Lab 5 : Syscall Hijack

实 验 报 告

刘子涵　518021910690

【1】实验要求

1. 编写一个内核模块。
    - 实验环境：ARM64，x86-64 均可，Linux 5.0 以上内核。
    - 要求替换系统调用表（sys_call_table) 中某一项系统调用，替换成自己编写的系统调用处理函数（例如`my_syscall()`），在新的系统调用函数中打印一句`"hello，I have hacked this syscall"`，然后再调用回原来的系统调用处理函数。
    - 比如以`ioctl`系统调用为例，它在系统调用表中的编号是`__NR_ioctl`。那么需要修改系统调用表`sys_call_table[__NR_ioctl]`的指向，让其指向`my_syscall()`函数，然后在`my_syscall()`函数中打印一句话，调用原来的`sys_call_table[__NR_ioctl]`指向的处理函数。
2. 卸载模块时把系统调用表恢复原样。
3. 用`clone`系统调用来验证你的驱动，clone系统调用号是`__NR_clone`。

【2】实验环境

- 实验平台：华为云弹性云服务器（1vCPUs | 1GiB | t6.small.1 | Ubuntu 20.04 server 64bit with **x86_64**）
- Linux 内核版本：**5.4.0**
- GCC 版本：9.3.0

【3】实验思路及具体实现

注：下文中列出的 Linux 内核源代码均为 5.4.0 版本。

① 系统调用及其处理

系统调用（System Call），是操作系统内核为上层用户提供的接口，当用户想要请求内核的某种服务时（读写文件等），用户可以调用相应的系统调用，访问内核提供的资源。对于 x86-64，当处理器执行到`syscall`汇编指令时，会引起异常陷入内核态，将控制权转移至系统调用处理程序，所有的异常处理程序都是放在内核代码中的。Linux 内核通过访问内存中的**系统调用表（Syscall Table）**来查找对应系统调用号的**系统调用处理函数（Syscall Handler）**。

本实验的总体思路即，找到内存中的系统调用表，解除相应页的写保护，修改系统调用表相应表项的指针，使其指向自定义的处理函数，这一过程可称为 **Syscall Hijack** 或 **Syscall Hook**。

对于 x86-64，系统调用表定义在文件 arch/x86/entry/syscall_64.c 中：

```
asmlinkage const sys_call_ptr_t x32_sys_call_table[__NR_syscall_x32_max+1] = {
    [0 ... __NR_syscall_x32_max] = &sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

可见，系统调用表 `sys_call_table` 的每一项都是 `sys_call_ptr_t` 类型的指针，该类型定义在 arch/x86/include/asm/syscall.h 中：

```
#ifdef CONFIG_X86_64
typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
#else
typedef asmlinkage long (*sys_call_ptr_t)(unsigned long, unsigned long,
                                          unsigned long, unsigned long,
                                          unsigned long, unsigned long);
#endif /* CONFIG_X86_64 */
extern const sys_call_ptr_t sys_call_table[];
```

在模块实现时，我自定义了 `sys_call_ptr_t` 类型，即参数和返回值均为空的函数指针类型，并声明了系统调用表 `syscall_table` （初始化为 NULL），用于存放获取的系统调用表起始地址。实现如下：

```
/* sys_clone_hook.c */
typedef void (*sys_call_ptr_t)(void);
sys_call_ptr_t *syscall_table = NULL;
```

可以通过 `kallsyms_lookup_name("sys_call_table")` 获取系统调用表在内存的起始地址，该函数定义在 include/linux/kallsyms.h 中，返回值类型为 `unsigned long` ：

```
/* Lookup the address for a symbol. Returns 0 if not found. */
unsigned long kallsyms_lookup_name(const char *name);
```

将其强制转换为 `sys_call_ptr_t *` 类型，即可得到指向系统调用表起始地址的指针。实现如下：

