems with it, rather than take the time to understand it. We hope the technique shown here will help make that approach less tempting.

TECHNIQUE 97 Reducing a container's attack surface with DockerSlim

In section 7.3 we discussed a few different ways to create a small image in response to reasonable concern about the amount of data being moved around a network. But there's another reason to do this—if your image has less in it, there's less for an attacker to exploit. As one concrete example, there's no way to get a shell in the container if there's no shell installed.

Building up an "expected behavior" profile for your container and then enforcing that at runtime means that unexpected actions have a realistic chance of being detected and prevented.

PROBLEM

You want to reduce an image to the bare essentials to reduce its attack surface.

SOLUTION

Use the DockerSlim tool to analyze your image and modify it for a reduced attack surface.

This tool is intended to take a Docker image and reduce it to its barest essentials. It's available at https://github.com/docker-slim/docker-slim.

DockerSlim reduces your Docker image in at least two distinct ways. First, it reduces your image to only the required files and places these files in a single layer. The end result is an image that's significantly smaller than its original, fat counterpart.

Second, it provides you with a seccomp profile. This is achieved through dynamic analysis of your running image. In lay terms, this means that it runs up your image and tracks which files and system calls are used. While DockerSlim is analyzing your running container, you need to use the app as it would be by all typical users, to ensure that the necessary files and system calls are picked up.

> **WARNING** If you reduce your image using a dynamic analysis tool like this, be absolutely sure you've exercised it enough in the analysis stage. This walkthrough uses a trivial image, but you may have a more complex image that's harder to exhaustively profile.

This technique will use a simple web example application to demonstrate the technique. You will

- Set up DockerSlim
- Build an image
- Run the image as a container with the DockerSlim tool
- Hit an endpoint of the application
- Run the slimmed image using the created seccomp profile

> **NOTE** A seccomp profile is essentially a whitelist of which system calls can be made from a container. When running the container, you can specify a seccomp profile with either reduced or raised permissions, depending on what your application needs. The default seccomp profile disables around 45 system calls out of over 300. Most applications need far fewer than this.

SETTING UP DOCKERSLIM

Run these commands to get the docker-slim binary downloaded and set up.

Listing 14.5 Downloading docker-slim and installing it to a directory

```
$ mkdir -p docker-slim/bin && cd docker-slim/bin        <- Makes the docker-slim folder and a bin subfolder
$ wget https://github.com/docker-slim/docker-slim/releases/download/1.18
➥ /dist_linux.zip                                      <- Gets the docker-slim zip file from its release folder
$ unzip dist_linux.zip                                  <- Unzips the retrieved zip file
$ cd ..                                                 <- Moves to the parent directory, docker-slim
```

NOTE This technique was tested against the preceding docker-slim version. You may want to visit GitHub at https://github.com/docker-slim/docker-slim/releases to see whether there have been any updates. This isn't a fast-moving project, so the updates shouldn't be too important.

Now you have the docker-slim binary in a bin subfolder.

BUILDING THE FAT IMAGE

Next you'll build a sample application that uses NodeJS. This is a trivial application that simply serves a string of JSON on port 8000. The following command clones the docker-slim repository, moves to the sample application code, and builds its Dockerfile into an image with the name sample-node-app.

Listing 14.6 Building an example docker-slim application

```
$ git clone https://github.com/docker-slim/docker-slim.git
$ cd docker-slim && git checkout 1.18
$ cd sample/apps/node
$ docker build -t sample-node-app .
$ cd -
```

Clones the docker-slim repository, which contains the sample application

Checks out a known-working version of the docker-slim repository

Moves to the NodeJS sample application folder

Builds the image, giving it the name sample-node-app

Returns to the previous directory, where the docker-slim binary is located

RUNNING THE FAT IMAGE

Now that you've created your fat image, the next step involves running it as a container with the docker-slim wrapper. Once the application has initialized, you then hit the application endpoint to exercise its code. Finally, bring the backgrounded docker-slim application to the foreground and wait for it to terminate.

```
$ ./docker-slim build --http-probe sample-node-app &
$ sleep 10 && curl localhost:32770
{"status":"success","info":"yes!!!","service":"node"}
$ fg
./docker-slim build --http-probe sample-node-app
INFO[0014] docker-slim: HTTP probe started...
INFO[0014] docker-slim: http probe - GET http://127.0.0.1:32770/ => 200
INFO[0014] docker-slim: HTTP probe done.
INFO[0015] docker-slim: shutting down 'fat' container...
INFO[0015] docker-slim: processing instrumented 'fat' container info...
INFO[0015] docker-slim: generating AppArmor profile...
INFO[0015] docker-slim: building 'slim' image...
```

Runs the docker-slim binary against the sample-node-app image. Backgrounds the process. http-probe will call the application on all exposed ports.

Sleeps for 10 seconds to allow the sample-node-app process to start, and then hits the port the application runs on

Sends the application's JSON response to the terminal

Foregrounds the docker-slim process and waits until it completes

The first section of output from docker-slim shows its working logs.

```
Step 1 : FROM scratch
 --->
Step 2 : COPY files /
 ---> 0953a87c8e4f
Removing intermediate container 51e4e625017e
Step 3 : WORKDIR /opt/my/service
 ---> Running in a2851dce6df7
 ---> 2d82f368c130
Removing intermediate container a2851dce6df7
Step 4 : ENV PATH "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
➥ /bin"
 ---> Running in ae1d211f118e
 ---> 4ef6d57d3230
Removing intermediate container ae1d211f118e
Step 5 : EXPOSE 8000/tcp
 ---> Running in 36e2ced2a1b6
 ---> 2616067ec78d
Removing intermediate container 36e2ced2a1b6
Step 6 : ENTRYPOINT node /opt/my/service/server.js
 ---> Running in 16a35fd2fb1c
 ---> 7451554aa807
Removing intermediate container 16a35fd2fb1c
Successfully built 7451554aa807
INFO[0016] docker-slim: created new image: sample-node-app.slim
$
$
```

Docker-slim builds the "slim" container.

When it completes, you may need to press Return to get a prompt.

In this case "exercising the code" just involves hitting one URL and getting a response. More sophisticated apps will need more varied and diverse types of poking and prodding to ensure they've been completely exercised.

Note that according to the documents, we don't need to hit the app on port 32770 ourselves because we've used the http-probe argument. If you enable the HTTP probe, it will default to running an HTTP and HTTPS GET request on the root URL ("/") on every exposed port. We do the `curl` by hand simply for demonstration purposes.

At this point, you've created the sample-node-app.slim version of your image. If you examine the output of `docker images`, you can see that its size has been drastically reduced.

```
$ docker images
REPOSITORY             TAG      IMAGE ID       CREATED              SIZE
sample-node-app.slim   latest   7451554aa807   About an hour ago    14.02 MB
 sample-node-app        latest   78776db92c2a   About an hour ago    418.5 MB
```

The sample-node-app.slim image is just over 14 MB in size.

The original sample-node-app image was over 400 MB in size.

If you compare the `docker history` output of the fat sample app with its slim counterpart, you'll see that they're quite different in structure.

The docker history command is run on the sample-node-app image.

