# Staging Package Deployment via Repository Management

Chris St. Pierre - stpierreca@ornl.gov
Matt Hermanson - mjhermanson@ornl.gov
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, USA*

**Abstract**

This paper describes an approach for managing package versions and updates in a homogenous manner across a heterogenous environment by intensively managing a set of software repositories rather than by managing the clients. This entails maintaining multiple local mirrors, each of which is aimed at a different class of client: One is directly synchronized from the upstream repositories, while others are maintained from that repository according to various policies that specify which packages are to be automatically pulled from upstream (and therefore automatically installed without any local vetting) and which are to be considered more carefully – likely installed in a testing environment, for instance – before they are deployed widely.

## Background

It is important to understand some points about our environment, as they provide important constraints to our solution.

We are lucky enough to run a fairly homogenous set of operating systems consisting primarily of Red Hat Enterprise Linux and CentOS servers, with fair numbers of Fedora and SuSE outliers. In short, we are dealing entirely with RPM-based packaging, and with operating systems that are capable of using yum [12]. As yum is the default package management utility for the majority of our servers, we opted to use yum rather than try to switch to another package management utility.

For configuration management, we chose to use Bcfg2 [3] for reasons wholly unrelated to package and software management. Bcfg2 is a Python and XML-based configuration management engine that "helps system administrators produce a consistent, reproducible, and verifiable description of their environment" [3]. It is in particular the focus on reproducibility and verification that forced us to consider updating and patching anew.

In order to guarantee that a given configuration – where a "configuration" is defined as the set of paths, files, packages, and so forth, that describes a single system – is fully replicable, Bcfg2 ensures that every package specified for a system is the latest available from that system's software repositories [8]. (As will be noted, this can be overridden by specifying an explicit package version.) This grants the system administrator two important abilities: to provision identical machines that will remain identical; and to reprovision machines to the exact same state they were previously in. But it also makes it unreasonable to simply use the vendor's software repositories (or other upstream repositories), since all updates will be installed immediately without any vetting. The same problem presents itself even with a local mirror.

Bcfg2 can also use "the client's response to the specification ... to assess the completeness of the specification" [3]. For this to happen, the Bcfg2 server must be able to understand what a "complete" specification entails, and so the server does not entirely delegate package installation to the Bcfg2 client. Instead, it performs package dependency resolution on the server rather than allowing the client to set its own configuration. This necessitates ensuring that the Bcfg2 Packages plugin uses the same

yum configuration as the clients; Bcfg2 has support for making this rather simple [8], but the Packages plugin does not support the full range of yum functionality, so certain functions like the "versionlock" plugin and even package excludes, are not available. Due to the architecture of Bcfg2 – architecture designed to guarantee replicability and verification of server configurations – it is not feasible or, in most cases, possible to do client-based package and repository management. This became critically important in selecting a solution.

# Other Solutions

There are a vast number of potential solutions to this problem that would seem to be low-hanging fruit – far simpler to implement, at least initially, than our ultimate solution – but that would not work, for various reasons.

## Yum Excludes

A core yum feature is the ability to exclude certain packages from updates or installation [13]. At first, this would seem to be a solution to the problem of package versioning: simply install the package version you want, and then exclude it from further updates. But this has several issues that made it unsuitable for our use (or, we believe, this use case in general):

