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) ： 信息安全

学 号 ：

姓 名 ：

任课教师 ： 王鹃

2023 年 11 月 1 日

目录

[实验四：Linux Kernel Hook 3](#_Toc150109033)

[1. 实验目的 3](#_Toc150109034)

[2. 实验环境 3](#_Toc150109035)

[3. 实验原理 3](#_Toc150109036)

[4. 实验要求 6](#_Toc150109037)

[5. 实验内容和步骤 7](#_Toc150109038)

[5.1 基于修改 sys\_call\_table 的系统调用挂钩 7](#_Toc150109039)

[5.2 基于系统调用挂钩的初级文件监视 9](#_Toc150109040)

[6. 实验报告 12](#_Toc150109041)

[7. 实验问题 12](#_Toc150109042)

[8. 参考资料 14](#_Toc150109043)

# 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](https://elixir.bootlin.com/linux/latest/ident/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](https://en.wikipedia.org/wiki/System.map) 中读取，可以查阅 [Hooking the Linux System CallTable](https://web.archive.org/web/20220627054149/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](https://elixir.free-electrons.com/linux/latest/ident/PAGE_OFFSET)是内核内存空间的起始地址。 因为[sys\_close](https://elixir.bootlin.com/linux/latest/ident/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#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](https://elixir.bootlin.com/linux/latest/ident/read_cr0) /[write\_cr0](https://elixir.bootlin.com/linux/latest/ident/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#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#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](https://elixir.bootlin.com/linux/latest/ident/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](https://elixir.bootlin.com/linux/latest/ident/sys_open),[sys\_unlink](https://elixir.bootlin.com/linux/latest/ident/sys_unlink),[sys\_unlinkat](https://elixir.bootlin.com/linux/latest/ident/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](https://elixir.bootlin.com/linux/latest/ident/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](https://elixir.bootlin.com/linux/latest/ident/sys_open) 的 flags中包含 [O\_CREAT](https://elixir.bootlin.com/linux/latest/ident/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](https://elixir.bootlin.com/linux/latest/ident/sys_unlink) 与 [sys\_unlinkat](https://elixir.bootlin.com/linux/latest/ident/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 基于修改 sys\_call\_table 的系统调用挂钩

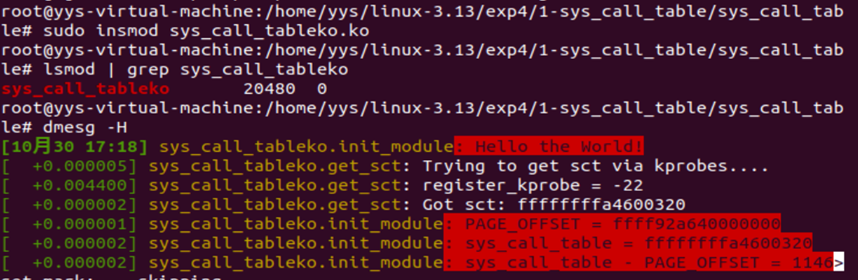
1. 获得 sys\_call\_table 的内存地址

1. unsigned **long** \*\*
2. get\_sct(**void**)
3. {
4. unsigned **long** \*\*entry = (unsigned **long** \*\*)PAGE\_OFFSET;
6. **for** (;(unsigned **long**)entry < ULONG\_MAX; entry += 1) {
7. **if** (entry[\_\_NR\_close] == (unsigned **long** \*)sys\_close) {
8. **return** entry;
9. }
10. }
12. **return** NULL;
13. }
14. #endif

上述代码的作用是获取系统调用表（sys\_call\_table）的地址。系统调用表是内核中的一个数组，存储着系统中所有系统调用的地址。在Linux内核中，sys\_close是一个导出的函数，我们可以直接获取它的地址。通过扫描内核内存空间，从PAGE\_OFFSET开始尝试每一个指针大小的内存，判断其中存储的地址是否和sys\_close函数的地址相同。如果匹配成功，则认为当前尝试的内存块就是sys\_call\_table的地址。\_\_NR\_close是系统调用close的编号，作为索引直接访问数组的一部分。

PAGE\_OFFSET是内核内存空间的起始地址，是一个宏定义。在该函数中，循环遍历了内核内存空间，直到尝试到ULONG\_MAX，即遍历了整个可能的内存空间，如果没有找到对应的sys\_call\_table的地址，就返回NULL。

