On Running systemd-nspawn Containers

11 min read

February 4, 2022

I'd like to talk more about a container technology that I really like that I touched upon in a previous article: **`systemd-nspawn`**.

`systemd-nspawn` is a container manager that allows you to run a full operating system or a command in a directory tree. Conceptually, it is similar to the venerable `chroot`, but it is much more secure.

While `chroot`s do perform filesystem isolation, they don't provide any of the security benefits that `cgroup`s and `namespaces` provide. Additionally, they're not easy to setup, unless, of course, you're using a tool like `debootstrap` or `pacstrap`.

> See my previous articles on using `chroot` in some pretty sweet ways:
>
> - On Running a Tor Onion Service in a Chroot
> - On Escaping a Chroot
> - On Stack Smashing, Part Two

`systemd-nspawn`, on the other hand, gives you as much security and configuration as you would want and expect and is as easily configurable as better-known tools like Docker (although it operates at a lower-level).

To create a container, `systemd-nspawn` expects a root filesystem and optionally a `JSON` container configuration file, which of course brings to mind an OCI runtime bundle, because `systemd-nspawn` is fully OCI compliant. Those familiar with tools like `runc` will be familiar with this requirement.

> One can use many of the same methods to get a root filesystem (rootfs) as documented in my article on `runc`.

By using the machine option (`--machine` or `-M`) with `systemd-nspawn`, the operating system tree (root filesystem) is automatically searched for in a couple places, most notably in `/var/lib/machines`, which is the recommended directory on the system.

The intent of this article is to quickly and succinctly outline several ways to get started using containers with `systemd-nspawn`. Hopefully, it will also encourage you to think more critically of tools like Docker and determine if they are as necessary as all the hype surrounding them would have you believe.

We'll be running the Tor browser in a container managed by `systemd-nspawn`.

Note that the following assumptions are made:

- `systemd-nspawn` is already installed on your system.

```
$ sudo apt install systemd-container
```

- All the following examples will assume that the current working directory is `/var/lib/machines`.
- All commands are run as the `root` user to save typing `sudo` for every command.

Hey, ho, let’s go.

`systemd-nspawn` Container Settings File

What is a container settings file? This is an optional INI-like file that contains startup configurations that will be applied to your container by the `systemd-nspawn` container manager. Any command-line option that is given to `systemd-nspawn` can be put in the settings file, although the names will

be different (see the docs). Simply write them to the file and let `systemd-nspawn` worry about the rest. Not a bad deal, friend.

If you're familiar with `systemd` service files, then this will be familiar to you.

The `systemd-nspawn` container settings file is named after the container to which it is applied. For instance, our container is called `tor-browser`, so the file should be called `tor-browser.nspawn`. That's easy enough.

Where should they go? That's a great question, geezer!

The algorithm searches the following locations, in order:

- `/etc/systemd/nspawn/`
- `/run/systemd/nspawn/`
- `/var/lib/machines/`

Persistent settings file should be placed in `/etc/systemd/nspawn/`, and, unlike the non-privileged location (see below), every setting contained therein will take effect since this is a privileged location (i.e., only privileged users should be able to access any configs in the `/etc` directory).

> Do **not** put anything in `/run/systemd/nspawn/` that you want to survive a reboot, as the `/run` filesystem is temporary and any runtime data put there is placed in volatile memory.

```
$ df /run --output=fstype
Type
tmpfs
```

However, any settings files found in the non-privileged `/var/lib/machines` location will only have a subset of those settings applied. As you may have guessed, any settings that grant elevated privileges or additional capabilities are ignored. This is so untrusted or unvetted files downloaded from the scary Internet don't cause undue harm and aren't automatically applied upon container creation.

In order for the Tor browser to be properly launched, the following `systemd-nspawn` file must be installed in `/etc/systemd/nspawn`:

`tor-browser.nspawn`

```
[Exec]
DropCapability=all
Environment=DISPLAY=:0
Hostname=kilgore-trout
NoNewPrivileges=true
Parameters=./start-tor-browser --log /dev/stdout
PrivateUsers=true
ProcessTwo=true
ResolvConf=copy-host
Timezone=copy
User=noroot
WorkingDirectory=/usr/local/bin/tor-browser
```

This is equivalent to the following command line statement:

```
$ sudo systemd-nspawn \
    --drop-capability all \
    --setenv DISPLAY=:0 \
    --hostname kilgore-trout \
```

```
    --no-new-privileges true \
    --private-users true \
    --as-pid2 \
    --resolv-conf copy-host \
    --timezone copy \
    --user noroot \
    --directory tor-browser \
    bash -c "/usr/local/bin/tor-browser/start-tor-browser --log /dev/st
```

Clearly, the settings file is much more convenient and allows us to start the container by simply typing:

```
$ sudo systemd-nspawn --machine tor-browser
```

In addition, there are more parameters we can set, such as filtering system calls, bind mounts, overlay or union mount points, and much more, but that is out of the scope of this article. And we haven't even covered the `[FILE]` and `[NETWORK]` sections of the settings file.

