Packaging software with RPM, Part 3: Accommodating software dependencies

Martin Streicher
Software Developer
Pixel, Byte, and Comma

12 January 2010

In this third article in a three-part series on the RPM Package Manager, discover the ins and outs of software dependencies, and learn how to control and customize your software packaging. (This series replaces an earlier series on RPM written by Dan Poirier.)
Share your expertise: What are your favorite RPM tricks? Add your comments below.

View more content in this series

In the early days of computing, each piece of software was monolithic. Excluding the ROM used to boot a machine and the operating system itself, each application provided all the libraries and code required to run. This approach was appropriate, largely because the computers of the day could not multitask. Later, however, computers advanced by leaps and bounds to support both multiple simultaneous users (time sharing) and multiple simultaneous applications. With many users running the same application and sharing system resources such as the file system and RAM, sharing code became a necessity and an optimization.

Today, nearly every operating system separates library code from application code and marries the two at run time. But a *shared library*—the name for shareable code—has pros and cons. In the plus column, each application is smaller, because each need not be monolithic. Further, a bug fix or performance enhancement in a commonly used shared library benefits all applications. However, the separation of application and library code is something of a handcuff: The shared library must be available and compatible, or the application cannot run.

The tie between an application and a library is one kind of *dependency.* If you distribute source code and require a particular set of header files to compile the code, that too is a dependency. Your code may also require a specific tool, such as Bison or Yacc, to compile. If you distribute your software in an RPM, you must verify that the target system provides all dependencies before you install your code. After all, if a system isn't suitable, there's little reason to install the application.

Part 1 of this series demonstrated how to distribute your software via RPM. Part 2 described the installation and uninstallation process in detail and explained how to install components when a complementary package is installed at some later time. This article, the last in the series, explores

another important topic—software dependencies—and discusses additional capabilities to control and customize your software packaging.

Defining dependencies

When you create an RPM package, you can declare four types of dependencies:

- If your package requires a capability provided by another, define a *requirement.*
- If other packages depend on or could eventually depend on a capability in your software, declare the capability your package *provides.*
- If your software (or part of your software) cannot coexist simultaneously with another package, specify a *conflict.*
- If your package deprecates another package or an older version of your own software, define what has become *obsolete.* If your package changes name, you should list the old name as obsolete.

Each dependency is listed separately in the RPM spec file. The syntax is `Type: Capability`, where *Type* is one of the four somewhat eponymous tags (`Requires`, `Provides`, `Conflicts`, or `Obsoletes`) and *Capability* is the name of an optional version number. If you have more than one `Capability` to list, enumerate all on the same line, delimited by a space or a comma.

For example, if your application required Perl to execute, you would specify that as follows:

```
Requires: perl
```

If your application required Perl and MySQL, you can write:

```
Requires: perl, mysql
```

Oftentimes, an application depends on a specific version or a specific major release of a package. For example, if you write Ruby code compliant with version 1.9, you depend on that version of the interpreter. To express a version dependency, add the version number to the `Capability`:

```
Requires: ruby >= 1.9
```

You can specify version numbers for any of the four types of dependencies. The syntax is identical. For instance, if your application was incompatible with versions of the Bash shell newer than version 2.0, you would write:

```
Conflicts: bash > 2.0
```

There are six comparators for version number:

- `package < revision` requires that the named `package` has a version number less than `revision`.
- `package > revision` specifies a `package` newer than `revision`.
- `package >= revision` asks for a `package` greater than or equal to `revision`.
- `package <= revision` requires a `package` less than or equal to `revision`.

- *`package = revision`* mandates a specific *`revision`*.
- *`package`* asks for any revision of the named *`package`*.

In general, the information for `Requires` and `Provides` is generated automatically based on RPM's analysis of your code and your spec file, respectively. (You can approximate what RPM computes for `Requires` using the `ldd` utility.) However, you can amend those two lists, if needed. `Conflicts` and `Obsoletes` are typically provided by the software developer.

Signing your RPM

Many developers choose RPM because it's easy to use and widely supported. However, simplicity also makes it easy for a bad actor to install the RPM, modify its contents, and repackage and redistribute the software as authentic. Mirror sites and torrents only hasten such "bootlegging." To protect yourself and those who choose to use your software, sign your RPM with a unique signature to guarantee its authenticity. Signing precludes modification: Any change in the file alters the signature, revealing a counterfeit.

