

Systemd for Administrators

Lennart Poettering
Author

Christian Rebischke
Editor

February 27, 2016

Contents

1	Abstract	2
2	Disclaimer	2
3	Verifying Bootup	3
4	Which Service Owns Which Processes?	4
5	How Do I Convert A SysV Init Script Into A systemd Service File?	7
6	Killing Services	11
7	The Three Levels of "Off"	12
8	Changing Roots	14
9	References	19

1 Abstract

As many of you know, systemd is the new Fedora init system, starting with F14, and it is also on its way to being adopted in a number of other distributions as well (for example, OpenSUSE). For administrators systemd provides a variety of new features and changes and enhances the administrative process substantially. This blog story is the first part of a series of articles I plan to post roughly every week for the next months. In every post I will try to explain one new feature of systemd. Many of these features are small and simple, so these stories should be interesting to a broader audience. However, from time to time we'll dive a little bit deeper into the great new features systemd provides you with.

2 Disclaimer

This handbook is written by Lennart Poettering. There are maybe some additions, cuts or other changes for increasing readability. Please visit Lennarts Blog for the original Blogposts:

<https://0pointer.de/blog>

3 Verifying Bootup

Traditionally, when booting up a Linux system, you see a lot of little messages passing by on your screen. As we work on speeding up and parallelizing the boot process these messages are becoming visible for a shorter and shorter time only and be less and less readable – if they are shown at all, given we use graphical boot splash technology like Plymouth these days. Nonetheless the information of the boot screens was and still is very relevant, because it shows you for each service that is being started as part of bootup, whether it managed to start up successfully or failed (with those green or red [OK] or [FAILED] indicators). To improve the situation for machines that boot up fast and parallelized and to make this information more nicely available during runtime, we added a feature to systemd that tracks and remembers for each service whether it started up successfully, whether it exited with a non-zero exit code, whether it timed out, or whether it terminated abnormally (by segfaulting or similar), both during start-up and runtime. By simply typing `systemctl` in your shell you can query the state of all services, both systemd native and SysV/LSB services:

```
[root@lambda] ~# systemctl
```

UNIT	LOAD	ACTIVE	SUB
dev-hugepages.automount	loaded	active	running
dev-mqueue.automount	loaded	active	running
proc-sys-fs-binfmt_misc.automount	loaded	active	waiting
sys-kernel-debug.automount	loaded	active	waiting
sys-kernel-security.automount	loaded	active	waiting
sys-devices-pc...0000:02:00.0-net-eth0.device	loaded	active	plugged
sys-devices-virtual-tty-tty9.device	loaded	active	plugged
-.mount	loaded	active	mounted
boot.mount	loaded	active	mounted
dev-hugepages.mount	loaded	active	mounted
dev-mqueue.mount	loaded	active	mounted
home.mount	loaded	active	mounted
proc-sys-fs-binfmt_misc.mount	loaded	active	mounted
abrtd.service	loaded	active	running
bus.service	loaded	active	running
getty@tty2.service	loaded	active	running
getty@tty3.service	loaded	active	running
getty@tty4.service	loaded	active	running
getty@tty5.service	loaded	active	running
getty@tty6.service	loaded	active	running
haldaemon.service	loaded	active	running
hdapsd@sda.service	loaded	active	running
irqbalance.service	loaded	active	running
iscsi.service	loaded	active	exited
iscsid.service	loaded	active	exited
livesys-late.service	loaded	active	exited
livesys.service	loaded	active	exited
lvm2-monitor.service	loaded	active	exited
mdmonitor.service	loaded	active	running
modem-manager.service	loaded	active	running
netfs.service	loaded	active	exited
NetworkManager.service	loaded	active	running
ntpd.service	loaded	maintenance	maintenance

LOAD = Reflects whether the unit definition was properly loaded.
 ACTIVE = The high-level unit activation state, i.e. generalization of SUB.
 SUB = The low-level unit activation state, values depend on unit type.
 JOB = Pending job for the unit.

