Hijacking Linux Syscalls & Writing a Linux Keylogger: A Case Study

by: Frank Stratton

original at: http://www.epanastasi.com/docs/syscall_talk.php

BACKGROUND
---

*User Space vs Kernel Space.
Linux is divided into two distinct modes of execution. User Space (aka User Mode) is where your everyday applications run (like Firefox, Apache, SSH, etc). User Space programs run in an unprivileged mode (from the operating system point of view). User Space applications usually do not have direct access to hardware or other critical sections of the system (packet handling, paging, interrupts, disk i/o, etc). The Kernel, however, is responsible for managing all these low-level operations. Kernel code runs in a privileged mode, made possible by hardware. On x86 the levels of privileged execution are called 'rings' with 'ring 0' used for the kernel and 'ring 3' used for user space.

Note: do not confuse the privileges of root with that of the kernel! Normal programs running as root do not run in the kernel.*

When you write a user space program that needs to use a particular resource, you must issue a request with the kernel. The typical way of doing this is via a system call.

*What is a syscall?

```
$man syscalls

The system call is the fundamental interface between an application and the
Linux kernel. As of Linux 2.4.17, there are 1100 system calls listed in
/usr/src/linux/include/asm-*/unistd.h.
```

Some of the most common syscalls are open, close, read, write, execve, fork, kill, and wait. A very useful tool to examine what syscalls an application makes is called strace. Here is the output of `strace ls` in a directory with three files: evil_file, good_file, good_file_number_2

```
$strace ls
execve("/bin/ls", ["ls"], [/* 17 vars */]) = 0
uname({sys="Linux", node="yt", ...})    = 0
brk(0)                                  = 0x805b000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f79000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=101364, ...}) = 0
old_mmap(NULL, 101364, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f60000
close(3)                                = 0

...

getdents64(3, /* 5 entries */, 4096)    = 152
getdents64(3, /* 0 entries */, 4096)    = 0
close(3)                                = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 9), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f78000
write(1, "evil_file  good_file  good_file_"..., 41evil_file  good_file  good_file_number_2
) = 41
munmap(0xb7f78000, 4096)                = 0
exit_group(0)                           = ?
```

*How syscalls work in the linux kerenel.

When a syscall is issued the application is interrupted and control is given to the kernel. In linux this is done by the instruction int 0x80 (some systems support the SYSENTER instruction to reduce the overhead of int 0x80). When the instruction int 0x80 is reached the kernel executes a special function: _system_call(). This function takes the syscall number and uses it to index into a huge table, the sys_call_table[] in fact, to find the address of the specific function that needs to be called.

REPLACING SYSCALLS IN 2.4

In the 2.4 kernel series, replacing syscalls is an easy matter. The sys_call_table[] is a globally exported symbol, meaning that you can declare and use it as follows:

```
extern void *sys_call_table[];
original_sys_open = sys_call_table[__NR_open];
sys_call_table[__NR_open] = new_sys_open;

...
asmlinkage int new_sys_open(const char *filename, int flags, int mode)
{
        printk(KERN_ALERT "CALLING SYS_OPEN");

        // call the original_sys_open
        ret = (*original_sys_open)(filename, flags, mode);

        return ret;
}
```

In the above example we simply patch the sys_open call to print a small message whenever it's used and then passes execution on to the original sys_open. We'll see later on some more creative uses for this...

(see [7] for if you want to know more about asmlinkage)

REPLACING SYSCALLS IN 2.6

The 2.6 kernel series is a different beast. There have been a number of changes as far as syscall hijackers are concerned about. The biggest change is the sys_call_table[] is no longer exported. Another problematic change is that some systems (Fedora for sure) include a kernel patch (write-protect-rodata) that make the sys_call_table memory page read-only and you get some ugly paging errors when you try to overwrite an entry.. Thus, we must invoke powerful magic to find and replace the correct syscall entry.

Before I detail exactly what magic we must use to find the sys_call_table[], I will first choose describe the three possible* entry points.

*From Userspace using /dev/kmem (defunct?)
The device /dev/kmem is basically a current snapshot of kernel memory. Since everything in Linux is a file(*) you can write a userspace program that can open/read/write to anywhere in this file and it will be as if you are playing around with the kernel's memory space directly. You must be root to access this /dev/kmem, and even if you change the permissions on the file there are other protections higher in the food chain that keep you from fiddling with all the bits...

It should be noted that /dev/kmem was removed from many a system, (including the one I work with) so no guarentees that it will be around for you to play with.

