Pp. 155-166 of the *Proceedings of LISA ’02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

An Analysis of RPM Validation Drift

John Hart and Jeffrey D’Amelia – Tufts University

ABSTRACT

Experiments that analyze dependencies in RedHat Linux and RpmFind.net show disturbing conflicts and overlaps between software packages that result in installing multiple differing versions of dynamic libraries. The final state of a system containing conflicting packages depends upon the order in which packages are installed, as well as user input during the installation process. This leads to system states that may or may not have been tested, lowering confidence that the resulting software configuration will function properly. We describe the details of the problem, potential effects, and potential solutions involving improving the practice of building RPM packages.

Introduction

RedHat Package Manager (RPM) files and their equivalents have revolutionized the ease with which one can add software to a Linux system. But do RPMs embody ease, or perhaps danger? The following excerpt from Ladislav Bodnar’s article “Is RPM Doomed?” [2] gives a very accurate account of a situation that most system administrators have experienced:

> “*You have just found this great software on the Internet and off you go to download and install it. It’s all free and GPL and, as luck would have it, the author provides a binary package in RPM format. It doesn’t take long to download it, then you run the customary* `rpm -Uvh package-name.rpm` *command. OOPS! The installation fails, reporting a missing dependent package without which it will not install or function correctly. Off you go again to search the Internet for the missing library.*
>
> *Unfortunately, installing that missing library fails because of three other missing libraries and two other libraries that come in incorrect versions. Depending on how badly you want the original package, you have two choices – either go and search for all missing dependent libraries as well as all libraries dependent on the dependent libraries, or you just give up. How many times have you given up?*
>
> *If you are persistent and lucky, you might eventually install the RPM package. If you are persistent and not lucky, then you have probably acquired a few bumps from banging your head against the nearest wall in sheer desperation. RPM dependency hell can be a hugely frustrating experience – anything from circular dependencies (the catch 22 situation) to incorrect library version when there wouldn’t be much left untouched had you really persisted in getting that badly wanted RPM installed.*”

The root of several problems with RPM (and many other kinds of package management) is that “order matters” [11]. It is common practice among many administrators to install and uninstall RPMs and other kinds of software packages with little concern for change control and without keeping a journal of the order of modifications. But case studies and theoretical analyses [10, 11] suggest that the only way to produce a predictable and reliable system is to decide upon some particular order for package installations and other configuration actions, and always perform the actions in that order. Another independent analysis [5] suggests that order matters whenever system configuration actions do not take a rather restrictive form in which all configuration actions are “homogeneous” with one another; this means that if two actions change the same file, they change it to have the exact same content. While this would seem a reasonable requirement, in our experience, RPM installations do not satisfy this restriction.

How dangerous is it in practice to ignore this discipline of ordering? It seems from practical experience that the danger is far greater than most of us realize. Many of us have managed to put systems into a state where “only starting over is feasible.” Why is this so?

This study looks at the risks associated with installing and uninstalling RPMs. We look at the nature of dependencies between packages to understand how one package has the potential to break another. We explain how use of poorly structured RPMs causes a “validation drift” in which the final system gradually “drifts” over time to a configuration that has not been tested. Using global analysis of existing RPM repositories, we identify subtle inconsistencies in well-known RPM packages that can lead us to doubt the results of installing them.

First we must comment that this study is limited in several ways. We only study the i386 distributions of RedHat 6.2, RedHat 7.2, and the Contrib directory, as listed on RedHat.com and RpmFind.net. These directories contain highly volatile data that will be quickly outdated. Much of the work was done in April of 2002 and repeated in July of 2002, with *differing* results due to changes (mostly improvements) in repository files! We are happy to report that one major example of repository rot that was discovered in April could not be reproduced from July data. We hope that *none* of the inconsistencies we report will ever be reported again, because implementors and repository managers will be motivated to address the problems we have found. Though there may be different inconsistencies, do not expect to necessarily find any of these particular problems in future RPM repositories.

Why RPM?

There are many package managers available to Linux developers and choosing one to focus on was difficult. Package managers such as Debian's DEB [13] or Slackware's TGZ [15] would have been fine choices for our analysis. However, the nature of RPM makes it a good basis to analyze the nature of installation failures and validation drift while also providing a framework for thinking about solutions to these problems. RPM is deployed in large and small network installations and many system administrators depend on it for installing and maintaining complex sets of software. It has a rich feature set that allows the installation and removal of individual packages or sets of packages and it maintains an internal database that records and verifies each change to the system.

Like most other package managers, RPM as a system allows a user to install packages in a computer, while checking an internal database to verify that all known prerequisites are met before allowing the package to install. After configuration, RPM allows the package to run arbitrary binary and script files prior to and for the completion of installation and uninstallation [1]. Because of this, we concentrate upon the Red Hat package management system [1] for Red Hat 6.2 and 7.2, including the contributed packages available from http://www.rpmfind.net.

Many readers have commented that we already know that the contrib directory is broken, so why analyze it? We respond that it helps to know *how* things break, so that we can avoid them in the future. We knew when we started that there would be serious problems, but had no idea of the nature of the problems we would find. And, surprisingly, the problems we found are not the problems that we expected!

RPM Dependencies

In every RPM package there exist several different kinds of dependencies. Declared dependencies external to the file are contained in the header information in each RPM package. Each package declares which services it "provides" and "requires." A service is nothing more than a string. RPM satisfies dependencies by forcing one to sequence software installations so that services are "provided" by installed packages before they are "required" by others.

