Kernel Korner

Loadable Kernel Module Exploits

Beat potential invaders at their own game by learning how to use cracker tools and improve your own security.

by William C. Benton

Many useful computer security tool ideas have a common genesis: the cracker world. Tools, like port scanners and password crackers, originally designed to aid black-hats in their attempts to compromise systems, have been profitably applied by systems administrators to audit the security of their own servers and user accounts. This article presents a cracker idea--the kernel module exploit--and shows how you can improve your system's security by using some of the same ideas and techniques. First, I will discuss the origin of my idea and how it works, then I will attempt to demystify the art of kernel module programming with a few short examples. Finally, we will walk through a substantial, useful example that will help prevent a class of attacks from compromising your system.

Before we get started, I need to mention the standard disclaimer. Be aware that a bug in kernel space is liable to crash your machine, and an endless loop in kernel space will hang your machine. Do not develop and test new modules on a production machine, and test modules thoroughly to ensure they do not destabilize your system or corrupt your data. To minimize data loss due to system crashes in the debugging cycle, I recommend that you either use a virtual machine or emulator (like bochs, plex86, the User-Mode Linux port or VMware) for testing, or install a journaling filesystem (like SGI's xfs) on your development workstation. Furthermore, none of the code examples in this article have been tested on an SMP machine, and most of it is likely not multiprocessor safe. Now that we have that out of the way, let's talk about modules.

A few months ago, I was developing a system called audit trail generator for Linux. For every process on a system, I wanted to keep track of all system calls and their arguments. To this end, I experimented with several approaches, but none was as successful as I would have liked. Wrapping the libc function for write(), for example, only enabled me to log write() invocations that originated from C programs, and dynamic binary instrumentation was limited by the sorts of executables the instrumentation library could parse (C, C++ and Fortran). Being limited to auditing executables produced by one of a few languages was only a small practical limitation, since virtually every program on a GNU/Linux system is written in C, C++ or some language that has a C- or C++-based runtime library, like Perl or Python. However, the incompleteness of these solutions really bothered me on a theoretical level. I knew how straightforward it would be to bypass this system by invoking a system call from a little-known language that didn't rely on C or C++, or even by handcrafting a system call in assembly language. It was clear that it would be impossible to write an insubversible user-space auditing tool, and it would be tough to write a really useful tool without hacking into the kernel. Since I didn't want to maintain a patch or deal with a lengthy recompile-reboot-debug cycle, I didn't think doing this in kernel space was feasible.

No sooner had I put these concerns on the back burner and started work on this project than I saw a message to my local LUG's mailing list that gave me an idea. This message was a forwarded advisory about a kernel module exploit. This particular module was a nasty one: it modified the behavior of certain system calls to hide itself from the lsmod command and to hide the presence of scanners, crackers, sniffer logs and other such files. I almost screamed ``Eureka!'' in my office. I didn't have to deal with maintaining a kernel patch, recompiling or rebooting; I could develop my tool as a loadable module. I recognized that the general

technique behind module exploits could be adapted to add many types of useful behavior to system calls, including a different security policy, finer-grained security than the UNIX model allows and, of course, my audit trail generator.

Hello, Kernel!

I will discuss some of the fun things you can do by altering and wrapping system calls a little later, but let us first get our hands dirty with an example kernel module. This is a simple example, akin to everyone's favorite first program, but it demonstrates the most basic parts of a loadable kernel module, the init_module and cleanup_module functions:

```
#include <linux/kernel.h>
#include <linux/module.h>
int init_module() {
   printk("<1> Hello, kernel!\n");
   return 0;
}
void cleanup_module() {
   printk("<1>I'm not offended that you"
          "unloaded me.  Have a pleasant day!\n");
}
```

You may have to use #define for the symbol MODVERSIONS and #include for the file linux/modversions.h from the Linux source tree, depending on how your system is set up. Call this short module hello.c and compile it with:

```
gcc -c -DMODULE -D__KERNEL__ hello.c
```

You should now have a file called hello.o in your current directory. If you're currently in X, switch over to a virtual console and (as root) type `insmod hello.o`. You should see ``Hello, kernel!'' on your screen. If you would like to check that your module is loaded, use the lsmod command; it should show that your hello module is loaded and taking up memory. You can now `rmmod` this module; it will politely inform you that you have unloaded it.