221 units listed. Pass --all to see inactive units, too.

```
[root@lambda] ~#
```

(I have shortened the output above a little, and removed a few lines not relevant for this blog post.) Look at the **ACTIVE** column, which shows you the high-level state of a service (or in fact of any kind of unit systemd maintains, which can be more than just services, but we'll have a look on this in a later blog posting), whether it is **active** (i.e. running), **inactive** (i.e. not running) or in any other state. If you look closely you'll see one item in the list that is marked

maintenance and highlighted in red. This informs you about a service that failed to run or otherwise encountered a problem. In this case this is ntpd. Now, let's find out what actually happened to ntpd, with the *systemctl status* command:

```
[root@lambda] ~# systemctl status ntpd.service
ntpd.service - Network Time Service
   Loaded: loaded (/etc/systemd/system/ntpd.service)
   Active: maintenance
     Main: 953 (code=exited, status=255)
   CGroup: name=systemd:/systemd-1/ntpd.service
[root@lambda] ~#
```

This shows us that NTP terminated during runtime (when it ran as PID 953), and tells us exactly the error condition: the process exited with an exit status of 255.

In a later systemd version, we plan to hook this up to ABRT, as soon as this enhancement request is fixed. Then, if systemctl status shows you information about a service that crashed it will direct you right-away to the appropriate crash dump in ABRT.

Summary: use *systemctl* and *systemctl status* as modern, more complete replacements for the traditional boot-up status messages of SysV services. *systemctl status* not only captures in more detail the error condition but also shows runtime errors in addition to start-up errors. That's it for this week, make sure to come back next week, for the next posting about systemd for administrators!

4 Which Service Owns Which Processes?

On most Linux systems the number of processes that are running by default is substantial. Knowing which process does what and where it belongs to becomes increasingly difficult. Some services even maintain a couple of worker processes which clutter the "ps" output with many additional processes that are often not easy to recognize. This is further complicated if daemons spawn arbitrary 3rd-party processes, as Apache does with CGI processes, or cron does with user jobs.

A slight remedy for this is often the process inheritance tree, as shown by "ps xaf". However this is usually not reliable, as processes whose parents die get reparented to PID 1, and hence all information about inheritance gets lost. If a process "double forks" it hence loses its relationships to the processes that started it. (This actually is supposed to be a feature and is relied on for the traditional Unix daemonizing logic.) Furthermore processes can freely change their names with PR_SETNAME or by patching argv[0], thus making it harder to recognize them. In fact they can play hide-and-seek with the administrator pretty nicely this way.

In systemd we place every process that is spawned in a control group named after its service. Control groups (or cgroups) at their most basic are simply groups

of processes that can be arranged in a hierarchy and labelled individually. When processes spawn other processes these children are automatically made members of the parents cgroup. Leaving a cgroup is not possible for unprivileged processes. Thus, cgroups can be used as an effective way to label processes after the service they belong to and be sure that the service cannot escape from the label, regardless how often it forks or renames itself. Furthermore this can be used to safely kill a service and all processes it created, again with no chance of escaping.

In today's installment I want to introduce you to two commands you may use to relate systemd services and processes. The first one, is the well known ps command which has been updated to show cgroup information along the other process details. And this is how it looks:

\$ ps xawf -eo	pid,user,cgroup,args	COMMAND
PID USER	CGROUP	
2 root	-	[kthreadd]
3 root	-	_ [ksftirqd/0]
[...]		
4281 root	-	_ [flush-8:0]
1 root	name=systemd:/systemd-1	/sbin/init
455 root	name=systemd:/systemd-1/sysinit.service	/sbin/udevd -d
28188 root	name=systemd:/systemd-1/sysinit.service	_ /sbin/udevd -d
28191 root	name=systemd:/systemd-1/sysinit.service	_ /sbin/udevd -d
1131 root	name=systemd:/systemd-1/auditd.service	auditd
1133 root	name=systemd:/systemd-1/auditd.service	_ /sbin/audispd
1135 root	name=systemd:/systemd-1/auditd.service	_ /usr/sbin/sedispd
1193 root	name=systemd:/systemd-1/rsyslog.service	/sbin/rsyslogd -c 4
1195 root	name=systemd:/systemd-1/cups.service	cupsd -C /etc/cups/cupsd.conf
1210 root	name=systemd:/systemd-1/irqbalance.service	irqbalance
1216 root	name=systemd:/systemd-1/dbus.service	/usr/sbin/modem-manager
1219 root	name=systemd:/systemd-1/dbus.service	/usr/libexec/polkit-1/polkitd
1317 root	name=systemd:/systemd-1/abrt.service	/usr/sbin/abrt -d -s
1332 root	name=systemd:/systemd-1/getty@.service	/tty2 /sbin/mingetty tty2
1339 root	name=systemd:/systemd-1/getty@.service	/tty3 /sbin/mingetty tty3
1342 root	name=systemd:/systemd-1/getty@.service	/tty5 /sbin/mingetty tty5
1343 root	name=systemd:/systemd-1/getty@.service	/tty4 /sbin/mingetty tty4
1344 root	name=systemd:/systemd-1/crond.service	crond
1346 root	name=systemd:/systemd-1/getty@.service	/tty6 /sbin/mingetty tty6
1362 root	name=systemd:/systemd-1/ssh.service	/usr/sbin/ssh
1759 lennart	name=systemd:/user/lennart/1	gnome-screensaver
909 lennart	name=systemd:/user/lennart/1	gnome-terminal
1913 lennart	name=systemd:/user/lennart/1	_ gnome-ptty-helper
1914 lennart	name=systemd:/user/lennart/1	_ bash
29231 lennart	name=systemd:/user/lennart/1	_ ssh tango
2221 lennart	name=systemd:/user/lennart/1	_ bash
4193 lennart	name=systemd:/user/lennart/1	_ ssh tango
2461 lennart	name=systemd:/user/lennart/1	_ bash
27251 lennart	name=systemd:/user/lennart/1	_ empathy

(Note that this output is shortened, I have removed most of the kernel threads here, since they are not relevant in the context of this blog story)

In the third column you see the cgroup systemd assigned to each process. You'll find that the udev processes are in the name=systemd:/systemd-1/sysinit.service cgroup, which is where systemd places all processes started by the sysinit.service service, which covers early boot.

My personal recommendation is to set the shell alias `psc` to the `ps` command line shown above:

```
alias psc='ps xawf -eo pid,user,cgroup,args '
```

With this service information of processes is just four keypresses away! A different way to present the same information is the `systemd-cgls` tool we ship with `systemd`. It shows the `cgroup` hierarchy in a pretty tree. Its output looks like this:

```
$ systemd-cgls
+ 2 [kthreadd]
[... ]
+ 4281 [flush -8:0]
+ user
| \ lennart
| | \ 1
| | | + 1495 pam: gdm-password
| | | + 1521 gnome-session
| | | + 1534 dbus-launch --sh-syntax --exit-with-session
| | | + 1603 /usr/libexec/gconfd-2
| | | + 1612 /usr/libexec/gnome-settings-daemon
| | | + 1615 /usr/libexec/gvfsd
| | \ 29519 systemd-cgls
| \ systemd-1
| | + 1 /sbin/init
| | + ntpd.service
| | | \ 4112 /usr/sbin/ntpd -n -u ntp:ntp -g
| | + systemd-logger.service
| | | \ 1499 /lib/systemd/systemd-logger
| | + accounts-daemon.service
| | | \ 1496 /usr/libexec/accounts-daemon
| | + rtkit-daemon.service
| | | \ 1473 /usr/libexec/rtkit-daemon
| | + console-kit-daemon.service
| | | \ 1408 /usr/sbin/console-kit-daemon --no-daemon
| | + prefdm.service
| | | + 1376 /usr/sbin/gdm-binary -nodaemon
| | | + 1419 /usr/bin/dbus-launch --exit-with-session
| | | \ 1511 /usr/bin/gnome-keyring-daemon --daemonize --login
| | + getty@.service
| | | + tty6
| | | | \ 1346 /sbin/mingetty tty6
| | | + tty4
| | | | \ 1343 /sbin/mingetty tty4
| | | + tty5
| | | | \ 1342 /sbin/mingetty tty5
| | | + tty3
| | | | \ 1339 /sbin/mingetty tty3
| | | \ tty2
| | | | \ 1332 /sbin/mingetty tty2
| | \ 28191 /sbin/udev -d
```