But also, each executable file in an RPM archive has requirements, some of which can be determined through use of techniques like those of sowhat [4]. These internal dependencies are intrinsic to the file and may or may not be related to dependencies declared in the RPM header. Sowhat's analysis utilizes the output of ldd and is not exhaustive; one can subvert it, e.g., by using dlopen to open a dynamic library by name, bypassing ldd and ld.so.conf.

Some kinds of errors are relatively insignificant. If an RPM package is over-declared (the set of dependencies in the header exceeds the set that the program needs), the consequence is that extra, possibly useless, programs are installed. If a package is under-declared, the packages actually used and required are greater then what is declared. This means that a package installation will fail even if the declared dependency requirements are fulfilled.

Are RPM Dependencies Sloppy?

The root of all evil in RPM seemed to be – at the outset – the way RPM packages are expected to declare what they need in order to operate. In RPM, there is a simple mechanism for notating dependencies between packages. In the package header, each package is declared to "provide" zero or more services. These are just strings with no real semantic meaning. A package that needs a service then "requires" it. This mechanism is mainly used to declare dependencies between packages using a dynamic library and the package(s) that might provide a copy of the library, so the "services" are typically library base-names.

Those of us who have experienced "dependency hell" (as documented in the excerpt in the introduction) have suspected major problems with the RedHat dependency system. We attempted to validate our suspicion by running a simple test to check the difference between actual and declared dependencies. The results were both more encouraging than expected and, in a way, depressing. The true dependency errors seem to be few in number and cannot account for the trouble many people report in using RPMs.

Checking Dynamic Library Dependencies

We could not in general determine all dependencies for a package, but we can determine all the dynamic libraries needed by a package. We did this by unpacking each package (using cpio) and skipping execution of the installation scripts. Then we ran ldd on each executable or dynamic library in the package. Finally, we compared the actual dependencies exposed by ldd with the declared dependencies in the header.

Comparing these was a complex process due to the free-form nature of dependencies. It was a multi-step process that takes into account all the ways a dependency can be declared in a collection of RPMs. The cases for each target RPM (the RPM from the collection currently under investigation) include:

a) The required library is part of the target RPM that requires it. In this case, no dependency listing is required.

b) The required library is explicitly required by the target RPM and provided by another. This is normal.

c) The required library is part of an RPM that provides *another* service that happens to be required by the target RPM. This is an implicit dependency based upon a service tag that is actually unrelated to the real library dependency (Figure 1). This is usually bad style for dependency declarations, at least for dynamic libraries. The only exception is that to save space, some implementors use blanket tags for service subsystems, e.g., "require qt". This is to avoid listing all the core libraries of the service explicitly.

d) The required library is part of the RedHat core distribution, for which dependencies are not explicitly listed, as their presence in any RedHat system is assured.

e) The required library is in another RPM with which the target RPM shares no explicit or implicit dependency. This is a dependency error.

To analyze the distribution of these kinds of dependencies within RPM repositories, we wrote two programs. Rift lists all of the dependencies in a set of packages that can be exposed through ldd. Tree reads all RPM files in an RPM distribution and outputs all dependency and checksum information from the distribution. The output of tree, together with the output of rift, is fed to a new program deps that categorizes dynamic library dependencies into each of the five classes above.

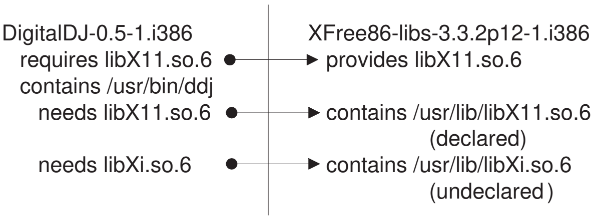

Figure 1: Implicit dependency of ddj upon libXi.so.6 via libX11.so.6.

Our results for the RedHat i386 distribution are shown in Table 1. The good news is that the distribution itself, as it comes from RedHat, is rather well-constructed with few errors. The errors seem to be statistical outliers. Errors we found were isolated to two packages. The anaconda runtime package anaconda-runtime-7.2-7.i386.rpm fails to require ld-linux.so.2, libc.so.6, libresolv.so.2, and libz.so.1. These are part of the core distribution so that this has no observable behavioral effect; it is just a bit sloppy. PyQt-2.4-1.i386.rpm fails to require libstdc++-libc6.1-1.so.2; this is a bit more serious and requires user intervention.

Our results for the RedHat contrib directory (i386) are shown in Table 2. Here things become more "interesting." Most packages are astoundingly well-behaved about declaring their needs. Outright errors are almost a statistical outlier. Errors we observed are listed in Table 3. These are annoying but minor at best.

# deps	Kind of Dependencies
741	normal: "requires" and "provides" correct.
26	internal: package contains library upon which it depends.
5	errors: dependency declaration omitted.

Table 1: Dependency types in RedHat 7.2 .

# deps	Kind of Dependencies
1201	normal: "requires" and "provides" correct.
21	internal: package contains library upon which it depends.
9	implicit: unrelated dependency includes file.
8	errors: dependency declaration omitted.

Table 2: Dependency types in RedHat Contrib.