The history of this image shows each command as it was originally created.

```
$ docker history sample-node-app
IMAGE         CREATED       CREATED BY                                  SIZE
78776db92c2a  42 hours ago  /bin/sh -c #(nop)  ENTRYPOINT ["node"      0 B
0f044b6540cd  42 hours ago  /bin/sh -c #(nop)  EXPOSE 8000/tcp         0 B
555cf79f13e8  42 hours ago  /bin/sh -c npm install                      14.71 MB
6c62e6b40d47  42 hours ago  /bin/sh -c #(nop)  WORKDIR /opt/my/ser     0 B
7871fb6df03b  42 hours ago  /bin/sh -c #(nop) COPY dir:298f558c6f2     656 B
618020744734  42 hours ago  /bin/sh -c apt-get update &&   apt-get     215.8 MB
dea1945146b9  7 weeks ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]       0 B
<missing>     7 weeks ago   /bin/sh -c mkdir -p /run/systemd && ec     7 B
<missing>     7 weeks ago   /bin/sh -c sed -i 's/^#\s*\(deb.*unive     2.753 kB
<missing>     7 weeks ago   /bin/sh -c rm -rf /var/lib/apt/lists/*     0 B
<missing>     7 weeks ago   /bin/sh -c set -xe   && echo '#!/bin/s     194.6 kB
<missing>     7 weeks ago   /bin/sh -c #(nop) ADD file:8f997234193     187.8 MB
$ docker history sample-node-app.slim
IMAGE         CREATED       CREATED BY                                  SIZE
7451554aa807  42 hours ago  /bin/sh -c #(nop)  ENTRYPOINT ["node"      0 B
2616067ec78d  42 hours ago  /bin/sh -c #(nop)  EXPOSE 8000/tcp         0 B
4ef6d57d3230  42 hours ago  /bin/sh -c #(nop)  ENV PATH=/usr/local     0 B
2d82f368c130  42 hours ago  /bin/sh -c #(nop)  WORKDIR /opt/my/ser     0 B
0953a87c8e4f  42 hours ago  /bin/sh -c #(nop) COPY dir:36323da1e97     14.02 MB
```

The docker history command is run on the sample-node-app.slim image.

The history of the slim container consists of fewer commands, including a COPY command not in the original fat image.

The preceding output gives a clue about part of what DockerSlim does. It manages to reduce the image size to (effectively) a single 14 MB layer by taking the final filesystem state, and copying that directory as the final layer of the image.

The other artifact produced by DockerSlim relates to its second purpose as described at the beginning of this technique. A seccomp.json file is produced (in this case, sample-node-app-seccomp.json), which can be used to limit the actions of the running container.

Let's take a look at this file's contents (edited here, as it's rather long).

Listing 14.7 A seccomp profile

Specifies the exit code for the process that tries to call any forbidden syscall

Captures the location of the seccomp file in the variable SECCOMPFILE

```
$ SECCOMPFILE=$(ls $(pwd)/.images/*/artifacts/sample-node-app-seccomp.json)
$ cat ${SECCOMPFILE}
{
"defaultAction": "SCMP_ACT_ERRNO",
  "architectures": [
  "SCMP_ARCH_X86_64"
  ],
  "syscalls": [
    {
      "name": "capset",
      "action": "SCMP_ACT_ALLOW"
```

Cats this file to view it

Specifies the hardware architectures this profile should be applied on

The syscalls controlled are whitelisted here by specifying the SCMP_ACT_ALLOW action against them.

```
      },
      {
        "name": "rt_sigaction",
        "action": "SCMP_ACT_ALLOW"
      },
      {
        "name": "write",
        "action": "SCMP_ACT_ALLOW"
      },
  [...]
     {
      "name": "execve",
        "action": "SCMP_ACT_ALLOW"
      },
      {
        "name": "getcwd",
        "action": "SCMP_ACT_ALLOW"
      }
    ]
}
```

The syscalls controlled are whitelisted here by specifying the SCMP_ACT_ALLOW action against them.

Finally, you're going to run up the slim image again with the seccomp profile and check that it works as expected:

Runs the slim image as a daemon, exposing the same port that DockerSlim exposed in its analysis phase, and applies the seccomp profile to it

Outputs the container ID to the terminal

```
$ docker run -p32770:8000 -d \
--security-opt seccomp=/root/docker-slim-bin/.images/${IMAGEID}/artifacts
➥ /sample-node-app-seccomp.json sample-node-app.slim
 4107409b61a03c3422e07973248e564f11c6dc248a6a5753a1db8b4c2902df55
 $ sleep 10 && curl localhost:32771
 {"status":"success","info":"yes!!!","service":"node"}
```

Reruns the curl command to confirm the application still works as before

The output is identical to the fat image you've slimmed.

DISCUSSION

This simple example has shown how an image can be reduced not just in size, but also in the scope of the actions it can perform. This is achieved by removing inessential files (also discussed in technique 59), and reducing the syscalls available to it to only those that are needed to run the application.

The means of "exercising" the application here was simple (one `curl` request to the default endpoint). For a real application, there are a number of approaches you can take to ensure you've covered all the possibilities. One way is to develop a set of tests against known endpoints, and another is to use a "fuzzer" to throw lots of inputs at the application in an automated way (this is one way to find bugs and security flaws in your software). The simplest way is to leave your application running for a longer period of time in the expectation that all the needed files and system calls will be referenced.

Many enterprise Docker security tools work on this principle, but in a more automated way. Typically they allow an application to run for some time, and track which syscalls are made, which files are accessed, and also (possibly) which operating system capabilities are used. Based on this—and a configurable learning period—they can determine what the expected behavior of an application is, and report any behavior that seems to be out of line. For example, if an attacker gains access to a running container and starts up the bash binary or opens unexpected ports, this might raise an alarm on the system. DockerSlim allows you to take control over this process up-front, reducing what an attacker might be capable of doing even if they got access.

Another way to consider slimming your application's attack surface is to constrain its capabilities. This is covered in technique 93.

TECHNIQUE 98 Removing secrets added during a build

When you're building images in a corporate environment, it's often necessary to use keys and credentials to retrieve data. If you're using a Dockerfile to build an application, these secrets will generally be present in the history, even if you delete it after use.

This can be a security problem: if someone got hold of the image, they might also get hold of the secret in the earlier layers.

PROBLEM

You want to remove a file from an image's history.

SOLUTION

Use `docker-squash` to remove layers from the image.

There are simple ways to solve this problem that work in theory. For example, you might delete the secret while it's being used, as follows.

Listing 14.8 Crude method of not leaving a secret within a layer

```
FROM ubuntu
RUN echo mysecret > secretfile && command_using_secret && rm secretfile
```

This approach suffers from a number of disadvantages. It requires the secret to be put into code in the Dockerfile, so it may be in plain text in your source control.

To avoid this problem, you might add the file to your .gitignore (or similar) file in your source control, and `ADD` it to the image while it's being built. This adds the file in a separate layer, which can't easily be removed from the resulting image.

Finally, you could use environment variables to store secrets, but this also creates security risks, with these variables being easily set in non-secure persistent stores like Jenkins jobs. In any case, you may be presented with an image by a user and asked to scrub the secret from it. First we're going to demonstrate the problem with a simple example, and then we'll show you a way to remove the secret from the base layer.

An image with a secret

The following Dockerfile will create an image using the file called secret_file as a placeholder for some secret data you've put in your image.

Listing 14.9 Simple Dockerfile with a secret

```
FROM ubuntu
CMD ls /
 ADD /secret_file secret_file
 RUN cat /secret_file
 RUN rm /secret_file
```

To save a bit of time, we override the default command with a file listing command. This will demonstrate whether the file is in the history.

Adds the secret file to the image build (this must exist in your current working directory along with the Dockerfile)

Uses the secret file as part of the build. In this case, we use the trivial cat command to output the file, but this could be a git clone or other more useful command.

Removes the secret file

Now you can build this image, calling the resulting image secret_build.

Listing 14.10 Building the simple Docker image with a secret

```
$ echo mysecret > secret_file
$ docker build -t secret_build .
Sending build context to Docker daemon  5.12 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> 08881219da4a
Step 1 : CMD ls /
 ---> Running in 7864e2311699
 ---> 5b39a3cba0b0
Removing intermediate container 7864e2311699
Step 2 : ADD /secret_file secret_file
 ---> a00886ff1240
Removing intermediate container 4f279a2af398
Step 3 : RUN cat /secret_file
 ---> Running in 601fdf2659dd
My secret
 ---> 2a4238c53408
Removing intermediate container 601fdf2659dd
Step 4 : RUN rm /secret_file
 ---> Running in 240a4e57153b
 ---> b8a62a826ddf
Removing intermediate container 240a4e57153b
Successfully built b8a62a826ddf
```

Once the image is built, you can demonstrate that it has the secret file by using technique 27.

