16 Docker in production: Dealing with challenges

This chapter covers

- Bypassing Docker's namespace functionality and using the host's resources directly
- Making sure your host OS doesn't kill processes in containers due to low memory
- Debugging a container's network directly, using your host's tooling
- Tracing system calls to determine why a container isn't working on your host

In this chapter we'll discuss what you can do when Docker's abstractions aren't working for you. These topics necessarily involve getting under the hood of Docker to understand why such solutions can be needed, and in the process we aim to provide you with a deeper awareness of what can go wrong when using Docker and how to go about fixing it.

16.1 Performance: You can't ignore the tin

Although Docker seeks to abstract the application from the host it's running on, one can never completely ignore the host. In order to provide its abstractions, Docker must add layers of indirection. These layers can have implications for your running system, and they sometimes need to be understood in order for operational challenges to be fixed or worked around.

In this section we'll look at how you can bypass some of these abstractions, ending up with a Docker container that has little of Docker left in it. We'll also show that although Docker appears to abstract away the details of the storage you use, this can sometimes come back to bite you.

TECHNIQUE 109 Accessing host resources from the container

We covered volumes, the most commonly used Docker abstraction bypass, in technique 34. They're convenient for sharing files from the host and for keeping larger files out of image layers. They can also be significantly faster for filesystem access than the container filesystem, as some storage backends impose significant overheads for certain workloads—this isn't useful for all applications, but it's crucial in some cases.

In addition to the overhead imposed by some storage backends, another performance hit comes about as a result of the network interfaces Docker sets up to give each container its own network. As with filesystem performance, network performance is definitely not a bottleneck for everyone, but it's something you may wish to benchmark for yourself (although the fine details of network tuning are very much outside the scope of this book). Alternatively, you may have other reasons to want to bypass Docker networking entirely—a server that opens random ports to listen on may not be well served by listening on port ranges with Docker, especially because exposing a range of ports will allocate them on the host whether they're in use or not.

Regardless of your reason, sometimes Docker abstractions get in the way, and Docker does offer the ability to opt out if you need to.

PROBLEM

You want to allow access to the host's resources from the container.

SOLUTION

Use the flags Docker offers for `docker run` to bypass the kernel namespace functionality that Docker uses.

> **TIP** Kernel namespaces are a service the kernel offers to programs, allowing them to get views of global resources in such a way that they appear to have their own separate instances of that resource. For example, a program can request a network namespace that will give you what appears to be a complete network stack. Docker uses and manages these namespaces to create its containers.

Table 16.1 summarizes how Docker uses namespaces, and how you can effectively switch them off.

Table 16.1 Namespaces and Docker

Kernel namespace	Description	Used in Docker?	"Switch off" option
Network	The network subsystem	Yes	`--net=host`
IPC	Inter-process communication: shared memory, semaphores, and so on	Yes	`--ipc=host`
UTS	Hostname and NIS domain	Yes	`--uts=host`
PID	Process IDs	Yes	`--pid=host`
Mount	Mount points	Yes	`--volume`, `--device`
User	User and group IDs	No	N/A

NOTE If any of these flags aren't available, it will likely be due to your version of Docker being out of date.

If your application is a heavy user of shared memory, for example, and you want to have your containers share this space with the host, you can use the `--ipc=host` flag to achieve this. This use is relatively advanced, so we'll focus on the other more common ones.

NETWORK AND HOSTNAME

To use the host's network, you run your container with the `--net` flag set to `host`, like this:

```
user@yourhostname:/$ docker run -ti --net=host ubuntu /bin/bash
root@yourhostname:/#
```

You'll notice that this immediately differs from a network-namespaced container in that the hostname within the container is the same as the host's. On a practical level, this can cause confusion, as it's not obvious that you're in a container.

In a network-isolated container, a quick `netstat` will show that there are no connections on startup:

```
host$ docker run -ti ubuntu
root@b1c4877a00cd:/# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
root@b1c4877a00cd:/#
```

A similar run using the host's network shows the usual network-busy host of a similarly busy technical author:

```
$ docker run -ti --net=host ubuntu
root@host:/# netstat -nap | head
Active Internet connections (servers and established)
```

```
Proto Recv-Q Send-Q Local Address   Foreign Address State       PID
➥ /Program name
tcp        0      0 127.0.0.1:47116 0.0.0.0:*       LISTEN      -
tcp        0      0 127.0.1.1:53    0.0.0.0:*       LISTEN      -
tcp        0      0 127.0.0.1:631   0.0.0.0:*       LISTEN      -
tcp        0      0 0.0.0.0:3000    0.0.0.0:*       LISTEN      -
tcp        0      0 127.0.0.1:54366 0.0.0.0:*       LISTEN      -
tcp        0      0 127.0.0.1:32888 127.0.0.1:47116 ESTABLISHED -
tcp        0      0 127.0.0.1:32889 127.0.0.1:47116 ESTABLISHED -
tcp        0      0 127.0.0.1:47116 127.0.0.1:32888 ESTABLISHED -
root@host:/#
```

NOTE `netstat` is a command that allows you to see information about networking on your local network stack. It's used most commonly to determine the state of network sockets.

