Package Management Security

Justin Cappos, Justin Samuel, Scott Baker, John H. Hartman
University of Arizona
Computer Science Department
{justin, jsamuel, bakers, jhh}@cs.arizona.edu

Abstract

Package management is the task of determining which packages should be installed on a host and then downloading and installing those packages. This paper examines the popular package managers APT and YUM and presents nine feasible attacks on them. There are attacks that install malicious packages, deny users package updates, or cause the host to crash. This work identifies three rules of package management security: don't trust the repository, the trusted entity with the most information should be the one who signs, and don't install untrusted packages. The violation of these rules leads to the described vulnerabilities. Unfortunately, many of the flaws are architectural in nature, so repair requires more than patches to APT and YUM.

While the rules of package management security argue that the design of existing package managers is insufficient, they do not prescribe how to provide security. This led to the development of three design principles for building a secure package manager: selective trust delegation, customized repository views, and explicitly treating the repository as untrusted. These principles were used to construct a package manager Stork which is not vulnerable to the attacks identified for YUM and APT. Stork has been in use for four years and has managed over half a million clients.

1 Introduction

Package managers are a popular way to distribute software (bundled into archives called packages) for modern operating systems [4, 11, 37]. Package managers provide a privileged, central mechanism for the management of software on a computer system. As many packages are installed in the root context of the operating system, package management security is essential to the overall security of the computer system.

This paper evaluates the security of the popular package managers YUM [47] and APT [2]. This work describes a set of possible attacks and discusses their feasibility given the resources the attacker requires to be able to launch the attack.

One reason for the vulnerability to attack is that there are inherent problems with how cryptographic signatures are handled. Signatures in existing package formats require the package to be downloaded before it can be verified. As a result, package metadata (information about the package used to determine what is installed) is not verified using the package signature. Trusting package signatures is typically a very coarse-grained operation where a developer's key is either trusted or not, leading to escalation of privilege attacks. If a developer's key is compromised, then key revocation is not handled well by APT and YUM, leading to an inability to mitigate damaged caused by the compromised key.

In addition, there are implicit trust assumptions that APT and YUM make when interacting with the repository. Unfortunately, many of these assumptions allow an attacker to be trusted if they can intercept and alter traffic to and from the repository.

This leads to vulnerabilities that can be exploited to perform escalation of privilege attacks, prevent clients from getting security updates, and exhaust resources on clients (commonly causing mail delivery to

stop, databases to be corrupted, and logging to fail). Unfortunately, fixing many of these problems requires fundamental changes in the security architectures of APT and YUM.

This work identifies three rules that are violated by APT and YUM which result in security vulnerabilities.

Don't trust the repository. There are two components to this rule. First, verify that the communication with the repository correctly follows the protocol. Second, verify that the data returned from the repository is correct and current.

The trusted entity with the most information should be the one who signs. Packages (and metadata) should be signed by the entity with the most knowledge of the data. This specifically argues against a party signing something simply because another party has signed it — a popular practice in many distributions today [11]. This also implies that package metadata should be signed by developers instead of repositories (since the repository has no idea if the package metadata is correct).

Don't install untrusted packages. There must be a secure method for deciding what packages should be installed as well as verifying that the obtained packages are correctly signed. Selectivity is important for both of these — an authority should only be trusted for a particular set of packages. This rule also implies that users must get what they intend and should not be able to be presented with out of date or alternative packages they do not want. Packages signed by keys known to be compromised must not be installed.

The three rules of package management security describe what should not be allowed in order to retain security. However, they do not describe how the necessary security mechanisms should be provided. To that end, three design principles for security in package management were developed for the construction of Stork. These principles are selective trust delegation, customized repository views, and explicitly treating the repository as untrusted.

Using selective trust delegation, users can trust another user (such as a developer) to know the validity of a limited subset of packages. In addition, selective trust delegation can be used to prevent exposing project keys to individual developers. It also provides a natural mechanism for key revocation. Having customized repository views means that each user "sees" a different repository, which is actually an amalgamation of their trusted packages on all repositories they are using. Customized repository views prevent malicious repositories or user-uploaded packages from compromising the security of users. If the repository is treated as an untrusted entity, then even a root and private key compromise of a package repository does not compromise the security of users.

This paper presents the design of a package manager Stork that follows these design principles. Stork also obeys the rules of package management security and is not vulnerable to the attacks on YUM and APT. Stork has been in use for 4 years and has managed over half a million clients.

This work makes several contributions:

- Nine attacks on APT and YUM are identified. These reason the attacks are effective is traced back to violations of three rules of security in package management.
- Package managers that ignore these rules are vulnerable to the nine attacks described in Section 3. The feasibility (Section 3.1) and effects (Section 3.2) of attacks that exploit these vulnerabilities, the rules violated that lead to the flaws (Section 3.3), and the potential for repairing YUM and APT (Section 3.4) are described.
- While some of the attacks may be mitigated by straightforward repairs to APT and YUM, other attacks are not preventable because the security architectures of APT and YUM are fundamentally broken. To identify how to build a security architecture with the functionality needed to retain security, three principles for package management security are described (Section 4).

- These principles are followed in the design of a secure package manager Stork. The architecture of Stork is described in Section 5.1. A discussion which describes how Stork mitigates the effectiveness of the previously discussed vulnerabilities follows in Section 5.2.

2 Background

This section provides background information about package managers which is important in order to better understand potential vulnerabilities. Most package managers can be split into two basic components: a package installer and a dependency resolver. A package installer is a low-level component that uses special files (called packages) to manage the software installed on a node. A dependency resolver is a high-level component that handles communication with external servers that host packages (called package repositories) as well as dependency resolution.

Before individual package managers are looked at in detail, this section also provides some statistics on which package managers are popular. Popularity is important because the more popular software is, the more systems will be compromised if a vulnerability exists.

According to DistroWatch [13] and Netcraft [29], distributions that use the DEB and RPM file formats are overwhelmingly the most popular. As a result, their corresponding package installers DPKG and RPM are also popular. With regard to dependency resolvers, on the desktop APT/APT-RPM (45.2%) is clearly the most popular, with YUM (6.2%), Portage (6.9%), and YaST (10.7%) all having significant market share [13]. For servers, YUM (52%) is the clear leader with APT (25%) also having strong market share [29]. Since APT/DPKG and YUM/RPM are the most popular package managers, their security is the most important. As a result this paper focuses on APT/DPKG and YUM/RPM.

2.1 Package Formats

Packages consist of an archive containing files and, in the case of RPM and DEB, additional metadata. For a given package, the additional 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.

In the RPM format there is space for one signature. DEB packages have no standard field for signatures (and DEB files are not usually signed) but extensions exists to support signatures [12, 15].

