Pid Eins

レナート
لينارت

POSTED ON FR 08 APRIL 2011

systemd for Administrators, Part VI

Here's another installment of my ongoing series on systemd for Administrators:

Changing Roots

As administrator or developer sooner or later you'll ecounter `chroot()` environments. The `chroot()` system call simply shifts what a process and all its children consider the root directory /, thus limiting what the process can see of the file hierarchy to a subtree of it. Primarily `chroot()` environments have two uses:

1. For security purposes: In this use a specific isolated daemon is chroot()ed into a private subdirectory, so that when exploited the attacker can see only the subdirectory instead of the full OS hierarchy: he is trapped inside the chroot() jail.
2. To set up and control a debugging, testing, building, installation or recovery image of an OS: For this a whole guest operating system hierarchy is mounted or bootstraped into a subdirectory of the host OS, and then a shell (or some other application) is started inside it, with this subdirectory turned into its /. To the shell it appears as if it was running inside a system that can differ greatly from the host OS. For example, it might run a different distribution or even a different architecture (Example: host x86_64, guest i386). The full hierarchy of the host OS it cannot see.

On a classic System-V-based operating system it is relatively easy to use chroot() environments. For example, to start a specific daemon for test or other reasons inside a chroot()-based guest OS tree, mount `/proc`, `/sys` and a few other API file systems into the tree, and then use `chroot(1)` to enter the chroot, and finally run the SysV init script via `/sbin/service` from inside the chroot.

On a systemd-based OS things are not that easy anymore. One of the big advantages of systemd is that all daemons are guaranteed to be invoked in a

completely clean and independent context which is in no way related to the context of the user asking for the service to be started. While in sysvinit-based systems a large part of the execution context (like resource limits, environment variables and suchlike) is inherited from the user shell invoking the init skript, in systemd the user just notifies the init daemon, and the init daemon will then fork off the daemon in a sane, well-defined and pristine execution context and no inheritance of the user context parameters takes place. While this is a formidable feature it actually breaks traditional approaches to invoke a service inside a chroot() environment: since the actual daemon is always spawned off PID 1 and thus inherits the chroot() settings from it, it is irrelevant whether the client which asked for the daemon to start is chroot()ed or not. On top of that, since systemd actually places its local communications sockets in `/run/systemd` a process in a chroot() environment will not even be able to talk to the init system (which however is probably a good thing, and the daring can work around this of course by making use of bind mounts.)

This of course opens the question how to use chroot()s properly in a systemd environment. And here's what we came up with for you, which hopefully answers this question thoroughly and comprehensively:

Let's cover the first usecase first: locking a daemon into a chroot() jail for security purposes. To begin with, chroot() as a security tool is actually quite dubious, since chroot() is not a one-way street. It is relatively easy to escape a chroot() environment, as even the man page points out. Only in combination with a few other techniques it can be made somewhat secure. Due to that it usually requires specific support in the applications to chroot() themselves in a tamper-proof way. On top of that it usually requires a deep understanding of the chroot()ed service to set up the chroot() environment properly, for example to know which directories to

bind mount from the host tree, in order to make available all communication channels in the chroot() the service actually needs. Putting this together, chroot()ing software for security purposes is almost always done best in the C code of the daemon itself. The developer knows best (or at least *should* know best) how to properly secure down the chroot(), and what the minimal set of files, file systems and directories is the daemon will need inside the chroot(). These days a number of daemons are capable of doing this, unfortunately however of those running by default on a normal Fedora installation only two are doing this: Avahi and RealtimeKit. Both apparently written by the same really smart dude. Chapeau! ;-) (Verify this easily by running `ls -l /proc/*/root` on your system.)

That all said, systemd of course does offer you a way to chroot() specific daemons and manage them like any other with the usual tools. This is supported via the `RootDirectory=` option in systemd service files. Here's an example:

```
[Unit]
Description=A chroot()ed Service

[Service]
RootDirectory=/srv/chroot/foobar
ExecStartPre=/usr/local/bin/setup-foobar-chroot.sh
ExecStart=/usr/bin/foobard
RootDirectoryStartOnly=yes
```

In this example, `RootDirectory=` configures where to chroot() to before invoking the daemon binary specified with `ExecStart=`. Note that the path specified in `ExecStart=` needs to refer to the binary inside the chroot(), it is not a path to the binary in the host tree (i.e. in this example the binary executed is seen as `/srv/chroot/foobar/usr/bin/foobard` from the host OS). Before the daemon is started a shell script `setup-foobar-chroot.sh` is invoked, whose purpose it is to set up the chroot environment as necessary, i.e. mount `/proc` and similar file

systems into it, depending on what the service might need. With the `RootDirectoryStartOnly=` switch we ensure that only the daemon as specified in `ExecStart=` is chrooted, but not the `ExecStartPre=` script which needs to have access to the full OS hierarchy so that it can bind mount directories from there. (For more information on these switches see the respective man pages.) If you place a unit file like this in `/etc/systemd/system/foobar.service` you can start your chroot()ed service by typing `systemctl start foobar.service`. You may then introspect it with `systemctl status foobar.service`. It is accessible to the administrator like any other service, the fact that it is chroot()ed does -- unlike on SysV -- not alter how your monitoring and control tools interact with it.

