Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*

Non Privileged User Package Management: Use Cases, Issues, Proposed Solutions

François-Denis Gonthier
Kryptiva, Inc.
fdgonthier@kryptiva.com

Steven Pigeon
École de Technologie Supérieure
Département de Génie Logiciel et
des Technologies de l'Information
spigeon@etsmtl.ca

Abstract

The package manager and the associated repositories play a central role in the usability and stability of user environments in GNU/Linux distributions. However, the current package management paradigm puts the control of the system exclusively in the hands of the system administrator, a root-like user. The non privileged user must rely on the administrator to install the packages he needs, while having to deal with delays or even refusal. We think that *non privileged package management* is the solution to the users' woes. We show that not only non privileged package management has realistic use cases, but also that it is quite feasible. We examine several possible existing solutions and show how they cannot be satisfactory for the deployment of unprivileged user package management. Finally, we analyse the dpkg package manager and show how it can be extended to include safe, consistent, non privileged user package management. Amongst results, we present the conflict resolution rules to include multiple databases, to ensure system consistence and proper dependency management. We also present how to modify user environment initialization to include alternate install locations. We show, finally, the feasibility and usefulness of unprivileged user package management and how small the changes to be made to a package manager such as dpkg are.

1 Introduction

Non privileged user package management is not considered as an important use case in package management. Package management focuses mainly on security and system stability, relying on a centralized model where control lies in the hands of the administrator(s). This model, essentially the only one used in Linux distributions, relies on the implicit assumption that users cannot manage their environment in any meaningful way, except for minor tweaks and configurations, and that decisions regarding packages can only be taken by administrators. We argue that while this model has proved itself effective, there is room for the users to manage their environments beyond mere tweaks.

The problem of non privileged user package management does not present itself when the user is the owner and administrator of the machine but poses itself clearly when the user is but one of many users of shared workstations or of a multi-user server for which he does not possess administrative privileges. In this case, the user must ask the administrator for the packages he needs, and his request may very likely be denied, either because the package is against the local policies, because the package conflicts with other packages on the system, because it would affect adversely the other users, because it represents a security risk, or even because the administrator decides that the benefits to the user from adding the requested packages are not worth the effort. Whatever the reasons, the net result is that the user does not get the needed packages and is left at his own devices.

Being left at his own devices, the user will simply resort to installing the needed software from tarballs, downloaded from some location without authentication. Launching the `configure` script, by specifying install location (usually through the `--prefix` switch), building the software using the created Makefile, and modifying his environment variables, he will eventually succeed in locally installing the software.

The tarball approach suffers from a number of important drawbacks, despite being the standard for develop-

ment builds. First, one must deal manually with the missing dependencies of the software being compiled and installed. As most software do not quite fail building but merely disable features when dependencies are missing, it is every difficult for novice users—and even more advanced ones—to configure software correctly. Eventually, after fetching and building all needed dependencies, the user manages to compile, install, and to get the software to run correctly, but he still faces the onus of making updates by himself, going through the tarballing all over again at each new release.

It is our opinion that users should be able to locally install software through the much simpler process of using the distribution-specific package manager, thus obtaining all the benefits of centralized, trusted, and simple to access distribution-specific repositories—as exist for all major distributions—that provide packages with all their dependencies. In addition to the simplified installation process, the user should get updates automatically through the distribution's update manager, without so much intervention as a simple confirmation.

However, non privileged user package management is seemingly a complex problem, and one can oppose it several objections (that we discuss in Section 2.2) of which security is the most obvious and the most serious. Indeed, one cannot grant any user the right to install any package as it may affect adversely other users or even render the system inoperable. Delegating package management via a simple mechanism like the sudoers list both is dangerous and insufficient.

The correct solution is therefore, in our opinion, to use *fully relocatable packages* and to allow *non privileged users* to perform private installs in their home directories (or some other accessible location) through the package manager. This implies that there is a local package database for each user, and that the package manager must maintain coherence between the user's and the system's databases, being fully capable of detecting and resolving package conflicts.

In this paper, we study the problem of non privileged user package management. We outline the problem, as we see it, and present venues for solutions. The paper is organised as follows. Section 2 presents the proposed use cases as well as possible objections to the proposed model. Section 3 reviews the existing strategies to delegate package management as well as existing package managers, and discusses their shortcomings. In Section 4 we outline our proposed solution using dpkg and apt as example implementations. Finally, we conclude in Section 5.

2 Non Privileged User Managed Packages

In this section, we propose use cases for non privileged user package management and discuss possible objections to the extension of classical package management to include non privileged user operations.

But before continuing, let us present the definitions of the terms we will be using in the remainder of this text and that should not be ambiguous to the reader.