- It does not (and cannot) guarantee a specific version. Using excludes to set a version depends on that version being installed (manually) prior to adding the package to the exclude list.
- There is no guarantee that the package is still in the repository. Many mainstream repositories[1] do not retain older versions in the same repository as current packages. Consequently, when reinstalling a machine where yum excludes have been used to set package versions (or when attempting to duplicate such a machine), there is no guarantee that the package version expected will even be available.
- In order to use yum excludes to control package versions, a very specific order of events must occur: first, the machine must be installed without the target package included (as Kickstart, the RHEL installation tool, does not support installing a specific version of a package [1]); next, the correct package version must be installed; and finally, the package must be added to the exclude list. If this happens out of order, then the wrong version of the package might be installed, or the package might not be installed at all.
- Supplying a permitted update to a package is even more difficult, as it involves removing the package exclusion, updating to the correct version, and then restoring the exclusion. A configuration management system would have to have tremendously granular control over the order in which actions are performed to accomplish this delicate goal.
- As discussed earlier, Bcfg2 performs dependency resolution on the server side in order to provide a guarantee that a client's configuration is fully specified. By using yum excludes – which cannot be configured in Bcfg2's internal dependency resolver – the relationship between the client and the server is broken, and Bcfg2 will in perpetuity claim that the client is out of sync with the server, thus reducing the usefulness of the Bcfg2 reporting tools.

While yum excludes appear at first to be a viable option, their use to set package versions is not replicable, consistent, and cannot be trivially automated.

## Specifying Versions in Bcfg2

Bcfg2 is capable of specifying specific versions of packages in the specification, e.g.:

```
<BoundPackage name="glibc" type="yum">
  <Instance version="2.13" release="1"
            arch="i686"/>
  <Instance version="2.13" release="1"
            arch="x86_64"/>
</BoundPackage>
```

This is obviously quite verbose (more so because the example uses a multi-arch package), and as a result of its verbosity it is also error-prone. Having to recopy the version, release, and architecture of a package – separately – is not always a trivial process, and the relatively few constraints of version and release strings makes it less so. For instance, given the package:

```
iomemory-vsl-2.6.35.12-88.fc14.x86_64-
    2.3.0.281-1.0.fc14.x86_64.rpm
```

The package name is "iomemory-vsl-2.6.35.12-88.fc14.x86_64" (which refers to the specific kernel for which it was built), the version is "2.3.0.281" and the release is "1.0.fc14".[2] This can be clarified through use of the `--queryformat` option to `rpm`, but the fact that more advanced RPM commands are necessary makes it clear that this approach is untenable in general. Even more worrisome is the package epoch, a sort of "super-version," which RPM cleverly hides by default, but could cause a newer package to be installed if it was not specified properly.

Maintenance is also tedious, as it involves endlessly updating verbose version strings; recall that a given version is just shorthand for what we actually care about – that a package *works*.

This approach also does not abrogate the use of yum on a system to update it beyond the appropriate point. The only thing keeping a package at the chosen version is Bcfg2's own self-restraint; if an admin on a machine lacks that same self-restraint, then he or she could easily update a package that was not to be updated, whereupon Bcfg2 would try to downgrade it.

Finally, this approach presents specific difficulties for us, as our adoption of Bcfg2 is far from complete; large swaths of the center still use Cfengine 2, and some machines – particularly compute and storage platforms – operate in a diskless manner and do not use configuration management tools in a traditional manner. They depend entirely on their images for package versions, so specifying versions in Bcfg2 would not help.

To clarify, using Bcfg2 forced us to reconsider this problem, and any solution must be capable of working with Bcfg2, but it cannot be assumed that the solution may leverage Bcfg2.

### Yum versionlock

Using yum's own version locking system would appear to improve upon pegging versions in Bcfg2: it works on all systems, regardless of whether or not they use Bcfg2; and a shortcut command, `yum versionlock <package-name>`, is provided to make the process of maintaining versions less error-prone.[3]

It also solves many of the problems of yum excludes, but suffers from a critical flaw in that approach: by setting package versions on the client, the relationship between the Bcfg2 client and server would be broken.

Combinations of these three approaches merely exhibit combinations of their flaws. For instance, the promising combination of yum's versionlock plugin and specifying the version in Bcfg2 would ensure that the Bcfg2 client and server were of a mind about package versions, and would work on non-Bcfg2 machines; however, it would forfeit versionlock's ease of use and require the administrator to once again manually copy package versions.