The `net=host` flag is the most often used for a couple of reasons. First, it can make connecting containers much easier. But you lose the benefits of port mapping for your containers. If you have two containers that listen on port 80, for example, you can't run them on the same host in this way. The second reason is that network performance is significantly improved over Docker's when using this flag.

Figure 16.1 shows at a high level the layers of overhead a network packet must go through in Docker versus a native network. Whereas the native network need only go through the TCP/IP stack of the host to the network interface card (NIC), Docker has to additionally maintain a virtual Ethernet pair (a "veth pair"—a virtual representation of a physical connection via an Ethernet cable), a network bridge between this veth pair and the host network, and a layer of network address translation (NAT). This overhead can cause the Docker network to be half the speed of a native host network in normal use cases.

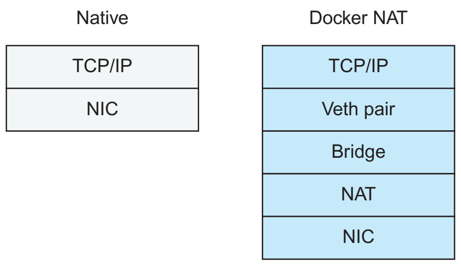

Figure 16.1 Docker networking vs. native networking

PID

The PID namespace flag is similar to the others:

The ps we're running is the only process in this container and is given the PID of 1.

Runs the ps command in a containerized environment, showing only the process that has a PID of 1

```
imiell@host:/$ docker run ubuntu ps -p 1
   PID TTY          TIME CMD
    1 ?        00:00:00 ps
 imiell@host:/$ docker run --pid=host ubuntu ps -p 1
   PID TTY          TIME CMD
    1 ?        00:00:27 systemd
```

Runs the same ps command with the PID namespace removed, giving us a view of the host's processes

This time the PID of 1 is the systemd command, which is the startup process of the host's operating system. This may differ for you, depending on your distribution.

The preceding example demonstrates that the systemd process of the host has process ID 1 in the container that has a view of the host PIDs, whereas without that view the only process seen is the `ps` command itself.

MOUNT

If you want access to the host's devices, use the `--device` flag to use a specific device, or mount the entire host's filesystem with the `--volume` flag:

```
docker run -ti --volume /:/host ubuntu /bin/bash
```

The preceding command mounts the host's `/` directory to the container's `/host` directory. You may be wondering why you can't mount the host's `/` directory to the container's `/` directory. This is explicitly disallowed by the `docker` command.

You may also be wondering whether you can use these flags to create a container that's virtually indistinguishable from the host. That leads us to the next section…

A HOST-LIKE CONTAINER

You can use the following flags to create a container that has an almost transparent view of the host:

Mounts the root filesystem of the host to a directory /host on the container. Docker disallows the mounting of volumes to the "/" folder, so you must specify the /host subfolder volume.

Runs a container with three host arguments (net, pid, ipc)

```
host:/$ docker run -ti --net=host --pid=host --ipc=host \
 --volume /:/host \
 busybox chroot /host
```

Starts up a BusyBox container. All you need is the chroot command, and this is a small image that contains that. Chroot is executed to make the mounted filesystem appear as the root to you.

It's ironic that Docker has been characterized as "`chroot` on steroids," and here we're using something characterized as a framework to run `chroot` in a way that subverts one of the principal purposes of `chroot`, which is to protect a host filesystem. It's usually at this point that we try not to think about it too hard.

In any case, it's hard to imagine a real-world use of that command (instructive as it is). If you think of one, please drop us a line.

That said, you might want to use it as a basis for more useful commands like this:

```
$ docker run -ti --workdir /host \
   --volume /:/host:ro ubuntu /bin/bash
```

In this example, `--workdir /host` sets the working directory on container startup to be the root of the host's filesystem, as mounted with the `--volume` argument. The `:ro` part of the volume specification means the host filesystem will be mounted as read-only.