Listing 14.11 Tagging each step and demonstrating the layer with the secret

```
$ x=0; for id in $(docker history -q secret_build:latest);
➥ do ((x++)); docker tag $id secret_build:step_$x; done
$ docker run secret_build:step_3 cat /secret_file'
mysecret
```

Tags each step of the build in numerical order

Demonstrates that the secret file is in this tag of the image

SQUASHING IMAGES TO REMOVE SECRETS

You've seen that secrets can remain in the history of images even if they're not in the final one. This is where `docker-squash` comes in—it removes the intervening layers but retains the Dockerfile commands (such as `CMD`, `PORT`, `ENV`, and so on) and the original base layer in your history.

The following listing downloads, installs, and uses `docker-squash` to compare the pre- and post-squashed images.

Listing 14.12 Using `docker_squash` to reduce layers of an image

Installs docker-squash. (You may need to refer to https://github.com/jwilder/docker-squash for the latest installation instructions.)

Saves the image to a TAR file that docker-squash operates on, and then loads the resulting image in, tagging it as "secret_build_squashed"

```
$ wget -qO- https://github.com/jwilder/docker-squash/releases/download
➥ /v0.2.0/docker-squash-linux-amd64-v0.2.0.tar.gz | \
  tar -zxvf -  && mv docker-squash /usr/local/bin
$ docker save secret_build:latest | \
  docker-squash -t secret_build_squashed | \
  docker load
$ docker history secret_build_squashed
IMAGE          CREATED         CREATED BY                            SIZE
ee41518cca25  2 seconds ago  /bin/sh -c #(nop) CMD ["/bin/sh" "  0 B
b1c283b3b20a  2 seconds ago  /bin/sh -c #(nop)  CMD ["/bin/bash  0 B
f443d173e026  2 seconds ago  /bin/sh -c #(squash) from 93c22f56  2.647 kB
93c22f563196  2 weeks ago    /bin/sh -c #(nop) ADD file:7529d28  128.9 MB
$ docker history secret_build
IMAGE          CREATED         CREATED BY                            SIZE
b8a62a826ddf  3 seconds ago  /bin/sh -c rm /secret_file          0 B
2a4238c53408  3 seconds ago  /bin/sh -c cat /secret_file         0 B
a00886ff1240  9 seconds ago  /bin/sh -c #(nop) ADD file:69e77f6  10 B
5b39a3cba0b0  9 seconds ago  /bin/sh -c #(nop) CMD ["/bin/sh" "  0 B
08881219da4a  2 weeks ago    /bin/sh -c #(nop)  CMD ["/bin/bash  0 B
6a4ec4bddc58  2 weeks ago    /bin/sh -c mkdir -p /run/systemd &  7 B
98697477f76a  2 weeks ago    /bin/sh -c sed -i 's/^#\s*\(deb.*u  1.895 kB
495ec797e6ba  2 weeks ago    /bin/sh -c rm -rf /var/lib/apt/lis  0 B
e3aa81f716f6  2 weeks ago    /bin/sh -c set -xe && echo '#!/bin  745 B
93c22f563196  2 weeks ago    /bin/sh -c #(nop) ADD file:7529d28  128.9 MB
$ docker run secret_build_squashed ls /secret_file
ls: cannot access '/secret_file': No such file or directory
$ docker run f443d173e026 ls /secret_file
ls: cannot access '/secret_file': No such file or directory
```

The history of the squashed image has no record of secret_file.

The origin image has the secret_file still in it.

Demonstrates that the secret_file is not in the squashed image

Demonstrates that the secret_file is not in the squashed image's "squashed" layer

A note on "missing" image layers

Docker changed the nature of layering in Docker 1.10. From that point on, images downloaded show up as "<missing>" in the history. This is expected and is because of changes made by Docker to improve the security of images' histories.

You can still get the contents of layers you've downloaded by `docker save`ing the image and then extracting the TAR files from within that TAR file. Here's an example session that does that for the already-downloaded Ubuntu image.

Listing 14.13 "Missing" layers in downloaded images

Uses the docker save command to output a TAR file of the image layers, which is piped straight to tar and extracted

Uses the docker history command to show the layer history of the Ubuntu image

```
$ docker history ubuntu
 IMAGE          CREATED       CREATED BY                                    SIZE
104bec311bcd  2 weeks ago  /bin/sh -c #(nop)  CMD ["/bin/bash"]      0 B
<missing>     2 weeks ago  /bin/sh -c mkdir -p /run/systemd && ech  7 B
<missing>     2 weeks ago  /bin/sh -c sed -i 's/^#\s*\(deb.*univer  1.9 kB
<missing>     2 weeks ago  /bin/sh -c rm -rf /var/lib/apt/lists/*   0 B
<missing>     2 weeks ago  /bin/sh -c set -xe   && echo '#!/bin/sh  745 B
<missing>     2 weeks ago  /bin/sh -c #(nop) ADD file:7529d28035b4  129 MB
$ docker save ubuntu | tar -xf -
 $ find . | grep tar$
 ./042e55060780206b2ceabe277a8beb9b10f48262a876fd21b495af318f2f2352/layer.tar
./1037e0a8442d212d5cc63d1bc706e0e82da0eaafd62a2033959cfc629f874b28/layer.tar
./25f649b30070b739bc2aa3dd877986bee4de30e43d6260b8872836cdf549fcfc/layer.tar
./3094e87864d918dfdb2502e3f5dc61ae40974cd957d5759b80f6df37e0e467e4/layer.tar
./41b8111724ab7cb6246c929857b0983a016f11346dcb25a551a778ef0cd8af20/layer.tar
./4c3b7294fe004590676fa2c27a9a952def0b71553cab4305aeed4d06c3b308ea/layer.tar
./5d1be8e6ec27a897e8b732c40911dcc799b6c043a8437149ab021ff713e1044f/layer.tar
./a594214bea5ead6d6774f7a09dbd7410d652f39cc4eba5c8571d5de3bcbe0057/layer.tar
./b18fcc335f7aeefd87c9d43db2888bf6ea0ac12645b7d2c33300744c770bcec7/layer.tar
./d899797a09bfcc6cb8e8a427bb358af546e7c2b18bf8e2f7b743ec36837b42f2/layer.tar
./ubuntu.tar
$ tar -tvf
➥ ./4c3b7294fe004590676fa2c27a9a952def0b71553cab4305aeed4d06c3b308ea
➥ /layer.tar
drwxr-xr-x  0 0      0           0 15 Dec 17:45 etc/
drwxr-xr-x  0 0      0           0 15 Dec 17:45 etc/apt/
-rw-r--r--  0 0      0        1895 15 Dec 17:45 etc/apt/sources.list
```

Demonstrates that the TAR files contain only file changes within that layer

Discussion

Although somewhat similar in intent to technique 52, the use of a specialized tool has some notable differences in the end result. In the preceding solution, you can see that metadata layers like `CMD` have been preserved, whereas the previous technique on this subject would discard them entirely, so you'd need to manually recreate those metadata layers through another Dockerfile.

This behavior means the docker-squash utility could be used to automatically clean up images as they arrive in a registry, if you're inclined not to trust your users to use secret data correctly within image builds—they should all work normally.

That said, you should be wary of your users putting secrets in any metadata layers—environment variables in particular are a threat and may well be preserved in the final image.

TECHNIQUE 99 OpenShift: An application platform as a service

OpenShift is a product managed by Red Hat that allows an organization to run an application platform as a service (aPaas). It offers application development teams a platform on which to run code without needing to be concerned about hardware details. Version 3 of the product was a ground-up rewrite in Go, with Docker as the container technology and Kubernetes and etcd for orchestration. On top of this, Red Hat has added enterprise features that enable it to be more easily deployed in a corporate and security-focused environment.

Although OpenShift has many features we could cover, we'll use it here as a means of managing security by taking away the user's ability to run Docker directly, but retaining the benefits of using Docker.

OpenShift is available both as an enterprise-supported product, and as an open source project called Origin, maintained at https://github.com/openshift/origin.

