GURU LABS

CREATING RPMS (Student version) v1.0

Featuring 36 pages of lecture
and a 48 page lab exercise

This document serves two purposes:
1. Representative sample to allow evaluation of our courseware manuals
2. Make available high quality RPM documentation to Linux administrators

About this material:
The blue background you see simulates the custom paper that all Guru Labs courseware is printed on. This student version does not contain the instructor notes and teaching tips present in the instructor version.

For more information on all the features of our unique layout, see:
http://www.gurulabs.com /courseware/courseware_layout.php

For more freely available Guru Labs content (and the latest version of this file), see:
http://www.gurulabs.com/goodies/

This sample validated on:
Red Hat Enterprise Linux 4 & Fedora Core v3
SUSE Linux Enterprise Server 9 & SUSE Linux Professional 9.2

About Guru Labs:
Guru Labs is a Linux training company started in 1999 by Linux experts to produce the best Linux training and courseware available. For a complete list, visit our website at:
http://www.gurulabs.com/

Guru Labs 801 N 500 W Ste 202 Bountiful, UT 84010 Ph: 801-298-5227
WWW.GURULABS.COM

Objectives:

- Understand the historical UNIX/Linux software management landscape
- Describe the features and architecture of the RPM system
- Overview of the day-to-day use of RPM and use of different RPM packages
- Rebuild and modify existing source RPMS
- Learn the syntax for RPM SPEC files and build infrastructure requirements
- Create SPEC files, supporting files and new RPMs from scratch
- Use advanced RPM creation techniques
- Manage RPM implementation differecnes between Red Hat and SUSE distributions

Section 12 CREATING RPMS

Managing Software

Unix Packaging

- System V packages

Linux Packaging

- Slackware Tarballs
- RPMs
- Debian DPKGs

Managing Installed Software

Traditionally, software for Unix systems has primarily been distributed as source code. To use this software, an administrator must compile it properly for the system on which it will be used, then install it. If it requires any supporting packages, they must also be installed. If any local customizations are required, they must be applied to the software.

When the software needs to be upgraded, an administrator must find all the original files for that software on the system and replace them with updated versions, taking care to preserve any configuration changes which have been made locally.

Now, consider a typical Unix workstation application, such as Mozilla. The basic Mozilla application consists of approximately 495 files scattered through approximately 10 different directories. When removing it, an administrator must find all those files and delete them. When upgrading it, an administrator must manually replace all those files. To operate, Mozilla requires that approximately 50 other system executables and libraries (many of which, in turn, require still other libraries) be installed.

Managing all this complexity manually is certainly possible, and Unix administrators have done so successfully for over 30 years now. However, a variety of vendors have developed tools which simplify the work necessary to cope. In the commercial Unix world, System V Unix defines a standard “package” system which is the basis for packaging software such as the pkg format used with Solaris, or the depot format used with HP-UX. These package systems supply tools which can be used to install, uninstall, and upgrade software obtained in a package format which contains both the software and metadata about the software (such as tracking of what other software the software needs installed in order to function).

In the Linux world, software was initially unpackaged, just as it was in the Unix world. When the Slackware distribution was first released in 1993, it introduced the use of a very rudimentary package format to Linux. Shortly thereafter, two new distributions, Red Hat and Debian, each began efforts to develop more full-featured packaging software for Linux. Debian introduced the DPKG packaging system, while Red Hat introduced the RPM packaging system. Since that time, the RPM packaging system has become a de facto standard in the Linux community. It is used by almost all prominent commercial Linux distributions, including those created by Red Hat, SUSE, and Mandrake. RPM is also the de jure standard for package management in the Linux world; the LSB (Linux Standards Base) effort which standardizes Linux requires that LSB-compliant systems support installation of RPM files.

RPM Features

Noninteractive, Scriptable Installation
Tracking of Installed Files
Verification of Installed Files
Queries of Installed Files
Dependency Tracking
Tracking of All Source and Build Process

- Pristine Source preservation

Digitally Signed Software

Considering RPM Features

RPM, the RPM Package Manager, provides a variety of different features which make it an excellent package manager for use by the Linux community, and a great tool to simplify the work life of Linux administrators.

RPM provides a set of tools which can be used to carry out noninteractive, scriptable installations of software. Once that software is installed, RPM provides tools which can track those installed files, making it easily possible to uninstall them later, or to upgrade them. It also provides several methods of verifying those installed files, allowing administrators to double-check that the installed software is installed correctly, and has not been inadvertently modified since installation. In addition, RPM provides a set of commands which can be used to search installed files to determine which application uses them.

RPM has excellent dependency tracking capabilities, meaning that when used to install new software, it can first ensure that any software required for that new software to work is already installed; when used to uninstall software, it can first verify that doing so will not break any other applications.

In addition, RPM provides a tool set which manages the entire process of patching, configuring, and compiling new applications. When compiling an application, RPM makes it possible to start with the pristine source code for that application, produce any needed patches for that application, then script the process of applying those patches, configuring the source code, and compiling the source code to produce executables. This aspect of RPM greatly simplifies the process of maintaining custom-configured applications in the enterprise.

Furthermore, RPM allows all packaged software to be signed digitally (using public-key technology). This feature allows the authenticity of software packaged for use with RPM to be verified, helping prevent the accidental installation of Trojan Horse software.

RPM Architecture

RPM package files

RPM database

- `/var/lib/rpm`

RPM utilities

- **`rpm`**
- **`rpmbuild`**
- **`rpmsign`**
- **`rpm2cpio`**

RPM configuration files

- macros used during preparation and installation of RPMs

Reviewing RPM Architecture

The RPM system consists of several components. Software which is to be installed using RPM must be supplied in a special format, the RPM package file. This RPM package file is an archive which contains the actual files to be installed, as well as metadata about those files which is used by the RPM system to ensure that those files are installed with the correct permissions and ownerships and in the correct locations.

As software is installed using RPM, the name and several other properties of each file being installed on the system are recorded in the RPM database, typically located in the `/var/lib/rpm` directory. This database contains a list of all installed applications, and the files which belong to those applications, allowing easy upgrade or uninstallation of applications at a later time. It also tracks properties of each file—such as its correct size, timestamp, and cryptographic checksum—ensuring that the file's correctness can be verified at a later date. This database also contains dependency information for every application, ensuring that administrators installing new applications can be certain that the applications the new software requires to operate are present, and that administrators removing applications can be certain that doing so will not break any other existing applications.

Several utilities are supplied for use with RPM. The basic utility used for most RPM administrative tasks is the `/bin/rpm` command. On dated systems, which use an older version of RPM, this command is the only command commonly used when working with RPM. On modern systems, which use newer versions of RPM, many of the functions formerly performed by the `/bin/rpm` command have been moved to helper utilities. For example, the `/usr/bin/rpmbuild` command is used to produce new RPM package files, while the `/usr/bin/rpmsign` command is used to sign new packages.

The `/usr/bin/rpm2cpio` command has always been available for conversion of RPM package files into a standard archive file format.

RPM also has several configuration files. In RHEL/FC, these are found in the `/etc/rpm` and `/usr/lib/rpm` directories, while on SUSE distributions, they are found in the `/usr/lib/rpm` directory only. These configuration files primarily define macros—short-cut commands which are used when running RPM commands, or when preparing RPM package files.

RPM Package Files

Naming Conventions
- name-version-release.architecture.rpm

Architectures
- source – .src.rpm
- noarch – .noarch.rpm
- binary – .i386.rpm

Format
- cpio archive plus a binary header

Considering RPM Files

Software prepared for use with RPM must be packaged in an RPM package file. These package files should be named in the format:

name-version-release.architecture.rpm

In this package file name, the ***name*** indicates the software which is packaged in that RPM. Usually, this ***name*** is the name of the application.

In the RPM package file name, the ***version*** indicates the version of the software which is packaged in that RPM. This ***version*** is usually the version of the application which is being packaged in that RPM package file. Although it is not required by the RPM package file format or RPM utilities, this ***version*** field should always be a number; use of words in this field can (and has) caused problems for several front-end applications which use RPM.

The ***release*** in the RPM package file name is used to indicate revisions of the packaging of that particular version of that application. Sometimes, mistakes are made when preparing RPM package files of a specific version of a specific application. When that occurs, a new package file of that specific version of that specific application can be prepared which fixes those mistakes. The ***release*** field allows these revisions of package files to be tracked. This field should be a number, and should be increased every time the package is revised.

The ***architecture*** is the platform on which that RPM can be executed, if binary. Typical values seen here include:

- i386 – the package can be used on any 32-bit Intel-compatible CPU
- i686 – the package can be used on any 686-class 32-bit Intel-compatible CPU
- ppc64 – the package can be used on the 64-bit PowerPC CPU
- x86_64 – the package can be used on AMD or Intel 64-bit CPUs
- ia64 – the package can be used on the 64-bit Itanium CPU
- sparc64 – the package can be used on a 64-bit UltraSparc CPU

In addition to binary executables, RPM package files can also be used to package platform-neutral executable code (such as programs written in an interpreted language like Lisp or Perl or Java). RPM package files which contain code which can run on any CPU, or other files -such as documentation- which are platform-neutral are packaged with in RPM package files for which the ***architecture*** is noarch.

RPM package files can also be used to package application source code, patches, and scripts specifying how that source code should be configured and compiled. These package files will have an architecture of ***src***, indicating that they contain source code rather than executables.

Regardless of their name, all RPM package files are a cpio archive with a binary header attached.

Working With RPMs

Installing without upgrading RPMs

- **rpm -i**

Upgrading/Installing & Freshing RPMs

- **rpm -U**
- **rpm -F**

Removing RPMs

- **rpm -e**

Querying RPMs and Files

- **rpmquery** / **rpm -q**

Verifying RPMs and Files

- **rpmverify** / **rpm -V**

Working With RPMs

RPM package files are normally installed using the `-U` option to the `rpm` command. Typically, this is used in conjunction with the `-v` option, which causes rpm to be more verbose, and the `-h` option, which causes `rpm` to display progress meters:

```
# rpm -Uvh ytalk-3.1.1-12.i386.rpm
Preparing...   ################################ [100%]
   1:ytalk     ################################ [100%]
#
```

If you wish to install a package AND keep the older version installed as well (this is only possible if there is no files that overlap), then use the `-i` option. This only typically done with installing a kernel package and you want to keep the older kernel installed in case the new kernel malfunctions.

Packages can also be upgraded using the `-F` option to the `rpm` command (which is typically also used with the `-v` and `-h` options). This option causes the package to be upgraded if already installed and of an older version, but not to be installed if it is not currently installed.

When performing upgrades or freshening installed packages, the rpm utility compares the version of the RPM being installed with the version, in the RPM database on the system, of the already installed RPM. If the version number of the package being installed is greater than the installed version number, RPM will upgrade. If the version number of the package being installed is less than the installed version number, RPM will refuse to downgrade. If the versions numbers of the two are the same, then RPM compares the release numbers, and upgrades only if the release number of the new package is greater than the release number of the installed package.

Once installed, applications can be removed using the `-e` option to the `rpm` command:

```
# rpm -e ytalk
#
```

When erasing installed software, only the name of the software (`ytalk`) is used. When installing new packages or upgrading existing packages, the complete package file name (`ytalk-3.1.1-12.i386.rpm`) is used.

Package files and installed packages can also be queried. The `rpmquery` command or the `rpm` command with the `-q` option can perform many powerful queries. Similarly, installed files can be verified using the `rpmverify` command or the `rpm` command with the `-V` option.

Source RPM Files

Contents

- Pristine source code
- Any needed patches and support files
- Script specifying how to build binary packages from the source code
  - .spec file

Sometimes similar contents in tarball format instead

Understanding Source RPM Files

The `rpm` command can also be used to work with source RPM files (SRPMs). SRPMs are package files which can be used to build binary RPMs, such as the `.i386.rpm` and `.noarch.rpm` RPM package files that are typically the end product and installed on systems. SRPMs contain several different files:

- Original source code for the application
- Any patches required to modify that source code
- Any needed auxiliary files for that application, such as System V init scripts
- A script specifying how to configure and compile the source code; this script is typically named application.spec

One design principal behind RPM is that packages should always be compiled from the original source code for that application. This criterion is important in the Linux community, where individual applications are developed by a large loosely organized collection of organizations all over the Internet, and then collated into a cohesive operating system by distribution vendors such as Red Hat or SUSE. Distribution vendors often need to modify the original source code to make it integrate better into their distribution. Similarly, end users often need to recompile, and occasionally need to patch, applications shipped by their distribution vendor to better suit their local environment. Preparing RPMs from original source code, plus separate patches, helps ensure long-term ease of maintenance. It also allows end-users to determine easily exactly how a package has been modified.

Typically, application developers who support creation of binary RPMs from their source code do so by releasing SRPMs. Some application developers, however, instead release their source code in a standard Unix tarball format, and simply include within the tarball a spec file specifying how to produce a binary RPM from that source code.

Using Source RPMs

Why?

- Typically, need software not included with distribution
- Sometimes need newer version of existing software, or existing software compiled with different features
- Want maintenance benefits of packaging for this needed software

Two approaches

- Rebuild existing src.rpm
  - Not modifying it in the process
  - Customizing the existing package
- Creating new src.rpm

Using Source RPMs

Although administrators can often obtain binary RPMs of the software which they need to install, sometimes administrators will instead need to compile software. There are several reasons why an administrator might need to compile software from source, including:

- A need for software not included in the Linux distribution
- A need for a newer version of an application included in the Linux distribution
- A need for an application compiled with different options or features than the build included in the Linux distribution
- A need to fix a bug in an application included in the Linux distribution

When compiling software from source, an administrator might still want the benefits of using a package management system like RPM, such as the ability to uninstall the software easily at a later date, or the ability to upgrade the software easily when necessary. Similarly, an administrator might want the benefits gained by building software using the RPM software, such as the inherent tracking of all build commands and the coordination of patches with software.

For reasons such as these, an administrator might choose to compile software from a SRPM to produce binary RPMs. To produce a binary RPM, the administrator will first need to have a source RPM which can be used to produce the binary RPM. The administrator will typically obtain a SRPM by either of two methods:

- Creating a new SRPM from scratch
- Using an existing SRPM

When working from an existing SRPM, the administrator might use it as-is, or might choose to customize it. All of these actions are possible using RPM.

Preparing to Build RPMs

Install Necessary Software Prerequisites
- gcc, make, and other development tools
- All needed dependencies
- Should be same environment as expected on target host

Do not build RPMs as root, to prevent changing current system

Prepare Non-root Build Environment On Red Hat
- Create needed directory structure
- Create `~/.rpmmacros`

Preparing to Build RPMs

When building binary RPMs from SRPMs, some preparatory steps on the system which will be used to build the RPMs are first required. The software needed to compile the binary RPM needs to be installed on the build host. Typically, this requirement means that common development tools such as the gcc C compiler and the `make` command need to be installed. Depending upon how the Linux distribution was installed, these tools may or may not be installed already. In addition to development tools, most software has build dependencies which must be present when the software is being compiled. Many applications require support libraries and header files which must be present when they are compiled, and these also need to be installed if not already present.

The binary, once compiled, will be compiled to execute on a system with the same system libraries and features as the system on which it was compiled. For this reason, it is best to compile software on the same operating system release as the system on which it will be installed.

When compiling binary RPMs from SRPMs, it is important to do so as a non-root user whenever possible.As part of the RPM build process, the RPM software performs an install of the compiled software into a virtual filesystem environment. Errors in the `spec` file controlling the RPM build process can cause RPM to install to the real filesystem instead. Building RPMs as a non-root user, though not possible for all RPMs, prevents the build process from accidentally writing to the real filesystem since non-root users do not typically have write access to the directories in which the RPM build process would attempt to install software.

On current releases of SUSE's Linux distributions, RPM is automatically configured to support RPM builds by non-root users within the `/usr/src/packages` subdirectory. On Red Hat distributions, the system will need to be configured to support non-root RPM builds.

To enable RPM building on Red Hat distributions as a normal user:

1. Log in as the non-privileged user who will be performing builds.
2. Create the directory structure needed by the RPM build process:

```
$ mkdir -p ~/rpmbuild/{BUILD,RPMS,S{OURCE,PEC,RPM}S}
or
$ cp -a /usr/src/redhat ~/rpmbuild
```

3. Configure RPM to use this new directory structure, rather than the default directory structure, when building RPMs:

```
$ echo "%_topdir $HOME/rpmbuild" > ~/.rpmmacros
```

Once these commands are completed, RPMs can be built under the `rpmbuild` directory in the user's home directory. To build RPMs elsewhere, simply adjust the paths as appropriate.

Rebuilding Existing Packages

Source RPMs

- `rpmbuild --rebuild`
- `rpmbuild --recompile`

Tarballs containing .spec files

- `rpmbuild -ta`

Rebuilding Existing Packages

Once a proper build environment has been prepared, any existing packages which need recompilation can be rebuilt. On modern systems, the commands:

```
$ rpmbuild --rebuild name-ver-rel.src.rpm
```

can be used to compile the ***SRPM*** to produce a binary RPM, while

```
$ rpmbuild --recompile name-ver-rel.src.rpm
```

can be used to compile an unpackaged binary from the ***SRPM***.

Or on older systems, the commands:

```
$ rpm --rebuild name-ver-rel.src.rpm
```

can be used to compile the ***SRPM*** to produce a binary RPM, while

```
$ rpm --recompile name-ver-rel.src.rpm
```

can be used to compile an unpackaged binary from the ***SRPM***.

Some developers release source code as tarballs which contain a spec file. RPM can also be used to compile this software. On modern systems, the command:

```
$ rpmbuild -ta tarball
```

can be used to compile the ***`tarball`*** to produce a SRPM and a binary RPM.

On older systems, the command

```
$ rpm -ta tarball
```

can be used to compile the ***`tarball`*** to produce a SRPM and a binary RPM.

