Jailing Programs with Linux Containers

Jay Beale
InGuardians
@jaybeale / jay.beale@inguardians.com
@inguardians

Jailing Processes

Chroot jails have been the best practice in Linux and UNIX for A Long Time™.

Chroot locks a process into a subset of the filesystem, by making a specific directory into the process' root.

If the process either starts with root privilege or can escalate to root privilege, an attacker can break out of the chroot trivially.

Breaking Chroot

If you have root privilege, breaking out of a chroot jail is simple.

```
#include <sys/stat.h>
#include <unistd.h>

int main() {
    mkdir("inguardians", 0755);
    chroot("inguardians");
    chroot("../../../../../../../../../../..");
    return execl("/bin/sh", "-i", NULL);
}
```

Linux Containers

Linux Containers in general, and Docker in particular, are an evolving technology.

At this stage, containers can certainly do a better job than a chroot.

Linux containers revolve around **namespaces** and **control groups**.

Namespaces

The chroot created a kind of filesystem namespace.

Containers bring even more types of namespaces:

- **PID** – process isolation
- **Network** – allows for differing network cards, IP addresses, routing tables, …
- **UTS** – allows different hostnames
- **Mount** – allows differing filesystem layouts/ properties
- **IPC** – isolates interprocess communication
- **User** – separate unique UID mapping, including root

Control Groups

Control groups (cgroups) were initially created to allow a system owner to set resource utilization limits on groups of processes. More specifically:

Resource Limitation: RAM and Swap limited by cgroup

Prioritization – CPU and disk I/O can favor a cgroup

Accounting – track utilization by group

Control – freezing processes, checkpointing, restarting

All of this is focused on dealing with resource accounting and control.

Abstraction: Containers vs VM's

Containers are the next evolutionary step in putting multiple "workloads" on the same hardware. Virtual machines were the previous step.

A virtual machine has its own kernel, core subsystems (syslog, cron, udev..) and far more running processes than one needs, just to separate one app from another.

Containers eliminate that duplicate kernel and can eliminate the other redundant processes.

Multi-tenancy

Many companies use containers for multi-tenancy.

I'm still a bit uncomfortable with even using virtual machine hypervisors for multi-tenancy.

My purpose here is to use containers to:

1) gain a level of containment within a machine/VM
2) develop and test code with speed, ease, and joy

Container Administration

There are a number of ways to manage containers, including Docker, LXC, LXD, and OpenVZ.

This talk first focuses on Docker, because of its ease and market leadership.

It also introduces LXD-managed containers because:

- LXD competes with Docker, forcing both to innovate.
- LXD can gateway us to LXC containers, which require no management daemon

Docker Concepts

Containers are the jails that Docker helps create and facilitate. A kind of "lightweight virtual machine."

Images are the persistent state of a container. They contain the filesystems and configuration.

In Docker, an image is made up of one or more **union-mounted filesystems**, where each layer overlays the filesystem below, overruling only those files it brings. Only the top layer in an image is read-write.

Demo Format

This talk uses demos, but we want the slides to be a reference you can use.

For this reason, every demo is both represented by an interaction and a set of slides from which you can copy-paste.

Docker Quickstart

- We can start using Docker by executing a single command:

```
docker run -it centos:7 /bin/bash
```

- This pulls an official Centos 7 image from Dockerhub, starts a container based on it, running only `/bin/bash`.
- Once the container starts, we'll get a shell. Try a `ps`:

```
[root@34f508fba5df /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 21:59 ?        00:00:00 /bin/bash
root          24       1  0 21:59 ?        00:00:00 ps -ef
```

Detach and Investigate

- Let's detach from the image with Ctrl-P-Q
- Next, run docker ps to see running containers

```
[root@localhost 73115]# docker ps
CONTAINER ID    IMAGE              COMMAND       CREATED
    STATUS              PORTS               NAMES
34f508fba5df    7322fb...:latest   "/bin/bash"   8 minutes ago
    Up 8 minutes                            hungry_pike
```

- This container is called 34f508..., but it's also called "hungry_pike".
- Its image is 7332fb...