PROBLEM

You want to manage the security risk of untrusted users invoking `docker run`.

SOLUTION

Use an aPaaS tool to manage and mediate the interaction with Docker via a proxying interface.

An aPaaS has many benefits, but the one we'll focus on here is its ability to manage user permissions and run Docker containers on the user's behalf, providing a secure audit point for users running Docker containers.

Why is this important? The users using this aPaaS have no direct access to the `docker` command, so they can't do any damage without subverting the security that OpenShift provides. For example, containers are deployed by non-root users by default, and overcoming this requires permission to be granted by an administrator. If you can't trust your users, using an aPaaS is a effective way of giving them access to Docker.

> **TIP** An aPaaS provides users with the ability to spin up applications on demand for development, testing, or production. Docker is a natural fit for these services, as it provides a reliable and isolated application delivery format, allowing an operations team to take care of the details of deployment.

In short, OpenShift builds on Kubernetes (see technique 88) but adds features to deliver a full-fledged aPaaS. These additional features include

- User management
- Permissioning
- Quotas
- Security contexts
- Routing

Installing OpenShift

A complete overview of OpenShift installation is beyond the scope of this book. If you'd like an automated install, using Vagrant, that we maintain, see https://github.com/docker-in-practice/shutit-openshift-origin. If you need help installing Vagrant, see appendix C.

Other options, such as a Docker-only installation (single-node only), or a full manual build are available and documented on the OpenShift Origin codebase at https://github.com/openshift/origin.git.

> **TIP** OpenShift Origin is the upstream version of OpenShift. *Upstream* means that it's the codebase from which Red Hat takes changes for OpenShift, its supported offering. Origin is open source and can be used and contributed to by anyone, but Red Hat's curated version of it is sold and supported as OpenShift. An upstream version is usually more cutting edge but less stable.

An OpenShift application

In this technique we're going to show a simple example of creating, building, running, and accessing an application using the OpenShift web interface. The application will be a basic NodeJS application that serves a simple web page.

The application will use Docker, Kubernetes, and S2I under the hood. Docker is used to encapsulate the build and deployment environments. The Source to Image (S2I) build method is a technique used by Red Hat in OpenShift to build the Docker container, and Kubernetes is used to run the application on the OpenShift cluster.

Logging in

To get started, run `./run.sh` from the shutit-openshift-origin folder, and then navigate to https://localhost:8443, bypassing all the security warnings. You'll see the login page shown in figure 14.3. Note that if you're using the Vagrant install, you'll need to start up a web browser in your VM. (See appendix C for help on getting a GUI with your VM.)

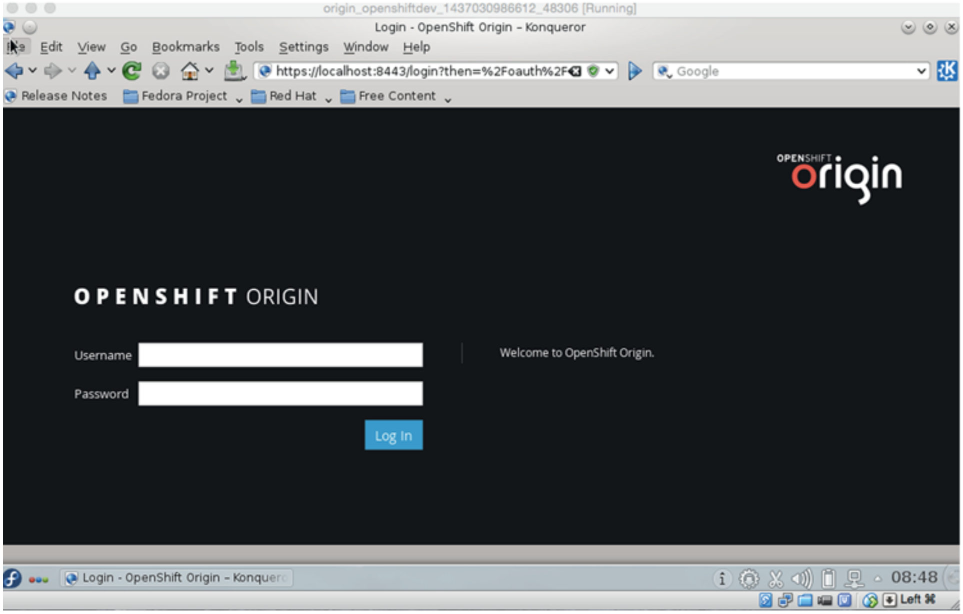

Figure 14.3 The OpenShift login page

Log in as `hal-1` with any password.

BUILDING A NODEJS APP

You're now logged into OpenShift as a developer (see figure 14.4).

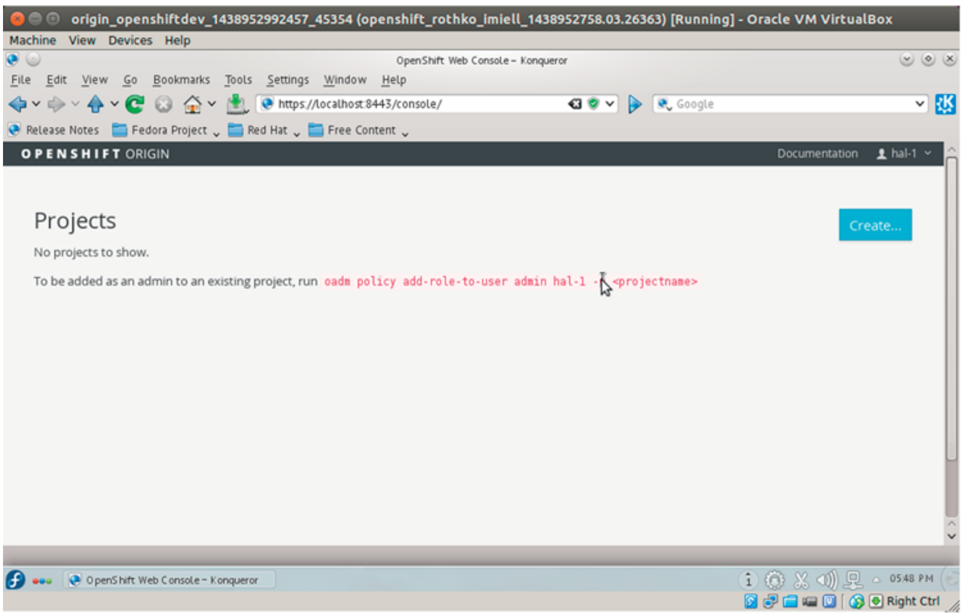

Figure 14.4 The OpenShift Projects page

Create a project by clicking Create. Fill out the form, as shown in figure 14.5. Then click Create again.

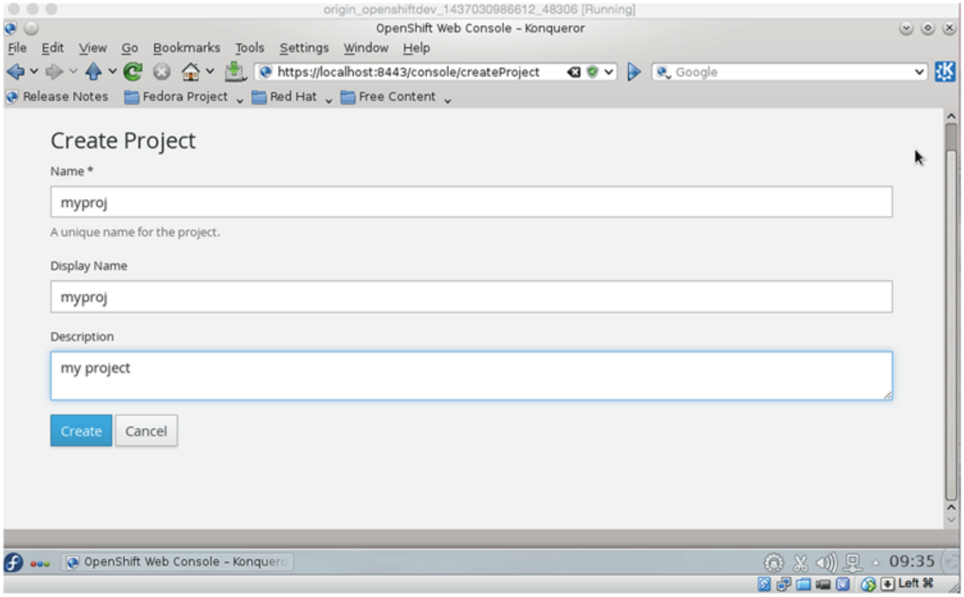

Figure 14.5 The OpenShift project-creation page

Once the project is set up, click Create again and input the suggested GitHub repo (https://github.com/openshift/nodejs-ex), as shown in figure 14.6.