Installing source RPMs

By installing a source RPM, using the command:

```
$ rpm -Uvh name-ver-rel.src.rpm
```

The pristine source tarball, patches and supporting files will be placed in your defined `SOURCES` directory. The spec file will be placed in your `SPECS` directory. This only needed if you want to change something before you make the binary RPM.

Creating New RPMs

Prepare any needed patches
- Changes to released source code

Prepare any needed support files
- Startup scripts and similar files

Create the .spec file
- “recipe” for compilation of software

Build and Test the RPM

Creating New RPMs

Many times, an existing SRPM is available for the software which needs to be compiled. If one is not available, however, one can be created from scratch by following a few simple steps:

- Preparing any needed patches
- Preparing any needed support files
- Creating the .spec file
- Building and testing the RPM

Sometimes, released software needs a patch to alter the way the code behaves. If the application being compiled needs any patches, they will need to be prepared.

Similarly, applications often require support files which typically are not distributed with the application. For example, system daemons often require a System V initialization script to run them at system start-up, but these init scripts are rarely supplied with the daemon. If any such support files are required, they must be created.

Instructions which tell the RPM software how to compile the software must also be prepared. These instructions are stored in a `spec` file which the RPM software will use when compiling the RPM.

After these basic components have been created, the RPM can be built and tested. Typically, doing so is an iterative process—during the build process, mistakes in patch files or spec files will be caught and corrected, then the build will be repeated. Eventually, a working build will be obtained.

It is a good idea to submit any non local-specific patches you had to create as well as the spec file back to the author of the software.

Patching Software

SRPM Contains Pristine Source and Patches
- Eases long-term maintenance
- Allows authenticity verification

Why Patch?
- Change hard-coded default values
- Back port changes from development tree
- Make changes to better integrate package with the system
- Fix software bugs

Generating Patches
- Create .org file then modify existing file
- Use **`/usr/bin/diff`** to create patch file

Patching Software

When preparing software for a package, the software often needs modifications to the source code. These modifications might be needed for any of several different reasons, including:

- A need to change hard-coded values, such as directory paths or usage limits, within the software
- A need to fix bugs in already released software
- A need to make the software better integrate with other software on the system

Any changes to the original source code of an application which are made by a packager need to be made in the form of a patch file, and not by directly modifying the original source of the application. SRPMs always contain the original source code for an application, as well as any patch files which are needed to make changes to that source code. This policy eases long-term maintenance, since it means any local changes are preserved in their own patch file and easily tracked. It also simplifies verification of the authenticity of the source code, an important consideration in today's Trojan Horse-flooded Internet.

Patch files are easily created. To create a patch file, first make a copy of the file to be modified under its original with a `.orig` extension. Then, make any modifications to the file which are needed. Next, use the `diff` command with the `-N`, `-a`, `-u`, and `-r` options to generate the differences between the original file and the modified file. Finally, save these differences as a patch file.

For example, suppose an application, `less`, required a couple of customizations to the files `edit.c` and `filename.c` in its source code. To prepare patch files of these changes:

1. Obtain and unpack the original source code.

```
$ tar -zxvf less-378.tar.gz
```

2. Back up the files to be modified.

```
$ cd less-378/
$ cp edit.c edit.c.orig
$ cp filename.c filename.c.orig
```

3. Modify the source files

```
$ vi edit.c
$ vi filename.c
```

4. Save the differences in the edited files to patch files

```
$ diff -Naur edit.c.orig edit.c > edit.patch
$ diff -Naur filename.c.orig filename.c >
filename.patch
```

At this point, two patch files for the less source code exist which can be used when preparing a SRPM to build the patched less program.

Creating Support Files

Does package need a scheduled job?
- Create file for /etc/cron.*

Does package need log file rotation?
- Create file for /etc/logrotate.d/

Does package need shell or environment changes?
- Create files for /etc/profile.d/

Consider special cases:
- Startup scripts for daemons
- Desktop file(s) for menus
- Web configuration files
- Authenticating daemons – /etc/pam.d/

Creating Support Files

In many cases, creation of support files needed for correct operation of an application will often be necessary. Many applications need to run scheduled tasks on a regular basis. For example, the SquirrelMail application creates temporary files which it deletes by running the tmpwatch utility periodically. On Red Hat and SUSE distributions, the following directories exist for use by programs which need to schedule tasks:

- `/etc/cron.d/`
- `/etc/cron.hourly/`
- `/etc/cron.daily/`
- `/etc/cron.weekly/`
- `/etc/cron.monthly/`

Packages can put shell scripts in the mentioned directories and those scripts will execute hourly, daily, weekly, or monthly. Packages which need more flexible scheduling can put crontabs in the `/etc/cron.d/` directory. When packaging an application which needs to run a command periodically, it will be necessary to create the appropriate file to put in one of these directories.

Similarly, some applications require shell or environment changes. Shell scripts making those changes can be placed in the `/etc/profile.d/` directory by the package. Programs which generate a logfile need to ensure that their logfile gets rotated periodically, and will need to place a configuration file in the `/etc/logrotate.d/` subdirectory.

Daemons which need to be executed on system start-up will need an initialization script created for them. For most daemons, this will entail creation of a System V init script, while for daemons which are started by a super-server, an xinetd or inetd initialization script will need to be created.

Some applications require further configuration changes. X-based applications need to add entries to the menu systems used in the X environment, and so need to provide a configuration file which adds them to the menu structures. Web applications and Apache modules often need to make configuration changes to the Apache web server, and so need to provide a configuration file to modify the behavior the of the Apache web server. Applications which perform authentication need to supply configuration files for the PAM subsystem.

Creating SysV init scripts

Standalone daemons require SysV init script
- **`/etc/init.d/scriptname`**

RHEL/FC enhanced SysV scripts
- Use standard functions from **`/etc/init.d/functions`**

  `daemon`
  `killproc`

SLES/SL SysV script commands provided in `sysvinit` RPM

checkconfig enabled SysV scripts
- Requires special comments in script
- Can be managed by **`/sbin/chkconfig`**
- RPM post-install should register script with chkconfig

SysV init scripts

The use of SysV init scripts allow standalone daemons to be managed using the standard method of running:

```
# /etc/init.d/scriptname (start|stop|restart)
```

When creating an RPM for a daemon it is good practice to create and install a SysV init script for the package so that system administrators can manage the daemon just as they do other stock daemons that came with the system.

When creating a SysV init script, it can be helpful to start with an existing, simple one and modify appropriately. One such example on RHEL/FC is the `/etc/init.d/smartd` SysV script. On SLES/SL the `/etc/init.d/skeleton` SysV script is provided.

The official documentation for RHEL/FC SysV init scripts is found in the `/usr/share/doc/initscripts-*/sysvinitfiles` file while on SLES/SL consult `/etc/init.d/README`.

Enabling chkconfig control of SysV init scripts

The **`chkconfig`** command allows system administrators to easily control what run level a daemon is started in, for example:

```
# chkconfig --level 2345 httpd on
```

To allow this control the SysV init script must have two special comment line near the top of the file, for example:

```
#!/bin/bash
#
# Startup script for the Apache Web Server
#
# chkconfig: 2345 85 15
# description: Apache is a World Wide Web server.
#
```

The `chkconfig:` line states that this daemon should by default be started in runlevels 2,3,4 and 5 and use have starting order 85 and stop order 15. If your daemon should not be started by default in any runlevel, use the minus sign in that position instead of the runlevel numbers. In that case, after installing your RPM, the system administrator would then run:

```
# chkconfig scriptname on
```

Registering the SysV init script with chkconfig

In the RPM spec file, in the post install script stanza, you should register your SysV init script to be controlled by chkconfig by having the command:

```
/sbin/chkconfig --add scriptname
```

Creating Menu Entries

freedesktop.org

- Multiple formal and informal specifications for interoperability
- Desktop Menu Specification
  - Standardized menu specification

***applicationname*.desktop files**

- Stored in the `/usr/share/applications/` directory

desktop-file-utils software

- **`desktop-file-install`**
- **`desktop-file-validate`**

Creating Menu Entries for User Applications

When packaging application software in RPM format as opposed to system software, it is recommended that a menu entry is created for the software so that users can easily find and launch the application.

Historically, the GNOME and KDE desktops used their own separate menus and menu files with different format. This was recognized as a problem and the two groups came together and created the Desktop Entry Standard.

The Desktop Entry Standard defines that the directory `/usr/share/applications/` stores XML files for each menu entry in the format `applicationname.desktop`.

Once a desktop file is created, it is installed using the **`desktop-file-install`** command. The files can be validated to ensure that the syntax is correct using the **`desktop-file-validate`** command.

For small desktop files that don't use translated descriptions, they are often defined and created directly within the package spec file.

Full details on the structure and rules for the desktop files can be found at the URL:

http://standards.freedesktop.org/menu-spec/latest/

Example spec file desktop generation and installation

```
%install
[snip lines unrelated to desktop]
# Desktop menu entry
cat > %{name}.desktop << EOF
[Desktop Entry]
Name=BZFlag
Comment=%{summary}
Exec=%{name}
Icon=bzflag-m.xpm
Terminal=0
Type=Application
EOF

mkdir -p %{buildroot}%{_datadir}/applications
desktop-file-install --vendor TimRiker        \
  --dir %{buildroot}%{_datadir}/applications  \
  --add-category X-Red-Hat-Extra              \
  --add-category Application                  \
  --add-category Game                         \
  %{name}.desktop
```

The .spec file

Defines meta-info about the package

Describes how to compile package

Describes what files to install

- Defines permissions with which to install files

Contains scripts to execute before and after installation and uninstallation

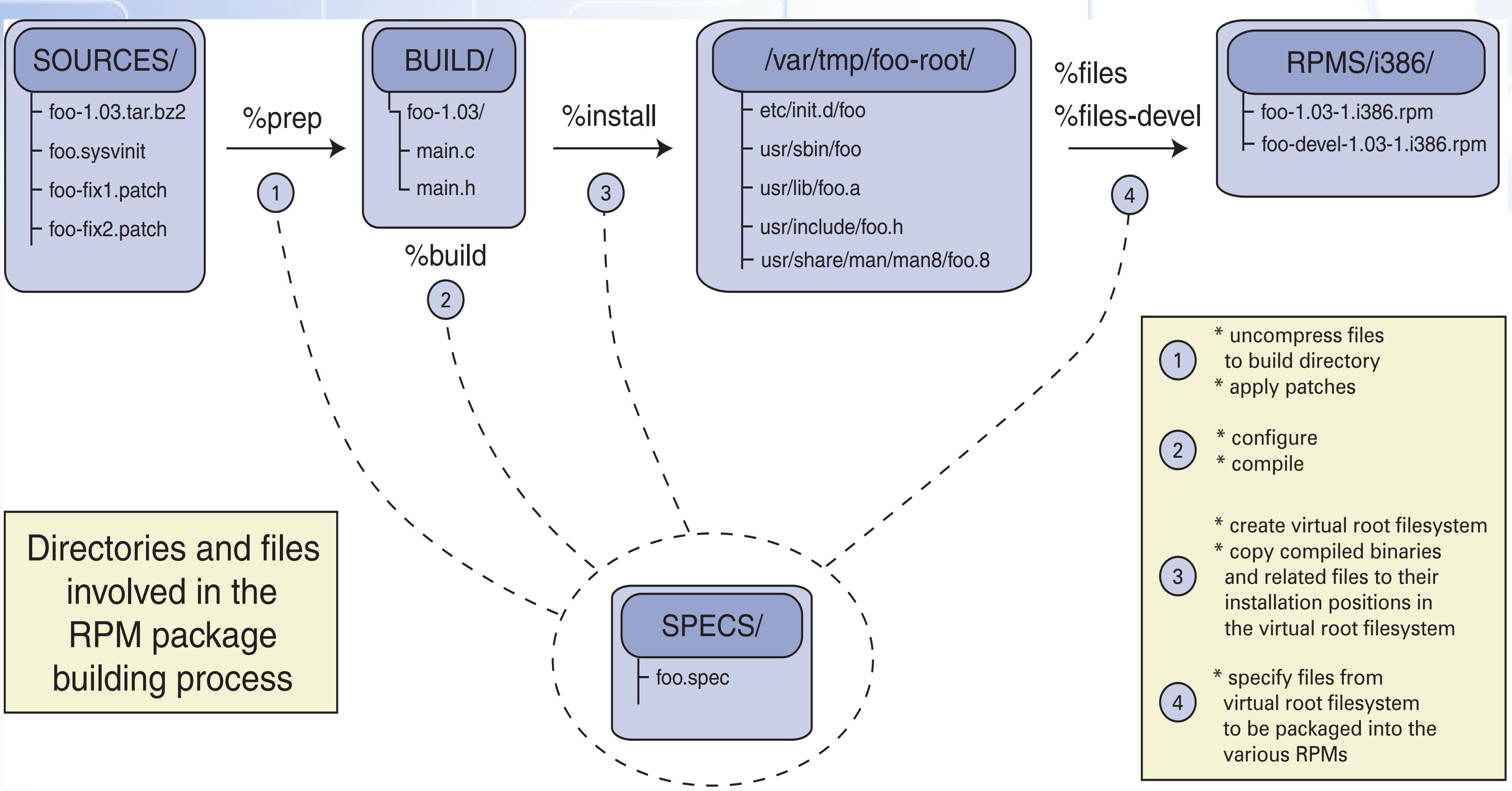
Examining .spec Files

After creating any required patch files and support files which are needed for the application, a spec file must be created. This spec file is written in a syntax which interleaves a macro programming language with shell commands and with descriptive text. In the spec file, the number sign (#) is used to denote comments, just as in most other Unix configuration files.

The spec file consists of several closely related sections:

- The Header stanza
- The Prep stanza
- The Build stanza
- The Install stanza
- The Files stanza
- The Scripts stanza
- The Changelog stanza

Together, these sections define the source files and patches which make up the application, provide detailed information about the source and use of the application, instruct RPM on how to compile the application, define the files which RPM needs to install when installing the application, as well as how to install those files. The spec file can also contain optional scripts which are executed before and after installation or uninstallation of the application.


SOURCES/
foo-1.03.tar.bz2
foo.sysvinit
foo-fix1.patch
foo-fix2.patch
%prep
1
BUILD/
foo-1.03/
main.c
main.h
%build
2
%install
3
/var/tmp/foo-root/
etc/init.d/foo
usr/sbin/foo
usr/lib/foo.a
usr/include/foo.h
usr/share/man/man8/foo.8
%files
%files-devel
4
RPMS/i386/
foo-1.03-1.i386.rpm
foo-devel-1.03-1.i386.rpm
Directories and files involved in the RPM package building process
SPECS/
foo.spec
1 * uncompress files to build directory * apply patches
2 * configure * compile
3 * create virtual root filesystem * copy compiled binaries and related files to their installation positions in the virtual root filesystem
4 * specify files from virtual root filesystem to be packaged into the various RPMs

The foo.spec file

```
Summary: The world famous foo
Name: foo
Version: 1.03
Release: 1
License: GPL
Group: Applications/System
Source0: foo-%{version}.tar.bz2
Source1: foo.sysvinit
Patch0: foo-fix1.patch
Patch1: foo-fix2.patch
BuildRoot: /var/tmp/%{name}-root

%description
This foo daemon serves bar clients.
```

```
%prep
%setup -q

%build
%configure
make

%install
rm -rf %{buildroot}
%makeinstall

%clean
rm -rf %{buildroot}
```

```
%files
%defattr(-,root,root)
/etc/init.d/foo
%{_sbindir}/foo
/usr/share/man/man8/foo.8

%files devel
%defattr(-,root,root)
/usr/include/foo.h
/usr/lib/foo.a

%changelog
* Mon Apr 30 2003 Dax Kelson
- ver 1.03
```

Using Macros

Widely used in .spec files

- Simplify creation of .spec files

System-wide macros

- `/usr/lib/rpm/macros`
- Vendor-specific files
  `/usr/lib/rpm/suse_macros`
  `/etc/rpm/macros*`

User-defined macros

- `~/.rpmmacros`

Using Macros

Macros are widely used throughout RPM to perform configuration. These configuration macros can be set globally, in the `/usr/lib/rpm/macros` configuration file. They can also be set on a per-user basis; whenever an RPM command is executed, the RPM software looks for the configuration file `.rpmmacros` in the user's home directory, and any configuration directives defined in this file override global configuration options.

Macros are commonly used in spec files as well. Pre-defined macros exist for several commonly used fields within the spec file, and macros also exist for many commonly used directories and paths on the system.

In configuration files and spec files, values are assigned to macro names by listing the macro name followed by its value. The statement

```
%_topdir /export/home/rpmmaker/rpmbuild
```

in `~/.rpmmacros`, for example, assigns the value

```
/export/home/rpmmaker/rpmbuild
```

to the macro

```
%_topdir
```

Within configuration files and spec files, the value of a macro is accessed by using the macro name encased in braces. For example, any time the statement

```
%{_topdir}
```

appears within a spec file for this *rpmmaker* user, it will automatically be replaced by the value

```
/export/home/rpmmaker/rpmbuild
```

Because macros are so useful when writing spec files, many distribution vendors supply a variety of pre-defined macros which can be used when writing spec files. On Red Hat distributions, these vendor-specific macros are put in the `/etc/rpm` subdirectory, while on SUSE distributions, these vendor-specific macros are put in the `/usr/lib/rpm/suse_macros` file. In addition, many standard macros are defined in the configuration files in the `/usr/lib/rpm` subdirectory. When preparing packages for building on multiple distributions, care should be taken to avoid using vendor-specific macros.

