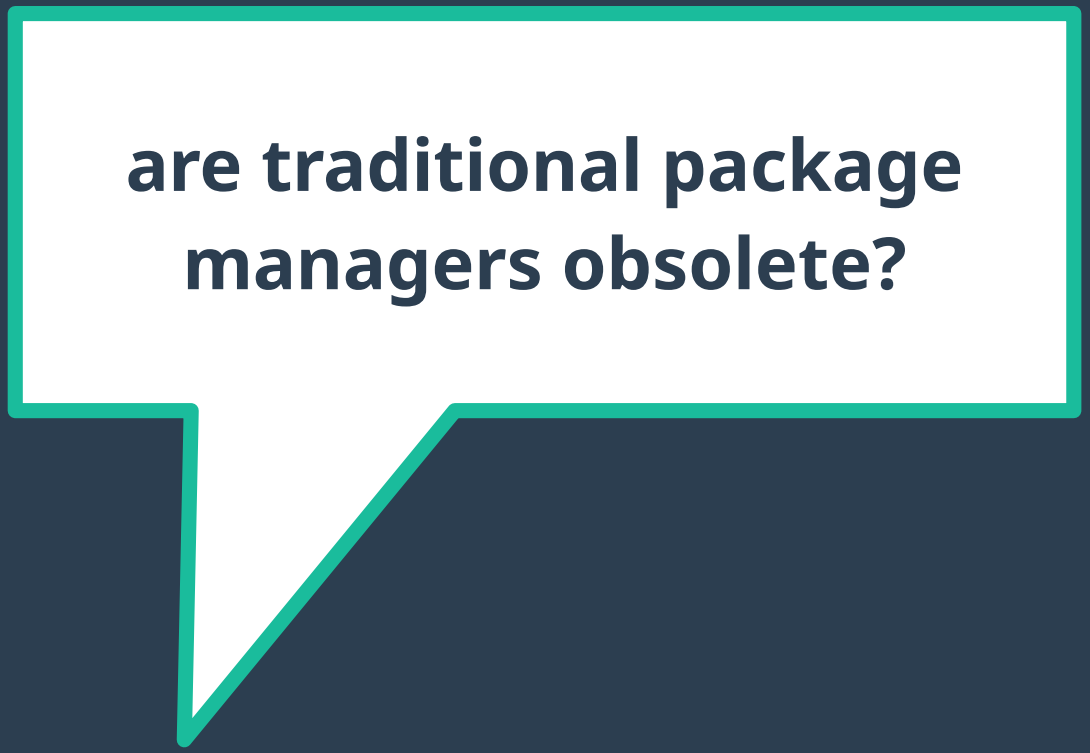
# Distributing Rust in RPMs for fun (relatively speaking) and profit

Fabio Valentini / decathorpe @ FOSDEM 2026

# about @decathorpe

- Fedora packager since 2016
- submitted my first Rust package to Fedora in 2020
- primary maintainer of rust2rpm since 2022
- Fedora Packaging Committee (FPC) member since 2018
- Fedora Engineering Steering Committee (FESCo) member since 2019


so what’s this talk about?


are traditional package managers obsolete?

# background: Rust

- compiled language (rustc / LLVM)
- typically statically linked (no stable Rust ABI)
- compile-time codegen / metaprogramming with macros or build scripts, conditional compilation
- package manager / build system: cargo
- official package registry: crates.io
- package / dependency management: easy (adding lots of dependencies: *also* easy)

# background: RPM packaging in Fedora

- hard requirement to build everything[1] from source
- hard requirement for compliance with license terms[2]
- strong preference to unbundle / not vendor dependencies
- preference for running upstream test suites (if possible)
- incentives to provide good system integration: shell completions, manual pages, etc.

# the approach for libraries

- packages for Rust crates contain only the crate sources and no compiled artifacts (again, no stable ABI)
- even with stable ABI, shipping compiled artifacts for all possible combinations of enabled feature flags is impractical (combinatorial explosion)

# the philosophy for applications

**The goal is to make**

```
$ dnf install foo
```

**provide something as close to**

```
$ cargo install foo
```

**as possible.**

**… or, if possible, to provide something even better.**

so ...
what can be done better?

