rpminspect

David Cantrell

Mar 03, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Goals

rpminspect is an RPM policy enforcing and deviation analysis tool. It looks at the output of an RPM build (e.g., the output of rpmbuild or a build system like Koji) and examines the contents of the build artifacts to report:

- Policy compliance
- Changes from a previous build to the current build
- General correctness and best practices

rpminspect is the frontend tool, and librpminspect provides the inspection engine and inspections. The program is configured through a configuration file and runtime data provided by a vendor-specific rpminspect data package. The **rpminspect-data-generic** package in this source tree provides a template for constructing the vendor-specific data package. For example, in Fedora Linux the rpminspect-data-fedora package provides configuration files and runtime data for rpminspect when checking Fedora Linux RPM packages.

1.2 Build Types Support

rpminspect expects its input to either be local RPM packages or a Koji build. Koji produces certain types of builds that are not supported by rpminspect. Right now, the following input types are supported:

- Local RPM packages (source and binary)
- Koji builds (i.e., a source RPM run with 'rpmbuild -ba' on all specified architectures)
- Modules

If comparing local RPM packages, rpminspect assumes the before and after specifications are peers whereas for a Koji build, rpminspect matches peer packages (e.g., in the gcc package, gcc-gfortran in the before build is peered with gcc-gfortran in the after build).

1.3 Intended Audience

Developers, QE, release engineers, and system administrators who regularly build RPM packages. for use in some environments or products.

From an individual user standpoint, rpminspect is a command-line tool you can use as a linter of sorts. rpminspect reports, and that's it. rpminspect can output information in JSON or xUnit format, which makes it easier to integrate with automated workflows or web frontends.

The reporting uses certain classifications for different things found, but it up to the end-user to determine what to do with that information. If used with an external build tool, the JSON or xUnit data may be more useful as you can construct decision making around those results.

CHAPTER

TWO

INSTALLATION

2.1 Program and Vendor Configuration

rpminspect works directly with RPM packages, so the focus is on RPM-based Linux distributions. The software does build and run on non-RPM distributions if all of the build requirements are available.

The easiest way to rpminspect is to use your distribution's packaging system. In Fedora Linux, you can use dnf to install rpminspect and the required dependencies:

```
dnf install rpminspect
```

That command will install the program, library, and required dependencies. The other important package you will need is your vendor's data package for rpminspect. Using Fedora Linux as an example, you will need to install rpminspect-data-fedora:

```
dnf install rpminspect-data-fedora
```

If you have reported a bug to the rpminspect project and are asked to test a development snapshot, you can do that by enabling the Copr repository for rpminspect and using the same dnf commands. Instructions on where to find the Copr repository are in the project README file in the rpminspect source tree.

The above instructions also apply to CentOS and Red Hat Enterprise Linux. Those products will have different vendor data packages, so be sure to install the one that applies to the environment your packages are built for. You can be running Fedora Linux and run rpminspect on packages built for CentOS. All you need to do is make sure you have the vendor data package installed for the target environment.

If you are installing rpminspect on another distribution and it is available a prebuilt package, please let us know so the documentation here can be updated.

2.2 Runtime Requirements

NOTE: This information is provided as an explanation of how to manually install the typical runtime requirements for rpminspect. The vendor data package for rpminspect, such as rpminspect-data-fedora, should carry dependencies on any optional requirements necessary. In most cases, it is sufficient to install the rpminspect package and the appropriate vendor data package.

In addition to the libraries that will be linked in to librpminspect, there are a number of userspace programs used:

Executable	Required?	Project
/usr/bin/desktop-file-validate	No	https://www.freedesktop.org/wiki/Software/desktop-file-utils/
/usr/bin/msgunfmt	**Yes**	https://www.gnu.org/software/gettext/
/usr/bin/abidiff	No	https://sourceware.org/libabigail/
/usr/bin/kmidiff	No	https://sourceware.org/libabigail/
/usr/bin/annocheck	No	https://sourceware.org/git/annobin.git
/usr/bin/diffstat	No	https://invisible-island.net/diffstat/

The provided RPM spec file template uses the Fedora Linux locations for these files, but in the program, they must be on the runtime system. Different inspections use these tools at runtime. Most distributions include the above tools. If they are available, you should use those packages.