Directory Macros

%_prefix	/usr
%_exec_prefix	%{_prefix}
%_bindir	%{_exec_prefix}/bin
%_sbindir	%{_exec_prefix}/sbin
%_libexecdir	%{_exec_prefix}/libexec
%_datadir	%{_prefix}/share
%_sysconfdir	%{_prefix}/etc
%_sharedstatedir	%{_prefix}/com
%_localstatedir	%{_prefix}/var
%_lib	lib
%_libdir	%{_exec_prefix}/%{_lib}
%_includedir	%{_prefix}/include
%_oldincludedir	/usr/include
%_infodir	%{_prefix}/info
%_mandir	%{_prefix}/man

The %configure macro

```
%configure \
CFLAGS="${CFLAGS:-%optflags}" ; export CFLAGS; \
CXXFLAGS="${CXXFLAGS:-%optflags}"; export CXXFLAGS; \
FFLAGS="${FFLAGS:-%optflags}" ; export FFLAGS ; \
./configure --host=%{_host} --build=%{_build} \\\
        --target=%{_target_platform} \\\
        --program-prefix=%{?_program_prefix} \\\
        --prefix=%{_prefix} \\\
        --exec-prefix=%{_exec_prefix} \\\
        --bindir=%{_bindir} \\\
        --sbindir=%{_sbindir} \\\
        --sysconfdir=%{_sysconfdir} \\\
        --datadir=%{_datadir} \\\
        --includedir=%{_includedir} \\\
        --libdir=%{_libdir} \\\
        --libexecdir=%{_libexecdir} \\\
        --localstatedir=%{_localstatedir} \\\
        --sharedstatedir=%{_sharedstatedir} \\\
        --mandir=%{_mandir} \\\
        --infodir=%{_infodir}
```

The %makeinstall macro

```
%makeinstall \
  make \\\
    prefix=%{buildroot}%{_prefix} \\\
    exec_prefix=%{buildroot}%{_exec_prefix} \\\
    bindir=%{buildroot}%{_bindir} \\\
    sbindir=%{buildroot}%{_sbindir} \\\
    sysconfdir=%{buildroot}%{_sysconfdir} \\\
    datadir=%{buildroot}%{_datadir} \\\
    includedir=%{buildroot}%{_includedir} \\\
    libdir=%{buildroot}%{_libdir} \\\
    libexecdir=%{buildroot}%{_libexecdir} \\\
    localstatedir=%{buildroot}%{_localstatedir} \\\
    sharedstatedir=%{buildroot}%{_sharedstatedir} \\\
    mandir=%{buildroot}%{_mandir} \\\
    infodir=%{buildroot}%{_infodir} \\\
 install
```

Common Header Fields

`Name` - the application
`Version` - the application's version
`Release` - the revision of that version's packaging
`License` - the license used for the application
`Group` - the category to which that application belongs
`Source` - source files for the application
`Patch` - patch files to modify the source files
`URL` - the location of the original application source
`Requires` - any software required for the application to work
`BuildRequires` - any software required to compile the app

Creating the Header

The `Header` stanza is usually the first section to appear in the spec file. It provides a lot of metadata about the application being packaged. Fields commonly found in the header include:

- Name - the application's name
- Version - the application's version
- Release - the revision of that version's packaging
- License - the license used for the application
- Group - the category to which that application belongs
- Source - source files for the application
- Patch - patch files to modify the source files for the application
- URL - the location of the original application source
- Requires - any software required for the application to work
- BuildRequires - any software required to compile the app

Valid values for the categories used in the `Group` header can be found in the */usr/share/doc/rpm-*/GROUPS* file on Red Hat systems.

Often, more than one `Source` file or more than one `Patch` file is needed for an RPM. In that case, the `Source` and `Patch` headers are simply numbered:

```
Source:
Source1:
Patch0:
Patch1:
```

The `Requires` and `BuildRequires` headers are optional. RPM will automatically calculate dependencies for software as it builds the software; the `Requires` header just allows developers to list run-time dependencies if they so desire. Similarly, the `BuildRequires` field is used to specify any applications which have to be present for the software to compile correctly. Although not always used, it is courteous to define any non-obvious build requirements to aid others trying to compile the software. Both `Requires` and `BuildRequires` can list the name of the software they require, or they can list the name and the version if specific versions are required.

Two other fields are always seen in the `Header`. The `Summary` field is used to provide a short, one-line blurb describing the application being packaged, while the %description field provides a longer, potentially multi-paragraph description of the application being packaged.

If any custom macros are being created and defined for use within the spec file, these macros will typically be defined within the `Header` stanza of the spec file.

Advanced Header Fields

Arch options

- BuildArch
- ExclusiveArch
- ExcludeArch

Epoch

Using Advanced Features

Other fields are also sometimes seen in the `Header` stanza for use in specialized circumstances. A variety of `Architecture` options exist which can be used to control the platforms on which the package will be built. The `BuildArch` directive is used to force a package to be built for a particular architecture, rather than the default architecture of the machine on which it is being build. Commonly, this is used to indicate that the built package should be a `.noarch.rpm`:

```
BuildArch: noarch
```

Not all software will compile on all architectures, just as not all applications are even usable on all architectures. Two directives exist which can be used to enforce these requirements. The `ExclusiveArch` statement can be used to list platforms on which the package is supported, instructing the RPM software not to build it on other platforms. The `ExcludeArch` statement can be used to list platforms on which the package is not supported, instructing the RPM software to build it on all other platforms. The statement

```
ExcludeArch: s390 s390x
```

specifies that the package should be built on all machines except 31-bit and 64-bit s/390 hardware, while the statement

```
ExclusiveArch: i386 s390 s390x x86_64
```

specifies that the package should be built only on 32-bit Intel-compatible hardware, 31-bit and 64-bit s/390 hardware, and 64-bit AMD (Opteron) hardware, but not on any other systems.

RPM also provides another field in the `Header` stanza, the `Epoch`. Normally, the RPM software uses the combination of package version and package release to determine if one package is newer than another. Some software uses non-standard versioning, or has changed versioning. For example, the Postfix MTA historically used YearMonthDate as its version (such as the postfix-19990906.tar.gz release). More recently, Postfix's author has adopted a more traditional two-digit versioning system (such as the postfix-2.0.9.tar.gz release). When RPM compares 19990906 with 2.0.9, 19990906 appears to be the newer software (since the version is higher), even though it is actually 4 years older than the 2.0.9 Postfix release. To correct these sorts of problems, RPM supports an `Epoch` field. Normally, this field is assumed to be zero if not present, but can be set to a higher number if necessary. Whenever package versions are compared, RPM first compares the `Epoch`, then the Version, then the Release. Because of this, a Postfix package with an `Epoch` of 1 and a Version of 2.0.9 will correctly appear newer than a Postfix package with an Epoch of 0 and a Version of 19990906.

The Prep Section

Used to prepare the software

- Untar the source code
- Apply any needed patches
- Perform any other required setup steps

Preparing the Software

After the `Header` stanza, the next stanza in the spec file is usually the `Prep` stanza. This section is always begun by the token

```
%prep
```

and is used to prepare the software to be compiled. Typically, this section unpacks the archived source code files, and applies any needed patches to these files. This can be done using standard shell commands, but is more typically done using predefined macros.

One macro commonly seen here is

```
%setup
```

This macro unpacks the source code and then changes directory to the directory where the source code was unpacked. Options are available to `%setup` as well. For example, after untarring the package, the `%setup` macro assumes the directory that is created is named:

```
%{name}-%{version}
```

If the directory inside the tarball is named differently, you can use the `-n` option to specify the directory name to cd into. For example:

```
%setup -n %{name}-April2003Rel
```

Another commonly used option to %setup is the `-q` option which turns off the verbose output from the tar command.

Another macro typically used here is

```
%patch
```

This macro applies the patches defined in the `Header` stanza. If multiple patches are being used, it accepts a numeric argument to indicate which patch file should be applied. It also accepts a `-b` *`extension`* argument to instruct the RPM software to back up files with the specified ***`extension`*** before patching them.

The following macro

```
%patch2 -b .test
```

instructs the RPM software to apply Patch2, and to save a backup of any files being patched with a `.test` extension before patching them.

The `-p` option to the `%patch` macro controls the -p option passed to the **`/usr/bin/patch`** command. Typically, `-p1` is used, however, it depends on how the patch file was created in the first place.

The Build Section

Used to compile the software

- Configure the software
- Compile the software

Defining the Build Process

After the `Prep` stanza, the spec file usually contains a `Build` stanza. The `Build` stanza always begins with the token:

```
%build
```

In this stanza, commands to configure the software and to compile the configured software are listed. As with the `Prep` section, these commands can be shell commands, or they can be macros.

If the software being compiled is designed for use with autoconf, the

```
%configure
```

macro should be used to configure the software. This macro automatically specifies the correct options to autoconf to install software correctly, and to compile it optimized for best performance.

If the software is not designed for configuration using autoconf, use shell commands to configure the software appropriately.

Once the software is configured, it must be compiled. Since compilation methods vary so widely from application to application, no macro exists for compilation. Simply list the shell commands which would be used to compile the software.

A shell variable,

```
$RPM_OPT_FLAGS
```

is commonly used when compiling software. This shell variable contains the correct optimization flags for the gcc suite of compilers. Using syntax such as

```
make CC=”gcc $RPM_OPT_FLAGS”
```

or

```
make CFLAGS=”$RPM_OPT_FLAGS”
```

will ensure that the appropriate optimization flags are always used. Other compiler flags and options can, of course, be specified as necessary.

The default value of `$RPM_OPT_FLAGS` is:

```
-O2 -g -march=i386 -mcpu=i686
```

The Install Section

Used to install the software

- Install the software into a virtual directory structure
- Clean up the temporary build directory

Relies on Buildroot set in Header stanza

Installing the Software

After the `Build` stanza, the next section of the spec file is the `Install` stanza. This stanza always begins with the token:

```
%install
```

This stanza is used to install the compiled software into a virtual directory structure so that it can be packaged up into an RPM.

In the `Header` stanza, the `Buildroot` can be specified. This `Buildroot` defines the location of a virtual directory tree into which the software can be installed. Usually, this will be declared as:

```
Buildroot: %{_tmppath}/%{name}-root
```

or

```
Buildroot: %{_tmppath}/%{name}-%{version}-root
```

using RPM's built-in macros to specify a private directory under the `/var/tmp` directory.

The shell variable

```
$RPM_BUILD_ROOT
```

can be used to access the value of `Buildroot` throughout the rest of the spec file.

```
mkdir -p $RPM_BUILD_ROOT/usr/share/icons/
cp %{SOURCE3} $RPM_BUILD_ROOT/usr/share/icons/
```

The `Install` stanza typically lists shell commands to install the compiled software within this `Buildroot`.

The macro `%makeinstall` can be used to install software which supports autoconf; this macro automatically installs software into the correct subdirectories under the `$RPM_BUILD_ROOT`.

Sometimes, a package needs to be built more than once, due to packaging errors or similar problems. Each time the package is build, the Install stanza will copy files into the `Buildroot`. To prevent incorrect packaging due to old files within the `Buildroot`, the `Buildroot` should always have any existing files deleted before installing new files into it. For this purpose, a clean script can be created within the `Install` stanza. This script is always denoted by the token

```
%clean
```

and usually just consists of the command

```
rm -rf $RPM_BUILD_ROOT
```

If present, this `%clean` section is run after preparing packages from the software installed in the `Install` stanza, ensuring that the `Buildroot` is correctly empty for the next time the package is built.

The Files Section

Used to define the files which should be packaged

- Only packages files installed into the Buildroot
- Should specify ownerships and permissions for the files being packaged

Controlling Installed Files

After the `Install` stanza, the next section of the spec file is the `Files` stanza, which lists the files and directories which should be packaged into an RPM. This stanza always begins with the token:

```
%files
```

Within this section, simply list, on a one-file-per-line-basis, the files and directories, relative to the `Buildroot`, which the RPM software should archive into packages. Wildcards can be used within this section, such as the statement

```
/usr/bin/*
```

which instructs the RPM software to package all files within the directory `$RPM_BUILD_ROOT/usr/bin`.

When listing files and directories, care must be taken to list all files which are needed, and not to list any files which should not be packaged. Great caution should be used when listing a directory. Listing a directory instructs the RPM software to package that directory and all files within it, so the statement

```
/usr/bin
```

instructs the RPM software to package all files within the directory `$RPM_BUILD_ROOT/usr/bin`, but also incorrectly produces a package which appears to own the `/usr/bin` directory!

Within the `Files` stanza, several macros can be used. The `%defattr` macro should always be used to specify default ownerships and permissions which apply to files listed after it. For example, the statement `%defattr(0770,root,root)` would specify that all files and directories listed after it and prior to any subsequent `%defattr` macros would be installed owned by the root user and group, and with permissions 0770. Further, individual files can have different permissions and ownership than the `%defattr` macro specifies by using the `%attr` macro. For example:

```
%files
%defattr(-,root,root)
%{_libdir}/libamanda*.so
%attr(660,amanda,disk) /var/lib/amanda/.amandahosts
```

Several macros exist to indicate that installed files have special properties. The macro

```
%dir
```

can be used to specify that a directory should be packaged, rather than the files within that directory. The macro

```
%config(noreplace)
```

can be used to indicate that the installed file is a configuration file, and shouldn't be overwritten on a package upgrade.

Optional Script Section(s)

preinstall / postinstall / preuninstall / postuninstall scripts

- Allow execution of commands before and after installation and deletion
- triggers can make script execution conditional

Commonly used to create necessary user accounts

Executing Commands During Installation or Deletion

Sometimes, commands need to be executed on the system before or after software gets installed. For example, after new shared libraries are installed, the ldconfig command needs to be run so that the system will use the newly installed libraries.

The RPM spec file can contain scripts which get executed before or after package installation or uninstallation. These scripts are usually listed after the `Files` stanza, and are simply Bourne shell scripts which are listed after a macro indicating when they should execute. Available scripts are:

- `%pre` – executed before the package is installed
- `%post` – executed after the package is installed
- `%preun` – executed before the package is uninstalled
- `%postun` – executed after the package is uninstalled

Most commonly, these scripts are used for packages which require a user account on the system to operate. The `%pre` script can be used to add the required account when installing the software, and the `%postun` script can be used to remove the no-longer-required account when uninstalling the software. When preparing a package for use on Red Hat systems which requires a user account, consult the file `/usr/share/doc/setup-*/uidgid`, which lists the UIDs and GIDs which are already used by software shipped with Red Hat Linux.

Sometimes, the actions which should be carried out in the `%pre`, `%post`, `%preun`, and `%postun` scripts are contingent upon the software which is installed on the system. For example, the Mailman mailing-list management software needs several email aliases to operated, so a package of it needs to add email aliases to the system in its `%post` script, and remove them from the system in its `%postun` script. Where those aliases get written is contingent upon the MTA installed on the system—Postfix uses the file `/etc/postfix/aliases` as its aliases database, while Sendmail uses the file `/etc/aliases` as its aliases database.

To deal with situations where the action to take is contingent upon the current system state, conditional shell scripting can be used. Alternately, RPM provides a trigger mechanism which can be used to list actions to carry out when other software is installed or uninstalled. These scripts are denoted by the tokens

`%triggerin -- `*`package`*

or

`%triggerun -- `*`package`*

The first token indicates a script to run when the software *`package`* is installed or upgraded, while the second token indicates a script to run when the software *`package`* is removed.

The Changelog Section

Used to record packaging changes

- List changes, and why made
- Provides a record of how the software is packaged

Tracking Packaging Changes

The final stanza of the RPM spec file is the `Changelog` stanza. This stanza always begins with the following token and is used to list any changes made to the package:

```
%changelog
```

Although the structure of this log is technically free-form, the following format is typically used by most RPM packages:

```
* date packager <packager’s email> version-release
- change made
- other change made
```

The date must be in the following format:

```
Wed Nov 20 2002
```

The current day in the required format can be obtained with the command:

```
date +"%a %b %d %Y"
```

Every time a new revision of the package is prepared, the packager simply adds a stanza similar to the above example to the beginning of the `Changelog` stanza.

Example Changelog Section from FC3 firefox RPM

The following example changelog comes from the Fedora Core v3 firefox RPM:

```
* Wed Mar 02 2005 Christopher Aillon
<caillon@redhat.com> 0:1.0.1-1.3.2

- Remerge firefox-1.0-pango-selection.patch

* Thu Feb 24 2005 Christopher Aillon
<caillon@redhat.com> 0:1.0.1-1.3.1

- Update to 1.0.1 fixing several security flaws.
- Mark some generated files as ghost (#136015)
- Add RPM version to the useragent
- BuildRequires pango-devel
- Enable pango rendering by default.
- Enable smooth scrolling by default
```

One convention that Red Hat uses it to include references to their bug tracking system bug Ids. In the example, there is reference to a bug about a RPM packing problem. The full details can be viewed here:

https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=136015

Putting It All Together....