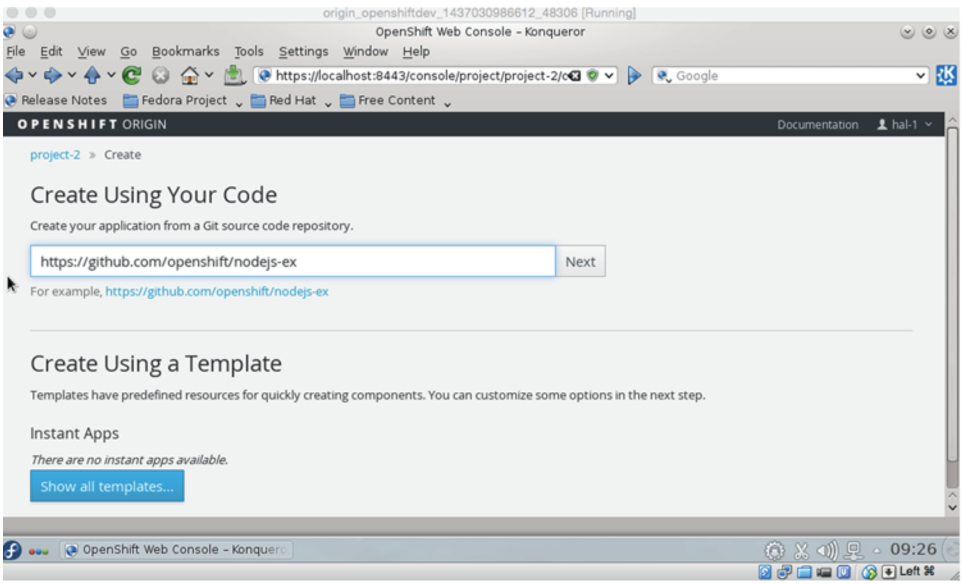

Figure 14.6 The OpenShift project source page

Click Next, and you'll be given a choice of builder images, as shown in figure 14.7. The build image defines the context in which the code will be built. Choose the NodeJS builder image.

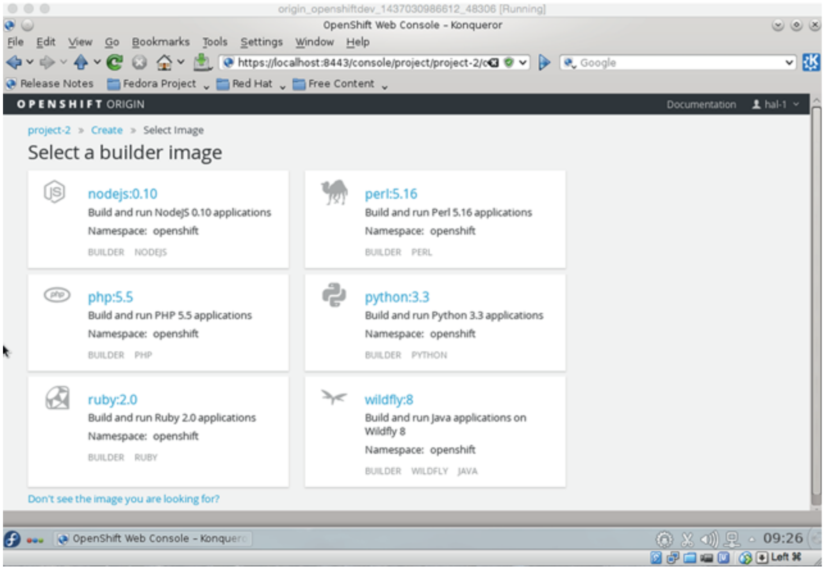

Figure 14.7 The OpenShift builder-image selection page

Now fill out the form, as shown in figure 14.8. Click Create on NodeJS at the bottom of the page as you scroll down the form.

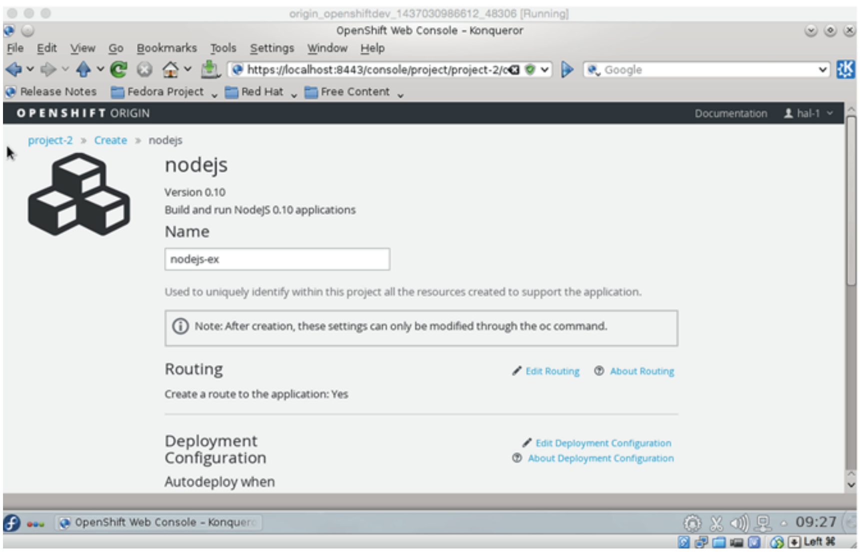

Figure 14.8 The OpenShift NodeJS template form

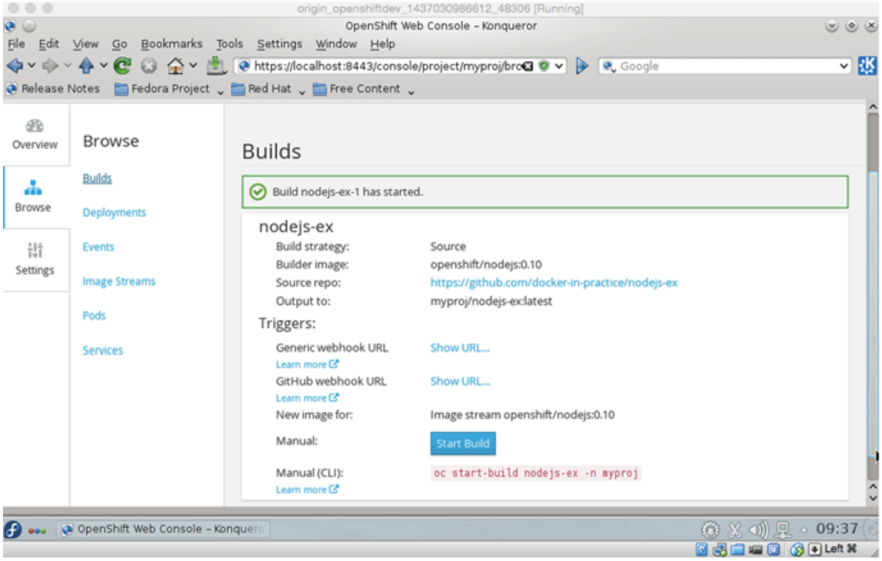

Figure 14.9 The OpenShift build-started page

After a few minutes, you should see a screen like the one in figure 14.9.

In a few moments, if you scroll down, you'll see that the build has started, as shown in figure 14.10.