(This too is shortened, the same way)

As you can see, this command shows the processes by their `cgroup` and hence service, as `systemd` labels the `cgroups` after the services. For example, you can easily see that the auditing service `auditd.service` spawns three individual processes, `auditd`, `auditp` and `sedispatch`.

If you look closely you will notice that a number of processes have been assigned to the `cgroup` `/user/1`. At this point let's simply leave it at that `systemd` not only maintains services in `cgroups`, but user session processes as well. In a later installment we'll discuss in more detail what this about.

So much for now, come back soon for the next installment!

5 How Do I Convert A SysV Init Script Into A systemd Service File?

Traditionally, Unix and Linux services (daemons) are started via SysV init scripts. These are Bourne Shell scripts, usually residing in a directory such as `/etc/rc.d/init.d/` which when called with one of a few standardized arguments (verbs) such as `start`, `stop` or `restart` controls, i.e. starts, stops or restarts the service in question. For starts this usually involves invoking the daemon binary, which then forks a background process (more precisely daemonizes). Shell scripts tend to be slow, needlessly hard to read, very verbose and fragile. Although they are immensely flexible (after all, they are just code) some things are very hard to do properly with shell scripts, such as ordering parallelized execution, correctly supervising processes or just configuring execution contexts in all detail. systemd provides compatibility with these shell scripts, but due to the shortcomings pointed out it is recommended to install native systemd service files for all daemons installed. Also, in contrast to SysV init scripts which have to be adjusted to the distribution systemd service files are compatible with any kind of distribution running systemd (which become more and more these days...). What follows is a terse guide how to take a SysV init script and translate it into a native systemd service file. Ideally, upstream projects should ship and install systemd service files in their tarballs. If you have successfully converted a SysV script according to the guidelines it might hence be a good idea to submit the file as patch to upstream. How to prepare a patch like that will be discussed in a later installment, suffice to say at this point that the `daemon(7)` manual page shipping with systemd contains a lot of useful information regarding this.

So, let's jump right in. As an example we'll convert the init script of the ABRT daemon into a systemd service file. ABRT is a standard component of every Fedora install, and is an acronym for Automatic Bug Reporting Tool, which pretty much describes what it does, i.e. it is a service for collecting crash dumps. Its SysV script I have uploaded [here](#).

The first step when converting such a script is to read it (surprise surprise!) and distill the useful information from the usually pretty long script. In almost all cases the script consists of mostly boilerplate code that is identical or at least very similar in all init scripts, and usually copied and pasted from one to the other. So, let's extract the interesting information from the script linked above:

- A description string for the service is "Daemon to detect crashing apps". As it turns out, the header comments include a redundant number of description strings, some of them describing less the actual service but the init script to start it. systemd services include a description too, and it should describe the service and not the service file.
- The LSB header[1] contains dependency information. systemd due to its design around socket-based activation usually needs no (or very little) manually configured dependencies. (For details regarding socket activation see the original announcement blog post.) In this case the dependency on \$syslog (which encodes that abrttd requires a syslog daemon), is the only valuable information. While the header lists another dependency (\$local.fs) this one is redundant with systemd as normal system services are always started with all local file systems available.
- The LSB header suggests that this service should be started in runlevels 3 (multi-user) and 5 (graphical).
- The daemon binary is /usr/sbin/abrttd

And that's already it. The entire remaining content of this 115-line shell script is simply boilerplate or otherwise redundant code: code that deals with synchronizing and serializing startup (i.e. the code regarding lock files) or that outputs status messages (i.e. the code calling echo), or simply parsing of the verbs (i.e. the big case block).