Example spec file

- Package for the `less` application
- All comments added, though production spec files often contain comments as well

The less Spec

```
# Header Stanza begins here
#
# Short description
Summary: A text file browser similar to more, but better.
# the application name
Name: less
# the version of the application
Version: 378
# the packaging revision of this particular version
Release: 8
# this software is licensed under the GPL
License: GPL
# this software is an application used with text files
Group: Applications/Text
# this software source file; not the use of variables
Source: http://www.greenwoodsoftware.com/less/%{name}-%{version}.tar.gz
# a shell script needed for use with less
Source1: lesspipe.sh
# script to change the environment when less is used in Bourne-compatible shells
Source2: less.sh
# script to change the environment when less is used in C shells
Source3: less.csh
```

```
# Patches
Patch0: less-378-rh1.patch
Patch1: less-378+iso247-20030108.diff
Patch2: less-378-multibyte.patch
# this software can be downloaded from the following location
URL: http://www.greenwoodsoftware.com/less/
# temporary dir where the software should be compiled
Buildroot: %{_tmppath}/%{name}-root
# non-obvious software required to build the software
BuildRequires: ncurses-devel

# long description
%description
The less utility is a text file browser that resembles more, but has
more capabilities.  Less allows you to move backwards in the file as
well as forwards.  Since less doesn't have to read the entire input file
before it starts, less starts up more quickly than text editors (for
example, vi).

You should install less because it is a basic utility for viewing text
files, and you'll use it frequently.

# Prep Stanza begins here
#
%prep
# unpack the source and cd into the source directory
%setup -q
# apply the first patch
%patch0 -p1 -b .rh1
# apply the second patch
%patch1 -p1 -b .jp
# apply the third patch
%patch2 -p1 -b .multibyte
# perform other needed setup
chmod -R a+w *
```

```
# Build Stanza begins here
#
%build
# less uses autoconf, so ./configure it w/ appropriate options
%configure
# compile the software
make CC="gcc $RPM_OPT_FLAGS -D_GNU_SOURCE -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOU
RCE -D_FILE_OFFSET_BITS=64" datadir=%{_docdir}

# Install Stanza begins here
#
%install
# as sanity protection, make sure the Buildroot is empty
rm -rf $RPM_BUILD_ROOT
# install software into the Buildroot
%makeinstall
strip -R .comment $RPM_BUILD_ROOT/usr/bin/less
mkdir -p $RPM_BUILD_ROOT/etc/profile.d
install -c -m 755 %{SOURCE1} $RPM_BUILD_ROOT/usr/bin/
install -c -m 755 %{SOURCE2} $RPM_BUILD_ROOT/etc/profile.d
install -c -m 755 %{SOURCE3} $RPM_BUILD_ROOT/etc/profile.d

# define a clean-up script to run after the software in Buildroot is pkg'ed
%clean
# the actual script -- just delete all files within the Buildroot
rm -rf $RPM_BUILD_ROOT

# Files Stanza begins here
#
%files
# set perms and ownerships of packaged files
# the - indicates that the current permissions on the files should be used
%defattr(-,root,root)
# package all files within the $RPM_BUILD_ROOT/etc/profile.d directory
/etc/profile.d/*
# package all files within the $RPM_BUILD_ROOT/usr/bin directory
/usr/bin/*
# package all files within the $RPM_BUILD_ROOT/usr/share/man/man1 directory
%{_mandir}/man1/*
```

```
# Scripts Stanza begin here
#
# No Scripts in this RPM

# Changelog begins here
#
%changelog
# newest Changelog entry
* Tue Feb  4 2003 Tim Waugh <twaugh@redhat.com> 378-7
- Part of multibyte patch was missing; fixed.

# 2nd-newest Changelog entry
* Mon Feb  3 2003 Tim Waugh <twaugh@redhat.com> 378-6
- Fix underlining multibyte characters (bug #83377).

# oldest Changelog entry
* Thu Jan 30 2003 Karsten Hopp <karsten@redhat.de> 378-5
- removed older, unused patches
- add patch from Yukihiro Nakai to fix display of japanese text
  (#79977)
```

Considering Advanced Needs

In addition to the basic spec file features mentioned so far, more advanced needs sometimes arise. Commonly, a single application needs to be packaged into two or more package files. For example, the LessTif software provides several different things: a runtime library used by applications linked against the Motif widget set, programming headers and libraries which developers can use to produce applications, and a window manager. These are three distinctly separate applications, and different users will need different combinations of these applications. For this reason, RPM spec files can be written to generate sub-packages, two or more binary packages built from the same source code.

Another common need is to package binary files for which the source code is not available. For example, Adobe releases the Acrobat Reader for Linux, but until version 7 they only released it as an tarball of the binary executables only. RPMs can be prepared of this binary application, allowing users to manage the application using the RPM suite of tools. To prepare a spec file for a binary application, simply omit the Build stanza. In the %install stanza manually copy the files to the virtual root filesystem.

Packagers often want to prepare RPMs which are interactive, meaning that they ask questions of the user during install. RPM deliberately does *NOT* support creation of interactive packages to ensure that all package installation can be scripted. Be sure to design packages so that no interactivity is required to install the RPM.

Packagers often need to prepare packages for more than one distribution. Several potential complications should be considered to ensure that a single spec file can be used to compile RPMs suitable for multiple distributions:

- Different distributions predefine different macros. Most macros defined on Red Hat distributions are a “base line” which can be assumed to be defined on most major distributions, including SUSE distributions, so using just the macros defined on Red Hat systems aids cross-distribution compatibility
- Different distributions use different versions of RPM. Older Linux distributions use an old RPM release, RPM 3, while modern systems use a newer release, RPM 4. There are both behavioral and syntax difference between different RPM releases which must be counteracted.
- Different distributions sometimes install files in different locations. The Linux Standards Base (LSB) effort has largely eliminated this factor, but it can occasionally still occur.

Packagers often want to release packages which support multiple languages, so that users can see the package description and similar information in their native language. RPM provides a specspo mechanism which can be used to store translated messages for package files.

Building Packages

Modern RPM (version 4.x and higher)

- **`rpmbuild -b`**

Legacy RPM (version 3.x and lower)

- **`rpm -b`**

Common options

- **`-a`**

Targeting a Platform

Overriding Macros

Building Packages

Once a `spec` file has been prepared, the next step is to build software using this `spec` file. The basic command to build software on modern systems is `rpmbuild -b spec`, while with older systems the command is `rpm -b spec`. These commands accept any of several different arguments, depending on what part of the `spec` file should be processed:

- `-bp` – Carry out the Prep stanza of the spec file
- `-bc` – Carry out the Prep and Build stanzas of the spec file
- `-bi` – Carry out the Prep, Build, and Install stanzas of the spec file
- `-bl` – Verify the Files stanza of the spec file
- `-bb` – Build a binary RPM based on the spec file
- `-bs` – Build a source RPM based on the spec file
- `-ba` – Build both source and binary RPMs based on the spec file

The most common command to build an RPM from a spec file is:

```
rpmbuild -ba spec
```

Sometimes, the `--target` option is used to force the RPM software to compile software for a specific platform. For example, on 32-bit Intel-compatible systems running RHEL/FC, RPM software will, by default, produce RPMs which run on any 32-bit Intel-compatible computer but which are optimized for use on 80686-class machines. This option can be used to force RPM to produce executables which only run on 80686-class machines, or which are better optimized for the AMD Athlon, or which are optimized for 80386-class machines. The command:

```
rpmbuild -ba --target i686-redhat-linux spec
```

builds RPMs based on the *spec* which are only compatible with 80686-class machines, and which are marginally more optimized than the default optimizations RPM performs would offer.

When building packages, RPM software provides the capability to override or modify RPM macros from the command-line. This feature is commonly used by packagers to produce `spec` files which can be used to compile the same software with different options. For example, the Postfix `spec` file in RHEL/FC contains a series of macro definitions in the `Header` stanza which enable or disable support, at compilation time, for various features. The statement:

```
%define LDAP 0
```

is used to define the value of the LDAP variable to 0, and later commands in the `Build` stanza use this to compile Postfix without LDAP support. Overriding this variable by the command-line

```
rpmbuild -ba --define ‘LDAP 1’ postfix.spec
```

changes the value of the variable, and Postfix is compiled with LDAP support.

Digitally Signing Packages

Why?
- Provides authentication
- Confirms integrity

How?
- Create GPG key
- Modify macros
- Sign package
  rpmsign –addsign

Verifying Signatures
- **`rpmsign -k`**

Signing Built Packages

After producing binary and source RPMs from a spec file, the resulting RPMs can be digitally signed using GPG (or PGP). Signed packages offer two important advantages to end users who download them:

- They authenticate the package, assuring the user that the package comes from the vendor it is supposed to come from
- They guarantee the package's integrity, assuring the user that the package has not been modified since the packager signed it (though RPMs also provide this guarantee using other mechanisms)

Both of these features are important on today's Trojan Horse-riddled Internet.

To sign a package, you must first have a GPG key. If you do not already have one, you can generate one easily:

```
$ gpg --gen-key
```

Once you have created a key, modify your RPM macros to instruct RPM to use GPG to sign packages:

```
echo "%_signature gpg" >> ~/.rpmmacros
```

Also, specify which key RPM should use when signing packages:

```
echo "%_gpg_name email_address" >> ~/.rpmmacros
```

Also tell RPM where to locate your GPG keys:

```
echo "%_gpg_path $HOME/.gnupg" >> ~/.rpmmacros
```

Once the RPM software is configured to use GPG, the rpmsign command can be used to sign packages you have built:

```
$ rpmsign --addsign /path/to/the.rpm
```

You can export your public key:

```
$ gpg --export --armor > gpg-pub-key
```

You can then give this public key to users installing your RPMs, and they can use it to verify that your RPMs are signed by you. To do this, they must first import your GPG public key into their RPM GPG keychain as root:

```
# rpmsign --import /path/to/gpg-pub-key
```

Once the public key is imported, the signature of packages can be verified:

```
$ rpmsign -K package-1.0-1.i386.rpm
package-1.0-1.i386.rpm: (sha1) dsa sha1 md5 gpg OK
```

If the package verifies, then the user knows the signature of it is that of a vendor they trust.

Revising a Package

Install current SRPM

Drop new source tarball in SOURCES

Update spec file to use new source

Try Prep

- If any patch has error applying, visually inspect to see whats up.
  - Already applied in new package, or reason for patch addressed via another method == drop patch
  - If patch needed but won't apply == redo new patch

Revising Packages

In addition to producing new RPMs from scratch, another common task for packagers is to update existing packages, rebuilding the package as newer versions of the application it contains become available. Revising an existing source RPM requires a few simple steps:

1. Install the latest available SRPM for that application
2. Add the latest version of the source code for that application in the SOURCES directory
3. Modify the spec file for that application to use the new source code.
4. Try to complete the `Prep` section of the spec file:

```
$ rpmbuild -bp application.spec
```

When trying to complete the `Prep` stanza for the software, some patches may not apply to the new application source code. If a patch does not apply, inspect the patch and the application source code to see why it does not apply.

1. The patch may not apply correctly because it is no longer needed. In this case, simply remove the patch from the spec file.
2. The patch may still be needed, but might no longer apply do to other code changes in the application. In this case, create a new version of the patch which applies to the new source code.

Once the `Prep` stanza can be completed successfully, update the `Changelog` and package versioning information in the spec file, then build new packages:

```
$ rpmbuild -ba application.spec
```

Other Resources

RPM references
Add-on Tools
Sources of RPMs

Further Resources

A variety of additional resources are available which provide more information about RPM and preparing packages for RPM. The best resource is http://www.rpm.org which provides a wide variety of documentation about RPM, as well as the source code for RPM. This site even has a freely downloadable book, *Maximum RPM*. Although now somewhat dated, this book is still a good supplemental resource.

Many tools are available which can simplify the creation of RPMs. These include tools like spec file editing modes for popular text editors:

http://www.tihlde.hist.no/~stigb/rpm-spec-mode.el

http://pegasus.rutgers.edu/~elflord/vim/syntax/spec.vim

There are also a variety of spec file generation tools available:

http://rpmrebuild.sourceforge.net/

http://www.cpan.org/modules/by-module/RPM/RPM-Specfile-1.17.tar.gz

http://checkinstall.izto.org/

One difficulty with preparing packages for multiple releases is the need to have a separate build environment for every release. There are several applications which can be used to simplify the work of creating multiple build environments on the same machine, including:

http://thomas.apestaart.org/projects/mach/

http://www.solucorp.qc.ca/miscprj/s_context.hc

Several sites exist which provide a variety of high-quality pre-created RPMs, or which can help locate existing RPMs:

http://freshrpms.net

http://www.fedoraproject.org

http://www.rpmfind.net

http://www.GuruLabs.com/downloads.html

Lots of tools are available which ease the work in installing RPMs, including:

http://current.tigris.org

http://www.linux.duke.edu/projects/yum/

http://www.autorpm.org/

http://www.mat.univie.ac.at/~gerald/ftp/autoupdate/

http://apt4rpm.sourceforge.net/

Lab 12.1 - Creating RPMs

Estimated Time: 1.5 Hours

Objectives:

- Set up non-root RPM build environment
- Rebuild a binary package from a src.rpm
- Update an existing source RPM with the latest software version
- Create a spec file from scratch for an unpackaged software application
- Revise packages to correct packaging errors
- Create multiple RPMs from a single source RPM
- Create a GPG key pair
- Sign and verify your RPMs

Lab 12 RPM CREATION

Task 1

- Set up non-root RPM build environment
- Rebuild a binary package from a `src.rpm`

This lab will primarily be done while logged in as a non-root user. Occasionally you will need root privileges. The lab will direct you to use the `su` command to become root when required. Start by logging in —or obtaining a shell as a non-root user.

This lab has been validated on the following distributions:

Red Hat Enterprise Linux v4 aka RHEL4

Fedora Core v3 aka FC3

SUSE Linux Enterprise Server v9 aka SLES9

SUSE Linux Professional v9.2 aka SL92

On RHEL/FC an "Everything" install is assumed to have been performed and on SLES/SL an install with all "package groups" selected is assumed to have been installed.

Additionally this lab requires several packages to downloaded to the /labfiles/ directory before starting. Download the need files by running the following commands:

```
# mkdir /labfiles/
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/ltris-1.0.4-2.src.rpm
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/lbreakout2-2.4.1.tar.gz
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/lbreakout2.spec-example
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/lbreakout2.spec-example2
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/nmap-3.70-1.src.rpm
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/nmap-3.81.tar.bz2
# wget http://www.gurulabs.com/GURULABS-RPM-LAB/template.spec
```

1. The creation of RPMs requires the use of several developer tools. These include both the standard development tools such as the GNU Compiler Collection, GNU make, and related packages, as well as specific RPM building

tools. Verify that you have the following standard development tool packages installed:

```
$ rpm -q gcc
gcc-3.2.2-5
$ rpm -q gcc-c++
rpm-c++-3.2.2-5
$ rpm -q make
make-3.79.1-17
$ rpm -q bison
bison-1.35-6
$ rpm -q binutils
binutils-2.13.90.0.18-9
```

- The exact version returned on this package and others will vary depending on what version of RHEL/FC or SLES/SL Linux you are using. What matters is that the packages are installed, not the exact versions which are installed.

Note that a full development environment will have many more related packages installed as well. However, if the above packages are installed, it is a strong indication that the other needed packages are also installed.

Now, verify that you have the proper RPM building utilities installed. On RHEL/FC the utilites are part of the `rpm-build` RPM package which can be optionally installed. On SLES/SL the utilties are part of the `rpm` package which is always installed. Run the following command only on RHEL/FC:

```
[RHEL/FC]$ rpm -q rpm-build
rpm-build-4.3.2-21
```

Also determine which RPM version you have installed:

```
$ rpm -q rpm
rpm-4.3.2-21
```

Different RPM versions support different features and are used differently, so it is important to determine which version of RPM is being used.

2. Remove any installed packages that will conflict with packages built manually in upcoming steps:

```
[SLES/SL]# rpm -e ltris nmap lbreakout
[RHEL/FC]# rpm -e nmap nmap-frontend
```

3. Out of the box on RHEL/FC, the system RPM build directory structure, `/usr/src/redhat/`, is only writable by the root user. On SLES/SL the RPM

build directory structure is located at `/usr/src/packages/` and is writable by everyone.

The recommended practice is NOT to build RPMs as the root user, create a RPM build directory tree in your non-root user's home directory for isolation from other users.

Make sure your present working directory is your home directory, and then create the RPM build directory structure:

```
$ cd
$ mkdir -p rpmbuild/{SOURCES,SPECS,BUILD,SRPMS,RPMS}
$ mkdir rpmbuild/RPMS/{i386,i586,i686}
```

4. When building RPMs using the `rpmbuild` command, the RPM software must locate the RPM build directory structure. To do this, it reads the value of the `%_topdir` RPM macro. The macro container files that macros are read from start with the RPM default `/usr/lib/rpm/macros` file, then the local system override file `/etc/rpm/macros` is consulted (if it exists), and finally, the per-user macro file is read, `~/.rpmmacros` (if it exists). Any macro found in the per-user macro file will override the same macro in the local system override file, and any macro found in the local system override file will override macros from the RPM default file.

   Create a `~/.rpmmacros` file and set the `%_topdir` RPM macro to point to the directory structure you created in the previous step. Also, set the `%debug_package` macro to turn off the automatic creation of the debuginfo RPM.

   Use your favorite text editor and **create** the file `~/.rpmmacros` with the following contents:

```
%_topdir                %(echo $HOME)/rpmbuild
%debug_package          %{nil}
```

- Red Hat Linux 9 and later RHEL/FC releases generate debuginfo packages by default. These debuginfo RPMs can be installed when installing an application to get more capability to debug the application. They are not typically necessary, and are often not desired, as they are quite large.
- This %debug_package macro is only needed on systems with RPM 4.2 or newer.

5. Test your non-root RPM building environment by rebuilding a binary RPM from an existing source RPM.

   The LTris game is a popular Tetris clone. Verify that your RPM build environment works by creating a binary LTris RPM from the source RPM:

```
$ rpmbuild --rebuild /labfiles/ltris-1.0.4-2.src.rpm
Installing /labfiles/ltris-1.0.4-2.src.rpm
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.2652
```

- This is something that is commonly done when obtaining RPMs off of the Internet. Unless you know for sure that an Internet binary RPM was built specifically against your version of Red Hat Linux, it is preferable to download the `src.rpm` of the package and rebuild a binary RPM with the application linked against the exact versions of the libraries provided by your system, and not other, hopefully compatible libraries, or even library versions that are not on your system.

```
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ LANG=C
+ export LANG
+ cd /home/guru/rpmbuild/BUILD
+ rm -rf ltris-1.0.4
+ /usr/bin/gzip -dc /home/guru/rpmbuild/SOURCES/ltris-
1.0.4.tar.gz
+ tar -xf -
+ STATUS=0
...Output Omitted...
```

After a few minutes have elapsed, look for the **Wrote:** line near the end of the output:

```
Requires: SDL >= 1.1.4 config(ltris) = 1.0.4-2 libSDL-1.2.so.0 libSDL_mixer-1.2.so.0 libc.so.6
libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1) libc.so.6(GLIBC_2.1.3) libc.so.6(GLIBC_2.3) libm.so.6
libpthread.so.0
Checking for unpackaged file(s): /usr/lib/rpm/check-files
/var/tmp/ltris-root
Wrote: /home/guru/rpmbuild/RPMS/i386/ltris-1.0.4-2.i386.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.9014
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd ltris-1.0.4
+ rm -rf /var/tmp/ltris-root
+ exit 0
```

- This is the line you are looking for.

