An Implementation of the Linux Software Repository Model for other Operating Systems

Neil McNab
Appupdater Project
neil@nabber.org

Anthony Bryan
Metalink Project
anthonybryan@gmail.com

Abstract

Year after year, the frequency of updated releases of software continues to increase. Without an automated install process, the result is either that a system installs software with known defects and/or vulnerabilities, or systems require increased manual labor to maintain up-to-date software installations. Linux packages, in conjunction with repositories, fill this need for automation to reduce both undesirable situations. This model can be modified to a generic operating system environment, such as Windows, which currently lacks the capability to update arbitrary software applications. Our application, Appupdater, demonstrates this concept of detecting, downloading, and installing upgrades automatically. This provides a completely automated upgrade cycle.

Categories and Subject Descriptors D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; D.2.9 [*SOFTWARE ENGINEERING*]: Management

General Terms Management, Security, Verification

Keywords upgrade, repository, packages, detection

1. Introduction

Maintaining up-to-date software on a computer without tightly integrated package management has never been an easy task for a casual user. Over the years, the internet has become critical for information about known software vulnerabilities as well as a distribution method for additional software releases to address them. There are many tracking tools that provide notification when a new release is available via a website or a software scanner tool. That is no longer the difficult part. The challenge today is actually finding the URL, downloading, and installing the latest versions soon after release.

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM ISBN 978-1-60558-723-3/09/10. . . $10.00

Installing the latest version of software is more critical than ever. There is a race between hackers and users to obtain software updates [Johansen 2008]. The availability of automated exploit generation is increasing the frequency of zero day exploits [Brumley 2008]. Time is critical in deploying new software versions. Mozilla Firefox 3.0.0 was released in June 2008[1]. The current release in June 2009 was Firefox 3.0.11. That is nearly a new security or bug fix release every month. Keeping up with this update pace on unmanaged systems is extremely difficult, especially when considering other installed software requires updating as well. A package deployment tool is the best choice for this task [Jansen 2005]. To test this concept, we have implemented a program called Appupdater[2].

Many commercial and open source software applications are available for managing software updates. However these applications have several limitations that do not make them useful on an internet scale. First, each organization using these products is responsible for creating and maintaining their own repository-like structure and packages that is usually only available internally. Providing access to the greater internet could exceed bandwidth available or violate license agreements. It is not practical to expect small networks or home users to deploy this type of system to keep software up to date. Second, packages created in this environment are not usually created for *every* version of a software release. An organization is only going to create and test packages when they have a compelling reason to upgrade. Third, it is not an efficient system. If ten independent organizations need a package to install the latest version of Adobe Reader, they each create one. This process is more efficient if the package only needs to be created once and is then available to all organizations, small network operators, and home users.

The products previously mentioned, Firefox and Adobe Reader, do include a self update capability. This type of feature usually involves polling a server on the internet and notifying the user when a new version is available. There are a number of problems with these. First, they usually

[1] https://developer.mozilla.org/devnews/index.php/2008/06/11/coming-tuesday-june-17th-firefox-3/

[2] http://www.nabber.org/projects/appupdater/

require user intervention to actually download and install the updated versions. To complicate the problem, the user is often prompted that updates are available when they do not have the necessary system permissions to install these updates. This can actually increase the workload on a system administrator because users may not have been aware that their system needed upgrading until prompted by the update service. Finally, this is not useful for private networks and those otherwise disconnected from the internet since they cannot poll the update server.

Critical goals for a platform independent package management system have still not been addressed in a single product. This includes detection of previously installed software outside of the package management system. Second is the client capability to perform fully automatic updates. This means a daemon that can run in the background with the necessary privileges to install packages and bypass any prompts and wizards that are part of a typical user initiated install. A decentralized repository model allows for community contributions and minimal bandwidth for a server. This also allows for a combination of both a single entity that can provide trusted updates in addition to allowing individual application authors to create and maintain their own repository. Resilient file downloads are required because of the decentralized nature and architecture considerations of the internet. This includes integrity checks and multiple download servers when possible. Also needed is basic dependency support. Finally, offline use to update a stand alone computer is a goal. Appupdater meets all of these requirements.

1.1 Linux Packages

Currently, there are many different package formats available for Linux, but they have common features. A package is typically a collection of related binary files combined into a single archive file that is easily portable. The package also includes meta-information about the software included in the package, such as a name, version number, and dependency information. In addition, install and uninstall scripts are used to extract the files from the archive and properly add them to the host system [Cosmo 2008]. The combination of these attributes into a package provides a self contained file with all information needed about that software.

