Lab06 Debug

李拙

1

- 用print_hex_dump打印 boot_command_line 和 saved_command_line 的地址空间内容

1.1 print_hex_dump

- 将二进制内容以16进制打印，并可选打印ascii字符
- 主要参数:
  - `const char *level, const char *prefix_str`
  - `const void *buf, size_t len, bool ascii`

1.1 print_hex_dump

```
* E.g.:
*   print_hex_dump(KERN_DEBUG, "raw data: ", DUMP_PREFIX_ADDRESS,
*                  16, 1, frame->data, frame->len, 1);
*
* Example output using %DUMP_PREFIX_OFFSET and 1-byte mode:
* 0009ab42: 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f  @ABCDEFGHIJKLMNO
```

1.2 command_line

- cmdline是内核启动的参数配置字符串
- 格式为“value1, param=value2”
- 在start_kernel()中处理
- 定义在init/main.c

1.2 command_line

- char __initdata boot_command_line[COMMAND_LINE_SIZE];
- char *saved_command_line;
- bootcmdline定义时加上宏__initdata，链接时会放在.init.data段，init阶段之后释放
- savedcmdline作用是保存未改动的cmdline

1.3 导出cmdline

- bootcmdline 和 savedcmdline都没有导出
- 修改源码:
  - EXPORT_SYMBOL(boot_command_line);
  - EXPORT_SYMBOL(saved_command_line);
- 重新编译、替换内核
- (导出的符号处于内核态)

1.4 编写模块

- 不知道要打印的长度
- 自己写了一个strlen…

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

1.4 编写模块

- 注：print_hex_dump要 #include <linux/printk.h>

```
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;
}
```

1.5 result

```
[ 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

- SysRQ is a ‘magical’ key combo you can hit which the kernel will respond to regardless of whatever else if is doing, unless it is completely locked up.
- https://www.mjmwired.net/kernel/Documentation/sysrq.txt

2.1 开启SysRQ

- 编译内核时需设置 CONFIG_MAGIC_SYSRQ=Y
- 启动内核后，修改/proc/sys/kernel/sysrq

```
 0 - disable sysrq completely
 1 - enable all functions of sysrq
>1 - bitmask of allowed sysrq functions (see below for detailed function
     description):
         2 =   0x2 - enable control of console logging level
         4 =   0x4 - enable control of keyboard (SAK, unraw)
         8 =   0x8 - enable debugging dumps of processes etc.
        16 =  0x10 - enable sync command
        32 =  0x20 - enable remount read-only
        64 =  0x40 - enable signalling of processes (term, kill, oom-kill)
       128 =  0x80 - allow reboot/poweroff
       256 = 0x100 - allow nicing of all RT tasks
```

2.2 use SysRq key

- x86: ALT+PrtSc+<command key>
- test: ALT+PrtSc+p – SysRq: Show Regs

2.2 use SysRq key