In Fedora Linux, for example, you can run the following to install these programs:

```
dnf install desktop-file-utils gettext diffstat libabigail /usr/bin/annocheck
```

The *shellsyntax* inspection uses the actual shell programs listed in the shells setting in the rpminspect configuration file. Since this can vary by system, you should make sure they are available in the system PATH. Or you can just not use the shellsyntax inspection. The RPM spec file for rpminspect includes weak dependencies on the default list of shells:

```
dnf install dash ksh zsh tcsh rc bash
```

An even easier option is to use the developer setup method described in the next section.

2.3 Help for non-RPM Distributions

The osdeps/ directory contains scripts and package listings to set up different Linux distributions so they can build rpminspect and run the test suite. You may find the information in here helpful if you are trying to use rpminspect on a non-RPM Linux distribution.

CHAPTER

THREE

CONFIGURATION

3.1 Main Configuration

rpminspect requires configuration and runtime data in order to perform the various checks on RPM packages. At the very least you will need to supply a main configuration file to rpminspect. This main configuration file is supplied by the vendor data package, such as rpminspect-data-fedora on Fedora Linux. The file is installed in **/usr/share/rpminspect** and usually matches the product name and ends with **.yaml**. In rpminspect-data-fedora you will find **/usr/share/rpminspect/fedora.yaml**. This configuration contains the settings that direct rpminspect to the other runtime data in **/usr/share/rpminspect**.

Note: You always need to tell rpminspect what main configuration file to use with the `-c` option. For example, when checking a Fedora Linux build, your command line must begin with:

```
rpminspect -c /usr/share/rpminspect/fedora.yaml
```

To make this easier on users, the vendor data packages usually include a wrapper script to do this for you. With rpminspect-data-fedora installed, you can run:

```
rpminspect-fedora
```

Instead of the longer command.

3.2 Product Releases

The program needs to know the product release when performing inspections. rpminspect uses the *dist tag* value to determine the product release. If you are comparing builds, the product releases must match, otherwise rpminspect will ask you to clarify which product you want to use for the inspection. Vendors can change policies as products evolve so the rules defined in the vendor data package for one product may be different from another, and so on.

Because dist tags vary, rpminspect can have a set of regular expressions to match those strings to products for the purposes of inspections. For example, in Fedora Linux the product release string for Fedora Linux 35 builds is **fc35** but that includes any dist tag that matches the **^.fc35.$** regular expression. These expressions are defined in the **fedora.yaml** file in the rpminspect-data-fedora package.

Note: It is important to note that the dist tag value matches what product environment you are building for, not necessarily what product your package is part of. For example, add on packages for Fedora Linux 35 are considered

part of the **fc35** product environment in rpminspect. It is important to keep this in mind if you are building a third party repository of packages intended for a particular vendor product.

In cases where rpminspect cannot determine a product release, you can always pass the `-r` option, such as `-r fc35`. This forces rpminspect to use the **fc35** product rules.

Looking in the rpminspect-data-fedora package, you will see the data files in there map to product release strings. Product release strings allow rpminspect to find per-product rules in the subdirectories in **/usr/share/rpminspect**.

3.3 Profiles

The next level of configuration available are profiles. These exist in **/usr/share/rpminspect/profiles/VENDOR**. They can contain any of the settings available in the main configuration file. The intent of profiles is to create further configuration customizations for categories of builds. For example, all kernel builds could use the *kernel* profile. That starts by loading the main configuration file then reading in the settings from the kernel profile. The kernel profile could then turn off inspections that do not apply to kernel builds or adjust other settings from the main configuration file.

Profiles are effectively free form. You can create any profiles you want. To share them across the product, profiles should be coordinated and grouped together in the vendor data package. In Fedora Linux we have the rpminspect-data-fedora package and any developer can submit a pull request to this project to add to the configuration data. You can add or modify profiles this way too.

To use a profile, specify the name with the `-p` option. For example:

```
rpminspect-fedora -p kernel
```

3.4 rpminspect.yaml

The last configuration option available to users, and probably the most common, will be the **rpminspect.yaml** file you can place in the current directory. This file lets you adjust configuration values for your rpminspect run. The **rpminspect.yaml** file can contain any settings available in the main configuration file. Common uses of this file are to set ignore lists per inspection.

Within Fedora Linux these files are committed to the git repository for the package on the applicable release branch. In Fedora CI these files are read by the rpminspect job. As the package maintainer you can control how rpminspect runs for your builds by adjusting this file in the git repository for the package.

