RED HAT SUMMIT

FROM SOURCE TO RPM IN 120 MINUTES

Adam Miller
Senior Software Engineer

2016-06-30

#redhat #rhsummit

WHY WE'RE HERE

TOPICS FOR TODAY

GENERAL TOPICS AND BACKGROUND

- What is source code?
- How programs are made.
- Building Software from Source
- Patching Software
- Installing Arbitrary Artifacts

RPM PACKAGING

- What is a RPM?
- What is a SPEC file?
- Buildroots
- RPM Macros
- Building RPMs
- Checking RPMs for Sanity
- Advanced Topics
    - Plenty of reference materials in the Appendix

ABOUT THIS LAB

LAB MANUAL

MEANT TO BE COMPREHENSIVE

- Meant to be a resource today and as 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
- Is living document, Upstream Project information on last page of manual.

ABOUT THIS LAB

PREREQUISITES - **PAGE 2**

PLEASE TURN TO PAGE 2 IN YOUR LAB MANUAL

- Here you will find the Prerequisites required, please run the command on your Student system

```
yum install gcc rpmbuild rpm-devel rpmlint make python bash coreutils diffutils patch
```

- Note that all prerequisites needed for this Lab are included as part of Red Hat Enterprise Linux 7 Server, meaning there is no need to enable any extra Subscription Channels to complete the exercises in this Lab.
    - **Note:** Extra resources are required for some of the advanced topics in the Appendix

GENERAL TOPICS AND BACKGROUND

WHAT IS SOURCE CODE?

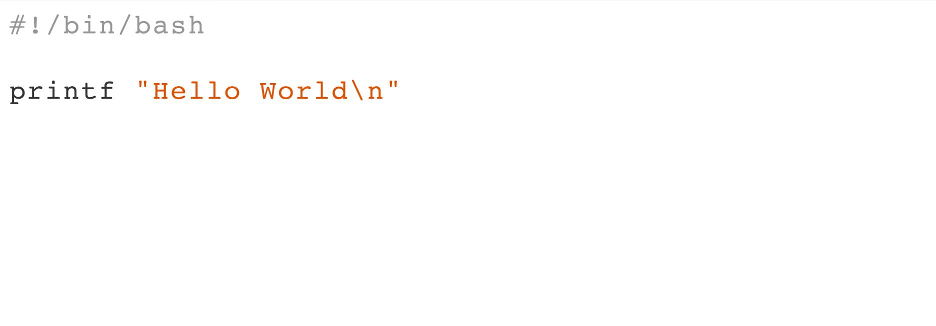
BRIEF OVERVIEW - **PAGE 3**

SOURCE CODE

Human friendly representation of instructions for the computer. Source code can be referred to as "program code" or a "script" depending on programming language or execution environment.

BASH SHELL

The Bash Shell is an interactive UNIX shell which happens to be "scriptable" (as most are), it's scripting language is in fact a programming language and therefore it's instructions to the computer can be considered Source Code



```
#!/bin/bash

printf "Hello World\n"
```

HOW PROGRAMS ARE MADE

UNDERSTANDING BUILDS - **PAGE 4**

COMPILATION

The process by which source code is translated into a representation the computer understands, native computer language or otherwise.

TYPES OF EXECUTION

Three types of execution. Two main categories.

- Natively Compiled
- Interpreted
  - Byte Compiled
  - Raw Interpreted

HOW PROGRAMS ARE MADE

UNDERSTANDING BUILDS - **PAGE 4**

NATIVE

- Translated (compiled) directly to machine code
- Can execute directly on the system
    - Examples: C/C++, Go, Objective-C, Fortran, COBOL, Vala

INTERPRETED (BYTE COMPILED)

- Translated into an optimized intermediate representation (byte-compiled)
- Needs an interpreter to execute ("wrapper" scripts are common)
    - Examples: Java, Python, Ruby, Node.js/JavaScript, Tcl, Lua, Perl

INTERPRETED (RAW)

- Interpreted and executed directly by it's runtime as the source code is parsed
- Needs an interpreter to execute
    - Examples: bash, zsh, batch

BUILDING SOFTWARE

(FROM SOURCE) - PAGE 6

BUILDING

- Software compilation is often referred to as "building"
- "build system" or "build tool" will often refer to things like GNU Make to automate this

NATIVE COMPILED SOURCE CODE

- Natively compiled code must be "built" in order to execute as it doesn't have an interpreter to execute it otherwise
- Hardware Architecture specific
  - Can not build on x86_64 and run on POWER, s390x, or AARCH64

INTERPRETED SOURCE CODE

- Source code written in interpreted programming languages that are byte-compiled must be "built" also
  - Some languages do this automatically for you (Python, Ruby, Node.js) others must be built by hand (Java).