TIP In early versions of OpenShift, the build would sometimes not begin automatically. If this is the case, click the Start Build button after a few minutes.

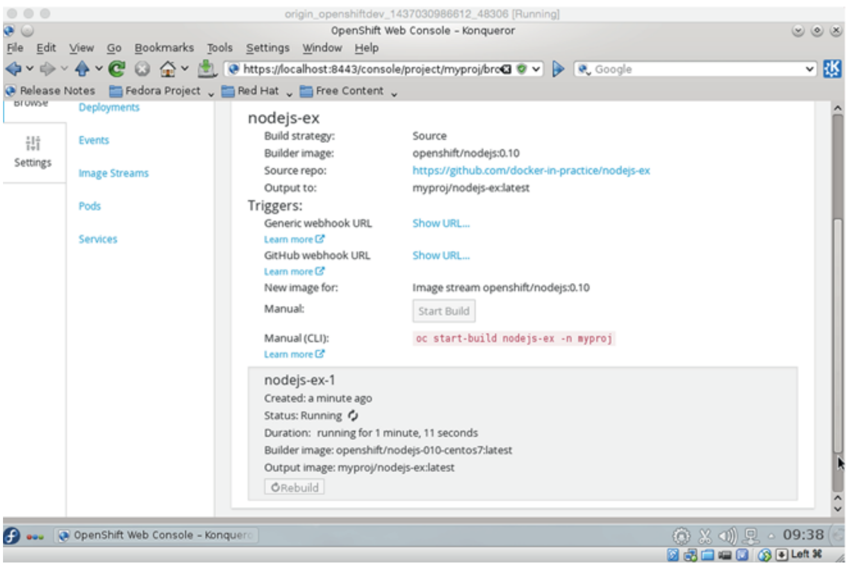

Figure 14.10 The OpenShift build-information window

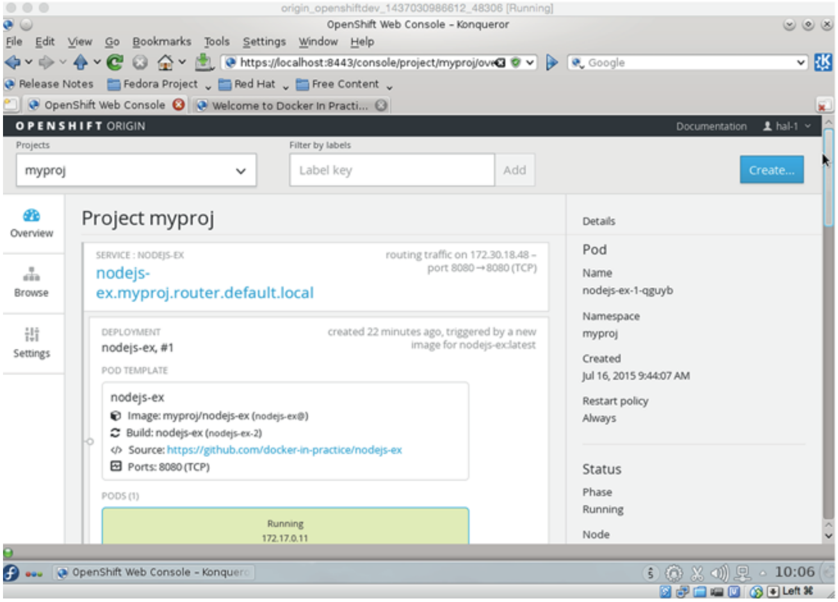

Figure 14.11 Application-running page

After some time you'll see that the app is running, as in figure 14.11.

By clicking Browse and Pods, you can see that the pod has been deployed, as in figure 14.12.

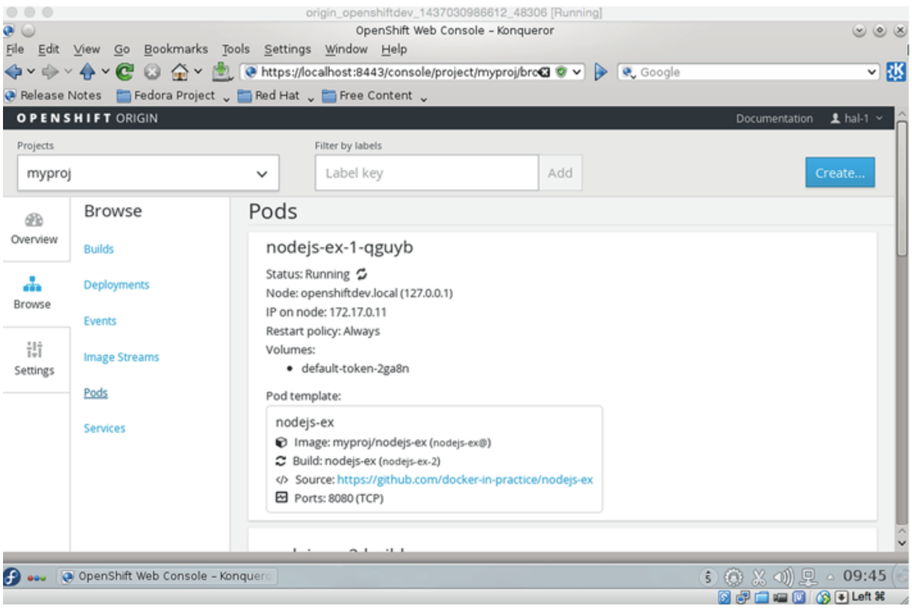

Figure 14.12 List of OpenShift pods

TIP See technique 88 for an explanation of what a pod is.

How do you access your pod? If you look at the Services tab (see figure 14.13), you'll see an IP address and port number to access.

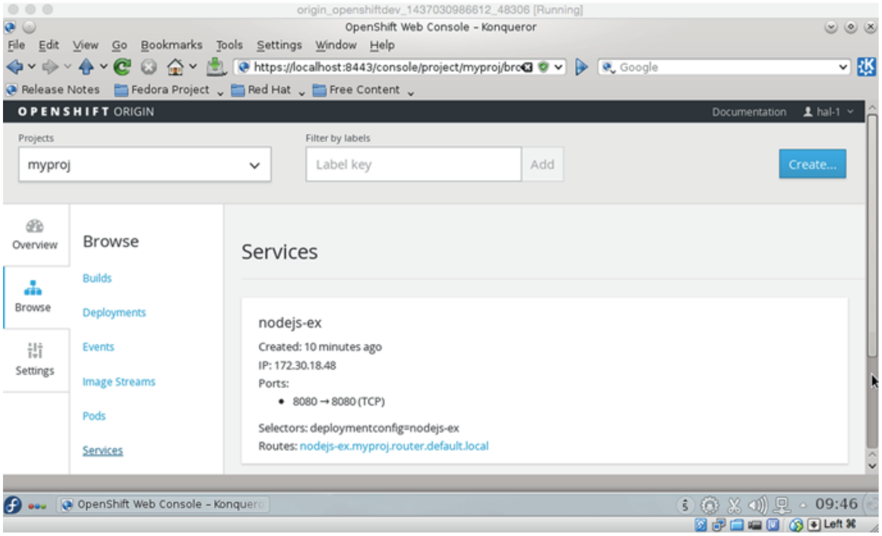

Figure 14.13 The OpenShift NodeJS application service details

Point your browser at that address, and voila, you'll have your NodeJS app, as in figure 14.14.