Here are some examples:

- The package provides no desktop files. You can disable the *desktop* inspection by creating an **rpminspect.yaml** file that contains:

```
---
inspections:
    desktop: off
```

- The package contains no Java **.class** files. You can disable the *javabytecode* inspection by creating an **rpminspect.yaml** file that contains:

```
---
inspections:
    javabytecode: off
```

- The *filesize* inspection is set too low and reports too many changes. The default is 20%, but you want to increase it to 47% so rpminspect will not report file size changes between builds that are under that percentage. You can create an **rpminspect.yaml** file that contains:

```
---
filesize:
    size_threshold: 47
```

- The package needs to allow **/usr/lib64/systemd** DT_RPATH values. You can add the path to the list of allowed DT_RPATH values by creating an **rpminspect.yaml** file that contains:

```
---
runpath:
    allowed_paths:
        - /usr/lib64/systemd
```

A full example is provided in the rpminspect source tree under the name **data/generic.yaml** (latest upstream one is available here). You should also check the main configuration in the vendor data package to see what the setting is for that product and then adjust accordingly in your local configuration file.

3.5 Runtime Data

Certain inspections pull their rules from runtime data files in **/usr/share/rpminspect**. These files map to the product release string and can be found in subdirectories in **/usr/share/rpminspect**. For an example of what can go in each file, see the corresponding files in the **data/** subdirectory of the rpminspect source tree.

Here is a description of the subdirectories found in **/usr/share/rpminspect**:

- **abi/**

  Per-product files (e.g., **el8**) defining the ABI compatibility levels and what packages belong to each one.

- **capabilities/**

  The capabilities list. A list consisting of 4 whitespace delimited fields. The first field is the package name, second field is the installed file path, third field is always **=**, and the fourth field is a comma-delimited list of `capabilities(7)` we expect and allow on that file in that package.

- **licenses/**

  Contains a JSON database of all approved licenses that packages may reference in the *License* field.

- **politics/**

  Per-product files listing allow or deny rules based on regular expression matching and comparing message digests. This check originated as a way to catch country or political substrings in filenames.

- **fileinfo/**

  Per-product files (e.g., **fc35**) containing `ls(1)` style output of expected permissions, owners, groups, and filenames in packages. The format is 4 whitespace-delimited fields. The first column is a human readable permission mask (e.g., **-rwsr-s-r-x**), the second field is the owner name, the third field is the group name, the fourth field is the filename.

- **rebaseable/**

  Per-product files that list package names that are always allowed to rebase. That is, the names of the before and after packages match but the version numbers differ. That is considered a rebase and in

some cases is not something we want to allow. Listing the file here permits it through the rebase check.

- **profiles/**

  This directory contains rpminspect profiles in the form of **NAME.yaml** where **NAME** is the name of the profile. Any setting in **rpminspect.yaml** is valid in a profile. Anyone can define a new profile. rpminspect uses the profile defined by the `-p` option. Profiles are intended for categories of packages that need to make the same adjustments to the rpminspect configuration at runtime. You can create a profile for those packages rather than modifying the **rpminspect.yaml** files for each of those packages.

- **security/**

  Contains per-product files that specify the actions to take for different security rules on a per package and version basis. For example, a product may have an older build of a package like openssl and needs to relax some of the security rules in rpminspect. Later product versions can define stricter rules. This file is also how you instruct rpminspect to ignore things it reports as needed product security inspection. The idea here is that the package maintainer will consult with their product security team to determine if the reported problem warrants a security rule and if so add it to this file for that product. Subsequent runs of rpminspect on the package in question will follow the rule here and ideally not continue reporting the security failure.

CHAPTER

FOUR

USAGE

4.1 Setup

Make sure you have installed the program and corresponding data file collection. Fedora Linux users can use dnf:

```
dnf install rpminspect rpminspect-data-fedora
```

The first package is the actual program. The rpminspect-data-fedora package delivers data files used by the various inspections performed by rpminspect. It also delivers the rpminspect configuration file. Have a look at this file and make sure it works for your environment. The idea is to support per-vendor data packages efficiently. For example, if CentOS needs different settings, it should provide a **rpminspect-data-centos** package.

Ideally, the configuration file should not need changing. If you do find mistakes, please file an issue or send a pull request to the project:

> https://github.com/rpminspect/rpminspect-data-fedora

The aim should be the vendor-specific data packages providing everything a developer needs to run rpminspect locally for that product successfully.