The *administrator* is a special user with access to all of the system's files and programs. Amongst other things, an administrator is a user that can manage packages on a system. The administrator is a user with root privileges. The *user*, on the other hand, is a user with no administrative rights. His privileges are limited to his personal files, programs made available to him, and he cannot install packages on the system.

In a virtualization context, the *host* is the machine or system that provides CPU and shared resources to one or more guests operating systems. The host has complete control over the guests it runs. A *guest* is a virtualized instance of an operating system running on a host. The guest can use the resources provided by the host, but in isolation from the other guests.

The *package manager* is the software suite that takes over the operations of installing, maintaining, or removing pre-compiled software packages in Linux distributions. Examples of packages managers are the Red Hat Package Manager (RPM) and Debian's dpkg. Selectable pre-compiled software packages are stored in one or many shared *repositories*.

A *maintainer script* is a script that is called during package maintenance whether installation, upgrade or removal. Maintainer scripts usually perform complex tasks such as creating users and groups, generating or removing application-specific configuration files, generate SSL or SSH keys, etc.

Repositories are storage locations containing a typically large number of software packages, which can be retrieved by the package manager. Repositories contain a certain amount of meta-data about the packages they contain. *Trusted* repositories contain digitally signed packages and are deemed safe (malware free).

2.1 Use Cases

The most difficult part, it seems, is to justify non privileged user package management as a valid and important use case for Linux-based computer users. While it may not seems *a priori* as an important use case, the proliferation of shared Linux workstations at work, in schools, and at home, warrants the question to be explored seriously. Indeed, how do we bring a superior user experience, better customization, and higher usability to users sharing computers, as in, for example, a computer science class or similar environments while minimizing the effort from the system's administrators?

Non privileged user package management may be the solutions to a number of administrative woes. Consider:

1. **Reduced workload for administrators**. Administrators would not be pressed to install user-requested packages. The current installation process asks for the administrator to personally intervene to install the packages, but only after having investigated whether or not the packages threaten the system in some way, and after deciding whether the user's benefit is worth the effort of installation.
2. **Reduced delays for users**. Users in large machine parks would not need to wait for their requests for given packages to be processed and possibly denied by the administrators. Users could install packages in their environment right away, to no detriment to other users.
3. **User Empowerment**. Users would gain direct control over their work environment without impeding on the other users. Users would be able to configure and streamline their environments for maximum usability and productivity, having all the application *they* need, rather than subset of software preestablished by the administrators.

What we advocate is, in essence, a shift away from the current centralized, somewhat totalitarian, management model to a distributed, delegated management model where users can setup their own environments, subject to the system's policies, while minimizing impacts onto other users.

As of today, adept users circumvent the impossibility of installing their own packages through the package manager by making local installs from tarballs. As already described, this is a tedious and error-prone process. Additionally, packages installed from tarballs are not recognized by the system's package manager and therefore are not automatically upgraded. Ideally, user-installed package should provide the same facilities than system packages, that is, minimizing installation effort, reducing considerably the risk of setup and configuration problems, while increasing maintainability through periodic and automatic package upgrades.

2.2 Objections

Even though "user empowerment" is a nice idea, one may object to non privileged user package management by raising a number of objections. A possible list of such objections could be as follows:

1. **Delegation using sudoers**. One could allow users to access the package manager without really giving them root privileges by adding them and the specific command to the sudoers list. However, allowing users to use the package manager to install global packages is tantamount to giving them root access as they are free to install whatever package they want, including broken or malicious packages. Even well intentioned, they can install software that affects adversely the other users and they can make the system inoperable as a whole. This would be prevented by user-managed local installs, since unprivileged users are granted permission to install packages from a possibly limited set of packages (defined by administrator policies) and only in their own user environments; and thus installed software runs with the users' privilege levels. Decisions on how to resolve packages conflicts are made by the user—but never to the detriment of the system. We will this discuss this issue in detail in Section 4.5.
2. **"Root" packages**. Packages containing software that must be run with privileges level higher than a normal user, such as kernel modules, services using protected resources like port numbers under 1024, etc., clearly cannot be left to user management. Therefore, there must be configurable policies built into the package management system to prevent users from installing such packages, and this issue cannot be removed by non privileged user package management. However, critical packages can be tagged in the repository as such and are therefore prevented from being installed by a non privileged user.
3. **Security of repositories**. Trust management for packages is a major issue. One cannot allow the

addition of arbitrary repositories, nor allow the installation of packages of unknown origin. Indeed, if one allows user-managed packages, does it imply that one also must allow user-managed repositories as well? If so, it means that the repositories allowed must be trusted repositories (for example, distribution-specific canonical repositories and their trusted mirrors) to prevent users from adding potentially harmful packages from arbitrary repositories. A system-level policy must be used to allow or disallow users from installing packages from unknown repositories or from a local directory.

