一. 实验目的

熟悉操作系统内核的编译和运行环境设置，并利用内核模块中存在的安全漏洞来完成权限提升

二. 实验内容

1. 编译 Linux 操作系统内核
2. 使用 QEMU 运行操作系统内核
3. 编写 C 用户程序与内核模块进行高效交互
4. 补全漏洞利用模板程序来完成权限提升

注：以下实验在 Ubuntu 20.04 系统中已完成测试。

三. 背景知识

1. 操作系统内核

内核程序是现代网络计算设备中安全级别最高的组件之一。以开源 Linux 内核为例，当今社会的网络计算设备中，从企业级服务器、超级计算机（如核电站），到个人手机设备（如安卓手机）、IoT 设备（如网络摄像头）几乎都运行着 Linux 操作系统，其安全性严重依赖底层的 Linux 内核。同样的，我国国产桌面及服务器操作系统（如统信操作系统 UOS，中标麒麟 NeoKylin，深之度 DeepIn），其底层同样运行 Linux 内核。不同于用户空间程序，内核程序运行在更高的权限层级，其中存在的漏洞通常更为严重。因此，内核程序天然成为各种恶意攻击者的首选目标，而未及时修复的内核漏洞不仅会破坏计算设备的正常运行，而且会在现实世界中造成不可预料的灾难性结果。例如，2016 年，脏牛漏洞（Dirty COW）允许攻击者利用 Linux 内核内存管理中的条件竞争完成本地提权。同时，攻击者还可以利用该漏洞获取 Android 7 Root 权限 [1]；2017 年，勒索软件 WannaCry 利用 Windows 内核漏洞破坏了超过 150 个国家的 20 多万台计算机设备，造成了近 40 亿美元的经济损失 [2]。因此，维护内核程序的代码质量和安全运行对于保障现实世界的正常运转和安全运行至关重要。

2. 释放后使用漏洞（Use-after-free，下文中统一使用简称 UAF）

free 函数用于回收在堆上动态分配后不再使用的数据对象。但是由于程序员的一些不适当的操作（如未对指针进行清空操作），会导致攻击者能够操控已经被释放的区域。

```
p = malloc(DRILL_ITEM_SIZE);
free(p); // 释放p所指向的对象1
// 使用残留指针 p 对已回收的数据区域进行操作
q = malloc(DRILL_ITEM_SIZE)
// 使用残留指针 p 对新分配的对象2进行操作
```

UAF 漏洞利用的一个关键点在于这块内存区域被重新释放。这样利用残留指针 p 来操作新分配的对象。如图所示，残留指针 p 可以控制 q 所指向的新的数据对象。

3. 空指针引用漏洞（Null Pointer Dereference）

如下所示，当数据指针或代码指针是 NULL 时，使用其进行内存访问的时候，就会触发空指针引用，导致程序崩溃。这种漏洞在用户态一般被认为只能进行拒绝服务攻击，无法进行高阶漏洞利用。但是在内核中则不然。

```
char *p = NULL;              // 数据指针
*p;                          // 数据指针引用
void (*p)(int) = NULL;       // 函数指针
p(0);                        // 函数指针调用
```

四. 内核模块 (drill_mod.c) 分析

```c
static int __init drill_init(void)
{
        struct dentry *act_file = NULL;

        pr_notice("drill: start hacking\n");

        drill.dir = debugfs_create_dir("drill", NULL);
        if (drill.dir == ERR_PTR(-ENODEV) || drill.dir == NULL) {
                pr_err("creating drill dir failed\n");
                return -ENOMEM;
        }

        act_file = debugfs_create_file("drill_act", S_IWUGO,
                                       drill.dir, NULL, &drill_act_fops);
        if (act_file == ERR_PTR(-ENODEV) || act_file == NULL) {
                pr_err("creating drill_act file failed\n");
                debugfs_remove_recursive(drill.dir);
                return -ENOMEM;
        }

        return 0;
}
```

```
static void __exit drill_exit(void)
{
        pr_notice("drill: stop hacking\n");
        debugfs_remove_recursive(drill.dir);
}

module_init(drill_init)
module_exit(drill_exit)
```

