Interested in learning
more about security?

SANS Institute
InfoSec Reading Room

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Kernel Rootkits

Loadable Kernel Modules (LKMs) allow the running operating system kernel to be extended dynamically. Most modern UNIX-like systems, including Solaris, Linux, and FreeBSD, use or support loadable kernel modules. The facility offers more flexibility than the traditional method of recompiling the kernel to add new hardware support or functionality; new drivers or functionality can be loaded at any time. A loaded kernel module has the same capabilities as code compiled into the kernel. This gives loadable drivers a lot of ...

Copyright SANS Institute
Author Retains Full Rights

AD

Kernel Rootkits

Dino Dai Zovi <dadaizo@sandia.gov>

Background

Loadable Kernel Modules (LKMs)

Loadable Kernel Modules (LKMs) allow the running operating system kernel to be extended dynamically. Most modern UNIX-like systems, including Solaris, Linux, and FreeBSD, use or support loadable kernel modules. The facility offers more flexibility than the traditional method of recompiling the kernel to add new hardware support or functionality; new drivers or functionality can be loaded at any time. A loaded kernel module has the same capabilities as code compiled into the kernel. This gives loadable drivers a lot of flexibility and power. However, it also allows a maliciously written kernel module to subvert the entire operating system kernel[8].

System installed modules reside in `/lib/modules`, `/modules`, and `/kernel` on Linux, FreeBSD, and Solaris, respectively. The system typically auto-loads modules from these directories. However, the modules can be loaded from anywhere on the system using `insmod`, `kldload`, or `modload` (depending on the operating system).

System Calls

Most modern processors support running in several privilege modes. Most processors support two modes, user mode and supervisor mode. Some processors, such as Intel 386 or greater processors, support more modes (although most operating systems only use two of them). User processes (even processes running as the superuser) run in user mode while only kernel routines run in supervisor mode. The mode distinction allows the operating system to force user processes to access hardware resources only through the operating system's interfaces. The mode distinction is very important in the operating system's virtual memory, multitasking, and hardware access subsystems.

The method by which a user mode process requests service from the operating system is the *system call*. System calls are used for file operations (open, read, write, close), process operations (fork, exec), network operations (socket, connect, bind, listen, accept), and many other low-level system operations.

System calls are typically listed in `/usr/include/sys/syscall.h` or in `/usr/include/bits/syscall.h` on Linux. In the kernel, the system calls are typically stored in a table (an array of pointers) indexed by the system call number. When a process initiates a system call, it places the number of the desired system call in a global

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

register or on the stack and initiates a processor interrupt or trap (depending on the processor architecture).

Rootkits

``Rootkits'' are software packages installed to allow a system intruder to keep privileged access. Traditional rootkits typically replace system binaries like `ls`, `ps`, and `netstat` to hide the attacker's files, processes, and connections, respectively. These rootkits were easily detected by checking the integrity of system binaries against known good copies (from vendor media) or checksums (from RPM database or Tripwire).

Kernel rootkits do not replace system binaries, they subvert them through the kernel. For example, `ps` may get process information from `/proc` (`procfs`). A kernel rootkit may subvert the kernel to hide specific processes from `procfs` so `ps` or even a known good copy from vendor media will report false information. In addition, a malicious kernel module can subvert the kernel so that it is not listed in kernel module listings (from the `lsmod` command). Kernel rootkits do this by redirecting system calls. As a kernel module has as much power as any other kernel code, it can replace system call handlers with its own wrappers to hide files, processes, connections, etc. The file access system calls can also be overwritten to cause false data to be read from or written to files or devices on the system.

A large collection of rootkits can be found on the PacketStorm archive at `http://packetstorm.securify.com/UNIX/penetration/rootkits/`. At present, at least eight LKM rootkits are archived there. Three such LKM rootkits will be briefly described below.

LKM Rootkits

Knark[2]

Knark is a rootkit written by Creed to explore the ideas he read in an article of Phrack [7]. The current release, v0.59, installs a kernel module `sysmod.o` that hides listening sockets, files, and directories. Knark's kernel module changes seven system calls: `fork`, `read`, `execve`, `kill`, `ioctl`, `settimeofday`, and `clone`.. Knark includes much more than just the kernel module, throwing in a small collection of exploits for the attacker to use to gain control of more systems.

Adore

Adore, written by Stealth, implements file hiding, process hiding, and privileged command execution. Adore does not implement any remote access features like Knark. A command-line utility, `ava` is used to give commands to the kernel module to specify

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

which files and processes to hide. A password (the ``elite command") is compiled into the module and `ava` to prevent unauthorized access and make fingerprinting more difficult.

Like Knark, Adore changes eight system calls: `fork`, `write`, `close`, `clone`, `kill`, `mkdir`, `clone`, `getdents`.

Rkit

Rkit, by TBob <tbob@techie.com> is the simplest rootkit described here. It simply changes the `setuid` system call to give root (uid = 0) access to a specific user.