With this command, you can give yourself a read-only view of the filesystem while having an environment where you can install tools (with the standard Ubuntu package manager) to inspect it. For example, you could use an image that runs a nifty tool that reports security problems on your host's filesystem, without having to install it on your host.

WARNING As the preceding discussion implies, using these flags opens you up to more security risks. In security terms, using them should be considered equivalent to running with the `--privileged` flag.

DISCUSSION

In this technique you've learned how to bypass Docker's abstractions within the container. Disabling these can give you speedups or other conveniences to make Docker better serve your needs. One variant we've used in the past is to install networking tools (perhaps like tcpflow, mentioned in technique 112) inside a container and expose host network interfaces. This lets you experiment with different tools on a temporary basis without having to install them.

The next technique looks at how you can bypass a restriction of Docker's underlying disk storage.

TECHNIQUE 110 Disabling the OOM killer

The "OOM killer" sounds like a bad horror film or severe disease, but it is in fact a thread within the Linux operating system kernel that decides what to do when the host is running out of memory. After the operating system has run out of hardware memory, used up any available swap space, and removed any cached files out of memory, it invokes the OOM killer to decide which processes should be killed off.

PROBLEM

You want to prevent containers from being killed by the OOM killer.

SOLUTION

Use the `--oom-kill-disable` flag when starting your container.

Solving this challenge is as simple as adding a flag to your Docker container. But as is often the case, the full story isn't that simple.

The following listing shows how you disable the OOM killer for a container:

Listing 16.1 `--oom-kill-disable` shows a warning

```
$ docker run -ti --oom-kill-disable ubuntu sleep 1
 WARNING: Disabling the OOM killer on containers without setting a
➥ '-m/--memory' limit may be dangerous.
```

The --oom-kill-disable flag is added to a normal docker run command. (first line)

A warning is output regarding another flag that might be set. (warning line)

The warning you see is important. It tells you that running with this setting is dangerous, but it doesn't tell you why. It's dangerous to set this option because if your host runs out of memory, the operating system will kill all other user processes before yours.

Sometimes that's desirable, such as if you have a critical piece of infrastructure you want to protect from failure—maybe an audit or logging process that runs across (or for) all containers on the host. Even then, you'll want to think twice about how disruptive this will be to your environment. For example, your container might depend on other running infrastructure on the same host. If you're running on a container platform like OpenShift, your container will survive even as key platform processes are killed off. You'd likely want that key infrastructure to stay up before that container.

Listing 16.2 `--oom-kill-disable` without a warning

This time, no warning is seen.

The --memory flag is added to a normal docker run command.

```
$ docker run -ti --oom-kill-disable --memory 4M ubuntu sleep 1
$
```

NOTE The minimum amount of memory you can allocate is 4M, where the "M" stands for megabytes. You can also allocate by "G" for gigabytes.

You may be wondering how to tell whether your container was killed by the OOM killer. This is easily done by using the `docker inspect` command:

Listing 16.3 Determining whether your container was "OOM-killed"

```
$ docker inspect logger | grep OOMKilled
            "OOMKilled": false,
```

This command outputs information about why the container was killed, including whether the OOM killer killed it.

DISCUSSION

The OOM killer doesn't require extended privileges to be set in a container, nor does it require you to be the root user—all you need is access to the `docker` command. This is yet another reason to be wary of giving unprivileged users access to the `docker` command without trusting them with root (see chapter 14 on security).

This is not only a security risk, but a stability risk too. If a user can run `docker`, they could run a process that gradually leaks memory (common in many production environments). If no boundaries are put on that memory, the operating system will step in once its options are exhausted and kill off the user process with the largest memory usage first (this is a simplification of the Linux OOM-killer algorithm, which has been battle-tested and grown over years). If the container has been started with the OOM killer disabled, however, it could trample over all containers on the host, causing far more destruction and instability for its users.

For a more fine-grained approach to memory management, you can adjust the container's "OOM score" with the `--oom-score-adj` flag. Another approach that may suit your purposes is to disable memory overcommit in the kernel. This has the effect of switching off the OOM killer globally, as memory is only granted if it's definitely available. However, this could limit the number of containers that can run on your hosts, which could also be undesirable.

As always, performance management is an art!

16.2 When containers leak—debugging Docker

In this section we'll cover some techniques that will help you understand and fix issues with applications running in Docker containers. We'll cover how to jump into a container's network while using tools from your host to debug issues, and we'll look at an alternative that avoids container manipulation by monitoring network interfaces directly.

