编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： Linux架构分析与安全设计

实验内容 ： 实验4 Linux Kernel Hook

专业(班) ：

学 号 ：

姓 名 ：

任课教师 ： 王鹃

2020 年 11 月 19 日

目 录

[实验四：Linux Kernel Hook 3](#_Toc59007590)

[1. 实验目的 3](#_Toc59007591)

[2. 实验环境 3](#_Toc59007592)

[3. 实验原理 3](#_Toc59007593)

[第一部分：基于修改 sys\_call\_table 的系统调用挂钩 3](#_Toc59007594)

[第二部分：基于系统调用挂钩的初级文件监视 6](#_Toc59007595)

[4. 实验要求 6](#_Toc59007596)

[5. 实验内容和步骤 7](#_Toc59007597)

[6. 实验问题 13](#_Toc59007598)

[7. 参考资料 14](#_Toc59007599)

# 实验四：Linux Kernel Hook

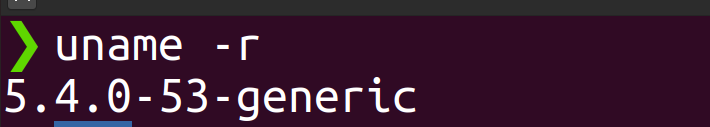
## 实验目的

在ubuntu下实现内核HOOK功能，观察HOOK的系统调用。

## 实验环境

（1）操作系统：Ubuntu 64位

（2）内核版本



## 实验原理

Rootkit的一种经典形式是通过Hook系统调用实现。在本次实验中，我们将实现简单的系统调用挂钩方案，并且基于这个方案实现最基本的文件监视工具，同时加深对LKM的理解。

### 第一部分：基于修改 [sys\_call\_table](http://lxr.free-electrons.com/ident?i=sys_call_table) 的系统调用挂钩

在系统调用挂钩技术中，最简单、最流行的方案是修改sys\_call\_table， 成员类型为函数指针的一维数组。

asmlinkage const sys\_call\_ptr\_t sys\_call\_table[\_\_NR\_syscall\_max+1] = {

 /\*

*\* Smells like a compiler bug -- it doesn't work*

*\* when the & below is removed.*

\*/

 [0 ... \_\_NR\_syscall\_max] = &sys\_ni\_syscall,

#include <asm/syscalls\_64.h>

};

要修改它，首先得拿到它在内存里的位置。然后，由于sys\_call\_table所在的内存是有写保护的，所以我们需要先去掉写保护，再做修改。

#### 1. 获得 sys\_call\_table 的内存地址

获取sys\_call\_table的方法有如下几种：

1. 从***/boot/System.map*** 中读取，可以查阅 [Hooking the Linux System CallTable](https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/)， 这篇文章便是使用这种方案来获取sys\_call\_table的地址的。
2. 从使用了sys\_call\_table的某些未导出函数的机器码里面进行特征搜索， 可以查阅[Kernel-LandRootkits](http://www.kernelhacking.com/rodrigo/docs/StMichael/kernel-land-rootkits.pdf)，作者阐述了如何从导出的函数中获取使用了sys\_call\_table的未导出函数，进而搜索那个未导出函数的机器码， 得到sys\_call\_table的地址；

本实验中采用直接暴力地址搜索的方法，查找sys\_call\_table的地址。代码如下：

unsigned long \*\*

get\_sys\_call\_table(void)

{

  unsigned long \*\*entry = (unsigned long \*\*)PAGE\_OFFSET;

  for (;(unsigned long)entry < ULONG\_MAX; entry += 1) {

    if (entry[\_\_NR\_close] == (unsigned long \*)sys\_close) {

        return entry;

      }

  }

  return NULL;

}

[PAGE\_OFFSET](http://lxr.free-electrons.com/ident?i=PAGE_OFFSET)是内核内存空间的起始地址。因为[sys\_close](http://lxr.free-electrons.com/ident?i=sys_close)是导出函数（需要指出的是， sys\_open 、 sys\_read 等并不是导出的），我们可以直接得到他的地址；因为系统调用号（也就是sys\_call\_table这个一维数组的索引）在同一[ABI](https://en.wikipedia.org/wiki/Application_binary_interface) （x86跟 x64 不是同一 ABI）上具有高度的后向兼容性，更重要的是，我们可以直接使用这个索引（代码中的 \_\_NR\_close ）！

