Signed, sealed, and delivered

Verifiable

How to avoid malicious code on Linux. *By Kurt Seifried*

Although it doesn't happen often, occasionally, an attacker breaks in to a server and manages to modify source code or binary packages that people install on their machines. This happened recently, well actually not that recently, when an attacker broke in and replaced the file `Unreal3.2.8.1.tar.gz` with one that contains a back door in the `DEBUG3_DOLOG_SYSTEM` macro [1]. Unfortunately, that attack happened in November 2009, and it wasn't until June 2010 that the problem was detected, apparently because someone compared the MD5 sum of the file to the original and found they weren't the same. But why wasn't it detected earlier?

Classic Public/Private Key Cryptography

Using private/public key cryptography, you can easily create a private and public key pair and then use the private key to create signatures that other people can verify using the public key. This method is better than relying only on published hash values of the files, because cryptographic signatures easily can be checked and verified in an automated manner (which will result in much earlier detection of a problem) and because it only requires the user to have the public key (as opposed to having to hunt for an email announcement or web page). The good news is that a lot of people are now signing their software releases, binary packages, and so on; however, the bad news is that not everyone does this yet (`UnrealIRCd` does now that they have been bitten). So, assuming the software you want to use is signed, how do you verify the signature to ensure it's okay?

Verifying Signatures

GnuPG

Because the most common archive formats (TAR files that are then compressed with gzip or bzip2) do not directly support signatures, the most common way to sign them is to create an external signature that is then provided alongside the source code (most commonly with the same file name but with a `.asc` or `.sign` extension). Simply download the files [2] and run the following command:

```
gpg --verify ↩
  patch-2.6.8.1.gz.sign patch-2.6.8.1.gz
```

This command most likely will result in the error message

```
gpg: Can't check signature: ↩
   public key not found
```

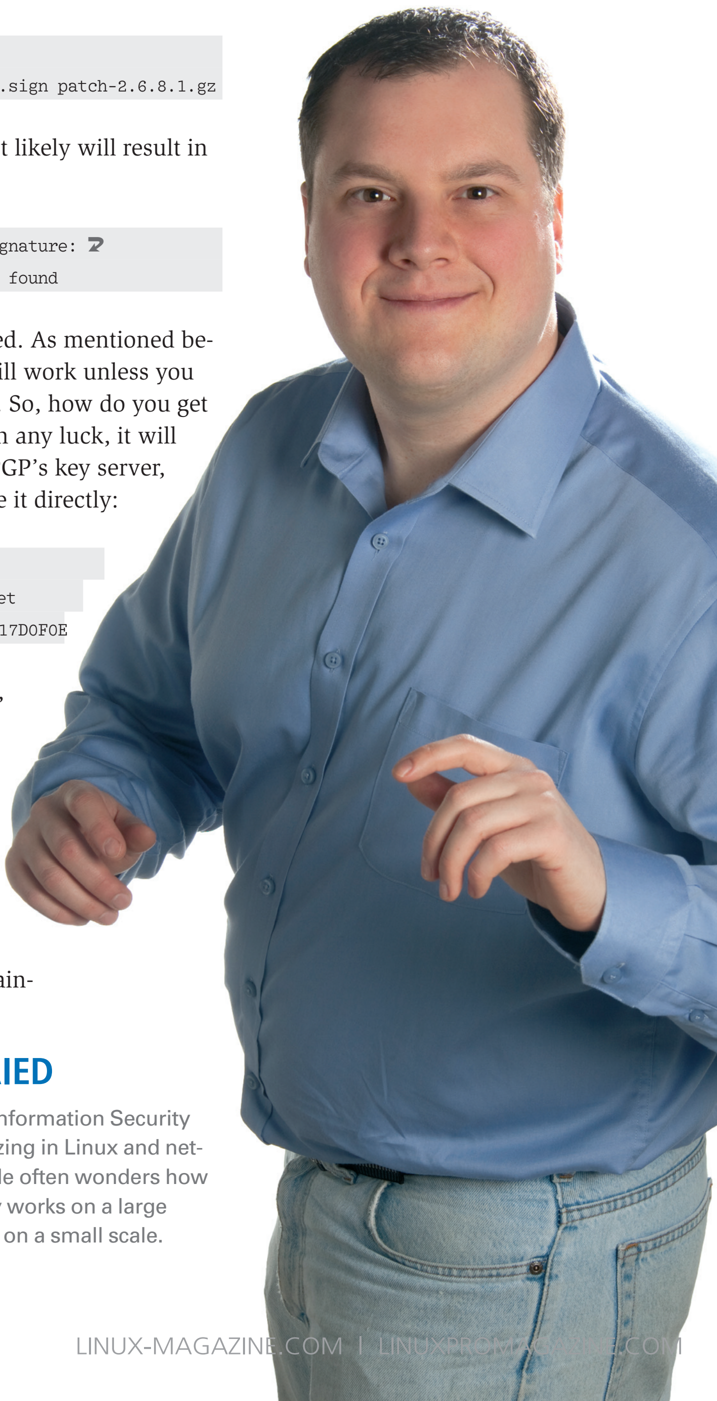
with the key ID listed. As mentioned before, none of this will work unless you have the public key. So, how do you get the public key? With any luck, it will be registered with PGP's key server, and you can retrieve it directly:

```
gpg --keyserver ↩
    wwwkeys.pgp.net
  --recv-keys 0x517D0F0E
```

