14 Years of systemd

Lennart Poettering
FOSDEM 2025
Brussels, Belgium

50 min

What is systemd again?

“systemd is a suite of basic building blocks for building a Linux OS. It provides a system and service manager that runs as PID 1 and starts the rest of the system.”

https://systemd.io/

Pre-History

Once upon a time there was sysvinit (*1992, design: *1983)

And then, there was Upstart (*2006, †2014)

Why not Upstart?

Didn’t solve the problem: admin/developer told the computer on which trigger to do what action, in order to build full tree of actions to reach some ultimate goal.

Tree is configured manually.

Our thinking instead: you specify the goal, computer figures out the rest.

Also: slow development, unfair contribution rules (“copyright assignment”).

Eventually: Babykit^Wsystemd

Transactional system.

Basic ideas hashed out during a flight back from Linux Plumbers Conference 2009, by Kay Sievers and yours truly.

Proper Open Source project, no copyright assignment, LGPL2.1

First commit *2009, first release *2010.

More than PID 1: a suite with all the basic components to bring up the system.

Influences

sysvinit & Upstart (duh!)

Apple’s launchd (*2005) (in particular: Socket Activation)

Solaris SMF (*2005)

+ one or two original thoughts (I think?)

Influences: UNIX?

What the heck is UNIX even?

Linux is not UNIX

UNIX: unified repository with kernel and userspace in one

Linux: everything split up, distributed ownership (aka: chaos)

Linux + systemd: still everything split up, just a little bit less.

Is this UNIX? Nope.

Closer to UNIX? Yeah, probably!

Influences: UNIX?

“Everything is a file”

(Is it really? Sorry, but my printer is not a file! 🙃)

“Shouldn’t running services be files too”?

`sysvinit`: nope, why would you think so?

systemd: kinda, maybe directories at least? (cgroupfs)

Influences: UNIX?

So is systemd UNIX?

Code-wise, legally: definitely not, no systemd code originates in UNIX.

Organization-wise: maybe a bit closer than Linux ever was?

Philosophically: yeah, I guess

Does it matter? Pretty sure it doesn’t really and it shouldn’t, OS design didn’t peak in 1969 and then suddenly fall of a cliff.

History, Part 2

And then, systemd became the default:

🌟 Fedora **2011** 🥳

openSUSE + Arch Linux **2012**

RHEL + SLES **2014**

Debian + Ubuntu **2015**

History, Part 3

2019: We have a logo!

(Thank you Tobias Bernard!)

https://brand.systemd.io/

[●◀] systemd

Conferences

systemd.conf 2015 + 2016, Berlin

Morphed into **All Systems Go!** 2017, still going strong today

https://all-systems-go.io/

See you in **Berlin Sep 30th – Oct 1st, 2025** for

All Systems Go! 2025

Where are we today?

All major distributions use it. *The world runs on it.* 🌍

Vibrant community (6 core maintainers, 60 people with commit access, > 2600 contributors)

~6 month primary release cycle (we want to do better) + regular minor stable releases

Grants by STF + donations managed by SPI

What is systemd today?

Suite of **~150** separate binaries

Quite modular

Most components optional, except for the core: the service manager, i.e. PID 1

(Almost) all in C

The common core of what a Linux-based OS needs, according to our PoV

Covers all kinds of basic OS functionality: `systemd-logind`, `systemd-resolved`, `systemd-networkd`, `systemd-timesyncd`, `systemd-homed`, …

👣 Footprint

~690K SLOC

(compare: `wpa_supplicant` ~460K or `glibc` ~1.4M)

On Fedora, full blown install is 36M bytes (compare: `bash` ~8M)

Weak dependencies → `dlopen()`

Hence, no hard deps besides **`glibc`**, **`libmount`**, **`libcap`**, `libselinux`, `libaudit`, `libseccomp` (latter three: build-time optional)

`dlopen()` → also think xz/Jia Tan mess

Executive summary: it’s not that big, but not tiny either

Suitable for inclusion in initrds, and containers

What belongs into systemd?

Needs to be generic

Needs to be foundational

Needs to be common

Needs to have a future

Needs to be clean

Needs to follow a common style

💡 systemd is Concepts

Example: Clear separation of `/etc/`, `/run/`, `/usr/`

Example: Hermetic `/usr/`

Example: Merging of drop-in configuration files (e.g. `/etc/tmpfiles.d/`, `/usr/lib/tmpfiles.d/`, `/run/tmpfiles.d/`)

Example: Declarative, not imperative (no shell)

Permits focus on high-level concepts, i.e. `ProtectHome=`, `ProtectSystem=`, which take benefit of above mentioned clear separation, and hermetic `/usr/`

systemd is Standards

We often intend to set standards, and we consume standards

Think: `/etc/os-release` – Now even on Solaris, and the BSDs

Place for our own and “adjacent” specs:

https://uapi-group.org/

Example: the DDI spec (“Discoverable Disk Images”) is based on the UEFI GPT spec (“GUID Partition Table”)

🚀 Goals and Challenges for the Future #1

(at least in Lennarts PoV, others on the team might have different priorities)

Let’s get Boot & System Integrity addressed.

All big OSes (Windows, Android + ChromeOS, Apple) have boot and system integrity implemented in one way or another.

Generic Linux though?

Uh. 😥

Why? It’s complex? Cultural issues? FUD by the FSF? [Jeez, they claim TPMs would be used for DRM!] Package-based world?

🔐 Boot & System Integrity #2

Step back: what does Boot & System Integrity mean?

Locking down what can be run on your system, at boot and during runtime:

Policy #1: only code your OS vendor considered OK (i.e. bind to your distro's code signing)

Or

Policy #2: only code *you* considered OK(i.e. bind to your own code signing key pair)

Or

Combination of both

🔐 Boot & System Integrity #3

Why?