编写测试程序，运行得到结果如下：



PAGE\_OFFSET：内核虚拟地址空间的起始地址。

sys\_call\_table：系统调用表（sys\_call\_table）的地址。

sys\_call\_table - PAGE\_OFFSET：这个值是sys\_call\_table与PAGE\_OFFSET之间的偏移量。在输出中显示为"1146"，这个偏移量表示了sys\_call\_table相对于PAGE\_OFFSET的距离，用于计算sys\_call\_table在内存中的实际位置。

2. 关闭写保护

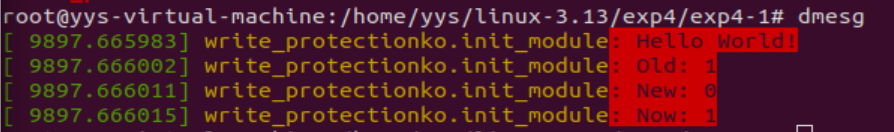
得到了sys\_call\_table的地址就可以开始关闭写保护了。

1. //关闭写保护
2. **void** disable\_wp(**void**)
3. {
4. unsigned **long** cr0;
6. preempt\_disable();
7. cr0 = read\_cr0();
8. clear\_bit(X86\_CR0\_WP\_BIT, &cr0);
9. old\_native\_write\_cr0(cr0);
10. preempt\_enable();
12. **return**;
13. }
14. //开启写保护
15. **void** enable\_wp(**void**)
16. {
17. unsigned **long** cr0;
19. preempt\_disable();
20. cr0 = read\_cr0();
21. set\_bit(X86\_CR0\_WP\_BIT, &cr0);
22. write\_cr0(cr0);
23. preempt\_enable();
25. **return**;
26. }

编写测试程序，代码如下：

1. init\_module(**void**)
2. {
3. unsigned **long** cr0;
4. unsigned **long** cr4;
6. fm\_alert("%s\n", "Hello World!");
8. cr0 = read\_cr0();
9. fm\_alert("Old: %d\n", test\_bit(X86\_CR0\_WP\_BIT, &cr0));
11. disable\_wp();
12. cr0 = read\_cr0();
13. fm\_alert("New: %d\n", test\_bit(X86\_CR0\_WP\_BIT, &cr0));
15. enable\_wp();
16. cr0 = read\_cr0();
17. fm\_alert("Now: %d\n", test\_bit(X86\_CR0\_WP\_BIT, &cr0));
19. cr4 = \_\_read\_cr4();
20. fm\_alert("old cr4 = %lx", cr4);
22. **return** 0;
23. }

测试代码运行结果如下：



可以看到关闭写保护前CR0的第16个bit为1，关闭后为0，开启后又重新恢复为1

3. 修改与恢复sys\_call\_table

本实验通过定义宏HOOK和UNHOOK来简化修改操作：

1. # define HOOK\_SCT(sct, name)                    \
2. **do** {                                        \
3. real\_##name = (**void** \*)sct[\_\_NR\_##name]; \
4. fm\_info("Original real\_" #name " = %lx", (unsigned **long**)real\_##name);    \
5. sct[\_\_NR\_##name] = (**void** \*)fake\_##name; \
6. } **while** (0)
8. # define UNHOOK\_SCT(sct, name)                  \
9. sct[\_\_NR\_##name] = (**void** \*)real\_##name

其中HOOK宏的作用是首先保存原来系统调用表中的地址到real\_name中，然后将我们要修改的系统调用的地址修改为自定义的函数fake\_name

UNHOOK宏的作用就是恢复系统调用表，将我们修改的部分重新修改回real\_name中保存的原始地址。通过这两个宏的定义，可以在代码中方便地对系统调用进行hook和取消hook的操作，而无需重复编写重复的代码。

### 5.2 基于系统调用挂钩的初级文件监视

sys\_open的钩子函数：fake\_open

1. asmlinkage **long** fake\_open(
2. **const** **char** \_\_user \*filename, **int** flags, umode\_t mode)
3. {
4. **char** kfn[1024];
5. **if** (strncpy\_from\_user(kfn, filename, **sizeof** kfn) <= 0) {
6. fm\_alert("strncpy\_from\_user failed!!!");
7. } **else** {
8. **if** ((flags & O\_CREAT) && strcmp(kfn, "/dev/null") != 0) {
9. fm\_alert("open: %s\n", kfn);
10. }
11. }
13. **return** real\_open（filename, flags, mode);
14. }