If this doesn't work, your best bet is to use Google to look for search terms like "Begin PGP public key block" and the key ID. Hopefully, you will find the key listed on the project or maintainer's website (and in a perfect world, they will provide it on an SSL-encrypted website so you know you're not being given a fake site or key by a man-in-the-middle attack).

Once you have imported the key, you will most likely need to edit the trust level assigned to it. Unless the key you just imported happens to be signed by someone you trust, chances are you will need to assign a trust level manually to the new key. For example, to modify the trust level of the key used to sign Linux kernels, use:

```
gpg --edit-key 0x517D0F0E
Command> trust
... [output about trust levels]
Command>4
```



KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

In my case, I chose to assign a trust level of `4` (I trust fully) because the key is widely known, and I was able to get it from the PGP key server and verify it from the kernel.org website. However, even a key as popular and as widely used as this one (which signs all source code releases, patch files, etc. for the Linux kernel) comes back with only 1,310 Google results, so chances are that lesser known keys won't be easy to find. This makes it critical for people to get their keys signed by other trusted and well-known keys.

Once you have done all this, you will now be able to check and verify Linux kernel source code releases for as long as they sign them with this key (which has no expiry date, so it could be in use for long time).

RPM

Disclaimer: RPM is my preferred package format for a number of reasons, one of which is the simplicity of signing and verifying signatures. RPM uses GnuPG to handle the signing and verification of packages, which is smart, because rolling your own cryptographic systems is almost always a good way to make a mess.

If you simply run `rpm` with the `-K` option, it will tell you whether the signature is valid. To import a GnuPG key, run:

```
rpm --import somekey.gpg
```

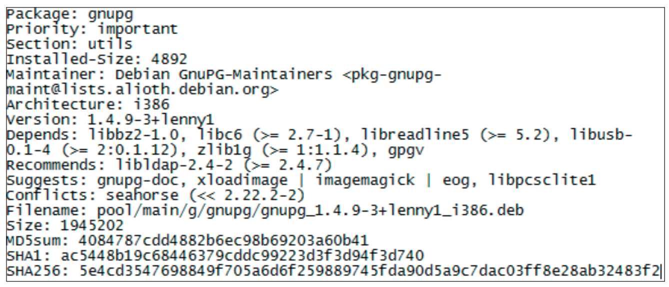
Typically, the first time you run `yum` you will be prompted to install the vendor key (which ships on the installation media), and all future package checks will happen automatically. Any failed packages will not be allowed to install, so this allows you to automate system updates safely and use shared resources (such as a publicly writable NFS server) to share the RPM update packages. If an attacker modifies a package, the signature check will fail and the package won't be installed (assuming, of course, you leave the default `gpgcheck=1` in `yum.conf`). Like a Ronco rotisserie, you "set it and forget it!"

dpkg

Now I come to a rather strange beast: dpkg. For a long time, dpkg has supported package signing with a set of external tools (`debsigs` and `debsig-verify`) and not within the default dpkg tools typically. But many dpkg-based Linux distributions (most notably Debian) do not use dpkg signatures for distributing files securely. What Debian does in fact do is require developers to sign their packages with their private key (i.e., *some-guy@debian.org*).