从内核内存的起始地址开始， 逐一尝试每一个指针大小的内存：把它当成是sys\_call\_table的地址，用某个系统调用的编号（也就是索引）访问数组中的成员，如果访问得到的值刚好是是这个系统调用号所对应的系统调用的地址，那么我们就认为当前尝试的这块指针大小的内存就是我们要找的sys\_call\_table的地址。

#### 2. 关闭写保护

写保护指的是写入只读内存时出错。这个特性可以通过CR0寄存器控制：开启或者关闭，只需要修改一个比特，也就是从 0 开始数的第 16个比特。我们可以使用read\_cr0 /write\_cr0 来读取 /写入 CR0 寄存器，免去我们自己写内联汇编的麻烦。函数原型如下：

static inline unsigned long read\_cr0(void);

static inline void write\_cr0(unsigned long x);

关闭写保护的源代码：将[CR0](https://en.wikipedia.org/wiki/Control_register" \l "CR0) 寄存器从0开始数的第16个比特置为0。

void

disable\_write\_protection(void)

{

  unsigned long cr0 = read\_cr0();

  clear\_bit(16, &cr0);

  write\_cr0(cr0);

}

开启写保护的源代码：将[CR0](https://en.wikipedia.org/wiki/Control_register" \l "CR0)寄存器从0开始数的第16个比特置为1。

void

enable\_write\_protection(void)

{

  unsigned long cr0 = read\_cr0();

  set\_bit(16, &cr0);

  write\_cr0(cr0);

}

在设置或者清除某个比特,使用了[set\_bit](https://www.kernel.org/doc/htmldocs/kernel-api/API-set-bit.html)与[clear\_bit](https://www.kernel.org/doc/htmldocs/kernel-api/API-clear-bit.html)。它们是 Linux 内核提供给内核模块使用的编程接口，简单易懂。函数原型如下：

static \_\_always\_inline void

set\_bit(long nr, volatile unsigned long \*addr);

static \_\_always\_inline void

clear\_bit(long nr, volatile unsigned long \*addr);

#### 3. 修改 [sys\_call\_table](http://lxr.free-electrons.com/ident?i=sys_call_table)

一维数组赋值。当然，我们需要先把真正的值保存好，以备后面之需。

disable\_write\_protection();

real\_open = (void \*)sys\_call\_table[\_\_NR\_open];

sys\_call\_table[\_\_NR\_open] = (unsigned long\*)fake\_open;

real\_unlink = (void \*)sys\_call\_table[\_\_NR\_unlink];

sys\_call\_table[\_\_NR\_unlink] = (unsigned long\*)fake\_unlink;

real\_unlinkat = (void \*)sys\_call\_table[\_\_NR\_unlinkat];

sys\_call\_table[\_\_NR\_unlinkat] = (unsigned long\*)fake\_unlinkat;

enable\_write\_protection();

#### 4. 恢复

disable\_write\_protection();

sys\_call\_table[\_\_NR\_open] = (unsigned long\*)real\_open;

sys\_call\_table[\_\_NR\_unlink] = (unsigned long\*)real\_unlink;

sys\_call\_table[\_\_NR\_unlinkat] = (unsigned long\*)real\_unlinkat;

enable\_write\_protection();

### 第二部分：基于系统调用挂钩的初级文件监视

监视文件的创建与删除。我们挂钩sys\_open,sys\_unlink,sys\_unlinkat这三个函数，并且在我们的钩子函数把操作到的文件名打印出来，然后把控制交给真正的系统调用处理。

#### 1. [sys\_open](http://lxr.free-electrons.com/ident?i=sys_open)的钩子函数：fake\_open

考虑到在系统运行时，对文件的读写操作从未中断，这里只打印了进行创建操作的文件名，准确地说是，[sys\_open](http://lxr.free-electrons.com/ident?i=sys_open) 的 flags中包含 [O\_CREAT](http://lxr.free-electrons.com/ident?i=O_CREAT) 。

asmlinkage long

fake\_open(const char \_\_user \*filename, int flags, umode\_t mode)

{

  if ((flags & O\_CREAT) && strcmp(filename, "/dev/null") != 0) {

    printk(KERN\_ALERT "open: %s\n", filename);

  }

  return real\_open(filename, flags, mode);

}

注：这里的[strcmp](https://www.kernel.org/doc/htmldocs/kernel-api/API-strcmp.html)也是内核提供的。

