Daemon Management Under Systemd

ZBIGNIEW JĘDRZEJEWSKI-SZMEK AND JÓHANN B. GUÐMUNDSSON

Zbigniew Jędrzejewski-Szmek works in a mixed experimental-computational neuroscience lab and writes stochastic simulators and programs for the analysis of experimental data. In his free time he works on systemd and the Fedora Linux distribution. zbyszek@in.waw.pl

Jóhann B. Guðmundsson, Penguin Farmer, IT Fireman, Archer, Enduro Rider, Viking-Reenactor, and general insignificant being in an insignificant world, living in the middle of the North Atlantic on an erupting rock on top of the world who has done a thing or two in distributions and open source. johannbg@gmail.com

The systemd project is the basic user-space building block used to construct a modern Linux OS. The main daemon, systemd, is the first process started by the kernel, and it brings up the system and acts as a service manager. This article shows how to start a daemon under systemd, describes the supervision and management capabilities that systemd provides, and shows how they can be applied to turn a simple application into a robust and secure daemon. It is a common misconception that systemd is somehow limited to desktop distributions. This is hardly true; similarly to the Linux kernel, systemd supports and is used on servers and desktops, but it is also in the cloud and extends all the way down to embedded devices. In general it tries to be as portable as the kernel. It is now the default on new installations in Debian, Ubuntu, Fedora/RHEL/CentOS, OpenSUSE/SUSE, Arch, Tizen, and various derivatives.

Systemd refers both to the system manager and to the project as a whole. In addition to systemd, the project provides a device manager (systemd-udevd), a logging framework (systemd-journald), and a daemon to keep track of user sessions (systemd-logind). For server and VM environments, reliability, control over daemons, and uniform management are possibly the most important focus points, and for desktop, emphasis is placed on a whole-system view of users, secure access to hardware, and quick boot times. The latter is also important for VMs, containers, and embedded devices. Watchdog integration, factory reset, and read-only root are useful for containers and embedded devices. Systemd also has very strong integration with LSMs, with SELinux and AppArmor support coming mainly from the server and container users, and SMACK (Simplified Mandatory Access Control Kernel) used in smartphones and embedded devices.

Uniformity is good for sysadmins and developers alike, and the systemd project has become the de facto standard base of Linux systems. After the last release of systemd, packages for Fedora, Debian, and Ubuntu appeared on the same day. This creates synergy and allows developers from all distributions to participate directly in upstream development, which in turn has led to a renewed focus on bug fixes and new features. Relying on the presence of systemd in all major distributions and across the stack makes it possible to make full use of functions that systemd provides. This article will strive to show how simpler and more secure daemons can be created.

Systemd Units

Systemd keeps the state of the system in a graph of interconnected "units." The most important are "services," i.e., daemons and scripts, with each service unit composed of one or more processes. Other unit types encapsulate resources ("device," "mount," "swap," "socket," "slice" units), group other units ("target," "snapshot," "scope"), or trigger other units under certain conditions ("timer," "path," and "socket" units). Units *may* be configured through files on disk, but this is not necessary for all unit types. Device units are dynamically created based on the device tree exported by the kernel and managed by udev. Similarly, mount and

swap units are usually generated from /etc/fstab but are also dynamically based on the current set of mounted file systems and used swap devices.

Unit files on disk have an extension that corresponds to the type of the unit (e.g., httpd.service, sshd.socket). Unit files are plaintext files in a syntax inspired by the desktop entry specification [1]. Out of the 12 unit types, this article only deals with two (.service and .socket) and only a few configuration options. The full set of configuration directives in unit files is rather large (258 as of systemd release 219), so the reader is referred to the copious documentation [2].

The Basics of Systemd Services

Systemd manages daemons and other jobs as "services." Each service is described by a simple declarative text file that lists the commands to execute. The service unit file also contains a short description of the service, pointers to documentation, and a list of dependencies on other services.

Service files have the .service extension and are usually located in /usr/lib/systemd/system, when they are installed as part of a package, or in /etc/systemd/system, when they are local configuration.

systemctl is the tool used to manipulate and check unit status.

This article will use a simple Python server as an example, and you can download the script and other files [3] and follow along. To keep things simple, but not totally trivial, the server will provide a hashing service and will respond with cryptographic hashes of the data it is sent.