Package	Fails to require
Eterm-0.8.8-1.i386.rpm	libungif.so.4
Frodo-4.1a-1.i386.rpm	Logo
ImageMagick-4.2.7-1.i386.rpm	libbz2.so.0
ImageMagick-perl-4.2.7-1.i386.rpm	libbz2.so.0
Qtabman-0.1-1.i386.rpm	libclntsh.so.1.0
XITE-3.3-3.i386.rpm	libjpeg.so.62
XITE-3.3-3.i386.rpm	libz.so.1
aktion-0.2.1-1.i386.rpm	libstdc++-libc6.1-1.so.2

Table 3: Dependency errors in RedHat i386 Contrib.

One disturbing tendency was some use of implicit loading of libraries. There were nine instances in which a dynamic library was required implicitly as a side-effect of another explicit requirement. The X11 library libXi.so.6 was implicitly required six times as a result of explicitly requiring libX11.so.6. Likewise, libdl.so.g was implicitly required three times as a result of explicitly requiring libc.so.6. These libraries were also among those that were formerly bundled with the libraries whose dependencies load them. The implicit loads of these libraries are probably due to packages being designed before that library design change took place.

The prognosis of this work is surprisingly good. While it would seem that the contributed RPM repository would be chaos, our simple checks showed contributed RPMs to be relatively organized and well-structured. Our obvious question, then, is "what is really wrong?" It is *not* the dependencies, because errors in these are statistically insignificant. We must look deeper for potential problems within RPMs. The key to this looking deeper is the concept of *validation* of the resulting system.

Validation

The central theme of this paper is not dependency analysis itself, but rather the relationship between dependency analysis and validation of software installation. In software engineering [9], there are two forms of sanity checking:

- ''verification'': ''are we making the product right?'' Does it conform to our own ideas of how it should work?
- ''validation'': are we making ''the right product?'' Does it conform to customer needs?

In system administration, we often concentrate upon validation to the exclusion of any concept of verification. We have less design freedom than software authors, and user requirements are usually more completely spelled out than for a software engineer, so that there is much less difference between ''verification'' and ''validation'' than there would be in a software development environment. The key to validation is rigorous testing [12] in a realistic setting. One must actually try the system and see if things work as expected.

Last year, Couch and Sun described global analysis [4] without emphasizing validation, its most important component. Any analysis of what went wrong with a system must be tempered by knowledge that at some time in the past, ''things were right.'' One's reasoning must always flow from knowledge that the system did work properly before. Otherwise, the question of ''what broke'' – central to use of sowhat – has no meaning. Unfortunately, it is often the case that the user thinks ''things were right'' in their system configuration when in fact they have been working with a system that was not completely validated. This perception by the user can contribute to the problem at hand. Dependency problems are always problems of validation: ''can we be sure that in making a change, we do not break anything?''. Given a rather strongly validated core distribution of RPMs, how can we avoid breaking anything in it that worked before?

Validation Rot

Brooks [3] points out that in software engineering, ''software rot'' occurs when too many small changes are made to a complex system, so that no one really understands the function of the software. Couch points out that a similar kind of ''filesystem rot'' occurs in software repositories managed over long time periods [6], so that meanings of specific files become unclear and inaccessible.

Our analyses show that RedHat machines managed via RPM suffer from a new kind of rot: ''validation rot.'' This is a gradual divergence from a fully tested configuration that invalidates and undermines prior testing. Here is how it works:

- We start with a ''baseline configuration,'' e.g., a RedHat distribution. This configuration is present on a multitude of hosts around the internet, so we can wait until this baseline has been comprehensively validated by an extensive community of users.
- Gradually, over time, we add functionality in the form of ''contributed RPMs.'' Each one of these adds some files and may replace others. The result is that the system diverges not only from the baseline, but also into a fairly unique state that may not be replicated anywhere else on the Internet.
- At any point in this process, one has a unique system that has never been validated by anyone.

The user community (and vendors) validate packages in relation to the baseline, not in relation to other packages. It is nearly impossible to test, yet alone reach, all possible package states due to combinatorial explosions. This means that a user is ''at risk'' when their system reaches a configuration state that has not been achieved by any previous population of users or the software developer. It is possible, then, that there are latent bugs that will show up only on that particular machine.

Validation Expense

Validating software is expensive. For commercial software, it often requires the expertise of a Software Quality Assurance(SQA) [9] team. For open-source freeware, the user community itself often serves that purpose over longer time periods, submitting bug reports and fixes. Either way, software can only be validated as working properly by extensive testing in multiple environments and with various kinds of inputs. There is no such thing as ''completely tested software'' [10] and one must always decide what form of testing is ''good enough'' or ''complete enough'' [12].

Transitive Validation

The expense of validation has led the Linux Standard Base [14] to employ a ''transitive validation'' strategy. Previously, a vendor wanting to market a software package for linux had to validate its function on every distribution of linux. The Linux Standard Base was created in order to give vendors the assurance that a package that works somewhere, works everywhere. If we control the couplings between a software package and its operating environment, and can validate the environment as possessing appropriate couplings, then validating it in one compliant environment validates it in all such environments.

There are two parts to the Linux Standard Base:

1) a code validator that indicates whether a specific binary file is compliant with the base. This

checks whether the system calls used by the binary file are loaded from correct versions of dynamic libraries.
2) an environment validator that checks whether the environment on a specific linux machine complies with the minimal requirements needed for system calls to work properly. This checks not only the existence and versions of key libraries, but also checks that particular system control files are found in standardized locations, e.g., /etc/hosts.