The linux/kernel.h and linux/module.h header files are the two most basic for any module development, and you are likely to need them for any module you write. It is best if these headers (unlike modversions.h) come from /usr/include/linux rather than a Linux source tree. (If your distribution vendor has made /usr/include/linux a link to the Linux source tree, complain--that practice is liable to cause major breakage and headaches for you.) You will use quite a few more of the kernel headers for any substantial module, and you will find that

```
grep -l /usr/include/linux
```

is a good friend while developing modules.

Think of init_module as an ``object constructor'' for your module. **init_module** should allocate storage, initialize data and alter the kernel state so that your module can do its work. In this case, init_module is merely announcing its presence and returning 0 to signify success, as in many C functions. Therefore, our initialization for the hello module consists solely of calling the printk function, a particularly handy function to have at your disposal. Essentially, it functions like the standard C printf function, but for two differences. First, and most obviously, printk allows you to specify a priority for a given message (the ``1'' in angle brackets). Second, printk sends its output to a circular buffer that is consumed by the kernel logger and (possibly) sent to syslogd. Since the output of syslog is flushed frequently, calling printk with judiciously placed, high-priority messages can greatly aid debugging--especially since any bug in kernel-space code is liable to crash your machine or at least cause a ``kernel oops''.

Why not just use printf, you ask? Simple: to do so would be impossible. The Linux kernel is not linked to the C library, so old friends like printf are unavailable in kernel-space code. However, there are many useful routines in the kernel that give you functionality similar to library routines, including workalikes for most of the str family of functions from the C library. To use these in your modules, merely include linux/string.h (be careful not to include the C library version).

If init_module is a constructor, remove_module is the destructor. Be sure to tidy up after your module as carefully as possible; if you don't free some memory or restore a data structure, you'll have to reboot to return your system to normal.

A More Interesting Module

Now we graduate to a more advanced example. Listing 1 presents a module that logs a message every time someone other than uid 0 (root) or uid 500 (me on my workstation) invokes the write system call with the word ``Linux" somewhere in the buffer. You may have to stretch a little to find a use for this module by itself, but I assure you it demonstrates several useful concepts. We are able to do this all by replacing the write system call with our own function that performs the checking and logging, and then calls write. Let's go through this example step by step.

Listing 1. Checking and Logging Function

Notice all of the include files. There sure are a lot of them, but don't despair, the ones we are going to worry about are linux/sched.h and asm/uaccess.h. The sched.h include allows you to access the current task_struct structure via the current macro, providing a great deal of useful information about the current process (see Table 1 for a list of some useful fields in task_struct), while uaccess.h provides useful macros for accessing user-space memory (more on this later).

Table 1. Useful task_struct Fields

Even these few fields in task_struct are enough to enable some really interesting modules. Should arbitrary users be allowed to su to root? You can prevent them from doing so by wrapping setuid and checking for one of several prespecified UIDs before allowing the ``real" setuid. This will allow you to develop, at the kernel level, an equivalent to the wheel group, or group of users that are allowed to su root. As an aside, the FSF has long held that the wheel group is a tool of fascist administrators (see the documentation for GNU su for more information).

Being able to audit or alter the behavior of system calls, simply on the basis of which uid invokes them, is obviously a powerful ability. It can make for good security policy to control and audit the actions of the ``nobody" user and its friends, the uucp, mail and postgres users carefully. However, an even more powerful technique is to alter behavior based on an argument. We will ignore sys_call_table and origwrite for now and proceed directly to wrapped_write, which examines both the uid of the invoking process and its buffer argument.