4. **Redundancy and Disk Space**. A given package may be installed by more than one user, resulting in multiple copies of the package's files (including configuration) that must be maintained. Moreover, if the user can access his account from machines with different architectures (via NIS and home directories over NFS, for example), the package may be installed for each architecture, if available. Unless users lock the package version (for reasons their own), these multiple copies should be updated correctly whenever system-wide updates are launched, resulting in extra computational cost. This is mitigated by using group-level installs (which we will discuss later on in Section 4.2) and by the fact that the number of user-managed packages on a given system is expected to remain relatively modest compared to the total number of packages installed on the system. Moreover, disk space quotas would sufficient to prevent users from using an inordinate amount of disk space, at the detriment of other users.
5. **Users ignore policies**. The administrator may want to prevent users from installing software such as games, BitTorrent software, etc. As simply informing the users of one's wishes is not sufficient to prevent them from installing forbidden software, the policies must be enforced in the package manager system itself. Policies define what is an acceptable package (and its provenance), and where it can be installed. We discuss policies in sect 4.6.
6. **Users don't know what they're doing**. While it may be true that not every user is familiar with the intricate details of package management, and that users may not understand the impacts of installing a given package, the package management system must prevent them from causing damage to the system and their own environments by applying its rules of conflict resolution.
7. **Packages and Repositories must be modified**. It is true that a major reworking of repositories, packages, and software they contain is needed to make user-managed packaging systems possible. Every package has to be tagged with the specific set of privileges required for its installation and use, but more importantly, has to be made completely *relocatable* so that user installs can be completed. Relocation means that maintainer scripts and meta-data, other than system-provided, must be rewritten in order to accept arbitrary locations for the packages—it may even imply change in the software itself so it can adapt to new locations. It also implies that the user environment setup scripts must be modified to ensure that correct environment variables, paths and priorities for packages are set.
8. **Package Managers must be modified**. The package management software must be modified as well to take into account the new meta-data found in repositories and packages. More importantly, package management software must be modified to manipulate multiple package databases and deal with package conflicts between the non privileged users installs and the system's packages, ensuring consistency of the system as a whole. In particular, conflict resolution rules must be extended to include several databases. Far from being impossible, we show that, indeed, conflict resolution rules may be extended to include several package databases.

Most of the preceding objections can be lifted, either totally or at least greatly mitigated. For example, one could use the potential explosion in needed disk space as an argument against user-managed packages, but in reality, this is not a problem given that there are other facilities within the operating system to limit a user's disk space usage (which would already be in use in a multi-user system), and that, for all intent, the cost of the disk space itself is negligible.

The only serious objection to user-managed packages is the amount of work needed to convert repositories and modify package management software. What would be an argument against the modification of packages is but an argument about workload, not about the philosophy of non privileged user package management itself. The amount of work needed to modify the repositories for added security and to allow truly relocatable package is not small, but may not be as important as first thought. We discuss the necessary changes to packages in Section 4.4 and 4.7.

As for package management software, we show in this paper that the changes are likely minimal and that conflict resolution rules may be extended to non privileged user package management as well, as we will show in Section 4.5.

3 Existing Solutions

While it is our opinion that no exact solution to our problem already exists, in this section, we consider the different solution venues. First, we discuss VServer, a system-level virtualization kernel modification that allows one to create distinct virtual copies of the kernel on a single machine. We discuss PackageKit, a package management API and GUI. We then discuss widely used package managers such RPM (Red Hat) and dpkg (Debian). For each, we explain why they are not exactly solutions to the problem we are interested in, as stated in Section 2.

VServer is a modification to the Linux kernel to allow system-level virtualization, enabling the computer to run several guest virtual instances of the same host kernel. This means that one can setup several isolated instances of the same Linux distribution, or even different distributions provided they use the same kernel. Each instance can be used by different owners, each enjoying root privileges but unable to influence other instances. VServer is therefore used to share the same hardware between users with different needs, as each user can install his own customized environment.

Delegating system management using virtualization is a too heavy-handed answer to our problem. Using VServer, the host system's administrator relinquish full control to his users (the administrators of the guests) as to what is installed in their instances. The administrator can still control which repositories the users can use, but this means reducing the customizability of the guest systems. Obviously, the VServer kernel extension provides no solutions for redundancy, as each guest has its own instances of files. However, redundancy can be limited somehow by hard-linking files across guests, but this also limits the freedom of the users to choose their packages, while adding the possibility of conflicts. Redundancy is also mitigated as, very often, but not always, VServer is used to create several server-type guest environments, which are, almost by definition, much lighter—in terms of the number of packages installed—than desktop environments.