A unit file for this server could be:

```
# /etc/systemd/system/hasher.service
[Unit]
Description=Text hashing service
Documentation=https://example.com/hasher
[Service]
ExecStart=/usr/bin/python -m hasher
```

That's it—after copying this file to one of the directories listed above, the daemon can be started with

```
systemctl start hasher
```

and stopped with

```
systemctl stop hasher
```

Wrapping daemons in initialization scripts that parse options and prepare state is discouraged. Systemd service configuration is intentionally not a programming language, and the logic of service initialization and state transitions is embedded in the boot manager (systemd) itself. Preferably, the daemon should be able to be launched directly. For the cases where this is not possible, it *is* possible to invoke arbitrary shell commands by either embedding them directly as calls to /bin/sh -c '...' in the unit file or by executing an external script.

/usr, /etc, /run Hierarchies and .d Snippets

Systemd units that are distributed as part of the operating system are installed in /usr/lib/systemd/system/ (/etc/systemd/system/ is reserved for the administrator). If a service with the same name appears in both places, the one in /etc is used. This allows the administrator to override distro configuration. /run/systemd/system serves a similar purpose and can be used to add temporary overrides, with a priority higher than /usr but lower than /etc.

Adding a unit file with the same name completely replaces the existing unit. Very often, just a modification or extension of the original unit is wanted. This is achieved by so called "drop-ins"—configuration files with like syntax that are located in a directory with the same name as the unit, suffixed with ".d," (e.g., hasher.service.d/).

For example, we would like to run our service under Python 3. We are not sure how this will work out, so we create an override in /run. It will be wiped out after reboot. "systemctl edit" can be used to conveniently create drop-in snippets: it will create the directory and launch an editor.

```
$ systemctl edit --runtime hasher
# /run/systemd/system/hasher.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/bin/python3 -m hasher
```

If we just added a new ExecStart line, systemd would merge them into a list of things to execute. By specifying an empty ExecStart=, we first clear the previous setting.

After updating unit configuration, the changes are not picked up automatically. This restarts the unit using the updated configuration:

```
$ systemctl daemon-reload
$ systemctl restart hasher
```

"systemctl cat" can be used to print the main service configuration file and all drop-ins.

Socket Activation

Socket activation was one of the early flagship features of systemd, introduced in current form in the original systemd announcement [4]. Motivation included simplification of daemons, simplification of dependencies between them, and uniform configuration of sockets on which daemons listen. Since then, the rest of systemd has grown, but socket activation remains a crucial building block and has become the basis of various security features.

Socket activation, depending on the context, can mean a few things. Let's tackle them one at a time.

The most important part of socket activation is that the daemon does not itself create the socket it will listen on, but it inherits the socket as a file descriptor (file descriptor 3, right after standard input, output, and error). This socket can either be a listening socket, which means that the daemon will have to listen(2) on it and serve incoming connections from clients, or just a single connection, that is, a socket received from accept(2). This latter version is rather inefficient, since a separate process is spawned for each connection, so this article will only describe the first version. In this version, after the daemon has been spawned, there is absolutely no difference in efficiency compared to the situation in which the daemon itself opens the sockets it listens on.

The way in which systemd informs the daemon that the sockets have already been opened for it is by means of two environment variables. $LISTEN_FDS contains the number of sockets. For example, for an httpd server, which listens on both HTTP and HTTPS (ports 80 and 443), those two sockets could be given as file descriptors 3 and 4, and $LISTEN_FDS would be 2. The second environment variable, $LISTEN_PID, sets the process identifier of the daemon. If the daemon spawns children but forgets to unset $LISTEN_FDS, this second variable acts as a safety feature because those children will know that $LISTEN_FDS was not addressed to them.

Systemd provides a library (libsystemd), which contains a utility function [5] to check $LISTEN_PID and query $LISTEN_FDS. sd_listen_fds() will unset those variables so they are not inherited by children. Nevertheless, if libsystemd is not a good fit for any reason, this protocol is so simple that it can be trivially reimplemented in any language that allows file descriptors to be manipulated.