PATCHING SOFTWARE

FIXING SOFTWARE - **PAGE 9**

PATCH

- A software patch is much like a cloth patch used in repair of a shirt, a blanket, a pair of pants or otherwise.
- Is meant to either repair a defect (bug) found in the software or add new functionality that was previously missing

WHY?

- This is important for RPM Packagers because we will often find ourselves needing to fix something or add functionality before the next major revision
- Original source code should remain pristine
    - Build audit chains
    - Reproducability
    - Debugging

INSTALLING ARTIFACTS

PLACING FILES - **PAGE 12**

INSTALLATION ON LINUX SYSTEMS

- Placing files in the "correct" place.
  - Everything is a file.

FILESYSTEM HIERARCHY STANDARD (FHS)

- Default directory structure
- Define context for arbitrary files based on location

INSTALL COMMAND

- Part of GNU Coreutils
- Copies files into their destination location
  - Also handles permissions/modes, owner, groups

```
install -m 0755 bello /usr/bin/bello
```

Filesystem Hierarchy of RHEL7

```
/
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

RPM PACKAGING GUIDE

#redhat #rhsummit

WHAT IS A RPM?

DEMYSTIFYING - **PAGE 19**

RPM PACKAGE

- File containing other files and metadata about them
- More specifically
    - Lead (96 Bytes of "magic")
        - No longer used, maintained for backwards compat
    - Signature - digital signatures
    - RPM Header - metadata
    - CPIO Archive - Payload

WORKSPACE SETUP

PREPPING FOR BUILD - **PAGE 20**

SETTING UP OUR PACKAGING ENVIRONMENT

- rpmdevtools
  - rpmdev-setuptree

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/maxamillion/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

LAB TIME!

SETUP YOUR RPM WORKSPACE

PAGE 20

TIME: 5 MINUTES

SSH Into your Student System and run the commands on Page 20 to familiarize yourself with setting up a RPM Workspace.

NOTE: Not as root or with sudo

WHAT IS A SPEC FILE?

THE RPM RECIPE - **PAGE 21**

THE SPEC FILE

- Recipe or set of instructions to tell rpmbuild how to actually build a RPM
- Composed of various sections and headings
  - populate metadata
  - build instructions
  - file manifest
- Where we define the Name-Version-Release (N-V-R)
  - This is used in RPM version comparison transactions as well as for yum installations