Since a VServer guest installation is a complete Linux installation, it gives its guest administrators full control on which packages are installed in the guest system (provided they are so allowed by the host administrator). Inside the guest system, the problem of allowing users to install packages is still complete. Users having access to the guest system but that are not administrator for that guest cannot install packages, and the guest administrator cannot delegate this right to his users in a way that does not raise the objections stated earlier. The multiplication of guest installations may also mean that the host administrator has an increased workload as he must now provide not only for the host, but also for the various guests' environments and users. Since all guests could potentially be very different distributions (but sharing the same kernel) the lack of uniformization in each environment clearly does not simplify the host's administrative work.

PackageKit's primary goal is to standardize package management across distributions. PackageKit abstracts the complexities of the various existing package managers by offering a consistent interface across the various distributions that already use it. It is composed of a privileged dæmon, `packagekitd`, several distribution-specific back-ends and GUIs. The front-ends communicate with the dæmon using the D-BUS desktop integration protocol, which in turn, delegates the actual package management to the distribution-specific back-end. By design, PackageKit deals with some of the objections we raised to the traditional package management strategy.

The use of a privileged dæmon means that users can be granted or denied its use in a secure fashion using privileges set by the PackageKit administrators. However, PackageKit is still limited by the inherent (in)capability of the underlying package managers as it uses them as back-ends to complete its tasks. Improper configuration of PackageKit opens the door to the same kinds of problems encountered when using delegation through sudoers. If we suppose that user requests can be filtered, the filters become a critical element of the package management system. Malicious—or simply unskilled—users could otherwise install packages that results in system-wide damage. Such policy filters are not implemented in PackageKit as of now; however, due to PackageKit's design, there should be no major obstacle to adding this feature [1, 2].

This being said, it is worthwhile to note that Pack-

ageKit's design is not fundamentally incompatible with non privileged package management, even though it currently offers no direct support for it. It would be indeed possible to do so, since a contribution to PackageKit allows the local install of specific audio/video codecs to a user's directory.

RPM, the Red Hat Package Manager—one of the two major package managers along with Debian's dpkg—implements package relocation in two ways, although the original intend was not to allow regular users to perform private installs [3]. The first type of relocation, using the `--prefix` switch, causes all of the package's contents to be installed in the specified directory, including configurations files that would normally be installed in locations such as `/etc/`. The `--prefix` option provides support to the maintainer scripts through the environment variable `RPM_INSTALL_PREFIX` which contains the path to the alternate location. Maintainer scripts can use the variable to detect relocation and use the new install location.

The second method, path replacement, is a simple technique that allows to install a subset of package's files in alternate locations and uses the `--relocate` switch to do so. For example, specifying `--relocate=/etc=$HOME/etc` would cause the package manager to substitute `/etc` for `$HOME/etc` wherever it occurs and, accordingly, all files that would have been installed under `/etc/` are now occupying the same relative location under the specified alternate location. Used without care, the `--relocate` option can lead to non-functional packages, as it offers no means to signal relocation to maintainer scripts.

RPM allows for relocating the package database, either by using the `--dbpath` switch, which specifies an explicit location, or by using the `--root` switch, which specifies a new root directory under which the relative location of files is preserved. However, relocating the database may necessitate the use of `--nodeps` and `--noscripts`, two switches bypassing important features of the package manager. Using `--nodeps` will cause the package manager to skip dependency checks as the package manager will refuse to install packages that have missing dependencies, either because they are not installed or because relocation prevents the package manager from detecting them correctly. The `--noscripts` may be necessary for non relocatable packages as their post-install scripts are not using relative, relocatable, paths.

While this gives users the opportunity to install packages in their own directories and manage their own databases without root privileges, the RPM approach has a number of serious drawbacks. First, relocation using path replacement may cause the installed software to fail as it may not be able to find the files it needs. Were the packages relocatable, they would be able to look for the configuration files in the new locations rather than in the default locations; something that is, as of now, provided neither by the RPM package manager nor by the packages themselves. Second, relocating a RPM package will likely cause it to be unmaintainable. RPM keeps track of packages that are installed, and where, but if the maintainer scripts are not able to manage relocation, fully automatic package updates will not be possible, even with user package databases. Relocation can also break dependencies when other packages depending on the relocated packages are installed as the installed dependencies are detected, but default locations still assumed. Third, and lastly, RPM is unable to maintain the consistency between the system's database and the user databases because it merely allows relocation of the database; not multiple databases, thus potentially causing the user packages to break whenever system-side dependencies are modified. The user must, each time, manually reconfigure his packages to adapt for system-side changes, a time consuming and error-prone process.