1.2 Linux Repositories

In most existing Linux distributions, software packages are distributed using a repository. Repositories are a way of loosely tying together individual packages for easy distribution. These repositories are typically made available via HTTP, FTP, CD, and DVD. Once Linux is installed on a system, package updates to fix errata are distributed via these repositories. Also new packages are installed in this manner. Modern repositories include security mechanisms such as checksums for integrity checks and digital signatures for source authentication to prevent malware.

2. Software Detection

In order for our new methodology to be "backward compatible" with already installed applications, we needed a method to detect this legacy software. Most existing software update/install tools do not do this. They are designed with a corporate or enterprise environment in mind where the systems are centrally managed and configurations are tightly controlled with the update and install tools. It is unnecessary to require users to install packages through our custom package concept. For example, in Windows, an installer often installs software onto a system. The installer itself contains the minimum information needed in order for a system to properly copy the included binary files and configure the environment. By having the capability to detect software, all applications installed on the system are properly updated as needed, not only the software installed through our package management application. There are two common methods of extracting software version information from Windows, probing the registry and using meta-information embedded in the files themselves. Both of these are insufficient for our purposes.

2.1 Registry Probing

Probing into the Windows registry is the most common method for determining what specific versions of programs are installed. There are a number of issues related to this. First, this is not platform independent. While this works for Windows platforms there is no concept of a registry on UNIX/Linux/Mac OS systems. Also it can be inaccurate. This can happen if an uninstall program fails. It may leave the program's information in the registry after the associated files for it have been removed from the system. Registries are also notorious for being fragile [Ganapathi 2004].

2.2 Embedded Version Numbers

A more accurate but difficult method for detecting installed software is by using embedded version numbers. Part of a standard executable file in Windows[3] is the ability to embed a version number directly into a .exe or .dll file. Again this is another method that is not platform independent. Most binaries compiled for Windows contain version information and it is usually very specific, sometimes including the build number. Problems can occur because the developer does not always utilize this feature.

2.3 Additional Problems

There is no way to get complete software meta-information from the Windows operating system. Inconsistent versioning schemes exist such as the Java transition from 1.5.X to 6.X. Also Firefox reports a different version number at the system level as compared to what a user is familiar with. For example, what users know as Firefox 3.0.10 is referenced as 1.9.0.3399 in the software itself. Some software even

[3] http://msdn.microsoft.com/en-us/library/aa381049(VS.85).aspx

lacks the granularity of minor version information, making it extremely difficult to determine the difference between releases and therefore what vulnerabilities may be present on a system. Some files have no registry entries and are present on the system without embedded version information, which causes an incomplete data set.

3. Installation

Most platforms other than Linux typically distribute software in the form of executable installer files. However, the information included in these does not comprise a complete package because there is no standardized way of expressing the software meta-information. Also, the installers frequently require user intervention to complete the install operation. To facilitate a more automatic installation, similar to Linux packages, we have added additional meta-information. Our package format includes silent install switches, which allow many installers to be run without requiring any user intervention.

A disadvantage to being dependent on silent install switches is that the application authors often leave these undocumented. Many of these switches only exist because they are part of the installer's standard build process. One approach is through trial and error. Using the silent install switches for well known installers we can determine what the silent install command switches actually are. Simple examples include /S, -s, or /qn, but they can be more complex. Once the proper switches are determined they also need to be thoroughly tested. This process is very similar to what a Linux package maintainer must do to determine the proper compilation instructions for each package.

Sometimes, due to customizations by the application author, the silent install is not completely silent. Dialog boxes are still presented that require a user's response, despite using a silent switch. Installer behavior can also change depending on if a new install is performed, compared to an upgrade[4]. We have had success using standard automation tools such as AutoIt[5] to address these problems, but this is a manual process. AutoIt provides a scripting language that is used to simulate keyboard input as well as mouse interaction with a graphical install interface. Each application installer requires its own AutoIt script which needs to be debugged and tested with each new release. The installer may change user prompts or the upgrade process might be different from the first time install process. When compared to Linux package management, these are most like source code patches provided by a distribution.

