1 print_hex_dump

由于boot_command_line和saved_command_line没有导出，普通程序没有权限访问，因此修改init/main.c导出符号：

```
EXPORT_SYMBOL(boot_command_line);
EXPORT_SYMBOL(saved_command_line);
```

重新编译安装内核。

编写dump模块，在其中使用print_hex_dump打印command_line的信息，只用KERN_INFO级别就可以在dmesg中查看。由于不知道这两个数组的长度，因此编写了一个my_strlen计算。

```
/* dump.c */
#include <linux/init.h>
#include <linux/printk.h>
#include <linux/kernel.h>
#include <linux/module.h>

extern char boot_command_line[];
extern char *saved_command_line;

int my_strlen(char *s)
{
  int i = 0;
  while (s[i++]);
  return i - 1;
}

static int __init dump_init(void)
{
    print_hex_dump(KERN_INFO, "boot_command_line: ", DUMP_PREFIX_NONE, 16, 1,
boot_command_line, my_strlen(boot_command_line), 1);
    print_hex_dump(KERN_INFO, "saved_command_line: ", DUMP_PREFIX_NONE, 16, 1,
saved_command_line, my_strlen(saved_command_line), 1);
    return 0;
}

static void __exit dump_exit(void)
{
    return;
}

module_init(dump_init);
module_exit(dump_exit);
```

得到的结果如下：

```
[ 1554.618976] boot_command_line: 6c 74 5f 74 61 72 67 65 74
lt_target
[ 1554.619689] saved_command_line: 42 4f 4f 54 5f 49 4d 41 47 45 3d 2f 62 6f 6f 74
BOOT_IMAGE=/boot
[ 1554.620258] saved_command_line: 2f 76 6d 6c 69 6e 75 7a 2d 34 2e 31 39 2e 30 2b
/vmlinuz-4.19.0+
[ 1554.620828] saved_command_line: 20 72 6f 6f 74 3d 55 55 49 44 3d 63 64 37 62 35
root=UUID=cd7b5
[ 1554.621371] saved_command_line: 37 65 36 2d 63 35 38 37 2d 31 31 65 38 2d 39 66
7e6-c587-11e8-9f
[ 1554.621925] saved_command_line: 31 63 2d 30 38 30 30 32 37 39 61 38 66 66 61 20  1c-
0800279a8ffa
[ 1554.622568] saved_command_line: 72 6f 20 6d 61 79 62 65 2d 75 62 69 71 75 69 74  ro
maybe-ubiquit
[ 1554.623151] saved_command_line: 79                                               y
```

2 SysRQ

开启SysRQ服务

```
echo 1 >/proc/sys/kernel/sysrq
```

测试按键组合，以下面几个组合为例：

```
# alt + prtsc + s
lzzz@ubuntu:~$ [] sysrq: SysRq : Emeregncy Sync
# alt + prtsc + u
lzzz@ubuntu:~$ [] sysrq: SysRq : Emeregncy Remount R/O
# alt + prtsc + b
reboot...
# alt + prtsc + w
lzzz@ubuntu:~$ [] sysrq: SysRq : Show Blocked State
# alt + prtsc + z
lzzz@ubuntu:~$ [] sysrq: SysRq : Dump ftrace buffer
[] Dumping ftrace buffer:
[]      (ftrace buffer empty)
# alt + prtsc + p
lzzz@ubuntu:~$ [] sysrq: SysRq : Show Regs
# alt + prtsc + r
lzzz@ubuntu:~$ [] sysrq: SysRq : Keyboard mode set to system default
```

3 dump_stack

在1中的dump_init中加入 dump_stack(); 结果如下：

```
[ 2128.509727] CPU: 3 PID: 5444 Comm: insmod Tainted: P         C OE     4.19.0+ #4
```

```
[ 2128.510422] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox
12/01/2006
[ 2128.510929] Call Trace:
[ 2128.511546]  dump_stack+0x63/0x85
[ 2128.512028]  ? 0xffffffffc0872000
[ 2128.512521]  dump_init+0xe/0x1000 [dump]
[ 2128.512995]  do_one_initcall+0x4a/0x1c9
[ 2128.513484]  ? _cond_resched+0x19/0x40
[ 2128.513960]  ? kmem_cache_alloc_trace+0x15c/0x1d0
[ 2128.514498]  do_init_module+0x5f/0x206
[ 2128.514972]  load_module+0x2213/0x2b00
[ 2128.515472]  __do_sys_finit_module+0xfc/0x120
[ 2128.515958]  ? __do_sys_finit_module+0xfc/0x120
[ 2128.516457]  __x64_sys_finit_module+0x1a/0x20
[ 2128.516923]  do_syscall_64+0x5a/0x120
[ 2128.517405]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
[ 2128.517868] RIP: 0033:0x7f6d8ac56839
[ 2128.518406] Code: 00 f3 c3 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 40 00 48 89 f8 48 89
f7 48 89 d6 48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d 01 f0 ff ff 73 01
c3 48 8b 0d 1f f6 2c 00 f7 d8 64 89 01 48
[ 2128.519887] RSP: 002b:00007ffe9082abe8 EFLAGS: 00000246 ORIG_RAX: 0000000000000139
[ 2128.520418] RAX: ffffffffffffffda RBX: 000055f041ae2770 RCX: 00007f6d8ac56839
[ 2128.520929] RDX: 0000000000000000 RSI: 000055f040370c2e RDI: 0000000000000003
[ 2128.521458] RBP: 000055f040370c2e R08: 0000000000000000 R09: 00007f6d8af29000
[ 2128.521976] R10: 0000000000000003 R11: 0000000000000246 R12: 0000000000000000
[ 2128.522564] R13: 000055f041ae2750 R14: 0000000000000000 R15: 0000000000000000
[ 2128.523179] boot_command_line: 6c 74 5f 74 61 72 67 65 74
lt_target
[ 2128.523750] saved_command_line: 42 4f 4f 54 5f 49 4d 41 47 45 3d 2f 62 6f 6f 74
BOOT_IMAGE=/boot
[ 2128.524408] saved_command_line: 2f 76 6d 6c 69 6e 75 7a 2d 34 2e 31 39 2e 30 2b
/vmlinuz-4.19.0+
[ 2128.525100] saved_command_line: 20 72 6f 6f 74 3d 55 55 49 44 3d 63 64 37 62 35
root=UUID=cd7b5
[ 2128.525670] saved_command_line: 37 65 36 2d 63 35 38 37 2d 31 31 65 38 2d 39 66
7e6-c587-11e8-9f
[ 2128.526446] saved_command_line: 31 63 2d 30 38 30 30 32 37 39 61 38 66 66 61 20  1c-
0800279a8ffa
[ 2128.527027] saved_command_line: 72 6f 20 6d 61 79 62 65 2d 75 62 69 71 75 69 74  ro
maybe-ubiquit
[ 2128.527575] saved_command_line: 79
```