### Spacewalk

Spacewalk was the first full-featured solution we looked at that aims to replace the mirroring portion of this relationship; all of the other potential solutions listed thus far have attempted to work with a "dumb" mirror and use yum features to work around the problem we have described. Spacewalk is a local mirror system that "manages software content updates for Red Hat derived [*sic*] distributions" [10]; it is a tremendously full-featured system, with support for custom "channels," collections of packages assembled in an ad-hoc basis.

Unfortunately, Spacewalk was a non-starter for us for the same reason that it has failed to gain much traction in the community at large: of the two versions of Spacewalk, only the Oracle version actually implements all of the features; the PostgreSQL version is deeply underfeatured, even after several years of work by the Spacewalk team to port all of the Oracle stored procedures.

As it turns out, Red Hat has a successor in mind for Spacewalk and Satellite: CloudForms [14]. The content management portion of CloudForms – roughly corresponding to the mirror and repository management functionality of Spacewalk – is Pulp.

## A solution: Pulp

Pulp is a tool "for managing software repositories and their associated content, such as packages, errata, and distributions" [7]. It is, as noted, the spiritual successor to Spacewalk, and so implements the vast majority of Spacewalk's repository management features without the dependency on Oracle.

Pulp's usage model involves syncing multiple upstream repositories locally; these repositories can then be *cloned*, which uses hard links to sync them locally with almost no disk space used. This allows us to sync a repository once, then duplicate it as many times as necessary to support multiple teams and multiple stability levels. The sync process supports *filters*, which allow us to blacklist or whitelist

packages and thus exclude "impactful" packages from automatic updates.

Pulp also supports manually adding packages to and removing packages from repositories, so we can later update a given package across all machines that use a repository with a single command. Adding and removing also tracks dependencies, so it's not possible to add a package to a repository without adding the dependencies necessary to install it.[4]

## Workflow

Pulp provides us with the framework to implement a solution to the problem outlined earlier, but even as featureful as it is it remains a fairly basic tool. Our workflow – enforced by the features Pulp provides, by segregating repositories, by policy, and by a nascent in-house web interface – provides the bulk of the solution. Briefly, we segregate repositories by tier to test packages before site-wide roll-outs, and by team to ensure operational separation. Packages are automatically synced between tiers based on package filters, which blacklist certain packages that must be promoted manually. This ensures that most packages benefit from up to two weeks of community testing before being deployed site-wide, and packages that we have judged to be more potentially "impactful" from more focused local testing as well.

### Tiered Repositories

We maintain different repository sets for different "levels" of stability. We chose to maintain three tiers:

`live` Synced daily from upstream repositories; not used on any machines, but maintained due to operational requirements within Pulp[5] and for reference.

`unstable` Synced daily from `live`, with the exception of selected "impactful" packages (more about which shortly), which can be manually promoted from `live`.

`stable` Synced daily from `unstable`, with the exception of the same "impactful" packages, which can be manually promoted from `unstable`.

This three-tiered approach guarantees that packages in `stable` are at least two days old, and "impactful" packages have been in testing by machines using the `unstable` branch. When a package is released from upstream and sync to public mirrors, those packages are pulled down into local repositories. From then on the package in under the control of Pulp. Initially, a package is considered unstable and is only deployed to those systems that look at the repositories in the `unstable` tier. After a period of time, the package is then promoted into the `stable` repositories, and thus to production machines.

In order to ensure that packages in `unstable` receive ample testing before being promoted to `stable`, we divide machines amongst those two tiers thusly:

- All internal test machines – that is, all machines whose sole purpose is to provide test and development platforms to customers within the group – use the `unstable` branch. Many of these machines are similar, if not identical, to production or external test machines.
- Where multiple identical machines exist for a single purpose, whether in an active-active or active-passive configuration, exactly one machine will use the `unstable` branch and the rest will use the `stable` branch.

Additionally, we maintain separate sets of repositories, branched from `live`, for different teams or projects that require different patching policies appropriate to the needs of those teams or projects. Pulp has strong built-in ACLs that support these divisions.