Keeping attackers out of your system

Define well known, secure state to return to

→ **no persistency**

Or with other words: so that you don’t have to always sleep with your laptop under your pillow.

🔐 Boot & System Integrity #4

Status quo on bigger distros: code signing of the kernel, bound to MSFT certificates (“UEFI Secure Boot”). Code signing ends right after kernel, does not cover initrd or anything else (i.e. you can trivially backdoor initrds, they are not protected whatsoever). Security value questionable: massive denylist of known bad EFI executables at best.

Not *reasonably* possible to replace keys on PCs with your own.

No binding of disk encryption security to any of this.

🔐 Boot & System Integrity #5

Goal for the future is to move focus away from *Secure Boot*, and more towards *Measured Boot* (i.e. “TPM protected boot”).

Benefit: it’s the disk encryption that is locked to the software of your choice, but not the software itself that is locked down.

Or in other words: no restriction on what is run

However: only if the software of your choice – that you trust – is run, disk encryption is unlocked.

This is more democratic and a lot more focussed: *you* are in control – instead of your OS/distribution vendor.

Open Source folks should *love* this.

🔐 Boot & System Integrity #6

Net result 1: a more secure system

Better chance that leaving your laptop in your hotel room (even with disk encryption) isn't an *obvious* risk for the device to be backdoored.

Net result 2: reasonable protection of unattended systems.

Path to get there: measurements, FDE with TPM, UKIs, `systemd-boot`, …

(For more on this: my talks today, 10:35am + tomorrow 9:00am, here at FOSDEM 2025)

🚀 Goals and Challenges for the Future #2

Repivot our IPC system.

Let’s do more Varlink.

Why: works during early boot, drastically fewer roundtrips means better performance, easier to debug, allows processing IPC requests in one service instance per connection mechanism, more easily extensible, sensible flow control, worker pool compatibility, conceptual compatibility with Web/JSON world, easier to learn, compatible with socket activation, better security model, sensible delegation model, protocol “upgrade” mechanism, better IDL with NULL and enum type, optional arguments, and so on, documentation strings, etc. pp.

💬 Varlink #2

One particularly interesting feature: “*allows processing IPC requests in one service instance per connection*”.

Why does this matter?

1. Consider classic UNIX tools: receive input on STDIN, command parameters via command line, output on STDOUT.
2. Consider modern UNIX tools: same same, but now we speak JSON on STDIN and STDOUT. (Example: `findmnt --json`)
3. Varlink is a natural extension to this: same same, but now the command parameters are also encoded in JSON, as structured part of the input.

💬 Varlink #3

Small addition, major effect: now we can just bind the command to an AF_UNIX SOCK_STREAM socket via systemd's socket activation and it becomes an IPC service.

THIS IS FANTASTIC!

Consider all these little command line tools: we can now *easily* expose them as IPC service, with proper sandboxing/isolation, lazy activation, introspection, and so on.

💬 Varlink #4

Net result: systemd already now has more services exposing Varlink IPC than D-Bus IPC.

Your project could be next?

→ https://varlink.org/

Goal hence is: let’s open up *everything* in systemd via Varlink IPC, so you never have to shell out anymore, scrape output, run with elevated privileges, and so on.

Idea: someone should build a shell, that deals in JSON objects (hey jq), works natively with Varlink IPC (and the various flavours of JSON over HTTP), as well as programs that take JSON as input and/or output, but retains UNIX concepts such as shell pipelines (JSON-SEQ?) and so on.

🚀 Goals and Challenges for the Future #3

C has limits

We are doing pretty well, vulnerability-wise: if you accept CVE as metric, we had 3 in 2023, none in 2024.

(CVEs mostly not memory related)

CI hooked up to static analyzers, sanitizers, fuzzers and so on. That helps.

We heavily use RAII concepts in C, via GCC cleanup attribute, `TAKE_PTR()` and similar. That helps a lot.

But: the future speaks 🦀 Rust (probably)

🦀 Rust #2

We started with the oxidation process (systemd-zram-generator)

(Not part of main repository yet though)

What so far slowed us down: we have a complex build, with many binaries and tests, with a many interdependencies. Not fit for Cargo. And Meson didn’t use to like Rust too much. The impedance mismatch sucked, and is not something we wanted to work around ourselves.

But: supposedly addressed now.

🦀 Rust #3

So, what’s the problem now?

systemd: 150 binaries + sensitive to footprint issues + heavily reliant on shared libraries (internally, and externally + with `dlopen()`)

Rust: dynamic libraries simply still *not there*

Static linking of 150 binaries simply not an option for us (too big!), and sticking to C linking only is neither.

Yes, doing Rust in kernel is much easier, since mostly a single big static binary.

Not an option for us: pioneer Rust development + systematically rely on unstable language features.

Hence, please, dear Rust people: *we need shared libraries to be 1st class supported*.

Also: currently, maintainers might know Rust, but they don’t ***know*** Rust.

🦀 Rust #4

Challenge:

Maybe we should push the Rust & Zig people (and Go?) into a contest:

Who can deliver a memory safe language that can check all our boxes, i.e.:

1. Reasonably stable shared libraries, with dlopen() and ABI for native language features
2. Good support for building hybrid codebases with meson & co
3. Nice C interfacing

🚀 Goals and Challenges for the Future #4

Let’s push the ecosystem away from package-based OS deployment to image-based OS deployment. Move image-based OSes into focus.

The systemd community has a tool for that: `mkosi`

Why? Robustness, reproducibility, security

What does this mean: focus more on cattle, less on pets. Have fewer local variables in the system, use a more uniform combination of software in your installations.

This includes desktop systems!

That’s All