(Almost) undetectable hardware-based rootkits

FOURTH-GENERATION ROOTKITS

We look at the history of the rootkit, including its newest incarnation, the DR RootKit. **BY KURT SEIFRIED**

Originally, I intended to write an article about the current state of rootkits and the tools that could be used to detect them. But I ran into a slight problem – the more modern rootkits tend to be really good at avoiding detection. By really good, I mean that you're unlikely to detect them unless you take action, such as a detailed analysis of a system memory dump, for example, comparing the actual kernel image with the expected.

History Lesson

Traditional rootkits were relatively simplistic programs, often running as a standalone daemon providing backdoor access. These were generally easy to detect by looking for a new process or newly installed software, which led attackers to start subverting system binaries. In turn, this led to attackers installing modified system binaries, such as hacked versions of OpenSSH that have a hard-coded administrative username and password to get root-level access. With the advent of tools such as Tripwire and the increasingly common use of package managers that can verify the integrity of installed files, such as RPM and dpkg, these became easy to detect [1].

Kernel-Based Rootkits

Soon attackers realized that more sophisticated hiding and subversion methods were needed to control a system, which led to kernel-based rootkits. By modifying the system call table, an attacker can avoid detection easily because, simply put, they control what you are seeing and how your programs are executing.

Typically attackers use one of two methods to modify the system kernel: either loading a malicious kernel module (e.g., heroin) or patching the in-memory kernel by writing to the special device */dev/kmem* (e.g., SucKIT). Because these attacks live in memory only, their disadvantage is that they typically do not survive a reboot.

Difficult as they can be to detect, these rootkits can be found by comparing the current system call table with the expected (i.e., by examining the file *System.map*). Dumps of system memory can be taken and used to verify that the kernel in memory is correct.

So what are attackers to do? Go deeper, of course.

Hardware-Based Rootkits and the Virtualized OS

Released in 2006 at Black Hat in Vegas, the first publicized hardware-based rootkit was called "Blue Pill" [2]. Modern CPU's from AMD and Intel include a number of features that support virtualization of operating systems. Because they no longer need to modify the operating system to work, these rootkits are harder to detect, so checking your system call table won't work. However, these rootkits do replace the Interrupt Descriptor Table (IDT), which is held within a CPU register (the IDTR) [3].

Because two IDTRs (the real one and the one being presented to the compromised operating system) now exist, the one being presented to the compromised operating system will be at a different memory location than usual. Fortunately, the privileged instruction Store Interrupt Descriptor Table (SIDT) can be run from user space and will reliably return the contents of the IDTR being presented to the operating system (which isn't very helpful because it has been compromised) and, more importantly, the memory location (which won't be in the normal location).

This appears to be a stalemate: The attackers have created new methods to

hide rootkits, and the defenders have found ways to detect them.

New Generation of Rootkit

Released in September 2008 by Immunity Inc., DR RootKit [4] implements system call hooking within Linux 2.6 kernels without modifying the system call table or the interrupt descriptor table. To do this, it places a hardware breakpoint on the syscall handler. This trap places a memory watch on the *syscall_table* entry *__NR_syscall*, which is used to export system call numbers. Basically, the rootkit behaves like a debugging tool, waiting for specific system calls to be executed, and when it sees them, it modifies them on the fly. Currently, the DR RootKit hooks the system calls listed in Table 1.

The DR RootKit includes capabilities such as hiding processes and preventing hidden processes (meaning that the attacker can run software that is hidden, and you can't kill it even if you manage to guess the process ID) from being terminated. Using the examples provided with the package, it is relatively easy to extend and create additional modified system calls. For example, you might want to modify the capset so that you can set process capabilities at will. The rootkit itself is a loadable kernel module, which makes it easy to insert once you have compromised a system, but like other memory-based rootkits, a system reboot will remove it from memory.

If you want to extend the rootkit, you can insert your own custom system calls. An example is given for replacing the exit system call. Simply put, the process consists of declaring your own hook to replace a system syscall and then writing a custom system call implementation – simple, really. The best place to start is with the kernel source – specifically, the *kernel/* subdirectory where most of the common system calls are defined. For example, if you have a system in which capabilities are in use to restrict what programs can and cannot do, you can modify the *do_sys_capset_other_tasks* system call to call a modified *cap_set_all*, which always returns all capabilities for a specific process ID, such as:

```
@@ -237,6 +237,9 @@
if (!capable(CAP_SETPCAP))
return -EPERM;
+  if (pid == 12345)⮐
/* magic process number*/
+  return cap_set_all_evil⮐
(effective, inheritable, ⮐
permitted);
```

As you can see, even a minor modification can have a significant effect. Suddenly, the process with ID 12345 will always have all capabilities, allowing it to do pretty much anything it wants. With just one system call, an attacker can create an effective backdoor.

Virtually the only way to detect this rootkit is through the measurement of timing or race conditions that are introduced by the rootkit. If a rootkit is present, the system should run a little slower than usual, but measuring this reliably is not an easy task, especially on production systems. Also, the software is relatively simplistic and can be extended easily to hide itself better, making detection even harder.

Alternative Approach

Of course, you can compromise a system and retain access in other ways while also staying hidden. Another penetration testing software company called Core Security [5] has taken the approach of injecting hostile code into the process that has been attacked. For example, if you exploit an Apache httpd server, you can inject code into the process that will allow you to have remote access. This technique is somewhat limited compared with a full kernel or hardware-based rootkit, and it also is less likely to affect the entire system, making it stealthier. The primary disadvantage of this technique is that operating system-level protection mechanisms, such as SELinux, will still be able to enforce security policy. However, for targeted attackers, this is often not a serious problem because they can either use local exploits to compromise the system further or stay within the behavioral confines of an SELinux policy and still extract information or use the system to execute other malicious attacks.

Conclusion

The good news is that by going to the hardware level, the attackers have (conceptually) run out of room to go. The bad news is that any number of hardware tricks can be used to maintain control over a compromised system. For example, a modern graphics card typically has direct memory access (meaning it can do pretty much anything it wants to the system memory without the operating system having much say in the process), its own onboard memory, and a large amount of processing power (to the point where people are using them as a poor man's computing cluster). Newer cards have flash memory to hold firmware that can be updated from software, and I have no doubt that one day, people will be working on how to use your video card to maintain control over a compromised system. ■

Table 1: System Calls

getdents64	Read directory entries
getdents	Read directory entries
chdir	Change working directory
open	Open a file or device
execve	Execute program
socketcall	Socket system calls
fork	Create a child process
exit	Terminate the current process
kill	Send signal to a process
getpriority	Get program scheduling priority

INFO

[1] "Secret Passage: Techniques for Building a Hidden Backdoor" by Amir Alsbih, *Linux Magazine*, April 2007: *http://www.linux-magazine.com/issues/2007/77/secret_passage*

[2] Blue Pill: *http://bluepillproject.org/*

[3] Red Pill: *http://www.invisiblethings.org/papers/redpill.html*

[4] DR RootKit: *http://www.immunityinc.com/resources-freesoftware.shtml*

[5] Core Security Technologies: *http://www.coresecurity.com/*

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He is married and has four cats but no fish (because the cats are more hungry than afraid of water). He often wonders how it is that technology works on a large scale but often fails on a small scale.