4. Repository and Package Structure

We propose a solution which addresses the gaps in other upgrade methods, meeting all requirements for creating a package structure and distribution system. In designing our package structure we began with the installer construct that already provides the basics for the installation process. However, as previously described, this does not provide enough information for a complete package. Other specifications were reviewed like The Open Software Description Format (OSD)[6] but also lacked the needed attributes such as information about installed files and download hash values. In addition to the installer we add attributes that are common to Linux packages, including meta-information of name, version, and dependencies. We also add an install and remove command structure in order to call the actual installer with silent install switches, if available. Next, we add two additional items that typical Linux packages do not have. First, is cryptographic hashes of the program executable once it is installed for detection purposes. Second, is the download location of the original installer. How the packager derives the components for inclusion in the XML data structure is illustrated in Figure 1.

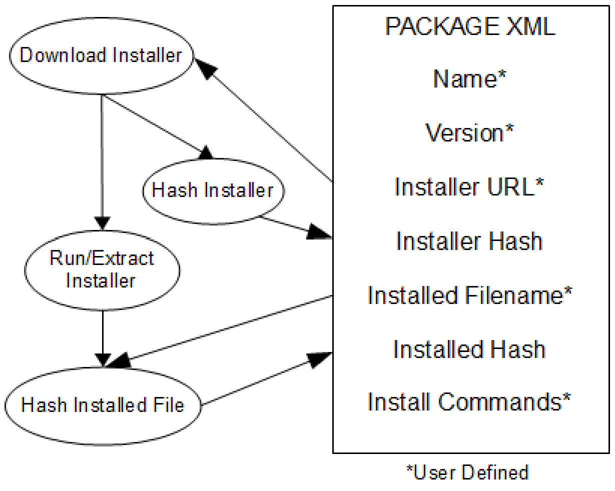

Figure 1. Creating a minimal package

For our purposes we chose a design decision that is significantly different from typical package models. We completely separate the package meta-information from the files that are related to it into its own file. First, this significantly reduces the disk space and bandwidth requirements to host a repository because we can use the URL of the program to have the Appupdater download directly from the application publisher's website. Second, it allows for a way to distribute the version meta-information without having to distribute the application itself, as may be required for proprietary software that is not licensed to be freely distributable. This directly compares to the Windows Update model [Gkantsidis 2006]. The concept of an update server is similar to the server that hosts our repositories. Our repositories provide

[4] http://unattended.sourceforge.net/installers.php

[5] http://www.autoitscript.com/AutoIt/

[6] http://www.w3.org/TR/NOTE-OSD.html

URLs pointing to the equivalent of the distribution servers in the Windows Update model.

4.1 Hash Solution for Detection

We developed our own custom meta-information system in order to create a more reliable detection method. First, for each application we determine a file that changes in every release. This is typically the application's main program executable file or library. Since it is different in each release, its hash will have a unique value for each release, including those compiled for different architectures. The hashes of these program files on a system can then be matched against a database of known software versions. This hash method gives us the needed capability to independently detect installed applications. It also adds assurance that the system in fact properly installed a program.

Hash values that cannot be found in our database can be submitted along with embedded version information as database candidates. This can be particularly useful for adding new releases to the database of known software versions. As an additional benefit, if a particular version of software is not detected properly it may indicate a failed integrity check. In this case, the original version of the software might have been altered.

4.2 Reliable Installer Distribution with Metalink

Repackaging an executable installer into a new single file archive with additional meta-information can be inefficient. This requires the repository to re-host a copy of the installer when the original website likely also has a copy. This can quickly add up to gigabytes of disk space and even more bandwidth to distribute meta-information added to an installer. Also there can be licensing issues when repackaging and redistributing an existing installer. That is why we adopted a reliable download description format called Metalink[7] to specify the installer download for the client. This solves many issues encountered when using websites outside of our control to provide the installers. If the URL is modified with a newer version, the download will be invalidated and our package maintains its version number integrity because it allows us to include cryptographic hash information for the installer file. Also, the Metalink format [Bryan 2006] [Bryan 2009] allows for specification of multiple URLs for a single file, so if a particular server is unreachable, an alternate server can be tried. Since our decision to use Metalinks, this technology has also been adopted by some Linux repositories[8] [9].

4.3 Dependency Structure

The package structure can contain optional dependency information. It it expressed similar to standard Linux repositories, simply by specifying the package's name and version number that is depended upon. The advantage is that this is an extremely simple method for managing dependencies, but certainly has major limitations [Dolstra 2008]. This is not critical for many of the packages we create because the vendor provided binary installer often handles dependencies automatically.

4.4 Privacy and Security

There are no privacy implications for a particular system when considering the list of what programs are installed. All repository information is downloaded every time, irrespective of what is already installed. The client downloads the installers from a third party website, therefore, the repository owner has no way of knowing which installers are downloaded.

