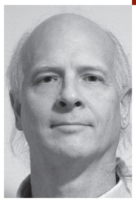
RIK FARROW

musings



Rik Farrow provides UNIX and Internet security consulting and training. He is the author of *UNIX System Security* and *System Administrator's Guide to System V*, and editor of the SAGE Short Topics in System Administration series.

rik@spirit.com

Corruption. The very thought sends shivers up and down my spine. And that is the goal of those who would break into your systems, so they can "own" them. They want to take control of your systems, preferably in a manner that is difficult to detect. Out of this desire came rootkits: corruption made simple.

I got my first rootkit from a friend at a university, my source for lots of examples of stuff left behind on compromised systems (nice, delicate term for being hacked). That rootkit was one of the first written, and contained trojans for SunOS 4. In the README file, the author of the rootkit had written (approximately), "I got tired of doing the same things over and over again, so I packaged them up." The rootkit contained trojans designed to hide the presence of certain files, processes, network connections, and a network sniffer. If you remember what networks and network protocols were like in 1993, you'll understand why this sniffer worked very well at collecting usernames and passwords.

Over the years, people added features to rootkits, such as the ability to edit logfiles or, better yet, prevent certain log entries from being appended to logfiles by trojaning the syslog daemon. New commands were added to the list of trojans. But the worst was yet to come.

The problem with command-level trojans is that it is relatively easy to detect them. Tools like Tripwire were written specifically with this in mind, as installation of trojans and other malware became commonplace. Most trojans rely on access to source code, and that leads to trojans for closed source systems being based upon open source software. If someone used the BSD source to ls, for example, the flags and behavior would not be the same as they would be for AIX or HP/UX. Close, but not exact. And systems like Solaris don't have just one version of ls, but several.

Going Deep

The solution, from the perspective of an attacker, was to move the rootkit deeper. If the rootkit runs at the kernel level, then nothing can be trusted. All software, whether on UNIX, Windows, Linux, or *BSD, relies on the kernel for all access to resources such as files, sockets, memory, and new processes. The system call interface provides this access. In UNIX-like systems, the system call interface provides a couple of hundred entry points for doing things like listing directories, files, programs, sockets, and active processes (189 in

OpenBSD 3.4, 315 in Linux 2.6). In Windows NT and its descendants, the number of entry points is more than 2000, but the concept is the same. In either case, if the attacker can insert code into the kernel, that attacker has the deepest level of control over a system.

The obvious way to insert code is to modify the kernel source directly. But there is a problem with that approach, in that a system must be rebooted before the changes take effect, and rebooting a UNIX-like system is rare enough that it would be noticed (in most cases). But there is also an obvious solution—use a method that permits patching the operating system without rebooting.

You have certainly heard of loadable kernel module (LKM) rootkits. LKMs permit sysadmins to install software in an operating system without rebooting it, or to configure a kernel at boot time without having all possible devices already linked into the kernel. While LKMs are convenient for sysadmins, they are just as convenient for any attacker who has acquired root access and wants to install the best in rootkit technology.

And Deeper

Over time, even LKM rootkit technology has improved. Early versions worked by replacing function addresses in the system call table with their own entry points. The original system call function still gets called, but the results of the system call get filtered to hide whatever the rootkit designer wants to hide. Initially, this was pretty much the same stuff that was done in the original, SunOS, command-level rootkit. But then it started to change.

One creative use of kernel-level rootkitting was file redirection. If you ran an integrity-checking tool like Tripwire (or anything that read a file), you would get the original version of the file. But if that file contains a program, when a request was made to execute it, a different program got run instead.

LKM rootkits can perform privilege elevation. In many of the rootkits around today (e.g., adore, adore-ng, all-root, kbdv3, rkit, shtroj2, and synapsys), the rootkit installer can either get a root shell or run a program as root by using whatever key the rootkit requires. In adore-ng, echoing the adore key to `/proc` elevates the privilege and capabilities of the shell to root without restrictions. This beats the pants off the old, SunOS rootkit technique of using back doors in SUID files like chsh and passwd. Adore-ng also prevents log records of hidden processes from being written.

Even the methods used to hide things have changed. Adore-ng, instead of hooking system calls, actually hooks into the Virtual File System (VFS) interface to perform its deeds. This works because both files and processes get listed via the VFS in Linux and some other operating systems (adore-ng works only on Linux). You can read Phrack (http://www.phrack.org/phrack/58/p58-0x06) if you want to learn how this is done.