2.2 Package Installers

The package installer is normally tied to a specific package format. The role of the package installer is to unpack the files from the package to the appropriate location, track installed packages, and remove or update packages. The package installer keeps a database that indicates which packages are installed. Most package installers also provide a signature verification mechanism to detect package tampering or corruption. There is typically only one package installer per computer system and it is normally standardized for the distribution.

RPM [34] is both the name for a package installer and the format of the package files that the package installer uses. The package installer is command line driven and is “capable of installing, uninstalling, verifying, querying, and updating computer software packages” [34]. While RPM is sometimes used standalone, it is common to run a dependency resolver like APT-RPM or YUM on top of RPM.

DEB packages are used by a package installer called DPKG [14]. DPKG performs similar actions as the RPM package installer, allowing the installation, uninstallation, and querying of computer software packages. There are subtle differences between DPKG and RPM [31], but they are interesting primarily from a functionality and ease of use standpoint rather than a security standpoint.

2.3 Dependency Resolvers

The dependency resolver gathers information about packages available on package repositories. Almost all dependency resolvers automatically download requested packages as well as any additional packages that are needed to correctly install the software (hence the name "dependency resolver"). 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 dependency resolver may be able to locate a package that provides `bar` on a repository. Note that this does not need to be a package with the name `bar`. A package may provide *virtual dependencies* of another name. This is useful if a package needs a program that provides some functionality but it doesn't matter what the program is (such as an email client or web browser).

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

The actual installation is done by the dependency resolver calling the package installer. It is common for multiple dependency resolvers [3, 39, 41, 47] to support a single lower-level package installer (RPM) and for different users of the same distribution to use different dependency resolvers.

One popular dependency resolver is APT. APT is a dependency resolver originally created to use the package installer DPKG in Debian. However, APT has also been ported to use other package installers, including RPM [3] (where it is named APT-RPM). APT automates the retrieval, configuration, and installation of software by resolving dependencies and handles communication with repositories. To address security concerns with APT (particularly with signatures), the secure-APT project [36] was created. Recent versions of APT use the changes added for secure-APT. This work focuses on the newer versions of APT with the secure-APT changes and ignores older versions since they are known to be insecure.

Another popular dependency resolver is YUM [47]. YUM is used to install and remove packages on systems with RPM package installers. YUM performs automatic dependency resolution and repository communication.

YUM and APT both have auto-update mechanisms available for them. When enabled, these auto-update mechanisms will upgrade older versions of installed packages with newer versions of the packages as they become available, typically within 1 day of the new packages being added.

2.4 Package Repository

Package repositories are usually just web servers used to provide packages and package metadata. The package metadata is the metadata in the package format which is extracted and stored separately. Often the metadata for all packages is put into a single tarball.

Package repositories store packages, package metadata, and the *root metadata* file. The root metadata file is called different things in APT (`Release`) and YUM (`repomd.xml`) but the contents are similar. The root metadata provides the location and secure hashes of the tarballs that contain the package metadata.

It is common for repositories to consist of multiple servers that host the same data. These additional servers are called mirrors and are used to offload traffic from the main repository. They typically contain the exact same content as the main repository and are updated via `rsync` or a similar tool.

2.5 Security Philosophy

APT and YUM use different techniques to provide security. APT focuses on securing the repository metadata rather than signing packages. An APT repository optionally provides a signature for the `Release` file

Attack Name	Description	Requirement	Result	Rule
Slow Retrieval	An attacker slows repository communication so that package managers will "hang" and will not error out or contact other repositories to retrieve package updates.	Repository	DoS	(1)
Endless Data	A malicious repository (or MITM) returns an endless stream of data in response to any file request.	Repository	DoS / Crash	(1)
Replay Old Metadata	An attacker provides old metadata that is correctly signed (perhaps to prevent new packages from being considered).	Repository	Outdated Package	(1)
Extraneous Dependency	An attacker changes the metadata for a package to indicate it depends on a package or packages of the attacker's choice.	Metadata Key	Any Signed Package	(2)
Depends on Everything	An attacker changes the metadata for a package to indicate it depends on everything.	Metadata Key	DoS / Crash	(2)
Unsatisfiable Dependencies	An attacker causes a package manager to ignore valid packages because forged metadata indicates unsatisfiable dependencies.	Metadata Key	DoS / Outdated Package	(2)
Provides Everything	An attacker changes the metadata for a package to indicate it provides any dependency the user requests.	Metadata Key	Any Signed Package	(2)
Use Revoked Keys	An attacker uses a revoked key to get users to install packages.	Revoked Key	Arbitrary Package	(3)
Escalation of Privilege	An attacker compromises a key trusted for signing a specific group of packages and then gets users to accept signed malicious versions of other packages.	Package Key	Arbitrary Package	(3)

Figure 1: This figure lists attacks, the requirements of the attacker, the result of a successful attack, and which security rule the developer violated. Revoked key attacks can work for either repository root metadata keys or package signing keys. The security rules are numbered (1) Don't trust the repository (2) The trusted entity with the most information should be the one who signs (3) Don't install untrusted packages. Package metadata implies the ability to change the package metadata the user receives when polling a repository.

(root metadata) in a file called `Release.gpg`. This allows APT to verify that the `Release` file is signed by the repository key and therefore came from the repository. The `Release` file contains the secure hashes of the package metadata on the repository. The package metadata contains secure hashes of the packages themselves.

Instead of signing the repository metadata, YUM uses signatures on packages to provide security. A YUM distribution maintainer signs all of the packages on the repository. YUM verifies package signatures after downloading packages.

In order to verify the identity of the repository, YUM uses HTTPS in Red Hat Enterprise Linux. Other distributions that use APT and YUM do not support HTTPS on their publicly available mirrors.

3 Vulnerabilities

This section introduces nine attacks that APT and YUM are vulnerable to. A summary of all of the attacks that will be discussed, the attacker's requirements to launch the attack, the result of the attack, and the rule for package management security that the package manager violated is listed in Figure 1.

There are four questions this section tries to answer:

1. Is it possible for attackers to obtain the necessary requirements to launch these attacks? (Section 3.1)
2. Do these attacks work on APT and YUM? (Section 3.2)
3. How could these attacks have been prevented? (Section 3.3)
4. What can be done to fix APT and YUM? (Section 3.4)

3.1 Avenues of Attack

This section examines the attacker's requirements to launch attacks on APT and YUM. There are three tiers of vulnerability that allow increasingly damaging attacks. The first tier is that the attacker must be able to impersonate a repository to launch a basic attack (Section 3.1.1). Once the attacker is able to impersonate a repository, the second tier is to be able to sign metadata, allowing more severe attacks (Section 3.1.2). If an attacker is able to impersonate a repository and sign metadata, then the third tier is to be able to sign packages, allowing the most severe attacks (Section 3.1.3).