The key assertion chain of LSB is that:

a) If a particular vendor software package passes code validation, i.e., only utilizes approved system and library calls, and
b) The vendor package has been tested on one LSB-compliant system, and
c) A particular linux system passes environmental validation, i.e., has all its libraries and files in appropriate places, then
d) The vendor software should function fine on *any* environmentally compliant system.

This is a "transitive validation" claim: software that works in one compliant environment works in every such environment. This can potentially save tons of money in validating software for different distributions of linux.

Is Validation Trust Transitive?

We are inspired by the LSB strategy and would like to apply a similar process to the problem of validating RPM-managed systems. The key question is "What can we trust?". Trying to answer this question cuts to the heart of the RPM problem. We contend that "we usually put too much trust in existing infrastructure."

For example, let's consider the following apparent assertions about "transitive trust":

1) If a program works properly on a RedHat 7.2 system, it will work properly on all RedHat 7.2 systems.
2) If a script works properly for a particular version of Perl, then it will work properly in the same version of Perl regardless of the environment in which it executes.
3) If a program or dynamic library compiled with one compiler works properly, then it will work properly if compiled with another compiler.
4) If a program works before new software is installed, it will work after the software is installed.

In each case, we decide to trust something in a new situation based upon validation in an old situation. All of the above assertions seem reasonable, and all are quite obviously *false* to the point of being ludicrous. Each of these points can be expressed in realistic terms as follows:

1) Just what is a RedHat 7.2 system? This is the baseline, but what has been done to the system since then? If a program depends, e.g., upon /etc/foo, then it will only work if one has installed /etc/foo. This has nothing to do with the baseline.
2) Any Perl programmer knows that Perl does some rather strange things to cope with system differences. For example, its implementation of lockf can take at least four forms depending upon support for locking in the operating system. These forms are semantically different.
3) Modern compilers have bugs, especially when optimization is turned on. Validation under one compiler is no guarantee of function when the same program is compiled with another.
4) Even in the simplest of cases, it is easy to break a program by installing another. The problem is "hidden dependencies" between programs and other programs and libraries.

These simple examples are obviously bogus, but administrators who install RPMs on an ad-hoc basis are using them as assumptions. We come to the inexorable conclusion that a system is validated as functional if:

1) it is constructed starting from a validated baseline,
2) all software packages installed in addition to the baseline are:
   a) validated against the baseline configuration by being installed against it and thoroughly tested.
   b) homogeneous [5], in the sense that overlaps between packages other than the baseline install the exact same content.
   c) uncoupled from the contents of other packages (excluding homogeneous overlaps), so that software within each package only refers to baseline content and the content of the specific package.

Analysis of RPM Failures

There are four main things that can interfere with the proper operation of a single package:

1) Hidden dependencies not known to the package designer.
2) Version skew between files and the programs that utilize them.
3) Relationships between files that are obscured by scripting.
4) Asynchronous operations other than package management that affect package files or required files.

Hidden dependencies are those unknown to the package designer or simply undeclared. Every package implicitly depends, e.g., upon the whole base distribution. If something in the base distribution changes, the package may break, but such dependencies are never made explicit. According to the results above, though, these may be statistically insignificant.

Version skew is a very common problem with which many people are familiar in both the Unix and Windows worlds. This happens when a library or program associated with more then one program is upgraded and the newer version is not functionally compatible with the older version.

Installation scripts, which are commonly included with programs today, are usually designed to move files and create directories that are custom to the package. But with larger, more complex packages, installation scripts are non-trivial and can perform tasks that have system-wide effects. Since the changes that an installation script can make are limited only by the rights of the user running it, (the user is typically root) any program has the ability to touch any other program or file. A common example is when an Apache RPM is installed. It makes modifications to the inetd.conf file that are not obvious if one is not aware that modifications are being made.

Asynchronous operations, which include installing non-RPMs, manually changing files that rpm controls, hacking, etc., can also have a compelling effect on the validity of installations. Since most complex systems span multiple volumes, when a package is located on one volume and a dependent package is located on a different volume, both volumes must be mounted or the dependent package may not work. This type of dependency is difficult to properly diagnose.

Global Analysis

The problem with the above descriptions of failure modes is that they are all module-centric. They can explain what's happening when one module is installed, but do not depict potentially subtle multi-module interactions. We applied the global analysis techniques of sowhat [4] to this problem and found potential and subtle failures in adding RPMs to the standard distributions. While the configuration language for RPM files allows expression of ''backward'' dependencies between referrer and required resource, ''forward'' dependencies between a new resource and an old program that uses it can lead to systems whose function has not been properly tested.

Our Experiments

We undertook several experiments to understand the scope of the problem of validation drift. Our first task was to identify the scope of inhomogeneity within RPM packages. We obtained several package distributions (for the i386 architecture) using the sites redhat.com and rpmfind.net. We then wrote several custom scripts to analyze their contents and pinpoint potential validation problems.

Inhomogeneities

Because of the computational difficulty of reading large distributions, we broke our analysis into several short scripts, each of which provides input to the next. Our first Perl script tree reads all RPM files in an RPM distribution and outputs all dependency and checksum information from the distribution as a text file. This data is then read by a second script, munch, which computes a list of files that are inhomogeneously provided by more than one package, as indicated by differing MD5 signatures for the exact same file path. We then went over these inhomogeneities using a filtering script punch to eliminate conflicts based upon hardware differences (i386 vs. i686) where needed. This was done based upon the naming conventions for RPM files.