In order to organize multiple tiers across multiple groups, we use a strict convention to specify the repository ID, which acts as the primary key across all repositories[6], namely:

```
<team name>-<tier>-<os name>-<os version>-
    <arch>-<repo name>
```

For example,
`infra-unstable-centos-6-x86_64-updates` would denote the Infrastructure team's `unstable` tier of the 64-bit CentOS 6 "updates" repository. This allows us to tell at a glance the parent-child relationships between repositories.

### Sync Filters

The syncs between the `live` and `unstable` and between `unstable` and `stable` tiers are mediated by filters[7]. Filters are regular expression lists of packages to either blacklist from the sync, or whitelist in the sync; in our workflow, only blacklists are used. A package filtered from the sync may still remain in the

repository; that is, if we specify `^kernel(-.*)?` as a blacklist filter, that does not remove `kernel` packages from the repository, but rather refuses to sync new `kernel` packages from the repository's parent. This is critical to our version-pegging system.

Given our needs, whitelist filters are unnecessary; our systems tend to fall into one of two types:

- Systems where we generally want updates to be installed insofar as is reasonable, with some prudence about installing updates to "impactful" packages.
- Systems where, due to vendor requirements, we must set all packages to a specific version. Most often this is in the form of a requirement for a minor release of RHEL[8], in which case there are no updates we wish to install on an automatic basis. (We may wish to update specific packages to respond to security threats, but that happens with manual package promotion, not with a sync; this workflow gives us the flexibility necessary to do so.)

A package that may potentially cause issues when updated can be blacklisted on a per-team basis[9]. Since the repositories are hierarchically tiered, a package that is blacklisted from the `unstable` tier will never make it to the `stable` tier.

### Manual Package Promotion and Removal

The lynchpin of this process is manually reviewing packages that have been blacklisted from the syncs and *promoting* them manually as necessary. For instance, if a filter for a set of repositories blacklisted `^kernel(-.*)?` from the sync, without manually promoting new kernel packages no new kernel would ever be installed.

To accomplish this, we use Pulp's *add package* functionality, exposed via the REST API as a POST to `/repositories/<id>/add_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.add_package()`, and via the CLI as `pulp-admin repo add_package`. In the CLI implementation, `add_package` follows dependencies, so promoting a package will promote everything that package requires that is not already in the target repository. This helps ensure that each repository stays consistent even as we manipulate it to contain only a subset of upstream packages[10].

Conversely, if a package is deployed and is later found to cause problems it can be removed from the tier and the previous version, if such is available in the repository, will be (re)installed. Bcfg2 will helpfully flag machines where a newer package is installed than is available in that machine's repositories, and will try to downgrade packages appropriately. Pulp can be configured to retain old packages when it performs a sync; this is helpful for repositories like EPEL that remove old packages themselves, and guarantees that a configurable number of older package versions are available to fall back on.

The *remove package* functionality is exposed via Pulp's REST API as a POST to `/repositories/<id>/delete_package/`, via the Python client API as `pulp.client.api.repository.RepositoryAPI.remove_package()`, and via the CLI as `pulp-admin repo remove_package`. As with `add_package`, the CLI implementation follows dependencies and will try to remove packages that require the package being removed; this also helps ensure repository consistency.

Optimally, security patches are applied 10 or 30 days after the initial patch release [2]; this workflow allows us to follow these recommendations to some degree, promoting new packages to the `unstable` tier on an approximately weekly basis. Packages that have been in the `unstable` tier for at least a week are also promoted to the `stable` tier every week; in this we deviate from Beattie et al.'s recommendations somewhat, but we do so because the updates being promoted to `stable` have been vetted and tested by the machines using the `unstable` tier.

This workflow also gives us something very important: the ability to install updates across all machines much sooner than the optimal 10- or 30-day period. High profile vulnerabilities require immediate action – even to the point of imperiling uptime – and by promoting a new package immediately to both `stable` and `unstable` tiers we can ensure that it is installed across all machines in our environment in a timely fashion.