#### 2. [sys\_unlink](http://lxr.free-electrons.com/ident?i=sys_unlink) 与 [sys\_unlinkat](http://lxr.free-electrons.com/ident?i=sys_unlinkat) 的钩子函数： fake\_unlink 与 fake\_unlinkat

简单处理，直接打印路径名。

asmlinkage long

fake\_unlink(const char \_\_user \*pathname)

{

  printk(KERN\_ALERT "unlink: %s\n", pathname);

  return real\_unlink(pathname);

}

asmlinkage long

fake\_unlinkat(int dfd, const char \_\_user \* pathname, int flag)

{

  printk(KERN\_ALERT "unlinkat: %s\n", pathname);

  return real\_unlinkat(dfd, pathname, flag);

}

## 实验要求

1. 通过认真学习linux kernel hook原理，明确实验目的、原理、方法以及注意事项等。
2. 实验过程中必须认真严肃，并认真学习和记录实验数据，从而进行科学分析。
3. 独立认真完成实验报告，语言简练、表达清晰，适当情况下增加相应的画图信息。

## 实验内容和步骤

#### 5.1 第一部分

在这一部分，我们将探究**基于修改系统调用表sys\_call\_table的系统挂钩的实现原理**。

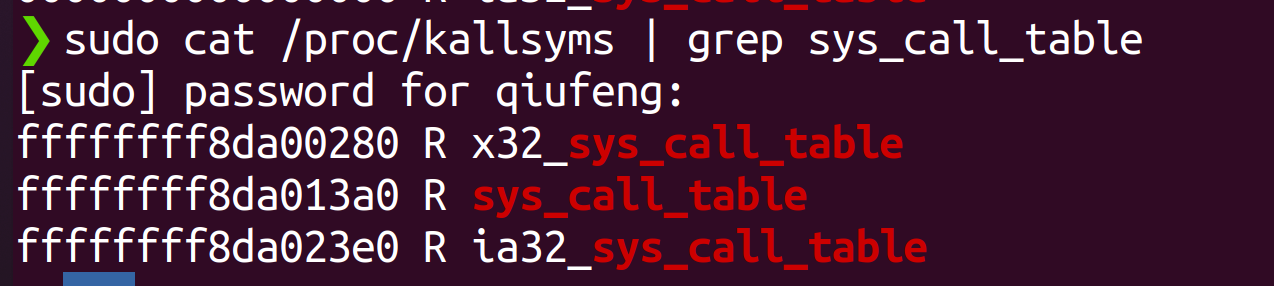
##### 5.1.1 获取sys\_call\_table内存地址

###### 使用/proc/kallsyms

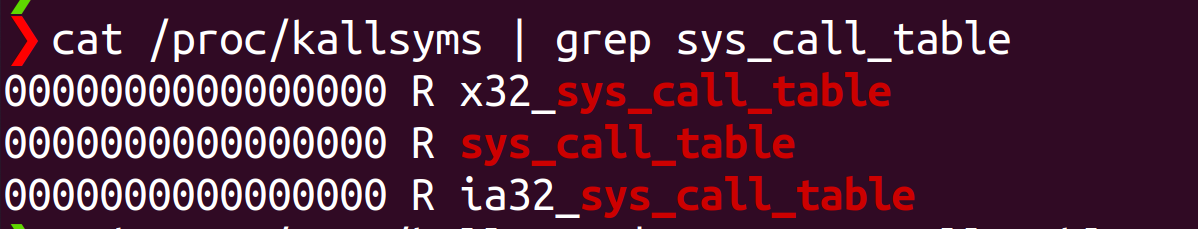
在***kallsyms***文件中包含了kernel image和动态加载模块的符号表，包括内核中的函数符号、全局变量等，因此我们可以通过如下命令简单的获取系统调用表的地址