From the information extracted above we can now write our systemd service file:

```
[Unit]
Description=Daemon to detect crashing apps
After=syslog.target

[Service]
ExecStart=/usr/sbin/abrttd
Type=forking

[Install]
WantedBy=multi-user.target
```

A little explanation of the contents of this file: The [Unit] section contains generic information about the service. systemd not only manages system services, but also devices, mount points, timer, and other components of the system. The generic term for all these objects in systemd is a unit, and the [Unit] section encodes information about it that might be applicable not only to services but also in to the other unit types systemd maintains. In this case we set the following unit settings: we set the description string and configure that

the daemon shall be started after Syslog[2], similar to what is encoded in the LSB header of the original init script. For this Syslog dependency we create a dependency of type `After=` on a systemd unit `syslog.target`. The latter is a special target unit in systemd and is the standardized name to pull in a syslog implementation. For more information about these standardized names see the `systemd.special(7)`. Note that a dependency of type `After=` only encodes the suggested ordering, but does not actually cause syslog to be started when `abrt` is – and this is exactly what we want, since `abrt` actually works fine even without syslog being around. However, if both are started (and usually they are) then the order in which they are is controlled with this dependency.

The next section is `[Service]` which encodes information about the service itself. It contains all those settings that apply only to services, and not the other kinds of units systemd maintains (mount points, devices, timers, ...). Two settings are used here: `ExecStart=` takes the path to the binary to execute when the service shall be started up. And with `Type=` we configure how the service notifies the init system that it finished starting up. Since traditional Unix daemons do this by returning to the parent process after having forked off and initialized the background daemon we set the type to `forking` here. That tells systemd to wait until the start-up binary returns and then consider the processes still running afterwards the daemon processes.

The final section is `[Install]`. It encodes information about how the suggested installation should look like, i.e. under which circumstances and by which triggers the service shall be started. In this case we simply say that this service shall be started when the `multi-user.target` unit is activated. This is a special unit (see above) that basically takes the role of the classic SysV Runlevel 3[3]. The setting `WantedBy=` has little effect on the daemon during runtime. It is only read by the `systemctl enable` command, which is the recommended way to enable a service in systemd. This command will simply ensure that our little service gets automatically activated as soon as `multi-user.target` is requested, which it is on all normal boots[4].

And that's it. Now we already have a minimal working systemd service file. To test it we copy it to `/etc/systemd/system/abrt.service` and invoke `systemctl daemon-reload`. This will make systemd take notice of it, and now we can start the service with it: `systemctl start abrt.service`. We can verify the status via `systemctl status abrt.service`. And we can stop it again via `systemctl stop abrt.service`. Finally, we can enable it, so that it is activated by default on future boots with `systemctl enable abrt.service`.

The service file above, while sufficient and basically a 1:1 translation (feature- and otherwise) of the SysV init script still has room for improvement. Here it is a little bit updated:

```
[Unit]
Description=ABRT Automated Bug Reporting Tool
After=syslog.target

[Service]
Type=dbus
BusName=com.redhat.abrt
ExecStart=/usr/sbin/abrt -d -s

[Install]
WantedBy=multi-user.target
```