Creating a Second Container

- Let's create a change in this container.

```
# docker attach hungry_pike
[root@34f508fba5df /]# echo "jay" >foo
```

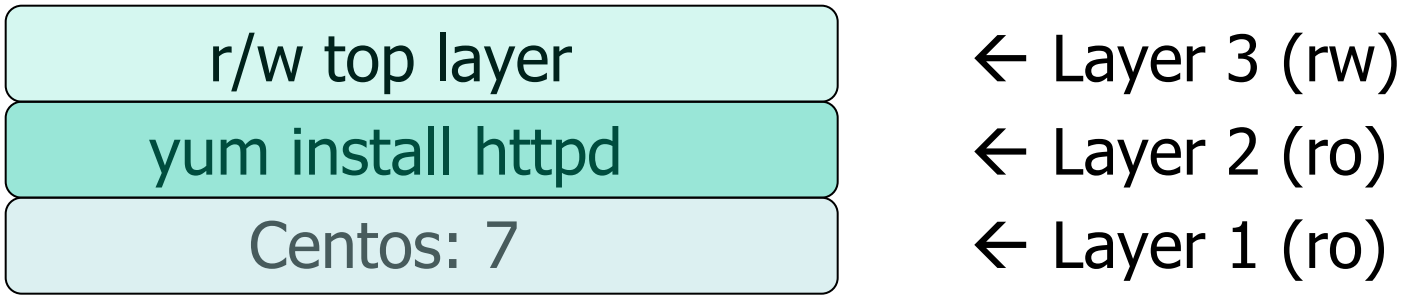
- Detach and start another container based on its image.

```
# docker run -it 7322fbe74aa5632b33a400959867c8ac4290e9c51 /bin/bash
[root@e1bf3790cc9e /]# ls
bin  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc
root  run  sbin  srv  sys  tmp  usr  var
[root@e1bf3790cc9e /]# echo "no jay here" >foo
```

- Detach and investigate both containers. They each have their own version of the /foo file.

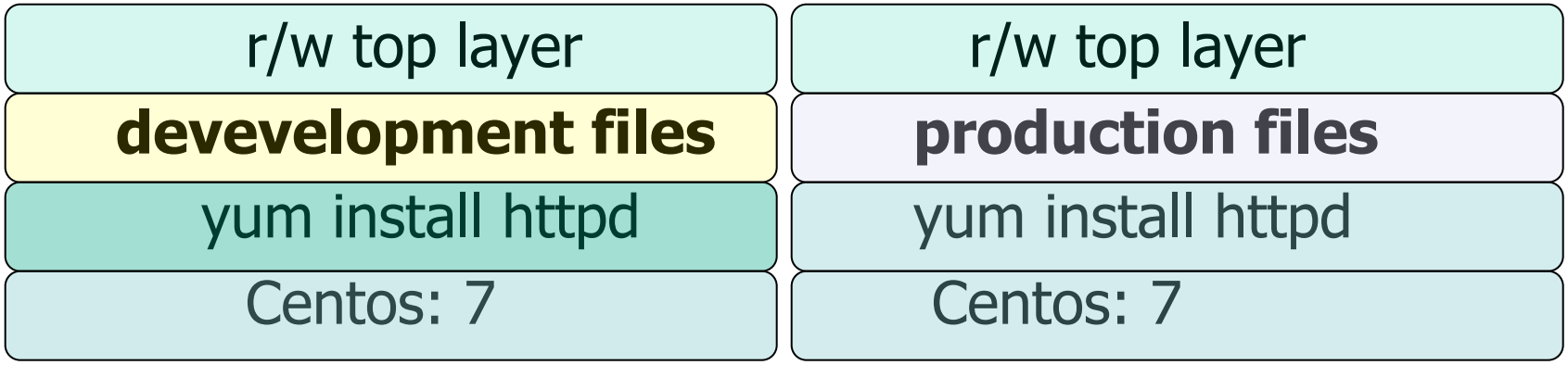
Docker Images

- A Docker image is made up of multiple layers
- Each layer is called an image.



We can now build another image from layer 1 and 2, without changing them.

