;login:
THE MAGAZINE OF USENIX & SAGE
November 2001 • Volume 26 • Number 7

Special Focus Issue: Security
Guest Editor: Rik Farrow

inside:

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

loadable kernel modules

The New Frontier for Incident Response

by Keith J. Jones

Keith Jones is a computer forensic consultant for Foundstone. His primary area of concentration is incident response program development and computer forensics.

Keith.jones@foundstone.com

What would you do if traditional incident response tools completely failed during an investigation? That is exactly what I experienced when up against a loaded evil kernel module. Loadable kernel modules are changing the techniques used to perform an incident response because the level of compromise is raised from user space to kernel space. Once the compromise breaches the kernel space, the effects trickle down to any user-space executable resident on the trojaned system. This effect allows an intruder to change the behavior for any command executed on the system without changing the program binaries themselves. With this in mind, any trusted toolkit you transfer to the victim machine will also be automatically compromised. Therefore, I will explain how one malicious kernel module works and describe a couple of tools I developed to cope with the problem.

Overview of Loadable Kernel Modules

Loadable kernel modules (LKM) are a blessing for the system administrator, but a nightmare for an incident responder. LKMs were initially designed to provide dynamic functionality by altering a running kernel without rebooting. The slight altering of a running kernel can provide additional support for other devices such as new file system types and network adapters. Additionally, since kernel modules can access all functions and memory areas of a kernel, the depth of what it can alter reaches the whole operating system without any controls. Therefore, every function and memory resident struct is in danger of being compromised by a malicious kernel module.

One well-known malicious module for Linux kernels is named knark. Once knark is compiled and loaded on a victim machine, the syscall table is altered, which changes the operating system behavior. Basically, the syscall table is the entry point into the operating system provided to user-level programs and resides in kernel space. The formal definition of "syscalls" is given in manual section two of most UNIX operating systems. Whenever the kernel executes on behalf of user space, the area of an operating system a typical user executes in, the OS maps all of the commands and functionality executed on the command line to system calls within this table. Therefore, when knark alters the syscall table, it is altering user command execution. The important system calls knark alters are the following:

- getdents – This system call gets the directory entries (i.e., the files and directories) of a given directory. By compromising this call, knark is able to hide files and directories from user-level programs.
- kill – This system call sends a signal to a process, typically to kill it. This call is compromised such that an extra unused signal, #31, will trigger the option flags of a process to be set to the "hidden" state. When a process is hidden, its entry from the /proc file system is removed and therefore hidden from ps. Signal #32 unhides the file by resetting the option flags of the task.
- read – This system call reads the contents of a given file. Knark compromises this call such that it hides intruder connection specifics from netstat. The specifics are hidden because they are read from the /proc file system as files.
- ioctl – This system call changes the behavior of files and devices. When knark compromises this system call, it is able to clear the promiscuous flag on the network

With control of the OS, an intruder can make it return false information to the user-space queries

interface cards. Additionally, knark also inserts the functionality of hiding and unhiding files into this function.

- fork – This system call spawns a new process. When knark compromises this system call, it will hide all child processes created from a hidden parent process.
- clone – This system call spawns a new process. When knark compromises this system call, it will hide all child processes created from a hidden parent process.
- execve – This system call executes a file. It is called every time a command is entered at the prompt by a user. When this system call is trojaned, the kernel module can manipulate how and what commands are executed. knark allows an intruder to point one executable to another, similar to a symbolic link but without the evidence. When execve is compromised by knark, the destination executable runs instead of the expected source program.
- settimeofday – This system call sets the system time. knark compromises this system call by watching for predetermined clock-setting times. When one of these times is sent to this call to reset the clock, knark can either execute some administrative tasks or give the current user the user ID and group ID of root immediately. This eliminates the need of changing a shell to SUID-root in order to give root privileges to an ordinary user.

Since the syscall table has been compromised, the functionality of administrative tools is altered. netstat reports a network interface card that is never in promiscuous mode, and connections from given locations disappear. ps and top do not report hidden processes because they disappear from the /proc file system. ls ignores hidden files and directories. All of this occurs because when the tools are run they rely on the operating system to supply information. With control of the OS, an intruder can make it return false information to the user-space queries. This occurs without changing the binary files for netstat, ps, top, and ls. Therefore file system checksum tools, such as Tripwire, are useless against this type of compromise. Checksum tools are also defenseless against the executable redirection capabilities of knark. If an intruder were to link a file hackme to cat, every time cat is run the program hackme is executed instead. This allows cat to remain on the file system with the same MD5 checksum, yet execute with different functionality.

Furthermore, transferring a new set of tools to a victim machine with knark installed will not change the data reported. Even a trusted tool set must make system calls, therefore the tools become untrusted immediately when running on the victim machine. There is currently no way to circumvent a kernel-level compromise without using a toolkit that also enters the kernel space. This was my motivation to develop tools and techniques to check for the installation of LKMs and capture processes when a system may have a malicious LKM installed.