### Selecting "impactful" packages

Throughout this paper, we have referred to "impactful" packages – those to which automatic updates we determined to be particularly dangerous – as a driving factor. Were it not for our reticence to automatically update all packages, we could have simply used an automatic update facility – `yum-cron` or

`yum-updatesd` are both popular – and been done with it.

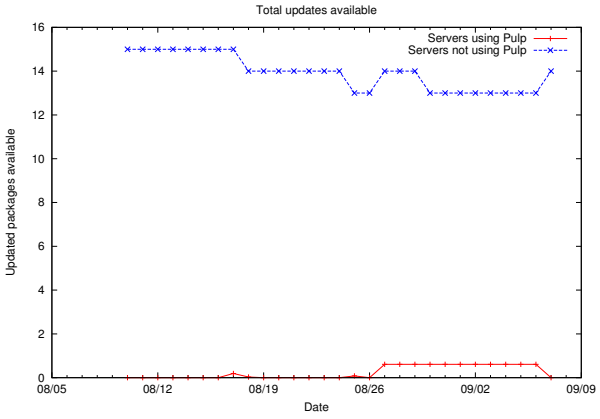
We didn't feel that was appropriate, though. For instance, installing a new kernel can be problematic – particularly in an environment with a wide variety of third-party kernel modules and other kernel-space modifications – and we wanted much closer control over that process. We flagged packages as "impactful" according to a simple set of criteria:

- The kernel, and packages otherwise directly tied to kernel space (e.g., kernel modules and Dynamic Kernel Module Support (DKMS) packages);
- Packages that provide significant, customer-facing services. On the Infrastructure team, this included packages like `bind`, `httpd` (and related modules), `mysql`, and so on.
- Packages related to InfiniBand and Lustre [9]; as one of the world's largest unclassified Lustre installations, it's very important that the Lustre versions on our systems stay in lockstep with all other systems in the center. Parts of Lustre reside directly in kernel space, an additional consideration.

The first two criteria provided around 20 packages to be excluded – a tiny fraction of the total packages installed across all of our machines. The vast majority of supporting packages continue to be automatically updated, albeit with a slight time delay for the multiple syncs that must occur.

## Results

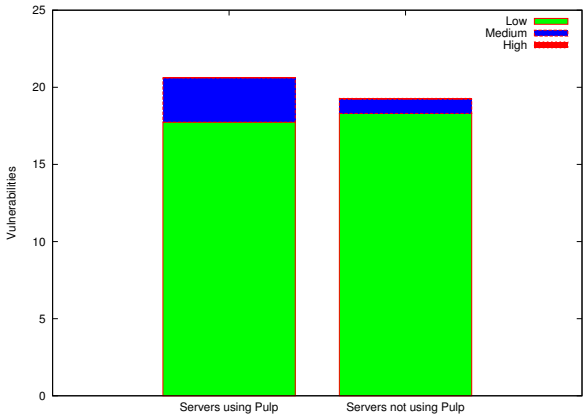
Our approach produces results in a number of areas that are difficult to quantify: improved automation reduces the amount of time we spend installing patches; not installing patches immediately improves patch quality and reduces the likelihood of flawed patches [2]; and increased compartmentalization makes it easier for our diverse teams to work to different purposes without stepping on toes. But it also provides testable, quantifiable improvements: since replacing a manual update process with Pulp and Bcfg2's automated update process, we can see that the number of available updates has decreased and remained low on the machines using Pulp.




The practice of staging package deployment makes is difficult to quantify just how out of date a client is, as yum on the client will only report the number of updates available from the repositories in `yum.conf`. To find the number of updates available from upstream, we collect an aggregate of all the package differences starting at the client and going up the heirarchy to the upstream repository. E.g., for a machine using the `unstable` tier, we calculate the number of updates available on the machine itself, and then the number of updates available to the `unstable` tier from the `live` tier.

