Software Release Practice HOWTO

Eric S. Raymond <esr@thyrsus.com> v2.0, 18 September 1999

This HOWTO describes good release practices for Linux open-source projects. By following these practices, you will make it as easy as possible for users to build your code and use it, and for other developers to understand your code and cooperate with you to improve it. This document is a must-read for novice developers. Experienced developers should review it when they are about to release a new project. It will be revised periodically to reflect the evolution of good-practice standards.

Contents

1 Introduction **2**

1.1 Why this document? . . . 2

1.2 New versions of this document . . . 2

2 Good project- and archive- naming practice **2**

2.1 Use GNU-style names with a stem and major.minor.patch numbering. . . . 2

2.2 But respect local conventions where appropriate . . . 4

2.3 Try hard to choose a name prefix that is unique and easy to type . . . 4

3 Good licensing and copyright practice: the theory **4**

3.1 Open source and copyrights . . . 4

3.2 What qualifies as open source . . . 5

4 Good licensing and copyright practice: the practice **5**

4.1 Make yourself or the FSF as the copright holder . . . 5

4.2 Use a license conformant to the Open Source Definition . . . 6

4.3 Don't write your own license if you can possibly avoid it. . . . 6

5 Good development practice **6**

5.1 Write either pure ANSI C or a portable scripting language . . . 6

5.2 Follow good C portability practices . . . 6

5.3 Use autoconf/automake/autoheader . . . 7

5.4 Sanity-check your code before release . . . 7

6 Good distribution-making practice **7**

6.1 Make sure tarballs always unpack into a single new directory . . . 7

6.2 Have a README . . . 8

6.3 Respect and follow standard file naming practices . . . 8

6.4 Provide RPMs . . . 9

7 Good communication practice **9**

7.1 Announce to c.o.l.a 9

7.2 Announce to a relevant topic newsgroup 9

7.3 Have a website 9

7.4 Host project mailing lists 10

7.5 Release to major archives 10

8 Good project-management practice **10**

1 Introduction

1.1 Why this document?

There is a large body of good-practice traditions for open-source code that helps other people port, use, and cooperate with developing it. Some of these conventions are traditional in the Unix world and predate Linux; others have developed recently in response to particular new tools and technologies such as the World Wide Web.

This document will help you learn good practice. It is organized into topic sections, each containing a series of checklist items. Think of these as a pre-flight checklist for your distribution.

1.2 New versions of this document

This document will be posted monthly to the newsgroups `comp.os.linux.answers` . The document is archived on a number of Linux FTP sites, including `metalab.unc.edu` in `pub/Linux/docs/HOWTO`.

You can also view the latest version of this HOWTO on the World Wide Web via the URL `<http://metalab.unc.edu/LDP/HOWTO/Software-Release-Practice.html>`.

Feel free to mail any questions or comments about this HOWTO to Eric S. Raymond, *esr@snark.thyrsus.com* `<mailto:esr@snark.thyrsus.com>`.

2 Good project- and archive- naming practice

As the load on maintainers of archives like Metalab, the PSA site and CPAN increases, there is an increasing trend for submissions to be processed partly or wholly by programs (rather than entirely by a human).

This makes it more important for project and archive-file names to fit regular patterns that computer programs can parse and understand.

2.1 Use GNU-style names with a stem and major.minor.patch numbering.

It's helpful to everybody if your archive files all have GNU-like names – all-lower-case alphanumeric stem prefix, followed by a dash, followed by a version number, extension, and other suffixes.

Let's suppose you have a project you call 'foobar' at version 1, release 2, level 3. If it's got just one archive part (presumably the sources), here's what its names should look

foobar-1.2.3.tar.gz

The source archive

foobar.lsm

The LSM file (assuming you're submitting to Metalab).

Please *don't* use these:

foobar123.tar.gz

This looks to many programs like an archive for a project called'foobar123' with no version number.

foobar1.2.3.tar.gz

This looks to many programs like an archive for a project called 'foobar1' at version 2.3.

foobar-v1.2.3.tar.gz

Many programs think this goes with a project called 'foobar-v1'.

foo_bar-1.2.3.tar.gz

The underscore is hard for people to speak, type, and remember

FooBar-1.2.3.tar.gz

Unless you *like* looking like a marketing weenie. This is also hard for people to speak, type, and remember.

If you have to differentiate between source and binary archives, or between different kinds of binary, or express some kind of build option in the file name, please treat that as a file extension to go *after* the version number. That is, please do this:

foobar-1.2.3.src.tar.gz

sources

foobar-1.2.3.bin.tar.gz

binaries, type not specified

foobar-1.2.3.bin.ELF.tar.gz

ELF binaries

foobar-1.2.3.bin.ELF.static.tar.gz

ELF binaries statically linked

foobar-1.2.3.bin.SPARC.tar.gz

SPARC binaries

Please *don't* use names like 'foobar-ELF-1.2.3.tar.gz', because programs have a hard time telling type infixes (like '-ELF') from the stem.