One caveat not previously mentioned is the existence of knark.o in the loaded module list reported by lsmod. Unfortunately for the investigator, there is a simple way to make this information disappear for the intruder. knark is packaged with another loadable kernel module named modhide, which makes itself and the last loaded module disappear. Once a module has disappeared, there is no way to unload it without rebooting the machine. Additionally, there is no easy method to even detect it is loaded, because all identifiable references to the module disappear. As has been shown, knark comes with all the tools to make it the ultimate stealthy rootkit.

Preventative Measures

If the ability to prevent a loadable kernel module compromise is available, it will obviously be the best solution. There are a few measures you can take to protect yourself from loadable kernel modules ahead of time. You can protect yourself from most of the maliciousness kernel modules can cause by securing your syscall table. A simple loadable kernel module can be constructed that watches the syscall table at periodic intervals and when other modules are loaded. If the sentry module discovers that the syscall table has been modified from its original state, it can alert the system administrator and even change it back to the original value. The following example code will work well with Linux 2.2 and 2.4. If you have a machine with more than one processor, it can be compiled by the following command: gcc –D__SMP__ -c syscall_sentry.c. If you have a machine with a single processor, just remove the –D__SMP__ definition. Once it is compiled, load it into the running kernel with insmod.

If the ability to prevent a loadable kernel module compromise is available, it will obviously be the best solution

```
/*
 * This LKM is designed to be a tripwire for the sys_call_table.
 */

#define MODULE_NAME  "syscall_sentry"

/* This definition is the time between periodic checks. */
#define TIMEOUT_SECS    10

#define MODULE
#define __KERNEL__

#include<linux/module.h>
#include<linux/config.h>
#include<linux/version.h>
#include<linux/kernel.h>
#include<linux/sys.h>
#include<linux/param.h>
#include<linux/sched.h>
#include<linux/timer.h>
#include<sys/syscall.h>

/* This function is a simple string comparison function */
static int mystrcmp( const char *str1, const char *str2)
{
      while(*str1 && *str2)
          if (*(str1++) != *(str2++))
              return -1;
      return 0;
}

/* This function builds a timer struct for versions of linux
 * less than Linux 2.4. It is used to set a timer
 */
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,4,0)
/* Initializes a timer */
void init_timer(struct timer_list * timer)
{
            timer->next = NULL;
            timer->prev = NULL;
}
#endif
```

```
/* This is our timer */
static struct timer_list syscall_timer;

/* This is the system's syscall table */
extern void *sys_call_table[];

/* This is the saved, valid syscall table */
static void *orig_sys_call_table[ NR_syscalls ];

/* This function is needed to protect yourself */
static unsigned long (*orig_init_module) (const char *, struct module*);

/* This function checks the syscalls for changes
 * and changes them back to the original if it has
 * been changed.
 */

static int check_syscalls( void )
{
      int i;

      /* Add a new timer for our next check */
      del_timer( &syscall_timer );
      init_timer( &syscall_timer );
      syscall_timer.function = (void *)check_syscalls;
      syscall_timer.expires = jiffies + TIMEOUT_SECS * HZ;
      add_timer( &syscall_timer );

      for ( i = 0; i < NR_syscalls - 1; i++ )
      {
          if (orig_sys_call_table[i] != sys_call_table[i])
          {
              printk(KERN_INFO "\nSysCallSentry - sys_call_table has been
                        modified in entry %d!\n", i);
              sys_call_table[i] = orig_sys_call_table[i];
          }
      }

      return 1;
}

/* Check sys_call_table anytime a new module is loaded. */
static int long sys_init_module_wrapper( const char *name, struct
                        module *mod )
{
      int i;
      int res = (*orig_init_module)(name,mod);

      for ( i = 0; i < NR_syscalls - 1; i++ )
      {
          if (orig_sys_call_table[i] != sys_call_table[i])
          {
              printk( KERN_INFO "\nSysCallSentry - sys_call_table has been
                        modified in entry %d!\n", i);
              sys_call_table[i] = orig_sys_call_table[i];
          }
      }
      return res;
}
```

```
/* Module Init Code */
static int init_module (void)
{
     int i;
     printk(KERN_INFO "\nSysCallSentry Inserted\n");

     /* Initiate the periodic timer */
     init_timer( &syscall_timer );

     /* Save the old values of the sys_call_table */
     orig_init_module = sys_call_table[SYS_init_module];

     /* Wrap the init_module syscall. This will check to see
     * if any calls have been altered when a new module loads.
     */
     sys_call_table[SYS_init_module] = sys_init_module_wrapper;

     for ( i=0; i < NR_syscalls - 1; i++ )
     {
        orig_sys_call_table[i] = sys_call_table[i];
     }

     /* Start our first check */
     check_syscalls();
     return(0);
}

/* Module Cleanup Code */
static void cleanup_module (void)
{
     /* Return system status to the original */
     sys_call_table[SYS_init_module] = orig_init_module;
     printk(KERN_INFO "\nSysCallSentry Removed\n");
}
```

The current "state of the art" in LKM rootkitting is to modify the syscall table

The current "state of the art" in LKM rootkitting is to modify the syscall table. Therefore, this method of placing a sentry on the syscall table is practical because the syscall table changes so infrequently. Possibly the best true preventative measure you can take to protect your machines from this type of compromise is to remove the ability to load kernel modules completely. Production servers should have all the code they need to run compiled into the kernel, and loadable kernel modules should not be used.