4.2 Input Types

rpminspect looks at files contained in RPM packages. But RPM packages have a lot of infrastructure built up around them too. For starters, a source RPM can generate multiple binary RPMs. Second, Linux distributions tend to build for multiple architectures. The collection of all of the built RPMs, or artifacts, is collectively referred to as a *build*. Fedora Linux tends to refer to a build using the *name-version-release*, or NVR, syntax. For example, **glibc-2.33-8.fc34**. Here the name is **glibc**, the version is **2.33**, and the release is **8.fc34**.

rpminspect requires at least an RPM of some sort as input. However, since it is very common for RPM maintenance to deal with all of the built RPMs and all applicable architectures, rpminspect can take a complete build as input. In Fedora Linux, when you specify an *NVR* build rpminspect will query Koji for information on that build and if it's found, download all of the RPM packages. You may also cache these Koji builds locally and refer to them by their filesystem path, in which case rpminspect will not go and download the build again.

If you are performing package work locally and just want to look at a single RPM, you can specify that as well. Just pass the path to the RPM on the command line. There are other RPM build systems out there and it would be interesting to see rpminspect expand to gain support for fetching builds from them.

4.3 Modes

rpminspect can operate in one of two modes when processing RPM packages. The first is called **analysis mode**. This mode is where you provide a single build or RPM package as the input. rpminspect will check that single input and perform all of the inspections that apply to single inputs only.

The second mode is a **comparison mode**. This mode performs all of the analysis mode checks but also performs checks that can happen when there is a before and after build to compare. These checks cover things like reporting when things change between builds. The analysis mode is more checks that enforce policies on built packages.

The mode cannot be specified via a command line option. The mode of operation is implied based on provided one or two builds as input on the command line.

4.4 Reporting Levels

rpminspect will exit with code 0 indicating success if all of the results reported are either at the **INFO** or **OK** level. Anything that is **VERIFY** or **BAD** will trigger an exit code of 1 indicating a failure. The failure reporting threshold can be adjusted with the `-t` option. The default failure threshold is **VERIFY**. The `-s` option allows you to specify a reporting level for results suppression. That is, anything below the specified level will be suppressed in the output. For example, `-s VERIFY` will suppress all results below the **VERIFY** level.

The results also contain a value indicating who is the appropriate entity or party to waive a particular result. This information can be changed by sending pull requests to the project. It has no bearing on the exit code of 0 and is only present for use by environments integrating rpminspect in to their workflows.

4.5 Workflow Examples

rpminspect runs from the command line. The inputs must be local RPM packages, a Koji build specification (*NVR*), a Koji scratch build task number, a Koji module specification, or a locally cached Koji build output (regular build or module). For inputs originating from Koji, rpminspect talks to Koji and download the build artifacts. For repeated runs, you may want to cache a remote build locally to avoid downloading it with each run. The examples below use Fedora Linux, so they will reference the **rpminspect-fedora** wrapper script provided by rpminspect-data-fedora as the command to run.

Here is a simple invocation using **tmux** as an example:

```
$ rpminspect-fedora -v tmux-2.9a-2.fc31 tmux-2.9a-3.fc31
```

This just runs with verbose mode enabled and compares **tmux-2.9a-2.fc31** to **tmux-2.9a-3.fc31**. rpminspect downloads the packages for these builds and runs the inspections.

If you want to keep temporary files created during the run, pass `-k`. rpminspect tells you where those files are when it finishes.

You can list available inspections with the `-l` option:

```
$ rpminspect-fedora -l
```

Say you only want to run the license inspection on the builds above:

```
$ rpminspect-fedora -v -k -T license tmux-2.9a-2.fc31 tmux-2.9a-3.fc31
```

Now let's say you want to run the license and manpage inspections:

```
$ rpminspect-fedora -v -k -T license,manpage tmux-2.9a-2.fc31 tmux-2.9a-3.fc31
```

And lastly, what if you want to run *all* inspections except the license one:

```
$ rpminspect-fedora -v -k -E license tmux-2.9a-2.fc31 tmux-2.9a-3.fc31
```

What about specify a locally cached build? First, let's start by caching the builds we have been using:

```
$ mkdir ~/builds
$ rpminspect-fedora -v -w ~/builds -f tmux-2.9a-2.fc31 tmux-2.9a-3.fc31
```

Now let's run all the inspections but specify the locally cached builds:

```
$ rpminspect-fedora -v ~/builds/tmux-2.9a-2.fc31 ~/builds/tmux-2.9a-3.fc31
```

Easy. Again, these locally cached builds must look like what rpminspect would download from Koji. Hence using rpminspect first to download it.

rpminspect can also run inspections on local RPM packages. Similar to Koji inputs, you may specify a single RPM or two RPM packages to compare. For example:

```
$ rpminspect-fedora -v ~/rpmbuild/RPMS/x86_64/tmux-2.9a-2.fc31.x86_64.rpm
```

Or:

```
$ cd ~/rpmbuild/RPMS/x86_64
$ rpminspect-fedora -v tmux-2.9a-2.fc31.x86_64.rpm tmux-2.9a-3.fc31.x86_64.rpm
```

All of the other command-line options that apply to Koji tests work for local RPM packages.