A good general form of name has these parts in order:

1. project prefix
2. dash

3. version number
4. dot
5. "src" or "bin" (optional)
6. dot or dash (dot preferred)
7. binary type and options (optional)
8. archiving and compression extensions

2.2 But respect local conventions where appropriate

Some projects and communities have well-defined conventions for names and version numbers that aren't necessarily compatible with the above advice. For instance, Apache modules are generally named like mod_foo, and have both their own version number and the version of Apache with which they work. Likewise, Perl modules have version numbers that can be treated as floating point numbers (e.g., you might see 1.303 rather than 1.3.3), and the distributions are generally named Foo-Bar-1.303.tar.gz for version 1.303 of module Foo::Bar.

Look for and respect the conventions of specialized communities and developers; for general use, follow the above guidelines.

2.3 Try hard to choose a name prefix that is unique and easy to type

The stem prefix should be common to all a project's files, and it should be easy to read, type, and remember. So please don't use underscores. And don't capitalize or BiCapitalize without extremely good reason – it messes up the natural human-eyeball search order and looks like some marketing weenie trying to be clever.

It confuses people when two different projects have the same stem name. So try to check for collisions before your first release. A good place to check is the *index file of Metalab* `<http://metalab.unc.edu/pub/Linux>`.

3 Good licensing and copyright practice: the theory

The license you choose defines the social contract you wish to set up among your co-developers and users. The copyright you put on the software will function mainly as a legal assertion of your right to set license terms on the software and derivative works of the software.

3.1 Open source and copyrights

Anything that is not public domain has a copyright, possibly more than one. Under the Berne Convention (which has been U.S. law since 1978), the copyright does not have to be explicit. That is, the authors of a work hold copyright even if there is no copyright notice.

Who counts as an author can be very complicated, especially for software that has been worked on by many hands. This is why licenses are important. By setting out the terms under which material can be used, they grant rights to the users that protect them from arbitrary actions by the copyright holders.

In proprietary software, the license terms are designed to protect the copyright. They're a way of granting a few rights to users while reserving as much legal territory is possible for the owner (the copyright holder). The copyright holder is very important, and the license logic so restrictive that the exact technicalities of the license terms are usually unimportant.

In open-source software, the situation is usually the exact opposite; the copyright exists to protect the license. The only rights the copyright holder always keeps are to enforce the license. Otherwise, only a few rights are reserved and most choices pass to the user. In particular, the copyright holder cannot change the terms on a copy you already have. Therefore, in open-source software the copyright holder is almost irrelevant – but the license terms are very important.

Normally the copyright holder of a project is the current project leader or sponsoring organization. Transfer of the project to a new leader is often signaled by changing the copyright holder. However, this is not a hard and fast rule; many open-source projects have multiple copyright holders, and there is no instance on record of this leading to legal problems.

Some projects choose to assign copyright to the Free Software Foundation, on the theory that it has an interest in defending open source and lawyers available to do it.

3.2 What qualifies as open source

For licensing purposes, we can distinguish several different kinds of rights that a license may convey. Rights to *copy and redistribute*, rights to *use*, rights to *modify for personal use*, and rights to *redistribute modified copies*. A license may restrict or attach conditions to any of these rights.

The *Open Source Initiative* `<http://www.opensource.org>` is the result of a great deal of thought about what makes software "open source" or (in older terminology) "free". Its constraints on licensing require that:

1. An unlimited right to copy be granted.
2. An unlimited right to use be granted.
3. An unlimited right to modify for personal use be granted.

The guidelines prohibit restrictions on redistribution of modified binaries; this meets the needs of software distributors, who need to be able to ship working code without encumbrance. It allows authors to require that modified sources be redistributed as pristine sources plus patches, thus establishing the author's intentions and an "audit trail" of any changes by others.

The OSD is the legal definition of the 'OSI Certified Open Source' certification mark, and as good a definition of "free software" as anyone has ever come up with. All of the standard licenses (MIT, BSD, Artistic, and GPL/LGPL) meet it (though some, like GPL, have other restrictions which you should understand before choosing it).

Note that licenses which allow noncommercial use only do *not* qualify as open-source licenses, even if they are decorated with "GPL" or some other standard license. They discriminate against particular occupations, persons, and groups. They make life too complicated for CD-ROM distributors and others trying to spread open-source software commercially.

4 Good licensing and copyright practice: the practice

Here's how to translate the theory above into practice:

4.1 Make yourself or the FSF as the copright holder

In some cases, if you have a sponsoring organization behind you with lawyers, youi might wish to give copyright to that organization.

4.2 Use a license conformant to the Open Source Definition

The Open Source Definition is the community gold standard for licenses. The OSD is not a license itself; rather, it defines a minimum set of rights that a license must guarantee in order to be considered an open-source license. The OSD, and supporting materials, may be found at the web site of the *Open Source Initiative* <http://www.opensource.org>.

4.3 Don't write your own license if you can possibly avoid it.

The widely-known OSD-conformant licenses have well-established interpretive traditions. Developers (and, to the extent they care, users) know what they imply, and have a reasonable take on the risks and tradeoffs they involve. Therefore, use one of the standard licenses carried on the OSI site if at all possible.

If you must write your own license, be sure to have it certified by OSI. This will avoid a lot of argument and overhead. Unless you've been through it, you have no idea how nasty a licensing flamewar can get; people become passionate because the licenses are regarded as almost-sacred covenants touching the core values of the open-source community.

Furthermore, the presence of an established interpretive tradition may prove important if your license is ever tested in court. At time of writing (late 1999) there is no case law either supporting or invalidating any open-source license. However, it is a legal doctrine (at least in the U.S., and probably in other common-law countries such as England and the rest of the British Commonwealth) that courts are supposed to interpret licenses and contracts according to the expectations and practices of the community in which they originated.

5 Good development practice

Most of these are concerned with ensuring portability, not only across Linuxes but to other Unixes as well. Being portable to other Unixes is not just a worthy form of professionalism and hackerly politeness, it's valuable insurance against future changes in Linux itself.

Finally, other people *will* try to build your code on non-Linux systems; portability minimizes the number of annoying perplexed email messages you will get.

5.1 Write either pure ANSI C or a portable scripting language

For portability and stability, you should write either in ANSI C or a scripting language that is guaranteed portable because it has just one cross-platform implementation.

Scripting languages that qualify include Python, Perl, Tcl, and Emacs Lisp. Plain old shell does *not* qualify; there are too many different implementations with subtle idiosyncracies, and the shell environment is subject to disruption by user customizations such as shell aliases.

Java holds promise as a portable language, but the Linux-available implementations are still scratchy and poorly integrated with Linux. Java is still a bleeding-edge choice, though one likely to become more popular as it matures.

5.2 Follow good C portability practices

If you are writing C, do feel free to use the full ANSI features – including function prototypes, which will help you spot cross-module inconsistancies. The old-style K&R compilers are history.

On the other hand, do *not* assume that GCC-specific features such as the '-pipe' option or nested functions are available. These will come around and bite you the second somebody ports to a non-Linux, non-GCC system.

5.3 Use autoconf/automake/autoheader

If you're writing C, use autoconf/automake/autoheader to handle portability issues, do system-configuration probes, and tailor your makefiles. People building from sources today expect to be able to type "configure; make" and get a clean build – and rightly so.

5.4 Sanity-check your code before release

If you're writing C, test-compile with -Wall and clean up the errors at least once before each release. This catches a surprising number of errors. For real thoroughness, compile with -pedantic as well.

If you're writing Perl, check your code with perl -c (and maybe -T, if applicable). Use perl -w and 'use strict' religiously. (See the Perl documentation for discussion.)

6 Good distribution-making practice

These guidelines describe how your distribution should look when someone downloads, retrieves and unpacks it.

6.1 Make sure tarballs always unpack into a single new directory

The single most annoying mistake newbie developers make is to build tarballs that unpack the files and directories in the distribution into the current directory, potentially stepping on files already located there. *Never do this!*

Instead, make sure your archive files all have a common directory part named after the project, so they will unpack into a single top-level directory directly *beneath* the current one.

Here's a makefile trick that, assuming your distribution directory is named 'foobar' and SRC contains a list of your distribution files, accomplishes this. It requires GNU tar 1.13

```
VERS=1.0
foobar-$(VERS).tar.gz:
        tar --name-prefix='foobar-$(VERS)/' -czf foobar-$(VERS).tar.gz $(SRC)
```

If you have an older tar program, do something like this:

```
foobar-$(VERS).tar.gz:
        @ls $(SRC) | sed s:^:foobar-$(VERS)/: >MANIFEST
        @(cd ..; ln -s foobar foobar-$(VERS))
        (cd ..; tar -czvf foobar/foobar-$(VERS).tar.gz `cat foobar/MANIFEST`)
        @(cd ..; rm foobar-$(VERS))
```

6.2 Have a README

Have a file called README or READ.ME that is a roadmap of your source distribution. By ancient convention, this is the first file intrepid explorers will read after unpacking the source.

Good things to have in the README include:

- A brief description of the project.
- A pointer to the project website (if it has one)
- Notes on the developer's build environment and potential portability problems.
- A roadmap describing important files and subdirectories.
- Either build/installation instructions or a pointer to a file containing same (usually INSTALL).
- Either a maintainers/credits list or a pointer to a file containing same (usually CREDITS).
- Either recent project news or a pointer to a file containing same (usually NEWS).

6.3 Respect and follow standard file naming practices

Before even looking at the README, your intrepid explorer will have scanned the filenames in the top-level directory of your unpacked distribution. Those names can themselves convey information. By adhering to certain standard naming practices, you can give the explorer valuable clues about what to look in next.

Here are some standard top-level file names and what they mean. Not every distribution needs all of these.

README or READ.ME
: the roadmap file, to be read first

INSTALL
: configuration, build, and installation instructions

CREDITS
: list of project contributers

NEWS
: recent project news

HISTORY
: project history

COPYING
: project license terms (GNU convention)

LICENSE
: project license terms

MANIFEST
: list of files in the distribution

FAQ
: plain-text Frequently-Asked-Questions document for the project

TAGS

generated tag file for use by Emacs or vi

Note the overall convention that filenames with all-caps names are human-readable metainformation about the package, rather than build components.

6.4 Provide RPMs

The de-facto standard format for installable binary packages is that used by the Red Hat Package manager, RPM. It's featured in the most popular Linux distribution, and supported by effectively all other Linux distributions (except Debian and Slackware; and Debian can install from RPMs).

Accordingly, it's a good idea for your project site to provide installable RPMs as well as source tarballs.

It's also a good idea for you to include in your source tarball the RPM spec file, with a production that makes RPMs from it in your Makefile. The spec file should have the extension '.spec'; that's how the rpm -t option finds it in a tarball.

For extra style points, generate your spec file with a shellscript that automatically plugs in the correct version number by analyzing the Makefile or a version.h.

7 Good communication practice

Your software won't do the world much good if nobody but you knows it exists. Also, developing a visible presence for the project on the Internet will assist you in recruiting users and co-developers. Here are the standard ways to do that.

7.1 Announce to c.o.l.a

Announce new releases to *comp.os.linux.announce* `<news:comp.os.linux.announce>`. Besides being widely read itself, this group is a major feeder for web-based what's-new sites like *Freshmeat* `<http://www.freshmeat.net>`.

7.2 Announce to a relevant topic newsgroup

Find USENET topics group directly relevant to your application, and announce there as well. Post only where the *function* of the code is relevant, and exercise restraint.

If (for example) you are releasing a program written in Perl that queries IMAP servers, you should certainly post to comp.mail.imap. But you should probably not post to comp.lang.perl unless the program is also an instructive example of cutting-edge Perl techniques.

Your announcement should include the URL of a project website.

7.3 Have a website

If you intend try to build any substantial user or developer community around your project, it should have a website. Standard things to have on the website include:

- The project charter (why it exists, who the audience is, etc).

- Download links for the project sources.
- Instructions on how to join the project mailing list(s).
- A FAQ (Frequently Asked Questions) list.
- HTMLized versions of the project documentation
- Links to related and/or competing projects.

Some project sites even have URLs for anonymous access to the master source tree.

7.4 Host project mailing lists

It's standard practice to have a private development list through which project collaborators can communicate and exchange patches. You may also want to have an announcements list for people who want to be kept informed of the project's process

7.5 Release to major archives

For the last several years, the *Metalab archive* <http://www.metalab.unc.edu/pub/Linux/> has been the most important interchange location for Linux software.

Other important locations include:

- the *Python Software Activity* <http://www.python.org> site (for software written in Python).
- the *CPAN* <http://language.perl.com/CPAN>, the Comprehensive Perl Archive Network, (for software written in Perl).

8 Good project-management practice

Managing a project well when all the participants are volunteers presents some unique challenges. This is too large a topic to cover in a HOWTO. Fortunately, there are some useful white papers available that will help you understand the major issues.

For discussion of basic development organization and the release-early-release-often 'bazaar mode', see *The Cathedral and the Bazaar* <http://www.tuxedo.org/~esr/writings/cathedral-bazaar/>.

For discussion of motivational psychology, community customs, and conflict resolution, see *Homesteading the Noosphere* <http://www.tuxedo.org/~esr/writings/homesteading/>.

For discussion of economics and appropriate business models, see *The Magic Cauldron* <http://www.tuxedo.org/~esr/writings/mafic-cauldron/>.

These papers are not the last word on open-source development. But they were the first serious analyses to be written, and have yet to be superseded.