Package Manager Security

Anish Athalye
aathalye@mit.edu

Rumen Hristov
rhristov@mit.edu

Tran Nguyen
viettran@mit.edu

Qui Nguyen
qui@mit.edu

Abstract

We analyze the security properties of package management software. First, we examine many package managers for basic security properties and we perform an in-depth security audit for several chosen package managers. Next, we construct and demonstrate an automated end-to-end attack against CPAN, a popular package manager for Perl. Finally, we make recommendations on how to build more secure package management programs.

1 Introduction

Millions of people rely on package management programs to install new packages on their machines, keep packages up-to-date, and manage library dependencies for other programs. Both developers and regular users invoke various package managers frequently, to install all kinds of software and libraries on their systems. Given that these package managers are responsible for managing the majority of the software and libraries running on a system, it's critical to the security of the system that these package managers and the protocols that they use are secure.

There are two basic types of package managers. Some manage system-wide programs and libraries for operating systems, like APT for Debian-based Linux distributions and Homebrew for Mac OS X. Others manage library dependencies in a development environment, such as pip for Python and CPAN for Perl.

Clearly, these package managers have a large security impact. These programs are sometimes run with ambient user privileges, and they are usually run with superuser privileges, so they are an attractive target for attackers. Libraries downloaded through package managers are often components of other software, amplifying the potential reach of malware. Therefore, it is critical that these package managers correctly install and update packages, against any attempted interference from attackers.

Our goal was to analyze potential weaknesses of package management programs in general, identify vulnerabilities, create automated tools to exploit these vulnerabilities, and recommend good security practices.

1.1 Threat Model

We analyze security properties of these package managers in the presence of man-in-the-middle attackers. In this model, an attacker may control the network that a user is on or control a router between the user and package repository. In this position, an attacker can sniff the user's traffic, send packets to the user, and manipulate traffic to and from the user. Given this threat model, we will evaluate the ability of attackers to compromise machines running various package management programs.

Organization The rest of this paper is organized as follows. In § 2, we discuss security properties of package managers and our methodology for analyzing tools. In § 3, we present security flaws in CPAN, the package manager for Perl, and in § 4, we take advantage of those vulnerabilities to construct an automated end-to-end attack against the package manager. In § 5, we discuss the scope and impact of our attack, and in § 6, we make recommendations on how to design, build, and ship more secure package management software. In § 7, we conclude the paper.

2 Analysis

We evaluated 28 different package management programs on a list of security criteria. The package managers we investigated included both system-level package managers such as APT, yum, and Homebrew, and development-level package managers like pip, cabal-install, and CPAN.

2.1 Criteria

We analyzed three security properties in the context of our threat model. These properties affect the security of the packages downloaded through these package managers, as well as the security of any downloaded metadata, such as information about package dependencies.

2.1.1 Communication Protocol

First, we considered the communication protocol that each package manager uses. Using a secure and authenticated protocol is necessary to ensure that downloaded packages are not tampered with. Most package managers use HTTP, or its secure version, HTTPS. For programs that do use HTTPS, we investigated whether they follow the protocol properly, by appropriately checking signatures and rejecting self-signed certificates.

2.1.2 Integrity Checking

We also evaluated how each package manager checked the integrity of downloads. A standard technique for integrity checking is to include a checksum with each downloaded file, and upon download, verifying that the checksum matches. We found some package managers, such as cabal-install for Haskell, do not support integrity checking. Others, like EasyInstall for Python, only verify checksums if provided by the developer. Some tools, such as CPAN, always require checksums.

2.1.3 Authenticity Checking

Integrity checking is not enough to guarantee that a download can be trusted. Even if a package matches its checksum, it could possibly contain malicious code. This could happen with a man-in-the-middle attacker who provides a checksum that matches a package that is infected with malware. Therefore, another important criterion is if and how a package manager ensures the authenticity of a package.

The most common way to check authenticity is with public-key cryptography. If it can be verified that a checksum is signed with the private key of the maintainers, then the file can be trusted. Some package managers we researched do not check signatures; others always check signatures; a few, such as CPAN, can optionally be configured to do so.

2.2 Methodology

To evaluate these package managers on the criteria listed above and identify specific attack vectors, we chose a subset of them. We examined documentation and performed detailed source code audits. We then explored potential attacks in virtual machines, using Wireshark to sniff the package manager traffic, and setting up our own mirrors of package repositories. Once we verified that an attack was possible from a malicious mirror, we could then move to trying a man-in-the-middle attack.

3 Vulnerabilities

After our initial analysis of security criteria, we chose to perform a more in-depth analysis of CPAN, the popular Perl package manager that has over 140,000 Perl modules.

