Detecting source code modification attacks

Mods

Learn how to protect yourself against malicious attacks by modified source code. *By Kurt Seifried*

Normally, when I think about intrusion detection, my thoughts go straight to solutions for things like network- and host-based intrusion detection – in other words, the usual suspects (Snort, OSSEC, Prelude, event logging and analysis, etc.) [1] – but an often overlooked area of intrusion detection is source code modification attacks.

In the past few months, several high-profile source code modification attacks have taken place. Fortunately, two of the largest were quickly detected and dealt with, but only because pre-existing systems and processes were in place that could detect the attack and allow it to be handled.

Code Reviews

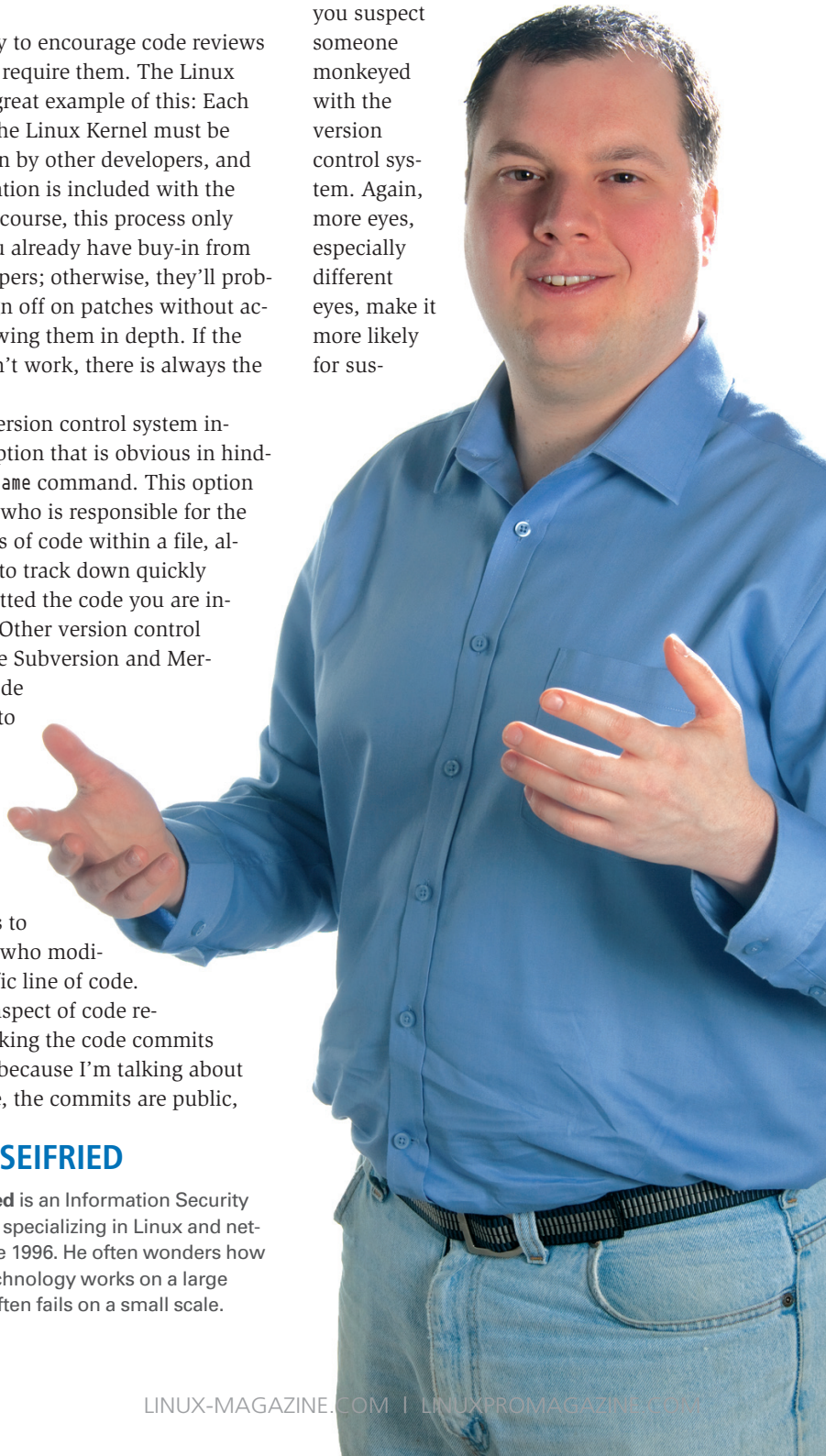
The Linux Kernel and open source projects in general excel at reviewing code. If attackers can insert malicious code into a popular project that people then install, they'll be able to waltz right in to a system. This happened several months ago with three WordPress plugins [2]: Add This, with 495,000+ downloads; W3 Total Cache, with 600,000+ downloads; and WPtouch, with 2,200,000+ downloads. Needless to say, if even a small percentage of users had installed the malicious versions of these plugins, the attacker would have had access to thousands, if not tens of thousands, of machines. Fortunately, the WordPress team noticed the code changes: All three back doors pass user-supplied content to a PHP `assert()` statement, which evaluates strings passed to it. Basically, this is a sneaky way to run a string variable as code.

So, what can you learn from this example? Code reviews work. The most obvious way to encourage code reviews is simply to require them. The Linux Kernel is a great example of this: Each commit to the Linux Kernel must be signed off on by other developers, and this information is included with the commit. Of course, this process only works if you already have buy-in from your developers; otherwise, they'll probably just sign off on patches without actually reviewing them in depth. If the carrot doesn't work, there is always the stick.