For more information, see the man page for `rpminspect(1)`. And see the `--help` output for information on command-line option syntax.

4.6 Command Line Options

Compare package builds for policy compliance and consistency.

Usage: `rpminspect [OPTIONS] [before build] [after build]`

Options:

-c FILE, --config=FILE Configuration file to use

-p NAME, --profile=NAME Configuration profile to use

-T LIST, --tests=LIST List of tests to run (default: ALL)

-E LIST, --exclude=LIST List of tests to exclude (default: none)

-a LIST, --arches=LIST List of architectures to check

-r STR, --release=STR Product release string

-n, --no-rebase Disable build rebase detection

-o FILE, --output=FILE Write results to FILE (default: stdout)

-F TYPE, --format=TYPE Format output results as TYPE (default: text)

-t TAG, --threshold=TAG Result threshold triggering exit failure (default: **VERIFY**)

-s TAG, --suppress=TAG Results suppression threshold (default: off, report everything)

-l, --list List available tests and formats

-w PATH, --workdir=PATH Temporary directory to use (default: `/var/tmp/rpminspect`)

-f, --fetch-only Fetch builds only, do not perform inspections (implies `-k`)

-k, --keep Do not remove the comparison working files

-d, --debug Debugging mode output

-D, --dump-config Dump configuration settings used (in YAML format)

-v, --verbose Verbose inspection output when finished, display full path

-?, –help Display usage information -V, –version Display program version

See the `rpminspect(1)` man page for more information.

4.7 Available Inspections

Analysis Mode

These inspections run when a single build is provided as input to the program.

- **license**

  Verify the string specified in the License tag of the RPM metadata describes permissible software licenses as defined by the license database. Also checks to see if the License tag contains any unprofessional words as defined in the configuration file.

- **emptyrpm**

  Check all binary RPM packages in the build for any empty payloads. When comparing two builds, report new packages in the after build with empty payloads.

- **metadata**

  Perform some RPM header checks. First, check that the Vendor contains the expected string as defined in the configuration file. Second, check that the build host is in the expected subdomain as defined in the configuration file. Third, check the Summary string for any unprofessional words. Fourth, check the Description for any unprofessional words. Lastly, if there is a before build specified, check for differences between the before and after build values of the previous RPM header values and report them.

- **manpage**

  Perform some checks on man pages in the RPM payload. First, check that each man page is compressed. Second, check that each man page contains valid content. Lastly, check that each man page is installed to the correct path.

- **xml**

  Check that XML files included in the RPM payload are well-formed.

- **elf**

  Perform several checks on ELF files. First, check that ELF objects do not contain an executable stack. Second, check that ELF objects do not contain text relocations. When comparing builds, check that the ELF objects in the after build did not lose a `PT_GNU_RELRO` segment. When comparing builds, check that the ELF objects in the after build did not lose `-D_FORTIFY_SOURCE`.

- **desktop**

Perform syntax and file reference checks on `*.desktop` files. Syntax errors and invalid file references are reported as errors.

- **disttag**

  Check that the Release tag in the RPM spec file includes the `%{?dist}` directive.

- **specname**

  Ensure the spec file name conforms to the *NAME.spec* naming format.

- **modularity**

  Ensure compliance with modularity build and packaging policies (only valid for module builds, no-op otherwise).

- **javabytecode**

  Check minimum required Java bytecode version in class files, report bytecode version changes between builds, and report if bytecode versions are exceeded. The bytecode version is vendor specific to releases and defined in the configuration file.

- **addedfiles**

  Report added files from the before build to the after build. Debuginfo files are ignored as are files that match the patterns defined in the configuration file. Files added to security paths generate special reporting in case a security review is required. New setuid and setgid files raise a security warning unless the file is in the whitelist.

- **ownership**

  Report files and directories owned by unexpected users and groups. Check to make sure executables are owned by the correct user and group. If a before and after build have been specified, also report ownership changes.

- **shellsyntax**

  For all shell scripts in the build, perform a syntax check on it using the shell defined in its #! line (shell must also be listed in shell section of the configuration data). If the syntax check returns non-zero, report it to the user and return a combined stdout and stderr. If comparing two builds, perform the previous check but also report if a previously bad script is now passing the syntax check.

- **annocheck**

  Perform annocheck tests defined in the configuration file on all ELF files in the build. A single build specified will perform an analysis only. Two builds specified will compare the test results between the before and after build. If no annocheck tests are defined in the configuration file, this inspection is skipped.

- **permissions**

  Report `stat(2)` mode changes between builds. Checks against the fileinfo lists for the product release specified or determined. Any setuid or setgid changes will raise a message indicating a security team should review it.

- **capabilities**

  Report `capabilities(7)` changes between builds. Checks against the capabilities list for the product release specified or determined. Any capabilities changes not on the list will raise a message indicating a security team should review the change. This inspection is only present if rpminspect was built with libcap support.

- **pathmigration**

Report files that are packaged in paths that have migrated to new locations. For example, packages should not package anything directly in `/bin` but rather `/usr/bin`. The path migrations are defined in the **rpminspect.yaml** file.

- **lto**

  Link Time Optimization (LTO) produces smaller and faster shared ELF executables and libraries. LTO bytecode is not stable from one release of gcc to the next. As such, LTO bytecode should not be present in `.a` and `.o` ELF objects shipped in packages. This inspection looks for LTO bytecode in ELF relocatable objects and reports if any is present.

- **symlinks**

  Symbolic links must be resolvable on the installed system. This inspection ensures absolute and relative symlinks are valid. It also checks for any symlink usage that will cause problems for RPM.

- **files**

  Reads each `%files` section in the spec file and looks for any forbidden path references. Forbidden path references are defined in the configuration file under the **files:forbidden_paths** section. An example might be reporting spec files that use `/usr/lib` rather than `%{_libdir}`.

- **patches**

  Report patches defined in the spec file that are under 4 bytes as invalid patch files. Report the percentage by which patches change between builds. Report how many lines are touched by a patch. Based on size and line count thresholds in the configuration file, report findings at either the INFO or VERIFY level.

- **virus**

  Check every file in the after build for viruses using libclamav. Any positive result is reported as a BAD result.

- **politics**

  Check for possible politically sensitive files in packages. The things to check for are defined in the per-product release files in `/usr/share/rpminspect/politics` that are provided by the vendor data package. This inspection was originally introduced to catch potentially political names or phrases in filenames.

- **badfuncs**

  Check for forbidden functions in ELF files. Forbidden functions are defined in the runtime configuration files. Usually this inspection is used to catch built packages that make use of deprecated API functions if you wish built packages to conform to replacement APIs.

- **runpath**

  Check for forbidden paths in both the `DT_RPATH` and `DT_RUNPATH` settings in ELF shared objects. If both `DT_RPATH` and `DT_RUNPATH` are found in an ELF object, report it as a BAD result since that would be a linker error.

- **unicode**

  Scan extracted and patched source code files, scripts, and RPM spec files for any prohibited Unicode code points, as defined in the configuration file. Any prohibited code points are reported as a possible security risk.

- **rpmdeps**

  Check for correct RPM dependency metadata. Report incorrect or conflicting findings as well as expected changes when comparing a new build to an older build. Changes are only reported when

comparing builds, but this inspection will check for correct RPM dependency metadata when inspecting a single build and report findings.

Comparison Mode

These inspections run when a before and after build are specified as the input to the program. All of the analysis mode inspections also run when in comparison mode.

- **lostpayload**

  Check all binary RPM packages in the before and after builds for any empty payloads. Packages that lost payload data from the before build to the after build are reported.

- **changedfiles**

  Report changed files from the before build to the after build. Certain file changes will raise additional warnings if the concern is more critical than just reporting changes (e.g., a suspected security impact). Any gzip, bzip2, or xz compressed files will have their uncompressed content compared only, which will allow changes through in the compression level used. Message catalog files (.mo) are unpacked and compared. Public C and C++ header files are preprocessed and compared. Any changes with unified diff output are included in the results.

- **movedfiles**

  Report files that have moved installation paths or across subpackages between builds. Files moved with a security path prefix generate special reporting in case a security review is required. Rebased packaged report these findings at the INFO level while non-rebased packages report them at the VERIFY level or higher.