The caveat to this approach is when, for instance, a package splits into two new packages. This results in two new packages, and one missing package, totaling three "updates" according to `yum check-update`, or zero "updates" when comparing repositories themselves, when in reality it is a single package update. For example, if package `foo` recieves an update that results in packages `foo-client` and `foo-server`, this could result in a margin of error of -1 or +2. This gives a slight potential benefit to machines using Pulp in our metrics, as updates of this sort are underestimated when calculating the difference between repositories, but overestimated when using `yum` to report on updates available to a machine. In practice, this is extremely rare, though, and should not significantly affect the results.

Ensuring, with a high degree of confidence, that updates are installed is wonderful, but even more important is ensuring that vulnerabilities are being mitigated. Using the data from monthly Nessus [11] vulnerability scans, we can see that machines using Pulp do indeed reap the benefits of being patched with more frequency:[11]




This graph is artificially skewed against Pulp due to the sorts of things Nessus scans for; for instance, web servers are more likely to be using Pulp at this time simply due to our implementation plan, and they also have disproportionately more vulnerabilities in Nessus because they have more services exposed.

# Future Development

## Sponge

At this time, Pulp is very early code; it has been in use in another Red Hat product for a while, so certain paths are well-tested, but other paths are pre-alpha. Consequently, its command line interface lacks polish, and many tasks within Pulp require extraordinary verbosity to accomplish. It is also not clear if Pulp is intended for standalone use, although such is possible.

To ease management of Pulp, we have written a web frontend for management of Pulp and its objects, called "Sponge." Sponge, powered by the Django [4] web framework, provides views into the state of Pulp repositories along with the ablity to manage its contents. Sponge leverages Pulp's Python client API to provide convience functions that ease our workflow.

By presenting the information visually, Sponge makes repository management much more intuitive. Sponge extends the functionality of Pulp by displaying the differences between a repository and its parent in the form of a diff. These diffs give greater insight into exactly how `stable`, `unstable`, and `live` tiers differ. They also provide insight into the implications of a package promotion or removal.

This is particularly important with package removal, since, as noted, removing a package will also remove anything that requires that specific package. Without Sponge's diff feature and a confirmation step, that is potentially very dangerous; Pulp itself only gives you confirmation of the packages removed without an opportunity to confirm or reject a removal. The contrapositive situation – promoting a package pulling in unintended dependencies – is also potentially dangerous, albeit less so. Sponge helps avert both dangers.

## Guaranteeing a minimum package age

As Beattie at al. observe [2], the optimal time to apply security patches is either 10 or 30 days after the patches have been released. Our workflow currently doesn't provide any way to guarantee this; our weekly manual promotion of new packages merely suggests that a patch be somewhere between 0 and 6 days old before it is promoted to `unstable`, and 7 and 13 days old before being promoted to `stable`. We plan to add a feature – either to Sponge or to Pulp – to promote packages only once they have aged properly.

## Other packaging formats

In this paper we have dealt with systems using yum and RPM, but the approach can, at least in theory, be expanded to other packaging systems. Pulp intends eventually to support not only Debian packages, but actually any sort of generic content at all [6], making it useful for any packaging system. Bcfg2, for its part, already has package drivers for a wide array of packaging systems, including APT, Solaris packages (Blastwave- or SystemV-style), Encap, FreeBSD packages, IPS, Mac Ports, Pacman, and Portage. This gives a hint of the future potential for this approach.

# Availability

Most of the software involved in the approach discussed in this paper is free and open source. The various elements of our solution can be found at:

**Pulp** `http://pulpproject.org`

**Bcfg2** `http://trac.mcs.anl.gov/projects/bcfg2`

**Yum** `http://yum.baseurl.org/`

Sponge, the web UI to Pulp listed in the Future Development section, is currently incomplete and unreleased. We have already worked closely with the Pulp developers to incorporate features into the Pulp core itself, and we will continue to do so. We hope that Sponge will become unnecessary as Pulp matures.

## Author Information

Chris St. Pierre leads the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He is deeply involved with the development of Bcfg2, contributing in particular to the specification validation tool and Packages plugin for the upcoming 1.2.0 release. He has taught widely on internal documentation, LDAP, and spam. Chris serves on the LOPSA Board of Directors.

