musings

by Rik Farrow

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.*

rik@spirit.com

The more I learn about security, the more paranoid I get. This won't surprise too many of you, because you very likely feel the same. When you know that kernel-level rootkits can hide anything the attacker desires from you (short of rebooting your system from a CD and running a thorough check), then I believe you have reason to be paranoid.

And are attackers out to get you? Really, in most cases, it's nothing personal. You just happened to be running some vulnerable bit of code, and the latest automated attack rooted your system, installed itself, installed the rootkit, and modified your system so that the tools will restart after any reboots. In a way, you are lucky if your system does start misbehaving right away, scanning Class B-sized blocks of IP addresses, as such activity should make you notice that something bad has happened. And if you have configured your firewall to block unusual outgoing traffic, no one else will even notice before you have time to boot from a CD and see what has happened to your system.

It would be nice if things like this just didn't happen. But even if you secured your system, installed the latest releases or patch levels, your system may still be exploited. There are zero-day exploits, ones that have received no publicity and for which no patches yet exist. Software is complex, and complexity guarantees there will be mistakes in coding or in implementation. Even the famed Wietze Venema expects that his Postfix code has at least one bug per thousand lines of code. Thirty bugs in Postfix, and we can expect that as Postfix continues to improve, the odds of more bugs being discovered will increase.

Paranoia

I could try the minimalist approach. Run a pared-down BSD system, with no X windows, no servers of any kind. Get my email from some other server, use only ed to read that mail on a VT100 terminal. By reducing my attack surface to the bare minimum, I can considerably reduce the chance of a successful attack.

But suppose I want to browse the Internet? I could use Lynx, but how will I display the latest satellite weather photos? Before you know it, I have millions of lines of code running X Windows, and millions more to provide a Web browser. Even if I avoid Windows and its considerably more complex IE browser, I still have increased my attack surface considerably.

Sandboxing

What I want to do, but have not yet done, is to sandbox my browser. Actually, I want to sandbox my several servers as well, but the browser is certainly important. "Sandboxing" means to isolate one application from the rest of the system; there are several ways to go about doing that.

One way would be to run a virtual machine that contains the browser. User-mode Linux (UML) makes it possible to run a virtual machine that is distinct from the actual system. If the virtual machine gets owned, I only lose what was on that system, which is essentially a large file that contains enough of a Linux install to run a browser (or mail server). Careful firewalling would prevent the virtual machine from doing anything that the browser or mail server would not have been doing. You might immediately figure out that this is not a perfect solution: A mail server exploit could still infect the virtual machine, and turn it into an attack engine that probed the same port, 25/TCP, that the firewall allowed the mail server to visit.

BSD offers its jail approach, but that may fall prey to the same problem, that is, an exploit which installs an attack tool that could, in turn, begin attacking from a BSD jail. Chroot by itself provides part of what the BSD jail does: It limits a process to a subtree of the file system. Jail does more by controlling access to network system calls as well.

There has been some research into yet another approach, one that I believe will eventually come into widespread use. That approach involves using the operating system itself to sandbox applications. Modifications have been made to both the Linux and BSD kernels to make this possible.

The Linux kernel modifications came about because several organizations, including the NSA, wanted to be able to enforce Mandatory Access Controls (MAC). Mandatory means just that. Even if you are root, you can't override these controls. You can gain authorization to configure these controls, but, theoretically at least, no exploit should be able to change these controls. Rather than modify the kernel for a single approach, 2.6.0-test2 and later kernels include hooks that support different approaches to MAC, including using the NSA SELinux approach (*http://www.nsa.gov/selinux*). *Lsm.immunix.org* has kernel patches for adding these hooks into older (2.4.20) Linux kernels.