- **removedfiles**

  Report removed files from the before build to the after build. Shared libraries get additional reporting output as they may be unexpected dependency removals. Files removed with a security path prefix generated special reporting in case a security review is required. Source RPM packages and debuginfo files are ignored by this inspection.

- **upstream**

  Report Source archives defined in the RPM spec file changing content between the before and after build. If the source archives change and the package is on the rebaseable list, the change is reported as informational. Otherwise the change is reported as a rebase of the package and requires inspection.

- **dsodeps**

  Compare `DT_NEEDED` entries in dynamic ELF executables and shared libraries between the before and after build and report changes.

- **filesize**

  Report file size changes between builds. If empty files became non-empty or non-empty files became empty, report those as results needing verification. Report file change percentages as info-only.

- **kmod**

  Report kernel module parameter, dependency, PCI ID, or symbol differences between builds. Added and removed parameters are reported and if the package version is unchanged, these messages are reported as failures. The same is true module dependencies, PCI IDs, and symbols. This inspection is only available is rpminspect was built with libkmod support.

- **arch**

  Report RPM architectures that appear and disappear between the before and after builds.

- **subpackages**

Report RPM subpackages that appear and disappear between the before and after builds.

- **changelog**

  Ensure packages contain an entry in the `%changelog` for the version built. Reports any other differences in the existing changelog between builds and that the new entry contains new text entries.

- **types**

  Compare MIME types of files between builds and report any changes for verification.

- **abidiff**

  When comparing two builds or two packages, compare ELF files using `abidiff(1)` from the libabigail project. Differences are reported. If the package is a rebase and not on the rebaseable list and the rebase inspection is enabled, ABI differences are reported as failures. The assumption here is that rpminspect is comparing builds for maintenance purposes and you do not want to introduce any ABI changes for users. If you do not care about that, turn off the abidiff inspection or add the package name to the rebaseable list.

- **kmidiff**

  kmidiff compares the binary kernel Module Interfaces of two Linux kernel trees. The binary KMI is the interface that the Linux kernel exposes to its modules. The trees we are interested in here are the result of the build of the Linux kernel source tree. If the builds compared are not considered a rebase, an incompatible change reported by `kmidiff(1)` is reported for verification.

- **config**

  Report `%config` files changing to/from `%config` status between the before and after builds. Report whitespace only changes as INFO messages and report content changes as VERIFY messages unless the comparison is for a rebased package in which case the INFO message level is used. For `%config` files that are symlinks, compare the link destinations and report changes using the reporting levels just mentioned.

- **doc**

  Report `%doc` files changing to/from `%doc` status between the before and after builds. These messages are at the INFO level for rebased builds and the VERIFY level otherwise. The main objective here is to catch projects that may rename documentation files (e.g., README to README.md) in a minor update that the package maintainer might overlook.

CHAPTER

FIVE

REPORTING BUGS

The upstream project is at https://github.com/rpminspect/rpminspect. The best way to report a bug is to open an Issue on this project. Please provide the command you used and note the inputs (e.g., RPM packages or builds) specified.

CHAPTER

SIX

DEVELOPMENT

6.1 Build Requirements

A typical Linux system with a toolchain for building C software using meson and ninja, plus the following libraries:

Requirement	URL	License
json-c	http://json-c.github.io/json-c	MIT
xmlrpc-c	http://xmlrpc-c.sourceforge.net/	BSD and MIT
libxml-2.0	http://xmlsoft.org/	MIT
rpm	https://github.com/rpm-software-management/rpm	GPL-2.0-or-later
libarchive	https://www.libarchive.org/	BSD
elfutils	https://sourceware.org/elfutils/	LGPL-3.0-or-later
kmod	https://git.kernel.org/pub/scm/utils/kernel/kmod/kmod.git	GPL-2.0-or-later
zlib	https://www.zlib.net/	Zlib and BSL-1.0
mandoc	https://mandoc.bsd.lv/	ISC
libyaml	https://github.com/yaml/libyaml	MIT
file	http://www.darwinsys.com/file/	BSD
OpenSSL	https://www.openssl.org/	OpenSSL
libcap	https://sites.google.com/site/fullycapable/	BSD-3-Clause

Additionally, the unit test suite requires the following packages:

Requirement	URL	License
CUnit	http://cunit.sourceforge.net/	LGPL-2.0-or-later

Most distributions include the above projects in prebuilt and packaged form. If those are available, you should use those packages.