fake\_open拦截了系统调用open的操作。在拦截之后，它首先尝试从用户空间复制文件名到内核空间中的kfn缓冲区，然后检查拷贝是否成功。如果拷贝失败，则会打印一条警告信息。接着，它检查传入的flags参数是否包含O\_CREAT标志，并且文件名不是/dev/null，如果满足条件，则打印一条关于打开的文件名的警告信息。最后，它调用了真实的open系统调用，将控制权交给原始的系统调用。

sys\_unlink的钩子函数：fake\_unlink

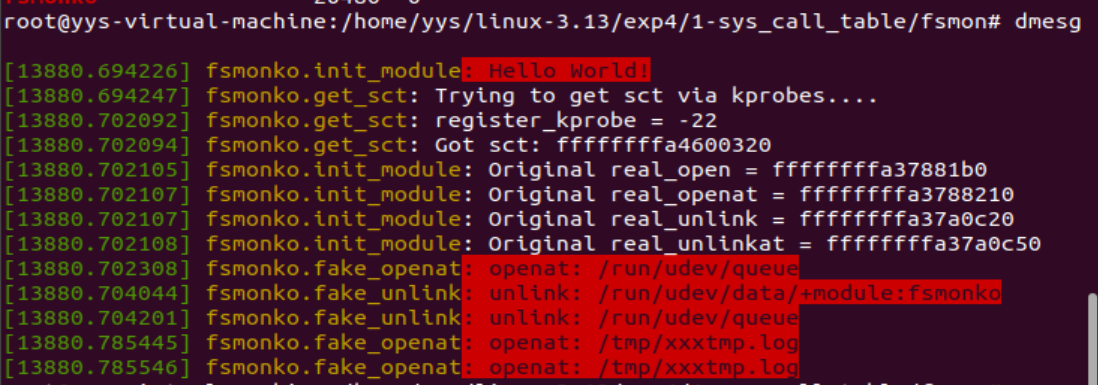
1. asmlinkage **long** fake\_unlink(**const** **char** \_\_user \*pathname)
2. {
3. **char** kfn[1024];
4. **if** (strncpy\_from\_user(kfn, pathname, **sizeof** kfn) <= 0) {
5. fm\_alert("strncpy\_from\_user failed!!!");
6. }
7. **else**
8. {
9. fm\_alert("unlink: %s\n", kfn);
10. }
12. **return** real\_unlink(pathname);
13. }

sys\_unlinkat 的钩子函数： fake\_unlinkat

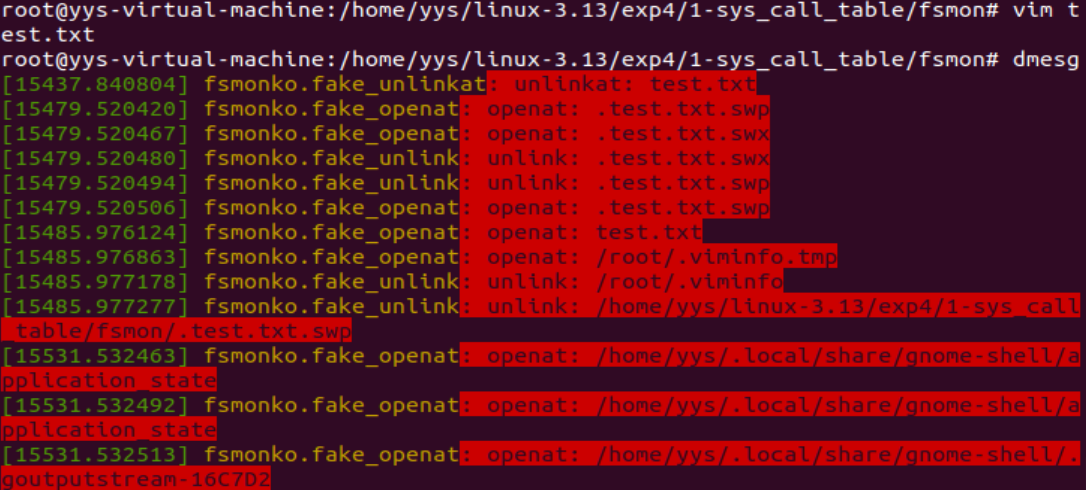
1. asmlinkage **long** fake\_unlinkat(
2. **int** dfd, **const** **char** \_\_user \* pathname, **int** flag)
3. {
4. **char** kfn[1024];
5. **if** (strncpy\_from\_user(kfn, pathname, **sizeof** kfn) <= 0) {
6. fm\_alert("strncpy\_from\_user failed!!!");
7. } **else**
8. {
9. fm\_alert("unlinkat: %s\n", kfn);
10. }
12. **return** real\_unlinkat(dfd, pathname,flag );
13. }

这两段代码拦截了unlink和unlikat的系统调用，跟踪在文件删除操作发生时记录被删除的文件名，并打印出来，最后调用真实的文件删除操作。

运行测试程序结果如下：

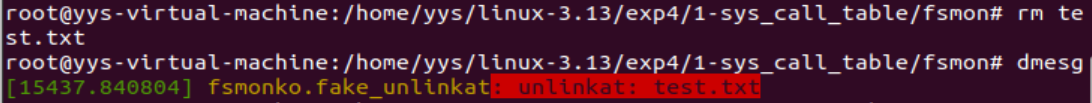


尝试创建一个文件test.txt



钩取到到test.txt创建过程中unlink和openat的操作

我们再把创建的test.txt删除



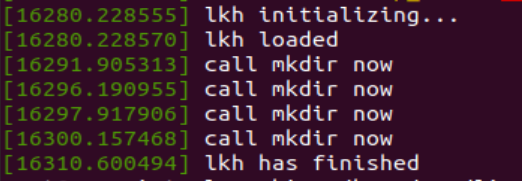
删除过程被成功钩取

mkdir的钩子函数mkdir\_hook

1. asmlinkage **int**
2. mkdir\_hook(**const** **char** \*path, **int** mode)
3. {
4. sys\_mkdir\_t sys\_mkdir;
5. printk(KERN\_INFO “call mkdir now\n”);
6. sys\_mkdir = (sys\_mkdir\_t)sys\_hook\_get\_orig64(lkh\_sys\_hook, \_\_NR\_mkdir);
8. **return** sys\_mkdir(path, mode);
9. }

Mkdir\_hook用于拦截mkdir系统调用。它首先通过sys\_hook\_get\_orig64函数从lkh\_sys\_hook中获取mkdir系统调用的原始函数地址，并将其保存在sys\_mkdir变量中。然后，它调用sys\_mkdir函数来执行原始的mkdir系统调用，从而实现了对mkdir系统调用的拦截和修改。最终，它返回sys\_mkdir的执行结果。