The Git version control system includes an option that is obvious in hindsight: the `blame` command. This option simply lists who is responsible for the various lines of code within a file, allowing you to track down quickly who committed the code you are interested in. Other version control systems, like Subversion and Mercurial, include commands to find out who committed to a file, but it's generally a two-step process to track down who modified a specific line of code.

Another aspect of code reviews is making the code commits public. But because I'm talking about open source, the commits are public, right? Well, yes and no. Unless someone mirrors the source code and checks the commit logs, they won't have any idea what is going on. On the other hand, if you also email your commits to a mailing list (e.g., OpenBSD sends all CVS commits to *source-changes@*), it is much easier for a developer or interested party (e.g., someone who repackages your source code) to keep an eye on things.

The other benefit is that most mailing lists are archived, so you now have additional sources to check in case you suspect someone monkeyed with the version control system. Again, more eyes, especially different eyes, make it more likely for sus-



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.

picious commits to be noticed and flagged as such.

Having a history of commits should also help make strange or bad commits more obvious. For example, the back doors put in the WordPress plugins all used the `assert()` PHP function, which is rarely used (it's a debugging function primarily). Also, hostile code probably won't conform to your coding standards (you do have coding standards, right?). It also may be obfuscated, which should be an obvious clue. (I hope you don't use obfuscated code.)

Source Code Signing and Distribution

Of course, having good, clean source code does no good if the distribution is done insecurely. Most open source projects simply use `tar` to package the source code files into a single file and then compress them with `gzip` or `bzip2`. Many will also include text files with SHA256, MD5, or both sums of the files, which makes it possible to verify that downloads are not corrupted, but this doesn't really prevent an attacker from replacing the files and simply updating the signatures file, so the obvious solution is to use good encryption software, like GnuPG, to sign the files.

The good news here is that many open source projects are signing their source code tarballs with PGP or GnuPG. A perfect example of this is the recent attack against vsftpd, where an attacker managed to replace the source code [3]. Fortunately, the attacker was not able to sign the code, and this omission was noticed (Listing 1).

But, what happens when you don't have the public key needed to verify the signature? The bad news is, for many projects, finding the signing key is often not that easy. When you check a signature, even if you don't have a copy of the public key needed to verify the signature, you will be able to find the Key ID of the key used to sign the data.

If you're lucky, the key details, such as the fingerprint, will be on a web page for the project. The Key ID is an eight-character hexadecimal number, so it only encompasses 32 bits (about 4 billion possibilities). Thus, in theory, an attacker can create his or her own key with the same Key ID as a legitimate key. The key fingerprint, on the other hand, is 32 characters long, or 128 bits, so the chances of creating a key with the same fingerprint (and Key ID) as a legitimate key are infinitesimal.

LISTING 1: GnuPG output of unsigned vsftpd tarball.

```
01 $ gpg ./vsftpd-2.3.4.tar.gz.asc
02 gpg: Signature made Tue 15 Feb 2011 02:38:11 PM PST using DSA key ID 3C0E751C
03 gpg: BAD signature from "Chris Evans <chris@scary.beasts.org>"
```

With the Key ID, you can search the Public Key Servers and probably find the key in question. However, once you have found the PGP key, you need to verify that it is legitimate. Again, if you're lucky, the key will be signed.

The key used to sign the Linux Kernel releases, for example, has more than 100 signatures; however, many, if not most, keys used to sign code are only self-signed (meaning that no third party has signed the key). If this is the case, your best bet is to use Google. Ideally, the key will be widely referenced, possibly going back for months or years, which will allow you to confirm that the key is legitimate.

The next step is for a vendor like Red Hat or Debian to take the source code and package it up. For more information on checking GPG and RPM/APT signatures look up my "Checking Signatures" article from September 2010 [4].

Key Management

Of course, properly signed source code and packages only work if the private PGP keys used to sign the data are protected properly. At minimum, the key needs to be protected by a good passphrase and placed on a secure system, If attackers can compromise the system and install a key logger, for example, the passphrase won't help much because they can just record it.

With this in mind, my advice is to keep your key offline. A simple way to accomplish this is to have a USB key/CD with the key on it. However, a better solution is to have a dedicated system that is not attached to the Internet (meaning it's very difficult to attack). A less expensive way to accomplish this is to use a bootable CD or USB key to fire up a system that you only use to sign source code [5].

The second aspect of key management is making it easy for users to verify your key. The two main ways to accomplish this are by key signing and by publishing your key widely. By getting a trusted third party to sign your key, chances are a user will have a key they already trust that they can use in turn to verify your key.

To publish your key widely, upload it to key servers and publish the key, the Key ID, and the fingerprint on your website. This approach works especially well if you have your own domain for the open source project and an SSL certificate (making it easier to verify the content served from the website).

Conclusion

Securing your network and systems begins with running software that has not been compromised by an attacker. Fortunately, on Linux, this process is pretty easy. You start with secure source code, which is then usually packaged by vendors, who in turn distribute it securely. But, users at all levels need to verify signatures for this process to work. ■■■

INFO

[1] "Security Lessons: Windows Logging" by Kurt Seifried, *Linux Magazine*, August 2011, pg. 72

[2] "Add This, W3 Total Cache, WPtouch backdoors" by Adam Harley, *http://adamharley.co.uk/2011/06/wordpress-plugin-backdoors/*

[3] "Alert: vsftpd download backdoored" by Chris Evans/Scarybeasts, *http://scarybeastsecurity.blogspot.com/2011/07/alert-vsftpd-download-backdoored.html*

[4] "Security Lessons: Checking Signatures" by Kurt Seifried, *Linux Magazine*, September 2010, pg. 44, *http://www.linux-magazine.com/Issues/2010/118/VERIFIABLE*

[5] "Security Lessons: Disposable Computers" by Kurt Seifried, *Linux Magazine*, November 2010, pg. 42