*From Userspace using /dev/mem (untested)

Not to fret if you do not have /dev/kmem, a similar device /dev/mem exists that maps the entirety of physical memory. The methods we discuss to find and replace the sys_call_table[] should work (mostly) for people using userspace /dev/mem access. Many Linux rootkits use this method to hijack kernel functions, and it can be used as a workaround for kernels that do not have module support. There are some necessary extra hacks that need to be involved, one of them is to find the address of kmalloc() to allocate kernel memory for your extra code so that your hijacked functions run in the correct context. This is described in [5].

*From Kernelspace via kernel module
This is the method I chose to implement as it requires less work, has acesss to necessary kernel functions, and it a way to learn kernel module programming. Now on to the good stuff...

Our first goal is to locate the address of the sys_call_table[]. If you remember back to our discussion on how syscalls work, you'll remember our friend int 0x80 . The hack that we're going to make goes like this:

1) Find the interrupt descriptor table
2) Find the entry for int 0x80 .
3) Get the location for the system_call() function.
4) Search through memory looking for the sys_call_table[] within this function.

Now a more detailed look at how to do this:

1) The address of the idt is held in a special register (idtr) there is an assembly function that can be used to obtain that address.

```
struct {
        unsigned short limit;
        unsigned int base;
} __attribute__ ((packed)) idtr;

/* ask the processor for the idt address and store it in idtr */
asm ("sidt %0" : "=m" (idtr));
```

2 & 3) Now we need to find the int 0x80 entry in the idt. Each entry in the idt is 8 bytes (hence the base+8*0x80) and the magic shifting of bits to reconstruct the location of system_call() used below.

```
struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;

unsigned sys_call_off;

/* read in IDT for int 0x80 (syscall) */
memcpy(&idt, idtr.base+8*0x80,sizeof(idt));
sys_call_off = (idt.off2 << 16) | idt.off1;
```

4) What we do now is the greatest leap of faith. If we look at the system_call() function in detail:

```
$ gdb -q /usr/src/linux/vmlinux
(no debugging symbols found)...(gdb) disass system_call
Dump of assembler code for function system_call:
0xc0106bc8 :       push   %eax
0xc0106bc9 :     cld
0xc0106bca :     push   %es
```

```
0xc0106bcb :     push   %ds
0xc0106bcc :     push   %eax
0xc0106bcd :     push   %ebp
0xc0106bce :     push   %edi
0xc0106bcf :     push   %esi
0xc0106bd0 :     push   %edx
0xc0106bd1 :     push   %ecx
0xc0106bd2 :    push   %ebx
0xc0106bd3 :    mov    $0x18,%edx
0xc0106bd8 :    mov    %edx,%ds
0xc0106bda :    mov    %edx,%es
0xc0106bdc :    mov    $0xffffe000,%ebx
0xc0106be1 :    and    %esp,%ebx
0xc0106be3 :    cmp    $0x100,%eax
0xc0106be8 :    jae    0xc0106c75
0xc0106bee :    testb  $0x2,0x18(%ebx)
0xc0106bf2 :    jne    0xc0106c48
0xc0106bf4 :    call   *0xc01e0f18(,%eax,4) <-- that's it
0xc0106bfb :    mov    %eax,0x18(%esp,1)
0xc0106bff :    nop
End of assembler dump.
(gdb) print &sys_call_table
$1 = ( *) 0xc01e0f18      <-- see ? it's same
(gdb) x/xw (system_call+44)
0xc0106bf4 :    0x188514ff <-- opcode (little endian)
(gdb)
```

What we see from the above output[5] is the actual call to a specific syscall. The line marked '<-- that's it' shows that call <address>(,%eax,4) where <address> matches that of the sys_call_table (seen a few lines later). To find the address of the sys_call_table we'll start at the system_call() function and inspect memory until we find something that looks like 'call <address>(,%eax,4)' and assume that <address> is the sys_call_table. The opcode for call is shown at the end of the above output. Thus, our code becomes:

```
char *p;
unsigned sys_call_table;

p = (char*)memmem (sc_asm,CALLOFF,"\xff\x14\x85",3);
sys_call_table = *(unsigned*)(p+3);
```

And there we have it, sys_call_table now holds the correct address and we can go about replacing syscalls... after we fix the page permissions thing.