以上两个函数为模块的初始化函数和退出函数。初始化函数 drill_init() 调用 debugfs_create_dir() 和 debugfs_create_file() 在 /sys/kernel/debug 目录下面创建一个目录 drill 以及在其中创建一个文件 drill_act。该函数还为 drill_act 文件创建了一个自定义的 write 函数。

```
static const struct file_operations drill_act_fops = {
        .write = drill_act_write,
};
```

此 write 函数，即，drill_act_write() 将用户传递或者说写入的字符，转换为数字，并执行它要完成的相应功能，具体功能详见 drill_act_exec 函数解析。

```
static ssize_t drill_act_write(struct file *file, const char __user *user_buf,
                               size_t count, loff_t *ppos)
{
        ssize_t ret = 0;
        char buf[ACT_SIZE] = { 0 };
        size_t size = ACT_SIZE - 1;
        long new_act = 0;

        BUG_ON(*ppos != 0);

        if (count < size)
                size = count;

        if (copy_from_user(&buf, user_buf, size)) {
                pr_err("drill: act_write: copy_from_user failed\n");
                return -EFAULT;
        }

        buf[size] = '\0';
        new_act = simple_strtol(buf, NULL, 0);

        ret = drill_act_exec(new_act);
        if (ret == 0)
                ret = count; /* success, claim we got the whole input */

        return ret;
}
```

```c
static int drill_act_exec(long act)
{
        int ret = 0;

        switch (act) {
        case DRILL_ACT_ALLOC:
                drill.item = kmalloc(DRILL_ITEM_SIZE, GFP_KERNEL);
                if (drill.item == NULL) {
                        pr_err("drill: not enough memory for item\n");
                        ret = -ENOMEM;
                        break;
                }

                pr_notice("drill: kmalloc'ed item at %lx (size %d)\n",
                                (unsigned long)drill.item, DRILL_ITEM_SIZE);

                drill.item->callback = drill_callback;
                break;

        case DRILL_ACT_CALLBACK:
                pr_notice("drill: exec callback %lx for item %lx\n",
                                        (unsigned long)drill.item->callback,
                                        (unsigned long)drill.item);
                drill.item->callback(); /* No check, BAD BAD BAD */
                break;

        case DRILL_ACT_FREE:
                pr_notice("drill: free item at %lx\n",
                                        (unsigned long)drill.item);
                kfree(drill.item);
                break;

        case DRILL_ACT_RESET:
                drill.item = NULL;
                pr_notice("drill: set item ptr to NULL\n");
                break;

        default:
                pr_err("drill: invalid act %ld\n", act);
                ret = -EINVAL;
                break;
        }

        return ret;
}
```

drill_act_exec 函数中仅支持四种功能，DRILL_ACT_ALLOC (= 1), DRILL_ACT_CALLBACK (= 2), DRILL_ACT_FREE (= 3), DRILL_ACT_RESET (= 4)。DRILL_ACT_ALLOC 为 drill.item 分配一个数据对象, 并设置 callback 函

数指针；DRILL_ACT_CALLBACK 调用 callback 函数；DRILL_ACT_FREE 回收之前分配的数据对象；DRILL_ACT_RESET 置空 drill.item。

```
enum drill_act_t {
        DRILL_ACT_NONE = 0,
        DRILL_ACT_ALLOC = 1,
        DRILL_ACT_CALLBACK = 2,
        DRILL_ACT_FREE = 3,
        DRILL_ACT_RESET = 4
};
```

五. 内核启动环境配置

QEMU 安装:
sudo apt-get update
sudo apt-get install qemu qemu-kvm

Linux 内核编译过程:
sudo apt-get install build-essential flex bison bc libelf-dev libssl-dev libncurses5-dev gcc-8
wget https://github.com/torvalds/linux/archive/v5.0-rc1.tar.gz
tar -xvf v5.0-rc1.tar.gz
cd linux-v5.0-rc1
make x86_64_defconfig
make -j8 CC=gcc-8