```
$ rpm -q python
python-2.7.5-34.el7.x86_64
```

WHAT IS A SPEC FILE?

CONTINUED - **PAGE 21**

PREAMBLE

- **Name** - name of the software being packaged
- **Version** - version of the software being packaged
- **Release** - release number of the package
- **Summary** - short summary of what software the package contains
- **License** - software license of the software being packaged
- **URL** - sofware or software vendor's website
- **Source0** - URL to where the software can be downloaded from
    - Can be multiple SourceX entries. Source1, Source2, Source3, etc.
- **Patch0** - File listing of a patch found in ~/rpmbuild/SOURCES/
    - Can be multiple PatchX entries. Patch1, Patch2, Patch3, etc.
- **BuildArch** - Architectures supported by this package (natively compiled code)
- **BuildRequires** - Packages required to be installed on build host to perform build
- **Requires** - Packages required to be installed on target host to actually run the software
- **ExcludeArch** - Architectures this package explicitly does not support (natively compiled code)

WHAT IS A SPEC FILE?

CONTINUED - **PAGE 22**

BODY

- **%description** - Long hand description of the software, can be many paragraphs.
- **%prep** - Command or series of commands to prepare the software for being built.
  - This is where you will unarchive/uncompress source code, etc.
- **%build** - Command or series of commands to build the software
- **%install** - Command or series of commands to install the software
  - Software is installed here in the context of the RPM BUILDROOT
- **%check** - Command or series of commands to run tests on the software
- **%files** - File manifest with metadata and default permissions attributes
- **%changelog** - Changelog for this package
  - Things like CVE fix listings and bug fix patches are normally listed here or information about a change to the SPEC file itself.

RPM MACROS

A LITTLE MAGIC - **PAGE 23**

MACROS

- Straight text substitution of a variable name
  - Can be conditionally called upon, meaning only expand this macro if some condition is true
  - Can be explored, evaluated before time
    - 'rpm --eval' - to evaluate a specific macro
    - 'rpm --showrc' - to see what macros are defined on the build host
      - A lot of output, normally used with 'grep' to search for something specific

COMMON MACROS

- Filesystem locations
  - %{_bindir} -> /usr/bin
  - %{_libexecdir} -> /usr/libexec
- Dist tag
  - %{?dist} - conditionally expanded if it exists in the context of our rpmbuild (?)

WORKING WITH SPEC FILES

GETTING STARTED- **PAGE 24**

CREATING SPEC FILES FROM SCRATCH

- Most RPM Packagers don't create SPEC files completely from scratch
  - Use built-in template tooling in their editor (vim/emacs/etc)
  - Use rpmdev-newspec
- We will be using rpmdev-newspec
  - Creates a template with some fields pre-populated and we can just fill it in
  - Template can be altered based on command line options passed

LAB TIME!

DOWNLOAD SOURCE

CREATE SPEC FILES

PAGE 24/25

TIME: 5 MINUTES

On Student System: download source files to ~/rpmbuild/SOURCES/

http://classroom.example.com/rpm

Then run commands on Page 25

NOTE: Not as root or with sudo

BELLO

FIRST RPM SPEC FILE- **PAGE 27**

EXERCISE TO PACKAGE SOFTWARE

- Example software written in bash, simple "Hello world" program
  - Note: This is a raw interpreted programming language and therefore doesn't need to be built.
    - A similar method could be used for arbitrary binaries as Source0 (not recommended but sometimes necessary)
- We will in this section of the lab create and modify the SPEC file for bello

LAB TIME!

BELLO SPEC FILE

PAGE 27-32

TIME: 15 MINUTES

Use Student system to perform exercise starting on Page 27 and ending on Page 32

NOTE: Not as root or with sudo

PELLO

SECOND RPM SPEC FILE- **PAGE 33**

EXERCISE TO PACKAGE SOFTWARE

- Example software written in Python, simple "Hello world" program
    - Note: This is a byte-compiled interpreted programming language and therefore does need to be built.
        - We will be using a simple example of how to do this, more sophisticated methods exist in the wild.
        - Also will be using a wrapper script (as discussed previously is common)
- We will in this section of the lab create and modify the SPEC file for pello

LAB TIME!

PELLO SPEC FILE

PAGE 33-40

TIME: 15 MINUTES

Use Student system to perform exercise starting on Page 33 and ending on Page 40

NOTE: Not as root or with sudo

CELLO

THIRD RPM SPEC FILE- **PAGE 41**

EXERCISE TO PACKAGE SOFTWARE

- Example software written in C, simple "Hello world" program
  - Note: This is a native compiled programming language and therefore does need to be built.
    - We will be using GNU Make
      - This is one of the most popular build automation tools in the world and you will almost certainly run into it as a RPM Packager
- We will in this section of the lab create and modify the SPEC file for pello

LAB TIME!

CELLO SPEC FILE

PAGE 41-46

TIME: 15 MINUTES

Use Student system to perform exercise starting on Page 41 and ending on Page 46

NOTE: Not as root or with sudo

BUILDING RPMS

RPMBUILD - **PAGE 47**

ACTUALLY PRODUCING RPMS

- Up until now we've been prepping ourselves for a rpmbuild
    - We learned what source code was
    - How software is built from source code
    - How arbitrary artifacts such as those built from source code are installed
    - Prepping our RPM build environment
    - How to instruct rpmbuild what to do (create a SPEC file)
- We will use rpmbuild to build Source RPMs (SRPMS) as well as Binary RPMs
- Explore some aspects of rpmbuild that can be surprising
- **Note:** rpmbuild should never be executed as root, if something is wrong in the SPEC file it could have negative affects on the system that is performing the build.

LAB TIME!

BUILDING RPMS

PAGE 47-51

TIME: 15 MINUTES

Use Student system to perform exercise starting on Page 47 and ending on Page 51

NOTE: Not as root or with sudo

CHECKING RPM SANITY

LINTING - **PAGE 52**

VERIFYING RPMS POST-BUILD FOR QUALITY

- rpmlint
  - Linter tool for RPMs and SPEC files
  - Checks common packaging errors
- We will use rpmlin to check the sanity of the SPEC files, RPMs, and SRPMs we have just created
  - **THERE WILL BE FAILURES AND WARNINGS**
  - Explore some reasons there are failures and warnings
    - Understanding these is a great tool in RPM Packaging
    - rpmlint can provide us will plenty of information about the errors and warnings

LAB TIME!

CHECKING RPM SANITY

PAGE 52-56

TIME: 15 MINUTES

Use Student system to perform exercise starting on Page 52 and ending on Page 56

NOTE: Not as root or with sudo

APPENDIX

ADVANCED RPM TOPICS - **PAGE 57**

RESOURCES FOR BEYOND THE LAB

The Appendix has been written in such a way that it will supplement what you have learned here today. The Lab Manual is meant to be a reference as is the Appendix.

TOPICS

- mock - pristine cross-distro and cross-release buildroots
- Version Control Systems - Following DevOps style workflows while building RPMs
- More on Macros
  - Defining Macros, %files directive, Buit-ins, Distribution specific, and more.
- Advanced SPEC Topics
  - Epoch - the final straw in versioning
  - Triggers and Scriptlets - modifying RPM transaction behavior
- References

THANK YOU!

RED HAT SUMMIT

LEARN. NETWORK.
EXPERIENCE OPEN SOURCE.

Adam Miller

maxamillion - Fedora

maxamillion - GitHu

@TheMaxamilli

#redhat #rhsummit

redhat.