```
/* sys_clone_hook.c */
static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");
    // ...
}
```

② clone 系统调用及其处理函数 sys_clone

`clone` 系统调用类似 `fork` ，创建一个子进程。相对于 `fork` ，它提供父子进程之间执行上下文共享机制更为精准的控制，比如可以控制父子进程是否共享虚拟地址空间、打开文件描述符表、信号处理程序表等。`clone` 系统调用对应的处理函数是 `sys_clone()` ，系统调用号是 `__NR_clone` 。其中 `sys_clone()` 定义在 include/linux/syscalls.h 中：

```
#ifdef CONFIG_CLONE_BACKWARDS
asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, unsigned
long,
          int __user *);
#else
#ifdef CONFIG_CLONE_BACKWARDS3
asmlinkage long sys_clone(unsigned long, unsigned long, int, int __user *,
             int __user *, unsigned long);
#else
asmlinkage long sys_clone(unsigned long, unsigned long, int __user *,
          int __user *, unsigned long);
#endif
#endif
```

此处我们需要使用第三个定义，即

`asmlinkage long sys_clone(unsigned long, unsigned long, int __user *, int __user *, unsigned long);`

在模块实现中，我依据以上函数原型自定义了 `sys_clone_t` 函数指针类型，并实例化了函数指针 `orig_clone` 用于保存替换前的处理函数。另外，我还依据以上函数原型自定义了替换的处理函数 `hooked_sys_clone`，该函数参数和返回值与 `sys_clone` 完全一致，实现中首先调用了原来的 `sys_clone` 函数（保证 clone 调用能被正常处理，否则会发生 kernel panic），然后打印一条消息到系统日志供测试查看，最后需要返回原 `sys_clone` 函数返回的返回值。

```
/* sys_clone_hook.c */
typedef asmlinkage long (*sys_clone_t)(unsigned long, unsigned long, int __user
*,
                                       int __user *, unsigned long);
sys_clone_t orig_clone = NULL;

asmlinkage long hooked_sys_clone(unsigned long x1, unsigned long x2, int __user
*x3,
                                 int __user *x4, unsigned long x5) {
    long ret_val = orig_clone(x1, x2, x3, x4, x5);
    printk(KERN_INFO "hello, I have hacked this syscall");
    return ret_val;
}
```

③ 系统调用表修改

（1）保存原来的系统调用处理函数。

```
/* sys_clone_hook.c */
static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");

    // save the original syscall handler
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];

    // ...
}
```

(2) 查找系统调用表内存页的页表项

这一步可以通过 `lookup_address()` 函数实现，该函数定义在 arch/x86/include/asm/pgtable_types.h 中：

```
/*
 * Helper function that returns the kernel pagetable entry controlling
 * the virtual address 'address'. NULL means no pagetable entry present.
 * NOTE: the return type is pte_t but if the pmd is PSE then we return it
 * as a pte too.
 */
extern pte_t *lookup_address(unsigned long address, unsigned int *level);
```

该函数查找虚拟地址 `address` 所在内存页表项指针，在 x86 中该函数的实现在 arch/x86/mm/pageattr.c 中，

```
pte_t *lookup_address(unsigned long address, unsigned int *level)
{
    return lookup_address_in_pgd(pgd_offset_k(address), address, level);
}
```

`lookup_address()` 函数调用了 `lookup_address_in_pgd()` 函数，该函数针对指定 pgd 下的虚拟地址，查找页表项，实现如下：

```
pte_t *lookup_address_in_pgd(pgd_t *pgd, unsigned long address,
                 unsigned int *level)
{
    p4d_t *p4d;
    pud_t *pud;
    pmd_t *pmd;

    *level = PG_LEVEL_NONE;

    if (pgd_none(*pgd))
        return NULL;

    p4d = p4d_offset(pgd, address);
    if (p4d_none(*p4d))
        return NULL;