QEMU 启动脚本（注：wheezy.img 使用学习通实验材料中提供的版本）：
qemu-system-x86_64 \
  -kernel linux-5.0-rc1/arch/x86/boot/bzImage \
  -append "console=ttyS0 root=/dev/sda debug earlyprintk=serial slub_debug=QUZ pti=off oops=panic ftrace_dump_on_oops nokaslr"\
  -hda wheezy.img \
  -net user,hostfwd=tcp::10021-:22 -net nic \
  -nographic -m 512M -smp 2 \
  -pidfile vm.pid 2>&1 | tee vm.log

SSH 连接:
ssh -i wheezy.id_rsa -p 10021 -o "StrictHostKeyChecking no" root@localhost

SCP 拷贝:
scp -r -i wheezy.id_rsa -p 10021 -o "StrictHostKeyChecking no" drill@localhost:/home/drill/

同学们也可以使用发布的 startvm, connectvm, scptovm, killvm 等脚本来进行虚拟机启动、连接、拷贝数据、销毁。

```
[    32.702946] drill_mod: loading out-of-tree module taints kernel.
[    32.708171] audit: type=1400 audit(1652075715.028:7): avc:  deni
[    32.740963] drill: start hacking
4+0 records in
4+0 records out
4 bytes (4 B) copied, 0.000667 s, 6.0 kB/s
[    32.989283] audit: type=1400 audit(1652075715.316:8): avc:  deni
[ ok ] Starting enhanced syslogd: rsyslogd.
Starting mcstransd:
[ ok ] Starting periodic command scheduler: cron.
[ ok ] Starting file context maintaining daemon: restorecond.
[ ok ] Starting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 7 syzkaller ttyS0

syzkaller login:
[0] 0:bash*
```

用户空间编程与内核模块交互：

1. echo “1” > /sys/kernel/debug/drill/drill_act

```
drill@syzkaller:~$ echo "c" > /sys/kernel/debug/drill/drill_act

-bash: echo: write error: Invalid argument
drill@syzkaller:~$ echo "1" > /sys/kernel/debug/drill/drill_act

drill@syzkaller:~$ echo "2" > /sys/kernel/debug/drill/drill_act

drill@syzkaller:~$ echo "3" > /sys/kernel/debug/drill/drill_act

drill@syzkaller:~$ echo "4" > /sys/kernel/debug/drill/drill_act
```

2. 编写 C 语言来交互

int fd = open("/sys/kernel/debug/drill/drill_act", O_WRONLY);
write(fd, "1", 1);

或采用右侧所示规格化函数，

```c
int act(int fd, char code)
{
        ssize_t bytes = 0;

        bytes = write(fd, &code, 1);
        if (bytes <= 0) {
                perror("[-] write");
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

六. 作业内容

1. 编译指定 Linux 内核，并配置 QEMU 环境运行指定内核；
2. 补全高效交互的 C 语言代码 (drill_operations.c)，UAF 漏洞的利用代码 (drill_exploit_uaf.c)，触发空指针引用漏洞的 PoC 代码 (drill_trigger_crash.c)，以及空指针引用漏洞的利用代码 (drill_exploit_nullderef.c)；
3. 提权后查看 /root/flags 中内容（*注意，该文件内容随内核启动更新*）

注意，这些代码中以“// MDL: ”开头的注释

七. 分析报告

编写分析报告来描述实验完成的全过程

八. 学有余力

学有余力的同学可以做如下两种尝试来修复或防御驱动代码中的漏洞：
1. 漏洞修复：对 UAF 漏洞进行修复，操作即在 kfree 之后，添加对于指针的置空操作；对空指针引用漏洞进行修复，操作即在使用指针之前判断该指针是否为空；
2. 开启防御措施 SMAP（Supervisor Mode Access Prevention）来禁止从内核空间访问用户空间，开启方法为“-cpu kvm64, smap”

九. 参考资料

1. UAF: https://cwe.mitre.org/data/definitions/416.html
2. SMAP: https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention
3. CVE-2019-9213: https://bugs.chromium.org/p/project-zero/issues/detail?id=1792
4. QEMU: https://www.qemu.org/