sudo cat /proc/kallsyms | grep sys\_call\_table

得到的结果如下所示



需要注意的是这里必须要使用sudo，否则无法读出正确的结果，如下图所示

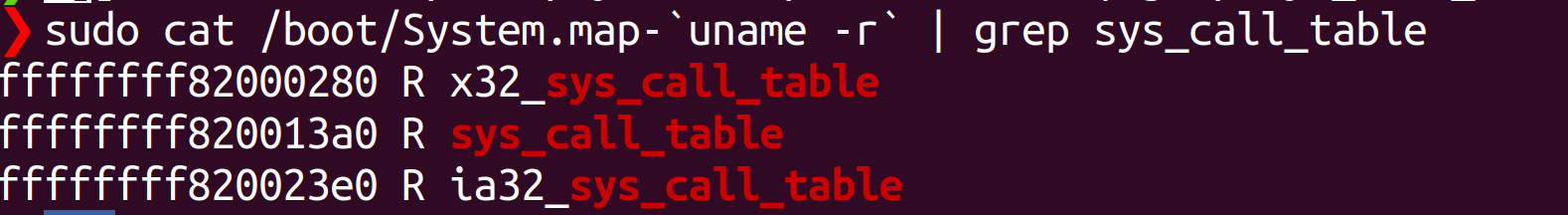


###### 使用System.map

***System.map***文件是一份内核符号表，它包含了内核中的变量名和函数名地址，在每次编译内核时自动生成，我们可以使用如下命令获取系统调用表的函数

sudo cat /boot/System.map-`uname -r` | grep sys\_call\_table

得到的结果如下图所示

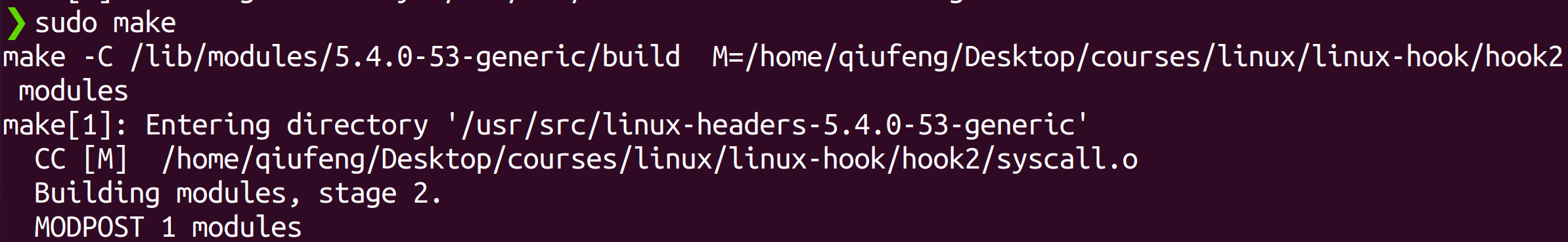


我们可以发现**用这种方法的到的结果与第一种方法结果不同**，这是因为正在运行的内核可能与System.map不匹配。而在***/proc/kallsyms***中增加的函数符号是后来通过安装层序引入的，因此相比较而言，使用***/proc/kallsyms***的到的系统调用表的地址更加可靠。

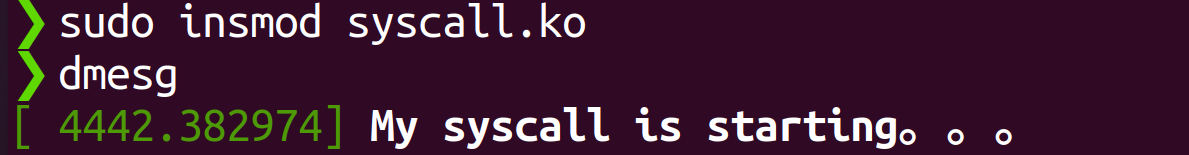
###### 内核模块暴力穷搜法

该方法利用了**导出函数*sys\_close***。我们从内核内存空间的起始地址***PAGE\_OFFSET***开始，逐一的尝试访问每一个指针大小的内存，根据索引***\_\_NR\_close***处的值是否与系统调用***sys\_close***所对应的系统调用地址相等，判断当前内存是否为***sys\_call\_table***的起始地址。