Layer Re-Use



Persisting the Container FS

- Unless we commit this image, it's not persistent.
- Let's commit the container's filesystem changes to an image.

```
# docker stop hungry_pike
# docker commit hungry_pike foo_is_jay
f2e7485f4d88544dacc4bb5476a24211fef4f3f5101aeef31ab13d3d866e2c91
```

- Now destroy the two containers.

```
# docker ps -a
CONTAINER ID        IMAGE                          COMMAND                CREATED
      STATUS                          PORTS               NAMES
e1bf3790cc9e        7322fbe74aa5632b…:latest    "/bin/bash"            31 minutes ago
      Exited  (137) 3 minutes ago                         sharp_yalow
34f508fba5df        7322fbe74aa5632b…:latest    "/bin/bash"            48 minutes ago
      Exited (137) 4 minutes ago                        hungry_pike
# docker rm sharp_yalow hungry_pike
```

Re-Use the Image

- Let's start a new container from the image.

```
# docker run -it foo_is_jay /bin/bash
[root@869793b6611e /]# ls
bin  dev  etc  foo  home  lib  lib64  lost+found  media  mnt  opt
proc  root  run  sbin  srv  sys  tmp  usr  var
[root@869793b6611e /]# cat foo
jay
```

- Detach and take a look at docker ps:

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED             STATUS
PORTS               NAMES
869793b6611e        foo_is_jay:latest   "/bin/bash"         5 minutes ago       Up 5 minutes
focused_bartik
```

Images and Repositories

- Look at a list of the images.

```
# docker images
```

- Commit an image to a repository

```
# docker commit <container> <repo>[:tag]
```

- Pull an image from a repository

```
# docker pull repo[:tag]
```

Observe the Overlay

Let's see how the overlay works.

```
# docker history foo_lacks_jay
```

IMAGE	CREATED	CREATED BY	SIZE
f2e7485f4d88	12 minutes ago	/bin/bash	**4 B**
7322fbe74aa5	4 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	**0 B**
c852f6d61e65	4 weeks ago	/bin/sh -c #(nop) ADD file:82835f82606420c764	172.2 MB
f1b10cd84249	12 weeks ago	/bin/sh -c #(nop) MAINTAINER The CentOS Proje	0 B

Inspect the Container

`docker inspect` gives you information about the container:

```
# docker inspect focused_bartik
[{
 "Config": {
"Cmd": [
            "/bin/bash"
"Hostname": "9426cbdfb662",
        "Image": "foo_is_jay",
 "Name": "/focused_bartik ",
    "NetworkSettings": {
        "Bridge": "docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.1"
```

Dockerfile's

- Let's create our own Dockerfile, then build it.

```
# ln –s Dockerfile-2* Dockerfile
# cat Dockerfile

FROM  centos:7
RUN   yum update –y && yum install –y httpd
EXPOSE  80/tcp
ENTRYPOINT   ["/usr/sbin/httpd"]
CMD ["-D","FOREGROUND"]
```

Building our Image

Let's build an image from that Dockerfile.

```
# docker build -t myimage .
Sending build context to Docker daemon 265.7 MB
…
Step 0 : FROM centos:7
 ---> 7322fbe74aa5
Step 1 : RUN yum update -y
 ---> Running in 849c8aa1931e
Complete!
 ---> 5c7b076b3015
Removing intermediate container ee35de591aa3
…
Step 3 : ENTRYPOINT /usr/sbin/httpd
 ---> Running in f07febdc721d
…
Removing intermediate container 92caf64ee809
Successfully built 844fd895bca4
```

Starting our Container

- Now let's launch a container from our image.
- First, list the images.

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL
SIZE
myimage             latest              844fd895bca4        2 minutes ago       269.5 MB
foo_is_jay          latest              9843d10249ab        19 hours ago        172.2 MB
```

- Start a container based on 89fb0290e248 AKA "myimage."

```
# docker run -d --name="mycontainer" myimage
a4a4f29ba888ff86325d68e96194ba6ebfb01beee86c...7807
```

- From the Docker host, surf to the container's IP address.

Examining the Logs

We can see the logs from the container with `docker logs`.

```
# docker logs mycontainer
AH00558: httpd: Could not reliably determine the server's fully
qualified domain name, using 172.17.0.10. Set the 'ServerName'
directive globally to suppress this message
```

Another useful command is `docker logs -f` which works the same way as `tail -f`.

Let's look in our container with `docker exec`.