4 syslog-counter

为了统计sys_mkdir调用的次数，修改Lab05中的syscall.c，定义全局变量mkdir_count，截获mkdir系统调用后，将计数器mkdir_count加一，然后调用原来的系统调用。

```
unsigned int mkdir_count;

asmlinkage int our_sys_mkdir(const char *filename, int flags, int mode)
{
    ++mkdir_count;
    return original_call(filename, flags, mode);
}
```

这样，劫持模块会统计sys_mkdir的调用次数，但是运行于用户态的程序无法获得这个变量。

于是再通过自己编写的系统效用进入内核态，获取counter的值。

在syscall.c中将mkdir_count导出：

```
EXPORT_SYMBOL(mkdir_count);
```

另外编写一个新的模块，用 extern unsigned int mkdir_count; 获取syscall.c中的counter，用相似的方法修改系统调用表，将233号系统调用修改为自己定义的函数，然后返回counter的值。

```
asmlinkage int print_mkdir_count(void)
{
    return mkdir_count;
}
int init_module()
{

    cr0 = read_cr0();
    write_cr0(cr- & ~CR0_WP);

    original_call = sys_call_table[233];
    sys_call_table[233] = print_mkdir_count;

    write_cr0(cr0);

    return 0;
}
```

这样，在用户代码中就可以通过233号系统调用获得counter的值。

监视系统调用：

```
#include <linux/unistd.h>
#include <syscall.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <syslog.h>

int main()
{
    unsigned int c0 = syscall(233);
    unsigned int c1 = c0;
```

```
    unsigned int c2;
    syslog(LOG_KERN, "Initial count: %d.\n", c0);
    while (1) {
        sleep(60);
        c2 = syscall(233);
        openlog("mkdir_count_log", LOG_KERN, 0);
        syslog(LOG_USER | LOG_DEBUG, "%d sys_mkdir calls in last one minute. %d calls
in all.\n", c2 - c1, c2 - c0);
        closelog();
        c1 = c2;
    }
    return 0;
}
```

insert module插入两个模块之后，运行这个程序，在/var/log/syslog中可以看到结果。（为了演示将60s间隔设为了0.06s）

```
    0 Oct 29 16:13:49 ubuntu a.out: Initial count: 0.
    1 Oct 29 16:13:49 ubuntu mkdir_count_log: 0 sys_mkdir calls in last one minute. 0
calls in all.
    2 Oct 29 16:14:00 ubuntu mkdir_count_log: message repeated 66603 times: [ 0
sys_mkdir calls in last one minute. 0 calls in all.]
    3 Oct 29 16:14:00 ubuntu mkdir_count_log: 1 sys_mkdir calls in last one minute. 1
calls in all.
    4 Oct 29 16:14:00 ubuntu mkdir_count_log: 0 sys_mkdir calls in last one minute. 1
calls in all.
    5 Oct 29 16:14:20 ubuntu mkdir_count_log: message repeated 113647 times: [ 0
sys_mkdir calls in last one minute. 1 calls in all.]
    6 Oct 29 16:14:20 ubuntu mkdir_count_log: 1 sys_mkdir calls in last one minute. 2
calls in all.
    7 Oct 29 16:14:20 ubuntu mkdir_count_log: 0 sys_mkdir calls in last one minute. 2
calls in all.
    8 Oct 29 16:14:24 ubuntu mkdir_count_log: message repeated 18025 times: [ 0
sys_mkdir calls in last one minute. 2 calls in all.]
    9 Oct 29 16:14:24 ubuntu mkdir_count_log: 1 sys_mkdir calls in last one minute. 3
calls in all.
   10 Oct 29 16:14:24 ubuntu mkdir_count_log: 0 sys_mkdir calls in last one minute. 3
calls in all.
```