6. Install the binary RPM which was created in the previous step. The path to the binary RPM was displayed on the **Wrote:** line. Remember, you must switch to the root user to install binary RPMs to the system.

```
$ su -
Password:
# rpm -Uvh /home/guru/rpmbuild/RPMS/i386/ltris-1.0.4-2.i386.rpm
# exit
$ ltris
```

- Test the LTris program as a non-root user. You must be working in the GUI environment for this to work.

Task 2

- Update an existing source RPM with the latest software version

Linux distributions are a collection of software packages that have been tested and validated together. The Linux distributions are released periodically, typically on a 6 month interval, while the individual packages are developed independently and released on separate release schedules.

Most Linux distribution vendors release updated packages only in two circumstances, to fix security vulnerabilities, and to fix major bugs that impact many people. In all other cases, Linux distribution vendors normally wait until the next version of their distribution to upgraded to any other updated versions. Sometimes there will be a newer version of a package available —with features that you need— than what came with the Linux distribution version you are using. However, this update does not meet the requirements to be released as an official update for the Linux distribution. In such circumstances, you can create your own RPM update for the application. This is desirable over just downloading, compiling and installing the package to `/usr/local/` in a un-tracked state.

1. To update an RPM that shipped with your distribution, start by installing an existing source RPM, such as one provided with your distribution. The distribution source RPMs for Nmap have been placed in the `/labfiles/` directory.

```
$ rpm -Uvh /labfiles/nmap-3.70-1.src.rpm
warning: /labfiles/nmap-3.70-1.src.rpm: V3 DSA signature: NOKEY, key ID 4f2a6fd2
   1:nmap          ########################################### [100%]
```

   Note that this may not be the actual source RPM for for Nmap for your specific Linux distribution but for the sake of consistency in the lab, will be used as such.

   This places the pristine source and any patches for this package into `the ~/rpmbuild/SOURCES/` directory, and the spec file for this package into the `~/rpmbuild/SPECS/` directory.

2. Copy the newer pristine tarball of Nmap source code into your `~/rpmbuild/SOURCES/` directory. The tarball has been placed in `/labfiles/` for you.

```
$ cp /labfiles/nmap-3.81.tar.bz2 ~/rpmbuild/SOURCES/
```

List the contents of the `~rpmbuild/SOURCES/` directory. You should see the tarball you just copied, plus the files provided by the source RPM you installed in the previous step.

```
$ ls -al ~/rpmbuild/SOURCES/
total 2574
drwxrwxr-x    2 guru    guru        1024 Apr 28 12:03 .
drwxrwxr-x    7 guru    guru        1024 Apr 26 23:45 ..
-rw-rw-r--    1 guru    guru         836 Sep  9  2004 inet_aton.patch
-rw-rw-r--    1 guru    guru         324 Sep  9  2004 makefile.patch
-rw-rw-r--    1 guru    guru      922293 Sep 13  2004 nmap-3.00.tar.bz2
-rw-r--r--    1 guru    guru      871101 Mar 22 11:29 nmap-3.81.tar.bz2
$
```

3. The next step requires modifying the spec file. This can involve two or more edits, depending on the differences between the older version and the current version. The two major changes which are always necessary include updating the version and release numbers (in one or more locations), and adding a **%changelog** entry. Additional changes that may be required are forward-porting patches to the current version, or removing patches that are no longer required. Finally, sometimes the newer package changes which files are to be installed, requiring adjustment of the **%files** section.

   Open the Nmap spec file with your favorite text editor, and display the headers of the file **~/rpmbuild/SPECS/nmap.spec**. The following example uses the nmap-3.70-1 spec file, with line numbers added for reference; small differences may exist if you are starting from a different spec file.

```
1  %{!?withgtk1:%define withgtk1 1}
2
3  Summary: Network exploration tool and security scanner
4  Name: nmap
5  Version: 3.70
6  Release: 1
7  License: GPL
8  Group: Applications/System
9  Source0: http://download.insecure.org/nmap/dist/%{name}-%{version}.tar.bz2
10 #Source1: nmapfe.desktop
```

```
11 Patch0: inet_aton.patch
12 Patch1: makefile.patch
13 URL: http://www.insecure.org/nmap/
14 BuildRoot: %{_tmppath}/%{name}-root
15 Epoch: 2
16 BuildRequires: openssl-devel, gtk+-devel, pcre-devel, libpcap
```

Change lines 5 to match the version of the newer Nmap, version **3.81**.

Verify that on line 6 the release number is set to **1** since this will be the first RPM release of version 3.81.

(SLES/SL only) Change part of line 16 from `gtk+-devel` to **`gtk-devel`** since on SLES/SL this package has the needed dependencies.

(SLES/SL only) On lines 30 and 31 make a similar changing references to `gtk+` and `gtk+-devel` to **`gtk`** and **`gtk-devel`**.

Move to the line starting with `%changelog`. Add a new entry by adding the following two lines —changing the date to today's date— followed by a blank line right beneath it, for example:

```
* Tue Mar 22 2005 Firstname Lastname <your@emailaddr.com> - 2:3.81-1
- ver 3.81
```

The number at the end of the first line is a common, yet not required convention, for including the epoch, version, and release info linked to each changelog entry.

Finally, **save** and **exit** the file.

4. The changes that have been made may be enough. The next step is to verify that all current patches still apply to the newer source code. To find out, change into the spec file directory, and execute the Prep stage of the build:

```
$ cd ~rpmbuild/SPECS/
$ rpmbuild -bp nmap.spec
...Output Omitted...
```

If everything works out, it should finish with no error messages or prompting for user input .

If there were problems, be sure to read the next two steps for examples problems and their solutions. If there were no problems, you may skip to step 8 if you wish.

5. You will know that the initial spec file changes are not enough if you get an error message. The most likely problem you might encounter is that patches which were previously applied to the older source code no longer apply against the newer source code. This problem might be caused by any of three different reasons:

- Perhaps the old patch has been adopted and applied by the software maintainer, and as such should be dropped from the spec file.
- Perhaps the patch is no longer needed since the software maintainer addressed the problem using some other mechanism, and as such should be dropped from the spec file.
- Perhaps the old patch is still needed, but will not apply to the new source. In this case, a new patch against the current software should be generated to replace the old patch in the spec file.

Which of these three situations is the cause of any errors, and what action to take, requires intelligent analysis of the patches and source code of the software. For example, the Nmap 3.00-4 spec file includes a patch to Nmap, `nmap-3.00-nowarn.patch`. When updating to 3.27 and leaving the patch in place in the spec file, the following error is produced when trying to prepare the new RPMs:

```
$ rpmbuild -bp nmap.spec
[snip]
+ echo 'Patch #2 (nmap-3.00-nowarn.patch):'
Patch #2 (nmap-3.00-nowarn.patch):
+ patch -p1 -b --suffix .nowarn -s
The text leading up to this was:
--------------------------
|--- nmap-3.00/tcpip.c.nowarn   2003-01-09 15:46:53.000000000 +0100
|+++ nmap-3.00/tcpip.c  2003-01-09 15:49:34.000000000 +0100
--------------------------
File to patch:
```

At this point the build process stops, with the patch command waiting for input. Press ***<CTRL-C>*** to abort.

Examining the directory listing of the `~rpmbuild/BUILD/nmap-3.27/` directory reveals that it is likely that the file `tcpip.c` in the 3.00 source has been renamed to `tcpip.cc` in the 3.27 source. This change prevents the patch from applying.

However, more inspection is needed. Examining the patch file itself, `~rpmbuild/SOURCE/nmap-3.00-nowarn.patch`, reveals that the patch is changing the following lines of code:

```
printf("Data portion:\n");
while(i < tot_len)  printf("%2X%c", data[i], (++i%16)? ' ' : '\n');
```

To:

```
printf("Data portion:\n");
while(i < tot_len)  {
       printf("%2X%c", data[i], ((i+1)%16)? ' ' : '\n');
       i++;
}
```

in two spots in the `tcpip.c` file. It is important to note that you **DO NOT** have to understand the C code to analyze if the patch has already been applied. Opening the file `~rpmbuild/BUILD/nmap-3.27/tcpip.cc` and searching for the string **Data portion**, we find the lines (in two different locations):

```
printf("Data portion:\n");
while(i < tot_len)  {
  printf("%2X%c", data[i], ((i+1) %16)? ' ' : '\n');
  i++;
}
```

So, it appears as if the official 3.27 Nmap has integrated the patch that was being applied in the RPM building process to the 3.00 Nmap. So, the proper course of action is to remove references to this patch in our new spec file.

6. **Open** the file `~rpmbuild/SPECS/nmap.spec` using a text editor and **remove** the two lines (prefixed here by their line numbers):

```
12 Patch2: nmap-3.00-nowarn.patch
36 %patch2 -p 1 -b .nowarn
```

Save the file and exit the text editor, and re-run the Prep by running the following command:

```
$ rpmbuild -bp nmap.spec
```

Repeat this process until all patching failures are resolved.

7. Once the spec file successfully patches the software without any errors, run the following command to compile packages of the Nmap software:

   ```
   $ rpmbuild -ba nmap.spec
   ```

8. Another common problem that can prevent the successful building of the new RPMs are difference in files installed to the virtual root filesystem. If any files are installed in the virtual root filesystem, but are not referenced in the **`%files`** section, RPM 4.1 and higher releases will display an error and abort the build process. You will encounter this problem in subsequent lab sequences, assuming you are using RPM version 4.1 or newer.

   When you encounter those sorts of problems, you will have two possible solutions:

- If the file is needed, add an entry to the `%files` section so that the file gets packaged.
- If the file is not needed, use commands in the Install stanza to delete the unwanted file from the virtual root filesystem.

On RHEL4/FC3 you will see an example of the 2nd scenario with the following error message will be displayed:

```
error: Installed (but unpackaged) file(s) found:
   /usr/share/applications/nmapfe.desktop


RPM build errors:
    Installed (but unpackaged) file(s) found:
   /usr/share/applications/nmapfe.desktop
```

This refers to a menu desktop file. Open the spec file with a text editor, and examine the `%install` section. You will notice the following lines:

```
# remove unused files
rm -f $RPM_BUILD_ROOT/usr/share/gnome/apps/Utilities/nmapfe.desktop
```

The intent is to delete the menu desktop file, but in this newer version it is being installed into a different location so this deletion attempt is failing. **Correct** the line to point to the new location by editing the line to read:

```
rm -f $RPM_BUILD_ROOT/usr/share/applications/nmapfe.desktop
```

Re-run the following command to compile packages of the Nmap software:

```
$ rpmbuild -ba nmap.spec
```

This should complete with out error.

On SLES9/SL92 you will encounter two problems. The first build attempt should end with the following error:

```
RPM build errors:
    File not found: /var/tmp/nmap-root/usr/share/nmap
```

This indicates that a required directory was not found in the virtual root. Under the `/usr/share/nmap/` directory the file `nmap.dtd` should exist. Do a search to find where it is is located inside of the virtual root:

```
$ find /var/tmp/nmap-root -name nmap.dtd
/var/tmp/nmap-root/var/tmp/nmap-root/usr/share/nmap/nmap.dtd
```

If you examine the path you'll notice that there is a `var/tmp/nmap-root/` inside of `/var/tmp/nmap-root/`. This is likely a error that occurs during the installation phase, open the spec file and locate the line `make install` line:

```
%makeinstall nmapdatadir=$RPM_BUILD_ROOT%{_datadir}/nmap
```

Edit the line and remove the extra parameter as the stock `%makeinstall` macro should be sufficient. Change the line to read:

```
%makeinstall
```

Re-run the following command to compile packages of the Nmap software:

```
$ rpmbuild -ba nmap.spec
```

You will see an example of the 2nd scenario with the following error message will be displayed:

```
error: Installed (but unpackaged) file(s) found:
   /usr/share/applications/nmapfe.desktop


RPM build errors:
    Installed (but unpackaged) file(s) found:
   /usr/share/applications/nmapfe.desktop
```

This refers to a menu desktop file. Open the spec file with a text editor, and examine the `%install` section. You will notice the following lines:

```
# remove unused files
rm -f $RPM_BUILD_ROOT/usr/share/gnome/apps/Utilities/nmapfe.desktop
```

The intent is to delete the menu desktop file, but in this newer version it is being installed into a different location so this deletion attempt is failing. Correct the line to point to the new location by editing the line to read:

```
rm -f $RPM_BUILD_ROOT/usr/share/applications/nmapfe.desktop
```

Re-run the following command to compile packages of the Nmap software:

```
$ rpmbuild -ba nmap.spec
```

This should complete with out error.

9. Install the new updated Nmap RPMs as root:

```
$ su -
Password:
[RHEL4/FC3 ]# rpm -Uvh /home/guru/rpmbuild/RPMS/i386/nmap-*
[SLES9/SL92]# rpm -Uvh /home/guru/rpmbuild/RPMS/i586/nmap-*
Preparing...                ########################################### [100%]
   1:nmap                   ########################################### [ 50%]
   2:nmap-frontend          ########################################### [100%]
```

10. Newer Nmap versions supports version scans that report the version of services. Try using this feature during a confirmation run to see that your new Nmap RPM is working properly.

```
# nmap -sV 127.0.0.1

Starting nmap 3.81 ( http://www.insecure.org/nmap/ ) at 2005-03-25 18:39 MST
Interesting ports on localhost (127.0.0.1):
(The 1659 ports scanned but not shown below are in state: closed)
PORT    STATE SERVICE VERSION
22/tcp  open  ssh     OpenSSH 3.9p1 (protocol 1.99)
25/tcp  open  smtp    Postfix smtpd
111/tcp open  rpcbind 2 (rpc #100000)
631/tcp open  ipp     CUPS 1.1

Nmap finished: 1 IP address (1 host up) scanned in 5.379 seconds
```

An older version of NMAP without the version scanning feature would have returned the error:

```
Illegal Argument to -P, use -P0, -PI, -PB, -PM, -PP, -PT, or -PT80 (or whatever number you want for
the TCP probe destination port)
QUITTING!
```

Task 3

- Create a spec file from scratch for an unpackaged software application.
- Revise packages to correct packaging errors.
- Create multiple RPMs from a single source RPM.

LBreakout2, http://lgames.sourceforge.net/index.php?project=LBreakout2, is a popular Breakout-style game for Linux. Unfortunately, it is not provided in RPM format, though the source code is readily available from the http://lgames.sourceforge.net/ home page. Being a devotee of all things Breakout and Breakout-related, you have decided to prepare an RPM of LBreakout2 for your system.

This lab task requires that following packages are installed so that LBreakout2 can be compiled:

```
SDL-devel
libpng-devel
```

The RPM packages have the same name on RHEL/FC and SLES/SL. Verify that the packages are installed, and if not, install them now.

1. When creating an RPM from scratch, most packagers start with a skeleton spec file which lists common lines used in most spec files. Through a trial-and-error process, this template file can be modified and tested until it produces a good set of RPMs. A template spec file has been provided in your **/labfiles** directory; begin by copying this spec file for use with LBreakout2:

   ```
   $ cp /labfiles/template.spec ~/rpmbuild/SPECS/lbreakout2.spec
   ```

2. A copy of the source code for LBreakout2 has been provided for you. Copy this source code into your SOURCES directory:

   ```
   $ cp /labfiles/lbreakout2-2.4.1.tar.gz ~/rpmbuild/SOURCES
   ```

3. Once the source code for an application is in place and a spec file exists, the next task is to modify the spec to compile your software. Open your spec file for editing:

   ```
   $ cd rpmbuild/SPECS/
   ```

```
$ vim lbreakout2.spec
```

4. Complete the Header stanza. At a minimum, you will need Summary, Name, Version, Release, Epoch, License, Group, Source, URL, and BuildRoot directives. Try creating these yourself, then compare your solution with the following possible solution:

```
Summary: Breakout clone
Name: lbreakout2
Version: 2.4.1
Release: 1
Epoch: 0
License: GPL
Group: Amusements/Games
Source0: http://ftp1.sourceforge.net/lgames/%{name}-%{version}.tar.gz
URL: http://lgames.sourceforge.net/index.php?project=LBreakout2
BuildRoot: %{_tmppath}/%{name}-root
```

5. The Header stanza also requires a **%description** directive which provides users with a fuller description of the purpose of the packaged application. Create your own **%description** block, similar to the following:

```
%description
The polished successor to LBreakout offers you a new challenge in more than 50 levels with loads of
new bonuses (goldshower, joker, explosive balls, bonus magnet ...), maluses (chaos, darkness, weak
balls, malus magnet ...) and special bricks (growing bricks, explosive bricks, regenerative bricks,
indestructible bricks, chaotic bricks).

And if you're through with all the levels you can create complete new level sets with the integrated
easy-to-use level editor!
```

6. Save your spec file and quit the editor. At this point, you can test your spec file to be certain that your Header stanza works:

```
$ rpmbuild -bp lbreakout2.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.39167
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
```

• RPM tells you it is executing the %prep section of the spec file.

```
+ LANG=C
+ export LANG
+ cd /home/guru/rpmbuild/BUILD
+ rm -rf lbreakout2-2.4.1
+ /usr/bin/gzip -dc /home/guru/rpmbuild/SOURCES/lbreakout2-2.4.1.tar.gz
+ tar -xf -
+ STATUS=0
+ '[' 0 -ne 0 ']'
+ cd lbreakout2-2.4.1
++ /usr/bin/id -u
+ '[' 50016 = 0 ']'
++ /usr/bin/id -u
+ '[' 50016 = 0 ']'
+ /bin/chmod -Rf a+rX,g-w,o-w .
+ exit 0
$
```

- RPM indicates that the %prep section was processed successfully.

7. The template spec file you used as a starting place has a commonly used Build stanza already created. Try compiling your software to see if this default Build stanza will work:

```
$ rpmbuild -bc lbreakout2.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.62876
...Output Omitted...
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.81466
...Output Omitted...
gcc  -O2 -g -pipe -march=i386 -mcpu=i686 -Wall -
I/usr/include/SDL -D_REENTRANT -o lbreakout2  credit.o
shine.o extras.o balls.o shrapnells.o shots.o event.o pad-
dle.o frame.o misc.o bricks.o difficulty.o player.o game.o
file.o levels.o config.o item.o menu.o manager.o value.o
chart.o editor.o help.o hint.o theme.o client.o
client_recv.o client_data.o client_game.o client_handlers.o
comm.o display.o main.o -lSDL_mixer ../common/libcommon.a
../gui/libGui.a -lpng -lz -lm  -L/usr/lib -Wl,-
rpath,/usr/lib -lSDL -lpthread
make[3]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
```

- First, the %prep stanza is processed
- After the software is prepared, the %build stanza compiles it
- This line shows the step of actually linking together the compiled objects to produce the `lbreakout2` executable.

```
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
Making all in docs
make[2]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[2]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
+ exit 0
$
```

- This exit 0 indicates that the %build completed successfully.

8. So far, it looks like the default Build stanza should work. Next, test out the commonly used Install stanza which was provided in the template spec file:

   On RHEL/FC run the following command and examine the output produced, on SLES/SL skip to step 18. on page 62 (unless you want to read about the types of problems that can occur).

```
$ rpmbuild -bi lbreakout2.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.89196
...Output Omitted...
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.89196
...Output Omitted...
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.70963
...Output Omitted...
Making install in client
make[1]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
Making install in gfx
make[2]: Entering directory
`/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx'
Making install in AbsoluteB
make[3]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx/AbsoluteB'
make[4]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx/AbsoluteB'
make[4]: Nothing to be done for `install-exec-am'.
/bin/sh ../../../mkinstalldirs /usr/share/games/lbreakout2/gfx/AbsoluteB
mkdir /usr/share/games/lbreakout2
mkdir: cannot create directory `/usr/share/games/lbreakout2': Permission denied
mkdir /usr/share/games/lbreakout2/gfx
```

- First, the %prep
- Next, the %build
- Next, the %install

```
mkdir: cannot create directory `/usr/share/games/lbreakout2/gfx': No such file or directory
mkdir /usr/share/games/lbreakout2/gfx/AbsoluteB
mkdir: cannot create directory `/usr/share/games/lbreakout2/gfx/AbsoluteB': No such file or directory
make[4]: *** [install-data-local] Error 1
make[4]: Leaving directory
`/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx/AbsoluteB'
make[3]: *** [install-am] Error 2
make[3]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx/AbsoluteB'
make[2]: *** [install-recursive] Error 1
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx'
make[1]: *** [install-recursive] Error 1
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
make: *** [install-recursive] Error 1
error: Bad exit status from /var/tmp/rpm-tmp.70963 (%install)


RPM build errors:
    Bad exit status from /var/tmp/rpm-tmp.70963 (%install)
$
```

- RPM indicates that the %install script did NOT complete successfully.

9. The default **%install** script will NOT work for this software. So, try to figure out why. Looking at the spec file, you'll see that the current **%build** script does:

```
%install
rm -rf %{buildroot}
%makeinstall
```

The problem here is that **%makeinstall** is not working. Looking through the output from the **%install**, you see that the **%makeinstall** did:

```
make prefix=/var/tmp/lbreakout2-root/usr
exec_prefix=/var/tmp/lbreakout2-root/usr
bindir=/var/tmp/lbreakout2-root/usr/bin
sbindir=/var/tmp/lbreakout2-root/usr/sbin syscon-
fdir=/var/tmp/lbreakout2-root/etc data-
dir=/var/tmp/lbreakout2-root/usr/share
includedir=/var/tmp/lbreakout2-root/usr/include lib-
dir=/var/tmp/lbreakout2-root/usr/lib libex-
```

```
ecdir=/var/tmp/lbreakout2-root/usr/libexec
localstatedir=/var/tmp/lbreakout2-root/var sharedstate-
dir=/var/tmp/lbreakout2-root/usr/com
mandir=/var/tmp/lbreakout2-root/usr/share/man info-
dir=/var/tmp/lbreakout2-root/usr/share/info install
```

That is, it ran the `make install` command with options intented to get it to install software to a virtual directory structure under the `/var/tmp/lbreakout2-root/` directory rather than under `/`. However, as the error indicates, this failed:

```
make[4]: Entering directory
`/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gfx/AbsoluteB'
make[4]: Nothing to be done for `install-exec-am'.
/bin/sh ../../../mkinstalldirs /usr/share/games/lbreakout2/gfx/AbsoluteB
mkdir /usr/share/games/lbreakout2
mkdir: cannot create directory
`/usr/share/games/lbreakout2': Permission denied
```

- This should have been /var/tmp/lbreakout2-root/usr/share/games/lbreakout2.

For some reason, the **`%makeinstall`** macro is not capable of getting this software to install into the Buildroot. These sorts of problems are quite common-place, and correcting them typically requires either a modification of the spec file, an addition of a patch to modify the `Makefiles` which specify how the software gets compiled and installed, or some combination of both spec modifications and code patches.

10. To get this software packaged, you must figure out why it cannot currently be installed into the Buildroot. So far, you know that executing the `make install` command in the client subdirectory of the source code fails because it tries to install software to `/usr/share/games/lbreakout2/` rather than `/var/tmp/lbreakout2-root/usr/share/games/lbreakout2/`. Looking at the `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/Makefile`, you'll see that it does:

```
...
hi_inst_flag = -DHI_DIR=\"/var/lib/games\"
inst_dir = /usr/share/games/lbreakout2
inst_flag = -DSRC_DIR=\"/usr/share/games/lbreakout2\"
...
```

The problem here is this middle line, line 81. If this `Makefile` were written correctly, that line would instead look something like:

```
inst_dir = $(datadir)/games/lbreakout2
```

where it uses a shell variable which can be overridden to specify where the software gets installed.

11. Fixing this problem can be done in a couple of different ways:

- You could create a patch for the `Makefile` and apply it during the **`%prep`** section. This is the "correct" solution, but is actually quite difficult, since the Makefiles for this program, like many other Linux applications, are automatically generated during the **`%build`** process by the autoconf software suite.
- You could add a script as the first line of the **`%install`** stanza which edits the incorrect Makefiles. This is a little bit easier to do, but is a less "correct" solution.

In many cases, though, a third option will also be available: you might simply be able to modify the incorrect `Makefile` variable by passing the correct value to the **`%makeinstall`** macro. If possible, that solution will be the simplest. In this case, the value of the inst_dir variable is incorrect, so modify the `spec` file to correct that variable. In the **`%install`** stanza, change the **`%makeinstall`** line to:

```
%makeinstall \
    inst_dir=${RPM_BUILD_ROOT}/usr/share/games/lbreakout2
```

- This line is too long to fit on a single 80-column line on the screen. For readability, it can be typed on two lines, as is done here, with a back-slash (`\`) at the end of the first line. This back-slash indicates to `rpm` that this line continues on to a subsequent line.

12. Once you have made this change, save the `spec` file and then try once again to compile the software and install it to a virtual root directory:

```
$ rpmbuild -bi lbreakout2.spec
...Output Omitted...
Making install in gui_theme
make[2]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gui_theme'
make[3]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/gui_theme'
make[3]: Nothing to be done for `install-exec-am'.
/bin/sh ../../mkinstalldirs /var/tmp/lbreakout2-root/usr/share/games/lbreakout2/gui_theme
mkdir /var/tmp/lbreakout2-
root/usr/share/games/lbreakout2/gui_theme
...Output Omitted...
```

- This solved the problem -- the directory is now being created in the virtual directory tree.

```
if ! test -f /var/lib/games/lbreakout2.hscr; then \
  /usr/bin/install -c -m 644 -m 666 empty.hscr /var/lib/games/lbreakout2.hscr; \
fi;
/usr/bin/install: cannot create regular file `/var/lib/games/lbreakout2.hscr': Permission denied
make[3]: *** [install-data-local] Error 1
make[3]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
make[2]: *** [install-am] Error 2
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
make[1]: *** [install-recursive] Error 1
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
make: *** [install-recursive] Error 1
error: Bad exit status from /var/tmp/rpm-tmp.72065 (%install)


RPM build errors:
    Bad exit status from /var/tmp/rpm-tmp.72065 (%install)
$
```

13. You've solved the first problem, but this revealed a new problem to solve. Here, the `make install` command is doing:

```
make[3]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client'
...
if ! test -f /var/lib/games/lbreakout2.hscr; then \
  /usr/bin/install -c -m 644 -m 666 empty.hscr
/var/lib/games/lbreakout2.hscr; \
fi;
/usr/bin/install: cannot create regular file
`/var/lib/games/lbreakout2.hscr': Permission denied
make[3]: *** [install-data-local] Error 1
```

This is a similar problem to the previous one. If you search through the `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/client/Makefile` file for the string **/var/lib/games**, you'll see that the `Makefile` does:

```
...
doc_dir = /usr/doc
hi_dir = /var/lib/games
hi_inst_flag = -DHI_DIR=\"/var/lib/games\"
```

```
...
```

The problem here is this middle line, line 79. If this `Makefile` were written correctly, that line would instead look something like:

```
hi_dir = $(localstatedir)/lib/games
```

where it uses a shell variable which can be overridden to specify where the software gets installed.

14. As with the previous problem, this problem can be fixed by patching the Makefiles, modifying the Makefiles within the **`%install`** script, or by passing a variable to the **`%makeinstall`** macro. Edit the `spec` file again, and modify the **`%makeinstall`** macro so that it reads:

```
%makeinstall \
     inst_dir=${RPM_BUILD_ROOT}/usr/share/games/lbreakout2 \
     hi_dir=${RPM_BUILD_ROOT}/var/lib/games
```

15. Once you have made this change, save the spec file and then try once again to compile the software and install it to a virtual root directory:

```
$ rpmbuild -bi lbreakout2.spec
...Output Omitted...
if ! test -f /var/tmp/lbreakout2-
root/var/lib/games/lbreakout2.hscr; then \
  /usr/bin/install -c -m 644 -m 666 empty.hscr
/var/tmp/lbreakout2-root/var/lib/games/lbreakout2.hscr; \
fi;
...Output Omitted...
Making install in docs
make[1]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[2]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[2]: Nothing to be done for `install-exec-am'.
/bin/sh ../mkinstalldirs /usr/doc/lbreakout2
mkdir /usr/doc
mkdir: cannot create directory `/usr/doc': Permission denied
mkdir /usr/doc/lbreakout2
mkdir: cannot create directory `/usr/doc/lbreakout2': No such file or directory
make[2]: *** [install-data-local] Error 1
```

- This solved the problem -- the directory is now being created in the virtual directory tree.

```
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[1]: *** [install-am] Error 2
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make: *** [install-recursive] Error 1
error: Bad exit status from /var/tmp/rpm-tmp.33062 (%install)


RPM build errors:
    Bad exit status from /var/tmp/rpm-tmp.33062 (%install)
$
```

16. You've solved the second problem, but now you've got a new problem to solve. Here, the make install command is doing:

```
make[1]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
...
/bin/sh ../mkinstalldirs /usr/doc/lbreakout2
mkdir /usr/doc
mkdir: cannot create directory `/usr/doc': Permission denied
...
```

If you look at the `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs/Makefile` file for the string /usr/doc, you'll find that it does:

```
...
audio_flag = -DAUDIO_ENABLED
doc_dir = /usr/doc
hi_dir = /var/lib/games
...
```

17. As with the other problems, the trouble here is this middle line, line 78. It needs to be corrected to install to the virtual directory tree. This line also has another flaw. On RHEL/FC systems, all documentation files should be put in the directory `/usr/share/doc` but this line specifies that they will be placed in the `/usr/doc` directory.

As with the previous problems, this mistake could be fixed in three different ways. To solve this problem, edit the lbreakout2 `spec` file again, and change the **`%makeinstall`** line to the following:

```
%makeinstall \
     inst_dir=${RPM_BUILD_ROOT}/usr/share/games/lbreakout2 \
      hi_dir=${RPM_BUILD_ROOT}/var/lib/games \
      doc_dir=${RPM_BUILD_ROOT}/usr/share/doc
```

18. Once you have made this change, save the `spec` file and then try once again to compile the software and install it to a virtual root directory:

```
$ rpmbuild -bi lbreakout2.spec
...Output Omitted...
mkdir /var/tmp/lbreakout2-root/usr/share/doc
mkdir /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2
/usr/bin/install -c -m 644 index.html /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2/index.html
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1/docs'
make[1]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
make[2]: Entering directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
make[2]: Nothing to be done for `install-exec-am'.
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
make[1]: Leaving directory `/home/guru/rpmbuild/BUILD/lbreakout2-2.4.1'
+ /usr/lib/rpm/redhat/brp-compress
+ /usr/lib/rpm/redhat/brp-strip /usr/bin/strip
+ /usr/lib/rpm/redhat/brp-strip-static-archive
/usr/bin/strip
+ /usr/lib/rpm/redhat/brp-strip-comment-note /usr/bin/strip /usr/bin/objdump
Processing files: lbreakout2-2.4.1-1
error: File not found by glob: /var/tmp/lbreakout2-root/usr/lib/*.so.*
error: File not found: /var/tmp/lbreakout2-root/usr/share/lbreakout2
error: File not found by glob: /var/tmp/lbreakout2-root/usr/share/man/man8/*
Executing(%doc): /bin/sh -e /var/tmp/rpm-tmp.22825
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd lbreakout2-2.4.1
+ DOCDIR=/var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ export DOCDIR
+ rm -rf /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
```

- Problem solved!
- At this point, the %install has finished, and RPM is now processing the %files stanza.

```
+ /bin/mkdir -p /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ cp -pr AUTHORS COPYING ChangeLog NEWS README TODO /var/tmp/lbreakout2-
root/usr/share/doc/lbreakout2-2.4.1
+ exit 0


RPM build errors:
    File not found by glob: /var/tmp/lbreakout2-root/usr/lib/*.so.*
    File not found: /var/tmp/lbreakout2-root/usr/share/lbreakout2
    File not found by glob: /var/tmp/lbreakout2-root/usr/share/man/man8/*
$
```

19. At this point, the **`%build`** and **`%install`** sections of the `spec` file are correct. Now, the RPM compile is failing because the default **`%files`** stanza in the template file is not correct. To solve this problem, first use the `ls` command to explore what files and directories are in the Buildroot, `/var/tmp/lbreakout2-root`. You should find the following files and directories:

```
/var/lib/games/lbreakout2.hscr
/usr/share/games/lbreakout2/{gfx,gui_theme,levels,sounds}
/usr/share/doc/lbreakout2
/usr/share/doc/lbreakout2-2.4.1
/usr/bin/lbreakout2
/usr/bin/lbreakout2server
```

20. There are two problems here. One is that these files and directories need to be listed in the `%files` section of the `spec` file.

    There is another, more subtle problem, however. Looking again at this `ls` output, you should notice that there are two documentation directories:

```
RHEL/FC: /usr/share/doc/lbreakout2
RHEL/FC: /usr/share/doc/lbreakout2-2.4.1
---------
SLES/SL: /usr/doc/lbreakout2
SLES/SL: /usr/share/doc/packages/lbreakout2
```

    There should only be one documentation directory.

On RHEL/FC it should be the one with the application version, `/usr/share/doc/lbreakout2-2.4.1`.

On SLES/SL it should be `/usr/share/doc/packages/lbreakout2/`.

21. If you look at the contents of `the wrongly located documentation directory` you should discover that it's all HTML documentation. To combine these two directories, you should put the contents of the rouge directory as subdirectory named `html` in the proper documentation directory.

    On RHEL/FC modify the `%install` stanza of the spec file to end with the line:

    `mv ${RPM_BUILD_ROOT}/usr/share/doc/lbreakout2 html`

    On SLES/SL modify the `%install` stanza of the spec file to end with the line:

    `mv ${RPM_BUILD_ROOT}/usr/doc/lbreakout2 html`

    On RHEL/FC When you finish, your Install stanza should read:

```
%install
rm -rf %{buildroot}
%makeinstall \
     inst_dir=${RPM_BUILD_ROOT}/usr/share/games/lbreakout2 \
      hi_dir=${RPM_BUILD_ROOT}/var/lib/games \
      doc_dir=${RPM_BUILD_ROOT}/usr/share/doc
mv ${RPM_BUILD_ROOT}/usr/share/doc/lbreakout2 html

%clean
rm -rf %{buildroot}
```

    On SLES/SL When you finish, your Install stanza should read:

```
%install
rm -rf %{buildroot}
%makeinstall
mv ${RPM_BUILD_ROOT}/usr/doc/lbreakout2 html

%clean
rm -rf %{buildroot}
```

22. After modifying the contents of the documentation directory, you need to adjust the `%files` section to list the newly included documentation files. Add the `html` directory to the `%doc` macro within the `%files` stanza, so that it reads:

```
%doc AUTHORS COPYING ChangeLog NEWS README TODO html
```

23. Look at the rest of your **`%files`** stanza. It currently reads:

```
%files
%defattr(-, root, root)
%doc AUTHORS COPYING ChangeLog NEWS README TODO html
%{_bindir}/*
%{_libdir}/*.so.*
%{_datadir}/%{name}
%{_mandir}/man8/*
```

- This statement sets ownership on the packaged files and directories.
- The files in /var/tmp/lbreakout2-root/usr/bin
- The files in /var/tmp/lbreakout2-root/usr/lib/*.so.*
- The files in /var/tmp/lbreakout2-root/usr/share/lbreakout2
- The files in /var/tmp/lbreakout2-root/usr/share/man/man8

Compare this with the list you obtained in step 19 of the files which you wish to package. When you compare the two lists, you will find that the **`%{_bindir}`** file-matching macro is needed, but that the **`%{_libdir}`** and the **`%{_mandir}`** statements need to be deleted.