Getting Inside the Container

We can add a process to a container with docker exec.

```
# docker exec -it mycontainer /bin/bash
[root@a4a4f29ba888 /]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
apache         5       1  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
apache         6       1  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
apache         7       1  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
apache         8       1  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
apache         9       1  0 18:32 ?        00:00:00 /usr/sbin/httpd -D FOREGROUND
root          10       0  0 18:37 ?        00:00:00 /bin/bash
root          26      10  0 18:37 ?        00:00:00 ps -ef
```

You can exit this without killing the container.

Publishing the Program's Ports

Remember that EXPOSE entry in the Dockerfile?

We can reach that port from the Docker host, but nowhere else.

If we want to **publish** the port to the outside world, add a **-p** argument to the docker run.

```
# docker run -d –p 8123:80 --name=webserver myimage
```

This forwards the host's external 8123/tcp to the container's port 80.

Logging with Syslog

- Docker doesn't log to syslog by default. In fact, it doesn't even have a /dev/log device! Let's add that.

```
# docker run -v /dev/log:/dev/log -it foo_is_jay /bin/bash
[root@9426cbdfb662 /]# logger "Log from the container"

# grep logger /var/log/messages
Jul 19 16:09:14 localhost logger: Log from the container
```

Volume Mounts

- Wait, what was that `-v` argument to `docker run`?

```
# docker run -v /dev/log:/dev/log -it foo_is_jay /bin/bash
```

- This shared the host's `/dev/log` with the container.
- In general, the syntax is:

```
-v /host_dir:/container_dir
```

- This shares the `/host_dir` directory from the host into the container's `/container_dir`.

IPTABLES in Docker

Docker creates iptables rules by itself, like this:

```
NAT Table:
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE

FILTER Table:
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j
ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
```

IPTABLES: Port Publishing

- When we published a port, it added these two rules:

```
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 8123 -j
DNAT --to-destination 172.17.0.11:80
-A DOCKER -d 172.17.0.11/32 ! -i docker0 -o docker0 -p
tcp -m tcp --dport 80 -j ACCEPT
```

- You can configure this with two Docker daemon command-line options, both of which default to true.
  - `-- icc=false` stop inter-container communications
  - `-- iptables=false` iptables should be manual, not automatic

Container without Root

```
# cat Dockerfile
FROM centos:7
RUN yum update -y
RUN yum install -y httpd
RUN cat /etc/httpd/conf/httpd.conf  | sed 's/Listen 80/Listen 8000/'
>/etc/httpd/conf/httpd.conf.2
RUN mv -f /etc/httpd/conf/httpd.conf.2 /etc/httpd/conf/httpd.conf
RUN chown -R apache /etc/httpd/ /var/run/httpd/ /var/log/httpd/
EXPOSE 8000/tcp

# docker build -t webunpriv .
# docker run -d -p 80:8000 –u apache webunpriv
```

Docker Root Capabilities

- Docker drops all root capabilities except:
  - CHOWN - Make arbitrary changes to file UIDs and GIDs (see **chown**(2)).
  - DAC_OVERRIDE - Bypass file read, write, and execute permission checks
  - FSETID - Don't clear set-user-ID and set-group-ID permission bits when a file is modified
  - FOWNER - Bypass perm checks on operations, set ACLs, …
  - MKNOD - Create special files using **mknod**(2)
  - NET_RAW - use RAW and PACKET sockets; bind to any address for transparent proxying.
  - SETGID - Make arbitrary manipulations of process GIDs
  - SETUID - Make arbitrary manipulations of process UIDs
  - SETFCAP - Set file capabilities.
  - SETPCAP - related to file capabilities
  - NET_BIND_SERVICE - Bind a socket to Internet domain privileged ports (<1024).
  - SYS_CHROOT - Use **chroot**(2).
  - KILL - Bypass permission checks for sending signals (see **kill**(2)).
  - AUDIT_WRITE - Write records to kernel auditing log.

Observe a Dropped Capability

Start a root container. Try an iptables command.

Dropping More Capabilities

(Demos end for now)

You can control what capabilities Docker retains from these, or add to these, by using `docker run --cap-add` and `--cap-drop`.

This would drop all capabilities except `net_bind_service`, which lets us bind to a privileged (<1024) port.

```
docker run --cap-drop ALL --cap-add net_bind_service image /bin/bash
```

Exercise: try running a root shell in a container with no capabilities.