"The good news: a lot of people are now signing their software releases."

Once the package is uploaded to Debian's servers, the signature is checked; if it passes, it is stripped. Information about the package (e.g., the file size, file name, and MD5 and SHA1 signatures) is then written to a file called `Packages` and the MD5, SHA1, and SHA256 sum of the `Packages` file is written to the `Release` file. The `Release` file in turn has an external GnuPG signature placed in the file `Release.gpg`, and all these files are posted to the Debian servers, from which they are then mirrored and made available for download (Figure 1).



```
Package: gnupg
Priority: important
Section: utils
Installed-Size: 4892
Maintainer: Debian GnuPG-Maintainers <pkg-gnupg-
maint@lists.alioth.debian.org>
Architecture: i386
Version: 1.4.9-3+lenny1
Depends: libbz2-1.0, libc6 (>= 2.7-1), libreadline5 (>= 5.2), libusb-
0.1-4 (>= 2:0.1.12), zlib1g (>= 1:1.1.4), gpgv
Recommends: libldap-2.4-2 (>= 2.4.7)
Suggests: gnupg-doc, xloadimage | imagemagick | eog, libpcsclite1
Conflicts: seahorse (<< 2.22.2-2)
Filename: pool/main/g/gnupg/gnupg_1.4.9-3+lenny1_i386.deb
Size: 1945202
MD5sum: 4084787cdd4882b6ec98b69203a60b41
SHA1: ac5448b19c68446379cddc99223d3f3d94f3d740
SHA256: 5e4cd3547698849f705a6d6f259889745fda90d5a9c7dac03ff8e28ab32483f2
```

Figure 1: This shows the package info for the GnuPG dpkg.

Verifying Signatures with Apt

Here is where Apt comes in. The apt-get binary downloads the `Package`, `Release`, and `Release.gpg` files; checks the signature on the `Release` file (with the same procedure used to check a GnuPG signature); and, if correct, verifies the MD5, SHA1, and SHA256 values for the `Package` file, which in turn holds dpkg package information (e.g., file name, size, and MD5 and SHA1 sums).

If this information matches, the dpkg is fine, and it can be installed safely. If the information doesn't match, apt-get will not install the files (unless you force it to using the `--allow-unauthenticated` option, which you really, really shouldn't do) [3].

Unsigned Files

Back in the old days, I would have said you should simply download unsigned files or packages from a few different mirror sites, compare them, then do the installation if they matched. However, with attackers now breaking into head distribution sites (e.g., `UnrealIRCd`), even if you download the file securely from a "trusted" site over an encrypted channel such as HTTPS, you can't be certain that you are safe.

In such a case as this, I would recommend using Google to search for an announcement of the version (plug in the file name and phrases like "latest release" or "bugs fixed in this version") and hope that they include an MD5 or similar hash of the file (fortunately, as `UnrealIRCd` did).

Assuming you can find such a message from a reputable source (i.e., an announcement mailing list archived at a site like GMAME or MARC), you can be reasonably sure you have a legitimate copy of the software. However, I also urge you to email the author or project and ask that they sign their releases. The more projects that do this, the harder it is for attackers to modify packages and go unnoticed, which is good for everyone.

Oh, and if you want to sign dpkg files, check out `dpkg-sig`. ■■■

INFO

[1] UnrealIRCd: *http://forums.unrealircd.com/viewtopic.php?t=6566*

[2] GnuPG – making and verifying signatures: *http://www.gnupg.org/gph/en/manual/x135.html*

[3] Using the GPG signature checking with Apt 0.6: *http://www.debian-administration.org/articles/174*