The first thing you should notice is that wrapped_write begins with a call to kmalloc. Why not malloc, you may ask? Remember, we're still in kernel space, and we don't have access to malloc and other standard library functions. Even if we did, calling malloc, which returns a pointer to user-space memory, would be worthless. We need to allocate some memory in kernel space to copy data into from the buf argument. This is an important point: the same memory visibility barrier between kernel and user space that keeps your programs from crashing the kernel also adds a little bit of complexity to your kernel programming. When you call write from a C program, you pass a pointer to a user-space memory block that is inaccessible from the kernel. Therefore, if you want to do any operations on data pointed to by a user-space pointer, you will have to first copy that memory area into kernel space. The copy_from_user macro does this for you.

copy_from_user takes three arguments, a ``to" pointer, a ``from" pointer and a count.

The remainder of wrapped_write is fairly straightforward, given what we know about current and task_struct. Perhaps a more interesting module would use strstr to check for the string ``Linux sucks", and if it existed, alter write_buf at that point to contain ``Linux rule", then transfer write_buf back to user space (with the copy_to_user macro) before calling the original write. Then, if unsuspecting users wrote ``Linux sucks", it would be replaced with ``Linux rules". **kfree** is important here. Leaking memory in the kernel is a bad thing, so be sure to `kfree` everything you `kmalloc`.

It is in init_module that we actually make the switch so that our function is called instead of the original write. Recall that syss_call_table is an array of pointers to functions. By altering the value at index SYS_write (a constant representing the system call number for write), we are able to cause another function to replace write. Be sure to save the original function, so you can replace it when the module is unloaded! You can test this module out by compiling and installing it with insmod; then `su` to some user other than 0 or 500, and type

```
% echo "I like Linux"
```

on a virtual console. You should get a message from the kernel that you're talking about Linux again. Congratulations! You are now ready for a module that does something useful.

A Final Example

Listing 2 [available at ftp://ftp.ssc.com/pub/lj/listings/issue89/4829.tgz] demonstrates a useful module that can help prevent your system from falling victim to stack-smashing attacks. A stack-smashing attack basically consists of writing past the end of a fixed-size buffer, so that the return address of the current function is overwritten, usually with a jump to exec (/bin/sh, ...). Since there really is no reason for programs like httpd, fingerd or wu-ftpd to **exec** a shell, we shall provide a mechanism to disallow it. By this point, you already have the knowledge to understand most of the code, with one small exception: the strncpy_from_user function. As you might expect, it functions much like its C-library counterpart, strncpy, and is a handy way to get a null-terminated string from user space. Since the code is straightforward, we'll briefly discuss the approach, and then I'll leave you to come up with great ideas of your own for improving your system's security.

The implementation in Listing 2 is straightforward. It is not as efficient or robust as one might want, but this code was written in the interest of clarity, and it is easy work to make it better by changing the linear search in wrapped_execve to something more efficient. Essentially, what this module does is overload the kill system call so that if you send signal 42 to a process; it is added to a list of ``unsafe" processes, processes that should not be allowed to execute any binary with ``sh" in its filename. (42 is one of the real-time signals; you probably aren't using it. If you are, feel free to substitute any number between 32 and 64.) The execve system call then checks to see whether the process is an unsafe one and, if so, checks to see if it is trying to execute a shell. If so, it returns success without doing anything. It is easy to use this module for all of your server processes; simply add a

```
kill -42 ...
```

to your init scripts.

Listing 2 represents an evolutionary step from Listing 1, but it shows that one can modify the behavior of calls, not just add behavior to the call path. It also does useful work. I hope that you are as excited as I am about the possibilities of writing kernel module exploits to improve your security. This article has given you the basic tools to get started. Fortunately, there is a wealth of documentation available to Linux programmers that will help you write more complex and functional modules; see the Resources section.

Resources



William C. Benton *(willb@acm.org) is a graduate student at the University of Wisconsin. He is interested in performance monitoring of parallel programs and in language and compiler approaches to security problems.*

Magazine Table of Contents