There is another option available to protect yourself against hostile LKMs. A tool called "St. Jude," when compiled with one called "St. Michael," both guard against the modification of the syscall table, and checks root transitions for evidence of attacks, based on a ruleset created during a learning phase. (*http://www.sourceforge.net/projects/stjude*).

Development of Investigative Tools and Techniques

It is obvious that the investigation must examine the victim machine's kernel space in order to effectively respond to a kernel-level compromise. Therefore, investigators must change their tools and techniques. It will be assumed that the response to an incident involving knark will include a forensic duplication of the victim machine's storage devices. Therefore, any hidden files will be available to the investigator using that method. What the investigator will miss, however, are hidden processes and network information. This can be remedied by developing a kernel level "ps-like" tool that also retrieves executable images of each process. This tool will be a loadable kernel

Access to the process executable image in Linux is not trivial, but it is possible

module so that it can be loaded after an incident occurs. This section will describe, at a high level, one such tool and how it works to circumvent the problems involved with kernel-level investigations on a Linux 2.2 platform.

The most important struct for a kernel level ps tool is task_struct. It is a circularly linked list where every process on the system is present. Every action available for the process is available inside this struct, such as opened files, an executable image of the process, opened network sockets, file operators, and more. The following are several fields of important information for the investigator which will be written to a log file.

- *Process ID (PID)* – This is the unique number used to identify a running process.
- *User ID* – This is the user number that executed the process. It is important to know what privilege level the process is running as.
- *Process Status* – This flag indicates how the process is currently running. Since a process cannot occupy the processing power of the CPU all of the time, it may be sleeping. This flag will indicate what running state the process is in.
- *Process Name* – This is the human-recognizable name for the process. It is the equivalent to a portion of the command line used to execute this process.
- *Start Time* – This is reported in "jiffies," which is the number of system clock ticks since the machine was booted. This field is used to determine when the process was started. It is obvious when a process is initiated using a startup script during system boot because the number is relatively small. It may also provide more clues as to when the intrusion occurred.
- *Open File Handles* – Since everything in UNIX is a file, viewing the open file handles of a process allows you to see all open regular files, network sockets, and FIFOs. This information will be pertinent when tracking down processes that save information to files, like sniffers, or open network sockets, like back doors.
- *Command Line Arguments* – The command line arguments are available in the task struct and are useful when deciphering the options with which a process was executed. For example, imagine an intruder started netcat. It would be difficult to observe where it was connecting unless you had the command line arguments. The command line arguments available in the task struct would include the IP addresses and ports for netcat.
- *Process Environment Variables* – Each process that is executed has its own version of an environment table. Typically, it is a duplicate of the environment that the initiating user had at the time of the execution. Therefore, extra information of the intruder's session will be available by examining the environment variables available in the task struct.

Therefore, a tool would iterate through this circular linked list, saving the information for each process to a log file. This information would be very similar to the ps –ef command, so most investigators will be able to read it easily. Additionally, a separate file will be created for the process containing the executable image, also found in the task struct, for further offline tool analysis. Access to the process executable image in Linux is not trivial, but it is possible. The image resides in the memory map struct of the task. Within that memory map struct there is a virtual memory area associated with the task. Within that area, there is a virtual-memory file, which contains a file operator array. Once we have found the proper read function, reading the executable image and writing it out to another file is simple. Theoretically, a process should always have an executable image completely loaded in memory because it is possible to delete binaries from the file system after they are running. The most difficult part of acquiring the image is finding where it resides in memory.

While your module is running, no other process can be scheduled to run. Interrupts and other system activity can still take place, but the module has preempted execution. Because this module "freezes" the process list, I named it "Carbonite." The source code is available from *http://www.foundstone.com* and is a good basis to develop tools like it for later versions of Linux and different operating systems.

One last fact you can use to determine if knark is loaded on your system is to view the network card status. When knark is loaded, it never lets the network interface report it is in promiscuous mode. This is to prevent a system administrator from observing an intruder's sniffer, which places the card into promiscuous mode. What you can do, however, is simply run tcpdump or any other sniffer that uses promiscuous mode and view the status of the network adapter using ifconfig. If the card does not report that it is in promiscuous mode, knark could be loaded.

Conclusion

The ability to load kernel modules is a significant blow to incident responders. The malicious loadable kernel module is already publicly available and probably used in many intrusions. It raises the bar of compromise and incident response to the kernel level. Although it may seem that kernel modules are a significant factor when performing investigations, they are not lethal. You can see that there are simple preventative measures that you can take to protect yourself against this type of compromise. Furthermore, investigative tools and techniques for this type of compromise fight fire with fire by also executing in kernel space.

The ability to load kernel modules is a significant blow to incident responders.