Abusing the RPM Package Manager to Compile Software

iliana destroyer of worlds*
Linux Witch
Amazon Web Services
iweller@amazon.com

ABSTRACT

The RPM Package Manager (RPM) is a popular Linux utility designed to destroy your computer. We extend its destructive capabilities to include dynamically building the list of upstream code sources based on the list of upstream code sources.

ACH Reference Format:
iliana destroyer of worlds. 2019. Abusing the RPM Package Manager to Compile Software. In *Proceedings of SIGBOVIK 2019, Pittsburgh, PA, USA, April 1, 2019 (SIGBOVIK '19)*, 3 pages.

1 INTRODUCTION

The RPM Package Manager (RPM) [4] is an extremely dangerous Linux program that should be avoided at all costs. Effects of use include destruction of data, privileged arbitrary code execution, and in the worst case, installation of a Linux operating system. Many Linux distributions use RPM as their primary package manager; even more don't.

There are two main modes for using RPM: compiling software, usually with the *rpmbuild* command, and installing software, usually with the *rpm -i --force --nodeps*[1] command.

Modern programming languages, such as Perl, have their own package management ecosystems. Within these ecosystems, seemingly innocuous software can depend on multiple incompatible versions of the same dependencies. This is, of course, an impossible issue to solve. It is core to every packager's instinct to add all existing development libraries to a Linux distribution, regardless of how useful those libraries are to an end user, and to build all tools against distribution-packaged libraries.

Firecracker, an advanced chroot developed by AWS, is one such package with many Rust dependencies. Nevertheless, we added Firecracker to Amazon Linux, a Linux distribution maintained by a book store, without building hundreds of useless RPMs. The solution we developed can, but should not, be applied to other programming language ecosystems with intractable dependency closures.

Most importantly, the solution is a complete hack that shouldn't work and should have never been written.

2 RPM: THE GOOD PARTS

Recipes for RPM are called *spec files* (a sample is provided in Listing 1). They begin with general metadata about a package, called the *preamble*, such as the name, mailing address, Social Security number, source tarball, and any patches. This is followed by the *scriptlet* sections, which are Bash scripts; *%prep* unpacks and sews patches onto the source code, *%build* builds it, and *%install* installs it into a build root. Finally, *%files* lists the files installed from the build root, and *%changelog* lists who broke your business-critical software. [1]

Listing 1: A spec file

```
Name: robotfindskitten
Version: 2.7182818.701
Release: 4%{?dist}
Summary: A zen simulation
Source0: http://robotfindskitten.org/.../%{name}-%{version}.tar.gz
Patch0: nki-makefile.patch

BuildRequires: ncurses-devel

%prep
%autosetup -p1

%build
%configure
%make_build

%install
%make_install

%files
%{_bindir}/%{name}

%changelog
* Sun Dec 10 2017 iliana weller <ilianaw@buttslol.net>
- Update to 2.7182818.701 (#1297151)
```

Note the complete lack of any common shell commands which one normally finds in Bash scripts. They're all hidden away behind *macros* to take away any artistic expression that software packaging once had, ensuring that all code is built with exactly the same *configure* and *make* options. [3] To ensure consistency in a distribution, this is coupled with a stringent and continuous review process… wait, spec files often get reviewed once and never again? Oh.

When a user feeds the spec file to *rpmbuild*, it most likely fails spectacularly. Sometimes it doesn't, though, and the user gets RPMs as output. RPMs are kind of like a candy-coated tarball; a hard, metadata-rich exterior breaks open to reveal a gooey *cpio* filling.

2.1 Thousands of Packages You Can't Use

A common rule in Linux distributions is "one project, one source package" [3]. This means that the TEX Live distribution in Fedora is represented by a single 220,000-line spec file generated by a script [5], while each of the 780,000 dependencies available on npm must have a separate spec file.

*Uppercasing of name is punishable by intergalactic law. Views expressed do not necessarily reflect Amazon's.

[1] -i --dont --force --nodeps and neither should you.

Dedicated Python scripts, assisted by obedient packagers, take this rule to its extreme. In Fedora 27 there were 1,129 package names prefixed with *nodejs*, 465 package names prefixed with *golang*, and 132 package names[2] prefixed with *rust*. [5] The scripts can recursively find dependencies, commit spec files, and start builds.