RPM offers enough options to allow non privileged users to manage their own databases and packages, but as we explained, packages cannot be maintained automatically and may be installed in a non-functional state that may require manual effort to reconfigure properly—if possible at all. In addition, none of the user's environment variables will be updated correctly, requiring further manual intervention.

Dpkg, Debian's package manager simply does not allow non privileged users to maintain private databases. Dpkg has no facilities to allow relocation, save for installation inside "chrooted" environments. Maintaining a local database and relocated packages will cause dpkg to use the `chroot` system call before launching maintainer scripts. However, `chroot` is a privileged command unavailable to normal users. Additionally, dpkg's maintainer scripts are often complex, and invariably assume `/` as the path prefix.

While adding relocation capabilities to dpkg was discussed [4], there are no serious plans to implement the

feature, as deemed an unimportant border case [5].

So, in essence, neither PackageKit nor VServer can be used to allow non privileged user managed package. PackageKit offers an abstraction to package management in order to standardize the package management API, relying on distribution-specific package management software to perform the actual management. PackageKit may be promising because it may allow the addition of policies to package management, but still lacks the possibility to relocate packages as it is fundamentally limited by the underlying package manager. VServer, on the other hand, only serves to create a nested version of the problem where the management of guest installs is delegated to their respective administrators (which are themselves subordinated to the host's administrator) and where the users of each guest are as normal users on a normal distribution, that is, unprivileged and unable to manage their own packages.

The principal package managers' limitation is that they expect their databases to be in a unique preassigned location and cannot deal with multiple, possibly conflicting, databases. Therefore, to minimize risks for the system, the databases are set to be accessed with root privileges only. Package managers may offer means to relocate the database, but they are limited to a simple relocation. If a user uses these features to create a database within his home directory, he could, theoretically, manage his own packages. But we saw that doing so forfeits most of the package manager's advantages such as automatic updates, conflict and dependency resolution, and automatic configuration.

Parts of the problems with relocation and automatic package configuration lie with the packages themselves; as their install scripts are unable to provide for arbitrary relocation of the files they contain, and even worse, not all software is capable of being relocated. This means that the users must perform the necessary configurations by hand, if allowed by the software at all.

In the next sections, we outline what we think would constitute a viable solution addressing all the objectives (and possible objections) stated until now, in particular package relocation, conflict and dependency resolution using multiple databases.

4 Proposed Solution

The minimal changes made to a package manager to accommodate unprivileged user package management necessarily depends strongly on the package manager one wants to modify. The first important modification is to allow the package manager to use user-specific databases in addition to the system database. The second is how the packages themselves are modified to allow fully relocatable installs. The third important modification concerns the users' run-time environment that must be set up correctly so that the users' packages are correctly configured. All modifications must be minimal, and, if possible, hidden to the user. Furthermore, a good solution would also be compatible with the File Hierarchy Standard (FHS) [6].

To outline our proposed solution, we will use dpkg as an example, especially that dpkg does not currently allow local installs at all. We will present how to extend dpkg to include unprivileged user package management.

4.1 Package Databases

The package manager must be extended to account for user-managed package databases. The user must be able to create a database for himself without further privileges than he already has. The creation of such a database would be automatic whenever the user invokes `apt-get install` or `dpkg -i` without root privileges. The location of the user's package database would be hidden in a dot file, or even a file within a dot directory. The database would maintain the list of packages installed by the user as well as their locations. Note that now, default location takes quite a different meaning. It can mean the usual FHS or a relocated file hierarchy relative to the user's home directory, depending on the privileges used during installation.

Without any special rights, the user could now (possibly with the help of some desktop applet) perform automatic and periodic updates of his installed packages. As packages are updated by either the user or by the system's administrators, dependencies conflicts must be resolved in an intelligent fashion. We will discuss conflict resolution at length in Section 4.5.

4.2 Managing Local Databases

The default user database location should be sufficient in most cases, but it could be explicitly relocated. By default, privileged user would use the system default database, location, and install paths, which are relative

to /. For an unprivileged user, the hierarchy would be relative to his $HOME. In addition to installed packages and their location, the users' databases must include a list of package-provided setup scripts that allow each package to configure its environment properly.

The package manager must now support at least a few new commands which we will outline. Let those new commands be accessed through the added program dpkg-env. Note that in addition to dpkg-env, one would modify dpkg itself (and any front-ends, such as apt) and the necessary modifications will be made clear in the following paragraphs.