    *level = PG_LEVEL_512G;
    if (p4d_large(*p4d) || !p4d_present(*p4d))
        return (pte_t *)p4d;

    pud = pud_offset(p4d, address);
    if (pud_none(*pud))
        return NULL;

    *level = PG_LEVEL_1G;
    if (pud_large(*pud) || !pud_present(*pud))
        return (pte_t *)pud;

    pmd = pmd_offset(pud, address);
    if (pmd_none(*pmd))
        return NULL;
```

```
    *level = PG_LEVEL_2M;
    if (pmd_large(*pmd) || !pmd_present(*pmd))
        return (pte_t *)pmd;

    *level = PG_LEVEL_4K;

    return pte_offset_kernel(pmd, address);
}
```

容易发现，`lookup_address()` 是通过逐级遍历页表的方式获取页表项的（walk the page table）。

所以，在模块实现中，需要定义指向页表项的指针 `pte` 和参数 `level`，然后在模块初始化函数中，获取系统调用表后进行查找，实现如下：

```
/* sys_clone_hook.c */
unsigned int level;
pte_t *pte;

// ...

static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];

    // lookup page table entry of syscall table address
    pte = lookup_address((unsigned long)syscall_table, &level);
    // ...
}
```

（3）解除系统调用表内存页的写保护

在 x86-64 中，这一步可以使用定义在 arch/ia64/include/asm/pgtable.h 中的宏 `pte_mkwrite` 来实现：

```
#define pte_mkwrite(pte)    (__pte(pte_val(pte) | _PAGE_AR_RW))
```

为保证操作的原子性，使用定义在 arch/x86/include/asm/pgtable.h 中的宏 `set_pte_atomic` 来实现。模块中实现如下：

```
/* sys_clone_hook.c */
static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];
    pte = lookup_address((unsigned long)syscall_table, &level);

    // change PTE to allow writing
    set_pte_atomic(pte, pte_mkwrite(*pte));
    // ...
}
```

(4) 替换系统调用表相应处理函数为自定义 hooked 函数

```
/* sys_clone_hook.c */
static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];
    pte = lookup_address((unsigned long)syscall_table, &level);
    set_pte_atomic(pte, pte_mkwrite(*pte));

    // overwrite the __NR_clone entry with address to our hook
    syscall_table[__NR_clone] = (sys_call_ptr_t)hooked_sys_clone;
    // ...
}
```

(5) 恢复系统调用表内存页的写保护

这一步类似解除写保护，需要使用原子操作进行修改，不同的是需要使用定义在 arch/x86/include/asm/pgtable.h 中的宏 `pte_clear_flags` 来恢复原来的写保护。

```
/* sys_clone_hook.c */
static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];
    pte = lookup_address((unsigned long)syscall_table, &level);
    set_pte_atomic(pte, pte_mkwrite(*pte));
    syscall_table[__NR_clone] = (sys_call_ptr_t)hooked_sys_clone;

    // reprotect page
    set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));
    // ...
}
```

(6) 卸载模块时把系统调用表恢复原样

实现模块退出函数时，同样使用上述方法实现解除写保护、修改处理程序、恢复写保护。实现如下：

```
static void __exit sys_clone_hook_exit(void) {
    // change PTE to allow writing
    set_pte_atomic(pte, pte_mkwrite(*pte));

    // restore syscall_table to the original state
    syscall_table[__NR_clone] = (sys_call_ptr_t)orig_clone;