Finally, we'll demonstrate how the Docker abstraction can break down, leading to containers working on one host and not another, and how to debug this on live systems.

TECHNIQUE 111 Debugging a container's network with nsenter

In an ideal world, you'd be able to use socat (see technique 4) in an *ambassador container* to diagnose issues with container communication. You'd start the extra container and make sure connections go to this new container, which acts as a proxy. The proxy allows you to diagnose and monitor the connections, and it then forwards them to the right place. Unfortunately it's not always convenient (or possible) to set up a container like this only for debugging purposes.

TIP See technique 74 for a description of the ambassador pattern.

You've already read about `docker exec` in techniques 15 and 19. This technique discusses *nsenter*, a tool that looks similar but allows you to use tools from your machine inside the container, rather than being limited to what the container has installed.

PROBLEM

You want to debug a network problem in a container, but the tools aren't in the container.

SOLUTION

Use nsenter to jump into the container's network but retain your host's tooling.

If you don't already have nsenter available on your Docker host, you can build it with the following command:

```
$ docker run -v /usr/local/bin:/target jpetazzo/nsenter
```

This will install nsenter in /usr/local/bin, and you'll be able to use it immediately. nsenter might also be available in your distro (in the util-linux package).

You may have noticed by now that the generally useful BusyBox image doesn't come with bash by default. As a demo of nsenter, we're going to show how you can enter a BusyBox container with your host's bash program:

```
$ docker run -ti busybox /bin/bash
FATA[0000] Error response from daemon: Cannot start container >
a81e7e6b2c030c29565ef7adb94de20ad516a6697deeeb617604e652e979fda6: >
```

Starts up a BusyBox container and saves the container ID (CID)

Inspects the container, extracting the process ID (PID) (see technique 30)

Runs nsenter, specifying the container to enter with the --target flag. The "sudo" may not be required.

Specifies the namespaces of the container to enter with the remaining flags

```
exec: "/bin/bash": stat /bin/bash: no such file or directory
$ CID=$(docker run -d busybox sleep 9999)
 $ PID=$(docker inspect --format {{.State.Pid}} $CID)
 $ sudo nsenter --target $PID \
 --uts --ipc --net /bin/bash
 root@781c1fed2b18:~#
```

See technique 109 for more detail on namespaces that nsenter understands. The critical point in the selection of namespaces is that you don't use the `--mount` flag, which would use the container's filesystem, because bash wouldn't be available. /bin/bash is specified as the executable to start.

It should be pointed out that you don't get direct access to the container's filesystem, but you do get all the tools your host has.

Something that we've needed before is a way to find out which veth interface device on the host corresponds to which container. For example, sometimes it's desirable to quickly knock a container off the network. An unprivileged container can't bring a network interface down, so you need to do it from the host by finding out the veth interface name.

We're unable to bring an interface in the container down. Note that your interface may not be eth0, so if this doesn't work, you may wish to use ip addr to find out your principal interface name.

Verifies that attempting to ping from inside a new container succeeds

Enters into the network space of the container, using the ethtool command from the host to look up the peer interface index—the other end of the virtual interface

Looks through the list of interfaces on the host to find the appropriate veth interface for the container

Brings down the virtual interface

Verifies that attempting to ping from inside the container fails

```
$ docker run -d --name offlinetest ubuntu:14.04.2 sleep infinity
fad037a77a2fc337b7b12bc484babb2145774fde7718d1b5b53fb7e9dc0ad7b3
$ docker exec offlinetest ping -q -c1 8.8.8.8
 PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.966/2.966/2.966/0.000 ms
$ docker exec offlinetest ifconfig eth0 down
 SIOCSIFFLAGS: Operation not permitted
$ PID=$(docker inspect --format {{.State.Pid}} offlinetest)
$ nsenter --target $PID --net ethtool -S eth0
 NIC statistics:
     peer_ifindex: 53
$ ip addr | grep '^53'
 53: veth2e7d114: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue >
master docker0 state UP
$ sudo ifconfig veth2e7d114 down
 $ docker exec offlinetest ping -q -c1 8.8.8.8
 PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

One final example of a program you might want to use from within a container is tcpdump, a tool that records all TCP packets on a network interface. To use it, you need to run `nsenter` with the `--net` command, allowing you to "see" the container's network from the host and therefore monitor the packets with tcpdump.

For example, the `tcpdump` command in the following code records all packets to the /tmp/google.tcpdump file (we assume you're still in the nsenter session you started previously). Some network traffic is then triggered by retrieving a web page:

```
root@781c1fed2b18:/# tcpdump -XXs 0 -w /tmp/google.tcpdump &
root@781c1fed2b18:/# wget google.com
--2015-08-07 15:12:04--  http://google.com/
Resolving google.com (google.com)... 216.58.208.46, 2a00:1450:4009:80d::200e
Connecting to google.com (google.com)|216.58.208.46|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ >
[following]
--2015-08-07 15:12:04--  >
http://www.google.co.uk/?gfe_rd=cr&ei=tLzEVcCXN7Lj8wepgarQAQ
Resolving www.google.co.uk (www.google.co.uk)... 216.58.208.67, >
2a00:1450:4009:80a::2003
Connecting to www.google.co.uk (www.google.co.uk)|216.58.208.67|:80... >
connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'index.html'

index.html             [ <=>              ]  18.28K  --.-KB/s   in 0.008s

2015-08-07 15:12:05 (2.18 MB/s) - 'index.html' saved [18720]

root@781c1fed2b18:# 15:12:04.839152 IP 172.17.0.26.52092 > >
google-public-dns-a.google.com.domain: 7950+ A? google.com. (28)
15:12:04.844754 IP 172.17.0.26.52092 > >
google-public-dns-a.google.com.domain: 18121+ AAAA? google.com. (28)
15:12:04.860430 IP google-public-dns-a.google.com.domain > >
172.17.0.26.52092: 7950 1/0/0 A 216.58.208.46 (44)
15:12:04.869571 IP google-public-dns-a.google.com.domain > >
172.17.0.26.52092: 18121 1/0/0 AAAA 2a00:1450:4009:80d::200e (56)
15:12:04.870246 IP 172.17.0.26.47834 > lhr08s07-in-f14.1e100.net.http: >
Flags [S], seq 2242275586, win 29200, options [mss 1460,sackOK,TS val >
49337583 ecr 0,nop,wscale 7], length 0
```

TIP Depending on your network setup, you may need to temporarily change your resolv.conf file to allow the DNS lookup to work. If you get a "Temporary failure in name resolution" error, try adding the line `nameserver 8.8.8.8` to the top of your /etc/resolv.conf file. Don't forget to revert it when you're finished.

DISCUSSION

This technique gives you a way to quickly alter the network behavior of containers without having to settle down with any of the tools from chapter 10 (techniques 78 and 79) to simulate network breakage.

You've also seen a compelling use case for Docker—it's much easier to debug network issues in the isolated network environment Docker provides than to do it in an uncontrolled environment. Trying to remember the correct arguments for tcpdump to appropriately filter out irrelevant packets in the middle of the night is an error-prone process. Using nsenter, you can forget about that and capture everything within the container, without tcpdump being installed (or having to install it) on the image.

TECHNIQUE 112 Using tcpflow to debug in flight without reconfiguring

tcpdump is the de facto standard in network investigation, and it's likely the first tool most people reach for if asked to dive into debugging a network issue.

But tcpdump is typically used for displaying packet summaries and examining packet headers and protocol information—it's not quite as full featured for displaying the application-level data flow between two programs. This can be quite important when investigating issues with two applications communicating.

PROBLEM

You need to monitor the communication data of a containerized application.

SOLUTION

Use tcpflow to capture traffic crossing an interface.

tcpflow is similar to tcpdump (accepting the same pattern-matching expressions) but it's designed to give you better insight into application data flows. tcpflow may be available from your system package manager, but, if not, we've prepared a Docker image you can use which should be virtually identical in functionality to an equivalent package manager install:

```
$ IMG=dockerinpractice/tcpflow
$ docker pull $IMG
$ alias tcpflow="docker run --rm --net host $IMG"
```

There are two ways you can use tcpflow with Docker: point it at the `docker0` interface and use a packet-filtering expression to retrieve only the packets you want, or use the trick from the previous technique to find the veth interface for the container you're interested in, and capture on that.

TIP You may wish to refer to figure 10.2 in chapter 10 to refresh your memory on how network traffic flows inside Docker and see why capturing on `docker0` will capture container traffic.

Expression filtering is a powerful feature of tcpflow to use after attaching to an interface, letting you drill down to the traffic you're interested in. We'll show a simple example to get you started:

```
$ docker run -d --name tcpflowtest alpine:3.2 sleep 30d
fa95f9763ab56e24b3a8f0d9f86204704b770ffb0fd55d4fd37c59dc1601ed11
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' tcpflowtest
172.17.0.1
$ tcpflow -c -J -i docker0 'host 172.17.0.1 and port 80'
tcpflow: listening on docker0
```

In the preceding example, you ask tcpflow to print a colorized stream of any traffic going to or from your container with a source or destination port of 80 (generally used for HTTP traffic). You can now try this by retrieving a web page in the container in a new terminal:

```
$ docker exec tcpflowtest wget -O /dev/null http://www.example.com/
Connecting to www.example.com (93.184.216.34:80)
null             100% |*******************************|  1270   0:00:00 ETA
```

You'll see colorized output in the tcpflow terminal. The cumulative output of the command so far will look something like this:

```
$ tcpflow -J -c -i docker0 'host 172.17.0.1 and (src or dst port 80)'
tcpflow: listening on docker0
172.017.000.001.36042-093.184.216.034.00080: >
GET / HTTP/1.1                                          <—— Blue coloring starts
 Host: www.example.com
User-Agent: Wget
Connection: close


093.184.216.034.00080-172.017.000.001.36042: >
HTTP/1.0 200 OK                                         <—— Red coloring starts
 Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Mon, 17 Aug 2015 12:22:21 GMT
[...]

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
[...]
```

DISCUSSION

tcpflow is an excellent addition to your toolbox, given how unobtrusive it is. You can start it against long-running containers to get a bit of insight into what they're transferring right now, or use it alongside tcpdump (the previous technique) to get a more complete picture of the kind of requests your application makes and what information is transferred.

As well as tcpdump, the previous technique also covers using nsenter to monitor traffic on just one container rather than all of them (which is what monitoring docker0 will do).

TECHNIQUE 113 Debugging containers that fail on specific hosts

The previous two techniques have shown how you can start investigating issues caused by the interaction between your containers and other locations (whether those "other locations" are more containers, or third parties on the internet).

If you've isolated a problem to one host, and you're sure that external interaction isn't the cause, the next step should be to try reducing the number of moving parts (removing volumes and ports) and to check the details of the host itself (free disk space, number of open file descriptors, and so on). It's probably also worth checking that each host is on the latest version of Docker.

In some cases, none of the above will help—you've got an image you can run with no arguments (such as `docker run imagename`) which should be perfectly contained, yet it runs differently on different hosts.

PROBLEM

You want to determine why a particular action within a container isn't working on a particular host.

SOLUTION

Strace the process to see what system calls it's making, and compare that to a working system.

Although Docker's stated aim is to allow users to "run any app anywhere," the means by which it tries to achieve this aren't always foolproof.

Docker treats the Linux kernel API as its *host* (the environment in which it can run). When they first learn how Docker works, many people ask how Docker handles changes to the Linux API. As far as we're aware, it doesn't yet. Fortunately, the Linux API is backwards-compatible, but it's not difficult to imagine a scenario in the future where a *new* Linux API call is created and used by a Dockerized application, and for that app to then be deployed to a kernel recent enough to run Docker but old enough to not support that particular API call.

> **NOTE** You may think that the Linux kernel API changing is something of a theoretical problem, but we came across this scenario while writing the first edition of this book. A project we were working on used the `memfd_create` Linux system call, which only exists on kernels versioned 3.17 and above. Because some hosts we were working on had older kernels, our containers failed on some systems and worked on others.

That scenario is not the only way in which the Docker abstraction can fail. Containers can fail on particular kernels because assumptions may be made by the application about files on the host. Although rare, it does happen, and it's important to be alert to that risk.

SELINUX INTERFERENCE WITH CONTAINERS

An example of where the Docker abstraction can break down is with anything that interacts with SELinux. As discussed in chapter 14, SELinux is a layer of security implemented in the kernel that works outside the normal user permissions.

Docker uses this layer to allow container security to be tightened up by managing what actions can be performed from within a container. For example, if you're root within a container, you're the same user as root on the host. Although it's hard to break out of the container so you obtain root on the host, it's not impossible; exploits have been found, and others may exist that the community is unaware of. What SELinux can do is provide another layer of protection so that even if a root user breaks out of the container to the host, there are limits on what actions they can perform on the host.

So far so good, but the problem for Docker is that SELinux is implemented on the host, and not within the container. This means that programs running in containers that query the status of SELinux and find it enabled might make certain assumptions about the environment in which they run, and fail in unexpected ways if these expectations aren't met.

In the following example, we're running a CentOS 7 Vagrant machine with Docker installed, and within that an Ubuntu 12.04 container. If we run a fairly straightforward command to add a user, the exit code is 12, indicating an error, and indeed the user has not been created:

```
[root@centos vagrant]# docker run -ti ubuntu:12.04
Unable to find image 'ubuntu:12.04' locally
Pulling repository ubuntu
78cef618c77e: Download complete
b5da78899d3a: Download complete
87183ecb6716: Download complete
82ed8e312318: Download complete
root@afade8b94d32:/# useradd -m -d /home/dockerinpractice dockerinpractice
root@afade8b94d32:/# echo $?
12
```

The same command run on an ubuntu:14.04 container works just fine. If you want to try to reproduce this result, you'll need a CentOS 7 machine (or similar). But for learning purposes, following the rest of the technique with any command and container will be sufficient.

TIP In bash, `$?` gives you the exit code of the last-run command. The meaning of the exit code varies from command to command, but typically an exit code of 0 means the call was successful, and a nonzero code indicates an error or exceptional condition of some kind.

DEBUGGING LINUX API CALLS

Because we know that the likely difference between the containers is due to differences between the kernel APIs running on the hosts, strace can help you determine the differences between calls to the kernel API.

strace is a tool that allows you to snoop on the calls made to the Linux API by a process (a.k.a. system calls). It's an extremely useful debugging and educational tool. You can see how it works in figure 16.2.

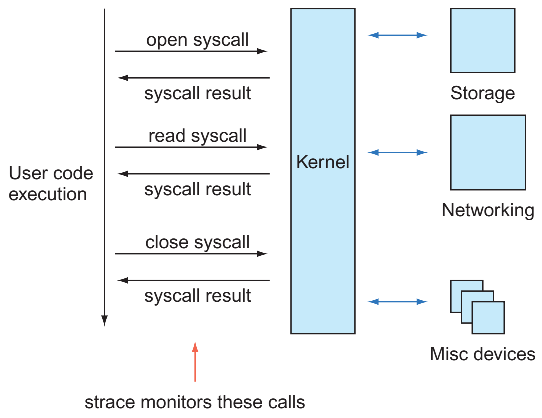

Figure 16.2 How strace works

First, you need to install strace on your container using the appropriate package manager, and then run the command that differs, with the `strace` command prepended. Here's some example output for the failed `useradd` call:

```
# strace -f \
 useradd -m -d /home/dockerinpractice dockerinpractice
 execve("/usr/sbin/useradd", ["useradd", "-m", "-d", >
 "/home/dockerinpractice", "dockerinpractice"], [/* 9 vars */]) = 0
[...]
open("/proc/self/task/39/attr/current", >
 O_RDONLY) = 9
read(9, "system_u:system_r:svirt_lxc_net_"..., >
 4095) = 46
close(9)                              = 0
 [...]
open("/etc/selinux/config", O_RDONLY)    = >
 -1 ENOENT (No such file or directory)
open("/etc/selinux/targeted/contexts/files/ >
 file_contexts.subs_dist", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/selinux/targeted/contexts/files/ >
 file_contexts.subs", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/etc/selinux/targeted/contexts/files/ >
 file_contexts", O_RDONLY) = -1 ENOENT (No such file or directory)
[...]
exit_group(12)
```

Runs strace on the command with the -f flag, which ensures that any process spawned by your command and any of its descendants are "followed" by strace

Appends the command you want to debug to the strace invocation

Each line of the strace output starts with the Linux API call. The execve call here executes the command you gave strace. The 0 at the end is the return value from the call (successful).

The "read" system call works on the previously opened file (with file descriptor 9) and returns the number of bytes read (46).

The "close" system call closes the file referenced with the file descriptor number.

The process exits with the value 12, which for useradd means that the directory couldn't be created.

The "open" system call opens a file for reading. The return value (9) is the file handle number used in subsequent calls to work on the file. In this case, the SELinux information is retrieved from the /proc filesystem, which holds information about running processes.

The program attempts to open the SELinux files it expects to be there, but in each case fails. strace helpfully tells you what the return value means: "No such file or directory."

The preceding output may seem confusing at first, but after a few times it becomes relatively easy to read. Each line represents a call to the Linux kernel to perform some action in what's known as *kernel space* (as opposed to *user space*, where actions are performed by programs without handing over responsibility to the kernel).

TIP If you want to learn more about a specific system call, you can run `man 2 callname`. You may need to install the man pages with `apt-get install manpages-dev` or a similar command for your packaging system. Alternatively, Googling "man 2 callname" will likely get you what you need.

This is an example of where Docker's abstractions break down. In this case, the action fails because the program expects SELinux files to be present, because SELinux appears to be enabled on the container, but the details of enforcement are kept on the host.

TIP It's incredibly useful to read over the `man 2` pages for all the system calls if you're serious about being a developer. At first they might seem full of jargon you don't understand, but as you read around the various subjects, you'll learn a great deal about fundamental Linux concepts. At some point, you'll start to see how most languages derive from this root, and some of their quirks and oddities will make more sense. Be patient, though, as you won't understand it all immediately.

DISCUSSION

Although such situations are rare, the ability to debug and understand how your program is interacting by using strace is an invaluable technique, not only with Docker but for more general development.

If you have very minimal Docker images, perhaps created by leveraging technique 57, and would prefer not to install strace on your container, it's possible to use strace from your host. You'll want to use `docker top <container_id>` to find the PID of the process in the container, and the `-p` argument to strace to attach to a specific running process. Don't forget to use sudo. Attaching to a process potentially allows you to read its secrets, so it requires extra permissions.

TECHNIQUE 114 Extracting a file from an image

Copying a file from a container is easily achieved using the `docker cp` command. Not infrequently, you'll want to extract a file from an image, but you don't have a clean container running to copy from. In these cases, you can artificially run a container of the image, run `docker cp`, and then remove the container. This is already three commands, and you may run into trouble if, for example, the image has a default entrypoint that demands a meaningful argument.

This technique gives you a single command alias that you can put into your shell startup scripts to do all this with one command and two arguments.

PROBLEM

You want to copy a file from an image to your host.

SOLUTION

Use an alias to run a container from the image with an entrypoint to `cat` the file's contents to a file on the host.

First we'll show you how to construct a `docker run` command to extract a file from an image, and then you'll see how to turn this into an alias for convenience.

Listing 16.4 Extracting a file from an image using `docker run`

Uses the --rm flag to delete the container immediately on running this command

Uses the -i flag to make the container interactive

Uses the -t flag to give the container a virtual terminal to write to

Sets the entrypoint for the container to 'cat'

The name of the image you want to extract the file from

The filename to output

Redirects the contents of the file to a local file on the host