Results of this process differed greatly depending upon where we tried to do it. RedHat 7.2 (i386) exposed no conflicts whatsoever. RedHat 6.2 (i386) exposed 14 conflicts, of which one was a difference between a software bundle and an individual package; three were due to packages that are mutually exclusive, e.g., mail delivery agents; six were due to multiple versions of the same software, and the remaining four were due to unforeseen and simple packaging mistakes. Each ''conflict'' is a system file that has multiple versions listed in the RPM repository, where there may be up to five versions available for a single conflict.

But it should be no surprise that as a result of performing this process on the contrib directory, we found 3499 inhomogeneities distributed as in Table 4. The majority of the problems were due to the presence of multiple versions of the same software, sometimes with recognizable naming patterns, sometimes not.

# Errors	Kind of Error
3095	differing versions of the same software.
238	file version conflicts in apache modules.
80	software packages are mutually exclusive by design.
52	software bundle disagrees with individual tool package.
25	inconsistent versions in overlapping software bundles.
9	other conflict

Table 4: Inhomogeneities found in contrib directory of rpmfind.net.

Apache contributed modules were a source of great chaos; conflicts encompassed everything from HTML and GIFs to dynamic libraries as described in Table 5. One big surprise is that a single apache add-on, php, was responsible for 125 of the 238 file version conflicts for apache modules. Most of this was due to replacing – for no reason apparent to us – much of the HTML documentation for apache itself inside apache_php3-1.3b6-1.i386.rpm. This is a classic case of ''repository poisoning;'' one RPM creates inconsistencies that affect several others.

Another much more potentially serious problem is that there were 36 dynamic libraries with multiple versions, as listed in Table 6. These are the libraries loaded by Apache httpd itself. This was due to the contents of only five modules as described in Table 7. Each of these modules contained copies of between 32 and 36 dynamic libraries.

# Conflicts	Kind of Conflict
113	HTML documentation (.html)
36	dynamic libraries (.so)
32	manual pages (.1-.8)
30	header files (.h)
10	executables
4	configuration files
3	other

Table 5: Kinds of file conflicts in Apache.

The remainder of the inhomogeneities were due to several kinds of problems. Several packages were mutually exclusive by design, e.g., mail delivery agents or service daemons for the same service. 52 times, a software bundle disagreed with the package containing an individual tool added to the bundle. 25 files were inconsistent among two or more different software bundles. nine conflicts were simply unforeseen couplings between files and modules, e.g., expect-5.31.2-2.rh6.1.i386.rpm surprisingly instantiates /usr/bin/rftp along with socks-4.3.beta2-2.i386.rpm.

Binary Differences

At a more detailed level, while executables and libraries that are exact binary copies of others are functionally identical, executables and libraries that exhibit *binary* differences may or may not be interchangeable. Binary differences are evidence that there may be a functional difference, but this functional difference may or may not exist when the programs are executed.

Binary differences between files can also occur for completely gratuitous reasons. One can use two different compilers to compile the same .c file to get two object files that are different in binary but identical in text *and* function. As indicated by the above analysis, validation of code is not invariant of choice of compiler.

Our goal was to look at the grouping of RPMs available to us and determine which executables and libraries showing binary differences were possibly problematic and not caused by gratuitous metadata. The basic issue is whether two files that differ do so in a way that changes the behavior of programs. The files upon which we concentrated are all the Extensible Link Format [8] files in a linux system, including executables and dynamic libraries (.so).

We compared two different versions of the same file through a C program (provided by our advisor Alva Couch) that compares the binary contents of the text, data, and bss segments of an ELF [8] file. If two ELF files – executables, libraries, etc. – do not differ in text, data, and bss segments, then they are functionally equivalent, even if they differ in other metadata such as date, compiler version, etc.

# Versions	Dynamic Library
5	/usr/lib/apache/mod_access.so
5	/usr/lib/apache/mod_actions.so
5	/usr/lib/apache/mod_alias.so
5	/usr/lib/apache/mod_asis.so
5	/usr/lib/apache/mod_auth.so
5	/usr/lib/apache/mod_auth_anon.so
5	/usr/lib/apache/mod_auth_db.so
5	/usr/lib/apache/mod_autoindex.so
5	/usr/lib/apache/mod_cern_meta.so
5	/usr/lib/apache/mod_cgi.so
5	/usr/lib/apache/mod_digest.so
5	/usr/lib/apache/mod_dir.so
5	/usr/lib/apache/mod_env.so
5	/usr/lib/apache/mod_example.so
5	/usr/lib/apache/mod_expires.so
5	/usr/lib/apache/mod_headers.so
5	/usr/lib/apache/mod_imap.so
5	/usr/lib/apache/mod_include.so
5	/usr/lib/apache/mod_info.so
5	/usr/lib/apache/mod_mime.so
5	/usr/lib/apache/mod_mime_magic.so
5	/usr/lib/apache/mod_mmap_static.so
5	/usr/lib/apache/mod_negotiation.so
5	/usr/lib/apache/mod_rewrite.so
5	/usr/lib/apache/mod_setenvif.so
5	/usr/lib/apache/mod_speling.so
5	/usr/lib/apache/mod_status.so
5	/usr/lib/apache/mod_userdir.so
5	/usr/lib/apache/mod_usertrack.so
4	/usr/lib/apache/libproxy.so
4	/usr/lib/apache/mod_log_agent.so
4	/usr/lib/apache/mod_log_config.so
4	/usr/lib/apache/mod_log_referer.so
4	/usr/lib/apache/mod_unique_id.so
3	/usr/lib/apache/mod_bandwidth.so
2	/usr/lib/apache/mod_vhost_alias.so

Table 6: Number of multiple versions of each Apache dynamic library.