Alternatively, instead of compromising the repository and signing keys, an attacker could simply compromise a developer key, allowing the attacker to launch attacks through the normal package update mechanisms in the repository (Section 3.1.4). The feasibility of obtaining the requirements to launch these attacks is now described.

3.1.1 Impersonate a Repository

The attacks against package managers require an attacker to have the ability to provide traffic on behalf of a repository. There are three ways this can be done: man-in-the-middle (MITM) attacks, control of a repository, or control of a mirror.

In MITM attacks, the attacker intercepts traffic between two communicating users and provides their own data instead. These attacks are relatively simple to launch [20, 46] and there are toolkits available that automate the process [16, 40]. When an insecure protocol such as HTTP is used, a MITM attacker is equivalent to an attacker who can control a repository through direct compromise. Using a secure protocol such as HTTPS can prevent MITM attackers from masquerading as a repository, however with the exception of Red Hat Enterprise Linux, HTTPS is not widely used.

A repository is typically very similar to a web server and attacks that compromise web servers, such as software exploits, may allow the attacker to modify content on the repository. Social engineering is another potential avenue of attack, allowing an attacker to gain control of a repository by manipulating the human administrators in charge of the repository.

Third-party developers often setup their own repositories to provide software that is not in the core of a distribution. An attacker could exploit this common practice by setting up their own repository purporting to provide third-party software and try to lure users to use it. However, this only effects the subset of users who use this software.

Another avenue of attack is for an attacker to obtain control of a mirror. To evaluate the feasibility of controlling mirrors of popular distributions, YUM and APT mirrors were set up for them. A fictitious company (Lockdown Hosting) with its own domain, website, and fictitious administrator (Jeremy Martin) were created to control the mirrors. A dedicated server was leased through The Planet (`www.theplanet.com`).

Setting up a public mirror for Debian, CentOS, and Fedora involved acquiring the packages and metadata and then notifying the distribution maintainers that the mirror is online. Debian and CentOS listed the mirror within a few hours, and Fedora listed the mirror in minutes.

With Fedora, a few minutes after the mirror was officially listed, the mirror began receiving requests from Fedora users. We believe this is the result of most Fedora users using `MirrorManager` [17] to dynamically select mirrors. `MirrorManager` has clients send requests to a central server that farms them out based upon the client IP, geographic location, country, etc. One interesting thing to note is that mirror administrators can specify an IP address range they want to serve packages to. In fact, targeting a subnet means that users in that subnet 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.

3.1.2 Metadata Key

While an attacker may be able to impersonate a repository, this may or may not mean that he is able to create forged packages or metadata on the repository. Some repositories use a private key (called a metadata key) to sign the root metadata that is published by the repository. If the repository keeps its metadata key offline or is a mirror of another repository which does all of the signing, a compromise of the repository may not imply a compromise of the key. A MITM who does not also possess the key cannot sign metadata.

In YUM and many APT distributions there is no metadata key to compromise. Any attacker with control of a repository can launch these additional attacks without having to compromise additional keys. If the repository signs metadata and has its key online, then an attacker who has compromised the repository can forge metadata. Similarly, if an attacker controls a third-party repository and lures users into adding the repository key to their keyring, the users become vulnerable.

One item of interest is that since the root metadata and the signature of the metadata are in separate files, they are not downloaded and verified atomically. This means that in some cases the signature and root metadata file will not match (for example, as they are updated) even in the absence of malicious users. These false positives may help to mask attacks.

Note that even when the root metadata is signed, an attacker who does not have the metadata key can still launch package metadata attacks if they can convince the repository to host a maliciously crafted package. This means that developers can trivially launch these attacks.

In cases where the root metadata is unsigned, one may argue that if the packages are signed, then the package metadata is protected from tampering. While YUM does not sign the root metadata, most YUM distributions do sign the packages themselves. One might assume that the package signature is used to verify the package metadata. This is not the case. The repository is trusted to generate accurate metadata that reflects the contents of the package. This metadata is used by the dependency resolver to decide which packages should be downloaded. Thus, the dependency resolver is explicitly trusting the repository to provide accurate information. The only way to be sure that the repository provided accurate metadata is to use the metadata from the downloaded and verified package. This means that, in practice, the package signature is not used to verify the package metadata! As a result, package metadata retrieved from a repository is (at best) only protected by the secure hash in the root metadata (signed using the metadata key in some APT distributions) and not the developer who packaged the software.

3.1.3 Package Key

In addition to keys used to sign the root repository metadata, some YUM distributions sign the packages themselves. If the key used to sign packages is compromised, then an attacker can sign arbitrary packages.

However, it is important to consider what happens when the dependency resolver decides a package that is unsigned or signed by an untrusted key needs to be installed to resolve dependencies. If a user willingly installs unsigned packages, then the result is equivalent to a private key compromise. With YUM, when a user tries to install a package that is signed with a key that is not in their keyring, YUM will return an error if the configuration option `gpgkey` is not set. If `gpgkey` is set, YUM will look for GPG keys at the URLs specified (often on the same repository) and ask the user if they want to install these keys in order to verify the package. So, if an attacker can provide their own key when the user requests a URL listed in `gpgkey`, all they need to be able to do is provide the user a package with a signature the user cannot verify. The user will be prompted to add the attacker's key to their keyring!

In APT distributions and YUM distributions that do not sign packages, there is no package key to compromise. In this case, repository and metadata key compromises imply the ability to launch any attacks that would otherwise require a package key compromise.

3.1.4 Developer Key

In addition to considering the feasibility of attacks on repositories, it is important to consider how packages are legitimately updated on repositories. If an attacker can introduce malicious content onto a repository through legitimate channels then they can also launch attacks.

To provide an example, the developer's package update process of Debian is described. To update packages for the popular APT distribution Debian, a developer uploads a package signed by their private key to the package archive. If the signature is valid, corresponds to a key in the list of developers, and the package is correctly formatted, the package file is put into a pool of packages that form the daily update. All of the packages in the daily update have new metadata generated for them. The root metadata file is signed automatically by a single Debian key that all Debian users have in their keyring. The daily update is then pushed to hundreds of mirrors around the world that serve as package repositories.

There are several items of interest with this process. First, the packaging and update process is fully automatic. From the description of security practices on the main Debian site [10], the only thing an attacker needs to do is compromise a developer's key to be able to upload malicious packages.

Second, any developer can upload an updated version of any package. This means that you are implicitly trusting *every* Debian developer's key even if you don't use the software they maintain. Considering there are more than 2448 keys listed in the Debian developer database, there are a significant number of keys that need to be secure for the distribution to remain secure. Further compounding the problem, there are keys that are as short as 768 bits and as old as 1993!

In addition to trusting developers who work on core distributions, it is important to consider third-party developers. If a user wants to verify third-party software before installation, then they need to add the third-party key to their keyring. Depending on the security measures undertaken by the third-party developers, this private key may be more or less secure than the user's distribution's private key. However, it always represents another avenue of attack.