# application installation: cargo

- cargo only supports “installing” applications by building them from source
- requires users to have a Rust toolchain (cargo, rustc, LLVM) installed
- external dependencies (C library headers) also required in common cases
- “cargo install” only installs executables into the user’s $PATH
- cannot handle other files

# application installation: cargo

- “cargo build” limitations incentivise workarounds that are usually considered unsafe (or at least “bad practice”):

  bundled C libraries, static linking – sometimes even pre-built object files

- “cargo install” limitations require using the “self contained executable” model for applications:

  embedding data files into (huge) executables instead of loading them from disk, no system integration

# application installation: RPM

- integrated with system package manager
- packages are built in a trusted environment and cryptographically signed
- no Rust toolchain or development headers required locally

- packages can ship support files for better system integration:

  shell completions
  manual pages
  config files / data files

# application installation: RPM

- smaller footprint using dynamically linked (C/C++) library dependencies shared between packages

  (deduplication both on disk and in memory)

- disentangles update cycles of C/C++ library dependencies and the application itself

  (allows shipping C/C++ library security updates without having to rebuild Rust applications)

# application updates: cargo

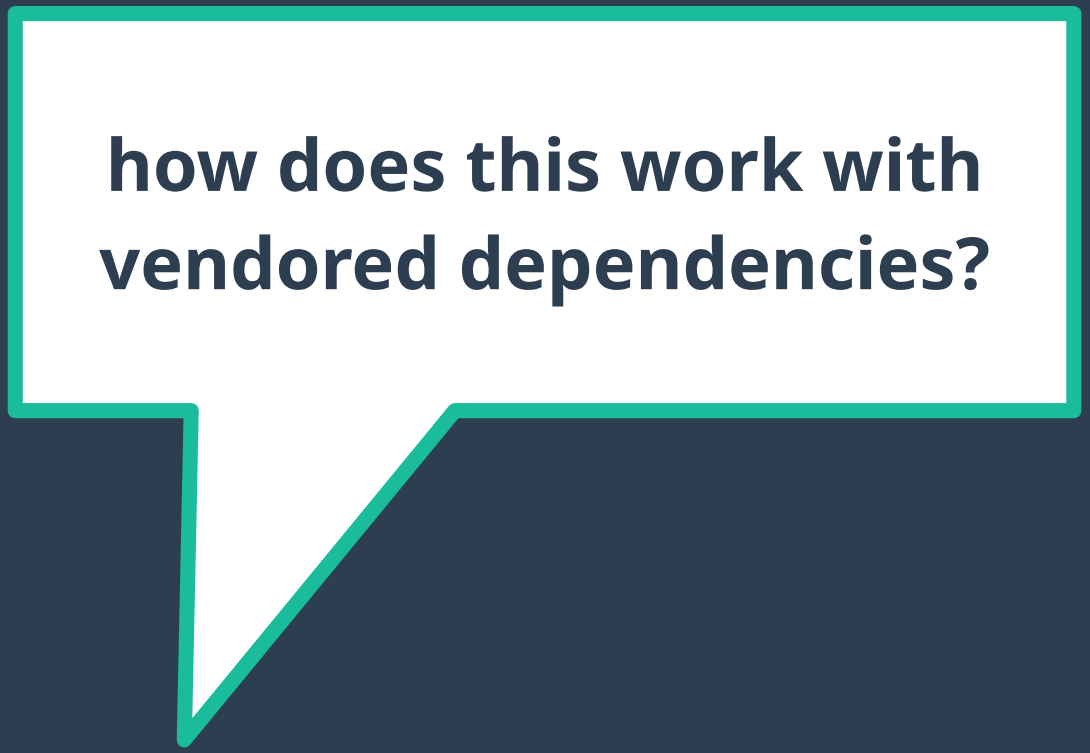
- requires user interaction (running “cargo install” *again* for all installed applications)
- no mechanism to notify users of update availability
- no mechanism to push security updates
- re-running “cargo install” is not enough
- does nothing by default if the *application* version has not changed (requires using “--force”)

# application updates: RPM

- integrated with system package manager
- supports notifying users of available (security) updates
- automatic installation of critical updates on servers or image-based systems
- decouples application and library dependency update cycle (mostly foreshadowing)

# bonus points: better test coverage

- package builds run project test suites (to the extent possible) against an environment that closely matches the user's environment (OS, OS version, library versions, CPU architecture, etc.)
- adds test coverage for environments usually not available in CI (powerpc64le, s390x)
- continuous testing against new dependency versions in package CI (mostly more foreshadowing)


how does this work with vendored dependencies?