# Lib's	RPM file
34	apache-fp-1.3.3-1.i386.rpm
36	apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
36	apache-php3perl-1.3.12-3nosyb.i386.rpm
32	apache-ssl-jserv-1.3.2-2.i386.rpm
35	apache_modperl-1.3.6-1.19-1.i386.rpm

Table 7: Apache RPMs asserting conflicting dynamic libraries.

The result of this analysis was that of the 981 inhomogeneities in ELF format, only 13 of these turned out to be functionally equivalent, and 10 of these equivalences arose from two differing revisions of the same package. The remaining three were unusual; It turns out that the two RPMs:

- apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
- apache-php3perl-1.3.12-3nosyb.i386.rpm

have exactly equivalent copies of mod_unique_id.so, mod_log_referer.so, and mod_log_agent.so, for reasons we do not understand.

In summary, lack of equivalence is the rule. There were 968 inhomogeneities that were not able to be proven as functionally equivalent because they differ in text, data, or bss segment.

Update Skew

We know that RedHat validates the core distribution and each update, and have verified that their core RPMs are homogeneous in RedHat 7.2. But our analyses of contributed RPMs show that it is easy to *partially* update a system so that updated files are version-skewed with respect to one another.

In several cases, notably involving contributed versions of Apache, updates *overlap* in asserting new contents for particular libraries. RPMs assure that *backward* dependencies are satisfied (so that all libraries used by executables in the update are updated), but fail to update so that forward dependencies are satisfied. Any *other* pre-existing program that happens to use the same library is at risk of malfunctioning unless it is updated or validated against the new library as well.

In analyzing global dependencies in the RedHat distribution, updates, and contributed modules, we have observed two main kinds of failure of validation. Either an existing program is forced to use new libraries with unpredictable results, or the contents of a specific library depend upon installation order.

Case Study: Updating Apache

In Apache, the exact contents of a library depend upon the sequence of RPM installations. It is considered good practice to encapsulate apache updates, extensions, and modules into individual RPM packages, where each package contains an Apache module and all files that the module might potentially need [1,7]. This means, however, that many files in the main Apache package are duplicated among the update packages. If duplicated files are identical across all packages, there are few potential problems, but if they differ, we have reason to suspect that system states are attainable that have not been tested by anyone.

By a simple signature analysis, we found, e.g., that in the updates to RedHat 7.2, there are five different versions of /usr/lib/apache/mod_autoindex.so contained in five module RPMs:

- apache_modperl-1.3.6-1.19-1.i386.rpm
- apache-mod_ssl-fp2000-1.3.12.2.6.2-0.6.0.i386.rpm
- apache-ssl-jserv-1.3.2-2.i386.rpm
- apache-fp-1.3.3-1.i386.rpm
- apache-php3perl-1.3.12-3nosyb.i386.rpm

In principle, all copies of mod_autoindex.so should be identical in function, but comparison of md5 signatures shows that all these files *differ* in binary content. This means that there are five different states for this file on an updated system, depending upon which updated module is installed *last*.

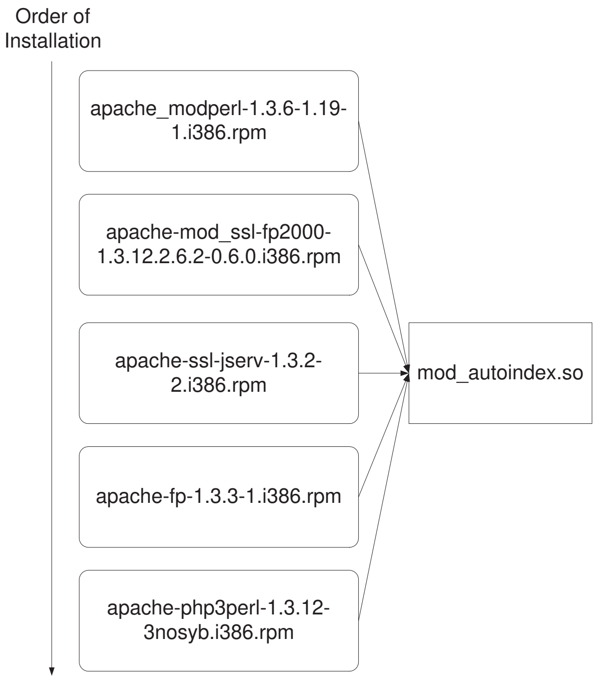

Figure 2: The unseen effects of updating apache: validation rot.

Figure 2 shows one possible configuration that could be used as an installation sequence for the five apache RPMs described above. The arrows from each RPM to mod_autoindex.so represent that RPMs installation of mod_autoindex.so. The problem here is that each RPM file has its own version of mod_autoindex.so. Thus, based on the order of installation and user input (about replacing files), the RPM that is installed last is the one that determines which version of mod_autoindex.so will be the one used by all of the packages once they are installed. This can cause the user to fall into an untested state and ultimately lead to a system failure.