As a recent attack [38] has shown, it is feasible for package repositories to be poisoned with malicious versions of packages. In this attack, an attacker obtained the password for a developer account. The attacker used the account to upload a modified version of the SquirrelMail package containing a flaw that allowed a remote user to execute arbitrary code. Interestingly, if the compromised account had access to the project website, the attacker would have been able to change the MD5 hash listed for the package and the attack may have gone unnoticed.

3.2 Attacks

Now that the feasibility of the attackers gaining the necessary access to launch these attacks is understood, it is important to understand if APT and YUM are vulnerable to attack.

3.2.1 Slow Retrieval

A simple attack is to allow the client to open a connection but then refuse to send data or send at a very slow rate. The attacker keeps the connection open for as long as possible. This prevents the client from installing updates from other repositories. APT and YUM don't log or print any useful information to help an administrator discover the attack has taken place. Other than "pausing" when it tries to contact a repository, there is no output to indicate the problem.

3.2.2 Endless Data

Another attack is to return an endless stream of data to the client whenever files are requested (an attack described in other work [42]). This attack has an odd effect on YUM and APT. Surprisingly, when YUM

is given a `repomd.xml` file of unlimited size, it exits silently after the filesystem is full with a 1 exit code — leaving the huge file on disk. Since no information is logged or printed about the error, this makes discovering the problem complicated (especially if YUM runs via auto-update).

APT is also vulnerable to this attack, but the size of the retrieved file is assumed to fit in a C unsigned long. This means that APT will willingly download up to 4 GB of data on 32 bit architectures for each file. If APT is compiled on a 64 bit architecture, then it will happily try to download files greater than 18,000,000 TB! Interestingly, APT could protect against this type of attack since the metadata in APT provides the data sizes of the files to be downloaded.

The endless data attack works to prevent clients from getting package updates from other repositories. However, this attack also consumes large amounts of disk space on the client system as well as network bandwidth and CPU. Exhausting resources, especially disk space, on a client machine can have disastrous effects. For example, on Fedora and Ubuntu this prevents logging, corrupts databases, and halts mail delivery.

The effectiveness of this attack can be mitigated by using a separate filesystem for the cache directory used by APT or YUM. However, this attack still prevents updates from other package repositories from being installed.

3.2.3 Replay Old Metadata

Some APT repositories sign the root metadata to prevent tampering by an attacker. The signature prevents an attacker who can not sign the root metadata from substituting arbitrary metadata. However, it does not prevent an attacker from replaying old metadata. An attacker may, for example, capture the root metadata from a date when a vulnerable package was released. At some time in the future, well after the package has been patched, the attacker may replay his captured metadata, causing clients who request the package to install the vulnerable version.

In APT, even though the metadata is signed to prevent tampering, there is no protection against replaying old metadata. APT ignores the date listed in the metadata file. It could be older than the previous file, or even have a date in the future, and there is no complaint. In fact, APT overwrites the existing metadata with the files it is downloading. This means that unless the user retains the old metadata file manually, they have no way to check what the previous state of the repository was.

For YUM and APT repositories that do not sign the root metadata, there is no need to replay metadata. An attacker can simply create metadata of their choosing.

3.2.4 Extraneous Dependencies

This attack is launched by providing false dependency information for a package the user will install to say that it also depends on another package. For example, an attacker can say that every package depends on some “extraneous” package of their choice.

This attack works on both APT and YUM. Surprisingly, neither APT nor YUM verify that the dependencies in the metadata of the packages they download match the package metadata they retrieve from the repository! The only restriction in the choice of extraneous package is that if package signatures are checked, then the package needs to be correctly signed. Even when signatures are used, this attack can result in new, vulnerable packages being installed.

3.2.5 Depends on Everything

Another potential attack involves returning package metadata that makes it look like a requested package has a huge number of package dependencies. In APT and YUM, all of these packages will be downloaded before any signatures are checked.

In addition to consuming disk and network resources on the client, this attack can be used by a malicious repository to launch an attack on other repositories. The malicious repository can advertise a new version of a package that depends on the entire set of core distribution packages and that the malicious repository hosts none of the core distribution packages. Assuming the client is configured to use multiple repositories, it will download all the distribution's packages from the other repositories.

3.2.6 Unsatisfiable Dependencies

To prevent installation of a package, an attacker can return a list of dependencies which indicate that a package the user is interested in has unresolvable dependencies. This prevents APT and YUM from installing the package, while to the user it appears that the repository or packager was in error, rather than it being a suspected attack. If an extension to YUM is used, YUM will attempt to install an alternate version of a package if the desired version is unavailable. This mechanism may allow an attacker to cause a particular desired (vulnerable) version of a package to be installed.

Note that the attacker need not be able to correctly sign the package of interest. Since packages are not downloaded until after dependency resolution, the client will not download the package (or check the signature).

3.2.7 Provides Everything

Another potential attack involves returning metadata that makes it look like a package resolves a huge number of virtual dependencies. Any time a package is needed to resolve a virtual dependency, this package will be considered. This package's metadata can be created to cause it to be installed over other packages that provide the same virtual dependency.

If there is a real package with the same name as a virtual dependency, the real package is always preferred. This allows an attacker to create a package `httpd` on their repository that will be preferred over any of the packages from the core distribution that provide this virtual dependency (such as `apache`). When there are multiple packages that provide the same virtual dependency, APT resolves dependencies on virtual dependencies by choosing the package whose name comes first alphabetically. YUM chooses the package whose name has the shortest length. Thus, an attacker can create a file named `a.deb` or `a.rpm` that will be preferred over all other packages.

This attack can be used by attackers in different ways. If package signatures are used, then an attacker can use this as another method to install extraneous packages. In some situations this is more effective than the extraneous dependencies attack because this will cause this package installation for dependencies of packages hosted on other repositories. If package signatures are not used or the attacker has compromised the package signing key, then an attacker can use this to have a malicious package of theirs be installed more frequently. Users who want to install other software will have the malicious package installed as well. This attack can also be used to launch depends-on-everything attacks if the package that provides everything has a huge dependency list.

3.2.8 Use Revoked Keys

One common need in any system that uses public key cryptography is a mechanism for revoking keys. Package managers are no exception. There are two issues to consider: the number of keys in the system and the method of revocation.

Unfortunately the popular APT and YUM distributions that do signing use few keys to sign everything. In Debian and Ubuntu, all of the root metadata files are signed by the same key. In Fedora every package is signed by the same key. There is no way to retroactively revoke trust in a signed item, without effectively revoking trust in all items signed by that key.

The typical mechanism for revoking a key is from a notification to the user that a key should be revoked in an ad hoc, out-of-band manner [8, 21, 23]. For example, such notification might take the form of a security announcement via email from an organization like CERT [8, 21]. The user then manually removes the key from their keyring.