Detecting LKM Rootkits

As mentioned before, most LKM rootkits redirect system calls by overwriting entries in the system call table. By examining the locations in memory of the system call entries, it can be determined which system calls are handled in the base kernel or which calls are intercepted by modules. On Linux, in particular, the `System.map` file (usually found in `/boot`), contains the memory addresses of kernel functions (including system calls). Comparing the addresses of system call functions in the `System.map` with the addresses in the system call table in the running kernel (read through `/dev/kmem`) identifies which system calls have been redirected by loaded modules.

KSTAT[6]

KSTAT (Kernel Security Therapy Anti-Trolls), available at `http://www.s0ftpj.org/tools/kstat.tgz`, is a tool for Linux that uses several methods to check the integrity of the running kernel. The `-s` option of KSTAT prints out the memory addresses of all system calls as read from `/dev/kmem`. If the address does not match the address listed in `System.map`, a warning message is displayed.

For example, the following is the output of `kstat -s` on a system that has the Adore rootkit installed.

```
sys_fork        0xc284652c WARNING! Should be at 0xc0108c88
sys_read        0xc2846868 WARNING! Should be at 0xc012699c
sys_execve      0xc2846bb8 WARNING! Should be at 0xc0108ce4
sys_kill        0xc28465d4 WARNING! Should be at 0xc01106b4
sys_ioctl       0xc2846640 WARNING! Should be at 0xc012ff78
sys_settimeofday 0xc2846a8c WARNING! Should be at 0xc0118364
sys_clone       0xc2846580 WARNING! Should be at 0xc0108ca4
```

listsyscalls

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

`listsyscalls` by Bruce M. Simpson aka User Datagram Protocol <udp@low-level.net> inspects the contents of the Solaris system call table. Redirected system calls can be identified by comparing the output with a previously generated baseline.

Some sample output of `listsyscalls` reporting on system calls 0-6 follows:

```
syscall: nosys [0]  sy_narg: 1 sy_flags: 0x02
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f0093a8c [no symbol]
syscall: rexit [1]  sy_narg: 1 sy_flags: 0x00
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f00b045c [no symbol]
syscall: fork [2]  sy_narg: 3 sy_flags: 0x00
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f00d7eb0 [no symbol]
syscall: read [3]  sy_narg: 1 sy_flags: 0x00
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f009dc78 [no symbol]
syscall: write [4]  sy_narg: 1 sy_flags: 0x00
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f012fbb0 [no symbol]
syscall: open [5]  sy_narg: 2 sy_flags: 0x00
sy_call: 0 [no symbol] sy_lock: 0  sy_callc: f009a108 [no symbol]
syscall: close [6]  sy_narg: 8 sy_flags: 0x02
sy_call: f00cd378 [no symbol] sy_lock: 0  sy_callc: f0093888 [no
symbol]
```

Fingerprinting LKM Rootkits

As mentioned before, most LKM Rootkits function by redirecting system calls. Many rootkits overwrite the same common system calls. However, the exact set of system calls that are redirected is relatively unique and can be used to identify the rootkit installed on a system. In addition, once it is determined that a module is installed, it may also be possible to determine which files it is hiding. These may include configuration files for the module and these could also possibly reveal the module's origin.

Defenses

Disable LKMs

The most obvious solution to LKM trojans and rootkits is to disable the entire Loadable Kernel Module facility. By sacrificing some system flexibility and ease of maintenance, we gain the security of not having to worry about foreign code being loaded into our kernel. Unfortunately, that is not the case. Even without LKM support, code can be injected by writing directly to `/dev/kmem` [1]. However, turning off LKMs will thwart most automated works and unskilled attackers.

Securelevels

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

`4.4BSD`[3] introduced new file flags and kernel security levels. Several flags could be set on a file, including *immutable* and *append-only*. To support this, the kernel runs in a specific security level. The security level can be raised by any superuser process, but it can only be lowered by the `init` process (`pid` 1). `init` needs to lower the secure level to shutdown the system, but no other process can lower the secure level for any other reason. The four secure levels and what they entail are detailed below.

-1.
: *Permanently insecure mode*: Kernel runs at securelevel 0 at all times.

0.
: *Insecure mode*: File flags may be set and reset and devices may be read from or written to according to their permissions.

1.
: *Secure mode*: Superuser-settable file flags cannot be turned off. Device files for system memory `/dev/mem`, `/dev/kmem` and for mounted filesystems are read-only.

2.
: *Highly secure mode*: Device files for filesystems are always read-only, whether they are mounted or not. Firewall rules may not be changed.

The higher security levels (1 and 2) require many system administration tasks to be done at console. To edit flagged files, change firewall rules, or create new filesystems, the system must be shutdown into a lower security level (typically single-user mode). However, this ensures that administration is done by authorized individuals combining physical security with computer security.

Securelevel support exists in all the BSD-derived operating systems and 2.0.3x versions of the Linux kernel. Linux has since removed the support from the 2.2 series in favor of POSIX capabilities (currently incomplete) [4].

