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 ：

年 月 日

目 录

[实验四：Linux Kernel Hook](#_Toc9377)

[1. 实验目的](#_Toc26601)

[2. 实验环境](#_Toc4854)

[3. 实验原理](#_Toc10514)

[4. 实验要求](#_Toc32551)

[5. 实验内容和步骤](#_Toc22808)

[6. 实验报告](#_Toc1637)

[7. 实验问题](#_Toc7696)

[8. 参考资料](#_Toc17802)

# 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）uname -r: 4.15.0-39-generic

## 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：

一，从[/boot/System.map](https://en.wikipedia.org/wiki/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。

二，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：

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](https://en.wikipedia.org/wiki/Control_register" \l "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](http://lxr.free-electrons.com/ident?i=read_cr0) /[write\_cr0](http://lxr.free-electrons.com/ident?i=write_cr0) 来读取 /写入 [CR0](https://en.wikipedia.org/wiki/Control_register#CR0) 寄存器，免去我们自己写内联汇编的麻烦。

函数原型。

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。

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](http://lxr.free-electrons.com/ident?i=sys_open),[sys\_unlink](http://lxr.free-electrons.com/ident?i=sys_unlink),[sys\_unlinkat](http://lxr.free-electrons.com/ident?i=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

项目地址：<https://github.com/jha/linux-kernel-hook>

为了方便观察实验结果，在hook.c:11中加入以下输出代码：

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

然后依次执行以下命令：

make

sudo ./load.sh

mkdir abc

mkdir 1

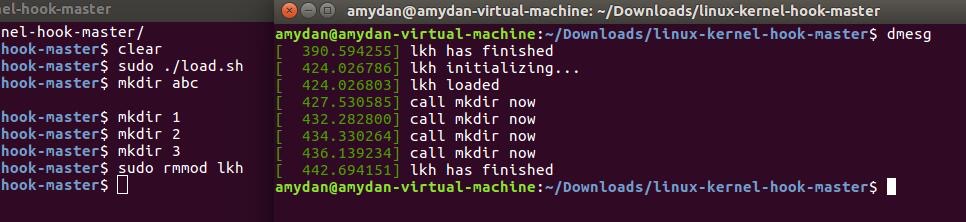
mkdir 2

mkdir 3

sudo rmmod lkh

dmesg

观察实验结果，如下图所示：



## 实验报告

根据实验内容完成实验报告。

## 实验问题

【问题一】

KERN\_ALERT的作用？

【问题二】

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？

## 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)