There are three ways to sign your package. You can sign the package when it is built. You can re-sign a package that's already been signed. And you can sign an existing RPM that has no signature. The latter two options build on the technique of the former, so let's focus on signing an RPM when it's built.

To begin, you must have a GPG private key-public key pair. If you lack one, such a key is simple to generate. The first step is to launch `gpg-agent`, which manages secret keys. (Systems typically run a single `gpg-agent` for all users. Check with your system administrator to determine whether this step is required. If the systems already runs the agent, ask how to connect to it.)

```
$ gpg-agent --daemon --enable-ssh-support \
  --write-env-file "${HOME}/.gpg-agent-info"
```

`gpg-agent` creates the file .gpg-agent-info in your home directory. The file contains shell environment settings required to connect to the running agent. Load the information with the following command. (Type the entire command at the prompt or, for convenience, add it to your shell startup file.)

```
$ if [ -f "${HOME}/.gpg-agent-info" ]; then
  . "${HOME}/.gpg-agent-info"
  export GPG_AGENT_INFO
  export SSH_AUTH_SOCK
  export SSH_AGENT_PID
fi
```

Finally, set an additional variable to point to the terminal device you are currently using. (Again, you can add these two lines to a shell startup file to be available to every interactive session.)

```
$ GPG_TTY=$(tty)
$ export GPG_TTY
```

You are now ready to generate a key. Run the command `gpg --gen-key`, and answer the prompts. An example key-generation session is shown in Listing 1. Data entry is shown in bold.

Listing 1. Example key-generation session

```
$ gpg --gen-key
gpg (GnuPG) 1.4.9; Copyright (C) 2008 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection? 1
DSA keypair will have 1024 bits.
ELG-E keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 1024
Requested keysize is 1024 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)  0
Key does not expire at all
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: Martin Streicher
Email address: martin.streicher@example.com
Comment: Example key for RPM
You selected this USER-ID:
    "Martin Streicher (Example key for RPM) <martin.streicher@example.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.

Enter passphrase: ******
Retype passphrase: ******

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
++++++++++++++++++.+++++.++++++++++++++++++.+++++>.+++++.+++++..>+++++.+++++
```

The first prompt chooses the type of key (the default is the preferred option). The next prompt sets the size of the key in bits. Here, the more bits, the better, although longer keys take additional time to generate. If you'd like, you can set an expiry on the key at the next prompt. The next three queries ask for information to identify this key. It is customary to provide your name and e-mail address to allow users to contact you for your public key. Finally, you are prompted twice for a passphrase to add an additional layer of security. No one can sign an RPM with your key without the proper passphrase. Depending on how busy your system is, key generation can take seconds or minutes.

Upon completion, the key generator creates a new directory named $HOME/.gnupg and populates it with files that represent your private and public keys. To see the keys you have available, run `gpg`

`--list-key`. Take note of the value of `uid`: It contains the name of the key you should use to sign your RPM.

```
$ gpg --list-key
/home/strike/.gnupg/pubring.gpg
------------------------------
pub   1024D/1811A3E4 2009-11-23
uid                  Martin Streicher (Example key for RPM) <martin.streicher@example.com>
sub   1024g/15BBCF06 2009-11-23
```

To continue, you must now set options for RPM to sign the package. Create or open the file $HOME/.rpmmacros, and add three lines:

```
%_signature gpg
%_gpg_path /home/strike/.gnupg
%_gpg_name Martin Streicher (Example key for RPM) <martin.streicher@example.com>
```

The `%_signature` line selects the type of signature. Here, it's set to `gpg`. The `%_gpg_name` specifies the ID of the key to sign with. During key generation, the name was set to `Martin Streicher (Example key for RPM) <martin.streicher@example.com>` (the user ID [UID] value above), so that is repeated here. Finally, `%_gpg_path` defines the path to your keys.