Recall that the package management system is designed to update packages rapidly and often automatically. This is at odds with the comparatively slow process of manual key revocation. Key revocation creates a race between the revoker (who is trying to remove trust in the compromised key) and a malicious party (who may be trying to utilize the key before it is revoked). Since key revocation is a slow manual process while updating packages is rapid, this gives malicious parties a strong advantage.

3.2.9 Escalation of Privilege

In order to verify package signatures, YUM uses a set of public keys which are considered trusted keys. To add a new public key for package verification, users add the key to their keyring. However, checking if a package's signing key is in a user's keyring is a true/false question — it is either there or it is not. This means that a user who wants to verify Apache project packages with the Apache project's public key will also implicitly (and silently) trust a `gcc` RPM signed by their key.

YUM has a mechanism to try to mitigate these types of attacks. Users can specify that a repository should be trusted for a specific package or packages using the configuration option `includepkgs`. Similarly, a repository can be restricted from being used for specific packages with the `exclude` configuration option. However, this doesn't prevent third party developers associated with one repository from infecting users with malicious versions of core packages if they can put their malicious packages on other repositories the user trusts. This is because the keys in the user's keyring that are used to verify packages are not tied to a repository or a set of packages. This also requires developers to set up their own third party repositories if they want to provide a greater degree of trust to their users.

The existence of escalation of privilege attacks creates a problem for administrators and users. Should a user who knows the correct version of Apache's public key install the key in their keyring if they aren't planning on installing Apache soon? If they don't add the key, then they will need to validate the key is correct if they decide to install Apache later (an administrative pain). If the key is added, then a compromise of the Apache key can impact the security of the system even if `apache` packages aren't installed.

3.3 Lessons Learned

The previous section pointed out attacks and suggested solutions in a one-off manner. This section draws correlations between vulnerabilities. If there are commonalities between programming and design errors, then perhaps system developers can avoid these issues. The attacks described throughout this section (depicted in Figure 1) can be attributed to failures to follow three security rules.

The first rule of package management security is don't trust the repository. This means scrutinizing the data returned by the repository and verifying it obeys correct data transfer protocol rules. Failure to correctly check the returned data leads to vulnerabilities with the replay of old metadata. Insufficient protocol examination leads to vulnerabilities with endless data and slow retrievals.

The trusted entity with the most information should be the one who signs is the second rule of package management security. Packages (and metadata) should be signed by the entity with the most knowledge. This implies that package metadata should be signed by developers instead of repositories (since the repository has no idea if the package metadata is correct). This rule is violated because package metadata isn't verified using the package signer's key which leads to attacks involving forging package metadata to change the dependencies and provides. This also argues against the distribution key automatically signing developer uploaded packages.

The final rule of package management security is don't install untrusted packages. This means that a central point of focus should be a built-in revocation mechanism to reduce the vulnerability of users to key compromises. Another important consideration for preventing the installation of untrusted packages is preventing developers of one package from presenting malicious versions of another package (escalation of privilege). Unfortunately, neither APT nor YUM has a mechanism for key revocation built-in and the mechanisms to restrict trust can be circumvented.

3.4 Securing APT and YUM

There are several simple actions that will mitigate the effectiveness of many of the attacks:

1. *Validate repository communication.* By checking that file sizes and data rates are reasonable, APT and YUM could limit the effectiveness of endless data and slow retrieval attacks.
2. *Track signature times.* APT (and YUM if it adds metadata signing) should refuse to accept older versions of signed data. This would limit the effectiveness of the replay old metadata attack.
3. *Use HTTPS.* HTTPS makes it more difficult for an attacker to launch any of the attacks because a man in the middle will have a harder time masquerading as the repository.
4. *Guard mirrors.* Delegating control of a mirror for a distribution should be treated with utmost caution. This will help to prevent most of the attacks because it will be harder for an attacker to obtain the ability to impersonate the repository.
5. *Sign metadata and packages.* Signing both metadata and packages makes it more difficult for an attacker to launch most types of attacks.
6. *Check metadata is correct.* Once APT or YUM has decided to install a package, it should download the package and verify its signature and that the metadata matches the metadata provided by the repository. This will help to prevent the depends on everything attack and extraneous depends attacks when the attacker cannot correctly sign the package.

These measures will increase the difficulty in launching many types of attacks. However, within the architectures of APT and YUM there is no way to fix key revocation, escalation of privilege, provides everything, and unsatisfiable dependencies attacks. This means that the security architectures of APT and YUM are fundamentally inadequate to address these issues.

4 Principles of Secure Package Management

The rules of package management security describe how to avoid security errors in package management. However, they do not describe how to provide the necessary functionality to provide security. This section discusses three principles for secure package management that provide this functionality. Then, examples of how these principles should be applied in practice are given.

4.1 Design Principles

Selective trust delegation allows a user to trust another user's signature only for specific packages. This is a means by which a user can prevent escalation of privilege attacks by delegating the minimum amount of trust necessary. A hierarchical model is used where an end user may trust a project leader or distributor, who in turn trusts individual developers. This is fundamentally different from having the distributor sign a

package simply because the developer signed it. This is because the distributor never signs the package, but instead signals trust in the individual developer's key. This provides a natural mechanism for key revocation and also removes the necessity to revoke a key to remove trust in a package.

Having customized repository views means that each user "sees" a different repository, which is actually an amalgamation of package metadata across all of the repositories they are using. However, only package metadata that the user trusts is included. Package metadata from untrusted users cannot impact the security of the package manager. This allows repository administrators to permit users to freely add their own packages without compromising the security of other users. Repository administrators then only need to worry about traditional problems for servers, such as preventing attackers from gaining root on the repository and disk space usage of individual users, instead of having to be concerned with the validity of the contents of the packages or the package metadata.

Contrary to popular belief, there are subtle ways that package repositories are trusted by clients running modern package repository software. For example, package repositories are implicitly trusted to allow file downloads as well as providing accurate and timely metadata about the packages on the repository. Unfortunately, in many cases a man-in-the-middle has the same trust assumptions. By treating all interactions with the repository (and everyone in between) as though they are with an untrusted entity, package managers can avoid these attacks.

4.2 Concepts and Examples

Selective delegation means that users can choose to delegate trust to other users in very fine-grained ways. As a simple example, if Alice knows Bob maintains the `foo` package, Alice can state that they will trust `foo` packages that are signed by Bob. This means that Alice will not trust Bob's signature of a package `bar`.

An example of how this might be used in practice is that the `foo` project itself may have a key that is used to manage the project. Alice can trust the `foo` project to know which `foo` packages are valid. The `foo` project can trust the current developers on the project to know which `foo` packages are valid. The developers can sign `foo` packages with their personal private key.