Matt Hermanson is a member of the Infrastructure team of the HPC Operations group at the National Center for Computational Sciences at Oak Ridge National Laboratory in Oak Ridge, Tennessee. He holds a B.A. in Computer Science from Tennessee Technological University.

## References

[1] Anaconda/Kickstart. `http://fedoraproject.org/wiki/Anaconda/Kickstart#Chapter_3._Package_Selection`.

[2] Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., and Shostack, A. Timing the Application of Security Patches for Optimal Uptime. Proceedings of LISA '02: Sixteenth Systems Administration Conference, USENIX, pp. 233–42.

[3] Desai, N. Bcfg2. `http://trac.mcs.anl.gov/projects/bcfg2`.

[4] Django Software Foundation. Django — The Web framework for perfectionists with deadlines. `https://www.djangoproject.com/`.

[5] Dobies, J. GCRepoApis. `https://fedorahosted.org/pulp/wiki/GCRepoApis`.

[6] Dobies, J. Generic Content Support. `http://blog.pulpproject.org/2011/08/08/generic-content-support/`.

[7] Dobies, J. Pulp - Juicy software repository management. `http://pulpproject.org`.

[8] Jerome, S., Laszlo, T., and St. Pierre, C. Packages. `http://docs.bcfg2.org/server/plugins/generators/packages.html`.

[9] Oracle Corporation. Lustre. `http://wiki.lustre.org/index.php/Main_Page`.

[10] Red Hat, Inc. Spacewalk: Free & Open Source Linux Systems Management. `http://spacewalk.redhat.com/`.

[11] Tenable Network Security. Tenable Nessus. `http://www.tenable.com/products/nessus`.

[12] Vidal, S. yum. `http://yum.baseurl.org/`.

[13] Vidal, S. yum.conf - configuration file for yum(8). `man 5 yum.conf`.

[14] Warner, T., and Sanders, T. The Future of RHN Satellite: A New Architecture Enabling the Traditional Data Center and the Cloud. Red Hat Summit, Red Hat, Inc.

## Notes

[1] For instance, Extra Packages for Enterprise Linux (EPEL) and the CentOS repositories themselves.

[2] Admittedly, this is a non-standard naming scheme, but no solution can be predicated on the idea that all RPMs are well-built.

[3] The command in question merely maintains a local file on a machine, so that file would still have to be copied into the Bcfg2 specification, but we believe this would be less error-prone than copying package version details.

[4] This is actually only true if the package is being added from another repository; it is possible to add a package directly from the filesystem, in which case dependency checking is not performed. This is not a use case for us, though.

[5] In Pulp, filters can only be applied to repositories with local feeds.

[6] This may change in future versions of Pulp, as multiple users, ourselves included, have asked for stronger grouping functionality [5].

[7] As noted earlier, in Pulp, filters can only be applied to repositories with local feeds, so no filter mediates the sync between upstream and `live`.

[8] It is lost on many vendors that it is unreasonable and foolish to require a specific RHEL minor release. As much work as has gone into this solution, it is still less than would be required to convince most vendors of this fact, though.

[9] Technically, filters can be applied on a per-repository basis, so black- and whitelists can be applied to individual repositories. This is very rare in our workflow, though.

[10] It is true that our approach does not *guarantee* consistency. A repository sync might result in an inconsistency if a package that was not listed on that sync's blacklist required a package that was listed on the blacklist. In practice this can be limited by using regular expressions to filter families of packages (e.g., `^mysql.*` or `^(.*-)?mysql.*` to blacklist all MySQL-related packages rather than just blacklisting the `mysql-server` package itself

[11] Unfortunately long-term data was not available for vulnerabilities for a number of reasons: CentOS 5 stopped shipping updates in their mainline repositories between July 21st and September 14th; the August security scan was partially skipped; and Pulp hasn't been in production long enough to get meaningful numbers prior to that. Still, the snapshot of data is compelling.