To change the permissions on a kernel page you can use the following method:

```
unsigned *sct;
struct page *pg;
pgprot_t prot;

// Get sys_call_table (as explained above)
find_sys_call_table(&sct);

// fix kernel perms
pg = virt_to_page(sct);
prot.pgprot = VM_READ | VM_WRITE | VM_EXEC; /* R-W-X */
change_page_attr(pg,1,prot);
```

virt_to_page() takes a virtual address and returns a pointer to the page to which that address belongs. change_page_attr() takes a starting page, number of pages, and new permissions. Simple, yes?

After all that work we can now do what we did in 2.4 and replace syscalls to our heart's content.

It should also be noted that the address of the sys_call_table[] and each individual syscalls (sys_open, etc) are located in the System.map file that is created for your kernel version. See code/example1-sct/README for more information.

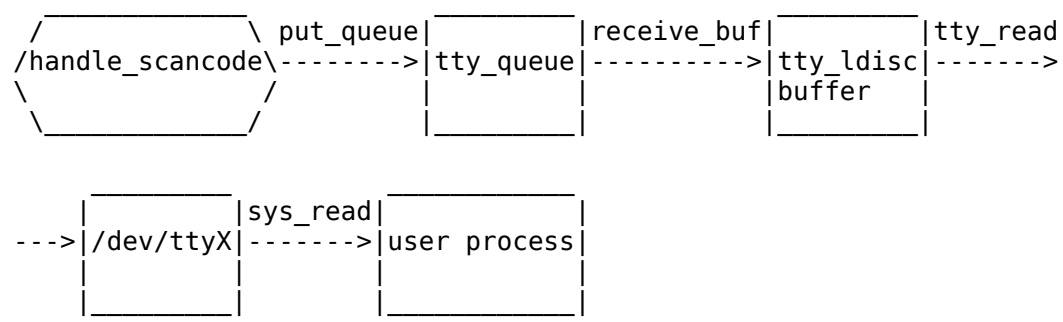
One final note on finding the sys_call_table[] in 2.6: if the system you are on does not support int 0x80 and instead makes use of sysenter, it is unknown whether the idt method will work. There is another similar hack to finding the syscall table [6] that looks like this:

```
for (ptr = (unsigned long) &loops_per_jiffy;
        ptr < (unsigned long) &boot_cpu_data;
    ptr += sizeof(void *))
{
        unsigned long *p;
        p = (unsigned long *)ptr;

        if (p[__NR_close] == (unsigned long) sys_close) {
                sct = (unsigned long **)p;
                break;
        }
}
```

WRITING A KEYLOGGER FOR LINUX

This is a crude diagram stolen to make this talk a little shorter. For a detailed look at this and other methods regarding how to write a linux keylogger see [4].



This shows the various steps taken when a key is pressed up to the point when it reaches the user. Any function in this chain can be hijacked (in various ways) and offer different levels of work/payouts. For instance, the simplest way to go about it is to hijack sys_read or sys_write and check to see if it is going or coming to /dev/ttyX and record keys that way. The other extreme would be to forget syscalls all together and write your own keyboard interrupt handler. Each has it's advantages and drawbacks. The middle ground approach lies in the receive_buf() function. receive_buf() is burried within special kernel structures and there is a function defined for each tty. Meaning, when a new tty (or pts in fact) is opened, a brand new struct_tty (and thus receive_buf()) is allocated. This means we will need to hijack the sys_open call and replace the receive_buf() function there. Going this route gives us some extra benifits, the biggest is the ability to log local and remote sessions, as well as being able to split incomming logs into their respective ttys instead of one giant mass of characters. It can also be used for so called 'smart modes' where you can detect when TTY_ECHO is turned off and log only user/passwords. For greater explaination see [4] as well as example code attached with this document.

REFERENCES

---

[1] "How System Calls Work on Linux/i86"
http://www.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

[2] System Call Optimization with the SYSENTER Instruction
http://www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8223/

[3] Introduction to UNIX Assembly Programming
http://linuxgazette.net/issue53/boldyshev.html

[4] Writing Linux Kernel Keylogger, Volume 0x0b, Issue 0x3b, Phile #0x0e of 0x12
http://www.phrack.org/show.php?p=59&a=14

[5] Linux on-the-fly kernel patching without LKM, Volume 0x0b, Issue 0x3a,
Phile #0x07 of 0x0e
http://www.phrack.org/show.php?p=58&a=7

[6] Local Honeypot Identification, Volume 0x0b, Issue 0x3e, Phile #0x07 of
0x0f
http://www.phrack.org/unoffical/p62/p62-0x07.txt

[7] What is asmlinkage?
http://www.kernelnewbies.org/faq/