Through dpkg-env, the administrators and users will be able to perform local database management and environment setup. The first important set of commands would allow the addition, management, and removal of local user (and group) databases. For example, a user can invoke dpkg-env --new without privileges to create his own local package database. If it already exists, the command succeeds. One could also invoke dpkg-env --new --user username with sufficient privileges to create a local database for user username (it would also be automatically created for the user invoking non privileged package management for the first time). Invoking dpkg-env --new --group groupname with sufficient privileges would create a group-specific database located in /var/lib/dpkg/groups/ or, if specified otherwise, in some other location. The packages themselves would be installed in /var/lib/dpkg/groups/$GROUP. The important nuance with group installs is that all users that are member of this group will inherit the group's packages and environment at the login. A corresponding --remove command would destroy a database after launching deinstallation of all related packages, preventing unusable packages as well as loss of disk space.

4.3 Configuring the environment

The next important command relates to the user's environment setup. For example, dpkg-env --setup would scan the system database, the groups' databases and finally the user database to set up correctly the environment for every installed package. The system database would provide only default location installation, so all packages in the system database can be swiftly dealt with by using the default environment settings (as it is currently done on Debian and related distributions). For groups and users, the procedure is similar. The database is scanned and the package-specific environment setup scripts are called. If a package offers (or needs) no such script, it inherits default group or user location through $PATH and $LD_LIBRARY_PATH. If a package does offer such a script, it is executed and the changes made to the environment are propagated to the session.

The problem with dpkg-env --setup is that it must be called just after a user logs in and before he begins his session, so that he can use all the available packages correctly, including packages such as window managers or interface extensions, for example. Currently, it can be done in three different ways: using PAM, using shell-specific profile configuration files, or using .xsession.

The Pluggable Authentication Module (PAM) is the standard Linux user authentication mechanism. It includes modules that are capable of launching actions when the user logs in, before shell or session scripts are executed. However, there are currently no simple way of modifying the environment variables from PAM. PAM uses the pam_env module that allows the user to add variables to his environment through the user's .pam_environment file. However, the file only contains a static list of environment variables and their values, and as such, cannot be used to run the package-specific scripts needed to configure the environment properly. The possibility of calling scripts from PAM might be an interesting addition to the session initialization process.

Using shell profile scripts is not universal. The scripts are recognized and executed by Bourne compatible shells (sh, bash, ksh, zsh, etc.) when they are invoked. The .profile is ignored until a shell is launched, and so is useless if the user logs in via a graphical interface that does not invoke a shell. This can be solved by using the system-wide X session configuration file. In both cases, it merely suffice to append a call to dpkg-env --setup to the files, and the program is run using the user's credentials with no need for special privileges.

4.4 Package-Specific Environment Setup Scripts

To support complete relocalisation, it is likely that the package will need, in addition to a field that tags it as

relocatable or not, a script that prepares its environment once installed so that it runs properly. If the package requires nothing more than the default (relocated) locations, the script is not required, as `dpkg-env` would already provide the correct values through `$PATH` and other environment variables. The packages might, however, require a special setup.

While executable programs need little more than modifying `$PATH` to be available, it is not so with all packages. Native libraries can be found with `$LD_LIBRARY_PATH` setup so that the proper precedences are respected. Manual packages can already install manpages in alternate location, provided that `$MANPATH` is set or `--manpath` specified. Desktop elements, such as icons and menu items can be installed and found through various environment variables that hopefully complies with the Base Directory Specification, already in use in major distributions [7, 8].

Configuration files are often expected to be found in the `/etc/` directory. Programs using absolute path to the configuration files will need to be modified to access them through environment variables and relative paths. Dynamically updated data, such as logs, are usually found in `/var/`. Just as with configuration file, should this directory be relocated, the packages must access it through the environment set up by `dpkg-env`. However, care must be taken to ensure that the relocated directory grants write access only to users that are entitled to use the package.

Finally, static data such as images, sound effects, etc, are usually installed in `/usr/share` and do not require much caring for. They can be relocated without harm provided that the access rights are set up properly for their intended users and that proper environment variables are set so that applications can find them.

4.5 Conflict Resolution

If both users and administrator install packages independently, conflict is bound to happen sooner or later. By conflict, we mean whenever the situation comes up where the user's environment is affected by a change in the system's environment or when the user installs conflicting packages. Rules must be applied to decide what will be the course of actions should conflict arise. In the next few paragraphs, we will present conflictual situations and how to resolve them, while ensuring the system's integrity as a whole, possibly to the inconvenience of the user or group. The main conflict resolution rules would be as follows:

1. If a package x is user-installed and subsequently system-installed, the package manager should proceed to uninstall the package from the user's packages while leaving his configuration files untouched. Same would occurs if the same package x is updated system-side and now meets the user's version.
2. The package x is installed system-side, but a user has a different version x' that depends on user-installed package y. If x and y are incompatible, accept or deny the removal of x' and y from the user's packages and system-side installation of x and y.
3. If a system-side package x is installed, and that x is user-installed and is depended upon by a user-installed package y, remove both x and y from the user's packages and install y server-side.
4. If a system-side package x is installed or upgraded and the users has a package y that depends on a different version of package x, x', also user-installed, and that y is incompatible with x, accept the removal of x' and y from the user's packages and proceed with the system-side installation of x, or fail the install or upgrade of x.
5. If a system-side package x is installed or upgraded while the user has package x' depended upon by user-side package y, and y is incompatible with x, accept the removal of y from the user's packages, or fail the install or upgrade of x.
6. If a user-side package x is upgraded and an older user-side package x' is installed, proceed with the install of x.
7. If a user-side package x is upgraded while depended upon by user-side package y, but y (and newer versions of y) is incompatible with the new version of x, confirm the upgrade of x and the removal of y, or fail the upgrade of x.
8. If a user-side package x is upgraded while depended upon user-side y, and the new version of x is incompatible with the current version of y but a newer version of y that is compatible with the new version of x exist, upgrade both packages after confirmation.
9. If a user tries to install a root package x his request is denied and the package manager bails out.

In the previous rules, the same applies when 'user' is changed for 'group'.

Unsurprisingly, the rules are reminiscent of the rules already existing in current package managers, except

that they are extended to include both system-side and user/group-side packages. As with system-side only package management, the administrator is proposed choices and must accept or refuse changes to the packages based on information provided by the package management engine. The difference is that now, the administrator can decide to forgo upgrade of a package because it breaks a user-side install, and if he does break user-side packages, he does so knowingly.

4.6 Policies

Currently, neither dpkg nor apt provide for policies. As discussed earlier in Section 2.2, it may be wise to prevent users from adding repositories and install certain types of packages, regardless of their origin. Preventing users from adding repositories greatly reduces security risks, as one cannot ascertain the trustworthiness of these new repositories. The same applies when the user tries to install a package from one of his directories.

If adding repositories is allowed, the data must be stored in a configuration file in the user's home directory, preventing effects on the system as a whole. Adding a repository asks for authentication, and this can be done using a white/black list system, where trusted repositories are listed. Such lists would be distribution-specific.

Filtering by package type (either categories, sections or tags), or even by repository, may help the administrators to ensure efficient usage of facilities. Software likely to disrupt work environment, such as games, BitTorrent software, resource-hungry applications, or software with unwanted licenses, may be blocked by administrator fiat. The granularity of the policies could be very coarse (repository level), coarse (package class level, tag sets), or even fine (a specific package with a specific version).

This limited type of policy management can be implemented by adding a separate module to offer the administrator the possibility of editing policies. Categories would include policies for all unprivileged users, for specific unprivileged users, for all groups, for specific groups. The package manager would simply access the policies database to determine whether or not a given operation can proceed. We think that the modifications to the package manager are minimal since it suffice to verify if the operation can be performed by checking against policies.

In Section 2.2, we also noted that the disk space usage problem can be mitigated (or even eliminated) by the standard quota facilities, and so we believe that it is unnecessary to modify the package manager to track a user's disk space usage. It would however need to test if sufficient space is left to perform the desired operations.

4.7 Modifications to existing Package Management Software

Let us pursue with dpkg as an example package manager and present the modifications needed to make unprivileged user package management possible.

The first step would be to modify the tools used to produce packages. In the case of debhelper (the main tool used for the creation of packages for dpkg, which provides a set of scripts to handle various repetitive tasks), for example, most of the modifications are contained within the default behavior of environment setup scripts. For the vast majority of packages, it will suffice for the setup script to add the package's file locations to `$PATH` and `$LD_LIBRARY_PATH` through a helper script API that provides `sh`-like functions to ensure correctness and avoid multiplicities. One would then simply rebuild the package with the added setup script, provided that the software contained in the package is relocation-aware.

The dpkg command line needs to be extended to support multiple database paths. By default, dpkg would be able to locate the system and the invoking user databases. How operations are performed would depend on the access rights provided on invocation: root access would affect the system database while unprivileged invocation would affect the user's database. Of course, it must be possible to specify explicitly operation mode regardless of current privilege level.

The most important internal change to dpkg will be to include install location to the package name and version to the database. Multiple versions of the same package can exist simultaneously in different locations and dpkg must be able to manage them correctly and independently.

Conflict resolution must obey the resolution rules presented in Section 4.5, with the effect that system-side packages have higher priority than user or group-installed packages, as the goal is to keep the system consistent, even if it means breaking a user's environment.

Therefore, to apply system-side conflict resolution rules as stated, dpkg must access and read *all* databases. By doing so, conflicts can be resolved, correct updates of users environments can be performed, and ensure the system is left in a stable configuration, even if it means that some users may have some of their packages upgraded or removed. Ideally, package removal due to conflicts must be notified to users. To apply user-side conflict resolution, only the system and the invoking user's database must be read and verified.