Now in an ideal world, all that might differ in the various copies of mod_autoindex.so is "circumstantial" and does not affect behavior, e.g., the time that the source file was compiled. We actually *think* this is the case here, but have no easy way to validate contents against source code. This configuration situation represents a *risk* that one or more of these copies exhibits different behavior than the others. To be sure that a system works properly after updating, we need to know that the version that we have has been validated with the other modules that use this library.

Lessons Learned

Our preliminary analysis of the "contrib" branch of the RedHat distribution indicates that there are roughly 324 potential library version skew conflicts, without even considering supporting executables and scripts. We believe that an even more detailed analysis will expose many more skew conflicts. These conflicts overshadow the dependency errors in RPM declarations, which are statistically insignificant by comparison. While these types of errors may be statistically insignificant, they can be annoying and can cause major problems in some instances

Many administrators suffer from the illusion that one can install and manage packages in a relatively ad-hoc way at low cost. This illusion is shattered by considering the implications and cost of testing of ad-hoc configurations with the same rigor with which core distributions are tested. Testing is very costly, we consider it the vendors' responsibility, and yet we put our systems into states the vendor could not test and validate. If we fail to test the system ourselves, then the user of the system will inadvertently test it for us.

Using global analysis techniques, it is possible to predict whether one is moving to an untested configuration and to take corrective action so that one's system remains one that is widely used and tested. It is possible to minimize divergences between one's configurations and those used by the broader community, and to understand the cost of ad-hoc or divergent administrative methods.

Changing Practices

So what can we do differently to avoid these problems and assure that what we install will work properly? The key seems to be a different attitude and technique in creating and using RPMs. We can demand homogeneity in RPMs contributed by outsiders. We can analyze their dependencies and validate this homogeneity to some extent. We can minimize the effects of scripting by re-architecting RPMs and systems to have simpler script requirements, e.g., by creating directories rather than files, i.e., xinetd.conf rather than inetd.conf.

Avoiding gratuitous changes to validated baseline distributions will help ease the problems as well. We must look at changes more carefully before we make them. Each change not only represents a modification to our system but also pushes us farther from the baseline system. By constantly updating and changing our systems, we are moving farther and farther away from the baseline and a system that we know is validated. By viewing system changes as both updates and jumps from the baseline, we must make more informed decisions before we gratuitously install packages.

We can also avoid conflicts over dynamic libraries by making crucial libraries specific to the packages they serve. If a package needs a special dynamic library, name it differently than the normal one to avoid conflicts. This will help alleviate the problem of a library getting updated from one package when another package relies on the old version of it. By coupling packages with the libraries that are crucial to their proper functioning, we can remove a problem that is difficult to recognize prior to system failures. This strategy trades memory for robustness; one must often make a library effectively unshared in order to isolate it from interactions.

Avoiding update skews in large packages by updating coupled executables and libraries simultaneously will continue to solve the problem of multiple packages relying on a single library. When several packages rely on an important library, an upgrade of any one of them can create unpredictable behavior. Even more problematic is the differing behavior we observe depending on the order that packages are updated. By coupling update packages together, these problems will no longer have to be addressed.

But as system administrators, we need to be able to deal with these issues now without waiting for package development practices to change. We must fully understand the problems that our unique systems may face through global analysis. Understanding the causes of potential problems and recognizing warning signs when updating our systems will only help make our machines and systems more stable. Even without changing the way packages are developed, we can help to protect ourselves from the headaches of validation drift by using proper practices. For example, installation order of packages must be carefully observed as it is up to the system administrator to detect problematic conditions and assure the packages are installed in the proper order.

Many of the practices mentioned above trade space for validation. Nowadays, space in systems is cheap, while validation is expensive. By changing the practices we use in administering Linux systems, particularly with RPMs, we can save ourselves time and effort. The cost of this is the added space that multiple copies of libraries and packages may take up. But when you weigh the benefits versus the drawbacks, it is clear that a change in practices will help everyone.

Future Work

While the global analysis techniques upon which we report are *based* upon the strategies in sowhat, we have yet to integrate these into the tool proper. While we have a good grasp of the problem, a truly practical methodology seems to require this integration. We expect to do this some time in the coming year.

Further, we intend to look at the arbitrary script actions performed by RPMs before and after installation. These scripts can cause numerous things in a system to change; things that the user probably does not know are being changed. By looking at these scripts and comparing them against one another, we will be able to get an even better handle on exactly what an RPM is doing when updating a system in a baseline state.

But this work is just the tip of the iceberg. Homogeneity of packages is *necessary*, but not *sufficient*. A truly practical strategy would account for *all* dependencies; not just those one can discover with ldd, but also dependencies that can only be discovered by tracing library references. We and Yizhan Sun are also working on wrapping library calls to trace perhaps non-conventional use of dynamic libraries (using dlopen), and even tracing – at a fine grain – actual file use in a live system. These measures will give us a better idea of the true dependencies in a running system that can be violated by poor practice.

Open Questions and Controversies

This work also brings up some rather important open questions for study by the whole community of RPM users. These are not questions that we feel that we can address ourselves with the technology we have. We leave them to other researchers.