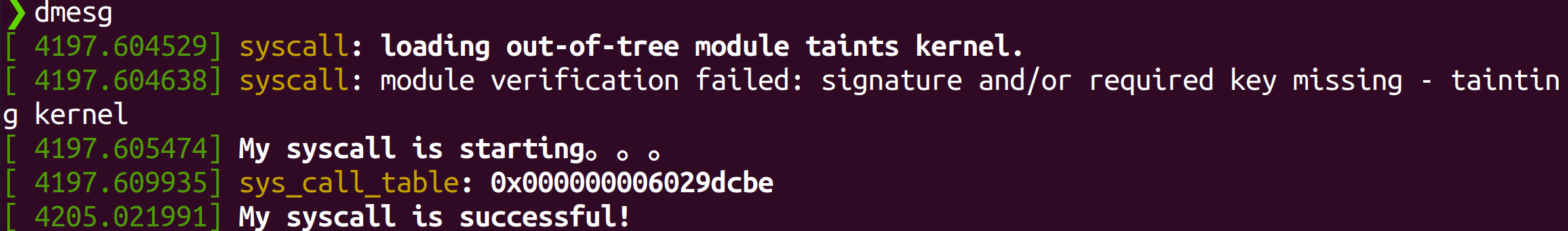
我们首先编译要加载的内核模块



加载内核模块

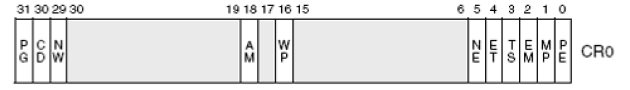


运行测试文件并查看系统调用表的地址



##### 5.1.2 关闭写保护

由于系统调用表**sys\_call\_table**所在的内存区域还具有写保护，即如果我们不能直接向其中写入数据。**该特性由CR0寄存器的WP位控制**。其中CR0寄存器的结构如下图所示



因此我们只需要将该位置0，即可以关闭写保护，对只读的物理页进行修改。在这里，我们借助函数***read\_cr0***和***write\_cr0***对CR0寄存器进行相关的操作。这两个函数的定义分别如下

***read\_cr0***实际上是一个指向***native\_read\_cr0***的函数指针，它利用了内联汇编对CR0寄存的值进行读取

static inline unsigned long native\_read\_cr0(void)

{

    unsigned long val;

    asm volatile("mov %%cr0,%0\n\t" : "=r" (val) : \_\_FORCE\_ORDER);

    return val;

}

***write\_cr0***实际上是指向***native\_write\_cr0***的函数指针，该函数的定义如下

void native\_write\_cr0(unsigned long val)

{

    unsigned long bits\_missing = 0;

set\_register:

    asm volatile("mov %0,%%cr0": "+r" (val) : : "memory");

    if (static\_branch\_likely(&cr\_pinning)) {

        if (unlikely((val & X86\_CR0\_WP) != X86\_CR0\_WP)) {

            bits\_missing = X86\_CR0\_WP;

            val |= bits\_missing;

            goto set\_register;

        }

        /\**Warn after we've set the missing bits.*\*/

        WARN\_ONCE(bits\_missing, "CR0 WP bit went missing!?\n");

    }

}

##### 5.1.3 修改sys\_call\_table

接下来我们用我们自定义的函数去覆盖系统调用表中相应的表项

* ***fake\_open***代替***sys\_call\_table[\_\_NR\_open]***
* ***fake\_unlink***代替***sys\_call\_table[\_\_NR\_unlink]***
* ***fake\_unlinkat***代替***sys\_call\_table[\_\_NR\_unlinkat]***

在写之前我们需要关闭写保护，写之后需要打开写保护

##### 5.1.4 恢复

当实验进行完毕之后，我们用原本系统调用函数恢复原始系统调用表

disable\_write\_protection();

sys\_call\_table[\_\_NR\_open] = (unsigned long\*)real\_open;

sys\_call\_table[\_\_NR\_unlink] = (unsigned long\*)real\_unlink;

sys\_call\_table[\_\_NR\_unlinkat] = (unsigned long\*)real\_unlinkat;

enable\_write\_protection();

#### 5.2 第二部分

##### 5.2.1 代码分析

实验代码基于[***linux-kernel-hool***](https://github.com/jha/linux-kernel-hook)。

该代码主要是修改了系统调用中的***sys\_mkdir***函数。挂钩函数如下

asmlinkage int

mkdir\_hook(const char \*path, int mode)

{

    sys\_mkdir\_t sys\_mkdir;

printk(KERN\_INFO “call mkdir now\n”);

    sys\_mkdir = (sys\_mkdir\_t)sys\_hook\_get\_orig64(lkh\_sys\_hook, \_\_NR\_mkdir);

    return sys\_mkdir(path, mode);

}

该挂钩函数主要在进行文件夹创建的系统调用时执行了如下操作

* 打印正在创建文件夹的信息
* 通过***sys\_hook\_get\_orig64***获取原始系统表中***sys\_mkdir***的函数地址
* 调用原始***sys\_mkdir***