To configure socket activation for a systemd service, a .socket file is used. Continuing with our example, the following would cause systemd to open TCP port 9001 for our daemon:

```
# /etc/systemd/system/hasher.socket
[Unit]
Description=Text hashing service socket
[Socket]
ListenStream=9001
```

By default a .socket unit is used with the .service unit of the same name, so we don't need to name hasher.service explicitly.

Types of Socket Activation

The word "activation" in "socket activation" implies that the connection to the socket causes the daemon to start. This used to be true (under inetd) but is just one possibility under systemd. In general, systemd can be configured to activate services for more than one reason, combining the functionality that was traditionally split between the init scripts, inetd, crond, and even anacron. A service can be configured to always start on boot or to start as a dependency of another service, which corresponds to the traditional "start at boot" semantics. It can also be started as a result of an incoming connection, like inetd would do. It can also be started at a specific time or date, a certain time after boot, after some interval after it last stopped running, which covers the functionality provided by crond and anacron, and a bit more. Some more esoteric triggers, like a file being created in a directory or another daemon failing, can also be used.

Having all this functionality in the system manager has certain advantages. The way that the daemon is started is configured only once. Race conditions between different activation mechanisms are handled gracefully: if a connection comes in while the daemon is still initializing, it will be serviced once the daemon is ready. If a daemon has to be started as a dependency of two different daemons, it will be started just once. If a cron-job-style service that is supposed to be started every day takes a few days to run, it will not be started twice.

Returning to a socket activated daemon, this daemon will inherit its socket or sockets already open. It can be configured to be started at boot, in which case socket activation only means that the sockets are opened before and not after the fork. The daemon can also be configured to start lazily on an incoming connection, in which case this first connection will not be lost, but it will be handled with a delay because the daemon needs to start first. In case of subsequent connections there is no difference in either case. Systemd "units" that describe the service and the socket can be written to allow both modes to be supported and can be enabled with a single command. How to do this will be described in the next section.

Systemd supports IPv4 and IPv6, and TCP and UDP sockets, but it also supports UNIX sockets, FIFOs, POSIX message queues, character devices, /proc and /sys special files, and netlink sockets. All those can be passed to the daemon using the socket activation protocol. Systemd will also configure various TCP/IP socket options, the congestion algorithm and listen queue size, binding to a specific network device, and permissions on UNIX sockets. The author of the daemon still needs to write code to support stream or datagram connections of course. The advantage is that they need not bother with writing code to parse addresses and configure different protocols and families. The advantage for the administrator is that all that can be uniformly

configured and enabled with simple declarative switches in the unit file, if the daemon provides necessary support.

So far our daemon was listening on the wildcard address (::). Let's say we would like to have it listen on the specific address 10.1.2.3 instead. The kernel will not allow binding to an address before it has been configured. To avoid having to synchronize with network initialization, we can use IP_FREEBIND, controlled by unit option FreeBind.

The updated unit file looks like this:

```
# /etc/systemd/system/hasher.socket
...
[Socket]
ListenDatagram=10.1.2.3:9001
FreeBind=yes
```

Enabling Units to Start by Default

The previous section mentioned that systemd services can be configured to start based on a few different conditions, including starting "at boot." Systemd groups services, which are described by .service units, into targets, described by .target units, for easier management. During boot, systemd starts a single target, including all of its dependencies. So starting a service during boot simply means adding it to the right target. For normal services this is multi-user.target, which contains everything that is part of a normally running system.

Back to our example, since our service does not work unless systemd hands it an open socket, we add a dependency on the .socket unit. We also specify how the service should be enabled. We append to the .service file:

```
# /etc/systemd/system/hasher.service
[Unit]
Description=Text hashing service
Requires=hasher.socket
[Service]
...
[Install]
WantedBy=multi-user.target
```

which allows the administrator to enable and disable the service. Let's do that:

```
$ systemctl enable hasher.service
Created symlink from /etc/systemd/system/multi-user.target.
wants/hasher.service to /etc/systemd/system/hasher.service.
```

This symlink encodes the dependency. Creating a symlink from a .wants/ directory is an alternative to specifying Wants=hasher in the multi-user.target file that does not require modifying the unit file. Specifying all dependencies a unit needs to successfully start is the principle underlying service management by systemd. In this specific case, systemd tries to start multi-user .target, which depends on hasher.service, which depends on hasher.socket, so the socket will be started first, then the service, and in the end systemd will announce that it has reached the specified target.