```
$ docker run --rm \
    -i \
    -t \
    --entrypoint=cat \
    ubuntu \
    /etc/os-release \
    > ubuntu_os-release
 $ cat ubuntu_os-release
NAME="Ubuntu"
VERSION="16.04.1 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.1 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
$ cat /etc/os-release
 cat: /etc/os-release: No such file or directory
```

To emphasize the point, we show that the /etc/os-release doesn't exist on the host.

You might be wondering why we use `entrypoint` here, and don't simply run the `cat` command to output the file. This is because some images will have set an entrypoint already. When this happens, docker would treat `cat` as the argument to the `entrypoint` command, resulting in behavior you wouldn't want.

For convenience, you might want to put this command into an alias.

Listing 16.5 Using an alias to extract a file from an image

Aliases the command to the name "imagecat", containing everything in the command from listing 16.4 up to the image and file arguments

Calls "imagecat" with the two arguments (image and filename)

```
$ alias imagecat='docker run --rm -i -t --entrypoint=cat'
 $ imagecat ubuntu /etc/os-release
 NAME="Ubuntu"
VERSION="16.04.1 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.1 LTS"
VERSION_ID="16.04"
```

```
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

This technique assumes the presence of `cat` in your containers. If you've been building minimal containers with technique 58, this may not be the case, as only your binary is present in the container—there are no standard Linux tools.

If that's the case, you'll want to consider using `docker export` from technique 73, but rather than sending them to another machine, you can just extract the file you want from them. Bear in mind that a container doesn't need to successfully start for you to export it—you can attempt to run it with a command that doesn't exist inside the container and then export the stopped container (or just use `docker create`, which prepares a container for execution without starting it).

Summary

- You can pass arguments to Docker to disable different kinds of isolation, either for greater flexibility of containers or for performance.
- You can disable the Linux OOM killer for individual containers to indicate that Linux should never try to reclaim limited memory by killing this process.
- nsenter can be used to get access to the network context of a container from the host.
- tcpflow allows you to monitor all traffic in and out of your containers without needing to reconfigure or restart anything.
- strace is a vital tool for identifying why a Docker container isn't working on a specific host.

This concludes the book! We hope we've opened your eyes to some of the uses of Docker and given you some ideas for integrating it in your company or personal projects. If you'd like to get in touch with us or give us some feedback, please create a thread in the Manning *Docker in Practice* forum (https://forums.manning.com/forums/docker-in-practice-second-edition) or raise an issue against one of the "docker-in-practice" GitHub repositories.