> If an `systemd-nspawn` settings file isn't present, the container will still launch, but to a virtual shell.

Let's now look at some examples.

Examples

`docker export`

Here is our old "friend" `docker export`. While Docker makes it easy to extract a container's root filesystem as a tarball, it needs, well, *Docker* to do it. That kinda sucks.

I don't know about you, but I don't want multiple container technologies/runtimes/managers on my base system. Since most distros are already using `systemd`, the ability to create and run containers is already installed and just waiting for your fat little fingers to type the necessary commands.

So, installing software additional software to run containers when you *already* have the ability to run containers is nonsensical. It's like installing an editor like Visual Studio Code when you already have Vim.

I've been grudgingly using Docker in my personal projects for the sake of convenience, and it's exactly why I am giddy about moving away from it. Convenience is the scourge of understanding.

Anyway, I digress. Here is a very simple way to run the Tor browser as a `systemd-nspawn` container:

```
$ sudo mkdir tor-browser \
    && docker export $(docker create btoll/tor-browser:latest) \
    | tar -x -C tor-browser
$ sudo systemd-nspawn -M tor-browser
```

The Dockerfile used to create this container image is straightforward.

Because of the convenience of the Dockerfile, Docker makes it easy to create a container with some provisioning already applied.

However, as I'll demonstrate next, it's not any effort to create a shell script from the Dockerfile to do the same thing. Shell scripts are some of our best friends!

And after all, it's pretty silly to install Docker only as a conduit for `systemd-nspawn`. Wouldn't it be better to learn other ways of getting a root filesystem?

Which leads us to…

`debootstrap`

I've been using `debootstrap` for years. It's a really great way to quickly and easily bootstrap a `chroot` by downloading a root filesystem with optional packages.

As mentioned in the previous example, I've created a shell script that provisions the container, and it's a simple step to copy it into the new OS tree created by `debootstrap`.

To run the script, we'll `chroot` into the container (well, what will *become* the container).

```
$ sudo debootstrap \
    --arch=amd64 \
    --variant=minbase \
    bullseye \
    tor-browser \
    http://deb.debian.org/debian
$ sudo cp install_tor_browser.sh tor-browser/
$ sudo chroot tor-browser/
---
### Run the installer script in the chroot.
---
root@sulla:/# ./install_tor_browser.sh
root@sulla:/# exit
$ sudo systemd-nspawn --machine tor-browser
```

That was easy! No big deal.

If we want to share this with a friend or import it into another tool, we can export the container as a tarball and upload it to a server. This can allow us to later download and create and run containers (the same concept as Docker Hub).

```
$ sudo machinectl export-tar tor-browser tor-browser.tar.xz
```

After the container is tarred up, anyone that wants to use it can simply download it and run it without having to do any of the setup steps above (copying and installing).

We'll soon see an example of how we can pull that tarball down from a remote server.

`mkosi`

A tool by Lennart Poettering, `mkosi` is an easy way to create an OSI (operating system image) or OS tree for use by `systemd-nspawn` and any container technology that can "consume" a root filesystem. Written in Python, it is well-documented (see its man page) and easy to use.

> There are many options and cool features but covering them is outside the scope of this article.

Creating an OSI is easy. Here you go:

```
$ sudo apt install mkosi -y
$ sudo mkosi \
    --distribution debian \
    --release bullseye \
    --format gpt_ext4 \
    --postinst-script install_tor_browser.sh
    --with-network \
    -o tor-browser.raw
$ sudo systemd-nspawn --machine tor-browser
```

Note that here I'm giving the `mkosi` tool the `install_tor_browser.sh` script as a value to the `--postinst-script`. This saves us a couple of the steps that we had to do manually when using `debootstrap` in the previous example, namely:

1. Copying the script from the host to the `chroot`.
2. Logging into the `chroot`.
3. Executing the script.

Easy peasy.

`machinectl pull-tar`

We're simply downloading the tarball from a previous example and running it as-is. No need to re-run the Tor browser installation script, of course.

```
$ sudo machinectl pull-tar \
    http://example.com/tor-browser.tar.xz \
    tor-browser
$ sudo systemd-nspawn \
    --resolv-conf copy-host \
    --machine tor-browser
```

> Although unnecessary here, I included the `--resolv-conf` option here to show how easy it is to get a DNS resolver for containers that need one.

`machinectl pull-raw`

I don't use this much, but I'm adding it here for its usefulness.

```
$ sudo machinectl pull-raw \
    http://cloud-images.ubuntu.com/focal/current/focal-server-cloudimg-
    rootfs
$ sudo systemd-nspawn --machine rootfs
Spawning container rootfs on /var/lib/machines/rootfs.raw.
Press ^] three times within 1s to kill container.
root@rootfs:~#
```

> Note that `mkosi` can also build an image that you could use here as the subject of `pull-raw`.