6.2 Developer Setup

If you are planning on contributing to rpminspect, you should use the developer setup target in the Makefile to make sure you have the required packages installed:

```
make instreqs
```

This target will install all BuildRequires, Requires, and Suggests dependencies from the spec file template, but also the additional test suite requirements listed in the files in the `osdeps/` subdirectory.

If you are working on a distribution other than Fedora/RHEL/CentOS, then have a look at the `rpminspect.spec.in` template for the packages listed as Requires, BuildRequires, and Suggests. These usually have similar names on other distributions. If you have instructions for setting up a development and runtime environment on another distribution, please send a pull request with the information.

6.3 Building

If you got the source from a released tarball or from cloned from git, you can do this:

```
meson setup build
ninja -C build
```

You can get verbose build output with `ninja -C build -v`.

Note: On CentOS 7.x and RHEL 7.x systems, the ninja command is named `ninja-build` and installed in `/usr/bin`.

6.4 Contributing

Patches are welcome, as are bug reports. There is a lot to do in this project. Some things to keep in mind:

- Please follow the existing coding style in files you modify. Things like variable and function naming, spacing, and indentation. I want to avoid wildly varying coding styles throughout the tree.
- New inspections in librpminspect need to be in the form of an `inspect_NAME.c` file with a driver added to the main struct. You may add static and non-static support functions to your `inspect_NAME.c` file and expose those as part of the librpminspect API. If the support functions are generic enough, feel free to start a new source file.
- Test cases or updates to existing test cases need to accompany patches and new code submissions.
- Use the standard C library whenever possible. Code using glib, libbsd or any other type of generic utility library is going to be reviewed and likely rejected until it is modified to use the standard C library.
- That said, inspections should make use of available libraries for performing their work. When given the option between a library and forking and executing a program, please use the library.
- Alternative libraries intended to replace a build requirement are ok so long as `meson.build` and `meson_options.txt` are modified to conditionalize the choices. For example:

  ```
  -Djson_lib=other_json_lib
  ```

  If this requires modification in the code, try to minimize the use of C preprocessor macros.
- See the TODO file for a current list of things that need work.

6.5 Licensing and Copyright

The program (`src/`), the test suite (`test/`), and various developer tools in this source tree are available under the GNU General Public License version 3 or, at your option, any later version.

The librpminspect library (`include/` and `lib/`) is available under the GNU Lesser General Public License version 3 or, at your option, any later version.

The data files (`data/`) are licensed under the Creative Commons Attribution 4.0 International Public License.

Some source files in the project carry Apache License 2.0 licenses or BSD licenses. Both of these cases are allowed under the GPLv3 and LGPLv3. The combined work is licensed as described in the previous paragraph.

Copyright statements are in the boilerplates of each source file.

CHAPTER

SEVEN

TESTING

Follow the steps in the Development section and then:

```
meson test -C build
```

You can get verbose build output with:

```
meson test -C build -v
```

The verbose mode is useful when tests fail because you can see the debugging information dumps and other output. You may also use the `Makefile` to run the test suite or a specific subset of tests. For example:

```
make check
```

Runs all of the tests where as:

```
make check xml
```

Will just run the xml inspection tests. The `Makefile` target is provided as a convenience so you don't need to remember the meson commands.

You can also use a Python virtualenv to run and debug the tests. Make sure pip and virtualenv are installed and then:

```
virtualenv -p python3 my_test_env
. my_test_env/bin/activate
pip-3 install meson ninja rpm-py-installer rpmfluff setuptools
export RPMINSPECT=$PWD/build/src/rpminspect
export RPMINSPECT_YAML=$PWD/data/generic.yaml
export RPMINSPECT_TEST_DATA_PATH=$PWD/test/data
```

To run all tests, execute the command:

```
python3 -Bm unittest discover -v test/
```

To run a specific test suite, execute:

```
python3 -Bm unittest discover -v test/ test_emptyrpm.py
```

CHAPTER

EIGHT

RELEASE PROCESS

The `make release` command creates a new release. The command handles the version number change, tests, git tag, and repo push.

Release entries on GitHub are created manually. The signed archive is attached manually to the release entry. Use the `make announce` target to generate a text file that can be used for the GitHub release entry and the blog post. The output of `make announce` also needs to be prepended to the CHANGES.md file and included in that release.

The `make koji` command submits official builds for Fedora Linux and EPEL.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search