At first glance this may seem the same as having a single project key for signing, but there are several important differences. First of all, the project key is only used when project membership changes and so can be kept off-line. Second, no developer needs access to the project's private key. As developers come and go there is no need to change the project key used to sign releases. Third, revocation of a developer's key is as easy as the project key removing delegation to that developer. End users need not even be aware that the project's membership has changed. Fourth, trust of individual packages can also be revoked without revoking the signing key.

Not every user needs to make their own trust delegation decisions. Distribution maintainers would likely selectively delegate trust to projects, who would further delegate trust to developers. They would request that all of their users either use their delegations or delegate trust to them for the packages that the distribution manages.

Another example of the usefulness of selective delegation is marking a package as untrusted. There are several groups that monitor software vulnerabilities [8, 21]. Once a vulnerability is uncovered, they notify users that affected packages are untrustworthy. Instead, they could ask users to trust them to know which packages to reject. Users can delegate trust to this group to know which packages are untrustworthy. As a result, the user's computer will refuse to install packages that are marked as having security vulnerabilities even if other users mark them as trusted.

A related principle to selective delegation is custom repository views. Custom repository views means that different users who use the same repository may see different packages. For example, perhaps Alice trusts Bob to know about the `foo` packages, while Charlie does not. Alice will have Bob's `foo` packages

in her view of the packages in the repository and Charlie will not.

Custom repository views extend to multiple repositories. A user may access packages and metadata on multiple repositories. This information is amalgamated to form a single, customized repository view with only the data that the user trusts. If some of the repositories are down or have been compromised, the user will still be able to install packages from valid sources.

In providing custom repository views, it is important to keep track of the latest version of the files that have been retrieved. The user should never accept an older version of a file it has seen.

It is important to treat data coming from the repository as untrusted. There are many subtle issues that are important to consider. For example, dependency resolution is performed using package metadata. This information needs to be validated so that maliciously-modified metadata cannot change the behavior of package dependency resolution. Also, clients retrieve data from repositories to install package updates including security updates. If a client is using multiple repositories, it should not be possible for one of the repositories to prevent the client from installing packages from others.

5 Stork

This section describes the security architecture of a package manager Stork which follows the principles for secure package management. A discussion of Stork's vulnerability to attack is presented.

Stork [6] is a package management system designed to fix shortcomings in APT and YUM. Stork has several non-security advantages over existing package management systems: it provides secure and efficient inter-VM package sharing on the same physical machine [28]; it provides centralized package management that allows users to determine which packages should be installed on their clients without configuring each client individually [35]; it allows multiple physical machines to download the same package efficiently; and it ensures that package updates are propagated to the VMs in a timely fashion. Stork is also unique from a development standpoint [7]. More information about Stork can be found in other sources [6, 7, 28, 35].

5.1 Basic Security Architecture in Stork

Stork mitigates the effectiveness of attacks on other package managers by following the three design principles for package management security: selective trust delegation, customized repository views, and explicitly treating the repository as untrusted.

This section begins with a description of a novel mechanism to provide selective trust delegation called a trusted packages file. Then follows a description of signature wrappers (a signature with additional fields) that provide resilience against replay attacks to support customized repository views. Next, Stork's repository communication (which gathers the trusted packages files and package metadata) is described. Finally, the use of self-certifying path names to support both customized repository views and validate repository communication is described.

5.1.1 Trusted Packages

The primary architectural security difference between Stork and existing package managers is the addition of a new type of file called a *trusted packages* file (or TP file). TP files are used to provide selective trust delegation and also aid in supporting customize repository views.

The user's trusted packages file indicates which packages the user considers valid. The TP file does not cause those packages to be installed, but instead indicates trust that the packages have valid contents and are candidates for installation. For the distributor of a package, it is common for them to have multiple versions of the same package listed in their TP file so that users can install older (perhaps more stable) versions of the package.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<TRUSTEDPACKAGES>

<!-- Refuse any packages the CERT user says are bad -->
<USER PATTERN="*" USERNAME="CERT" PUBLICKEY="MFwwDQYJK+sdlkjASFDGjsdfGJSDFGJAsd2qg34_45...sdfSJGs4pADFgsW34wEAAQ" ACTION="DENY"/>


<!-- Trust some packages that the user specifically allows -->
<FILE PATTERN="emacs-2.2-5.i386.rpm"   HASH="aed4959915ad09a2b02f384d140c4626b0eba732" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.01.i386.rpm" HASH="16b6d22332963d54e0a034c11376a2066005c470" ACTION="ALLOW"/>
<FILE PATTERN="foobar-1.0.i386.rpm"  HASH="3945fd48567738a28374c3b23847309634ee37fd" ACTION="ALLOW"/>
<FILE PATTERN="simple-1.0.tar.gz"    HASH="23434850ba2934c39485d293403e3293510fd341" ACTION="ALLOW"/>

<!-- Trust the apache user for apache packages -->
<USER PATTERN="apache*" PROVIDES="apache*,httpd" USERNAME="apache" PUBLICKEY="MFwwDQYJKtdd_...4bi1w4o6JMCAwEAAQ" ACTION="ALLOW"/>

<!-- Trust the 'stork' user for stork and arizona packages -->
<USER PATTERN="stork*,arizona*" PROVIDES="stork*,arizona*" USERNAME="stork" PUBLICKEY="MFwwDQYJKIh...3DMCAwEAAQ" ACTION="ALLOW"/>