Newer Linux kernels support file system namespaces. These are similar to `chroot()` but a lot more powerful, and they do not suffer by the same security problems as `chroot()`. systemd exposes a subset of what you can do with file system namespaces right in the unit files themselves. Often these are a useful and simpler alternative to setting up full chroot() environment in a subdirectory. With the switches `ReadOnlyDirectories=` and `InaccessibleDirectories=` you may setup a file system namespace jail for your service. Initially, it will be identical to your host OS' file system namespace. By listing directories in these directives you may then mark certain directories or mount points of the host OS as read-only or even completely inaccessible to the daemon. Example:

```
[Unit]
Description=A Service With No Access to /home

[Service]
ExecStart=/usr/bin/foobard
InaccessibleDirectories=/home
```

This service will have access to the entire file system tree of the host OS with one exception: /home will not be visible to it, thus protecting the user's data from potential exploiters. (See the man page for details on these options.)

File system namespaces are in fact a better replacement for `chroot()`s in many many ways. Eventually Avahi and RealtimeKit should probably be updated to make use of namespaces replacing `chroot()`s.

So much about the security usecase. Now, let's look at the other use case: setting up and controlling OS images for debugging, testing, building, installing or recovering.

chroot() environments are relatively simple things: they only virtualize the file system hierarchy. By chroot()ing into a subdirectory a process still has complete access to all system calls, can kill all processes and shares about everything else with the host it is running on. To run an OS (or a small part of an OS) inside a chroot() is hence a dangerous affair: the isolation between host and guest is limited to the file system, everything else can be freely accessed from inside the chroot(). For example, if you upgrade a distribution inside a chroot(), and the package scripts send a SIGTERM to PID 1 to trigger a reexecution of the init system, this will actually take place in the host OS! On top of that, SysV shared memory, abstract namespace sockets and other IPC primitives are shared between host and guest. While a completely secure isolation for testing, debugging, building, installing or recovering an OS is probably not necessary, a basic isolation to avoid *accidental* modifications of the host OS from inside the chroot() environment is desirable: you never know what code package scripts execute which might interfere with the host OS.

To deal with chroot() setups for this use systemd offers you a couple of features:

First of all, `systemctl` detects when it is run in a chroot. If so, most of its operations will become NOPs, with the exception of `systemctl enable` and `systemctl disable`. If a package installation script hence calls these two commands, services will be enabled in the guest OS. However, should a package installation script include a command like `systemctl restart` as part of the package upgrade process this will have no effect at all when run in a chroot() environment.

More importantly however systemd comes out-of-the-box with the systemd-nspawn tool which acts as chroot(1) on steroids: it makes use of file system and PID namespaces to boot a simple lightweight container on a file system tree. It can be used almost like chroot(1), except that the isolation from the host OS is much more complete, a lot more secure and even easier to use. In fact, `systemd-nspawn` is capable of booting a *complete* systemd or sysvinit OS in container with a single command. Since it virtualizes PIDs, the init system in the container can act as PID 1 and thus do its job as normal. In contrast to chroot(1) this tool will implicitly mount `/proc`, `/sys` for you.

Here's an example how in three commands you can boot a Debian OS on your Fedora machine inside an nspawn container:

```
# yum install debootstrap
# debootstrap --arch=amd64 unstable debian-tree/
# systemd-nspawn -D debian-tree/
```

This will bootstrap the OS directory tree and then simply invoke a shell in it. If you want to boot a full system in the container, use a command like this:

```
# systemd-nspawn -D debian-tree/ /sbin/init
```

And after a quick bootup you should have a shell prompt, inside a complete OS, booted in your container. The container will not be able to see any of the processes outside of it. It will share the network configuration, but not be able to modify it. (Expect a couple of EPERMs during boot for that, which however should not be fatal). Directories like `/sys` and `/proc/sys` are available in the container, but mounted read-only in order to avoid that the container can modify kernel or hardware configuration. Note however that this protects the host OS only from *accidental* changes of its parameters. A process in the container can manually remount the file systems read-writeable and then change whatever it wants to change.

So, what's so great about `systemd-nspawn` again?

1. It's really easy to use. No need to manually mount `/proc` and `/sys` into your chroot() environment. The tool will do it for you and the kernel automatically cleans it up when the container terminates.
2. The isolation is much more complete, protecting the host OS from accidental changes from inside the container.
3. It's so good that you can actually boot a full OS in the container, not just a single lonesome shell.
4. It's actually tiny and installed everywhere where systemd is installed. No complicated installation or setup.

systemd itself has been modified to work very well in such a container. For example, when shutting down and detecting that it is run in a container, it just calls exit(), instead of reboot() as last step.

Note that `systemd-nspawn` is not a full container solution. If you need that LXC is the better choice for you. It uses the same underlying kernel technology but offers

a lot more, including network virtualization. If you so will, `systemd-nspawn` is the GNOME 3 of container solutions: slick and trivially easy to use -- but with few configuration options. LXC OTOH is more like KDE: more configuration options than lines of code. I wrote `systemd-nspawn` specifically to cover testing, debugging, building, installing, recovering. That's what you should use it for and what it is really good at, and where it is a much much nicer alternative to chroot(1).

So, let's get this finished, this was already long enough. Here's what to take home from this little blog story:

1. Secure chroot()s are best done natively in the C sources of your program.
2. `ReadOnlyDirectories=`, `InaccessibleDirectories=` might be suitable alternatives to a full chroot() environment.
3. `RootDirectory=` is your friend if you want to chroot() a specific service.
4. `systemd-nspawn` is made of awesome.
5. chroot()s are lame, file system namespaces are totally l33t.

All of this is readily available on your Fedora 15 system.

And that's it for today. See you again for the next installment.

Category: projects

← BACK TO INDEX