Cryptographically Signed Kernel Modules

One way to ensure that only trusted code runs in the kernel is to cryptographically sign all kernel modules. At kernel compile-time, a random RSA (or another asymmetric algorithm that supports signatures) keypair could be generated. All modules compiled with the kernel would be signed by the private key. Afterwards, the private key would be discarded in a secure manner. It would be necessary to ensure that the temporary private key is not written out to disk or swap. The generated public key could be stored in the kernel image.

To be secure, this system would require the use of securelevels (or an equivalent). For example, an attacker could overwrite the trusted public key through `/dev/kmem`. An attacker could also overwrite the public key in the kernel image on the filesystem (or through the raw disk device) and reboot. The kernel image and kernel memory devices must be protected by securelevels to ensure that even a process with superuser access could not write to them.

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

Unfortunately, perhaps the best solution (signed modules), does not yet exist in a usable form[5]. Under Linux, module relocation is done by userland processes `insmod` or `kerneld`. Because the modules are altered by these processes before being handed off to the kernel, signatures must be verified by the userland processes; relocation would make the hash of the module incorrect. However, these userland processes can easily be subverted by an attacker with superuser access. They could use their own version of `insmod` that does not verify signatures. In response, the kernel could only accept module load system calls from a process run from a specific inode (guaranteeing that the system's `insmod` was being used). However, the attacker could then use the process debugging facilities `ptrace` or `procfs` to alter the text segment of `insmod` in its memory image to not check the module's signature by overwriting the call to `checki_signature()` (or whatever the function would be called) with machine-code NOPs (no-operations). Similar issues may exist in other operating systems that support kernel modules.

Bibliography

1
Silvio Cesare.
Runtime kernel kmem patching.
http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt.

2
Jonathan P. Clemens.
An email interview with creed, the author of knark.
http://jclemens.org/knark/creed_interview1.html.

3
Marshall Kirk McKusick et al.
The Design and Implementation of the 4.4BSD Operating System.
Addison-Wesley, 1996.

4
Goran Gajic.
Securelevel support for linux 2.2.xx.
http://147.91.1.113/~ggajic/securelevel.html.

5
Pavel Kankovsky.
Re: Announcement: Solaris loadable kernel module backdoor.
Posting to BUGTRAQ mailing list, December 28 1999.

6
Toby Miller.
Detecting loadable kernel modules (lkm).
http://www.s0ftpj.org/docs/lkm.htm.

7
plaguez.
Weakening the linux kernel.
Phrack, 8(52), January 1998.

8

© SANS Institute 2001, As part of the Information Security Reading Room. Author retains full rights.

pragmatic.
(nearly) complete linux loadable kernel modules.
http://www.pimmel.com/articles/lkm-hacking.html.

© SANS Institute 2001, Author retains full rights

Last Updated: May 24th, 2013

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Malaysia @ MCMC 2013	Cyberjaya, MY	Jun 03, 2013 - Jun 08, 2013	Live Event
SANS Pen Test Berlin 2013	Berlin, DE	Jun 03, 2013 - Jun 08, 2013	Live Event
Industial Control Systems Security Training - Houston	Houston, TXUS	Jun 10, 2013 - Jun 15, 2013	Live Event
Security Impact of IPv6 Summit 2013	Washington, DCUS	Jun 14, 2013 - Jun 16, 2013	Live Event
SANSFIRE 2013	Washington, DCUS	Jun 14, 2013 - Jun 22, 2013	Live Event
SANS Canberra 2013	Canberra, AU	Jul 01, 2013 - Jul 13, 2013	Live Event
Digital Forensics & Incident Response Summit 2013	Austin, TXUS	Jul 09, 2013 - Jul 16, 2013	Live Event
SANS London Summer 2013	London, GB	Jul 09, 2013 - Jul 16, 2013	Live Event
SANS Rocky Mountain 2013	Denver, COUS	Jul 14, 2013 - Jul 20, 2013	Live Event
SANS Mumbai 2013	Mumbai, IN	Jul 22, 2013 - Jul 27, 2013	Live Event
SEC528 SANS Training Program for the CompTIA® New Advanced Security Practitioner (CASP) Certification	Washington, DCUS	Jul 22, 2013 - Jul 26, 2013	Live Event
SANS San Francisco 2013	San Francisco, CAUS	Jul 29, 2013 - Aug 03, 2013	Live Event
SANS SEC 560: Network Penetration Testing @ Bangalore 2013	Bangalore, IN	Aug 05, 2013 - Aug 10, 2013	Live Event
SANS Boston 2013	Boston, MAUS	Aug 05, 2013 - Aug 10, 2013	Live Event
Critical Security Controls Summit	Washington, DCUS	Aug 12, 2013 - Aug 18, 2013	Live Event
Industrial Control Systems Security Training - DC	Washington, DCUS	Aug 12, 2013 - Aug 16, 2013	Live Event
SANS Thailand 2013	Bangkok, TH	Aug 19, 2013 - Aug 31, 2013	Live Event
SANS Virginia Beach 2013	Virginia Beach, VAUS	Aug 19, 2013 - Aug 30, 2013	Live Event
Mobile Device Security Summit 2013	OnlineCAUS	May 30, 2013 - Jun 06, 2013	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced