A Look In the Mirror: Attacks on Package Managers

Author Names Removed for Anonymous Submission

ABSTRACT

Package managers are a privileged, centralized mechanism for software update and are essential to the security of modern computers. This work studies the security of ten popular package managers. These package managers use different mechanisms to provide security including signatures embedded in the package, signatures on metadata detached from the packages, or a signature on the root metadata (a file that contains the secure hashes of the package metadata). The security models used by these package managers are compared and contrasted.

The threat model used to evaluate security in this paper is an attacker that controls a mirror (a copy of the main repository's contents for a distribution). We demonstrate that it is trivial for an attacker to control an official mirror for a popular distribution. An attacker can compromise a client who either installs software created by the attacker or installs an outdated version of a package with a vulnerability the attacker knows how to exploit. Furthermore, every package manager studied can be compromised by an attacker who controls a mirror *without compromising a private key.* In fact, 5 of the 10 package managers studied have security flaws that allow an attacker to compromise every client that requests a package from the mirror. We estimate that an attacker with a mirror that costs \$50 per week could compromise between 150 and 1500 clients per week depending on the package manager.

An existing package manager is modified to add a layered approach to security where multiple signatures are used. The updated package manager is evaluated in practical use. By using a layered approach to security, the package manager provides a high degree of usability and is not vulnerable to the attacks on existing package managers. The overhead of additional security mechanisms is 2-5% in practice and so should not be a deterrent.

The purpose of this work is to not only point out security issues and provide solutions but also to raise an alarm to the imminent threat of attacks on package managers. Package managers are a weak point in the security of modern computers. Given the simplicity of compromising systems through package managers, developers and distributions must act quickly and intelligently to avert disaster.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; K.4.1 [**Social Issues**]: Abuse and Crime Involving Computers

General Terms

Security

Keywords

Package Management, Mirrors, Replay Attack

1. INTRODUCTION

Package managers are a popular way to distribute software (bundled into archives called packages) for modern operating systems [1, 2, 3, 16, 17, 20, 21, 23, 26, 27]. Package managers provide a privileged, central mechanism for the management of software on a computer system. As packages are installed by the superuser (root) and normally are shared by all users of the computer, package management security is essential to the overall security of the computer system.

This work demonstrates that it is trivial for an attacker to control an official package mirror. Mirrors are used to provide fault tolerance and offload traffic from the distribution's main repository but are usually hosted by outside organizations. Therefore, package managers must recognize and mitigate the dangers posed by malicious mirrors given a threat model in which they cannot trust the mirrors from which they obtain content.

This paper evaluates the security of the eight most popular [8, 14] package managers in use on Linux: APT [1], APT-RPM [2], Pacman [3], Portage [16], Slaktool [20], urpmi [23], YaST [26], and YUM [27]. Also examined is the popular package manager for BSD systems called ports [17] and a popular package manager in the research community called Stork [21]. These package managers use one of four different security models: no security, signatures embedded within packages, signatures on detached package metadata, or signatures on the root metadata (a file that contains the secure hashes of the package metadata).

The package managers that provide some form of security recognize the threat posed by mirrors that are under the control of a malicious entity. However, each of the ten package managers studied is vulnerable to attacks by malicious mirrors. *This means that any attacker that controls a mirror can compromise a large number of computers on the Internet today.*

Remarkably, many the security flaws that enable these attacks are conceptually simple to understand as well as easy fix in practice. Neither the flaws discovered nor the solutions to them are novel. However, surprisingly these problems exist across a wide range of package managers that were implemented by different developers.

This work demonstrates there is an ordering to the security of package manager techniques and this order is preserved even as security fixes are applied. Having no signatures allows the most egregious attacks, followed by package signatures, signatures on detached package metadata, and finally signatures on the root metadata present the most security.

However, there are usability concerns with many package managers, most notably the ability to verify a stand-alone package (a package obtained from a source other than the main repository). Signatures on root metadata do not provide a convenient way to verify stand-alone packages and so the user is likely to install such packages without using security checks. In contrast, package managers that use signatures on detached package metadata or package signatures can verify stand-alone packages.

Because of the usability strengths and weaknesses of different techniques for providing security, this work recommends a layered approach created by combining two techniques: signatures on detached package metadata and signatures on the root metadata. This technique provides the security strengths and the usability strengths of both types of signatures. The overhead of using multiple security techniques is between 2-5%, which demonstrates that overhead is not a deterrent. The layered approach advocated by this work has been added to a package manager and is now in practical use by thousands of clients around the world.

1.1 Contributions

This work makes several contributions:

- The security mechanisms in package managers are analyzed and classified by effectiveness, usability, and efficiency. The relative strengths and weaknesses of different mechanisms are compared and contrasted. This work also describes means by which to improve the security of existing package managers without adding additional signature mechanisms.
- An existing package manager is modified to incorporate the changes discussed in this work. This package manager is deployed and evaluated in practice with conclusions drawn based upon real-world experience.
- The attacks in this paper are shown to be more broadly applicable and affect users who do not use mirrors when the attacker can intercept and modify traffic.
- Using traces of traffic from mirrors, the overall threat posed by mirror compromises is estimated. All popular package managers are shown to be vulnerable to compromise by an attacker that controls a mirror. *This work raises awareness to the threat of attack using mirrors.*

1.2 Map

This paper begins by providing background information about how package managers work (Section 2). The scope and feasibility of an attacker obtaining a mirror are then discussed, including the resulting threat model (Section 3). Then follows an analysis of the security of package managers in this threat model (Section 4). Following this is a discussion of the usability provided by these security models (Section 5). Next is a discussion about how security and usability can be maximized by utilizing signatures on both the root metadata and the detached package metadata (Section 6). Results are then provided to demonstrate the efficiency and security of package managers (Section 7). The paper discusses related work (Section 8) and then concludes (Section 9).

2. BACKGROUND

This section provides background information about package managers which is important in order to better understand potential vulnerabilities.

2.1 Package Formats

Packages consist of an archive containing files and, in most cases, additional *embedded package metadata*. For a given package, the embedded package metadata contains information about the other packages it needs to be able to operate (the *dependencies*), functionality the package possesses (what the package *provides*), and various other information about the package itself. The most popular package format [18] has space for one signature. Other popular package formats have no standard field for signatures, although in some cases extensions exists to support signatures [7, 9].

2.2 Package Managers

Clients use a package manager to install packages on their system. A package manager gathers information about packages available on package repositories. Almost all package managers automatically download requested packages as well as any additional packages that are needed to correctly install the software. This process is called *dependency resolution*. For example, a requested package `foo` may depend on `libc` and `bar`. If `libc` is already installed, then `libc` is a dependency that has been resolved (so no package needs to be added for this dependency). If there is no installed package that provides `bar`, then `bar` is an *unresolved dependency* and a package that *provides* `bar` must be installed before `foo` may be installed. The package manager may be able to locate a package that provides `bar` on a repository.

The packages that are chosen to fulfill dependencies may have unresolved dependencies of their own. Packages are continually added to the list of packages to be installed until

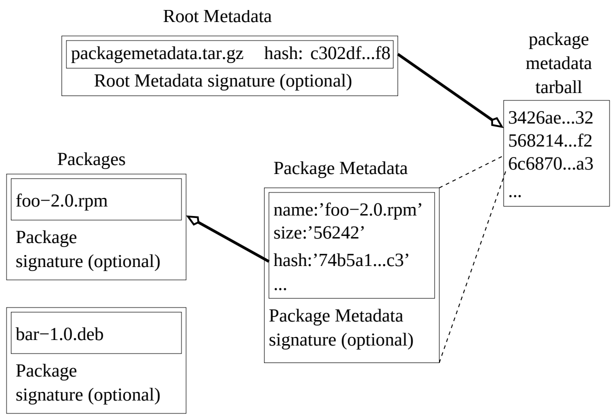

Figure 1: Repository Layout. The root metadata, package metadata, and packages may all optionally have signatures depending on the support of the package manager. Arrows point from a secure hash to the file it references.

either the package manager cannot resolve a dependency (and produces an error) or all dependencies are resolved.

2.3 Repository

A package repository is usually an HTTP or FTP server that clients can obtain packages and *package metadata* from. The package metadata for a package is usually just a copy of the embedded package metadata in the package. Package managers download the package metadata from a repository so that they know which packages are available from that repository. This also provides the package manager with dependency information needed perform dependency resolution. To facilitate convenient downloading of package metadata, most repositories store all of the package metadata in a small number of compressed files, often a tarball.

In addition to the package metadata files, each repository has a *root metadata* file. The name and location of the root metadata file varies for different repository formats, but the contents are similar. The root metadata provides the location and secure hashes of the files that contain the package metadata.

Figure 1 shows the layout of a repository. A package manager downloads the root metadata and uses that to locate the files containing the package metadata. The package manager then downloads the package metadata. The package metadata is used to determine package availability as well as for dependency resolution. Packages are then downloaded and installed. The root metadata, package metadata, and packages may be signed depending on the security model of the package manager.

2.4 Mirror

It is common for a distribution to have more than one server from which users can download packages and package metadata. There is usually a *main repository* for a distribution whose contents are copied by many separate *mirrors*. A mirror typically contains exactly the same content as the main repository and is updated via `rsync` or a similar tool. A mirror differs from a main repository in that a mirror is not intended to have packages directly added to it or removed from it by its administrators. Packages are added or removed only on the main repository and the mirrors later obtain the changes when copying the main repository.

A mirror can be *public* (available for anyone to use) or *private* (restricted to a specific organization). A mirror may also be endorsed by a distribution for public use, typically when that the distribution is in contact with the mirror maintainers. This type of mirror is called an *official* mirror (the terminology used outside of this document varies by the distribution). Official mirrors are by definition public because the distribution is endorsing their use to the public.

It should be noted that some distributions do not use official mirrors hosted by outside organizations. One type of distribution without official mirrors are tiny distributions that can support all of their clients by a small number of repositories that the distribution directly controls. Another example of a type of distribution without official mirrors is a distribution that requires users to pay for the distribution. These costs often are used to support a set of internally maintained mirrors for the distribution. Alternatively, a distribution may allow or require each organization using the distribution to set up their own private mirrors for the organization's own use.

However, official mirrors hosted by outside organizations are the predominant mechanism for software distribution with all but two popular distributions [8, 14] relying on official mirrors. Official mirrors are essential for most distributions to reduce cost and management overhead.

3. MIRRORS

This section examines the feasibility of obtaining a mirror to attack package managers (Section 3.1) and the resulting threat model (Section 3.2).

3.1 Obtaining a Mirror

To evaluate the feasibility of controlling mirrors of popular distributions, we attempted to set up official public mirrors for the CentOS, Debian, Fedora, openSUSE, and Ubuntu distributions. A fictitious company (Lockdown Hosting) with its own domain, website, and fictitious administrator (Jeremy Martin) were used as the organization maintaining the mirrors. A server with a monthly bandwidth quota of 1500 GB was leased for $200 per month through The Planet (`www.theplanet.com`).

Setting up a public mirror for each distribution involved acquiring the packages and metadata from an existing mirror and then notifying the distribution maintainers that the mirror was online and available for public usage. The distributions varied in terms of the degree of automation in the public mirror application and approval process as well as whether newly listed mirrors have traffic immediately and automatically directed to them. Regardless of whether the application and approval process was completely automated, the same basic information was required, including the name of the organization providing the mirror, the contact email address for the mirror administrator, and the mirror's available bandwidth. For the distributions that automatically direct their clients to mirrors (CentOS, Fedora, openSUSE), the specific mirrors a client is directed to are based upon a combination of the geographic locations of the client and the mirrors (determined by their IP addresses) as well as the number of clients each mirror can handle, based upon their available bandwidth. More details about signing up individual mirrors can be found in the Appendix.

3.2 Threat Model

There are several ways that an attacker can compromise a client. First, if a package manager installs arbitrary code provided by the attacker, the attacker has compromised the client's system. Second, the attacker can cause a client to install an outdated package with a vulnerability the attacker knows how to exploit (called a *vulnerable package*). The attacker can then compromise the client by exploiting the vulnerable package.

The threat model used in this paper involves an attacker that controls an official mirror for a distribution. The threat model can be summarized as:

- The attacker may modify any files served by the mirror. — This is logical because the attacker has root access to the mirror.
- The attacker does not know what package the client will request a priori. — While the attacker knows this for some package managers, this is not true for all package managers, therefore our threat model assumes the attacker does not have this ability.
- The attacker *does not* have a key trusted to sign packages, package metadata, or the root metadata. — Mirrors do not usually possess the private key used to sign files, they only copy previously signed files from the main repository.
- The attacker has access to outdated packages, outdated package metadata, and outdated root metadata. — There are many outdated mirrors on the Internet where the attacker can obtain these files if they haven't already saved copies.
- The attacker is aware of vulnerabilities in some outdated packages and is able to exploit those vulnerabilities. — By looking at change logs and updates to software source files, it is possible for an attacker to discover vulnerabilities. Also, some websites [13] make toolkits and proof-of-concept code available that can exploit known vulnerabilities.
- The attacker does not know of a vulnerability in the latest version of any package. — Zero-day vulnerabilities are obviously useful to an attacker, but are unlikely to be known by most attackers.
- If a package manager supports signatures, signatures are used. — If a client or distribution chooses not to use signatures supported by their package manager, they are as vulnerable as if they used a package manager that does not support signatures.
- Expiration times the root metadata are used if supported and vulnerable versions of packages are only listed in expired root metadata. — The root metadata is a single, small file so it is feasible for the main repository to sign it relatively frequently with short expiration times.

More formally, let p_i be a package in the total set of packages P, where each package p_i consists of a set of versions $V_i = \{v_{i0}, v_{i1}, \ldots, v_{in}\}$. The attacker knows how to exploit vulnerabilities in some (possibly empty) subset of the total versions of the package $\hat{V}_i \subset V_i$. Note that $\hat{V}_i \neq V_i$ since the most current version is not in $\hat{V}_i$. If a client installs any package in $\hat{V}$ (comprised of all versions of all packages the attacker knows how to exploit), the attacker can compromise the client. Each version of a package also has a start time t^s_{ij} and end time t^e_{ij} when the version was available on the main repository. Note that it not uncommon for the older version to be removed when the new version is added $t^e_{ij} = t^s_{ij+1}$ or for the old version and new version to overlap $t^e_{ij} > t^s_{ij+1}$ but it is rare for no version to be available for any time period $t^e_{ij} < t^s_{ij+1}$. Each package also has a frequency of request (denoted f_i) which is the fraction of package requests that are for p_i.

This formalism will be used in Sections 4 and 7 to quantify the vulnerability to attack of the security mechanisms used by different package managers.

4. SECURITY OF PACKAGE MANAGERS

The security of a package manager varies depending on what the package manager signs. This section explores the security strengths and weakness of signatures on different data along with implementation pitfalls observed in package managers (and how to fix them). This section then classifies the security of different signatures into a list ordered by increased security.

The discussion groups package managers with similar security characteristics together. The first group of package managers do not use signatures (Section 4.1). The second group of package managers sign packages but do not include signatures on package metadata or root metadata (Section 4.2). The third group of package managers sign package metadata but do not sign the root metadata (Section 4.3). The final group of package managers sign the root metadata (Section 4.4).

4.1 Package Managers Without Security

There are three popular package managers that do not provide security: Pacman, ports[1], and Slaktool. These package managers do not sign packages, package metadata, or the root metadata file. As a result any attacker that controls a mirror can trivially create an arbitrary, malicious package and clients will install the attacker's software.

4.2 Package Signatures

[1]A version of ports used by NetBSD did support package signatures at one time [6], but this has been obsoleted and is not maintained or used.

The popular package managers YUM and urpmi rely on signatures embedded in packages to provide security. Both package managers use package signatures for security. There is no protection of package metadata or the root metadata.

Package Metadata An attacker can create arbitrary package metadata and neither YUM nor urpmi verifies the package metadata at any point *even if a package is downloaded.* After downloading a signed package and verifying the package's signature, the embedded package metadata in the signed package is not compared against the previously downloaded package metadata. Due to this, an attacker can cause additional packages to be installed along with the package the client intended to install. For example, suppose an attacker knows how to compromise a vulnerable package v_{ij}, the attacker can then change the package metadata of every package on the mirror to depend on v_{ij}. Any client that wants to installs any package will also install the vulnerable package because of the dependency. It is possible to modify YUM and urpmi to prevent this attack by verifying that the embedded package metadata is the same as the downloaded.

Packages An attacker can choose which versions of packages to include on the mirror. This means that an attacker can choose to have the mirror provide versions of packages that were on the main repository at different times. As a result, an attacker can compromise a client that requests any package for which the attacker has a vulnerable version.

Analysis The rate of compromise of clients using the mirror varies depending on whether or not the package manager verifies that the package metadata matches the package. If the package manager does not verify the package metadata matches, the attacker can add a dependency on a specific package and version in $\hat{V}$ to the package metadata of every package. Thus every client that requests a package will be compromised.

If the package managers are modified to verify that the package metadata matches the package, they are still vulnerable to an attacker who places older, vulnerable package versions on the mirror. The probability of the client downloading a vulnerable package is the chance of downloading a package for which there exists a vulnerable version. The probability of a package download by an uncompromised client resulting in a compromise (denoted X) can be estimated as:

$$X = \forall p_i \exists v_{ij} \in \hat{V}_i (\sum f_i) \tag{1}$$

4.3 Package Metadata Signatures

The Portage and Stork package managers sign package metadata, however they do so in different ways. Each package in Portage has a separate, signed package metadata file for each version of the package. The package metadata contains the secure hash of the package (possibly along with related files such as patches). In contrast, Stork users create a single file that contains a timestamp and the secure hash of the package metadata for all of the packages that the user trusts. Users can also delegate trust to other users and all users typically delegate trust to a single "distribution" user. The analysis of Stork focuses on the security of the packages trusted by the distribution user because the security of the distribution user affects all clients.

Package Metadata The tampering attacks described for the package managers that sign only packages are not effective when the package metadata is signed. That is true because the client can verify the signature on the package metadata before using the package metadata to do dependency resolution. There is no need to verify that the package metadata matches the embedded metadata in the package because the secure hash of the package in the package metadata protects the package from tampering.

Packages Resistance to metadata tampering does not imply that the package manager is resistant to attack. In Portage, since each package has a different file for metadata signatures, an attacker can choose to have any combination of packages (such as those that include only older versions with known vulnerabilities) available on the mirror. This an attacker to choose to host vulnerable versions of packages that existed on the main repository at different times.

However, in Stork all of the package metadata signatures are in the same file. This prevents an attacker from providing package metadata that existed on the repository at different times. An attacker must instead provide a consistent snapshot of the main repository at a specific time. Timestamps prevent the attacker from providing a client with an older file than it has seen.

It is worth noting that this does not prevent an attacker who controls a mirror from freezing the mirror contents at a particular point (i.e. stop updating from the main repository). This means that an attacker can convince clients to use vulnerable packages so long as the client has not downloaded a file with a newer timestamp.

Analysis The vulnerability to attack of having individually signed package metadata is the same as is presented in Equation 1. This is because the attacker can choose vulnerable packages that were on the mirror at different times.

However, the same is not true when the signatures is over a group of package metadata. This prevents the attacker from choosing to provide a client with versions of packages that did not exist on the main repository at the same time. This means that if $\hat{V}_i = v_{ia}$ and $\hat{V}_j = v_{jb}$ and $t^e_{ia} < t^s_{jb}$, then the attacker must choose to compromise either clients who use package p_i or clients who use p_j. An intelligent attacker will choose a time frame that has the maximum probability of a client downloading a vulnerable package. Thus the attacker will want to maximize the probability of compromise (denoted X) and will choose the time frame t' (out of the time period when the main repository has been online T) so that:

$$X = \max_{t' \in T} \left\{ \forall p_i \exists v_{ij} \in \hat{V}_i \Big|_{t^s_{ij} \le t' \le t^e_{ij}} (\sum f_i) \right\} \tag{2}$$

4.4 Root Metadata Signatures

The package managers APT, APT-RPM, and YaST sign the root metadata. All three of these package managers optionally support packages signatures as well, but this functionality is not widely used in practice.

Package Metadata Package metadata is stored in compressed files and the secure hashes of those files are stored in the root metadata file. As the root metadata is protected by a signature, the package metadata is protected from tampering.

Packages The signature on the root metadata prevents a mirror from hosting versions of packages that were on the main repository during different time periods. The attacker must choose a time period of the main repository to copy and provide exactly those files.

The signature on the root metadata does not prevent an attacker from replaying old versions of the root metadata at a later date. An attacker may want to do this in response to newly discovered vulnerabilities in older packages. An attacker also may choose to change the set of packages it provides after compromising one set of clients so as to go after clients with different package interests.

It is easy for package managers to prevent the replay of older metadata by adding a timestamp to the root metadata and checking that the downloaded version is newer than the version the client last downloaded. In fact APT and APT-RPM have timestamps in the root metadata and merely need to add the check. YaST, which can use different repository structures and metadata formats, only has a timestamp available in the root metadata for some of its supported formats. However, there is a timestamp within the gpg signature of the root metadata and this signature timestamp could be used without requiring a modification to any root metadata file formats.

Another avenue of attack comes from an attacker freezing the mirror contents at a particular point in time. The clients will continue to use the old mirror data and may install vulnerable packages as vulnerabilities become known within the frozen set of packages. This attack is similar to the freezing attack described in the previous section.

To mitigate the effectiveness of freezing attacks, packages managers could add an expiration time to the root metadata. Clients would refuse to use a root metadata file if the current time is greater than the expiration time. Since the root metadata is a single, small file it is feasible to re-sign this file often and require every mirror to be frequently updated (most distributions already require their public mirrors to update no less frequently than once a day).

YaST uses gpg to check the signature on the root metadata and gpg supports expiration times. However, YaST does not recheck the root metadata's signature if the file has not changed since the last time it downloaded it. This means that without additional changes to YaST, adding root metadata signature expiration times would prevent an attacker from providing a client old metadata files that had already expired, but old metadata files that had not yet expired could be given to the client and frozen at that point in time by continuing to provide the client the same root metadata files. During testing it was noted that gpg expiration times are not in practical use. For these reasons, YaST is vulnerable to freezing attacks.

Name	Signs	Package Installation	Metadata Abuse
pacman	nothing	arbitrary	arbitrary
slaktool	nothing	arbitrary	arbitrary
YUM	(1)	alongside	arbitrary
urpmi	(1)	alongside	arbitrary
Portage	(2)*	mismatch	replay / freeze
Stork	(1)*, (2)	consistent	freeze
APT	(1)*, (3)	consistent	replay / freeze
YaST	(1)*, (3)	consistent	replay / freeze
APT-RPM	(1)*, (3)*	consistent	replay / freeze

Figure 2: Package managers, their protection mechanisms and vulnerabilities. The protection mechanisms are numbered (1) packages, (2) package metadata, (3) root metadata. '*' indicates that support exists but is not in common use.

Analysis Root metadata signatures that are vulnerable to replay and freezing attacks still prevent the attacker from providing a client with versions of packages that did not exist on the main repository at the same time. This has a similar effect as having a single file that contains all of the package metadata signatures. As a result, it has the same vulnerability as displayed in Equation 2 because the attacker must choose package versions that existed on the main repository at the same time.

If the package manager supports root metadata signatures and uses expiration times and protect against replay attacks, it is not vulnerable to compromise in our threat model. As a result, it is safe for a client to use a compromised mirror.

4.5 Classification

The security mechanisms and vulnerabilities of the package managers are summarized in Figure 2. All of the package managers studied are vulnerable to metadata tampering by an attacker that controls a mirror. The result of metadata tampering is that an attacker may cause clients to install vulnerable packages. Depending on the package manager's security mechanisms, the result can be any of the following, where those listed first also imply those listed after: *arbitrary* packages created by the attacker are installed, any vulnerable package can be installed *alongside* non-vulnerable packages a client installs, *mismatched* outdated packages are installed (in that they existed on the main repository at different times), or *consistent* outdated packages are installed (in that they existed on the main repository at the same time but are outdated).

Based upon the observation and analysis of the security in existing package managers, it is possible to similarly classify the security mechanisms. Given these classifications, one can obtain an ordering of the security of the mechanism.

Figure 3 shows a classification of different security mechanisms. Clearly, having no signatures allows the most attacks and is the most vulnerable. Signatures on packages provide a definite improvement over no signatures, but gives the attacker the ability to manipulate metadata arbitrarily and provides attackers the ability to populate a mirror with packages of mismatched versions, or, if package metadata isn't verified using the signed packages, the ability to causing vulnerable packages to be installed alongside any non-vulnerable packages. Signatures on package metadata pre-

Classification Name	Best Case		Common Case	
	Package	Metadata Abuse	Package	Metadata Abuse
No Security	arbitrary	arbitrary	arbitrary	arbitrary
Package	mismatch	replay / freeze	alongside	arbitrary
Package Metadata	mismatch / consistent	freeze	mismatch	replay / freeze
Root Metadata	current	none	consistent	replay / freeze

Figure 3: Classification of package manager protection schemes. This demonstrates both the security that is possible to achieve using a scheme as well as what is commonly provided by existing implementations.

vent the attacker from doing more than replaying or freezing the package metadata but if the signatures are in separate files, the attacker can still mismatch versions of packages. By preventing replay and freezing attacks in package managers that sign the root metadata, a package manager will only install current packages and is immune to metadata tampering.

5. ADDITIONAL USABILITY NEEDS

This section focuses on additional usability requirements users have for package management. Most importantly, the use case where a user has an uninstalled package on their computer they need to verify.

The standard use case of the package managers and their security mechanisms is where a user needs to securely install software from a remote, trusted repository. However, it is not uncommon for a user to have a stand-alone package that they need to verify is free from tampering and was created by a party they trust. Stand-alone packages are packages that are not obtained through the package manager's normal channels at install time. Stand-alone packages may have been obtained manually from unofficial sources or may even be packages a user has created.

The reasons [24] that users commonly state in support of stand-alone package verification can be summarized as follows.

- Stand-alone packages can be checked for tampering immediately without requiring the relevant signed repository metadata for the specific packages.
- There are sources that gather and distribute packages originating from many other places, such as `rpmfind.net` and `rpm.pbone.net`. It is important for users downloading packages from these sites to be able to verify that the packages they download haven't been tampered with.
- Packages may be created by a user who does not run a repository. The user may want to verify their own packages at a later time (for instance, if the system the packages were stored on was compromised at some point).
- There are "derived distributions" that are created by slight modification to an existing distribution's files. When using packages from a derived distribution, users are able to identify whether a given package is the original from the parent distribution.
- If the method for verifying the package allows multiple signatures, these signatures allow the user to trace the path a package has taken (developer signature, distribution signature, etc.).
- Developers would like to verify packages as they move around their own infrastructure.

The signing of only root metadata does not allow any practical way to verify stand-alone packages. Package managers that use signed root metadata could be modified to keep copies of all metadata obtained from the repository for future verification of stand-alone packages, but this only helps for packages a user manually downloads from the same repository that they access through their package manager. This also fails to satisfy one of the primary reasons given for being able to verify stand-alone package signatures: verifying signatures for files when they are only available for manual download and installation, not through a repository.

Package managers that sign package metadata tend to be more able to meet the needs of stand-alone package verification than the package managers who only sign root metadata. However, the way in which package metadata is stored has a major impact on usability in this case. Similarly with package managers that sign only root metadata, package managers would need to store old package metadata and this would only be of use for verifying stand-alone packages that came from a repository the user normally users. In other cases the user would need to be sure to always keep the signed package metadata with the package for verification purposes. This is far from an ideal option.

Signatures embedded in packages are thus the most practical option and provide the greatest ease of use when stand-alone packages must be verified. All that a user needs in order to verify a package is the package itself. A drawback with having signatures in the package is that signatures are constrained by the limits of the package format. The most popular package format [18] has space for one signature. Other popular package formats have no standard field for signatures, although in some cases extensions exists to support signatures [7, 9].

Using signatures embedded in packages for stand-alone package verification does have complications, though. Notably, users must have the requisite public keys available in order to verify package signatures. They must also ensure on their own that packages they are installing are not outdated and have vulnerabilities. However, there are many scenarios where a user can use embedded package signatures in a way that increases security in their specific situation.

6. DISCUSSION

It is important to remember that all of the package managers that use some form of security mechanism are considered to be secure by their users. The implication by providing cryptographic signatures of data is that the data is secure.

Many security-aware people have been involved in the design and development of these package management systems over long periods of time and yet every one of them suffers from critical security problems. Therefore, it is vital to carefully consider what the threats against package managers are, what each form of security specifically protects against, and to identify ways in which existing package managers can increase their level of security.

Each of the security mechanisms available have trade-offs. Signatures on packages present unacceptable security risks for many users but are conceptually simple to use (often with unfortunate limitations on the number of signatures per package). Package metadata signatures have much better security and allow multiple signatures per package. Signatures on the root metadata have the best security but have low usability when stand-alone package verification is a requirement.

6.1 Deployment Experiences

To gain more experience with what works well in practice, we took the existing package manager Stork and added root metadata signing. We also ensured that Stork was not susceptible to replay or freeze attacks. Since Stork already supported both package signatures and package metadata signatures, this allowed us to experiment with all types of signatures in a single package manager.

Stork was chosen for three reasons. First, since Stork is a research project, the authors were receptive to "experimental" changes to Stork. Second, Stork already incorporated two of the three types of signatures, including package metadata signing which seemed the most complex to add to an existing package manager. Third, Stork is used almost entirely by researchers and we thought it would be easier to convince researchers to try experimental code.

Once the changes to Stork for root metadata signing were tested and verified by the Stork team, they pushed the changes in their main release. Interestingly, Stork differed from all other security-using package managers in that there was no key already trusted by clients to validate communication from the repository. As the only signed files in Stork were the package metadata files signed by individual users, there had never been a need for the repository to have its own key that the clients needed to trust. This resulted in the necessity of distributing a repository key to clients in order for them to make use of the new root metadata signatures. The key was included with the initial release of the updated version of Stork. This initial key distribution was able to be done securely because the majority of users, through their trusted packages files, delegate trust to the Stork team to provide them updated Stork packages. — It is important to note that, unlike in other package managers, Stork's design meant that users would not be using this key for trusting packages, but rather only verifying metadata files downloaded from the repository.

The resulting changes were transparent to the users. Ultimately, there were few comments about the addition of root metadata signatures since the the existing security mechanism (package metadata signatures) was retained without modification. Though transparent to the users, they gained increased security through the addition of root metadata signatures.

We then tested the different signature mechanisms and found that with the use of package metadata signatures, there was no reason to use package signatures. We examined packages on the Stork repository to find that user-uploaded packages did not include package signatures, indicating that researchers were not using the optional package signature feature of Stork. From this we conclude that package metadata signatures and package signatures are redundant and it is sufficient for only one of these to be available to users.

6.2 Deployment Conclusion

Since no one scheme works well from both a security and a usability standpoint, we propose that package managers should use multiple security mechanisms. It is clear that root metadata signatures should be included because of the security benefits. It also seems advantageous to have either package metadata signing or package signatures. Whether signatures on package metadata are superior to signatures on packages from a security and usability context depends upon the specifics of how a package manager is implemented. In either case, there appears to be little security or usability gain from including both signed packages and package metadata signatures. By combining root metadata signatures with either signed package metadata or signed packages, a package manager can obtain a high degree of security and excellent usability.

6.3 Broader Applicability

While the focus of this work is on compromises involving a mirror, this threat model has broader applicability. For package managers that do not use HTTPS, an attacker who controls a mirror has the same characteristics as an attacker who can launch a man-in-the-middle attack or otherwise divert repository traffic to their system. For this reason, even clients that use the main repository for their distribution are not safe from attack.

However, the threat of man-in-the-middle attacks can be reduced by using HTTPS. Unfortunately, many package managers do not support HTTPS. Even for those package managers that do support HTTPS, it is only widely used in one popular distribution, Red Hat Enterprise Linux. From a practical standpoint, HTTPS removes the threat of a man-in-the-middle attack, but it is not applicable to the problem of a compromised mirror because the client is communicating with the computer it intends to.

A more dire threat than a man-in-the-middle attack is if an attacker were able to control the main repository for a distribution. This is a serious threat because the mirrors for the distribution will all copy the contents of the main repository. Distributions tend to be very protective and proactive with the main repository, so this seems unlikely. However, if mirrors do not use a secure connection to synchronize with the main repository (as is the case in most of the popular distributions), then the mirrors are susceptible to man-in-the-middle attackers who pose as the main repository.

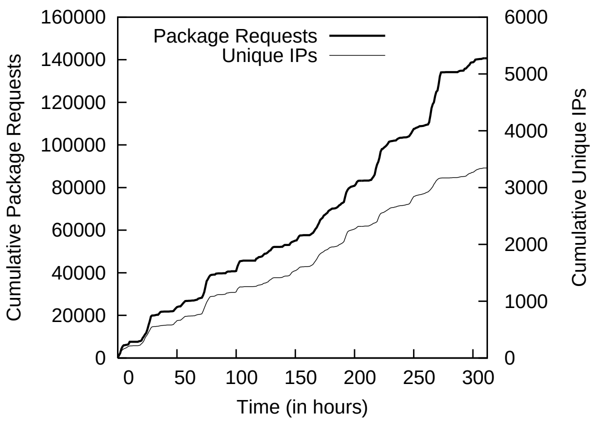

Figure 4: CentOS Mirror Traffic. This figure presents the cumulative package downloads and requests from unique IP addresses for the CentOS mirror over a 13 day period.

6.4 (More) Safely Using an Insecure Package Manager

Many fundamental security issues remain for the users of insecure package managers. However, both the distributions and users can help to mitigate the effectiveness of attacks even when using an insecure package manager.

- If the package manager supports HTTPS, the distribution can set up repositories or mirrors that support HTTPS transfers. This will mitigate man-in-the-middle attacks.
- The distribution can review their mirror policy carefully and validate administrator credentials before putting a mirror on the official mirror list. This will make it harder for an attacker to obtain a mirror.
- The distribution's mirrors should use a secure connection (e.g. SSH) to synchronize with the main repository. This will make it harder for an attacker to impersonate the main repository in an attempt to get the mirrors to copy their content.
- Users can check that the versions of the packages their package manager recommends for installation are recent through another source. This will help to detect malicious mirrors serving old packages.

7. RESULTS

To understand the impact of an attacker that controls a mirror, a trace of package requests was conducted on a CentOS mirror and used to estimate the number of clients that could be compromised by an attacker. Using an estimation was necessary as it would be unacceptable to deploy live compromises on the public Internet.

7.1 Mirror Traces

The CentOS mirror was chosen for this analysis because it was the longest mirror experiment that was conducted, lasting 13 days. The package access trace gathered from the CentOS mirror is shown in Figure 4. The number of package requests and number of requests from unique IP addresses increase roughly linearly over this time period. Assuming the CentOS user base is not growing faster than our mirror serves clients, we would expect the number of unique IP addresses to flatten out over time, however our trace is not long enough to capture that effect. Since clients are counted as unique by IP address, multiple clients behind a NAT box or proxy are counted as a single client. There are many instances where a single IP address has several orders of magnitude more package requests than the median client, often with many requests for the same package. This implies that clients are using NAT boxes or proxies in practice.

The openSUSE and Fedora mirrors (not shown) had similar traffic effects as the CentOS mirror. The Debian and Ubuntu mirrors (not shown) were both up for only a short period, but did not demonstrate this effect. We suspect this is because they do not automatically distribute requests among the mirrors, instead requiring manual selection by a user. Since our mirrors were only listed a few days, they did not attract a large number of users on Debian and Ubuntu.

7.2 Package Versions and Vulnerability

To perform our analysis, we needed to know the distribution of package versions over time on the mirror, as well as which of those versions are vulnerable to attack. Information about the 58165 versions of the 3020 RPM CentOS packages used in the last year was captured. The update times were captured and used in the data set to determine if different versions existed on the main repository at the same time. This information was used to estimate compromises for those package managers that require a consistent set of packages (packages that were all on a repository during the same time period).

Determining which package versions are vulnerable to attack proved to be more difficult as we are unaware of a data set that provides a good model of this. We randomly chose a set of vulnerable versions from all non-current versions of packages. In practice, we believe that an attacker would be more likely to work to discover vulnerabilities in old versions of popular packages as these would allow the attacker to compromise more clients. This work does not capture this effect.

7.3 Number of Compromised Clients

Using the CentOS trace and version information, the number of clients compromised by a malicious mirror was estimated (Figure 5). As the true number of vulnerable packages is not known, we used a conservative estimate of 25 vulnerable packages. These plots show that the security model of the package manager has a great impact on the number of clients that can be compromised. Every client that uses a mirror with a package manager that has missing or inadequate security (pacman, slaktool, ports, YUM, urpmi) is vulnerable to compromise. Using a package manager that allows an attacker to mismatch vulnerable package versions that were on the main repository at different times (Portage) reduces the number of compromised clients by about a factor of 4 to around 900 over the 13 day period. Package managers whose security mechanisms require a consistent set of packages (Stork, APT, APT-RPM, YaST) reduce the number of clients compromised to under 500. A package manager with

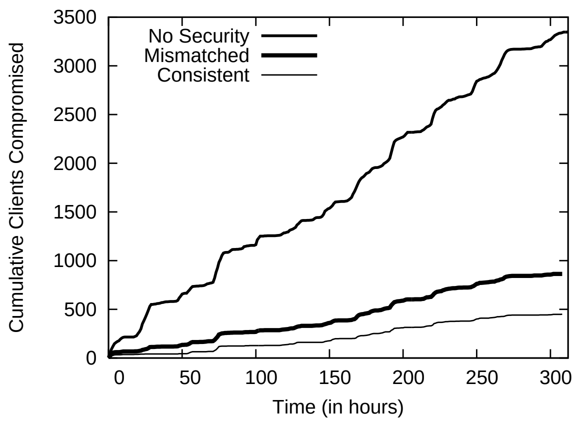

Figure 5: Compromised Clients CDF. This figure presents the cumulative number of clients compromised over a 13 day period for an attacker with 25 vulnerable versions. This figure shows the effect of the security mechanism on the number of clients compromised.

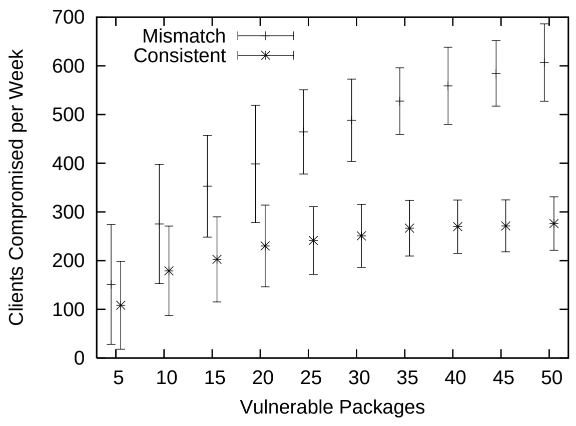

Figure 6: Clients Compromised Per Week. This figure presents the estimated number of clients compromised per week by an attacker that controls a mirror. The error bars show one standard deviation. The plots are offset slightly for readability.

signatures on the root metadata and protection against replay and freezing attacks (for example, modified Stork) will not have any compromises from an attacker that controls a mirror.

The package managers that allow the attacker to mismatch packages or that require a consistent set of packages vary based upon the number of vulnerable packages. The effect of varying the number of vulnerabilities is shown in Figure 6. The number of packages with vulnerabilities that the attacker can exploit is on the x-axis. The plots show that package managers which allow an attacker to choose different versions of packages that existed on the root repository at different times (mismatch) are more vulnerable to an attacker that has many vulnerable versions. Package managers that require an attacker to present a set of packages that was all on the repository at the same time (consistent) provide less motivation for an attacker to uncover more vulnerabilities. Equation 2 was used to decide which set of vulnerable versions the attacker should choose. This figure clearly shows that a package manager that requires an attacker to present a consistent set of packages provides better security than one that allows an attacker to mismatch packages. Somewhat disturbing is the significant number of clients that can be compromised if there are only 5 vulnerable package versions.

Our leased server was bandwidth-limited and mirrored multiple distributions simultaneously in order to control the cost of the experiment. An attacker would likely expend more bandwidth or set up multiple mirrors to capture additional traffic, thus leading to more compromises.

8. RELATED WORK

Many package managers have GUI front-ends [10, 22]. These GUI-based tools are usually just a different interface to the functionality provided by a command line package manager and so are identical from a security standpoint.

There are many techniques that help to support software security such as systems that ensure the authenticity and integrity of software (including SFS-RO [11], SUNDR [12], Deployme [15], and Self-Signed Executables [25]), and code signing certificates [5]. These are complimentary to the solutions presented in this paper.

There are several systems that access multiple mirrors to improve download performance and avoid DoS attacks. Byers et al. [4] describe using Tornado codes to improve performance by downloading from several mirrors simultaneously. This has the side-effect of allowing the client to make progress even if one of the mirror misbehaves. Sharma et al. [19] describe having the client "hop" between mirrors while downloading a file. This prevents an attacker from launching a DoS attack on the client because the attacker does not know which mirror the client will use next. We hope that raised awareness of the threat of malicious mirrors will result in organizations trying these techniques in practice.

9. CONCLUSION

This work identifies security issues in ten popular package managers in use today. These security issues allow an attacker who has control of a mirror to compromise clients that use the mirror. Our estimates show that an attacker with a mirror that costs $50 per week can compromise between 150 and 1500 clients each week. However, we describe how to address these security issues to reduce the effectiveness and in some cases eliminate the risk to clients.

This work also classifies the security models used in package managers and provides an ordering that demonstrates that root metadata signing provides the best security, followed by metadata signing, package signing, and no security. In addition to root metadata signing, we argue a package manager should support another signature mechanism to support the ability to verify stand-alone packages. This work demonstrates the overhead of using multiple signature methods is negligible in practice (2-5%).

10. REFERENCES

[1] Debian APT tool ported to RedHat Linux. `http://www.apt-get.org/`.

[2] APT-RPM. `http://apt-rpm.org/`.

[3] Arch Linux (Don't Panic) Installation Guide. `http://www.archlinux.org/static/docs/arch-install-guide.txt`.

[4] J. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: using Tornado codes tospeed up downloads. *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1, 1999.

[5] Introduction to Code Signing. `http://msdn2.microsoft.com/en-us/library/ms537361.aspx`.

[6] A. Crooks. The netbsd update system. In *ATEC '04: Proceedings of the USENIX Annual Technical Conference*, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.

[7] debsigs - What is debsigs. `http://linux.about.com/cs/linux101/g/debsigs.htm`.

[8] DistroWatch.com: Editorial: How Popular is a Distribution? `http://distrowatch.com/weekly.php?issue=20070827#feature`.

[9] man dpkg-sig. `http://pwet.fr/man/linux/commandes/dpkg_sig`.

[10] The KPackage Handbook. `http://docs.kde.org/development/en/kdeadmin/kpackage/`.

[11] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th SOSP*, pages 124–139, Kiawah Island Resort, SC, Dec 1999.

[12] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117, New York, NY, USA, 2002. ACM.

[13] milw0rm - exploits : vulnerabilities : videos : papers : shellcode. `http://www.milw0rm.com`.

[14] Netcraft: Strong growth for Debian. `http://news.netcraft.com/archives/2005/12/05/strong_growth_for_debian.html`.

[15] K. Oppenheim and P. McCormick. Deployme: Tellme's Package Management and Deployment System. In *Proc. 14th Systems Administration Conference (LISA '00)*, pages 187–196, New Orleans, LA, Dec 2000.

[16] Gentoo-Portage. `http://gentoo-portage.com/`.

[17] Installing Applications: Packages and Ports. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/ports.html`.

[18] RPM Package Manager. `http://www.rpm.org/`.

[19] P. Sharma, P. Shah, and S. Bhattacharya. Mirror hopping approach for selective denial of service prevention. *Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on*, pages 200–208, 2003.

[20] Slackware Package Management. `http://www.slacksite.com/slackware/packages.html`.

[21] Stork. `http://www.cs.arizona.edu/stork`.

[22] Synaptic Package Manager - Home. `http://www.nongnu.org/synaptic/`.

[23] URPMI. `http://www.urpmi.org/`.

[24] dkpg-sig support wanted? `http://nixforums.org/about101637-asc-15.html`.

[25] G. Wurster and P. van Oorschot. Self-Signed Executables: Restricting Replacement of Program Binaries by Malware. In *2nd USENIX Workshop on Hot Topics in Security*, Boston, MA, Aug 2007.

[26] YaST - openSuSE. `http://en.opensuse.org/YaST`.

[27] Yum: Yellow Dog Updater Modified. `http://linux.duke.edu/projects/yum/`.

APPENDIX

A. MIRROR DETAILS

Fedora's mirror system is the most automated of the five distributions for which mirrors were setup. Their system, called `MirrorManager`, allows mirror administrators to create an account where they can add and edit the details of their mirror. Mirror administrators add their mirror's information and run a reporting script on their mirror which submits to the `MirrorManager` website a list of files currently available on the mirror. After we had completed these steps, our mirror began receiving traffic from YUM clients within minutes. At no point was any direct communication with individuals from Fedora required. One interesting thing to note with Fedora's `MirrorManager` is that mirror administrators can specify an IP address range they want to serve packages to. In fact, targeting a netblock means that users in that netblock will use *only* that mirror. This allows easy targeting of attacks (to a specific country or organization) and reduces the number of other parties who will consume resources on the mirror.

For CentOS, new mirrors announce themselves by email through a public mailing list. This list is monitored by a person from CentOS who maintains the database of public mirrors. The only information we sent in our announcement email to this mailing list was the minimal contact, organizational, and mirror bandwidth information we'd seen others provide when announcing their mirrors. We received back later that day a thank you email from the mirror database maintainer letting us know our mirror had been added. Around the same time, we began receiving traffic from YUM clients who were being automatically directed to us by CentOS. To try to ensure the correctness of mirror content, CentOS uses an automated monitoring system that periodically makes requests to each mirror in order to ascertain whether the mirror is online and how frequently it has been updated.

Debian mirror information is submitted through an online form. After we had submitted the information for our mirror, we were contacted through email by someone from Debian who asked us for a few additional details about our mirror that hadn't been specifically requested in the online form, such as available bandwidth. We provided this additional information and within one day our mirror was included in the list of mirrors on Debian's website. This mirror received only a small amount of traffic during the time we had it online. We believe this is due to Debian not distributing client requests among all mirrors by default. There are utilities such as `netselect` available to Debian users for selecting mirrors by their speed relative to one's location, but we did not keep our mirror online long enough to attract many clients.

The information on Ubuntu's website indicates that new mirror approval is fast and automated, with approval taking up to 48 hours. However, three weeks passed after registering our mirror and the mirror had not been probed nor had there been any contact from Ubuntu. One week after filing a bug report, we were contacted through email by a person from Ubuntu who informed us that there had been internal delays in processing mirror verifications and that the issues had been resolved. At that time our mirror was finally probed and included on the Ubuntu website's list of official mirrors. As with Debian, client traffic was not automatically directed to our mirror since Ubuntu uses the `netselect` utility as well.

Registering a mirror for openSUSE involved emailing an openSUSE contact email address to announce our mirror was online and available for public usage. When we sent our new mirror announcement email, we received a response the same day saying our mirror had been added to the mirror database. We began receiving traffic to our mirror later that day.