In order to improve the hash database, Appupdater reports back to the server only unknown application versions that it finds. This minimal level of automated reporting can easily be disabled in the settings if desired.

Just as with Linux, repositories have the capability to be digitally signed with OpenPGP[10]. This provides a trust chain for all cryptographic hashes of the files included in the repository itself. By trusting the key that signs the entire repository, all subsequent file hashes are also trusted. This includes the hashes of installed files to be detected as well as downloaded installer software.

Rabin fingerprinting [Rabin 1981] has been proposed for software detection in Mirage [Crameri 2007], but this method can be bypassed for malware attacks that are possible in our scenario. The hash algorithms currently supported are MD5, SHA1, and the entire SHA family. Given several vulnerabilities in the MD5 hash [Lenstra 2005], SHA1 hashes are the preferred algorithm and migration to SHA256 is in progress. MD5 is retained for compatibility reasons, especially considering most existing file hash databases use this type. The XML specification used for hashes is based on the Metalink specification which is flexible enough to allow multiple hash types if available as well as an easy transition to newer hash algorithms as they are implemented.

5. Implementation as Appupdater

Appupdater provides advanced functionality to Windows, similar to apt-get or yum on Linux. It automates the process of detecting, downloading, and installing up to date versions of programs.

5.1 Other Operating Systems

While Appupdater is currently most useful for Windows operating systems, it was written with other operating systems in mind. Python[11] was chosen as the implementation language for exactly this reason. On Linux, the client soft-

[7] http://www.metalinker.org/

[8] http://fedoraproject.org/wiki/Infrastructure/MirrorManager

[9] http://mirrorbrain.org/features/

[10] http://tools.ietf.org/html/rfc4880

[11] http://www.python.org

ware was tested with a minimal repository. This gives us high confidence that it will function on other POSIX architectures as well. The architecture has been tested on these two platforms, if any changes to the software are needed for other operating systems they will likely be operating system specific. At this time, the only production repositories created are for Windows because it is the platform that benefits the most from Appupdater. The next operating system candidate is Mac OS X due to its similar usage profile when compared to Windows. Users are often installing third-party software not maintained by the built in operating system update service. It should also be noted that a package itself is *not* platform independent. Due to binary incompatibilities and operating system specific installation calls, each operating system has its own packages. This also suggests that maintaining separate repositories for each operating system is a good practice, but not a requirement.

5.2 Client

We wrote about 10,000 lines of cross-platform Python code as a package deployment tool [Jansen 2005]. It includes many interfaces with the system including a daemon, command line program, Windows service, and graphical user interface. It is based on the repository structure previously described and is the client side implementation. Appupdater completes a full cycle on the client by detecting previous versions, downloading, and installing as seen in Figure 2. No user intervention is required to maintain a system full of up to date software. The download engine reuses source code from Metalink Checker[12].

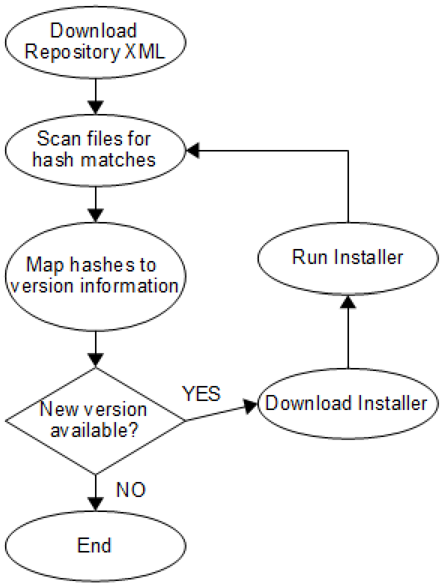

Figure 2. Client upgrade process

[12] http://www.nabber.org/projects/metalink/checker/

The Appupdater client side implementation differs in a few ways from a standard Linux package deployment tool. First, the installed package database is independent of whether or not Appupdater installed the software. After the initial install of Appupdater it scans the system and compares the file hashes to those in its database to determine what has previously been installed. Second, this same information also provides a way to validate that an install process completed successfully with Appupdater. Integrity checking of installed files is not generally available in a package deployment tool. A non-zero return value from the installer indicates a possible error and a notification is provided.

The Appupdater client supports multiple repositories and is designed to operate in a very decentralized model. This is very similar to how the Maemo Linux[13] distribution operates in practice. Generally, each developer creates a repository for distributing the completed applications and dependencies to other users. The client-side user must make a one time configuration change to add the URL for each repository. From that point on the user is "subscribed" to easily obtain any future updates directly from the application's author.