# to vendor or not to vendor

**Pros:**

- quickly get to a building and working package
- avoid dealing with dependencies

**Cons:**

- often difficult to get from “working package” to “acceptable package”
- you cannot *actually* avoid dealing with dependencies

# just vendor dependencies (problems included)

- building with vendored dependencies is not a shortcut for avoiding responsibilities
- running test suites for library dependencies is basically not possible[3]

- are project licenses declared correctly?
- do all projects include mandatory license texts?
- are the licenses of all dependencies acceptable[2]?
- are all dependencies legally acceptable[4]?

# doing things the hard way (Fedora, debian)

- one RPM package per Rust crate / library (which only ship source code*)
- shared responsibilities for most[5] dependencies
- avoid duplicated audit effort (legal / technical)
- run test suites (find bugs[6]!)
- providing multiple parallel-installable library versions is easy (if necessary)
- submitting patches with fixes / improvements to upstream improves the ecosystem for all

# making the “right” way as easy as possible

- substantial work has gone into improving the Rust / RPM packager tooling (rust2rpm, etc.)
- busywork basically eliminated from initial packaging + update workflows
- more improvements coming in the future
- currently some features are not enabled by default due to RHEL shipping old RPM versions
- we still need to support EPEL 9 and 10 for … a long time

# recent developments / future improvements

**dynamically generated subpackages (“specparts”):**

- dramatically reduced boilerplate in RPM spec files
- makes re-running rust2rpm for most crate updates unnecessary
- requires RPM 4.19+
- available on Fedora and EPEL 10 (not enabled by default)

**declarative BuildSystem:**

- almost zero boilerplate for common cases
- requires RPM 4.20+
- available on Fedora only (not enabled by default)

so …
*are* traditional package managers obsolete?


Q & A

# [1] exceptions to the “build from source” req

- proprietary firmware blobs
- compiler toolchains (for bootstrapping purposes only)
- generated code does not necessarily need to be regenerated (mostly)

**Fedora Packaging Guidelines:**
**What can be packaged**

# ² license compliance

- many popular FOSS licenses require that (re)distributed sources contain a copy of the original license text

  (including Apache-2.0 and MIT – both popular in the Rust ecosystem)

- some licenses are not acceptable for Fedora

  notably, CC0-1.0 is no longer on the list of licenses that are allowed for “code” (only allowed for “content”)

# [3] running test suites of library dependencies

- the “cargo vendor” command only includes and downloads test-specific dependencies of the toplevel project – but not those of dependencies (or transitive deps)
- even if vendored sources *would* contain those dependencies … dealing with running tests for all libraries would be difficult
- packaged Rust crates avoid this issue entirely

# [4] legal acceptability

- US export restrictions (elliptic curve cryptography)

Fedora Legal Docs:
Elliptic curve cryptography
(allowed curves)

- software covered by patents (mostly related to audio / video codecs – aptX, H.264, H.265 / HEVC, H.266 / VCC, ...)

# [5] shared responsibilities

- most “common” Rust crate dependencies are already packaged for Fedora

  (~3300 Rust packages including packages for different versions of the same crate)

- the Rust crate ecosystem has mostly stabilized in recent years – many projects have significant overlaps between their dependency trees

# 6 finding bugs

- running test suites allows catching issues and regressions early and proactively
- many failures are not only test failures but actual code issues
- we also regularly find bugs in LLVM (usually in backends for less-used platforms)

common failure cases include invalid / platform-specific assumptions:

- endianness bugs
- wrong use of atomics
- page size assumptions