In brief, the architecture of the package manager is as follows. There is a command line interface, `App::CPAN`, for the module that interfaces with package mirrors, `CPAN.pm`. Optionally, the tool uses `Module::Signature` to add cryptographic authentication checks to CPAN. All packages have `CHECKSUM` files that are signed by the CPAN maintainers, and public keys for the maintainers are added to the gpg keychain when the signature module is installed. This tool makes calls to the command line `gpg` application to actually perform verification. Signature checking can be enabled by setting the `check_sigs` parameter for CPAN. This is the setting recommended by the author of CPAN [1].

Through packet inspection and source code audits, we found vulnerabilities in the design of the signature checking scheme in this tool.

We performed an in-depth source code audit of `Module::Signature` [2], and we found several serious issues related to verifying signatures. If needed to verify a signature, the tool automatically fetches keys from a keyserver (by default, `pool.sks-keyservers.net`). Normally, this is not a concern, but because of the way the rest of the system works, this can lead to disastrous outcomes. The module uses the return value from the `gpg` command line tool to verify a signature, which makes it fairly difficult to check for trust rather than just validity, an unfortunate design decision on the part of the `gpg` designers. GPG only prints a warning to standard error if a signature is not trusted, so the module checks for trust using a regular expression match on the text output of `gpg` to `stderr`.

```
if ($output =~ /((?: +[\dA-F]{4}){10,})/) {
    # emit warning to stdout
}
```

Using a regular expression match is fragile and error-prone, and `Module::Signature` only outputs a warning to standard output if a signature is untrusted. The return value from the check still indicates success. This is unfortunate, as it makes it incredibly difficult for a tool using the library to check for trust. In addition, because automatic key download is enabled, a signature made

with an untrusted key will validate if the key has been uploaded to a public keyserver.

The code for `CPAN.pm` [3] performs a signature check by making a call to the signature module, but that module does not check for trust, only for validity (given in the return value). Even though `CPAN.pm` always checks that checksums are signed, the check is ineffective. For reference, the signature checking code from `CPAN.pm` is shown below in reduced form:

```
my $rv = eval { Module::Signature::_verify($chk_file) };
if ($rv == Module::Signature::SIGNATURE_OK()) {
    $CPAN::Frontend->myprint("Signature for $chk_file ok\n");
    return $self->{SIG_STATUS} = "OK";
} else {
    # abort, notify user about invalid signature
}
```

The vulnerabilities that we found are serious design-level vulnerabilities indicating a fundamental misunderstanding between *validity* and *trust*.

4 Attack

In our threat model, we assume the attacker can be a man-in-the-middle between a user and a package repository. Achieving a man-in-the-middle position was not the primary focus of our attack, but we still tried several different approaches.

If the user is on an unsecure network, then the attacker can send a malicious DNS response to poison the user's DNS cache and direct all the user's requests to the attacker. We attempted to do this, but unfortunately, the only hardware we had available were our personal computers, so our malicious DNS responses were much slower than responses from the real DNS server, so our attack did not have a 100% success rate. We hypothesize that the network cards on personal computers are not optimized for low latency. We believe that this method is possible with higher performance hardware.

An attacker can also achieve a man-in-the-middle position by controlling a router to which the user is connected. Because our primary goal was not a man-in-the-middle attack, we ultimately decided to simulate a man-in-the-middle position with a simpler approach. We set up two machines: one acting as the user and the other as a server. We rerouted all of the user's requests to the real CPAN mirror to our own server, simulating DNS poisoning.

Our malicious server acts as a malicious mirror of CPAN. It receives requests for packages and dynamically injects them with malware before returning them to the client.

When our server receives a request for a package, it first fetches the correct tar archive from the real CPAN mirror. Then, it decompresses the package and inserts malware as a test file. The CPAN tool automatically runs all tests on install, so this ensures our malware will be executed immediately on the victim's machine. CPAN also verifies a signed checksum, so we synthesize a new checksum for the modified package and sign it with our own PGP key. We uploaded this PGP key to public PGP keyservers so that CPAN would fetch it when installing our package.

Our malware executes with the privileges that `cpan install` is invoked with. Often, CPAN packages require root privileges during installation. This allows our malware to run arbitrary code on the user's machine with root privileges. For demonstration purposes, we wrote malware to erase the user's file system. If desired, we could have silently infected the users machine.

5 Evaluation

Even though we didn't setup an active man in the middle attack, we know that it is possible to do [4]. Considering our thread model, it is fairly easy to add malicious code to packages that the user is downloading. Most commonly, users do not look at the code that they download. This makes the attack very hard to detect.

Some package managers run the code that they download during installation. This was the case with CPAN, which allowed us to destroy the user's machine. Even if they don't run code or if they don't run it with root privileges, the attack is very effective. The source code that is installed might later be used for building applications, and the malware of the attacker will be also part of them. This creates huge security vulnerabilities, because the attacker not only infects a single machine, but also all the applications that are going to use it.

6 Recommendations

Based on our findings, we make several recommendations on designing, building, and distributing more secure package management software.

