PACKAGING FOR BEGINNERS

Carl George
CPE EPEL Team Lead

carl@redhat.com
@carlwgeorge

Troy Dawson
EPEL Steering Committee Chair

tdawson@redhat.com

Nils Philippsen
CPE Engineer

nils@redhat.com

SHAMELESS PLUG FOR THE EPEL SURVEY

In case you missed it during the introductory remarks, EPEL is doing a survey and we'd love your feedback.

Survey link:
https://fedoraproject.limequery.com/396386
or
https://tinyurl.com/epelsurvey2022

carl@redhat.com

TOPICS FOR TODAY

General topics

- What is source code?
- How programs are made.
- Building software from source.
- Patching software.
- Installing arbitrary artifacts.

RPM packaging

- What is an RPM?
- What is a spec file?
- Buildroots
- RPM macros
- Building RPMs
- Quality checking RPMs

ABOUT THIS GUIDE

Our workshop today will roughly follow the Red Hat Developer RPM Packaging Guide.

- The guide will be a resource today and a reference later.
- We will **not** cover the whole thing in this session.
- Appendix is full of advanced topics we will not have time for.
- It is a living document.
- Pull requests are welcome and encouraged.

https://github.com/redhat-developer/rpm-packaging-guide

ABOUT THIS GUIDE CONT.

Packaging guide online:
https://rpm-packaging-guide.github.io/rpm-packaging-guide.pdf
or
https://tinyurl.com/nestrpm

PREREQUISITES

- You will need access to a Fedora, CentOS, RHEL, or RHEL derivative (Alma, Rocky, Oracle, etc.) to perform the labs in today’s workshop.
- This can be the machine you’re using or a remote system you have SSH access to.

LAB - PREREQUISITES (PAGE 3)

Run the specified dnf/yum command on this page to install the packages you will need to complete the workshop.

If anyone is running an EL8 system, see the adjustments needed here: https://github.com/redhat-developer/rpm-packaging-guide/pull/90

Time: <5 minutes

GENERAL TOPICS AND BACKGROUND

WHAT IS SOURCE CODE? (PAGE 7)

- Human friendly representation of instructions for the computer.
- The Bash shell is an interactive shell which happens to be “scriptable” (as most shells are). Its scripting language is in fact a programming language and therefore its instructions to the computer can be considered source code.

HOW PROGRAMS ARE MADE (PAGE 8)

- Compilation
  - The process by which source code is translated into a representation the computer understands, native computer language or otherwise.
- Types of execution
  - Natively compiled
  - Interpreted
    - Byte compiled
    - Raw interpreted

TYPES OF BUILDS (PAGE 8)

- Native
  - Translated (compiled) directly to machine code
  - Can execute directly on the system
- Interpreted (byte compiled)
  - Translated into an optimized intermediate representation known as byte code
  - Needs an interpreter to execute
- Interpreted (raw)
  - Interpreted and executed directly by its runtime as the source code is parsed
  - Needs an interpreter to execute

BUILDING SOFTWARE (PAGES 9-12)

- Building
  - Software compilation is often referred to as “building”.
  - “Build systems” or “build tools” such as make or meson automate this process.
- Natively compiled source code
  - Must be “built” in order to execute as it doesn’t have an interpreter to execute it otherwise.
  - Hardware architecture specific.
- Interpreted source code
  - If byte-compiled, byte code must must be “built”.
  - Some byte-compiled languages do this automatically (Python, Ruby) and others must be built by hand (Java).

PATCHING SOFTWARE (PAGES 12-14)

- A software patch is much like a cloth patch used in repair of a shirt, a blanket, a pair of pants, etc.
- It’s meant to either repair a defect (bug) found in the software or add new functionality that was previously missing.
- This is important for RPM packagers because we will often find ourselves needing to fix something or add functionality before the next upstream version.
- Original source code remains pristine for auditability, reproducibility, and debugging purposes.

INSTALLING ARTIFACTS (PAGES 14-16)

- Installation on Linux systems
  - Placing file in the “correct” place.
- Filesystem Hierarchy Standard (FHS)
  - Default directory structure.
  - Defines context for arbitrary files based on location (`/etc`, `/usr/bin`, `/usr/share`, and so on).
- `install` command
  - Part of GNU coreutils.
  - Copies files into their destination.
  - Handles modes, ownership, etc.

RPM PACKAGING GUIDE

WHAT IS AN RPM PACKAGE? (PAGE 20)

- File containing other files and metadata about them.
- More specifically
    - Lead 96 bytes of “magic” (no longer used, retained for backwards compatibility).
    - Digital signature.
    - RPM header containing the metadata.
    - CPIO archive containing the payload (the actual files to be installed on the target system).

WORKSPACE SETUP (PAGE 21)

- The rpmdevtools package includes the `rpmdev-setuptree` command.
- Running that command on a system will create the following directories.
  - `~/rpmbuild/BUILD`
  - `~/rpmbuild/RPMS`
  - `~/rpmbuild/SOURCES`
  - `~/rpmbuild/SPECS`
  - `~/rpmbuild/SRPMS`

LAB - WORKSPACE SETUP (PAGE 21)

Run the `rpmdev-setuptree` command on your system to set up the directory structure we will be using for the rest of the labs.