Maintainer scripts that allow correct package relocation can be instructed of the new location via an environment variable or a direct argument; in either case dpkg must be modified to provide the correct value to the maintainer scripts for install, updates or removals. The current version of dpkg uses the `chroot()` system call before calling a maintainer script to provide it with the correct relative location and while this feature is useful by itself, it would be activated only through the explicit use of the `--root` command line option.

Apt would need to behave correctly when called by a non privileged user. This means it would access the calling user's database or bail out elegantly if this is not possible. This is needed to ensure that invoking `apt-get install` functions correctly, installing packages in the user's environment. Apt must locate automatically the relevant databases, searching into user-configured or standard locations. User-configured locations could reside in `$HOME/.apt.conf`, which would be read each time apt is invoked. A personal `source.list` must also be locally stored and checked by apt to allow a user to maintain his personal repository list (while subject to system's policies). This file should use the same syntax as the global `source.list` file.

To test for a relocatable package, it suffice to add a boolean field that indicates whether or not the package is relocatable, allowing apt to bail out gracefully if a non-relocatable package is installed with insufficient privileges. Since in Debian-like repositories meta-information about packages can be copied in the repository index, it would be easy to make apt warn the user about the non-relocatable nature of a package even before downloading. This functionality must therefore be included in GUI-based front-ends like Synaptic to help users manage their packages.

Policies for repositories and packages available to the user also have to be included at this level. There already is a configuration file, `/etc/apt/preferences`, that enables the system administrator to pin particular packages to inhibit changes. This configuration file could be extended with similar syntax to include information about the prioritization and exclusion of repositories. For example, fields such as `Rep` and `Rep-Priority`.

The package manager (and its front-ends) must also be able to prevent the installation of a package with untrusted origins. A user should not be able to install a package, should the administrators decide so, from a non authenticated location such as one of his directory. In most distributions, the repositories already contains the quasi-totality of safe packages, so the need to use untrusted packages is quite lowered—the only possible exception would be development tarballs corresponding to packages too recent to have made their way to the trusted repository.

5 Conclusion

In this paper, we have presented the problem of non privileged user package management. In Section 2 we presented the proposed use cases and possible objections. The use case we stressed most is when the user shares workstations in a work or school environment and he does not have sufficient privileges to install packages he needs and therefore has to rely entirely on the system's administrators, which may lead to unacceptable delays or even plain refusal. We presented, in Section 3, the existing solutions to the problems and why they were not quite satisfactory. In particular, we discussed the current package manager and how they fail at providing all the tools necessary to make non privileged user package managing possible.

Finally, in Section 4 we outlined the various modification to be brought to package managers to make non privileged user package management possible, in particular conflict resolution rules and how dpkg, taken as an example of package manager, should be modified to accommodate our proposal. We discussed policies that would be needed to ensure the system's stability as a whole. We also showed that setting up the user environments at login would be rather simple despite the lack of standard initialisation procedure across Linux distribution. Finally, we think that we made clear that not only non privileged user package management would be beneficent to users, it is also quite possible to implement

using relatively little effort compared to what one might have thought initially. Having shown the feasibility of non privileged user package management, the next logical step is to proceed to implementation, which is the object of future work.

References

[1] *The PackageKit Package Management Front-End*, `http://www.packagekit.org`

[2] *The PackageKit Source Repository*, `http://www.packagekit.org/pk-faq.html`

[3] Edward C. Bailey, *Taking the Red Hat Package Manager to the Limit*, Red Hat Inc, 2000. Chapter 15, `http://rpm.org/max-rpm/ch-rpm-reloc.html`

[4] Kenneth Arnold, *Relocatable Package Development proposal*, Dpkg's Wiki post, `http://www.dpkg.org/dpkg/RelocatablePackages`

[5] Daniel Burrows, *Re: Non-privileged user package management*, Gmane forum message, `http://permalink.gmane.org/gmane.linux.debian.apt.devel/15021`

[6] Rusty Russell, Daniel Quinlan, Christopher Yeoh (eds), *The File Hierarchy Standard, V2.3*, `http://www.pathname.com/fhs/pub/fhs-2.3.html`

[7] Waldo Bastian, Francois Gouget, Alex Graveley, George Lebl, Havoc Pennington, Heinrich Wendel, *Desktop Menu Specification, V1.0*, `http://standards.freedesktop.org/menu-spec/menu-spec-1.0.html`

[8] Waldo Bastian, *Base Directory Specification, V0.6*, `http://standards.freedesktop.org/basedir-spec/basedir-spec-0.6.html`