Fedora is not alone; Debian has over 450 Rust source packages; each one builds several binary packages for different feature flags.

These packages commonly install source code to distribution-defined file paths that are ignored by the tools that know how to compile this code. They are dead weight in repository metadata, and adding too many Rust packages incurs the risk of package repository oxidization.

2.2 One Version Ought To Be Enough For Anybody[3]

fd[4] is a Rust utility for finding files. We inspected the Cargo.lock file for fd version 7.3.0[5], which lists its full dependency closure. In order to compile fd, we need two versions of *rand_core* (0.3.x and 0.4.x), as well as two versions of *winapi* (0.2.x and 0.3.x).

In the usual Linux distribution dependency model, this means you need to have two distinct versions of the *winapi* package installed simultaneously, which RPM front-ends[6] did not support until recently, and you need the ability to specify a permitted version range (e.g. $0.3.0 \leq v < 0.4.0$), which RPM did not support until late 2017 (version $4.14 \leq v < \infty$).

This leaves longer-support distributions no option for building Rust code, which is becoming required for common staples of Linux distributions (most notably Firefox, librsvg, and vape[7]).

Well. There is *an* option, but we don't like it.

3 (DEFUN GETSOURCES () (GETSOURCES))

RPM macros are very powerful, incredibly dangerous, and an excellent example of write-only code [2]. The *%autosetup* macro in Listing 1 is responsible for unpacking and patching code. Its definition in Fedora 29 is reproduced in Listing 2.

RPM macros can perform option parsing and introspection of sources and patches defined in the spec file. They can also run arbitrary Bash or Lua code. All of this can be mixed together, and nothing is off-limits; macros can write out entire scriptlet sections, preamble fields, and even Harry Potter slash fiction.

What if we could abuse macros and kill off those thousands of unusable packages? Our macros would need to:

- Introspect a source file for all its Rust dependencies, and add *Source#* lines to the preamble
- Fake a Cargo registry containing all these dependencies
- Build and install the target binary

We built these macros, which contain an unhealthy mixture of JSON parsing, AWK, and Lua. Not only did we build them, we distributed them in an actual Linux distribution, we built Firecracker using them, and we made the code rebuildable by users via retrieving the source RPM and using *rpmbuild --rebuild*.

Listing 2: The %autosetup macro

```
%autosetup(a:b:cDn:TvNS:p:)\
%setup %{-a} %{-b} %{-c} %{-D} %{-n} %{-T} %{!-v:-q}\
%{-S:%global __scm %{-S*}}\
%{expand:%__scm_setup_%{__scm} %{!-v:-q}}\
%{!-N:%autopatch %{-v} %{-p:-p%{-p*}}}
```

For your safety (and because TEX is difficult), the macros[8] are only partially reproduced in Listing 3. At your own risk, they are available on GitHub[9] and in the Amazon Linux source RPMs.

For a packager to download all the required source files, they must run the spec file through a spec file parser (such as *rpmspec -P*), then use a source download tool (such as *spectool -g*).

3.1 Okay, So Not Everything's Perfect

For some dependencies to build correctly, packagers must include the necessary *BuildRequires* and *Patch#* lines to allow them to build. In this system, these must be duplicated across every package that shares the same dependency. This can be resolved by arbitrarily limiting the number of packages that use this solution, perhaps by utilizing the tried-and-true mechanism of "laziness".

4 FURTHER APPLICATIONS

The general approach described here can be used for any programming language with lock files describing their dependency enclosures and with standard URLs to fetch source code, such as Node.js or what people wish Go would be.

We *really* shouldn't though.

5 FURTHER TANGENTIALLY RELATED RESEARCH TOPICS

Calculate the total bandwidth used to transfer the *%changelog* section of the TEX Live spec file, duplicated in each of its nearly 6,000 binary RPMs [5], across Fedora and its derivative distributions.

Something something Nix Docker Flatpak AppImage snaps.

ACKNOWLEDGMENTS

The author acknowledges Amazon for inexplicably employing her.

Thanks to Natasha Jarus, Samuel Karp, Euan Kemp, Tom Kirchner, and Colleen Quine for their reviews.