So, what did we change? Two things: we improved the description string a bit. More importantly however, we changed the type of the service to dbus and configured the D-Bus bus name of the service. Why did we do this? As mentioned classic SysV services daemonize after startup, which usually involves double forking and detaching from any terminal. While this is useful and necessary when daemons are invoked via a script, this is unnecessary (and slow) as well as counterproductive when a proper process babysitter such as systemd is used. The reason for that is that the forked off daemon process usually has little relation to the original process started by systemd (after all the daemonizing scheme's whole idea is to remove this relation), and hence it is difficult for systemd to figure out after the fork is finished which process belonging to the service is actually the main process and which processes might just be auxiliary. But that information is crucial to implement advanced babysitting, i.e. supervising the process, automatic respawning on abnormal termination, collecting crash and exit code information and suchlike. In order to make it easier for systemd to figure out the main process of the daemon we changed the service type to dbus. The semantics of this service type are appropriate for all services that take a name on the D-Bus system bus as last step of their initialization[5]. ABRT is one of those. With this setting systemd will spawn the ABRT process, which will no longer fork (this is configured via the `-d -s` switches to the daemon), and systemd will consider the service fully started up as soon as `com.redhat.abrt` appears on the bus. This way the process spawned by systemd is the main process of the daemon, systemd has a reliable way to figure out when the daemon is fully started up and systemd can easily supervise it.

And that's all there is to it. We have a simple systemd service file now that encodes in 10 lines more information than the original SysV init script encoded in 115. And even now there's a lot of room left for further improvement utilizing more features systemd offers. For example, we could set `Restart=restart-always` to tell systemd to automatically restart this service when it dies. Or, we could use `OOMScoreAdjust=-500` to ask the kernel to please leave this process around when the OOM killer wreaks havoc. Or, we could use `CPU schedulingPolicy=idle` to ensure that `abrt` processes crash dumps in background only, always allowing the kernel to give preference to whatever else might be running and needing CPU time.

For more information about the configuration options mentioned here, see the respective man pages `systemd.unit(5)`, `systemd.service(5)`, `systemd.exec(5)`. Or, browse all of `systemd`'s man pages.

Of course, not all SysV scripts are as easy to convert as this one. But gladly, as it turns out the vast majority actually are.

That's it for today, come back soon for the next installment in our series.

6 Killing Services

Killing a system daemon is easy, right? Or is it?

Sure, as long as your daemon persists only of a single process this might actually be somewhat true. You type `killall rsyslogd` and the syslog daemon is gone. However it is a bit dirty to do it like that given that this will kill all processes which happen to be called like this, including those an unlucky user might have named that way by accident. A slightly more correct version would be to read the `.pid` file, i.e. `kill 'cat /var/run/syslogd.pid'`. That already gets us much further, but still, is this really what we want?

More often than not it actually isn't. Consider a service like Apache, or `crond`, or `atd`, which as part of their usual operation spawn child processes. Arbitrary, user configurable child processes, such as cron or at jobs, or CGI scripts, even full application servers. If you kill the main `apache/crond/atd` process this might or might not pull down the child processes too, and it's up to those processes whether they want to stay around or go down as well. Basically that means that terminating Apache might very well cause its CGI scripts to stay around, reassigned to be children of `init`, and difficult to track down.

`systemd` to the rescue: With `systemctl kill` you can easily send a signal to all processes of a service. Example:

```
# systemctl kill crond.service
```

This will ensure that `SIGTERM` is delivered to all processes of the `crond` service, not just the main process. Of course, you can also send a different signal if you wish. For example, if you are bad-ass you might want to go for `SIGKILL` right-away:

```
# systemctl kill -s SIGKILL crond.service
```

And there you go, the service will be brutally slaughtered in its entirety, regardless how many times it forked, whether it tried to escape supervision by double forking or fork bombing.

Sometimes all you need is to send a specific signal to the main process of a

service, maybe because you want to trigger a reload via SIGHUP. Instead of going via the PID file, here's an easier way to do this:

```
# systemctl kill -s HUP --kill-who=main crond.service
```

So again, what is so new and fancy about killing services in systemd? Well, for the first time on Linux we can actually properly do that. Previous solutions were always depending on the daemons to actually cooperate to bring down everything they spawned if they themselves terminate. However, usually if you want to use SIGTERM or SIGKILL you are doing that because they actually do not cooperate properly with you.