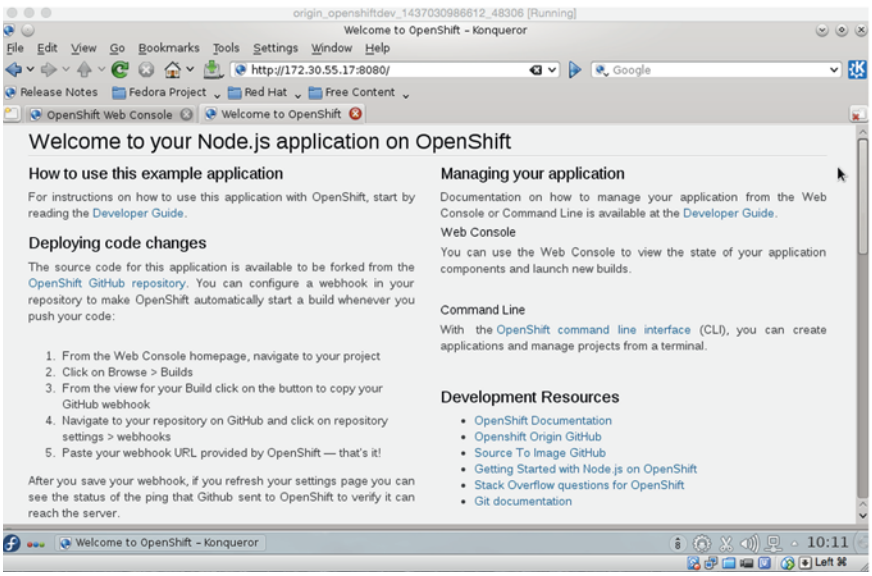

Figure 14.14 The NodeJS application landing page

DISCUSSION

Let's recap what we've achieved here, and why it's important for security.

From the point of view of the user, they logged into a web application and deployed an application using Docker-based technologies without going near a Dockerfile or the `docker run` command.

The administrator of OpenShift can

- Control user access
- Limit resource use by project
- Provision resources centrally
- Ensure code is run with non-privileged status by default

This is far more secure than giving users direct access to `docker run`.

If you want to build on this application and see how an aPaaS facilitates an iterative approach, you can fork the Git repository, change the code in that forked repository, and then create a new application. We've done that here: https://github.com/docker-in-practice/nodejs-ex.

To read more about OpenShift, go to http://www.openshift.org.

TECHNIQUE 100 Using security options

You've already seen in previous techniques how, by default, you're given root in the Docker container, and that this user is the same root as the root user on the host. To alleviate this, we've shown you how this user can have its capabilities as root reduced, so that even if it escapes the container, there are still actions the kernel won't allow this user to perform.

But you can go further than this. By using Docker's security-options flag you can protect resources on the host from being affected by actions performed within a container. This constrains the container to only affecting resources it has been given permission to by the host.

PROBLEM

You want to secure your host against the actions of containers.

SOLUTION

Use SELinux to impose constraints on your containers.

Here we're going to use SELinux as our kernel-supported mandatory access control (MAC) tool. SELinux is more or less the industry standard and is most likely to be used by organizations that particularly care about security. It was originally developed by the NSA to protect their systems and was subsequently open-sourced. It's used in Red Hat–based systems as a standard.

SELinux is a big subject, so we can't cover it in depth in this book. We're going to show you how to write and enforce a simple policy so that you can get a feel for how it works. You can take things further and experiment if you need to.

TIP Mandatory access control (MAC) tools in Linux enforce security rules beyond the standard ones you may be used to. Put briefly, they ensure that not only are the *normal* rules of read-write-execute on files and processes enforced, but more fine-grained rules can be applied to processes at the kernel level. For example, a MySQL process may only be allowed to write files under specific directories, such as /var/lib/mysql. The equivalent standard for Debian-based systems is AppArmor.

This technique assumes you have a SELinux-enabled host. This means you must first install SELinux (assuming it's not already installed). If you're running Fedora or some other Red Hat–based system, you likely have it already.

To determine whether you have SELinux enabled, run the command `sestatus`:

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

The first line of the output will tell you whether SELinux is enabled. If the command isn't available, you don't have SELinux installed on your host.

You'll also need to have the relevant SELinux policy-creation tools available. On a yum-capable machine, for example, you'll need to run `yum -y install selinux-policy -devel`.

SELinux on a Vagrant machine

If you don't have SELinux and want it to be built for you, you can use a ShutIt script to build a VM inside your host machine, with Docker and SELinux preinstalled. What it does is explained at a high level in figure 14.15.

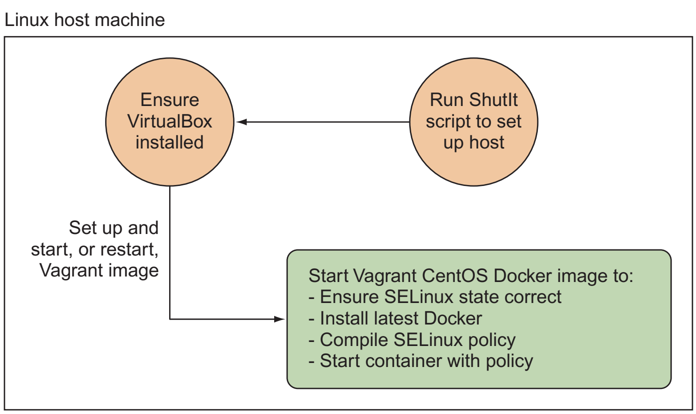

Figure 14.15 Script to provision a SELinux VM

TIP ShutIt is a generic shell automation tool that we created to overcome some limitations of Dockerfiles. If you want to read more about it, see the GitHub page: http://ianmiell.github.io/shutit.

Figure 14.5 identifies the steps required to get a policy set up. The script will do the following:

1. Set up VirtualBox
2. Start an appropriate Vagrant image
3. Log into the VM
4. Ensure the state of SELinux is correct
5. Install the latest version of Docker
6. Install the SELinux policy development tools
7. Give you a shell

Here are the commands to set up and run it (tested on Debian and Red Hat–based distributions):

Listing 14.14 Installing ShutIt

```
sudo su -
 apt-get install -y git python-pip docker.io || \
yum install -y git python-pip docker.io
 pip install shutit
 git clone https://github.com/ianmiell/docker-selinux.git
 cd docker-selinux
 shutit build --delivery bash \
 -s io.dockerinpractice.docker_selinux.docker_selinux \
  compile_policy no
```

Ensures you are root before starting the run

Ensures the required packages are installed on the host

Installs ShutIt

Clones the SELinux ShutIt script and enters its directory

Runs the ShutIt script. "--delivery bash" means commands are executed in bash rather than via SSH or in a Docker container.

Configures the script to not compile a SELinux policy, as we'll do this by hand

After running this script, you should eventually see output like this:

```
Pause point:
Have a shell:
You can now type in commands and alter the state of the target.
Hit return to see the prompt
Hit CTRL and ] at the same time to continue with build

Hit CTRL and u to save the state
```

You now have a shell running inside a VM with SELinux on it. If you type `sestatus`, you'll see that SELinux is enabled in permissive mode (as shown in listing 14.14). To return to your host's shell, press Ctrl-].

Compiling an SELinux policy

Whether you used the ShutIt script or not, we assume you now have a host with SELinux enabled. Type `sestatus` to get a status summary.

Listing 14.15 SELinux status once installed and enabled

```
# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

In this case, we're in permissive mode, which means that SELinux is recording violations of security in logs, but isn't enforcing them. This is good for safely testing new policies without rendering your system unusable. To move your SELinux status to permissive, type `setenforce Permissive` as root. If you can't do this on your host for security reasons, don't worry; there's an option to set the policy as permissive outlined in listing 14.15.

> **NOTE** If you're installing SELinux and Docker yourself on a host, ensure that the Docker daemon has `--selinux-enabled` set as a flag. You can check this with `ps -ef | grep 'docker -d.*--selinux-enabled`, which should return a matching process on the output.

Create a folder for your policy and move to it. Then create a policy file with the following content as root, named docker_apache.te. This policy file contains a policy we'll try to apply.

Listing 14.16 Creating a SELinux policy

Creates a folder to store the policy files, and moves into it

Creates the policy file that will be compiled as a "here" document

Creates the SELinux policy module docker_apache with the policy_module directive

```
mkdir -p /root/httpd_selinux_policy && >
cd /root/httpd_selinux_policy
 cat > docker_apache.te << END
 policy_module(docker_apache,1.0)
 virt_sandbox_domain_template(docker_apache)
 allow docker_apache_t self: capability { chown dac_override kill setgid >
setuid net_bind_service sys_chroot sys_nice >
sys_tty_config } ;
```

The Apache web server requires these capabilities to run; adds them here with the allow directive.

Uses the provided template to create the docker_apache_t SELinux type, which can be run as a Docker container. This template gives the docker_apache SELinux domain the fewest privileges required to run. We'll add to these privileges to make a useful container environment.