    // reprotect page
    set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));

    printk(KERN_INFO "uninstalled sys_clone_hook");
}
```

【4】实验测试及效果截图

1. 预期实验效果

实验附件包括 `test.o` 和 `bench.o`，运行：

- `dmesg`：系统日志输出 1 次 hack 信息
- `./test.o`：系统日志输出 1 次 hack 信息
- `./bench.o`：系统日志输出 6 次 hack 信息

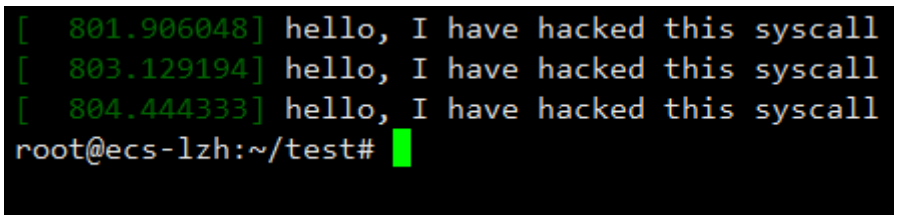
2. 编译并插入模块

编写 Makefile，对模块源文件 `sys_clone_hook.c` 进行编译 `make`。编译后使用 `insmod` 命令插入模块。

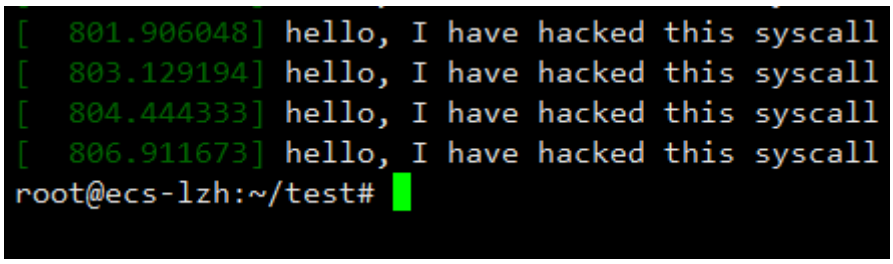
```
obj-m := sys_clone_hook.o
KDIR:=/lib/modules/$(shell uname -r)/build
PWD:=$(shell pwd)
all:
        make -C $(KDIR) M=$(PWD) modules
clean:
        make -C $(KDIR) M=$(PWD) clean
```

3. 测试截图

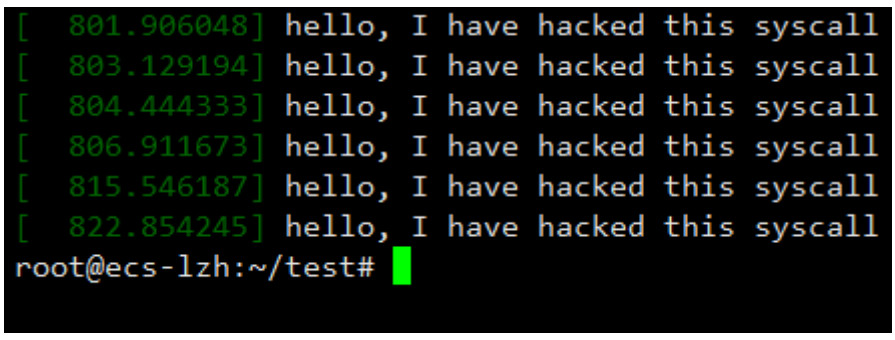
① 初始使用 `dmesg` 查看系统日志：**801-804（3 条）**



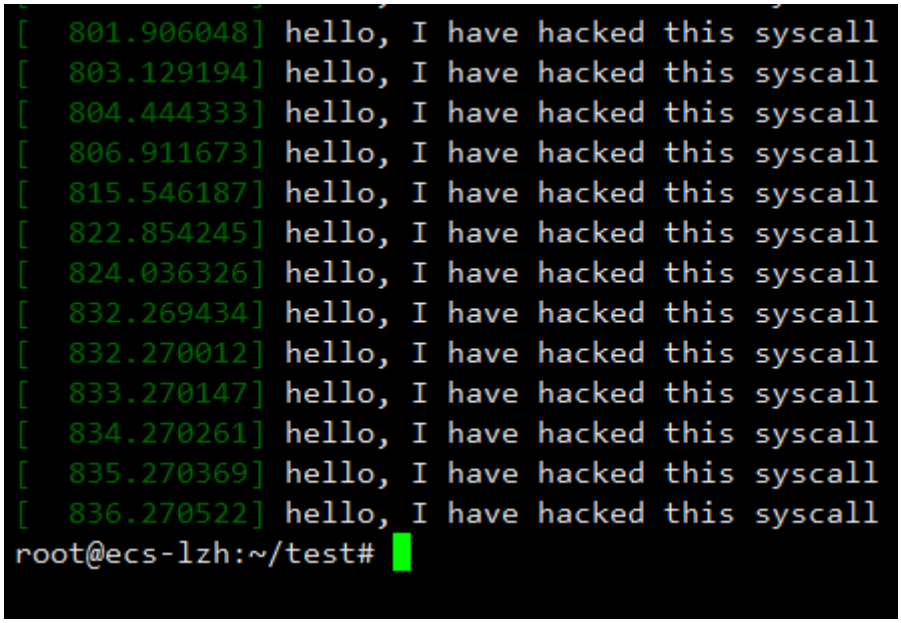
② 再次使用 `dmesg` 查看系统日志：**801-806（4 条），相比上次新增了 1 条输出，正确**



③ 运行 `./test.o`，使用 `dmesg` 查看系统日志：**801-822（6 条），相比上次新增了 2 条输出，正确**（test.o 和 dmesg 各 1 次）



④ 运行 `./bench.o`，使用 `dmesg` 查看系统日志：**801-836（13 条），相比上次新增了 7 条输出，正确**（bench.o 6 次 + dmesg 1 次）



【5】实验心得

本次实验相对综合，涉及模块编程、系统调用及处理、页表遍历等内容。在实践过程中， 我通过 Google 查找了大量资料，阅读了许多内核关于系统调用和内存页表的源码，提高了我阅读分析大规模系统软件源码的能力和调试能力。总之，本次实验收获不少！感谢老师和助教的指导！

【附 录】模块源代码 sys_clone_hook.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kallsyms.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Zihan Liu");
MODULE_DESCRIPTION("a LKM to hook sys_clone");

typedef void (*sys_call_ptr_t)(void);
typedef asmlinkage long (*sys_clone_t)(unsigned long, unsigned long, int __user
*,
                                       int __user *, unsigned long);