6.1 Communication Protocol

We recommend that all package managers use a secure, well-established communication protocol. One of the reasons why our attack works is that we are able to sniff and modify the traffic to and from the victim machine. Our particular construction of the attack can be mitigated if these package managers use secure protocol such as HTTPS with correct certificate checking. Many package managers including Maven, npm, and Homebrew have slowly migrated to HTTPS. Forging SSL certificates is not beyond the capabilities of state-level attackers, but it is still a great first-layer defense against most attackers.

6.2 Bootstrapping

We recommend that all distributors of package managers are careful with how they ship their package manager, protecting against attacks on the distribution itself during the initial download or bootstrapping phase. This can include protecting the download of the package manager itself and the download of any cryptographic keys. It is bad practice to download public keys over an insecure connection, because attackers could modify the keys to later subvert the package installation process.

A good approach to distributing a package manager is as follows. The package manager should come prepackaged with cryptographic keys, and the download itself should be provided over an HTTPS connection with a verified SSL certificate. This way, users can trust the download.

6.3 Validity versus Trustworthiness

We recommend that great attention is paid to both integrity and authenticity checking of packages.

We found serious issues indicating a fundamental confusion between validity versus trustworthiness. Many package managers use signature schemes to verify the authenticity of downloads, which is a great security practice. However, signatures that are only checked for *validity* as opposed to *trustworthiness* are only as good as checksums! Checking a signature just for validity is almost useless in terms of providing authenticity guarantees. This provides no protection against malicious attackers, only offering protection against accidental packet corruption.

6.4 Design of Interfaces

We recommend that interfaces of critical security-related components are designed carefully, and that a great deal of attention is given to the interfaces between components, being mindful of which component is responsible for what. We found several serious design-related vulnerabilities in the tools that we examined. Implementation vulnerabilities are easier to fix – they are usually small oversights, and a couple lines of code can patch a security bug. Design-related vulnerabilities, however, are incredibly difficult to fix while preserving compatibility with existing tools that rely on these insecure software packages.

The most serious issue that we found was with GnuPG. When verifying signatures, there are several situations that may need to be communicated with a client program (or user):

$$\{valid, invalid\} \times \{untrusted, trusted\}$$

The GnuPG command line program returns 0 (success) for the situation $(valid, *)$, and it returns 1 (failure) for the situation $(invalid, *)$. Deciding on a return value is complicated, because of having to somehow encode 4 possible results into a binary return value. GnuPG makes the wrong decision in terms of this encoding. On their own, either validity or trustworthiness do not mean much – either one can be forged by an attacker. A better interface would return success only for the case $(valid, trusted)$ and return failure for the other three cases. Right now, it seems that the only way to check a signature for both validity and trust is to check the return value from the `gpg` tool and examine what is printed to `stderr`. This is not ideal even when humans are interacting with the command line tool. It is incredibly fragile to have programs perform string parsing of output formatted for humans for security purposes, especially when attackers control part of the input.

This issue is exacerbated by the fact that the GPG keychain is shared between different applications that use it, where any application can add public keys to the keychain. Often times, automatic key download is enabled, making this issue even worse, because anyone can upload keys to public keyservers. Validity means almost nothing in terms of security guarantees, and because of the interface of the GnuPG command line tool, it is incredibly difficult for tools that depend on `gpg` to perform the proper security check in a robust manner.

6.5 Testing

Some protocols are provably secure, but this doesn't stop implementation bugs to expose security vulnerabilities. An important practice is to have good test coverage. This is true for all software in general, and it is especially important for security-related packages. Tests should check for correct behavior for situations where signatures and checksums are correct, and tests should also check for correct behavior for situations when signatures or checksums fail to validate.

For example, CPAN's `Module::Signature` has tests for their signature module, but it only contains test cases with messages signed by a trusted key. Given better test coverage, it might have been possible to catch the vulnerabilities that we exploited.

7 Conclusion

We analyzed security properties of package managers, and we uncovered several design-level vulnerabilities. To demonstrate the seriousness of these issues, we constructed an automated end-to-end attack tool against CPAN, the popular package manager for Perl. Finally,

we made recommendations on how developers could design, implement, and ship more secure package managers in the future.

References

[1] A. J. König, "CPAN::FirstTime." `http://search.cpan.org/~andk/CPAN-2.05/lib/CPAN/FirstTime.pm`, Apr. 2014.

[2] A. Tang, "module-signature." `https://github.com/audreyt/module-signature`, Feb. 2014. commit 02f7a91.

[3] "cpanpn." `https://github.com/andk/cpanpm`, Apr. 2014. commit 87d8281.

[4] M. Eriksson, "An example of a man-in-the-middle attack against server authenticated ssl-sessions," 2009.

[5] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," *Conference on Computer and Communications Security*, 2008.