Adore-ng also offers a new technique for hiding its own presence. The adore-ng.o file can be linked with an existing kernel module, so that when that module gets loaded at boot time, so will adore-ng. This makes adore-ng much more difficult to detect, and quite neatly solves the problem for the attacker of how to reload it after the next reboot. For details, you can check out Phrack again (http://www.phrack.org/phrack/61/p61-0x0a_Infecting_Loadable_Kernel_Modules.txt). It turns out neither to be difficult nor difficult to understand, and relies on a documented feature of ld plus a little symbol name manipulation.

By moving the hooks into a deeper level of the file system, tools that monitor the system call table for changes will miss the installation of rootkits like adore-ng. I did uncover a paper by Kruegel, Robertson, and Vigna (http://www.cs.ucsb.edu/

~vigna/pub/2004_kruegel_robertson_vigna_ACSAC04.pdf) that performs binary analysis of LKMs and detects rootkits by checking for the memory they seek to modify. Most LKMs stick to the regions of memory that a device driver would need to modify in order for initialization to succeed, but not rootkits, which stray to regions only miscreants would go. Certainly an interesting approach.

Another "interesting approach" comes in the form of SUCKIT, a kernel-level rootkit that does not rely on using LKM hooks. This charmingly named rootkit does its work by reading and writing directly to /dev/kmem. Unlike the LKM approach, which relies on being able to locate the kernel symbol tables, this rootkit searches through kernel memory looking for the pattern of bytes typically found within the soft interrupt handler, the entry point to the kernel and the system call table. The soft interrupt handler address can be gleaned from a single Intel assembler instruction, sidt %0, and then the code searches for the offset to the actual call to the system call table. You can read about this in Phrack too: http://www.phrack.org/show.php?p=58&a=7.

So, even if you compile a kernel without LKM support, someone can still patch your kernel. As I read the Phrack article about this technique, I shuddered again. While getting your system rootkitted is bad, SUCKIT (like LKM rootkits) might just abort your kernel if it doesn't work perfectly.

The authors of SUCKIT suggest modifying your kernel so that writes to /dev/kmem are prohibited, even to root. This will stop this rootkit, without stopping you from tuning your kernel using the /proc interface. They even suggest a one-line patch to mem.c that will do this. Some solution.

But what about stopping LKM rootkits? I mentioned earlier that there were three ways of rootkitting kernels. The third way I was alluding to works with Windows and involves installing a device driver (for information, see http://www.rootkit.com). Microsoft certainly deserves a lot of the bad marks it gets for security, but you may have noticed that Microsoft not only supports but encourages the use of signed device drivers. If a device driver has been signed, you know it has not been modified to include a rootkit and (relying on the signer of the device driver) is not a rootkit. I will confess to being less than current as a Windows sysadmin, but there was a time when someone who could administer printers could also install device drivers. And I do know that the default on XP is to make the first (and often the only) user a member of the Administrator group.

I wondered if LKM signing had been accomplished in the Linux world and found a "discussion" (a polite term for it) on an archive of the Linux-kernel mailing list. It seems that most of those involved are not interested in adding more bloat to the Linux kernel (I certainly understand that concern) by adding support for checking the signatures of LKMs before loading them. David Howell even posted patches that support checking GPG signatures of kernel modules (http://people.redhat.com/~dhowells/modsign/), but his solution appeared overwhelmed by opposition. Perhaps RedHat will decide to do this on their own for their commercial Linux version.

Proper use of LKM signing implies that any time you build kernel modules, you copy them to another system, sign them, and copy them back to the system where they will be used. As long as the signing system cannot be compromised, the signature checking mechanism will guarantee that only signed, unmodified modules get loaded into your kernel. RedHat could certainly offer signed LKMs with their distros, and those that build their own kernels could include the mechanism and the public key, in the kernels they build. Combined with disabling writing to /dev/kmem, LKM signing would appear to block an entire class of popular attacks. And it might even provide a use for the TCPA chip, in that it could hold the public key and be involved in signature checking.

I do want to add a note that FreeBSD kernels after 4.0 have the securelevel flag, which, when set to one or two, prevents kernel modules from being loaded. A positive securelevel also blocks writing to kernel memory (goodbye SUCKIT). Evil kernel modules could still be placed in a directory where they would be automatically loaded during the next reboot.

The history of computer (in)security has been one of attacks, defenses, and new attacks designed to counter those defenses. Signing LKMs could be just another failed defense. But some form of kernel defense does appear to be justified.

Anything beats corruption.