Lift and shift: Modernising a legacy LAMP application with systemd-nspawn

Martin Lucina, martin@lucina.net
@mato@mastodon.1984.cz

FOSDEM 2024, Containers devroom

I. Setting the stage

A phone call one evening:

“Our sole IT person is leaving after 20 years. They developed the entire system our school runs on. We need someone who understands software, and who we can trust.”

“Our business depends on it. We need to keep this running, one way or another, for the next 20 years. *I have no idea what state it's in*, except that it's not very good.”

“They leave in a month.”

“Can you help?”

II. Setting the stage

They weren't kidding ... about the system being in a less than ideal state.

```
# ls -l /etc/debian_version
-rw-r--r-- 1 root root 6 2009-08-30 22:01 /etc/debian_version
# cat /etc/debian_version
5.0.3
```

```
ii  apache                          1.3.34-4.1+etch1
ii  apache-common                   1.3.34-4.1+etch1
rc  apache2.2-common               2.2.3-4
ii  libapache-mod-php4              4.4.4-8+etch6
rc  libapache2-mod-php5             5.2.0-8+etch7
ii  libssl0.9.8                     0.9.8g-15+lenny3
ii  mysql-server                    5.0.51a-24+lenny2
ii  mysql-server-5.0                5.0.51a-24+lenny2
ii  php4-common                     4.4.4-8+etch6
ii  php5-common                     5.2.6.dfsg.1-1+lenny3
ii  php4-mysql                      4.4.4-8+etch6
```

And a quick, naive, SLOCCount of `/var/www/html` reveals:

```
Totals grouped by language (dominant language first):
php:          226839 (80.62%)
```

III. Salvage!

My *naive* plan:

Extract as much business and technical knowledge as possible, before the original author dissapears forever.

Then:

1. Virtualize all the things!
2. Secure the obvious attack surfaces.
3. Split off the business-critical system.
4. Do so in a way that will be future-proof and maintainable.
5. All while keeping it running with minimal disruption thoughout the school year.

IV. Enter containers, stage left.

I wanted a way of *reproducibly* re-building the 15 year old enviroment that the legacy LAMP application depended on, which I could host on modern infrastructure and *maintain* with modern tools.

The Debian community have developed reproducible container images of EOL releases, available on Docker Hub as debian/eol.

However, Docker doesn't really fit the bill — this application is very much a *pet*, not *cattle*!

I wanted something more like FreeBSD jails, or Solaris zones.

Enter systemd-nspawn(1), the Swiss Army Knife of Linux containers.

V. 2009 meets 2023, kicking and screaming.

Getting from a Docker image to a rootfs that we can run with systemd-nspawn is easy:

```
$ skopeo copy -f oci docker://registry.docker.com/debian/eol:lenny oci:build/lenny-oci
$ oci-image-tool create --ref platform.os=linux build/lenny-oci build/lenny-bundle
$ cp -pRdu --reflink=always build/lenny-bundle/rootfs run/rootfs
# chown -hR root:root run/rootfs
```

Then we try and run it, and 2009 calls and wants its syscalls back:

```
# systemd-nspawn --directory=run/rootfs -M ctr-lenny /bin/sh
Spawning container ctr-lenny on /home/fosdem-containers/run/rootfs.
Press ^] three times within 1s to kill container.

Container ctr-lenny terminated by signal SEGV.
```

```
newbug systemd[1]: Started machine-ctr\x2dlenny.scope - Container ctr-lenny.
newbug kernel: sh[2263] vsyscall attempted with vsyscall=none ip:ffffffffff>
newbug kernel: sh[2263]: segfault at ffffffffff600400 ip ffffffffff600400 s>
newbug kernel: Code: Unable to access opcode bytes at 0xffffffffff6003d6.
```

VI. 2009 meets 2023 and they make up.

Adding `vsyscall=emulate` to our kernel command line and rebooting gets us:

```
# systemd-nspawn --directory=run/rootfs -M ctr-lenny /bin/sh
Spawning container ctr-lenny on /home/fosdem-containers/run/rootfs.
Press ^] three times within 1s to kill container.
sh-3.2# cat /etc/debian_version
5.0.10
sh-3.2# exit
exit
Container ctr-lenny exited successfully.
```

VII. From /bin/sh to /sbin/init.

If all we want is a `/bin/sh` and `apt-get` in a Debian lenny container, then so far, so good!

But, we want more than that:

1. Run a full `/sbin/init` in the container, which will manage the LAMP stack services.
2. Integrate the container's networking with the host system's `systemd-networkd`.
3. Integrate the container's `/dev/log` with the host system's `systemd-journald`.
4. Use user namespacing to give us a private set of UIDs/GIDs in the container.
5. Start and stop the container as a systemd service on the host.

I made a script, so that you don't have to! Please follow along at https://gist.github.com/mato/35477cf13a68411555546f650b8a391e

VIII. INIT: version 2.86 booting

You can now boot the resulting rootfs as follows:

```
# systemd-nspawn \
        -M ctr-lenny \
        --directory=run/rootfs \
    --private-users=pick \
    --network-veth \
    --kill-signal=SIGINT \
    --boot
```

Login on the console, and note that `^C` will shut down the container cleanly. This is desirable when it is started as a systemd unit.

IX. Gotchas: Networking.

Networking:

- `systemd-networkd` *almost* does all the things by default.
- The host needs the `net.ipv4.ip_forward = 1` sysctl enabled.
- If you're doing anything in your FORWARD chain, ensure you have something like:

```
iptables -I FORWARD 1 -i ve-+ -j ACCEPT
iptables -I FORWARD 1 -o ve-+ -j ACCEPT
```

- Old DHCP clients are picky about checksums:

```
iptables -t mangle -A POSTROUTING -o ve-+ -p udp --dport 68 -j CHECKSUM --checksum-fill
```

- Your mileage *will* vary.

X. Gotchas: Logging.

`systemd-journald` namespaces are great. But, using them naively results in the systemd-journald for our namespace exiting immediately, due to no clients connecting.

- Copy `/etc/systemd/journald.conf` to `journald@ctr-lenny.conf` and set **`MaxRetentionSec=1year`** to fix this.
- `systemctl start systemd-journald@ctr-lenny.service` gets us a socket at `/run/systemd/journal.ctr-lenny/dev-log`.
- Bind mount this into the container with `--bind=/run/systemd/journal.ctr-lenny/dev-log:/dev/log`.

Apart from that, well documented in `systemd-journald(8)`.

XI. Startup and shutdown.

`systemd-nspawn` comes with a default unit file, at

`/lib/systemd/system/systemd-nspawn@.service`.

Start with that and customize to suit as e.g. `/etc/systemd/system/app-nspawn@.service`.

Useful runes:

```
Wants=systemd-journald@%i.service
```

... gets you a proper dependency on the service providing the matching journald namespace.

And, start with this for ExecStart:

```
ExecStart=systemd-nspawn --quiet --keep-unit --settings=override --boot \
--machine=%i --directory=/srv/app/%i/rootfs --private-users=pick --network-veth \
--kill-signal=SIGINT --bind=/run/systemd/journal.%i/dev-log:/dev/log (...)
```

XII. Conclusion.

The application is running fine in production!

There is *way* more I could rant about for a long time, mainly about PHP 4 and MySQL 5.

Did I mention it's all running in the `CP1250` character set?

But, at least it's 64-bit now, so it won't fall over in 2038!

I'll end with a conversation with my friend Martin Sústrik:

“What is more complicated? The unikernel research and development work you did for the last few years prior, or this work?”

“Oh this work, definitely!”