运行结果如下：



## 实验报告

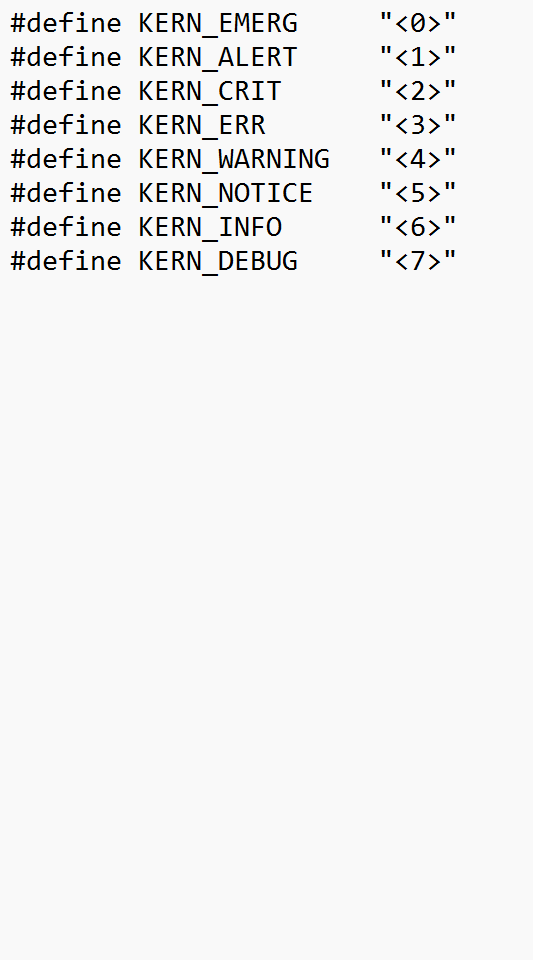
根据实验内容完成实验报告。

## 实验问题

【问题一】

KERN\_ALERT的作用？

Linux内核基本上有八个日志级别，由linux内核发送消息，日志级别从高到低，level 0 ~ level 7



0 KERN\_EMERG 这是按严重性排序的最高级别：它被有关系统不稳定或即将崩溃的消息采用。

1 KERN\_ALERT 此级别用于立即需要用户注意的情况。

2 KERN\_CRIT此严重性级别用于通知与硬件或软件相关的严重错误。

3 KERN\_ERR通常用于通知用户非关键错误，例如设备识别失败或有问题，或者更一般的驱动程序相关问题。

4 KERN\_WARNING或 loglevel 它是大多数 Linux 发行版中通常用作默认值的日志级别。此级别用于显示有关非即将发生的错误的警告或消息。

5 KERN\_NOTICE此级别的消息是关于可能值得注意的事件。

6 KERN\_INFO此级别的消息用于有关内核执行的操作的信息性消息的日志级别。

7 KERN\_DEBUG此级别的消息主要用于调试。

我们检查一下系统上使用的默认日志级别。



1. 第一个值表示当前使用的日志级别。这意味着只有具有高于该级别的严重性的消息才会显示在控制台上。

2. 第二个值用于没有特定日志级别的消息。如果消息未与特定的日志级别关联，则将使用此值作为默认级别。

3. 第三个值指示最低可用于控制台日志的日志级别。在本实验中，它被设置为1，表示只有具有严重性级别1（KERN\_ALERT）的消息才会显示在控制台上。

4. 最后一个值代表系统在引导时使用的默认日志级别。默认情况下，系统将使用这个级别来决定哪些消息会显示在控制台上。

因此，KERN\_ALERT作为一种日志级别，用于标识紧急情况下的系统警报消息。它表示某些需要立即引起注意的严重错误或状态，这可能会导致系统的不可恢复性损坏或数据丢失。

这个级别通常用于记录需要管理员立即采取行动的系统事件，如硬件故障、内核崩溃或其他导致系统不可用或数据完整性受损的情况。由于其高优先级，KERN\_ALERT级别的消息很少被用于常规的系统日志记录，而是用于紧急情况下的通知。

【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？

1.方法一参考[Hooking the Linux System CallTable](https://web.archive.org/web/20220627054149/https://tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/)主要依赖于/boot/System.map文件。/boot/System.map通常包含了内核中已加载模块和符号的地址映射。该文件提供了内核符号和其对应地址的列表，使得内核的调试和分析工作变得更加方便。

该方法通过打开并读取/boot/System.map文件的内容，找到sys\_call\_table字符串，并将其解析为对应的地址，从而顺利获取系统调用表的地址。

优点：该方法的实现相对直观，不需要对系统的内存进行任何操作，因此易于理解和实现。通过读取/boot/System.map文件，可以确保获取到准确的sys\_call\_table地址，方法较为适用且准确。该方法无需访问内存结构，避免了对内核内存结构的直接操作可能带来的风险。

缺点：该方法依赖于/boot/System.map文件，倘若该文件内容发生变化甚至损坏，就可能导致无法准确获取sys\_call\_table的地址。由于需要读取文件内容并进行解析，该方法在频繁调用时可能会带来一定的性能开销。

2.方法二参考[Kernel-LandRootkits](http://www.kernelhacking.com/rodrigo/docs/StMichael/kernel-land-rootkits.pdf)，本实验中采用直接暴力搜索的方法来获取sys\_call\_table的地址。本例使用了\_\_NR\_close作为系统调用的编号，这是一个已经导出的系统调用函数。利用这个系统调用号索引，代码尝试访问数组中的成员，以确定当前尝试的这块指针大小的内存是否是sys\_call\_table的地址。它利用了系统调用号的后向兼容性和已知导出函数的地址，从内核内存的起始地址开始逐个尝试，直到找到对应的sys\_call\_table地址。

优点：该方法直接搜索内存地址，无需依赖于外部文件系统，因此更加直接有效。由于系统调用号具有后向兼容性，并且已知导出函数的地址可以直接获取，因此这种方法具有一定的通用性，适用于各种不同版本的Linux内核。

缺点：由于需要遍历整个内存地址空间来搜索目标地址，因此其计算量可能相对较大，尤其是在内存较大的系统中可能会影响性能。该方法要求对底层内核结构的了解，不如方法一更加直观、简便。

## 参考资料

* [Linux Cross Reference](https://elixir.bootlin.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/linux-syscall-2.html)
* [CR0](https://en.wikipedia.org/wiki/Control_register#CR0)
* [Hooking the Linux System CallTable](https://web.archive.org/web/20220627054149/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)