24. Correct your **`%files`** stanza to define the correct files produced with LBreakout2 by changing it to the following:

```
%files
%defattr(-, root, root)
%doc AUTHORS COPYING ChangeLog NEWS README TODO html
%{_bindir}/*
%{_datadir}/%{name}
```

25. Once you have made this change, save the `spec` file and then try once again to compile the software and install it to a virtual root directory:

```
$ rpmbuild -bi lbreakout2.spec
...Output Omitted...
Processing files: lbreakout2-2.4.1-1
error: File not found: /var/tmp/lbreakout2-
root/usr/share/lbreakout2
Executing(%doc): /bin/sh -e /var/tmp/rpm-tmp.546
```

- The %files stanza listed a file which could not be found.

```
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd lbreakout2-2.4.1
+ DOCDIR=/var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ export DOCDIR
+ rm -rf /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ /bin/mkdir -p /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ cp -pr AUTHORS COPYING ChangeLog NEWS README TODO html
/var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ exit 0


RPM build errors:
    File not found: /var/tmp/lbreakout2-root/usr/share/lbreakout2
$
```

26. At this point, the `spec` file is almost working. This procedure failed because the `%files` stanza is still slightly incorrect. The `%files` stanza is currently trying to package the directory `/usr/share/lbreakout2`, which does not exist. Looking back at the list of files and directories produced in step 19. on page 63, you'll see that this should instead be `/usr/share/games/lbreakout2`. To fix this problem, edit the spec file and change the line:

    ```
    %{_datadir}/%{name}
    ```

    to:

    ```
    %{_datadir}/games/%{name}
    ```

27. Once you have made this change, save the spec file and then try once again to compile the software and install it to a virtual root directory:

    ```
    $ rpmbuild -bi lbreakout2.spec
    ...Output Omitted...
    ```

28. If you are using RPM 4.1 or later versions, when you complete the previous step, you will get output similar to the following:

    ```
    Checking for unpackaged file(s): /usr/lib/rpm/check-files
    /var/tmp/lbreakout2-root
    ```

```
error: Installed (but unpackaged) file(s) found:
   /var/lib/games/lbreakout2.hscr


RPM build errors:
    Installed (but unpackaged) file(s) found:
   /var/lib/games/lbreakout2.hscr
```

- This error indicates that a file is installed in the virtual root, but is not listed in the %files stanza.

This build attempt is getting closer to succeeding. Now, the build is exiting because a file, `/var/lib/games/lbreakout2.hscr`, was found in the virtual root, but was not listed in the `%files` stanza. Skip the next step, and proceed to step 30 which explains how to add this missing file to the spec file.

- This is the problem mentioned in Step 8 of Task 2. RPM 4.1 and later releases check for installed but unpackaged files and produce a warning or error when any are encountered.

29. If you are using RPM 4.0 or older versions, you will not get output similar to that in the previous step. Instead, your build will actually appear to complete successfully, producing output similar to the following:

```
...Output Omitted...
+ exit 0
Finding  Provides: (using /usr/lib/rpm/find-provides)...
Finding  Requires: (using /usr/lib/rpm/find-requires)...
PreReq: rpmlib(PayloadFilesHavePrefix) <= 4.0-1 rpmlib(Com-
pressedFileNames) <= 3.0.4-1
Requires(rpmlib): rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rpmlib(CompressedFileNames) <= 3.0.4-1
Requires: ld-linux.so.2 libartsc.so.0 libaudiofile.so.0
libc.so.6 libdl.so.2 libesd.so.0 libm.so.6 libpng.so.2
libpthread.so.0 libSDL-1.2.so.0 libX11.so.6 libXext.so.6
libz.so.1 libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1)
libc.so.6(GLIBC_2.1.3) libm.so.6(GLIBC_2.0)
libpthread.so.0(GLIBC_2.0)
```

- If you are using RPM 4.1 or newer versions, you DO NOT have to complete the commands in this step. Be grateful!

RPM 4.1 and later automatically "validate" installations to make sure that all files installed into the `${RPM_BUILD_ROOT}`, complaining if any files were installed but not packaged. RPM 4.0 and earlier releases do not do this, so you must manually validate your installation. To do this, first prepare a list of the files installed by the completed LBreakout2 install, and store these files into a text file in your home directory:

- Remember, these commands are only used if you are using RPM 4.0 or earlier releases.

```
$ find /var/tmp/lbreakout2-root | sort > ~/lbreakout2-installed-files
```

This list of files will incorrectly prepend the `${RPM_BUILD_ROOT}` at the beginning of every file name. Strip this path off:

```
$ perl -pi -e 's/^\/var\/tmp\/lbreakout2-root//g' ~/lbreakout2-installed-files
```

Now, actually build a package:

```
$ rpmbuild -ba lbreakout2.spec
...Output Omitted...
Wrote: /home/guru/rpmbuild/SRPMS/lbreakout2-2.4.1-1.src.rpm
Wrote: /home/guru/rpmbuild/RPMS/i386/lbreakout2-2.4.1-
1.i386.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.63876
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd lbreakout2-2.4.1
+ rm -rf /var/tmp/lbreakout2-root
+ exit 0
```

- This is the location of the built package.

Once your package is built, query it for the files it contains, and store those file names in a text file in your home directory:

```
$ rpm -qlp ../RPMS/i386/lbreakout2-2.4.1-1.i386.rpm | sort >
~/lbreakout2-packaged-files
```

Finally, compare the list of packaged files with the list of installed files:

```
$ diff -Naur ~/lbreakout2-packaged-files ~/lbreakout2-
installed-files
--- /home/guru/lbreakout2-packaged-files      Thu May 22 14:43:42 2003
+++ /home/guru/lbreakout2-installed-files     Thu May 22 14:44:15 2003
@@ -1,5 +1,10 @@
+
+/usr
+/usr/bin
 /usr/bin/lbreakout2
 /usr/bin/lbreakout2server
+/usr/share
+/usr/share/doc
 /usr/share/doc/lbreakout2-2.4.1
 /usr/share/doc/lbreakout2-2.4.1/AUTHORS
```

- In this output, a plus sign (+) indicates a file which was installed but not packaged.

```
 /usr/share/doc/lbreakout2-2.4.1/ChangeLog
@@ -42,6 +47,7 @@
 /usr/share/doc/lbreakout2-2.4.1/NEWS
 /usr/share/doc/lbreakout2-2.4.1/README
 /usr/share/doc/lbreakout2-2.4.1/TODO
+/usr/share/games
 /usr/share/games/lbreakout2
 /usr/share/games/lbreakout2/gfx
 /usr/share/games/lbreakout2/gfx/AbsoluteB
@@ -247,3 +253,7 @@
 /usr/share/games/lbreakout2/sounds/wall.wav
 /usr/share/games/lbreakout2/sounds/weak_ball.wav
 /usr/share/games/lbreakout2/sounds/wontgiveup.wav
+/var
+/var/lib
+/var/lib/games
+/var/lib/games/lbreakout2.hscr
```

Looking over that list, you'll see that the following files or directories are installed, but are not included in the package:

```
/usr
/usr/bin
/usr/share
/usr/share/doc
/usr/share/games
/var
/var/lib
/var/lib/games
/var/lib/games/lbreakout2.hscr
```

Of those files and directories, most are shared directories which should NOT be part of the package, but one file, `/var/lib/games/lbreakout2.hscr`, does need to be added to the package.

30. Your spec file currently correctly compiles LBreakout2, but it fails to install one file which is needed by LBreakout2. To fix this problem, add this file to the **%files** list in the `spec` file by adding the line:

    ```
    %{_localstatedir}/lib/games/lbreakout2.hscr
    ```

    When you finish, your Files stanza should now look like:

    ```
    %files
    %defattr(-, root, root)
    %doc AUTHORS COPYING ChangeLog NEWS README TODO html
    %{_bindir}/*
    %{_datadir}/games/%{name}
    %{_localstatedir}/lib/games/lbreakout2.hscr
    ```

31. Once you have made this change, save the spec file and then try once again to compile the software and install it to a virtual root directory:

    ```
    $ rpmbuild -bi lbreakout2.spec
    ...Output Omitted...
    Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1
    rpmlib(PayloadFilesHavePrefix) <= 4.0-1
    Requires: /sbin/ldconfig libSDL-1.2.so.0 libSDL_mixer-1.2.so.0 libc.so.6
    libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1) libc.so.6(GLIBC_2.1.3)
    libc.so.6(GLIBC_2.3) libm.so.6 libm.so.6(GLIBC_2.0) libpng12.so.0
    libpthread.so.0 libpthread.so.0(GLIBC_2.0) libz.so.1
    Checking for unpackaged file(s): /usr/lib/rpm/check-files
    /var/tmp/lbreakout2-root
    $
    ```

32. This time, the build worked! To finish packaging your software, edit the `spec` file one final time and update the Changelog with contents similar to the following:

    ```
    %changelog
    * Tue Apr 02 2005 John Doe <student@gurulabs.com> 2.4.1-1
    - Initial package
    ```

33. Once you have made this change, save the `spec` file and then try once again to compile the software, install it to a virtual root directory, and package it:

```
$ rpmbuild -ba lbreakout2.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.1332
...Output Omitted...
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.1332
...Output Omitted...
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.41692
...Output Omitted...
Processing files: lbreakout2-2.4.1-3
Executing(%doc): /bin/sh -e /var/tmp/rpm-tmp.60710
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd lbreakout2-2.4.1
+ DOCDIR=/var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ export DOCDIR
+ rm -rf /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ /bin/mkdir -p /var/tmp/lbreakout2-root/usr/share/doc/lbreakout2-2.4.1
+ cp -pr AUTHORS COPYING ChangeLog NEWS README TODO html /var/tmp/lbreakout2-
root/usr/share/doc/lbreakout2-2.4.1
+ exit 0
Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1 rpmlib(PayloadFilesHavePrefix) <= 4.0-1
Requires: /sbin/ldconfig libSDL-1.2.so.0 libSDL_mixer-1.2.so.0 libc.so.6 libc.so.6(GLIBC_2.0)
libc.so.6(GLIBC_2.1) libc.so.6(GLIBC_2.1.3) libc.so.6(GLIBC_2.3) libm.so.6 libm.so.6(GLIBC_2.0)
libpng12.so.0 libpthread.so.0 libpthread.so.0(GLIBC_2.0) libz.so.1
Checking for unpackaged file(s): /usr/lib/rpm/check-files /var/tmp/lbreakout2-root
Wrote: /home/guru/rpmbuild/SRPMS/lbreakout2-2.4.1-3.src.rpm
Wrote: /home/guru/rpmbuild/RPMS/i386/lbreakout2-2.4.1-
3.i386.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.81877
+ umask 022
+ cd /home/guru/rpmbuild/BUILD
+ cd lbreakout2-2.4.1
+ rm -rf /var/tmp/lbreakout2-root
+ exit 0
$
```

- First, the %prep
- Next, the %build
- Next, the %install
- Next, the %files
- The %doc macro in %files
- Next, build the SRPM
- Next, build all RPMs
- Finish by executing the %clean macro from the %install section

34. You've now got the RPM `/home/guru/rpmbuild/RPMS/i386/lbreakout2-2.4.1-1.i386.rpm`!
To try it out, install it as root:

```
RHEL/FC $ su -c "rpm -Uvh ../RPMS/i386/lbreakout2-2.4.1-1.i386.rpm"
SLES/SL $ su -c "rpm -Uvh ../RPMS/i586/lbreakout2-2.4.1-1.i586.rpm"

Password:
Preparing...                ########################################### [100%]
   1:lbreakout2             ########################################### [100%]
$
```

35. Rn the LBreakout2 program to verify that it works:

```
$ lbreakout2
```

- You must be in X for this program to run.

36. After playing a game of LBreakout2, verify the RPM:

```
$ rpmverify -V lbreakout2
S.5....T   /var/lib/games/lbreakout2.hscr
$
```

- This is the high-scores file. It will appear modified, as it does here, as long as you've beaten one of the previous high scores.

Here, you see that the file `/var/lib/games/lbreakout2.hscr` has been modified in various ways:

- Its size is modified (S)
- Its MD5 hash is different, meaning its contents are changed (5)
- Its timestamps are changed (T)

37. Good Linux administrators periodically verify the files on their systems to make sure they are not modified, since most installed files should never change. However, some files (such as this high scores file) might reasonably be modified. RPM allows files which might change to be flagged as configuration files, so that administrators verifying the system won't panic when they notice changes to these files.

To accomodate your administrators, you decide to revise your lbreakout2 RPM. Fix your lbreakout2 spec file to indicate that the high-scores file for LBreakout2 is a configuration file. To do this, **edit** your `spec` file and change the line in the `%files` stanza for the high-scores file from:

```
%{_localstatedir}/lib/games/lbreakout2.hscr
```

to

```
%config %{_localstatedir}/lib/games/lbreakout2.hscr
```

Also, **modify** the Release field in the Header stanza, changing it to:

```
Release: 2
```

Finally, **document** the change you've made to the package by adding an entry to the **%changelog** section similar to the following:

```
* Tue Apr 02 2005 John Doe <student@gurulabs.com> 2.4.1-2
- Mark high-scores file as config file
```

- The %config macro in the %files section indicates that the packaged file is a configuration file, and might reasonably change on a production system.

38. Once you have made these three changes to revise your package, save the spec file and then try once again to compile the software, install it to a virtual root directory, and package it:

```
$ rpmbuild -ba lbreakout2.spec
...Output Omitted...
Wrote: /home/guru/rpmbuild/SRPMS/lbreakout2-2.4.1-2.src.rpm
Wrote: /home/guru/rpmbuild/RPMS/i386/lbreakout2-2.4.1-2.i386.rpm
...Output Omitted...
$
```

39. Now that you've built this revised package, upgrade using the **-F** option to the **rpm** command and test it out:

```
RHEL/FC $ su -c "rpm -Fvh ../RPMS/i386/lbreakout2-2.4.1-2.i386.rpm"
SLES/SL $ su -c "rpm -Fvh ../RPMS/i586/lbreakout2-2.4.1-2.i586.rpm"
Password:
Preparing...                ########################### [100%]
   1:lbreakout2             ########################### [100%]
$ lbreakout2
```

40. Play a couple more games of LBreakout2, making sure to get on the High Scores list, then verify the RPM again:

```
$ rpmverify -V lbreakout2
S.5....T c /var/lib/games/lbreakout2.hscr
$
```

The high-scores file is now correctly flagged as a configuration file (c), so your administrators will not panic when they notice it has changed.

41. At this point, you have created an RPM from scratch for LBreakout2, and have successfully revised it to fix a minor packaging error. Depending on how thoroughly you have tested LBreakout2, you might have discovered one other problem with your package -- due to a packaging mistake, your LBreakout2 game does not actually handle high score logging correctly, though you probably will not notice this unless you play as multiple different users.

The basic mistake made here is this: when a program runs on Unix or Linux, it normally runs as the user who executes it, which means it only has write access to the files which that user can write. For high-score tracking to work correctly for LBreakout2, its high-score file, `/var/lib/games/lbreakout2.hscr`, must be writable by every user who runs the `lbreakout2` binary.

To accomplish this, two changes are necessary:

- The `/var/lib/games/lbreakout2.hscr` file should have permissions of `rw-rw-r--` and be owned by the *games* user and *games* group
- The `/usr/bin/lbreakout2` file should have permissions of `r-xr-sr-x` and owned by the *root* user and the *games* group

To make these changes, edit your `spec` file and modify the **`%files`** section to the following:

```
%files
%defattr(-, root, root)
%doc AUTHORS COPYING ChangeLog NEWS README TODO html
%{_bindir}/lbreakout2server

%attr(2555,root,games) %{_bindir}/lbreakout2
%{_datadir}/games/%{name}
%defattr(0664,games,games)
%config %{_localstatedir}/lib/games/lbreakout2.hscr
```

- List each binary separately, since they need different permissions.
- Make /usr/bin/lbreakout2 belong to the games group and SGID.
- Make the /var/lib/games/lbreakout2.hscr file group-writable and owned by the games group.