Time: <5 minutes

SPEC FILES

WHAT IS A SPEC FILE? (PAGE 21)

- Recipe or set of instructions to tell rpmbuild how to actually build an RPM.
- Composed of various sections and headings.
  - Metadata
  - Build instructions
  - File manifest
- Where we define the name, version, and release (NVR)
  - This is used to compare packages to determine which available packages are upgrades for installed packages.
  - Example: `bash-5.1.16-2.fc36`

SPEC FILE PREAMBLE (PAGE 22)

- `Name` - name of the software being packaged
- `Version` - upstream version of the software
- `Release` - release of the package
- `Summary` - short summary of what the package contains
- `License` - software license of the software being packaged
- `URL` - software or software vendor's website
- `Source` - path or URL for software source code archive or other files to be included in the package
- `Patch` - file name of patch files to apply to the software

SPEC FILE PREAMBLE CONT. (PAGE 22)

- `BuildArch` - used to declare a package as architecture independent (noarch)
- `BuildRequires` - packages that must be installed on the system building the package
- `Requires` - packages that must be installed on the system installing the package
- `ExcludeArch` - architectures this package explicitly does not support
- `ExclusiveArch` - architectures this package only supports

SPEC FILE BODY (PAGE 23)

- `%description` - full description of the software
- `%prep` - commands to prepare the source code for being built (unpacking archives, applying patches, etc.)
- `%build` - commands for actually building the software into machine code (compiled languages) or byte code (for byte-compiled interpreted languages)
- `%install` - commands to install the built files into appropriate filesystems locations relative to the %buildroot directory
- `%check` - commands to test the software, e.g. run unit tests
- `%files` - list of files that will be installed on the target system
- `%changelog` - record of changes that have happened to the package between different versions/releases

RPM MACROS (PAGE 24)

- Variable for text substitution.
- Can be conditional, meaning only expand the macro if some condition is true.
- Can be explored outside of an RPM build.
  - `rpm --eval` to evaluate a specific macro
  - `rpm --define` to define a macro to influence other macros being evaluated
  - `rpm --showrc` to print all defined macros

COMMON MACROS (PAGES 24-25)

- Filesystem locations
  - `%{_bindir}` → `/usr/bin`
  - `%{_libexecdir}` → `/usr/libexec`
- Distribution properties
  - `%{centos}` → `9`
  - `%{el9}` → `1`
  - `%{dist}` → `.el9`

WORKING WITH SPEC FILES (PAGE 25)

- A big part of packaging software into RPMs is editing spec files.
- Most packagers don’t create spec files completely from scratch.
  - Use built in templates from their text editor
  - Use `rpmdev-newspec`, which creates a spec file with the basic structure (preamble and body sections) that is then adjusted for the software being package.

LAB - WORKING WITH SPEC FILES (PAGE 25)

Download the tarballs and patch files mentioned on this page. Place them in the `~/rpmbuild/SOURCES` directory.

We will be working with three example “hello world” programs today. Create a new spec file for each of them using `rpmdev-newspec` as detailed on this page.

Time: 5 minutes

LAB - BELLO SPEC FILE (PAGES 26-31)

In this lab we will write the spec file for the bello program. It is an example “hello world” program written in Bash.

Time: 15 minutes

LAB - PELLO SPEC FILE (PAGES 31-38)

In this lab we will write the spec file for the pello program. It is an example “hello world” program written in Python.

Time: 15 minutes

LAB - CELLO SPEC FILE (PAGES 39-44)

In this lab we will write the spec file for the cello program. It is an example “hello world” program written in C.

Time: 15 minutes

BUILDING RPMS (PAGES 44-47)

- Up until now we’ve been preparing ourselves for `rpmbuild`.
- We’ve covered:
  - How software is built from source code.
  - How arbitrary artifacts built from source code are installed.
  - Preparing our RPM build environment.
  - How to instruct `rpmbuild` what to do (the spec file).
- We will use `rpmbuild` to build source RPMs (SRPMs) as well as binary RPMs.
- We will also explore some aspects of `rpmbuild` that can be surprising.

LAB - BUILDING RPMS (PAGES 44-47)

In this lab we will build source RPMs and binary RPMs for the bello, pello, and cello programs.

Make sure to only run the rpmbuild command as a non-root user. Errors in a spec file can have negative effects on the system that is performing the build.

Time: 15 minutes

QUALITY CHECKING RPMS (PAGES 47-51)

- `rpmlint` is a linter tool for spec files, SRPMs, and RPMs.
- Can report common packaging errors.
- Fedora 35+ has `rpmlint` version 2, whose output will not match the examples in the guide.

LAB - QUALITY CHECKING RPMS (PAGES 47-51)

In this lab we will check our spec files, SRPMs, and RPMs for quality using `rpmlint`.

Time: 15 minutes

ADVANCED TOPICS (PAGES 52-74)

- Resources for your packaging adventures after this workshop.
- Of particular note, we recommend reading up about:
  - mock
  - dist-git
  - Defining your own macros
  - Epochs
  - Scriptlets
  - Conditionals

MOCK (PAGES 55-59)

- Drawbacks of using rpmbuild directly:
    - Build requirements must be installed on the system running rpmbuild.
    - A build requirement that is already installed is easy to forget to list in the spec file.
    - Can only build RPMs targeting the same operating system and release.
- mock is a tool that builds packages in isolated chroots.
    - Build requirements are installed in chroot, not system.
    - Can build RPMs for different operating systems and releases than your system.
    - Chroots are automatically created and removed.

THAT’S ALL FOLKS!