函数***sys\_hook\_get\_orig64()***主要是通过遍历结构体***sys\_hook***并通过比较***syscall\_id***获取原始函数的地址

uintptr\_t

sys\_hook\_get\_orig64(struct sys\_hook \*hook, unsigned int syscall\_id)

{

    struct sys\_hook\_ent \*curr;

    for (curr = hook->head; curr != NULL; curr = curr->next) {

        if (curr->type == SHT\_X64 && curr->syscall\_id == syscall\_id)

            return curr->original;

    }

    return 0;

}

其中***sys\_hook***是一个链表，其结构如下所示

struct sys\_hook {

    unsigned int \*x86\_sct;

    unsigned long long \*x64\_sct;

    struct sys\_hook\_ent \*head, \*tail;

};

结构体***sys\_hook\_ent***包含每一个系统调用的相关信息，包含***syscall\_id***、原始系统调用函数的地址、hook函数的地址等

struct sys\_hook\_ent {

    struct sys\_hook\_ent \*next;

    unsigned int syscall\_id;

    uintptr\_t original, hooked;

    enum sys\_hook\_type type;

};

我们通过函数***sys\_hook\_add64***来添加新的hook函数，其执行的主要操作如下

* 初始化结构体***sys\_hook\_ent***
* 通过写入***CR0***寄存器关闭写保护
* 修改系统调用表***sys\_call\_table***
* 通过写入***CR0***寄存器打开写保护

bool\_t

sys\_hook\_add64(struct sys\_hook \*hook, unsigned int syscall\_id, void \*func)

{

    /\**Create our new hook entry*\*/

    ent->next = NULL;

    ent->syscall\_id = syscall\_id;

    ent->original = hook->x64\_sct[syscall\_id];

    ent->hooked = (uintptr\_t)func;

    ent->type = SHT\_X64;

    /\**Overwrite the entry in the syscall table*\*/

    set\_cr0(get\_cr0() & ~CR0\_WRITE\_PROTECT);

    hook->x64\_sct[syscall\_id] = (unsigned long long)ent->hooked;

    set\_cr0(get\_cr0() | CR0\_WRITE\_PROTECT);

}

函数***sys\_hook\_free***提供了释放hook函数并将系统调用表还原的功能，它执行的主要操作如下

* 通过写入***CR0***寄存器关闭写保护
* 遍历链表***sys\_hook***，根据***original***字段恢复原始系统调用表
* 通过写入***CR0***寄存器打开写保护

void

sys\_hook\_free(struct sys\_hook \*hook)

{

    set\_cr0(get\_cr0() & ~CR0\_WRITE\_PROTECT);

    for (curr = hook->head; curr != NULL;) {

            hook->x64\_sct[curr->syscall\_id] = (unsigned long long)curr->original;

            break;

        }

        tmp = curr->next;

        kfree(curr);

        curr = tmp;

    }

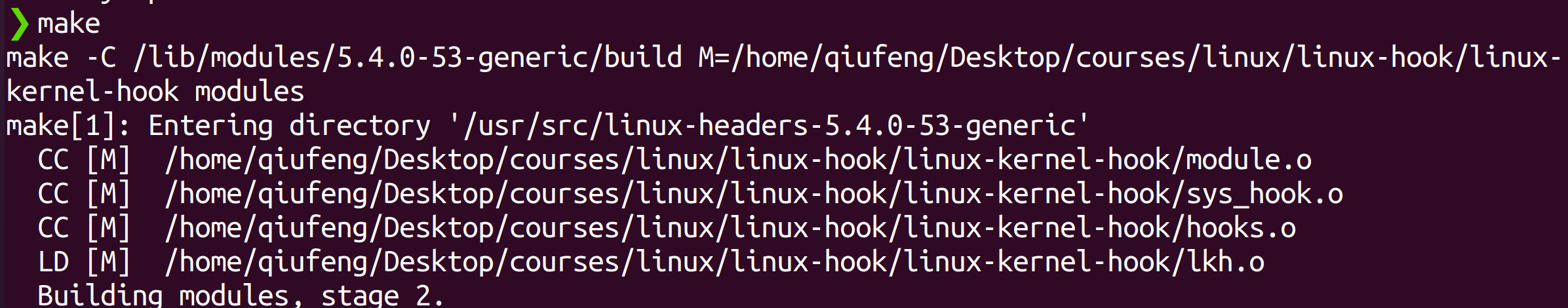
    set\_cr0(get\_cr0() | CR0\_WRITE\_PROTECT);

    kfree(hook);