```
[ 2806.675327] sysrq: SysRq : Show Regs
[ 2806.676018] CPU: 2 PID: 0 Comm: swapper/2 Tainted: P         C OE     4.19.0+ #4
[ 2806.676020] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 2806.676028] RIP: 0010:native_safe_halt+0x6/0x10
[ 2806.676033] Code: 71 ff ff ff 7f 5d c3 65 48 8b 04 25 00 5c 01 00 f0 80 48 02 20 48 8b 00 a8 08 74 8b eb c1 90 90 90 90 90 90 55 48 8
9 e5 fb f4 <5d> c3 0f 1f 84 00 00 00 00 00 55 48 89 e5 f4 5d c3 90 90 90 90 90
[ 2806.676035] RSP: 0018:ffffac9a0069be70 EFLAGS: 00000246 ORIG_RAX: ffffffffffffffdd
[ 2806.676038] RAX: ffffffff8ea14620 RBX: 0000000000000002 RCX: 0000000000000001
[ 2806.676040] RDX: ffff8e1f90d23940 RSI: 0000000000000087 RDI: 0000000000000002
[ 2806.676042] RBP: ffffac9a0069be70 R08: 000000000001ca80 R09: 0000000000000020
[ 2806.676043] R10: 0000000000000004 R11: 0000000000000000 R12: 0000000000000002
[ 2806.676044] R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
[ 2806.676054] FS:  0000000000000000(0000) GS:ffff8e1f90d00000(0000) knlGS:0000000000000000
[ 2806.676056] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 2806.676057] CR2: 00007f59dbfb21d0 CR3: 00000000a9348000 CR4: 00000000000406e0
[ 2806.676062] Call Trace:
[ 2806.676071]  default_idle+0x22/0x150
[ 2806.676076]  arch_cpu_idle+0x15/0x20
[ 2806.676078]  default_idle_call+0x23/0x30
[ 2806.676083]  do_idle+0x1cb/0x280
[ 2806.676086]  cpu_startup_entry+0x1d/0x20
[ 2806.676090]  start_secondary+0x1ab/0x200
[ 2806.676095]  secondary_startup_64+0xa4/0xb0
[ 2806.676098] CPU#2: active:     0000000000000000
[ 2806.676107] CPU#2:   gen-PMC0 ctrl:  0000000000000000
[ 2806.676108] CPU#2:   gen-PMC0 count: 0000000000000000
[ 2806.676110] CPU#2:   gen-PMC0 left:  0000000000000000
[ 2806.676116] CPU#2:   gen-PMC1 ctrl:  0000000000000000
[ 2806.676117] CPU#2:   gen-PMC1 count: 0000000000000000
[ 2806.676118] CPU#2:   gen-PMC1 left:  0000000000000000
[ 2806.676124] CPU#2:   gen-PMC2 ctrl:  0000000000000000
[ 2806.676126] CPU#2:   gen-PMC2 count: 0000000000000000
[ 2806.676127] CPU#2:   gen-PMC2 left:  0000000000000000
[ 2806.676133] CPU#2:   gen-PMC3 ctrl:  0000000000000000
[ 2806.676134] CPU#2:   gen-PMC3 count: 0000000000000000
[ 2806.676135] CPU#2:   gen-PMC3 left:  0000000000000000
```

3 dump_stack

- 打印内核调用栈
- 从栈顶开始查找，判断地址是否在内核代码段

```
for (; stack < top; stack += 4) {
        addr = *(unsigned int *)stack;
        if (kernel_text_addressp(addr))
                printk(" [<%p>] %pS\n", (void *)addr, (void *)addr);
}
```

3.1 1中的call_stack

- 在dump_init()中调用call_stack();

```
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;
}
```

3.1 1中的call_stack

- 在dump_init()中调用call_stack();

```
[ 3494.130061] Call Trace:
[ 3494.130069]  dump_stack+0x63/0x85
[ 3494.130071]  ? 0xffffffffc072e000
[ 3494.130074]  dump_init+0xe/0x1000 [dump]
[ 3494.130077]  do_one_initcall+0x4a/0x1c9
[ 3494.130080]  ? _cond_resched+0x19/0x40
[ 3494.130083]  ? kmem_cache_alloc_trace+0x15c/0x1d0
[ 3494.130086]  do_init_module+0x5f/0x206
[ 3494.130088]  load_module+0x2213/0x2b00
[ 3494.130091]  __do_sys_finit_module+0xfc/0x120
[ 3494.130092]  ? __do_sys_finit_module+0xfc/0x120
[ 3494.130094]  __x64_sys_finit_module+0x1a/0x20
[ 3494.130095]  do_syscall_64+0x5a/0x120
[ 3494.130097]  entry_SYSCALL_64_after_hwframe+0x44/0xa9
```

4 syslog

- Syslog is a standard for message logging.
- It allows separation of the software that generates messages.

- See linux manual:
  - http://man7.org/linux/man-pages/man3/syslog.3.html

4.1 截获系统调用并计数

- 按照Lab05的方式截获系统调用
- 修改源码导出sys_call_table
- 重新编译、加载内核

```
unsigned int mkdir_count;

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

- 用户程序如何获取该变量?

4.2 内核空间与用户空间

- 编写系统调用
- 编写驱动程序
- 使用proc文件系统
- 使用虚拟文件系统

4.3 修改系统调用

- 与劫持系统调用的方法相同
- 加入内核模块，修改系统调用表

4.3 修改系统调用

```
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;
}
```

4.3 修改系统调用

- 劫持系统调用的模块A计数 EXPORT_SYMBOL(mkdir_count);
- and
- 用户调用系统调用B返回mkdir_count
  - extern unsigned int mkdir_count;

4.4 用户程序

```c
#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;
}

```

4.5 查看结果

- vim /var/log/syslog

```
    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.
```

Thanks.