Socket units are described by .socket files. They too can be configured to be created at boot by adding them to a target. For sockets this is sockets.target, which itself is part of multi-user.target. Similarly to .service units, we add installation instructions to the socket unit:

```
# /etc/systemd/system/hasher.socket
...
[Install]
WantedBy=sockets.target
```

When the socket+service pair is written this way, the administrator can enable just the socket to have lazy activation on the first connection, or can enable the service to always start it during boot.

Let's make our daemon lazily activated:

```
$ systemctl disable hasher.service
Removed symlink /etc/systemd/system/multi-user.target.wants/
hasher.service.
$ systemctl enable hasher.socket
Created symlink from /etc/systemd/system/sockets.target.
wants/hasher.socket to /etc/systemd/system/hasher.socket.
```

Using Socket Activation to Resolve Dependencies

Sockets.target is actually started early during boot. This means that systemd opens the sockets even before the services that will handle the connections can be started. Traditionally, the administrator had to make sure that services are started in the right order, so that they manage to open their sockets before the services which will try to connect to those sockets are started. Socket activated services can be started in parallel, even if they are interdependent. A service that connects to a socket belonging to a daemon that hasn't started yet will simply wait. As long as no loops exist, those dependencies are resolved without any explicit configuration.

Incidentally, opening sockets in this fashion ensures that they are not accidentally opened by a different program, solving the problem that portreserve deals with.

Failure Handling

There are two schools of thought on how to handle crashing daemons. The first states that daemons should not crash and must be fixed. The second states that crashing daemons are a fact of life and have to be dealt with. Systemd watches the status of all services and will notice if the main process of a service exits for

any reason. The manager can be configured to take some action. Restart=on-failure, a common setting that causes a service to be restarted when it exits with a non-zero exit code, is killed by SIGSEGV, SIGABRT, or similar, or if a timeout is hit.

To monitor service health, systemd supports basic watchdog functionality. When WatchdogSec= is specified in the service file, systemd provides the $WATCHDOG_USEC variable in the environment of the service and expects periodic notifications with the sd_notify() call. When the service does not send the heartbeat signal, it will be aborted, and possibly restarted, depending on the settings described above.

The watcher is also watched. The manager can be configured to enable the hardware watchdog (with RuntimeWatchdogSec= and ShutdownWatchdogSec= settings) to allow the system to be automatically restarted if PID 1 stops responding. This way a chain of supervision from the hardware to the leaf services is established.

The Journal and Log Labeling

One of the most debated aspects of systemd is its log handling. Systemd-journald is one of the non-optional parts of systemd and is one of the first services to start and one of the last services to be stopped. All messages are stored in a binary format in /var/log/journal/<machine-id> and can be read using journalctl or using sd_journal_get_data(3) and related functions found in libsystemd. Even traditional syslog daemons nowadays usually do this, and get their data from binary journal files before writing them to text files. There are reasons for this organization.

Systemd tries very hard not to lose any log messages. All messages from services will be captured and processed by systemd-journald. This includes syslog messages and anything written to standard output and standard error. Daemons that crash, especially during startup, will often print the cause to standard error, so it is nice to capture those messages too. In addition, structured messages can be sent to the journal through a custom UNIX socket. sd_journal_print(), and related functions in libsystemd provide this functionality. The advantage is that in addition to the main message, additional fields can be attached. Each entry in the journal is composed of a group of FIELD=VALUE pairs. The "message" is stored as MESSAGE field, and other fields can carry text or binary content. In fact, sd_journal_print() will by default attach the source code filename and line.

The binary format which systemd-journald uses indexes messages based on field names and field values. Taking our hashing daemon as an example, it could add the remote address and port as additional fields when using native journal logging. The journal can then be queried for messages about a certain remote from certain dates without searching through all logs:

```
    journal.send('New connection on fd={} from {}:{}'.format(fd,
address, port),
                 ADDRESS=address[0],
                 PORT=port)
```

Traditional syslog messages include an identifier, usually the program name. In journal messages this field is called SYSLOG_IDENTIFIER and is controlled by the sender.

```
systemctl -t <identifier>
```

can be used to query messages with a certain identifier.

Another important aspect of structured logs is that journald attaches some fields that specify the provenance of the message and cannot be faked or modified by the sender. Those fields are labeled with a leading underscore (e.g., _PID, _UID, _GID, _SYSTEMD_UNIT for the process identifier, user and group of the sender, and the systemd service containing the process). Traditional syslog does not have anything like this and allows any sender to send messages on behalf of any daemon. Systemd makes extensive use of those additional fields. When starting a service or logging anything related to a service, messages about the service are tagged with the name of the service. In addition, privileged daemons like setroubleshoot will also tag messages as pertaining to a certain service. Systemd-journald also reads audit messages using the netlink socket and parses them to extract process identifiers and other metadata. This allows all messages *from* a service and *about* a service to be retrieved with a simple

```
journalctl -u <service>
```

command. This same functionality is used when showing service status:

```
systemctl status <service>
```

will list the processes being part of the service, and what systemd knows about the service, but also the last ten lines of logs pertaining to the service.

If we were to start hasher.service

```
$ systemctl start hasher
```

we could see messages from systemd and from the daemon interwoven:

```
$ journalctl -u hasher
Feb 22 19:15:12 fedora systemd[1]: Starting Text hashing
service...
Feb 22 19:15:12 fedora python3[222]: /usr/bin/python3: No
module named hasher
Feb 22 19:15:12 fedora systemd[1]: hasher.service: main
process exited, code=exited, status=1/FAILURE
Feb 22 19:15:12 fedora systemd[1]: Unit hasher.service
entered failed state.
```

How does journalctl know which messages to show? If we also show the auxiliary data, we can see that messages are tagged with either _PID=1 and UNIT=hasher.service or _SYSTEMD_UNIT=hasher.service. The first comes from PID 1, and journalctl knows that it can trust the UNIT= field. The second is tagged as coming from the service itself.

Systemd-journald is started very early. In fact, if systemd is used in the initramfs, systemd-journald runs in the initramfs, writing logs to temporary storage under /run/log/. After the transition to the main file system, systemd-journald continues writing to /run/log, and then flushes those logs to /var/log/ after /var/ has been mounted. This means that logs from early boot are available just like the rest.

Systemd-journald watches the amount of free disk space and will not allow journal files to eat up all available space. In the default configuration it will cap the total space used and also leave a certain percentage of the disk free.

Security Features

A very simple yet effective way to limit the damage that a hacked or misbehaving service can do is to run it under its own user. The primary reason for socket activation is the simplification of network daemons and their lazy activation. But it has implications for security, too. If a daemon does not open sockets by itself, it can be less privileged. A daemon that wants to listen on a port below 1024 can be started under root, open the port itself, and then do the fairly complicated transition to unprivileged user itself. But if systemd opens the port for it, it can run as the unprivileged user from the start.

Letting systemd take care of the user transition is trivial: use the set User= option.

Our process could still transition back by running a SUID binary. We can disallow this and any other transitions or privilege escalation with NoNewPrivileges=yes.

Some daemons need to run as root, but systemd can still restrict them by using mount and network namespaces which limit their view of the world. A group of settings use mount namespaces [6] to curtail access to the file system. ProtectHome= and ProtectSystem= are the high-level options. The first can be used to present /home to the daemon as either empty or read-only, and the second will make /usr, /boot, and optionally /etc read-only for the daemon. ReadOnlyDirectories=, ReadWriteDirectories=, InaccessibleDirectories= are the low level settings that do what their names suggest.

Using predictable file names in shared temporary directories is a common source of denial-of-service and security vulnerabilities. PrivateTmp= setting uses mount namespaces to give private /tmp and /var/tmp directories to the daemon. This protects both the daemon from users and other daemons, and others from the daemon.

Systemd uses network namespaces to prevent a daemon from using the network. A service running with PrivateNetwork=yes sees only a private loopback device. If the daemon is compromised, it cannot be used to exfiltrate data or attack other hosts.

Paradoxically, socket-activated network daemons are often started with PrivateNetwork=yes. This means that they can be run locked down as an unprivileged user, and their only means of contact with the network is through the sockets inherited from systemd.

Let's turn those additional protections on for our service:

```
# /etc/systemd/system/hasher.service
…
[Service]
User=hasher          ← the primary group of the user is used
too if Group= is not specified
NoNewPrivileges=yes
ProtectHome=yes
ProtectSystem=full    ← "full" includes /etc in addition to /
usr and /boot
PrivateTmp=yes
PrivateNetwork=yes
```

After the service is restarted, it has its own network namespace with a private lo device, cannot see /home or write to /usr, /boot, and /etc even if the file access mode would allow. What the service sees as /tmp is really a directory /tmp/systemd-private-<bootid>-hasher.service-<gibberish>/tmp.

SELinux and Other Linux Security Modules

There are two parts to the integration with a security module: the first part is that systemd will perform initial SELinux configuration when the system is brought up. Systemd is aware of SELinux contexts, so when creating files or opening sockets systemd will label them properly. The second part is the ability to override default domain transitions with configuration in unit files. The SELinuxContext= setting can be used to set the context of executed processes. Similar support and settings exist for AppArmor and SMACK. Integrating support directly in the boot manager means that initialization is performed very early, thus LSM protection includes the early boot, and any initialization errors are caught and will be treated as fatal if necessary. It should be noted that SELinuxContext is logged as a trusted metadata in the journal.

Resource Limits

Traditional UNIX resource limits were applied per process (niceness, virtual memory size, CPU usage, open files) or per user (number of processes). Those limits are still supported and

can be set automatically with Nice=, LimitDATA=, LimitCPU=, LimitNOFILE=, LimitNPROC=, and others [7].

Each systemd service runs in its own control group. Cgroups allow resource limits to be applied at the level of the whole service or group of services, and to partition resources more fairly. By default, systemd only uses the cgroup hierarchy to keep track of forked children of various services. Additional settings can be used to turn on specific controllers and constrain resource usage. It should be noted that this is not free, and especially the memory controller is known for its high overhead. Systemd presents a simplified subset of the functionality provided by the kernel, and can limit and partition CPU usage (CPUShares=), memory usage (MemoryLimit=), and block device bandwidth (BlockIOWeight=, BlockIOReadBandwidth=, BlockIOWriteBandwidth=).

Automated Management

Systemd binaries generally produce two kinds of output on the console: colorful tabularized output for human consumption, and plain output useful for scripting. This second type is stable [8] and can be used as a basis for management tools. The most popular tools like Chef, Puppet, and Salt all provide similar functionality that wraps calls to systemctl enable/disable/start/stop/restart/is-active and can be used to manage units in a centralized manner.

Systemctl is actually a wrapper around the D-Bus API of systemd and defers operations on units to it. The status of all units is also available over D-Bus, and D-Bus property notifications can be used for live updates. The kcmsystemd control module for KDE uses this, and hopefully more tools will in the future.

Summary

The systemd stack has grown over the last few years, to the point where it is simply impossible to describe more than some aspects in a short article like this. We show how the facilities provided by systemd can be used to build a secure daemon by making use of user separation, namespaces, control groups, and resource limits. Those futures are provided by the Linux kernel but are not as widely used as they should be because of the additional work required to support them. Our daemon (although written in Python) is also fairly efficient and robust: it uses epoll, provides extensive logs, and will be monitored and restarted if necessary. We hope that this proves that systemd makes the lives of developers and programmers more pleasant.

Resources

[1] http://standards.freedesktop.org/desktop-entry-spec/latest/.

[2] http://www.freedesktop.org/software/systemd/man/systemd.directives.html#Unit%20directives.

[3] Python script and configuration examples: http://in.waw.pl/git/login-article-example/.

[4] The original systemd announcement: http://0pointer.de/blog/projects/systemd.html.

[5] sd_listen_fds() in libsystemd: http://www.freedesktop.org/software/systemd/man/sd_listen_fds.html.

[6] James Bottomley and Pavel Emelyanov, "Containers," *;login:*, vol. 39, no. 5, October 2014: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/containers.

[7] CPU limits for systemd services: http://www.freedesktop.org/software/systemd/man/systemd.exec.html#LimitCPU=.

[8] Systemd interface stability: http://www.freedesktop.org/wiki/Software/systemd/InterfacePortabilityAndStabilityChart/.