Also, modify the Release token in the Header stanza to:

```
Release: 3
```

Finally, document your changes in the **`%changelog`** with a new entry similar to the following:

```
* Tue Apr 02 2005 John Doe <student@gurulabs.com> 2.4.1-3
- Make SGID games and fix ownership and perms of
```

```
high-scores file so that high scores work
```

42. Once you have made these three changes to revise your package, save the spec file and then try once again to compile the software, install it to a virtual root directory, and package it:

```
$ rpmbuild -ba lbreakout2.spec
...Output Omitted...
Wrote: /home/guru/rpmbuild/SRPMS/lbreakout2-2.4.1-3.src.rpm
Wrote: /home/guru/rpmbuild/RPMS/i386/lbreakout2-2.4.1-3.i386.rpm
...Output Omitted...
$
```

43. Now that you've built this revised package, uninstall the existing package and install the new package:

```
$ su -c "rpm -e lbreakout2"
Password:
$ su -c "rpm -Uvh ../RPMS/i386/lbreakout2-2.4.1-3.i386.rpm"
Password:
Preparing...                ########################### [100%]
   1:lbreakout2             ########################### [100%]
$
```

44. Notice the new permissions on the files installed by your revised package:

```
$ ls -l /usr/bin/lbreakout2* /var/lib/games/lbreakout2.hscr
-r-xr-sr-x    1 root     games      279698 Apr 29 12:58 /usr/bin/lbreakout2
-rwxr-xr-x    1 root     root        49665 Apr 29 12:58 /usr/bin/lbreakout2server
-rw-rw-r--    1 games    games         227 Apr 29 12:58 /var/lib/games/lbreakout2.hscr
$
```

Congratulations! At this point you have a high-quality, usable RPM for the LBreakout2 application which you can give to your users....

45. There is one additional customization which you can make to your LBreakout2 RPM which might be useful to some of your users. RPM supports "sub

packages" – the creation of multiple binary RPMs from a single source RPM file. Look at the binaries installed by your current lbreakout2 RPM:

```
$ rpm -ql lbreakout2 | grep bin
/usr/bin/lbreakout2
/usr/bin/lbreakout2server
$
```

Also, examine the number and size of the graphics and sound files installed by LBreakout2:

```
$ rpm -ql lbreakout2 | grep "/usr/share/games" | wc -l
    205
$ du -sh /usr/share/games/lbreakout2
3.4M    /usr/share/games/lbreakout2
$
```

The `/usr/bin/lbreakout2` binary is the stand-alone application users run to play LBreakout2. In addition, however, LBreakout2 also provides the `/usr/bin/lbreakout2server` binary. This is a network daemon which can be run on a server. Once started on a server, clients can run `/usr/bin/lbreakout2` on their systems, connect to this server daemon, and start a network game of LBreakout2. The directory `/usr/share/games/lbreakout2` contains 205 files consuming about 3.4 megabytes of space for artwork required by the client, but not by the server.

For convenience, you might choose to package LBreakout2 in four files:

- `lbreakout2`, which provides the client binary
- `lbreakout2-server`, which provides the server binary
- `lbreakout2-graphics`, which provides the graphics and sound files needed by the client and optionally useful on the server
- `lbreakout2-doc`, which provides the documentation files

Splitting LBreakout2 in this fashion will provide your users more flexibility -- they can install just the parts of LBreakout2 they need, on just the machines on which they need them.

46. To split LBreakout2 into sub packages, edit the `spec` file one final time. Sub packages are made simply by adding additional `%package` and `%description` stanzas to the Header section, then adding additional `%files` stanzas which indicate which files get put in which sub packages. First, add a

line to the current %description stanza, indicating that the lbreakout2 package supplies the LBreakout2 client, like the following:

```
This package supplies only the lbreakout2 client, suitable
for use in single-player games.
```

- The "main" %description and %package information applies to the "base" package, in this case lbreakout2

47. After that %description stanza, add additional %package and %description stanzas for the new sub packages:

```
%package server
Summary: lbreakout2 network server
Group: Amusements/Games

%description server
This package supplies the lbreakout2server daemon, useful for creating
multi-player LBreakout2 servers

%package graphics
Summary: lbreakout2 graphics, sound, and level files
Group: Amusements/Games

%description graphics
This package supplies LBreakout2 graphics, sound, and level files. This package must be installed on
client systems, and can be installed on server systems if desired.

%package doc
Summary: lbreakout2 documentation
Group: Amusements/Games

%description doc
This package supplies documentation of LBreakout2.
```

- Create a new sub package, lbreakout2-server
- Create a new sub package, lbreakout2-graphics
- Create a new sub package, lbreakout2-doc

48. Next, split the %files section up into individual sections for each sub package. When you finish, your %files section should look something like:

```
%files
%attr(2555,root,games) %{_bindir}/lbreakout2
%defattr(0664,games,games)
```

- Files to put in the lbreakout2 package

```
%config %{_localstatedir}/lib/games/lbreakout2.hscr

%files server
%defattr(-, root, root)
%{_bindir}/lbreakout2server

%files graphics
%defattr(-, root, root)
%{_datadir}/games/%{name}

%files doc
%defattr(-, root, root)
%doc AUTHORS COPYING ChangeLog NEWS README TODO html
```

- Files to put in the lbreakout2-server package
- Files to put in the lbreakout2-graphics package
- Files to put in the lbreakout2-doc package

49. Now, add an entry to the Changelog documenting your latest change, similar to the following:

```
* Tue Apr 02 2005 John Doe <student@gurulabs.com> 2.4.1-4
- Split into multiple packages for user convenience
```

50. In the Header stanza, increase the Release field one more time:

```
Release: 4
```

51. One additional change is required in the Header stanza. The `/usr/bin/lbreakout2` binary will not run without the graphics, sound, and level files installed, so the lbreakout2 package needs to require the lbreakout2-graphics package. Change the Requires field in the Header to require the lbreakout2-graphics package:

```
Requires: /sbin/ldconfig lbreakout2-graphics
```

- The graphics, art, and music files could just be put in the lbreakout2 package, but then they could not be installed on the server easily without also unnecessarily installing the client....

52. Once you have made these changes to revise your package, save the `spec` file and then try once again to compile the software, install it to a virtual root directory, and package it:

```
$ rpmbuild -ba lbreakout2.spec
...Output Omitted...
Wrote: /export/home/rpmtestuser/rpmbuild/SRPMS/lbreakout2-2.4.1-4.src.rpm
```

```
Wrote: /export/home/rpmtestuser/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
Wrote: /export/home/rpmtestuser/rpmbuild/RPMS/i386/lbreakout2-server-2.4.1-4.i386.rpm
Wrote: /export/home/rpmtestuser/rpmbuild/RPMS/i386/lbreakout2-graphics-2.4.1-4.i386.rpm
Wrote: /export/home/rpmtestuser/rpmbuild/RPMS/i386/lbreakout2-doc-2.4.1-4.i386.rpm
...Output Omitted...
$
```

53. Congratulations! You've now packaged LBreakout2 from scratch, creating four sub packages from one source package.

Task 4

- Create a GPG key pair.
- Sign and verify your LBreakout2 RPMs.

This lab explores the functionality built into the RPM command for signing packages, ensuring package authenticity and integrity.

1. In a previous lab task, you created an RPM for the LBreakout2 application. RPM provides a variety of built-in integrity and authenticity signatures which can be checked on RPM packages. First, check the existing signatures of your `lbreakout2-2.4.1-4` RPM file:

```
RHEL/FC $ rpmsign -K ~/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
SLES/SL $ rpmsign -K ~/rpmbuild/RPMS/i586/lbreakout2-2.4.1-4.i586.rpm
rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm: sha1 md5 OK
$
```

- On RPM 4.1 and later, you should see both SHA-1 digests and MD5 checksums. On RPM 4.0 and earlier, you should only see MD5 checksums. Both MD5 and SHA-1 are used to provide digital signatures of the package.

Here, you see that the SHA-1 and MD5 signatures of this package are correct. These are both file integrity checks, assuring you that the contents of the package have not been modified.

2. RPM also provides the capability to digitally sign packages using GPG or PGP. GPG / PGP signatures are useful to the users who install your software because these types of signatures allow both the integrity of the package to be verified (much like MD5 and SHA-1), and also the authenticity of the package to be verified – if you sign your packages with GPG, your users can be guaranteed that the software they install comes from you, and not from someone creating Trojan Horses which appear to come from you.

To utilize this GPG functionality, you must first create GPG keys which will be used to sign your packages. To create keys, first run the `gpg` command to create the GPG configuration files in your home directory:

- The output below is from GnuPG 1.2. Older releases of GnuPG, such as GnuPG 1.0, will produce different output, and will actually exit immediately after creating initial configuration files

```
RHEL/FC $ mkdir ~/.gnupg; chmod 700 ~/.gnupg
$ gpg
gpg: /home/guru/.gnupg: directory created
gpg: new configuration file `/home/guru/.gnupg/gpg.conf' created
gpg: keyblock resource `/home/guru/.gnupg/secring.gpg': file open error
gpg: keyring `/home/guru/.gnupg/pubring.gpg' created
```

```
gpg: Go ahead and type your message ...

CTRL-C
gpg: some signal caught ... exiting

$
```

- Press CTRL-C to exit out of **gpg**. You will not need to do this on older versions of **gpg**.

3. Once the GPG configuration files have been created, generate GPG keys:

```
$ gpg --gen-key
gpg (GnuPG) 1.2.6; Copyright (C) 2004 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

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.
About to generate a new ELG-E keypair.
              minimum keysize is  768 bits
              default keysize is 1024 bits
    highest suggested keysize is 2048 bits
What keysize do you want? (1024) 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
```

- Output seen here will vary slightly with GnuPG version. Select the default on your version of GnuPG.
- Select 1 here.
- 1024 bits is fine here, or 2048 bits if you're more paranoid.
- 0 is fine here. In production use, people typically expire keys on a periodic basis, however.
- If you haven't made any mistakes, say y here. If you have made mistakes, say n and correct them.

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

Real name: RPM Test
Email address: guru@gurulabs.com
Comment: RPM Package Builder
You selected this USER-ID:
    "RPM Test (RPM Package Builder) <guru@gurulabs.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:password
Repeat passphrase:password
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.
++++++++++.+++++++++++++++++++++++++++++++++++...++++++++++
++++++++++++++++++++..+++++++++++++++++++++++++++++.+++++.
...++++++++++....++++++++++>++++++++++..>+++++.<.+++++.....
.>+++++.....<+++++.>.+++++......<+++++..............+++++
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.
+++++++++++++++++++++++++.+++++++++++++++++++.++++++++++++
+++.++++++++++++++++++++++++++++++++++++++++++++.+++++.+++
+++++++++++++++++>+++++.......+++++^^^
gpg: /home/guru/.gnupg/trustdb.gpg: trustdb created
public and secret key created and signed.
key marked as ultimately trusted.
```

- Enter your first and last name here.
- Enter your email address.
- If you want, enter a descriptive comment here.
- If you haven't made any mistakes, say o here. If you have made mistakes, correct them.
- Enter a password here. This password is all that prevents someone from stealing your key file and impersonating you, so guard it carefully.
- Entropy (sources of "random" data) are needed by GnuPG. It uses seemingly random events like keyboard key presses or mouse movement as sources of entropy. If enough entropy is not available, it will actually prompt you to generate more entropy by carrying out activities such as moving the mouse, typing on the keyboard, or generating hard disk traffic.
- The output seen here will vary slightly with GnuPG version.

```
pub  1024D/87786C00 2003-04-29 RPM Test (RPM Package Builder) <guru@gurulabs.com>
     Key fingerprint = 24ED F0C7 2148 E88C 51B8  1CA2 8D2A FB9A 8778 6C00
sub  1024g/9B24A1DC 2003-04-29

$
```

4. At this point, you have generated public and private GPG keys. To verify that they were created correctly, you can view them:

```
$ gpg --list-keys
/home/guru/.gnupg/pubring.gpg
--------------------------
pub  1024D/87786C00 2003-04-29 RPM Test (RPM Package Builder) <guru@gurulabs.com>
sub  1024g/9B24A1DC 2003-04-29

$
```

5. Now that you have created a GPG key, you must configure RPM to allow you to use it. First, configure RPM to use GPG:

```
$ echo "%_signature gpg" >> ~/.rpmmacros
$ echo "%_gpg_path $HOME/.gnupg" >> ~/.rpmmacros
$
```

Also, configure RPM to locate the GPG key you just created:

```
$ echo "%_gpg_name guru@gurulabs.com" >> ~/.rpmmacros
$
```

- Make sure you use the same email address here you used in step 3 when you generated your GPG keys.

6. Once RPM is configured to support GPG, use it to sign a package with your new GPG private key:

```
RHEL/FC $ rpmsign --addsign ~/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
SLES/SL $ rpmsign --addsign ~/rpmbuild/RPMS/i586/lbreakout2-2.4.1-4.i586.rpm
Enter pass phrase: password
Pass phrase is good.
rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm:
$
```

- Enter your GPG password that you created in step 3.

7. Next, examine the signatures of your freshly signed package. If you are using RPM 4.0 or older, you will see:

```
RHEL/FC $ rpmsign -K ~/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
SLES/SL $ rpmsign -K ~/rpmbuild/RPMS/i586/lbreakout2-2.4.1-4.i586.rpm
rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm: md5 gpg OK
$
```

This indicates that the package has a correct MD5 signature, is signed using GPG, and that the GPG signature of the package can be decrypted using a GPG public key on your GPG keyring. Congratulations! You have signed a package, and verified that it is correctly signed. You may skip the rest of the steps in this lab; they are only necessary for users running RPM 4.1 and newer releases.

- When you generated your public and private key pair in step 3, GPG automatically added your public key to your personal public keyring. When you ran **rpmsign -K**, **rpmsign** used this public key to decrypt the GPG signature within the package file.

8. If you are using RPM 4.1 and newer releases, when you examine the signatures of your package, you will instead see:

```
RHEL/FC $ rpmsign -K ~/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
SLES/SL $ rpmsign -K ~/rpmbuild/RPMS/i586/lbreakout2-2.4.1-4.i586.rpm
rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm: (SHA1) DSA
sha1 md5 (GPG) NOT OK (MISSING KEYS: GPG#87786c00)
$
```

Although your package is signed with your correct GPG key, it still shows up as NOT OK with RPM 4.1 and newer releases. This is because RPM 4.1 and newer releases use their own separate GPG keyring, rather than reading the keyring of the user running the RPM command. RPM is trying to verify the signature on your package using the RPM GPG keyring, and RPM does not yet have your public key installed on its keyring, so it cannot verify the signature.

9. To correct this, you will first need to export your public key in a format RPM can understand. To see your public key, use the command:

```
$ gpg --export --armor

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.2.1 (GNU/Linux)

mQGiBD6u2hQRBAC1wEn+cJvjz8P94ogeE/PCwX4mUBbY4QvKiklrXgr+bpD
d9OmaM+JqbnqyBIq9KfsxVLK651VFB7jBqKwzjBVQER+xsDJW79vKE2mDv0
zNRJuxG2ull8iwwlyub3H9Jky+VXJ4upA3CxhwDRMjZ1PShhJzPSX0paHwC
```

- Fortunately, this output makes more sense to the GPG software than it does to you!

```
x29FQOTKZ6tfgDopoDbq4bUD/1rwocepHES4EOrzz4TVKXAtwASP+Z18OkI
RPmRQ2nkawnRMhIl+taTcbjBJapm1YiKAyqkGwiWtBMTQER9Ox+Uz1p4SmQ
VO0SAGpi+2jjmD79ViRAqdAd+JcgLSI8c9TEZo3D3UFWZI/8yv5PHJMgGQo
nmbFA/0XeJH5aluTFgFnJCP2JMFvMDG2SwiGmwOyDiAWZ7/RTulOt7Y8fMk
OgLlN9wsAZ6JNCmkUP/jiQFYpyMULI1Soca2h+PKYUenEjmwo1LRWUGkdLa
3t8PA92mAhcJynRG+JEEYL5x3NHDVSIzBxz3mTedWj7YQLkkCrRBVGVzdCB
YWdlIChSUE0gUGFja2FnZSBCdWlsZGVyKSA8a2Fib29tQHdpbnN1Y2tzLmd
YWJzLmNvbT6IWQQTEQIAGQUCPq7aFAQLBwMCAxUCAwMWAgECHgECF4AACgk
mod4bABO7QCfWJoP+QFploVay2FPAskbrle4DjgAnAhdbdFJ27Wbolk1xob
IPd/uQENBD6u2hUQBACOgzjL5Wz0Tg/avYSfiswAC5wwKGXTxG9RZPF3cdB
F/k2KsFpoDg/5p9nzSM/h3MjH+1Lr/ZOJgMr/Bz8TunnJg+I+vx00vh0Tw2
tV3B+0jx+ots3A+Opmn77eUGSm+lOv1PQYfQ9kwRsE7+BO9ylgHsrbx9myA
BQP/aJl2yC5H9MTyiK8PvVzr2hrAdykZYGCmJzcmSJ1guiLEgQ7qG0i3Ycm
z+YxIvon42w6EqaoDBp9RHkK2KQpuvNgQfUSMUcr5tIUf3EqDAy37wgdFPx
eh4yem1zigQ12M/Vk5BkQGBs/8Ol+7VfZ1pvgtvZiXtKSoqIRgQYEQIABgU
FQAKCRCNKvuah3hsAGH6AJ9jgyNiBSEf9a5LcetnZqQez/a9dwCfeNEmgUo
/ZxE3l/sTe5Nb+8=
=OnkZ
-----END PGP PUBLIC KEY BLOCK-----
$
```

10. Now, save your GPG public key to a file:

```
$ gpg --export --armor > /tmp/gpg-public-key.asc
$
```

11. Next, import your public key into RPM's GPG keyring:

```
$ su -c "rpmsign --import /tmp/gpg-public-key.asc"
Password:
$
```

12. Once your public key is imported into the RPM GPG keyring, examine the signed LBreakout2 package signatures again:

```
$ rpmsign -K ~/rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm
rpmbuild/RPMS/i386/lbreakout2-2.4.1-4.i386.rpm: (sha1) dsa sha1 md5 gpg OK
$
```

13. This time, your GPG signature verified correctly because RPM now has your GPG public key installed correctly. You can also give your GPG public key to your users, and they can import it into RPM using the same process you did in step 11. Once they have imported your public key, they can verify the authenticity and integrity of any RPMs you create for them!