5.3 Creating Packages to Build a Repository

Just as with Linux repositories, maintaining more than a few packages can quickly take a significant amount of time, especially when tracking *every* version released. In addition many Windows programs require the installer to be run manually to extract the files, further slowing the package creation process. When generating a repository the time required for gathering all the meta-information for a package can be reduced by automatically installing an application or extracting files from installers without having to actually install the software using an extraction program such as 7-zip[14].

5.4 Server Tools

In addition to the Appupdater client, we provide a set of server tools to help streamline the process of building a new version of a package. By creating a configuration file with information that usually does not change between releases like download URL, silent install switches, and installed program executable file names, the process can nearly be automatic. The server side tools for Appupdater can automate the download, running of the installer, and generating the appropriate hashes to create a package instead of it being a slow, manual process for each new version.

6. Additional Applications

We now have a constantly growing database of hashes of known application versions. We can now cross reference detected applications against the Common Vulnerabilities

[13] http://maemo.org/

[14] http://www.7-zip.org/

and Exposures (CVE)[15] database to find known software flaws on a system [Martin 2001].

Malware detection is possible with the database of hashes of known application versions. For example if a system reports a version of an application is installed and its hash does not match a "known good" as defined in the database, then the file has been modified and could contain malware or a trojan.

7. Future Work

The current repository structure requires a client to download all meta-information from a repository, not just the information for applications that it needs. As the size of a repository increases, the XML data structure begins to break down. These files take too long to download and parse. This process might be more efficient as a web service, but also has privacy implications.

We wrote Appupdater with cross-platform applications in mind. Any operating system with a Python implementation can use this software when provided with a repository for that operating system. Mac OS X and it's .dmg installers should be compatible with Appupdater, but there is no repository data at this time.

Further development could include a "crawler" used to check application websites for version updates instead of waiting for submissions from users, similar to a search engine. The crawler could pass this information to the server tools to further automate the package generation process.

8. Conclusion

Using the Linux package and repository structure as a model, we created a more generic system to include other operating systems. This is accomplished by adding additional attributes to what typically defines a Linux package. We are also able to make downloads more resilient through the network oriented design of the repository information. Using this new repository structure we created a client that is able to implement automatic software upgrades.

Acknowledgments

The Metalink Project for the development of the Metalink Specification. This provided a basis for a robust download description format which was extended for use as a software repository.

Jeffrey Ellen for his mentorship and valuable comments regarding this publication.

References

David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In Proc. of the IEEE Symp. on Security and Privacy, pages 143157, Oakland, CA, USA, 2008.

Anthony Bryan. Metalink 3.0 Specification (Second Edition). http://www.metalinker.org/Metalink_3.0_Spec.pdf

A. Bryan, T. Tsujikawa, N. McNab, P. Poeml. The Metalink Download Description Format (Draft). http://tools.ietf.org/html/draft-bryan-metalink

Roberto Di Cosmo, Stefano Zacchiroli, Paulo Trezentos. Package upgrades in FOSS distributions: details and challenges. In ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), 2008.

Olivier Crameri, Ricardo Bianchini, Nikola Knezevic, Willy Zwaenepoel, Dejan Kostic. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. SOSP'07, October 14-17, 2007, Stevenson, Washington, USA.

E. Dolstra and A. Löh. NixOS: A purely functional Linux distribution. In ICFP'08.

Archana Ganapathi, Yi-Min Wang, Ni Lao, Ji-Rong Wen. Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems. Proc. IEEE International Conference on Dependable Systems and Networks (DSN), June 2004.

Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnovic. Planet Scale Software Updates. In Proceedings of SIGCOMM, September 2006.

Slinger Jansen, Gerco Ballintijn, and Sjaak Brinkkemper. A process model and typology for software product updaters. In 9th European Conference on Software Maintenance and Reengineering (CSMR), 2005.

Havard Johansen and Dag Johansen. Resilient Software Mirroring With Untrusted Third Parties. In ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), 2008.

Arjen Lenstra and Benne de Weger. On the possibility of constructing meaningful hash collisions for public keys. ACISP 2005, Brisbane, Australia, July 4-6, 2005.

Robert A. Martin. Managing Vulnerabilities in Networked Systems. Computer, vol. 34, no. 11, pp. 32-38, Nov. 2001, doi:10.1109/2.963441

Michael Rabin. Fingerprinting by Random Polynomials. Center for Research in Computing Technology Harvard University Report TR-15-81, 1982.

[15] http://cve.mitre.org/