REFERENCES

[1] Edward C. Bailey. 1997. *Maximum RPM.* Sams Publishing. http://ftp.rpm.org/max-rpm/

[2] Wikipedia contributors. 2019. Write-only language. *Wikipedia* (2019). https://en.wikipedia.org/w/index.php?title=Write-only_language&oldid=880031540

[3] Tom 'spot' Callaway et al. [n. d.]. Fedora Packaging Guidelines. https://docs.fedoraproject.org/en-US/packaging-guidelines/

[4] Erik Troan, Marc Ewing, et al. 1995. RPM Package Manager. http://rpm.org

[5] Will Woods. 2018. Unpacking RPM: package names. https://weldr.io/Unpacking-RPM-names/

[2] 132 is a lot for the first Fedora release where Rust packages were permitted at all; the current count is over 500.

[3] With apologies to Bill Gates, noted proponent of Linux.

[4] fd stands for Finger Donuts.

[5] https://github.com/sharkdp/fd/blob/7f58e8f7064346ffe569563b657fe92f24830e6a/Cargo.lock

[6] Common RPM front-ends include Zypper, yum, and DNF (which stands for Do Not Fuck).

[7] https://github.com/JoshuaRLi/vape

[8] Object Class: Euclid

[9] https://github.com/awslabs/rust-bundled-packaging/tree/b9c50d16b6d517c4e7483c6842b6f3cc77969b9d

Listing 3: RPM macros for bundling Rust dependencies

```
# SPDX-License-Identifier: MIT

# Copyright (c) 2017 Igor Gnatenko
# Copyright 2018 Amazon.com, Inc. or its affiliates.

%__cargo %{_bindir}/cargo
%__cargo_common_opts %{?_smp_mflags}

%_cargometadir %{_datadir}/cargo-metadata

%cargo_prep (\
set -eu \
%{__mkdir} -p .registry \
REGISTRY="$(realpath .registry)" \
%{__mkdir} -p .cargo \
cat > .cargo/config << EOF \
[build]\
rustc = "%{__rustc}"\
rustdoc = "%{__rustdoc}"\
rustflags = %{__global_rustflags_toml}\
\
[term]\
verbose = true\
\
[source]\
\
[source.local-registry]\
directory = "$REGISTRY"\
\
[source.crates-io]\
registry = "https://crates.io"\
replace-with = "local-registry"\
EOF\
do_cargo_build_registry() { \
%__cargo_build_registry \
} \
do_cargo_build_registry $REGISTRY \
)

%cargo_build %{__cargo} build %{__cargo_common_opts} --release %{?cargo_args}

%cargo_test %{__cargo} test %{__cargo_common_opts} --release --no-fail-fast %{?cargo_args}

%cargo_install \
%{__cargo} install %{__cargo_common_opts} --path . --root %{buildroot}%{_prefix} %{?cargo_args} \
%{__rm} %{buildroot}%{_prefix}/.crates.toml \
%{__mkdir_p} %{buildroot}%{_cargometadir} \
%{__cargo} metadata --format-version 1 %{?cargo_args} > %{buildroot}%{_cargometadir}/%{name}.json

%__cargo_crate_source_url() https://crates.io/api/v1/crates/%1/%2/download#/%1-%2.crate
%__cargo_crate_source_urls grep '^"checksum' | awk '{ print "Source" NR+9999 ": %%{__cargo_crate_source_url " $2 " " $3 "}" } END { print "%%global __cargo_first_crate 10

%__cargo_build_registry %{lua:for i = rpm.expand("%__cargo_first_crate"),rpm.expand("%__cargo_last_crate") do \
  uncompress = rpm.expand("%{uncompress:%{S:" .. i .. "}}") \
  print(uncompress .. " | tar -x -C $1\\n") \
  template = '{"files":{},"package":"%s"}' \
  print("printf '" .. template .. "' $(sha256sum " .. rpm.expand("%{S:" .. i .. "}") .. " | awk '{ print $1 }') > $1/$(" .. uncompress .. " | tar -t | head -n 1 | cut -d / -f 1)/.cargo-chec
end}

%cargo_bundle_crates(n:t:l:) \
%{-t:%{-l:%{error:cargo_bundle_crates: Can't specify both -t and -l}}} \
%{!-t:%{!-l:%{error:cargo_bundle_crates: Must specify one of -t or -l}}} \
%{-t:%{expand:%(%{uncompress:%{S:%{-t*}}} | tar -xO %{-n:%{-n*}}%{!-n:%{name}-%{version}}/Cargo.lock | %__cargo_crate_source_urls)}} \
```