sys_call_ptr_t *syscall_table = NULL;
sys_clone_t orig_clone = NULL;
unsigned int level;
pte_t *pte;


asmlinkage long hooked_sys_clone(unsigned long x1, unsigned long x2, int __user
*x3,
                                 int __user *x4, unsigned long x5) {
    long ret_val = orig_clone(x1, x2, x3, x4, x5);
    printk(KERN_INFO "hello, I have hacked this syscall");
    return ret_val;
```

```
}


static int __init sys_clone_hook_init(void) {
    syscall_table = (sys_call_ptr_t *)kallsyms_lookup_name("sys_call_table");

    // save the original syscall handler
    orig_clone = (sys_clone_t)syscall_table[__NR_clone];

    // unprotect syscall_table memory page
    pte = lookup_address((unsigned long)syscall_table, &level);

    // change PTE to allow writing
    set_pte_atomic(pte, pte_mkwrite(*pte));

    // overwrite the __NR_clone entry with address to our hook
    syscall_table[__NR_clone] = (sys_call_ptr_t)hooked_sys_clone;

    // reprotect page
    set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));

    printk(KERN_INFO "installed sys_clone_hook");
    return 0;
}


static void __exit sys_clone_hook_exit(void) {
    // change PTE to allow writing
    set_pte_atomic(pte, pte_mkwrite(*pte));

    // restore syscall_table to the original state
    syscall_table[__NR_clone] = (sys_call_ptr_t)orig_clone;

    // reprotect page
    set_pte_atomic(pte, pte_clear_flags(*pte, _PAGE_RW));

    printk(KERN_INFO "uninstalled sys_clone_hook");
}


module_init(sys_clone_hook_init);
module_exit(sys_clone_hook_exit);
```