With keys and configuration in place, signing an RPM requires one additional option, `--sign`.

```
$ rpmbuild -v --sign -bb --clean SPECS/wget.spec
Enter pass phrase:
Pass phrase is good.
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.46786
+ umask 022
...
```

The `rpmbuild` command shown is the same one used in Part 1—the only addition is `--sign`. When prompted for your password, enter the same passphrase you provided during key generation. The RPM build continues and signs the resulting package.

You can verify your signature to ensure that your package is pristine. To verify the signature of an RPM, run `rpm -K` on the file itself:

```
$ rpm -K wget-1.12-1.i386.rpm
wget-1.12-1.i386.rpm: (SHA1) DSA sha1 md5 (GPG) OK
```

If the command reads `OK`, the file is legitimate. If you download another developer's RPM, consider verification before you install the package.

Additional tools for RPM development

If you use RPM frequently for packaging, spec files become second nature. Nonetheless, a wide variety of tools is available to make spec file authoring easier. Here's a sample (see Resources for links to each):

- `rpmlint` validates RPM files. The tool catches a good number of mistakes—installing a binary in /etc and a configuration file in /usr are two common pitfalls `rpmlint` can detect—and you

can extend it with custom modules. `rpmlint` is written in Python and requires a handful of libraries to perform its functions.
- Easy RPM Builder is a K Desktop Environment (KDE)-based tool for assembling packages. The tool provides a number of templates to help jump-start your RPM development and provides a graphical user interface (GUI) to portions of the spec file. Easy RPM Builder does not replace `rpmbuild`, and some familiarity with RPM is required to use it effectively.
- If you don't use KDE or prefer to work with the spec file directly, you can extend both Vim and Emacs to include a special spec mode, which highlights spec file syntax and directs the creation and maintenance of a spec file.
- Although not a tool per se, the Fedora Project's RPM Guidelines page provides a vast list of best practices. Indeed, if you intend to distribute your software on Fedora, pay particular attention to the requirements for new packages. In addition, the guidelines describe how to package applications based on a number of popular platforms, including Eclipse, Perl, and Ruby. Packaging software for a particular interpreter is especially involved, because packaged software can include scripts, binaries, and source that must be rebuilt during installation directly on the target machine. For example, a Perl module may be part Perl and part `c` code. You can also find a version of the official RPM software guide on the Fedora site (see Resources below for a link) .

Additional tools help install and manage RPMs. In general, these tools are intended for systems administrators, but you may find them useful to help validate your installations and manage the software of you own development system. If you use your own development system for testing, you may find the tools helpful to purge defunct packages.

Conclusion

RPM is actively maintained. You can keep track of the efforts at the project's new home page (see Resources). RPM tools exist for most Linux distributions, including Debian. The latest version of RPM is 5.2.0, which was released in July 2009. The goals of the RPM project remain the same: "Make it easy to get packages on and off the system. Make it easy to verify a package was installed correctly. Make it easy for the package builder. Make it start with the original source code. Make it work on different computer architectures."

RPM development can be complex, because installing software is equally complex. This series touched on but a few topics. You can find a great deal of information about RPM development on the Web. There are tutorials, forums, and even an IRC channel dedicated to the topic (visit #rpm.org on Freenode). Moreover, you can find hundreds if not thousands of other RPMs online. If you face an especially knotty problem, find another RPM package with similarities and probe its spec file to deduce a solution.

If you're a software developer or a system administrator, providing your application as a package makes installations, upgrades, and maintenance much easier. Again, if you build it and package it with RPM, they will come.

Resources

Learn

- Read all three articles in this series (developerWorks, January 2010) on the RPM Package Manager.
- Read Wikipedia's overview of the RPM Package Manager.
- Visit the official RPM site for more information about package management in Linux.
- Learn about RPM in Red Hat Linux.
- Check out Fedora's RPM Guidelines.
- Get the official RPM software guide from Fedora.
- Read all of Martin's articles on developerWorks
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Learn more and download `rpmlint`.
- Learn more and download Easy RPM Builder.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Martin Streicher

Martin Streicher is a freelance Ruby on Rails developer and the former Editor-in-Chief of *Linux Magazine*. Martin holds a Master of Science degree in computer science from Purdue University, and has programmed UNIX-like systems since 1986. He collects art and toys.