One of the most heated controversies in current systems management is whether binary equivalence is necessary for behavioral equivalence of programs, libraries, or systems [10, 11]. It seems that the community is strongly divided into factions of ''theorists'' and ''practitioners.'' Some ''practitioners'' believe that only identical binary files are guaranteed to behave identically (our premise) and that differing compilers, for example, cannot be trusted to compile the same source code with identical behavioral results. Some ''theorists'' believe, however, that ''we should be able to write compilers that perfect'' and that the source code should be the real measure of equivalence. Some extremists also argue that even differing source code can be proved behaviorally equivalent by compiler optimization techniques. We take the very conservative position that ''only practical techniques can be applied now'' and thus believe binary equivalence the only currently practical measure of behavioral equivalence. Only time will tell whether the other ideas of equivalence will be practical.

Another open question concerns the general nature of dependency. So far, we can only describe dependencies in a very coarse way, by saying which files should be present or which packages should be installed. Dependency, in general, is a much more complex thing. It creates limits on the *contents* of files as well as their presence and location. How far can we go with describing dependencies before the cure (of discovering and declaring dependencies) is worse than the disease (of dependency failure)?

Conclusions

We have shown that problems in RedHat installations are *not* always caused by problems with dependencies between packages, but instead (and perhaps more commonly) by *overlaps* between packages. Dependencies declared inside a typical collection of RPMs are surprisingly accurate. But overlaps between package files seem to be a plague upon both closed and open repositories containing reusable binary RPMs.

In casually installing RPMs in a Linux system, it is easily possible to put the system into a state that no one has validated or tested. While for a ''home computer'' the risk of down time is fairly low, in an enterprise management strategy, such ad-hoc system updates should be avoided in favor of staying near configurations that have been extensively tested and ''burned in'' by the community. We show that deviations from tested states can sometimes be detected before an RPM is installed by global analysis of all RPMs available. We also suggest that RPMs be constructed so that any combination, in any order, always results in a validated system state. This is easy to accomplish by isolating dependencies and avoiding inhomogeneous overlaps, but seemingly only the major distributions have managed to do this properly.

Acknowledgements

Our advisor Alva Couch helped us with the scripts, resources, and writing involved in completing this paper. We would also like to recognize the ongoing and much appreciated efforts of RedHat, RpmFind, and other organizations dedicated to making software package installation more efficient, predictable, and robust. We do not intend anything in this article as a criticism of their noble efforts. In this paper we have shown how hard their job really is.

Author Biographies

Jeffrey D'Amelia is a graduate student at Tufts University working towards his Masters degree which will be completed in the Spring of 2003. Beginning in January 2002, Jeff was awarded a fellowship in the NSF GK-12 program. Through this program, he works at a junior high school in Malden, MA with the goal of infusing computer science problem solving approaches into the K-12 math curriculum. He has also worked at the college level as both an undergraduate and graduate teaching assistant for several different courses. Jeff can be reached via U. S. Mail at Tufts University, Department of Computer Science, Halligan Hall, 161 College Ave., Medford, MA 02155 or electronically at jdamelia@eecs.tufts.edu

John Hart is a graduate student at Tufts University working towards his Masters degree which will be completed in the Spring of 2003. He received his Bachelor of Engineering degree from Tufts University in Computer Engineering and has worked as undergraduate and graduate teaching assistant. John can be reached via U. S. Mail at Tufts University, Department of Computer Science, Halligan Hall, 161 College Ave., Medford, MA 02155 or electronically at jhart@eecs.tufts.edu

References

[1] Bailey, Ed, ''Maximum RPM,'' SAMS, Inc., 1997.
[2] Bodnar, Ladislav, ''Is RPM Doomed?'', http://www.distrowatch.com/article-rpm.php .

[3] Brooks, Fredrick, "The Mythical Man-Month," Addison-Wesley, Inc, 1995.

[4] Couch, Alva and Yizhan Sun, "Global Analysis of Dynamic Library Dependencies," *Proceedings LISA 2001*, San Diego, CA, 2001.

[5] Couch, Alva, "The Maelstrom: Network Service Debugging via 'Ineffective Procedures'," *Proceedings LISA 2001*, San Diego, CA, 2001.

[6] Couch, Alva, "SLINK: Effective Abstractions for Community-Based Administration," *Proceedings LISA 96*, San Diego, CA, 1996.

[7] Hess, Joey, "A comparison of the deb, rpm, tgz, slp, and pkg package formats," http://www.kitenet.net/~joey/pkg-comp/ .

[8] Levine, John, "Linkers and Loaders," First Edition, Morgan Kaufmann Publishers, 1999.

[9] Pressman, Roger, "Software Engineering: A Practitioners' Approach," Fifth Edition, McGraw-Hill, Inc, 2001.

[10] Traugott, Steve and Joel Huddleston, "Bootstrapping an Infrastructure," *Proceedings LISA XII*, USENIX Association, 1998.

[11] Traugott, Steve, and Lance Brown, "Why Order Matters: Turing Equivalence in Automated Systems Administration," Proceedings LISA XVI, USENIX Association, 2002.

[12] Watkins, John, "Testing IT: an Off-The-Shelf Software Testing Process," Cambridge University Press, 2001.

[13] "Debian GNU/Linux," http://www.debian.org .

[14] The Linux Standard Base, "The Linux Standard Base Project," http://www.linuxbase.org .

[15] The Slackware Linux Project, http://www.slackware.com .