</TRUSTEDPACKAGES>
```

Figure 2: **Example TP File.** This file specifies which packages and users are trusted. Only packages allowed by a TP file may be installed. FILE actions are used to trust individual packages. USER actions allow hierarchical trust by specifying a user whose TP file is included.

A trusted packages file allows a user to delegate trust and specify individual package files that they trust. In order to trust a package, the package name and the hash of the package's metadata are added to the trusted packages file. *This is not the same as adding the hash of the package to the trusted packages file as the metadata may now be verified independently of the package.* Since the package metadata contains a secure hash of the package, the package is still protected from tampering.

To delegate trust to a user, the user's name and public key are specified along with the packages and dependencies they are allowed to provide. This information is added to the trusted packages file. Users can be trusted to know which packages that meet these specifications to install (using `ALLOW`), which packages not to install (using `DENY`), or both (using `ANY`).

Lines in a trusted packages file are processed in the order they are seen in the file. The first rule that a package matches classifies the package as either available for installation (allowed) or removes it from consideration (denied). Any packages that are unmatched are automatically rejected (there is an implicit <`FILE PATTERN="*" ACTION="DENY"/`> at the end). Thus, the order of lines in a TP file is relevant when determining which packages are allowed to be candidates for installation.

Figure 2 shows a TP file without a signature wrapper. This TP file rejects any packages that the CERT user rejects. It specifically allows `emacs-2.2-5.i386.rpm`, several versions of `foobar`, and `customapp-1.0.tar.gz` to be installed if their metadata matches the secure hash listed. It also trusts the Apache user for packages named `apache*`. Additionally, the shown TP file indicates that the TP file of the Stork user will be used to determine trust of any package whose name starts with `stork` or `arizona`.

5.1.2 Signature Wrappers

Stork uses signature wrappers to support customized repository views. Signature wrappers protect TP files and the repository's root metadata from tampering, replay of old files, and a party permanently returning the same version of a file. Signature wrappers contain the timestamp, expiration time, signature, and a hash of the public key that was used to generate the signature. The timestamp prevents older versions of files from being considered over newer versions. The expiration time stops old files from being used indefinitely. The signature protects the file from tampering. The public key is included for reasons explained in Section 5.1.4.

The timestamp specifies when the file was created. By default, the timestamp is generated from the time in seconds since the epoch. Clients track the latest version of each file they have seen and will never accept an older version, thus preventing replay of old files.

Files can also be created with a negative timestamp. If a file has a negative timestamp, it means that the signing key should be treated as invalid. Any client with a file that has a negative timestamp will reject any file signed by the same key downloaded thereafter regardless of timestamp. This is an effective way for a party to indicate that a key has been compromised and that the key's signature should not be trusted.

The signature wrapper also has an expiration time. This is checked against the time on the local system where the file is used. If the current number of seconds since the epoch is greater than the expiration time, then the file is regarded as invalid.

The order of the timestamp and expiration time comparisons may be relevant. For example, suppose that there are two files: an older file that has an expiration time in the future and a newer file that has already expired. If the expiration comparison is done first, then the older file will be used because the newer file has expired. If the expiration comparison is done after the timestamp comparison, neither file is valid file because the newest version has an expiration time in the past.

Stork performs the timestamp comparison first so that the file with the newest timestamp will always be used. This helps to prevent replay of old metadata because users will never accept older files once they have seen a newer file, no matter the situation.

The expiration time mechanism requires that clients have roughly synchronized clocks. If this is a problem, a client may choose to check the signature and keep timestamp ordering but disable expiration checking. This allows users to trade convenience for security (as there is no protection against an attacker continuously returning the same version of a signed file).

The signature is used to protect the file from tampering. It covers the expiration time and timestamp as well as the embedded contents.

Stork uses signature wrappers to create the customized repository view that a client sees. This is the set of good package metadata and TP files from all repositories the user references, including good information previously downloaded from the repositories. Package metadata is checked for validity (described in Section 5.1.4) and newer versions of files are selected. The resulting valid package metadata forms the customized repository view.

5.1.3 Communicating with Repositories

Instead of a simply opening a repository connection for file download and waiting for download completion, Stork monitors the connection. If the connection is not transferring data above a user-configurable minimum rate, the repository is treated as dead and the connection is aborted. Stork also restricts the amount of data downloaded from a repository. If a repository responds with more than the expected amount of data or more than a user configurable maximum then Stork drops the connection.

Stork supports multiple data transfer protocols, including HTTPS.

5.1.4 Self-certifying path names

Along with signatures, Stork uses self-certifying path names [26] to detect tampering. These identifiers are present in the URLs or file names of TP files, packages, and package metadata. This allows an entity, such as a repository, to be able to check the validity of a file.

A package or package metadata is stored on a repository at a URL containing its secure hash. All packages and package metadata are considered to be immutable. Any user can verify that a file has not been tampered with by checking the secure hash of the retrieved data. This allows a repository to check that uploaded packages and package metadata is valid. This is also used by the client to validate that information retrieved from a repository is correct.

TP files have public keys embedded in their names instead[1]. Using an embedded public key, a repository can verify the file is unmodified by checking that the TP file is correctly signed and that the public key in the secure identifier corresponds to the private key that signed the file.

Since TP files may be changed by the user, there may be cases where multiple valid copies of a TP file are uploaded to a Stork repository. In this case, the Stork repository keeps the version with the latest timestamp. If any of the files specify that the key should be revoked (via a negative timestamp), then the file with the negative timestamp will be retained.

Having a Stork repository keep the newest version of a file is an optimization that prevents attackers from replaying old copies of files. However, a repository is not trusted or relied on to perform this action. Recall that even if a repository colludes with an attacker to accept old files, Stork's use of expiration dates limits the effectiveness of replaying old TP files to a client.

5.2 Mitigating Vulnerabilities in Stork

By following the principles for secure package management, Stork has tried to mitigate the vulnerabilities present in APT and YUM. This section revisits these vulnerabilities to examine their effectiveness on Stork.

5.2.1 Slow Retrieval

Communication monitoring by the client prevents slow communications from "pausing" the package manager. If a connection is too slow, the transfer is aborted and the repository is treated as down.

5.2.2 Endless Data

Stork protects against endless data attacks. In some cases a Stork client does not know the correct size of data it is downloading. Two such cases are when an attacker controls a repository (and can arbitrarily set the size) or when retrieving the initial root metadata. There are maximum file sizes for every type of file downloaded as well as a maximum size for a repository as a whole.

In the case of an uncompromised repository, all files other than the root metadata are of a known size. Stork verifies that the files it downloads do not exceed that size.

5.2.3 Replay Old Metadata

Since the root metadata and TP files have a signature wrapper, an attacker cannot replay older metadata than the client has seen. Also, the expiration time prevents files from being used indefinitely. In fact, repository metadata will only be used for the short time frame before it expires. Similarly, user TP files have an expiration time determined by the creator that prevents them from being used indefinitely.

5.2.4 Extraneous Dependencies