Capabilities Documentation

To read more about Linux capabilities, consult:

```
man 7 capabilities
```

SELinux

On Red Hat, Docker'd programs all run with s_virt types, which are able to read only Docker-related files.

Each container gets its own MCS compartment:

```
system_u:system_r:svirt_lxc_net_t:s0:c326,c871 process1
system_u:system_r:svirt_lxc_net_t:s0:c286,c581 process2
```

You can specialize SELinux confinement more by creating your own svirt subset types, then start containers like so:

```
docker run -d -p 80:8000 --security-opt label:type:svirt_apache_t web
```

Docker and OSSEC

- How would you apply host-based intrusion detection here?
- We can apply OSSEC in two different ways:
  - Run OSSEC in each container.
  - Monitor the logs and filesystem from outside all the containers.
- The latter approach is safer and aggregates better.
- OSSEC's primary focus is a log-based IDS.
  - Containers can log to a central syslog on the Docker host.
- What about file integrity checking?
  - Use volume mounts to share the container's filesystem.

Container Efficiency

- Just to get a feel for how fast you can start containers vs virtual machines, try starting about 20.

```
for ltr in a b c d e f g h i j k l m n o p q r s t u v ; do
 docker run -d --name container$ltr myimage ; done
docker ps
for ltr in a b c d e f g h i j k l m n o p q r s t u v ; do
 docker stop container$ltr ; done
for ltr in a b c d e f g h i j k l m n o p q r s t u v ; do
 docker rm container$ltr ; done
```

Docker Man Pages

- When in doubt, read the docs. Each of these is a man page!

```
docker-attach(1)    Attach to a running container
docker-build(1)  Build an image from a Dockerfile
docker-commit(1) Create a new image from a container's changes
docker-cp(1)     Copy files/folders from a container's filesystem to the host
docker-create(1) Create a new container
docker-diff(1)   Inspect changes on a container's filesystem
docker-events(1) Get real time events from the server
docker-exec(1)   Run a command in a running container
docker-export(1) Stream the contents of a container as a tar archive
docker-history(1)   Show the history of an image
docker-images(1) List images
docker-import(1) Create a new filesystem image from the contents of a tarball
docker-info(1)   Display system-wide information
```

Docker Man Pages: 2 of 3

docker-inspect(1) Return low-level information on a container or image
docker-kill(1) Kill a running container (which includes the wrapper process and everything inside it)
docker-load(1) Load an image from a tar archive
docker-login(1) Register or login to a Docker Registry Service
docker-logout(1) Log the user out of a Docker Registry Service
docker-logs(1) Fetch the logs of a container
docker-pause(1) Pause all processes within a container
docker-port(1) Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
docker-ps(1) List containers
docker-pull(1) Pull an image or a repository from a Docker Registry Service
docker-push(1) Push an image or a repository to a Docker Registry Service
docker-restart(1) Restart a running container
docker-rm(1) Remove one or more containers
docker-rmi(1) Remove one or more images
docker-run(1) Run a command in a new container

Docker Man Pages: 3 of 3

docker-save(1)	Save an image to a tar archive
docker-search(1)	Search for an image in the Docker index
docker-start(1)	Start a stopped container
docker-stats(1)	Display a live stream of one or more containers' resource usage statistics
docker-stop(1)	Stop a running container
docker-tag(1)	Tag an image into a repository
docker-top(1)	Lookup the running processes of a container
docker-unpause(1)	Unpause all processes within a container
docker-version(1)	Show the Docker version information
docker-wait(1)	Block until a container stops, then print its exit codeindex

Docker Cheat Sheet

- docker run -it <image> [<command>]
- docker run -d <image> [<command>]
- docker run -it —name <container> <image>
- docker run -d -u <user> <image>
- docker run -p <hostport>:<container_port> -it <image> <command>
- docker run -it —cap-drop ALL —cap-add net_bind_service <image> <command>
- docker commit <container> <repo/image_name>[:<tag>]
- docker exec -it <container> <command>
- docker images
- docker stop <container>
- docker pull <repo>[:<tag>]
- docker rm <container>
- docker rmi <image>
- docker ps
- docker ps –a
- docker history
- docker inspect

- docker logs
- docker logs -f
- docker -v <host_dir>:<container_dir>
- docker -d
- docker -d -l <debug|info|error|fatal> >>logfile >&1
- docker -d —icc=false —iptables=false
- docker build -t <image> .