```
 allow docker_apache_t self:tcp_socket >
create_stream_socket_perms;
 allow docker_apache_t self:udp_socket >
create_socket_perms;
 corenet_tcp_bind_all_nodes(docker_apache_t)
 corenet_tcp_bind_http_port(docker_apache_t)
 corenet_udp_bind_all_nodes(docker_apache_t)
 corenet_udp_bind_http_port(docker_apache_t)
 sysnet_dns_name_resolve(docker_apache_t)
 #permissive docker_apache_t
 END
```

These allow and corenet rules give permission for the container to listen to Apache ports on the network.

Allows DNS server resolution with the sysnet directive

Optionally makes the docker_apache_t type permissive so this policy isn't enforced even if the host is enforcing SELinux. Use this if you can't set the SELinux mode of the host.

Terminates the "here" document, which writes it out to disk

TIP For more information about the preceding permissions, and to explore others, you can install the selinux-policy-doc package and use a browser to browse the documentation on file:///usr/share/doc-base/selinux-policy-doc/html/index.html. The docs are also available online at http://oss.tresys.com/docs/refpolicy/api/.

Now you're going to compile this policy and see your application fail to start against this policy in enforcing mode. Then you'll restart it in permissive mode to check the violations and correct it later:

```
$ make -f /usr/share/selinux/devel/Makefile \
docker_apache.te
 Compiling targeted docker_apache module
/usr/bin/checkmodule:  loading policy configuration from >
tmp/docker_apache.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing binary representation (version 17) >
to tmp/docker_apache.mod
Creating targeted docker_apache.pp policy package
rm tmp/docker_apache.mod tmp/docker_apache.mod.fc
$ semodule -i docker_apache.pp
 $ setenforce Enforcing
 $ docker run -ti --name selinuxdock >
--security-opt label:type:docker_apache_t httpd
 Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
2a341c7141bd: Pull complete
[...]
Status: Downloaded newer image for httpd:latest
permission denied
Error response from daemon: Cannot start container >
650c446b20da6867e6e13bdd6ab53f3ba3c3c565abb56c4490b487b9e8868985: >
[8] System error: permission denied
$ docker rm -f selinuxdock
 selinuxdock
```

Compiles the docker_apache.te file to a binary SELinux module with a .pp suffix

Installs the module

Sets the SELinux mode to "enforcing"

Runs the httpd image as a daemon, applying the security label type of docker_apache_t you defined in the SELinux module. This command should fail because it violates the SELinux security configuration.

Removes the newly created container

```
$ setenforce Permissive
 $ docker run -d --name selinuxdock >
--security-opt label:type:docker_apache_t httpd
```

Sets the SELinux mode to "permissive" to allow the application to start up

Runs the httpd image as a daemon, applying the security label type of docker_apache_t you defined in the SELinux module. This command should run successfully.

Checking for violations

Up to this point you've created a SELinux module and applied it to your host. Because the enforcement mode of SELinux is set to permissive on this host, actions that would be disallowed in enforcing mode are allowed with a log line in the audit log. You can check these messages by running the following command:

The type of message in the audit log is always AVC for SELinux violations, and timestamps are given as the number of seconds since the epoch (which is defined as 1st Jan 1970).

The type of action denied is shown in the curly brackets.

The process ID and name of the command that triggered the violation

The path, device, and inode of the target file

```
$ grep -w denied /var/log/audit/audit.log
type=AVC msg=audit(1433073250.049:392): avc:  >
 denied  { transition } for >
 pid=2379 comm="docker" >
 path="/usr/local/bin/httpd-foreground" dev="dm-1" ino=530204 >
 scontext=system_u:system_r:init_t:s0 >
tcontext=system_u:system_r:docker_apache_t:s0:c740,c787 >
 tclass=process
 type=AVC msg=audit(1433073250.049:392): avc:  denied  { write } for  >
pid=2379 comm="httpd-foregroun" path="pipe:[19550]" dev="pipefs" >
ino=19550 scontext=system_u:system_r:docker_apache_t:s0:c740,c787 >
tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
type=AVC msg=audit(1433073250.236:394): avc:  denied  { append } for  >
pid=2379 comm="httpd" dev="pipefs" ino=19551 >
scontext=system_u:system_r:docker_apache_t:s0:c740,c787 >
tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
type=AVC msg=audit(1433073250.236:394): avc:  denied  { open } for  >
pid=2379 comm="httpd" path="pipe:[19551]" dev="pipefs" ino=19551 >
scontext=system_u:system_r:docker_apache_t:s0:c740,c787 >
tcontext=system_u:system_r:init_t:s0 tclass=fifo_file
[...]
```

The SELinux context of the target

The class of the target object

Phew! There's a lot of jargon there, and we don't have time to teach you everything you might need to know about SELinux. If you want to find out more, a good place to start is with Red Hat's SELinux documentation: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/ch-selinux.html.

For now, you need to check that the violations are nothing untoward. What might look untoward? If an application tries to open a port or a file you didn't expect, you might think twice about doing what we'll show you next: patch these violations with a new SELinux module.

In this case, we're happy that the httpd can write pipes. We've worked out that this is what SELinux was preventing because the "denied" actions mentioned are `append`, `write`, and `open` for pipefs files on the VM.

Patching SELinux violations

Once you've decided that the violations you've seen are acceptable, there are tools that can automatically generate the policy file you need to apply, so you don't need to go through the pain and risk of writing one yourself. The following example uses the audit2allow tool to achieve this.

Listing 14.17 Creating a new SELinux policy

```
mkdir -p /root/selinux_policy_httpd_auto
 cd /root/selinux_policy_httpd_auto
audit2allow -a -w
 audit2allow -a -M newmodname create policy
 semodule -i newmodname.pp
```

- `mkdir -p ...`: Creates a fresh folder to store the new SELinux module
- `audit2allow -a -w`: Uses the audit2allow tool to display the policy that would be generated from reading the audit logs. Review this again to make sure it looks sensible.
- `audit2allow -a -M ...`: Creates your module with the -M flag and a name for the module you've chosen
- `semodule -i ...`: Installs the module from the newly created .pp file

It's important to understand that this new SELinux module we've created "includes" (or "requires") and alters the one we created before by referencing and adding permissions to the docker_apache_t type. You can combine the two into a complete and discrete policy in a single .te file if you choose.

Testing your new module

Now that you have your new module installed, you can try re-enabling SELinux and restarting the container.

> **TIP** If you couldn't set your host to permissive earlier (and you added the hashed-out line to your original docker_apache.te file), then recompile and reinstall the original docker_apache.te file (with the permissive line hashed-out) before continuing.

Listing 14.18 Starting a container with SELinux restrictions

```
docker rm -f selinuxdock
setenforce Enforcing
docker run -d --name selinuxdock \
--security-opt label:type:docker_apache_t httpd
docker logs selinuxdock
grep -w denied /var/log/audit/audit.log
```

There should be no new errors in the audit log. Your application has started within the context of this SELinux regime.

DISCUSSION

SELinux has a reputation for being complex and hard to manage, with the most frequently heard complaint being that it's more often switched off than debugged. That's hardly secure at all. Although the finer points of SELinux do require serious effort to master, we hope this technique has shown you how to create something that a security expert can review—and ideally sign off on—if Docker isn't acceptable out of the box.

Summary

- You can granularly control the power of root within your containers with capabilities.
- You can authenticate people using your Docker API via HTTP.
- Docker has built-in support for API encryption using certificates.
- SELinux is a well-tested way to reduce the danger of containers running as root.
- An application platform as a service (aPaaS) can be used to control access to the Docker runtime.