More Commands

This is not even close to a comprehensive list. For example, you can copy files to and from a running container, but I haven't any examples for that.

As always, read the docs.

Exporting

As we've already seen, we can easily export a container's root filesystem as a tarball. Then, simply upload this to an accessible storage area for other people and processes.

This is the same workflow that has been around for hundreds of thousands of years.

```
$ sudo machinectl export-tar tor-browser tor-browser.tar.xz
```

> Also, export as image: `machinectl export-raw`

List Running Containers

```
$ machinectl list
MACHINE      CLASS     SERVICE        OS     VERSION ADDRESSES
tor-browser  container systemd-nspawn debian 11      -
ubuntu-focal container systemd-nspawn ubuntu 20.04   -
```

List All Containers

From the man page:

```
list-images
       Show a list of locally installed container and VM images.  This
       disk images and container directories and subvolumes in /var/lib
       other search paths, see below). Use start (see above) to run a c
       of the listed images.  Note that, by default, containers whose r
       a dot (".") are not shown. To show these too, specify --all. Not
       image ".host" always implicitly exists and refers to the image t
       booted from.
```

```
$ machinectl list-images
NAME        TYPE      RO USAGE CREATED MODIFIED
hugo        directory no   n/a n/a     n/a
tor-browser directory no   n/a n/a     n/a

2 images listed.
```

List all containers without the header and footer:

```
$ machinectl list-images --no-legend
hugo        directory no n/a n/a n/a
tor-browser directory no n/a n/a n/a
```

List Transfers

Downloading and exporting can take a while. Let's check the status!

```
$ sudo machinectl list-transfers
ID PERCENT TYPE       LOCAL       REMOTE
 1     n/a export-tar tor-browser

 1 transfers listed.
```

Querying the Container Status

```
$ sudo machinectl status tor-browser
tor-browser(88544b92092430bc5d3fbbffc12a2f04)
           Since: Fri 2022-02-04 19:54:28 EST; 4h 29min ago
          Leader: 1380829 ((sd-stubinit))
         Service: systemd-nspawn; class container
            Root: /var/lib/machines/tor-browser
              OS: Debian GNU/Linux 11 (bullseye)
            Unit: machine-tor\x2dbrowser.scope
                  ...
```

Removing the Container

When you're *absolutely* sure that you're done with it, you can remove both the machine and the `systemd-nspawn` service file in one fell swoop:

```
$ sudo machinectl remove tor-browser
```

Running Miscellaneous Commands in the OS Tree

```
$ sudo systemd-nspawn -M tor-browser --quiet uname -a
Linux kilgore-trout 5.11.0-49-generic #55-Ubuntu SMP Wed Jan 12 17:36:3

$ sudo systemd-nspawn -M tor-browser --quiet du -hs
264M    .

$ sudo systemd-nspawn -M tor-browser --quiet cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

$ sudo systemd-nspawn -M tor-browser --quiet df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/nvme0n1p2  468G   61G  384G  14% /
tmpfs           1.6G     0  1.6G   0% /tmp
tmpfs           4.0M     0  4.0M   0% /dev
tmpfs           1.6G     0  1.6G   0% /dev/shm
```

```
tmpfs           3.1G   12K  3.1G   1% /run
tmpfs           1.6G  1.9M  1.6G   1% /run/host/incoming
tmpfs           4.0M     0  4.0M   0% /sys/fs/cgroup
```

`fzf`

Using the amazing command-line fuzzy finder tool (`fzf`), I wrote a simple `bash` function that will list all of the machine images in `/var/lib/machines` and allow you to select one. Once the selection is made, it will create and launch the container:

```
nspawn() {
    sudo systemd-nspawn --machine \
        $(machinectl list-images --no-legend | awk '{ print $1 }' | fzf
        --quiet
}
```

Conclusion

This article could also be called `"On Getting Rid of Docker"`, since that it is one of my goals. After all, if you're running a Linux distro, chances are that the init system is `systemd`, so why not use `systemd-nspawn`?

There's no need to install `containerd` and `runc`, which Docker needs and installs by default. I don't have anything against them, mind you, and `systemd-nspawn` may not be the best tool for the job.

Unfortunately, though, most developers don't even know that there are options outside of Docker, or that they're not as "convenient". Hopefully, this article has disabused some of that notion.

References

- systemd-nspawn (Debian docs)
- systemd-nspawn (Arch Linux docs)
- Running sid in systemd-nspawn
- Running containers with systemd-nspawn
- mkosi — A Tool for Generating OS Images
- Ubuntu Cloud Images
- Debian Official Cloud Images