LXD

Canonical has been building its own container manager, LXD.

LXD, pronounced Lex-Dee, plugs into OpenStack and offers a REST API.

We'll control LXD with the lxc CLI program.

LXD is at version 0.25, though the underlying containers it uses are mature. Prior to LXD, you created containers by downloading templates and using them to build images.

Installing LXD

Get an Ubuntu version >= 14.04.

Set up to pull container images from LinuxContainers.org:

```
lxc remote add images images.linuxcontainers.org
lxc image list images:
```

Launch your first container:

```
lxc launch images:ubuntu/trusty/amd64 ubuntu
```

* For the very first container you may need to use the lxd-images script.

Installing LXD

This may take some time, as we pull the container across the Internet. Let's see what images we have cached.

```
$ lxc image list
```

```
-------+--------------+--------+-------------------------------------+--------+----------+------------------------------+
| ALIAS | FINGERPRINT  | PUBLIC |            DESCRIPTION              |  ARCH  |   SIZE   |         UPLOAD DATE          |
+-------+--------------+--------+-------------------------------------+--------+----------+------------------------------+
|       | 0afd3f6ac0d7 | no     | Ubuntu 14.04 LTS server (20151218)  | x86_64 | 118.24MB | Jan 10, 2016 at 2:31pm (PST) |
|       | 58897960204b | no     | Debian wheezy (amd64)               | x86_64 | 97.16MB  | Jan 10, 2016 at 7:04pm (PST) |
|       | c1e88a2e8681 | no     | Ubuntu wily (amd64)                 | x86_64 | 72.56MB  | Jan 10, 2016 at 6:34pm (PST) |
|       | ed679a91fc8c | no     | Centos 6 (amd64)                    | x86_64 | 49.75MB  | Jan 10, 2016 at 6:51pm (PST) |
|       | f60c0d925ae1 | no     | Centos 7 (amd64)                    | x86_64 | 59.25MB  | Jan 10, 2016 at 6:55pm (PST) |
+-------+--------------+--------+-------------------------------------+--------+----------+------------------------------+
```

Let's launch a Centos 7 box.

```
$ lxc launch f60c0d925ae1 c7
```

List Running Containers

Let's see what containers are running:

```
$ lxc list
+------------------+---------+------------------+------+-----------+-----------+
|       NAME       |  STATE  |       IPV4       | IPV6 | EPHEMERAL | SNAPSHOTS |
+------------------+---------+------------------+------+-----------+-----------+
| c7               | RUNNING | 10.0.3.27 (eth0) |      | NO        |         0 |
+------------------+---------+------------------+------+-----------+-----------+
| ubuntu-wily-64   | STOPPED |                  |      | NO        |         0 |
+------------------+---------+------------------+------+-----------+-----------+
```

Investigating a Container

Let's learn a bit more about our new Centos 7 system.

```
$ lxc info c7
Name: c7
Status: Running
Init: 5759
Processcount: 7
Ips:
  eth0: IPV4    10.0.3.27       veth79CT9K
  lo:   IPV4    127.0.0.1
  lo:   IPV6    ::1
```

Configuration as YAML

LXD can export configurations as YAML, though the REST API exports JSON.

```
$ lxc config show c7
name: c7
profiles:
- default
config:
  volatile.base_image:
0afd3f6ac0d751fb121ad9a77a163926208ee71c57d68bd75cc253ce2c733a60
  volatile.eth0.hwaddr: 00:16:3e:38:f8:e1
  volatile.eth0.name: eth0
  volatile.last_state.idmap: '[{"Isuid":true,"Isgid":false,"Hostid":
165536,"Nsid":0,"Maprange":65536},{"Isuid":false,"Isgid":true,"Hostid":
165536,"Nsid":0,"Maprange":65536}]'
devices: {}
ephemeral: false
```