This attack cannot be launched by an attacker who compromises a repository since the developer's signature protects the package metadata. The key difference between the Stork client and existing package managers is that, with Stork, the *project or developer's key* protects the package metadata, not the *repository's key*.

[1]On some filesystems, keys with many bits are too long to be embedded in the file names. In this case the public key is placed in the signature wrapper and a secure hash of the key is placed in the URL

5.2.5 Depends on Everything

Similar to extraneous dependencies, this attack cannot be launched by an attacker who compromises a repository since the developer's signature protects the package metadata.

5.2.6 Unsatisfiable Dependencies

This reason why this attack is prevented is the same as extraneous dependencies. A repository compromise does not imply the ability to forge package metadata.

5.2.7 Provides Everything

This attack cannot be launched by an attacker who compromises a repository since the developer's signature protects the package metadata. The other consideration is whether or not a malicious developer can cause their package to be installed when not requested using this attack. In Stork a user can restrict the provides that will be allowed by package metadata signed by a specific user to prevent this attack.

5.2.8 Use Revoked Keys

Delegation using a trusted packages file provides a mechanism for trust revocation. Assuming that project keys are used only to change the list of developers trusted to sign project files, the most likely avenue for compromise is an individual developer key. These can be revoked individually by the project key without involving other users. If a project key is compromised, a trusted packages file with a negative timestamp can be created so that users will not continue to trust the compromised key.

Unlike APT and YUM, the speed of revocation in Stork is the same as the speed of trust. This means that individuals that revoke trust are not at a disadvantage because of the inherent speed of the mechanism used to do so.

5.2.9 Escalation of Privilege

Trusted packages files allow very fine-grained trust decisions to be delegated to other users. This means that if a user trusts Apache to know about the validity of `apache` packages but never installs an `apache` package, an Apache key compromise will not compromise their security. This removes the administrator's dilemma about whether to add keys they can verify when the package may not be needed (from Section 3.2.9). The simple and secure answer for Stork users is to trust all of the project keys that a user knows are valid and to trust them only for packages provided by that project. By using selective trust delegation, only systems that install a given project's packages are at risk if that project's key is compromised.

6 Related Work

There are a large number of package managers for Linux including Slaktool [37], pacman [4], YaST [45], urpmi [41], and Portage [33]. These package managers have flaws similar to those in APT and YUM.

A popular BSD package manager is `pkg_add` [5] with `freebsd-update` [32] also in use. `pkg_add` relies on users checking that detached signatures match the provided tarball. `freebsd-update` has improved security and signs root metadata but does not sign packages or use HTTPS making it similar to APT from a security standpoint.

Other systems make use shared filesystems for software deployment [1, 25, 18]. While Safari [18] does support package signatures, in general these systems do not address the larger problem of key distribution, revocation, or trust delegation.

There are a variety of ways to update and install software on modern operating systems besides package managers. There are software update systems [19, 24], systems that ensure the authenticity and integrity of software (including SFS-RO [26], SUNDR [27], Deployme [30], and Self-Signed Executables [44]), software installers [22, 43], and code signing certificates [9]. These systems do not support selective delegation or protect dependency information. They are meant for cases in which every user knows the organization who is supposed to be distributing each piece of new software and there are no software dependencies — unrealistic requirements for many scenarios.

7 Conclusion

This paper presents nine feasible attacks on APT and YUM. Unfortunately many of the flaws that make these attacks viable are architectural in nature so cannot be easily fixed. This work provides a security architecture that demonstrates how to avoid these vulnerabilities. A secure package manager Stork was built upon this security architecture. Stork is analyzed and shown to be robust to the attacks that are feasible on APT and YUM. Stork has been in use for four years and has serviced over half a million clients.

References

[1] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling File Servers via Cooperative Caching. In *Proc. 2nd NSDI*, Boston, MA, May 2005.

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

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

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

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

[6] Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John Hartman. Stork: Package Management for Distributed VM Environments. In *Proc. 21th Systems Administration Conference (LISA '07)*, Dallas, TX, 2007.

[7] Justin Cappos and John Hartman. Why It Is Hard to Build a Long Running Service on Planetlab. In *Proc. of the 2nd Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec 2005.

[8] CERT. `http://www.cert.org/`.

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

[10] Debian Developer's Reference. `http://www.debian.org/doc/packaging-manuals/developers-reference/`.

[11] Debian - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Debian`.

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

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

[14] Debian – dpkg. `http://packages.debian.org/stable/base/dpkg`.

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

[16] dsniff. `http://monkey.org/~dugsong/dsniff/`.

[17] Infrastructure/Mirroring – Fedora Project Wiki. `http://fedoraproject.org/wiki/Infrastructure/Mirroring`.

[18] Bill Fithen, Steve Kalinowski, Jeff Carpenter, and Jed Pickel. Infrastructure: A Prerequisite for Effective Security. In *Proc. 11th Systems Administration Conference (LISA '98)*, pages 11–26, Boston, MA, Dec 1998.

[19] Christos Gkantsidis, Thomas Karagiannis, and Milan VojnoviC. Planet scale software updates. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 423–434, New York, NY, USA, 2006. ACM.

[20] A. Godber and P. Dasgupta. Countering rogues in wireless networks. In *2003 International Conference on Parallel Processing Workshops*, Oct 2003.

[21] GovCertUK. `http://www.govcertuk.gov.uk/`.

[22] InstallShield – Installation Tool. `http://www.macrovision.com/products/installation/installshield.htm`.

[23] MMi Public Service Announcement – Malicious Installer Source Warning. `http://www.modmyifone.com/forums/showthread.php?t=24323`.

[24] Apple – Software Update. `http://www.apple.com/softwareupdate/`.

[25] Kenneth Manheimer, Barry A. Warsaw, Stephen N. Clark, and Walter Rowe. The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries. In *Proc. 11th Systems Administration Conference (LISA '90)*, pages 37–46, Colorado Springs, CO, Oct 1990.

[26] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proc. 17th SOSP*, pages 124–139, Kiawah Island Resort, SC, Dec 1999.

[27] David Mazières and Dennis 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.

[28] Steve Muir, Larry Peterson, Marc Fiuczynski, Justin Cappos, and John Hartman. Proper: Privileged Operations in a Virtualised System Environment. In *Proc. USENIX '05*, Anaheim, CA, Apr 2005.

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

[30] Kyle Oppenheim and Patrick 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.

[31] Re: Differences of Debian vs. the Other Guys. `http://lists.debian.org/debian-devel/1998/06/msg00128.html`.

[32] Colin Percival. An Automated Binary Security Update System for FreeBSD. In *BSDCon '03*, pages 29–34, San Mateo, CA, Sep 2003.

[33] '[gentoo-security] The state of ebuild signing in portage' - MARC. `http://marc.info/?l=gentoo-security&m=105073449619892&w=2`.

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

[35] Justin Samuel, Jeremy Plichta, and Justin Cappos. Centralized Package Management Using Stork. *To appear in ;login:*, Feb 2008.

[36] SecureApt - Debian Wiki. `http://wiki.debian.org/SecureApt`.

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

[38] SquirrelMail Repository Poisoned with Critical flaw. `http://www.beskerming.com/commentary/2007/12/19/313/SquirrelMail_Repository_Poisoned_with_Critical_flaw`.

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

[40] RSA Alert: New Universal Man-in-the-Middle Phishing Kit Discovered. `http://www.rsa.com/press_release.aspx?id=7667`.

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

[42] Wietse Venema. Murphy's law and computer security. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 19–19, Berkeley, CA, USA, 1996. USENIX Association.

[43] Software Packaging and Installation Authoring Tools – Altris, Inc. `http://www.wise.com/`.

[44] Glenn Wurster and P.C. 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.

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

[46] Lihua Yuan, K. Kant, P. Mohapatra, and Chen-Nee Chuah. DoX: A Peer-to-Peer Antidote for DNS Cache Poisoning Attacks. In *2006 IEEE International Conference on Communications*, Jun 2006.

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