How does this relate to `systemctl stop`? `kill` goes directly and sends a signal to every process in the group, however `stop` goes through the official configured way to shut down a service, i.e. invokes the stop command configured with `ExecStop=` in the service file. Usually `stop` should be sufficient. `kill` is the tougher version, for cases where you either don't want the official shutdown command of a service to run, or when the service is hosed and hung in other ways.

(It's up to you BTW to specify signal names with or without the SIG prefix on the `-s` switch. Both works.)

It's a bit surprising that we have come so far on Linux without even being able to properly kill services. `systemd` for the first time enables you to do this properly.

7 The Three Levels of "Off"

In `systemd`, there are three levels of turning off a service (or other unit). Let's have a look which those are:

1. You can stop a service. That simply terminates the running instance of the service and does little else. If due to some form of activation (such as manual activation, socket activation, bus activation, activation by system boot or activation by hardware plug) the service is requested again afterwards it will be started. Stopping a service is hence a very simple, temporary and superficial operation. Here's an example how to do this for the NTP service:

```
$ systemctl stop ntpd.service
```

This is roughly equivalent to the following traditional command which is available on most SysV inspired systems:

```
$ service ntpd stop
```

In fact, on Fedora 15, if you execute the latter command it will be transparently converted to the former.

2. You can disable a service. This unhooks a service from its activation triggers. That means, that depending on your service it will no longer be activated on boot, by socket or bus activation or by hardware plug (or any other trigger that applies to it). However, you can still start it manually if you wish. If there is already a started instance disabling a service will not have the effect of stopping it. Here's an example how to disable a service:

```
$ systemctl disable ntpd.service
```

On traditional Fedora systems, this is roughly equivalent to the following command:

```
$ chkconfig ntpd off
```

And here too, on Fedora 15, the latter command will be transparently converted to the former, if necessary.

Often you want to combine stopping and disabling a service, to get rid of the current instance and make sure it is not started again (except when manually triggered):

```
$ systemctl disable ntpd.service
$ systemctl stop ntpd.service
```

Commands like this are for example used during package deinstallation of systemd services on Fedora.

Disabling a service is a permanent change; until you undo it it will be kept, even across reboots.

3. You can mask a service. This is like disabling a service, but on steroids. It not only makes sure that service is not started automatically anymore, but even ensures that a service cannot even be started manually anymore. This is a bit of a hidden feature in systemd, since it is not commonly useful and might be confusing the user. But here's how you do it:

```
$ ln -s /dev/null /etc/systemd/system/ntpd.service
$ systemctl daemon-reload
```

By symlinking a service file to `/dev/null` you tell systemd to never start the service in question and completely block its execution. Unit files stored in `/etc/systemd/system` override those from `/lib/systemd/system` that carry the same name. The former directory is administrator territory, the latter territory of your package manager. By installing your symlink in `/etc/systemd/system/ntpd.service` you hence make sure that systemd will never read the upstream shipped service file `/lib/systemd/system/ntpd.service`.

systemd will recognize units symlinked to `/dev/null` and show them as masked. If you try to start such a service manually (via `systemctl start`

for example) this will fail with an error.

A similar trick on SysV systems does not (officially) exist. However, there are a few unofficial hacks, such as editing the init script and placing an exit 0 at the top, or removing its execution bit. However, these solutions have various drawbacks, for example they interfere with the package manager.

Masking a service is a permanent change, much like disabling a service.

Now that we learned how to turn off services on three levels, there's only one question left: how do we turn them on again? Well, it's quite symmetric. Use `systemctl start` to undo `systemctl stop`. Use `systemctl enable` to undo `systemctl disable` and use `rm` to undo `ln`.

8 Changing Roots

As administrator or developer sooner or later you'll encounter `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 bootstrapped 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 script,

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

ing 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 com-

plete 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.

9 References