Enter the Container

Let's add a process to that running container, say, a shell.

```
$ lxc exec c7 /bin/bash
[root@c7 ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 00:05 ?        00:00:00 /sbin/init
root         105       1  0 00:05 ?        00:00:00 /usr/lib/systemd/systemd-journald
dbus         217       1  0 00:05 ?        00:00:00 /bin/dbus-daemon --system --
address=systemd: --nofork --nopidfile --systemd-ac
root         240       1  0 00:05 ?        00:00:00 /usr/sbin/rsyslogd -n
root         256       1  0 00:05 ?        00:00:00 /usr/lib/systemd/systemd-logind
root         438       1  0 00:05 ?        00:00:00 /sbin/dhclient -H c7 -1 -q -lf /var/lib/
dhclient/dhclient--eth0.lease -pf /v
root         866       1  0 00:12 ?        00:00:00 /sbin/agetty --noclear --keep-baud console
115200 38400 9600 vt220
root         875       0  0 00:12 ?        00:00:00 /bin/bash
root         884     875  0 00:12 ?        00:00:00 ps -ef
```

More Processes than Docker

We immediately see that this isn't a single program container, the way Docker's are intended to be.

```
[root@c7 ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 00:05 ?        00:00:00 /sbin/init
root         105       1  0 00:05 ?        00:00:00 /usr/lib/systemd/systemd-journald
```

Does this mean we're running root processes in the real system?

```
jay@ubuntu:~$ ps -ef | grep init
165536     5759   5718  0 16:05 ?        00:00:00 /sbin/init
```

User Namespaces

init isn't running as UID 0 for real!

It's running as UID 165536.

This number is defined in /etc/subuid and /etc/subgid.

```
$ cat /etc/subuid
jay:100000:65536
lxd:165536:65536
root:165536:65536
mike:231072:65536
```

Snapshot and Publish

We can snapshot and restore LXC containers.

```
$ lxc snapshot c7 snapshot1
$ lxc exec c7 -- touch /file
$ lxc exec c7 -- ls –l /file
$ lxc restore c7 snapshot1
$ lxc exec c7 -- ls –l /file
```

We can publish an image to a remote LXD server or to our local one.

```
$ lxc publish c7/snapshot1 --alias="c7-snapshot1"
$ lxc delete c7
$ lxc launch c7-snapshot1
```

Launching Machines Remotely

If we can publish, we can also start it on a remote LXD server.

```
$ lxc remote add lxdserver lxd.example.com
$ lxc launch c7-snapshot1 lxdserver:c7
```

Go see this in action using the "Try It" submenu item, from the "LXD" menu drop down on:

`https://www.linuxcontainers.org`

AppArmor and SELinux

You can use either AppArmor or SELinux profiles with LXD.

```
$ lxc config set c7 raw.lxc 'lxc.aa_profile = jaycustom1'
$ lxc config set c7 raw.lxc 'lxc.se_context =
system_u:system_r:lxc_t:s0:c22'
```

Capability Dropping

You can use either a whitelist or blacklist of capabilities.

Obviously, we'd choose a whitelist.

```
$ lxc config set c7 raw.lxc 'lxc.cap.keep = CAP_NET_BIND_SERVICE'
```

```
Really useful
```

```
https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.
5.html
```

Firewalling Containers

You create firewall rules for an LXD container.

```
$ipt = "/sbin/iptables"
for chain in FORWARD OUTPUT; do
  $ipt -A $chain -d 10.0.3.79 -o lxcbr0 -p tcp --dport 22 \
       -s 56.12.12.12 -j ACCEPT
done
for chain in FORWARD INPUT; do
  $ipt -A $chain -s 10.0.3.79 -i lxcbr0 -s 10.0.3.79 -j DROP
done
```

Wrap Up

Both Docker and LXD give us the ability to do some nice containment, but they also make testing and developing software so much easier.

I strongly recommend that you try Docker out this week.

LXD is helpful, but it may not stabilize for six to twelve months.

Speaker Bio

Jay Beale has created several defensive security tools, including Bastille Linux and the CIS Linux Scoring Tool, both of which are used throughout industry and government. He has served as an invited speaker at many industry and government conferences, a columnist for Information Security Magazine, SecurityPortal and SecurityFocus, and a contributor to nine books, including those in his Open Source Security Series and the "Stealing the Network" series. Jay is a founder and the CTO/COO of the information security consulting company InGuardians.