MAC has been around for a while, with systems using MAC, including UNIX systems, having been built all through the '80s. The big problem with any system that included MAC was that ease-of-use went flying out the window. You had pain-of-use instead. But this is where the notion of sandboxing comes in. Instead of chaining down the entire system, you focus on sandboxing particular applications. You sandbox your most dangerous apps, such as any network server, your Web browser, email client, and anything like chat or IM. You can also get rid of set-user-ID programs and use MAC to control access to privileged operations, such as access to a raw network socket or writing the password file.

The big problem with any system that included MAC was that ease-of-use went flying out the window.

And how do you configure your sandbox? Fortunately, there has been research into how to do that as well. One group that has worked on configuring sandboxing called it a computer immune system. You can read some of their papers at *http://www.cs.unm.edu/~immsec/begin.html*. The basic notion is simple enough. You run your application with your sandbox in learning mode. You will need to exercise the application so that all of the important execution paths have been taken in order to create an accurate profile of normal system-call activity. Systrace, Niels Provos's approach to this in OpenBSD (*http://niels.xtdnet.nl/systrace/*) has been ported to FreeBSD and Linux, although work on the Linux port has apparently ceased.

Systrace uses execution profiling to handle the configuration. And you can run a front end to systrace (with or without X) so that it will report exceptions, allowing you to update the profile interactively.

There are a couple of approaches to doing this in Linuxland. Immunix has been around for a while, having started with buffer overflow busting approaches (StackGuard), and now sells a Linux distribution with MAC features (*http://www.immunix.org*). Grsecurity resembles the better known systrace in some ways. It, too, has a learning mode, but the latest version comes with a default least-privilege configuration.

The University of New Mexico papers (referenced above at *www.cs.unm.edu*) do suggest one feature that both systrace and grsecurity appear to lack. Sanasecurity, the company where one of the authors of the UNM papers now works, is building a product that takes a step beyond security policies that specify permitted system calls and arguments by adding in the notion of grouping system calls temporally. In other words, a policy that sandboxes an application should include not just system calls, but some context. The context chosen includes the order of recently used system calls. This appears to nail down the definition of policy a bit more than either systrace or grsecurity.

The only way that MAC versions of operating systems will succeed is if they can overcome the ease-of-use problems found in older MAC operating systems. AT&T MLS (Multi-Level Security) added over a hundred new command-line tools for routine administration back in the late ’80s, which made it difficult to use. Well, impossible for most people would be a better description. While this might be great for security, it is not going to help encourage people, even paranoid ones like myself, to move to a MAC-based system.

Still, I am worried. I just might go there . . .

KNOPPIX

Earlier in this column, I mentioned using a CD to check out a system where a kernel rootkit might be installed. One approach is to use the install CD that is available for just about any version of UNIX (hard to imagine that such a CD wouldn’t exist). You boot the install CD, then escape to a shell. Often, the menu system that comes with most install CDs will make this easy to do. Then you mount your hard drive partitions and start looking for evidence of a rootkit.

If you are working with Intel or PowerPC-based systems, there is something even better to work with. KNOPPIX (*knoppix.org*) is a single-boot CD that contains a huge collection of GNU software, running on top of a Linux kernel. You might be tempted to ignore KNOPPIX, because you already have bootable Linux install CDs. But what makes KNOPPIX special is the care taken by its author, Klaus Knopper, in writing setup scripts. I found that KNOPPIX worked better than many Linux distros (most!) at finding key devices, like your monitor and graphics card, and setting up a useful environment. I plan on using KNOPPIX in my Boston USENIX class.

KNOPPIX also probes hard drive partitions and sets up an `/etc/fstab` for read-only mounting of the partitions it finds. You can even examine Windows NTFS partitions. I had my only Win2K box bluescreen, all by itself, for no good reason – but with an unbacked-up copy of my 2003 tax data. The Win2K install disk couldn't mount this partition, but KNOPPIX could, allowing me to recover key files.

I highly recommend that you make certain you have bootable CDs handy for any Linux or UNIX system you manage. In times of crises, you don’t want to waste time searching for one – you want to have what you need at hand, and know how to use it.