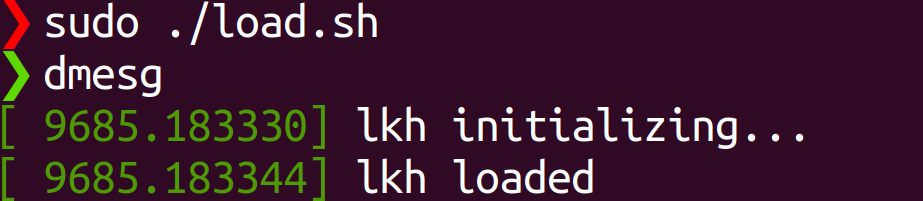
}

##### 5.2.2 实验结果

首先执行***make***进行编译



装载内核模块



依次执行创建文件夹的命令

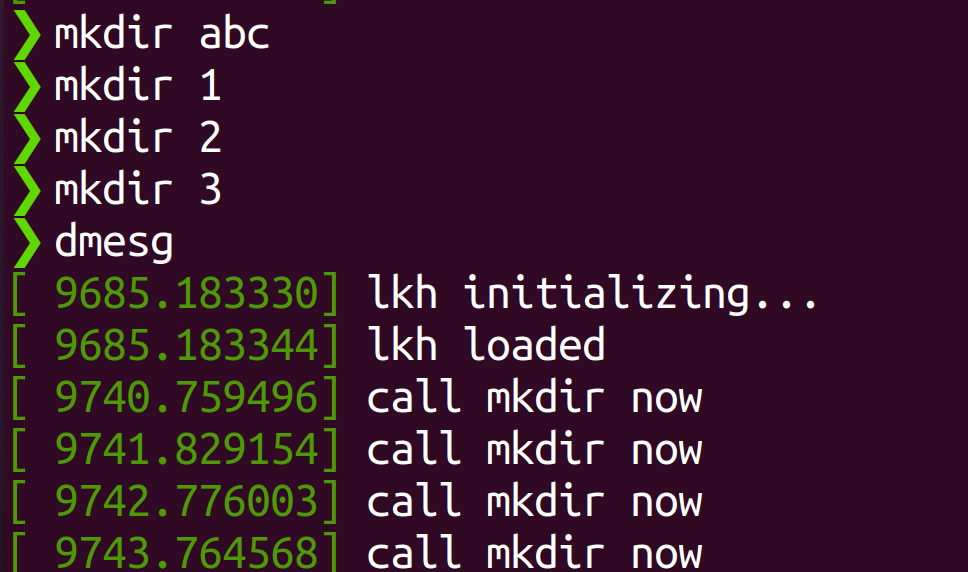
mkdir abc

mkdir 1

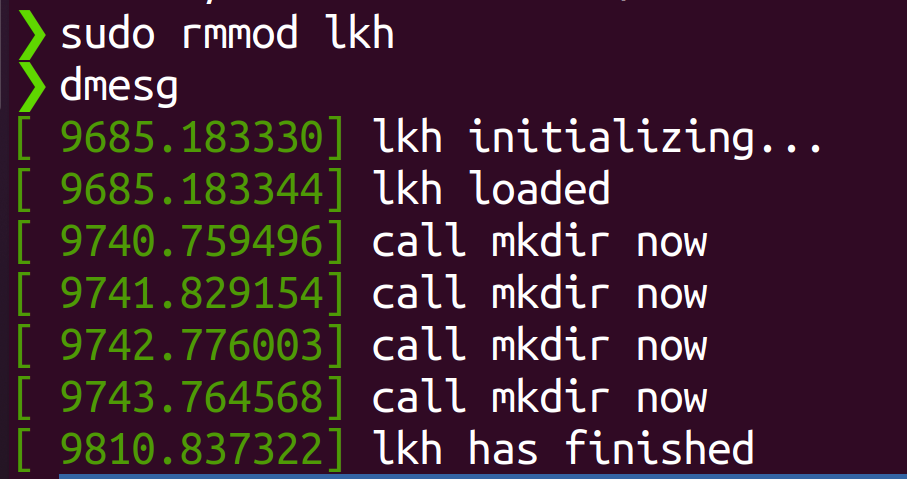
mkdir 2

mkdir 3

可以发现文件夹创建被成功监控



卸载内核模块



## 实验问题

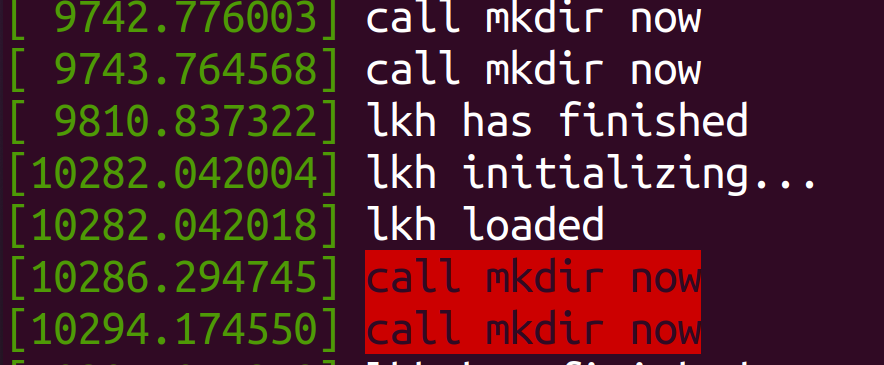
#### KERN\_ALERT的作用？

内核在使用***printk***打印信息时必须要指定日志的级别，***KERN\_ALERT***即是其中的一种级别，常见的级别还有以下几种

* ***KERN\_EMERG***：用于紧急事件消息，一般是系统崩溃之前提示的消息
* ***KERN\_ALERT***：用于需要立即采取动作的情况
* ***KERN\_CRIT***：临界状态，通常涉及严重的硬件或软件操作失败
* ***KERN\_ERR***：用于报告错误状态，设备驱动程序会经常使用KERN\_ERR来报告来自硬件的问题
* ***KERN\_WARNING***：对可能出现问题的情况进行警告，但这类情况通常不会对系统造成严重问题
* ***KERN\_NOTICE***：有必要进行提示的正常情形，许多与安全相关的状况用这个级别进行汇报
* ***KERN\_INFO***：提示性信息，很多驱动程序在启动的时候以这个级别来打印他们找到的硬件信息
* ***KERN\_DEBUG***：用于调试信息

通过使用***KERN\_ALERT***可以表明此时系统可能遇到的严重的问题，需要立即采取相应的动作。

例如，我们将上述实验从***KERN\_INFO***改为***KERN\_ALERT***，可以发现打印出的信息用红色高亮显示，表明出现了某些紧急情况



#### 二、本实验中提到了几种获取sys\_call\_table地址的方法，请思考它们的优缺点？

本实验中提到的获取***sys\_call\_table***地址的方法主要有以下几种

* 从***/boot/System.map***中读取
* 通过***sys\_close***函数暴力搜索

实际上还可以有如下两种方法

* 在***/proc/kallsyms***中读取
* 使用***kallsyms\_lookup\_name()***函数获取

关于如何获取可以参考[上面的实验](#_5.1.1_获取sys_call_table内存地址)，下面只是分析一下各自的优缺点。

* ***System.map***
  + ***System.map***是一份内核符号表，它包含了内核中的变量名和函数名地址，**在每次编译内核时自动生成**。因此可能会出现**正在运行的内核与*System.map*不匹配的问题**。导致结果可能不正确
* 暴力搜索
  + 通过暴力搜索的方法获取的结果**正确性较高**，**但必须使用内核模块的方式，并且速度较慢**。
* ***/proc/kallsyms***
  + ***kallsyms***包含了内核镜像和动态加载模块的符号表，因此获取的结果较为准确，但如果函数被编译器**内联优化**，则在***/proc/kallsyms***中可能找不到
* ***kallsyms\_lookup\_name***
  + 该函数同暴力搜索类似，只能够在内核模块使用，涉及到***CR0***寄存器的读写以及内核模块的装载，较为复杂但准确率较高

## 参考资料

* [Linux Cross Reference](http://lxr.free-electrons.com/)
* [The Linux KernelAPI](https://www.kernel.org/doc/htmldocs/kernel-api/index.html)
* [How the Linux kernel handles a systemcall](https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-2.html)
* [CR0](https://en.wikipedia.org/wiki/Control_register#CR0)
* [Hooking the Linux System CallTable](https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/)
* [Kernel-LandRootkits](http://www.kernelhacking.com/rodrigo/docs/StMichael/kernel-land-rootkits.pdf)